



Desenvolvimento de Sistemas

Programação de computadores: fundamentos, código, compiladores, código de máquina

Fundamentos



Figura 1 – Cientista da computação Margaret Hamilton ao lado do código de programação do sistema de voo e pouso do programa Apollo, da NASA

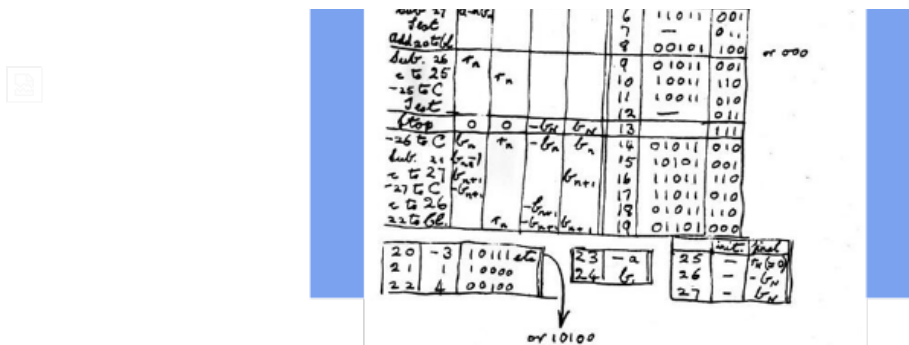
Fonte: <<https://solarsystem.nasa.gov/people/320/margaret-hamilton/>>. Acesso em: 28 out. 2021.

Programação de computadores nada mais é do que projetar, desenvolver e construir um programa executável em um computador, para que o mesmo cumpra uma determinada tarefa ou responda com determinado e/ou específico resultado.

Programar computadores exige o cumprimento de certas tarefas, tais como análises, geração de algoritmos, análise de recursos computacionais, implementação e testes de algoritmos em uma linguagem de programação escolhida, este último processo é comumente conhecido como codificação.

1967/68 *Kilburn Highest Factor Routine (amended)*

Index	C	24	26	27	Line	0	1	2	3	4	5	13	14	15
24	C	-G ₁	-	-	1	0	0	0	1	1	0	1	0	0
26	C	-G ₁	-	-	2	0	1	0	1	1	1	1	0	0
27	C	-G ₁	-	-	3	0	1	0	1	1	0	1	0	0
28	C	-G ₁	-	-	4	1	1	0	1	1	1	1	0	0
29	C	-G ₁	-	-	5	1	1	0	1	0	1	0	1	0



Address	Operation	Data
0	Clear	
1	Read	
2	Write	
3	Clear	
4	Read	
5	Write	
6	Clear	
7	Read	
8	Write	
9	Clear	
10	Read	
11	Write	
12	Clear	
13	Read	
14	Write	
15	Clear	
16	Read	
17	Write	
18	Clear	
19	Read	
20	Write	
21	Clear	
22	Read	
23	Write	
24	Clear	
25	Read	
26	Write	
27	Clear	

or 10100

Figura 2 – Demo original do primeiro código a ser testado no primeiro computador eletrônico (Eniac)

Fonte: <<https://computerhistory.org/wp-content/uploads/2019/08/programing-the-eniac-BabyProgram.jpg>>.

Acesso em: 28 out. 2021.



Figura 3 – Programador atual criando programas de computador

Fonte: <<https://www.businessinsider.com/what-is-coding>>. Acesso em: 28 out. 2021.

Pense sobre as diversas maneiras com as quais computadores são utilizados por pessoas no dia a dia. Em escolas, estudantes usam computadores para escrever artigos, fazer pesquisas na Internet, enviar *e-mail* e até participar de aulas *on-line*. No trabalho, as pessoas usam computadores para gerar planilhas eletrônicas, fazer cálculos, alimentar agendas, criar e participar de reuniões, fazer transações bancárias e conduzir apresentações e palestras. Em casa, computadores são comumente usados para entretenimento, compras *on-line*, conversas com amigos e jogos eletrônicos, entre outros. Não se deve esquecer, é claro, que cada aparelho de telefone celular é também um computador e tem tudo o que é necessário para executar cada uma dessas mesmas tarefas. O uso de computadores no mundo atual é imprescindível, e é praticamente impossível entrar em um ambiente sem que um computador esteja presente, tanto no local quanto nos bolsos dos que estão

naquele mesmo local.



Computadores podem executar tal gama de tarefas exatamente porque podem ser programados. Isso quer dizer que um computador não é criado para resolver uma tarefa em específico, e sim para executar qualquer tarefa que o programador de computadores desejar.

Um programa de computador é um conjunto de instruções e algoritmos que o programador escreve para que o computador cumpra e resolva problemas ou tarefas. Por exemplo, o Adobe Photoshop é um programa de computador que foi programado de tal maneira a manipular imagens digitais, como fotos.

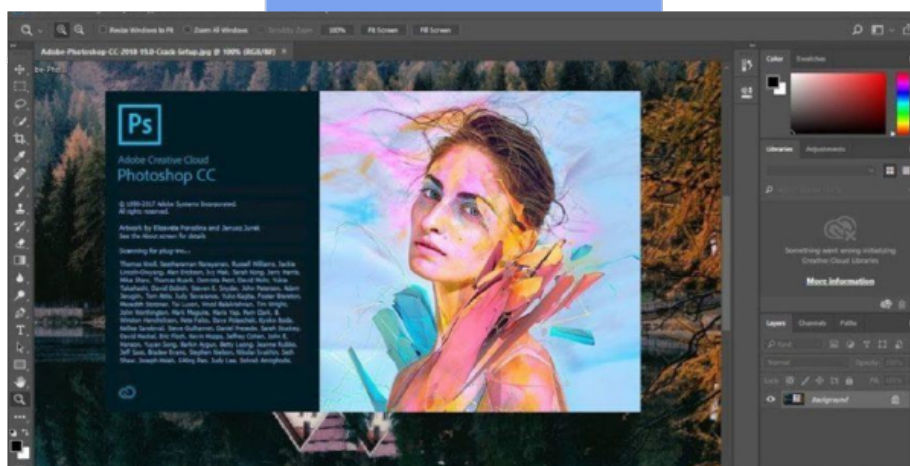


Figura 4 – Adobe Photoshop, programa de computador para manipulação de imagens digitais

Fonte: <<http://corporativa.unesc.net/cursos/introducao-ao-adobe-photoshop/>>. Acesso em: 28 out. 2021.

Programas de computadores são geralmente chamados de *softwares*, e um computador sem nenhum *software* não contém função alguma, já que é o *software* que controla todas as partes físicas do computador, partes essas chamadas de *hardware*.

Um programador ou um desenvolvedor de *software* é a pessoa responsável por criar os algoritmos, códigos e executáveis para controlar e fazer uso de um computador. É o programador que cria os programas de computador.

Programar computadores é uma carreira divertida, promissora e atual, pois, hoje em dia, programadores são encontrados em todas as áreas de trabalho e campos de pesquisa no mundo, como, por exemplo, nas áreas de medicina, governamentais, ciências, agricultura, entretenimento e muitos outros campos.





Figura 5 – Colheitadeira totalmente automatizada e assessorada por diversos *softwares* e computadores
Fonte: <<https://gingerichfarmsonline.com/about/>>. Acesso em: 28 out. 2021.

Código

Saber como um computador armazena informação será importante para entender, subsequentemente, como um programa de computador é criado, compilado e executado.

Como você pôde ver no conteúdo sobre **Sistemas Digitais** desta unidade curricular, os computadores digitais operam sob base binária. Assim, toda a informação armazenada em um computador acaba convertida em uma sequência de 0s, 1s. Isso é chamado de código binário, e cada valor desses (0 ou 1) é um interruptor ligado (1) ou desligado (0). Todos esses valores, chamados de *bits* (*binary digit*, ou, em português, “dígito binário”), são armazenados na memória principal ou na memória secundária de um computador.

As primeiras linguagens de programação se baseavam em instruções descritas com sequências de *bytes*, ou seja, eram instruções binárias. Um exemplo de instrução seria a seguinte sequência de *bytes*:

01000011 00111010 00111011 01000001 00101011 01000100

Observe que pode ser bastante complexo compreender o que cada sequência de 0s e 1s significa no programa. O programador deveria ter profundo conhecimento de quais componentes do dispositivo está manipulando, o que tornava a programação trabalhosa. Assim, surgiram linguagens mais compreensíveis ao ser humano com a criação das linguagens montadoras, ou Assembly. As instruções recebiam nomes, também denominados mnemônicos, que ficam mais compreensíveis. A instrução a seguir, por exemplo, soma dois valores – um armazenado em um registrador (memória) de endereço “CL” e outro valor no registrador “BH”.

ADD CL,BH

Isso seria correspondente em binário à sequência 00000010 11001111.



A partir do sucesso dos Assembly, novas linguagens foram sendo desenvolvidas.

Em suma, programadores de computador criam código de programação em alguma dessas linguagens. Esse código, por sua vez, é interpretado por um compilador e então convertido em código de máquina, que é ilegível para seres humanos, mas rapidamente “lido” por computadores.

No começo da história da programação de computadores, as CPUs (*central process unit*, ou “unidades centrais de processamento”) tinham poucas instruções e eram capazes de fazer algumas das seguintes operações:

- ◆ Ler informação de uma memória principal
- ◆ Somar dois números
- ◆ Subtrair um número de outro
- ◆ Multiplicar dois números
- ◆ Dividir um número por outro
- ◆ “Movimentar” uma parte de informação de um lugar para outro na memória
- ◆ Comparar dois valores quanto a seu tamanho

Como se pode ver, uma CPU do início da época dos computadores digitais era capaz de fazer pequenas tarefas em pequenas quantidades de informação.

Deve-se ter sempre em mente que uma CPU não executa nada sozinha, apenas cumpre as funções que estão no código de programação.

Linguagens de programação de alto nível

Em 1950, a linguagem de programação comumente usada era o Assembly, mas, por ser uma linguagem de baixo nível, ou seja, difícil para um ser humano entender e “visualizar” o código, começaram a aparecer linguagens de programação chamadas de alto nível, permitindo que programadores humanos construíssem códigos muito mais complexos e capazes de realizar inúmeras tarefas.

As linguagens de alto nível, assim, escondem instruções mais básicas da computação em instruções mais sucintas. Assim, uma instrução acaba sendo capaz de executar várias operações de uma vez.

Uma das primeiras linguagens de alto nível criadas foi o Cobol e o código de programação que se usa em Cobol para criar um programa de computador que mostre a mensagem “Olá, Mundo” na tela de um computador, escreve-se da seguinte maneira:

DISPLAY "Olá Mundo"



Java, que é uma linguagem moderna muito utilizada atualmente, pode executar o mesmo trabalho de mostrar uma mensagem na tela de um computador por meio do seguinte código de programação.

```
System.out.println("Olá Mundo");
```

Esses dois códigos de programação, mesmo que simples, são fáceis de serem entendidos por humanos. Caso esse mesmo programa seja escrito em Assembly, que é de bem mais baixo nível, ficaria da seguinte forma:

```
section .text align=0
global _start
mensagem db 'OláMundo', 0x0a
lenequ $ - mensagem
_start:
moveax, 4
movebx, 1
movecx, mensagem
movedx, len
int 0x80
moveax, 1
int 0x80
```

Como se pode ver, o código em Assembly é bem mais difícil de ser rapidamente interpretado por um ser humano, e com certeza levou muito mais tempo para ser escrito.

A primeira imagem vista neste conteúdo foi a da programadora Margaret Hamilton, que criou o código de navegação e pouso da missão Apollo para que houvesse pouso com segurança na Lua. Todo aquele código visto na imagem foi escrito em Assembly!





Figura 6 – Parte do código de pouso do veículo de pouso lunar da Apollo 11, escrito em Assembly pela equipe e coordenado por Margaret Hamilton

Fonte: <https://www.i-programmer.info/history/people/10030-margaret-hamilton-apollo-and-beyond.html?start=1>. Acesso em: 29 out. 2021.

Cada linguagem de programação de alto nível contém sua maneira de programar e respeita a sua documentação.

Confira algumas das linguagens de programação de alto nível e um pouco sobre elas:

Ada

Linguagem de programação criada nos anos de 1970 e primeiramente adotada pelo departamento de defesa dos Estados Unidos da América. Teve seu nome dado em homenagem à condessa Ada Lovelace, que foi uma grande influência na história da computação, sendo considerada a primeira programadora da história.

Basic

Beginners all-purpose symbolic instruction code ou, como em português, “código de instrução simbólico para todos os fins para iniciantes” é uma linguagem de programação originalmente criada no começo da década de 1960 com o intuito de ser simples o suficiente para novos programadores aprenderem a usar sem maiores problemas. Hoje em dia, ainda existem diversas versões diferentes do Basic.

Fortran

Formula translator ou “tradução de fórmulas” foi desenvolvida nos anos de 1950 e é considerada a primeira linguagem de alto nível criada para realizar cálculos matemáticos complexos.

Cobol

Common business oriented language ou, como em português, “linguagem comum orientada para os

negócios” foi criada nos anos de 1950, e teve como propósito as aplicações para negócios.



Pascal

O Pascal foi criado nos anos de 1970 originalmente para ensinar programação, e teve seu nome dado em homenagem ao matemático, físico e filósofo Blaise Pascal.

C e C++

C e C++ são linguagens de programação poderosas e de uso geral. A primeira foi criada nos laboratórios da Bell, no ano de 1972, e a segunda, em 1983.

Java

Java (não confundir com JavaScript) é uma linguagem que foi criada pela empresa Sun Microsystems no começo dos anos de 1990, e pode ser utilizada para desenvolvimento de programas que rodem em qualquer máquina capaz de rodar a JVM (Java Virtual Machine).

JavaScript

Criada nos anos de 1990, essa linguagem pode ser utilizada principalmente em páginas *web* e, apesar do seu nome, não tem relação nenhuma com a linguagem Java.

Python

Python é uma linguagem de propósito geral criada nos anos de 1990, e que hoje é muito popular em negócios e trabalhos acadêmicos.

Ruby

Linguagem de propósito geral, criada nos anos de 1990 e popular em servidores *web*.

Visual Basic



Também conhecida como VB, é uma linguagem de programação criada pela empresa Microsoft no início dos anos 1990, e que permite que programadores criem rapidamente programas de computador para o seu sistema operacional Windows.

Toda linguagem de programação contém suas **palavras reservadas**. Essas palavras têm funções específicas que dão ordens específicas ao compilador para que ele transforme esse código de programação em uma linguagem de mais baixo nível, que pode ser interpretada pelo CPU de um computador.



<code>abstract</code>	<code>assert</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>	<code>case</code>
<code>catch</code>	<code>char</code>	<code>class</code>	<code>const</code>	<code>continue</code>	<code>default</code>
<code>double</code>	<code>do</code>	<code>else</code>	<code>enum</code>	<code>extends</code>	<code>false</code>
<code>final</code>	<code>finally</code>	<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>
<code>implements</code>	<code>import</code>	<code>instanceof</code>	<code>int</code>	<code>interface</code>	<code>long</code>
<code>native</code>	<code>new</code>	<code>null</code>	<code>package</code>	<code>private</code>	<code>protected</code>
<code>public</code>	<code>return</code>	<code>short</code>	<code>static</code>	<code>strictfp</code>	<code>super</code>
<code>switch</code>	<code>synchronized</code>	<code>this</code>	<code>throw</code>	<code>throws</code>	<code>transient</code>
<code>true</code>	<code>try</code>	<code>void</code>	<code>volatile</code>	<code>while</code>	

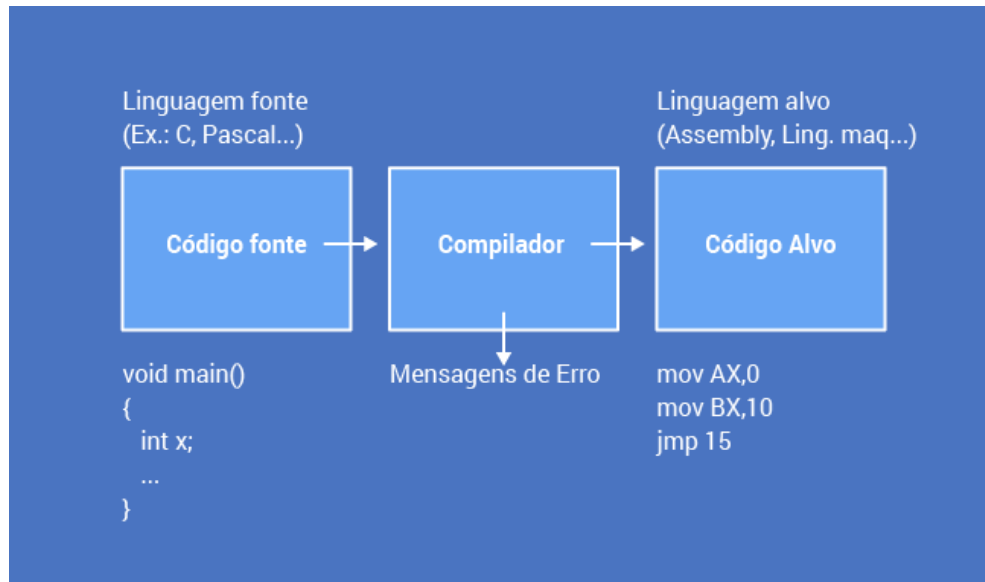
Figura 7 – Exemplos de palavras reservadas da linguagem de programação Java

Compiladores e código de máquina

Como visto anteriormente, as CPUs de computadores só entendem instruções de baixo nível, por isso é preciso uma maneira de converter o código de programação a um nível que a CPU entenda. É aí que entram os compiladores e interpretadores.

Compiladores e interpretadores são os programas que fazem a conversão do código de programação em alto nível, o qual foi escrito pelo programador em instruções que a CPU de um computador consiga “entender”. Compiladores são utilizados em linguagens que precisam ser compiladas e interpretadores em linguagens que são apenas interpretadas.

Em geral, um compilador não produz diretamente código de máquina (binário) mas sim um código Assembly que faça as operações equivalentes às instruções do código em alto nível sendo compilado (lembre-se, pelo exemplo do “Olá, Mundo”, de que uma instrução em código de alto nível pode levar a dezenas de instruções em Assembly).



Esquema 1 – Processo de compilação básico

Fonte: <<http://www.gpec.ucdb.br/pistori/disciplinas/compiladores/compilador.gif>>. Acesso em: 29 out. 2021.

Como se pode notar, o código, após compilado, é convertido em código Assembly, que é então executado pelo processador do computador.

Será trabalho do compilador ler o código escrito em alto nível, identificar cada palavra reservada da linguagem e interpretar a sequência de termos e símbolos presentes na linguagem para definir exatamente quais tarefas em Assembly devem ser executadas. Veja, como exemplo, o seguinte trecho de código em Java:

```
int a, b;
a = 10;
b = a + 2;
```

A primeira linha realiza uma declaração de variável (um espaço para guardar dados). O compilador precisa compreender como se declaram variáveis no Java para interpretar corretamente a instrução dessa primeira linha. Em Java, a declaração usa a sintaxe `,`. Assim, o compilador identifica “int” como um tipo válido e “a” e “b” como suas identificações. Caso fosse escrito algo fora desse padrão, o compilador acusaria erro de compilação, como no código a seguir:

```
int 10, b;
```

Assessment

Observe a seguir um exemplo de compilação de código em C++.

1. Primeiro, o programador escreve o código:

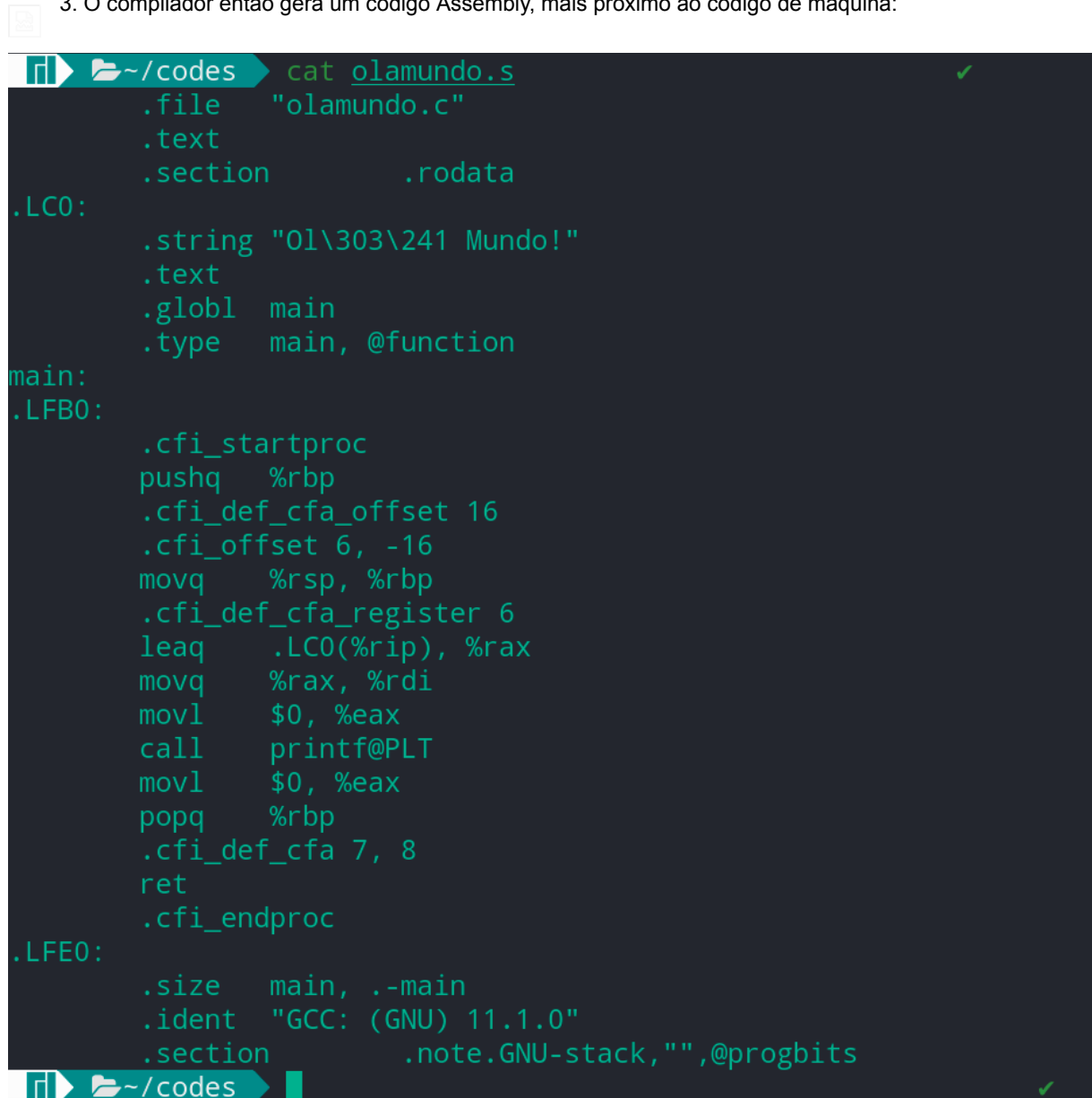
```
#include <stdio.h>
int main() {
    printf("Olá Mundo!");
    return 0;|
}
~
~
~
~
~
~
~
~
~
~
~
olamundo.c [+] 4,13 Tudo
-- INSERÇÃO --
```



2. Depois, com linhas de comando, é necessário acionar o compilador:

```
~/codes gcc -S olamundo.c ✓
~/codes ls ✓
olamundo.c olamundo.s
~/codes
```

3. O compilador então gera um código Assembly, mais próximo ao código de máquina:



```
cat olamundo.s
.file "olamundo.c"
.text
.section .rodata
.LC0:
.string "Ol\303\241 Mundo!"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
leaq .LC0(%rip), %rax
movq %rax, %rdi
movl $0, %eax
call printf@PLT
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 11.1.0"
.section .note.GNU-stack,"",@progbits
```

4. A partir desse código, é gerado o executável (binário), que, nesse caso, simplesmente mostra na tela "Olá Mundo":



```
~/codes ➤ ./a.out ✓  
Olá Mundo!%  
~/codes ➤
```

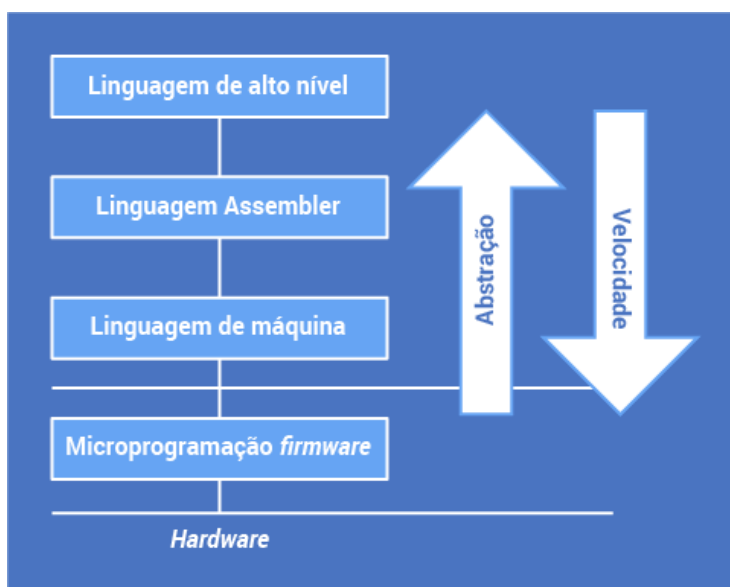
No exemplo, foi utilizado o C++, por ter um processo mais claro de compilação que outras linguagens, como Java.



Algumas linguagens, como C++, Java e C#, passam por um processo explícito de compilação, como o descrito anteriormente; ou seja, é necessário que o programador acione o compilador manualmente antes de executar seu programa. A partir dessa compilação, gera-se o arquivo executável (no Windows, esses arquivos geralmente possuem extensão exe).

Outras linguagens como JavaScript e Python, são interpretadas, ou seja, o processo de compilação é transparente ao programador – basta escrever o código e já é possível executá-lo. O interpretador aqui é o correspondente ao compilador e é responsável em converter instrução por instrução em código que pode ser interpretado pelo processador do computador.

Computadores gostam de 1s e 0s, seres humanos gostam de símbolos. O Assembler é o programa responsável por converter símbolos em instruções de máquina.



Nesse fluxograma, é possível constatar que, quanto mais alto for o nível da linguagem, mais alta será sua abstração e mais fácil será a leitura. No entanto, mais lento seria para um computador ler.

Quanto mais baixo é o nível da linguagem, mais fácil é para a máquina interpretar e mais abstrato é para seres humanos.

Lógica proposicional e lógica booliana

A lógica proposicional é uma das bases da computação e se aplica ao sistema binário utilizado nos

sistemas digitais. É também essencial para a compreensão da lógica de programação – especialmente no que diz respeito a decisões, ou seja, à capacidade de um código realizar comparações e executar uma ou outra operação de acordo com o resultado dessa comparação.

A lógica proposicional se baseia no conceito de **proposições**, uma sentença declarativa (expressa por palavras ou símbolos) cujo valor pode ser considerado verdadeiro ou falso (e nenhum outro diferente desses). Por exemplo:

- ◆ “A Terra é maior que a lua” é uma proposição verdadeira.
- ◆ “ $10 > 50$ ” é uma proposição falsa.

Algumas sentenças como “feliz ano-novo”, ou “qual é o seu nome?” não podem ser consideradas como proposições, já que não têm um valor “verdadeiro” ou “falso” agregados a ela. Geralmente são usadas as letras **p**, **q**, **r**, e assim por diante, para representar uma proposição e montar combinações entre elas. Observe o exemplo:

- ◆ **p**: João é médico.
- ◆ **q**: Luíza foi ao cinema ontem.
- ◆ **r**: $5 > 8$

Três princípios são importantes nas proposições:

- ◆ Uma proposição verdadeira é verdadeira; uma proposição falsa é falsa (princípio da identidade).
- ◆ Nenhuma proposição poderá ser verdadeira e falsa ao mesmo tempo. (princípio da não contradição).
- ◆ Uma proposição ou será verdadeira, ou será falsa: não há outra possibilidade (princípio do terceiro excluído).

As proposições combinadas é que ganham força na computação. Observe novamente as proposições **p**, **q** e **r** citadas. É possível combiná-las com **conectivos lógicos**:

- ◆ **p “e” q** \rightarrow João é médico **e** Luíza foi ao cinema ontem.
- ◆ **p “ou” r** \rightarrow João é médico **ou** $5 > 8$.

Para determinar se uma proposição composta é verdadeira ou falsa, isso depende de duas coisas: 1.º) do valor lógico das proposições componentes; e 2.º) do tipo de conectivo que as une.

Os principais conectivos são “e” e “ou”. O conectivo “e” (chamado de **conjunção**) define que uma

expressão ou sentença é verdadeira quando ambas as proposições são verdadeiras. Repare que, quando as pessoas conversam, em linguagem natural, e usa-se o termo “e”, está-se, de fato, indicando que duas situações aconteceram ou que dois fatos são verdadeiros. Assim é também nas proposições, nas quais se usa o símbolo \wedge para representar conjunção.

◆ $p \wedge q \rightarrow$ João é médico e Luíza foi ao cinema ontem.

A expressão anterior só será verdadeira se João for médico e se de fato Luíza foi ao cinema ontem. Caso uma dessas proposições seja falsa, a sentença se torna falsa (é mentira que João é médico ou é mentira que Luíza foi ao cinema ontem).

Outro conectivo é o “ou” (chamado **disjunção**). Quando se inclui a palavra “ou” na linguagem natural, estão sendo oferecidas alternativas – uma das partes conectadas é verdadeira. Assim é também na lógica proposicional: uma expressão com “ou” será verdadeira se alguma das proposições for verdadeira. Usa-se para isso o símbolo \vee .

◆ $p \vee r \rightarrow$ João é médico ou $5 > 8$.

A expressão será verdadeira se João for médico; caso ele não seja, haverá duas proposições falsas, já que 5 não é maior que 8, o que levará a expressão a um valor falso.

Em resumo:

- ◆ Uma conjunção (“e”) só será verdadeira se ambas as proposições componentes forem também verdadeiras. Basta que uma das proposições componentes seja falsa para que a conjunção seja toda ela falsa.
- ◆ Uma disjunção (“ou”) será falsa quando as duas partes que a compõem forem falsas. Nos demais casos, a disjunção será verdadeira.

O que isso tem a ver com programação e sistemas digitais? Muitas coisas! Essas operações são as bases para os circuitos lógicos, ou seja, um processador não existiria sem essas definições. O que acontece é que, nos circuitos, ao invés de “verdadeiro” e “falso”, trata-se com os dígitos 0 e 1. A partir disso, extrai-se a lógica (ou álgebra) booliana, um sistema de operações matemáticas (ou lógicas) desenvolvida por George Boole sobre os dígitos 0 e 1.

Voltando às proposições, confira a tabela utilizada para representar todos os valores possíveis – o verdadeiro (V) e o falso (F) – e seus resultados em conjunções:



p	q	$p \wedge q$	Descrição	Exemplo
V	V	V	Se p é verdadeiro e q é verdadeiro, então $p \wedge q$ é verdadeiro.	p = 5 é ímpar q = um dia tem 24 horas $p \wedge q$ = 5 é ímpar “e” um dia tem 24 horas (verdade, pois ambas as sentenças estão corretas.)
V	F	F	Se p é verdadeiro e q é falso, então $p \wedge q$ é falso.	p = 7 é maior que 5 q = uma hora tem 100 segundos $p \wedge q$ = 7 é maior que 5 “e” uma hora tem 100 segundos (falso, pois apesar de 7 ser maior que 5, uma hora tem 3600 segundos e não 100.)
F	V	F	Se p é falso e q é verdadeiro, então $p \wedge q$ é falso.	p = peixes têm penas q = 2 é par $p \wedge q$ = peixes têm penas “e” 2 é par (falso, pois peixes não têm penas.)
F	F	F	Se p é falso e q é falso, então $p \wedge q$ é falso.	p = 10 é maior que 70 q = 2 é igual a 3 $p \wedge q$ = 10 é maior que 70 “e” 2 é igual a 3 (falso, pois ambas as afirmações são incorretas)

Esses valores V e F podem ser trocados, com naturalidade, por 0 e 1.

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

De maneira análoga, é possível montar a tabela de disjunção:



p	q	$p \vee q$	Descrição	Exemplo
V	V	V	Se p é verdadeiro e q é verdadeiro então $p \vee q$ é verdadeiro.	p = 5 é ímpar q = um dia tem 24 horas $p \vee q$ = 5 é ímpar “ou” um dia tem 24 horas (verdade, pois ambas as sentenças estão corretas.)
V	F	V	Se p é verdadeiro e q é falso, então $p \vee q$ é falso.	p = 7 é maior 5 q = uma hora tem 100 segundos $p \vee q$ = 7 é maior que 5 “ou” uma hora tem 100 segundos (verdade, pois 7 é de fato maior que cinco, apesar de uma hora ter 3600 segundos e não 100.)
F	V	V	Se p é falso e q é verdadeiro, então $p \vee q$ é falso.	p = peixes têm penas q = 2 é par $p \vee q$ = peixes têm penas “ou” 2 é par (verdade, pois, independentemente dos peixes, 2 é par.)
F	F	F	Se p é falso e q é falso, então $p \vee q$ é falso.	p = 10 é maior que 70 q = 2 é igual a 3 $p \vee q$ = 10 é maior que 70 “ou” 2 é igual a 3 (falso, pois nem 10 é maior que 70, nem 2 é igual a 3.)

Com dígitos binários, obtém-se:

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

Outra operação ainda pode ser aplicada a uma proposição: a negação. Trata-se da inversão de um valor – se ele é verdadeiro, torna-se falso e vice-versa. Em binário, o 0 torna-se 1 e vice-versa. É representado pelo símbolo \sim ou \neg .



p	$\sim p$
V	F
F	V

p	$\sim p$
0	1
1	0

Pode-se combinar essas conjunções ainda em uma única tabela, chamada de **tabela da verdade**. Confira a seguir:

p	q	$p \wedge q$	$p \vee q$	$\sim p$
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

p	q	$p \wedge q$	$p \vee q$	$\sim p$
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Note que é possível aplicar expressões mais complexas, como $((p \vee q) \wedge \sim(p \vee q))$, por exemplo, ou incluir mais de duas proposições como $((p \wedge q) \vee (p \wedge r))$.

Há ainda dois outros operadores na lógica proposicional:

◆ “Se então” (\rightarrow)



- ◆ o $p \rightarrow q$ (se p então q) será falso apenas se q for falso

◆ “Se e somente se” (\leftrightarrow)

- ◆ o $p \leftrightarrow q$ (p se e somente se q) será verdadeiro quando p e q forem ambos verdadeiros ou ambos falsos.

Experimente preencher colunas com essas operações na tabela da verdade.

A compreensão dessas operações lógicas será, como comentado anteriormente, de enorme importância na programação, permitindo a verificação de situações de desvio de código a partir de decisões a serem tomadas. Nos dispositivos do computador, no entanto, ganham especial importância, uma vez que portas lógicas definem operações de soma, subtração e comparação, tudo isso usando o sistema binário. Para projetar circuitos, geralmente se usam símbolos para as operações lógicas.

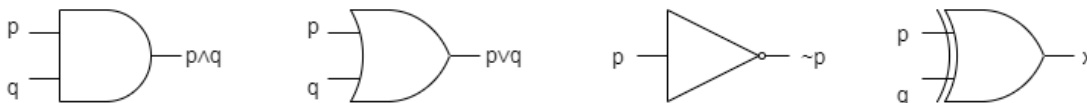


Figura 8 – Principais portas lógicas: respectivamente, para “e”, “ou”, “não” e “ou exclusivo” (verdadeiro apenas quando p e q são diferentes)

Fonte: autor

As portas lógicas definem, cada uma, entradas e saídas, resultando nas operações e valores observados na tabela da verdade. Nos circuitos eletrônicos, elas são encadeadas de maneira a permitir cálculos e processamentos variados.

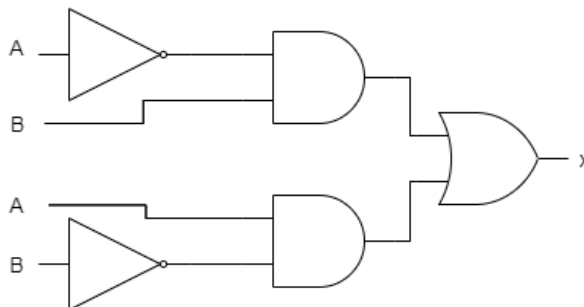


Figura 9 – Exemplo de circuito encadeando portas lógicas (os valores 0 ou 1 vindos pelas entradas A e B passam pelas operações de “não”, “e” e “ou” para resultar em um valor x ao final)

O conhecimento de portas lógicas e construção de circuitos é especialmente requisitado na engenharia da

computação, que estuda com particular interesse os projetos de *hardware*.



O que você aprendeu sobre programação de computadores?

Aprendeu o que é necessário para que um computador digital moderno funcione. Aprendeu também que *hardware* é necessário para que o mesmo possa executar tarefas.

Você notou que um computador digital moderno, sem um *software*, de nada serve, pois é a programação desse computador que o torna útil.

Também foi visto um pouco sobre linguagens de programação e suas utilidades, assim como o que é um código de programação.

Além disso, você pôde ver como um código de programação é compilado por um compilador.