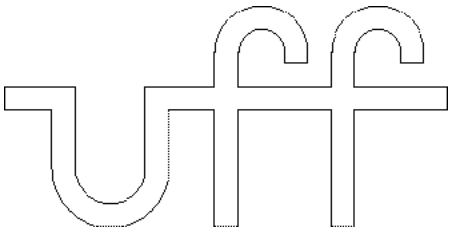
Apostila
com
Códigos Demonstrativos
usando a linguagem Verilog
para
Circuitos Digitais

(Versão A2026M03D02)



Universidade Federal Fluminense

Alexandre Santos de la Vega

Departamento de Engenharia de Telecomunicações – TET

Escola de Engenharia – TCE

Universidade Federal Fluminense – UFF

Março – 2026

.

621.3192 (*) D278 (*) 2026	de la Vega, Alexandre Santos Apostila com Códigos Demonstrativos usando a linguagem Verilog para Circuitos Digitais / Alexandre Santos de la Vega. – Niterói: UFF/TCE/TET, 2026. 194 (sem romanos) ou 214 (com romanos) p. (*) Apostila com Códigos Demonstrativos – Graduação, Engenharia de Telecomunicações, UFF/TCE/TET, 2026. 1. Circuitos Digitais. 2. Técnicas Digitais. 3. Telecomunicações. I. Título.

(*) OBTER INFO NA BIBLIOTECA E ATUALIZAR !!!

Aos meus alunos.

Prefácio

O trabalho em questão aborda os tópicos a serem apresentados na disciplina Circuitos Digitais.

O material completo, formado por apostilas e tutoriais, encontra-se dividido nos seguintes volumes:

1. O conteúdo teórico principal pode ser encontrado no volume entitulado Apostila de Teoria para Circuitos Digitais.

2. Uma introdução teórica sobre dispositivos configuráveis é apresentada no volume entitulado Tutorial sobre Circuitos Digitais Configuráveis para Circuitos Digitais.

3. O conteúdo prático pode ser encontrado nos seguinte volumes:

   (a) Apostila com Códigos Demonstrativos usando a linguagem VHDL para Circuitos Digitais.

   (b) Apostila com Códigos Demonstrativos usando a linguagem Verilog para Circuitos Digitais.

Os documentos foram escritos com o intuito de servir como uma referência rápida para os alunos dos cursos de graduação e de mestrado em Engenharia de Telecomunicações da Universidade Federal Fluminense (UFF).

O material básico utilizado para o conteúdo teórico foram as minhas notas de aula, que, por sua vez, originaram-se em uma coletânea de livros sobre os assuntos abordados. Por outro lado, os códigos demonstrativos e as especificações dos trabalhos propostos são completamente autorais.

A motivação inicial para o desenvolvimento desse trabalho foi a de aumentar o dinamismo das aulas. Logo, deve ficar bem claro que os documentos produzidos não pretendem substituir os livros textos ou outros livros de referência. Pelo contrário, espera-se que eles sejam utilizados como ponto de partida para estudos mais aprofundados, utilizando-se a literatura existente.

Espero conseguir manter o presente texto em constante atualização e ampliação.

Correções e sugestões são sempre bem-vindas.

Alexandre Santos de la Vega
UFF / TCE / TET

Agradecimentos

Àqueles alunos do Curso de Engenharia de Telecomunicações e professores do Departamento de Engenharia de Telecomunicações (TET), da Escola de Engenharia (TCE), da Universidade Federal Fluminense (UFF), que colaboraram com críticas e sugestões bastante úteis à finalização da versão inicial deste trabalho.

Aos ex-funcionários do TET, Arlei, Carmen Lúcia, Eduardo Wallace, Francisco e Jussara, pelo apoio constante.

Aos meus alunos, que, além de servirem de motivação principal, obrigam-me sempre a me tentar melhorar, em todos os sentidos.

Mais uma vez, e sempre, aos meus pais, por tudo.

Alexandre Santos de la Vega
UFF / TCE / TET

Apresentação do material didático

Metodologia de construção

- O material aqui apresentado não é fruto de um projeto educacional envolvendo idealização, planejamento, pesquisa, estudo, estruturação, desenvolvimento, revisão e edição.
- Pelo contrário, ele nasceu, evoluiu e tem sido mantido de uma forma bem orgânica.

Histórico

- Em 1995, o autor ingressou no Departamento de Engenharia de Telecomunicações (TET) da Universidade Federal Fluminense (UFF) e, desde então, tem sido responsável por diversas disciplinas oferecidas pelo TET para o Curso de Engenharia de Telecomunicações, da Escola de Engenharia da UFF (TCE/UFF), e para o Curso de Ciência da Computação, do Instituto de Computação da UFF (IC/UFF).
- Na época do seu ingresso, Circuitos Digitais era trabalhado em duas disciplinas (Técnicas Digitais I e Técnicas Digitais II) oferecidas pelo TET para o Curso de Engenharia de Telecomunicações e em uma disciplina única (Circuitos Digitais) para os demais Cursos.
- Após algumas reformas curriculares, as disciplinas e os conteúdos convergiram para uma disciplina única, entitulada Circuitos Digitais.
- Ao longo do tempo, o autor colaborou com todas as citadas disciplinas.
- Para dar suporte às aulas, foram elaboradas as primeiras notas de aula (manuscritas), usando a referência [HP81] como livro texto.
- As notas de aulas manuscritas foram sofrendo alterações, ao longo do tempo e nas diversas trocas de disciplinas, incorporando informações de outras referências bibliográficas.
- No final da década de 2010, o TET realizou uma nova reforma curricular, a qual acarretou uma redução na quantidade e na carga horária das disciplinas. Isso provocou uma reformulação na abordagem dos tópicos da disciplina.
- Assim, em 2010, com os objetivos iniciais de melhor organizar os manuscritos e de atender aos apelos dos alunos por cópia dos manuscritos, eles foram apenas transcritos para o Sistema de Preparação de Documentos LaTeX [KD04] , [MG04]. Assim, surgiu a primeira versão da apostila de teoria.

- A partir daí, com a maturação gradual que a disciplina foi ganhando a cada período letivo, novos conteúdos foram surgindo. Ora por curiosidade do autor, procurando incorporar um determinado tópico na disciplina. Ora por curiosidade dos alunos, por demandarem algum assunto em especial. Ora por necessidade pedagógica, pois, ao se perceberem dúvidas recorrentes dos alunos, novas formas de abordagem têm sido testadas.
- Além disso, como filosofia educacional do autor, as questões que fazem parte de toda e qualquer forma de avaliação formal da disciplina (testes, provas, trabalhos) são anexadas ao conteúdo, na forma de exercícios propostos.
- Ainda como filosofia educacional do autor, a apostila de teoria não apresenta um conjunto completo de figuras que ilustrem os assuntos abordados. Pelo contrário, é demandado aos alunos que eles gerem as suas próprias figuras, a partir de um aplicativo computacional adequado.
- Posteriormente, objetivando incentivar os alunos a utilizarem ferramentas de simulação, ou mesmo a modificarem códigos existentes, o autor organizou um texto introdutório sobre Linguagem de Descrição de *Hardware* (ou *Hardware Description Language* ou HDL) e sobre a linguagem (VHDL), bem como desenvolveu diversos códigos autorais nesta linguagem.
- Assim, em 2017, surgiu a primeira versão de uma nova apostila, contendo códigos demonstrativos, para simulação, relativos aos tópicos abordados na apostila de teoria, em sala de aula e/ou em alguma forma de avaliação formal da disciplina.
- Procurando trabalhar o surgimento e a evolução de um tipo de implementação denominada Dispositivo Lógico Programável (ou *Programmable Logic Device* ou PLD), o autor organizou um texto introdutório sobre o assunto.
- Assim, em 2019, surgiu a primeira versão de um novo tutorial sobre Dispositivos Lógicos Configuráveis.
- No primeiro mês do isolamento social, imposto pela Pandemia de COVID-19, o autor realizou uma pesquisa e organizou um texto introdutório sobre uma outra HDL (Verilog), bem como desenvolveu diversos códigos autorais nesta linguagem.
- Assim, em 2020, surgiu a primeira versão de uma nova apostila, contendo códigos demonstrativos, para simulação, relativos aos tópicos abordados na apostila de teoria, em sala de aula e/ou em alguma forma de avaliação formal da disciplina.

Comentários gerais:

- Conforme foi exposto acima, desde o início da sua confecção até o presente momento, sempre foram preparadas diversas versões de cada documento autoral ao longo de um mesmo período letivo. Por essa razão, o identificador "Versão A<ano>M<mês>D<dia>" aparece logo abaixo do título de cada apostila.
- No tocante à apresentação do conteúdo teórico, os manuscritos originais continham apenas tópicos, destinados à abordagem do conteúdo programático durante as aulas. Pode-se dizer que tais manuscritos representavam apenas um roteiro de aula. Gradativamente, com a evolução do material, os tópicos têm sido trocados por textos dissertativos, relativos ao conteúdo abordado.

- No ponto de vista estrutural é que o aspecto dinâmico dos documentos mais se tem feito presente. Buscando uma melhor apresentação dos tópicos abordados, os diversos seccionamentos de texto (capítulos, seções, subseções, etc.) comumente surgem, são mesclados e desaparecem, a cada nova versão.
- Por tudo isso, pode-se asseguradamente dizer que todo o material produzido encontra-se em constante atualização.
- Na preparação das aulas, têm sido utilizados os seguintes livros:
  - Livros indicados pela ementa da disciplina: [IC08], [Tau82].
  - Outros livros indicados: [HP81], [Rhy73], [TWM07], [Uye02].

Teoria abordada no material didático

Um resumo do conteúdo programático é apresentado a seguir, assumindo um período letivo de 15 semanas, com uma carga horária total de 60 horas.

Circuitos digitais combinacionais <28 horas>

- Conceitos básicos: busca contextualizar o tópico no âmbito do curso de graduação e apresentar os conceitos que serão necessários ao longo do texto. <2 horas>
- Funções lógicas: que define as bases para a representação de informações não numéricas em circuitos digitais tradicionais. <6 horas>
- Álgebra de Boole: que apresenta um formalismo matemático para a estrutura algébrica da lógica implementada em circuitos digitais tradicionais. <2 horas>
- Formas padrões para representação de expressões booleanas: que define formas de expressões booleanas adequadas a um processo sistemático de simplificação das mesmas. <4 horas>
- Simplificação algébrica de expressões booleanas: que ilustra um processo algébrico para a simplificação sistemática de expressões booleanas. <4 horas>
- Mapa de Karnaugh: que apresenta uma ferramenta tabular para a simplificação sistemática de expressões booleanas. <4 horas>
- Sistemas de numeração: que define as bases para a representação de quantidades numéricas em circuitos digitais. <6 horas>
- Circuitos combinacionais básicos: que apresenta os blocos funcionais combinacionais básicos utilizados em sistemas digitais. <Distribuídos ao longo da apresentação do conteúdo e exercitados na forma de trabalhos>

Circuitos digitais sequenciais <16 horas>

- Conceitos básicos: busca contextualizar o tópico no âmbito do curso de graduação e apresentar os conceitos que serão necessários ao longo do texto. <2 horas>
- Elementos básicos de armazenamento: apresenta os elementos de armazenamento utilizados nos circuitos seqüenciais abordados neste texto. <6 horas>
- Circuitos seqüenciais do tipo *clock-mode*: define as características dessa classe de circuitos e aborda os procedimentos, as técnicas e as ferramentas de análise e de projeto para circuitos da classe. <8 horas>

- Circuitos seqüenciais do tipo *pulsed*: define as características dessa classe de circuitos e aborda os procedimentos, as técnicas e as ferramentas de análise e de projeto para circuitos da classe. <Apenas contextualizado em Conceitos básicos>
- Circuitos seqüenciais do tipo *level-mode*: define as características dessa classe de circuitos e aborda os procedimentos, as técnicas e as ferramentas de análise e de projeto para circuitos da classe. <Apenas contextualizado em Conceitos básicos>

Circuitos digitais configuráveis <8 horas>

- Circuitos configuráveis: apresenta uma breve descrição sobre circuitos digitais que podem ser configurados para cumprir diferentes funções, combinacionais e/ou sequenciais. <2 horas>
- Noções básicas sobre Linguagem de Descrição de *Hardware* (*Hardware Description Language* ou HDL). <2 horas>
- Noções básicas sobre as HDLs comumente utilizadas (VHDL e Verilog). <4 horas>

Aplicações

- Aplicações: exemplos de aplicações são distribuídos ao longo da apresentação do conteúdo e exercitados na forma de trabalhos.

Objetivos da disciplina

- Apresentar a base matemática que fundamenta a análise e o projeto de Circuitos Digitais convencionais.
- Abordar os seguintes conteúdos:
  - Circuitos digitais combinacionais (circuitos lógicos e circuitos numéricos).
  - Circuitos digitais sequenciais.
  - Noções de circuitos digitais configuráveis.
  - Noções de linguagens de descrição de *hardware*.
- Trabalhar a análise e o projeto de circuitos digitais mais comumente utilizados.

Sumário

Prefácio v

Agradecimentos vii

Apresentação do material didático ix

Teoria abordada no material didático xiii

Objetivos da disciplina xv

Sumário xvii

I Introdução 1

1 Linguagens de descrição de *hardware* 3
1.1 Introdução . . . 3
1.2 Abordagem hierárquica . . . 3
1.3 Níveis de abstração . . . 4
1.4 Linguagens de descrição de *hardware* . . . 5

2 Introdução à linguagem Verilog 7
2.1 Histórico da linguagem Verilog . . . 7
2.2 Verilog como linguagem . . . 8
2.2.1 Considerações gerais . . . 9
2.2.2 Identificadores . . . 9
2.2.3 Palavras reservadas . . . 9
2.2.4 Identificadores definidos pelo usuário . . . 11

II Circuitos Combinacionais: Construções básicas 13

3 Exemplos de construções básicas 15
3.1 Introdução . . . 15
3.2 Códigos para Tabela Verdade . . . 15
3.3 Códigos para equivalentes formas de expressão lógica . . . 20
3.4 Códigos para equivalentes formas de descrição . . . 27

4 Exemplos de aplicação direta de portas lógicas 39
4.1 Introdução . . . 39
4.2 Elemento de controle . . . 39

4.3 Elemento de paridade 42
4.4 Elemento de igualdade 44

III Circuitos Combinacionais: Blocos funcionais 47

5 Exemplos de blocos funcionais 49

6 Detector de igualdade entre dois grupos de *bits* 51
6.1 Detector de igualdade para um *bit* 51
6.2 Detector de igualdade para quatro *bits* 55
6.3 Detectores de igualdade baseados em detector para um *bit* 58

7 Selecionadores: MUX, DEMUX e *address decoder* 63
7.1 Introdução 63
7.2 Multiplexador (MUX) 63
7.2.1 MUX 2x1 63
7.2.2 MUX 4x1 65
7.2.3 MUX 8x1 68
7.3 Demultiplexador (DEMUX) 69
7.3.1 DEMUX 2x1 69
7.3.2 DEMUX 4x1 71
7.3.3 DEMUX 8x1 74
7.4 Decodificador de endereços (*address decoder*) 75
7.4.1 *Address decoder* 1x2 75
7.4.2 *Address decoder* 2x4 78
7.4.3 *Address decoder* 3x8 81

8 Deslocador configurável (*barrel shifter*) 85
8.1 Introdução 85
8.2 Deslocador parametrizado 85
8.3 *Barrel shifter* com deslocamento genérico 93

9 Decodificador 97
9.1 Introdução 97
9.2 Decodificador com validação de código 97

10 Conversor de códigos 103
10.1 Introdução 103
10.2 Conversor Binário-*One-hot* 103
10.3 Conversor Binário-Gray 103

11 Separador e relacionador de *bits* 1 e 0 em palavra de N *bits* 105
11.1 Introdução 105
11.2 Separador de *bits* 1 e 0 em palavra de N *bits* 105
11.3 Árbitro da relação entre *bits* 1 e 0 em palavra de N *bits* 108
11.4 Relacionador de *bits* 1 e 0 em palavra de N *bits* 113

12 Somadores básicos **119**
12.1 Introdução . . . 119
12.2 Somador de 2 *bits* ou *half adder* (HA) . . . 119
12.3 Somador de 3 *bits* ou *full adder* (FA) . . . 120
12.4 Somador com propagação de *carry* (CPA ou RCA) . . . 121
12.5 Somador binário com modelo comportamental . . . 125

13 Detector de *overflow* e saturador **129**
13.1 Introdução . . . 129
13.2 Detector de *overflow* . . . 129
13.3 Saturador . . . 130

IV Circuitos Sequenciais: Construções básicas **135**

14 Elemento básico de armazenamento: *flip-flop* **137**
14.1 Introdução . . . 137
14.2 *Flip-flop unclocked*: *latch* SR . . . 137
14.3 *Flip-flop clocked* elementar: *latch* SR *clocked* . . . 140
14.4 *Flip-flop clocked* elementar: *latch* D (*clocked*) . . . 142
14.5 *Flip-flop* com estrutura *master-slave* . . . 145
14.6 *Flip-flop edge-triggered* (estrutura realimentada) . . . 150

V Circuitos Sequenciais: Blocos funcionais **159**

15 Exemplos de blocos funcionais **161**

16 Elementos de armazenamento: registradores **163**
16.1 Introdução . . . 163
16.2 Tipos básicos de registradores . . . 163
16.3 Registrador SISO (*Serial-Input Serial-Output*) . . . 164
16.4 Registrador SIPO (*Serial-Input Parallel-Output*) . . . 167
16.5 Registrador PISO (*Parallel-Input Serial-Output*) . . . 170
16.6 Registrador PIPO (*Parallel-Input Parallel-Output*) . . . 176

17 Divisores de freqüência **185**
17.1 Introdução . . . 185
17.2 Divisores de freqüência . . . 185

18 Máquinas de Estados Finitos simples **189**
18.1 Introdução . . . 189
18.2 Máquinas de Estados Finitos simples . . . 189

Referências bibliográficas **193**

Parte I

Introdução

Capítulo 1

Linguagens de descrição de *hardware*

1.1 Introdução

- Desde a implementação do primeiro dispositivo eletrônico em circuito integrado, os avanços tecnológicos têm possibilitado um rápido aumento na quantidade de elementos que podem ser combinados em um único circuito nesse tipo de implementação.
- Naturalmente, com a oferta de uma maior densidade de componentes, a complexidade dos circuitos projetados cresce na mesma taxa.
- Porém, a capacidade de um ser humano em lidar com a idealização, o projeto, a documentação e a manutenção de sistemas com um grande número de componentes é extremamente limitada.
- Dessa forma, torna-se necessário o uso de ferramentas de apoio, adequadas a tal tipo de problema.
- Existem duas técnicas de projeto largamente utilizadas na abordagem de problemas de elevada complexidade:
  - Adotar uma visão hierárquica na elaboração do sistema, de forma que, em cada nível de representação, toda a complexidade dos níveis inferiores seja ocultada.
  - Aumentar o nível de abstração na descrição do sistema, de forma que o foco esteja mais na função desempenhada e menos na implementação propriamente dita.

1.2 Abordagem hierárquica

- Na definição de um sistema de baixa complexidade, pode-se descrever a sua operação de uma forma simples e direta.
- Por outro lado, na definição de sistemas com complexidade elevada, pode-se utilizar o conceito organizacional de hierarquia.
- Em uma abordagem hierárquica, um sistema de complexidade genérica é recursivamente dividido em módulos ou unidades mais simples. O ponto de parada da recursividade é subjetivo e costuma ser escolhido como a descrição comportamental mais simples possível e/ou desejada.

- Nesse sentido, o sistema completo pode ser interpretado como o módulo mais complexo ou mais externo da hierarquia.

- A abordagem hierárquica facilita a descrição, a análise e o projeto dos circuitos, uma vez que cada módulo pode ser tratado como um circuito único, isolado dos demais.

- A descrição hierárquica no sentido do todo para as partes mais simples é chamada de *top-down*.

- Por outro lado, a descrição hierárquica no sentido das partes mais simples para o todo é chamada de *bottom-up*.

- Normalmente, aplica-se um processo *top-down* para a especificação e um processo *bottom-up* para a implementação de sistemas.

1.3 Níveis de abstração

A descrição, a análise e a síntese de circuitos podem ser realizadas em diversos níveis de abstração.

Do ponto de vista do comportamento modelado, os seguintes níveis podem ser considerados:

- Físico-matemático: que adota equações matemáticas para descrever um modelo físico de comportamento. Obviamente, é o modelo mais próximo do comportamento físico do circuito. É tipicamente utilizado na descrição funcional de circuitos analógicos.

- Lógico: que emprega equações lógicas na sua descrição. É naturalmente utilizado na descrição funcional de circuitos digitais.

- Estrutural: que utiliza, intrinsicamente, uma descrição hierárquica do circuito modelado. Inicialmente, são definidos, testados e validados, blocos de baixa complexidade. Em seguida, realizando-se instanciações desses blocos, são definidos blocos de complexidade mais elevada. Tal processo é repetido até que o sistema desejado seja adequadamente definido. Portanto, esse é um modelo que preserva a visão da estrutura física dos circuitos.

- Comportamental: que apresenta um nível de representação mais abstrato e mais distante do sistema físico. Encontra aplicação direta em testes de funcionalidade dos circuitos.

Do ponto de vista da complexidade dos sistemas, os seguintes níveis podem ser considerados:

- Componentes: que representam os elementos básicos de circuitos.

- Células básicas: que são circuitos de baixa complexidade.

- Blocos funcionais: que são circuitos de média complexidade.

- Sistemas: que são circuitos de alta complexidade.

1.4 Linguagens de descrição de *hardware*

- Na área de projeto de circuitos integrados, o uso de uma Linguagem de Descrição de *Hardware* (*Hardware Description Language* ou HDL) tem sido proposto, a fim de permitir uma descrição mais abstrata dos seus elementos constituintes e de possibilitar que estes sejam organizados de forma hierárquica.
- Uma HDL é uma linguagem de modelagem, utilizada para descrever tanto a estrutura quanto a operação de um *hardware* digital.
- Em linguagens de programação comuns, os processos de interpretação e de compilação podem ser modelados como a tradução de uma linguagem entendida por uma máquina virtual para uma outra linguagem asssociada a uma outra máquina virtual.
- No caso de uma HDL, o modelo é um pouco diferente. A partir da descrição apresentada pelo código elaborado, o compilador deve inferir um *hardware* digital equivalente.
- Portanto, por meio de uma HDL, além de um mapeamento lingüístico e/ou matemático, é realizado um mapemento físico.
- De acordo com o seu comportamento funcional, um *hardware* digital pode ser classificado da seguinte forma:
  - Combinacional: sistema instantâneo (ou sem memória), com operação concorrente de eventos.
  - Seqüencial: sistema dinâmico (ou com memória), com operação seqüencial de eventos.
- Logo, uma HDL deve ser capaz de descrever ambos os comportamentos: o concorrente e o seqüencial.
- As aplicações típicas para uma HDL são as seguintes:
  - Documentação de circuitos digitais.
  - Análises de circuitos digitais, tais como: simulações e checagens diversas.
  - Síntese (projeto) de circuitos digitais. Uma vez definida uma implementação alvo, o compilador infere um circuito equivalente à descrição HDL e gera um código adequado para tal implementação. Sínteses típicas são as seguintes: a geração de código para configurar dispositivos lógicos programáveis e a geração de máscaras (*layout*) para fabricação de circuitos integrados.
- Algumas características que levam uma HDL a ser largamente empregada são as seguintes:
  - Apresentar padrões bem estabelecidos e bem documentados.
  - Apresentar características encontradas em outras HDLs.
  - Possuir vasta literatura disponível.
  - A existência de várias ferramentas computacionais para a HDL em questão, tais como: editores, compiladores, simuladores, checadores diversos.
  - A existência de ferramentas computacionais de diversos tipos, tais como:
    - Domínio público ou comercial.

    - Implementada isoladamente ou incluída em ambiente de desenvolvimento integrado (*Integrated Development Environment* ou IDE).
    - Implementada em diversas plataformas computacionais.

- Exemplos de HDL
  - Independentes de tecnologia e de fabricante: VHDL, Verilog, SystemVerilog.
  - Dependentes de fabricante: AHDL (Altera HDL).

- A Tabela 1.1 apresenta uma lista de fabricantes, produtos e funções, que lidam com HDL.

Fabricante	Produto	Função
Altera	Quartus II	Síntese e simulação
Xilinx	ISE	Síntese e simulação
Menthor Graphics	Precision RTL	Síntese
	ModelSim	Simulação
Synopys/Synplicity	Design Compiler Ultra	Síntese
	Synplify Pro/Premier	
	VCS	Simulação
Cadence	NC-Sim	Simulação

Tabela 1.1: Lista de fabricantes, produtos e funções, que lidam com HDL.

Capítulo 2

Introdução à linguagem Verilog

A linguagem Verilog é apresentada a seguir, de forma introdutória. Para que se adquira um conhecimento mais aprofundado sobre a linguagem, é recomendado consultar uma literatura específica (manuais e livros especializados).

2.1 Histórico da linguagem Verilog

- O histórico descrito a seguir foi montado a partir dos seguintes documentos:
  - Aleksandar Milenković, (System)Verilog Tutorial [Tuta].
  - Deepak Kumar Tala, Verilog Tutorial [Tutb].
  - Verilog IEEE Standard 1364-1995 [Stda].
  - Verilog IEEE Standard 1364-2001 [Stdb].
  - Verilog IEEE Standard 1364-2005 [Stdc].
  - SystemVerilog IEEE Standard 1800-2009 [Stdd].
  - SystemVerilog IEEE Standard 1800-2012 [Stde].
  - SystemVerilog IEEE Standard 1800-2017 [Stdf].
- Verilog foi desenvolvida por Phil Moorby, como uma linguagem proprietária da empresa Gateway Design Automation Inc., por volta de 1984, para ser usada em simulação lógica.
- A linguagem não era padronizada e sofreu várias revisões entre 1984 e 1990.
- Ela foi influenciada pela HDL HiLo, a qual foi desenvolvida na Universidade de Brunel sob um contrato com o Ministério da Defesa Britânico, e por linguagens de computação tradicionais, possuindo uma sintaxe concisa e similar à linguagem C.
- Um simulador para Verilog foi desenvolvido em 1985, pela Gateway, e substancialmente expandido em 1987. Ele era um interpretador denominado Verilog-XL.
- Em 1989, a empresa Gateway foi comprada pela empresa Cadence Design System.
- Em 1990, para não perder mercado para a linguagem VHDL, a Cadence organizou a Open Verilog International (OVI), que realizou melhorias no *Language Reference Manual* (LRM), e, em 1991, transformou Verilog em uma linguagem aberta, de domínio público.

- Com a abertura da linguagem, em 1992 e 1993, vários simuladores foram desenvolvidos por diversas empresas. Por exemplo, o *Verilog Compiled Simulator* (VCS), da empresa Chronologic Simulation, era um compilador que acelerava em muito o tempo de simulação.

- Verilog é uma linguagem extensível, por meio de um conjunto de rotinas organizadas nos conjuntos denominados de *Programming Language Interface* (PLI) e *Verilog Procedural Interface* (VPI).

- Em 1992, a fim de transformar o LRM de Verilog em um padrão, os diretores da OVI solicitaram e, em 1993, o IEEE formou o *Working Group* 1364, que desenvolveu o padrão IEEE Std. 1364-1995, que combinou a sintaxe de Verilog e a sua *Programming Language Interface* (PLI) em um único documento.

- Os padrões IEEE são revisados, pelo menos, a cada cinco anos.

- Uma revisão do padrão inicial resultou no padrão IEEE Std. 1364-2001.

- Em 2005, o padrão foi novamente atualizado, sob a denominação IEEE Std. 1364-2005.

- Em 2001, a empresa Accellera deu início a uma tentativa de padronizar uma extensão de Verilog, denominada SystemVerilog.

- Para evitar incompatibilidades, o IEEE formou o *Working Group* 1800, ligado ao 1364, para tratarem da padronização de SystemVerilog, resultando no padrão denominado IEEE Std. 1800-2005.

- Em 2009, os padrões IEEE Std. 1364-2005 e IEEE Std. 1800-2005 foram fundidos sob a denominação IEEE Std. 1800-2009.

- Em 2012, o padrão foi atualizado, sob a denominação IEEE Std. 1800-2012.

- Em 2017, o padrão foi atualizado, sob a denominação IEEE Std. 1800-2017.

2.2 Verilog como linguagem

Como qualquer linguagem escrita, Verilog utiliza um conjunto específico de símbolos e de regras que definem aspectos de sintaxe e de semântica.

Verilog não é uma linguagem de programação de computadores. Ela é uma linguagem que tem uma finalidade distinta e específica, sendo classificada como Linguagem de Descrição de *Hardware* (*Hardware Description Language* ou HDL). Nesse sentido, Verilog foi desenvolvida para a descrição de circuitos eletrônicos digitais, sendo aplicada na documentação, simulação e síntese automática de tais circuitos.

É fundamental compreender a diferença entre uma linguagem de programação e uma HDL. Enquanto o código de uma linguagem de programação é traduzido em comandos reconhecidos e executáveis por uma máquina computacional, o código de uma HDL é traduzido em um circuito que é utilizado para construir tais máquinas.

Mesmo assim, Verilog apresenta elementos comuns a diversas linguagens de programação modernas, alguns dos quais são discutidos a seguir.

2.2.1 Considerações gerais

- Arquivos contendo código Verilog são formatados em tipo TEXTO.
- O nome do arquivo que contém o código Verilog (`nome.v`) deve ser o mesmo nome da entidade mais externa na hierarquia de circuitos descrita pelo arquivo em questão.
- Comentários podem ser incluídos de duas formas:
  em linha única (// texto) ou em várias linhas (/* texto ... texto */).
- Verilog é *case sensitive*.

  Logo: palavra $\neq$ PaLaVrA $\neq$ PALAVRA.
- Um sinal físico, com valores digitais típicos (0, 1, X, Z), pode ser definido pelo tipos NET e VARIABLE. Por sua vez, um conjunto (vetor) de tais sinais é definido pela especificação de faixa [$valor1 : valor2$].
- Exemplos de declarações para sinais físicos do tipo NET são:

  ```
  wire sinal_simples;
  ```

  e

  ```
  wire [valor1:valor2] sinal_vetor;.
  ```
- Exemplos de declarações para sinais físicos do tipo VARIABLE são:

  ```
  reg sinal_simples;
  ```

  e

  ```
  reg [valor1:valor2] sinal_vetor;.
  ```
- Duas operações básicas no uso de Verilog são a compilação e a simulação. Em uma forma bastante simplificada, elas podem ser definidas como a seguir. A partir de uma descrição Verilog do circuito digital, armazenada em arquivo do tipo texto, o compilador Verilog infere um circuito equivalente na implementação alvo e armazena tal informação em um outro arquivo do tipo texto. A partir do arquivo que contém informação sobre o circuito compilado e de uma descrição de sinais de teste, armazenada em arquivo do tipo texto, o simulador Verilog calcula os sinais gerados pelas saídas do circuito.

2.2.2 Identificadores

Um identificador é uma cadeia de caracteres (*string*), usada para designar um nome único a um objeto, de forma que ele possa ser adequadamente referenciado.

Por padrão, Verilog aceita identificadores da ordem de 1000 caracteres.

2.2.3 Palavras reservadas

As palavras reservadas (*reserved words* ou *keywords*) são identificadores que possuem um significado especial dentro da linguagem. Assim, seu uso é restrito à sua definição original e, uma vez que não podem ser redefinidas, elas não podem ser empregadas para nenhum outro propósito.

A Figura 2.1 apresenta as palavras reservadas de Verilog, as quais são definidas apenas em tipo minúsculo.

always	for	nand	scalared
and	force	negedge	showcancelled
assign	forever	nmos	signed
automatic	fork	nor	small
	function	noshowcancelled	specify
begin		not	specparam
buf	generate	notif0	strong0
bufif0	genvar	notif1	strong1
bufif1			supply0
	highz0	or	supply1
case	highz1	output	
casex			table
casez	if	parameter	task
cell	ifnone	pmos	time
cmos	incdir	posedge	tran
config	include	primitive	tranif0
	initial	pull0	tranif1
deassign	inout	pull1	tri
default	input	pulldown	tri0
defparam	instance	pullup	tri1
design	integer	pulsestyle_onevent	triand
disable		pulsestyle_ondetect	trior
	join		trireg
edge		rcmos	
else	large	real	unsigned (*)
end	liblist	realtime	use
endcase	library	reg	uwire
endconfig	localparam	release	
endfunction		repeat	vectored
endgenerate	macromodule	rnmos	
endmodule	medium	rpmos	wait
endprimitive	module	rtran	wand
endspecify		rtranif0	weak0
endtable		rtranif1	weak1
endtask			while
event			wire
			wor
			xnor
			xor

(*) unsigned é reservado para um possível uso futuro.

Figura 2.1: Palavras reservadas de Verilog.

2.2.4 Identificadores definidos pelo usuário

Algumas regras básicas para a construção de identificadores são as seguintes:

- Todo identificador é formado por uma seqüência de caracteres (*string*) única, sendo o seu comprimento máximo definido pela ferramenta utilizada.
- Os identificadores podem ser formados apenas com letras minúsculas e/ou maiúsculas (a até z e A até Z), com dígitos numéricos de 0 a 9, com o símbolo "$" (dólar) e com o símbolo "_" (sublinhado ou *underscore*).
- O primeiro caracter de um identificador não pode ser um dígito numérico.
- Quando o primeiro caracter de um identificador é o símbolo $, ele possui um significado especial, passando a definir uma *System Task* ou uma *System Function*.
- Verilog é *case sensitive*. Logo: identificador $\neq$ IdEnTiFiCaDoR $\neq$ IDENTIFICADOR.

Parte II

Circuitos Combinacionais: Construções básicas

Capítulo 3

Exemplos de construções básicas

3.1 Introdução

Nesse capítulo são apresentados exemplos de construções básicas na HDL em questão. A seguir, são abordados os seguintes itens:

- Códigos para Tabela Verdade.
- Códigos para equivalentes formas de expressão lógica.
- Códigos para equivalentes formas de descrição.

3.2 Códigos para Tabela Verdade

- A Listagem 1 apresenta um exemplo de descrição de Tabela Verdade, com a mesma quantidade de valores '0' e '1'.
- A Listagem 2 apresenta um exemplo de descrição de Tabela Verdade, com quantidades diferentes de valores '0' e '1'.
- A Listagem 3 apresenta um exemplo de descrição de Tabela Verdade, com a criação de um sinal intermediário.

Listagem 1 - Exemplo de descrição de Tabela Verdade ('0' e '1' em mesma quantidade):

```
//
// ----- ----- ----- ----- ----- -----
// Design: Truth table
//         by using case()
//         with #0's = #1's
// Filename: truth_table_00_case.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module truth_table_00_case (
```

```
a, // input 1
b, // input 2
c, // input 3
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;

  // Reg Declaration
reg f; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(a or b or c)
  begin
    //
    case ({a,b,c})
      3'b011  : f = 1;
      3'b100  : f = 1;
      3'b110  : f = 1;
      3'b111  : f = 1;
      default : f = 0;
    endcase
    //
  end // always

endmodule // truth_table_00_case

//
// EOF
//
```

Listagem 2 - Exemplo de descrição de Tabela Verdade (‘0’ e ‘1’ em quantidades diferentes):

```
//
// ----- ----- ----- ----- ----- -----
// Design: Truth table
//         by using case()
//         with #0's != #1's
// Filename: truth_table_01_case.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module truth_table_01_case (
a, // input 1
b, // input 2
c, // input 3
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;

  // Reg Declaration
reg f; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(a or b or c)
  begin
    //
    case ({a,b,c})
      3'b000  : f = 0;
      3'b110  : f = 0;
      default : f = 1;
```

```
    endcase
    //
  end // always

endmodule // truth_table_01_case

//
// EOF
//
```

Listagem 3 - Exemplo de descrição de Tabela Verdade (criação de sinal intermediário):

```
//
// ----- ----- ----- ----- ----- -----
// Design: Truth table
//         by using if()
//         with an intermediate signal
// Filename: truth_table_02_case.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module truth_table_02_if (
a, // input 1
b, // input 2
c, // input 3
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;

  // Port Reg Declaration
reg f; // reg because of "always"...

  // Internal Reg Declaration
```

```
reg x; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
// always @(a or b or c or x)   <-- Também é aceito e funciona igual...
 always @(a or b or c)
  // using blocking assignment "=' for combinational synthesis !!!
  begin
    //
    if (b==c)
      x = 0;
      else
        x = 1;
    //
    if (a==x)
      f = 0;
      else
        f = 1;
    //
  end // always

endmodule // truth_table_02_if

//
// EOF
//
```

3.3 Códigos para equivalentes formas de expressão lógica

- As listagens abaixo descrevem a mesma função.
- A Listagem 4 apresenta a descrição da Tabela Verdade, implementando cada valor "1" separadamente.
- A Listagem 5 apresenta a implementação da forma SOP padrão.
- A Listagem 6 apresenta a implementação da forma SOP mínima.
- A Listagem 7 apresenta a implementação da forma mínima global.

Listagem 4 - Exemplo de descrição da Tabela Verdade:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Truth table
//         by using case()
//         for minterms
// Filename: truth_table_minterms.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module truth_table_minterms (
a, // input 1
b, // input 2
c, // input 3
d, // input 4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
wire d;

  // Reg Declaration
```

```
reg f; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(a or b or c or d)
  begin
    //
    case ({a,b,c,d})
      4'b0101 : f = 1;
      4'b0110 : f = 1;
      4'b1001 : f = 1;
      4'b1010 : f = 1;
      4'b1101 : f = 1;
      4'b1110 : f = 1;
      default : f = 0;
    endcase
    //
  end // always

endmodule // truth_table_minterms

//
// EOF
//
```

Listagem 5 - Exemplo de descrição da forma SOP padrão:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Standard Sum Of Products
//         by using primitive gates
//         for building minterms
// Filename: std_sop.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module std_sop (
   a, // input  1
   b, // input  2
```

```
   c, // input  3
   d, // input  4
min1, // output minterm 1
min2, // output minterm 2
min3, // output minterm 3
min4, // output minterm 4
min5, // output minterm 5
min6, // output minterm 6
   f  // output
);

  // I/O Port Declaration
input     a;
input     b;
input     c;
input     d;
output min1;
output min2;
output min3;
output min4;
output min5;
output min6;
output    f;

  // Port Wire Declaration
wire    a;
wire    b;
wire    c;
wire    d;
wire min1;
wire min2;
wire min3;
wire min4;
wire min5;
wire min6;
wire    f;

  // Internal Wire Declaration
wire w_not_a;
wire w_not_b;
wire w_not_c;
wire w_not_d;
//
wire w_and_1;
wire w_and_2;
wire w_and_3;
wire w_and_4;
wire w_and_5;
wire w_and_6;
```

```
  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_a  (w_not_a, a);
not  not_b  (w_not_b, b);
not  not_c  (w_not_c, c);
not  not_d  (w_not_d, d);
//
and  and_1  (w_and_1, w_not_a,       b, w_not_c,       d);
and  and_2  (w_and_2, w_not_a,       b,       c, w_not_d);
and  and_3  (w_and_3,       a, w_not_b, w_not_c,       d);
and  and_4  (w_and_4,       a, w_not_b,       c, w_not_d);
and  and_5  (w_and_5,       a,       b, w_not_c,       d);
and  and_6  (w_and_6,       a,       b,       c, w_not_d);
//
or   or_out (f, w_and_1, w_and_2, w_and_3, w_and_4, w_and_5, w_and_6);

    // Continuous Assignment Statement
//
assign min1 = w_and_1;
assign min2 = w_and_2;
assign min3 = w_and_3;
assign min4 = w_and_4;
assign min5 = w_and_5;
assign min6 = w_and_6;

endmodule // std_sop

//
// EOF
//
```

Listagem 6 - Exemplo de descrição da forma SOP mínima:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Minimum Sum Of Products
//         by using primitive gates
//         for building implicants
// Filename: min_sop.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//
```

```
// Module Declaration
module min_sop (
   a, // input  1
   b, // input  2
   c, // input  3
   d, // input  4
imp1, // output implicant 1
imp2, // output implicant 2
imp3, // output implicant 3
imp4, // output implicant 4
   f  // output
);

  // I/O Port Declaration
input     a;
input     b;
input     c;
input     d;
output imp1;
output imp2;
output imp3;
output imp4;
output    f;

  // Port Wire Declaration
wire    a;
wire    b;
wire    c;
wire    d;
wire imp1;
wire imp2;
wire imp3;
wire imp4;
wire    f;

  // Internal Wire Declaration
wire w_not_c;
wire w_not_d;
//
wire w_and_1;
wire w_and_2;
wire w_and_3;
wire w_and_4;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
```

```
//
not  not_c  (w_not_c, c);
not  not_d  (w_not_d, d);
//
and  and_1  (w_and_1,       b, w_not_c,       d);
and  and_2  (w_and_2,       b,       c, w_not_d);
and  and_3  (w_and_3,       a, w_not_c,       d);
and  and_4  (w_and_4,       a,       c, w_not_d);
//
or   or_out (f, w_and_1, w_and_2, w_and_3, w_and_4);

    // Continuous Assignment Statement
//
assign imp1 = w_and_1;
assign imp2 = w_and_2;
assign imp3 = w_and_3;
assign imp4 = w_and_4;

endmodule // min_sop

//
// EOF
//
```

Listagem 7 - Exemplo de descrição da forma mínima global:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Global Minimum
//         by using primitive gates
//         for building signals
// Filename: min_global.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module min_global (
   a, // input  1
   b, // input  2
   c, // input  3
   d, // input  4
sig1, // output signal 1
sig2, // output signal 2
sig3, // output signal 3
sig4, // output signal 4
```

```
   f  // output
);

  // I/O Port Declaration
input     a;
input     b;
input     c;
input     d;
output sig1;
output sig2;
output sig3;
output sig4;
output    f;

  // Port Wire Declaration
wire    a;
wire    b;
wire    c;
wire    d;
wire sig1;
wire sig2;
wire sig3;
wire sig4;
wire    f;

  // Internal Wire Declaration
wire w_not_c;
wire w_not_d;
//
wire w_and_l1s1;
wire w_and_l1s2;
wire  w_or_l1s3;
wire  w_or_l2s1;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_c  (w_not_c, c);
not  not_d  (w_not_d, d);
//
and  and_l1s1  (w_and_l1s1, w_not_c,       d);
and  and_l1s2  (w_and_l1s2,       c, w_not_d);
or    or_l1s3  ( w_or_l1s3,       a,       b);
//
or    or_l2s1  ( w_or_l2s1, w_and_l1s1, w_and_l1s2);
//
and  and_out   (f, w_or_l1s3, w_or_l2s1);
```

```
    // Continuous Assignment Statement
//
assign sig1 = w_and_l1s1;
assign sig2 = w_and_l1s2;
assign sig3 = w_or_l1s3;
assign sig4 = w_or_l2s1;

endmodule // min_global

//
// EOF
//
```

3.4 Códigos para equivalentes formas de descrição

- As listagens abaixo descrevem a mesma função.
- A Listagem 8 apresenta uma descrição usando comando condicional.
- A Listagem 9 apresenta uma descrição equivalente usando operadores.
- A Listagem 10 apresenta uma descrição equivalente usando componentes instanciados primitivos.
- A Listagem 11 apresenta uma descrição equivalente usando componentes instanciados em um mesmo arquivo.
- A Listagem 12 apresenta uma descrição equivalente usando componentes instanciados em arquivos separados.

Listagem 8 - Exemplo de descrição usando comando condicional:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using case()
// Filename: funct_case.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module funct_case (
a, // input 1
b, // input 2
c, // input 3
```

```
d, // input 4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
wire d;

  // Reg Declaration
reg f; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(a or b or c or d)
  begin
    //
    case ({a,b,c,d})
      4'b1100 : f = 1;
      4'b1101 : f = 1;
      4'b1110 : f = 1;
      //
      4'b0011 : f = 1;
      4'b0111 : f = 1;
      4'b1011 : f = 1;
      //
      4'b1111 : f = 1;
      //
      default : f = 0;
    endcase
    //
  end // always

endmodule // funct_case

//
```

```
// EOF
//
```

Listagem 9 - Exemplo de descrição usando operadores:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using logical operators
// Filename: funct_op.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module funct_op (
a, // input  1
b, // input  2
c, // input  3
d, // input  4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
wire d;
wire f;

  // Internal Wire Declaration
wire w_and_ab;
wire w_and_cd;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign w_and_ab = a & b;
assign w_and_cd = c & d;
//
assign f = w_and_ab | w_and_cd;
```

```
endmodule // funct_op

//
// EOF
//
```

Listagem 10 - Exemplo de descrição usando componentes instanciados primitivos:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using primitive gates
// Filename: funct_estrut_primitive.v
// Coder: Alexandre Santos de la Vega
// Versions: /feb_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module funct_estrut_primitive (
a, // input  1
b, // input  2
c, // input  3
d, // input  4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
wire d;
wire f;

  // Internal Wire Declaration
wire w_and_ab;
wire w_and_cd;


  // Concurrent Assignment
```

```
    // Predefined Logic Primitives Instantiation
//
and  and_ab  (w_and_ab, a, b);
and  and_cd  (w_and_cd, c, d);
//
or   or_out (f, w_and_ab, w_and_cd);

endmodule // funct_estrut_primitive

//
// EOF
//
```

Listagem 11 - Exemplo de descrição usando componentes instanciados em um mesmo arquivo:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using instantiation
//         on the same file
// Filename: funct_estrut_same_file.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module funct_estrut_same_file (
a, // input  1
b, // input  2
c, // input  3
d, // input  4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
```

```
wire d;
wire f;

  // Internal Wire Declaration
wire w_and_ab;
wire w_and_cd;


  // Concurrent Assignment

    // Module Instantiation
//
and2_submodule  and_ab (a, b, w_and_ab);
and2_submodule  and_cd (c, d, w_and_cd);
//
 or2_submodule  or_out (w_and_ab, w_and_cd, f);

endmodule // funct_estrut_same_file


// --------------------------
//  Definicao dos submodulos
// --------------------------


// --------------------------
// Module Declaration
module and2_submodule (
a, // input  1
b, // input  2
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire f;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
and  and_ab (f, a, b);
```

```
endmodule // and2_submodule
// ---------------------------


// ---------------------------
// Module Declaration
module or2_submodule (
a, // input  1
b, // input  2
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire f;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
or  or_ab (f, a, b);

endmodule // or2_submodule
// ---------------------------

//
// EOF
//
```

Listagem 12 - Exemplo de descrição usando componentes instanciados em arquivos separados:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using instantiation
//         not on the same file
// Filename: funct_estrut_not_same_file.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module funct_estrut_not_same_file (
a, // input  1
b, // input  2
c, // input  3
d, // input  4
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
input  c;
input  d;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire c;
wire d;
wire f;

  // Internal Wire Declaration
wire w_and_ab;
wire w_and_cd;


  // Concurrent Assignment

    // Module Instantiation
//
and2_submodule  and_ab (a, b, w_and_ab);
and2_submodule  and_cd (c, d, w_and_cd);
//
 or2_submodule  or_out (w_and_ab, w_and_cd, f);
```

```
endmodule // funct_estrut_not_same_file

//
// EOF
//



//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using instantiation
//         not on the same file
//         AND2 submodule declaration
// Filename: and2_submodule.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module and2_submodule (
a, // input  1
b, // input  2
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire f;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
and  and_ab (f, a, b);

endmodule // and2_submodule

//
// EOF
```

```
//



//
// ----- ----- ----- ----- ----- -----
// Design: Function description
//         by using instantiation
//         not on the same file
//         OR2 submodule declaration
// Filename: or2_submodule.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module or2_submodule (
a, // input  1
b, // input  2
f  // output
);

  // I/O Port Declaration
input  a;
input  b;
output f;

  // Port Wire Declaration
wire a;
wire b;
wire f;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
or  or_ab (f, a, b);

endmodule // or2_submodule

//
// EOF
//
```

Capítulo 4

Exemplos de aplicação direta de portas lógicas

4.1 Introdução

Nesse capítulo são apresentados exemplos de circuitos digitais simples, com o emprego direto de uma porta lógica. A seguir, são abordados os seguintes itens:

- Elemento de controle.
- Elemento de paridade.
- Elemento de igualdade.

4.2 Elemento de controle

- A Listagem 1 apresenta um bloco de controle de interrupção.
- A Listagem 2 apresenta um elemento inversor configurável.

Listagem 1 - Bloco de controle de interrupção:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Interruption control.
//           Masking block.
//           Four inputs (1 NMI + 3 MI)
// Filename: four_pins_int_ctrl.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module four_pins_int_ctrl (
NMI_pin, // input  non_maskable interrrupt request
MI1_pin, // input      maskable interrrupt request 1
MI2_pin, // input      maskable interrrupt request 2
```

```
MI3_pin, // input      maskable interrrupt request 3
//
IM_bit1, // input  interrupt mask bit 1
IM_bit2, // input  interrupt mask bit 2
IM_bit3, // input  interrupt mask bit 3
//
NMI_req, // output non_maskable interrrupt request
MI1_req, // output     maskable interrrupt request 1
MI2_req, // output     maskable interrrupt request 2
MI3_req  // output     maskable interrrupt request 3
);

  // I/O Port Declaration
input NMI_pin;
input MI1_pin;
input MI2_pin;
input MI3_pin;
//
input IM_bit1;
input IM_bit2;
input IM_bit3;
//
output NMI_req;
output MI1_req;
output MI2_req;
output MI3_req;

  // Port Wire Declaration
wire NMI_pin;
wire MI1_pin;
wire MI2_pin;
wire MI3_pin;
//
wire IM_bit1;
wire IM_bit2;
wire IM_bit3;
//
wire NMI_req;
wire MI1_req;
wire MI2_req;
wire MI3_req;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign NMI_req = NMI_pin;
```

```
    // Predefined Logic Primitives Instantiation
//
and and_1 (MI1_req, MI1_pin, IM_bit1);
and and_2 (MI2_req, MI2_pin, IM_bit2);
and and_3 (MI3_req, MI3_pin, IM_bit3);

endmodule // four_pins_int_ctrl

//
// EOF
//
```

Listagem 2 - Elemento inversor configurável:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Configurable inverter.
// Filename: config_not.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module config_not (
cfg_ctrl   , // input  control
cfg_not_in , // input  not in
cfg_not_out  // output not out
);

  // I/O Port Declaration
input  cfg_ctrl   ;
input  cfg_not_in ;
output cfg_not_out;

  // Port Wire Declaration
wire cfg_ctrl   ;
wire cfg_not_in ;
wire cfg_not_out;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xor xor_config_not (cfg_not_out, cfg_not_in, cfg_ctrl);
```

```
endmodule // config_not

//
// EOF
//
```

4.3 Elemento de paridade

- A Listagem 3 apresenta a implementação de um gerador de paridade par e ímpar.
- A Listagem 4 apresenta a implementação de um identificador de paridade par e ímpar.

Listagem 3 - Exemplo de descrição de gerador de paridade par e ímpar:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Parity generator.
//           Two input bits.
//           Odd and even parity outputs.
// Filename: parity_gen.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module parity_gen (
data_bit1      , // input bit 1
data_bit2      , // input bit 2
 odd_parity_bit, // output  odd_parity bit
even_parity_bit  // output even_parity bit
);

  // I/O Port Declaration
input  data_bit1      ;
input  data_bit2      ;
output  odd_parity_bit;
output even_parity_bit;

  // Port Wire Declaration
wire data_bit1      ;
wire data_bit2      ;
wire  odd_parity_bit;
wire even_parity_bit;
```

```
  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xnor xnor_odd ( odd_parity_bit, data_bit1, data_bit2);
//
xor  xor_odd  (even_parity_bit, data_bit1, data_bit2);

endmodule // parity_gen

//
// EOF
//
```

Listagem 4 - Exemplo de descrição de identificador de paridade par e ímpar:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Parity identifier.
//           Two input bits.
//           Odd and even parity outputs.
// Filename: parity_idf.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module parity_idf (
  data_bit     , // input   data bit
parity_bit     , // input parity bit
 odd_parity_bit, // output  odd_parity bit
even_parity_bit  // output even_parity bit
);

  // I/O Port Declaration
input    data_bit     ;
input  parity_bit     ;
output  odd_parity_bit;
output even_parity_bit;

  // Port Wire Declaration
wire   data_bit     ;
wire parity_bit     ;
wire  odd_parity_bit;
wire even_parity_bit;
```

```
  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xor   xor_odd  ( odd_parity_bit, data_bit, parity_bit);
//
xnor  xnor_odd (even_parity_bit, data_bit, parity_bit);

endmodule // parity_idf

//
// EOF
//
```

4.4 Elemento de igualdade

- A Listagem 5 apresenta a implementação de um identificador de igualdade entre padrões binários.

Listagem 5 - Exemplo de descrição de identificador de igualdade:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Equality identifier.
//           Two input bits.
//           Equal and not equal outputs.
// Filename: one_bit_equal_idf.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module one_bit_equal_idf (
          data1, // input data 1
          data2, // input data 2
    equal_data , // output     equal
not_equal_data   // output not equal
);

  // I/O Port Declaration
input            data1;
input            data2;
output     equal_data ;
output not_equal_data ;
```

```
  // Port Wire Declaration
wire           data1;
wire           data2;
wire     equal_data ;
wire not_equal_data ;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xnor xnor_odd (    equal_data, data1, data2);
//
xor  xor_odd  (not_equal_data, data1, data2);

endmodule // one_bit_equal_idf

//
// EOF
//
```

Parte III

Circuitos Combinacionais: Blocos funcionais

Capítulo 5

Exemplos de blocos funcionais

Nessa parte, são apresentados exemplos de circuitos digitais que implementam funções comumente encontradas em sistemas digitais.

A seguir, são abordados os seguintes itens:

- Detector de igualdade entre dois grupos de *bits*.
- Selecionadores:
  - Multiplexador (MUX).
  - Demultiplexador (DEMUX).
  - Decodificador de endereço (*address decoder*) ou decodificador de linha (*line decoder*).
- Deslocador configurável (*barrel shifter*).
- Decodificador.
- Conversor de códigos.
- Separador e relacionador de *bits* 1 e 0 em palavra de N *bits*.
- Somadores básicos.
- Detector de *overflow* e saturador.

Capítulo 6

Detector de igualdade entre dois grupos de *bits*

6.1 Detector de igualdade para um *bit*

- A Listagem 1 apresenta um detector de igualdade para um *bit*, baseado em Tabela Verdade.
- A Listagem 2 apresenta um detector de igualdade para um *bit*, baseado na comparação direta.
- A Listagem 3 apresenta um detector de igualdade para um *bit*, baseado em operador lógico.

Listagem 1 - Detector de igualdade para um *bit*, baseado em Tabela Verdade:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on truth table
//         by using case()
//         1-bit pattern
// Filename: tt_1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module tt_1bit_x_equal_y (
  nbr1_k, // input pattern 1
  nbr2_k, // input pattern 2
eq_inp_k, // equal signal input
eq_out_k  // equal signal output
);

  // I/O Port Declaration
input    nbr1_k;
```

```
input    nbr2_k;
input  eq_inp_k;
output eq_out_k;

  // Port Wire Declaration
wire   nbr1_k;
wire   nbr2_k;
wire eq_inp_k;

  // Reg Declaration
reg eq_out_k; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(eq_inp_k or nbr1_k or nbr2_k)
  begin
    //
    case ({eq_inp_k, nbr1_k, nbr2_k})
      3'b100  : eq_out_k = 1;
      3'b111  : eq_out_k = 1;
      default : eq_out_k = 0;
    endcase
    //
  end // always

endmodule // tt_1bit_x_equal_y

//
// EOF
//
```

Listagem 2 - Detector de igualdade para um *bit*, baseado na comparação direta:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on strict comparison
//         by using if()
//         1-bit pattern
// Filename: eq_1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
```

```
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module eq_1bit_x_equal_y (
  nbr1_k, // input pattern 1
  nbr2_k, // input pattern 2
eq_inp_k, // equal signal input
eq_out_k  // equal signal output
);

  // I/O Port Declaration
input    nbr1_k;
input    nbr2_k;
input  eq_inp_k;
output eq_out_k;

  // Port Wire Declaration
wire   nbr1_k;
wire   nbr2_k;
wire eq_inp_k;

  // Reg Declaration
reg eq_out_k; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(eq_inp_k or nbr1_k or nbr2_k)
  begin
    //
    if ( (eq_inp_k == 1) & (nbr1_k == nbr2_k) )
      eq_out_k = 1;
    else
      eq_out_k = 0;
    //
  end // always

endmodule // eq_1bit_x_equal_y

//
// EOF
//
```

Listagem 3 - Detector de igualdade para um *bit*, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on logic operators
//         1-bit pattern
// Filename: op_1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_1bit_x_equal_y (
  nbr1_k, // input pattern 1
  nbr2_k, // input pattern 2
eq_inp_k, // equal signal input
eq_out_k  // equal signal output
);

  // I/O Port Declaration
input    nbr1_k;
input    nbr2_k;
input  eq_inp_k;
output eq_out_k;

  // Port Wire Declaration
wire   nbr1_k;
wire   nbr2_k;
wire eq_inp_k;
wire eq_out_k;

  // Internal Wire Declaration
wire w_eq;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xnor  xnor_patt (    w_eq, nbr1_k,   nbr2_k);
and   and_inpk  (eq_out_k,   w_eq, eq_inp_k);

endmodule // op_1bit_x_equal_y

//
// EOF
//
```

6.2 Detector de igualdade para quatro *bits*

- A Listagem 4 apresenta um detector de igualdade para quatro *bits*, baseado em operador lógico.
- A Listagem 5 apresenta um detector de igualdade para dezesseis *bits*, empregando o detector de quatro *bits* da Listagem 4.

Listagem 4 - Detector de igualdade para quatro *bits*, baseado baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on logic operators
//         4-bit pattern
// Filename: op_4bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_4bit_x_equal_y (
  nbr1, // input pattern 1
  nbr2, // input pattern 2
eq_inp, // equal signal input
eq_out  // equal signal output
);

  // I/O Port Declaration
input  [3:0]   nbr1;
input  [3:0]   nbr2;
input        eq_inp;
output       eq_out;

  // Port Wire Declaration
wire [3:0]   nbr1;
wire [3:0]   nbr2;
wire       eq_inp;
wire       eq_out;

  // Internal Wire Declaration
wire [3:0] w_eq;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
// It can be like that:
```

```
xnor  xnor_bit3 (w_eq[3], nbr1[3], nbr2[3]);
xnor  xnor_bit2 (w_eq[2], nbr1[2], nbr2[2]);
//        or like that:
xnor  xnor_bit1 (w_eq[1], nbr1[1], nbr2[1]),
      xnor_bit0 (w_eq[0], nbr1[0], nbr2[0]);
//
and   and_all_eq  (eq_out, w_eq[3], w_eq[2], w_eq[1], w_eq[0], eq_inp);

endmodule // op_4bit_x_equal_y

//
// EOF
//
```

Listagem 5 - Detector de igualdade para dezesseis *bits*, empregando o detector de quatro *bits*:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on instantiation
//         of modules that are
//         based on logic operators
//         16-bit pattern
// Filename: op_4x4bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_4x4bit_x_equal_y (
  nbr1, // input pattern 1
  nbr2, // input pattern 2
eq_inp, // equal signal input
eq_out  // equal signal output
);

  // Parameter Declaration (optional)
parameter pattern_width = 16;  // number of bits
parameter nom = 4;             // number of modules

  // I/O Port Declaration
input  [(pattern_width-1):0] nbr1;
input  [(pattern_width-1):0] nbr2;
input  eq_inp;
output eq_out;
```

```
  // Port Wire Declaration
wire [(pattern_width-1):0] nbr1;
wire [(pattern_width-1):0] nbr2;
wire  eq_inp;
wire  eq_out;

  // Internal Wire Declaration
wire [(nom-1):0] w_eq;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
// It can be like that:
op_4bit_x_equal_y  mod_3 (.nbr1(nbr1[15:12]), .nbr2(nbr2[15:12]),
                          .eq_inp(eq_inp)   , .eq_out(w_eq[3])    );
//
op_4bit_x_equal_y  mod_2 (.nbr1(nbr1[11:8]) , .nbr2(nbr2[11:8] ),
                          .eq_inp(w_eq[3])  , .eq_out(w_eq[2])    );
//        or like that:
op_4bit_x_equal_y  mod_1 (.nbr1(nbr1[7:4])  , .nbr2(nbr2[7:4])  ,
                          .eq_inp(w_eq[2])  , .eq_out(w_eq[1])    ),
//
                   mod_0 (.nbr1(nbr1[3:0])  , .nbr2(nbr2[3:0])  ,
                          .eq_inp(w_eq[1])  , .eq_out(w_eq[0])    );

    // Continuous Assignment Statement
//
assign eq_out = w_eq[0];

endmodule // op_4x4bit_x_equal_y

//
// EOF
//
```

6.3 Detectores de igualdade baseados em detector para um *bit*

- A Listagem 6 apresenta um detector de igualdade para quatro *bits*, empregando o detector de um *bit* da Listagem 3.
- A Listagem 7 apresenta um detector de igualdade para um *bit*, baseado na comparação direta, parametrizável.
- A Listagem 8 apresenta um detector de igualdade para oito *bits*, empregando o detector de um *bit* parametrizável da Listagem 7.

Listagem 6 - Detector de igualdade para quatro *bits*, empregando o detector de um *bit*:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on instantiation
//         of modules that are
//         based on logic operators
//         4-bit pattern
// Filename: op_4x1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_4x1bit_x_equal_y (
  nbr1, // input pattern 1
  nbr2, // input pattern 2
eq_inp, // equal signal input
eq_out  // equal signal output
);

  // Parameter Declaration (optional)
parameter pattern_width = 4;  // number of bits
parameter nom = 4;            // number of modules

  // I/O Port Declaration
input  [(pattern_width-1):0] nbr1;
input  [(pattern_width-1):0] nbr2;
input  eq_inp;
output eq_out;

  // Port Wire Declaration
wire [(pattern_width-1):0] nbr1;
wire [(pattern_width-1):0] nbr2;
wire  eq_inp;
wire  eq_out;
```

```
  // Internal Wire Declaration
wire [(nom-1):0] w_eq;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
// It can be like that:
op_1bit_x_equal_y  mod_3 (.nbr1_k(nbr1[3])  , .nbr2_k(nbr2[3])  ,
                          .eq_inp_k(eq_inp) , .eq_out_k(w_eq[3])  );
//
op_1bit_x_equal_y  mod_2 (.nbr1_k(nbr1[2])  , .nbr2_k(nbr2[2])  ,
                          .eq_inp_k(w_eq[3]), .eq_out_k(w_eq[2])  );
//        or like that:
op_1bit_x_equal_y  mod_1 (.nbr1_k(nbr1[1])  , .nbr2_k(nbr2[1])  ,
                          .eq_inp_k(w_eq[2]), .eq_out_k(w_eq[1])  ),
//
                  mod_0 (.nbr1_k(nbr1[0])  , .nbr2_k(nbr2[0])  ,
                          .eq_inp_k(w_eq[1]), .eq_out_k(w_eq[0])  );

    // Continuous Assignment Statement
//
assign eq_out = w_eq[0];

endmodule // op_4x1bit_x_equal_y

//
// EOF
//
```

Listagem 7 - Detector de igualdade para um *bit*, parametrizável:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on strict comparison
//         by using if().
//         Org parameter: 1-bit pattern.
// Filename: eq_P1_1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module eq_P1_1bit_x_equal_y (
  nbr1, // input pattern 1
  nbr2, // input pattern 2
eq_inp, // equal signal input
eq_out  // equal signal output
);

  // Parameter Declaration
parameter WIDTH1 = 1;

  // I/O Port Declaration
input  [(WIDTH1-1):0]   nbr1;
input  [(WIDTH1-1):0]   nbr2;
input                  eq_inp;
output                 eq_out;

  // Port Wire Declaration
wire [(WIDTH1-1):0]   nbr1;
wire [(WIDTH1-1):0]   nbr2;
wire                 eq_inp;

  // Reg Declaration
reg eq_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(eq_inp or nbr1 or nbr2)
  begin
    //
    if ( (eq_inp == 1) & (nbr1 == nbr2) )
      eq_out = 1;
    else
      eq_out = 0;
    //
  end // always

endmodule // eq_P1_1bit_x_equal_y

//
// EOF
//
```

Listagem 8 - Detector de igualdade para oito *bits*, empregando o detector parametrizável:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: Equal pattern detector
//         based on strict comparison
//         by using instantiation of
//         1-bit-pattern module.
//         This one: 8-bit pattern.
// Filename: eq_P8_1bit_x_equal_y.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module eq_P8_1bit_x_equal_y (
  nbr1, // input pattern 1
  nbr2, // input pattern 2
eq_inp, // equal signal input
eq_out  // equal signal output
);

  // Parameter Declaration
parameter WIDTH8 = 8;

  // I/O Port Declaration
input  [(WIDTH8-1):0]   nbr1;
input  [(WIDTH8-1):0]   nbr2;
input                 eq_inp;
output                eq_out;

  // Port Wire Declaration
wire [(WIDTH8-1):0]   nbr1;
wire [(WIDTH8-1):0]   nbr2;
wire                eq_inp;
wire                eq_out;


  // Concurrent Assignment

    // Module Instantiation
//
// Usando Padrão:
//
//   submodule_name  #(new_param_value)  instance_name (instance_port);
//
// MaxPlusII: Não funciona !!!
// Recomenda: defparam ...
```

```
    //

    //
    // Usando defparam:
    //
    eq_P1_1bit_x_equal_y  eq_detect_1_to_8 (.nbr1(nbr1)    , .nbr2(nbr2)    ,
                                            .eq_inp(eq_inp), .eq_out(eq_out)  );
    defparam eq_detect_1_to_8.WIDTH1 = WIDTH8;
    //

    endmodule // eq_P1_1bit_x_equal_y

    //
    // EOF
    //
```

Capítulo 7

Selecionadores: MUX, DEMUX e *address decoder*

7.1 Introdução

Nesse capítulo, são apresentados códigos para três blocos funcionais relacionados entre si:

- Multiplexador (MUX).
- Demultiplexador (DEMUX).
- Decodificador de endereço (*address decoder*) ou decodificador de linha (*line decoder*).

Todos eles representam um tipo de selecionador.

7.2 Multiplexador (MUX)

A seguir, são apresentados códigos para multiplexadores com N entradas de B *bits*, denominados de MUX NxB.

7.2.1 MUX 2x1

- A Listagem 1 apresenta um MUX 2x1, baseado em operador lógico.

Listagem 1 - MUX 2x1, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: MULTIPLEXER
//         based on logic operators
//         (MUX 2x1)
// Filename: op_mux_2x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module op_mux_2x1 (
mux_in0, // input  data 0
mux_in1, // input  data 1
   sel0, // input  selector 0
mux_out  // output selected data
);

  // I/O Port Declaration
input  mux_in0;
input  mux_in1;
input     sel0;
output mux_out;

  // Port Wire Declaration
wire mux_in0;
wire mux_in1;
wire    sel0;
wire mux_out;

  // Internal Wire Declaration
wire w_not_sel0;
//
wire     w_min0;
wire     w_min1;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
//
and  and_min0 (w_min0, w_not_sel0, mux_in0);
and  and_min1 (w_min1,       sel0, mux_in1);
//
or   or_out   (mux_out, w_min0, w_min1);
//

endmodule // op_mux_2x1

//
// EOF
//
```

7.2.2 MUX 4x1

- A Listagem 2 apresenta um MUX 4x1, empregando o MUX 2x1 da Listagem 1.
- A Listagem 3 apresenta um MUX 4x1, baseado em operador lógico.

Listagem 2 - MUX 4x1, empregando o MUX 2x1:

```
//
// ----- ----- ----- ----- ----- -----
// Design: MULTIPLEXER
//         based on hierarchy
//         (MUX 4x1 by using MUX 2x1)
// Filename: hier_mux_4x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_mux_4x1 (
mux_in0, // input  data 0
mux_in1, // input  data 1
mux_in2, // input  data 2
mux_in3, // input  data 3
   sel0, // input  selector 0
   sel1, // input  selector 1
mux_out  // output selected data
);

  // I/O Port Declaration
input  mux_in0;
input  mux_in1;
input  mux_in2;
input  mux_in3;
input     sel0;
input     sel1;
output mux_out;

  // Port Wire Declaration
wire mux_in0;
wire mux_in1;
wire mux_in2;
wire mux_in3;
wire    sel0;
wire    sel1;
wire mux_out;

  // Internal Wire Declaration
wire w_out_mux_01;
```

```
wire w_out_mux_23;

  // Concurrent Assignment

    // Module Instantiation
//
op_mux_2x1  mux_23 (     mux_in2,      mux_in3, sel0, w_out_mux_23);
//
op_mux_2x1  mux_01 (     mux_in0,      mux_in1, sel0, w_out_mux_01);
//
op_mux_2x1  mux_LH (w_out_mux_01, w_out_mux_23, sel1,      mux_out);
//

endmodule // hier_mux_4x1

//
// EOF
//
```

Listagem 3 - MUX 4x1, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: MULTIPLEXER
//         based on logic operators
//         (MUX 4x1)
// Filename: op_mux_4x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_mux_4x1 (
mux_in0, // input  data 0
mux_in1, // input  data 1
mux_in2, // input  data 2
mux_in3, // input  data 3
   sel0, // input  selector 0
   sel1, // input  selector 1
mux_out  // output selected data
);

  // I/O Port Declaration
input  mux_in0;
input  mux_in1;
input  mux_in2;
```

```
input  mux_in3;
input     sel0;
input     sel1;
output mux_out;

  // Port Wire Declaration
wire mux_in0;
wire mux_in1;
wire mux_in2;
wire mux_in3;
wire    sel0;
wire    sel1;
wire mux_out;

  // Internal Wire Declaration
wire w_not_sel0;
wire w_not_sel1;
//
wire     w_min0;
wire     w_min1;
wire     w_min2;
wire     w_min3;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
not  not_sel1 (w_not_sel1, sel1);
//
and  and_min0 (w_min0, w_not_sel1, w_not_sel0, mux_in0);
and  and_min1 (w_min1, w_not_sel1,       sel0, mux_in1);
and  and_min2 (w_min2,       sel1, w_not_sel0, mux_in2);
and  and_min3 (w_min3,       sel1,       sel0, mux_in3);
//
or   or_out   (mux_out, w_min0, w_min1, w_min2, w_min3);
//

endmodule // op_mux_4x1

//
// EOF
//
```

7.2.3 MUX 8x1

- A Listagem 4 apresenta um MUX 8x1, empregando comando condicional.

Listagem 4 - MUX 8x1, empregando comando condicional:

```
//
// ----- ----- ----- ----- ----- -----
// Design: MULTIPLEXER
//         by using case()
//         (MUX 8x1)
// Filename: cond_case_mux_8x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module cond_case_mux_8x1 (
 mux_in, // input  data
    sel, // input  selector
mux_out  // output selected data
);

  // Parameter Declaration
parameter WIDTH_sel3 = 3;
parameter WIDTH_inp8 = 8;

  // I/O Port Declaration
input  [(WIDTH_inp8-1):0]  mux_in;
input  [(WIDTH_sel3-1):0]     sel;
output                    mux_out;

  // Port Wire Declaration
wire  [(WIDTH_inp8-1):0] mux_in;
wire  [(WIDTH_sel3-1):0]    sel;

  // Reg Declaration
reg mux_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(mux_in or sel)
  begin
    //
```

```
    case (sel)
      3'b000 : mux_out = mux_in[0];
      3'b001 : mux_out = mux_in[1];
      3'b010 : mux_out = mux_in[2];
      3'b011 : mux_out = mux_in[3];
      3'b100 : mux_out = mux_in[4];
      3'b101 : mux_out = mux_in[5];
      3'b110 : mux_out = mux_in[6];
      3'b111 : mux_out = mux_in[7];
      // no default action...
      // default :  ;
    endcase
    //
  end // always

endmodule // cond_case_mux_8x1

//
// EOF
//
```

7.3 Demultiplexador (DEMUX)

A seguir, são apresentados códigos para demultiplexadores com N saídas de B *bits*, denominados de DEMUX NxB.

7.3.1 DEMUX 2x1

- A Listagem 5 apresenta um DEMUX 2x1, baseado em operador lógico.

Listagem 5 - DEMUX 2x1, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: DEMULTIPLEXER
//         based on logic operators
//         (DEMUX 2x1)
// Filename: op_demux_2x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_demux_2x1 (
```

```
  demux_in, // input  data
      sel0, // input  selector 0
demux_out0, // output selected data 0
demux_out1  // output selected data 1
);

  // I/O Port Declaration
input    demux_in;
input        sel0;
output demux_out0;
output demux_out1;

  // Port Wire Declaration
wire   demux_in;
wire       sel0;
wire demux_out0;
wire demux_out1;

  // Internal Wire Declaration
wire w_not_sel0;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
//
and  and_out0 (demux_out0, w_not_sel0, demux_in);
and  and_out1 (demux_out1,       sel0, demux_in);
//

endmodule // op_demux_2x1

//
// EOF
//
```

7.3.2 DEMUX 4x1

- A Listagem 6 apresenta um DEMUX 4x1, empregando o DEMUX 2x1 da Listagem 5.
- A Listagem 7 apresenta um DEMUX 4x1, baseado em operador lógico.

Listagem 6 - DEMUX 4x1, empregando o DEMUX 2x1:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: DEMULTIPLEXER
//         based on hierarchy
//         (DEMUX 4x1 by using DEMUX 2x1)
// Filename: hier_demux_4x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_demux_4x1 (
  demux_in, // input  data
      sel0, // input  selector 0
      sel1, // input  selector 1
demux_out0, // output selected data 0
demux_out1, // output selected data 1
demux_out2, // output selected data 2
demux_out3  // output selected data 3
);

  // I/O Port Declaration
input    demux_in;
input        sel0;
input        sel1;
output demux_out0;
output demux_out1;
output demux_out2;
output demux_out3;

  // Port Wire Declaration
wire   demux_in;
wire       sel0;
wire       sel1;
wire demux_out0;
wire demux_out1;
wire demux_out2;
wire demux_out3;

  // Internal Wire Declaration
wire w_out_demux_LH0;
```

```
wire w_out_demux_LH1;


  // Concurrent Assignment

    // Module Instantiation
//
op_demux_2x1  demux_LH (       demux_in, sel1,
                        w_out_demux_LH0, w_out_demux_LH1);
//
op_demux_2x1  demux_01 (w_out_demux_LH0, sel0,
                             demux_out0, demux_out1);
//
op_demux_2x1  demux_23 (w_out_demux_LH1, sel0,
                             demux_out2, demux_out3);
//

endmodule // hier_demux_4x1

//
// EOF
//
```

Listagem 7 - DEMUX 4x1, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: DEMULTIPLEXER
//         based on logic operators
//         (DEMUX 4x1)
// Filename: op_demux_4x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_demux_4x1 (
  demux_in, // input  data
      sel0, // input  selector 0
      sel1, // input  selector 1
demux_out0, // output selected data 0
demux_out1, // output selected data 1
demux_out2, // output selected data 2
demux_out3  // output selected data 3
);
```

```
  // I/O Port Declaration
input    demux_in;
input        sel0;
input        sel1;
output demux_out0;
output demux_out1;
output demux_out2;
output demux_out3;

  // Port Wire Declaration
wire   demux_in;
wire       sel0;
wire       sel1;
wire demux_out0;
wire demux_out1;
wire demux_out2;
wire demux_out3;

  // Internal Wire Declaration
wire w_not_sel0;
wire w_not_sel1;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
not  not_sel1 (w_not_sel1, sel1);
//
and  and_out0 (demux_out0, w_not_sel1, w_not_sel0, demux_in);
and  and_out1 (demux_out1, w_not_sel1,       sel0, demux_in);
and  and_out2 (demux_out2,       sel1, w_not_sel0, demux_in);
and  and_out3 (demux_out3,       sel1,       sel0, demux_in);
//

endmodule // op_demux_4x1

//
// EOF
//
```

7.3.3 DEMUX 8x1

- A Listagem 8 apresenta um DEMUX 8x1, empregando comando condicional.

Listagem 8 - DEMUX 8x1, empregando comando condicional:

```
//
// ----- ----- ----- ----- ----- -----
// Design: DEMULTIPLEXER
//         by using case()
//         (DEMUX 8x1)
// Filename: cond_case_mux_8x1.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module cond_case_demux_8x1 (
 demux_in, // input  data
      sel, // input  selector
demux_out, // output selected data
);

  // Parameter Declaration
parameter WIDTH_sel3 = 3;
parameter WIDTH_out8 = 8;

  // I/O Port Declaration
input                       demux_in;
input  [(WIDTH_sel3-1):0]       sel;
output [(WIDTH_out8-1):0] demux_out;

  // Port Wire Declaration
wire                      demux_in;
wire  [(WIDTH_sel3-1):0]      sel;

  // Reg Declaration
reg [(WIDTH_out8-1):0] demux_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(demux_in or sel)
  begin
    //
```

```
    case (sel)
      3'b000 : demux_out = {7'b0, demux_in      };
      3'b001 : demux_out = {6'b0, demux_in, 1'b0};
      3'b010 : demux_out = {5'b0, demux_in, 2'b0};
      3'b011 : demux_out = {4'b0, demux_in, 3'b0};
      3'b100 : demux_out = {3'b0, demux_in, 4'b0};
      3'b101 : demux_out = {2'b0, demux_in, 5'b0};
      3'b110 : demux_out = {1'b0, demux_in, 6'b0};
      3'b111 : demux_out = {      demux_in, 7'b0};
      // no default action...
      // default :  ;
    endcase
    //
  end // always

endmodule // cond_case_demux_8x1

//
// EOF
//
```

7.4 Decodificador de endereços (*address decoder*)

A seguir, são apresentados códigos para decodificadores de endereços com B *bits* e $N = 2^B$ saídas de 1 *bit*, denominados de *address decoder* BxN ou *line decoder* BxN.

7.4.1 *Address decoder* 1x2

- A Listagem 9 apresenta um *address decoder* 1x2, baseado em operador lógico.
- A Listagem 10 apresenta um *address decoder* 1x2, com *enable*, baseado em operador lógico.

Listagem 9 - *Address decoder* 1x2, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         based on logic operators.
//         1-bit address / 2 outputs.
// Filename: op_address_decoder_1x2.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module op_address_decoder_1x2 (
                sel0, // input  selector 0
address_decoder_out0, // output selected address 0
address_decoder_out1  // output selected address 1
);

  // I/O Port Declaration
input                   sel0;
output address_decoder_out0;
output address_decoder_out1;

  // Port Wire Declaration
wire                 sel0;
wire address_decoder_out0;
wire address_decoder_out1;

  // Internal Wire Declaration
wire w_not_sel0;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);

    // Continuous Assignment Statement
//
assign address_decoder_out0 = w_not_sel0;
assign address_decoder_out1 =       sel0;


endmodule // op_address_decoder_1x2

//
// EOF
//
```

Listagem 10 - *Address decoder* 1x2, com *enable*, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         based on logic operators.
//         1-bit address / 2 outputs.
//         Enable control.
// Filename: op_address_decoder_1x2_ena.v
```

```
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_address_decoder_1x2_ena (
                 ena, // input  enable control
                sel0, // input  selector 0
address_decoder_out0, // output selected address 0
address_decoder_out1  // output selected address 1
);

  // I/O Port Declaration
input                   ena;
input                  sel0;
output address_decoder_out0;
output address_decoder_out1;

  // Port Wire Declaration
wire                  ena;
wire                 sel0;
wire address_decoder_out0;
wire address_decoder_out1;

  // Internal Wire Declaration
wire w_not_sel0;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
//
and  and_out0 (address_decoder_out0, w_not_sel0, ena);
and  and_out1 (address_decoder_out1,       sel0, ena);
//

endmodule // op_address_decoder_1x2_ena

//
// EOF
//
```

7.4.2 *Address decoder* 2x4

- A Listagem 11 apresenta um *address decoder* 2x4, baseado em operador lógico.
- A Listagem 12 apresenta um *address decoder* 2x4, com *enable*, baseado em operador lógico.

Listagem 11 - *Address decoder* 2x4, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         based on logic operators.
//         2-bit address / 4 outputs.
// Filename: op_address_decoder_2x4.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_address_decoder_2x4 (
                sel0, // input  selector 0
                sel1, // input  selector 1
address_decoder_out0, // output selected address 0
address_decoder_out1, // output selected address 1
address_decoder_out2, // output selected address 2
address_decoder_out3  // output selected address 3
);

  // I/O Port Declaration
input                  sel0;
input                  sel1;
output address_decoder_out0;
output address_decoder_out1;
output address_decoder_out2;
output address_decoder_out3;

  // Port Wire Declaration
wire                 sel0;
wire                 sel1;
wire address_decoder_out0;
wire address_decoder_out1;
wire address_decoder_out2;
wire address_decoder_out3;

  // Internal Wire Declaration
wire w_not_sel0;
wire w_not_sel1;
```

```
  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
not  not_sel1 (w_not_sel1, sel1);
//
and  and_out0 (address_decoder_out0, w_not_sel1, w_not_sel0);
and  and_out1 (address_decoder_out1, w_not_sel1,       sel0);
and  and_out2 (address_decoder_out2,       sel1, w_not_sel0);
and  and_out3 (address_decoder_out3,       sel1,       sel0);
//

endmodule // op_address_decoder_2x4

//
// EOF
//
```

Listagem 12 - *Address decoder* 2x4, com *enable*, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         based on logic operators.
//         2-bit address / 4 outputs.
//         Enable control.
// Filename: op_address_decoder_2x4_ena.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_address_decoder_2x4_ena (
                 ena, // input  enable control
                sel0, // input  selector 0
                sel1, // input  selector 1
address_decoder_out0, // output selected address 0
address_decoder_out1, // output selected address 1
address_decoder_out2, // output selected address 2
address_decoder_out3  // output selected address 3
);

  // I/O Port Declaration
input                   ena;
```

```
input                  sel0;
input                  sel1;
output address_decoder_out0;
output address_decoder_out1;
output address_decoder_out2;
output address_decoder_out3;

  // Port Wire Declaration
wire                  ena;
wire                 sel0;
wire                 sel1;
wire address_decoder_out0;
wire address_decoder_out1;
wire address_decoder_out2;
wire address_decoder_out3;

  // Internal Wire Declaration
wire w_not_sel0;
wire w_not_sel1;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
not  not_sel0 (w_not_sel0, sel0);
not  not_sel1 (w_not_sel1, sel1);
//
and  and_out0 (address_decoder_out0, w_not_sel1, w_not_sel0, ena);
and  and_out1 (address_decoder_out1, w_not_sel1,       sel0, ena);
and  and_out2 (address_decoder_out2,       sel1, w_not_sel0, ena);
and  and_out3 (address_decoder_out3,       sel1,       sel0, ena);
//

endmodule // op_address_decoder_2x4_ena

//
// EOF
//
```

7.4.3 *Address decoder* 3x8

- A Listagem 13 apresenta um *address decoder* 3x8, empregando comando condicional.
- A Listagem 14 apresenta um *address decoder* 3x8, com *enable*, empregando comando condicional.

Listagem 13 - *Address decoder* 3x8, empregando comando condicional:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         by using case().
//         3-bit address / 8 outputs.
// Filename: cond_case_address_decoder_3x8.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module cond_case_address_decoder_3x8 (
                sel, // input  selector
address_decoder_out, // output selected data
);

  // Parameter Declaration
parameter WIDTH_sel3 = 3;
parameter WIDTH_out8 = 8;

  // I/O Port Declaration
input  [(WIDTH_sel3-1):0]                sel;
output [(WIDTH_out8-1):0] address_decoder_out;

  // Port Wire Declaration
wire  [(WIDTH_sel3-1):0]      sel;

  // Reg Declaration
reg [(WIDTH_out8-1):0] address_decoder_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(sel)
  begin
    //
```

```
    case (sel)
      3'b000 : address_decoder_out = {7'b0, 1'b1      };
      3'b001 : address_decoder_out = {6'b0, 1'b1, 1'b0};
      3'b010 : address_decoder_out = {5'b0, 1'b1, 2'b0};
      3'b011 : address_decoder_out = {4'b0, 1'b1, 3'b0};
      3'b100 : address_decoder_out = {3'b0, 1'b1, 4'b0};
      3'b101 : address_decoder_out = {2'b0, 1'b1, 5'b0};
      3'b110 : address_decoder_out = {1'b0, 1'b1, 6'b0};
      3'b111 : address_decoder_out = {      1'b1, 7'b0};
      // no default action...
      // default :  ;
    endcase
    //
  end // always

endmodule // cond_case_address_decoder_3x8

//
// EOF
//
```

Listagem 14 - *Address decoder* 3x8, com *enable*, empregando comando condicional:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design: Address decoder
//         by using case().
//         3-bit address / 8 outputs.
//         Enable control.
// Filename: cond_case_address_decoder_3x8_ena.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module cond_case_address_decoder_3x8_ena (
                ena, // input  enable control
                sel, // input  selector
address_decoder_out, // output selected data
);

  // Parameter Declaration
parameter WIDTH_sel3 = 3;
parameter WIDTH_out8 = 8;

  // I/O Port Declaration
```

```
input                                       ena;
input  [(WIDTH_sel3-1):0]                   sel;
output [(WIDTH_out8-1):0] address_decoder_out;

  // Port Wire Declaration
wire                          ena;
wire  [(WIDTH_sel3-1):0]      sel;

  // Reg Declaration
reg [(WIDTH_out8-1):0] address_decoder_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
 always @(sel or ena)
  begin
    //
    if (ena == 0)
      address_decoder_out = 8'b0;
    else
      begin
        case (sel)
          3'b000 : address_decoder_out = {7'b0, 1'b1      };
          3'b001 : address_decoder_out = {6'b0, 1'b1, 1'b0};
          3'b010 : address_decoder_out = {5'b0, 1'b1, 2'b0};
          3'b011 : address_decoder_out = {4'b0, 1'b1, 3'b0};
          3'b100 : address_decoder_out = {3'b0, 1'b1, 4'b0};
          3'b101 : address_decoder_out = {2'b0, 1'b1, 5'b0};
          3'b110 : address_decoder_out = {1'b0, 1'b1, 6'b0};
          3'b111 : address_decoder_out = {      1'b1, 7'b0};
          // no default action...
          // default :  ;
          //
        endcase
      end // else
  end // always

endmodule // cond_case_address_decoder_3x8_ena

//
// EOF
//
```

Capítulo 8

Deslocador configurável (*barrel shifter*)

8.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de deslocador configurável (*barrel shifter*). Serão levadas em consideração as seguintes opções:

- Quantidade de deslocamentos.
- Deslocamento lógico e aritmético.
- Deslocamento para esquerda e para direita.
- Rotação para esquerda e para direita.

8.2 Deslocador parametrizado

- A Listagem 1 apresenta um deslocador parametrizado, baseado em descrição comportamental, com parametrização da quantidade de *bits* e da quantidade de deslocamentos, bem como seleção de deslocamento para a esquerda ou para a direita.
- A Listagem 2 apresenta um deslocador genérico parametrizado, baseado em uma descrição mista, comportamental e estrutural, utilizando o deslocador *left-right* da Listagem 1. Ele possui parametrização da quantidade de *bits* e da quantidade de deslocamentos. O deslocador aceita seleção de deslocamento para a esquerda ou para a direita e rotação para a esquerda ou para a direita, ambos dos tipos lógico ou aritmético, também selecionáveis.
- A Listagem 3 apresenta o mesmo deslocador genérico parametrizado da Listagem 2, porém baseado em uma descrição puramente comportamental. Do ponto de vista da descrição, ele pode ser pensado como a aglutinação de ambas as descrições anteriores.

Listagem 1 - Deslocador parametrizado (*left-right*) baseado em descrição comportamental:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design: Data shifter.
//         Selectable shift: left-right.
//         Enabled shift.
//         Parametrized D-bits data.
//         Parametrized S-bits shift.
//         Conditional by using case().
// Filename: param_Dbits_data_Sbits_shifter.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module param_Dbits_data_Sbits_shifter (
left_right, // input  control  left_0  right_1
    enable, // input  enable   shift
  vect_apd, // input  appended data
   vect_in, // input  original data
  vect_out  // output shifted  data
);

  // Parameter Declaration
parameter DATA_WIDTH = 8;
parameter SHFT_WIDTH = 3;

  // I/O Port Declaration
input  left_right;
input      enable;
//
input  [(SHFT_WIDTH-1):0]   vect_apd;
input  [(DATA_WIDTH-1):0]    vect_in;
output [(DATA_WIDTH-1):0]   vect_out;

  // Port Wire Declaration
wire   left_right;
wire       enable;
//
wire   [(SHFT_WIDTH-1):0]   vect_apd;
wire   [(DATA_WIDTH-1):0]    vect_in;

  // Reg Declaration
reg    [(DATA_WIDTH-1):0]   vect_out; // reg because of "always"...


  // Concurrent Assignment
```

```
    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
always @(left_right or vect_apd or vect_in)
  begin
    if (enable== 1)
      // if (SHFT_WIDTH < DATA_WIDTH)
      case (left_right)
        //
        // left
        0 : vect_out = { vect_in[((DATA_WIDTH-1)-(SHFT_WIDTH)):0],
                         vect_apd };
        //
        // right
        1 : vect_out = { vect_apd,
                         vect_in[(DATA_WIDTH-1):(SHFT_WIDTH)] };
        //
        // no default action... only for testing...
        //default: vect_out = vect_in;
        //
      endcase
      //
    else
      vect_out = vect_in;
    //
  end // always
//

endmodule  // param_Dbits_data_Sbits_shifter

//
// EOF
//
```

Listagem 2 - Deslocador genérico parametrizado, baseado no deslocador da Listagem 1:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design: Generic Data shifter
//         Selectable shift:
//           logic_arithmetic,
//           shift_rotate,
//           left_right.
//         Enabled shift.
//         Parametrized D-bits data.
//         Parametrized S-bits shift.
//         Hierarchical and
//           conditional by using case().
// Filename: hier_gen_param_Dbits_data_Sbits_shifter.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_gen_param_Dbits_data_Sbits_shifter (
logic_arithmetic, // input  control  logic_0 arithmetic_1
    shift_rotate, // input  control  shift_0 rotate_1
      left_right, // input  control  left_0  right_1
          enable, // input  enable   shift
         vect_in, // input  original data
        vect_out  // output shifted  data
);

  // Parameter Declaration
parameter DATA_WIDTH = 8;
parameter SHFT_WIDTH = 4;

  // I/O Port Declaration
input logic_arithmetic;
input     shift_rotate;
input       left_right;
input           enable;
//
input  [(DATA_WIDTH-1):0]   vect_in;
output [(DATA_WIDTH-1):0]  vect_out;

  // Port Wire Declaration
wire logic_arithmetic;
wire     shift_rotate;
wire       left_right;
wire           enable;
//
```

```
wire [(DATA_WIDTH-1):0]   vect_in;

  // Reg Declaration
reg  [(SHFT_WIDTH-1):0]  vect_apd; // reg because of "always"...
reg  [(DATA_WIDTH-1):0]  vect_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
always @(logic_arithmetic or shift_rotate or left_right or vect_in)
  begin
    // if (SHFT_WIDTH < DATA_WIDTH)
    case ({logic_arithmetic, shift_rotate, left_right})
      //
      // logic      shift  left
      // logic      shift  right
      3'b000,
      3'b001 : vect_apd = {SHFT_WIDTH{1'b0}};
      //
      // logic      rotate left
      // arithmetic rotate left
      3'b010,
      3'b110 :
      vect_apd = vect_in[(DATA_WIDTH-1):((DATA_WIDTH-1)-(SHFT_WIDTH-1))];
      //
      // logic      rotate right
      // arithmetic rotate right
      3'b011,
      3'b111 : vect_apd = vect_in[(SHFT_WIDTH-1):0];
      //
      // arithmetic shift  left
      3'b100 :
               vect_apd = {SHFT_WIDTH{vect_in[0:0]}};
      //
      // arithmetic shift  right
      3'b101 :
      vect_apd = {SHFT_WIDTH{vect_in[(DATA_WIDTH-1):(DATA_WIDTH-1)]}};
      //
      // no default action... only for testing...
      //default: vect_out = vect_in;
      //
    endcase
    //
  end // always
//
```

```
    // Module Instantiation
//
// Usando Padrão:
//
//   submodule_name  #(new_param_value)  instance_name (instance_port);
//
// MaxPlusII: Não funciona !!!
// Recomenda: defparam ...
//

//
// Usando defparam:
//
param_Dbits_data_Sbits_shifter pddss (.left_right(left_right),
                                      .enable(enable),
                                      .vect_apd(vect_apd),
                                      .vect_in(vect_in),
                                      .vect_out(vect_out)
                                     );
defparam pddss.SHFT_WIDTH = SHFT_WIDTH;
//

endmodule  // hier_gen_param_Dbits_data_Sbits_shifter

//
// EOF
//
```

Listagem 3 - Deslocador genérico parametrizado, sem hierarquia estrutural:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design: Generic Data shifter
//         Selectable shift:
//           logic_arithmetic,
//           shift_rotate,
//           left_right.
//         Enabled shift.
//         Parametrized D-bits data.
//         Parametrized S-bits shift.
//         Conditional by using case().
// Filename: gen_param_Dbits_data_Sbits_shifter.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//
```

```
// Module Declaration
module gen_param_Dbits_data_Sbits_shifter (
logic_arithmetic, // input  control  logic_0 arithmetic_1
    shift_rotate, // input  control  shift_0 rotate_1
      left_right, // input  control  left_0  right_1
          enable, // input  enable   shift
         vect_in, // input  original data
        vect_out  // output shifted  data
);

  // Parameter Declaration
parameter DATA_WIDTH = 8;
parameter SHFT_WIDTH = 4;
//

  // I/O Port Declaration
input logic_arithmetic;
input     shift_rotate;
input       left_right;
input           enable;
//
input  [(DATA_WIDTH-1):0]   vect_in;
output [(DATA_WIDTH-1):0]  vect_out;

  // Port Wire Declaration
wire logic_arithmetic;
wire     shift_rotate;
wire       left_right;
wire           enable;
//
wire [(DATA_WIDTH-1):0]   vect_in;

  // Reg Declaration
reg  [(DATA_WIDTH-1):0]  vect_out; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
always @(logic_arithmetic or shift_rotate or left_right or
         vect_in or enable)
  begin
    if (enable== 1)
      // if (SHFT_WIDTH < DATA_WIDTH)
      case ({logic_arithmetic, shift_rotate, left_right})
```

```
        //
        // logic      shift  left
        3'b000 : vect_out = { vect_in[((DATA_WIDTH-1)-(SHFT_WIDTH)):0],
                              {SHFT_WIDTH{1'b0}} };
        //
        // logic      shift  right
        3'b001 : vect_out = { {SHFT_WIDTH{1'b0}},
                              vect_in[(DATA_WIDTH-1):(SHFT_WIDTH)] };
        //
        // logic      rotate left
        // arithmetic rotate left
        3'b010,
        3'b110 :
        vect_out = { vect_in[((DATA_WIDTH-1)-(SHFT_WIDTH)):0],
                vect_in[(DATA_WIDTH-1):((DATA_WIDTH-1)-(SHFT_WIDTH-1))] };
        //
        // logic      rotate right
        // arithmetic rotate right
        3'b011,
        3'b111 : vect_out = { vect_in[(SHFT_WIDTH-1):0],
                              vect_in[(DATA_WIDTH-1):(SHFT_WIDTH)] };
        //
        // arithmetic shift  left
        3'b100 :
                 vect_out = { vect_in[((DATA_WIDTH-1)-(SHFT_WIDTH)):0],
                              {SHFT_WIDTH{vect_in[0:0]}} };
        //
        // arithmetic shift  right
        3'b101 :
        vect_out = { {SHFT_WIDTH{vect_in[(DATA_WIDTH-1):(DATA_WIDTH-1)]}},
                     vect_in[(DATA_WIDTH-1):(SHFT_WIDTH)]};
        //
        // no default action... only for testing...
        //default: vect_out = vect_in;
        //
      endcase
      //
    else
      vect_out = vect_in;
    //
  end // always
//

endmodule  // gen_param_Dbits_data_Sbits_shifter

//
// EOF
//
```

8.3 *Barrel shifter* com deslocamento genérico

- A Listagem 4 apresenta um deslocador genérico parametrizado e selecionável, baseado em uma descrição mista, comportamental e estrutural, utilizando o deslocador da Listagem 3. Ele possui parametrização da quantidade de *bits* e da quantidade de deslocamentos. O deslocador aceita seleção de deslocamento para a esquerda ou para a direita e rotação para a esquerda ou para a direita, ambos dos tipos lógico ou aritmético, também selecionáveis. Ele aceita ainda a seleção da quantidade de deslocamentos.

Listagem 4 - *Barrel shifter* com deslocamento genérico, baseado no deslocador da Listagem 3:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design: Generic Data shifter
//         Selectable shift:
//           logic_arithmetic,
//           shift_rotate,
//           left_right.
//         Parametrized D-bits data.
//         Selectable   3-bits shift.
//         Hierarchical and
//           conditional by using case().
// Filename: hier_gen_param_Dbits_data_3ctrl_shifter.v
// Coder: Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_gen_param_Dbits_data_3ctrl_shifter (
logic_arithmetic, // input  control  logic_0 arithmetic_1
    shift_rotate, // input  control  shift_0 rotate_1
      left_right, // input  control  left_0  right_1
       vect_shft, // input  control  shift binary value
         vect_in, // input  original data
        vect_out  // output shifted  data
);

  // Parameter Declaration
parameter DATA_WIDTH = 8;
parameter CTRL_WIDTH = 3;
//

  // I/O Port Declaration
input logic_arithmetic;
input     shift_rotate;
input       left_right;
//
input  [(CTRL_WIDTH-1):0] vect_shft;
input  [(DATA_WIDTH-1):0]   vect_in;
```

```
output [(DATA_WIDTH-1):0]  vect_out;

  // Port Wire Declaration
wire logic_arithmetic;
wire     shift_rotate;
wire       left_right;
//
wire [(CTRL_WIDTH-1):0] vect_shft;
wire [(DATA_WIDTH-1):0]   vect_in;

  // Reg Declaration
reg  [(DATA_WIDTH-1):0]  vect_out; // reg because of "always"...
reg  [(DATA_WIDTH-1):0]  vect_sh0; // reg because of "always"...
reg  [(DATA_WIDTH-1):0]  vect_sh1; // reg because of "always"...


  // Concurrent Assignment

//
gen_param_Dbits_data_Sbits_shifter
  gpddss0 (
           .logic_arithmetic(logic_arithmetic),
           .shift_rotate(shift_rotate),
           .left_right(left_right),
           .enable(vect_shft[0]),
           .vect_in(vect_in),
           .vect_out(vect_sh0)
          );
defparam gpddss0.SHFT_WIDTH = 1;
//
gen_param_Dbits_data_Sbits_shifter
  gpddss1 (
           .logic_arithmetic(logic_arithmetic),
           .shift_rotate(shift_rotate),
           .left_right(left_right),
           .enable(vect_shft[1]),
           .vect_in(vect_sh0),
           .vect_out(vect_sh1)
          );
defparam gpddss1.SHFT_WIDTH = 2;
//
gen_param_Dbits_data_Sbits_shifter
  gpddss2 (
           .logic_arithmetic(logic_arithmetic),
           .shift_rotate(shift_rotate),
           .left_right(left_right),
           .enable(vect_shft[2]),
           .vect_in(vect_sh1),
           .vect_out(vect_out)
```

```
          );
defparam gpddss2.SHFT_WIDTH = 4;
//

endmodule  // hier_gen_param_Dbits_data_3ctrl_shifter

//
// EOF
//
```

Capítulo 9

Decodificador

9.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de decodificador.

9.2 Decodificador com validação de código

A seguir, são apresentados códigos para um decodificador com validação de código. O decodificador em questão possui um sinal de saída que alerta quando o código de entrada é válido (A = '1') ou não (A = '0'). A especificação do conjunto dos códigos de entrada válidos, bem como do mapeamento entre os códigos de entrada e de saída, pode ser obtida diretamente da Listagem 1.

- A Listagem 1 apresenta um decodificador com validação de código, baseado em código condicional.
- A Listagem 2 apresenta um decodificador com validação de código, baseado em operador lógico.

Listagem 1 - Decodificador com validação de código, baseado em código condicional:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Decoder.
//           Input code validation.
//           Conditional description
//             by using case().
// Filename: cond_case_decoder_4x8_2017_2_a2.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module cond_case_decoder_4x8_2017_2_a2 (
x, // input  code
y, // output code
```

```
a  // output input-code-validation signal
);

  // Parameter Declaration
parameter INP_WIDTH = 4;
parameter OUT_WIDTH = 8;

  // I/O Port Declaration
input  [(INP_WIDTH-1):0] x;
output [(OUT_WIDTH-1):0] y;
output                   a;

  // Port Wire Declaration
wire [(INP_WIDTH-1):0] x;

  // Reg Declaration
reg [(OUT_WIDTH-1):0] y; // reg because of "always"...
reg                   a; // reg because of "always"...


  // Concurrent Assignment

    // Always Statement
//
// always @(*)   <-- MaxPlusII não aceitou...
//
always @(x)
  begin
    //
    case (x)
      4'b0001 :
        begin
          a = 1;
          y = 8'b01000011;
        end
      4'b0011 :
        begin
          a = 1;
          y = 8'b01100100;
        end
      4'b0100 :
        begin
          a = 1;
          y = 8'b10010110;
        end
      4'b0110 :
        begin
          a = 1;
          y = 8'b10111001;
```

```
        end
      //
      4'b1001 :
        begin
          a = 1;
          y = 8'b11001011;
        end
      4'b1011 :
        begin
          a = 1;
          y = 8'b11101100;
        end
      4'b1100 :
        begin
          a = 1;
          y = 8'b00011110;
        end
      4'b1110 :
        begin
          a = 1;
          y = 8'b00110001;
        end
      //
      // Error condition: invalid input code !!!
      // Output value   : not specified...
      //
      default :
        begin
          a = 0;
          y = 8'b00000000;
        end
    endcase
    //
  end // always
//

endmodule // cond_case_decoder_4x8_2017_2_a2

//
// EOF
//
```

Listagem 2 - Decodificador com validação de código, baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Decoder.
//           Input code validation.
//           By using logical operators
//              and continuous assignment.
// Filename: op_decoder_4x8_2017_2_a2.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_decoder_4x8_2017_2_a2 (
x, // input  code
y, // output code
a  // output input-code-validation signal
);

  // Parameter Declaration
parameter INP_WIDTH = 4;
parameter OUT_WIDTH = 8;

  // I/O Port Declaration
input  [(INP_WIDTH-1):0] x;
output [(OUT_WIDTH-1):0] y;
output                   a;

  // Port Wire Declaration
wire [(INP_WIDTH-1):0] x;
wire [(OUT_WIDTH-1):0] y;
wire                   a;

  // Internal Wire Declaration
wire w_and_1;
wire w_and_2;
wire w_and_3;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xor   xor_a   (a, x[2], x[0]);
//
//
xor   xor_y7  (   y[7], x[3], x[2]);
```

```
not   not_y6  (   y[6], x[2]      );
//
and   and_1   (w_and_1,   ~x[3],    x[2],    x[1]);
and   and_2   (w_and_2,    x[3],   ~x[1]         );
and   and_3   (w_and_3,    x[3],    x[0]         );
or    or_y3   (   y[3], w_and_1, w_and_2, w_and_3);
//
xnor  xnor_y2 (y[2], x[1], x[0]);
not   not_y1  (y[1], x[1]      );
xor   xor_y0  (y[0], x[1], x[0]);
//

    // Continuous Assignment Statement
//
assign y[5] = x[1];
assign y[4] = x[2];
//

endmodule // op_decoder_4x8_2017_2_a2

//
// EOF
//
```

Capítulo 10

Conversor de códigos

10.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de conversores de códigos. Serão considerados os seguintes códigos: Binário Seqüencial, *One-hot*, Gray, Johnson, BCD (*Binary-Coded Decimal*).

10.2 Conversor Binário-*One-hot*

Um conversor de código Binário Seqüencial para código *One-hot* é um bloco funcional com B *bits* de entrada e $N = 2^B$ *bits* de saída. A saída S_0 representa o LSB, enquanto a saída S_{N-1} representa o MSB. A saída S_k, para $0 \leq k \leq (N-1)$, é equivalente ao mintermo m_k. Portanto, por definição, esse conversor é um *address decoder* BxN.

10.3 Conversor Binário-Gray

- A Listagem 1 apresenta um conversor de código Binário Seqüencial para código Gray, baseado em operadores.

Listagem 1 - Conversor de Binário Seqüencial para Gray, baseado em operadores:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Code converter.
//           Binary-to-Gray.
//           Building block: 4 bits.
//           By using logical operators
//              and continuous assignment.
// Filename: binary_to_gray_4bits.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module binary_to_gray_4bits (
```

```
msb_plus, // input  MSB + 1
 data_in, // input  code_in
data_out, // output code_out
lsb_less  // output LSB - 1
);

  // Parameter Declaration
parameter DATA_WIDTH = 4;

  // I/O Port Declaration
input                      msb_plus;
input  [(DATA_WIDTH-1):0]  data_in;
output [(DATA_WIDTH-1):0] data_out;
output                     lsb_less;

  // Port Wire Declaration
wire                       msb_plus;
wire   [(DATA_WIDTH-1):0]  data_in;
wire   [(DATA_WIDTH-1):0] data_out;
wire                       lsb_less;


  // Concurrent Assignment

    // Predefined Logic Primitives Instantiation
//
xor  xor_3 (data_out[3],   msb_plus, data_in[3]);
xor  xor_2 (data_out[2], data_in[3], data_in[2]);
xor  xor_1 (data_out[1], data_in[2], data_in[1]);
xor  xor_0 (data_out[0], data_in[1], data_in[0]);
//

    // Continuous Assignment Statement
//
assign lsb_less = data_in[0];
//

endmodule // binary_to_gray_4bits

//
// EOF
//
```

Capítulo 11

Separador e relacionador de *bits* 1 e 0 em palavra de N *bits*

11.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de separador e de relacionador de *bits* 1 e 0, em palavra de N *bits*.

11.2 Separador de *bits* 1 e 0 em palavra de N *bits*

A seguir, é apresentado código para um separador de *bits* 1 e 0, em uma palavra de N *bits*. O separador em questão aloca os *bits* 1 da palavra de entrada nos *bits* mais significativos da palavra de saída. Ele é parametrizável, apresentando $N = 5$ como tamanho de palavra original.

- A Listagem 1 apresenta um separador de *bits* 1 e 0, em uma palavra de N *bits*, modular e não otimizado, descrito por um modelo comportamental, parametrizável, que utiliza dois ciclos de códigos aninhados (*nested loops*).
- A Listagem 2 apresenta uma instanciação do separador de *bits* 1 e 0 da Listagem 1, empregando $N = 8$ *bits*.

Listagem 1 - Separador de *bits* 1 e 0, em palavra de N *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Bit-1 and bit-0 sets separation
//             Bit-1 set at MSB-side
//             Bit-0 set at LSB-side
//             Generic N - Based on org param N=5
//           Model: Behavioral - Two nested for-loops
// Filename: separa_bits_1_0_word_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module separa_bits_1_0_word_N (
data_in , // input vector WIDTH-bits
data_out  // input vector WIDTH-bits
);

  // Parameter Declaration
parameter WIDTH_5 = 5;

  // I/O Port Declaration
input  [(WIDTH_5-1):0] data_in ;
output [(WIDTH_5-1):0] data_out;

  // Port Wire Declaration
wire [(WIDTH_5-1):0] data_in ;
wire [(WIDTH_5-1):0] data_out;

  // Internal Reg Declaration
reg [(WIDTH_5-1):0] loop_inp_vector;
reg [(WIDTH_5-1):0] loop_out_vector;
reg [(WIDTH_5-1):0] loop_aux_vector;
//
reg loop_inp_propagate_bit;
reg loop_out_propagate_bit;

  // Internal Var Declaration
integer lin;
integer col;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(data_in)
  begin
    //
    loop_out_vector = data_in;
    for (lin = (WIDTH_5-1); lin >= 0; lin = lin - 1)
      begin
        loop_inp_vector = loop_out_vector;
        //
        loop_out_propagate_bit = 0;
        for (col = (WIDTH_5-1); col >= 0; col = col - 1)
          begin
            loop_inp_propagate_bit = loop_out_propagate_bit;
            //
            loop_out_vector[col]   = loop_inp_vector[col]
                                     &
                                     loop_inp_propagate_bit;
```

```
            //
            loop_out_propagate_bit = loop_inp_vector[col]
                                     |
                                     loop_inp_propagate_bit;
            //
          end
        //
        loop_aux_vector[lin] = loop_out_propagate_bit;
      end
    //
  end
//

    // Continuous Assignment Statement
//
assign data_out = loop_aux_vector;
//

endmodule // separa_bits_1_0_word_N

//
// EOF
//
```

Listagem 2 - Instanciação do separador de *bits* 1 e 0, com palavra de $N = 8$ *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Bit-1 and bit-0 sets separation
//             Bit-1 set at MSB-side
//             Bit-0 set at LSB-side
//           Test version: N = 8
//           Model: Hierarchy - Instantiation
// Filename: separa_bits_1_0_word_N8.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module separa_bits_1_0_word_N8 (
data_in , // input vector WIDTH-bits
data_out  // input vector WIDTH-bits
);

  // Parameter Declaration
parameter WIDTH_8 = 8;
```

```
  // I/O Port Declaration
input  [(WIDTH_8-1):0] data_in ;
output [(WIDTH_8-1):0] data_out;

  // Port Wire Declaration
wire [(WIDTH_8-1):0] data_in ;
wire [(WIDTH_8-1):0] data_out;


  // Concurrent Assignment

    // Module Instantiation
//
separa_bits_1_0_word_N
  separa_org_width_5
    ( .data_in(data_in), .data_out(data_out) );
//
defparam separa_org_width_5.WIDTH_5 = WIDTH_8;
//

endmodule // separa_bits_1_0_word_N8

//
// EOF
//
```

11.3 Árbitro da relação entre *bits* 1 e 0 em palavra de N *bits*

A seguir, é apresentado código para um árbitro da relação entre *bits* 1 e 0, em palavra de N *bits*. O árbitro em questão indica se a quantidade de *bits* 1 é menor_que, igual_a ou maior_que os *bits* 0 da palavra. Ele analisa os *bits* medianos de uma palavra, previamente organizada por um separador de *bits* 1 e 0, e toma a decisão.

- A Listagem 3 apresenta um árbitro para *bits* 1 alocados nos *bits* menos significativos da palavra.
- A Listagem 4 apresenta um árbitro para *bits* 1 alocados nos *bits* mais significativos da palavra.
- A Listagem 5 apresenta um árbitro que é independente da ordem com que os dois grupos de *bits* são alocados na palavra.

Listagem 3 - Árbitro para *bits* 1 alocados nos *bits* menos significativos da palavra:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Total bits-1 and bits-0 comparison:
//             1's  "Less - Equal - Greater"  0's
//           Model: logic equation
//           Type : LSB
//                    Bit-1 set at LSB-side
//                    Bit-0 set at MSB-side
// Filename: total_1_lt_eq_gt_0_after_separation_lsb.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module total_1_lt_eq_gt_0_after_separation_lsb (
// --> Even0_Odd1 == 0 for even N
// --> Even0_Odd1 == 1 for odd  N
Even0_Odd1, // input
//
// --> Bits at: N/2 and N/2 - 1
Mid_N,         // input
Mid_N_minus_1, // input
//
// --> Less, Equal, Greater: 1 versus 0
L1T0, // output
E1A0, // output
G1T0  // output
);

  // I/O Port Declaration
input  Even0_Odd1;
//
input  Mid_N;
input  Mid_N_minus_1;
//
output L1T0;
output E1A0;
output G1T0;

  // Port Wire Declaration
wire Even0_Odd1;
//
wire Mid_N;
wire Mid_N_minus_1;
//
wire L1T0;
```

```
wire E1A0;
wire G1T0;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign
//
  L1T0 = (~Mid_N_minus_1) | (~Mid_N & Even0_Odd1),
//
  G1T0 = ( Mid_N),
//
//       E1A0 = (~L1T0) & (~G1T0);
  E1A0 = (~Mid_N &  Mid_N_minus_1 & ~Even0_Odd1);
//

endmodule // total_1_lt_eq_gt_0_after_separation_lsb

//
// EOF
//
```

Listagem 4 - Árbitro para *bits* 1 alocados nos *bits* mais significativos da palavra:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Total bits-1 and bits-0 comparison:
//             1's  "Less - Equal - Greater"  0's
//           Model: logic equation
//           Type : MSB
//                    Bit-1 set at MSB-side
//                    Bit-0 set at LSB-side
// Filename: total_1_lt_eq_gt_0_after_separation_msb.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module total_1_lt_eq_gt_0_after_separation_msb (
// --> Even0_Odd1 == 0 for even N
// --> Even0_Odd1 == 1 for odd  N
Even0_Odd1, // input
//
// --> Bits at: N/2 and N/2 - 1
```

```
Mid_N,         // input
Mid_N_minus_1, // input
//
// --> Less, Equal, Greater: 1 versus 0
L1T0, // output
E1A0, // output
G1T0  // output
);

  // I/O Port Declaration
input  Even0_Odd1;
//
input  Mid_N;
input  Mid_N_minus_1;
//
output L1T0;
output E1A0;
output G1T0;

  // Port Wire Declaration
wire Even0_Odd1;
//
wire Mid_N;
wire Mid_N_minus_1;
//
wire L1T0;
wire E1A0;
wire G1T0;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign
//
  L1T0 = (~Mid_N),
//
  G1T0 = ( Mid_N_minus_1) | ( Mid_N & Even0_Odd1),
//
//       E1A0 = (~L1T0) & (~G1T0);
  E1A0 = ( Mid_N & ~Mid_N_minus_1 & ~Even0_Odd1);
//

endmodule // total_1_lt_eq_gt_0_after_separation_msb

//
// EOF
//
```

Listagem 5 - Árbitro que é independente da ordem de alocação dos dois grupos de *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Total bits-1 and bits-0 comparison:
//             1's  "Less - Equal - Greater"  0's
//           Model: logic equation
//           Type : General
//                    Bit-1 set at MSB-side
//                    Bit-0 set at LSB-side
//                    or
//                    Bit-1 set at LSB-side
//                    Bit-0 set at MSB-side
// Filename: total_1_lt_eq_gt_0_after_separation.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module total_1_lt_eq_gt_0_after_separation (
// --> Even0_Odd1 == 0 for even N
// --> Even0_Odd1 == 1 for odd  N
Even0_Odd1, // input
//
// --> Bits at: N/2 and N/2 - 1
Mid_N,         // input
Mid_N_minus_1, // input
//
// --> Less, Equal, Greater: 1 versus 0
L1T0, // output
E1A0, // output
G1T0  // output
);

  // I/O Port Declaration
input  Even0_Odd1;
//
input  Mid_N;
input  Mid_N_minus_1;
//
output L1T0;
output E1A0;
output G1T0;

  // Port Wire Declaration
wire Even0_Odd1;
//
wire Mid_N;
```

```
wire Mid_N_minus_1;
//
wire L1T0;
wire E1A0;
wire G1T0;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign
//
  L1T0 = (~Mid_N & ~Mid_N_minus_1) | (~Mid_N & Even0_Odd1),
//
  G1T0 = ( Mid_N &  Mid_N_minus_1) | ( Mid_N & Even0_Odd1),
//
//       E1A0 = (~L1T0) & (~G1T0);
  E1A0 = (~Mid_N &  Mid_N_minus_1 & ~Even0_Odd1) |
         ( Mid_N & ~Mid_N_minus_1 & ~Even0_Odd1);
//

endmodule // total_1_lt_eq_gt_0_after_separation

//
// EOF
//
```

11.4 Relacionador de *bits* 1 e 0 em palavra de N *bits*

A seguir, é apresentado código para um relacionador de *bits* 1 e 0, em uma palavra de N *bits*. O relacionador em questão indica se a quantidade de *bits* 1 é menor_que, igual_a ou maior_que os *bits* 0 da palavra. Inicialmente, ele realiza um agrupamento dos *bits*, utilizando um separador de *bits* apresentado anteriormente. Em seguida, ele analisa os *bits* medianos do agrupamento, utilizando um árbitro apresentado anteriormente.

- A Listagem 6 apresenta o relacionador de *bits* 1 e 0, hierárquico e parametrizável, baseado no separador de *bits* da Listagem 1 e no árbitro da Listagem 5.
- A Listagem 7 apresenta uma instanciação do relacionador de *bits* 1 e 0 da Listagem 6, empregando $N = 7$ *bits*.

Listagem 6 - Relacionador de *bits* 1 e 0, em palavra de N *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Total bits-1 and bits-0 separation & comparison:
//             1's  "Less - Equal - Greater"  0's
//           Model: Hierarchy - Instantiation
// Filename: separa_e_relaciona_bits_1_0_word_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module separa_e_relaciona_bits_1_0_word_N (
data_in , // input  vector N-bits
data_out, // output vector N-bits
    L1T0, // output single-bit
    E1A0, // output single-bit
    G1T0  // output single-bit
);

  // Parameter Declaration
//
parameter WIDTH_N = 4;

  // I/O Port Declaration
input  [(WIDTH_N-1):0] data_in ;
output [(WIDTH_N-1):0] data_out;
output                 L1T0;
output                 E1A0;
output                 G1T0;

  // Port Wire Declaration
wire [(WIDTH_N-1):0] data_in ;
wire [(WIDTH_N-1):0] data_out;
wire                 L1T0;
wire                 E1A0;
wire                 G1T0;

  // Internal Wire Declaration
wire [(WIDTH_N-1):0] data_out_bus;

  // Internal Reg Declaration
reg Even0_Odd1;
reg Mid_N;
reg Mid_N_minus_1;
```

```
  // Concurrent Assignment

    // Module Instantiation
//
separa_bits_1_0_word_N
  separa_org_width_5
    ( .data_in(data_in), .data_out(data_out_bus) );
//
defparam separa_org_width_5.WIDTH_5 = WIDTH_N;
//

    // Continuous Assignment Statement
//
assign data_out = data_out_bus;
//

    // Always Statement
//
always @(data_out_bus)
begin
if ( (WIDTH_N % 2) == 0 )
  begin
    Even0_Odd1    = 1'b0;  //  WIDTH_N is even...
    Mid_N         = data_out_bus[ ( (WIDTH_N-1)/2 ) + 1 ];
    Mid_N_minus_1 = data_out_bus[ ( (WIDTH_N-1)/2 )     ];
  end
else
  begin
    Even0_Odd1    = 1'b1;  //  WIDTH_N is odd...
    Mid_N         = data_out_bus[   (WIDTH_N-1)/2       ];
    Mid_N_minus_1 = data_out_bus[ ( (WIDTH_N-1)/2 ) - 1 ];
  end
end
//
    // Module Instantiation
//
total_1_lt_eq_gt_0_after_separation
  total_after_separation
    (.Even0_Odd1(Even0_Odd1),
     .Mid_N(Mid_N),
     .Mid_N_minus_1(Mid_N_minus_1),
     .L1T0(L1T0),
     .E1A0(E1A0),
     .G1T0(G1T0) );
//

endmodule // separa_e_relaciona_bits_1_0_word_N

//
```

```
// EOF
//
```

Listagem 7 - Instanciação do relacionador de *bits* 1 e 0, com palavra de $N = 7$ *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Total bits-1 and bits-0 separation & comparison:
//             1's  "Less - Equal - Greater"  0's
//           Model: Hierarchy - Instantiation
//           Test version: N = 7
// Filename: separa_e_relaciona_bits_1_0_word_N7.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module separa_e_relaciona_bits_1_0_word_N7 (
data_in , // input  vector N-bits
data_out, // output vector N-bits
    L1T0, // output single-bit
    E1A0, // output single-bit
    G1T0  // output single-bit
);

  // Parameter Declaration
//
parameter WIDTH_N = 7;

  // I/O Port Declaration
input  [(WIDTH_N-1):0] data_in ;
output [(WIDTH_N-1):0] data_out;
output                 L1T0;
output                 E1A0;
output                 G1T0;

  // Port Wire Declaration
wire [(WIDTH_N-1):0] data_in ;
wire [(WIDTH_N-1):0] data_out;
wire                 L1T0;
wire                 E1A0;
wire                 G1T0;


  // Concurrent Assignment
```

```
    // Module Instantiation
//
separa_e_relaciona_bits_1_0_word_N
  separa_e_relaciona
    ( .data_in(data_in),
      .data_out(data_out),
      .L1T0(L1T0),
      .E1A0(E1A0),
      .G1T0(G1T0) );
//
defparam separa_e_relaciona.WIDTH_N = WIDTH_N;
//

endmodule // separa_e_relaciona_bits_1_0_word_N7

//
// EOF
//
```

Capítulo 12

Somadores básicos

12.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de somadores básicos.

12.2 Somador de 2 *bits* ou *half adder* (HA)

- A Listagem 1 apresenta um somador de 2 *bits* ou meio somador ou *half adder* (HA), baseado em operador lógico.

Listagem 1 - Somador de 2 *bits* ou *half adder* (HA), baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Half Adder (HA)
//           by using logical operators.
// Filename: op_half_adder.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_half_adder (
op1, // op 1
op2, // op 2
  r, // result
cout // carry_out
);

  // I/O Port Declaration
input   op1;
input   op2;
output    r;
output cout;

  // Port Wire Declaration
```

```
wire  op1;
wire  op2;
wire    r;
wire cout;

  // Internal Wire Declaration
// no internal wires

  // Concurrent Assignment
    // Continuous Assignment Statement
assign  r    = op1 ^ op2;
assign  cout = (op1 & op2);

endmodule // op_half_adder

//
// EOF
//
```

12.3 Somador de 3 *bits* ou *full adder* (FA)

- A Listagem 2 apresenta um somador de 3 *bits* ou somador completo ou *full adder* (FA), baseado em operador lógico.

Listagem 2 - Somador de 3 *bits* ou *full adder* (FA), baseado em operador lógico:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Full Adder (FA)
//           by using logical operators.
// Filename: op_full_adder.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_full_adder (
op1, // op 1
op2, // op 2
cin, // carry_in
  r, // result
cout // carry_out
);

  // I/O Port Declaration
```

```
input   op1;
input   op2;
input   cin;
output    r;
output cout;

  // Port Wire Declaration
wire  op1;
wire  op2;
wire  cin;
wire    r;
wire cout;

  // Internal Wire Declaration
// no internal wires

  // Concurrent Assignment
    // Continuous Assignment Statement
assign  r    = op1 ^ op2 ^ cin;
assign  cout = (op1 & op2) | (op1 & cin) | (op2 & cin);

endmodule // op_full_adder

//
// EOF
//
```

12.4 Somador com propagação de *carry* (CPA ou RCA)

A seguir, são apresentados códigos para um somador com propagação de *carry* ou *Carry Propagate Adder* (CPA) ou *Ripple Carry Adder* (RCA).

- A Listagem 3 apresenta um RCA de 4 *bits*, empregando o *full adder* da Listagem 2.
- A Listagem 4 apresenta um RCA de 8 *bits*, empregando o RCA de 4 *bits* da Listagem 3.

Listagem 3 - Somador RCA de 4 *bits*, empregando o *full adder*:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Carry Propagate Adder (CPA)
//           or
//           Ripple Carry Adder (RCA)
//           4 bits
//           based on Full Adder.
// Filename: hier_CPA_RCA_4_bits.v
// Coder   : Alexandre Santos de la Vega
```

```
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_CPA_RCA_4_bits (
vet1, // op 1
vet2, // op 2
cin,  // carry_in
vetr, // result
cout  // carry_out
);

  // Parameter Declaration
parameter WIDTH=4;

  // I/O Port Declaration
input  [(WIDTH-1):0] vet1;
input  [(WIDTH-1):0] vet2;
input                cin;
output [(WIDTH-1):0] vetr;
output               cout;

  // Port Wire Declaration
wire [(WIDTH-1):0] vet1;
wire [(WIDTH-1):0] vet2;
wire               cin;
wire [(WIDTH-1):0] vetr;
wire               cout;

  // Internal Wire Declaration
wire [(WIDTH-1):0] carry_chain;

  // Concurrent Assignment

    // Module Instantiation
//
op_full_adder  fa0 ( .op1(vet1[0]), .op2(vet2[0]),
                     .cin(cin),
                     .res(vetr[0]), .cout(carry_chain[0]) );
//
op_full_adder  fa1 ( .op1(vet1[1]), .op2(vet2[1]),
                     .cin(carry_chain[0]),
                     .res(vetr[1]), .cout(carry_chain[1]) );
//
op_full_adder  fa2 ( .op1(vet1[2]), .op2(vet2[2]),
                     .cin(carry_chain[1]),
                     .res(vetr[2]), .cout(carry_chain[2]) );
//
```

```
op_full_adder  fa3 ( .op1(vet1[3]), .op2(vet2[3]),
                     .cin(carry_chain[2]),
                     .res(vetr[3]), .cout(carry_chain[3]) );
//

    // Continuous Assignment Statement
//
assign  cout = carry_chain[3];
//

endmodule // hier_CPA_RCA_4_bits

//
// EOF
//
```

Listagem 4 - Somador RCA de 8 *bits*, empregando o RCA de 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Carry Propagate Adder (CPA)
//           or
//           Ripple Carry Adder (RCA)
//           2 modules, 4 bits each,
//           based on CPA/RCA.
// Filename: hier_CPA_RCA_2x4_bits.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module hier_CPA_RCA_2x4_bits (
vet1, // op 1
vet2, // op 2
cin,  // carry_in
vetr, // result
cout  // carry_out
);

  // Parameter Declaration
parameter WIDTH_OP  = 8;
parameter NbrOfMod  = 2;
parameter WIDTH_MOD = WIDTH_OP / NbrOfMod;

  // I/O Port Declaration
```

```
input  [(WIDTH_OP-1):0] vet1;
input  [(WIDTH_OP-1):0] vet2;
input                    cin;
output [(WIDTH_OP-1):0] vetr;
output                  cout;

  // Port Wire Declaration
wire [(WIDTH_OP-1):0] vet1;
wire [(WIDTH_OP-1):0] vet2;
wire                   cin;
wire [(WIDTH_OP-1):0] vetr;
wire                  cout;

  // Internal Wire Declaration
wire [(NbrOfMod-1):0] carry_chain;

  // Concurrent Assignment

    // Module Instantiation
//
hier_CPA_RCA_4_bits  RCA4_0 ( .vet1(vet1[(WIDTH_MOD-1):0]),
                              .vet2(vet2[(WIDTH_MOD-1):0]),
                              .cin (cin),
                              .vetr(vetr[(WIDTH_MOD-1):0]),
                              .cout(carry_chain[0])        );
//
hier_CPA_RCA_4_bits  RCA4_1 ( .vet1(vet1[(WIDTH_OP-1):(WIDTH_MOD)]),
                              .vet2(vet2[(WIDTH_OP-1):(WIDTH_MOD)]),
                              .cin(carry_chain[0]),
                              .vetr(vetr[(WIDTH_OP-1):(WIDTH_MOD)]),
                              .cout(carry_chain[1])                );
//

    // Continuous Assignment Statement
//
assign  cout = carry_chain[1];
//

endmodule // hier_CPA_RCA_2x4_bits

//
// EOF
//
```

12.5 Somador binário com modelo comportamental

A seguir, são apresentados códigos para um somador binário com modelo comportamental. No primeiro, o total de *bits* dos operandos é definido por meio da declaração de um parâmetro. Nos demais, é feita uma instanciação do primeiro e alteração do valor do parâmetro inicial.

- A Listagem 5 apresenta um somador binário de 4 *bits*, empregando modelo comportamental e declaração de parâmetro.
- A Listagem 6 apresenta um somador binário de 6 *bits*, empregando instanciação do somador da Listagem 5, com alteração do valor do parâmetro inicial.
- A Listagem 7 apresenta um somador binário de 2 *bits*, empregando instanciação do somador da Listagem 5, com alteração do valor do parâmetro inicial.

Listagem 5 - Somador binário de 4 *bits*, empregando modelo comportamental:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Binary adder
//           Model: assign
//           Width: 4 by param
// Filename: binary_adder_assign_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module binary_adder_assign_param_width_4 (
 cin, // carry in
 op1, // op 1
 op2, // op 2
 res, // result
cout  // carry out
);

  // Parameter Declaration
parameter WIDTH = 4;

  // I/O Port Declaration
input                 cin;
input  [(WIDTH-1):0]  op1;
input  [(WIDTH-1):0]  op2;
output [(WIDTH-1):0]  res;
output               cout;

  // Port Wire Declaration
wire                cin;
wire [(WIDTH-1):0]  op1;
wire [(WIDTH-1):0]  op2;
```

```
wire [(WIDTH-1):0]  res;
wire               cout;

  // Concurrent Assignment

    // Continuous Assignment Statement
assign {cout,res} = op1 + op2 + cin;

endmodule // binary_adder_assign_param_width_4

//
// EOF
//
```

Listagem 6 - Somador binário de 6 *bits*, empregando instanciação e alteração de parâmetro:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Binary adder
//           Model: instantiation
//           Width: 6 by defparam
// Filename: binary_adder_defparam_width_6.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module binary_adder_defparam_width_6 (
 cin, // carry in
 op1, // op 1
 op2, // op 2
 res, // result
cout  // carry out
);

  // Parameter Declaration
parameter WIDTH = 6;

  // I/O Port Declaration
input                 cin;
input  [(WIDTH-1):0]  op1;
input  [(WIDTH-1):0]  op2;
output [(WIDTH-1):0]  res;
output               cout;

  // Port Wire Declaration
```

```
wire                cin;
wire [(WIDTH-1):0]  op1;
wire [(WIDTH-1):0]  op2;
wire [(WIDTH-1):0]  res;
wire               cout;

  // Concurrent Assignment

    // Module Instantiation

//
// MaxPlusII: Não funciona !!!
// Recomenda: defparam ...
//
// binary_adder_assign_param_width_4 #(6) adder6 (cin,op1,op2,r,cout);
//

//
binary_adder_assign_param_width_4 adder6 (cin,op1,op2,res,cout);
defparam adder6.WIDTH = WIDTH;
//

endmodule // binary_adder_defparam_width_6

//
// EOF
//
```

Listagem 7 - Somador binário de 2 *bits*, empregando instanciação e alteração de parâmetro:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Binary adder
//           Model: instantiation
//           Width: 2 by defparam
// Filename: binary_adder_defparam_width_2.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module binary_adder_defparam_width_2 (
 cin, // carry in
 op1, // op 1
 op2, // op 2
 res, // result
```

```
cout  // carry out
);

  // Parameter Declaration
parameter WIDTH = 2;

  // I/O Port Declaration
input                 cin;
input  [(WIDTH-1):0]  op1;
input  [(WIDTH-1):0]  op2;
output [(WIDTH-1):0]  res;
output               cout;

  // Port Wire Declaration
wire                cin;
wire [(WIDTH-1):0]  op1;
wire [(WIDTH-1):0]  op2;
wire [(WIDTH-1):0]  res;
wire               cout;

  // Concurrent Assignment

    // Module Instantiation

//
// MaxPlusII: Não funciona !!!
// Recomenda: defparam ...
//
// binary_adder_assign_param_width_4 #(2) adder2 (cin,op1,op2,r,cout);
//

//
binary_adder_assign_param_width_4 adder2 (cin,op1,op2,res,cout);
defparam adder2.WIDTH = WIDTH;
//

endmodule // binary_adder_defparam_width_2

//
// EOF
//
```

Capítulo 13

Detector de *overflow* e saturador

13.1 Introdução

Nesse capítulo, são apresentados códigos para alguns exemplos de detectores de *overflow* e de saturadores.

13.2 Detector de *overflow*

- A Listagem 1 apresenta um detector de *overflow* em adição de operandos codificados em complemento-a-2.

Listagem 1 - Detector de *overflow* em adição de operandos codificados em complemento-a-2:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Detector de overflow em adicao
//           Dados numericos codificados em complemento a 2
//           Model: logic equation
// Filename: equ_detector_overflow_adicao_complemento_2.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mai_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module equ_detector_overflow_adicao_complemento_2 (
op1_sign,     // input  sign op1
op2_sign,     // input  sign op2
res_sign,     // input  sign addition result
add_overflow, // output overflow on addition
);

  // I/O Port Declaration
input  op1_sign;
input  op2_sign;
input  res_sign;
output add_overflow;
```

```
  // Port Wire Declaration
wire op1_sign;
wire op2_sign;
wire res_sign;
wire add_overflow;


  // Concurrent Assignment

    // Continuous Assignment Statement
//
assign add_overflow = (~op1_sign & ~op2_sign &  res_sign)
                      |
                      ( op1_sign &  op2_sign & ~res_sign);
//

endmodule //  equ_detector_overflow_adicao_complemento_2

//
// EOF
//
```

13.3 Saturador

A seguir, são apresentados códigos para saturadores. O saturador é um bloco funcional responsável por corrigir um valor proveniente de operação numérica com ocorrência de *overflow*.

- Os saturadores a seguir levam em consideração que os dados numéricos estão codificados em complemento-a-2.
- A Listagem 2 apresenta um saturador para valores codificados em complemento-a-2, configurável e parametrizável.
- A Listagem 3 apresenta um saturador para valores codificados em complemento-a-2, de 5 *bits*, empregando o saturador configurável e parametrizável da Listagem 2.

Listagem 2 - Saturador configurável e parametrizável:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Saturador
//           Valor numerico codificado em complemento a 2
//           Saturacao configuravel:
//             0 = Max Representable Number
//             1 = Max Representable Module
//           Model: Behavioral: 'Always + Logic Equations'
```

```
// Filename: equ_saturador_complemento_2_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mai_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module equ_saturador_complemento_2_N (
overflow,            // input  overflow signal
sat_0MaxNbr_1MaxMod, // input  saturation type:
                     //          0 = Max Representable Number
                     //          1 = Max Representable Module
inp_value,           // input  numeric value 2's complement
out_value            // output numeric value 2's complement
);

  // Parameter Declaration
parameter WIDTH_N = 3;

  // I/O Port Declaration
//
input overflow;
input sat_0MaxNbr_1MaxMod;
//
input  [(WIDTH_N-1):0] inp_value;
output [(WIDTH_N-1):0] out_value;

  // Port Wire Declaration
//
wire overflow;
wire sat_0MaxNbr_1MaxMod;
//
wire [(WIDTH_N-1):0] inp_value;

  // Port Reg Declaration
reg [(WIDTH_N-1):0] out_value;  // 'reg' because of always...

  // Internal Var Declaration
integer bit_ind;


  // Concurrent Assignment

    // Always Statement
//
always @(overflow or sat_0MaxNbr_1MaxMod or inp_value)
  begin
    // signal bit = MSB
    out_value[WIDTH_N-1] = ( ~overflow &  inp_value[WIDTH_N-1] )
```

```
                          |
                          (  overflow & ~inp_value[WIDTH_N-1] );
    // mantissa bits, but LSB
    for ( bit_ind = (WIDTH_N-2); bit_ind > 0; bit_ind = (bit_ind - 1) )
      begin
        out_value[bit_ind] = ( ~overflow & inp_value[bit_ind]  )
                             |
                             (  overflow & inp_value[WIDTH_N-1] );
      end
    // mantissa bit = LSB
    out_value[0] = ( ~overflow & inp_value[0] )
                   |
                   (  overflow & inp_value[WIDTH_N-1] )
                   |
                   (  overflow & sat_0MaxNbr_1MaxMod );
  end
//

endmodule //  equ_saturador_complemento_2_N

//
// EOF
//
```

Listagem 3 - Saturador de 5 *bits*, empregando o saturador configurável e parametrizável:

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Saturador
//           Valor numerico codificado em complemento a 2
//           Saturacao configuravel:
//             0 = Max Representable Number
//             1 = Max Representable Module
//           Model: Hierarchy - Instantiation
//           Test version: N = 5
// Filename: equ_saturador_complemento_2_N5.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mai_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module equ_saturador_complemento_2_N5 (
overflow,            // input  overflow signal
sat_0MaxNbr_1MaxMod, // input  saturation type:
                     //          0 = Max Representable Number
                     //          1 = Max Representable Module
```

```
inp_value,           // input  numeric value 2's complement
out_value            // output numeric value 2's complement
);

  // Parameter Declaration
parameter WIDTH_N = 5;

  // I/O Port Declaration
//
input overflow;
input sat_0MaxNbr_1MaxMod;
//
input  [(WIDTH_N-1):0] inp_value;
output [(WIDTH_N-1):0] out_value;

  // Port Wire Declaration
//
wire overflow;
wire sat_0MaxNbr_1MaxMod;
//
wire [(WIDTH_N-1):0] inp_value;
wire [(WIDTH_N-1):0] out_value;


  // Concurrent Assignment

    // Module Instantiation
//
equ_saturador_complemento_2_N
  saturador_complemento_2
    ( .overflow(overflow),
      .sat_0MaxNbr_1MaxMod(sat_0MaxNbr_1MaxMod),
      .inp_value(inp_value),
      .out_value(out_value) );
//
defparam saturador_complemento_2.WIDTH_N = WIDTH_N;
//

endmodule //  equ_saturador_complemento_2_N5

//
// EOF
//
```

Parte IV

Circuitos Sequenciais: Construções básicas

Capítulo 14

Elemento básico de armazenamento: *flip-flop*

14.1 Introdução

Flip-flop é uma designação genérica para um circuito digital capaz de armazenar um *bit*. Portanto, um *flip-flop* pode ser chamado de Elemento Básico de Armazenamento (EBA).

Nesse capítulo, são apresentados códigos para alguns tipos de *flip-flops*.

Será usada a seguinte classificação para os *flip-flops*: *unclocked* (*latch*) e *clocked* (elementar, *master-slave* e *edge-triggered*). O *flip-flop* do tipo *clocked* elementar é um *latch* com controle de sincronismo temporal (*clock*). Os outros dois tipos são estruturas que se utilizam do *latch* como bloco fundamental.

14.2 *Flip-flop unclocked*: *latch* SR

- A Listagem 1 apresenta um *latch* do tipo SR, com saídas complementares, sensível a nível alto, baseado em NOR.
- A Listagem 2 apresenta um *latch* do tipo SR, com saídas complementares, sensível a nível alto, baseado em NAND.

Listagem 1 - *Latch* SR, baseado em NOR:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch SR
//           NOR based
//           No CTRL input
//           No CLR/PRE input
// Filename: op_sr_nor.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_sr_nor (
```

```
   q, // output    q
qbar, // output   ~q
   s, // input   set
   r  // input reset
);

  // I/O Port Declaration
output    q;
output qbar;
input     s;
input     r;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    s;
wire    r;

  // Concurrent Assignment

    // Primitive Instantiation

      // two nor gates and their interconnections
nor nor_q     (   q, qbar, r),
    nor_q_bar (qbar,    q, s);

endmodule // op_sr_nor

//
// EOF
//
```

Listagem 2 - *Latch* SR, baseado em NAND:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch SR
//           NAND based
//           No CTRL input
//           No CLR/PRE input
// Filename: op_sr_nand.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module op_sr_nand (
   q, // output    q
qbar, // output   ~q
   s, // input   set
   r  // input reset
);

  // I/O Port Declaration
output    q;
output qbar;
input     s;
input     r;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    s;
wire    r;

  // Concurrent Assignment

    // Primitive Instantiation

      // two nand gates and their interconnections
nand nand_q     (   q, qbar, ~s),
     nand_q_bar (qbar,    q, ~r);

endmodule // op_sr_nand

//
// EOF
//
```

14.3 *Flip-flop clocked* elementar: *latch* SR *clocked*

- A Listagem 3 apresenta um *latch* do tipo SR *clocked*, com saídas complementares, sensível a nível alto, baseado em NOR.
- A Listagem 4 apresenta um *latch* do tipo SR *clocked*, com saídas complementares, sensível a nível alto, baseado em NAND.

Listagem 3 - *Latch* SR *clocked*, baseado em NOR:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch SR
//           NOR based
//           CTRL input
//           No CLR/PRE input
// Filename: op_sr_c_nor.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_sr_c_nor (
   q, // output    q
qbar, // output   ~q
   s, // input   set
   r, // input reset
   c  // input  ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     s;
input     r;
input     c;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    s;
wire    r;
wire    c;

  // Internal Wire Declaration
wire wire_s;
wire wire_r;

  // Concurrent Assignment
```

```
    // Primitive Instantiation

      // two and gates and their interconnections
and and_s (wire_s, c, s),
    and_r (wire_r, c, r);

      // two nor gates and their interconnections
nor nor_q     (   q, qbar, wire_r),
    nor_q_bar (qbar,    q, wire_s);

endmodule // op_sr_c_nor

//
// EOF
//
```

Listagem 4 - *Latch* SR *clocked*, baseado em NAND:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch SR
//           NAND based
//           CTRL input
//           No CLR/PRE input
// Filename: op_sr_c_nand.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_sr_c_nand (
   q, // output    q
qbar, // output   ~q
   s, // input   set
   r, // input reset
   c  // input  ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     s;
input     r;
input     c;
```

```
  // Port Wire Declaration
wire    q;
wire qbar;
wire    s;
wire    r;
wire    c;

  // Internal Wire Declaration
wire wire_s;
wire wire_r;

  // Concurrent Assignment

    // Primitive Instantiation

      // two and gates and their interconnections
and and_s (wire_s, c, s),
    and_r (wire_r, c, r);

      // two nand gates and their interconnections
nand nand_q     (   q, qbar, ~wire_s),
     nand_q_bar (qbar,    q, ~wire_r);

endmodule // op_sr_c_nand

//
// EOF
//
```

14.4 *Flip-flop clocked* elementar: *latch* D (*clocked*)

- A Listagem 5 apresenta um *latch* do tipo D (*clocked*), com saídas complementares, sensível a nível alto, baseado no *latch* SR *clocked* da Listagem 3.
- A Listagem 6 apresenta um *latch* do tipo D (*clocked*), com saídas complementares, sensível a nível alto, baseado no *latch* SR *clocked* da Listagem 4.

Listagem 5 - *Latch* D (*clocked*), baseado no *Latch* SR *clocked* NOR:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch D
//           Latch (nor) SR based
//           CTRL input
//           No CLR/PRE input
// Filename: estrut_sr_c_nor_to_d.v
```

```
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module estrut_sr_c_nor_to_d (
   q, // output    q
qbar, // output   ~q
   d, // input     d
   c  // input  ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;

  // Concurrent Assignment

    // Module Instantiation

      // instantiation of a nor latch SR circuit
op_sr_c_nor sr_c_nor_kernel ( .q(q), .qbar(qbar),
                              .s(d), .r(~d)     , .c(c) );

endmodule // estrut_sr_c_nor_to_d

//
// EOF
//
```

Listagem 6 - *Latch* D (*clocked*), baseado no *Latch* SR *clocked* NAND:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Latch D
//           Latch (nand) SR based
//           CTRL input
//           No CLR/PRE input
```

```
// Filename: estrut_sr_c_nand_to_d.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module estrut_sr_c_nand_to_d (
   q, // output    q
qbar, // output   ~q
   d, // input     d
   c  // input  ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;

  // Concurrent Assignment

    // Module Instantiation

      // instantiation of a nand latch SR circuit
op_sr_c_nand sr_c_nand_kernel ( .q(q), .qbar(qbar),
                                .s(d), .r(~d)     , .c(c) );

endmodule // estrut_sr_c_nand_to_d

//
// EOF
//
```

14.5 *Flip-flop* com estrutura *master-slave*

- A Listagem 7 apresenta um *flip-flop* com estrutura *master-slave*, do tipo D, com saídas complementares, sensível à borda negativa (*master-high*, *slave-low*), com *clock* aplicado a ambos os estágios, sem controle de *CLEAR/PRESET*, baseado no *latch* SR *clocked* da Listagem 4.

- A Listagem 8 apresenta um *flip-flop* com estrutura *master-slave*, do tipo JK, com saídas complementares, sensível à borda negativa (*master-high*, *slave-low*), com *clock* aplicado a ambos os estágios, possuindo controle de *CLEAR/PRESET*, ambos ativos em nível baixo, baseado em NAND.

- A Listagem 9 apresenta um *flip-flop* com estrutura *master-slave*, do tipo JK, com saídas complementares, sensível à borda negativa (*master-high*, *slave-low*), com *clock* aplicado apenas ao *master*, possuindo controle de *CLEAR/PRESET*, ambos ativos em nível baixo, baseado em NAND.

Listagem 7 - *Flip-flop* D com estrutura *master-slave*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Master-Slave D flip-flop
//           Latch (nand) SR based
//           CTRL input: Master-High, Slave-Low
//           No CLR/PRE input
// Filename: estrut_sr_c_nand_to_ms_d.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module estrut_sr_c_nand_to_ms_d (
   q, // output    q
qbar, // output   ~q
   d, // input     d
   c  // input  ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;
```

```
  // Internal Wire Declaration
wire wire_q_m;
wire wire_qbar_m;

  // Concurrent Assignment

    // Module Instantiation

      // instantiation of a nand latch SR circuit
op_sr_c_nand sr_c_nand_master ( .q(wire_q_m), .qbar(wire_qbar_m),
                                .s(d)       , .r(~d)            ,
                                .c( c)                            );

      // instantiation of a nand latch SR circuit
op_sr_c_nand sr_c_nand_slave  ( .q(q)       , .qbar(qbar)       ,
                                .s(wire_q_m), .r(wire_qbar_m)   ,
                                .c(~c)                            );

endmodule // estrut_sr_c_nand_to_ms_d

//
// EOF
//
```

Listagem 8 - *Flip-flop* JK com estrutura *master-slave* e *clock* nos dois estágios:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Master-Slave JK flip-flop
//           All-NAND based
//           CTRL input: Master-High, Slave-Low
//             (into both stages)
//           CLR/PRE input (low active)
// Filename: op_ms_jk_all_nand_cm_cs.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_ms_jk_all_nand_cm_cs (
   q, // output         q
qbar, // output        ~q
   j, // input          j
   k, // input          k
   c, // input       ctrl
```

```
clrb, // input  clear_bar
preb  // input preset_bar
);

  // I/O Port Declaration
output    q;
output qbar;
input     j;
input     k;
input     c;
input  clrb;
input  preb;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    j;
wire    k;
wire    c;
wire clrb;
wire preb;

  // Port Wire Declaration

    // master wires
wire wire_sbar_mst;
wire wire_rbar_mst;

wire wire_q_mst;
wire wire_qbar_mst;

    // slave wires
wire wire_sbar_slv;
wire wire_rbar_slv;

wire wire_q_slv;
wire wire_qbar_slv;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

        // master
nand nand_s_mst (wire_sbar_mst,  c, j, wire_qbar_slv, clrb),
     nand_r_mst (wire_rbar_mst,  c, k, wire_q_slv   , preb);

nand nand_q_mst     (wire_q_mst   , wire_qbar_mst, wire_sbar_mst, preb),
```

```
     nand_q_bar_mst (wire_qbar_mst, wire_q_mst   , wire_rbar_mst, clrb);

        // slave
nand nand_s_slv (wire_sbar_slv, ~c, wire_q_mst   , clrb),
     nand_r_slv (wire_rbar_slv, ~c, wire_qbar_mst, preb);

nand nand_q_slv     (wire_q_slv   , wire_qbar_slv, wire_sbar_slv, preb),
     nand_q_bar_slv (wire_qbar_slv, wire_q_slv   , wire_rbar_slv, clrb);

    // Continuous Assignment Statement

      // output connection
assign q    = wire_q_slv,
       qbar = wire_qbar_slv;

endmodule // op_ms_jk_all_nand_cm_cs

//
// EOF
//
```

Listagem 9 - *Flip-flop* JK com estrutura *master-slave* e *clock* apenas no *master*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Master-Slave JK flip-flop
//           All-NAND based
//           CTRL input: Master-High, Slave-Low
//             (into master stage only)
//           CLR/PRE input (low active)
// Filename: op_ms_jk_all_nand_cm.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_ms_jk_all_nand_cm (
   q, // output         q
qbar, // output        ~q
   j, // input          j
   k, // input          k
   c, // input       ctrl
clrb, // input  clear_bar
preb  // input preset_bar
);
```

```
  // I/O Port Declaration
output    q;
output qbar;
input     j;
input     k;
input     c;
input  clrb;
input  preb;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    j;
wire    k;
wire    c;
wire clrb;
wire preb;

  // Port Wire Declaration

    // master wires
wire wire_sbar_mst;
wire wire_rbar_mst;

wire wire_q_mst;
wire wire_qbar_mst;

    // slave wires
wire wire_sbar_slv;
wire wire_rbar_slv;

wire wire_q_slv;
wire wire_qbar_slv;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

        // master
nand nand_s_mst (wire_sbar_mst,  c, j, wire_qbar_slv, clrb),
     nand_r_mst (wire_rbar_mst,  c, k, wire_q_slv   , preb);

nand nand_q_mst     (wire_q_mst   , wire_qbar_mst, wire_sbar_mst, preb),
     nand_q_bar_mst (wire_qbar_mst, wire_q_mst   , wire_rbar_mst, clrb);

        // slave
nand nand_s_slv (wire_sbar_slv, wire_sbar_mst, wire_q_mst   , clrb),
```

```
     nand_r_slv (wire_rbar_slv, wire_rbar_mst, wire_qbar_mst, preb);

nand nand_q_slv     (wire_q_slv   , wire_qbar_slv, wire_sbar_slv, preb),
     nand_q_bar_slv (wire_qbar_slv, wire_q_slv   , wire_rbar_slv, clrb);

    // Continuous Assignment Statement

      // output connection
assign q    = wire_q_slv,
       qbar = wire_qbar_slv;

endmodule // op_ms_jk_all_nand_cm

//
// EOF
//
```

14.6 *Flip-flop edge-triggered* (estrutura realimentada)

- A Listagem 10 apresenta um *flip-flop edge-triggered*, com estrutura realimentada, do tipo D, com saídas complementares, sensível à transição positiva, baseado em NAND, retirado de [HP81], sem o controle de *CLEAR/PRESET*.
- A Listagem 11 apresenta um *flip-flop edge-triggered*, com estrutura realimentada, do tipo D, com saídas complementares, sensível à transição positiva, baseado em NAND, retirado de [HP81], com o controle de *CLEAR/PRESET*.
- A Listagem 12 apresenta um *flip-flop edge-triggered*, com estrutura realimentada, do tipo D, com saídas complementares, sensível à transição positiva, baseado em NAND, retirado de [HP81], com o controle de *CLEAR/PRESET* modificado pelo autor do código.
- A Listagem 13 apresenta um *flip-flop edge-triggered*, com estrutura realimentada, do tipo JK, com saídas complementares, sensível à transição positiva, baseado em NAND, retirado de [HP81], modificado pelo autor do código.

Listagem 10 - *Flip-flop* D *edge-triggered*, com estrutura realimentada, sem CLR/PRE:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Feedback D flip-flop
//           All-NAND based
//           CTRL input
//           No CLR/PRE input
//           Get from Hill & Peterson
// Filename: op_fb_d_all_nand_HP.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
```

```
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_fb_d_all_nand_HP (
   q, // output         q
qbar, // output        ~q
   d, // input          d
   c  // input       ctrl
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;

  // Internal Wire Declaration

    // D wires
wire wire_q_d_sr;
wire wire_qbar_d_sr;

    // C wires
wire wire_q_c_sr;
wire wire_qbar_c_sr;

    // output wires
wire wire_q_out_sr;
wire wire_qbar_out_sr;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

        // D SR
nand nand_q_d       (wire_q_d_sr   , wire_qbar_d_sr,  d               ),
     nand_q_bar_d   (wire_qbar_d_sr, wire_q_d_sr   ,  c, wire_qbar_c_sr);
        // C SR
nand nand_q_c       (wire_q_c_sr   , wire_qbar_c_sr, wire_q_d_sr),
     nand_q_bar_c   (wire_qbar_c_sr, wire_q_c_sr   ,           c);
```

```
        // output SR
nand nand_q_out     (wire_q_out_sr   , wire_qbar_out_sr, wire_qbar_c_sr),
     nand_q_bar_out (wire_qbar_out_sr, wire_q_out_sr   , wire_qbar_d_sr);

    // Continuous Assignment Statement

      // output connection
assign q    = wire_q_out_sr,
       qbar = wire_qbar_out_sr;

endmodule // op_fb_d_all_nand_HP

//
// EOF
//
```

Listagem 11 - *Flip-flop* D *edge-triggered*, com estrutura realimentada, com CLR/PRE:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Feedback D flip-flop
//           All-NAND based
//           CTRL input
//           CLR/PRE (low active)
//           Get from Hill & Peterson
// Filename: op_fb_d_all_nand_cp_HP.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_fb_d_all_nand_cp_HP (
   q, // output         q
qbar, // output        ~q
   d, // input          d
   c, // input       ctrl
clrb, // input  clear_bar
preb  // input preset_bar
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;
input  clrb;
```

```
input  preb;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;
wire clrb;
wire preb;

  // Internal Wire Declaration

    // D wires
wire wire_q_d_sr;
wire wire_qbar_d_sr;

    // C wires
wire wire_q_c_sr;
wire wire_qbar_c_sr;

    // output wires
wire wire_q_out_sr;
wire wire_qbar_out_sr;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

        // D SR
nand nand_q_d       (wire_q_d_sr   , wire_qbar_d_sr,  d, clrb          ),
     nand_q_bar_d   (wire_qbar_d_sr, wire_q_d_sr   ,  c, wire_qbar_c_sr);

        // C SR
nand nand_q_c       (wire_q_c_sr   , wire_qbar_c_sr, wire_q_d_sr, preb),
     nand_q_bar_c   (wire_qbar_c_sr, wire_q_c_sr   ,           c, clrb);

        // output SR
nand nand_q_out     (wire_q_out_sr   , wire_qbar_out_sr, wire_qbar_c_sr,
                     preb),
     nand_q_bar_out (wire_qbar_out_sr, wire_q_out_sr   , wire_qbar_d_sr,
                     clrb);

    // Continuous Assignment Statement

      // output connection
assign q    = wire_q_out_sr,
       qbar = wire_qbar_out_sr;
```

```
endmodule // op_fb_d_all_nand_cp_HP

//
// EOF
//
```

Listagem 12 - *Flip-flop* D *edge-triggered*, estrutura realimentada, CLR/PRE modificado:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Feedback D flip-flop
//           All-NAND based
//           CTRL input
//           No CLR/PRE input
//           Get from Hill & Peterson
//           CLR/PRE (low active) by ASV
// Filename: op_fb_d_all_nand_cp_ASV.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_fb_d_all_nand_cp_ASV (
   q, // output         q
qbar, // output        ~q
   d, // input          d
   c, // input       ctrl
clrb, // input  clear_bar
preb  // input preset_bar
);

  // I/O Port Declaration
output    q;
output qbar;
input     d;
input     c;
input  clrb;
input  preb;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    d;
wire    c;
wire clrb;
```

```
wire preb;

  // Internal Wire Declaration

    // D wires
wire wire_q_d_sr;
wire wire_qbar_d_sr;

    // C wires
wire wire_q_c_sr;
wire wire_qbar_c_sr;

    // output wires
wire wire_q_out_sr;
wire wire_qbar_out_sr;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

        // D SR
nand nand_q_d       (wire_q_d_sr   , wire_qbar_d_sr,  d),
     nand_q_bar_d   (wire_qbar_d_sr, wire_q_d_sr   ,  c, wire_qbar_c_sr,
                     preb);

        // C SR
nand nand_q_c       (wire_q_c_sr   , wire_qbar_c_sr, wire_q_d_sr      ),
     nand_q_bar_c   (wire_qbar_c_sr, wire_q_c_sr   ,           c, clrb);

        // output SR
nand nand_q_out     (wire_q_out_sr   , wire_qbar_out_sr, wire_qbar_c_sr,
                     preb),
     nand_q_bar_out (wire_qbar_out_sr, wire_q_out_sr   , wire_qbar_d_sr,
                     clrb);

    // Continuous Assignment Statement

      // output connection
assign q    = wire_q_out_sr,
       qbar = wire_qbar_out_sr;

endmodule // op_fb_d_all_nand_cp_ASV

//
// EOF
//
```

Listagem 13 - *Flip-flop* JK *edge-triggered*, estrutura realimentada, modificado:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Feedback JK flip-flop
//           CTRL input
//           Kbar input
//           CLR/PRE (high active)
//           SN74109
//           Get from Hill & Peterson
//           K instead Kbar     by ASV
//           CLR/PRE low active by ASV
//           AOI all-NAND based by ASV
// Filename: op_fb_jk_cp_HP_SN74109_ASV.v
// Coder   : Alexandre Santos de la Vega
// Versions: /mar_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module op_fb_jk_cp_HP_SN74109_ASV (
   q, // output         q
qbar, // output        ~q
   j, // input          d
   k, // input          d
   c, // input       ctrl
clrb, // input  clear_bar
preb  // input preset_bar
);

  // I/O Port Declaration
output    q;
output qbar;
input     j;
input     k;
input     c;
input  clrb;
input  preb;

  // Port Wire Declaration
wire    q;
wire qbar;
wire    j;
wire    k;
wire    c;
wire clrb;
wire preb;

  // Internal Wire Declaration
```

```
    // JK wires
wire wire_nand_j;
wire wire_nand_k;
wire wire_aoi_jk;

    // PRE wires
wire wire_nand_pre;

    // Kernel wires
wire wire_nand_s;
wire wire_nand_r;

    // output wires
wire wire_q_out_sr;
wire wire_qbar_out_sr;

  // Concurrent Assignment

    // Primitive Instantiation

      // instantiation of nand gates

  //
  // AOI by NANDs  = Nand_Nand_Inv
  //
  // Nand_Nand_Inv = Nand_Nand
  //                 +
  //                 inverters at the inputs of the receiving blocks
  //
nand nand_j   (wire_nand_j,  j, wire_qbar_out_sr, clrb, wire_nand_r),
     nand_k   (wire_nand_k, ~k, wire_q_out_sr   , clrb, wire_nand_r),
     nand_aoi (wire_aoi_jk, wire_nand_j, wire_nand_k);

        // nand PRE
nand nand_pre (wire_nand_pre, wire_nand_s, preb, ~wire_aoi_jk);

        // input SR
nand nand_s (wire_nand_s, wire_nand_pre, c,         clrb),
     nand_r (wire_nand_r, wire_nand_s  , c, ~wire_aoi_jk);

        // output SR
nand nand_q_out     (wire_q_out_sr   , wire_qbar_out_sr, wire_nand_s,
                     preb),
     nand_q_bar_out (wire_qbar_out_sr, wire_q_out_sr   , wire_nand_r,
                     clrb);

    // Continuous Assignment Statement
```

```
      // output connection
assign q    = wire_q_out_sr,
       qbar = wire_qbar_out_sr;

endmodule // op_fb_jk_cp_HP_SN74109_ASV

//
// EOF
//
```

Parte V

Circuitos Sequenciais: Blocos funcionais

Capítulo 15

Exemplos de blocos funcionais

Nessa parte, são apresentados exemplos de circuitos digitais que implementam funções comumente encontradas em sistemas digitais.

A seguir, são abordados os seguintes itens:

- Elementos de armazenamento: registradores.
- Divisores de freqüência.
- Máquinas de Estados Finitos (*Finite-State Machines*) simples.

Capítulo 16

Elementos de armazenamento: registradores

16.1 Introdução

Registrador é uma designação genérica para um circuito digital capaz de armazenar um conjunto de N *bits*.

Nesse capítulo, são apresentados códigos para alguns tipos de registradores.

16.2 Tipos básicos de registradores

- As entradas e as saídas dos registradores podem ser dos seguintes tipos:
  - Serial.
  - Paralela.
- De acordo com o tipo da sua entrada e da sua saída, um registrador pode ser classificado da seguinte forma:
  - SISO (*Serial-Input Serial-Output*).
  - SIPO (*Serial-Input Parallel-Output*).
  - PISO (*Parallel-Input Serial-Output*).
  - PIPO (*Parallel-Input Parallel-Output*).
- Para os registradores que permitem deslocamento dos dados, este pode ser de dois tipos:
  - Deslocamento para a esquerda (*shift left*)
  - Deslocamento para a direita (*shift right*).

16.3 Registrador SISO (*Serial-Input Serial-Output*)

- A Listagem 1 apresenta um registrador do tipo SISO (*Serial-Input Serial-Output*), com deslocamento para a direita, usando modelo comportamental, parametrizado com 4 *bits*.
- A Listagem 2 apresenta um registrador do tipo SISO (*Serial-Input Serial-Output*), com deslocamento para a direita, usando modelo estrutural hierárquico, que emprega o registrador da Listagem 1, parametrizado com N=6 *bits*.

Listagem 1 - Registrador SISO, *shift right*, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           SISO (Serial-In Serial-Out)
//           Shift-right
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_siso_shr_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_siso_shr_for_param_width_4 (
reg_si, // input   serial
   clk, // control clock
reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input  reg_si;
input     clk;
output reg_so;

  // Port Wire Declaration
wire reg_si;
wire    clk;
wire reg_so;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;

  // Internal Var Declaration
integer k;
```

```
  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  // shift_right: from LSB up to MSB
  for (k = 0; k < (WIDTH_4-1); k = k + 1)
    begin
      internal_data[k] = internal_data[k+1];
    end
  //
  // load serial_in at MSB
  internal_data[(WIDTH_4-1)] = reg_si;
  //
end

// load serial_out from LSB
assign reg_so = internal_data[0];
//

endmodule // reg_siso_shr_for_param_width_4

//
// EOF
//
```

Listagem 2 - Registrador SISO, *shift right*, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           SISO (Serial-In Serial-Out)
//           Shift-right
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: reg_siso_shr_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_siso_shr_defparam_width_N (
reg_si, // input   serial
   clk, // control clock
```

```
reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input  reg_si;
input     clk;
output reg_so;

  // Port Wire Declaration
wire reg_si;
wire    clk;
wire reg_so;


  // Concurrent Assignment

    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
reg_siso_shr_for_param_width_4 reg_siso_width_4 ( .reg_si(reg_si),
                                                  .clk(clk),
                                                  .reg_so(reg_so) );
defparam reg_siso_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // reg_siso_shr_defparam_width_N

//
// EOF
//
```

16.4 Registrador SIPO (*Serial-Input Parallel-Output*)

- A Listagem 3 apresenta um registrador do tipo SIPO (*Serial-Input Parallel-Output*), com deslocamento para a direita, usando modelo comportamental, parametrizado com 4 *bits*.
- A Listagem 4 apresenta um registrador do tipo SIPO (*Serial-Input Parallel-Output*), com deslocamento para a direita, usando modelo estrutural hierárquico, que emprega o registrador da Listagem 3, parametrizado com N=6 *bits*.

Listagem 3 - Registrador SIPO, *shift right*, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           SIPO (Serial-In Parallel-Out)
//           Shift-right
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_sipo_shr_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_sipo_shr_for_param_width_4 (
reg_si, // input   serial
   clk, // control clock
reg_po  // output  parallel
);

  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input                  reg_si;
input                     clk;
output [(WIDTH_4-1):0] reg_po;

  // Port Wire Declaration
wire                 reg_si;
wire                    clk;
wire [(WIDTH_4-1):0] reg_po;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;

  // Internal Var Declaration
integer k;
```

```
  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  // shift_right: from LSB up to MSB
  for (k = 0; k < (WIDTH_4-1); k = k + 1)
    begin
      internal_data[k] = internal_data[k+1];
    end
  //
  // load serial_in at MSB
  internal_data[(WIDTH_4-1)] = reg_si;
  //
end

// load parallel_out
assign reg_po = internal_data;
//

endmodule // reg_sipo_shr_for_param_width_4

//
// EOF
//
```

Listagem 4 - Registrador SIPO, *shift right*, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           SIPO (Serial-In Parallel-Out)
//           Shift-right
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: reg_sipo_shr_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_sipo_shr_defparam_width_N (
reg_si, // input   serial
   clk, // control clock
```

```
reg_po  // output  parallel
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input                  reg_si;
input                     clk;
output [(WIDTH_N-1):0] reg_po;

  // Port Wire Declaration
wire                 reg_si;
wire                    clk;
wire [(WIDTH_N-1):0] reg_po;


  // Concurrent Assignment

    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
reg_sipo_shr_for_param_width_4 reg_sipo_width_4 ( .reg_si(reg_si),
                                                  .clk(clk),
                                                  .reg_po(reg_po) );
defparam reg_sipo_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // reg_sipo_shr_defparam_width_N

//
// EOF
//
```

16.5 Registrador PISO (*Parallel-Input Serial-Output*)

- A Listagem 5 apresenta um registrador do tipo PISO (*Parallel-Input Serial-Output*), com deslocamento para a esquerda, usando modelo comportamental, parametrizado com 4 *bits*.

- A Listagem 6 apresenta um registrador do tipo PISO (*Parallel-Input Serial-Output*), com deslocamento para a esquerda, usando modelo estrutural hierárquico, que emprega o registrador da Listagem 5, parametrizado com N=6 *bits*.

- A Listagem 7 apresenta um registrador do tipo PISO (*Parallel-Input Serial-Output*), com deslocamento para a direita, usando modelo comportamental, parametrizado com 4 *bits*.

- A Listagem 8 apresenta um registrador do tipo PISO (*Parallel-Input Serial-Output*), com deslocamento para a direita, usando modelo estrutural hierárquico, que emprega o registrador da Listagem 7, parametrizado com N=6 *bits*.

Listagem 5 - Registrador PISO, *shift left*, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PISO (Parallel-In Serial-Out)
//           Shift-left
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_piso_shl_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_piso_shl_for_param_width_4 (
      reg_pi, // input   parallel
      reg_si, // input   serial
load0_shift1, // control load=0 shift=1
         clk, // control clock
      reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input  [(WIDTH_4-1):0]       reg_pi;
input                        reg_si;
input                  load0_shift1;
input                           clk;
output                       reg_so;
```

```
  // Port Wire Declaration
wire [(WIDTH_4-1):0]       reg_pi;
wire                       reg_si;
wire                 load0_shift1;
wire                          clk;
wire                       reg_so;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;

  // Internal Var Declaration
integer k;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  if (load0_shift1 == 0)
    //
    // load parallel_in
    internal_data = reg_pi;
    //
  else begin
    //
    // shift_left: from MSB up to LSB
    for (k = (WIDTH_4-1); k > 0; k = k - 1)
      begin
        internal_data[k] = internal_data[k-1];
      end
    //
    // load serial_in at LSB
    internal_data[0] = reg_si;
    //
  end
end

// load serial_out from MSB
assign reg_so = internal_data[(WIDTH_4-1)];
//

endmodule // reg_piso_shl_for_param_width_4

//
// EOF
//
```

Listagem 6 - Registrador PISO, *shift left*, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PISO (Parallel-In Serial-Out)
//           Shift-left
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: reg_piso_shl_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_piso_shl_defparam_width_N (
      reg_pi, // input   parallel
      reg_si, // input   serial
load0_shift1, // control load=0 shift=1
         clk, // control clock
      reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input  [(WIDTH_N-1):0]       reg_pi;
input                        reg_si;
input                  load0_shift1;
input                           clk;
output                       reg_so;

  // Port Wire Declaration
wire [(WIDTH_N-1):0]       reg_pi;
wire                       reg_si;
wire                 load0_shift1;
wire                          clk;
wire                       reg_so;


  // Concurrent Assignment

    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
```

```
//
// submodule #(N) instance_name_W (.);
//

//
reg_piso_shl_for_param_width_4
          reg_piso_shl_width_4 ( .reg_pi(reg_pi),
                                 .reg_si(reg_si),
                                 .load0_shift1(load0_shift1),
                                 .clk(clk),
                                 .reg_so(reg_so) );
//
defparam reg_piso_shl_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // reg_piso_shl_defparam_width_N

//
// EOF
//
```

Listagem 7 - Registrador PISO, *shift right*, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PISO (Parallel-In Serial-Out)
//           Shift-right
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_piso_shr_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_piso_shr_for_param_width_4 (
      reg_pi, // input   parallel
      reg_si, // input   serial
load0_shift1, // control load=0 shift=1
         clk, // control clock
      reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_4 = 4;
```

```
  // I/O Port Declaration
input  [(WIDTH_4-1):0]       reg_pi;
input                        reg_si;
input                  load0_shift1;
input                           clk;
output                       reg_so;

  // Port Wire Declaration
wire [(WIDTH_4-1):0]       reg_pi;
wire                       reg_si;
wire                 load0_shift1;
wire                          clk;
wire                       reg_so;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;

  // Internal Var Declaration
integer k;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  if (load0_shift1 == 0)
    //
    // load parallel_in
    internal_data = reg_pi;
    //
  else begin
    //
    // shift_right: from LSB up to MSB
    for (k = 0; k < (WIDTH_4-1); k = k + 1)
      begin
        internal_data[k] = internal_data[k+1];
      end
    //
    // load serial_in at MSB
    internal_data[(WIDTH_4-1)] = reg_si;
    //
  end
end

// load serial_out from LSB
assign reg_so = internal_data[0];
```

```
//

endmodule // reg_piso_shr_for_param_width_4

//
// EOF
//
```

Listagem 8 - Registrador PISO, *shift right*, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PISO (Parallel-In Serial-Out)
//           Shift-right
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: reg_piso_shr_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_piso_shr_defparam_width_N (
      reg_pi, // input   parallel
      reg_si, // input   serial
load0_shift1, // control load=0 shift=1
         clk, // control clock
      reg_so  // output  serial
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input  [(WIDTH_N-1):0]       reg_pi;
input                        reg_si;
input                  load0_shift1;
input                           clk;
output                       reg_so;

  // Port Wire Declaration
wire [(WIDTH_N-1):0]       reg_pi;
wire                       reg_si;
wire                 load0_shift1;
wire                          clk;
```

```
wire                        reg_so;


  // Concurrent Assignment

    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
reg_piso_shr_for_param_width_4
          reg_piso_shr_width_4 ( .reg_pi(reg_pi),
                                 .reg_si(reg_si),
                                 .load0_shift1(load0_shift1),
                                 .clk(clk),
                                 .reg_so(reg_so) );
//
defparam reg_piso_shr_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // reg_piso_shr_defparam_width_N

//
// EOF
//
```

16.6 Registrador PIPO (*Parallel-Input Parallel-Output*)

- A Listagem 9 apresenta um registrador do tipo PIPO (*Parallel-Input Parallel-Output*), sem deslocamento, usando modelo comportamental, parametrizado com 4 *bits*.
- A Listagem 10 apresenta um registrador do tipo PIPO (*Parallel-Input Parallel-Output*), sem deslocamento, usando modelo estrutural hierárquico, usando o registrador da Listagem 9, parametrizado com N=6 *bits*.
- A Listagem 11 apresenta um registrador do tipo PIPO (*Parallel-Input Parallel-Output*), com deslocamento programável (*shift left/right*), usando modelo comportamental, parametrizado com 4 *bits*.
- A Listagem 12 apresenta um registrador do tipo PIPO (*Parallel-Input Parallel-Output*), com deslocamento programável (*shift left/right*), usando modelo estrutural hierárquico, que emprega o registrador da Listagem 11, parametrizado com N=6 *bits*.

Listagem 9 - Registrador PIPO, sem deslocamento, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PIPO (Parallel-In Parallel-Out)
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_pipo_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_pipo_for_param_width_4 (
reg_pi, // input   parallel
   clk, // control clock
reg_po  // output  parallel
);

  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input  [(WIDTH_4-1):0] reg_pi;
input                  clk;
output [(WIDTH_4-1):0] reg_po;

  // Port Wire Declaration
wire [(WIDTH_4-1):0] reg_pi;
wire                 clk;
wire [(WIDTH_4-1):0] reg_po;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  // load parallel_in
  internal_data = reg_pi;
  //
end
```

```
// load parallel_out
assign reg_po = internal_data;
//

endmodule // reg_pipo_for_param_width_4

//
// EOF
//
```

Listagem 10 - Registrador PIPO, sem deslocamento, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PIPO (Parallel-In Parallel-Out)
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: reg_pipo_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_pipo_defparam_width_N (
reg_pi, // input   parallel
   clk, // control clock
reg_po  // output  parallel
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input  [(WIDTH_N-1):0] reg_pi;
input                  clk;
output [(WIDTH_N-1):0] reg_po;

  // Port Wire Declaration
wire [(WIDTH_N-1):0] reg_pi;
wire                 clk;
wire [(WIDTH_N-1):0] reg_po;


  // Concurrent Assignment
```

```
    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
reg_pipo_for_param_width_4 reg_pipo_width_4 ( .reg_pi(reg_pi),
                                              .clk(clk),
                                              .reg_po(reg_po) );
defparam reg_pipo_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // reg_pipo_defparam_width_N

//
// EOF
//
```

Listagem 11 - Registrador PIPO, *shift left/right*, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Register
//           PIPO (Parallel-In Parallel-Out)
//           Shift: left - right
//           Model: always - for - assign
//           Width: 4 by param
// Filename: reg_pipo_shl_shr_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module reg_pipo_shl_shr_for_param_width_4 (
      reg_pi, // input   parallel
  reg_si_msb, // input   serial MSB
  reg_si_lsb, // input   serial LSB
load0_shift1, // input   load=0 shift=1
   shl0_shr1, // input   shift_left=0 shift_right=1
         clk, // control clock
      reg_po  // output  parallel
);
```

```
  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input  [(WIDTH_4-1):0]       reg_pi;
input                     reg_si_msb;
input                     reg_si_lsb;
input                   load0_shift1;
input                      shl0_shr1;
input                           clk;
output [(WIDTH_4-1):0]       reg_po;

  // Port Wire Declaration
wire [(WIDTH_4-1):0]       reg_pi;
wire                     reg_si_msb;
wire                     reg_si_lsb;
wire                   load0_shift1;
wire                      shl0_shr1;
wire                           clk;
wire [(WIDTH_4-1):0]       reg_po;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] internal_data;

  // Internal Var Declaration
integer k;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk) begin
  //
  if (load0_shift1 == 0)
    //
    // load parallel_in
    internal_data = reg_pi;
    //
  else if (shl0_shr1 == 0) begin
         //
         // shift_left: from MSB up to LSB
         for (k = (WIDTH_4-1); k > 0; k = k - 1)
           begin
             internal_data[k] = internal_data[k-1];
           end
         //
         // load serial_in at LSB
```

```
          internal_data[0] = reg_si_lsb;
          //
        end
        else begin
          //
          // shift_right: from LSB up to MSB
          for (k = 0; k < (WIDTH_4-1); k = k + 1)
            begin
              internal_data[k] = internal_data[k+1];
            end
          //
          // load serial_in at MSB
          internal_data[(WIDTH_4-1)] = reg_si_msb;
          //
        end
    end

    // load parallel_out
    assign reg_po = internal_data;
    //

    endmodule // reg_pipo_shl_shr_for_param_width_4

    //
    // EOF
    //
```

Listagem 12 - Registrador PIPO, *shift left/right*, com N=6 *bits*:

```
    //
    // ----- ----- ----- ----- ----- ----- ----- -----
    // Design  : Register
    //           PIPO (Parallel-In Parallel-Out)
    //           Shift: left - right
    //           Model: hierarchy - defparam
    //           Width: N by defparam
    // Filename: reg_pipo_shl_shr_defparam_width_N.v
    // Coder   : Alexandre Santos de la Vega
    // Versions: /abr_2020/
    // ----- ----- ----- ----- ----- ----- ----- -----
    //

    // Module Declaration
    module reg_pipo_shl_shr_defparam_width_N (
          reg_pi, // input   parallel
      reg_si_msb, // input   serial MSB
      reg_si_lsb, // input   serial LSB
```

```
load0_shift1, // input   load=0 shift=1
   shl0_shr1, // input   shift_left=0 shift_right=1
         clk, // control clock
      reg_po  // output  parallel
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input  [(WIDTH_N-1):0]       reg_pi;
input                     reg_si_msb;
input                     reg_si_lsb;
input                   load0_shift1;
input                      shl0_shr1;
input                            clk;
output [(WIDTH_N-1):0]       reg_po;

  // Port Wire Declaration
wire [(WIDTH_N-1):0]       reg_pi;
wire                     reg_si_msb;
wire                     reg_si_lsb;
wire                   load0_shift1;
wire                      shl0_shr1;
wire                           clk;
wire [(WIDTH_N-1):0]       reg_po;



  // Concurrent Assignment

    // Module Instantiation

//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
reg_pipo_shl_shr_for_param_width_4
          reg_pipo_shl_shr_width_4 ( .reg_pi(reg_pi),
                                     .reg_si_msb(reg_si_msb),
                                     .reg_si_lsb(reg_si_lsb),
                                     .load0_shift1(load0_shift1),
                                     .shl0_shr1(shl0_shr1),
                                     .clk(clk),
                                     .reg_po(reg_po) );
```

```
    //
    defparam reg_pipo_shl_shr_width_4.WIDTH_4 = WIDTH_N;
    //

    endmodule // reg_pipo_shl_shr_defparam_width_N

    //
    // EOF
    //
```

Capítulo 17

Divisores de freqüência

17.1 Introdução

Divisor de freqüência é um circuito digital que recebe um sinal períodico com período T_0, ou freqüência $F_0 = 1/T_0$, e gera um ou mais sinais periódicos, com períodos $T_k = kT_0$, ou freqüências $F_k = 1/kT_0 = F_0/k$.

Nesse capítulo, são apresentados códigos para alguns divisores de freqüência.

17.2 Divisores de freqüência

- A Listagem 1 apresenta um divisor de freqüência, usando modelo comportamental, parametrizado com 4 *bits*. A descrição é fundamentada em um contador *ripple clock* (ou assíncrono), que utiliza apenas *flip-flops* do tipo T, sem entrada seletora.
- A Listagem 2 apresenta um divisor de freqüência, usando modelo estrutural hierárquico, que emprega o divisor da Listagem 1, parametrizado com N=6 *bits*.

Listagem 1 - Divisor de freqüência, baseado em contador assíncrono, com 4 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Frequency divider
//           Based on: ripple-clock counter
//           Model: always - for - assign
//           Width: 4 by param
// Filename: div_freq_rc_for_param_width_4.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module div_freq_rc_for_param_width_4 (
clk, // input  clk
  q  // output div clk vector
);
```

```
  // Parameter Declaration
parameter WIDTH_4 = 4;

  // I/O Port Declaration
input                   clk;
output [(WIDTH_4-1):0]   q;

  // Port Wire Declaration
wire                  clk;
wire [(WIDTH_4-1):0]   q;

  // Internal Reg Declaration
reg [(WIDTH_4-1):0] vector;
reg old_bit;

  // Internal Var Declaration
integer k;


  // Concurrent Assignment

    // Procedure by means of always
//
always @(posedge clk)
  begin
    //
    //  toggle bit 0, if clk...
    old_bit   =  vector[0];
    vector[0] = ~vector[0];
    //
    //  ripple the clk
    for (k = 1; k < (WIDTH_4); k = k + 1)
      begin
        //
        // toggle bit k, if bit (k-1)...
        if ( (old_bit==0) & (vector[k-1]==1) )
          begin
            old_bit   =  vector[k];
            vector[k] = ~vector[k];
          end
        else
          begin
            old_bit   =  vector[k];
            vector[k] =  vector[k];
          end
      end
    //
  end
//
```

```
// load parallel_out
assign q = vector;
//

endmodule // div_freq_rc_for_param_width_4

//
// EOF
//
```

Listagem 2 - Divisor de freqüência, baseado em contador assíncrono, com N=6 *bits*:

```
//
// ----- ----- ----- ----- ----- ----- -----
// Design  : Frequency divider
//           Based on: ripple-clock counter
//           Model: hierarchy - defparam
//           Width: N by defparam
// Filename: div_freq_rc_defparam_width_N.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
module div_freq_rc_defparam_width_N (
clk, // input  clk
  q  // output div clk vector
);

  // Parameter Declaration
parameter WIDTH_N = 6;

  // I/O Port Declaration
input                   clk;
output [(WIDTH_N-1):0]   q;

  // Port Wire Declaration
wire                  clk;
wire [(WIDTH_N-1):0]   q;



  // Concurrent Assignment

    // Module Instantiation
```

```
//
// Não funciona !!!
// Recomenda: defparam ...
//
// submodule #(N) instance_name_W (.);
//

//
div_freq_rc_for_param_width_4 div_freq_width_4 ( .clk(clk), .q(q) );
//
defparam div_freq_width_4.WIDTH_4 = WIDTH_N;
//

endmodule // div_freq_rc_defparam_width_N

//
// EOF
//
```

Capítulo 18

Máquinas de Estados Finitos simples

18.1 Introdução

Nesse capítulo, são apresentados códigos para algumas Máquinas de Estados Finitos (*Finite State Machines*) simples.

18.2 Máquinas de Estados Finitos simples

- A Listagem 1 apresenta um circuito digital sequencial, do tipo Mealy, que implementa um contador, com contagem crescente, de módulo 4, cujas saídas são codificadas em binário puro, que atende a um sinal de inicialização ativo em nível baixo ($\overline{RST}$). Os valores da contagem, em representação decimal, são os seguintes: z = (02/03, 04/07, 10/11, 16/19).

Listagem 1 - Contador, crescente, módulo 4, com *reset* (02/03, 04/07, 10/11, 16/19):

```
//
// ----- ----- ----- ----- ----- ----- ----- ----- -----
// Design  : Finite State Machine (FSM)
//             Input   : x
//             Output  : z
//             Control : clk, rstb
//             Type    : Mealy
//             Function: Counter
//               x = 0 --> {...,02,04,10,16,02,...}
//               x = 1 --> {...,03,07,11,19,03,...}
//               rstb  --> z = 02/03
//           Model: always - if - case
//           Widths by param
// Filename: fsm_counter_mealy_0203_0407_1011_1619.v
// Coder   : Alexandre Santos de la Vega
// Versions: /abr_2020/
// ----- ----- ----- ----- ----- ----- ----- ----- -----
//

// Module Declaration
```

```
module fsm_counter_mealy_0203_0407_1011_1619 (
 clk, // input  control clock
rstb, // input  control reset_bar
   x, // input  single data
   z  // output vector data
);

  // Parameter Declaration
parameter WIDTH_OUT = 5;
parameter NbrOfFFs  = 2;

  // I/O Port Declaration
input                     clk;
input                    rstb;
input                       x;
output [(WIDTH_OUT-1):0]    z;

  // Port Wire Declaration
wire                     clk;
wire                    rstb;
wire                       x;

  // Reg Declaration
reg [(WIDTH_OUT-1):0] z;

  // Internal Reg Declaration
reg [(NbrOfFFs-1):0] state_vect;
reg [(NbrOfFFs-1):0] excit_vect;  // assuming D-type Flip-Flop (DFF)...


  // Concurrent Assignment

    // Procedure by means of always
//
// -- -- --  G & A  -- -- --
always @(negedge rstb or posedge clk)
  begin
    if ( rstb == 0 )
      // set initial state = 00
      state_vect = 0;
    else
      // set next state
      state_vect = excit_vect;
  end
// -- -- --  G & A  -- -- --
//


    // Procedure by means of always
```

```
//
// -- -- --   C C   -- -- --
always @(state_vect or x)
  begin
    //
    case (state_vect)
      //
      0 :
        begin
          // set output
          if (x == 0)
            z = 5'b00010;  //-- 02;
          else
            z = 5'b00011;  //-- 03;
          // set excitation vector
          excit_vect = 1;
        end
      //
      1 :
        begin
          // set output
          if (x == 0)
            z = 5'b00100;  //-- 04;
          else
            z = 5'b00111;  //-- 07;
          // set excitation vector
          excit_vect = 2;
        end
      //
      2 :
        begin
          // set output
          if (x == 0)
            z = 5'b01010;  //-- 10;
          else
            z = 5'b01011;  //-- 11;
          // set excitation vector
          excit_vect = 3;
        end
      //
      3 :
        begin
          // set output
          if (x == 0)
            z = 5'b10000;  //-- 16;
          else
            z = 5'b10011;  //-- 19;
          // set excitation vector
          excit_vect = 0;
```

```
        end
      // no default...
      //default : z = 5'b00000;  //-- 00;
    endcase
    //
  end // always
// -- -- --   C C   -- -- --
//

endmodule // fsm_counter_mealy_0203_0407_1011_1619

//
// EOF
//
```

Referências bibliográficas

[HP81] F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley, New York, NY, 3rd edition, 1981.

[IC08] I. V. Idoeta and F. G. Capuano. *Elementos de Eletrônica Digital*. Editora Érica, 40^a edição edition, 2008.

[KD04] H. Kopka and P. W. Daly. *A Guide to LaTeX and Electronic Publishing*. Addison-Wesley, Harlow, England, 4th edition, 2004.

[MG04] F. Mittelbach and M. Goosens. *The LaTeX Companion*. Addison-Wesley, Boston, MA, USA, 2th edition, 2004.

[Rhy73] V. T. Rhyne. *Fundamentals of Digital Systems Design*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Stda] IEEE 1364-1995 Verilog Std. http://ce.sharif.edu/courses/85-86/2/ce483/resources/root/IEEE%20standard%20hardware%20description%20language%20based%20on%20the%20Ver.pdf.

[Stdb] IEEE 1364-2001 Verilog Std. http://www-inst.eecs.berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf.

[Stdc] IEEE 1364-2005 Verilog Std. http://staff.ustc.edu.cn/~songch/download/IEEE.1364-2005.pdf.

[Stdd] IEEE 1800-2009 SystemVerilog Std. http://93.174.95.29/main/9070E369D26B03AA2F7A9D951ACD79E4.

[Stde] IEEE 1800-2012 SystemVerilog Std. http://www.ece.uah.edu/~gaede/cpe526/2012%20System%20Verilog%20Language%20Reference%20Manual.pdf.

[Stdf] IEEE 1800-2017 SystemVerilog Std. http://ecee.colorado.edu/~mathys/ecen2350/IntelSoftware/pdf/IEEE_Std1800-2017_8299595.pdf.

[Tau82] H. Taub. *Digital Circuits and Microprocessors*. McGraw-Hill, New York, NY, 1982. Em português: McGraw-Hill, Rio de Janeiro, 1984.

[Tuta] Aleksandar Milenkovic. (System)Verilog Tutorial. http://www.mrc.uidaho.edu/mrc/people/jff/EO_440/Handouts/SystemVerilog%20and%20Modeling/verilog_synth.pdf.

[Tutb] Deepak Kumar Tala. Verilog Tutorial. http://classweb.ece.umd.edu/enee359a/verilog_tutorial.pdf.

[TWM07] R. J. Tocci, N. S. Widmer, and G. L. Moss. *Sistemas Digitais: Princípios e Aplicações*. Prentice Hall, Pearson Education, 10^a edição edition, 2007.

[Uye02] J. P. Uyemura. *Sistemas Digitais: Uma abordagem integrada*. Thomson Pioneira, São Paulo, SP, 2002.