2. CONCEITOS INICIAIS DE ELETRÔNICA DIGITAL

Olá! Seja bem-vindo à nossa segunda unidade de BCC265 – **Eletrônica para Computação**!

Neste encontro, conversaremos sobre como podem ser construídos os sistemas digitais. Porém, para isso, vamos passar por alguns pontos conceituais antes. Um desses pontos relaciona-se às abordagens de projeto e, também, relembrarmos um pouco sobre sistemas de numeração e os pontos básicos da álgebra que traça as diretrizes para o pensamento digital: a álgebra booleana.

Assim, para conversarmos sobre os circuitos digitais e, consequentemente, sobre a álgebra booleana, este capítulo trará alguns conceitos iniciais da álgebra booleana para que possamos abstrair os conceitos básicos que norteiam os circuitos digitais.

2.1 Desenvolvimento de Projetos Digitais

Antes de iniciarmos a nossa conversa sobre os circuitos digitais, é interessante falarmos sobre desenvolvimento de sistemas. Tanto nos sistemas baseados em *software* quanto naqueles baseados em *hardware*, devemos passar por algumas etapas – desde a sua proposição inicial, *design* (projeto) de seus módulos funcionais até a completa implementação.

De forma sumária, o desenvolvimento de sistemas deve compreender as seguintes etapas:

- Especificação
- Projeto
- Implementação
- Testes e análises

Cada etapa compreende um conjunto de ferramentas e metodologias de modo a facilitar o projeto e, também, proporcionar uma maior qualidade e eficiência do produto sob implementação.

Em linhas gerais, o desenvolvimento dos sistemas segue a sequência de etapas ilustradas na figura a seguir:

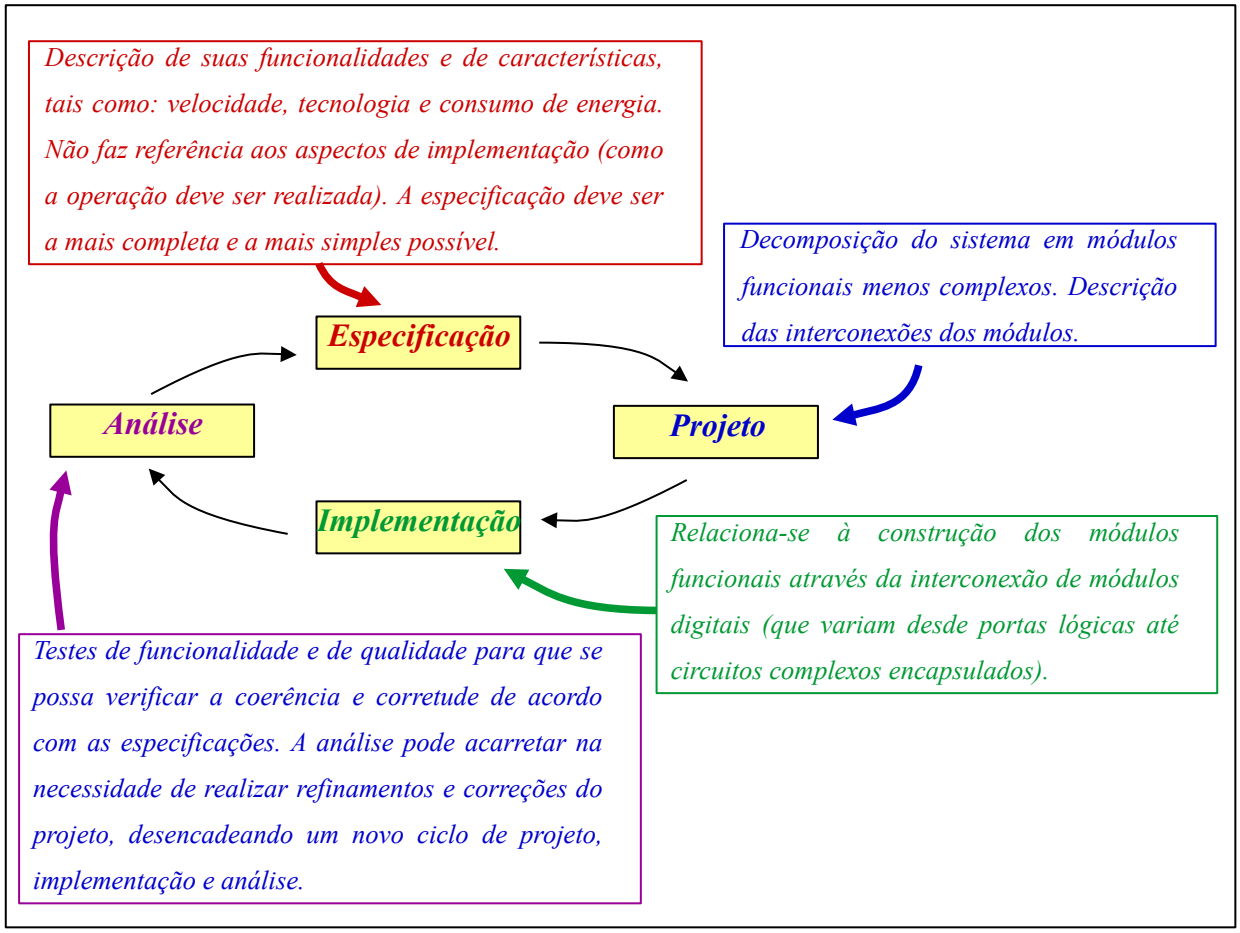


Figura 01 – Etapas para a elaboração de um sistema de *hardware*.

Fonte: Elaborada pelo autor, 2020.

Como podemos observar na figura acima, as etapas para o composição de um sistema de *hardware* é cíclico e pode permanecer ativo durante toda a vida útil do componente.

Você quer ler?

Atualmente, várias informações de diagnóstico podem ser coletadas durante toda a vida útil de um sistema computacional. Essas informações alimentam sistemas denominados como HMS (Health Management Systems – Sistemas de Gerenciamento da Saúde de Sistemas). Para ler sobre uma aplicação de HMS, você poderá ler o artigo escrito por (CHENG, 2010) que aborda PHM (*Prognostics and Health Management* – Prognósticos e Gerenciamento da Saúde) – uma das subáreas de HMS. O artigo encontra-se disponível em <https://www.mdpi.com/1424-8220/10/6/5774/pdf>.

Quando refletimos sobre as etapas de projeto e implementação, podemos pensar, ainda,

em duas estratégias: a abordagem *top-down* e a abordagem *botton-up*. A figura a seguir ilustra essas duas abordagens.

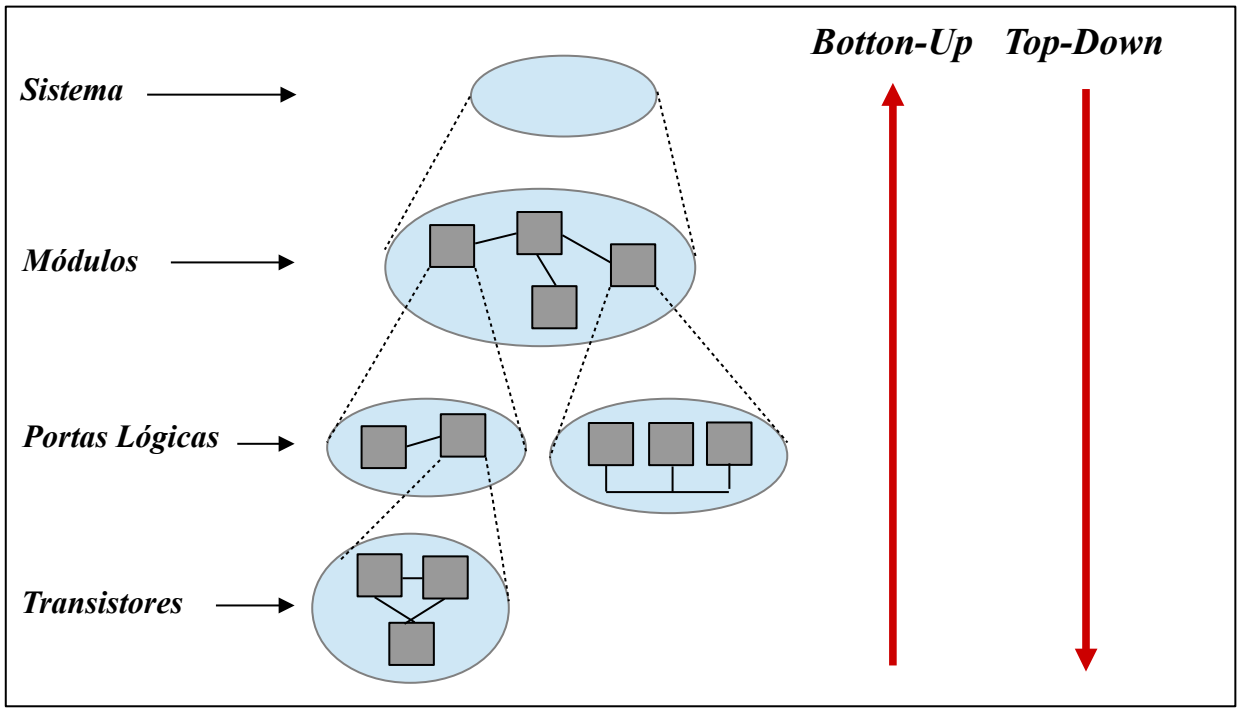


Figura 02 – Abordagens *botton-up* e *top-down* para a construção de um sistema de *hardware*.

Fonte: Elaborada pelo autor, 2020.

Na figura acima, temos ilustradas as abordagens *botton-up* e *top-down* para a exemplificação de construção de um sistema de *hardware*. O nível de abstração depende dos componentes disponíveis ou do nível de detalhamento que desejamos chegar no projeto.

Na abordagem *botton-up*, partimos de módulos pré existentes para que, a partir de interconexões, possamos criar estruturas mais complexas até culminar no sistema como um todo. Por sua vez, na abordagem *top-down*, os módulos são decompostos de modo a obtermos módulos mais simples. Esse processo de decomposição pode ser novamente aplicada até que se alcance o nível de complexidade desejada.

Mas o que fazer com esses módulos simples da abordagem *top-down*? De onde coletar os módulos para implementarmos o sistema na abordagem *botton-up*? A figura abaixo já começa a responder tais perguntas.

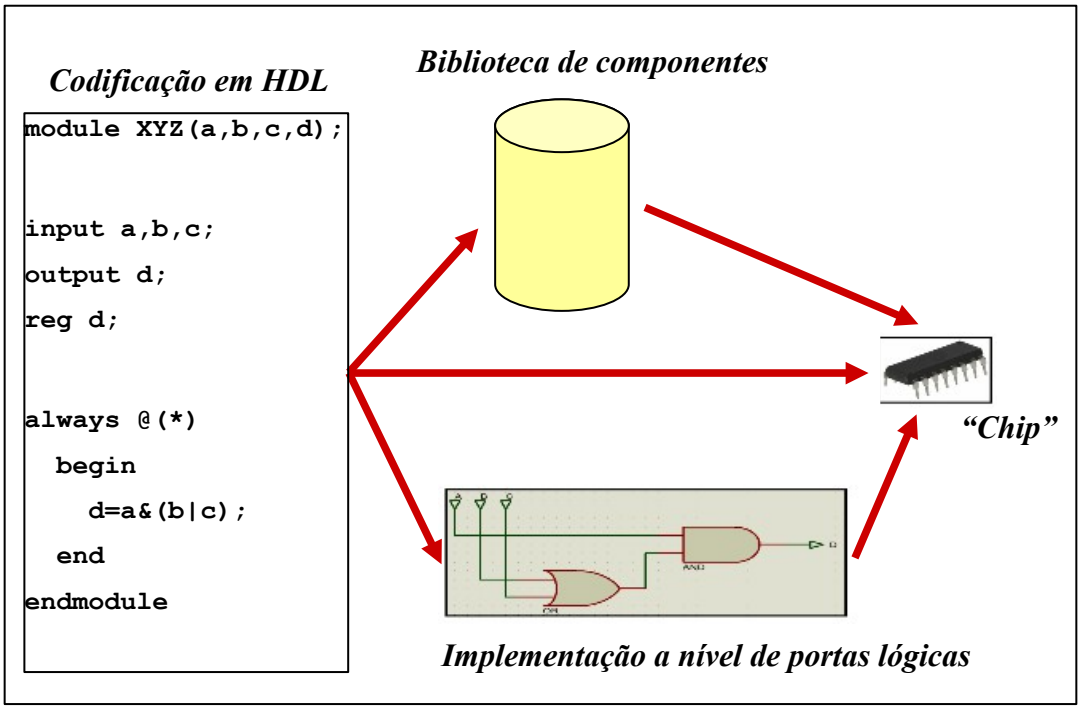


Figura 03 – Possíveis sequências para as abordagens *botton-up* e *top-down*.

Fonte: Elaborada pelo autor, 2020.

Na figura acima, a abordagem *top-down* pode ser representada pelo completo desenvolvimento do circuito desde os mais altos níveis de abstração (por exemplo, codificação em HDL – *Hardware Description Language* – Linguage de Descrição de *Hardware*), implementação a nível de portas lógicas e culminando no circuito físico final. Por outro lado, a codificação em HDL pode gerar um componente que poderá ser utilizado (e reaproveitado) na abordagem *botton-up* – tal componente integrará uma base de componentes (repositório de componentes). Nesta abordagem, o circuito é gerado através da interconexão de componentes já implementados (por exemplo, disponibilizados no repositório).

Os componentes criados e que poderão ser utilizados em diversos projetos são denominados como *IP Core* (*Intellectual Property Core* – Núcleo (ou Bloco) de Propriedade Intelectual).

Você sabia?

Existem projetos de IP Cores abertos que poderão ser utilizados nos mais diversos projetos e aplicações. Um dos exemplos consiste no repositório denominado OpenCores, acessado através do *link* <https://opencores.org/projects>.

Mencionamos o termo HDL. Mas, o que vem a ser uma Linguagem de Descrição de

Hardware? Trata-se de uma linguagem cujo objetivo consiste na criação de sistemas digitais. O produto obtido pela codificação de um *hardware* via HDL pode gerar, como exemplos: um *ASIC* (*Application Specific Integrated Circuit* – Circuito Integrado de Aplicação Específica), um *bitstream* para ser descarregado em uma *FPGA* (*Field Programmable Gate Array* – Arranjo de Portas Programáveis em Campo) ou, ainda, uma simulação do circuito.

Você quer ler?

Podemos citar, como exemplos de HDL, o Verilog e o HDL. Como trabalharemos em cima do *Verilog*, um tutorial para essa linguagem de descrição de *hardware* poderá ser obtido acessando o endereço <https://edisciplinas.usp.br/pluginfile.php/5550730/mod_assign/intro/Curso%20Basico%20de%20Verilog.pdf>.

Todos os circuitos que abordaremos ao longo de nosso semestre serão também codificados em Verilog. Optou-se por Verilog devido à sua maior facilidade de abstração – sua codificação em muito se assemelha à linguagem C de programação.

O desenvolvimento de processadores e demais sistemas digitais podem ser realizados por intermédio de ferramentas e ambientes integrados. Como exemplos, podemos citar:

- Quartus Prime (Altera), LeonardoSpectrum (Mentor Graphics) e Xilinx Vivado (Xilinx): ferramentas para realizar síntese de circuitos a partir de códigos HDL (Verilog e VHDL).
- ModelSim SE (Mentor Graphics) e Simulink (MathWorks): ferramentas de simulação de sistemas;
- IC Station (Mentor Graphics) e Design Architecture (Mentor Graphics): criação de *layouts* e diagramas esquemáticos
- Calibre (Mentor Graphics): verificações dos layouts criados pelo IC Station e Design Architecture.

Além destas ferramentas, podemos citar as desenvolvidas pela empresa Cadence. Entre as ferramentas, encontram-se OrCad/PSpice e o JasperGold.

Você quer ver?

Para possibilitar uma maior abstração dos pontos a serem conversados durante a nossa disciplina, trabalharemos com simulações tanto a nível de utilização de componentes eletrônicos digitais quanto a nível de utilização de Verilog. Para o início da ambientação, você poderá ver os

seguintes vídeos: <https://www.youtube.com/watch?v=QmHU5Cz9N1I> e <https://www.youtube.com/watch?v=6sA7wAqcFMA>. Nestes vídeos são abordados os seguintes ambientes: Quartus Prime, ModelSim e Icarus Verilog. Você também poderá ver os vídeos produzidos pelo laboratório CESLa (Circuits and Embedded Systems Lab), da Universidade Federal do Piauí, disponíveis no "Canal do CESLa" através do *link* <https://www.youtube.com/channel/UC044OtMEmF5QYyKIalIaaPw>. Além destes ambientes, trabalharemos também como o ambiente *online* <https://www.tinkercad.com/>.

Existem diversas outras ferramentas para possibilitar o desenvolvimento de sistemas digitais. Porém, não devemos esquecer de realizar análises e verificações dos projetos idealizados – para tanto, existem também ferramentas e técnicas, dentro as quais, a verificação formal.

Vamos praticar?

Acesse o link <https://opencores.org/projects> e escolha algum projeto de microprocessador. Analise a estruturação dos arquivos de codificação relacionando os módulos implementados.

Antes de iniciarmos a nossa conversa sobre circuitos digitais, vamos rever os conceitos de sistemas de numeração, mais especificamente o sistema binário?

2.2 Conversão entre sistemas de numeração

Antes de iniciarmos nossa conversa sobre conversão entre sistemas de numeração, convém falarmos sobre o que vem a ser um sistema de numeração e quais são os principais, do ponto de vista computacional. De acordo com (TOCCI, 2018), os sistemas numéricos mais comuns, sob o ponto de vista computacional, são: decimal; hexadecimal; octal; e binário. Mas, o que eles têm em comum? É o que você saberá a seguir, com a leitura deste tópico.

2.2.1 Sistemas de Numeração

Sistemas de numeração representam a maneira de como um valor numérico é formalizado, ou seja, representa as regras para a representação de um número. Bom, retomando: o que os sistemas decimal, hexadecimal, octal e binário têm em comum é que todos eles podem ser fatorados. Por exemplo, caso tenhamos um número "Q" de quatro dígitos representado

através de uma base hipotética qualquer denominada B, esse número poderia ser decomposto nos seguintes fatores:

$$Q_3 * B^3 + Q_2 * B^2 + Q_1 * B^1 + Q_0 * B^0$$

Sendo Q_3 o dígito mais significativo e, consequentemente, Q_0 o dígito menos significativo. Para ficar mais claro, vamos supor um número de quatro dígitos na nossa forma decimal de representação:

$$3546 = 3 * 10^3 + 5 * 10^2 + 4 * 10^1 + 6 * 10^0$$

Citando o exemplo acima de decomposição de um número, acabamos de mencionar a base decimal. A base decimal é a base que estamos acostumados a manipular em nosso cotidiano. Iniciando a nossa conversa pela base decimal, fica mais fácil abstrair dois conceitos: os símbolos admissíveis e sua quantidade.

- **Quantidade de símbolos admissíveis**: a quantidade de símbolos que podemos usar em cada dígito é derivado do próprio nome da base. Na base decimal, por exemplo, podemos representar um dígito com um dos 10 símbolos (ou valores) possíveis.
- **Símbolos admissíveis**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Os computadores não são eficientes para manipular valores na base decimal em função de sua forma de implementação, que é baseada em sistemas lógicos digitais. Em função de sua estrutura eletrônica, em que se admite apenas dois valores possíveis de seus sinais internos, utiliza-se a **base binária**.

E o que vem a ser sinais internos? Chamamos de “sinais internos”, no caso dos circuitos eletrônicos, os valores que trafegam nos fios e componentes eletrônicos que compõem o computador. Esses valores são vinculados às voltagens aplicadas sobre uma determinada parte do circuito. Geralmente, podemos associar a presença de tensão elétrica positiva como “nível lógico 1”, e a ausência de tensão elétrica como “nível lógico 0”. A figura a seguir ilustra esses dois níveis.

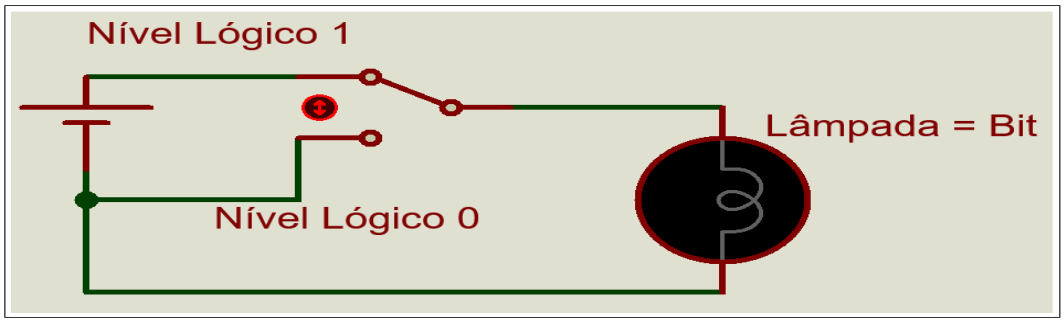


Figura 4 – Analogia dos níveis lógicos “0” e “1” com a tensão inserida em uma lâmpada. Lâmpada acesa denota que está recebendo o nível “1”, e, apagada, o nível lógico “0”.
Fonte: Elaborada pelo autor, 2020.

Na figura, a lâmpada representa um dígito de um valor binário – o chamado “bit” (***B**inary dig**IT***, ou dígito binário). Em tal situação, a lâmpada estará representando o nível “1” quando estiver acesa ou o nível “0” quando apagada. Assim, temos:

- quantidade de símbolos admissíveis: 2.
- símbolos admissíveis: 0, 1.

Podemos falar que um número binário é formado por vários *bits*. O conjunto de *bits*. chamamos de “palavra”. Por exemplo, uma palavra de 8 *bits* tem o tamanho (ou largura) de 1 *byte*. Mas, então, podemos realizar a decomposição sobre um valor escrito em binário? A base binária não foge à regra de decomposição em fatores. Por exemplo, supondo o valor binário $1101_{(2)}$, temos:

$$1101_{(2)} = \mathbf{1} * \mathbf{2}^3 + \mathbf{1} * \mathbf{2}^2 + \mathbf{0} * \mathbf{2}^1 + \mathbf{1} * \mathbf{2}^0$$

Conversamos sobre o que vem a ser um sistema de numeração, mas como realizar a conversão entre valores representados por bases numéricas distintas? Isso é o que dialogaremos a seguir.

2.2.2 – Conversão entre bases numéricas

Acabamos de falar sobre a decomposição de um valor na base binária. Retomando o exemplo:

$$1101_{(2)} = \mathbf{1} * \mathbf{2}^3 + \mathbf{1} * \mathbf{2}^2 + \mathbf{0} * \mathbf{2}^1 + \mathbf{1} * \mathbf{2}^0$$

O que se pode abstrair dessa decomposição? Inicialmente, podemos usar os valores de $\mathbf{2}^n$ para definir o nome dessa representação binária: BCD8421. O nome BCD significa "decimal codificado em binário" (*Binary-coded decimal*). O valor "8421" é obtido pelas exponenciações ($\mathbf{2}^3 = 8$; $\mathbf{2}^2 = 4$; $\mathbf{2}^1 = 2$; $\mathbf{2}^0 = 1$).

Você quer ler?

Existem outras formas de representação numérica usando sistema binário. Para se adequar a outras aplicações ou circunstâncias, um valor numérico pode ser representado de formas distintas por meio de outros sistemas de representação binários. Para saber um pouco mais, você poderá acessar o link: https://wiki.sj.ifsc.edu.br/wiki/index.php/DIG222802_AULA07.

Voltando à decomposição acima, podemos realizar uma outra abstração: qual o resultado obtido se realizarmos a soma "$\mathbf{1} * \mathbf{2}^3 + \mathbf{1} * \mathbf{2}^2 + \mathbf{0} * \mathbf{2}^1 + \mathbf{1} * \mathbf{2}^0$"? O resultado dessa expressão vale "13". Então, podemos falar que:

$$1101_{(2)} = 13_{(10)}$$

Esse processo pode ser aplicado para qualquer base de numeração quando desejamos transformar o número expresso em base decimal. Para demonstrar isso, vamos falar sobre dois outros sistemas de numeração: o sistema hexadecimal e o sistema octal. O sistema hexadecimal é muito utilizado pelos programadores e projetistas de sistemas computacionais por representar um número de forma mais condensada em relação ao sistema decimal ou ao sistema binário de representação numérica.

- Quantidade de símbolos admissíveis: 16.
- Símbolos admissíveis: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Observando os símbolos admissíveis, você notará a presença das letras "A" a "F". Nesse caso, se fizermos uma correspondência direta com o sistema decimal, temos a faixa de "A" representando o valor 10 até a letra "F" denotando o valor "15". A conversão de hexadecimal para decimal segue o mesmo processo de decomposição. Vamos supor o valor: $3A8C_{(16)}$:

$$3A8C_{(16)} = 3 * 16^3 + 10 * 16^2 + 8 * 16^1 + 12 * 16^0 = 12288 + 2560 + 128 + 12 = 14988_{(10)}$$

Por fim, vamos demonstrar que esse processo de decomposição também serve para realizar a conversão de um valor representado na base octal para binário. O sistema octal foi muito usado pelos programadores e projetistas de sistemas computacionais quando a capacidade dos equipamentos era extremamente menor em relação aos recursos computacionais atualmente disponíveis. Por esse motivo, grave bem, o sistema octal cedeu espaço para o sistema hexadecimal. Em relação às suas características básicas, temos:

- quantidade de símbolos admissíveis: 8.
- símbolos admissíveis: 0, 1, 2, 3, 4, 5, 6, 7.

Para proceder à conversão, vamos supor o valor $5461_{(8)}$:

$$5461_{(8)} = 5 * 8^3 + 4 * 8^2 + 6 * 8^1 + 1 * 8^0 = 2560 + 256 + 48 + 1 = 2865_{(10)}$$

Até o momento, falamos sobre o processo de conversão da base "Q" para a base decimal. Mas como realizar o processo contrário? Como converter um valor expresso na base decimal para a base Q? Basta realizar divisões sucessivas do valor decimal pela base em questão.

A figura a seguir ilustra o processo de conversão dos valores acima calculados para as suas bases de origem.

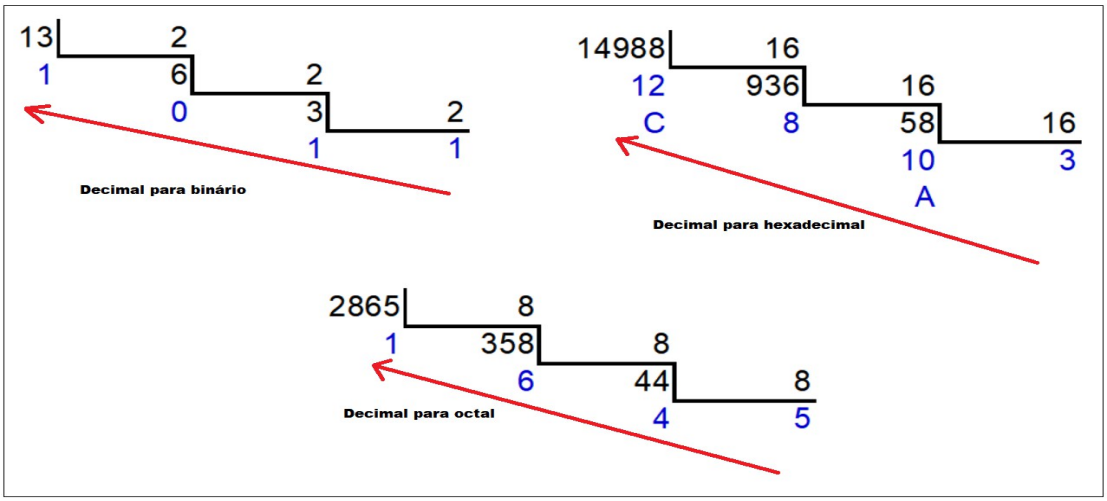


Figura 5 – Processo de divisão de números representados na base decimal para as bases binária, hexadecimal e octal, utilizando-se divisões sucessivas.

Fonte: Elaborada pelo autor, 2020.

Na figura anterior, podemos notar que o valor resultante da conversão é construído coletando-se o último quociente e todos os restos da divisão na ordem inversa de aparição.

Mencionamos que as bases octal e hexadecimal são usadas principalmente por programadores e projetistas de sistemas computacionais, certo? Como elas são mais próximas do sistema binário, será que existe uma forma mais fácil para realizar a conversão entre binário para hexadecimal ou octal e hexadecimal ou octal para binário? Para respondermos essa questão, vamos analisar a quantidade de *bits* necessária para se representar um valor em hexadecimal ou octal:

- Hexadecimal:
  - Quantidade de símbolos admissíveis: 16
  - Quantidade de *bits* necessários para cada dígito: $\log_2(16) = 4$
- Octal:
  - Quantidade de símbolos admissíveis: 8
  - Quantidade de *bits* necessários para cada dígito: $\log_2(8) = 3$

Assim, cada dígito hexadecimal gerará uma sequência de quatro *bits*, e cada dígito octal será convertido em três bits. A figura a seguir exemplifica o processo de conversão entre essas bases de representação numérica tomando-se como exemplo o valor binário de 16 *bits*: "1011011010101101".

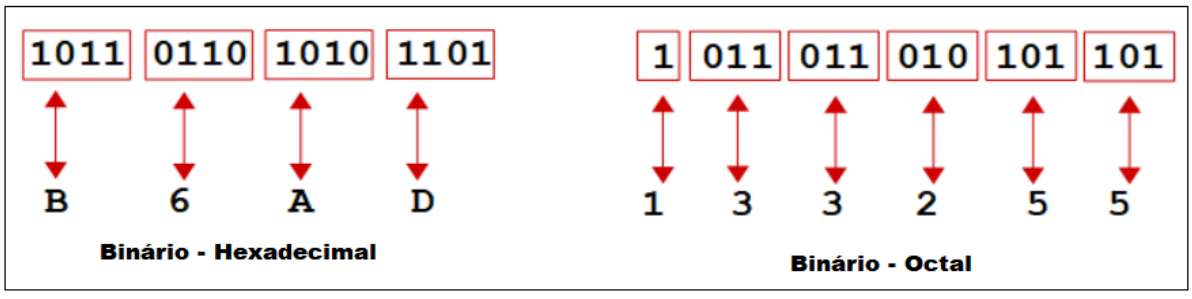


Figura 6 – Conversão entre a base binária e as bases hexadecimal e octal. O processo de agrupamento de 4 ou 3 *bits* é realizado nos dois sentidos da conversão.

Fonte: Elaborada pelo autor, 2020.

Na figura, perceba que há relação entre os agrupamentos de quatro ou três *bits* para cada dígito hexadecimal ou octal, respectivamente. O valor binário é representado de acordo com o padrão BCD8421.

Mas o computador manipula números binários representando valores positivos e negativos. Como ele consegue diferenciar essas faixas de valores? Para responder essa questão, podemos falar que, dentro da representação binária, merece destaque um sistema conhecido como "complemento 2".

Para uma melhor abstração, vamos começar a falar sobre complemento 2 exemplificando com o tipo "**char**" da linguagem "C" de programação. Esse tipo comumente é representado por uma palavra de 8 *bits* (1 *byte*). Assim, temos 2^8 valores possíveis, ou seja, 256 valores que podem ser representados em uma variável do tipo "**char**".

Mas quais são esses valores? Porque essa faixa vai de -127 a 128? Porque que se utilizássemos na criação da variável o modificador de tipo "**unsigned**" (ficando "**unsigned char**"), a faixa iria de 0 a 255? Tudo isso ocorre devido ao fato de que o *bit* mais significativo (MSB – *Most Significant Bit*), nos casos em que não há a utilização do "**unsigned**", representa o sinal.

Dessa forma, dos oito *bits* do "**char**", temos um *bit* de sinal e sete contendo a informação propriamente dita. Consequentemente, 2^7 denota que temos 128 valores possíveis: 128 valores negativos (de -128 a -1) e 128 valores positivos (de 0 a 127).

E como representar um valor negativo? Bem, vamos tomar como exemplo prático, para facilitar a apresentação e os cálculos, um valor de quatro bits "**$1110_{(2)}$**". De acordo com o que foi apresentado, temos:

- Bit MSB (*bit* sinal): "**1**" → indicando que o valor é negativo. Caso o MSB fosse 0, o número representado seria positivo.
- Valor propriamente dito: "**$110_{(2)}$**". Caso o *bit* sinal fosse 0 (indicando um número positivo), poderíamos aplicar o BCD8421 diretamente sobre "**$110_{(2)}$**". Porém, como o *bit* sinal vale "**1**", não podemos falar que o valor representado é "**$-6_{(10)}$**".

Nesse caso, com o *bit* sinal indicando um valor negativo, para se conhecer o valor, temos que aplicar dois passos:

- Passo 1: inverte-se todos os *bits* do valor representado: "**$1110_{(2)}$**" → "**$0001_{(2)}$**".
- Passo 2: soma-se ao valor obtido da inversão dos *bits*: "**$0001_{(2)}$**" + "**$0001_{(2)}$**" = "**$0010_{(2)}$**" (1 + 1 = 2).

Assim, entendemos que o valor "**$1110_{(2)}$**" equivale a "**$-2_{(10)}$**".

Para achar o correspondente negativo a partir de um valor positivo ou para achar o correspondente positivo a partir de um valor negativo, segue-se os mesmos dois passos apresentados acima. Porém, tenha em mente que o computador não precisa realizar tais operações. O valor em complemento 2 já é diretamente obtido em suas operações, por exemplo, de subtração.

Como mencionamos, um computador manipula palavras expressas no sistema binário de representação numérica. Assim, como temos uma álgebra matemática para manipularmos o nosso sistema decimal, deve haver uma álgebra adequada para a manipulação de valores binários. A seguir, conversaremos sobre a álgebra booleana.

2.3 Álgebra booleana: simbologia e teoremas

Conhecer sobre a álgebra booleana não somente é importante para abstrair o funcionamento dos sistemas computacionais, mas também quando estivermos modelando os sistemas digitais. A princípio, um sistema digital pode ser representado por meio de uma expressão booleana, além de podermos utilizar outros elementos que serão vistos oportunamente.

2.3.1 – Variáveis e expressões booleanas

De acordo com (Vahid, 2008), uma álgebra objetiva a representação de valores por meio de letras e símbolos. Mas o que vem a ser tais letras e símbolos? Para facilitar nossa abstração, vamos supor um exemplo prático. Suponha que a iluminação de um cômodo de uma casa tenha acendimento automático de acordo com a seguinte condição:

```
Se (alguém_presente_no_cômodo E já_anoiteceu) então luz = acesa
Senão luz = apagada
```

No nosso trecho apresentado de pseudocódigo, podemos identificar variáveis de entrada e saída.

- **Variáveis de entrada**: as variáveis de entrada poderão, nesse caso, ser relacionadas a um sensor de presença (P) e um sensor de luminosidade (L). Ambos os sensores entregarão, ao sistema digital, valores binários.

- **Variável de saída**: essa variável é associada a uma saída (S) conectada a uma lâmpada que será acesa ou permanecerá apagada.

Note que todos os itens envolvidos manipulam valores binários. Então, poderemos reescrever o pseudocódigo usando uma expressão booleana:

S = P "AND" L

Dessa forma, podemos identificar os primeiros elementos da álgebra booleana:

- Expressão booleana: **S = P "AND" L**
- Variáveis booleanas: **S ; P ; L**
- Operador lógico: **AND**

Você o conhece?

Sistemas lógicos digitais e a computação como um todo são baseados na manipulação de valores binários. Valores binários remetem à lógica booleana e seria impossível falar de álgebra booleana sem falar sobre George Boole, o idealizador da álgebra booleana. Para saber um pouco da vida dele, você poderá ler (SOUSA, 2005), disponível em <https://repositorio.ufrn.br/jspui/bitstream/123456789/14224/1/GiselleCS.pdf>.

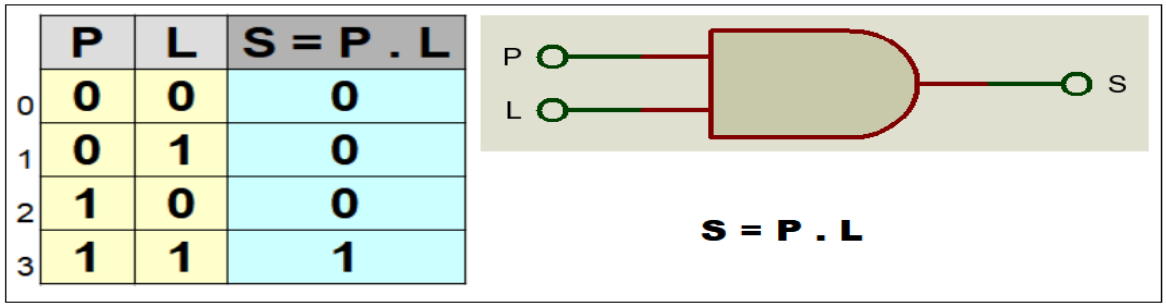
Mencionamos, a pouco, o termo "operador lógico". Mas, o que vem a ser um operador lógico? Conversaremos sobre isso a seguir.

2.3.2 Operadores lógicos

O que vem a ser operador lógico? Um operador lógico realiza a conexão entre duas variáveis para que seja estabelecido um resultado. Os operadores lógicos básicos são: "NOT", "AND", "OR" e, em sistemas digitais, são representados pelas "**portas lógicas**". Assim, podemos falar que as portas lógicas são os elementos de abstração de mais baixo nível em sistemas digitais.

Antes de falarmos sobre as propriedades e teoremas da álgebra booleana, vamos conversar sobre as **portas lógicas**. Convém mencionarmos que toda porta lógica estará associada a uma representação em diagrama esquemático (a notação utilizada no desenho do circuito), uma

simbologia na álgebra booleana (para representar a porta nas expressões booleanas) e uma tabela-verdade (que contempla todas as combinações possíveis dos valores que poderão ser assumidos pelas variáveis de entrada). Assim, para antecipar, podemos, também, representar o exemplo dado anteriormente (do acendimento automático da luz) por meio da expressão booleana, diagrama esquemático e tabela-verdade. Tais elementos são contemplados na figura a seguir.



	P	L	S = P . L
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

Figura 7 – Tabela-verdade, expressão booleana e diagrama esquemático do sistema lógico digital para o acendimento automático da luz.
Fonte: Elaborada pelo autor, 2020.

Na figura, perceba, temos a tabela-verdade, a expressão booleana e o diagrama esquemático do sistema lógico digital para o acendimento automático da luz em função das variáveis “P” (presença de alguma pessoa no recinto) e “L” (intensidade da luminosidade do dia). Para que a luz se acenda, deve haver alguma pessoa no recinto “E” e a luminosidade externa ao recinto deve estar baixa.

Note, ainda, que a tabela-verdade (à esquerda da figura) representa todas as combinações possíveis das variáveis de entrada — combinação 0 até combinação 3. O número de linhas da tabela-verdade é representado por 2^n, em que *n* é o número de variáveis de entrada envolvidas. Os valores inseridos na tabela representam a própria sequência numérica no formato BCD8421.

Você quer ver?

Podemos dizer que a base para a implementação de sistemas lógicos digitais é a construção da tabela-verdade. A tabela-verdade reflete os resultados da expressão booleana, levando-se em conta todas as combinações possíveis das variáveis de entrada. Para saber um pouco mais sobre como preencher uma tabela-verdade, assista ao vídeo: https://www.youtube.com/watch?v=mVJXZit3msA.

Mas o que realmente representam tais símbolos? Veremos, agora, as representações das

portas lógicas – as portas lógicas correspondem aos próprios operadores lógicos.

2.3.2.1 Porta NOT

A porta “NOT” realiza a inversão (ou complemento) da variável apresentada à sua entrada. Trata-se, portanto, de uma porta unária, ou seja, é representada por uma função que envolve apenas uma variável de entrada.

- Simbologias: S = ~A ; S = A'; S = $\overline{A}$; S = ¬A

A figura a seguir ilustra o diagrama esquemático e a tabela-verdade referente à porta “NOT”.

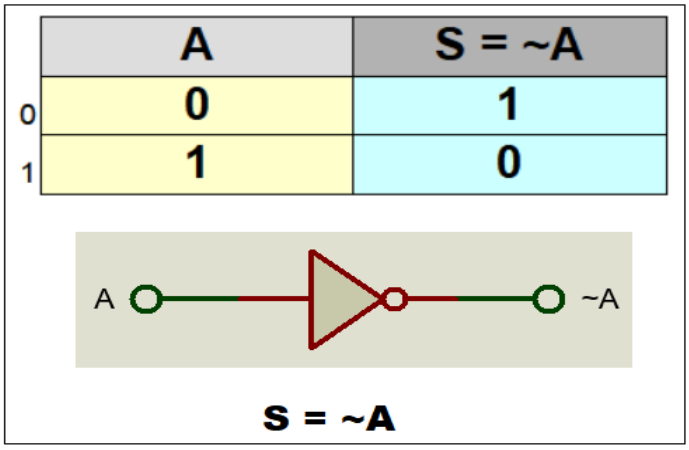


Figura 8 – Diagrama esquemático e tabela-verdade de uma porta “NOT”

Fonte: Elaborada pelo autor, 2020.

O diagrama esquemático da porta “NOT” também pode ser denotado apenas com a circunferência quando associado a alguma outra estrutura ou porta lógica. Em Verilog, o operador NOT é representado de forma idêntica ao operador *bitwise* em C, ou seja, através do símbolo “~”. Desta forma, tem-se, por exemplo, “`S = ~A;`” (S recebe NOT A).

2.3.2.2 Portas AND e OR

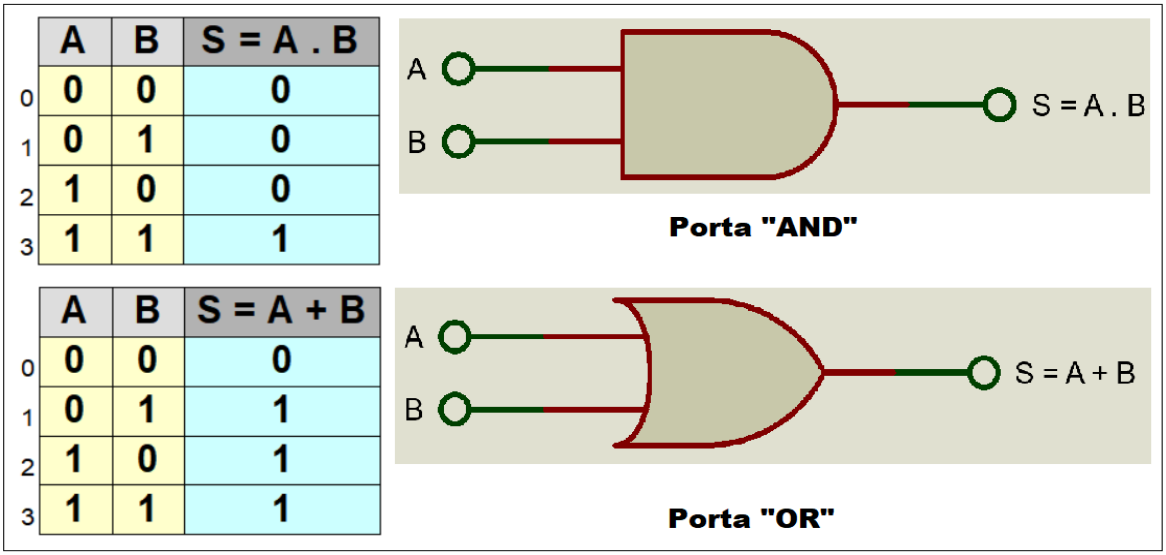
A porta “AND” é uma porta binária, ou seja, para externar o seu valor de saída, são necessárias duas variáveis de entrada. A saída será “1” apenas se as duas entradas também forem “1” simultaneamente.

- Simbologias da porta AND: $S = A \cdot B$; $S = A\,B$; $S = A \wedge B$

Na simbologia da porta “AND”, o símbolo de “.” pode ser omitido. Assim, por exemplo, a expressão $S = A \cdot B$ também poderá ser escrita na forma $S = AB$. Por sua vez, a porta “OR” é, assim como a porta “AND”, uma porta binária. A saída será “0” apenas se as duas entradas também forem “0” simultaneamente.

- Simbologias da porta “OR”: $S = A + B$; $S = A \vee B$

A figura a seguir ilustra os diagramas esquemáticos e as tabelas-verdade referentes às portas “AND” e “OR”.



	A	B	S = A . B
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

	A	B	S = A + B
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

Figura 9 – Tabelas-verdade e diagramas esquemáticos relativos às portas “AND” e “OR”
Fonte: Elaborada pelo autor, 2020.

Assim como o operador “NOT”, os operadores “AND” e “OR” também possuem as suas representações em Verilog conforme as linhas a seguir:

- AND: `S = A & B;`
- OR: `S = A | B;`

As portas “AND” e “OR” poderão ser conectadas a uma porta “NOT” em suas saídas, gerando as portas universais “NAND” e “NOR”, respectivamente. Saiba mais sobre elas a seguir.

2.3.2.3 Portas NAND e NOR

Como mencionado anteriormente, as portas “NAND” e “NOR” são portas ditas “universais”, ou seja, podemos usar somente portas “NAND” ou somente portas “NOR” para implementar nosso sistema lógico digital. Os valores de saída das suas tabelas-verdade são, portanto, os valores complementados das saídas das portas “AND” e “OR”, respectivamente.

A figura a seguir ilustra os diagramas esquemáticos e as tabelas-verdade referentes às portas “NAND” e “NOR”.

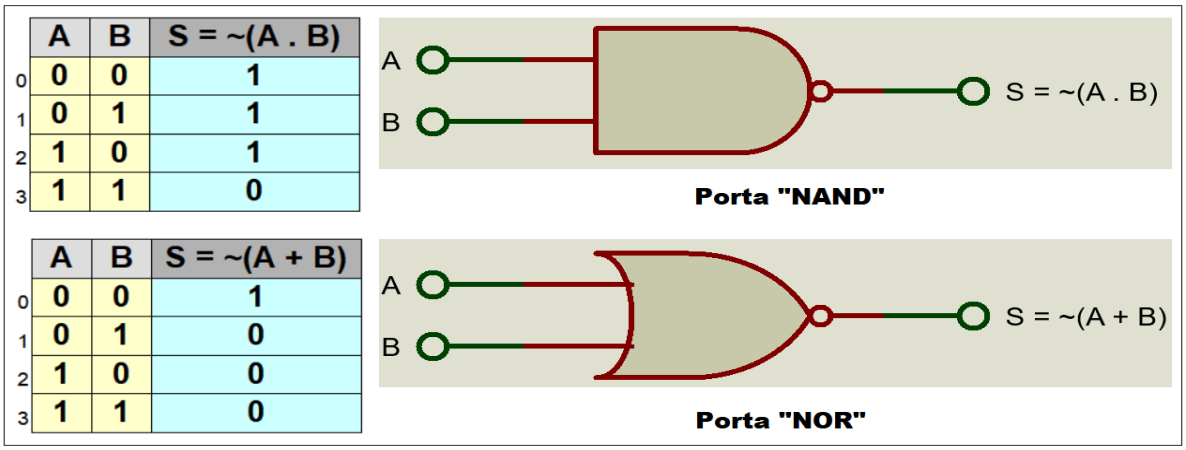


Figura 10 – Tabelas-verdade e diagramas esquemáticos relativos às portas “NAND” e “NOR”

Fonte: Elaborada pelo autor, 2020.

Como você pode notar nos diagramas esquemáticos da figura anterior, o sinal da negação anexado às saídas das portas “AND” e “OR” consiste apenas na circunferência. Como mencionado anteriormente, caso a negação esteja atrelada a alguma outra estrutura, para se representar a negação, basta utilizar a circunferência sem a seta indicativa da direção do fluxo do sinal (entrada → saída).

2.3.2.4 Portas XOR e XNOR

Por fim, temos as portas “EXCLUSIVE-OR” (OU-EXCLUSIVO ou XOR) e a porta “COINCIDÊNCIA” (ou XNOR). A porta “XOR” apresenta a saída valendo 1 se, e somente se, tivermos um único valor “1” representado pelas suas entradas. Por outro lado, a porta “XNOR” retrata o inverso da porta “XOR”, ou seja, a saída será “1” caso os dois valores de entrada sejam iguais.

A figura a seguir ilustra os diagramas esquemáticos, as simbologias e as tabelas-verdade referentes às portas “XOR” e “XNOR”.

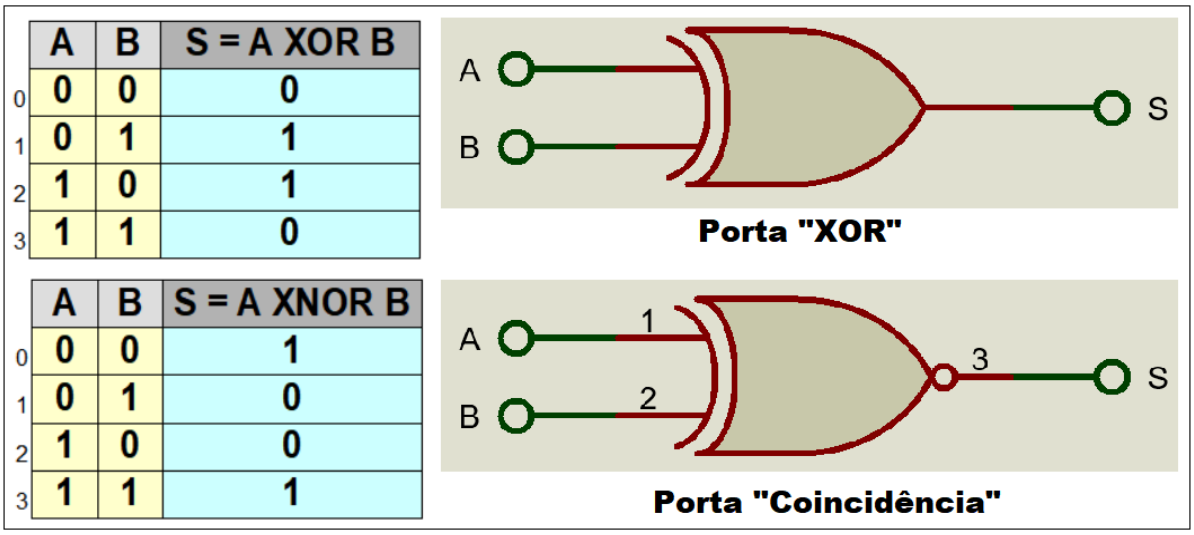


Figura 11 – Tabelas-verdade e diagramas esquemáticos relativos às portas “XOR” e “XNOR”

Fonte: Elaborada pelo autor, 2020.

Na figura anterior, podemos observar o diagrama esquemático da porta “XNOR” como sendo a porta “XOR” complementada. Porém, podemos encontrar também a porta “XNOR” sendo representada como uma porta “AND” com linha dupla junto às suas entradas.

Em Verilog, existe apenas a correspondência direta ao operador “XOR”:

- XOR: `S = A ^ B;`

Por sua vez, o operador coincidência poderá ser implementado negando-se o operador “XOR”:

- XNOR: `S = ~(A ^ B);`

Você sabia?

Você sabia que a maioria dos operadores vistos aqui podem ser aplicados na programação para realizar a manipulação “bit-a-bit”? Essa técnica, denominada “*bitwise*” permite que, por exemplo, *bits* sejam alterados dentro de uma variável. Para saber mais, você poderá acessar a apostila escrita por (RENNA, 2014), disponível em <https://www.telecom.uff.br/pet/petws/downloads/apostilas/arduino/apostila_de_programacao_a

rduino.pdf>.

As portas aqui mencionadas farão parte das expressões booleanas (representando os operadores lógicos ou operadores booleanos). Para que possamos manipular as expressões, teremos que aplicar os teoremas da álgebra booleana, os quais veremos a seguir.

2.3.3 Teoremas da Álgebra Booleana

Os teoremas da álgebra booleana nortearão a manipulação das expressões booleanas para, por exemplo, desenhar o diagrama esquemático ou calcular o valor resultante a partir da combinação dos valores das variáveis de entrada. Para começarmos, podemos mencionar a precedência dos operadores:

- 1º: parênteses
- 2º negação
- 3º operador "AND"
- 4 º operador "OU", "XOR", "XNOR"

Assim, ao nos depararmos com, por exemplo, a expressão booleana "**S = (~A + B) . C**", teremos, o diagrama esquemático e a tabela-verdade ilustrados a seguir.

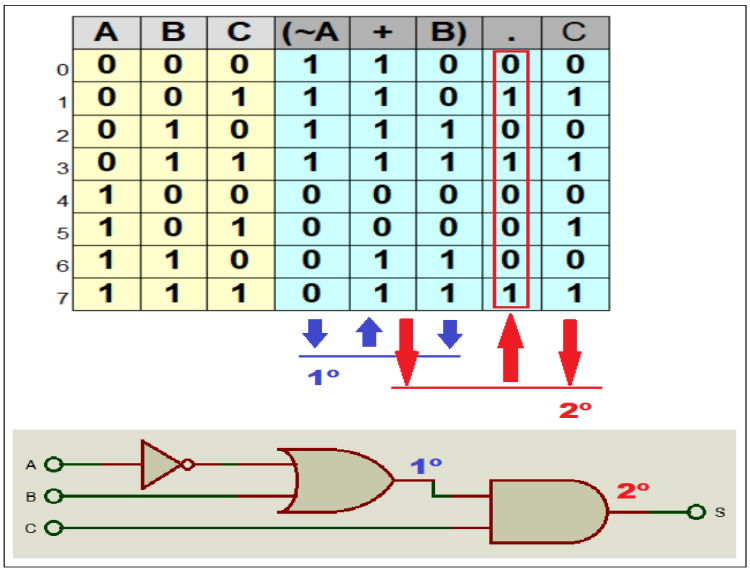


Figura 12 – Tabela-verdade e diagrama esquemático relativos à expressão **S = (~A + B) . C**

Fonte: Elaborada pelo autor, 2020.

A figura anterior ilustra a tabela-verdade e o diagrama esquemático relativo à expressão "**S = (~A + B) . C**". Perceba que a ordem de manipulação dos operadores segue a precedência da álgebra booleana. A presença de uma inversão ligada à uma variável faz com que, no caso, a coluna relativa à "~A" receba o complemento de "A". Em função dos parênteses, a primeira operação a ser realizada consiste no operando "OR" (marcado em azul e denotado por "1º"). Após o preenchimento da coluna referente ao resultado do operador "OR", efetua-se o operador "AND" — marcado na cor vermelha e referenciado como "2º" passo a ser tomado. A coluna referente ao resultado do operador "AND" (delimitada pelo retângulo vermelho) representa os resultados da expressão booleana, tendo em vista todas as combinações possíveis envolvendo as variáveis de entrada "A", "B" e "C".

Além das precedências, temos que nos atentar que, assim como a álgebra matemática, a álgebra booleana também apresenta propriedades comutativa, associativa e distributiva:

- **Propriedade Comutativa**: a ordem das variáveis não interfere no resultado produzido. Exemplos:

  A . B = B . A

  A + B = B + A

  A ⊕ B = B ⊕ A

- **Propriedade Associativa**: caso tenhamos variáveis conectadas por meio do mesmo tipo de operador, poderemos criar grupos (associar) para que possamos definir a ordem de manipulação. Exemplos:

  (A . B) . C = A . (B . C) = A . B . C

  (A + B) + C = A + (B + C) = A + B + C

  (A ⊕ B) ⊕ C = A ⊕ (B ⊕ C) = A ⊕ B ⊕ C

- **Propriedade Distributiva**: na propriedade distributiva, procedemos à distribuição não somente da variável, mas também do operador associado à variável. Exemplos:

  A . (B + C) = A . B + A . C

  A + B . C = (A + B) . (A + C)

Além das propriedades citadas, uma quarta propriedade merece destaque na manipulação de expressões booleanas. Trata-se do "Teorema de De Morgan". Este teorema pode ser aplicado

quando um operador “NOT” incidir, além das variáveis, sobre o próprio operador “AND” ou “OR”. Nesse caso, temos que proceder, além da inversão das variáveis, à alteração do próprio operador, trocando o “AND” pelo “OR” e vice-versa:

a) ~(A + B) = ~A . ~B
b) ~(A . B) = ~A + ~B

Nesses casos, temos, em (a), a incidência da negação sobre o operador “OR”. Nota-se que, após a aplicação do Teorema de De Morgan, o operador “OR” foi alterado para “AND” e as variáveis foram complementadas. Em (b), a negação abrange o operador “AND”, razão pela qual apareceu, após a aplicação do teorema, o operador “OR”.

Além dos teoremas e das propriedades, a álgebra booleana também contempla os seguintes postulados (IDOETA, 2019):

- **Complemento**: pode ser aplicado sobre um valor booleano ou envolvendo operadores. No caso de valores lógicos e das portas “AND” e “OR”, temos:

  ~0 = 1
  ~1 = 0
  A . ~A = 0
  A + ~A = 1

- **Idempotência**: envolve a aplicação de um número par de negações resultando na própria variável envolvida:

  ~(~ A) = A

- **Identidade**: ao aplicarmos um operador “AND” ou “OR” sobre apenas uma variável temos, como resultado, a própria variável:

  A + A = A
  A . A = A

- **Elemento neutro**: o elemento neutro representa o valor lógico (0 ou 1) que, quando aplicado a um operador, não modifica o valor resultante:

  A + 0 = A
  A . 1 = A

- **Elemento de absorção**: o elemento de absorção, ao ser aplicado sobre um operador, faz com que, independentemente do valor da variável envolvida, o operador resulte no próprio valor aplicado:

  A + 1 = 1

  A . 0 = 0

Se aplicarmos as propriedades e postulados citados, podemos abstrair as seguintes propriedades:

A + A . B = A

A + ~A . B = A + B

(A + ~B) . B = A . B

A . B + A . ~B = A

(A + B) . (A + ~B) = A

A . (A + B) = A

A . (~A + B) = A . B

A .B + ~A . C = (A + C) . (~A + B)

Juntando o Teorema de De Morgan com a idempotência, conseguimos uma aplicabilidade prática: reescrever a expressão de forma mais simples. Por exemplo, imagine que temos a seguinte expressão lógica:

$$S = \sim A \,.\, \sim B$$

Nessa expressão, temos operador "NOT" sendo aplicado a cada variável. Assim, podemos aplicar os seguintes passos:

1. **~A . ~B = ~(~(~A . ~B))** → nega-se a expressão duas vezes para que, de acordo com a idempotência, não se altere o valor da expressão.
2. **~(~(~A . ~B)) = ~(~~A + ~~B)** → aplica-se o Teorema de De Morgan na inversão mais interna.
3. **~(~~A + ~~B)= ~(A + B)** → aplica-se a idempotência nas inversões que precedem as variáveis "A" e "B".
4. **S = ~(A + B)** → expressão reescrita usando apenas um operador "NOR".

Foi mencionado, anteriormente, que operadores “NAND” e “NOR” são ditos como universais. Dessa forma, aplicando os princípios vistos aqui, vamos representar os operadores básicos utilizando-se somente “NAND” e “NOR”:

- Operador “NOT”:

  $S = \sim A \rightarrow S = \sim(A \,.\, A)$

  $S = \sim A \rightarrow S = \sim(A + A)$

- Operador “AND”:

  $S = A \,.\, B \rightarrow \sim(\sim(A \,.\, B) \,.\, \sim(A \,.\, B))$

  $S = A \,.\, B \rightarrow \sim(\sim(A + A) + \sim(B + B))$

- Operador “OR”:

  $S = A + B \rightarrow \sim(\sim(A \,.\, A) \,.\, \sim(B \,.\, B))$

  $S = A + B \rightarrow \sim(\sim(A + B) + \sim(A + B))$

Para finalizar, convém mencionar um fato interessante na álgebra booleana: o princípio da dualidade. Esse princípio menciona que a partir de uma expressão booleana, ao efetuarmos a troca dos operadores lógicos (de “AND” para “OR” e vice-versa), além de trocarmos os valores de “0” para “1” e vice-versa, mantemos a equivalência entre as duas expressões booleanas.

Como exemplos, podemos citar as expressões booleanas abaixo:

$\mathbf{A + 0 = A}$	$\leftrightarrow$	$\mathbf{A \,.\, 1 = A}$
$\mathbf{A + 1 = 1}$	$\leftrightarrow$	$\mathbf{A \,.\, 0 = 0}$
$\mathbf{A + A = A}$	$\leftrightarrow$	$\mathbf{A \,.\, A = A}$
$\mathbf{A + \sim A = 1}$	$\leftrightarrow$	$\mathbf{A \,.\, \sim A = 0}$

Com a aplicação das propriedades citadas, já começamos a enveredar pelo campo da otimização, simplificação ou otimização de expressões booleanas. Simplificar uma expressão resulta em várias melhorias as realizarmos a implementação física dos sistemas lógicos digitais.

Vamos praticar?

Conversamos sobre como preencher uma tabela-verdade produzindo, inclusive, o resultado da expressão booleana para cada combinação das variáveis de entrada. Construa a tabela-verdade das expressões a seguir:

a) S = A + ~B.C

b) S = A.(B + C) + ~A(C + ~D)

c) S = A + ~A.B

d) S = (A ⊕ B).(A + ~B)

Síntese

Chegamos ao fim desta unidade. Nesse nosso diálogo, você teve o primeiro contato com o mundo no qual se encontram os sistemas lógicos digitais. Inicialmente, abordamos os sistemas de numeração e como fazemos para realizar a conversão dentre os sistemas mais importantes do ponto de visto computacional: decimal; hexadecimal; octal; e binário. Como você pôde ver, essa área é baseada na manipulação de palavras cujos elementos (dígitos) são representados por *bits*. Por ser baseado na manipulação de *bits*, temos que conhecer, entender e manipular a álgebra booleana, seus operadores, propriedades e postulados. Na parte prática, os operadores lógicos são representados pelas portas lógicas, componentes básicos da eletrônica digital.

Aqui, você teve a oportunidade de:

- ter um contato inicial com sistemas de numeração
- conhecer, compreender e manipular informações utilizando a álgebra booleana.
- construir tabelas-verdade para representar o comportamento de uma expressão booleana em função de todas as combinações possíveis de suas variáveis de entrada.
- identificar e saber utilizar as propriedades, operadores e postulados.

Bibliografia

CEFOR – Instituto Federal do Espírito Santo. **Videoaula – Criação de Tabela-verdade**. Publicado em 12/05/2014. Disponível em <https://www.youtube.com/watch?v=mVJXZit3msA>. Acesso em 18/10/2020.

CHENG, S.; AZARIAN, M. H.; PECHT, M. G. Sensors Systems for Prognostics and Health Management. Publicado em 08/06/2010. Disponível em <https://www.mdpi.com/1424-

8220/10/6/5774/pdf>. Acessado em 17/10/2020.

IDOETA, I.V.; CAPUANO, F. G. **Elementos de Eletrônica Digital**. 42 Ed. São Paulo: Érica, 2019. Disponível na Minha Biblioteca <https://integrada.minhabiblioteca.com.br/#/books/9788536530390>. Acessível via Minha UFOP – Biblioteca Digital).

IFSC. **DIG222802 AULA07**. Publicado em 01/06/2016. Disponível em <https://wiki.sj.ifsc.edu.br/wiki/index.php/DIG222802_AULA07>. Acessado em 18/10/2020.

MATTOS, W. **Tabelas-verdade Obtidas a partir de Expressões Booleanas**. Publicado em 02/12/2015. Disponível em <https://www.youtube.com/watch?v=cNu0wQk7f0Q>. Acessado em 18/10/2020.

RENNA, R. B.; PAIVA, L. M. **Apostila de Programação (Versão: A2014M05D02)**. Niterói: 2014. Universidade Federal Fluminense. Apostila de Tópicos Especiais em Eletrônica II – Introdução ao microcontrolador Arduino. Publicado em 02/05/2014. Disponível em <https://www.telecom.uff.br/pet/petws/downloads/apostilas/arduino/apostila_de_programacao_arduino.pdf>. Acessado em 18/10/2020.

SOUSA, G. S. **Uma Reavaliação do Pensamento Lógico de George Boole à Luz da História da Matemática**. Publicado em 14/10/2005. Disponível em <https://repositorio.ufrn.br/jspui/bitstream/123456789/14224/1/GiselleCS.pdf>. Acessado em 18/10/2020.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas Digitais: Princípios e Aplicações**. 12 Ed. São Paulo: Pearson Education do Brasil, 2018. Disponível na Biblioteca Virtual Pearson <https://plataforma.bvirtual.com.br/Acervo/Publicacao/168497>. Acessível via Minha UFOP – Biblioteca Digital).

VAHID, F.; LASCHUK, A. **Sistemas Digitais: Projeto, otimização e HDLs.** Porto Alegre: Bookman, 2008. Disponível na Minha Biblioteca <https://integrada.minhabiblioteca.com.br/#/books/9788577802371>. Acessível via Minha UFOP – Biblioteca Digital).