Eletrônica Digital, Verilog e FPGA

João Ranhel

1ª. Edição

São Paulo, 2021.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Ranhel, João
Eletrônica digital, VERILOG e FPGA / João Ranhel. -- 1. ed. -- São Paulo : Ed. do Autor, 2021.

ISBN 978-65-00-17065-8

1. Engenharia 2. Computação 3. Eletrônica digital 4. Eletrônica digital - Processamento de dados 5. FPGA (Circuitos) 6. Sistemas 7. Verilog - Linguagem de computador I. Título.

21-56104 CDD-621.395

Índices para catálogo sistemático:

1. Sistemas digitais em Verilog e FPGA : Engenharia eletrônica 621.395

Aos meus alunos, monitores dessa disciplina, e colegas professores que compartilharam e compartilham o ministério da disciplina Eletrônica Digital na Universidade Federal do Pernambuco e na Universidade Federal do ABC (UFABC).

Agradeço a meus amigos professores por valiosas sugestões e revisões no texto: João Paulo Cerquinho Cajueiro, Sandro Rodrigo Gonçalves Bastos, José Rodrigues de Oliveira Neto, Helói Francisco Gentil Genari, Cacilda Vilela de Lima, e ao Prof. Marcio Lobo Netto pela escrita do prefácio dessa edição. Mesmo sem mencioná-los, sou muito grato a todos aqueles que tenham contribuído de alguma forma para a execução desse projeto...

João Ranhel

Prefácio

Os avanços tecnológicos na área da engenharia eletrônica têm sido muito rápidos e as conquistas surpreendentes até mesmo para aqueles envolvidos com o tema. Os mais diversos dispositivos eletrônicos nos acompanham hoje em muitas das nossas tarefas diárias, as vezes até mesmo sem que tenham noção disso. Dentre os mais evidentes estão os *smart phones*, dispositivos que integram perfeitamente uma diversificada forma de comunicação e computação, tornando-se ferramentas imprescindíveis no nosso dia a dia. E muito se deve ao avanço no domínio dos circuitos digitais, e ambientes computacionais, cuja flexibilidade permite seu uso nas mais variadas aplicações.

Mas se por um lado os benefícios são evidentes e permeiam praticamente todas as atividades modernas, por outro o ensino destas tecnologias enfrenta dificuldades. É recorrente o questionamento do que deve ser ensinado e como isso deve ser feito. Os conhecimentos dos princípios fundamentais são naturalmente essenciais, e permitem a continua adaptação e a possibilidade do acompanhamento dos desenvolvimentos tecnológicos. Mas a cada momento aquelas que são as tecnologias mais promissoras e usadas precisam também participar da composição dos cursos oferecidos aos alunos de engenharia.

Como professor universitário, ex orientador de doutorado e colega do autor deste livro, é com grande prazer que acompanho o seu sucesso como professor universitário, e seu contínuo entusiasmo pela ciência e engenharia. Neste livro, o Prof. João Ranhel realiza uma tarefa árdua – consegue abranger desde os fundamentos da eletrônica digital até as mais atuais técnicas de projetos de circuitos digitais, oferecendo aos estudantes um livro referencial sobre este tema. Além de uma vasta e sólida revisão de conceitos, o livro oferece aos estudantes muitos exemplos e exercícios fundamentais para a melhor compreensão dos assuntos por ele tratados, dando assim aos estudantes a oportunidade para bem assimilar tais conceitos, e então a possibilidade para usar estas técnicas no desenvolvimento de circuitos eletrônicos digitais para os mais diversos fins, e de diferentes níveis de complexidade.

Num mundo globalizado as interpenetrações dos mercados e oportunidades de trabalho são cada vez maiores. Desta forma, engenheiros atuais devem atuar em projetos desenvolvidos num contexto de parcerias internacionais, e somente aqueles com boa formação profissional terão de fato condições de bem explorar estas oportunidades. Mas há também uma enorme demanda por sistemas dedicados e especializados que exigem projetos específicos, criando assim mercados locais. Os engenheiros devem ser capacitados para trabalhar em todas as frentes, desde aquelas locais, focadas na resolução de um problema mais específico, como naquelas globais, e que só podem ser desenvolvidas com a participação de muitos profissionais compartilhando seus conhecimentos e contribuindo para as grandes realizações. Seja qual for a situação, a excelência na formação dos alunos é uma exigência para que tenham condições para bem realizar suas tarefas. E neste sentido, este livro oferece uma grande contribuição, de modo que espero que nossos estudantes tenham aqui a oportunidade de aprofundar e consolidar seus conhecimentos em eletrônica digital. E que a partir desta base continuem se empenhando para aprimorar seus conhecimentos nas tecnologias que continuarão evoluindo de forma cada vez mais acentuada.

Por fim parabenizo o autor pela empreitada e contribuição para o desenvolvimento do ensino de engenharia, esperando que os estudantes venham a ter este mesmo empenho e entusiasmo que são a base para uma carreira profissional de sucesso.

Prof. Dr. Marcio Lobo Netto,

Fevereiro de 2021.

Sumário

Prefácio 5
Sumário 6
Introdução 1
Breve Histórico 1
Parte 1 – FUNDAMENTOS 5
Capítulo I – Representações 7
1.1 – Números e Sistemas Numéricos 8
1.1.1 – Sistema Decimal 8
1.1.2 – Sistema Numérico com N Símbolos 9
1.1.3 – Sistema Binário 9
1.1.4 – Números Binários e Eletrônica 10
1.1.5 – Sistema Hexadecimal 11
1.1.6 – Conversão Entre Sistemas de Numeração 11
1.1.6.1 – Binário Para Decimal 11
1.1.6.2 – Decimal Para Binário 12
1.1.6.3 – Binário Para Hexadecimal e Vice-Versa 12
1.1.7 – Números Naturais x Inteiros 12
1.1.7.1 – Sinal-Magnitude 12
1.1.7.2 – Complemento Para 1 13
1.1.7.3 – Complemento Para 2 13
1.1.7.4 – Excesso-K (Offset Binary) 14
1.1.8 – Números Reais 14
1.1.8.1 – Ponto Fixo 15
1.1.8.2 – Ponto Flutuante 15
1.1.9 – Representando Símbolos Não-Numéricos 16
1.2 – Aritmética Binária 17
1.2.1 – Adição 17
1.2.2 – Subtração 18
1.2.3 – Multiplicação 20
1.2.4 – Divisão 21
Resumo e Conclusões do Capítulo 21
Capítulo II – Lógica 23
2.1 – Funções Lógicas 24
2.2 – Representações das Funções Lógicas 24
2.2.1 – Função **E** (**AND**) 25
2.2.2 – Função **OU** (**OR**) 25
2.2.3 – Função **NÃO** (**NOT**) 26
2.2.4 – Função **NÃO-E** (**NAND**) 26
2.2.5 – Função **NÃO-OU** (**NOR**) 27
2.2.6 – Função **OU-EXCLUSIVO** (**XOR**) 27

2.2.7 – Função **NÃO-OU-EXCLUSIVO** (**XNOR**) 28
2.3 – Circuitos Digitais 28
2.3.1 – Análise de Circuitos 29
2.3.2 – Síntese de Circuitos 30
2.3.2.1 – Projeto Simples com Componentes Discretos 30
2.3.2.2 – Projeto Simples com PLD e Verilog 31
2.4 – Exercício Para Prática 33
2.5 – Resumo do Capítulo 35
Capítulo III – Álgebra de Boole e Simplificação de Circuitos 37
3.1 – Exemplo de Simplificação 37
3.2 – Álgebra de Boole 37
3.3 – Revendo o Exemplo Primeiro 40
3.4 – Mapas de Veitch-Karnaugh 40
3.4.1 – Mapa com Duas Variáveis 40
3.4.2 – Mapa com Três Variáveis 41
3.4.3 – Mapa com Quatro Variáveis 41
3.5 – Ferramentas de Simplificação 43
Parte 2 – CIRCUITOS COMBINACIONAIS 45
Capítulo IV – Codificadores e Decodificadores 47
4.1 – Projeto de Um Encoder 48
4.1.1 – Resistores Pull-Up e Pull-Down 49
4.1.2 – Tabela Verdade do Encoder da Cisterna 50
4.1.3 – Código Verilog do Encoder da Cisterna 51
4.1.3.1 – Vetores em Verilog 52
4.1.3.2 – Estrutura **always** @() 52
4.1.3.3 – Estrutura **case**() 53
4.1.4 – Compilando e Gerando o Encoder 54
4.2 – Projeto de Decoder Binário Para 7-Segmentos 54
4.2.1 – Código Verilog do Decoder 7-Segmentos 56
4.3 – Projeto Cisterna Completo 56
4.4 – Outros Códigos em Sistemas Digitais 57
4.4.1 – Exemplos de Codificadores e Decodificadores 57
4.4.2 – Código Gray 57
4.4.3 – Código Johnson 58
4.4.4 – Decodificador Seletor N Para 2^N 58
Capítulo V – Circuitos Aritméticos e Lógicos 61
5.1 – Carry Versus Overflow 61
5.2 – Revendo Aritmética Binária em Números Sem Sinais 61
5.3 – Circuito Somador 62
5.4 – Circuito Subtrator 65
5.5 – Circuito Somador/Subtrator Complemento 65

5.6 – Circuito Multiplicador 66
5.7 – Circuito Divisor 68
5.8 – Aritmética em Verilog 68
5.8.1 – Redimensionando Vetores em Verilog 68
5.8.2 – Redimensionamento de Vetores Numéricos 69
5.8.3 – Operações Aritméticas Sem Sinal em Verilog 70
5.8.4 – Operações Aritméticas Com Sinal em Verilog 72
5.9 – Comparadores 73
5.9.1 – Comparação de Números Com Sinais 75
5.9.2 – Comparadores em Verilog 75
Capítulo VI – Tecnologia Digital e Multiplexadores 77
6.1 – Lógica com Diodos 77
6.2 – Níveis Lógicos Reais, Fan-IN e Fan-OUT 78
6.3 – Lógica com Transistores 79
6.3.1 – Field Effect Transistors 79
6.4 – Inversores com MOSFET e CMOS 84
6.5 – Portas Lógicas com CMOS 86
6.5.1 – Porta **NAND** CMOS 86
6.5.2 – Porta **NOR** CMOS 87
6.5.3 – Porta **XNOR** CMOS 87
6.5.4 – Porta **XOR** CMOS 88
6.6 – Circuitos com FET 89
6.6.1 – Transistores de Passagem e Pass-Transistor Logic 90
6.6.2 – Circuitos com Saída Tri-State 91
6.6.3 – Circuitos com Saída Open-Drain 93
6.7 – Multiplexador 94
6.8 – Demultiplexador 96
6.9 – Multiplexadores Como Gerador de Funções Lógicas 98
6.9.1 – Transformando Declarações em LUT 99
6.10 – Porta de Transmissão - Transmission Gate 100
6.11 – Comparador de Tensão 102
6.12 – Disparador de Schmitt (Schmitt Trigger) 104
6.12.1 – Circuito de Reset 105
6.13 – Datapath 106
6.14 – Codificação de MUX e DEMUX 108
6.15 – Codificação 3-STATE em Verilog 110
Parte 3 – CIRCUITOS SEQUENCIAIS 111
Capítulo VII - Lógica Sequencial 113
7.1 – Gerador de Clock com Inversor 113
7.1.1 – Estados e Realimentação 114
7.2 – Latch com Inversores 114

7.2.1 – Latch com Transistores de Passagem 115
7.2.2 – Tabela e Análise de Estados do Latch Transparente 116
7.2.3 – Resumo do Latch transparente 118
7.3 – Latch com Portas Lógicas 118
7.3.1 – Latch Transparente Tipo D com Portas Lógicas 119
7.3.2 – Latch Tipo D com Preset e Clear 120
7.4 – Circuitos Sensíveis à Bordas 120
7.4.1 – Diferença Entre Latch e Flip-Flop 121
7.4.2 – Circuito Flip-Flops Mestre-Escravo 121
7.5 – Símbolos dos FF e Latches 122
7.6 – Setup Time, Hold Time e Metaestabilidade 123
7.7 – Latches e Flip-Flops em Verilog 124
7.7.1 – Declarações Blocking e Non-Blocking em Verilog 124
7.7.2 – Codificação de Latch e Flip-flop em Verilog 125
7.6 – Registrador com FF Tipo D 126
7.6.1 – RTL e Aplicações de Registradores 127
7.7 – Registradores em Verilog 128
Capítulo VIII - Máquina de Estados Finitos 131
8.1 – Conceito de Máquina de Estados Finitos (FSM) 131
8.2 – Formas de Representação 132
8.2.1 – Diagrama de Estados 132
8.2.2 – Tabela de Estados / Eventos 134
8.2.3 – Blocos Lógicos e Diagrama Elétrico 134
8.3 – Máquinas Moore e Mealy 136
8.3.1 – Representações de Máquinas Moore 136
8.3.2 – Representações de Máquinas Mealy 137
8.3.3 – Vetores de Entrada e Saída 138
8.4 – Máquinas de Estado Assíncronas 139
8.5 – MEF um Projeto Prático 139
8.6 – Codificação de FSM em Verilog 143
8.6.1 – Codificação em Verilog da Máquina Mealy 143
8.6.2 – Codificação em Verilog da Máquina Moore 144
8.6.3 – Codificação em Verilog da Máquina Moore – estilo 2 146
8.6.4 – Erros Comuns na Codificação Verilog 147
8.6.5 – Codificando em Verilog o Projeto Prático 148
Capítulo IX - Circuitos Sequenciais 151
9.1 – Registradores de Deslocamento 151
9.1.1 – Registradores de Deslocamento com Flip-Flops Discretos 151
9.1.2 – Registradores de Deslocamento Como MEF (FSM) 153
9.1.3 – Registradores de Deslocamento Universal - Símbolo 153
9.1.4 – Codificando Registrador de Deslocamento em Verilog 154
9.1.5 – Comunicação Serial com Registrador de Deslocamento 156

9.1.6 – Registrador de Deslocamento e Dispositivos Programáveis 157
9.2 – Contadores 157
9.2.1 – Alternância nos FF-D (toggling) 157
9.2.2 – Contadores Assíncronos – Ripple Counter 158
9.2.3 – Contadores Síncronos 159
9.2.4 – Contadores Como MEF (FSM) 160
9.2.5 – Flip-Flops J-K e Contadores à Moda Antiga 161
9.2.6 – Contadores em Circuitos Discretos 162
9.3 – Contadores em Verilog 162
9.3.1 – Contador Up/Down em Verilog 163
9.3.2 – Contadores de Módulo M em Verilog 164
9.4 – Projetos Modulares em Verilog - Relógio 165
9.4.1 – Módulo Divisor de Clock 166
9.4.2 – Juntando Módulos e Compondo o Projeto 166
9.4.2.1 – Instanciando Componentes – Parâmetros 167
9.4.2.2 – Instanciando Componentes – Nome Local 168
9.4.2.3 – Instanciando Componentes – Conexões 168
9.4.3 – Instanciando Contadores 168
9.5 – Finite-State Machine com Datapath (FSMD) 170
9.5.1 – Exemplo de Datapath 172
9.6 – Pipeline 174
Capítulo X - Memórias 177
10.1 – Memória RAM Conceitual 177
10.1.1 – Escrevendo na RAM 178
10.1.2 – Lendo na RAM 180
10.1.3 – Cartas de Tempo das Memórias 181
10.1.4 – Resumo de Memória Conceitual 182
10.2 – Tipos e Características das Memórias 182
10.3 – Memória Eletrônicas 183
10.3.1 – Memória SRAM 183
10.3.1.1 – RAM Estática com Porta Dupla (Dual-Port) 185
10.3.1.2 – RAM Estática Single-Port em Verilog 186
10.3.1.3 – RAM Estática Dual-Port em Verilog 187
10.3.2 – Memória com Capacitores 188
10.3.2.1 – Leitura e Escrita na DRAM 190
10.3.2.2 – Evoluções da RAM Dinâmica 192
10.3.3 – Memória com Resistores 193
10.3.3.1 – PROM (Programável Uma Vez) 193
10.3.4 – Memristores - RRAM 193
10.3.5 – Memória Magnetorresistiva - MRAM 194
10.3.5 – Memória Ferroelétrica - FeRAM 195
10.4 – Memória com Transistores MOS 196

10.4.1 – Floating Gate (Porta Flutuante) 196
10.4.2 – EPROM 198
10.4.3 – Memórias EEPROM 199
10.4.4 – Memórias NOR x NAND Flash 200
10.4.5 – NOR Flash 201
10.4.6 – NAND Flash 203
Capítulo XI - Conversão e Quantização 207
11.1 – Amostragem 208
11.2 – Resolução dos Conversores 209
11.3 – Quantização e Erro de Quantização 209
11.4 – Considerações na Escolha do Conversor 211
11.4.1 – Parâmetros Gerais dos Conversores 211
11.4.1 – Requisitos de DAC 211
11.4.1 – Requisitos de ADC 211
11.5 – Conversores Digital para Analógico (DAC) 212
11.5.1 – Resistores Ponderados 212
11.5.2 – Malha R-2R 213
11.5.3 – Divisor de Tensão 215
11.5.4 – Circuito PWM 216
11.6 – Conversores Analógico Para Digital (ADC) 219
11.6.1 – Circuito Sample & Hold 219
11.6.2 – Conversores Flash 220
11.6.2 – Conversores Integradores 222
11.6.3 – Conversores por Aproximação Sucessiva 224
11.6.4 – Conversores Delta-Sigma ΔΣ 225
Parte 4 – DISPOSITIVOS PROGRAMÁVEIS E PRÁTICAS 231
Capítulo XII - Dispositivos Lógicos Programáveis (PLDs) 233
12.1 – Programmable Logic Device – PLD 233
12.2 – Aprender Com ou Sem Kit de FPGA? 234
12.3 – Lógica Programável: Histórico 236
12.3.1 – Lógica Programável em ROM: 237
12.3.2 – PAL (Matriz lógica programável) 237
12.3.3 – GAL (Generic Array Logic) 239
12.3.4 – PLA (Arranjo Lógico Programável) 240
12.3.5 – Elementos Lógicos Programáveis 241
12.3.6 – CPLDs (Complex Programmable Logic Devices) 242
12.3.7 – FPGA (Field Programmable Gate Array) 243
12.4 – Configuração, Núcleos e Computação Heterogênea 245
12.5 – JTAG (Joint Test Action Group) 247
Capítulo XIII – EDA e Simulações 251
13.1 – Fluxo de Projetos nos EDAs 251

13.2 – Criando um Projeto no Quartus© 252
13.3 – Relacionando Pinos e Sinais 260
13.4 – Compilação e Execução do Projeto no FPGA 262
13.3 – Simulações no Quartus© e ModelSim© 264
13.4 – Método Estímulo Manual no ModelSim 264
13.5 – Método Testbench Verilog 269
Capítulo XIV – Práticas 275
14.1 – Prática 1 – Projeto: Sala e Interruptores 276
14.2 – Prática 2 – Encoder e Decoder 278
14.3 – Prática 3 – Hierarquia em Verilog e Comparador 280
14.4 – Prática 4 – Somador e Subtrator 283
14.5 – Prática 5 – Projeto: ULA 285
14.6 – Prática 6 – Flip-Flop e Registrador 286
14.7 – Prática 7 – Registrador de Deslocamento 288
14.8 – Prática 8 – Contadores 290
14.9 – Prática 9 – Máquina de Estados Finitos e Debouncing 292
14.10 – Prática 10 – Cronômetro Digital 295
Referências: 301

Introdução

Este livro aborda eletrônica digital desde fundamentos básicos até a aplicação de tecnologias atuais para construção de circuitos lógicos e sistemas digitais complexos. Chamamos de *sistema* um conjunto de elementos interdependentes, concretos ou abstratos, estruturados de forma apropriada a fim de que formem um todo organizado. No caso, *sistemas digitais* são circuitos eletrônicos organizados de tal forma que executam funções de processamento de informação na forma digital, ou binária. Exemplos de sistemas digitais são os (micro)processadores, presentes em quase todos os dispositivos e equipamentos atuais. Processadores, seguindo o estilo de processamento de computadores tradicionais, executam instruções pré-programadas de forma sequencial. A partir da evolução no processo de fabricação de processadores, do surgimento de sistemas operacionais que realizam múltiplas tarefas concomitantes, do advento de técnicas de Inteligência Artificial (IA) e de aprendizado de máquina modernas, além de outros fatores; o estilo de processamento de informações tem sofrido um deslocamento do eixo serial e sequencial para o processamento em paralelo de informação. Em outras palavras, circuitos digitais especializados têm sido projetados para executar processamento em conjunto, em paralelo, com os processadores tradicionais, diminuindo o tempo de processamento, o custo operacional, em geral, o consumo de energia, dentre outros benefícios.

Nesse cenário, surge a *computação heterogênea*, que é uma forma de processamento de informação realizada por processadores tradicionais associados a hardware dedicado. Tais processadores dedicados são criados por projetistas de sistemas digitais para aumentar a performance do sistema. Quando a complexidade das funções executadas num sistema digital aumenta, aumentam também a complexidade dos circuitos, dos componentes envolvidos, da forma de planejar, de criar e de executar o projeto do sistema como um todo. Dessa forma, nas últimas décadas surgiram circuitos digitais programáveis (PLDs – *Programmable Logic Devices*), dentre eles os FPGAs (*Field Programmable Gate Array*) e os SoC (*System on Chips*). FPGAs e SoCs são circuitos digitais nos quais o projetista pode implementar um circuito digital complexo sem necessidade de recorrer a componentes discretos, a placa de ensaio (*protoboards* e *breadboards*), sem usar fios, *jumpers*, *wire-wrap* ou método similar, muito usados em prototipagem até os anos 1980 ou 1990.

Atualmente, projetos são feitos em um computador pessoal, usando um software dedicado, e , em geral, o que o projetista faz é *descrever* o comportamento desejado para o circuito. Para descrever o comportamento de circuitos foram desenvolvidas algumas linguagens de descrição de hardware (HDL – *Hardware Description Language*). Uma das linguagens de descrição de circuitos mais comuns é a Verilog, que concorre com várias outras linguagens de descrição de hardware, mas que ao meu ver é a mais intuitiva e de menor complexidade para o leitor que está iniciando nesse assunto.

Essa é uma descrição breve do título do livro, e uma explanação geral do que o usuário encontrará no livro: explicação sobre sistema binário e lógica, sobre circuitos eletrônicos que executam funções lógicas, sobre criações de blocos funcionais e sobre como juntar blocos lógicos para criar processamento de informações de forma complexa. O livro tem um viés prático; assim, vamos usar FPGA e Verilog para sintetizar os blocos de circuitos mais importantes, até mesmo pequenos sistemas, na medida em que forem descritos durante o livro. O leitor não vai encontrar nessa série explicações minuciosas e análises profundas de certos assuntos. Ao contrário, tentaremos ser pragmáticos.

Até a parte III o livro trata de fundamentos de eletrônica digital, ou seja, tem a característica de um livro-texto didático. Na parte IV o livro é mais parecido com tutorial, explicando como usar Quartus© e ModelSim©, e como criar circuitos programáveis de forma prática. O leitor com conhecimento sólido em eletrônica digital, interessado em aprender Verilog e FPG, pode começar pelos tutoriais, mas muito da explicação de como Verilog funciona está diluído pelo livro. O último capítulo traz proposições práticas e exercícios de sintetização e simulação de circuitos em Verilog, propostas estas que temos usado como aulas práticas de laboratório há algum tempo.

Breve Histórico

Ainda nos anos 1960 surgiram os primeiros circuitos integrados lógicos[1], criados pela Fairchild em 1961, seguidos pelas famílias TTL e CMOS. Naquele tempo, os circuitos integrados (CIs) disponibilizavam portas lógicas e alguns circuitos funcionais básicos. Na medida em que o número de transistores integrados aumentava,

surgiram em anos subsequentes os blocos lógicos integrados em LSI (*large-scale integration*: larga escala de integração). Até o final dos anos 1980, grande parte do tempo de desenvolvimento em eletrônica digital era gasto em simplificações de circuitos. Era – e ainda é –necessário economizar o número de CIs nas placas de circuito impresso (PCB – *printed circuit board*), diminuir o consumo de energia, diminuir o tamanho da placa, diminuir custos em geral, facilitar testes, dentre vários outros motivos técnicos.

A partir de meados dos anos 1980 a indústria de semicondutores desenvolveu então os CIs de lógica programável, que culminou nos atuais CIs CPLDs (*Complex Programmable Logic Devices*) e nos FPGA (Field Programmable Gate Array). Ao mesmo tempo, foram desenvolvidas linguagens para descrever circuitos digitais (HDL - *Hardware Description Language*), sendo as mais populares a VHDL e Verilog.

Primeiramente, o uso de HDL seria para descrever sistemas digitais em linguagem mais acessível; uma vez que eram comuns projetos com várias folhas de desenho de diversos tamanhos, feitos seguindo distintas normas, com a finalidade de representarem numerosos blocos lógicos que compunham o projeto. Num segundo momento, verificou-se a utilidade das HDLs para simular circuitos, e assim verificar funcionalidade antes de serem implementados quaisquer protótipos. Num terceiro momento, empresas que criavam circuitos de lógica programável começaram a desenvolver compiladores que transformavam textos HDLs em circuitos físicos para programar CPLDs e FPGAs, processo esse chamado *síntese* de circuitos.

A partir desse período, a tarefa principal do projetista começa a mudar. Se antes a simplificação de circuitos e a construção de PCBs eficientes tomava grande parte do tempo dos projetos, a partir da virada do milênio os projetistas têm desafios de descrever comportamentos eficientes de circuitos. Há assim enorme ganho de tempo na criação de sistemas digitais. Uma forma abstrata de fazer projetos de sistemas digitais emergiu, a RTL (*register transfer level*), ou lógica no nível de transferência entre registradores. Em RTL, o comportamento do circuito é descrito em termos do fluxo de dados entre registradores por meio de caminhos para os dados (*datapath*).

Voltando um pouco no tempo, os processadores de uso genérico tiveram avanços enormes a partir dos anos 1980. De processadores de 8 bits operando em alguns MHz de *clock*, baixa capacidade de endereçamento de memória, periféricos e memória geralmente externos, os processadores saltaram para 32 e 64 bits, operando em GHz, com memória interna, endereçamento de alguns GBytes (e.g. 32 bits podem endereçar 4GBytes). Processadores incorporaram blocos de funções, subsistemas, periféricos, e tornaram-se sistemas completos em um único *chip*. Uma classe de CIs, conhecida como microcontroladores (***μ***C), possui atualmente vários periféricos integrados, memória não volátil interna e CPU avançada de 32 bits, além de outros recursos. Por efetivamente serem sistemas completos em um único CI são chamados SoC (acrônimo para *System on Chip*). Nesse cenário do uso de ***μ***C, o trabalho do projetista de sistemas digitais consiste também em decidir se utiliza um SoC para solucionar um problema, ou se deve criar uma solução em hardware, um circuito específico para realizar certo tipo de processamento de informação.

Recentemente, fabricantes de FPGA integraram núcleos de processadores tradicionais dentro de seus CIs, sendo o ARM (*Advanced Risc Machine*) o mais comum. Isso significa que o projetista poder utilizar parte da solução de um problema usando a flexibilidade do software rodado no processador tradicional e a parte mais exigente do processamento pode ser transferida para hardware, ou seja, para um processador específico desenvolvido pelo projetista, que é o que se denomina computação heterogênea, embora o termo computação heterogênea também se refira ao processamento em múltiplos cores de processadores e ao uso das Unidade de Processamento Gráfico de Propósito Geral ou GPGPU (*General Purpose Graphics Processing Unit*).

Mais recente, notou-se que RTL e as linguagens HDL têm consumido muito tempo para desenvolvimento de sistemas cuja complexidade aumenta em larga escala. Sistemas que há alguns anos eram descritos em textos curtos, atualmente podem exigir dezenas ou centenas de milhares de linhas de código. Além do tempo gasto para escrever o código, esforços de mesma magnitude são gastos em simulações para testar tais sistemas. Surgiram compiladores capazes de sintetizar hardware a partir da descrição de sistemas em linguagem de alto nível, como "C"/ "C++", SystemC, OpenCL, e outras.

Programas podem ser testados com maior flexibilidade e simplicidade, economizando tempo na criação de projetos. Duas frentes têm se destacado: HLS (*High-Level Synthesis*), um processo de design automatizado que interpreta um algoritmo (um software) e cria um hardware digital que implementa o comportamento descrito pelo algoritmo; e OpenCL®, uma arquitetura para escrever programas que funcionam em plataformas onde

coexistem CPUs de cores múltiplos, FPGAs, GPUs, etc. Desenvolvida pela Apple®, agora padrão aberto, OpenCL é uma plataforma para desenvolvimento de processamento de informações e programação paralela, usando tanto o paralelismo de tarefas como de dados.

Dado esse contexto, um iniciante que pretende aprender eletrônica digital a partir do zero precisa ter uma noção global de onde pode ou deve chegar. Processamento de informações hoje é um misto de hardware e software e não há fronteiras distintas entre esses dois campos como havia há pouco tempo. Um projetista de hardware terá que ter domínio de software e algoritmos para compreender e descrever sistemas em RTL e HLS ou OpenCL. Por outro lado, é provável que programadores, engenheiro de software, cientista da computação e outros profissionais da área também necessitem se envolver com o projeto de processadores dedicados e algum hardware específico.

Muito conhecimento foi introduzido nos últimos 60 anos, e como consequência é aconselhável que um estudante gaste seu tempo aprendendo conceitos relevantes para o estado da arte em projetos de sistemas. A questão que surge é: é possível aprender eletrônica digital, HDL, microprocessadores, RTL, síntese de alto nível e computação heterogênea sem se ater a conceitos antigos? A resposta é sim. E espero que o leitor veja durante o livro.

Parte 1 – FUNDAMENTOS

Capítulo I – Representações

Pense em uma bicicleta. Conseguimos descrevê-la dizendo que ela possui duas rodas, um quadro, um banco e um guidão, duas manoplas de freio, etc. Nenhum desses componentes da bicicleta existe de fato dentro da sua cabeça. Nosso encéfalo manipula "representações" do mundo. Dentro dos encéfalos[1] os neurônios manipulam pequenos pulsos com duração de aproximadamente 1 milésimo de segundo, chamados potenciais de ação, ou *spikes*. Possivelmente ocorrem milhões ou bilhões de *spikes* para você interpretar a palavra "bicicleta" e em seguida deflagrar uma cadeia de processos que resulta na visualização e na descrição de uma bicicleta. Nós transformamos coisas reais (e imaginárias) do mundo em informações representadas por *spikes*. Nossos encéfalos são máquinas especializadas em processar informações. Interessante é que *spikes* são sinais binários: ou acontecem de forma bem definida, ou não ocorrem. Podemos dizer que a quase totalidade, ou a grande maioria dos neurônios operam de forma binária: disparando *spikes* ou permanecendo silentes.

Imitando a funcionalidade de nossos encéfalos, queremos criar sistemas para processar informações. A primeira coisa a considerar é a forma como representaremos tais informações. Como faziam certas máquinas mecânicas de cálculo, podemos escolher um sistema mecânico para representar e processar informações; mas de longe eletricidade é a forma mais indicada por uma série de razões como: maior velocidade no processamento, menor tamanho por elemento de processamento, menor consumo de energia; a lista seria longa. A natureza levou milhões de anos para desenvolver sistemas nervosos baseados também no processamento de sinais elétricos associado a processos químicos que ocorrem nos encéfalos.

Fenômenos físicos, como luz, pressão, velocidade, força, vários outros, existem de forma contínua na natureza. Existem componentes que convertem uma forma de energia em outra, chamados *transdutores*. Assim, é possível converter os diversos tipos de fenômenos físicos em sinais elétricos proporcionais às intensidades com que ocorrem por meio dos transdutores. De uma forma geral chamamos de *sensores* tais transdutores e alguns exemplos são fotosensores, termopares, células de carga, potenciômetros, etc. Em resumo, esses *dispositivos* ou *sensores* são capazes de converter um tipo de fenômeno físico, de energia, etc. em sinais elétricos proporcionais ao estímulo que recebem. Quando o sinal resultante é "proporcional" ao estímulo dizemos que o sinal obtido num sensor é "*análogo*" ao fenômeno. Em outras palavras, a *representação* de um fenômeno físico é feita na forma de uma corrente ou potencial elétrico proporcional, *análoga* à intensidade do fenômeno causador.

Existe uma parte da engenharia eletrônica especialista em obter, tratar, processar e gerar informação tratando sinais na forma analógica. Nesse tipo de processamento de sinais o interesse do projetista é manter o sinal do começo ao fim sem distorções. Em outras palavras, neste tipo de processamento, via de regra, espera-se que o sinal não sofra distorções desde a entrada até a saída do circuito. Entre a captação do sinal e a entrega do sinal processado, é possível realizar eletronicamente várias operações sobre o sinal, por exemplo, filtrar para eliminar ruídos, pode-se executar operações aritméticas (multiplicar, somar, subtrair, dividir), comparar sinais e tomar decisões, realizar operações de cálculo como integração e derivação. Isso quer dizer que poderíamos criar um computador analógico? Sim, eles estão em desuso atualmente, mas já estiveram em voga há alguns anos. Contudo, esse tipo de computador não é flexível. Por exemplo, um integrador utiliza um circuito com um resistor e capacitor e um amplificador operacional (*OpAmp*). Se precisarmos modificar as constantes de integração teremos que trocar fisicamente os resistores e capacitores.

Por outro lado, considere representar o número Pi com três casas decimais depois da vírgula: $\pi = 3{,}141$; ou seja, nesse processador o circuito teria que representar 3 volts e ser preciso e imune a ruídos até 0.001 V (ou 1 milivolt), o que pode ser obtido com um circuito relativamente simples. Contudo, considere que o circuito deve executar cálculo usando Pi com precisão nove casas decimais, ou seja: $\pi = 3{,}141592653$. Nesse caso, o circuito tem que tratar de sinais com magnitude de +3 V com imunidade à ruídos e precisão de nano volts (10^{-9} V). Circuitos analógicos com essa característica são caros porque precisam ter componentes muito precisos e imunidade à ruído da ordem de nano volts. Todavia, inúmeras soluções de processamento de dados em engenharia, economia, ciência, astrofísica e outras áreas, exigem cálculos cujas precisões vão além de 9 casas decimais.

Nesse contexto, para solucionar esse e outros tipos de problemas surge a opção da eletrônica digital. Como os neurônios em nossos encéfalos, eletrônica digital trata dados na forma binária; ou seja, o sinal existe dentro de

[1] Encéfalo é o centro do sistema nervoso em todos os animais vertebrados; e cérebro é uma das partes que compõem o encéfalo.

uma faixa bem definida, por exemplo, um valor igual ou próximo da tensão de alimentação do sistema (***+V***), ou um valor próximo ou igual ao valor de referência da fonte (***GND*** ou *ground*) – chamado *terra*, *referência*, ou zero-volt. Falaremos das vantagens do sistema representar informação no formato binário/digital adiante. Primeiro, vamos compreender como é possível representar fenômenos físicos, valores, abstrações, palavras, imagens e outras coisas do mundo na forma digital, e vamos começar por números e quantidades.

1.1 – Números e Sistemas Numéricos

Filósofos travam interessantes discussões sobre a natureza dos números, se são ou não abstrações ou objetos, se são construções mentais, se dependem ou não da linguagem. Certas tribos humanas[2], por exemplo os Pirahãs na Amazônia[3][4], não possuem em sua linguagem a representação detalhada de números como temos. Os Pirahãs usam a palavra *hói*, que representa a quantidade "1", e.g. 1 dedo, mas que também representa pouco ou pequeno; a palavra *hoí* que representa "2", e.g. 2 dedos, e a palavra *aibai* que representa muitos. Parece que os Pirahãs não têm necessidade de serem precisos quando representam quantidades. Contudo, qualquer grupo tecnologicamente desenvolvido precisará tratar quantidades com maior acurácia.

Vários grupos humanos encontraram diferentes formas de representar números. Por exemplo, 3000 anos AC os egípcios usavam um sistema de numeração representado por hieróglifos. Possivelmente motivados pelos 10 dedos das mãos, o instrumento de contagem mais acessível aos humanos, o sistema egípcio usava a base 10. Eles usavam figuras distintas para representar um, dez, mil, dez mil, cem mil e milhão, e, para representar números quaisquer, repetiam N vezes o hieróglifo com símbolos mais significativos em cima dos menos significativos[2]. Os babilônicos desenvolveram o sistema sexagesimal, cuja base é 60, aproximadamente 3 séculos AC. Eles usavam o polegar e as 12 falanges dos quatro dedos restantes em uma das mãos para contar até 12, e os 5 dedos da mão livre para contar até 60 (5x12). Além do mais, sessenta é um número facilmente divisível por 2, 3, 4, 5 e 6. Esse sistema é usado ainda hoje em algumas atividades humanas, como a marcação das horas, contagem por dúzias, etc. Os babilônicos introduziram ainda o sistema de notação posicional para registrar um valor; ou seja, a posição em que um símbolo se encontra indica a quantidade de 60 unidades contadas. Os babilônicos não tinham um símbolo para o zero, assim, deixavam em branco as posições onde nada devia ser marcado, ou faziam um pequeno símbolo de apóstrofe. Por outro lado, os romanos criaram um complexo sistema de representação baseado em letras do alfabeto. O sistema era complexo porque não permitia operações dos símbolos diretamente, tanto que poucos além dos escribas sabiam realizar operações de multiplicação ou divisão. Imagine, por exemplo, dividir MMXIX por CCCLVI. Na China, foi inventado o sistema algorítmico para se fazer contas, colocando varetas representando números de 1 a 9 em colunas, cada coluna representando as unidades, centenas, milhares, etc.; contudo, para registrar números os chineses usavam um método mais complicado, porque inventaram símbolos para representar dezena, centena, milhar, etc. Os chineses não conheciam o número "zero" e foram os indianos que introduziram esse conceito na matemática, possivelmente no século IX DC. O que antes significava apenas um marcador de lugar ou um espaço vazio para outros povos, os indianos transformaram em um número com significado por si mesmo. Os algarismos usados hoje para representar números são conhecidos como indo-arábicos, com os símbolos {9,8,7,6,5,4,3,2,1 e 0} representando os símbolos da base de 10, a notação posicional representada pela posição do número indica quantas dezenas, centenas ou milhares eles representam, e os algoritmos para operações aritméticas foi herdada de milhares de anos de evolução do raciocínio de representações e aprendizado de como operar com números.

1.1.1 – Sistema Decimal

No sistema decimal a base numérica é o 10, ou seja, usamos 10 símbolos (*algarismos*) para representar cada um dos valores da base. Para representar um valor qualquer usamos a notação posicional de sorte que um valor X pode ser representado pelo polinômio:

$$X = \ldots + \psi.10^3 + \psi.10^2 + \psi.10^1 + \psi.10^0 + \psi.10^{-1} + \psi.10^{-2} + \psi.10^{-3} + \ldots \qquad (1.1)$$

onde **10** é a base, elevada a uma potência de acordo com a posição que ocupa, e ψ é um dos 10 símbolos do conjunto $\Psi = \{9, 8, 7, 6, 5, 4, 3, 2, 1, 0\}$. Assim, o valor 5.364,79 é representado por:

$X = 5.10^3+3.10^2+6.10^1+4.10^0+7.10^{-1}+9.10^{-2}$, ou seja: $5.1000+3.100+6.10+4.1+7.0{,}1+9.0{,}01$.

[2] Ver Papiro de Rhind ou Ahmes: https://pt.wikipedia.org/wiki/Papiro_de_Rhind

1.1.2 – Sistema Numérico com N Símbolos

Para criar um sistema numérico com qualquer quantidade de símbolos usando a regra da notação posicional, em primeiro lugar define-se o conjunto de símbolos $\boldsymbol{\Psi}_n = \{\lozenge, \ldots, \smile, ¤\}$ a serem usados. Por exemplo, para criarmos um sistema com 3 símbolos, usando o conjunto acima, teremos $\boldsymbol{\Psi}_3 = \{\lozenge, \smile, ¤\}$ e o polinômio que define um número nesse sistema seria reescrito da equação (1.1) substituindo a base 10 pelo 3, que é a quantidade de símbolos do sistema novo:

$$\boldsymbol{X} = \ldots + \boldsymbol{\psi}.3^4 + \boldsymbol{\psi}.3^3 + \boldsymbol{\psi}.3^2 + \boldsymbol{\psi}.3^1 + \boldsymbol{\psi}.3^0 + \boldsymbol{\psi}.3^{-1} + \boldsymbol{\psi}.3^{-2} + \boldsymbol{\psi}.3^{-3} + \boldsymbol{\psi}.3^{-4} + \ldots \quad (1.2)$$

Nesse sistema, os valores seriam escritos seguindo a regra posicional; zero é o algarismo ¤, a unidade é o símbolo $\smile$, e duas unidades são representadas pelo símbolo $\lozenge$. Assim como fazemos no sistema decimal quando chegamos ao algarismo "9", não temos mais símbolos para representar o valor 3 com apenas um algarismo nesse sistema; então criamos uma coluna à esquerda para indicar que já foram contadas três ocorrências e voltamos a coluna da unidade para zero, quer dizer, contamos um trio e zero unidades. Quando as duas colunas estiverem completas com o último símbolo do conjunto, criamos uma terceira coluna para representar quantas vezes já foram contados 3^2, depois uma nova coluna para 3^3, e assim por diante. A tabela I ilustra a contagem neste sistema de três símbolos. Em um exemplo, o valor "$\smile\lozenge\smile¤\lozenge$" nesse sistema usa 5 dígitos, e representa o valor: $1.3^4 + 2.3^3 + 1.3^2 + 0.3^1 + 2.3^0 = 146$; ou $1.81 + 2.27 + 1.9 + 0.3 + 2.1 = 146$.

TABELA I

número			decimal	
		¤	zero	0
		$\smile$	um	1
		$\lozenge$	dois	2
	$\smile$	¤	três	3
	$\smile$	$\smile$	quatro	4
	$\smile$	$\lozenge$	cinco	5
	$\lozenge$	¤	seis	6
	$\lozenge$	$\smile$	sete	7
	$\lozenge$	$\lozenge$	oito	8
$\smile$	¤	¤	nove	9
$\smile$	¤	$\smile$	dez	10
$\smile$	¤	$\lozenge$	onze	11
$\smile$	$\smile$	¤	doze	12

Note que, seguindo o exposto acima, é possível criar um sistema numérico com apenas 2 símbolos.

1.1.3 – Sistema Binário

O sistema binário é a base da eletrônica digital. Dissemos na introdução de representações que queremos trabalhar com valores discretos; ou seja, com ***+V*** e ***zero*** Volt (***GND***) porque, dentre outros motivos, os sinais se tornam mais imunes a ruídos. No sistema binário usa-se apenas dois símbolos: $\boldsymbol{\Psi}_2 = \{\boldsymbol{1}, \boldsymbol{0}\}$.

Assim, o polinômio que define um número nesse sistema seria reescrito da equação (1.1) substituindo base 10 pela base 2:

$$\boldsymbol{X} = \ldots + \boldsymbol{\psi}.2^4 + \boldsymbol{\psi}.2^3 + \boldsymbol{\psi}.2^2 + \boldsymbol{\psi}.2^1 + \boldsymbol{\psi}.2^0 + \boldsymbol{\psi}.2^{-1} + \boldsymbol{\psi}.2^{-2} + \boldsymbol{\psi}.2^{-3} + \boldsymbol{\psi}.2^{-4} + \ldots \quad (1.3)$$

Nesse sistema, cada algarismo só pode assumir o valor '1' ou '0'.

Seguimos a regra da notação posicional e vamos inserindo dígitos à esquerda na medida em que aumentamos os valores representados, e à direita na medida em que fracionamos o valor.

A tabela II ilustra como escrevemos valores inteiros positivos em binário até 16. A coluna mais à direita mostra como são representados os valores em hexadecimal, discutido adiante.

Uma vez que um sistema numérico é adotado para representar valores, espera-se que todas as propriedades das operações aritméticas sejam válidas nesse sistema. Dessa forma, vamos adiante investigar como realizar operações aritméticas com números binários.

TABELA II					
número binário				decimal	hexadecimal
			0	zero	***0***
			1	um	***1***
		1	0	dois	***2***
		1	1	três	***3***
	1	0	0	quatro	***4***
	1	0	1	cinco	***5***
	1	1	0	seis	***6***
	1	1	1	sete	***7***
1	0	0	0	oito	***8***
1	0	0	1	nove	***9***
1	0	1	0	dez	***A***
1	0	1	1	onze	***B***
1	1	0	0	doze	***C***
1	1	0	1	treze	***D***
1	1	1	0	quatorze	***E***
1	1	1	1	quinze	***F***

1.1.4 – Números Binários e Eletrônica

A seguir vamos ver como representamos números em circuito digital; ou seja, vejamos a ideia por trás de usar representação binária em eletrônica.

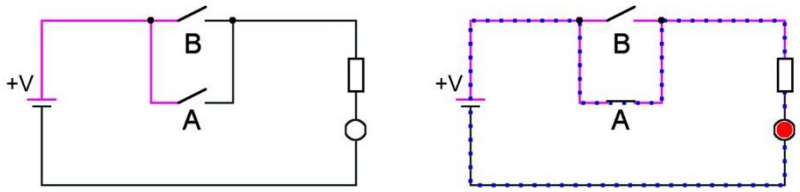


Fig. 1.1: circuitos representando duas chaves que atuam como variáveis independentes A e B, e uma saída, o LED que acende em função dos estados das chaves A e B.

As principais vantagens de se utilizar sistema binário em eletrônica é que os circuitos ficam muito mais simples e consideravelmente mais imunes à ruídos, como veremos adiante. Nos circuitos ilustrados na Fig. 1.1, se arbitrarmos que chave fechada representa '1', então a chave aberta representará um '0'. Se arbitrarmos que um LED aceso representa um '1' então o LED apagado representa um '0'.

Note que os *estados* das variáveis são discretos: 1 = fechada, 0 = aberta. A saída também tem valores bem definidos e complementares: 1 = LED aceso, 0 = LED apagado. Não há um estado de saída com o LED meio aceso, ou com ¼ de brilho. O circuito não tem ruídos aparentes. Considerando então que temos apenas os níveis de tensão ***+V*** (terminal positivo da bateria) e ***GND*** ou terra (zero Volt da bateria), como podemos representar números usando um circuito eletrônico dessa natureza? Considere que temos apenas um fio, quantos valores podemos representar? Apenas dois: ***+V*** e ***GND***; ou seja, só podemos conectar esse fio com o polo ***+V*** da bateria ou ligar esse fio ao ***GND***.

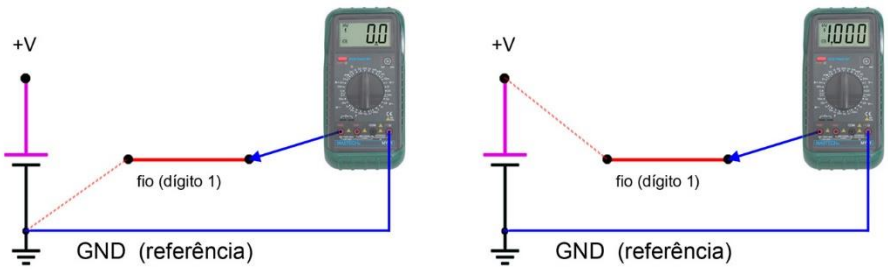


*Fig. 1.2: com apenas 1 fio (1 dígito) só podemos conectá-lo ao **+V** ou ao **GND** e assim representar '1' e '0' respectivamente.*

Da mesma forma como fizemos na tabela II, se precisarmos representar mais de dois valores temos que incrementar o número de fios em paralelo no sistema digital. Assim, quando colocamos dois fios podemos ligá-los ao ***+V*** ou à ***GND*** e obteremos uma combinação de {00, 01, 10, 11}.

Note que é dever do projetista definir qual dos fios será mais significativo. Em outras palavras, um dos fios representará o algarismo/dígito 2^0 e fio será o menos significativo; já o outro fio representará 2^1 e será o dígito mais significativo. Cada um desses fios pode representar uma informação binária mínima (0 ou 1) e em inglês essa informação recebe o nome de ***b**inary dig**it*** (dígito binário – ou *bit*). O bit mais significativo de um conjunto de fios ganha o nome de MSb (*most significant bit*) e o bit menos significativo recebe o nome de LSb (*least significant bit*). A Fig. 1.3 ilustra uma forma de representar números com 2 bits, usando 2 fios paralelos e independentes. Em um dado momento, olhamos para a combinação de sinais ***+V*** e ***GND*** nos fios, que será o valor representado naquele momento.

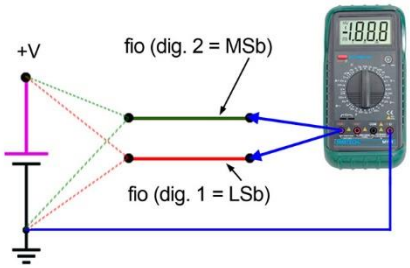


*Fig. 1.3: com 2 fios representa-se 2^2 combinações simultâneas de **+V** ou **GND** no par de fios.*

Um sistema com ***N*** fios pode então representar 2^N combinações de níveis '1' (***+V***) e '0' (***GND***). Os primeiros processadores criados nos anos 1970/80 usavam 8 fios em paralelo – daí os processadores serem chamados de 8-bits. Com 8 fios em paralelo podemos representar $2^8 = 256$ valores, e essa quantidade de bits ganhou o nome de *byte* – usado até hoje como referência de quantidade de memória. Atualmente, é comum ter processadores de 32 bits usados em automação, celulares, etc. Ou seja, são capazes de representar num instante "***t***" valores com a combinação de ***+V*** e ***GND*** em 32 fios paralelos, o que significa 2^{32} combinações, ou valores de 0 até 4.294.967.295.

1.1.5 – Sistema Hexadecimal

Um sistema de numeração muito usado em computação e em sistemas digitais é o *hexadecimal*, definido pelo conjunto $\Psi_h = \{F, E, D, C, B, A, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0\}$, com 16 símbolos. Esse sistema é prático porque representa os valores possíveis em um conjunto de 4 bits. O conjunto de 4 bits é chamado *nibble*. Assim, o valor 11 em decimal é representado por 1011_b e B_h, com os sub índices ***b*** e ***h*** indicando representações em binário e em hexadecimal. Quando não há sub índices consideramos valor em decimal. Uma forma comumente usada para representar hexadecimal na computação é *0x*B. Nesse caso, o prefixo *0x* serve para indicar que o valor é em *hexa* (versão curta de hexadecimal). A notação faz sentido porque zero à esquerda não altera em nada o valor de um número e indica ao programador e ao compilador que se trata de um valor numérico, e o ***x*** em seguida indica que o valor está codificado em hexadecimal.

Hexadecimal é especialmente prático porque tomamos um byte e dividimos em dois grupos de 4 bits (2 *nibbles*) e apontamos diretamente o valor que cada conjunto representa. Por exemplo, um byte com o valor 1010 0111 pode ser escrito em hexa como *0x*A7 (7 para os 4 bits menos significativos e ***A*** para os 4 bits mais significativos).

1.1.6 – Conversão Entre Sistemas de Numeração

Vamos analisar brevemente como converter um valor de uma base para outra dentre os sistemas numéricos binário, decimal e hexadecimal.

1.1.6.1 – Binário Para Decimal

Para fazer a conversão de um valor binário para decimal basta usar o polinômio da equação (1.3). Assim, o valor 10111010_b em decimal = $1.2^7+0.2^6+1.2^5+1.2^4+1.2^3+0.2^2+1.2^1+0.2^0 = 1.128+0.64+1.32+1.16+1.8+0.4+1.2+0.1 = 186$. Outro exemplo: $10001_b = 1.2^4+0.2^3+0.2^2+0.2^1+1.2^0 = 17$; ou seja, 1.16 + 1.1 = 17.

1.1.6.2 – Decimal Para Binário

A conversão de decimal para binário é um pouco mais trabalhosa porque envolve fatoração, sucessivas divisões por 2. O valor decimal é dividido por 2 até que o último quociente seja q < 2. Por exemplo, 69 e 68 são convertidos como:

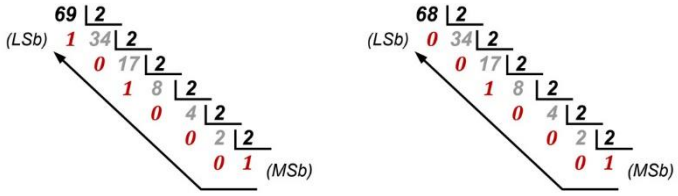


Fig. 1.4: Obtenção do número binário por divisão sucessiva do valor decimal por 2.

Como o divisor será sempre o "2", o resto de cada etapa das divisões sucessivas será sempre 1 ou 0. O resto da primeira divisão será o bit menos significativo (LSb). Note que 69, ímpar, gera LSb = 1; enquanto 68, par, gera o LSb = 0. O último dividendo pode ser um número 3 ou 2. Quando for 3, MSb = 1 e o segundo bit mais significativo será também 1, porque o resto da divisão 3 por 2 é 1. Quando for 2, MSb = 1, porém o segundo bit mais significativo será 0, porque o resto da divisão 2 por 2 é zero.

1.1.6.3 – Binário Para Hexadecimal e Vice-Versa

Para converter um valor binário para hexa basta agrupar, da direita para a esquerda, grupos de 4 em 4 bits e trocar o valor dos quatro bits pelo respectivo valor em hexadecimal, como na tabela II. Por exemplo: 1000101010000111_b deve ser agrupado: 1_0001_0101_0000_1111 => 0x1150F (ou $1150F_h$).

Na conversão de hexa para binário faz-se o contrário, ou seja, cada algarismo, cada dígito em hexa gera 4 bits em binário, sendo que cada bit deve representar um valor 1 ou 0 que resulte no valor em hexa, ver tabela II. Por exemplo, 0x30F7 deve ser convertido como: $0011_0000_1111_0111_b$

O procedimento para conversão de decimal para hexa segue o algoritmo descrito na conversão de decimal para binário, porém com o divisor igual a 16. Conversões de decimal para hexa e vice-versa são menos usuais. Há outros sistemas de numeração em sistemas digitais, como octal; porém de pouca importância prática e que não serão abordados no contexto desse livro.

1.1.7 – Números Naturais x Inteiros

Até o momento vimos como representar em binário os números Naturais ($\mathbb{N}$), pois consideramos apenas número positivos. Como representar os números Inteiros ($\mathbb{Z}$) em binário? O conjunto $\mathbb{Z}$ é composto por todos os valores inteiros negativos e positivos; vamos ver como representar negativos em binário.

No final do capítulo, a Fig. 1.12 resume algumas formas de interpretação dos valores que um vetor de cinco bits binários pode ter. No anel mais externo vemos os valores do vetor binário, seguido das interpretações em Complemento para 2, Sinal-Magnitude, e no anel mais interno os números sem sinal, Naturais ($\mathbb{N}$), como será visto em seções seguintes.

1.1.7.1 – Sinal-Magnitude

Uma forma de representar valores negativos é reservar um bit para sinal enquanto os outros bits mostram a magnitude do número, representação esta chamada sinal-magnitude (*Signed Magnitude Representation*, SMR).

Usamos o MSb para o sinal e os bits restantes para a magnitude do valor. Duas razões para selecionar o MSb são que esse bit varia menos, portanto, separa dois conjuntos de valores mais facilmente, além de ficar mais parecido com a forma como escrevemos números em matemática, e.g. -2, +14, -115. O valor '0' no MSb é usado para sinal (+) e '1' define valores negativos (-).

A tabela III mostra a representação sinal-magnitude para um byte. Note que se trata de uma codificação, uma forma de representação que parte de uma escolha. Define-se o papel dos bits dentro do byte e, a partir dessa escolha, o sistema deve processar informações respeitando esta escolha arbitrária. Podemos observar que na

notação sinal-magnitude acima temos dois valores que representam o zero, o +0 e -0. Sinal-magnitude foi usado na construção de computadores antigos, com IBM 7090[3].

Tabela III

campo	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		$\mathbb{Z}$	$\mathbb{N}$
+	0	0	0	0	0	0	0	0		+0	0
	0	0	0	0	0	0	0	1		+1	1
	...									...	...
	0	1	1	1	1	1	1	0		+126	126
	0	1	1	1	1	1	1	1		+127	127
–	1	0	0	0	0	0	0	0		-0	128
	1	0	0	0	0	0	0	1		-1	129
	...									...	...
	1	1	1	1	1	1	1	0		-126	254
	1	1	1	1	1	1	1	1		-127	255
	sinal	*magnitude do número*									

Apesar de o sistema ter alguns inconvenientes, ainda tem uso em algumas aplicações.

1.1.7.2 – Complemento Para 1

Como será visto, um dos problemas com representações de números com sinal são as operações aritméticas que podem ser feitas de forma mais ou menos direta, influenciando no nível de complexidade do hardware.

Tabela IV

campo	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		$\mathbb{Z}$	$\mathbb{N}$
+	0	0	0	0	0	0	0	0		+0	0
	0	0	0	0	0	0	0	1		+1	1
	0	0	0	0	0	0	1	0		+2	2
	...										
	0	1	1	1	1	1	1	0		+126	126
	0	1	1	1	1	1	1	1		+127	127
–	1	1	1	1	1	1	1	1		-0	255
	1	1	1	1	1	1	1	0		-1	254
	1	1	1	1	1	1	0	1		-2	253
	...										
	1	0	0	0	0	0	1	0		-125	130
	1	0	0	0	0	0	0	1		-126	129
	1	0	0	0	0	0	0	0		-127	128

O formato de representação "*complemento para 1*" é obtido invertendo-se todos os bits de um número binário; ou seja, obtendo-se o complemento desse valor. Dessa forma, todos os bits 0 se tornam 1 e vice-versa. A tabela IV ilustra como são representados os valores em complemento para 1. A notação complemento para 1 foi utilizada em vários computadores nos anos 1960 e 1970, como PDP-1, da Digital[4] e os da série UNIVAC 11xx[5]. Atualmente, a maioria dos processadores utiliza a forma de representação complemento para 2, discutida a seguir.

1.1.7.3 – Complemento Para 2

Vários livros sobre eletrônica digital explicam como se obtém o "*complemento para 2*" (*Cp2*), porém há uma propriedade nessa forma de codificação que é mais importante do que a forma como se obtém, e que será explicada em seguida; embora a explicação tradicional é que obtemos *Cp2* fazendo-se o complemento para 1 adicionando +1. Exemplificando, suponha o valor +69, em binário: 01000101; para se obter o *Cp2* primeiro inverte-se todos os bits: 10111010 e em seguida adiciona-se +1 a esse número. Nesse caso, -69 = 10111010 + 00000001 = 10111011. A tabela V ilustra como são representados valores em *complemento para 2* de um byte.

[3] Ver IBM7090 series: https://en.wikipedia.org/wiki/IBM_7090
[4] PDP-1, DEC: https://en.wikipedia.org/wiki/PDP-1 (acesso 25/08/2019)
[5] UNIVAC,Unisys: https://en.wikipedia.org/wiki/UNIVAC_1100/2200_series#1107 (acesso 25/07/2019)

Contudo, formalmente, um número codificado em *Cp2* consiste em se inverter o sinal do bit mais significativo do número da seguinte forma:

$$X = -x_{(n-1)}.2^{(n-1)} + x_{(n-2)}.2^{(n-2)} + \ldots + x_{(3)}.2^{(3)} + x_{(2)}.2^{(2)} + x_{(1)}.2^{(1)} + x_{(0)}.2^{(0)} \quad (1.4)$$

Esse tipo de representação apresenta características que se traduzem em várias vantagens sobre os anteriores e faz dele o modo dominante de representação binária para valores negativos. As principais características e seus benefícios são destacados a seguir. Considere um byte, 8 bits, codificado em *Cp2*. Quando $x_{(n-1)} = 1$ teremos: $-1.2^{(7)}+\ldots+x_{(0)}.2^{(0)}$; ou seja, -128 que será somado ao restante dos termos do polinômio.

Tabela V											
campo	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		ℤ	ℕ
+	0	0	0	0	0	0	0	0		+0	0
	0	0	0	0	0	0	0	1		+1	1
	0	0	0	0	0	0	1	0		+2	2
	...										
	0	1	1	1	1	1	1	0		+126	126
	0	1	1	1	1	1	1	1		+127	127
–	1	0	0	0	0	0	0	0		-128	128
	1	0	0	0	0	0	0	1		-127	129
	...										
	1	1	1	1	1	1	0	1		-3	253
	1	1	1	1	1	1	1	0		-2	254
	1	1	1	1	1	1	1	1		-1	255

Se todos os outros bits forem '0' temos $-1.2^{(7)}+ 0.2^{(n-2)}\ldots + 0.2^{(0)}$ = -128. Esse será o menor número negativo que se pode escrever em *Cp2*. Quando $x_{(n-1)}$ = 1 e os outros bits forem '1', temos $-1.2^{(7)}+ 1.2^{(n-2)}\ldots + 1.2^{(0)}$ = -128+127 = -1; ou seja, todos os termos anteriores ao bit mais significativo $x_{(n-1)}$ somados resultam uma unidade a menos que $1.2^{(N-1)}$. Isso resulta que, não importa a quantidade de bits do número, o valor -1 será sempre representado por todos os bits iguais a '1'. No caso do exemplo com o byte, -1 = 1111 1111, e com16 bits, -1 = 1111 1111 1111 1111 (*0x*FFFF). O valor -2 será $-1.2^{(7)} + 1.2^{(n-2)}\ldots +1.2^{(1)} + 0.2^{(0)}$ = -128+126.

Essa forma de representação elimina a situação de +0 e -0. O maior valor positivo que poderemos representar será $-0.2^{(7)}+ 1.2^{(n-2)} \ldots + 1.2^{(0)}$ = 127. Portanto, quando o bit mais significativo $x_{(n-1)}$ = 0, esse código tem capacidade de representar do valor zero até o maior valor positivo, que é $2^{(N-1)}$-1. Essa representação é assimétrica em relação ao zero. O bit mais significativo (MSb) representa o sinal do número, quando esse bit for '0', o valor é positivo, e quando for '1', o valor será negativo. Hoje, a grande maioria dos circuitos aritméticos, que serão discutidos adiante, usa essa forma de representação de números inteiros positivos e negativos. Verilog, por exemplo, considera esse tipo de representação numérica para tratar com valores com sinal.

1.1.7.4 – Excesso-K (Offset Binary)

Um valor em Excesso-K é representado por um valor sem sinal que é K unidades maior que o valor que se pretende representar; ou seja, o valor X = *excesso-*K. Assim, o valor zero será representado pelo valor K, porque $X = K-K = 0$; e o valor negativo ($-K$) é representado por todos os bits zerados, porque $X = 0-K$. A representação de *Excesso-*K usa um número previamente especificado K como o valor do viés, da polarização. Como exemplo, essa notação é usada no valor dos expoentes do padrão IEEE-754 para representação valores reais em ponto flutuante, que veremos adiante.

1.1.8 – Números Reais

Números Reais (ℝ) são usados para representar uma quantidade contínua que inclui o zero, os positivos e os negativos. Trata-se de uma expansão dos números Racionais (ℚ), que é o conjunto dos números representados por uma *razão* ou *fração* de um valor a/b, ambos números inteiros. Em outras palavras, um número real deve considerar a parte da fração de um número, e essa fração pode ser infinita (como 1/3, π, etc.).

Como em sistemas digitais o número de bits (de fios paralelos) é finito, um dos problemas na representação de números Reais é que tais valores não podem ser representados com precisão infinita em suas partes fracionárias. Tendo isso em mente, é possível usar de várias formas para se representar a parte fracionária dos valores Reais.

1.1.8.1 – Ponto Fixo

Considerando um valor qualquer de ***N*** bits, pode-se representar um número real $\mathbb{R}$ dividindo os ***N*** bits para representar uma parte inteira e outra fracionária. É possível trabalhar com ponto fixo sem sinal ($+\mathbb{R}$), mas o mais comum é dividir a palavra digital em três partes: 1 bit de sinal, ***m*** bits para a parte inteira e ***n*** bits para a parte fracionária. O formato mais conhecido é ***Qm.n***, onde ***Q*** é o bit de sinal, ***m*** é a quantidade de bits inteiros, ***n*** a quantidade de bits decimais; com números representados no formato *Cp2*. O bit MSb é usado para sinal, assim temos, por exemplo Q6.9:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	***I***	***I***	***I***	***I***	***I***	***I***	***F***	***F***	***F***	***F***	***F***	***F***	***F***	***F***	***F***

O exemplo tem 16 bits, o dígito 15 é usado como sinal, os bits 14, 13, 12, 11, 10 e 9, ou seja, os índices [14:9], formam a parte inteira do número; enquanto os bits restantes, do índice 8 até 0 [8:0], formam a parte fracionária. Nesse exemplo, o maior valor positivo *inteiro* será 2^m-1: ***0***111111,000000000 = 63 e o menor *inteiro* será ***1***000000,000000000 = – 64.

O menor valor real positivo nessa representação, ***+MinR***, consiste do bit mais à direita da parte fracionária igual a '1' com os outros bits zerados, que pode ser calculado por 2^{-n}. No exemplo, ***0***000000,000000001 = 2^{-9} = 0,001953125; e o valor simétrico negativo, ***–MinR***, em *Cp2* será escrito = ***1***111111,111111111.

O maior valor real positivo, ***+MaxR***, ocorre com todos os bits '1' exceto o sinal, calculado por $(2^m\text{-}1)+(1–\textbf{\textit{MinR}})$; e.g. ***0***111111,111111111 = 63,998046875, e o simétrico negativo = ***1***000000,000000001 = –63,998046875; embora em *Cp2* o menor real negativo do exemplo, ***–MaxR*** = ***1***000000,000000000 = – 64.

Esse formato de representação não pode representar nenhum valor no intervalo {***–MinR*** > ***x*** < ***+MinR***} que não seja o valor zero. Essa região é denominada *underflow*, na qual qualquer parte fracionária menor que 2^{-n} se torna zero. Por outro lado, valores fora do intervalo {***–MaxR*** < ***x*** > ***+MaxR***} também não podem ser representados, porque estão na região denominada *overflow*; ou seja, são maiores que o maior real positivo, ***+MaxR***, ou menores que o menor valor real negativo ***–MaxR***.

Note que o número é representado por 1's e 0's e a posição da vírgula (*ponto em notação inglesa*) é imaginária. Veja um exemplo para mostrar o que essa posição representa: considere +37 em binário: ***0***100101. Dividir por 2 em binário é o mesmo que deslocar uma casa para a direita; assim, ao dividir por 4 deslocamos 2 bits para a direita (da casa decimal imaginária). Dividindo 37/4 = 9,25; o resultado fica: ***0***1001,01. Ao converter esse valor para decimal pela equação (1.3): 1101,01 = **1**•2^3+0•2^2+0•2^1+**1**•2^0+0•2^{-1}+**1**•2^{-2} = **1**•8+**1**•1+**1**•0,25 = 9,25.

Portanto, o projetista de sistemas digitais deve compreender que representar um número com *ponto fixo* é uma escolha mental, que para cada valor representado nos circuitos tem que manter-se a coerência entre o formato ***Qm.n*** e o valor no mundo externo que ele pretende representar.

1.1.8.2 – Ponto Flutuante

A representação em ponto flutuante (*Floating Point*)[6] por vezes é mais eficiente que a de ponto fixo. Nesse tipo de representação dividimos a quantidade de bits em três campos: o *sinal*, a *mantissa* e o *expoente*. Os números são representados da forma $x = (\pm)\ m.b^{exp}$, onde ***m*** é a mantissa, ***b*** é a base usada para representar os dígitos, e ***exp*** é o expoente do número. No caso da base decimal, estamos acostumados a usar essa notação em engenharia, por exemplo: 352**.**10^{-2} para 3,52; ou 1,89**.**10^6 para 1.890.000; ou ainda -17645**.**10^{-3} para -17,645. A diferença para uso em binários é que a base é "2". Assim, temos que a fórmula geral será $x = (\pm)\ m.2^e$.

Como vários computadores poderiam implementar esta mesma fórmula em diferentes formatos, por exemplo, com quantidade de bits variáveis, distintas posições para a mantissa e expoente, distintas quantidades de bits para mantissa e expoente; o IEEE (*Institute of Electrical and Electronic Engineers*) foi incumbido de padronizar um formato para representação em ponto flutuante a ser adotado por todos os projetistas de computadores e softwares[5]. O padrão IEEE-754 foi estabelecido em 1985 e revisado em 2008. A partir de seu estabelecimento não apenas o hardware mas também os softwares, as linguagens de programação e seus compiladores usam o mesmo formato para representar valores de ponto flutuante em 16 bits (*half-precision*), 32 bits (*single-*

[6] Usamos nessa seção o padrão IEEE-754: O Padrão IEEE para Aritmética de Ponto Flutuante (IEEE 754) é um padrão técnico para representação numérica e para aritmética de ponto flutuante estabelecido em 1985 pelo Instituto de Engenheiros Elétricos e Eletrônicos (IEEE).

precision), 64 bits (*double-precision*), 128 bits (*quadruple-precision*), e 256 bits (*octuple-precision*) na base binária. O padrão IEEE-754 também prevê formatos na base decimal com 32, 64 e 128 bits.

Foge ao escopo dessa seção detalhar o padrão IEEE-754, porém vamos descrever como o padrão representa *half-precision* e *single-precision*. As tabelas seguintes mostram as representações em 16 e 32 bits. A notação *half-precision* usa 16 bits e serve para armazenar dados, mas não é indicada para operações de aritmética em ponto flutuante. O bit 15 é o sinal, 5 bits são usados para o expoente e 10 bits são usados para a mantissa.

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	E	E	E	E	E	m	m	m	m	m	m	m	m	m	m

31	30	29	28	27	26	25	24	23	22	21	20	…	4	3	2	1	0
S	E	E	E	E	E	E	E	E	m	m	m	…	m	m	m	m	m

O formato IEEE-754 usa a notação *excesso-N* (ver acima) para representar os expoentes. No expoente em 16 bits o valor do excesso (*bias*) é 15 e em 32 bits o excesso é 127. Assim, o expoente em 16 bits é calculado subtraindo-se 15 do valor escrito nos 5 bits reservados para o expoente (D_{14} ... D_{10}). Em 32 bits o expoente é calculado subtraindo-se 127 do valor escrito nos 8 bits do expoente (D_{30} ... D_{23}). Assim, as fórmulas para calcular o valor que os 16 bits, e 32 bits expressam, respectivamente são: $x_h = (-1)^s \cdot m \cdot 2^{(E-15)}$; e $x_f = (-1)^s \cdot m \cdot 2^{(E-127)}$; onde o subscrito x_h significa o valor de x em *floating half-precision* e x_f significa o valor de x em *floating single precision*.

Há muito mais detalhes a saber sobre esse tipo de formato de representação numérica, uma primeira consulta pode ser feita na Wikipedia[7]. Livros de computação também costumam dar uma boa introdução ao padrão; contudo, caso necessário, consulte o próprio padrão[5].

Importante é compreender que podemos usar múltiplos bytes para representar um único valor em ponto flutuante. No caso do *half-precision* usamos 2 bytes e no *single-precision* usamos 4 bytes. Eles podem ser tratados em hardware ou software, seja em um circuito que realiza operação direta sobre os N bits, ou um algoritmo que calcular o resultado byte a byte, como é o caso dos processadores de 8 bits quando realizam operações em ponto flutuante precisão simples. O que importa é que o padrão da representação seja respeitado; ou seja, cada bit dentro da cadeia tem sua função e o software ou hardware deve tratar a representação da forma como foi arbitrado.

Até o momento vimos a representação de números, ou de valores em sistemas binários. Usamos um ou mais bytes e vimos que podemos representar com uns e zeros qualquer valor numérico, observando que teremos limitações porque não temos infinitos dígitos para representar todos os números reais. Ainda assim, mesmo com aproximações, podemos representar quaisquer valores numéricos de forma que atendam nossas necessidades de forma pragmática. Veremos a seguir como representar outras "coisas" nos sistemas digitais, outros "fenômenos" ou "objetos" que não números ou valores numéricos.

1.1.9 – Representando Símbolos Não-Numéricos

Um byte pode facilmente representar um caractere, ou pode representar um tom de cinza de um pixel em uma imagem, um nível de tensão em um sinal de áudio, ou qualquer coisa que se queira. Como consequência, um conjunto de bytes pode representar um texto e um conjunto de pixels pode representar uma imagem, ou ainda uma sequência de valores pode representar uma onda sonora em um áudio. Dentro do contexto de um editor de textos, um byte pode não representar um valor numérico, mas letras e sinais de pontuação. Dentro de um editor de imagem um conjunto de bytes representa valores de cada um dos elementos da imagem – os pixels. Todavia, os circuitos eletrônicos que armazenam, processam e mostram tal informação veem apenas valores $+V$ ou GND ("1s" e "0s") nos fios e na memória que representam tais valores.

Dois exemplos de códigos de representações de caracteres alfanuméricos são o ASCII e o Unicode. ASCII é uma sigla para "*American Standard Code for Information Interchange*"[8] (*código padrão norte-americano para*

[7] Ver: https://en.wikipedia.org/wiki/IEEE_754 (ou em português). (acesso 26/07/2019).
[8] A tabela ASCII, incluindo caracteres estendidos pode ser facilmente encontrada na Internet, e.g.: (acesso 27/07/2019) http://www.asciitable.com/

intercâmbio de informações), que foi criado visando padronizar os códigos para letras, sinais, números e acentos. Abaixo temos uma ilustração de como um *byte* representa uma letra, um símbolo e um número em ASCII – observando que o código ASCII original usa apenas 7 bits – inserimos '0' no MSb.

Com a popularização mundial dos computadores pessoais surgiu a necessidade de representação de caracteres em diversas línguas, com caracteres outros que não apenas os usados para a língua inglesa, incluindo notação matemática. Assim, criou-se o Unicode[9], um padrão que permite a sistemas digitais representar e manipular texto de qualquer sistema de escrita existente, que exigem diacríticos e acentos, por exemplo.

Tabela VI - Três valores em ASCII									
Caractere		Valor em binário							
@		0	1	0	0	0	0	0	0
a		0	1	1	0	0	0	0	1
5		0	0	1	1	0	1	0	1

O padrão é regido por uma organização não governamental e consiste de pouco mais de 107 mil caracteres, uma metodologia para codificação, um conjunto de diagramas de referência visual, e um conjunto de códigos padrões de caracteres e suas propriedades, como caixa alta e baixa, um conjunto de arquivos com dados de referência, além de regras para normalização, decomposição, ordenação alfabética e renderização em telas de interfaces digitais.

Uma das causas do grande sucesso da eletrônica digital sobre sistemas analógicos é que o sistema binário pode facilmente representar qualquer coisa, desde que se obedeça a convenção sobre o que está sendo representado e o contexto de uso da informação.

1.2 – Aritmética Binária

Vimos nas seções anteriores que podemos representar qualquer valor numérico no sistema binário, incluindo formatos de números negativos e fracionários. Portanto, se é possível representar números em binário, deve ser possível realizar operações aritméticas com eles.

Vamos escolher para representação binária a regra da *notação posicional*; ou seja, a posição do algarismo indica sua potência no sistema. Vamos ainda usar em binário os mesmos algoritmos que utilizamos para matemática decimal {1,0}. Fazendo isso, a diferença da aritmética binária com a matemática com números decimais surge em poucas regras a serem observadas quando trabalhando em binário, como veremos.

Em decimal, somamos ou subtraímos os números que estão nas mesmas posições na representação: unidade ± unidade, dezena ± dezena, centena ± centena, e assim por diante; o mesmo procedimento é aplicado ao binário.

A adição, quando um valor extrapola o valor máximo representado em decimal, no caso um valor maior que nove, enviamos um transporte (em inglês *carry*) para a posição adjacente. Na subtração, quando o valor resultante é uma representação menor que zero, tomamos emprestado uma unidade na posição adjacente.

Em ambos os casos, adição ou subtração, faremos o mesmo usando números binários; ou seja, somamos ou subtraímos bits nas posições correspondentes: $2^0 \pm 2^0$, então $2^1 \pm 2^1$, e assim por diante, e transportamos ou tomamos emprestado bits das posições adjacentes quando necessário. Vejamos as regras para aritmética em binário em cada um dos casos, de adição e subtração.

1.2.1 – Adição

A adição binária é direta e intuitiva: 0+0 = 0, 0+1 = 1, 1+0 = 1, e 1+1 = 2; mas, não existe uma representação do algarismo "2" em binário, temos apenas os algarismos {1,0}. Em binário o valor $2 = 10_b$. Por esse motivo, o resultado da soma em binário S = 0 e transportamos 1 para a posição à esquerda da coluna que estamos somando, ou seja, fazemos o *carry* = 1 para a posição adjacente à esquerda.

Em outras palavras, da mesma forma como fazemos em decimal, transportamos o valor *carry* para o bit de ordem superior. Se estamos somando a posição 2^3, o carry vai para 2^4, se somamos 2^7 o carry vai para 2^8.

[9] Ver: https://www.unicode.org/standard/standard.html (27/07/2019)

Tabela VII - Regras para ADIÇÃO binária					
A		B		Carry	S
0	+	0	=	0	0
0	+	1	=	0	1
1	+	0	=	0	1
1	+	1	=	1	0

Seguindo as regras da Tab. VII, vamos ver um exemplo de soma em binário. Considere o c*arry out* (c_{ok} = *vai 1*) o bit de transporte resultante da soma na posição ***k***; que será transportado para a coluna adjacente à esquerda ***k***+1 no número. Da mesma forma, considere *carry in* ($\boldsymbol{c_{ik}}$ = *vem 1*) o bit que veio transportado da posição anterior (***k***-1) e deve ser somado na posição atual ***k***:

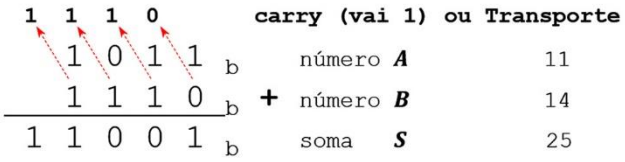


Fig. 1.5: Exemplo de soma em binário com transporte (carry).

No exemplo da Fig. 1.5, na posição da unidade (2^0) o carry de entrada é zero. Assim, temos que somar apenas dois bits: $\boldsymbol{S_0}$ = 1+0 = 1 e o transporte c_{o0} = 0. Por outro lado, na posição 2^1, e subsequentes, devemos considerar o *carry in* ($\boldsymbol{c_{ik}} = \boldsymbol{c_{ok-1}}$), então a adição nessas posições são feitas com três bits. No exemplo, temos $c_{i1} = c_{o0} = 0$; assim, a soma fica $\boldsymbol{S_1}$ = 0+1+1 = 0 e transporte c_{o1} = 1. Na posição 2^2, $c_{i2} = c_{o1}$ = 1 e somamos 3 bits: 1+0+1 = 0, com *carry out* c_{o2} = 1. Note que na posição 2^3 vamos somar o $c_{i3} = c_{o2}$ = 1 com mais dois bits uns, a soma fica: 1+1+1 = 1 com transporte c_{o3} = 1. Faz sentido porque 1+1+1 = 3 e o valor 3 em binário se escreve "11", ou seja, a soma na posição ***k*** = 1 e transporta-se 1 para a posição ***k***+1. Na posição 2^4 somamos $c_{i4} = c_{o3}$ = 1 e consideramos 0 na posição 2^4 para os números ***A*** e ***B***, porque zeros à esquerda não alteram os valores dos números. O resultado da posição 2^4 é S_4 = 1+0+0 = 1 e $\boldsymbol{c_{o4}}$ = 0. O resultado aparece abaixo como o número S = 11001_b. Compare os valores somados em decimal, somamos 11 com 14, obtendo S = 25 (ou S = $1.2^4+1.2^3+0.2^2+0.2^1+1.2^0$).

1.2.2 – Subtração

A subtração é executada da mesma forma como executamos o algoritmo acima, e como aprendemos na matemática decimal, ou seja: 0-0 = 0, 1-0 = 1, 1-1 = 0, e no caso especial de 0-1 = 1 com pedido de empréstimo para a posição de ordem superior. As regras para o binário são sintetizadas na tabela VIII, na qual representamos ***S*** = ***A*** - ***B***:

Tabela VIII - Regras para SUBTRAÇÃO binária					
A		B		Carry	S
0	-	0	=	0	0
0	-	1	=	1	1
1	-	0	=	0	1
1	-	1	=	0	0

Como na soma, a interpretação é intuitiva, porque se ***A*** = 0 e ***B*** = 0, 0-0 = 0; e se ***A*** = 1 e ***B*** = 1, então 1-1 = 0. Se ***A*** = 1 e ***B*** = 0 o resultado será 1-0 = 1, ou seja, ***A*** = 1 e tiramos nada de ***A***, portanto ficamos com S = 1. A única interpretação que pode gerar confusão é ***A*** = 0 e ***B*** = 1, onde 0-1 = 1 e *carry* = 1. Para explicar o que ocorre nessa situação, vejamos um exemplo, com o minuendo 93 e o subtraendo 78.

```
  93
- 78
= 15
```

Primeiro testamos os valores das unidades: de 3 não podemos tirar 8, portanto, tomamos emprestado uma dezena da posição seguinte e somamos ao 3. Agora, 13-8 = 5. Para continuarmos o algoritmo há três formas de considerarmos o desconto: (***a***) somamos +1 ao subtraendo da dezena, ou seja, fazemos 7+1 = 8 dezenas, e então subtraímos 9-8 = 1; (***b***) subtrairmos -1 do minuendo na coluna da dezena, ou seja, fazemos 9-1 = 8, e então desse resultado subtraímos o valor do subtraendo da dezena, ou seja 8-7 = 1; ou (***c***) subtraímos o valor do

minuendo menos o subtraendo da coluna da dezena 9-7 = 2, e então descontamos o valor que emprestamos desse resultado 2-1 = 1.

Não importa qual algoritmo alguém queira seguir, o resultado é que a dezena ficará com 1. O resultado de toda a operação será 15. Agora vejamos as três formas de resolver em binário a partir do exemplo da Fig. 1.6:

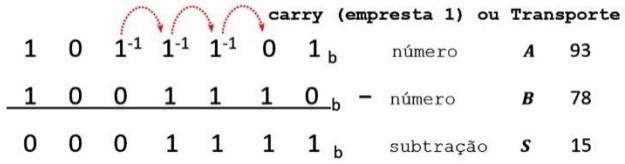


Fig. 1.6: Exemplo de subtração em binário com empréstimo (carry).

Na coluna da unidade (2^0), 1-0 = 1 sem *carry out*. Na coluna 2^1 vemos que de 0 não se pode subtrair 1, portanto, pedimos emprestado da coluna 2^2. Quando se toma emprestado, o minuendo na coluna 2^1 fica ($10_b = 2$), portanto a subtração na coluna 2^1 agora será 2-1 = 1, por isso o resultado na coluna 2^1 fica S = 1. Usando o algoritmo acima (***a***) descontamos o 1 emprestado para a coluna 2^1 do minuendo da coluna 2^2, que fica 1-1 = 0. Retornamos ao caso da coluna 2^1, porque de 0 não poderemos subtrair 1 que está no subtraendo 2^1, então temos que tomar 1 emprestado da coluna 2^3, e assim por diante. Até que na coluna 2^4 temos um minuendo 1 e um subtraendo 0, portanto, ao pagar o *empréstimo* para a coluna 2^3 ficamos com 0 e não precisamos mais pedir emprestado para subtrair $S_4 = (1-1)-0 = 0-0 = 0$.

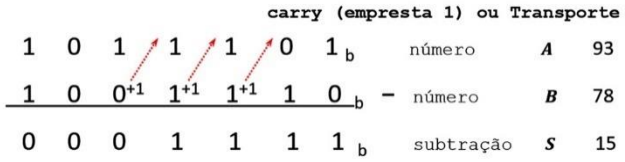


Fig. 1.7: Exemplo de subtração binária com empréstimo, algoritmo (a) descontando do minuendo k+1.

Usando o algoritmo (***b***), emprestamos para a coluna 2^1 e para ajustar o empréstimo somamos 1 no subtraendo da coluna 2^2, que fica com $10_b = 2$. O valor do minuendo na coluna 2^2 é 1, temos que subtrair dele um valor 2; portanto, tomamos emprestado do subtraendo de 2^3. Fazendo isso, o valor do minuendo de 2^2 agora é $11_b = 3$, que subtraído do subtraendo de 2^2 dá uma resposta 3-2 = 1, colocado no resultado da coluna 2^1. Como foi pedido emprestado um bit na coluna 2^3, repete-se o ajuste somando 1 ao subtraendo 2^3 e o processo se repete até que, na coluna 2^4 quando somamos 1 ao subtraendo 0+1 = 1 que, subtraído do minuendo $2^4 = 1$, resulta 1-1 = 0.

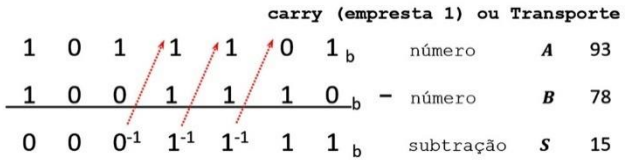


Fig. 1.8: Exemplo de subtração binária com empréstimo, algoritmo (b) tomando emprestado do subtraendo k+1.

Com o terceiro algoritmo (***c***), tomamos emprestado 1 da coluna 2^2 para a 2^1. Ao ajustar o empréstimo descontamos do resultado da subtração do minuendo 2^2 pelo subtraendo 2^2. Nesse caso, 1-1 = 0, mas, se o resultado da coluna 2^2 é zero, como emprestar para a coluna 2^1? Tomando emprestado da coluna 2^3; assim, o resultado seria $10_b = 2$, e ajustando o empréstimo fica 2-1 = 1, porém, devendo para a coluna de potência superior. Continuando, o resultado do minuendo 2^3 menos o subtraendo 2^3 também é zero e não é possível pagar o empréstimo anterior; então temos que tomar emprestado da coluna 2^4. Por sua vez, o resultado do minuendo 2^4 menos o subtraendo 2^4 é 1-0 = 1, então, essa coluna tem como pagar o empréstimo tomado pela coluna 2^3, e assim o resultado da coluna 2^4 será 0.

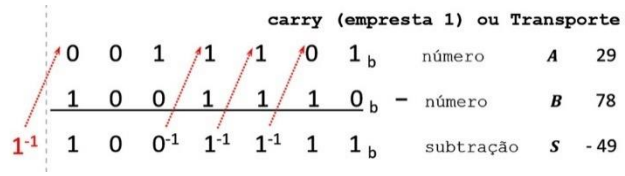


Fig. 1.9: Exemplo de subtração binária com empréstimo, algoritmo (c) descontando do resultado k+1.