UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

AMBIENTE COMPUTACIONAL PARA PROJETOS DE SISTEMAS COM TECNOLOGIA MISTA

TIAGO DA SILVA ALMEIDA

Ilha Solteira
Estado de São Paulo - Brasil
2009

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

AMBIENTE COMPUTACIONAL PARA PROJETOS DE SISTEMAS COM TECNOLOGIA MISTA

TIAGO DA SILVA ALMEIDA
Orientado

Prof. Dr. ALEXANDRE CÉSAR RODRIGUES DA SILVA
Orientador

Dissertação apresentada à Faculdade de Engenharia - UNESP – Campus de Ilha Solteira, para obtenção do título de Mestre em Engenharia Elétrica.

Área de Conhecimento: Automação.

Ilha Solteira
Estado de São Paulo - Brasil
2009

FICHA CATALOGRÁFICA

Elaborada pela Seção Técnica de Aquisição e Tratamento da Informação
Serviço Técnico de Biblioteca e Documentação da UNESP - Ilha Solteira.

Almeida, Tiago da Silva.
A447a Ambiente computacional para projetos de sistemas com tecnologia mista / Tiago da Silva Almeida. -- Ilha Solteira : [s.n.], 2009.
143 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Automação, 2009

Orientador: Alexandre César Rodrigues da Silva
Bibliografia: p. 110-112

1. Circuitos eletrônicos - Projetos. 2. Hardware – Linguagens descritivas. 3. Conversores digitais-analógicos.

unesp UNIVERSIDADE ESTADUAL PAULISTA
CAMPUS DE ILHA SOLTEIRA
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA

CERTIFICADO DE APROVAÇÃO

TÍTULO: AMBIENTE COMPUTACIONAL PARA PROJETOS DE SISTEMAS COM TECNOLOGIA MISTA

AUTOR: TIAGO DA SILVA ALMEIDA
ORIENTADOR: Prof. Dr. ALEXANDRE CESAR RODRIGUES DA SILVA

Aprovado como parte das exigências para obtenção do Título de MESTRE em ENGENHARIA ELÉTRICA , Área: AUTOMAÇÃO, pela Comissão Examinadora:

Prof. Dr. ALEXANDRE CESAR RODRIGUES DA SILVA
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Profa. Dra. DALVA MARIA DE O VILLARREAL
Departamento de Matemática / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. MAURO CONTI PEREIRA
Departamento de Engenharia Mecatrônica / Universidade Catolica Dom Bosco

Data da realização: 30 de outubro de 2009.

Dedicatória

Dedico este trabalho ao meu avô materno Sr Ulisses Rodrigues da Silva. Suas lições estarão sempre vívidas em minha memória.

Agradecimentos

Muitas pessoas contribuíram para que esta pesquisa fosse realizada, porém, alguns merecem um destaque especial. Dessa forma, agradeço primeiramente a minha família pelo apoio incondicional em minhas decisões ao longo de minha vida.

Agradeço imensamente ao meu orientador Alexandre César Rodrigues da Silva, por ter acreditado no meu trabalho e na minha capacidade em estar desenvolvendo este trabalho.

Aos meus professores de graduação Marcos Antônio Estremote e Tércio Alberto dos Santos Filho, por terem depositado em mim um importante voto de confiança que foi indispensável para a realização deste trabalho.

Ao pesquisador Silvano Renato Rossi, por ter ajudado a expandir meus horizontes na busca pelo saber.

À banca examinadora, composta pelos professores Dalva Maria de Oliveira Villarreal e Mauro Conti Pereira, pela valiosa contribuição ao trabalho.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo auxílio no desenvolvimento desse trabalho em forma de fomento.

Agradeço a todos que ajudaram de forma direta ou indireta nessa caminhada, pois, caso contrário, eu não seria metade da pessoa que sou e essa pesquisa não existiria tal qual como foi realizada.

Resumo

Neste trabalho, apresenta-se o desenvolvimento e a avaliação de duas ferramentas que auxiliam projetos de circuitos eletrônicos, sejam eles projetos de sistemas digitais ou de sistemas mistos (sinais digitais e sinais analógicos). A partir de um diagrama de transição de estados, modelado em ambiente Stateflow®, a primeira ferramenta, denominada SF^{2}HDL, realiza a extração de linguagens de descrição de *hardware*, podendo ser VHDL ou Verilog HDL. Sendo ainda capaz de extrair uma tabela de transição de estados padronizada, que, posteriormente, foi utilizada como entrada pelo programa TABELA, o qual realiza a minimização do sistema digital. A máquina de estados finitos, alvo da tradução, pode ser descrita tanto pelo modelo de Mealy como pelo modelo de Moore. Como estudos de caso, foram utilizados quatro códigos de linhas empregados em sistemas de telecomunicações. A segunda ferramenta é um aperfeiçoamento de uma ferramenta já existente, denominada MS^{2}SV, empregada na síntese de sistemas mistos. O MS^{2}SV é capaz de gerar uma descrição em VHDL-AMS estrutural, a partir de um modelo descrito em alto nível de abstração no ambiente Simulink®. Toda a estrutura de projeto necessária para a simulação e análise do sistema no ambiente SystemVision™, também é gerado pelo MS^{2}SV. Foram utilizados quatro modelos de conversor de dados do tipo DAC (*Digital to Analog Conversor*), para avaliar o desempenho da ferramenta. Nesse contexto, as duas ferramentas permitem maior flexibilidade ao projetista, traduzindo descrições em níveis de abstração diferentes, o que permite uma análise mais detalhada do funcionamento do sistema e facilitando a sua implementação física.

Palavras-chaves: Síntese, extração, máquina de estados finitos, linguagem de descrição de *hardware*, conversor de dados.

Abstract

In this work, it's shown the development and evaluation of two tools to aid in electronic circuits projects, be them digital systems projects or for mixed systems (digital and analogical signs). From a states transition diagram modeled in Stateflow® environment, the first tool, named SF^{2}HDL, performs the extraction of hardware description languages, which could be VHDL or Verilog HDL. It is also capable of extracting states transition table standardized, which later was used as a TABELA program, which accomplishes the minimization of the digital system. The target finite state machine of the translated can be described by the Mealy model as much as the Moore model. As case studies were used four code lines employed in telecommunications systems. The second tool is an improvement of an already existent tool, known as MS^{2}SV, used in the synthesis of mixed systems. The MS^{2}SV is able to generate a description in structural VHDL-AMS, from a model described in high level of abstraction in the Simulink® environment. The whole project structure necessary for the simulation and analysis of the system by the SystemVision™ environment is also generated by MS^{2}SV. Four DAC (Digital to Analog Converter) were used to evaluate the tool is performance. In that context, both tools allow a greater flexibility to the planner, translating descriptions in different abstraction levels, which allows a more detailed analysis of the systems behavior and making its physical implementation easier.

Keyword: Synthesis, extraction, finite state machine, hardware description language, data type converter.

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas

2B1Q	*Two Binary, One Quaternary*
ADC	*Analog to Digital Converter*
AMI	*Alternate Mark Inversion*
ASIC	*Application Specific Integrated Circuit*
CAD	*Computer Aided Design*
CI	Circuito Integrado
CPLD	*Complex Programmable Logic Device*
DAC	*Digital to Analog Converter*
DAE	*Differential Algebraic Equations*
DOME	*Honeywell Domain Modeling Environment*
EDULIB	*Educational Library*
FPGA	*Field Programmable Gate Array*
GUI	*Graphical User Interface*
HDB1	*High Density Binary of order 1 code*
HDB3	*High Density Binary of order 3 code*
HDL	*Hardware Description Language*
IEEE	*Institute of Electrical and Electronics Engineers*
LAB	*Logic Array Blocks*
LSB	*Least Significant Bit*
LTL	*Linear Temporal Logic*
LUT	*Lookup Table*
MAST AHDL	*MAST Analog Hardware Description Language*
MLT-3	*Multi Level Transmit 3*
MS^2SV	*Matlab / Simulink to SystemVision*
MSB	*Most Significant Bit*
PROMELA	*Process or Protocol Meta Language*
RAM	*Random Access Memory*
RTL	*Register Transfer Level*
SF^2HDL	*Stateflow to Hardware Description Language or state transition table*
SPICE	*Simulation Program with Integrated Circuit Emphasis*

SRAM	*Static Random Access Memory*
TAB2VHDL	*TABELA to VDHL*
TTL	*Transistor Transistor Logic*
UML	*Unified Modeling Language*
VHDL	*VHSIC Hardware Description Language*
VHDL-AMS	*VHSIC Hardware Description Language - Analog Mixed Signal*
VHSIC	*Very High Speed Integrated Circuits*
XML	*Extensible Markup Language*

Sumário

Capítulo 1. Introdução geral

Neste capítulo, são apresentadas as metodologias utilizadas na modelagem de projetos de sistemas eletrônicos e alguns trabalhos que foram desenvolvidos para ajudar a resolver problemas de projetos desses sistemas.

Dessa forma, na seção 1.2, apresenta-se uma breve introdução sobre como é realizado o processo de síntese e as metodologias utilizadas em projetos de sistemas eletrônicos, tanto sistemas digitais, como sistemas mistos. Na seção 1.2, apresenta-se uma revisão bibliográfica dos principais trabalhos encontrados que empregam ferramentas de tradução no processo de síntese. Por fim, na seção 1.3, apresenta-se o objetivo deste trabalho, a metodologia utilizada na tradução dos estudos de caso.

No Capítulo 2, são apresentados os conceitos básicos utilizados no desenvolvimento do trabalho, como o conceito de máquina de estados finitos, apresentado na seção 2.1. Na seção 2.2, apresentam-se o funcionamento e as características dos dispositivos conversores de dados de um modo geral. Na seção 2.3, apresenta-se o conceito básico dos códigos de linha de sistemas de telecomunicações como também os códigos que foram utilizados para validação da metodologia da ferramenta SF^{2}HDL. Na seção 2.4, apresentam-se as linguagens de descrição de hardware utilizadas, como se deu o surgimento dessas linguagens e os benefícios em sua utilização.

No Capítulo 3, são apresentadas as ferramentas que foram utilizadas nas metodologias propostas, sejam elas ferramentas de software ou não. Logo, na seção 3.1, apresentam-se, de forma resumida, as principais características do ambiente Matlab®, Simulink® e Stateflow®. Na seção 3.2, apresentam-se o ambiente de modelagem, a análise e simulação SystemVision™. Na seção 3.3, apresentam-se os modelos de conversores de dados utilizados na avaliação da metodologia da ferramenta MS^{2}SV, de acordo com o manual de cada dispositivo. Na seção 3.4, apresentam-se as características do programa TABELA. Na seção 3.5, apresentam-se as características do programa TAB2VHDL e um exemplo do arquivo utilizado como entrada e um exemplo de arquivo de saída gerado pelo programa. Na seção 3.6, apresentam-se as características de funcionamento e a metodologia utilizada pela primeira versão da ferramenta MS^{2}SV.

No Capítulo 4, são apresentadas as ferramentas desenvolvidas neste trabalho. Na seção 4.1, apresentam-se a interface de utilização da ferramenta SF^{2}HDL e os passos envolvidos na extração de máquinas de estados finitos. Na seção 4.2, apresentam-se a interface de utilização e a nova metodologia proposta para síntese dos conversores de dados da ferramenta MS^{2}SV, agora na versão 1.7.

No Capítulo 5, são apresentados os estudos de caso e o resultado da avaliação das ferramentas desenvolvidas. Na seção 5.1, apresenta-se o resultado apresentados pelo SF^{2}HDL no estudo de caso dos códigos de linha apresentados no Capítulo 2. Na seção 5.2, apresenta-se o resultado da validação da nova versão da ferramenta MS^{2}SV para o estudo de caso dos conversores apresentados no Capítulo 3.

Por fim, no Capítulo 6, apresenta-se a conclusão obtida com o desenvolvimento deste trabalho de pesquisa.

1.1. Introdução à síntese de circuitos eletrônicos

Fatores decisivos como redução de custos e tempo na fabricação de CIs (Circuito Integrado), automação de processos industriais e maior eficiência em sistemas de aquisição de dados, levaram a um grande avanço tecnológico nas últimas décadas. Tais fatores fizeram surgir ASICs (*Application Specific Integrated Circuit*) mais complexos e desenvolvidos para uma grande diversidade de aplicações. Como por exemplo, a indústria automotiva, aeroespacial, telecomunicações etc.

Porém, o trabalho de desenvolvimento de novas tecnologias não é uma tarefa trivial e demanda muito esforço do projetista e de ferramentas que auxiliam na realização desses projetos, fazendo-se necessário o surgimento de novas metodologias e ferramentas de *software* que possam ajudar a resolver problemas de projeto.

A ausência de um padrão bem definido para a especificação de projetos em alto nível de abstração, fez com que surgisse uma variedade muito grande de metodologias e ferramentas, fazendo-se necessária a sistematização dos passos envolvidos na criação de um determinado projeto. Os primeiros autores a tratar de forma sistemática as metodologias de projetos existentes foram Gajski e Kuhn (1983), constatando-se que existem basicamente três abordagens.

Na primeira abordagem, acredita-se que todas as decisões de projetos podem ser tomadas por projetistas, pois, ao longo dos anos, se ganhou uma boa prática de projeto. Essa abordagem tende a ser uma abordagem *bottom-up*, visto que, blocos construídos são

projetados inicialmente em baixo nível de abstração e, posteriormente, usados para realizar estruturas em alto nível. Por outro lado, um projetista é lento e sujeito a erros.

Na segunda abordagem, o conhecimento humano tem sido capturado em forma de regras de projeto e armazenado em bases de conhecimento. Sistemas que tentam reproduzir o desempenho de um ou mais projetistas experientes, denominadas sistemas inteligentes, são menos sujeitos a erros e, quando esses sistemas são direcionados, se tornam mais eficientes que um projetista. Por outro lado, sistemas inteligentes são eficientes somente na análise de um projeto.

Na terceira abordagem, denominada *top-down*, acredita-se que, na prática de projetos de CI (Circuito Integrado), o conhecimento é algorítmico e que tradutores podem ser escritos para sintetizar partes ou todo o projeto automaticamente a partir da descrição do problema em alto nível. Nela, é necessária a análise em um nível mais alto de abstração, antes de se chegar à implementação física.

Já no trabalho de Riesgo, Torroja e De La Torre (1999), são consideradas apenas duas metodologias de projeto: *buttom-up* e *top-down*, ambas conforme o descrito por Gajski e Kuhn (1983), sendo a primeira abordagem considerada atualmente como um método tradicional, já que a maioria dos projetistas tem voltado seus esforços à metodologia *top-down*.

Gajski e Kuhn (1983) também foram os primeiros autores a abordar os níveis de abstração que um projeto exige através de um diagrama conhecido como “Diagrama Y”. Esse, por sua vez, sofreu alterações ao longo dos anos até chegar ao diagrama proposto na Figura 1.1, e que também é descrito de diferentes formas por diversos autores.

Ao longo de cada um dos três eixos denominados: domínio comportamental, domínio estrutural e domínio físico, o refinamento do projeto é conduzido em níveis. E quanto mais o projeto se aproxima da origem, mais baixo será o nível de abstração, e de forma análoga, quanto mais a descrição do projeto se afasta da origem, mais alto será o nível de abstração.

No domínio comportamental, o interesse do projetista está na funcionalidade do projeto, e no que o sistema faz. A representação do projeto no domínio comportamental pode ser realizada por vários níveis, iniciando com instruções e redes de Petri, linguagens algorítmicas, expressões booleanas e máquina de estados finitos e equações diferenciais.

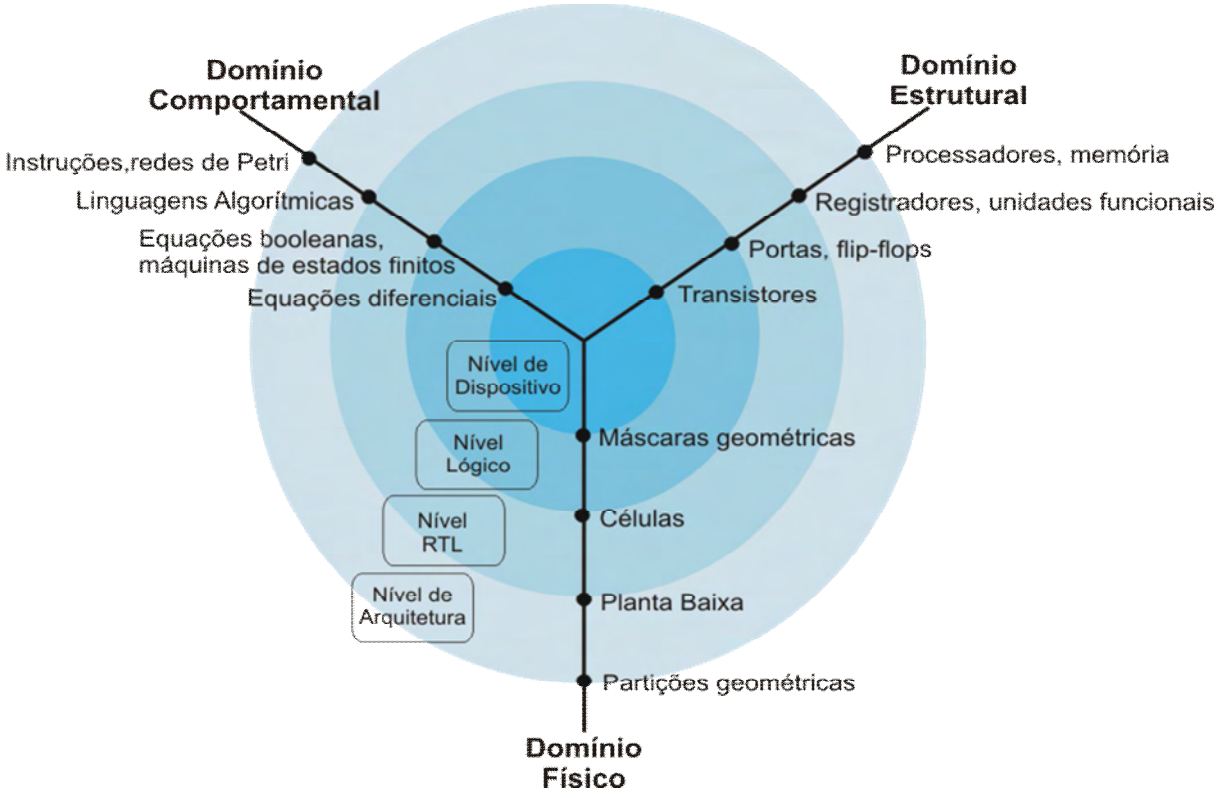

Figura 1.1. Níveis de abstração de um projeto representado no diagrama Y.

O domínio físico especifica parâmetros físicos, como a disposição de portas lógicas em uma placa de circuito impresso ou *chip* de silício. Os níveis da representação física são máscaras geométricas, células, planta baixa com tamanho de blocos arbitrários e partições geométricas.

O domínio estrutural representa a ponte entre o domínio comportamental e o domínio físico. Trata-se de um mapeamento do domínio comportamental em um conjunto de componentes e conexões sujeito às restrições de custo, área e tempo. Os níveis comumente usados na representação estrutural são transferências entre memória e processador, transferência entre registradores e unidades funcionais, portas lógicas e *flip-flops*, e transistores.

Também de acordo com Riesgo, Torroja e De La Torre (1999), a transformação de um nível mais alto de abstração em um nível mais baixo, é comumente chamado de “síntese”. Assim como a transformação de uma descrição comportamental em uma descrição estrutural do mesmo nível também é chamada de “síntese”, embora, se alguma tecnologia ou biblioteca em particular for considerada, esse passo pode ser chamado de “mapeamento”. A transformação de um nível de abstração em um determinado domínio para fornecer o desempenho do sistema pode ser chamada de “otimização” ou “refinamento”. Por fim, a

transformação de um nível mais baixo de abstração em um nível mais alto é conhecida como "extração".

As metodologias de projetos, bem como as transformações entre níveis de abstrações diferentes, podem ser mais bem entendidas através da seção 1.2, em que é apresentada uma revisão bibliográfica dos trabalhos que empregam ferramentas de *software* no processo de síntese, extração e otimização em projetos de circuitos eletrônicos.

1.2. Revisão da bibliografia

A utilização de domínios e níveis de abstração de projetos é extremamente útil, já que a complexidade dos projetos tem aumentado nos últimos anos, os quais tendem a ser cada vez mais complexos, pois as necessidades de mercado aumentam a cada dia. Em decorrência, um projetista faz uso de uma grande variedade de ferramentas disponíveis comercialmente e para um número muito elevado de aplicações. Essas aplicações vão desde a análise de dados científicos até a modelagem e simulação de circuitos eletrônicos. Com isso, surge a necessidade de uma linguagem flexível para suportar a tradução de modelos descritos em outras formas de modelagem, já que, atualmente, não existe um padrão para especificação, modelagem e simulação de projetos de sistemas eletrônicos.

Algumas ferramentas de modelagem disponíveis comercialmente incorporam a necessidade de tradução para formas de modelagem diferentes. Um exemplo é a ferramenta Matlab C Compiler, desenvolvida pela Mathworks, que traduz o código fonte dos programas escritos em Matlab para um código na linguagem C. Essa ferramenta pode ser instalada separadamente do ambiente Matlab®, e seu funcionamento e suas principais vantagens são descritas no trabalho de Mirotznik (1996).

Máquinas de estados finitos representam uma importante forma de modelagem para circuitos eletrônicos, possibilitando várias formas de verificação. No trabalho de Wang e Edsall (1998), máquinas de estados finitos utilizam um estilo de codificação padrão proposta pelos autores, através da linguagem Verilog HDL (*Hardware Description Language*) no domínio comportamental. A partir daí, foi utilizado um algoritmo para traduzir o modelo da máquina de estados finitos descrito pela linguagem de descrição de *hardware* para uma

representação visual na forma de diagrama de bolha em PostScript[1], já que, o diagrama de bolha é uma importante forma de verificação e documentação de projetos.

Em 1999, Mukherjee e Fedder (1999) apresentaram três pontos importantes em um projeto de sistemas microeletromecânicos. O primeiro é a simulação do circuito em domínio misto (domínio elétrico e mecânico), utilizando um conjunto de elementos micro eletromecânicos, podendo ser conectados hierarquicamente para criar sistemas e componentes mais complexos. O segundo ponto é uma estratégia de síntese em nível de componente, através da utilização de bibliotecas parametrizadas, que traduzem especificações do dispositivo em parâmetros de *layout* geométrico. O terceiro ponto é uma estratégia de extração do *layout* geométrico em uma representação do circuito em domínio misto, através de uma rede de conexões entre os componentes do sistema. O ponto forte no desenvolvimento desse trabalho foi a utilização da metodologia *top-down*, o que permite uma rápida verificação do sistema a um baixo custo computacional.

Ainda que a principal ênfase no desenvolvimento de linguagens e suas ferramentas de suporte têm sido, para atividades de projeto, atividades relacionadas com testes, cada vez mais importantes. Porém, testes para circuitos integrados de sinais mistos sempre foram um problema para a indústria de microeletrônica devido à natureza e complexidade dos circuitos. Uma verificação visual pode auxiliar os projetistas na fase de testes de CI. No trabalho de Stone e Manolakos (2000), esse fato foi evidenciado com o desenvolvimento da ferramenta DG2VHDL (*Dependence Graph to VHDL – VHSIC Hardware Description Language*). Essa ferramenta é capaz de traduzir descrições de algoritmos abstratos altamente complexos, conhecidos como dependência gráfica, para um modelo em VHDL comportamental, o que facilita uma validação rápida do projeto. Utilizando um algoritmo heurístico, o DG2VHDL gera um modelo VHDL quase ótimo. Por fim, o projeto é implementado em um FPGA da Altera.

No trabalho de Grout e Keane (2000), foi desenvolvido um protótipo de *software* capaz de analisar e sintetizar um modelo de diagrama de blocos de um controlador de motores de malha fechada, descritos em ambiente Simulink®, para uma descrição em VHDL. A descrição em VHDL representa uma combinação de domínios de abstração: comportamental, funcional e estrutural. Através do modelo em VHDL, é possível a implementação do sistema em dispositivos como FPGAs (*Field Programmable Gate Array*) e ASICs.

---

[1] PostScript é uma linguagem de programação especializada para visualização de informações, ou uma linguagem de descrição de páginas, originalmente criada para impressão e, posteriormente, modifica para o uso em monitores.

Em 2001, Gannod e Gupta (2001) desenvolvem uma ferramenta que realiza uma tradução automática de especificações de uma rede de Petri, modelada no ambiente DOME[2], para uma descrição equivalente na linguagem PROMELA[3] (*Process or Protocol Meta Language*). O código em PROMELA obtido na tradução foi executado no ambiente Spin[4]. A abordagem para a tradução é baseada na semântica das transições existentes na rede de Petri, o que reduz os esforços na fase de verificação do projeto.

A possibilidade de integrar em único CI (Circuito Integrado) partes analógicas e digitais tem alavancado o crescimento da indústria eletrônica e, consequentemente, tornou-se uma importante área de pesquisa. Como, por exemplo, no trabalho de Soudris et al. (2001), em que foi analisada e apresentada uma ferramenta capaz de gerar uma descrição estrutural em VHDL de conversores de dados, a partir de um sistema numérico de resíduo. Essa abordagem tenta minimizar o tempo necessário para o projetista investigar as soluções alternativas do sistema numérico de resíduo. O código VHDL, gerado pela ferramenta desenvolvida, foi simulado no ambiente Synopsys da Mentor Graphics.

No trabalho de Camera (2001), foi apresentada a ferramenta intitulada SF2VHD. A ferramenta é capaz de traduzir modelos de máquinas de estados finitos para uma descrição estrutural em VHDL. As máquinas de estados finitos foram modeladas hierarquicamente no ambiente Stateflow® da Mathworks. A ferramenta foi desenvolvida em linguagem C++ e apresentou uma tradução eficiente para os casos analisados e simulados no ambiente Synopsys.

No trabalho de Horta (2002), foi descrita uma metodologia simbólica aplicada à exploração e caracterização de topologias e arquiteturas de sistema de sinais analógicos e mistos, permitindo o processo de síntese em alto nível de abstração e auxiliando o desenvolvimento de novas estruturas e modelos de conversores de dados de alta resolução.

Em 2003, Zorzi, Franzè e Speciale (2003) desenvolveram um simulador para a linguagem VHDL-AMS (*VHSIC Hardware Description Language – Analog Mixed Signals*), desenvolvido no ambiente Matlab, denominado S.A.M.S.A. Nesse trabalho, foi utilizado como estudo de caso um modelo de conversor ADC (*Analog to Digital Conversor*) de baixa

---

[2] DOME (*Honeywell Domain Modeling Environment*) é uma ferramenta que suporta projetos de sistemas usando uma grande variedade de modelagens, incluindo diagramas UML e Redes de Petri.

[3] PROMELA é uma linguagem de modelagem e verificação. A linguagem permite a criação dinâmica de processos concorrentes para modelos, por exemplo: sistemas distribuídos.

[4] SPIN é uma ferramenta geral para verificação e correção de modelos de *software* distribuídos em um modo rigoroso e geralmente automático. Os sistemas a serem verificados são descritos em PROMELA e suas propriedades são expressas como fórmulas de lógica linear temporal (LTL).

potência. A ferramenta foi comparada com outros simuladores de código VHDL-AMS disponíveis comercialmente, a qual foi capaz de simular o código VHDL-AMS corretamente e os resultados foram os mesmo em relação a outras ferramentas comerciais de simulação.

No trabalho de Nehme e Ludqvist (2003), foi apresentada uma ferramenta capaz de traduzir uma máquina de estados finitos, descrita através da linguagem VHDL, em um diagrama de transição de estados dessa determinada máquina. O foco deste trabalho é a modelagem e a verificação (através de máquinas de estados finitos) de sistemas de aplicações de missão crítica como, por exemplo, sistemas de controle de tráfego aéreo.

No trabalho de Sbarcea e Nicula (2004), foi desenvolvida uma ferramenta denominada Sim2HDL, que realiza a tradução automática de modelos de projetos do Simulink em uma linguagem de descrição de *hardware*, na tentativa de reduzir drasticamente o tempo de projeto. A linguagem gerada pode ser tanto a VHDL como a Verilog, ambas em uma descrição comportamental e, a partir dessa descrição, o projeto pode ser implementado em FPGAs utilizando sintetizadores comerciais.

Também em 2004, Zorzi et al. (2004) desenvolveu uma metodologia, através de ferramentas CAD (*Computer Aided-Design*), para a integração da linguagem VHDL-AMS com a linguagem SPICE. Nesse trabalho, os modelos em VHDL-AMS são compilados e, posteriormente, convertidos em uma biblioteca de modelos SPICE, para serem simulados como um modelo puramente em SPICE.

No trabalho de Shanblatt e Foulds (2005), foi apresentada uma metodologia para implementação em FPGA de projetos modelados no ambiente Simulink®. Os modelos do Simulink® são inicialmente traduzidos em um gráfico de fluxo de dados de controle (*Control Data Flow Graph*), utilizando a linguagem XML (*eXtensible Markup Language*) e, posteriormente, traduzidos em um código VHDL estrutural, podendo ser sintetizados e implementados em FPGA por ambientes comerciais de síntese.

No trabalho de Guihal et. al. (2006), foi criada uma metodologia para a tradução de projetos de sinais mistos (sinais digitais e analógicos), modelados em MAST[5], para uma descrição correspondente em VHDL-AMS. Os modelos traduzidos foram simulados em ambiente SystemVision™ da Mentor Graphics, por possibilitar um comparação visual do modelo VHDL-AMS. Para realizar a metodologia de tradução, foi desenvolvido um ambiente

[5]MAST é a primeira linguagem de modelagem de sinais analógicos. Mais conhecida como MAST AHDL (*Analog Hardware Description Language*).

denominado Sig_Cad. Porém, o ambiente ainda precisa de outras ferramentas para realizar uma tradução precisa.

No trabalho de Markovic, Richards e Brodersen (2006), foi apresentada uma metodologia de projeto para a geração automática de HDL (*Hardware Description Language*) em domínio comportamental, a partir de um diagrama de blocos modelado no ambiente Simulink®. Geração automática é feita através da ferramenta intitulada In-House, desenvolvida no ambiente Matlab®. Essa metodologia apresentou resultados satisfatórios, através do estudo de caso do algoritmo de Sobel[6]. A partir dessa tradução, é possível uma implementação otimizada do modelo em FPGA ou ASIC.

No trabalho de Krasniewski (2008), foi proposta uma metodologia de detecção de erro em projetos de máquinas de estados finitos, utilizando blocos de memória embarcada de FPGAs baseados em SRAM (*Static Random Access Memory*). A metodologia proposta detecta falhas permanentes e temporárias que são associadas a uma única saída ou entrada de algum componente do circuito que resulte em uma transição de estado ou uma saída incorreta. O trabalho foi desenvolvido utilizando uma ferramenta de síntese dedicada, denominada FSMdec. Esta, por sua vez, apresentou um baixo custo computacional.

O avanço no desenvolvimento de novas ferramentas de tradução que auxiliam em projetos de CI, sejam, sistemas com células complexas que trabalhem ou não com partes analógicas e partes digitais, são ilustrados através do gráfico na Figura 1.2. No gráfico, apresenta-se a quantidade de publicações referentes às ferramentas de traduções empregadas na síntese, ou mesmo que auxiliam a síntese de projetos eletrônicos.

Observa-se na Figura 1.2 que as publicações se mantiverem estáveis até o ano de 1999, onde houve um fortalecimento das pesquisas relacionadas à síntese de circuitos eletrônicos, sempre envolvendo a metodologia *top-down* na sistematização do projeto. Porém, há um pico no ano de 2003, em que surgiram muitos trabalhos relevantes, como, por exemplo, o trabalho de Zorzi, Franzè e Speciale (2003), Nehme e Ludqvist (2003), sendo que, em ambos os trabalhos, são empregadas as HDLs como forma de modelagem de projetos eletrônicos. Diferem-se na metodologia utilizada em cada um dos trabalhos, já que o primeiro aborda projetos de sinais mistos e o segundo utiliza a HDL como alvo de tradução para máquinas de estados finitos, em projetos digitais.

---

[6] O Algoritmo de Sobel é utilizado em processamento de imagens aplicado em algoritmos de detecção de borda.

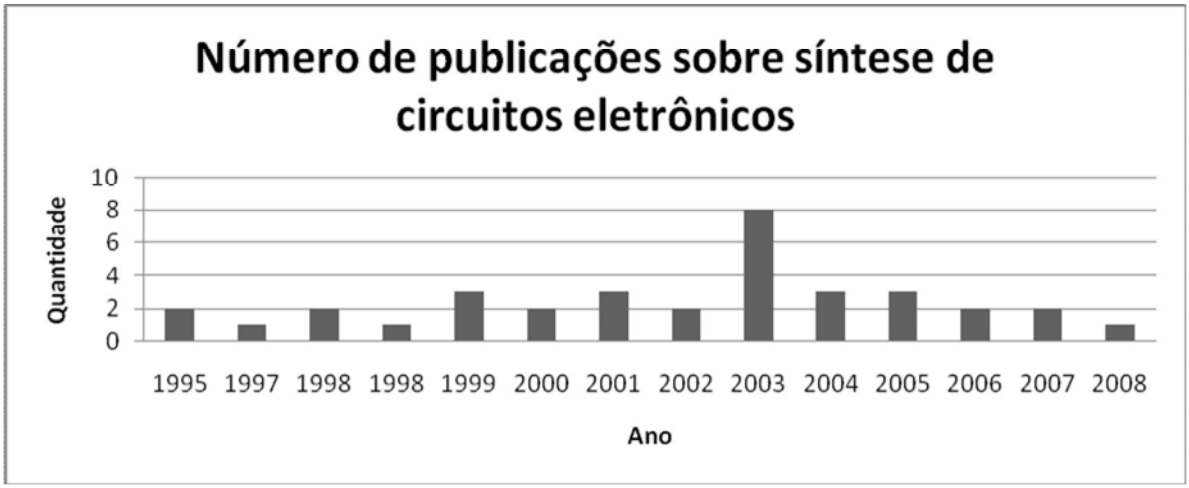

Figura 1.2. Gráfico de publicações realizadas nos anos de 1995 a 2008.

Nesse contexto, a maioria dos trabalhos desenvolvidos no ano de 2003 envolve a utilização de HDLs, seja como alvo de tradução, como resultado de tradução, ou até mesmo empregadas somente na análise e verificação dos sistemas através de ferramentas de simulação. Outros trabalhos ainda abordam a síntese através da combinação de modelos matemáticos e HDLs, demonstrando o foco no domínio comportamental e na metodologia *top-down*. O emprego das HDLs nesses trabalhos é extremamente conveniente, pois, em alguns deles, são utilizadas FPGAs para implementação final do projeto, já que FPGAs fornecem uma flexibilidade muito grande em diversas aplicações.

A Figura 1.2 ilustra uma grande queda nas pesquisas até o ano de 2008. Ainda sim, o trabalho de síntese não é uma tarefa trivial e ainda há muita pesquisa a ser realizada, no que diz respeito a novas metodologias que possam permitir a redução de tempo e custo no desenvolvimento de projetos cada vez mais complexos, já que o desenvolvimento tecnológico é uma área em constante expansão.

É importante ressaltar que o gráfico apresentado na Figura 1.2 foi confeccionado a partir dos trabalhos utilizados para o desenvolvimento desta pesquisa, sendo somente um gráfico aproximado das publicações ocorridas entre os anos de 1995 a 2008.

Na próxima seção apresenta-se o objetivo deste trabalho, bem como a organização dos Capítulos, seções e subseções.

1.3. Ferramentas desenvolvidas neste trabalho

Com o intuito de reduzir o tempo e, consequentemente, os custos no desenvolvimento de projetos de circuitos eletrônicos, neste trabalho, são apresentadas duas ferramentas distintas que possibilitam maior flexibilidade à fase de modelagem de projetos. Seguindo a tendência de mercado, ambas as ferramentas se apoiam na metodologia *top-down*, fazendo-se

necessária a modelagem e simulação de um projeto em níveis mais altos de abstração, até chegar à implementação física do projeto.

A primeira ferramenta desenvolvida, denominada SF^{2}HDL (*Stateflow to Hardware Description Language or states transition table*), é responsável por extrair máquinas de estados finitos, modeladas no ambiente Stateflow®, para uma descrição correspondente em VHDL ou Verilog HDL, ou ainda, em uma tabela de transição de estados padronizada. Na Figura 1.3, ilustra-se o comportamento da metodologia de projeto proposta para a ferramenta SF^{2}HDL, dentro do diagrama Y.

O projeto de máquinas de estados finitos é inicialmente modelado no domínio comportamental em nível lógico utilizando o ambiente Stateflow®. Esse ambiente é considerado uma ferramenta adicional ao ambiente Matlab® desenvolvido pela Mathworks. A ferramenta SF^{2}HDL extrai o projeto para uma linguagem algorítmica e para uma tabela de transição de estados padronizada, ambas em nível RTL (*Register Transfer Level*). Em seguida, o programa TABELA (SILVA, 1989) sintetiza a tabela de transição de estados padrão em um modelo de expressões booleanas minimizadas em nível lógico.

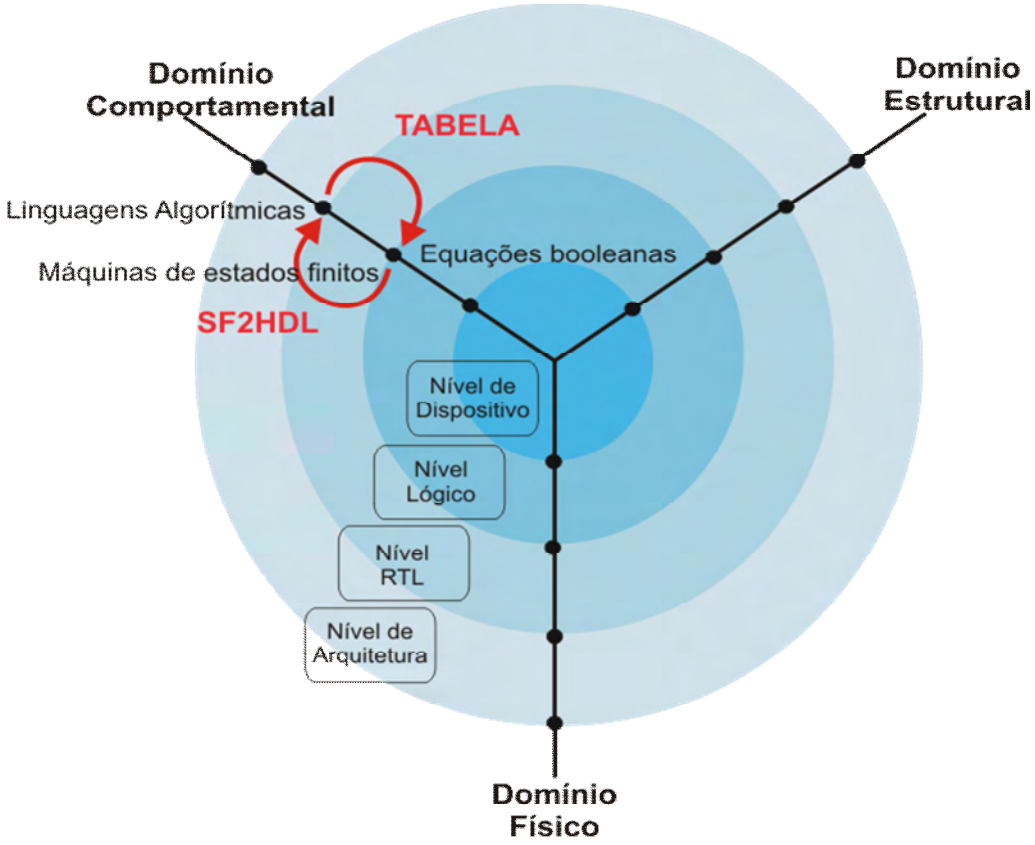

Figura 1.3. Metodologia de projeto no diagrama Y para o SF^{2}HDL e TABELA.

O programa TAB2VHDL (TANCREDO, 2002) faz a leitura do arquivo gerado na saída do programa TABELA e extrai novamente o projeto para uma linguagem algorítmica em

nível RTL. A partir daí, os sintetizadores comerciais, no caso o ambiente Quartus II, sintetizam a descrição para domínio físico em nível lógico, cuja implementação pode ser realizada em FPGA. A metodologia de projeto para o TAB2VHDL e para o ambiente Quartus II, apresenta-se na Figura 1.4.

As máquinas de estados finitos podem ser descritas tanto pelo modelo de Mealy como pelo modelo de Moore, podendo ser completa ou incompletamente especificada.

Para avaliar a metodologia proposta neste trabalho para a ferramenta SF^{2}HDL, foram utilizados códigos de linha, como, por exemplo, o código HDB3 e o 2B1Q, que são amplamente empregados em sistemas de telecomunicações.

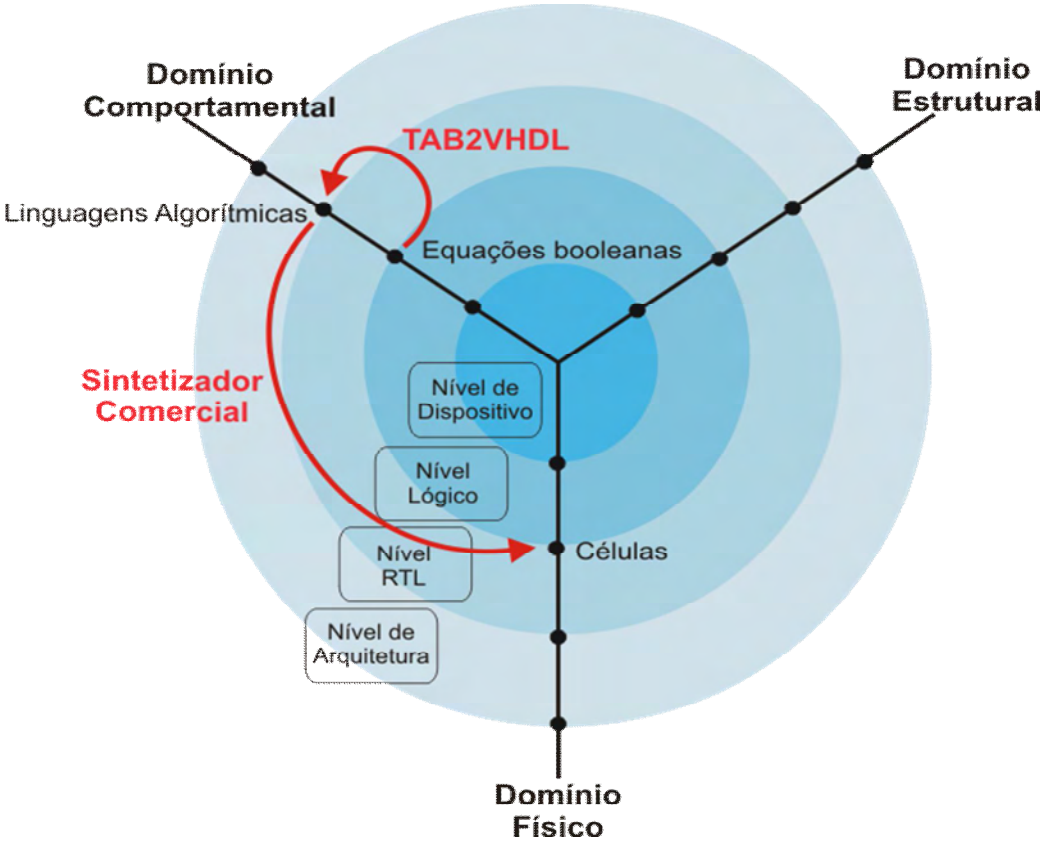

Figura 1.4. Metodologia de projeto no diagrama Y para o TAB2VHDL e para o Quartus II.

A segunda ferramenta desenvolvida trata-se de um aperfeiçoamento de uma ferramenta já existente, denominada MS^{2}SV (*Matlab / Simulink to SystemVision*™) (SILVA, 2007). Essa ferramenta é capaz de sintetizar projetos de sinais mistos em VHDL-AMS e gerar toda a estrutura de projeto para o ambiente SystemVision™ da Mentor Graphics. Na Figura 1.5, apresenta-se o comportamento da metodologia de projeto proposta para a ferramenta MS^{2}SV, também dentro do diagrama Y.

O projeto de sinais mistos é inicialmente modelado como diagrama de blocos em nível de arquitetura no domínio comportamental. O MS^{2}SV sintetiza o projeto gerando todas as ligações entre os componentes contidos no modelo através de uma linguagem algorítmica em nível RTL. Posteriormente, o ambiente SystemVision™ realiza mais uma vez a síntese do projeto, porém, dessa vez para o nível lógico no domínio físico, assim como o ambiente Quartus II.

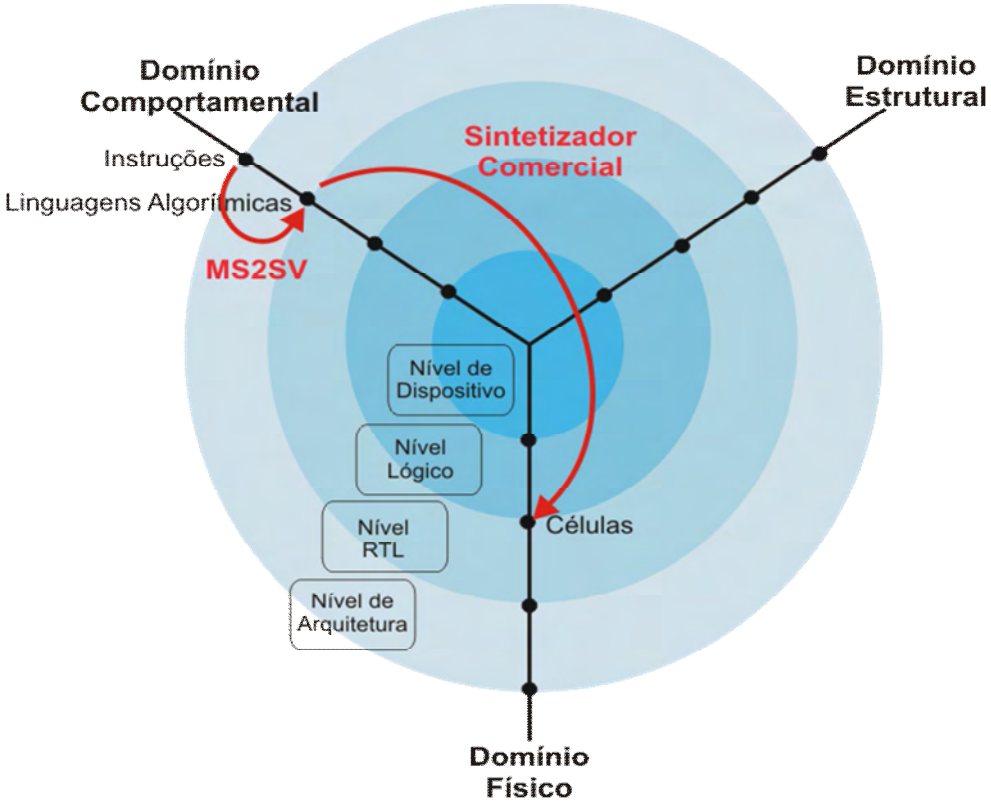

Figura 1.5. Metodologia de projeto no diagrama Y para o MS^{2}SV.

A validação da metodologia para o MS^{2}SV foi realizada através do uso de conversores de dados do tipo DAC (*Digital to Analog Converter*). Quatro modelos foram avaliados: DAC08, AD7524, AD7528 e AD5450, sendo que três modelos realizam a leitura dos dados de forma paralela e apenas um de forma serial.

Capítulo 2. Conceitos básicos utilizados

Neste capítulo, são passados os conceitos iniciais, que foram necessários para o desenvolvimento deste trabalho. Sendo assim, na seção 2.2 é apresentado o conceito de máquinas de estados finitos, como a forma de descrição e modelagem através de diagramas de transição de estados. Na seção 2.3, são apresentadas as formas de representação de sinais (analógicas e digitais) e o funcionamento dos conversores de dados. Já na seção 2.4, são apresentados alguns dos códigos de linha, de sistemas de telecomunicações, utilizados como estudo de caso da ferramenta SF^2HDL. E, finalmente, na seção 2.5, são apresentadas três linguagens de descrição de *hardware* frequentemente utilizadas na descrição de sistemas eletrônicos e disponíveis nas ferramentas de sínteses comerciais, o VHDL, o Verilog HDL e o VHDL-AMS.

2.1. Máquinas de estados finitos

Máquinas de estados finitos ou máquinas sequenciais são representações abstratas de um circuito sequencial real. Seu comportamento pode ser descrito como uma sequência de eventos que acontecem em instantes de tempo discreto.

Máquinas de estados finitos são usadas em vários momentos do nosso dia a dia, como por exemplo, em controle de elevadores, semáforos etc. Existe em sistemas digitais a necessidade de armazenar e efetuar operações lógicas e/ou aritméticas, sendo que tais sistemas podem ser representados (modelados) como máquinas de estados (SILVA, 1989).

Uma máquina de estados finitos pode ser representada através do esquemático apresentado na Figura 2.1. O circuito possui um número finito de entradas, constituindo o conjunto das variáveis de entrada $N = \{N_1,N_2,...,N_n\}$. Assim, o circuito tem um número finito de saídas, determinado pelo conjunto de variáveis de saída $M = \{M_1,M_2,...,M_m\}$. O valor contido em cada elemento de memória é chamado de variáveis de estado, formando o conjunto das variáveis de estado $K = \{K_1,K_2,...,K_k\}$. Os valores contidos nos *K* elementos de memória definem o estado atual da máquina. As funções de transições internas geram o conjunto de próximo estado $S = \{S_1,S_2,...,S_s\}$, que dependem das entradas *N* e dos estados atuais *K* da máquina e são definidas através de circuitos combinacionais. Os valores de *S*, que aparecem na função de transição da máquina de estados no instante *t*, determinam os valores

das variáveis de estado no instante *t+1*, e, portanto, definem o próximo estado da máquina (SILVA, 1989).

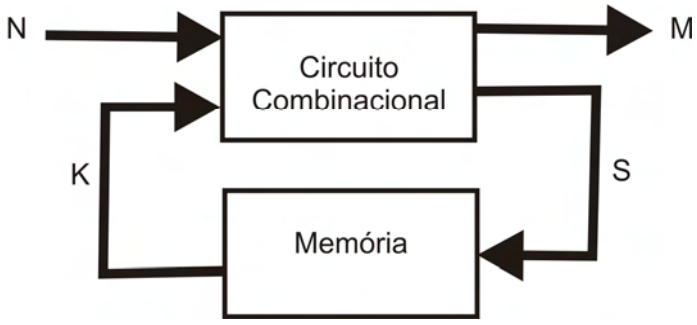

Figura 2.1. Esquemático de máquina de estados finitos.

O comportamento de uma máquina de estados finitos pode ser descrito através de um diagrama de transição de estados ou através de uma tabela de transição de estados.

Um diagrama de transição de estados ou uma tabela de transição de estados relacionam estado atual, próximo estado, entrada e saída. Uma tabela de transição de estados tem 2^N colunas, uma para cada ocorrência do conjunto de entrada e 2^K linhas, uma para cada ocorrência do conjunto de estado.

O diagrama de transição é um grafo orientado, onde cada nó representa um estado, e de cada nó emanam *p* arcos orientados correspondendo às transições de estado. Cada arco orientado é rotulado com a entrada que determina a transição e a saída gerada. As máquinas de estados finitos determinam o próximo estado *K(t+1)*, somente com base no estado atual *K(t)* e na entrada atual *N(t)*. As máquinas de estados podem ser representadas pela equação 2.1:

$$K(t+1) = f[K(t), N(t)] \tag{2.1}$$

onde *f* é uma função de transição de estados. O valor da saída *M(t)* é obtido através de duas equações, 2.2 e 2.3:

$$M(t) = g[K(t)] \tag{2.2}$$

$$M(t) = g[K(t), N(t)] \tag{2.3}$$

onde *g* é uma função de saída.

Uma máquina com propriedades descritas nas equações 2.1 e 2.2 é chamada de *Máquina de Moore* e uma máquina descrita através das equações 2.1 e 2.3 é chamada de *Máquina de Mealy*. Assim, na Figura 2.2, ilustra-se um modelo de diagrama de transição de estados descrito através da máquina de Mealy. Esse modelo representa um circuito detector de

três zeros consecutivos sem sobreposição e de forma equivalente. Na Figura 2.3, é apresentado um diagrama de transição de estados descrito pelo modelo de Moore para o mesmo circuito detector de três zeros consecutivos (SILVA, 1989).

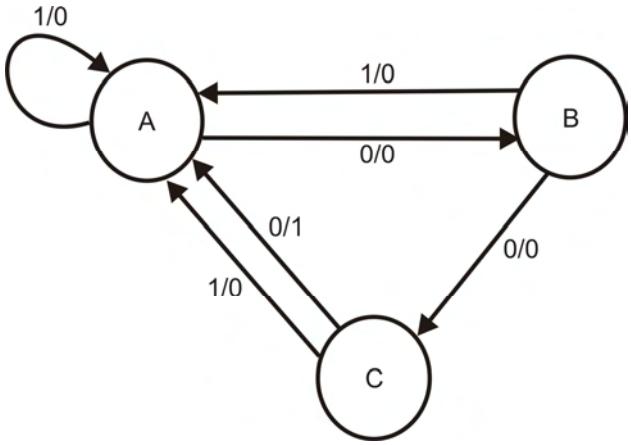

Figura 2.2. Diagrama de transição de estados descrito pelo modelo de Mealy.

De fato, toda máquina representada pelo modelo de Mealy pode ser descrita pelo modelo de Moore, e, de forma análoga, um modelo descrito pelo modelo de Moore pode ser representado pelo modelo de Mealy. Porém, é importante ressaltar que o modelo de Mealy (Figura 2.2) tem três estados e o modelo Moore (Figura 2.3) tem quatro estados, isto porque, em geral, as máquinas descritas pelo modelo de Moore contêm mais estados que a correspondente de Mealy.

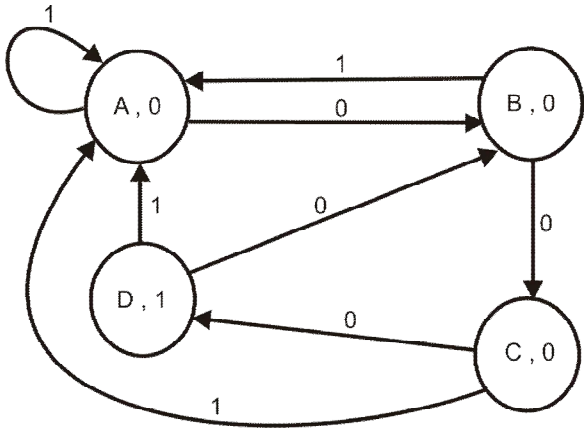

Figura 2.3. Diagrama de transição de estados descrito pelo modelo de Moore.

Como mencionado anteriormente, a modelagem de um circuito sequencial pode ser feita por um diagrama de transição de estados, onde, tal diagrama define as transições de estados e suas saídas quando submetidas às entradas. Com a utilização de elementos de memória chamados *flip-flops*, é projetado um circuito capaz de operar de acordo com as especificações desejadas. Os estados desses elementos de memória são associados aos estados da máquina

por um processo chamado alocação de estados. Cada estado é representado por um código binário único.

O problema da alocação de estados é bastante importante no que diz respeito ao desempenho e custos dos circuitos finais, já que, diferentes alocações podem dar origem a diferentes circuitos, ou seja, um circuito contendo um número maior ou menor de operadores lógicos. Existem muitos trabalhos relevantes que tratam o problema de alocação de estado, como, por exemplo, o trabalho de Santos (2005), desenvolvido na FEIS (Faculdade de Engenharia de Ilha Solteira), onde é desenvolvido e comparado com outros algoritmos existentes, um algoritmo genético modificado com propriedades de substituição. O algoritmo desenvolvido apresentou bons resultados de alocação de estados em relação aos algoritmos comparados.

A sistematização e automação do projeto de circuitos digitais é extremamente conveniente, pois retira do projetista as tarefas cansativas, sujeitas a erros, para permitir-lhe uma análise crítica das soluções possíveis.

2.2. Conversores de dados

Um sinal é uma quantidade ou qualidade física, capaz de transmitir informações. Embora muitos sinais possam originar de sistemas não eletrônicos, tais como os sinais gerados de forma biológica, mecânica e fontes acústicas, geralmente é possível representar uma quantidade física, por meio de sinal elétrico, como uma corrente ou tensão.

Um sinal digital binário aparece como uma série de símbolos, podendo ser expresso apenas com duas possibilidades: 0 ou 1 (pulso alto ou pulso baixo). Para que esses valores sejam entendidos, as tensões estão numa faixa preestabelecida, como por exemplo, para lógica TTL (*Transistor-Transistor Logic*) os valores de tensão compreendidos entre 0 V e 0,8 V representam nível lógico 0 e valores compreendidos entre 2 V e 5 V representam nível lógico 1. Os valores exatos de tensão não são importantes, já que todos os circuitos digitais possuem o mesmo comportamento para as tensões que estão dentro da faixa estabelecida.

Já nos sistemas de sinais analógicos, as tensões elétricas podem assumir quaisquer valores, sendo que deve ser medido o seu valor exato, caso contrário, os resultados estariam incorretos. Por exemplo, um determinado sensor de temperatura ambiente converte uma variável física em uma tensão elétrica no valor de 3,02 V, o que pode representar uma temperatura de 30,2 ºC. Se a tensão convertida fosse de 2,52 V, isso representaria uma

temperatura 2,52 ºC. Dessa forma, um sinal analógico nada mais é do que uma tensão ou corrente que varia em função de um tempo discreto.

Os sinais analógicos podem apresentar problemas de transmissão quanto a ruídos ou distorções do sinal em um sistema de transmissão, ou quando o sinal analógico sofre alguma operação como a de multiplicação, a precisão com a qual o sinal é reconhecido é reduzida.

Esses problemas do sinal analógico podem ser compensados com as vantagens adquiridas com a representação digital, que é bem menos sensível às imperfeições na transmissão (distorção, ruído). No sistema digital, somente algumas características como amplitude e duração são levadas em consideração. Além disso, o sinal digital é mais fácil de ser armazenado e transmitido, por outro lado, a faixa requerida para transmitir a informação digital é muito grande comparada com a representação da informação analógica.

Conversores de dados são circuitos que transformam um dado de uma representação de sinal para outra forma de representação. Dessa forma, os conversores ADCs (*Analog to Digital Conversor*) são usados para converter dados analógicos para a representação correspondente em digital e DACs (*Digital to Analog Conversor*) realizam o trabalho inverso, convertendo um sinal digital para um sinal analógico proporcional ao valor digital. Neste trabalho, foram levados em consideração somente os conversores do tipo DAC, os quais foram utilizados como estudo de casos e na avaliação da ferramenta MS^2SV.

Um conversor DAC recebe uma quantidade *n* de entradas binárias representando um valor decimal codificado em binário, ou seja, existe 2^n possibilidades de combinações binárias na entrada. Existe ainda uma entrada adicional usada como entrada de referência, representada por V_{ref}, que serve para determinar o fundo de escala do nível de saída, ou o valor máximo que o conversor pode gerar em sua saída. O sinal é então convertido em uma saída analógica representando o valor decimal de entrada, sendo o valor analógico gerado pela soma ponderada das *n* entradas. Porém, a saída de um DAC não é realmente uma saída analógica, já que ela apenas assume valores específicos enquanto V_{ref} for constante, caracterizando uma saída pseudoanalógica, no entanto, quanto maior for a quantidade de bits de entrada, mais próximo será o valor da saída de uma quantidade analógica real.

As entradas são ponderadas de acordo com a posição de cada bit, sendo que, o LSB (bit menos significativo) recebe o peso $2^0=1$, consecutivamente, os pesos dos próximos bits recebem os valores $2^1=2$, $2^2=4$, $2^3=8$, até chegar ao MSB (bit mais significativo). Dessa forma, conclui-se que os pesos são sucessivamente dobrados a partir do LSB.

Um DAC possui um atributo chamado resolução ou tamanho do degrau que representa a menor variação de valor que a saída analógica pode assumir, funcionando como uma escada

numa escala crescente de valores. O tamanho do degrau é sempre igual ao valor do peso atribuído ao bit menos significativo.

O valor de referência (V_{ref}) é um valor fixo, isso implica que, quanto maior for a quantidade de entradas menor será o tamanho do degrau da escala e quanto maior for o número de entradas também será maior o custo de produção do DAC. Por isso, é muito importante o uso somente das entradas que forem realmente necessárias.

Quando as entradas são enviadas de forma paralela, implica numa maior velocidade de conversão, mas, em compensação, ocupam uma quantidade maior de bits de entrada. Assim, alguns DACs trabalham de forma serial, enviando um bit de cada vez para o conversor, o que produz uma economia significativa de bits de entrada, porém, provoca uma perda no tempo de conversão.

O sinal pode ser muitas vezes convertido em uma amplitude positiva ou negativa e deve ser capaz de distinguir entre dois valores iguais, mas com amplitudes diferentes. Para isso, os DACs podem ter circuitos internos extras e aceitar números com sinal na forma de complemento de 2. Esse método tem vantagens sobre outros métodos, pois, em sinais que atravessam a faixa do zero, podem causar a mudança de todos os bits, o que pode resultar em graves erros. Um número representado por *n* bits, pode ser representados por *n+1* bits, a quantidade continua a mesma, mas o tamanho da escala completa dobrou, estes são os chamados DACs bipolares.

Existem vários métodos para implementar a operação de um DAC. Um dos métodos é a utilização de uma chave semicondutora em cada entrada para controlar os níveis de tensão das entradas. Essa chave é acionada através da própria entrada e controlada através do sinal de referência. Quando a entrada estiver em nível lógico alto (1), a chave é fechada contendo o sinal do resistor e liberando o sinal da fonte de referência, já que a chave está ligada tanto no resistor como na entrada de referência. Quando o sinal de entrada estiver em nível lógico baixo (0), a chave é aberta para receber o sinal direto do resistor, pois, a tensão do sinal de referência é mais precisa do que a tensão vinda do resistor para fornecer a entrada do DAC. Nesse contexto, o tamanho do degrau é proporcional à maneira que o sinal de referência também for modificado. Esse modelo de DAC possui uma grande limitação para a atual tecnologia de circuitos integrados, pois, com um número maior de entradas e menor valor de degrau torna-se difícil projetar resistências que mantenham um nível de precisão com qualidade.

Uma alternativa para este problema são as malhas R/2R, onde apenas dois valores para os pesos são usados: R e 2R, e a corrente de saída depende das chaves que são controladas pelas entradas.

A equação 2.4 descreve o valor da saída analógica para a malha R/2R para o caso de um conversor com resolução igual a 8 bits, ou seja, $2^8 = 256$, o que implica com que a tensão de referência seja dividida por 256:

$$V_{out} = \frac{-V_{ref}}{256} * n \tag{2.4}$$

onde *n* são as entradas digitais, podendo variar de 0 a 255, ou seja, de $(00000000)_2$ a $(11111111)_2$. Na Figura 2.4, ilustra-se o modelo básico de funcionamento da malha R/2R.

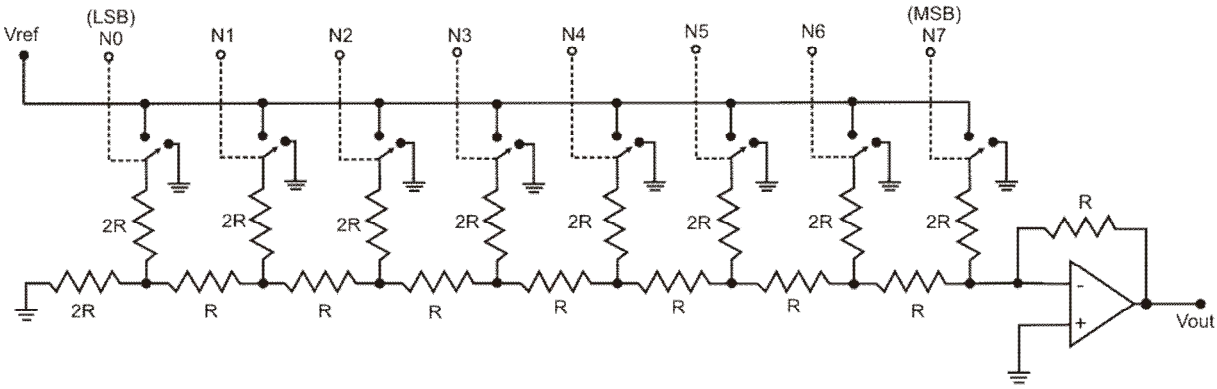

Figura 2.4. Modelo básico da malha R/2R.

Para especificar a precisão de um DAC, os fabricantes utilizam diversas formas, sendo as mais comuns o erro de fundo de escala e o erro de linearidade, em que ambas são expressas como uma porcentagem da saída de fundo de escala. O erro de fundo de escala é o valor máximo de desvio que pode haver na saída do conversor a partir do seu valor ideal, e o erro de linearidade é o valor máximo de desvio que pode haver no tamanho do degrau, também a partir do seu valor ideal.

Quando todos os bits assumem o valor 0, tecnicamente, o valor analógico da saída também será 0 V, mas, na prática, isso não acontece. Existe uma tensão bastante pequena na saída, mesmo quando todas as entradas são 0, esse é o chamado erro de *offset*. Esse erro é somado na saída do conversor em todos os casos de entrada, daí a importância de sua correção. Muitos DACs possuem um ajuste de *offset* externo, permitindo zerar o *offset*.

Os DACs são disponibilizados pelos fabricantes em forma de CI (Circuito Integrado) e possuem uma aplicabilidade muito grande. Podem ser utilizados em sistemas de controle, no

qual é possível converter sinais digitais provenientes de um microcomputador para controlar motores, temperatura, aparelhos sonoros, ou seja, qualquer variável física.

Os DACs podem apresentar diversos problemas quanto ao seu funcionamento correto, no entanto, é possível fazer alguns testes para avaliar esses problemas. Como, por exemplo, o teste de escada, que consiste em analisar se a saída aumenta degrau a degrau, conforme a entrada é incrementada. Esse teste pode ajudar a detectar tanto defeitos internos como externos de um DAC (TOCCI; WIDMER; MOSS, 2003).

2.3. Códigos de linha

O avanço dos circuitos eletrônicos possibilitou o crescimento de vários setores da economia como, por exemplo, o setor de telecomunicações. O surgimento dos novos serviços de telecomunicações digital, fez surgirem vários códigos de linha, que atuam sobre certa quantidade de bits conhecida como palavra, alterando esses bits e transportando-os em um sistema de transmissão. Nessa codificação, o objetivo é preservar a faixa de frequência dos pulsos originais, otimizando o consumo de energia na transmissão do sinal ou até mesmo fornecendo mecanismos de sincronismo no próprio sinal. O funcionamento básico dos códigos de linha utilizados como estudo de caso neste trabalho é apresentado nas subseções seguintes.

2.3.1. Código AMI

O código AMI (*Alternative Mark Inversion*) é utilizado para eliminar o nível da corrente contínua na linha de transmissão. No AMI, os bits 0 são transmitidos como 0 e os bits 1 são transmitidos como +1 ou -1, de forma inversa, a polaridade do bit anterior, conhecida como violações de código, podendo ser positivas ou negativas. Porém, o AMI é limitado no que diz respeito ao sincronismo do *clock* (KEOGH, 1984). Neste trabalho, foi utilizado um bit adicional para representar a alteração ou não da polaridade, ou seja, quando for 0 representa a não alteração de polaridade e 1 representa a alteração da polaridade.

O exemplo, a seguir, demonstra o funcionamento do código AMI:

Sequência de entrada: 1 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1

Sequência codificada: +1 0 -1 0 +1 -1 0 0 0 +1 0 -1 +1 0 0 -1 0 +1 0 0 0 -1 +1

O código AMI pode ser representado em forma de diagrama de transição de estados, descrita através do modelo de Moore, conforme pode ser observado na Figura 2.5.

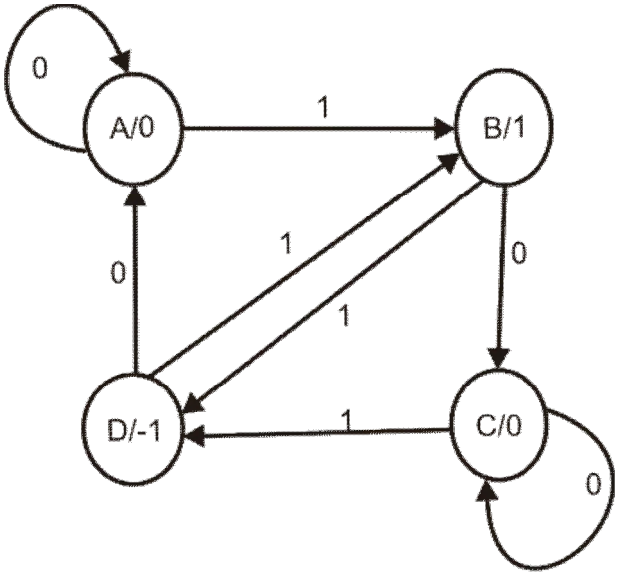

Figura 2.5. Diagrama de transição de estados descrevendo o código AMI.

2.3.2. Código HDB1

O código de linha HDB1 (*High Density Bipolar of order 1 code*) foi o primeiro código da família HDB, sendo muito parecido com o código AMI com pequenas modificações. Quando ocorrem bits 0 isolados, ou seja, entre dois 1, a codificação é feita da mesma forma que o código AMI (o 0 é transmitido como 0). Porém, quando ocorrem dois bits 0 consecutivos eles são transmitidos como +1+1, se o último sinal antecessor for -1, e são transmitidos como -1-1, caso o último sinal antecessor for +1 (TANCREDO, 2002). De acordo com o exemplo a seguir:

Sequência de entrada: 1 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1
Sequência codificada: +1 0 -1 0 +1 -1 +1 +1 0 -1 0 +1 -1 +1 +1 -1 0 +1 -1 -1 +1 +1 -1

Na Figura 2.9, ilustra-se o diagrama de transição de estado para o código HDB1, descrito pelo modelo de Mealy.

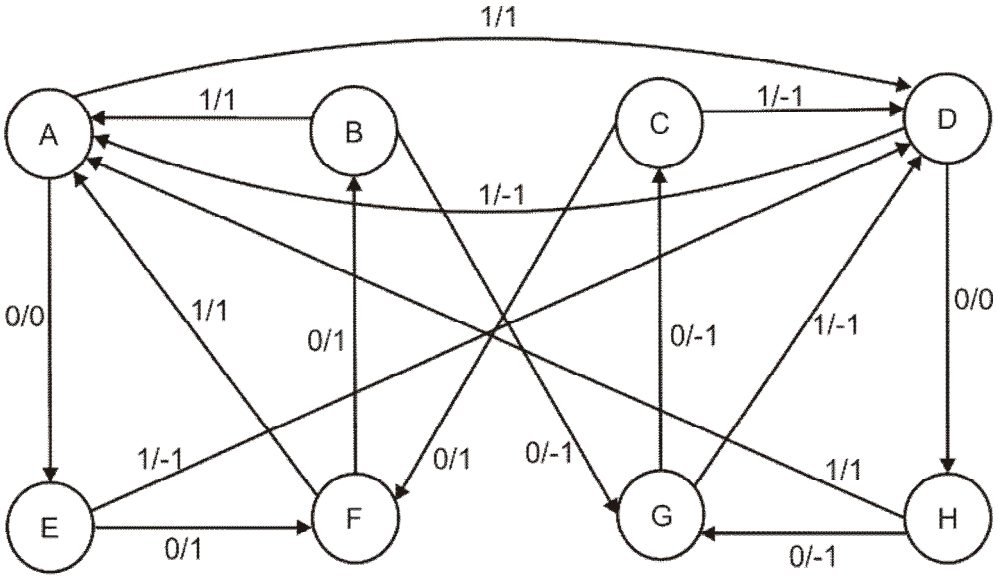

Figura 2.6. Diagrama de transição de estados descrevendo o código HDB1.

2.3.3. Código HDB3

O código de linha HDB3 (*High Density Bipolar of order* 3 *code*) é uma técnica de detecção de sincronismo, com sinalização bipolar e vem sendo utilizada ao longo dos anos em sistemas de linhas digitais primárias de 2 Gbit/s.

O HDB3 é uma derivação do código AMI (*Alternative Mark Inversion*), utilizando as mesmas regras de violação de pulso. O HDB3 procura remediar o efeito de uma palavra comprida com mais de quatro 0 alterando a codificação de saída, para que não ocorra perda de informação na sincronização do *clock*, codificando quatro 0 consecutivos. Assim, o primeiro bit 0 da sequência será codificado como 0, se a marca precedente do sinal tiver polaridade oposta à violação precedente, ou será codificado como marca sem violação (+1 ou -1), caso as marcas de violação precedente tiverem a mesma polaridade. O segundo e terceiro bits 0 da sequência são codificados como 0. O último ou quarto bit da sequência é codificado com uma marca e a polaridade deve ser a violação do código AMI.

O exemplo a seguir ilustra o funcionamento do código HDB3:

Sequência de entrada: 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 1
Sequência codificada: +1 -1 0 0 0 -1 +1 -1 0 0 0 -1 +1 -1 0 0 0 -1 +1 -1

O codificador necessita de três partes de memória para armazenar e identificar os quatros 0_2 sucessivos. É necessário também um *flip-flop* para armazenar a polaridade do pulso (0_2 para + ou 1_2 para -) e, finalmente, outro *flip-flop* para armazenar a polaridade da violação.

Assim, o codificador requer um mínimo de 2^5=32 estados (KEOGH, 1984). Na Figura 2.6, ilustra-se o diagrama de transição de estados do código HDB3, descrito pelo modelo de Mealy, em representação hexadecimal.

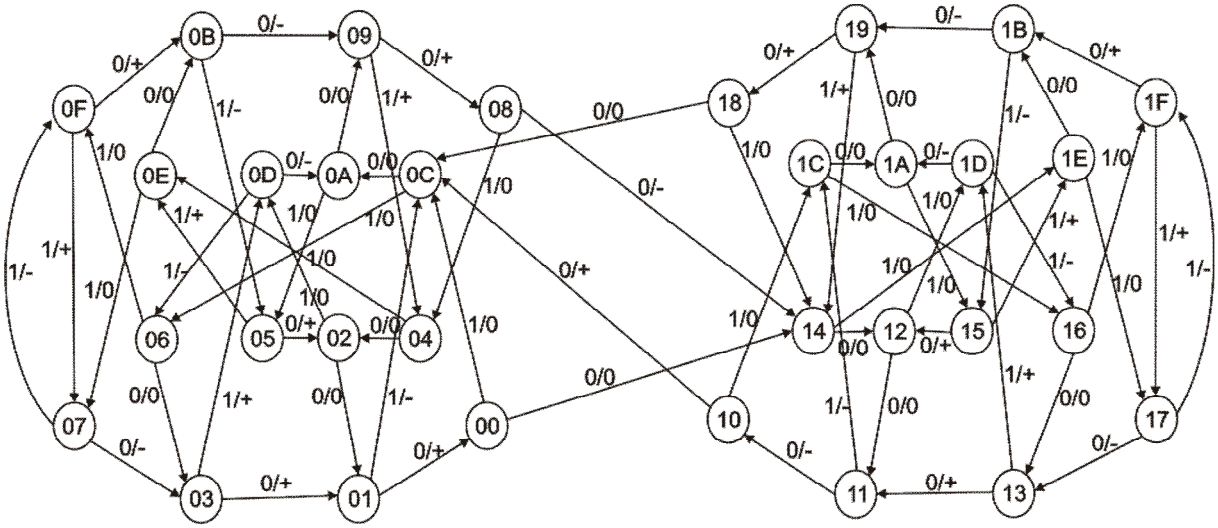

Figura 2.7. Diagrama de transição de estados descrevendo o código HDB3.

2.3.4. Código MLT-3

O código MLT-3 (*Muti Level Transmission 3*) é um código utilizado para melhorar a eficiência de uma rede de transmissão FDDI (*Fiber Distributed Data Inferface*). A rede FDDI utiliza-se de cabos de fibra ótica como meio de transmissão em uma arquitetura em anel duplo.

O diagrama de transição de estados do código de linha MLT-3, descrito pelo modelo de Mealy, é apresentado na Figura 2.7. De acordo com a figura, é possível observar que o código MLT-3 varia entre +1, -1 e 0, de modo que possa provocar um equilíbrio entre as tensões (TANCREDO, 2002). Quando ocorrem bit 0 nada acontece, se o bit for 1 e o sinal anterior for positivo, subtrai-se 1 e, quando o bit for 1 e o sinal anterior for negativo, soma-se 1, de acordo com o exemplo a seguir:

Sequência de entrada: 0 1 0 1 1 0 1 1 0 0 1 1 1 0 0 1 1 0 1 1

Sequência codificada: 0 +1 +1 0 -1 -1 0 +1 +1 +1 0 -1 0 0 0 +1 0 0 -1 0

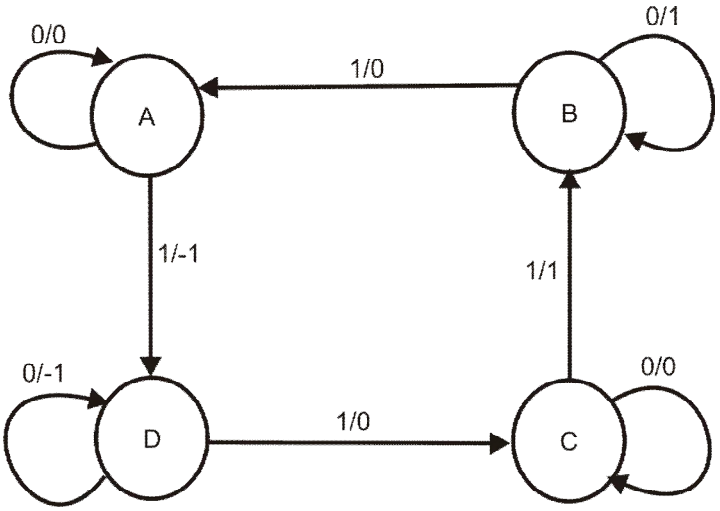

Figura 2.8. Diagrama de transição de estados descrevendo o código MLT-3.

2.3.5. Código 2B1Q

A função do código 2B1Q (*Two Binary, One Quaternary*) é converter a transmissão de um grupo de dois bits em um símbolo quaternário, como também o seu inverso, um símbolo quaternário em um conjunto de dois bits. Dessa forma, quando existem os bits $(10)_2$ no sinal, eles são codificados em +3, quando existem os $(11)_2$, eles são codificados em +1, quando existem os bits $(01)_2$, eles são codificados em -1, e, por fim, quando existem os bits $(00)_2$, eles são codificados em -3, de acordo com a Tabela 2.1 (TANCREDO, 2002).

Tabela 2.1. Regra de codificação do código 2B1Q.

1º bit (bit de polaridade)	2º bit (bit de magnitude)	Símbolo quaternário
1	0	+3
1	1	+1
0	1	-1
0	0	-3

Na Figura 2.8, apresenta-se o diagrama de transição de estados proposto para o código 2B1Q, descrito pelo modelo de Moore. No diagrama de transição de estados, as entradas e as saídas estão representadas em notação decimal, ao invés da notação binária.

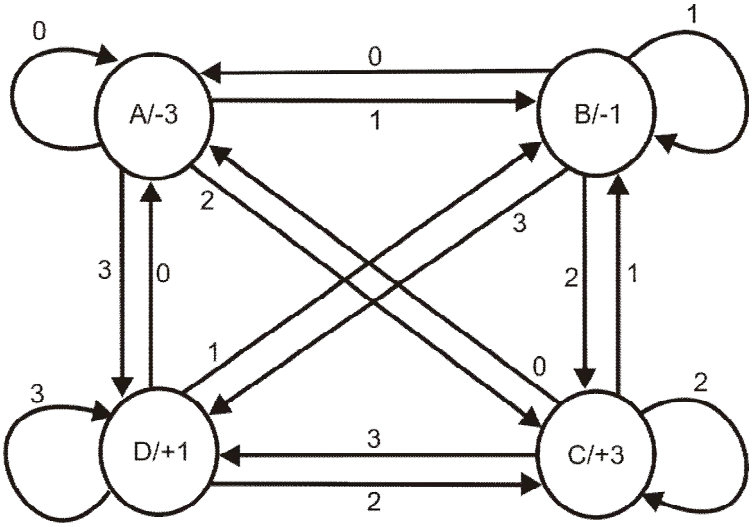

Figura 2.9. Diagrama de transição de estados descrevendo o código 2B1Q.

2.4. Linguagens de descrição de Hardware (HDLs)

As linguagens de descrição de *hardware* ou HDLs (*Hardware Description Language*) são linguagens criadas para descrever o comportamento de dispositivos e processos em uma tarefa comumente chamada de modelagem. As HDLs vêm sendo usadas desde a década de 60 para modelar e simular diversas aplicações, sejam elas digitais e / ou analógicas, concentração de fluidos em processos químicos, e muitos outros modelos. HDLs podem ser divididas em sistemas digitais, sistemas analógicos e sistemas de sinais mistos.

HDLs tem como objetivo resolver alguns dos problemas relacionados às fases de desenvolvimento de um projeto de circuitos eletrônicos. Como, por exemplo, o tempo e o esforço empregado no desenvolvimento, a capacidade de reutilização do projeto, como também a divisão do projeto em partes menores entre os envolvidos.

Assim, as HDLs permitem que o projeto seja dividido em projetos menores, fornecendo a possibilidade de verificação e checagem contínua do desempenho e comportamento do sistema, se tornando em uma linguagem comum entre os integrantes e as ferramentas utilizadas no desenvolvimento do sistema (RIESGO; TORROJA; LA TORRE,1999).

As HDLs também permitem ao projetista desenvolver sistemas de circuitos eletrônicos sem a necessidade do conhecimento de detalhes da tecnologia de implementação, facilitando, dessa forma, as modificações e a implementação do sistema. Elas são amplamente utilizadas no processo de síntese de circuitos eletrônicos, sendo disponibilizadas pelas ferramentas de síntese e como alvo de muitas pesquisas científicas.

Nos tópicos 2.4.1, 2.4.2 e 2.4.3, são descritas algumas das características básicas das linguagens: VHDL, Verilog HDL e VHDL-AMS (respectivamente).

2.4.1. VHDL

A VHDL (*VHSIC - Very High Speed Integrated Circuits - Hardware Description Language*) é uma linguagem de descrição de *hardware* usada para descrever o comportamento e a estrutura de sistemas digitais. A linguagem VHDL emprega o conceito de paralelismo, em que é possível executar linhas de códigos diferentes no mesmo instante de tempo, diferente do que ocorre com as linguagens de programação de computadores em que é executada uma instrução ou linha de código por vez.

Com relação ao comportamento, podem-se descrever os sistemas digitais em três níveis de abstração: comportamental, funcional ou RTL (*Register Transfer Level*) e estrutural. O estilo estrutural é uma descrição em nível mais baixo de abstração, onde são descritas as ligações dos componentes existentes no modelo. O estilo comportamental consiste em declarações concorrentes para descrever o comportamento de determinados eventos em um circuito e declarações simultâneas para demonstrar um comportamento contínuo. Já o nível funcional ou RTL representa uma ponte entre o estilo comportamental e o estrutural, pois nesse estilo também é descrito o comportamento do modelo, porém a um nível mais baixo de abstração comparado em um modelo puramente comportamental. É possível, ainda, uma combinação entre os três níveis de abstração citados.

É importante ressaltar que a VHDL foi desenvolvida para ser independente da tecnologia utilizada, o que significa que, se um modelo específico for implementado em uma tecnologia utilizada nos dias atuais, este também poderá ser implementado em tecnologias futuras.

A VHDL pode ser empregada para descrever e simular uma grande variedade de circuitos digitais, em que o grau de complexidade varia de algumas poucas portas lógicas a interconexões de circuitos complexos. Essa linguagem teve início nos laboratórios militares, com o propósito de uniformizar métodos para especificação de sistemas digitais, mas, após ter sido padronizada pelo IEEE (*Institute of Electrical and Electronics Engineers*), passou a ser utilizada pela indústria eletrônica (D'AMORE, 2005).

Um código em VHDL é composto basicamente por uma entidade (*entity*), onde são declarados os portos de entrada e saída, ou seja, a interface para com outros modelos e parâmetros genéricos e uma ou mais arquitetura (*architecture*), a qual descreve o comportamento ou elementos internos do circuito, de acordo com o nível de abstração.

2.4.2. Verilog HDL

A linguagem Verilog HDL (*Hardware Description Language*) foi criada por Phil Moorby na década de 80, já se tornando bastante popular no final da mesma década. A semelhança da linguagem Verilog com a linguagem C foi um fator que contribui para essa popularização, sendo bastante utilizada pelos projetistas tanto para simulação como para síntese de sistemas digitais.

Durante muitos anos, a Verilog HDL foi uma linguagem patenteada e somente em 1995 foi definido o padrão IEEE std. 1364, homologado pelo IEEE, vindo a se tornar de domínio público (OHIRA, 2003).

Os níveis de abstração empregados na descrição de projetos utilizando a Verilog são os mesmos empregados na linguagem VHDL (comportamental, funcional ou RTL e estrutural), ou até mesmo fazendo-se uma combinação entre os níveis, utilizando o conceito de paralelismo, diferentemente das linguagens de programação de computadores assim como a VHDL.

A estrutura básica do código Verilog é formada por um módulo (*module*), que é a unidade básica e única da linguagem, contendo toda configuração dos portos de entrada e saída e também todas as funções que determinam o comportamento e a estrutura do sistema. A declaração dos portos também é obrigatória na sintaxe da linguagem (OHIRA, 2003).

A Verilog é considerada a principal concorrente da linguagem VHDL, possuindo algumas vantagens em relação à VHDL como, por exemplo, simplificação do código, o tempo de compilação e, consequentemente, de simulação são menores. A síntese da linguagem Verilog em níveis mais baixos de abstração também é mais eficiente comparada à VHDL, porém, a VHDL possui uma quantidade maior de recursos para a modelagem em níveis mais altos de abstração, principalmente no que diz respeito à utilização de dados abstratos, já que a Verilog não utiliza bibliotecas em seus modelos como a linguagem VHDL (OHIRA, 2003).

2.4.3. VHDL-AMS

Com o avanço dos projetos capazes de operar com sinais mistos, surgiram novas HDLs que trabalham com sinais digitais, analógicos e mistos. No ano de 1999, o IEEE criou o padrão IEEE std. 1076.1. Esse padrão define a sintaxe e a forma de trabalho da linguagem VHDL-AMS (*VHSIC Hardware Description Language – Analog Mixed Signals*). O VHDL-AMS tem a sensibilidade de operar com sinais digitais (incorporado do VHDL), analógicos e sinais mistos, tanto no âmbito comportamental, como no estrutural (CHRISTEN; BAKALAR,

1999). A escolha da linguagem VHDL-AMS foi motivada por se tratar de uma linguagem de descrição de *hardware* padrão, disponível em muitos dos ambientes de síntese comercial.

Um modelo VHDL-AMS é uma extensão da linguagem VHDL e também consiste em uma entidade e uma ou mais arquiteturas. Ele pode ser codificado usando um nível de descrição estrutural, comportamental, funcional ou RTL, ou uma combinação desses níveis.

A linguagem VHDL-AMS é capaz de trabalhar não só com sistemas digitais e analógicos, mas também com sistemas térmicos, mecânicos, hidráulicos, pneumáticos, entre outros.

A VHDL-AMS introduz uma nova classe de objetos chamados *quantity*, para representar um valor desconhecido em uma equação algébrica e diferencial (DAE – *Differential Algebric Equations*). Uma *quantity* pode ser um escalar ou um composto (vetores e registros), mas deve ter um sub-elemento escalar do tipo ponto flutuante. É permitido o objeto *quantity* possuir qualquer valor em uma expressão particular, além de outros elementos que foram adicionados à sintaxe da linguagem, que permite a modelagem de sistemas de sinais mistos (CHRISTEN; BAKALAR, 1999).

No Capítulo 3, apresentam-se as ferramentas de *software* utilizadas no desenvolvimento deste trabalho, ou seja, os ambientes de modelagem e simulação utilizadas. Também são apresentados modelos de conversores de dados utilizados como estudo de casos na avaliação do programa MS^2SV.

Capítulo 3. Ferramentas utilizadas

Neste capítulo, serão apresentados de forma sucinta os ambientes comerciais utilizados no desenvolvimento deste trabalho, como *softwares* e conversores de dados utilizados na avaliação da metodologia proposta. Dessa forma, na seção 3.1 são feitos alguns comentários sobre o ambiente Matlab®, pois os modelos descritos no mesmo são traduzidos pelas ferramentas desenvolvidas. Na seção 3.2, é apresentado o ambiente SystemVision™ e suas principais características. Na seção 3.3, são apresentadas as especificações técnicas dos conversores de dados, utilizados como estudo de casos, de acordo com cada manual disponibilizado pelos respectivos fabricantes. Nas seções 3.4 e 3.5, são apresentadas duas ferramentas utilizadas para validar a ferramenta SF^{2}HDL, o programa TABELA e o programa TAB2VHDL. Na seção 3.6, apresentam-se as características e o funcionamento da primeira versão da ferramenta MS^{2}SV.

3.1. O ambiente Matlab®

O Matlab® é um ambiente voltado para a análise numérica integrando cálculo com matrizes, processamento de sinais e construção de gráficos, cujos problemas e soluções são expressos somente como eles são escritos matematicamente, ao contrário da programação tradicional. O Simulink® é o seu principal *toolbox*, composto por uma grande variedade de componentes e interface visual. O Matlab® é utilizado principalmente para executar cálculos matemáticos, visualizando e analisando os dados e escrevendo novos programas de computadores. Já o Simulink® é empregado na simulação, análise e no desenvolvimento de sistemas dinâmicos complexos, sejam eles contínuos, discretos ou híbridos. O Simulink® possui uma grande variedade de ferramentas agrupadas denominadas *toolboxes*, que possuem uma interface gráfica que facilita a utilização por parte do projetista (KARRIS, 2007). A motivação para o uso do Simulink® na modelagem de projeto em alto nível deu-se pelo fato de ser um ambiente padrão usado, por exemplo, na área de controle.

Outro *toolbox* disponível no Matlab® é o Stateflow® que pode ser considerado como uma extensão do Simulink®. É uma importante ferramenta utilizada na modelagem e simulação de máquinas de estados finitos. O Stateflow® permite tanto a modelagem de máquinas de Mealy quanto às máquinas de Moore e ainda permite uma descrição clássica,

nativa da própria ferramenta, como também a modelagem de projetos hierárquicos. A modelagem pode ser feita de forma independente utilizando somente o Stateflow®, como também pode haver uma interação com o Simulink® e o Matlab®. O Stateflow® fornece os objetos gráficos necessários para construir uma máquina de estados finitos. Assim como no Simulink®, pode-se arrastar e soltar objetos para construir os estados e as transições gráficas, as quais, fluem de forma lógica de um estado para outro (KARRIS, 2007).

O ambiente Matlab® é largamente utilizado no âmbito acadêmico como importante ferramenta de pesquisa, auxiliando nos projetos de maneira rápida e eficiente nas mais diversas áreas de aplicação. Por esse motivo, o ambiente Matlab® foi utilizado neste projeto como ferramenta inicial de modelagem para posterior tradução de seus modelos para as linguagens de descrição de *hardware*.

3.2. O ambiente SystemVision™

O ambiente de modelagem SystemVision™, desenvolvido pela Mentor Graphics, possui a capacidade de modelagem de projetos mistos (modelos analógicos e digitais) e simulações integrando o processador de projetos DxDesigner com o simulador de sinais mistos ADVance MS. O simulador ADVance MS permite a simulação, análise e verificação através de forma de ondas. O ambiente suporta modelos e técnicas de modelagem de VHDL-AMS, VHDL, C e simulação em SPICE (SILVA, 2007).

Algumas capacidades chaves do sistema de modelagem SystemVision™ são:

- Simulação de modelos VHDL-AMS, SPICE ou a combinação de ambos.
- Os mesmos modelos podem ser uma mistura de subcircuitos SPICE e VHDL-AMS.
- O SystemVision™ fornece um modelo de biblioteca (EDULIB - *Educational Library*) que contém a representação de dispositivos capazes de descrever várias tecnologias, incluindo modelos elétricos (analógico/digital), mecânica, hidráulica, magnética e térmica.
- O SystemVision™ pode gerar símbolos automaticamente, auxiliando o projetista na criação de modelos.
- São suportadas metodologias de projetos hierárquicos e esquemáticos em várias páginas (*sheet*).
- São suportados projetos completos de sinais mistos, incluídos barramentos digitais e analógicos.

- Dados do projeto são organizados e armazenados usando uma abordagem orientada a projetos.
- Pode ser usado o ModelSim (também desenvolvido pela Mentor Graphics) para compilar modelos digitais VHDL em um determinado projeto, permitindo a simulação no ambiente SystemVision™.

O SystemVision™ usa dois tipos de bibliotecas:

- Biblioteca de símbolos: as aplicações do SystemVision™ fornecem uma biblioteca para símbolos (componentes), sendo dividida em categorias: Sistema de Controle, Tecnologia Mista, Digital, Rotacional, Elétrica, Térmica, Hidráulica, Translacional, Magnético, Primitivas SPICE, Sinais Mistos, Semicondutores SPICE e Macro modelos SPICE (SILVA, 2007).
- Biblioteca de modelos: a coleção de modelos em VHDL-AMS é referenciada como EDULIB e também é dividida em categorias como: Sistemas de Controle, Sinais Mistos, Digitais, Tecnologia Mista, Elétrica, Rotacional, Hidráulica, Térmica, Magnética e Translacional (SILVA, 2007).

Na Figura 3.1, é ilustrada a relação dos módulos funcionais usados pelo SystemVision™.

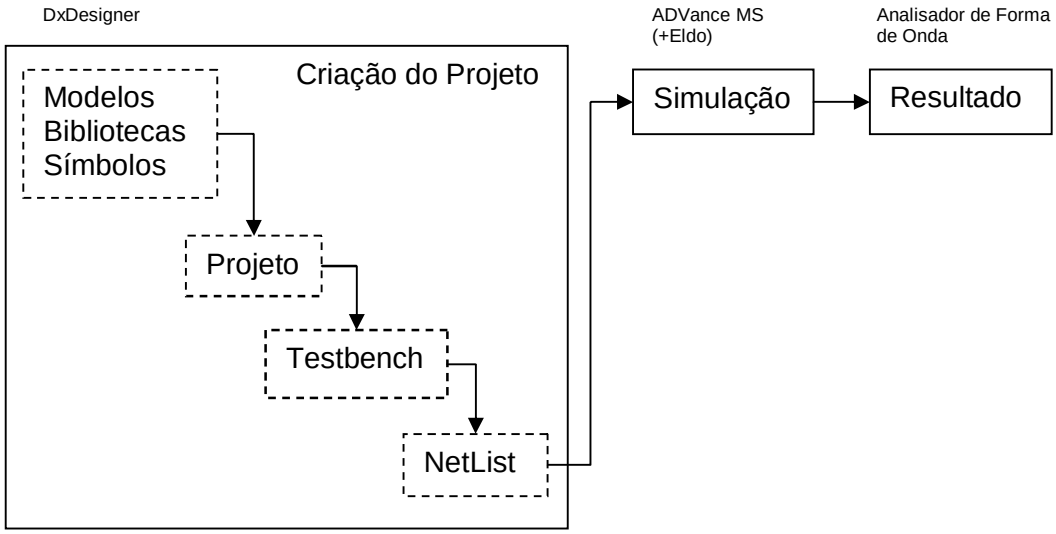

Figura 3.1. Relação dos módulos funcionais do ambiente SystemVision™.

O DxDesigner™ fornece uma interface gráfica ao usuário (GUI – *Graphical User Interface*) para criação e simulação de projetos. O ADVance MS é capaz de simular projetos contendo modelos VHDL-AMS que possuem sinais analógicos e/ou digitais, através da incorporação do simulador Eldo (simulador de sinais analógicos), o qual opera em modelos

em SPICE. Através das opções de *testbench* do projeto ativo no SystemVision™, é possível eleger os seguintes tipos de projetos para serem simulados:

- ♦ Somente modelos SPICE.
- ♦ Somente modelos VHDL-AMS.
- ♦ Modelos SPICE e VHDL-AMS juntos em um *netlist* SPICE.
- ♦ Modelos SPICE e VHDL-AMS juntos em um *netlist* VHDL.

Quando é executada uma simulação no SystemVision™, junto com o *netlist*, é compilado o *testbench* ativo de acordo com a convenção do IEEE para o diretório do VHDL e a estrutura de bibliotecas.

A principal característica do ambiente SystemVision™, que influenciou na sua utilização neste trabalho, é a possibilidade de descrever o modelo usando primitivas disponíveis nas bibliotecas e visualizar o VHDL-AMS criado pelo modelo (SILVA, 2007).

3.3. Conversores de dados utilizados

Para avaliar a nova versão da ferramenta MS^{2}SV, foram utilizados quatro conversores de dados disponíveis comercialmente, cujos modelos foram descritos em Simulink® e traduzidos para a linguagem VHDL-AMS, para serem simulados no ambiente SystemVision™.

3.3.1. DAC08

De todos os modelos de conversores de dados estudados, o DAC08 é o modelo com funcionamento mais simplificado sendo constituído basicamente de um conversor com resolução de 8 bits de entrada paralela, que realiza a conversão digital para analógico. O tamanho compacto e o baixo consumo de energia tornam o DAC08 atrativo para aplicações portáteis e militares / aeroespaciais. Na Figura 3.2, apresenta-se o diagrama funcional do conversor DAC08.

A corrente de saída do DAC08 é obtida pela divisão do valor da entrada digital por 256, já que o conversor possui uma resolução de 8 bits. O resultado dessa divisão é multiplicado pela corrente de referência, podendo ser uma corrente fixa ou que varie na faixa de 0 mA a 4.0 mA. A conversão dos dados digitais em analógicos é feita com a utilização da malha R/2R, onde as entradas binárias controlam o chaveamento entre a corrente vinda dos resistores e a corrente vinda direto da corrente de referência, conforme o apresentado na Figura 3.2 (ANALOG DEVICES INC., 2009).

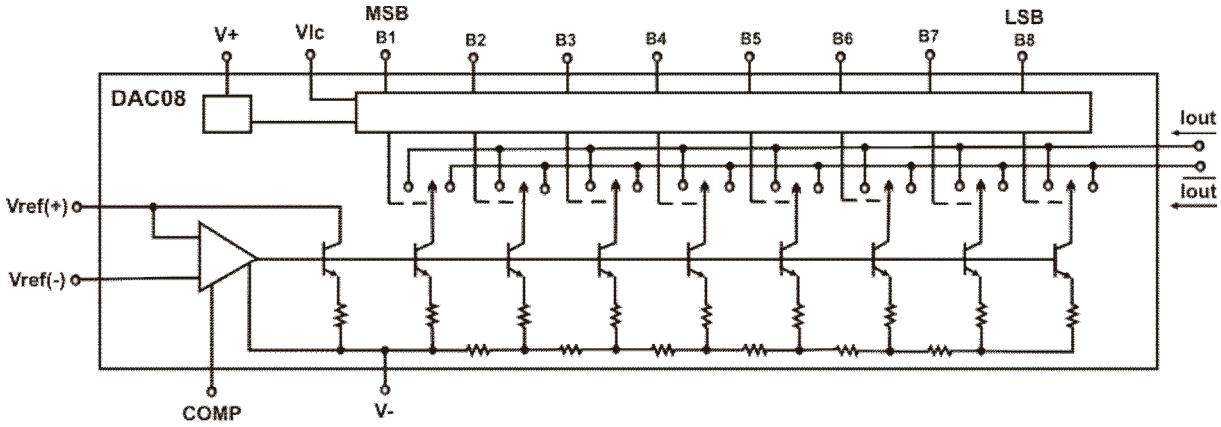

Figura 3.2. Diagrama funcional do conversor DAC08.

Aspectos de caráter técnico, como tecnologia utilizada na construção do CI, consumo de energia, dissipação de calor, foram desconsiderados, pois o foco deste trabalho é somente analisar o funcionamento dos conversores para a sua modelagem em alto nível de abstração.

3.3.2. AD7524

O AD7524 é um conversor de dados do tipo DAC (*Digital to Analog Conversor*). O AD7524 apresenta uma resolução de 8 bits de entrada paralela, utilizando também a malha R/2R na realização da conversão do sinal. O AD7524 é caracterizado por ser utilizado em aplicações de:

- Microprocessadores controlando circuitos de ganho.
- Microprocessadores que controlam circuitos utilizados na redução de corrente alternada.
- Microprocessadores controlando a geração de funções.
- Em circuitos que controlam a geração de ganho automático.
- Instrumentação em barramento estruturado.

A principal diferença entre o DAC08 e o AD7524 é que o AD7524 possui internamente um conjunto de *flip-flops* do tipo *latch* capaz de armazenar a última entrada digital e uma interface lógica capaz de realizar o controle da leitura e do armazenamento dessa entrada digital, tornando seu ciclo de escrita semelhante ao ciclo de escrita da memória RAM.

O modo de seleção é controlado pelas entradas CS_B e WR_B. Quando CS_B e WR_B estão em nível lógico baixo (0), é habilitado o modo de escrita, ou seja, a saída analógica representa o valor binário no barramento de entrada DB_0-DB_7. Já quando CS_B ou WR_B assume o nível lógico alto (1), o conversor está no modo de armazenamento, a saída analógica detém o valor correspondente à última entrada digital presente no DB_0-DB_7 antes de WR_B ou CS_B assumir nível lógico alto.

Na Tabela 3.1, é apresentada a relação das entradas de controle e o modo de seleção do AD7524 e, na Figura 3.3, é apresentado o diagrama funcional do conversor AD7524 (ANALOG DEVICES INC., 2009).

Tabela 3.1. Relação das entradas com o modo de seleção do AD7524.

CS_B	WR_B	Modo de Seleção	Comentários
0	0	**Escrita**	A saída corresponde à atividade no barramento de entrada (DB_0 a DB_7).
1	X	Espera	A saída corresponde à última entrada válida, armazenada nos flip-flops.
X	1		

O AD7524 é formado pela malha R/2R e oito canais de corrente chaveada em um *chip* monolítico. É utilizada uma estrutura de malha R/2R invertida, cuja corrente carregada é chaveada entre as saídas OUT1 e OUT2, assim, a corrente de saída é mantida constante, independente do estado da chave. De acordo com o apresentado na Figura 3.3, o AD7524 possui ainda uma interface lógica que habilita ou não saída analógica e armazena a última entrada válida presente no barramento de dados, conforme o apresentado na Tabela 3.1.

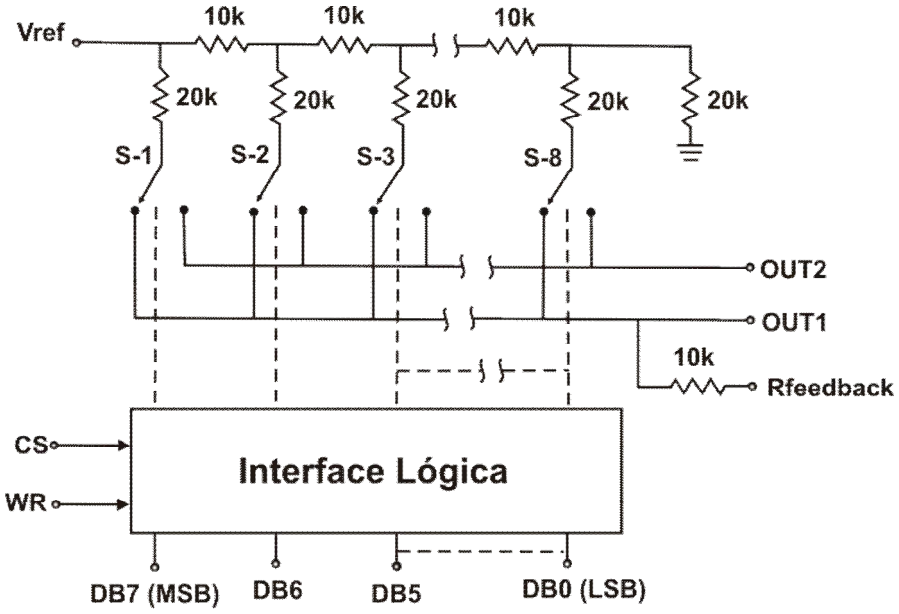

Figura 3.3. Diagrama Funcional do Conversor AD7524.

3.3.3. AD7528

O conversor AD7528 pertence à família do conversor AD7524, também sendo um conversor do tipo DAC, com um barramento de oito bits de entrada de dados. O AD7528 contém dois conversores DAC idênticos, o DAC A e o DAC B. Todos os DAC possuem uma

malha R/2R e um conjunto de *flip-flops* do tipo *latch* para armazenar a última entrada válida, exatamente igual ao conversor AD7524. A entrada de ambos os DACs são provenientes de um mesmo barramento de dados, caracterizando, assim, dois estados distintos, o estado de escrita e o estado de espera em cada DAC independente. Na Figura 3.4, ilustra-se o diagrama funcional do AD7528 (ANALOG DEVICES INC, 2009).

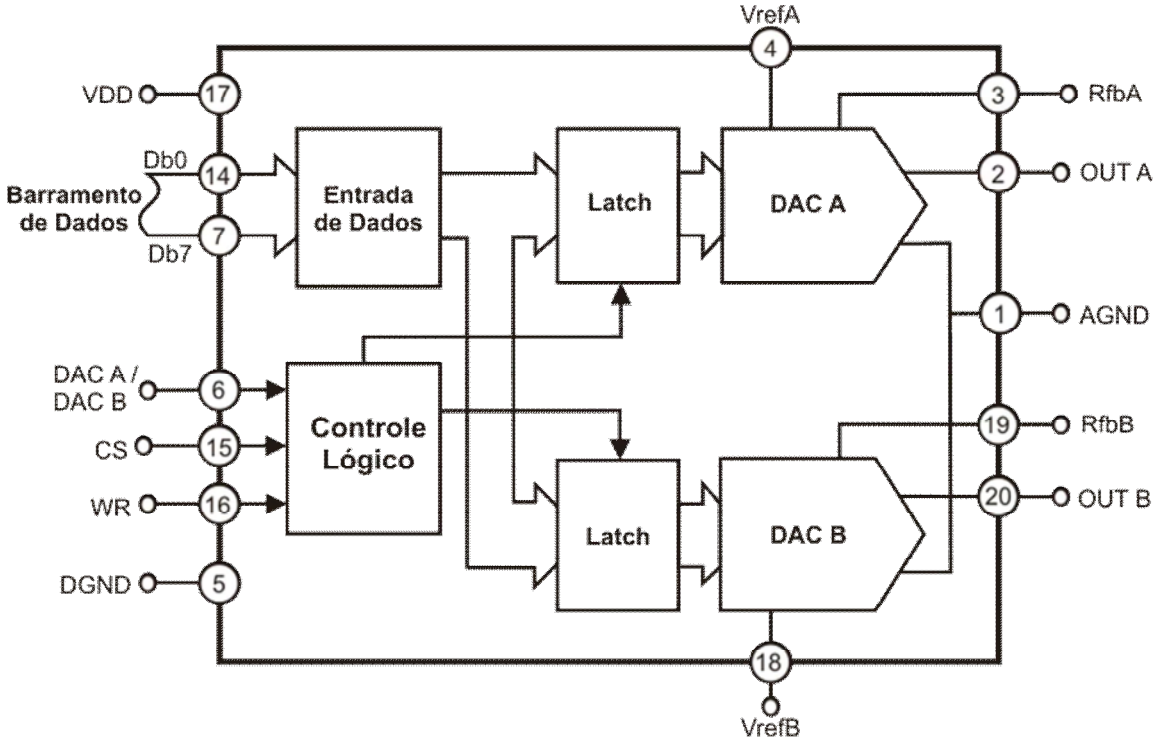

Figura 3.4. Diagrama funcional do conversor AD7528.

O controle de seleção, que especifica qual DAC será utilizado, é feito por um único pino o $\overline{DACA}/DACB$. Quando este está em nível lógico baixo (0), o DAC A é selecionado para escrita e o DAC B está em modo de espera. Quando está em nível lógico alto (1), o DAC B é selecionado para escrita e o DAC A está em modo de espera. O controle do modo de escrita e espera também é feito de forma independente em todos os DACs através de dois pinos, o CS_B e o WR_B. Quando ambos os pinos estão em nível lógico baixo (0), o DAC selecionado está no modo de escrita, portanto, a saída do DAC representa o valor binário no barramento de entrada DB_0 a DB_7. Quando um dos pinos CS_B e WR_B estão em nível lógico alto (1) o DAC selecionado está em modo de espera. Nesse caso, o conjunto de *latches* armazena a última entrada que ocorreu no barramento de dados, antes do DAC entrar em modo de espera. Na Tabela 3.2, é apresentada a relação dos três pinos de controle citados (ANALOG DEVICES INC, 2009).

Tabela 3.2. Relação do modo de seleção do AD7528.

$\overline{DACA}$ / DACB	CS_B	WR_B	DAC A	DAC B
0	0	0	**Escrita**	Espera
1	0	0	Espera	**Escrita**
X	1	X	Espera	Espera
X	X	1	Espera	Espera

3.3.4. AD5450

O AD5450 realiza conversão de sinais digitais em sinais analógicos, com uma resolução de 8 bits, utilizando-se da malha R/2R para realização da conversão. O conversor AD5450 é o mais complexo dos quatro conversores estudados, pois esse conversor trabalha de forma serial ao invés de paralela. A família do conversor AD545n pode ser configurada para trabalhar com resoluções de 8 bits, 10 bits, 12 bits e 14 bits.

Os dados são inseridos em uma palavra de 16 bits, sendo os 2 bits mais significativos utilizados para controle, chamados de C0 e C1, os 8 bits seguintes são os dados (para o caso de uma resolução de 8 bits) e os 6 bits menos significativos são descartados pelo circuito interno de controle (ANALOG DEVICES INC, 2009). A palavra de dados de entrada solicitada pelo AD5450 com uma resolução de 8 bits, de 10 bits, 12 bits e 14 bits, são estruturadas conforme o apresentado na Figura 3.5 (a), (b), (c) e (d), respectivamente.

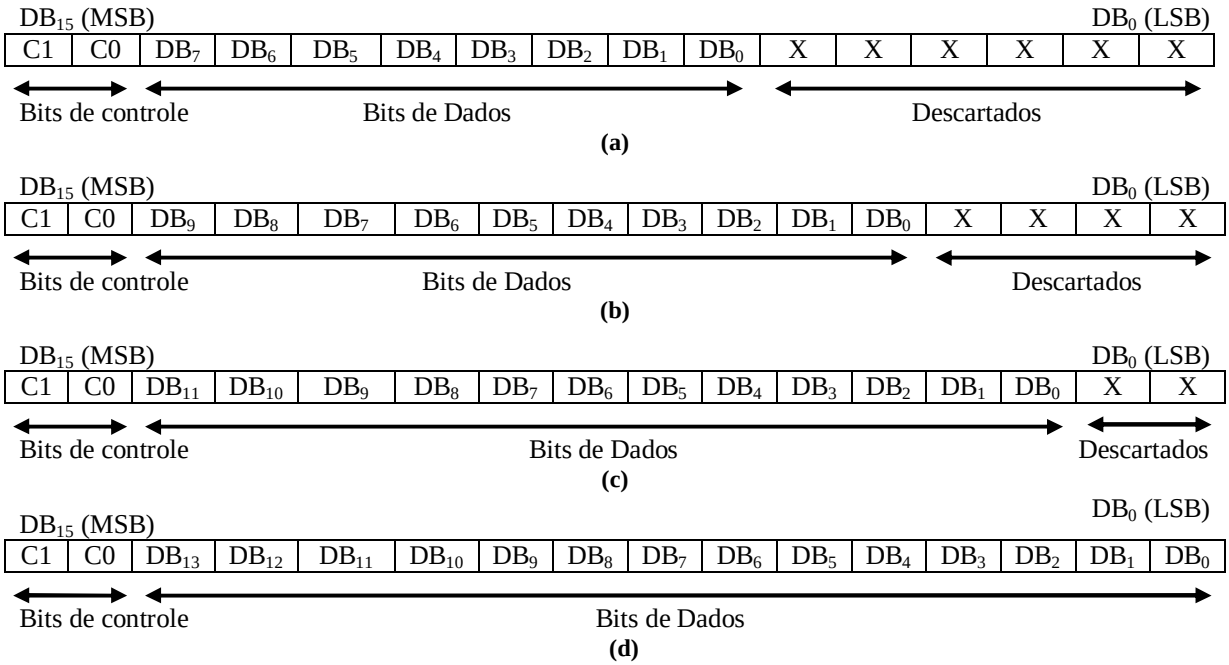

Figura 3.5. Palavra de dados do conversor AD5450.

Os sinais de C1 e C0 são empregados na configuração do conversor, ou seja, quando os dois sinais possuem nível lógico baixo (0), o conversor é configurado para realizar a leitura dos dados na borda de descida do *clock*. Já quando os dois sinais estão em nível lógico alto (1), o conversor realizará a leitura na borda de subida do *clock*. Os níveis lógicos (1) e (0), ou (0) e (1), para os sinais C1 e C0, respectivamente, são reservados para outras implementações na configuração do conversor. Os detalhes são apresentados na Tabela 3.3.

Tabela 3.3. Relação dos sinais de entrada dos pinos C1 e C0.

C1	C0	Função implementada
0	0	Carrega e Atualiza no *clock* de descida (Padrão)
0	1	Reservado
1	0	Reservado
1	1	Carrega e Atualiza no *clock* de subida

A sincronização da conversão é realizada pela função SYNC, a qual é ativada em nível lógico baixo (0). Os dados só podem ser transferidos quando a função SYNC estiver em nível lógico baixo (0) e, assim, é mantido até o décimo sexto pulso de *clock*, quando é terminada a transferência dos dados. Em seguida, o SYNC assume o nível lógico alto (1) e os dados acumulados no registrador de deslocamento (*Shift Register*) são transferidos para o registrador DAC de forma paralela, e só então transferidos para a malha R/2R, para realizar a conversão dos dados obtidos (ANALOG DEVICES Inc, 2009). A Tabela 3.4 apresenta a relação dos níveis empregados no sinal SYNC.

Tabela 3.4. Relação dos sinais de entrada do pino !SYNC.

SYNC	Entrada
0	A entrada é ativada e inicia-se a transferência da palavra até o 16º pulso de *clock*
1	A entrada é desativada

O AD5450 pode ser configurado para operar no modo unipolar ou no modo bipolar ativando a saída do amplificador operacional. Para o caso de uma configuração unipolar, a saída é descrita de acordo com a equação 3.1:

$$V_{out} = \frac{-D}{2^n} * V_{ref} \tag{3.1}$$

onde *D* é a representação fracionária da entrada digital, e *n* é a quantidade de bits, ou seja, a resolução do dispositivo. Para o caso de uma resolução de 8 bits, *D* pode assumir o valor de 0 a 255. Já para o caso de uma saída bipolar, a saída é obtida com a utilização de um amplificador operacional externo adicional e também resistências externas. Nesse caso a tensão de saída é descrita através da equação 3.2:

$$V_{out} = \left(\frac{V_{ref} * D}{2^{n-1}} \right) - V_{ref} \tag{3.2}$$

onde, da mesma forma que no modo unipolar, *D* é a representação fracionária da entrada digital, e *n* é a quantidade de bits, ou seja, a resolução do dispositivo. Para o caso de uma resolução de 8 bits, o D pode assumir o valor de 0 a 255.

Na Figura 3.6 é ilustrado o diagrama funcional básico para o conversor AD5450.

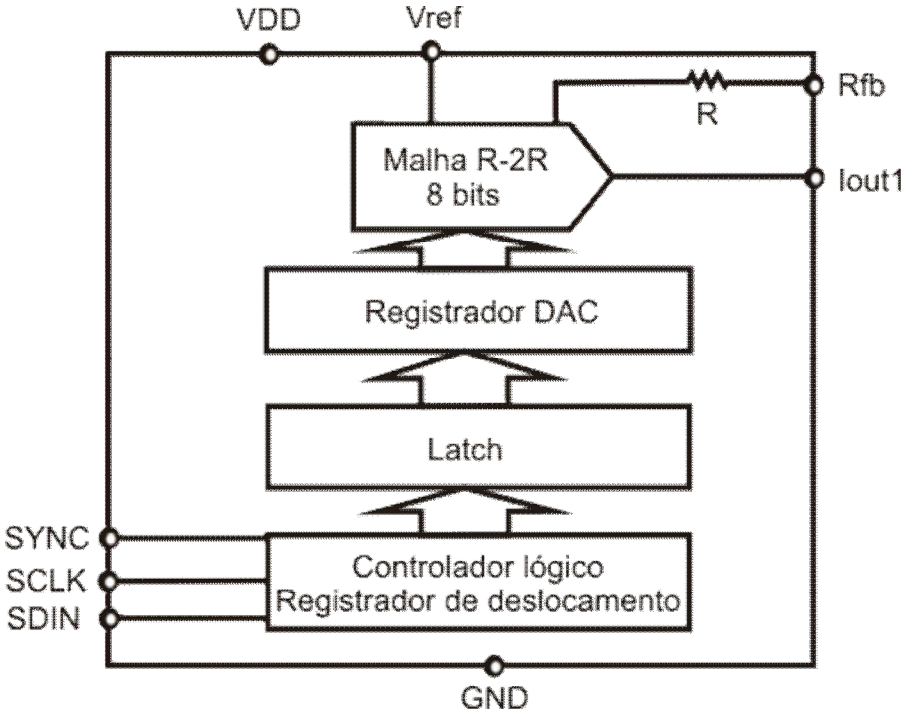

Figura 3.6. Diagrama Funcional Genérico do Conversor AD5450.

O AD5450 é formado basicamente de quatro blocos: um controlador lógico, que realiza a leitura dos dados de forma serial e a sincronização do *clock*, um registrador de deslocamento, que transforma a entrada digital vinda de forma serial em dados paralelos, um conjunto de *latches* para armazenar a entrada digital até que ela esteja completa, e por fim, a malha R/2R que realizara a conversão propriamente dita (ANALOG DEVICES INC., 2009).

3.4. O programa TABELA

O programa TABELA foi desenvolvido para sintetizar máquinas de estados finitos e foi desenvolvido em linguagem Pascal por pesquisadores da UNICAMP (Universidade Estadual de Campinas) (SILVA, 1989). O programa TABELA gera a tabela de transição de estados de uma máquina de estados finitos a partir de seu diagrama de transição de estados e minimiza as funções de transições internas correspondentes aos elementos de memória utilizados e as funções de saída do circuito.

Os dados solicitados pelo programa são: nome do dispositivo de saída de resultados (não deve conter a extensão); número de *flip-flops*; tipo de cada um deles (D ou JK); número de variáveis de entrada; número de variáveis de saída; tabela de próximo estado, na forma: estado atual, próximo estado, entrada e saída. O final da descrição é representado pela notação "-100".

Os estados, as entradas e as saídas devem estar na forma decimal. As máquinas podem ser completa ou incompletamente especificadas. Na Figura 3.7, é ilustrada a estrutura do arquivo de entrada para o programa TABELA, o qual deve ser na forma de tabela de transição de estados.

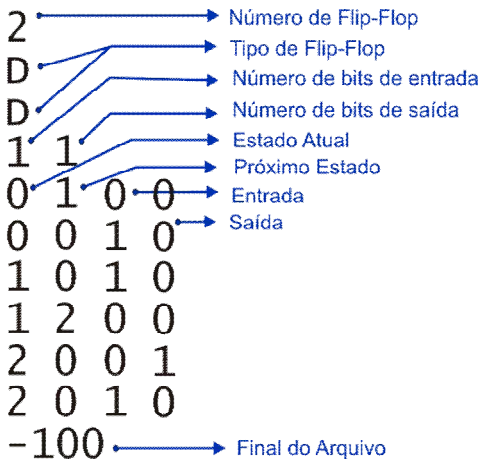

Figura 3.7. Arquivo de entrada do programa TABELA.

O programa monta uma tabela de transição de estados, armazenando-a no arquivo de saída. A partir desta tabela são obtidos os mintermos e os *don't care states* das funções de transição internas de todos os *flip-flops* e da saída do circuito. Utilizando o algoritmo de minimização das funções booleana de Quine-McCluskey, essas funções são obtidas nas suas fórmulas mínimas. O algoritmo de Quine-McCluskey é um método clássico que possui duas fases: a obtenção dos implicantes primos e a cobertura irredutante, que, a partir do conjunto de implicantes primos são obtidos os implicantes essenciais para a realização da função

(SILVA, 1989). Na Figura 3.8, é apresentado o diagrama de bloco funcional que ilustra as fases envolvidas no funcionamento do programa TABELA.

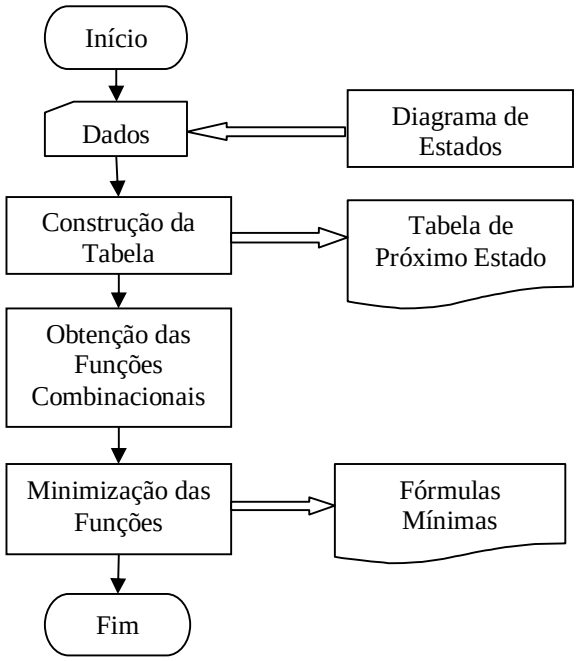

Figura 3.8. Diagrama funcional do programa TABELA.

Após a minimização das funções booleanas contidas na máquina de estados finitos, o TABELA gera um arquivo de texto contendo toda a descrição da minimização obtida e os custos de cada função (em forma de soma de produtos). Para a obtenção de um circuito mínimo, é necessário uma ferramenta que realize uma boa alocação de estados, para a máquina de estados, como o proposto por Santos (2005), já que o programa TABELA não realiza alocação de estados automaticamente.

3.5. O programa TAB2VHDL

O programa TAB2VHDL (*TABELA to VHDL*) foi utilizado neste trabalho para efeitos comparativos com o programa SF^{2}HDL. O TAB2VHDL é responsável por gerar o código VHDL, em domínio funcional, a partir do arquivo gerado pelo programa TABELA, inferindo os *flip-flops* utilizados e gerando o circuito combinacional referente ao comportamento de uma determinada máquina de estados finitos. Inicialmente, o TAB2VHDL faz a leitura do arquivo de texto gerado pelo TABELA e a partir dele é gerado um arquivo com a extensão ".vhd", contendo o código VHDL. Na Figura 3.9 (a) e 3.9 (b), são apresentados um exemplo de arquivo de saída do TABELA, que é usado como entrada pelo TAB2VHDL e um exemplo do arquivo de saída gerado pelo TAB2VHDL, ou seja, um código VHDL funcional

(TANCREDO, 2002). Os dois exemplos representam o circuito detector de três zeros consecutivos sem sobreposição, apresentados na Figura 2.2.

```
     CASO    :  Detector de 3 zeros consecutivos

     DATA    :  2 de julho de 2009

     USUARIO :  Tiago da Silva Almeida

!DE!P/!ENTRADA !MINT!!Q1!Q1+!D1!!Q0!Q0+!D0!!Z0!
! 2! 0! 1 (1)  !   6!! 1!  0! 0!! 0!  0! 0!! 0!
! 2! 0! 0 (0)  !   2!! 1!  0! 0!! 0!  0! 0!! 1!
! 1! 2! 0 (0)  !   1!! 0!  1! 1!! 1!  0! 0!! 0!
! 1! 0! 1 (1)  !   5!! 0!  0! 0!! 1!  0! 0!! 0!
! 0! 0! 1 (1)  !   4!! 0!  0! 0!! 0!  0! 0!! 0!
! 0! 1! 0 (0)  !   0!! 0!  0! 0!! 0!  1! 1!! 0!


DON'T CARE STATES GERAIS :
!DE!P/!ENTRADA !DONT!!Q1!Q1+!D1!!Q0!Q0+!D0!!Z0!
! 3! X! 0 (0)  !   3!! 1! X ! X!! 1! X ! X!! X!
! 3! X! 1 (1)  !   7!! 1! X ! X!! 1! X ! X!! X!

FUNCAO D1
=========
MINTERMOS :  1;
DON'T CARE STATES :  7;  3;
IMPLICANTES PRIMOS ESSENCIAIS :
ESSENCIAL:  1     REDUNDANCIA:       2 ->  0X1
CUSTO FINAL DE D1 = 2

FUNCAO D0
=========
MINTERMOS :  0;
DON'T CARE STATES :  7;  3;
IMPLICANTES PRIMOS ESSENCIAIS :
ESSENCIAL:  0     REDUNDANCIA:       0 ->  000
CUSTO FINAL DE D0 = 3

FUNCAO Z0
=========
MINTERMOS :  2;
DON'T CARE STATES :  7;  3;
IMPLICANTES PRIMOS ESSENCIAIS :
ESSENCIAL:  2     REDUNDANCIA:       1 ->  01X
CUSTO FINAL DE Z0 = 2

CUSTO TOTAL DAS 3 FUNCOES = 7
```

(a) Arquivo gerado pelo programa TABELA

```
ENTITY TESTE IS
     PORT(
     CLK, CLR : IN BIT;
     X0 : IN BIT;
     Q0, Q1 : OUT BIT;
     Z0 : OUT BIT );
END TESTE;

ARCHITECTURE RTL OF TESTE IS

SIGNAL  VE0, VE1: BIT;
SIGNAL D1, D0 : BIT;

 BEGIN

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1' THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE1 <= '0';
    ELSIF CLK'EVENT and CLK = '1' THEN
       VE1 <= D1;
    END IF;
    Q1 <= VE1;
END PROCESS;

  D1 <=  ( NOT(X0) AND (VE0));
  D0 <=  ( NOT(X0) AND NOT(VE1) AND NOT(VE0));
  Z0 <=  ( NOT(X0) AND (VE1));

END RTL;
```

(b) Arquivo gerado pelo programa TAB2VHDL

Figura 3.9. Arquivos utilizados pelo programa TAB2VHDL.

3.6. A ferramenta MS²SV

A ferramenta MS^{2}SV (*Matlab®/Simulink® to SystemVision™*) lê um arquivo contendo o modelo em Simulink® e traduz esse modelo para o código VHDL-AMS estrutural. Ele também cria toda a estrutura de arquivos necessária para simular o modelo traduzido no ambiente SystemVision™ da Mentor Graphics. A ferramenta MS^{2}SV foi desenvolvida em linguagem de programação C e possui um número predefinido de componentes de bibliotecas necessários no processo de tradução (SILVA, 2007).

Durante a tradução, todos os componentes apresentados no modelo Simulink® são identificados na biblioteca de componentes previamente desenvolvida. Os passos envolvidos na metodologia de projeto são descritos a seguir:

- Especificação inicial do modelo e simulação usando Simulink®.
- Conversão do modelo do Simulink® para o modelo VHDL-AMS correspondente.

- Simulação e análise do modelo convertido usando o ambiente SystemVision™.
- Comparação dos resultados de simulação de ambos os modelos.
- Continuação do processo de síntese de partes do código VHDL-AMS, se os resultados da comparação forem aceitáveis, ou retornar para uma nova especificação.

Na Figura 3.10, é ilustrado o diagrama da metodologia utilizada pelo MS^{2}SV no processo de tradução.

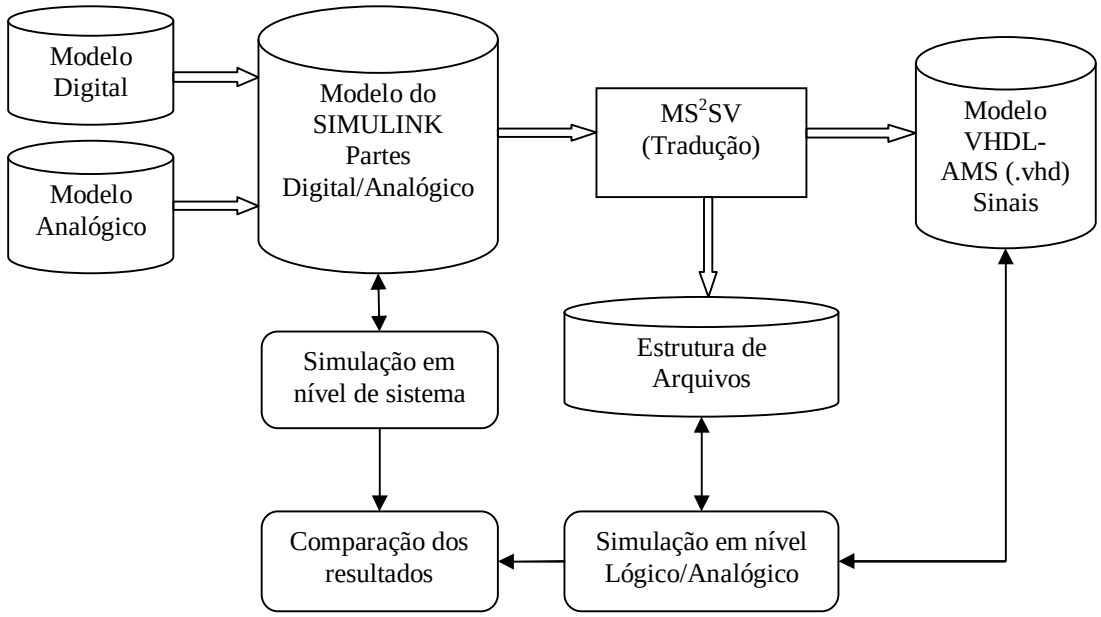

Figura 3.10. Diagrama da metodologia utilizada pelo MS^{2}SV.

Na fase inicial da especificação, o usuário pode usar somente os componentes disponíveis na biblioteca LIB_MS2SV. Essa biblioteca possui um conjunto de primitivas de componentes sequenciais e combinacionais, como operadores lógicos, *flip-flops*, contadores, registradores de deslocamento etc. Ela contém ainda primitivas analógicas, como ganhos, produtos e somas. Também é analisado o número de subsistemas criados para uso específico, junto com tipos constantes e fontes geradoras de pulso (SILVA, 2007).

Um aspecto importante da metodologia do projeto é relacionado ao uso de nomes para especificar os portos de entrada / saída do sistema. O nome de todos os sinais digitais deve ser terminado com "_D" e não deve haver nomes iguais às palavras reservadas da linguagem VHDL-AMS. Os nomes dos componentes são diferentes dos nomes convencionais, porque, no processo de conversão, os nomes convencionais são palavras reservadas no modelo a ser traduzido. Eles também não podem terminar com números, como por exemplo, "CONVDA8". Isso porque, se forem utilizados dois ou mais componentes CONVDA8, por exemplo, o Simulink® mudará o nome do componente para CONVDA1, CONVDA2 etc. Dessa forma, a ferramenta não reconhece os componentes corretamente.

Essa regra de nomes não é aplicada aos operadores lógicos. Neste caso, o programa identifica somente o nome GAND, GOR etc. O número de entradas é especificado dentro do arquivo ".mdl".

Quando a ferramenta MS^2SV é executada, são identificas todas as informações importantes dentro do arquivo do modelo Simulink® e reconhece essas informações na biblioteca de componentes LIB_MS2SV (SILVA, 2007).

O mapeamento automático do modelo Matlab® / Simulink® para a descrição estrutural do VHDL-AMS é suportado por um conjunto de primitivas e subsistemas criados e disponibilizados na biblioteca LIB_MS2SV. Esse é um arquivo do Simulink® que contém todos os componentes que podem ser usados para descrever os modelos utilizados no desenvolvimento do trabalho de Silva (2007). O usuário não pode usar nenhum outro componente que não esteja na biblioteca.

Essa biblioteca possui um conjunto de grupos de componentes e funções, que são descritas a seguir:

- Primitivas Digitais - as primitivas digitais são os componentes que representam uma função digital básica como operadores lógicos, AND, OR, NOT, *flip-flops*.
- Primitivas Analógicas - as primitivas analógicas são compostas pelas instâncias de Ganho, Produto, Soma (duas entradas) e SUM4 (soma de quatro entradas).
- Entrada / Saída - as primitivas de entrada e saída têm uma função importante na descrição dos modelos. Elas visualizam se o sinal externo é digital ou analógico.
- Constantes - as primitivas disponíveis podem ser usadas como valores constantes e fontes geradoras de pulso. O pacote EDULIB define somente a constante VCC implementada através da primitiva SET. Este sinal constante está permanentemente em nível lógico alto (1). Para gerar a constante GND, implementada através da primitiva RESET, foi definido o sinal permanentemente em nível lógico baixo (0).
- Fontes geradoras de pulso - essa versão do sistema permite traduzir somente o pulso gerado diretamente pela simulação. Quando o projetista necessita usar o WORKSPACE do Matlab® para avaliar a biblioteca no Simulink®, é necessário usar o código VHDL-AMS que implementa o modelo e realiza a leitura dos dados no arquivo de entrada.
- Conversão de tipos de dados - na conversão de digital para analógico, trabalhou-se com dois diferentes tipos de sinais, o sinal digital e o sinal analógico. No Simulink®, não existe a necessidade de um componente de conversão entre o componente

representando um sinal digital e outro que represente um sinal analógico, o uso do conversor de tipo de dados é opcional. Por outro lado, é necessário o uso deste componente no SystemVision™, por isso convencionou-se em utilizar os conversores de tipos de dados em todos os modelos utilizados no desenvolvimento deste trabalho;

- Subsistemas - os subsistemas foram criados somente para uso especifico. Esses subsistemas são compostos por outros subsistemas em uma estrutura hierárquica (SILVA, 2007).

No Capítulo 4, apresenta-se a interface de utilização das ferramentas propostas neste trabalho e a metodologia utilizada na realização da tradução dos modelos de sistemas digitais e de sistemas mistos.

Capítulo 4. Ferramentas desenvolvidas

Neste capítulo, são apresentadas as metodologias utilizadas na tradução dos estudos de caso para cada uma das ferramentas desenvolvidas neste trabalho. Logo, na seção 4.1, apresenta-se a ferramenta SF^{2}HDL, por meio da metodologia utilizada na tradução dos modelos de máquinas de estados finitos para os códigos VHDL, Verilog e a tabela de transição de estados padrão. A interface de utilização da ferramenta também é apresentada. Na seção 4.2, apresentam-se as modificações realizadas na ferramenta MS^{2}SV, a nova metodologia de trabalho que foi empregada na tradução dos projetos de sinais mistos e os recursos incorporados à nova versão da ferramenta.

4.1. A ferramenta SF^{2}HDL

A ferramenta SF^{2}HDL (*Stateflow® to Hardware Description Language or table of transition states*) possui uma interface gráfica simplificada similar à maioria dos programas desenvolvidos para a plataforma Windows. Essa interface permite ao projetista selecionar qual arquivo se deseja gerar: o arquivo de texto contendo a tabela de transição de estados que será a entrada para o programa TABELA, o código VHDL comportamental, o código Verilog comportamental, ou uma combinação de todas. Na interface, é possível observar qual o arquivo está sendo traduzido e o tempo gasto na tradução em milissegundos, ambos por meio da barra de *status*. É necessário ainda que o projetista selecione em qual modelo o diagrama de transição de estados foi descrito, pelo modelo de Mealy ou de Moore. Se o projetista selecionar o modelo errado, é exibida na tela do usuário uma mensagem reportando o erro.

Quando o projetista seleciona gerar o arquivo de texto, contendo a tabela de transição de estados, é habilitada na interface uma caixa de *combo*, contendo o tipo de *flip-flop* que será utilizado. Nesse *combo*, aparecem somente duas opções de *flip-flop* (D ou JK), ou seja, as duas opções suportadas pelo programa TABELA. Na Figura 4.1, apresenta-se a interface da ferramenta SF^{2}HDL.

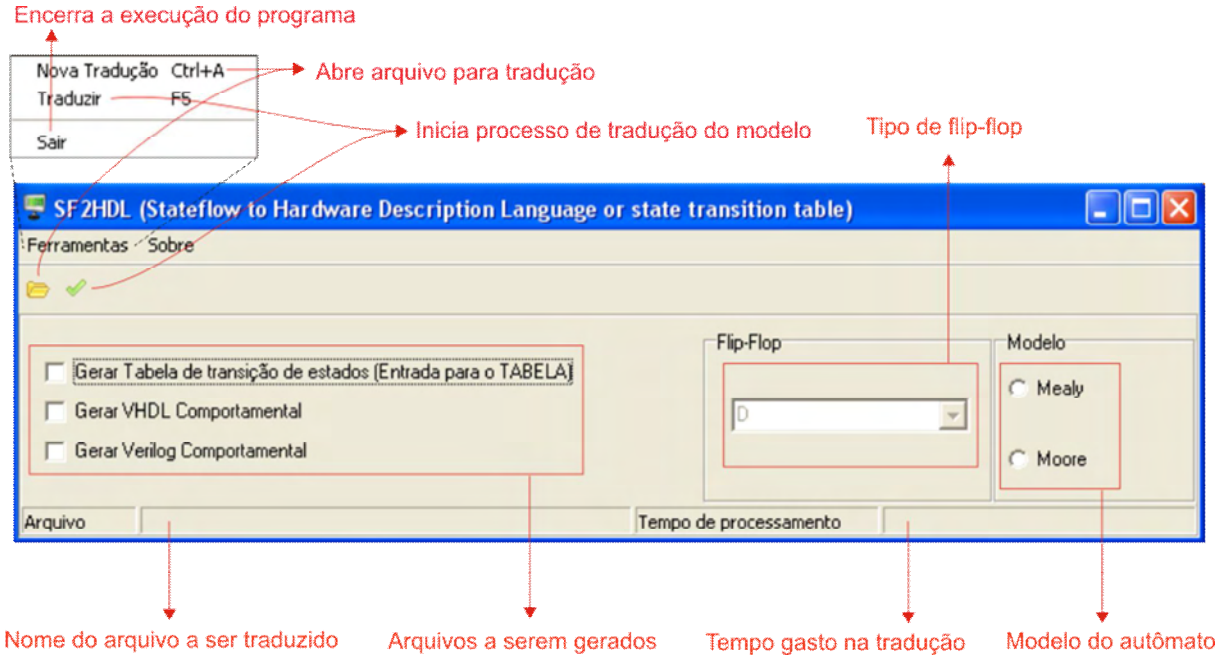

Figura 4.1. Interface do programa SF²HDL.

O programa possui um menu de opções denominado Ferramentas, conforme ilustrado na Figura 4.1. Esse menu possui as seguintes opções:

- Nova Tradução: utilizado para abrir o arquivo com a máquina de estados finitos modelado em ambiente Stateflow® (o arquivo a ser traduzido), descrita pelo modelo de Mealy ou Moore.
- Traduzir: essa opção realiza a tradução do modelo propriamente dita.
- Sair: encerra a execução do programa.

Existe ainda a opção Sobre, que exibe uma tela com um breve comentário sobre a ferramenta desenvolvida.

Inicialmente, a ferramenta SF²HDL realiza a leitura do arquivo contendo a máquina de estados descrita através do modelo de Mealy ou Moore, simulada no Stateflow®. Em seguida, a ferramenta localiza no arquivo toda a estrutura da máquina de estados finitos já que um mesmo arquivo do Simulink®, com extensão ".mdl", pode possuir modelos complexos que integrem componentes do *toolboxes* do Simulink® junto com máquinas de estados finitos no Stateflow®. A estrutura de máquina de estados finitos é armazenada em um arquivo temporário. A partir dessa informação, o programa faz a localização de todos os estados e de todas as transições e armazena essas informações também em arquivos temporários.

O arquivo temporário de estado possui um atributo de identificação chamado "id" e há um atributo do nome do estado chamado "labelString". Esses atributos são carregados em tempo de execução, para serem utilizados no processo de criação das HDLs ou da tabela de

transição de estados padrão. As outras informações não são relevantes para a tradução do modelo, automaticamente não são carregadas em tempo de execução. A listagem a seguir exibe um exemplo das informações armazenadas no arquivo temporário de estado.

```
id                      3
labelString             "A"
position                [149.2449 52.2716 66.7649 48.3259]
fontSize                12
chart                   2
treeNode                [2 0 4 5]
subviewer               2
type                    OR_STATE
decomposition           CLUSTER_STATE
```

É importante salientar que essa listagem exibe o exemplo de um estado referente a uma máquina de Mealy.

A listagem a seguir exibe um exemplo das informações relevantes às transições armazenadas em outro arquivo temporário.

```
id                      7
labelString             "[u[t]==0]{z=0;}"
labelPosition           [266.627 65.001 76.282 15.033]
fontSize                12
src {
 id                     3
 intersection           [2 1 0 0.6492 216.0098 83.6448 0 9.0345]
}
dst {
 id                     4
 intersection           [4 -1 0 0.302 377.9835 83.6448 0 -9.0345]
}
midPoint                [296.2571 83.6435]
chart                   2
linkNode                [2 10 12]
dataLimits              [216.01 377.984 81.245 86.045]
subviewer               2
drawStyle               SMART
slide {
}
```

Nessa listagem, as transições também possuem um atributo de identificação chamado “id” e um atributo de nome, que possui a condição de saída, se submetida à entrada também apresentada no atributo nome, chamado de “labelString”. Porém, no arquivo temporário de transição, existem dois atributos de importantes, o “src” e o “dst”. O atributo “src” possui o “id” do estado de origem desta determinada transição, no exemplo de listagem o estado de origem é o estado de “id” igual a 3. Já o atributo “dst” possui o “id” do estado resultante na ocorrência dessa determinada transição, no exemplo de listagem apresentada, o estado

resultante é o estado de “id” igual a 4. As outras informações contidas no arquivo temporário não são relevantes.

Destacando que esse exemplo de arquivo temporário de transição também representa um modelo de Mealy, a única diferença para um modelo de Moore seria o valor de saída, que estaria contido no arquivo temporário de estado, ao invés do arquivo temporário de transição.

Após localizar os estados, é feita uma lista de todos os estados presentes na máquina e, através da localização das transições, é criada uma lista das transições, das condições de entrada, das saídas existentes e das ligações entre os estados e as transições. Todas essas informações são armazenadas em memória RAM (*Random Access Memory*). No caso do projetista selecionar, gerar o arquivo de entrada para o programa TABELA, a ferramenta SF^{2}HDL necessita da entrada manual do elemento de memória que será utilizado na tabela de transição de estados padronizada.

Na Figura 4.2, ilustra-se o diagrama funcional do programa SF^{2}HDL, com todos os passos envolvidos na tradução.

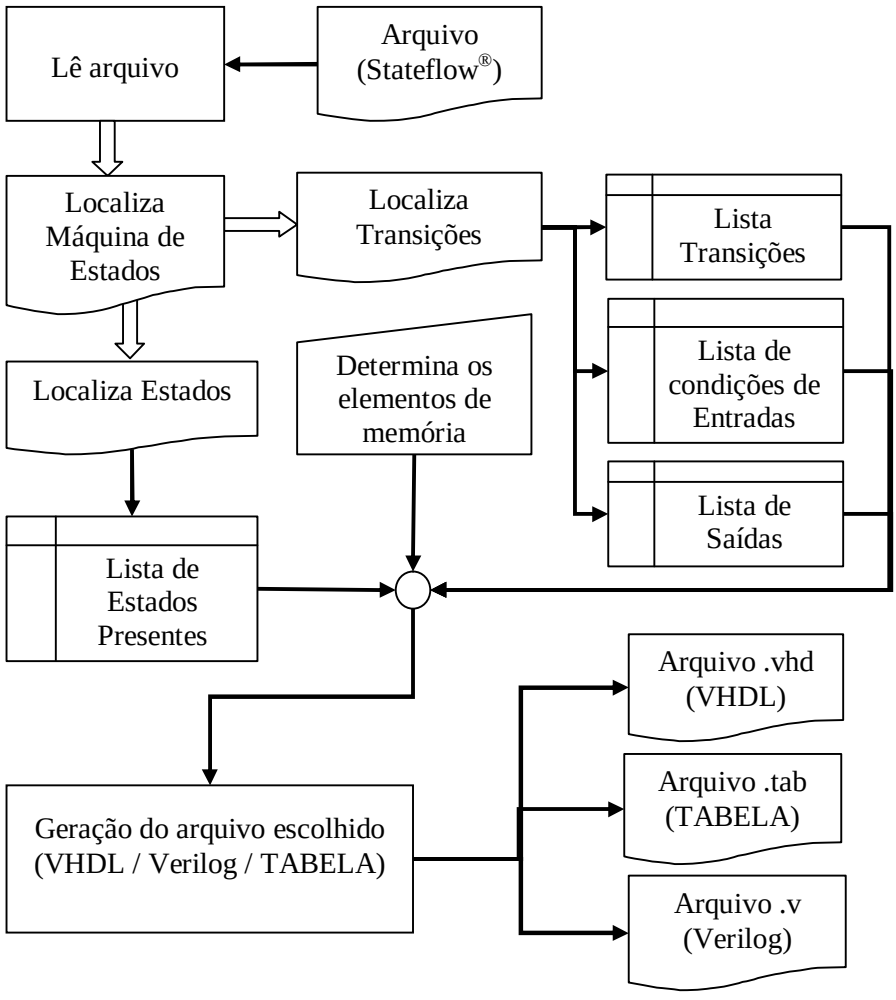

Figura 4.2. Diagrama funcional do programa SF^{2}HDL.

Por meio dessa metodologia, é possível verificar o relacionamento da máquina de estados, permitindo, assim, determinar o estado atual, o próximo estado, a entrada e saída. De acordo com a estrutura de cada arquivo, todas as informações são agrupadas e só então são gerados os arquivos selecionados pelo projetista, de acordo com as opções do menu, ou seja, pode ser gerado um arquivo com a extensão .TAB, que é entrada para o programa TABELA, um arquivo com a descrição em VHDL ou um arquivo com a descrição em Verilog HDL.

A avaliação da metodologia de projeto proposta para o SF^{2}HDL é apresentada no Capítulo 5.

4.2. A ferramenta MS²SV versão 1.7

A versão 1.7 do MS^{2}SV (*Matlab® / Simulink® to SystemVision™*) aperfeiçoou a ferramenta MS^{2}SV. Na primeira versão do programa, o projetista ficava limitado à biblioteca LIB_MS2SV, utilizando somente os componentes existentes em biblioteca previamente definida. Por esse motivo, foi criada uma nova metodologia que oferece mais flexibilidade ao projetista, dois recursos foram agregados:

- O primeiro prevê a possibilidade de o projetista adicionar novos componentes do *toolboxes* do Simulink®. Com isso, o projetista poderá utilizar elementos diferentes dos elementos padrões previamente definidos em biblioteca.
- O segundo prevê a adição de novas bibliotecas desenvolvidas pelo projetista, com modelos mais complexos ou até mesmo a alteração da biblioteca LIB_MS2SV.

Outro aperfeiçoamento realizado na ferramenta MS^{2}SV é a criação de uma interface gráfica que facilite a utilização da ferramenta pelo projetista. Pela possibilidade de criação de um ambiente gráfico interativo, foi escolhida a linguagem C++ e o ambiente C++ Builder da Borland, para a remodelagem da ferramenta MS^{2}SV.

A nova metodologia utiliza uma estrutura de arquivos, que contêm os modelos reconhecidos pela ferramenta e fazem referência a eles. Existe um diretório chamado “bin”, nele encontram-se os arquivos “lib_ref.ini” e “blk_ref.ini”. Esses arquivos referenciam as bibliotecas e os elementos do *toolboxes* do Simulink®, respectivamente. Ainda dentro do diretório “bin” estão os diretórios “blk”, onde estão os códigos VHDL-AMS correspondente aos componentes do *toolboxes* do Simulink® e o diretório “lib” onde estão os códigos VHDL-AMS correspondentes às bibliotecas utilizadas. Dentro do diretório “lib” existe outro diretório e um arquivo de configuração, ambos com o nome da biblioteca utilizada pelo modelo, por exemplo, o diretório “LIB_MS2SV” e o arquivo “LIB_MS2SV.ini”. Somente dentro do

diretório “LIB_MS2SV” estão realmente os elementos utilizados na biblioteca criada pelo projetista. O arquivo de configuração com extensão “.ini” referencia os modelos contidos nessa biblioteca. No caso da adição de novas bibliotecas, serão criados um arquivo “.ini” e um diretório para cada biblioteca adicionada. Na Figura 4.3, é ilustrada a estrutura de arquivos e diretórios utilizados pela nova versão da ferramenta MS^{2}SV.

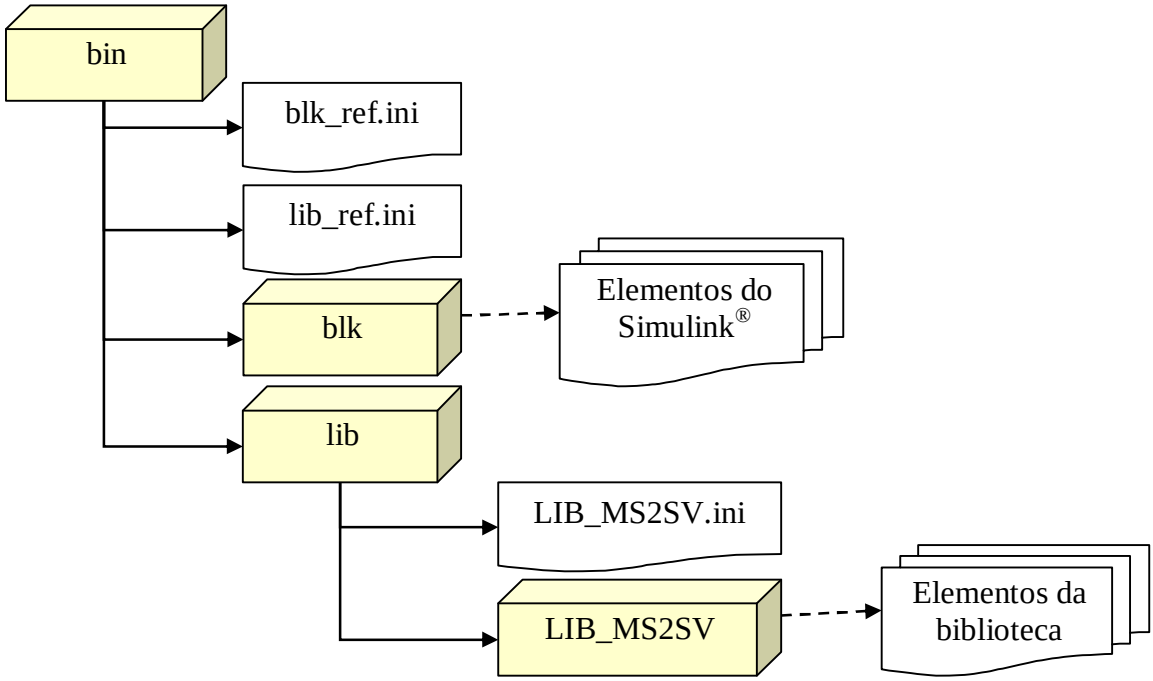

Figura 4.3. Estrutura de diretórios e arquivos utilizados pelo MS^{2}SV.

A tradução do modelo é iniciada com a leitura do arquivo “.mdl”, que contém o modelo a ser traduzido. Primeiramente, é feita uma verificação dos elementos e das bibliotecas utilizadas no modelo através de uma chamada a classe “TranslateCode”. Se não existirem elementos ou bibliotecas desconhecidas, inicia-se o processo de tradução. Em seguida, uma classe chamada “SystemFile” gera toda a estrutura de diretórios necessária para simulação e análise do projeto no ambiente SystemVision™, de acordo com o local que o projetista selecionou para salvar o projeto.

Essa estrutura de projeto, necessária para a simulação no SystemVision™, possui um grupo de diretórios, cada qual utilizado para armazenar os arquivos necessários no projeto como esquemáticos, símbolos, simulações, códigos VHDL utilizados etc. Na Figura 4.4, apresenta-se a estrutura de diretórios do projeto em ambiente SystemVision™.

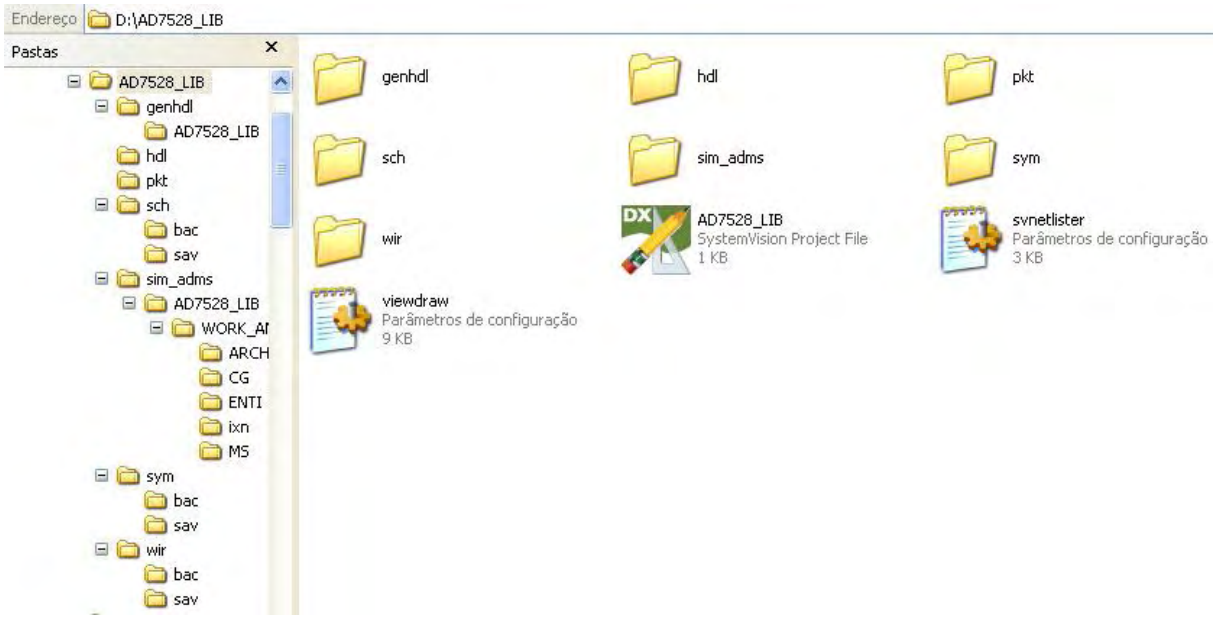

Figura 4.4. Estrutura de diretórios do System Vision™.

Após a chamada a classe "SystemFile", é feita uma nova chamada a classe "TranslateCode". O processo de tradução é feito capturando toda a informação relevante contida no modelo. Em seguida, é montada a estrutura do código VHDL-AMS. No caso de existirem subsistemas ou algum componente de biblioteca, são gerados códigos em VHDL-AMS que são utilizados pelo modelo. Todos os códigos são salvos no diretório "genhdl\Nome_Modelo".

O programa possui uma interface gráfica que também é semelhante à maioria dos programas da plataforma Windows, conforme apresentado na Figura 4.5. Na interface principal, existe um menu contendo as seguintes opções:

- Arquivo: nessa opção, existem duas outras opções. Na primeira, o usuário seleciona o modelo do Simulink® a ser traduzido. Quando selecionada essa opção, aparece na barra de *status* o nome do arquivo que será traduzido. E a segunda opção é um botão para encerrar a execução da ferramenta.
- Ferramentas: nessa opção, será exibida uma nova interface para que o usuário selecione o local onde será salvo o projeto. Só então é iniciado o processo de tradução. No caso de existirem modelos desconhecidos pela ferramenta, será exibida uma mensagem reportando o fato.
- Configuração: essa opção contém uma opção chamada "Adicionar / Remover Componentes (*toolboxes* do Simulink)" e outra chamada "Adicionar / Remover Bibliotecas (biblioteca do Simulink)". Essas opções são utilizadas para adicionar ou remover elementos do *toolboxes* do Simulink® e bibliotecas desenvolvidas pelo

projetista, respectivamente. Se não houver uma configuração do modelo a ser traduzido, a ferramenta não é capaz de reconhecê-lo e traduzi-lo.

- Ajuda: opção de ajuda possui alguns comentários sobre a ferramenta e como ela deve ser utilizada.

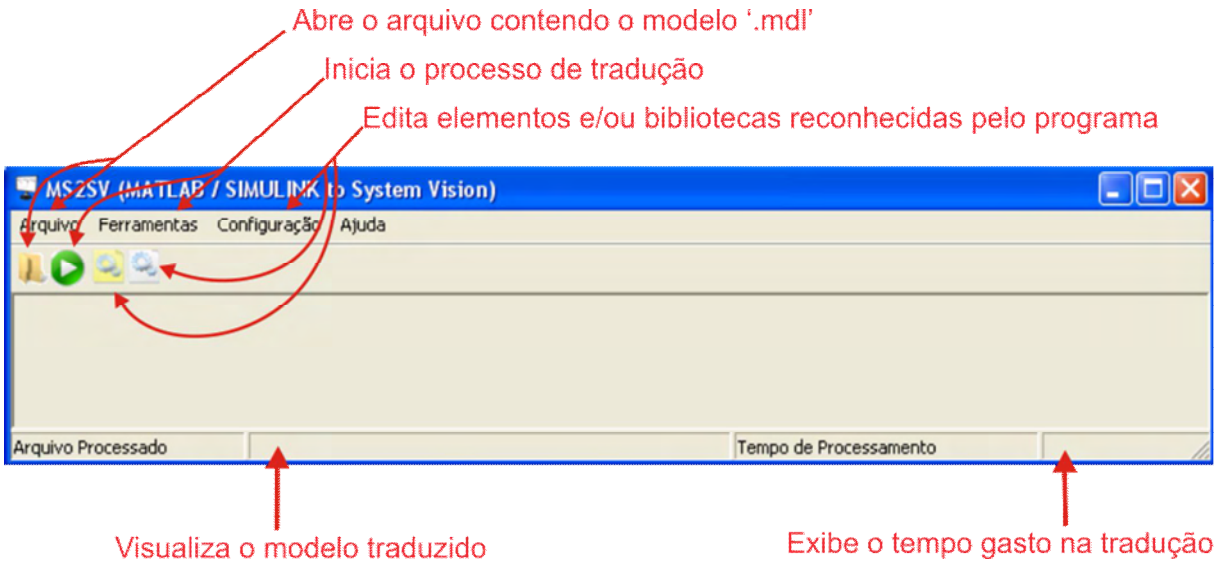

Figura 4.5. Interface principal do MS²SV.

Para a adição de novos elementos do *toolboxes* do Simulink®, a ferramenta MS²SV exibe uma interface amigável, na qual são exibidos os elementos reconhecidos pela ferramenta, conforme o ilustrado na Figura 4.6.

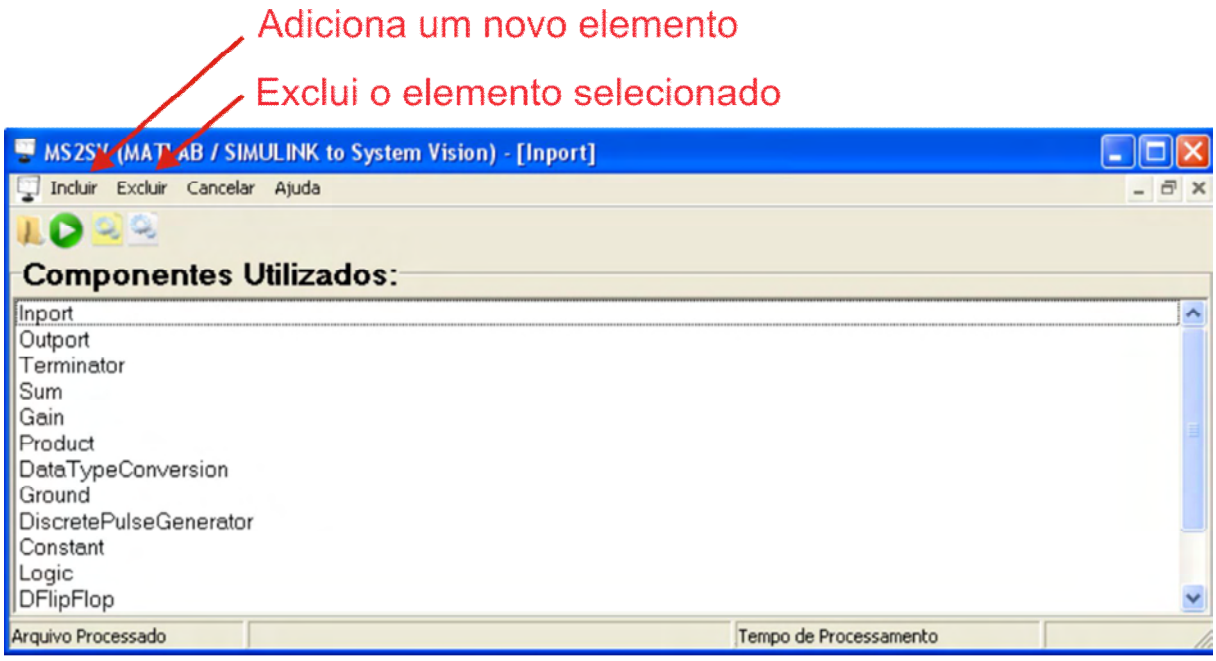

Figura 4.6. Interface de edição dos elementos reconhecidos pelo MS²SV.

Essa interface possui um botão para adicionar novos elementos, que, ao ser selecionado, exibe uma nova interface, em que o projetista pode inserir um novo elemento preenchendo o

campo para inserção de nomes e o campo para a descrição do modelo em VHDL-AMS que representa esses novos elementos, conforme apresentado na Figura 4.7. É importante salientar que o nome do novo elemento deve ser exatamente igual ao nome do elemento no ambiente Simulink®, fazendo-se distinção entre maiúsculas e minúsculas. Outro aspecto importante é que todos os portos devem estar em uma linha de código e os portos que são inicializados com valores constantes não são referenciados na estrutura do *netlist*.

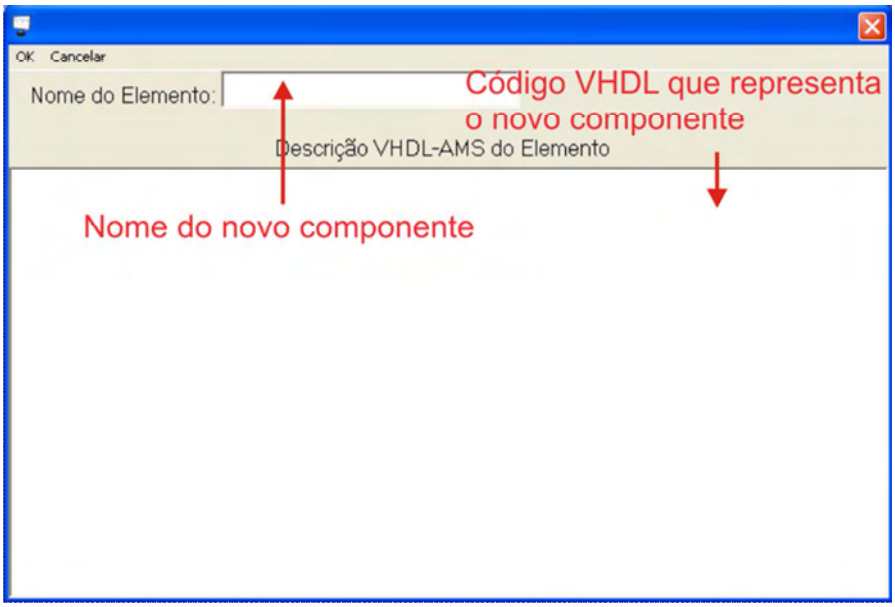

Figura 4.7. Interface de adição de novos elementos.

Para remover um elemento, basta selecionar o elemento escolhido com um clique do mouse e selecionar a opção “Excluir”, conforme opção de menu apresentada na Figura 4.6.

A interface de adição de novas bibliotecas, ilustrada na Figura 4.8, segue o mesmo princípio da interface apresentada na Figura 4.6. Assim, no campo à esquerda, são exibidas todas as bibliotecas que a ferramenta MS^{2}SV é capaz de reconhecer. Quando uma determinada biblioteca é selecionada com um clique do mouse, no campo à direita, são exibidos todos os componentes dessa biblioteca que são reconhecidos pela ferramenta. No caso de exclusões, basta selecionar o componente ou a biblioteca (para o caso de exclusão de uma biblioteca) e clicar em “Excluir”, conforme opção de menu apresentada na Figura 4.8.

Para adicionar uma nova biblioteca de modelos, basta clicar em “Criar”, que é exibida uma mensagem solicitando o nome da nova biblioteca. Para adicionar os componentes que fazem parte de uma nova biblioteca, ou uma biblioteca já existente, basta clicar duas vezes com o mouse no nome dessa determinada biblioteca, que será exibida uma interface com um campo para os nomes do componente e outro para a descrição VHDL-AMS, que representa esse determinado componente. Vale ressaltar que o nome da biblioteca e o nome dos

componentes devem ser exatamente os mesmos nomes constantes na biblioteca criada pelo projetista e dos componentes presentes nessa biblioteca, assim como todos os portos devem estar em uma linha de código.

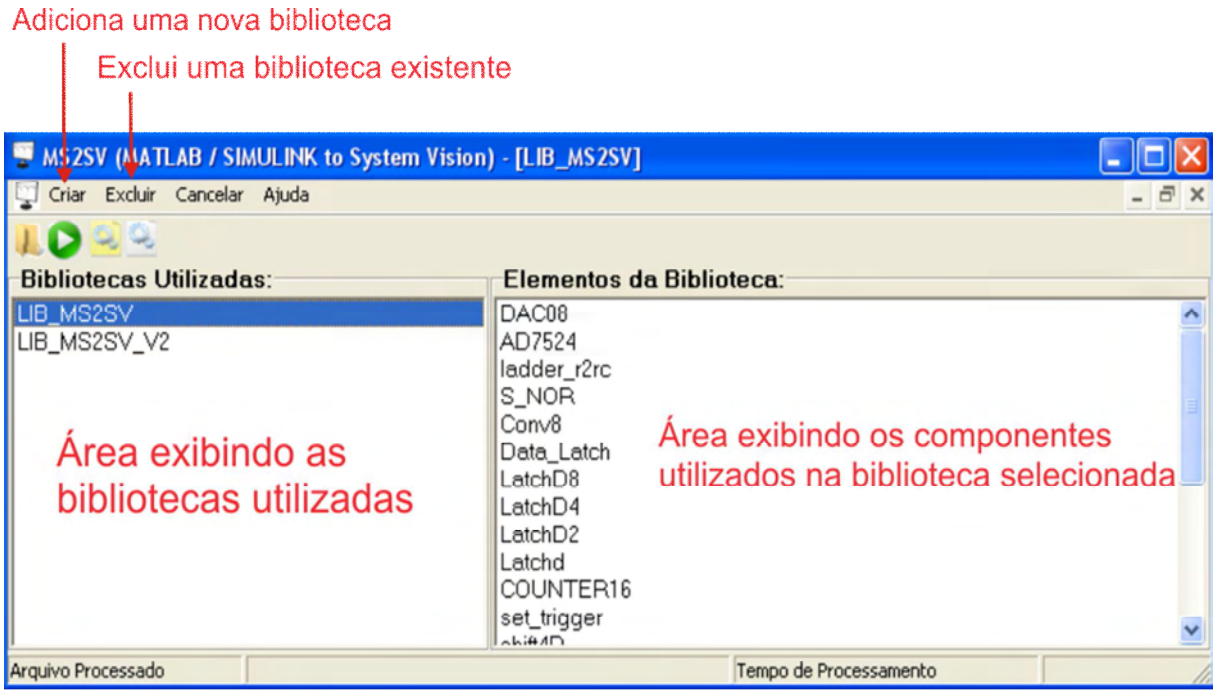

Figura 4.8. Interface de edição de bibliotecas reconhecidas pelo MS^{2}SV.

Alguns componentes podem necessitar de outros componentes existentes na mesma biblioteca. Esses componentes representam subsistemas dentro de subsistemas. Nesse caso, com dois cliques em um componente de uma determinada biblioteca, é exibida uma interface que permite ao usuário selecionar outros componentes de uma mesma biblioteca que necessitam ser incluídos juntos em um mesmo projeto. Na Figura 4.9, apresenta-se a interface que possibilita esse relacionamento hierárquico entre os elementos de uma mesma biblioteca.

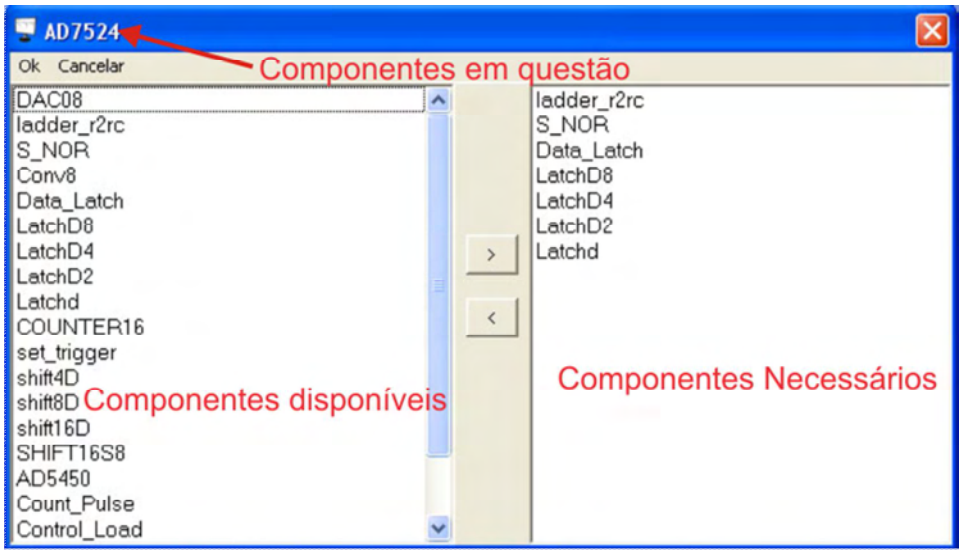

Figura 4.9. Interface de relacionamento dos elementos de uma mesma biblioteca.

Em relação ao nome dos elementos de biblioteca e subsistemas existem as seguintes observações:

- Os portos digitais devem possuir o sufixo "_D".
- Os portos elétricos de entrada devem possuir o sufixo "_I".
- Os subsistemas que possuem entradas digitais devem possuir o sufixo "_D".
- Os blocos Terminator do *toolboxes* do Simulink® que encerrem um sinal digital devem possuir o sufixo "_D".
- Jamais nomear um bloco ou subsistema com palavras reservadas da sintaxe do VHDL-AMS.

Com a nova versão da ferramenta MS^{2}SV, não existe mais a necessidade do projetista ficar restrito à biblioteca LIB_MS2SV e utilizar somente os elementos que compõem essa biblioteca. O projetista agora tem a possibilidade de criar e adicionar suas próprias bibliotecas, assim como adicionar elementos do Simulink® que a ferramenta não é capaz de reconhecer, tornando-o capaz de identificar e traduzir elementos antes desconhecidos. A funcionalidade da ferramenta ganha nova forma a partir dessa metodologia. Na Figura 4.10 apresenta-se o diagrama funcional da ferramenta MS^{2}SV, sendo possível observar os blocos na cor preta, que demonstram o funcionamento já existente na primeira versão da ferramenta e que não foram alterados. Já os blocos destacados na cor azul representam o que não existe mais na funcionalidade da ferramenta, ou seja, a dependência da biblioteca LIB_MS2SV. Os blocos destacados em vermelho representam os blocos que foram adicionados à funcionalidade do programa, ou seja, a nova metodologia de maior praticidade e interação do projetista.

Inicialmente, a ferramenta MS^{2}SV faz a leitura do arquivo com o modelo de projeto misto já simulado no ambiente Simulink®. É necessária uma verificação dos elementos existentes em biblioteca e dos componentes do *toolboxes* do Simulink®, a fim de verificar se existem componentes e / ou bibliotecas desconhecidas. A ferramenta captura no arquivo com extensão ".mdl" todas as informações relevantes para a construção do circuito, armazenando, em memória RAM, a lista dos componentes existentes no modelo, a lista de ligações entre esses componentes e outras informações importantes para a geração do *netlist* do circuito. Em seguida, a ferramenta MS^{2}SV cria toda a estrutura de projeto necessária para a simulação do projeto no ambiente SystemVision™. Só então são geradas todas as descrições em VHDL-AMS que o projeto necessita e os arquivos necessários para o *debug* do projeto no SystemVision™.

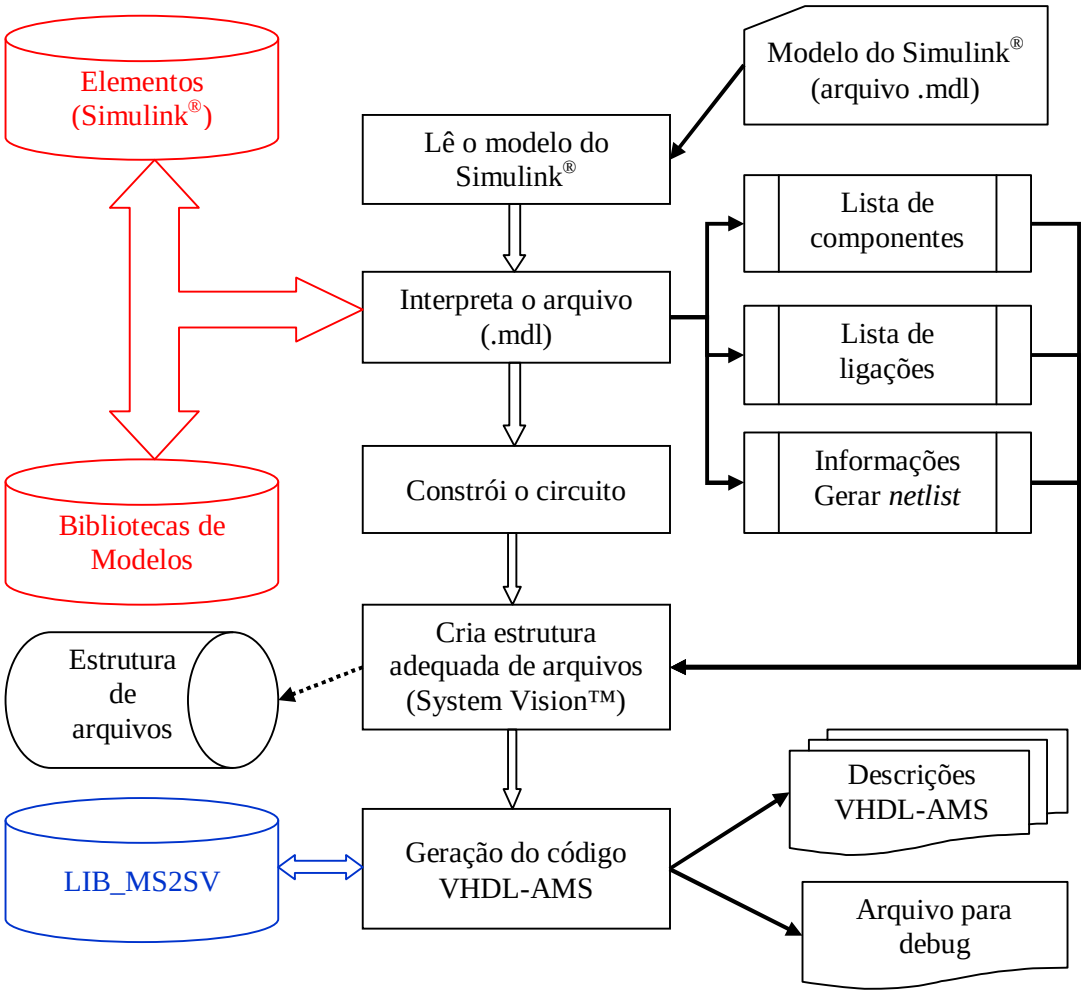

Figura 4.10. Diagrama funcional do programa MS²SV.

A validação da nova metodologia empregada no MS²SV versão 1.7 é apresentada no Capítulo 5, através dos conversores de dados do tipo DAC, que foram apresentados no Capítulo 3.

Capítulo 5. Avaliação das ferramentas desenvolvidas

Neste capítulo, são apresentados os resultados obtidos com a avaliação das ferramentas desenvolvidas neste trabalho. Inicialmente, na seção 5.2, são apresentados os estudos de caso utilizados para avaliar o SF^{2}HDL e, na seção 5.3, são apresentados os estudos de caso que avaliaram a nova versão do MS^{2}SV, bem como os resultados obtidos a partir dessas avaliações.

5.1. Avaliação da ferramenta SF²HDL

Uma importante forma de avaliar a ferramenta SF^{2}HDL é utilizando, como estudo de caso, os códigos de linha de sistemas de telecomunicações. Assim, os códigos de linhas apresentados no Capítulo 2 foram modelados como diagrama de transição de estados no ambiente Stateflow®. A partir da tabela de descrição de estados gerada pelo SF^{2}HDL, foi utilizado o programa TABELA para realizar a minimização das funções booleanas da máquina de estados e, a partir do arquivo gerado pelo programa TABELA, foi utilizado o programa TAB2VHDL, para gerar o código VHDL funcional da máquina de estados finitos. Foram obtidas três descrições em linguagem de descrição de *hardware* de uma mesma máquina de estados finitos, pois, outros modelos foram gerados diretamente a partir do arquivo Stateflow® um modelo VHDL e outro Verilog HDL. Dessa forma, obtiveram-se, a partir da uma descrição em Stateflow®, três diferentes modelos em duas linguagens de descrição de *hardware*, ou seja:

- ♦ VHDL comportamental.
- ♦ VHDL funcional.
- ♦ Verilog HDL comportamental.

A partir dessas descrições, foi possível realizar um comparativo entre as diferentes formas de modelagem e uma comparação entre a síntese gerada pelas ferramentas disponíveis comercialmente. Como ambiente de síntese, utilizou-se ambiente Quartus II desenvolvido pela Altera Corporation. Esse ambiente é utilizado para síntese de circuitos digitais e seus FPGAs (*Field Programmable Gate Array*) e CPLDs (*Complex Programmable Logic Device*).

Para a avaliação e simulação da implementação das máquinas de estados, foi utilizada a FPGA Cyclone II modelo EP2C20F484C7, também da Altera. Porém, é importante ressaltar

que a utilização do componente Cyclone II foi apenas em simulação, ou seja, não foi realizada a implementação física do circuito em FPGA.

A FPGA Cyclone II contém linhas bidimensionais e colunas baseadas em arquiteturas. As colunas e linhas são interconectadas entre blocos de vetores lógicos (LAB – *Logic Array Bloks*), blocos de memória embarcada e multiplicadores embarcados. Os LABs são agrupados em linhas e colunas através da FPGA e em cada LAB existem 16 elementos lógicos (LE – *Logic Element*). Um LE é uma pequena unidade lógica que permite a implementação de funções lógicas pelo projetista. O LE possui as seguintes características:

- Quatro *lookup table* (LUT), o qual é um gerador de funções que pode implementar qualquer função de quatro variáveis.
- Um registrador programável.
- Uma conexão de circuito carregado (*carry chain*).
- Uma conexão de registrador de circuito.
- A habilidade de direcionar todas as interconexões: local, linha, coluna, registrador de circuito e interconexões de *link* direto.
- Suporte para pacotes de registradores.
- Suporte para avaliação do registrador.

Cada registrador programável do LE pode ser configurado para operações com *flip-flop* do tipo D, JK, T e SR.

Para modelar os códigos de linha em ambiente Stateflow®, convencionou-se em adicionar na palavra do código um bit para representar o sinal negativo. Dessa forma, o negativo é representado pelo nível lógico alto (1) e o sinal positivo pelo nível lógico baixo (0). O bit do sinal é o bit mais significativo. Outra convenção utilizada é a notação dos valores de entrada e saída, pois todos os modelos do Stateflow® estão em notação decimal ao invés de binária.

Em alguns casos, a descrição gerada pelo SF^{2}HDL é muito extensa, por isso, convencionou-se em utilizar o símbolo de "..." para representar no texto uma continuação do código que é exatamente igual à sequência de comandos anterior e posterior ao símbolo de "...".

Para a realização das simulações, foram utilizadas palavras seriais de 24 bits, tanto para o ambiente Stateflow como para o ambiente Quartus II. Essa palavra foi empregada na simulação de todos os códigos de linha, sendo formada pelos seguintes bits:

- $(000100110000110101000100)_2$.

A única exceção é em relação ao código 2B1Q, já que esse código utiliza dois bits de entrada, pois o bit menos significativo recebeu a seguinte palavra:

♦ $(001001100001101010001000)_2$.

E o bit mais significativo recebeu a seguinte palavra:

♦ $(000100100001101010001000)_2$.

Ambas com um total de 24 bits, formando uma entrada decimal de:

♦ $(001201320001321212001200)_{10}$.

5.1.1. Código AMI

Na Figura 5.1, apresenta-se o diagrama de transição de estados do código AMI descrito pelo modelo de Moore em ambiente Stateflow®. Nela, é possível observar o estado inicial A, os estados B, C e D e as saídas *z* contidas em cada estado. A letra *u*, nas transições (arcos), representa a variável de entrada em um instante *t* de tempo discreto, que representa a transição de um estado de origem, definido pelo início do arco de transição, para um estado destino, definido pela seta do arco. Os símbolos de chaves demonstram a sintaxe adotada pelo ambiente Stateflow® para a descrição do modelo de Moore. A simulação do código AMI no ambiente Stateflow® apresentou um comportamento conforme o esperado em relação à palavra de dados de entrada utilizada na simulação e seu resultado de simulação é apresentado na Figura 5.2.

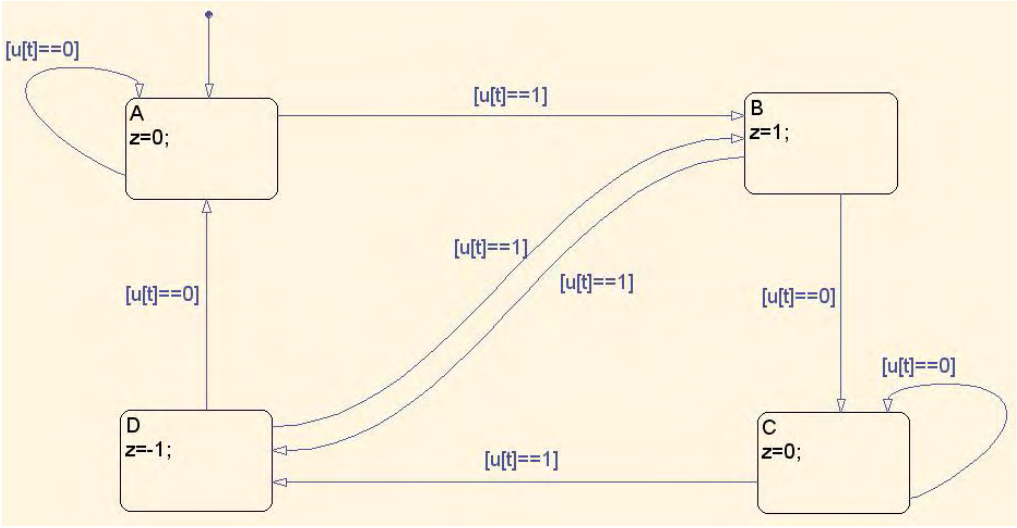

Figura 5.1. Código de linha AMI em ambiente Stateflow®.

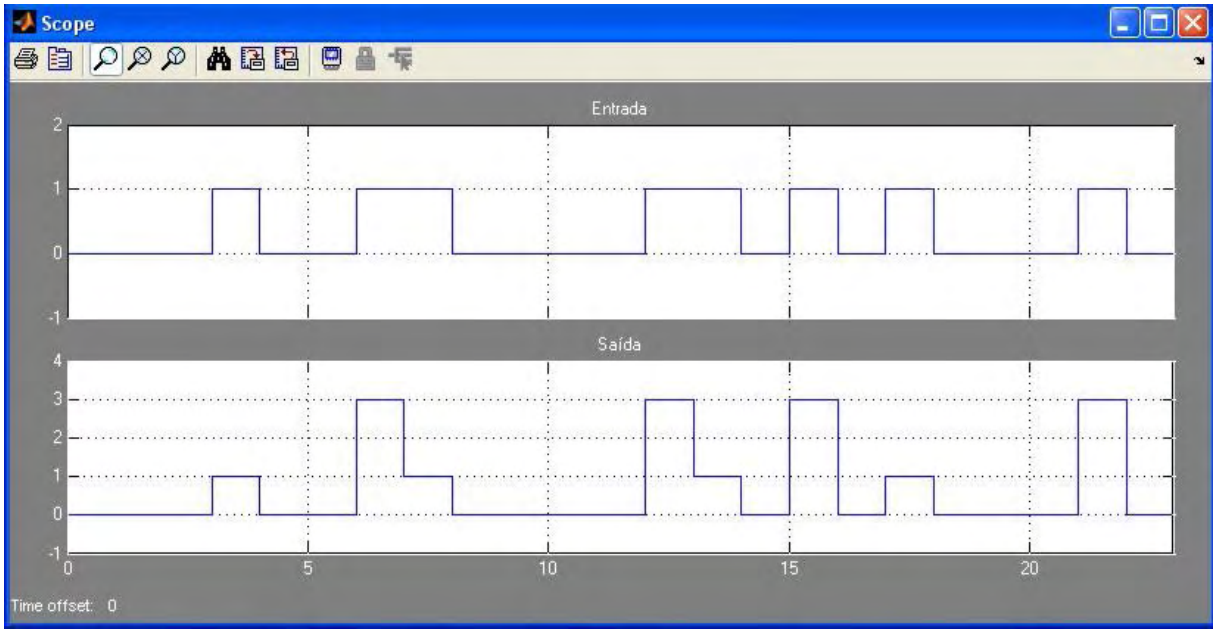

Figura 5.2. Simulação do código AMI no Stateflow.

Na Figura 5.3 (a), (b) e (c), são apresentados os códigos VHDL comportamental, Verilog comportamental e VHDL funcional, respectivamente. Após serem obtidas as três descrições, foi realizada a fase de simulação no ambiente Quartus II e, tanto a VHDL comportamental, a Verilog comportamental e o VHDL funcional apresentaram os mesmos resultados gerados pelo Stateflow®. A Figura 5.4 apresenta o resultado da simulação do código Verilog comportamental no Quartus II e a Figura 5.5 apresenta a simulação do código VHDL funcional também no ambiente Quartus II.

Na Tabela 5.1, é apresentado o resultado da síntese obtida pelo ambiente Quartus II das duas descrições geradas pelo SF^{2}HDL e pelo TAB2VHDL. A síntese obtida, para o VHDL comportamental e para o Verilog comportamental, utilizou a mesma quantidade de elementos lógicos, funções combinacionais e registradores, mas a síntese obtida com VHDL funcional utilizou menos elementos lógicos, funções combinacionais e resgitradores, porém com a utilização de dois pinos a mais, pois, na descrição funcional, foram inseridos pinos referentes aos estados internos da máquina de estados somente para fins didáticos. Porém, é importante salientar que o modelo foi previamente minimizado através do TABELA, o que não ocorre com os outros dois modelos.

```
ENTITY Moore_AM_C IS
 PORT(
  input : IN INTEGER RANGE 0 TO 1;
  output : OUT INTEGER RANGE 0 TO 1;
  reset : IN BIT;
  clk : IN BIT);
END Moore_AM_C;

ARCHITECTURE lpssd OF Moore_AM_C IS

TYPE type_state IS (S0,S1,S2,S3);
SIGNAL state , nxtstate : type_state;

BEGIN
 behavior: PROCESS (reset, input,
state)
 BEGIN
  nxtstate <= state;
  output <= 0;
  IF reset = '1' THEN
   nxtstate <= S0;
  ELSE
   CASE state IS
    WHEN S0 =>
     output <= 0;
     IF (input = 1) THEN
      nxtstate <= S1;
     ELSE
      nxtstate <= S0;
     END IF;
    WHEN S1 =>
     output <= 1;
     IF (input = 1) THEN
      nxtstate <= S3;
     ELSE
      nxtstate <= S2;
     END IF;
    WHEN S2 =>
     output <= 0;
     IF (input = 1) THEN
      nxtstate <= S3;
     ELSE
      nxtstate <= S2;
     END IF;
    WHEN S3 =>
     output <= 3;
     IF (input = 1) THEN
      nxtstate <= S1;
     ELSE
      nxtstate <= S0;
     END IF;
   END CASE;
  END IF;
 END PROCESS behavior;


 clock: PROCESS
 BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  state <= nxtstate;
 END PROCESS clock;
END lpssd;
```

```
module Moore_AM (clk, in, reset,
out);

 input clk, reset;
 input [0:0] in;
 output out;
 reg [0:0] out;
 reg [1:0] state;
 reg [1:0] nxtstate;
 parameter [1:0]
  S0 = 0,
  S1 = 1,
  S2 = 2,
  S3 = 3;

 always @ (in or reset or state)
begin
  nxtstate = state;
  out = 0;
  if (reset) begin
   nxtstate = S0;
  end
  else begin
   case (state)
    S0: begin
     out = 0;
     if (in == 1) begin
      nxtstate = S1;
     end
     else begin
      nxtstate = S0;
     end
    end
    S1: begin
     out = 1;
     if (in == 1) begin
      nxtstate = S3;
     end
     else begin
      nxtstate = S2;
     end
    end
    S2: begin
     out = 0;
     if (in == 1) begin
      nxtstate = S3;
     end
     else begin
      nxtstate = S2;
     end
    end
    S3: begin
     out = 3;
     if (in == 1) begin
      nxtstate = S1;
     end
     else begin
      nxtstate = S0;
     end
    end
   endcase
  end
 end

 always @ (posedge clk) begin
  state = nxtstate;
 end
endmodule
```

```
ENTITY Moore_AM_E IS
     PORT(
     CLK, CLR : IN BIT;
     X0 : IN BIT;
     Q0, Q1 : OUT BIT;
     Z0 : OUT BIT );
END Moore_AM_E;

ARCHITECTURE RTL OF Moore_AM_E IS
SIGNAL  VE0, VE1: BIT;
SIGNAL D1, D0 : BIT;


 BEGIN
PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE1 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE1 <= D1;
    END IF;
    Q1 <= VE1;
END PROCESS;

  D1 <=  ( (VE1) AND NOT(VE0)) OR
(NOT(VE1) AND (VE0));
  D0 <=  ( (X0));
  Z0 <=  ( NOT(VE1) AND (VE0));

END RTL;
```

(a) VHDL comportamental **(b) Verilog HDL comportamental** **(c) VHDL Funcional**

Figura 5.3. Descrições HDL obtidas pela ferramenta desenvolvida para o código AMI.

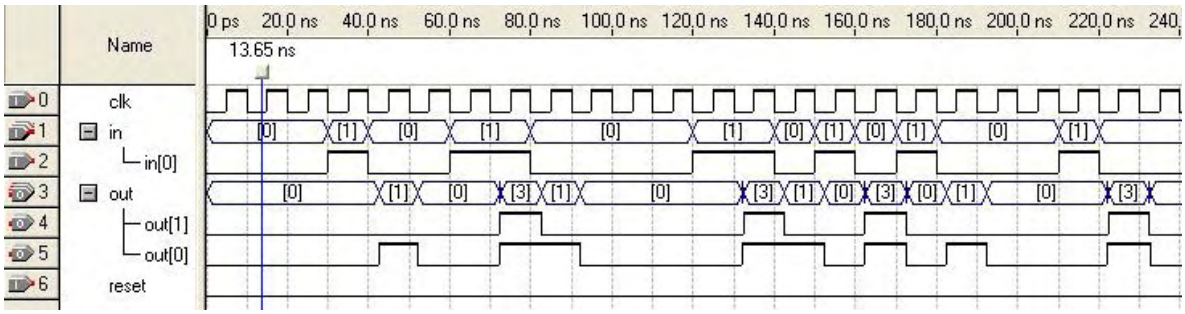

Figura 5.4. Simulação do código AMI no ambiente Quartus com o Verilog.

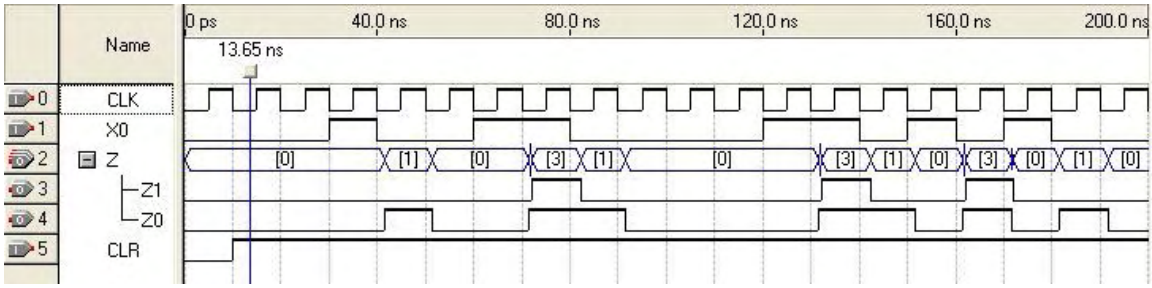

Figura 5.5. Simulação do código AMI no Quartus com o VHDL funcional.

Tabela 5.1. Síntese obtida pelo ambiente Quartus para o código AMI.

	VHDL (SF²HDL)	**Verilog (SF²HDL)**	**VHDL (TAB2VHDL)**	**Total**
Elementos lógicos	6 (0,031 %)	6 (0,031 %)	2 (0,01 %)	18.752
Funções combinacionais	6 (0,031 %)	6 (0,031 %)	2 (0,01 %)	18.752
Registradores	4 (0,021 %)	4 (0,021 %)	2 (0,01 %)	18.752
Pinos	5 (1,58 %)	5 (1,58 %)	7 (2,22 %)	315

5.1.2. Código HDB1

Na Figura 5.18, é apresentado o código de linha HDB1 em ambiente Stateflow®, descrito pelo modelo de Mealy, cuja a máquina é iniciada no estado S1. O rótulo da saída *z* e a entrada *u* está contida na transição de cada estado, ocorrendo em um instante de tempo discreto *t*. A simulação em ambiente Stateflow® apresentou um atraso no modelo proposto em 1 instante de tempo, em que a saída esperada para o caso de dois 0 consecutivos aconteceu somente no instante *t+1*, deixando visível a necessidade de realizar futuramente uma remodelagem do código HDB1, em uma máquina de estados finitos, diferente da proposta neste trabalho. Porém, esse modelo apresentou uma ocorrência menor de atrasos no sinal de saída em comparação ao primeiro modelo proposto no início do desenvolvimento deste trabalho.

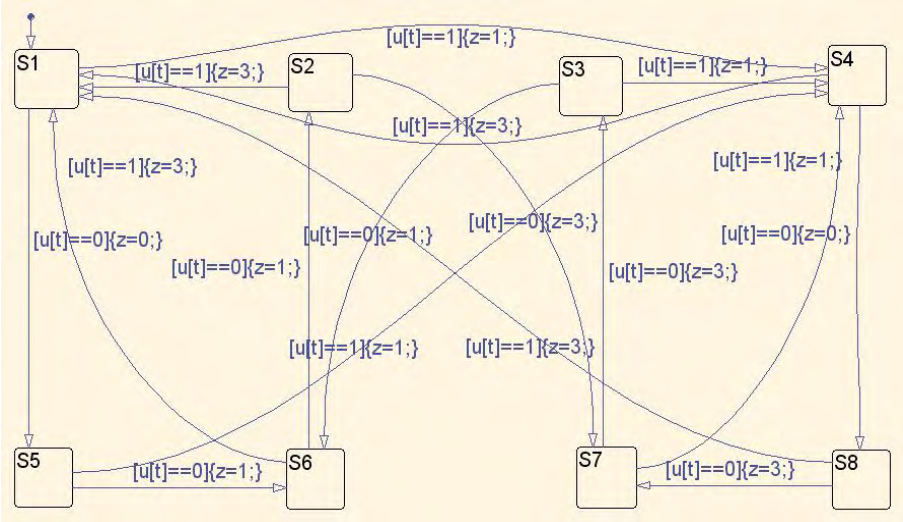

Figura 5.6. Código de linha HDB1 em ambiente Stateflow®.

Na Figura 5.19 (a), (b) e (c), apresentam-se os códigos gerados VHDL comportamental, Verilog comportamental e VHDL funcional, respectivamente, através da tradução da ferramenta SF^{2}HDL, da minimização do TABELA e da tradução do TAB2VHDL.

```
ENTITY Moore_HD_C IS
 PORT(
  input : IN INTEGER RANGE 0 TO 1;
  output : OUT INTEGER RANGE 0 TO 3;
  reset : IN BIT;
  clk : IN BIT);
END Moore_HD_C;

ARCHITECTURE lpssd OF Moore_HD_C IS

TYPE type_state IS
(S0,S1,S2,S3,S4,S5);
SIGNAL state , nxtstate : type_state;

BEGIN
 behavior: PROCESS (reset, input,
state)
 BEGIN
  nxtstate <= state;
  output <= 0;
  IF reset = '1' THEN
   nxtstate <= S2;
  ELSE
   CASE state IS
    WHEN S0 =>
     output <= 1;
     IF (input = 0) THEN
      nxtstate <= S3;
     ELSE
      nxtstate <= S5;
     END IF;
          ...
          ...
    WHEN S5 =>
     output <= 3;
     IF (input = 1) THEN
      nxtstate <= S0;
     ELSE
      nxtstate <= S2;
     END IF;
   END CASE;
  END IF;
 END PROCESS behavior;

 clock: PROCESS
 BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  state <= nxtstate;
 END PROCESS clock;
END lpssd;
```

(a) VHDL comportamental

```
module Moore_HD (clk, in, reset,
out);

 input clk, reset;
 input [0:0] in;
 output out;
 reg [1:0] out;
 reg [2:0] state;
 reg [2:0] nxtstate;
 parameter [2:0]
  S0 = 0,
  S1 = 1,
  S2 = 2,
  S3 = 3,
  S4 = 4,
  S5 = 5;

 always @ (in or reset or state)
begin
  nxtstate = state;
  out = 0;
  if (reset) begin
   nxtstate = S2;
  end
  else begin
   case (state)
    S0: begin
     out = 1;
     if (in == 0) begin
      nxtstate = S3;
     end
     else begin
      nxtstate = S5;
     end
    end
          ...
          ...
    S5: begin
     out = 3;
     if (in == 1) begin
      nxtstate = S0;
     end
     else begin
      nxtstate = S2;
     end
    end
   endcase
  end
 end

 always @ (posedge clk) begin
  state = nxtstate;
 end
endmodule
```

(b) Verilog HDL comportamental

```
ENTITY Moore_HD_E IS
     PORT(
     CLK, CLR : IN BIT;
     X0 : IN BIT;
     Q0, Q1, Q2 : OUT BIT;
     Z0, Z1 : OUT BIT );
END Moore_HD_E;

ARCHITECTURE RTL OF Moore_HD_E IS

SIGNAL  VE0, VE1, VE2: BIT;
SIGNAL D2, D1, D0 : BIT;

 BEGIN

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;
          ...
          ...
PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE2 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE2 <= D2;
    END IF;
    Q2 <= VE2;
END PROCESS;

  D2 <=  ( (VE2) AND NOT(VE0)) OR
((X0) AND (VE1) AND (VE0)) OR
(NOT(X0) AND (VE1) AND NOT(VE0))
OR ((X0) AND NOT(VE1) AND
NOT(VE0));
  D1 <=  ( NOT(X0) AND (VE2) AND
(VE0)) OR (NOT(X0) AND NOT(VE2)
AND NOT(VE1) AND NOT(VE0));
  D0 <=  ( (X0) AND NOT(VE1) AND
NOT(VE0)) OR (NOT(X0) AND
NOT(VE2) AND NOT(VE1)) OR ((VE1)
AND (VE0));
  Z1 <=  ( NOT(VE1) AND (VE0));
  Z0 <=  ( NOT(VE1));

END RTL;
```

(c) VHDL funcional

Figura 5.7. Descrições obtidas pela ferramenta desenvolvida para o código HDB1.

Os resultados obtidos nas simulações foram aceitáveis para a proposta do código HDB1, mas todas as simulações apresentaram os mesmos resultados, tanto nas três descrições utilizadas, como também no ambiente Stateflow®.

A Figura 5.20 apresenta o resultado para a descrição em VHDL comportamental e a Figura 5.21 apresenta o resultado para o código VHDL funcional.

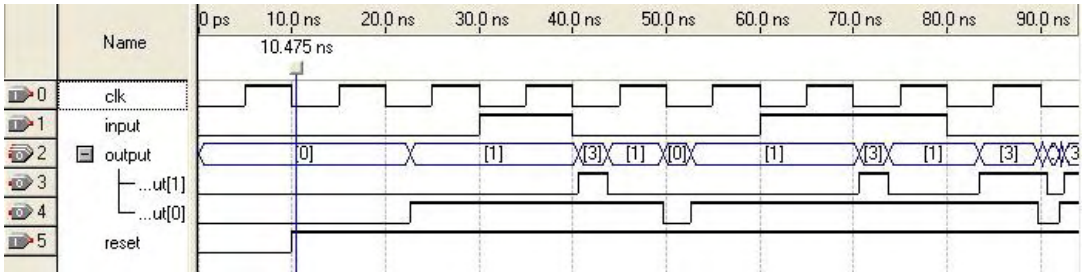

Figura 5.8. Simulação do código HDB1 no Quartus com o VHDL comportamental.

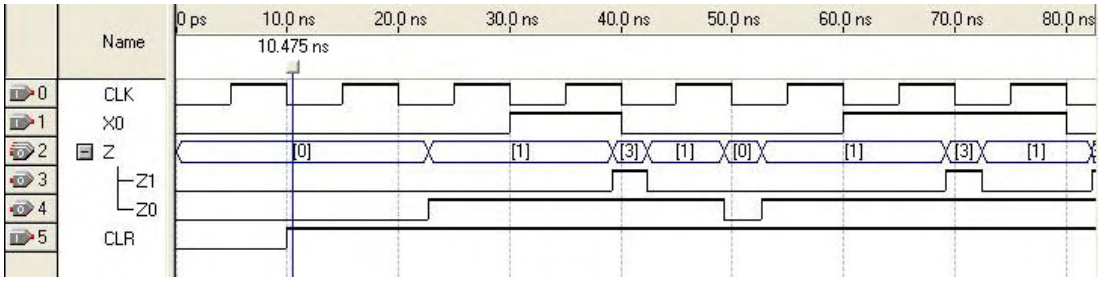

Figura 5.9. Simulação do código HDB1 no Quartus com o VHDL funcional.

A descrição em Verilog comportamental apresentou o pior resultado da síntese gerada pelo Quartus II, utilizando 10 elementos lógicos e 10 funções combinacionais, ao invés dos 9 elementos lógicos e das 9 funções combinacionais utilizadas na descrição comportamental da VHDL e dos 4 elementos lógicos e das 4 funções combinacionais utilizadas na descrição funcional do VHDL, que, por sua vez, utilizou uma quantidade maior de pinos e uma quantidade menor de registradores em comparação as outras duas descrições. O resultado completo da síntese realizada pelo Quartus II, para as três descrições, pode ser observado na Tabela 5.5.

Tabela 5.2. Síntese obtida pelo ambiente Quartus para o código HDB1.

	VHDL (SF^2HDL)	**Verilog (SF^2HDL)**	**VHDL (TAB2VHDL)**	**Total**
Elementos lógicos	12 (0,063 %)	13 (0,069 %)	5 (0,026 %)	18.752
Funções combinacionais	12 (0,063 %)	13 (0,069 %)	5 (0,026 %)	18.752
Registradores	8 (0,042 %)	8 (0,042 %)	3 (0,015 %)	18.752
Pinos	5 (1,58 %)	5 (1,58 %)	8 (2,53 %)	315

5.1.3. Código HDB3

Na Figura 5.6, apresenta-se o código de linha HDB3, descrito pelo modelo de Mealy, em ambiente Stateflow®. Em uma descrição pelo modelo de Mealy, a saída *z* está contida no rótulo da transição junto com entrada *u*. O estado S00 representa o estado inicial da máquina, e os dois pares de estado S07 e S0F (no canto esquerdo) e S1F e S17 (no canto direito) representam a descrição AMI, o que ocasionam a codificação alternada entre +1 e -1. O resultado da simulação é apresentado na Figura 5.7.

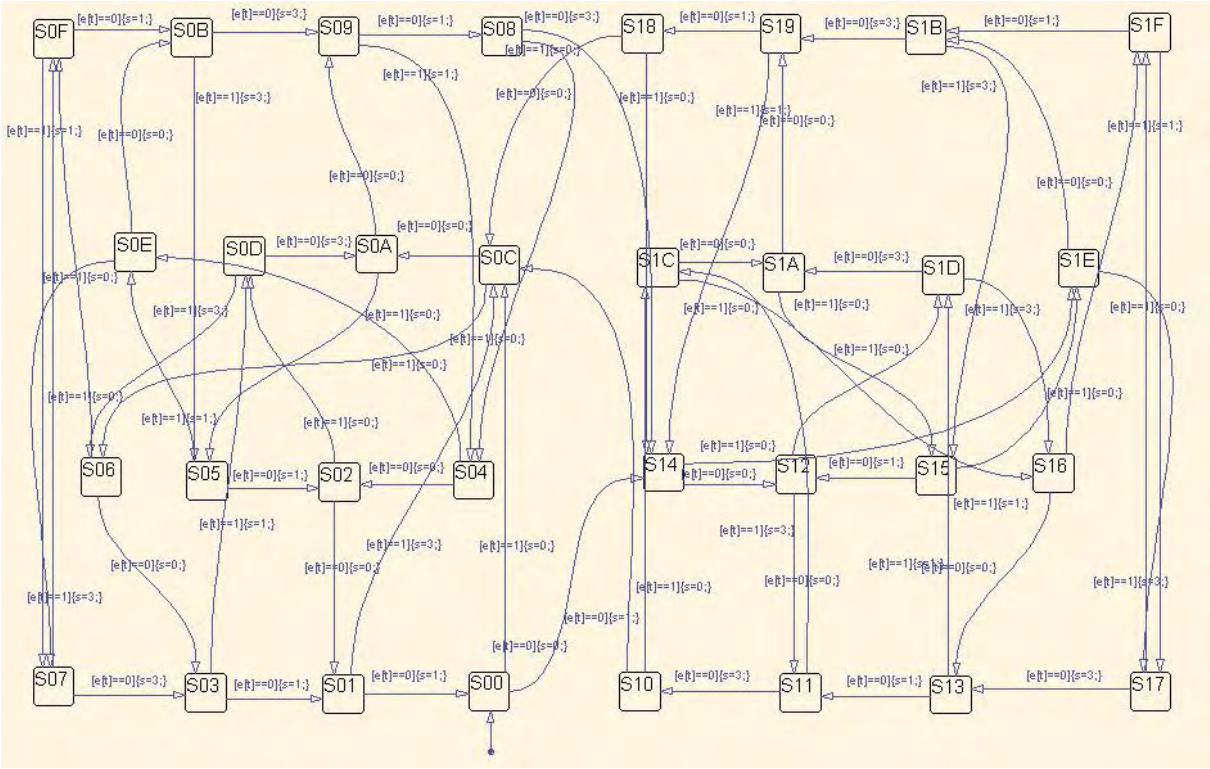

Figura 5.10. Código de linha HDB3 em ambiente Stateflow®.

Na Figura 5.8, apresenta-se o código HDB3 descrito nas linguagens de descrição de hardware, VHDL comportamental (Figura 5.8 (a)), Verilog comportamental (Figura 5.8 (b)) e VHDL funcional (Figura 5.8 (c)). Na Figura 5.9, apresenta-se o resultado da simulação obtida pela descrição do código HDB3 em VHDL comportamental.

Já na Tabela 5.2, apresenta-se o resultado da síntese obtida pelo ambiente Quartus II para as três descrições. Da forma semelhante ao código AMI, a VHDL comportamental e a Verilog comportamental tiveram o mesmo resultado, já a VHDL funcional apresentou um resultado com menor quantidade de elementos lógicos, funções combinacionais e registradores, porém, foram necessários 5 pinos a mais que nas outras descrições de HDL.

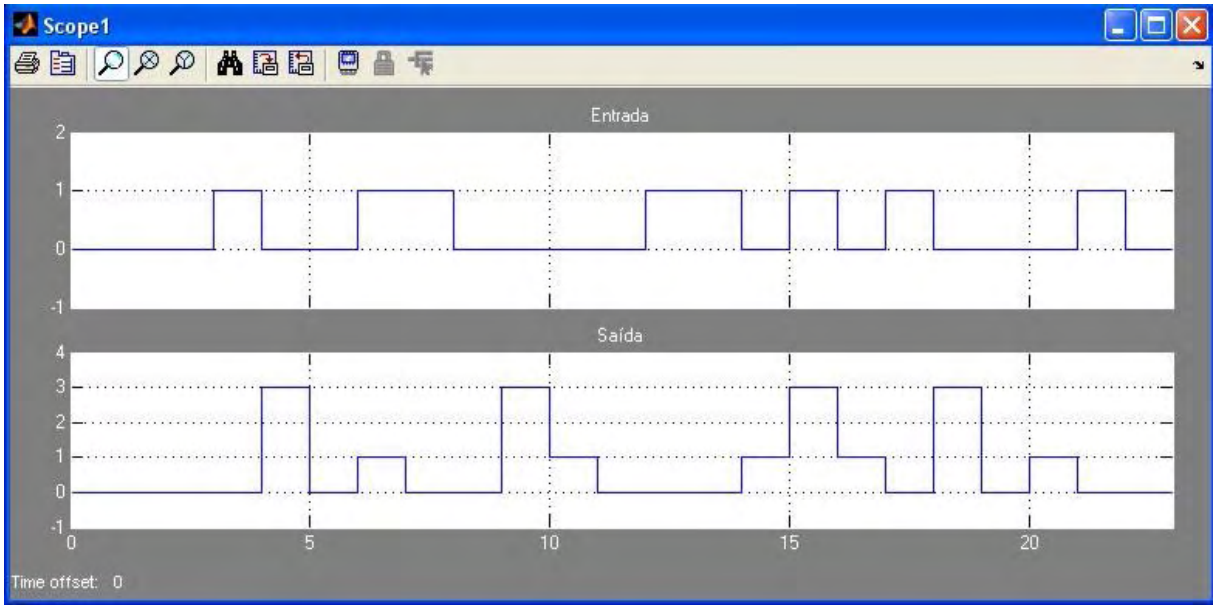

Figura 5.11. Simulação do código HDB3 no Stateflow.

```
ENTITY Mealy_HD_C IS
 PORT(
  input : IN INTEGER RANGE 0 TO 1;
  output : OUT INTEGER RANGE 0 TO 3;
  reset : IN BIT;
  clk : IN BIT);
END Mealy_HD_C;

ARCHITECTURE lpssd OF Mealy_HD_C IS

TYPE type_state IS (S0,S1,...,S31);
SIGNAL state , nxtstate : type_state;

BEGIN
 behavior: PROCESS (reset, input,
state)
 BEGIN
  nxtstate <= state;
  output <= 0;
  IF reset = '0' THEN
   nxtstate <= S14;
  ELSE
   CASE state IS
    WHEN S0 =>
     IF (input = 1) THEN
      output <= 1;
      nxtstate <= S12;
     ELSE
      output <= 1;
      nxtstate <= S2;
     END IF;
    WHEN S1 =>
     IF (input = 1) THEN
      output <= 0;
      nxtstate <= S12;
     ELSE
      output <= 0;
      nxtstate <= S2;
     END IF;
          ...
          ...
WHEN S31 =>
     IF (input = 0) THEN
      output <= 0;
      nxtstate <= S4;
     ELSE
      output <= 0;
      nxtstate <= S23;
     END IF;
   END CASE;
  END IF;
 END PROCESS behavior;

 clock: PROCESS
 BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  state <= nxtstate;
 END PROCESS clock;
END lpssd;
```

```
module Mealy_HD (clk, in, reset,
out);

 input clk, reset;
 input [0:0] in;
 output out;

 reg [1:0] out;
 reg [4:0] state;
 reg [4:0] nxtstate;

 parameter [4:0]
  S0 = 0,
  S1 = 1,
          ...
          ...
  S31 = 31;

 always @ (in or reset or state)
begin
  nxtstate = state;
  out = 0;
  if (reset) begin
   nxtstate = S14;
  end
  else begin
   case (state)
    S0:
     if (in == 1) begin
      nxtstate = S12;
      out = 1;
     end
     else begin
      nxtstate = S2;
      out = 1;
     end
    S1:
     if (in == 1) begin
      nxtstate = S12;
      out = 0;
     end
     else begin
      nxtstate = S2;
      out = 0;
     end
          ...
          ...
    S31:
     if (in == 0) begin
      nxtstate = S4;
      out = 0;
     end
     else begin
      nxtstate = S23;
      out = 0;
     end
   endcase
  end
 end

 always @ (posedge clk) begin
  state = nxtstate;
 end
endmodule
```

```
ENTITY Mealy_HD_E IS
     PORT(
     CLK, CLR : IN BIT;
     X0 : IN BIT;
     Q0, Q1, Q2, Q3, Q4 : OUT
BIT;
     Z0, Z1 : OUT BIT );
END Mealy_HD_E;

ARCHITECTURE RTL OF Mealy_HD_E IS

SIGNAL  VE0, VE1, VE2, VE3, VE4:
BIT;
SIGNAL D4, D3, D2, D1, D0 : BIT;

 BEGIN

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;

          ...
          ...

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE4 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE4 <= D4;
    END IF;
    Q4 <= VE4;
END PROCESS;

  D4 <=  ( (VE4) AND (VE2)) OR
((VE4) AND (VE1)) OR ((VE4) AND
(VE0)) OR ((X0) AND (VE4)) OR
(NOT(X0) AND NOT(VE4) AND
NOT(VE2) AND NOT(VE1) AND
NOT(VE0));
          ...
          ...
  Z0 <=  ( (VE3) AND (VE2) AND
NOT(VE1) AND (VE0)) OR ((VE3) AND
NOT(VE2) AND (VE1) AND (VE0)) OR
(NOT(X0) AND NOT(VE4) AND (VE3)
AND NOT(VE2) AND NOT(VE1) AND
NOT(VE0)) OR (NOT(VE3) AND (VE2)
AND (VE1) AND (VE0)) OR (NOT(VE3)
AND NOT(VE2) AND NOT(VE1) AND
(VE0));

END RTL;
```

(a) VHDL comportamental **(b) Verilog HDL comportamental** **(c) VHDL funcional**

Figura 5.12. Descrições obtidas pela ferramenta desenvolvida para o código HDB3.

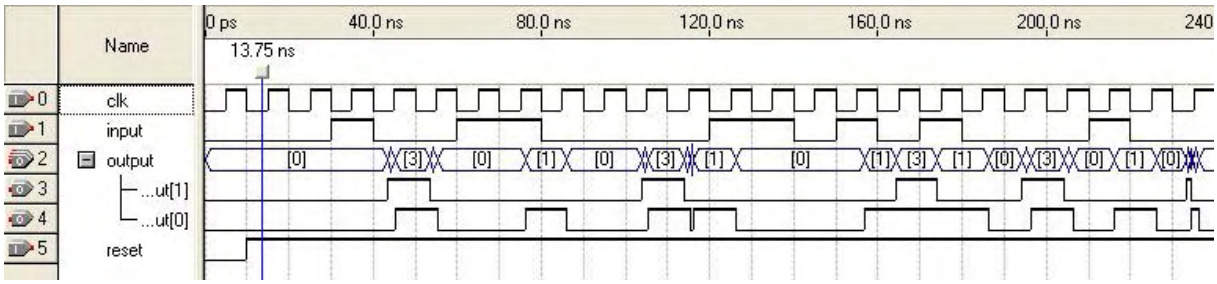

Figura 5.13. Simulação do código HDB3 no Quartus com o VHDL comportamental.

Tabela 5.3. Síntese obtida pelo ambiente Quartus para o código HDB3.

	VHDL (SF²HDL)	Verilog (SF²HDL)	VHDL (TAB2VHDL)	Total
Elementos lógicos	44 (0,23 %)	44 (0,23 %)	8 (0,042 %)	18.752
Funções combinacionais	44 (0,23 %)	44 (0,23 %)	7 (0,037 %)	18.752
Registradores	32 (0,17 %)	32 (0,17 %)	5 (0,026 %)	18.752
Pinos	5 (1,58 %)	5 (1,58 %)	10 (3,17 %)	315

5.1.4. Código MLT-3

Na Figura 5.10, é ilustrado o código de linha MLT-3 em ambiente Stateflow® descrito pelo modelo de Mealy, em que o estado A representa o estado inicial da máquina, os estados B, C e D provocam a alternância da polaridade do sinal entre -1, 0 e 1. A variável *u* (rotulada nas transições) representa a entrada em um instante de tempo discreto *t*, para que seja obtida a saída *z* nesse determinado instante de tempo. O resultado da simulação do código MLT-3 é apresentado na Figura 5.11.

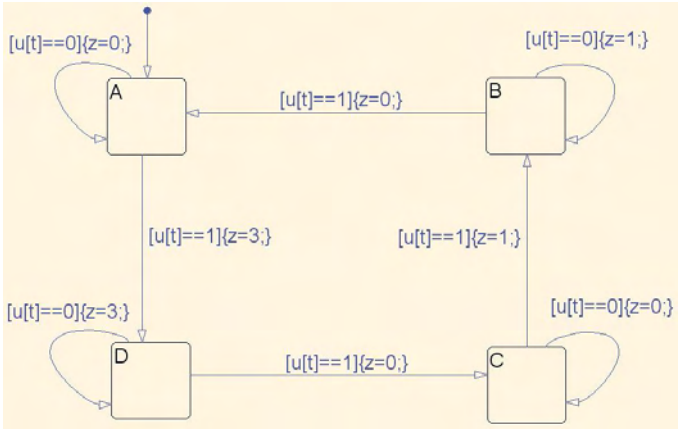

Figura 5.14. Código de linha MLT-3 em ambiente Stateflow®.

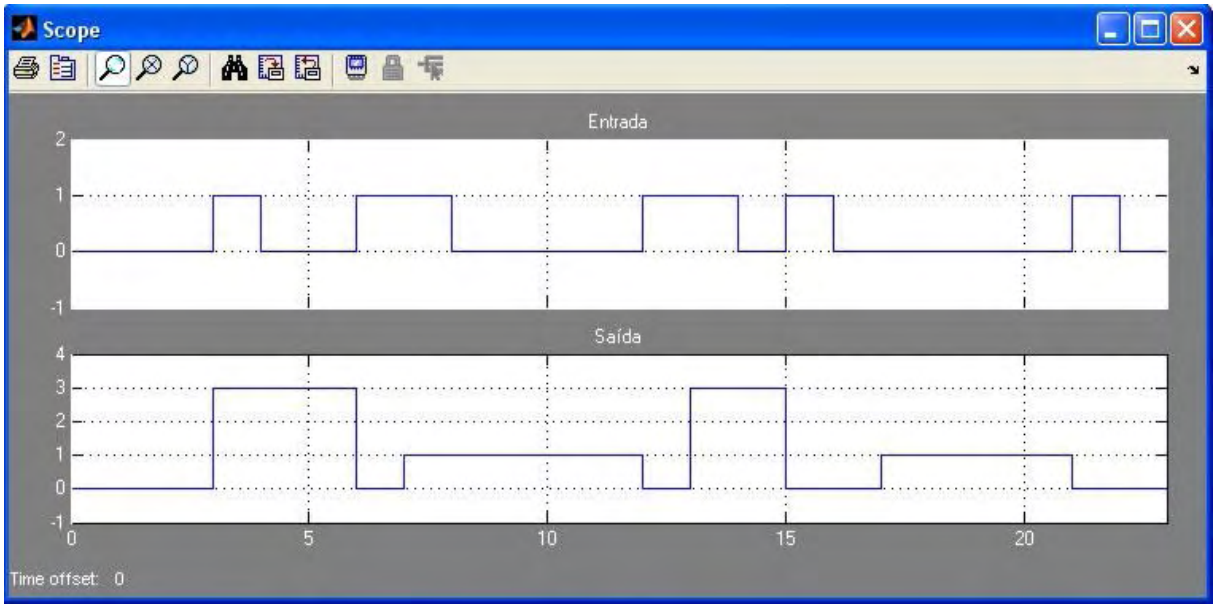

Figura 5.15. Simulação do código MLT-3 no Stateflow.

Após a modelagem em Stateflow®, a tradução realizada pelo SF^{2}HDL, a minimização do programa TABELA e a tradução do programa TAB2VHDL, foram gerados os códigos completos apresentados nas Figuras 5.12 (a), (b), (c), sendo eles o VHDL comportamental, Verilog comportamental e VHDL funcional, respectivamente.

Na Figura 5.13, apresentam-se os resultados da simulação no Quartus II para o Verilog comportamental.

No processo de síntese realizado pelo Quartus II, a descrição em VHDL funcional apresentou o melhor resultado, lembrando que este foi previamente minimizado pelo TABELA, porém, utilizando dois pinos a mais que nas outras duas descrições. Já o Verilog comportamental apresentou o pior resultado das três descrições, utilizando um elemento lógico e uma função combinacional a mais em relação ao VHDL comportamental, e quatro elementos lógicos e quatro funções combinacionais a mais em comparação ao VHDL funcional, conforme apresentado na Tabela 5.3.

```
ENTITY Mealy_ML_C IS
 PORT(
  input : IN INTEGER RANGE 0 TO 1;
  output : OUT INTEGER RANGE 0 TO 3;
  reset : IN BIT;
  clk : IN BIT);
END Mealy_ML_C;

ARCHITECTURE lpssd OF Mealy_ML_C IS

TYPE type_state IS (S0,S1,S2,S3);
SIGNAL state , nxtstate : type_state;

BEGIN
 behavior: PROCESS (reset, input,
state)
 BEGIN
  nxtstate <= state;
  output <= 0;
  IF reset = '0' THEN
   nxtstate <= S0;
  ELSE
   CASE state IS
    WHEN S0 =>
     IF (input = 1) THEN
      output <= 3;
      nxtstate <= S3;
     ELSE
      output <= 0;
      nxtstate <= S0;
     END IF;
          ...
          ...
    WHEN S3 =>
     IF (input = 1) THEN
      output <= 0;
      nxtstate <= S2;
     ELSE
      output <= 3;
      nxtstate <= S3;
     END IF;
   END CASE;
  END IF;
 END PROCESS behavior;

 clock: PROCESS
 BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  state <= nxtstate;
 END PROCESS clock;
END lpssd;
```

(a) VHDL comportamental

```
module Mealy_ML (clk, in, reset,
out);

 input clk, reset;
 input [0:0] in;
 output out;
 reg [1:0] out;
 reg [1:0] state;
 reg [1:0] nxtstate;
 parameter [1:0]
  S0 = 0,
  S1 = 1,
  S2 = 2,
  S3 = 3;

 always @ (in or reset or state)
begin
  nxtstate = state;
  out = 0;
  if (reset) begin
   nxtstate = S0;
  end
  else begin
   case (state)
    S0:
     if (in == 1) begin
      nxtstate = S3;
      out = 3;
     end
     else begin
      nxtstate = S0;
      out = 0;
     end
          ...
          ...
    S3:
     if (in == 1) begin
      nxtstate = S2;
      out = 0;
     end
     else begin
      nxtstate = S3;
      out = 3;
     end
   endcase
  end
 end

 always @ (posedge clk) begin
  state = nxtstate;
 end
endmodule
```

(b) Verilog HDL comportamental

```
ENTITY Mealy_ML_E IS
     PORT(
     CLK, CLR : IN BIT;
     X0 : IN BIT;
     Q0, Q1 : OUT BIT;
     Z0, Z1 : OUT BIT );
END Mealy_ML_E;

ARCHITECTURE RTL OF Mealy_ML_E IS

SIGNAL  VE0, VE1: BIT;
SIGNAL D1, D0 : BIT;

 BEGIN
PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE1 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE1 <= D1;
    END IF;
    Q1 <= VE1;
END PROCESS;

  D1 <=  ( (VE1) AND (VE0)) OR
(NOT(X0) AND (VE1)) OR ((X0) AND
NOT(VE1) AND NOT(VE0));
  D0 <=  ( NOT(X0) AND (VE0)) OR
((X0) AND NOT(VE0));
  Z1 <=  ( NOT(X0) AND (VE1) AND
(VE0)) OR ((X0) AND NOT(VE1) AND
NOT(VE0));
  Z0 <=  ( NOT(X0) AND (VE0)) OR
((X0) AND NOT(VE0));

END RTL;
```

(c) VHDL funcional

Figura 5.16. Descrições obtidas pela ferramenta desenvolvida para o código MLT-3.

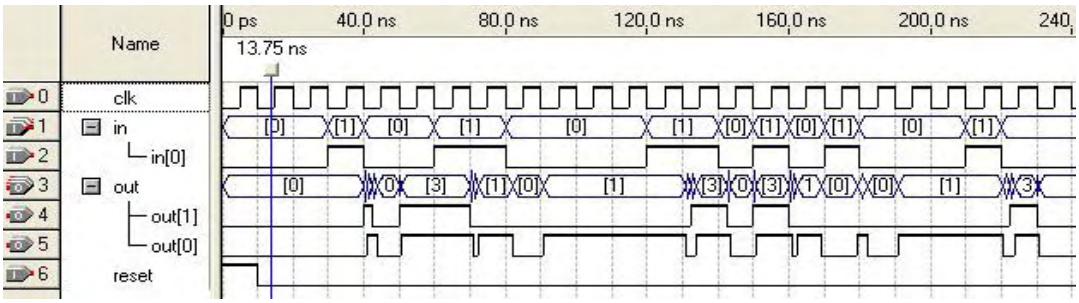

Figura 5.17. Simulação do código MLT-3 no Quartus com o Verilog.

Tabela 5.4. Síntese obtida pelo ambiente Quartus para o código MLT-3.

	VHDL (SF^2HDL)	Verilog (SF^2HDL)	VHDL (TAB2VHDL)	Total
Elementos lógicos	6 (0,031 %)	7 (0,037 %)	3 (0,015 %)	18.752
Funções combinacionais	6 (0,031 %)	7 (0,037 %)	3 (0,015 %)	18.752
Registradores	4 (0,021 %)	4 (0,021 %)	2 (0,01 %)	18.752
Pinos	5 (1,58 %)	5 (1,58 %)	7 (2,22 %)	315

5.1.5. Código 2B1Q

No caso do código 2B1Q, foi modelada uma máquina de estados descrita pelo modelo de Moore completamente especificada, e convencionou-se em utilizar a codificação decimal. Dessa forma, os pares de bits de entrada 00 foram codificados como 0 decimal, os bits de entrada 01 como 1 decimal, os bits de entrada 10 como 2 em decimal e os bits de entrada 11 como 3 em decimal. Para representar a saída, foi utilizado um bit a mais para representar a polaridade do sinal, sendo, 1 para polaridade negativa e 0 para polaridade positiva, assim, o sinal de saída -3 foi representado como 7 (“111” em binário), o sinal -1 foi representado como 5 (“101” em binário), o sinal +3 (“011” em binário) foi representado como 3 e o sinal +1 foi representado como 1 (“001” em binário). Na Figura 5.14, é ilustrada essa codificação decimal do 2B1Q em ambiente Stateflow®.

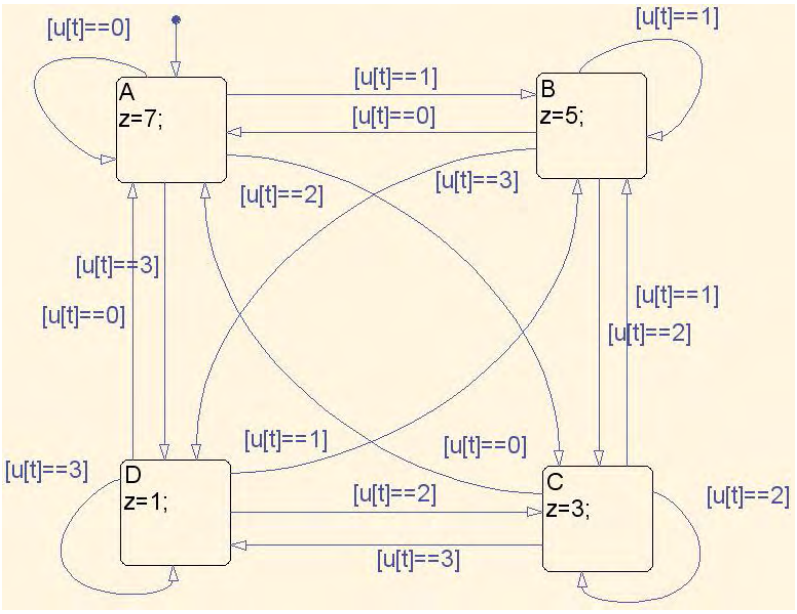

Figura 5.18. Código de linha 2B1Q em ambiente Stateflow®.

Após a tradução do SF^{2}HDL, a minimização do programa TABELA e a tradução do TAB2VHDL, foram gerados os códigos VHDL comportamental, Verilog comportamental e VHDL funcional apresentados nas Figuras 5.15 (a), (b) e (c) , respectivamente.

```
ENTITY Moore_2B_C IS
 PORT(
  input : IN INTEGER RANGE 0 TO 3;
  output : OUT INTEGER RANGE 0 TO 7;
  reset : IN BIT;
  clk : IN BIT);
END Moore_2B_C;

ARCHITECTURE lpssd OF Moore_2B_C IS

TYPE type_state IS (S0,S1,S2,S3);
SIGNAL state , nxtstate : type_state;

BEGIN
 behavior: PROCESS (reset, input,
state)
 BEGIN
  nxtstate <= state;
  output <= 0;
  IF reset = '1' THEN
   nxtstate <= S0;
  ELSE
   CASE state IS
    WHEN S0 =>
     output <= 7;
     IF (input = 1) THEN
      nxtstate <= S1;
     ELSIF (input = 0) THEN
      nxtstate <= S0;
     ELSIF (input = 2) THEN
      nxtstate <= S2;
     ELSE
      nxtstate <= S3;
     END IF;
          ...
          ...
    WHEN S3 =>
     output <= 1;
     IF (input = 3) THEN
      nxtstate <= S3;
     ELSIF (input = 2) THEN
      nxtstate <= S2;
     ELSIF (input = 0) THEN
      nxtstate <= S0;
     ELSE
      nxtstate <= S1;
     END IF;
   END CASE;
  END IF;
 END PROCESS behavior;

 clock: PROCESS
 BEGIN
  WAIT UNTIL clk'EVENT AND clk = '1';
  state <= nxtstate;
 END PROCESS clock;
END lpssd;
```

```
module Moore_2B (clk, in, reset,
out);
 input clk, reset;
 input [1:0] in;
 output out;
 reg [2:0] out;
 reg [1:0] state;
 reg [1:0] nxtstate;
 parameter [1:0]
  S0 = 0,
  S1 = 1,
  S2 = 2,
  S3 = 3;
 always @ (in or reset or state)
begin
  nxtstate = state;
  out = 0;
  if (reset) begin
   nxtstate = S0;
  end
  else begin
   case (state)
    S0: begin
     out = 7;
     if (in == 1) begin
      nxtstate = S1;
     end
     else if (in == 0) begin
      nxtstate = S0;
     end
     else if (in == 2) begin
      nxtstate = S2;
     end
     else begin
      nxtstate = S3;
     end
    end
          ...
          ...
    S3: begin
     out = 1;
     if (in == 3) begin
      nxtstate = S3;
     end
     else if (in == 2) begin
      nxtstate = S2;
     end
     else if (in == 0) begin
      nxtstate = S0;
     end
     else begin
      nxtstate = S1;
     end
    end
   endcase
  end
 end
 always @ (posedge clk) begin
  state = nxtstate;
 end
endmodule
```

```
ENTITY Moore_2B_E IS
     PORT(
     CLK, CLR : IN BIT;
     X0, X1 : IN BIT;
     Q0, Q1 : OUT BIT;
     Z0, Z1, Z2 : OUT BIT );
END Moore_2B_E;

ARCHITECTURE RTL OF Moore_2B_E IS

SIGNAL  VE0, VE1: BIT;
SIGNAL D1, D0 : BIT;

 BEGIN

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE0 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE0 <= D0;
    END IF;
    Q0 <= VE0;
END PROCESS;

PROCESS(CLK, CLR)
BEGIN
    IF CLR = '0' THEN
       VE1 <= '0';
    ELSIF CLK'EVENT and CLK = '1'
THEN
       VE1 <= D1;
    END IF;
    Q1 <= VE1;
END PROCESS;

  D1 <=  ( (X1));
  D0 <=  ( (X0));
  Z2 <=  ( NOT(VE1));
  Z1 <=  ( NOT(VE0));
  Z0 <=   '1';

END RTL;
```

(a) VHDL comportamental **(b) Verilog HDL comportamental** **(c) VHDL funcional**

Figura 5.19. Descrições obtidas pela ferramenta desenvolvida para o código 2B1Q.

Todas as simulações apresentaram os resultados esperados, assim, na Figura 5.16, apresentam-se os resultados da simulação da descrição em Verilog comportamental e, na Figura 5.17, apresenta-se o resultado em VHDL funcional.

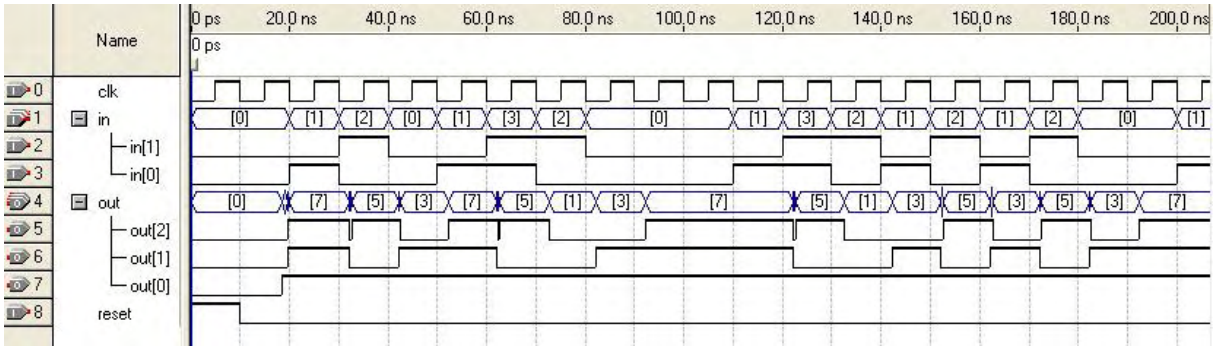

Figura 5.20. Simulação do código 2B1Q no Quartus com o Verilog.

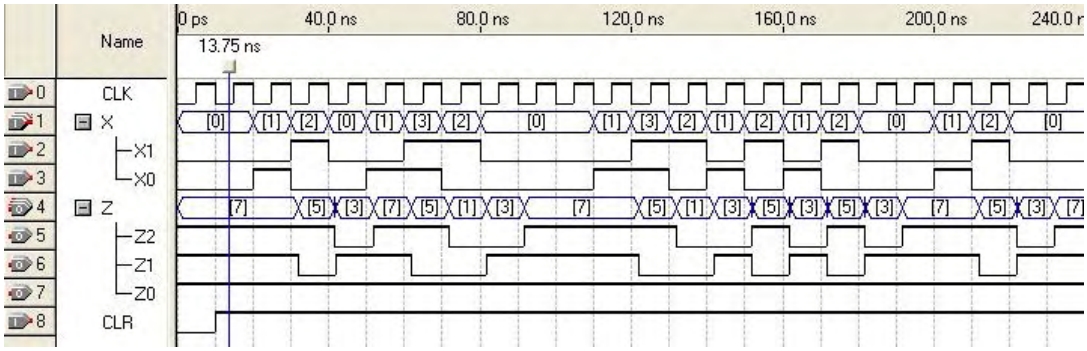

Figura 5.21. Simulação do código 2B1Q no Quartus com o VHDL funcional.

De acordo com a minimização realizada pelo programa TABELA, a descrição em VHDL funcional apresentou o melhor resultado na síntese realizada pelo Quartus II. Uma importante ressalva é que não foi utilizada nenhuma função combinacional da FPGA nessa descrição, porém foram utilizados três pinos a mais que nas outras duas descrições. Já as descrições comportamentais em VHDL e Verilog apresentaram exatamente os mesmos resultados. Os resultados da síntese obtida podem ser visualizados na Tabela 5.4.

Tabela 5.5. Síntese obtida pelo ambiente Quartus para o código 2B1Q.

	VHDL (SF^2HDL)	**Verilog (SF^2HDL)**	**VHDL (TAB2VHDL)**	**Total**
Elementos lógicos	5 (0,026 %)	5 (0,026 %)	2 (0,01 %)	18.752
Funções combinacionais	5 (0,026 %)	5 (0,026 %)	0 (0 %)	18.752
Registradores	3 (0,015 %)	3 (0,015 %)	2 (0,01 %)	18.752
Pinos	7 (2,22 %)	7 (2,22 %)	9 (2,87 %)	315

A partir da extração dos casos estudados, foi avaliado o desempenho da ferramenta desenvolvida. Para a realização das traduções e a avaliação do consumo computacional, foi utilizado um computador com processador Athlon 64 X2, com 2GB de memória RAM e sistema operacional Windows XP *Service Pack* 3.

Ambos os casos apresentaram excelentes tempos na tradução, sendo que, na tradução do código HDB3, a ferramenta consumiu o maior tempo de processamento, com cerca de 1,563 segundos, devido também a maior complexidade do modelo em relação aos outros casos analisados. O código MLT-3 levou o menor tempo de processamento, cerca de 47 milissegundos.

Já em relação ao consumo de memória RAM (*Random Access Memory*), os casos analisados tiveram um consumo médio de 8,58 MB, porém, o código 2B1Q consumiu uma menor quantidade de memória RAM, cerca de 8,28 MB. O maior consumo de memória foi

obtido pelo código HDB3, consumindo de 8,75 MB. O resultado completo da avaliação do consumo computacional, para os casos analisados, apresenta-se na Tabela 5.6.

Tabela 5.6. Consumo computacional dos casos analisados.

Caso	Tempo	RAM
AMI (Moore)	63 ms	8,42 MB
HDB1 (Mealy)	93 ms	8,74 MB
HDB3 (Mealy)	1,563 s	8,75 MB
MLT-3 (Mealy)	47 ms	8,71 MB
2B1Q (Moore)	109 ms	8,28 MB

É importante destacar que esses resultados podem variar de acordo com o microcomputador utilizado no desenvolvimento do projeto.

5.2. Avaliação da ferramenta MS²SV versão 1.7

Com as características de funcionamento dos conversores de dados apresentadas no Capítulo 3, foi possível a modelagem em nível de sistema em ambiente Simulink®, utilizando apenas primitivas básicas (como operadores lógicos, ganhos, somadores, *flip-flops*, entre outros) existentes nos *toolboxes* do Simulink®. No caso de estruturas mais complexas, o Simulink® fornece a possibilidade da criação de subsistemas, o que simplifica o desenvolvimento, permitindo uma melhor compreensão e visualização do conversor modelado. É o caso da malha R/2R, que foi utilizada por todos os modelos de conversores estudados, dessa forma, foi criado um subsistema denominado "*Ladder_r2r*" para ser utilizado em todos os modelos. Na Figura 5.22, é ilustrado esse subsistema, representando a malha R/2R. Nela estão contidas oito entradas paralelas, multiplicadas pelos pesos e, posteriormente, somando-os para se obter a saída analógica.

É importante ressaltar que os portos digitais devem ser seguidos do sufixo "_D", para que o programa possa reconhecê-lo como um porto digital.

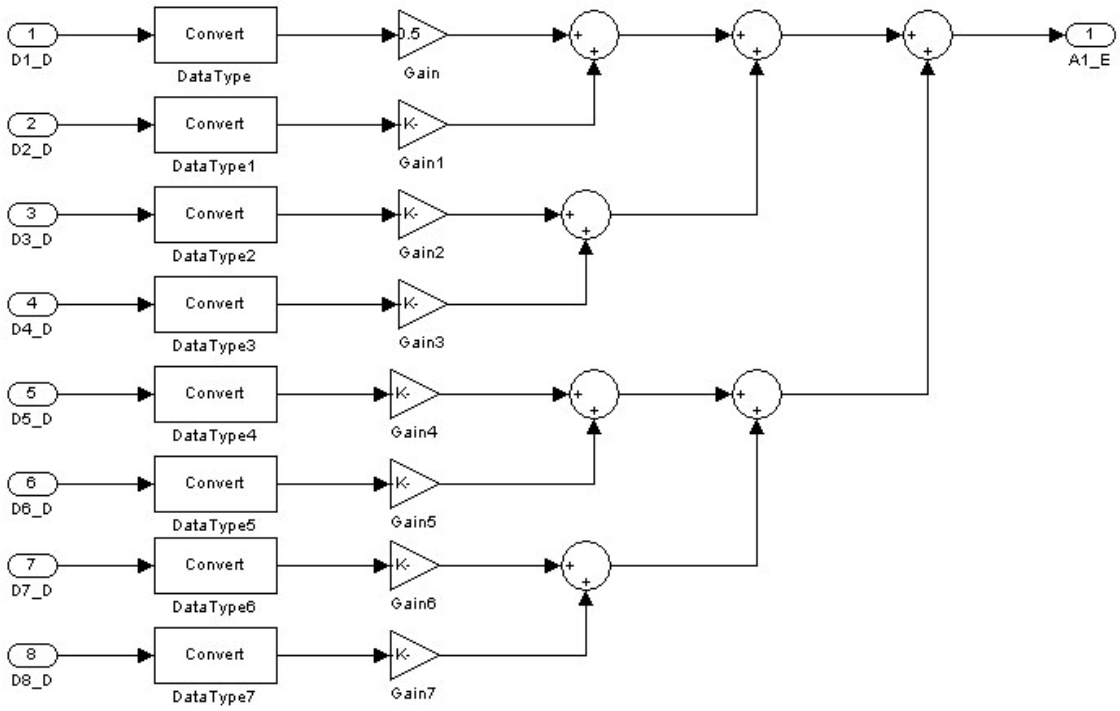

Figura 5.22. Malha R/2R modelada em ambiente Simulink®.

O valor de cada peso segue o mesmo princípio apresentado no Capítulo 2, com o LSB (*Least Significant Bit*) recebendo o menor peso, e o MSB (*Most Significant Bit*), recebendo o maior peso. Na Tabela 5.7, apresentam-se os pesos utilizados em todos os bits da malha R/2R.

Outro padrão adotado foi a utilização da estrutura de *flip-flop* utilizada nos modelos: AD7524 e AD7528, pois, esses modelos necessitam de uma estrutura capaz de armazenar as entradas digitais, antes de convertê-las em uma saída analógica. Nesse caso, foi modelado um novo subsistema denominado "*Data Latch*", o qual possui oito *flip-flops* do tipo *latch*, para armazenar as entradas. Esse modelo de *flip-flop* possui apenas duas entradas, uma para armazenar o bit de entrada e outra para habilitar ou não a saída da informação armazenada e duas saídas, sendo que a segunda representa a primeira saída negada (invertida). A escolha desse *flip-flop* deu-se pelo fato de que não é necessário um sincronismo com o sinal de *clock* para modelar os conversores AD7524 e AD7528 em alto nível de abstração. A estrutura do subsistema "*Data Latch*" é apresentada na Figura 5.23.

Tabela 5.7. Valor do peso de cada bit na malha R/2R.

Relevância de cada bit	Valor do peso
1 (MSB)	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8 (LSB)	0.00390625

Na avaliação realizada com o conversor AD5450, necessitou-se de uma estrutura similar, porém, utilizando *flip-flops* do tipo D, pois esse modelo necessita do sincronismo com sinal de *clock*, já que a entrada é feita de forma serial ao invés de paralela como apresentado nos outros modelos estudados.

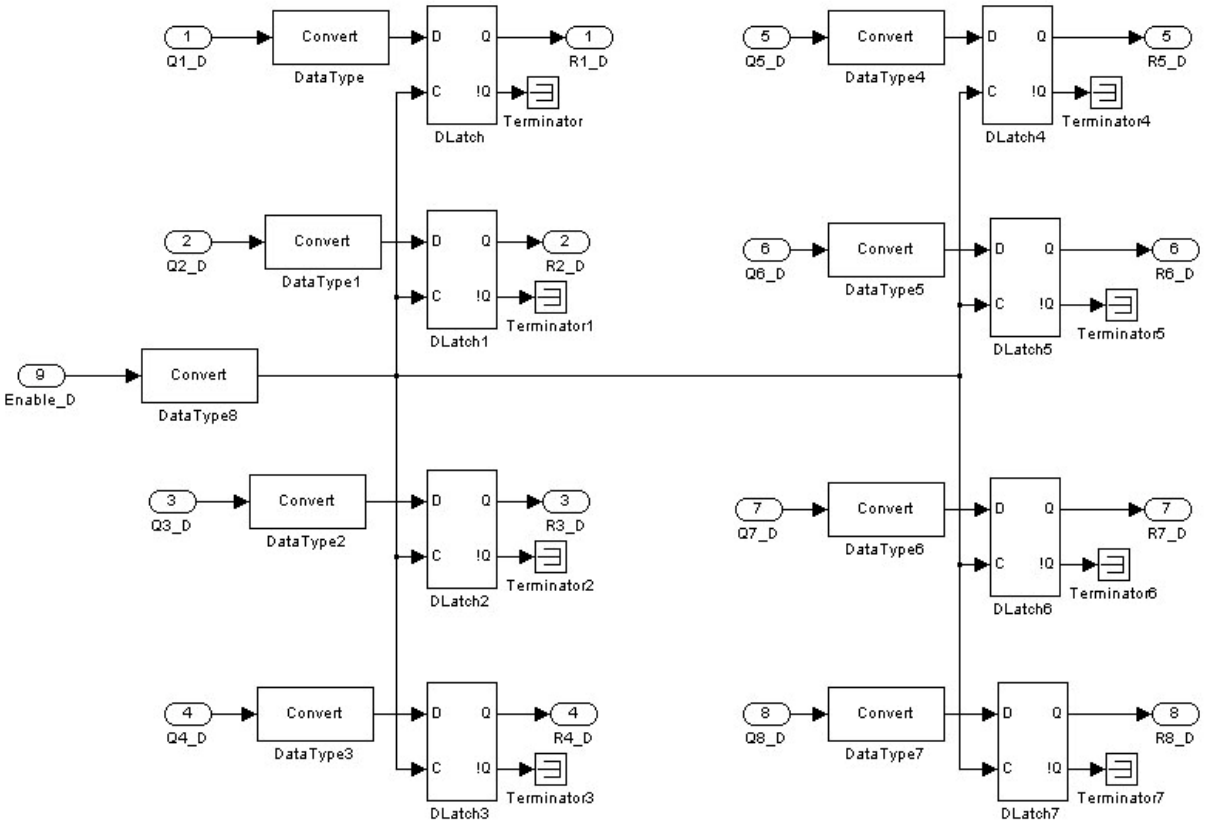

Figura 5.23. Subsistema *Data Latch* modelado em ambiente Simulink®.

5.2.1. Conversor DAC08

Dos quatro modelos estudados, o DAC08 é o conversor de dados com o funcionamento mais simplificado, sendo formado apenas pela malha R/2R, com sua saída sendo multiplicada pela tensão de referência (valor máximo obtido na saída) e formado por duas saídas, sendo uma adicional que representa a diferença entre a saída analógica e a tensão de referência. Na Figura 5.24 é, ilustrado o DAC08 em ambiente Simulink®.

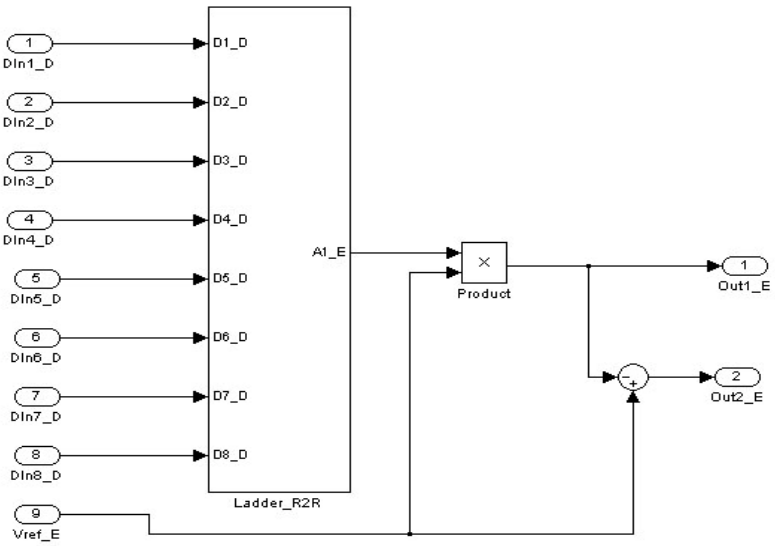

Figura 5.24. Conversor DAC08 em ambiente Simulink®.

Para a realização da simulação no ambiente Simulink®, foi utilizada uma tensão de referência de 10 V para esse modelo de conversor, a qual também foi utilizada nos outros modelos paralelos. Também sendo utilizados blocos geradores de pulso, porém, com períodos de atrasos ajustados, para que se chegasse a uma entrada alternada de 2^8 bits diferentes. Dessa forma, convencionou-se em utilizar os atrasos de frequência apresentados na Tabela 5.8.

Tabela 5.8. Atrasos utilizados nos geradores de pulso.

Relevância de cada bit	Atraso na frequência
1 (MSB)	1.0
2	0.5
3	0.25
4	0.125
5	0.0625
6	0.03125
7	0.015625
8 (LSB)	0.0078125

Na Figura 5.25, é apresentada a forma de onda obtida na simulação no ambiente Simulink® com os geradores de pulso, sendo obtida uma onda em forma de rampa, com um ciclo repetido de acordo com a frequência utilizada na simulação.

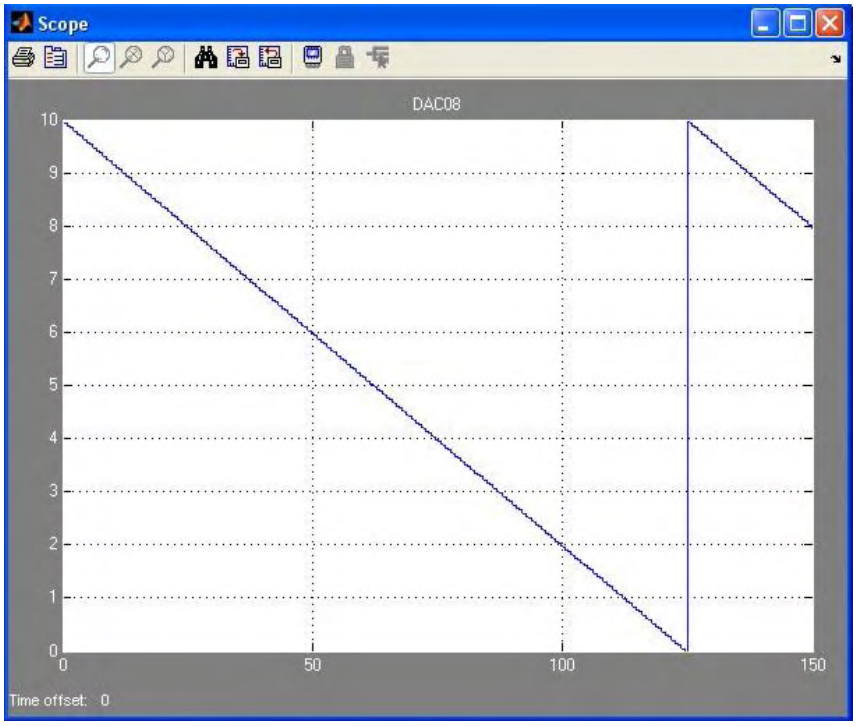

Figura 5.25. Simulação do DAC08 no Simulink.

Após a simulação do modelo, foi realizada a tradução do modelo pelo MS^{2}SV, gerando duas descrições em VHDL-AMS, que devem estar no *testbench* ativo do projeto no ambiente SystemVision™. Nesse caso, a inclusão no *testbench* deve ser feita por meio do SystemVision™, e não pela ferramenta MS^{2}SV. O código completo obtido na tradução do DAC08 é apresentado no Apêndice A. Já na Figura 5.26, apresentam-se os resultados da simulação no ambiente SystemVision™, utilizando-se também os geradores de pulso com os mesmos atrasos utilizados no Simulink® apresentados na Tabela 5.8.

A simulação da Figura 5.26 apresenta o mesmo resultado que a simulação apresentada na Figura 5.25. A única diferença é que, no ambiente Simulink®, o sinal do *clock* é iniciado em nível lógico alto (1), o contrário do que ocorre no ambiente SystemVision™, onde o *clock* é iniciado em nível lógico baixo (0).

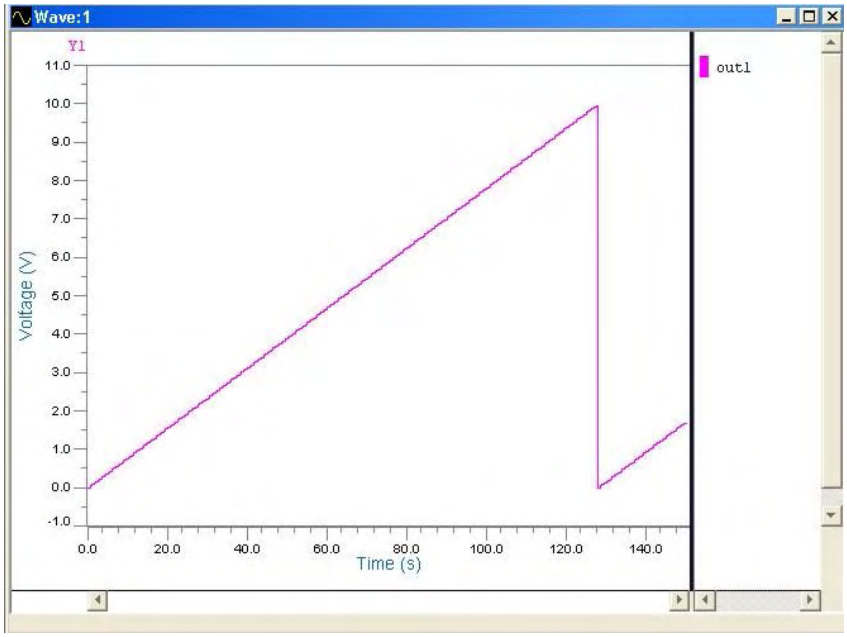

Figura 5.26. Simulação do DAC08 no SystemVision™.

Para validar a metodologia, também foi utilizado um código adicional em VHDL, para a leitura de um arquivo de entrada, o qual possibilitou a geração de uma forma de onda senoidal na saída do conversor. Essa forma de onda é apresentada na Figura 5.27. A listagem a seguir mostra a código VHDL empregado na leitura do arquivo de simulação.

```
use std.textio.all;
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.std_logic_textio.all;

entity lerarqp is
  port (relogio: in std_logic;
        saida: out std_ulogic_vector (8 downto 1));
end entity lerarqp;

architecture behavior of lerarqp is
begin
   testing: process
     file f_in: text open READ_MODE is "input.dat";
     variable L: LINE;
     variable L_BIT: std_ulogic_vector (8 downto 1);
   begin
     while not endfile(f_in) loop
        readline (f_in,L);
        hread (L,L_BIT);
        wait until (relogio'event) and (relogio='1');
        saida <= L_BIT;
     end loop;
   end process testing;
end architecture behavior;
```

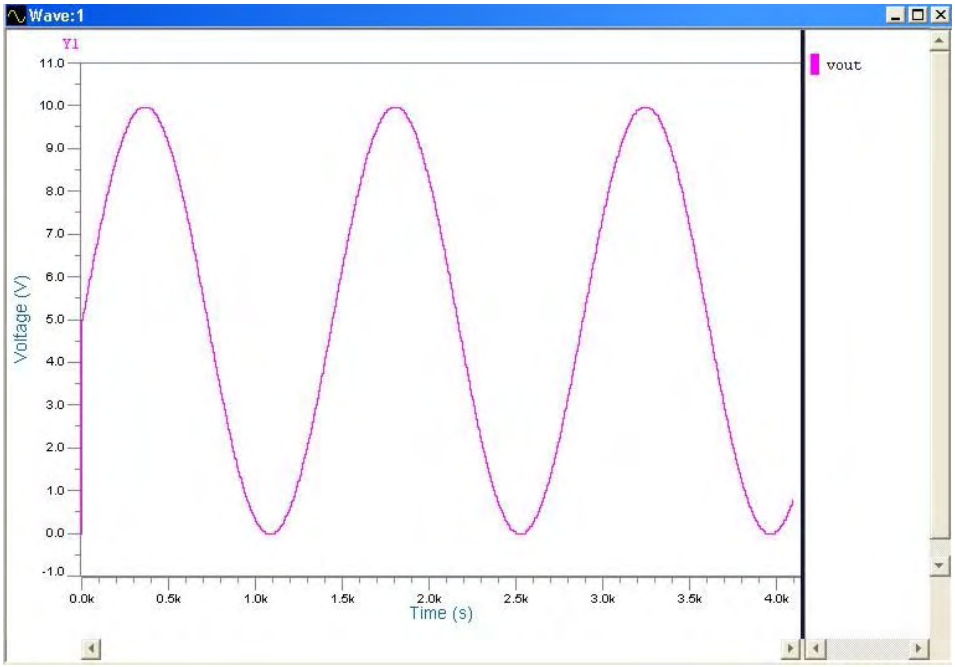

Figura 5.27. Forma de onda senoidal gerada para o DAC08.

5.2.2. Conversor AD7524

Para a modelagem do conversor AD7524, foi necessária a criação de um circuito lógico capaz de controlar o modo de escrita e de espera, conforme a especificação do manual do conversor, formado basicamente por um operador AND com as entradas negadas. Esse circuito lógico foi denominado "*S_NOR*" e seu modelo em ambiente Simulink® é apresentado na Figura 5.28.

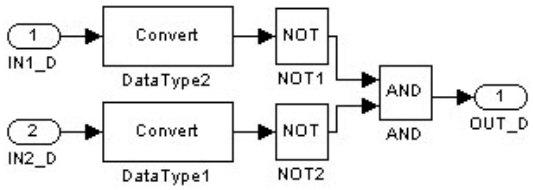

Figura 5.28. Circuito S_NOR em ambiente Simulink®.

O AD7524 possui ainda mais dois subsistemas, o "Data Latch" (representando o conjunto de *flip-flop* capaz de armazenar a última entrada válida) e a "*Ladder_r2r*" (representando a malha R/2R). O AD7524 possui ainda a tensão de referência, multiplicando a saída da malha R/2R e uma saída adicional representando a diferença entre a saída analógica e a tensão de referência. O conversor AD7524 em ambiente Simulink® é apresentado na Figura 5.29.

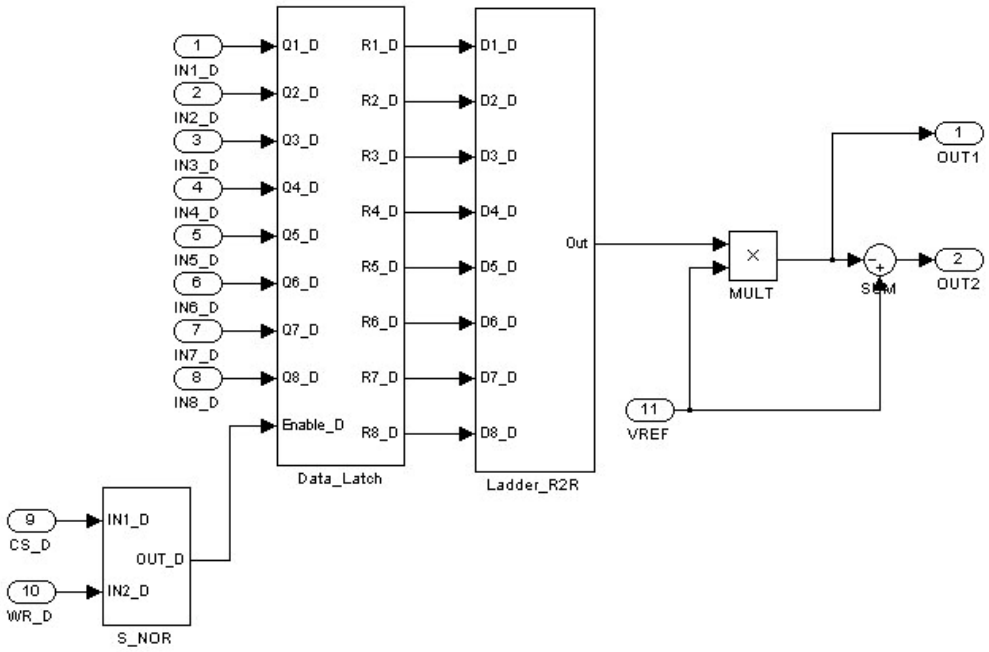

Figura 5.29. Conversor AD7524 em ambiente Simulink®.

A simulação do AD7524 no ambiente Simulink® apresentou o mesmo resultado que o DAC08, já apresentado na Figura 5.25. Já que foram utilizados os mesmos atrasos nos blocos geradores de pulso, de acordo com a Tabela 5.8 e a mesma tensão de referência delimitando o valor máximo a ser obtido na saída, cerca de 10 V. A onda em forma de rampa da simulação do AD7524 é apresentada na Figura 5.30.

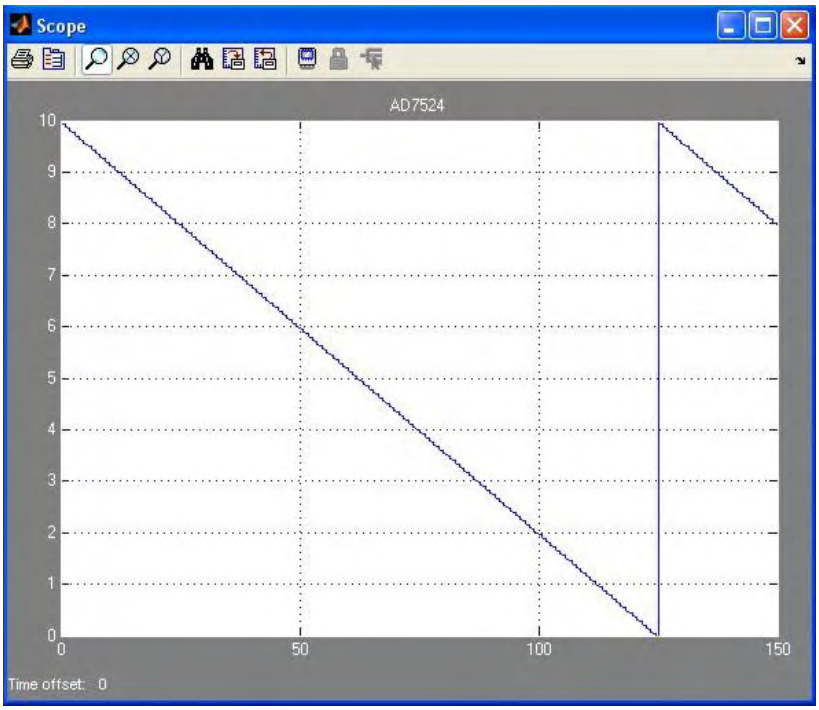

Figura 5.30. Simulação do AD7524 no Simulink®.

A tradução do AD7524 gerou quatro descrições em VHDL-AMS e o código completo é apresentado no Apêndice B.

A onda em forma de rampa também é a mesma no ambiente SystemVision™, sendo apresentada na Figura 5.31.

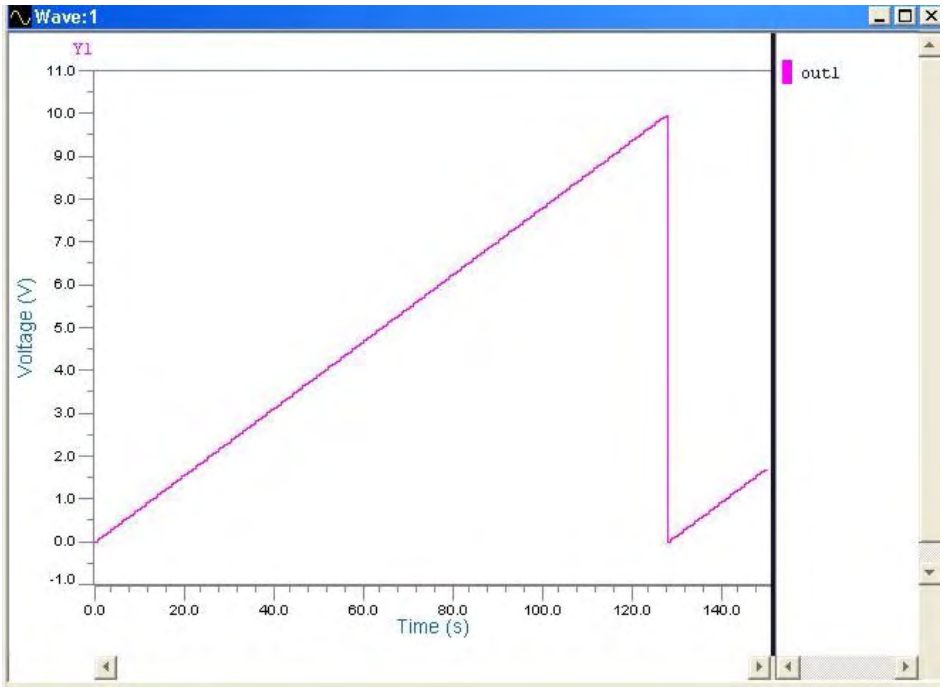

Figura 5.31. Simulação do AD7524 no SystemVision.

Para validar a descrição em VHDL-AMS obtida na tradução, da mesma forma que no caso anterior, foi utilizado o código VHDL para gerar uma forma de onda senoidal na saída analógica do conversor AD7524, a qual é apresentada na Figura 5.32.

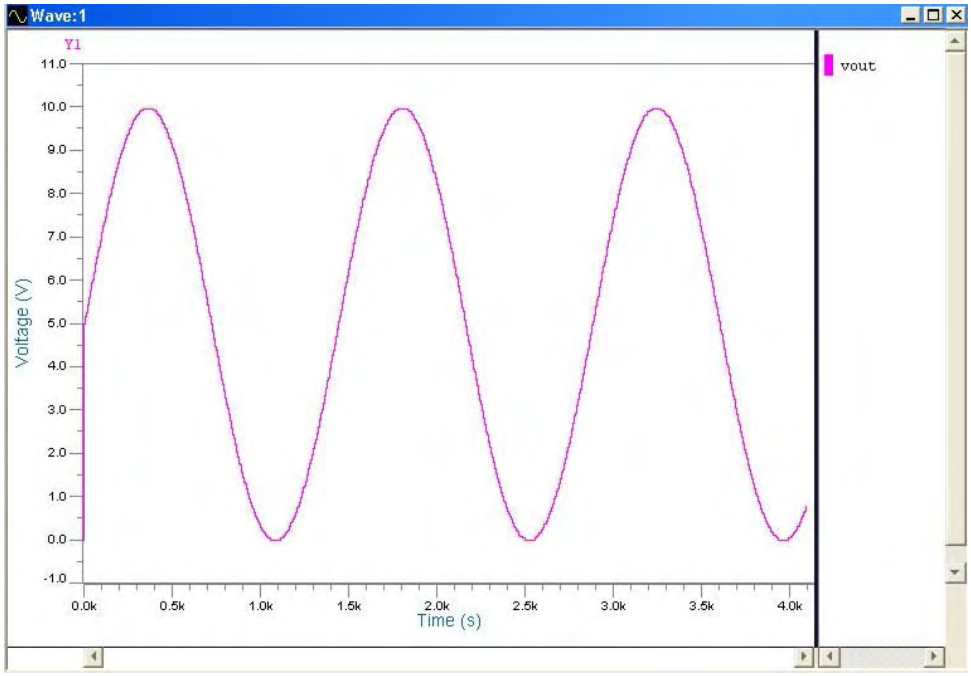

Figura 5.32. Forma de onda senoidal gerada para o AD7524.

5.2.3. Conversor AD7528

O conversor AD7528 possui uma estrutura semelhante ao AD7524, porém, com a diferença de que existem dois modelos de DAC dentro de um único CI, conforme já apresentado no Capítulo 3. Para tal, foi criado um controlador lógico, dentro de um subsistema denominado "*Control Logic*", capaz de acionar um ou outro DAC. Esse circuito é formado basicamente por três operadores AND e três operadores NOT, negando as entradas antes de serem processadas as operações de AND. Na Figura 5.33, é apresentado o controlador lógico modelado no Simulink®.

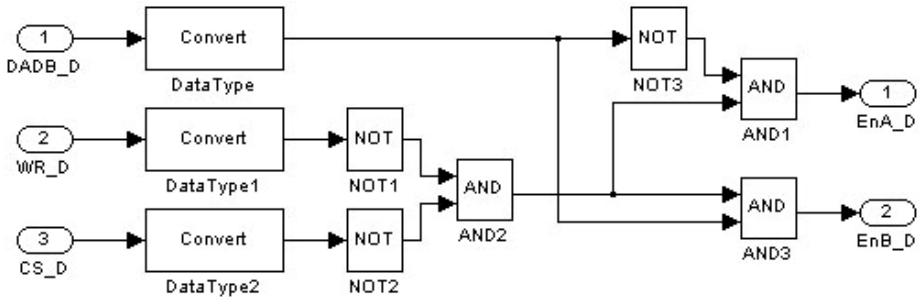

Figura 5.33. Controlador lógico modelado em ambiente Simulink®.

Na modelagem do circuito, foram criadas duas malhas R/2R e dois conjuntos de *flip-flops* do tipo *latch*, um para cada DAC e cada um dentro de um subsistema diferente. Na Figura 5.34, é ilustrado o AD7528 modelado em ambiente Simulink®. Os subsistemas de cada DAC, denominados "DAC_A" e "DAC_B", são iguais ao conversor AD7524, com exceção do subsistema "S_NOR" que não está contido no modelo e as entradas digitais são provenientes do mesmo barramento de dados.

Para diferenciar os dois conversores internos, DAC A e DAC B, foram utilizadas tensões de referência diferentes, 16 V no DAC A e 10 V no DAC B. Porém, foram utilizados blocos geradores de pulso, também com os atrasos apresentados na Tabela 5.8. O resultado da simulação no Simulink® para o DAC A e o DAC B, com a onda em forma de rampa, é apresentado nas Figuras 5.35 e 5.36, respectivamente.

A partir do modelo do conversor AD7528, foi utilizada a ferramenta MS^{2}SV para realizar a tradução para o código VHDL-AMS. Da mesma forma que nos casos anteriores, foram geradas várias descrições em VHDL-AMS, sendo uma para cada subsistema criado no modelo em Simulink®. Ressalta-se que o código completo gerado pela ferramenta MS^{2}SV para o conversor AD7528 é apresentado no Apêndice C.

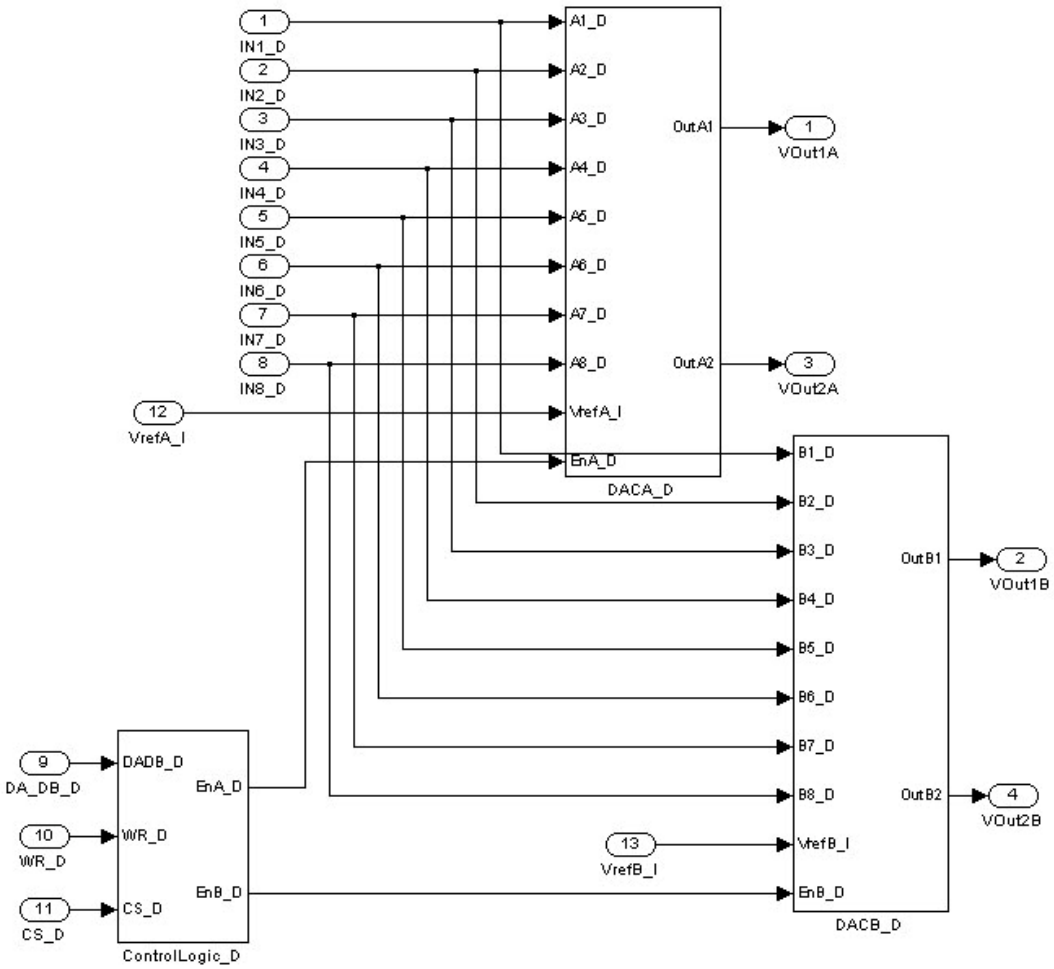

Figura 5.34. Conversor AD7528 em ambiente Simulink®.

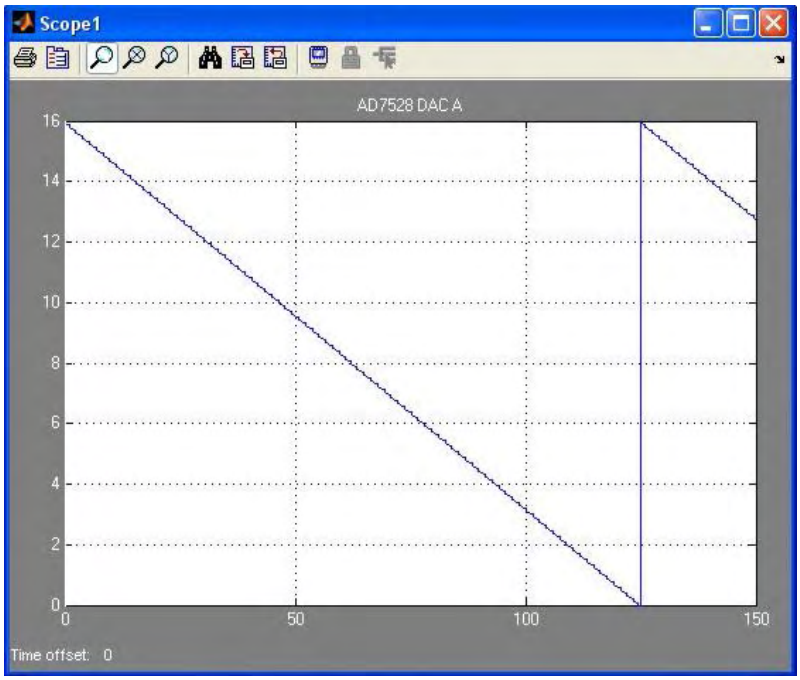

Figura 5.35. Simulação do DAC A no AD7528 no Simulink®.

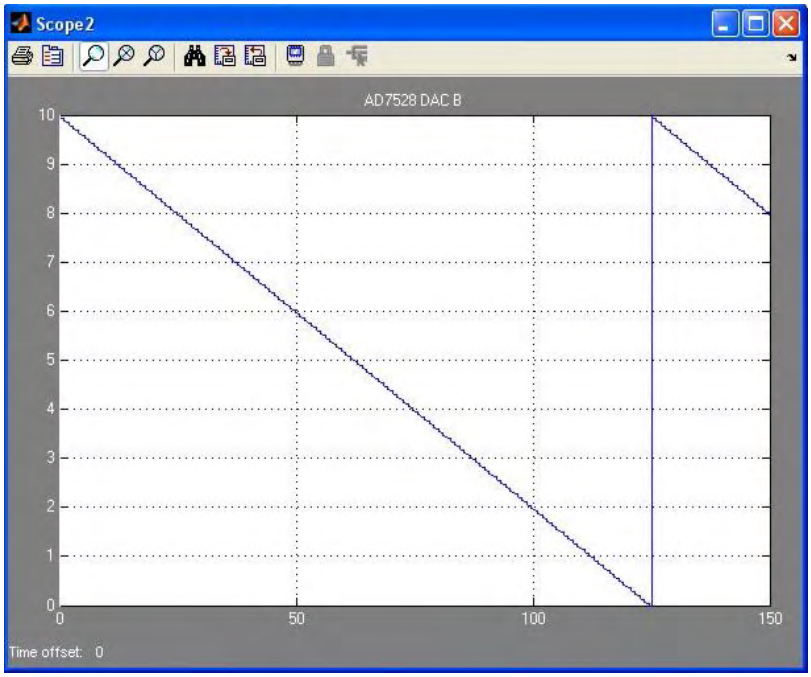

Figura 5.36. Simulação do DAC B no AD7528 no Simulink®.

A forma de onda apresentada na saída do AD7528 para o DAC A e o DAC B, para a simulação no ambiente SystemVision™, é apresentada nas Figuras 5.37 e 5.38, respectivamente.

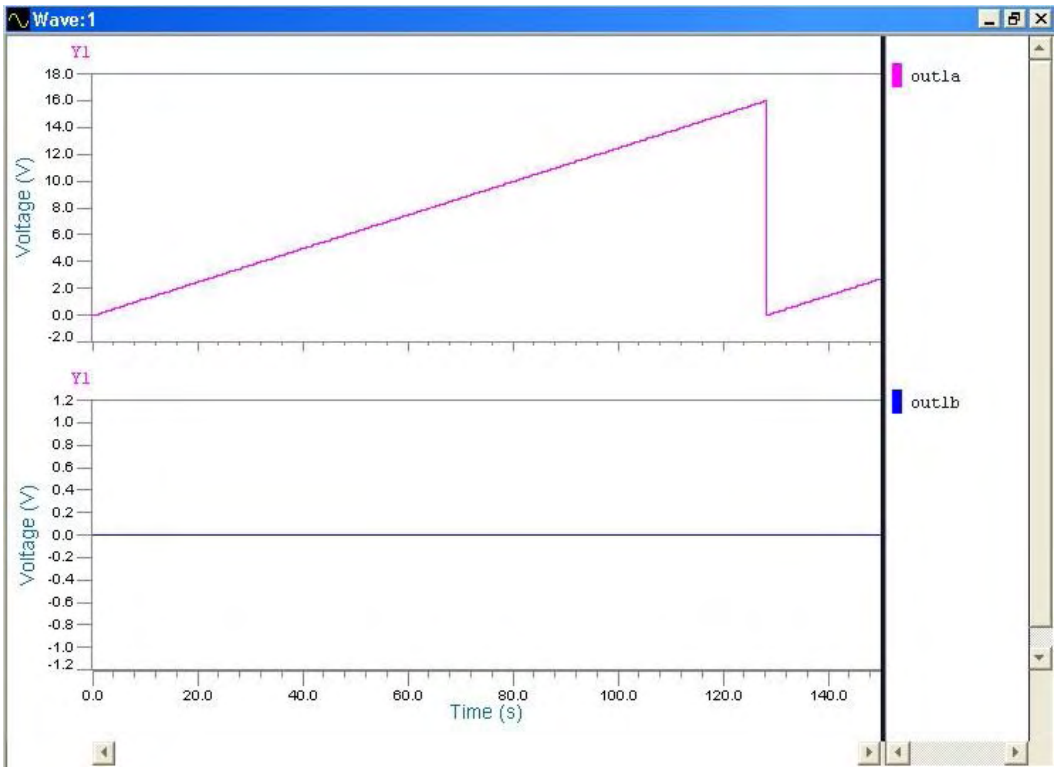

Figura 5.37. Simulação do DAC A no AD7528 no SystemVision™.

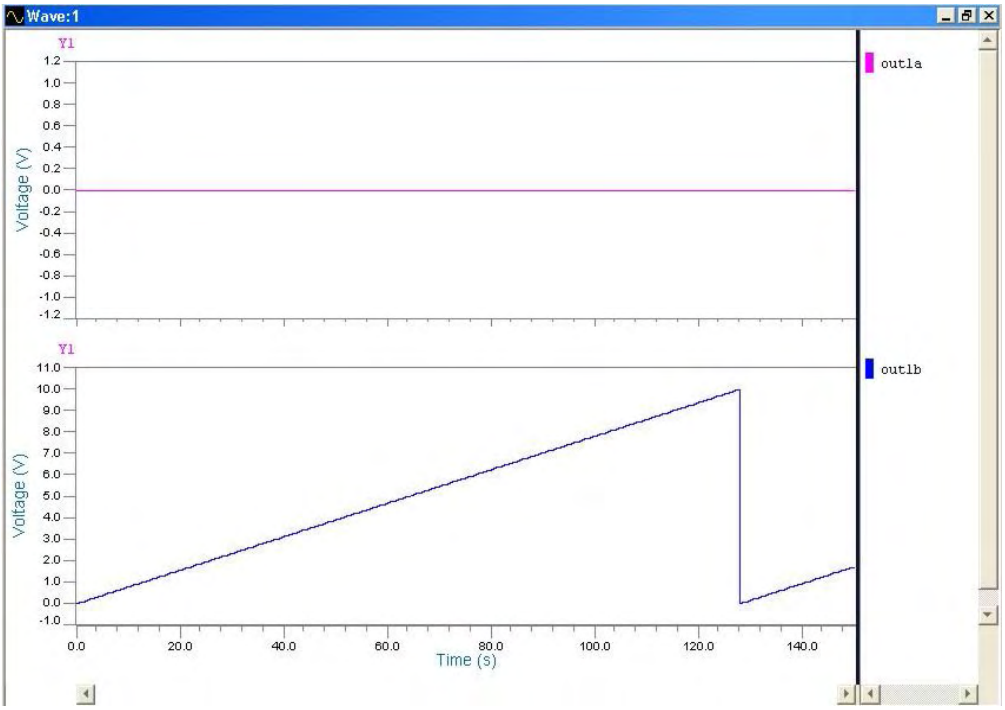

Figura 5.38. Simulação do DAC B no AD7528 no SystemVision™.

De forma análoga, as formas de onda senoidal obtidas nas saídas analógicas do DAC A e do DAC B, são apresentadas nas Figuras 5.39 e 5.40, respectivamente. Para tal, também foi utilizado um código adicional em VHDL, empregado na leitura de arquivo contendo uma entrada que gere a forma de onda senoidal.

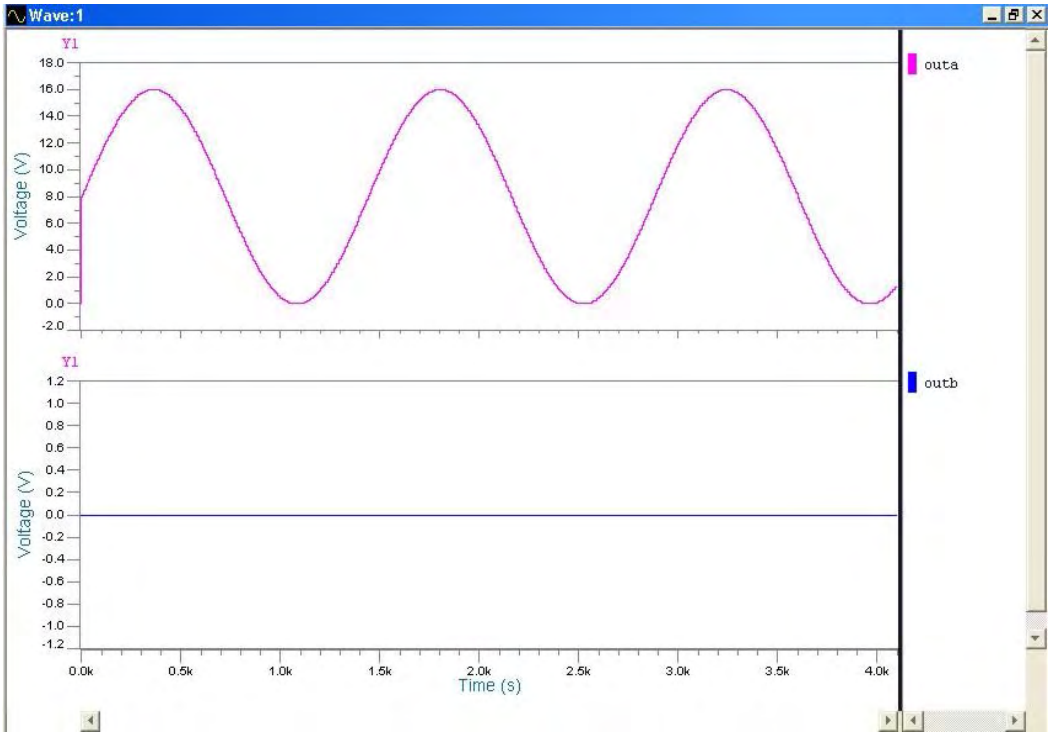

Figura 5.39. Forma de onda senoidal gerada pelo DAC A no AD7528.

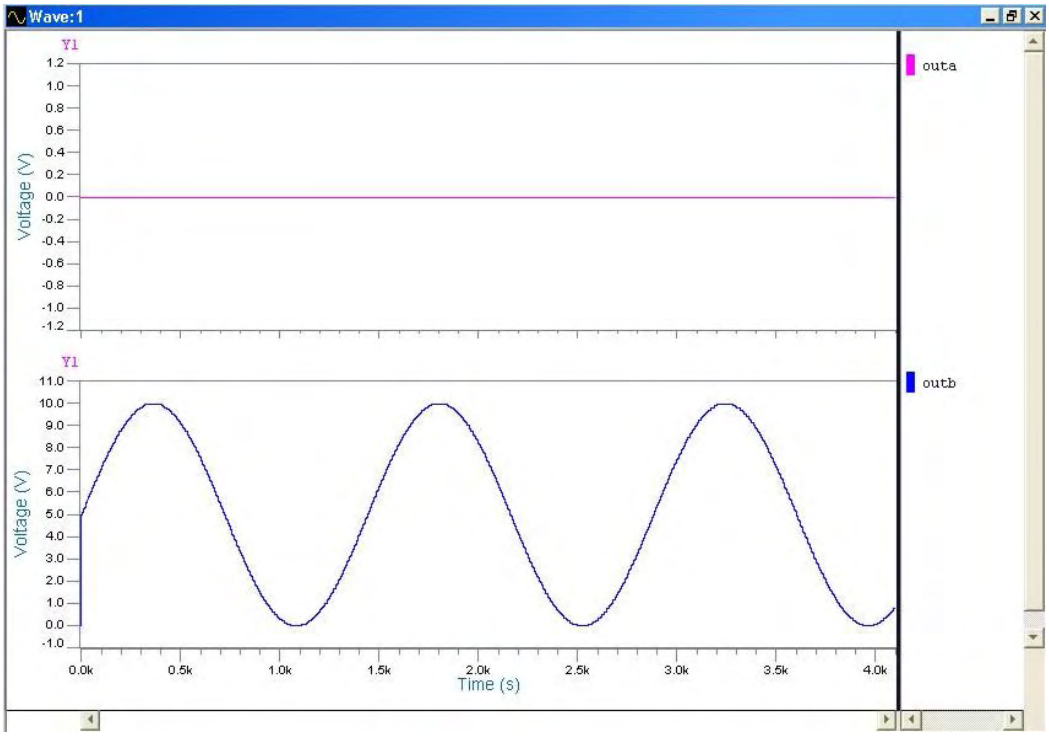

Figura 5.40. Forma de onda senoidal gerada pelo DAC B no AD7528.

5.2.4. Conversor AD5450

Dos quatro modelos de conversores estudados, o AD5450 é o modelo que apresenta maior grau de complexidade, pois, a leitura dos dados digitais é feita de forma serial. Foi necessária a confecção de vários subsistemas para a modelagem no Simulink®. O primeiro deles é denominado "*DAC Data Latch*", o qual contém toda a estrutura responsável por receber os dados seriais, convertê-los em paralelo (para passá-los a malha R/2R) e sincronizar o *clock* do conversor, conforme é apresentado na Figura 5.41.

Dentro do subsistema "*DAC Data Latch*", existem mais quatro subsistemas e três elementos, denominado "*Transport Delay*", encontrado no *toolboxes* do Simulink®, responsável por provocar um atraso no sinal, para que haja o sincronismo do *clock*. Em VHDL-AMS não existe essa necessidade já que as linguagens de descrição de *hardware* utilizam o conceito de paralelismo.

O primeiro subsistema é denominado "*Control Load*", responsável por carregar os dados digitais, realizar a contagem dos pulsos de *clock*, realizado pelo subsistema denominado "*Count Pulse*" e iniciar a leitura dos dados seriais realizado pelo subsistema denominado "*Set Trigger*". O modelo "*Control Load*" em ambiente Simulink® é apresentado na Figura 5.42.

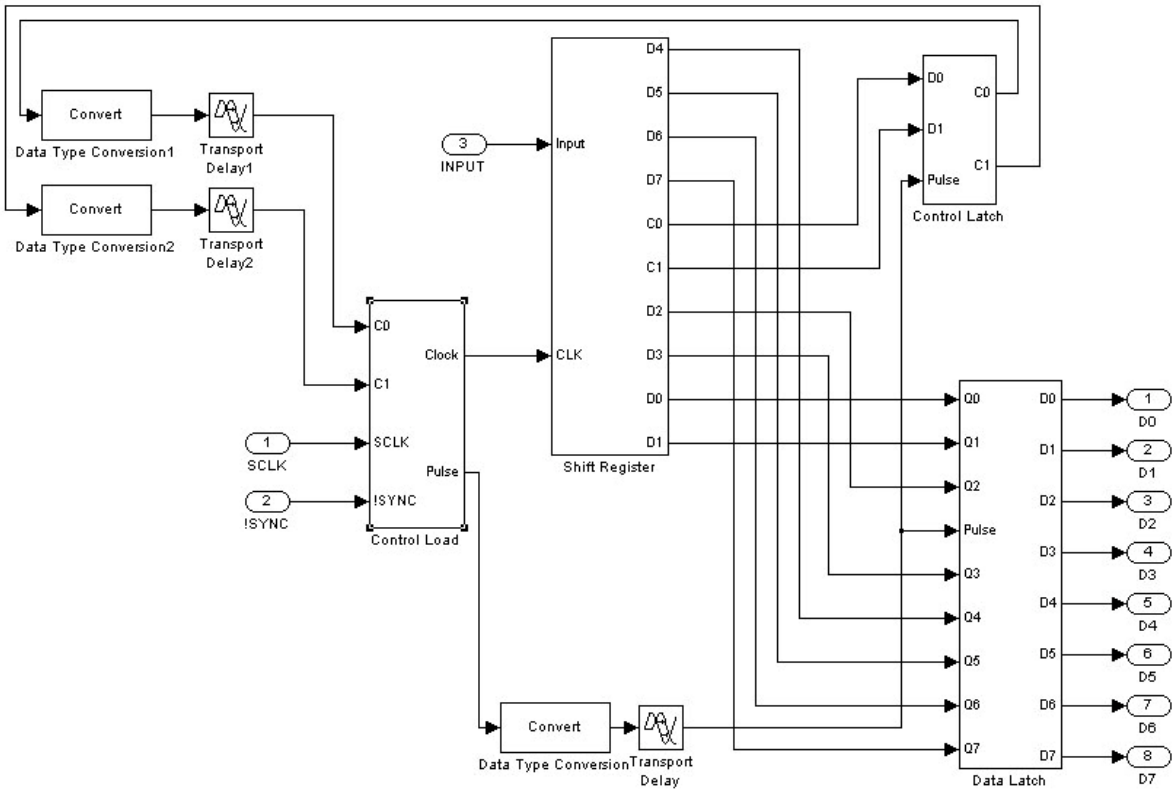

Figura 5.41. Subsistema DAC Data Latch em ambiente Simulink®.

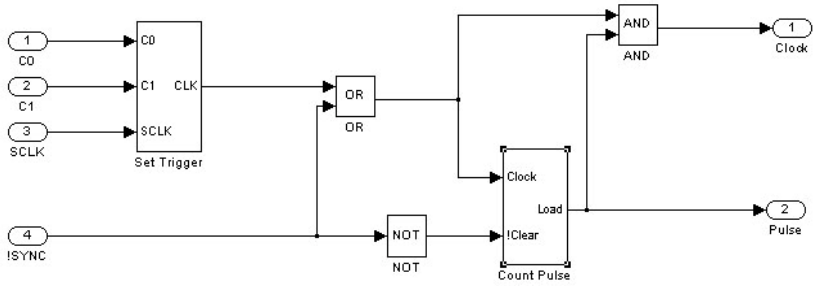

Figura 5.42. Subsistema Control Load em ambiente Simulink®.

Nas Figuras 5.43 e 5.44 são apresentados os subsistemas "*Set Trigger*" e "*Count Pulse*" respectivamente. É possível observar que esses subsistemas são circuitos digitais formados simplesmente por operadores lógicos e *flip-flops* do tipo D.

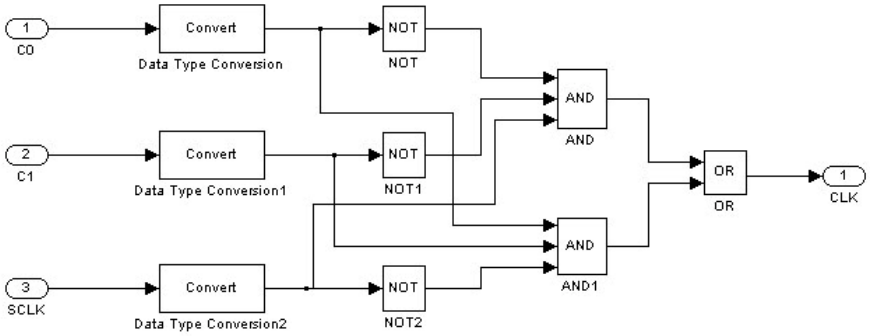

Figura 5.43. Subsistema Set Trigger modelado em ambiente Simulink®.

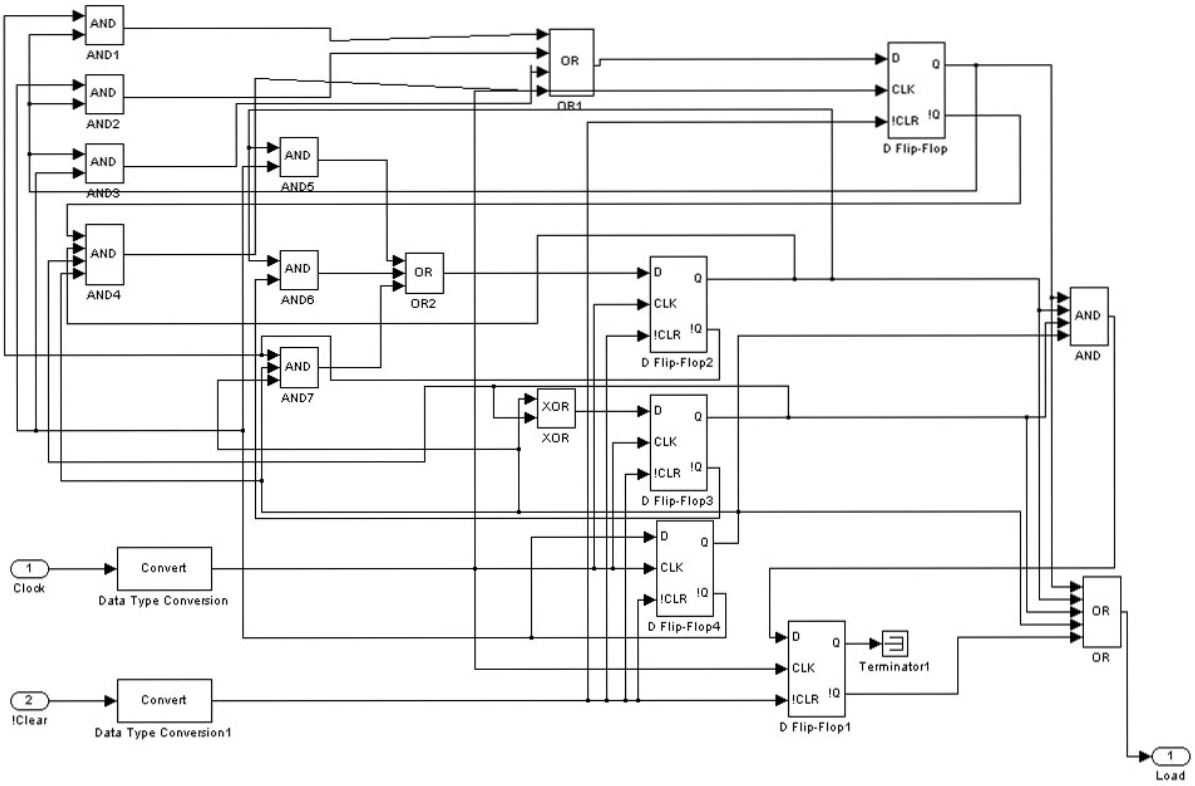

Figura 5.44. Subsistema Count Pulse modelado em ambiente Simulink®.

Outro importante subsistema contido no subsistema "*DAC Data Latch*" é o "*Shift Register*", que realiza a função de um registrador de deslocamento. Ele é responsável por receber bit por bit de forma serial, armazená-los e enviá-los, após o décimo sexto pulso de *clock* ao subsistema "*Data Latch*". O "*Data Latch*" é semelhante ao já apresentado na seção 5.2, com a diferença de que os bits são armazenados em *flip-flops* do tipo D, pois existe a necessidade do sincronismo com o *clock*. Na Figura 5.45, apresenta-se o subsistema "*Shift Register*", o qual também é composto por um conjunto de *flip-flops* do tipo D.

É importante ressaltar que somente os 10 primeiros bits de dados são considerados, os demais são descartados pelo circuito, já que, como estudo de caso, foi utilizado um conversor com resolução de 8 bits. Os dois primeiros bits, chamados C0 e C1, são bits empregados na configuração do sincronismo do *clock*.

Na Figura 5.46, apresenta-se o subsistema "*Control Latch*" contido dentro do subsistema "*DAC Data Latch*". Esse subsistema é responsável por capturar os dois bits mais significativos (C0 e C1), para realizar o sincronismo do "*Shift Register*" com o "*Data Latch*" e configurar o conversor.

O conjunto de todos esses subsistemas, a multiplicação da tensão de referência pela saída obtida na malha R/2R, a entrada do sinal de *clock* (SCLK), a entrada que habilitada a

leitura dos dados (SYNC) e entrada de dados (INPUT), formam o modelo no conversor AD5450 modelado no Simulink®, conforme apresenta-se na Figura 5.27.

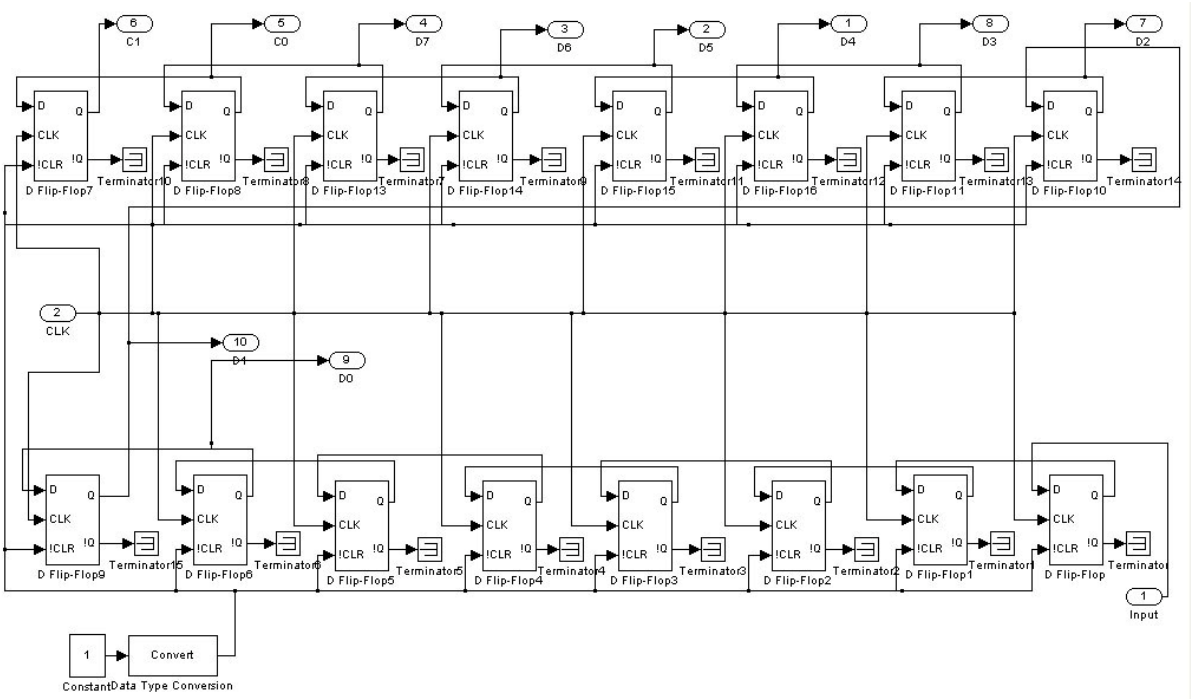

Figura 5.45. Subsistema Shift Register modelado em ambiente Simulink®.

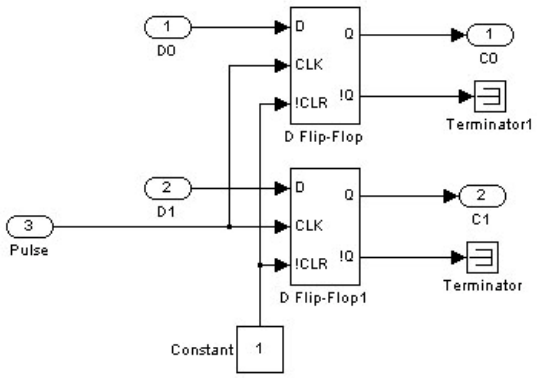

Figura 5.46. Subsistema Control Latch modelado em ambiente Simulink®.

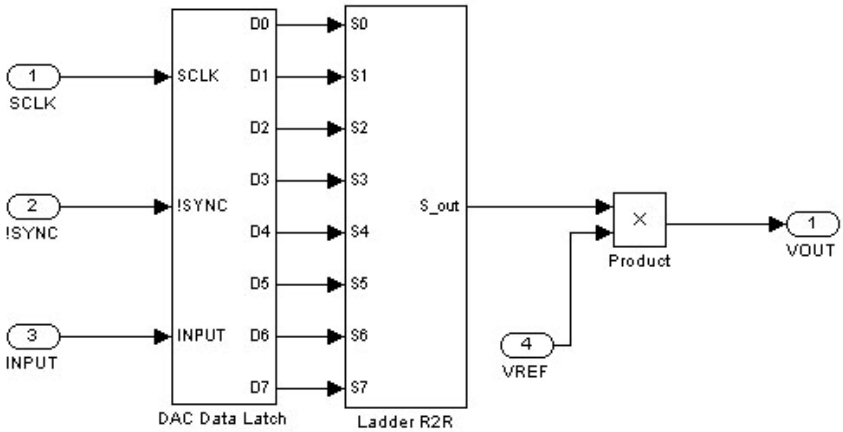

Figura 5.47. Conversor AD7528 em ambiente Simulink®.

```
-- Digital Systems and Signals Processing Laboratory
-- genhdl\..\AD5450.vhd
-- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
-- File created Wed Sep 09 08:14:38 2009

library IEEE;
use ieee.std_logic_1164.all;
use ieee.electrical_systems.all;
use ieee.mechanical_systems.all;
use ieee.fluidic_systems.all;
use ieee.thermal_systems.all;
use ieee.radiant_systems.all;
library EduLib;
use WORK.all;
library fundamentals_vda;
library spice2vhd;

entity AD5450 is
    port map (signal \!SYNC\: IN STD_LOGIC;
              signal SCLK: IN STD_LOGIC;
              signal SDIN: IN STD_LOGIC;
              terminal VOUT: ELETRICAL;
              terminal VREF: ELETRICAL);
end entity AD5450;

architecture arch_AD5450 of AD5450 is

    terminal \$1N47\: ELETRICAL;
    signal \$1N25\: STD_LOGIC;
    terminal \$1N37\: ELETRICAL;
    signal \$1N15\: STD_LOGIC;
    terminal \$1N39\: ELETRICAL;
    signal \$1N17\: STD_LOGIC;
    signal \$1N29\: STD_LOGIC;
    signal \$1N19\: STD_LOGIC;
    terminal \$1N41\: ELETRICAL;
    terminal \$1N31\: ELETRICAL;
    terminal \$1N43\: ELETRICAL;
    signal \$1N21\: STD_LOGIC;
    terminal \$1N33\: ELETRICAL;
    terminal \$1N45\: ELETRICAL;
    signal \$1N23\: STD_LOGIC;
    signal \$1N12\: STD_LOGIC;
    terminal \$1N35\: ELETRICAL;

begin

    \$1|1\: entity WORK.LADDER_R2RC (ARCH_LADDER_R2RC)
          port map (D0 => \$1N45\,
                    D1 => \$1N43\,
                    D2 => \$1N41\,
                    D3 => \$1N39\,
                    D4 => \$1N37\,
                    D5 => \$1N35\,
                    D6 => \$1N33\,
                    D7 => \$1N31\,
                    OUT_E => \$1N47\);

    D2A_BIT7: entity EDULIB.D2A_BIT (IDEAL)
          generic map (VHIGH => 1.0)
          port map (A => \$1N25\,
                    D => \$1N33\);

    D2A_BIT8: entity EDULIB.D2A_BIT (IDEAL)
          generic map (VHIGH => 1.0)
          port map (A => \$1N29\,
                    D => \$1N31\);

    \$1|2\: entity WORK.DATA_LATCH_DRIVES (ARCH_DATA_LATCH_DRIVES)
          port map (\!SYNC\ => \!SYNC\,
                    D0 => \$1N12\,
                    D1 => \$1N15\,
                    D2 => \$1N17\,
                    D3 => \$1N19\,
                    D4 => \$1N21\,
                    D5 => \$1N23\,
                    D6 => \$1N25\,
                    D7 => \$1N29\,
                    INPUT => SDIN,
                    SCLK => SCLK);

    E_MULT1: entity EDULIB.E_MULT
          port map (IN1 => \$1N47\,
                    IN2 => VREF,
                    OUTPUT => VOUT);

end architecture arch_AD5450;
```

Figura 5.48. Código VHDL-AMS gerado pelo MS^{2}SV para o conversor AD5450.

Na Figura 5.48, apresenta-se o código VHDL-AMS principal, após a tradução realizada pela ferramenta MS^{2}SV. É importante ressaltar que foram também gerados vários códigos VHDL-AMS, um para cada subsistema criado no modelo, mas todos devem estar contidos no *testbench* ativo do projeto no SystemVision™, porém, eles devem ser ativados por meio do ambiente SystemVision™ e não pela ferramenta MS^{2}SV. O código VHDL-AMS completo, para o caso do conversor AD5450, é apresentado no Apêndice D.

A simulação do conversor AD5450 não foi realizada por nenhum estímulo de entrada utilizado nos testes, diferente dos conversores paralelos. Com isso, em trabalhos futuros, é necessária uma revisão sobre os detalhes de funcionamento e implementação desse modelo de conversor serial.

Na avaliação da ferramenta MS^{2}SV, assim como na avaliação da ferramenta SF^{2}HDL, foi medido consumo computacional da ferramenta para os casos analisados neste trabalho. É importante ressaltar que foi utilizado o mesmo computador na avaliação das duas ferramentas: um Athlon 64 X2, com 2 GB de RAM e sistema operacional Windows XP *Service Pack* 3. O resultado foi proporcional à complexidade de cada caso, logo a tradução do modelo do conversor DAC08 obteve o menor consumo de tempo, 165 milissegundos, e menor consumo de memória RAM, 7,47 MB, e a tradução do conversor AD5450 obteve o maior consumo computacional, levando 281 milissegundos, consumindo 7,59 MB de memória.

Na Tabela 5.9, apresenta-se o consumo computacional apresentado em todos os casos avaliados pelo MS^{2}SV, destacando-se que esses resultados podem variar de acordo com o microcomputador utilizado no desenvolvimento do projeto.

Tabela 5.9. Comparação do consumo computacional dos casos.

Caso	Tempo	RAM
DAC08	165 ms	7, 47 MB
AD7524	172 ms	7,53 MB
AD7528	234 ms	7,57 MB
AD5450	281 ms	7,59 MB

Capítulo 6. Conclusão

Neste trabalho, foram apresentadas duas ferramentas que operam a síntese e extração entre diferentes níveis de abstração de um mesmo projeto. Visou-se fornecer ferramentas que auxiliem o desenvolvimento de projetos de circuitos eletrônicos em alto nível de abstração, já que, atualmente, não existe um padrão para modelagem, análise e simulação de projetos em alto nível.

A primeira ferramenta, denominada SF^{2}HDL, é capaz de realizar a extração de linguagens algorítmicas, como a VHDL e a Verilog HDL, a partir de um diagrama de transição de estados, modelado em ambiente Stateflow®. A ferramenta SF^{2}HDL é capaz ainda de extrair uma tabela de transição de estados padronizada, que foi utilizada como entrada para o programa TABELA, o qual foi responsável por minimizar o projeto através do método de Quine-McCluskey. Em seguida, foi utilizado o programa TAB2VHDL, que foi responsável por gerar um descrição em VHDL funcional, a partir da minimização obtida pelo TABELA.

O SF^{2}HDL foi avaliado através da modelagem de códigos de linha de sistemas de telecomunicações: AMI, HDB3, MLT-3, 2B1Q e HDB1. A ferramenta apresentou-se bastante flexível a um custo computacional satisfatório, tanto para as máquinas descritas pelo modelo de Mealy como para as descritas pelo modelo de Moore. Os casos foram sintetizados através do ambiente comercial de síntese Quartus II.

No ambiente Quartus II, foi possível verificar que os resultados de síntese dos códigos VHDL funcionais são extremante melhores em relação aos códigos VHDL comportamental e Verilog HDL comportamental, já que os códigos VHDL funcionais foram previamente minimizados pelo programa TABELA. Isso demonstra a necessidade de utilização de algoritmos de minimização antes da implementação física de um projeto. É importante ressaltar que a síntese de códigos em níveis mais altos de abstração são inferiores, se comparados com códigos em níveis mais baixos de abstração, o que já faz parte do escopo das linguagens de descrição de *hardware*. Para a geração do código VHDL funcional, é necessária a minimização do circuito, o que também faz parte do escopo das linguagens de descrição de *hardware*.

A segunda ferramenta é um aprimoramento de uma ferramenta já existente denominada MS^{2}SV, que é capaz de traduzir projetos modelados em nível de sistema no ambiente

Simulink®, para uma linguagem algorítmica, utilizando também linguagens de descrição de *hardware*, no caso a linguagem VHDL-AMS no domínio estrutural. Neste trabalho, foi utilizado o ambiente SystemVision™ da Mentor Graphics, o qual permite a simulação, análise e integração de sistemas mistos.

Essa nova versão da ferramenta proporciona uma maior flexibilidade por meio da adição de elementos de bibliotecas criadas pelo projetista, ou mesmo a adição de elementos do *toolboxes* do Simulink® que, inicialmente, o MS^{2}SV não é capaz de reconhecer e traduzi-los de forma automática para uma HDL correspondente. Na versão anterior, o projetista ficava limitado a utilizar somente os elementos existentes na biblioteca LIB_MS2SV. A LIB_MS2SV é uma biblioteca do Simulink®, a qual possui os elementos básicos utilizados nos projetos dos conversores de dados.

A nova versão da ferramenta MS^{2}SV possui uma interface gráfica similar à maioria dos programas desenvolvidos para a plataforma Windows, facilitando a utilização do projetista, o que não existia na primeira versão da ferramenta.

A avaliação da nova versão foi realizada com o estudo de caso de quatro modelos de conversores de dados, o DAC08, AD7524, AD7528 e AD5450. Nesses casos, a ferramenta foi eficiente na síntese dos modelos a um custo computacional satisfatório, comparado a primeira versão do MS^{2}SV, sendo possível melhorar o desempenho da ferramenta. Porém, não foi realizada a simulação do conversor serial AD5450, mesmo utilizando estímulos de entrada diferentes. Com isso, é necessário um estudo mais aprofundado e detalhado sobre o funcionamento e implementação desse modelo de conversor.

Ambas as ferramentas são bastante flexíveis, já que elas inibem a necessidade de utilização de ambientes caros e proprietários e, também, facilita o trabalho por meio da possibilidade de criação de bibliotecas externas e, assim, reutilizar determinados elementos de projeto.

É conveniente que, em trabalhos futuros, as duas ferramentas desenvolvidas sejam integradas em um ambiente único de síntese de circuitos. Existe também a necessidade de adicionar mais recursos às ferramentas, como por exemplo, algoritmos de redução de estados, algoritmos de alocação de estados, e novos algoritmos de minimização.

É importante que, nos próximos trabalhos, seja fechado o ciclo de projeto com a implementação física do circuito em FPGA, por sua grande flexibilidade em diversas aplicações. Salienta-se que as avaliações das duas ferramentas foram feitas somente por meio de simulação, disponíveis nas ferramentas utilizadas no desenvolvimento deste trabalho.

Referências

ANALOG DEVICES INC. (Org.). **8/10/12/14-bit high bandwidth multiplying DACs with serial interface.** United States of America: [s.n., 2009?]. Disponível em: <http:// www.datasheetcatalog.org/datasheets/134/148042_DS.pdf>. Acesso em: 6 jan. 2009.

ANALOG DEVICES INC. (Org.). **8-bit, high-speed, multiplying D/A converter:** universal digital logic interface. United States of America: [s.n., 2009?]. Disponível em: <http://www.datasheetcatalog.org/datasheet/analogdevices/80487346DAC08_b.pdf>. Acesso em: 6 jan. 2009.

ANALOG DEVICES INC. (Org.). **CMOS 8-bit buffered multiplying DAC.** United States of America: [s.n., 2009?] Disponível em: <http://www.datasheetcatalog.org/datasheet/ analogdevices/888358036ad7524.pdf>. Acesso em: 6 jan. 2009.

ANALOG DEVICES INC. (Org.). **CMOS dual 8-bit buffered multiplying DAC.** United States of America: [s.n., 2009?]. Disponível em: <http://www.datasheetcatalog.org /datasheet/analogdevices/718586868AD7528_b.pdf>. Acesso em: 2 maio 2009.

CAMERA, K. **SF2VHD:** a Stateflow to VHDL translator. 2001. 165f. Dissertação (Mestrado em Electrical Engineering and Computer Science) – University of California, Berkeley, 2001.

CHRISTEN, E.; BAKALAR, K. VHDL-AMS - a hardware description language for analog and mixed-signal applications. In: IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, 10., 1999, Rome. **Anais…** Rome: [s.n.], 1999. p. 1263-1272.

D'AMORE, R. **VHDL:** descrição e síntese de circuitos digitais. Rio de Janeiro: LTC, 2005. 259 p.

GAJSKI, D.D.; KUHN, R.H.. New VLSI tools. **Guest Editor's Introduction**, USA, v. 16, n. 12, p.11-14, 1983.

GANNOD, G.C.; GUPTA, S. an automated tool for analyzing Petri nets using Spin. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 16., 2001, Washington. **Anais...** Washington; [s.n.], 2001. p. 404 - 407.

GROUT I.A.; Keane K. A Matlab to VHDL conversion toolbox for digital control. In: IFAC SYMPOSIUM ON COMPUTER AIDED CONTROL SYSTEMS DESIGN (CACSD 2000), 2000, Salford. **Anais...** Salford: [s.n.], 2000. p. 1 -5.

GUIHAL, D. et al. VHDL-AMS model creation. In: INTERNATIONAL CONFERENCE MIXED DESIGN OF INTEGRATED CIRCUITS AND SYSTEM, 13., 2006, Poland. **Anais...** Poland: [s.n.], 2006. p. 549 - 554.

HORTA, N. Analogue and mixed-signal systems topologies exploration using symbolic methods. **Analog Integrated Circuits And Signal Processing,** Netherlands, p. 161-176. 2002.

KARRIS, S.T. **Introduction to stateflow with applications.** United States Of America: Orchard Publications, 2007. 366 p.

KEOGH, D. B.. The state diagram of HDB3. In: IEEE TRANSACTIONS ON COMMUNICATIONS, 11., 1984, Turin. **Anais…** Turin; [s.n.], p. 1222 - 1224.

KRASNIEWSKI, A. Concurrent error detection for finite state machines implemented with embedded memory blocks of SRAM-based FPGAs. **Microprocessors and Microsystems**, Poland, v. 32, p. 303 – 313, 2008.

KRISHNASWAMY, V.; GUPTA, R.; BANERJEE, P. A procedure for software synthesis from VHDL models. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2., 1997, Japan. **Anais...** Japan: [s.n.], 1997. p. 593 - 598.

MARKOVIĆ, D.; RICHARDS, B.; BRODERSEN, R.W. Technology driven DSP architecture optimization within a high-level block diagram based design flow. In: CONFERENCE ON SIGNALS, SYSTEMS AND COMPUTERS, 2006, Pacific Grove. **Anais…** Pacific Grove: [s.n.], 2006. p. 89 - 93.

MIROTZNIK, Mark S.. Translating Matlab programs into C code. **IEEE Spectrum**, New York, v. 33, p.63-64, 1996.

MUKHERJEE, T.; FEDDER, G.K. Hierarchical mixed-domain circuit simulation, synthesis and extraction methodology for MEMS. **Jornal of VLSI Singal Processing**, Netherlands, v. 21, p. 233 - 249, 1999.

NEHME, C.; LUNDQVIST, K. A tool for translating VHDL to finite state machines. In: DIGITAL AVIONICS SYSTEMS CONFERENCE, 22., 2003, USA. **Anais...** USA: [s.n.], 2003. p. 1 - 7.

OHIRA, M.I. **Estudo comparativo entre linguagens de descrição de hardware, VHDL e Verilog HDL.** 2003. 57 f. Trabalho de Formatura (Graduação em Engenharia Elétrica) - Curso de Engenharia Elétrica, Universidade Estadual Paulista, Ilha Solteira, 2003.

RIESGO, T.; TORROJA, Y.; LA TORRE, E. Design methodologies based on hardware description languages. In: IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, 1999, Auburn. **Anais...** Auburn: [s.n.], 1999. v. 46, p. 3 - 12.

SANTOS, C.E.A.S. **Algoritmo genético empregado na alocação de estados em máquinas markovianas.** 2005. 89 f. Dissertação (Mestrado em Engenharia Elétrica) – Faculdade de Engenharia, Universidade Estadual Paulista, Ilha Solteira, 2005.

SBARCEA, B.; NICULA, D. **Automatic conversion of Matlab/Simulink models to HDL models.** [S.l.: s.n., 2004?]. Disponível em: <http://www.fcd.co.il/doc/optim2004.pdf>. Acesso em: 19 ago. 2004.

SHANBLATT, M.A.; FOULDS, B. A Simulink-to-FPGA implementation tool for enhanced design flow. In: IEEE INTERNATIONAL CONFERENCE ON MICROELECTRONICS SYSTEMS EDUCATION, 2005, Washington. **Anais...** Washington: [s.n.], 2005. p. 1 - 2.

SILVA, A.C.R. **Contribuição à minimização e simulação de circuitos lógicos**. 1989. 138 f. Dissertação (Mestrado em Engenharia Elétrica) - Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 1989.

SILVA, A.C.R. **Data converter design and synthesis using hardware description languages.** 2007. 173 f. Tese (Pós-doutorado em Electrical And Computer Engineering) - Department of Electrical And Computer Engineering, University Limerik, Limerik, 2007.

SOUDRIS, D. J. et al. A CAD tool for architecture level exploration and automatic generation of rns converters. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 7., 2001, Australia. **Anais...** Australia: [s.n.], 2001. p. 730 - 733.

STONE, A.; MANOLAKOS, E.S. DG2VHDL: a tool to facilitate the high level synthesis of parallel processing array architectures. **The Journal Of VLSI Signal Processing,** Netherlands, v..., n…., p. 99-120. 2000.

TANCREDO, L.O. **TAB2VHDL:** um ambiente de síntese lógica para máquinas de estados finitos**.** 2002. 130 f. Dissertação (Mestrado em Engenharia Elétrica) – Faculdade de Engenharia, Universidade Estadual Paulista, Ilha Solteira, 2002.

TOCCI, R. J.; WIDMER, N.S.; MOSS, G. L.. Interface com o mundo analógico. In: TOCCI, R.J.; WIDMER, N.S.; MOSS, G.L.. **Sistemas digitais:** princípios e aplicações. 10. ed. São Paulo: Pearson, 2008. Cap. 11, p. 245-295.

WANG, Tsu-Hua; EDSALL, T. Practical FSM analysis for Verilog. In: VERILOG HDL CONFERENCE AND VHDL INTERNATIONAL USERS FORUM, 1998, San Jose. **Anais...** San Jose: [s.n.], 1998. p. 52-58.

ZORZI, M; FRANZÈ, F; SPECIALE, N.; MASETTI, G. A tool for integration of new VHDL-AMS models in Spice. In: PROCEEDINGS OF THE 2004 INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 4., 2004, Vancouver. **Anais…** Vancouver, [s.n.], 2004. p. IV637-640.

ZORZI M.; FRANZÈ F.; SPECIALE N. Construction of VHDL-AMS simulator in Matlab™. In: PROCEEDINGS OF THE 2003 INTERNATIONAL WORKSHOP ON BEHAVIORAL MODELING, 2003, San Jose. **Anais…** San Jose: [s.n.], 2003. p.113-117.

Apêndice A

A seguir é listado o código VHDL-AMS completo gerado pela ferramenta MS^{2}SV para o caso do conversor DAC08.

- Ladder_R2R

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\Ladder_R2R.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 16:00:12 2009
5.
6.    library IEEE;
7.    use ieee.std_logic_1164.all;
8.    use ieee.electrical_systems.all;
9.    use ieee.mechanical_systems.all;
10.   use ieee.fluidic_systems.all;
11.   use ieee.thermal_systems.all;
12.   use ieee.radiant_systems.all;
13.   library EduLib;
14.   use WORK.all;
15.   library fundamentals_vda;
16.   library spice2vhd;
17.
18.   entity Ladder_R2R is
19.     port (
20.         signal D1_D: in std_logic;
21.         signal D2_D: in std_logic;
22.         signal D3_D: in std_logic;
23.         signal D4_D: in std_logic;
24.         signal D5_D: in std_logic;
25.         signal D6_D: in std_logic;
26.         signal D7_D: in std_logic;
27.         signal D8_D: in std_logic;
28.         terminal A1: electrical
29.     );
30.   end entity Ladder_R2R;
31.
32.   architecture arch_Ladder_R2R of Ladder_R2R is
33.         terminal \1$N0\: electrical;
34.         terminal \1$N1\: electrical;
35.         terminal \1$N2\: electrical;
36.         terminal \1$N3\: electrical;
37.         terminal \1$N4\: electrical;
38.         terminal \1$N5\: electrical;
39.         terminal \1$N6\: electrical;
40.         terminal \1$N7\: electrical;
41.         terminal \1$N8\: electrical;
42.         terminal \1$N9\: electrical;
43.         terminal \1$N10\: electrical;
44.         terminal \1$N11\: electrical;
45.         terminal \1$N12\: electrical;
46.         terminal \1$N13\: electrical;
47.         terminal \1$N14\: electrical;
48.         terminal \1$N15\: electrical;
```

```
49.         terminal \1$N16\: electrical;
50.         terminal \1$N17\: electrical;
51.         terminal \1$N18\: electrical;
52.         terminal \1$N19\: electrical;
53.         terminal \1$N20\: electrical;
54.         terminal \1$N21\: electrical;
55.   begin
56.         DataType: entity EDULIB.D2A_BIT(IDEAL)
57.               generic map ( VHIGH => 1.0,
58.                             VLOW => 0.0 )
59.               port map (
60.                     D => D1_D,
61.                     A => \1$N0\
62.               );
63.         DataType1: entity EDULIB.D2A_BIT(IDEAL)
64.               generic map ( VHIGH => 1.0,
65.                             VLOW => 0.0 )
66.               port map (
67.                     D => D2_D,
68.                     A => \1$N1\
69.               );
70.         DataType2: entity EDULIB.D2A_BIT(IDEAL)
71.               generic map ( VHIGH => 1.0,
72.                             VLOW => 0.0 )
73.               port map (
74.                     D => D3_D,
75.                     A => \1$N2\
76.               );
77.         DataType3: entity EDULIB.D2A_BIT(IDEAL)
78.               generic map ( VHIGH => 1.0,
79.                             VLOW => 0.0 )
80.               port map (
81.                     D => D4_D,
82.                     A => \1$N3\
83.               );
84.         DataType4: entity EDULIB.D2A_BIT(IDEAL)
85.               generic map ( VHIGH => 1.0,
86.                             VLOW => 0.0 )
87.               port map (
88.                     D => D5_D,
89.                     A => \1$N4\
90.               );
91.         DataType5: entity EDULIB.D2A_BIT(IDEAL)
92.               generic map ( VHIGH => 1.0,
93.                             VLOW => 0.0 )
94.               port map (
95.                     D => D6_D,
96.                     A => \1$N5\
97.               );
98.         DataType6: entity EDULIB.D2A_BIT(IDEAL)
99.               generic map ( VHIGH => 1.0,
100.                            VLOW => 0.0 )
101.              port map (
102.                    D => D7_D,
103.                    A => \1$N6\
104.              );
105.        DataType7: entity EDULIB.D2A_BIT(IDEAL)
106.              generic map ( VHIGH => 1.0,
107.                            VLOW => 0.0 )
108.              port map (
109.                    D => D8_D,
```

```
110.                  A => \1$N7\
111.            );
112.      E_Gain: entity EDULIB.E_GAIN(BEHAVIORAL)
113.            generic map ( K => 0.5 )
114.            port map (
115.                  INPUT => \1$N0\,
116.                  OUTPUT => \1$N8\
117.            );
118.      E_Gain1: entity EDULIB.E_GAIN(BEHAVIORAL)
119.            generic map ( K => 0.25 )
120.            port map (
121.                  INPUT => \1$N1\,
122.                  OUTPUT => \1$N9\
123.            );
124.      E_Gain2: entity EDULIB.E_GAIN(BEHAVIORAL)
125.            generic map ( K => 0.125 )
126.            port map (
127.                  INPUT => \1$N2\,
128.                  OUTPUT => \1$N10\
129.            );
130.      E_Gain3: entity EDULIB.E_GAIN(BEHAVIORAL)
131.            generic map ( K => 0.0625 )
132.            port map (
133.                  INPUT => \1$N3\,
134.                  OUTPUT => \1$N11\
135.            );
136.      E_Gain4: entity EDULIB.E_GAIN(BEHAVIORAL)
137.            generic map ( K => 0.03125 )
138.            port map (
139.                  INPUT => \1$N4\,
140.                  OUTPUT => \1$N12\
141.            );
142.      E_Gain5: entity EDULIB.E_GAIN(BEHAVIORAL)
143.            generic map ( K => 0.015625 )
144.            port map (
145.                  INPUT => \1$N5\,
146.                  OUTPUT => \1$N13\
147.            );
148.      E_Gain6: entity EDULIB.E_GAIN(BEHAVIORAL)
149.            generic map ( K => 0.0078125 )
150.            port map (
151.                  INPUT => \1$N6\,
152.                  OUTPUT => \1$N14\
153.            );
154.      E_Gain7: entity EDULIB.E_GAIN(BEHAVIORAL)
155.            generic map ( K => 0.00390625 )
156.            port map (
157.                  INPUT => \1$N7\,
158.                  OUTPUT => \1$N15\
159.            );
160.      E_Sum: entity EDULIB.E_SUM
161.            port map (
162.                  IN1 => \1$N8\,
163.                  IN2 => \1$N9\,
164.                  OUTPUT => \1$N16\
165.            );
166.      E_Sum1: entity EDULIB.E_SUM
167.            port map (
168.                  IN1 => \1$N10\,
169.                  IN2 => \1$N11\,
170.                  OUTPUT => \1$N17\
```

```
171.            );
172.      E_Sum2: entity EDULIB.E_SUM
173.            port map (
174.                  IN1 => \1$N20\,
175.                  IN2 => \1$N21\,
176.                  OUTPUT => A1
177.            );
178.      E_Sum3: entity EDULIB.E_SUM
179.            port map (
180.                  IN1 => \1$N12\,
181.                  IN2 => \1$N13\,
182.                  OUTPUT => \1$N18\
183.            );
184.      E_Sum4: entity EDULIB.E_SUM
185.            port map (
186.                  IN1 => \1$N14\,
187.                  IN2 => \1$N15\,
188.                  OUTPUT => \1$N19\
189.            );
190.      E_Sum5: entity EDULIB.E_SUM
191.            port map (
192.                  IN1 => \1$N16\,
193.                  IN2 => \1$N17\,
194.                  OUTPUT => \1$N20\
195.            );
196.      E_Sum6: entity EDULIB.E_SUM
197.            port map (
198.                  IN1 => \1$N18\,
199.                  IN2 => \1$N19\,
200.                  OUTPUT => \1$N21\
201.            );
202. end architecture arch_Ladder_R2R;
```

♦ DAC08

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\DAC08.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 16:00:12 2009
5.
6.    library IEEE;
7.    use ieee.std_logic_1164.all;
8.    use ieee.electrical_systems.all;
9.    use ieee.mechanical_systems.all;
10.   use ieee.fluidic_systems.all;
11.   use ieee.thermal_systems.all;
12.   use ieee.radiant_systems.all;
13.   library EduLib;
14.   use WORK.all;
15.   library fundamentals_vda;
16.   library spice2vhd;
17.
18.   entity DAC08 is
19.   end entity DAC08;
20.
21.   architecture arch_DAC08 of DAC08 is
22.         terminal \1$N39\: electrical;
23.         terminal \1$N41\: electrical;
24.         signal DIn1_D: std_logic;
25.         signal DIn2_D: std_logic;
```

```
26.         signal DIn3_D: std_logic;
27.         signal DIn4_D: std_logic;
28.         signal DIn5_D: std_logic;
29.         signal DIn6_D: std_logic;
30.         signal DIn7_D: std_logic;
31.         signal DIn8_D: std_logic;
32.         terminal Vref: electrical;
33.         terminal Out1: electrical;
34.   begin
35.         Ladder_R2R: entity WORK.Ladder_R2R(arch_Ladder_R2R)
36.               port map (
37.                     D1_D => DIn1_D,
38.                     D2_D => DIn2_D,
39.                     D3_D => DIn3_D,
40.                     D4_D => DIn4_D,
41.                     D5_D => DIn5_D,
42.                     D6_D => DIn6_D,
43.                     D7_D => DIn7_D,
44.                     D8_D => DIn8_D,
45.                     A1 => \1$N39\
46.               );
47.         E_MULT: entity EDULIB.E_MULT
48.               port map (
49.                     IN1 => \1$N39\,
50.                     IN2 => Vref,
51.                     OUTPUT => Out1
52.               );
53.
54.         E_Pulse: entity EDULIB.CLOCK_FREQ(IDEAL)
55.             generic map ( FREQ => 1.0 )
56.             port map ( CLK_OUT => DIn8_D );
57.
58.         E_Pulse1: entity EDULIB.CLOCK_FREQ(IDEAL)
59.             generic map ( FREQ => 0.5 )
60.             port map ( CLK_OUT => DIn7_D );
61.
62.         E_Pulse2: entity EDULIB.CLOCK_FREQ(IDEAL)
63.             generic map ( FREQ => 0.25 )
64.             port map ( CLK_OUT => DIn6_D );
65.
66.         E_Pulse3: entity EDULIB.CLOCK_FREQ(IDEAL)
67.             generic map ( FREQ => 0.125 )
68.             port map ( CLK_OUT => DIn5_D );
69.
70.         E_Pulse4: entity EDULIB.CLOCK_FREQ(IDEAL)
71.             generic map ( FREQ => 0.0625 )
72.             port map ( CLK_OUT => DIn4_D );
73.
74.         E_Pulse5: entity EDULIB.CLOCK_FREQ(IDEAL)
75.             generic map ( FREQ => 0.03125 )
76.             port map ( CLK_OUT => DIn3_D );
77.
78.         E_Pulse6: entity EDULIB.CLOCK_FREQ(IDEAL)
79.             generic map ( FREQ => 0.015625 )
80.             port map ( CLK_OUT => DIn2_D );
81.
82.         E_Pulse7: entity EDULIB.CLOCK_FREQ(IDEAL)
83.             generic map ( FREQ => 0.0078125 )
84.             port map ( CLK_OUT => DIn1_D );
85.
86.         V_VREF: entity EDULIB.V_CONSTANT(IDEAL)
```

```
87.             generic map ( LEVEL => 10.0 )
88.             port map ( POS => Vref,
89.                        NEG => ELECTRICAL_REF);
90.
91.   end architecture arch_DAC08;
```

Apêndice B

A seguir é listado o código VHDL-AMS completo gerado pela ferramenta MS^{2}SV para o caso do conversor AD7524. A malha R2R utilizada nesse modelo é exatamente igual à descrita no Apêndice A.

- Data_Latch

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\DataLatch_D.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 20:15:55 2009
5.
6.    library IEEE;
7.    use ieee.std_logic_1164.all;
8.    use ieee.electrical_systems.all;
9.    use ieee.mechanical_systems.all;
10.   use ieee.fluidic_systems.all;
11.   use ieee.thermal_systems.all;
12.   use ieee.radiant_systems.all;
13.   library EduLib;
14.   use WORK.all;
15.   library fundamentals_vda;
16.   library spice2vhd;
17.
18.   entity DataLatch_D is
19.     port (
20.         signal Q1_D: in std_logic;
21.         signal Q2_D: in std_logic;
22.         signal Q3_D: in std_logic;
23.         signal Q4_D: in std_logic;
24.         signal Q5_D: in std_logic;
25.         signal Q6_D: in std_logic;
26.         signal Q7_D: in std_logic;
27.         signal Q8_D: in std_logic;
28.         signal Enable_D: in std_logic;
29.         signal R1_D: out std_logic;
30.         signal R2_D: out std_logic;
31.         signal R3_D: out std_logic;
32.         signal R4_D: out std_logic;
33.         signal R5_D: out std_logic;
34.         signal R6_D: out std_logic;
35.         signal R7_D: out std_logic;
36.         signal R8_D: out std_logic
37.     );
38.   end entity DataLatch_D;
39.
40.   architecture arch_DataLatch_D of DataLatch_D is
41.         signal \1$N2\: std_logic;
42.         signal \1$N5\: std_logic;
43.         signal \1$N8\: std_logic;
44.         signal \1$N11\: std_logic;
45.         signal \1$N23\: std_logic;
46.         signal \1$N16\: std_logic;
```

```
47.         signal \1$N19\: std_logic;
48.         signal \1$N22\: std_logic;
49.   begin
50.         DLatch: entity WORK.DLatch(arch_DLatch)
51.               port map (
52.                     D => Q1_D,
53.                     CLK => Enable_D,
54.                     Q => R1_D,
55.                     QN => \1$N2\
56.               );
57.         DLatch1: entity WORK.DLatch(arch_DLatch)
58.               port map (
59.                     D => Q2_D,
60.                     CLK => Enable_D,
61.                     Q => R2_D,
62.                     QN => \1$N5\
63.               );
64.         DLatch2: entity WORK.DLatch(arch_DLatch)
65.               port map (
66.                     D => Q3_D,
67.                     CLK => Enable_D,
68.                     Q => R3_D,
69.                     QN => \1$N8\
70.               );
71.         DLatch3: entity WORK.DLatch(arch_DLatch)
72.               port map (
73.                     D => Q4_D,
74.                     CLK => Enable_D,
75.                     Q => R4_D,
76.                     QN => \1$N11\
77.               );
78.         DLatch4: entity WORK.DLatch(arch_DLatch)
79.               port map (
80.                     D => Q5_D,
81.                     CLK => Enable_D,
82.                     Q => R5_D,
83.                     QN => \1$N23\
84.               );
85.         DLatch5: entity WORK.DLatch(arch_DLatch)
86.               port map (
87.                     D => Q6_D,
88.                     CLK => Enable_D,
89.                     Q => R6_D,
90.                     QN => \1$N16\
91.               );
92.         DLatch6: entity WORK.DLatch(arch_DLatch)
93.               port map (
94.                     D => Q7_D,
95.                     CLK => Enable_D,
96.                     Q => R7_D,
97.                     QN => \1$N19\
98.               );
99.         DLatch7: entity WORK.DLatch(arch_DLatch)
100.              port map (
101.                    D => Q8_D,
102.                    CLK => Enable_D,
103.                    Q => R8_D,
104.                    QN => \1$N22\
105.              );
106.  end architecture arch_DataLatch_D;
```

♦ DLatch

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\DLatch.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 20:15:55 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity DLatch is
19.      port (
20.         signal D: in std_logic;
21.         signal CLK: in std_logic;
22.         signal Q: out std_logic;
23.         signal QN: out std_logic
24.      );
25.   end entity DLatch;
26.
27.   architecture arch_DLatch of DLatch is
28.   begin
29.
30.     process(D,CLK)
31.     begin
32.       if clk = '1' then
33.         Q <= D;
34.         QN <= not D;
35.       end if;
36.     end process;
37.
38.   end architecture arch_DLatch;
```

♦ S_NOR

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\SNOR_D.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 20:15:55 2009
5.
6.    library IEEE;
7.    use ieee.std_logic_1164.all;
8.    use ieee.electrical_systems.all;
9.    use ieee.mechanical_systems.all;
10.   use ieee.fluidic_systems.all;
11.   use ieee.thermal_systems.all;
12.   use ieee.radiant_systems.all;
13.   library EduLib;
14.   use WORK.all;
15.   library fundamentals_vda;
16.   library spice2vhd;
17.
```

```
18.  entity SNOR_D is
19.    port (
20.        signal IN1_D: in std_logic;
21.        signal IN2_D: in std_logic;
22.        signal OUT_D: out std_logic
23.    );
24.  end entity SNOR_D;
25.
26.  architecture arch_SNOR_D of SNOR_D is
27.        signal \1$N63\: std_logic;
28.        signal \1$N64\: std_logic;
29.  begin
30.        G_AND1: entity EDULIB.AND2
31.              port map (
32.                    IN1 => \1$N63\,
33.                    IN2 => \1$N64\,
34.                    OUTPUT => OUT_D
35.              );
36.        G_NOT1: entity EDULIB.INVERTER
37.              port map (
38.                    INPUT => IN1_D,
39.                    OUTPUT => \1$N63\
40.              );
41.        G_NOT2: entity EDULIB.INVERTER
42.              port map (
43.                    INPUT => IN2_D,
44.                    OUTPUT => \1$N64\
45.              );
46.  end architecture arch_SNOR_D;
```

♦ AD7524

```
1.   -- Digital Systems and Signals Processing Laboratory
2.   -- genhdl\..\AD7524.vhd
3.   -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.   -- File created Tue Sep 08 20:15:55 2009
5.
6.   library IEEE;
7.   use ieee.std_logic_1164.all;
8.   use ieee.electrical_systems.all;
9.   use ieee.mechanical_systems.all;
10.  use ieee.fluidic_systems.all;
11.  use ieee.thermal_systems.all;
12.  use ieee.radiant_systems.all;
13.  library EduLib;
14.  use WORK.all;
15.  library fundamentals_vda;
16.  library spice2vhd;
17.
18.  entity AD7524 is
19.  end entity AD7524;
20.
21.  architecture arch_AD7524 of AD7524 is
22.      signal IN1_D: std_logic;
23.        signal IN2_D: std_logic;
24.        signal IN3_D: std_logic;
25.        signal IN4_D: std_logic;
26.        signal IN5_D: std_logic;
27.        signal IN6_D: std_logic;
28.        signal IN7_D: std_logic;
29.        signal IN8_D: std_logic;
```

```
30.         signal CS_D: std_logic;
31.         signal WR_D: std_logic;
32.         terminal VREF: electrical;
33.         terminal OUT1: electrical;
34.         terminal OUT2: electrical;
35.         signal \1$N68\: std_logic;
36.         signal \1$N69\: std_logic;
37.         signal \1$N70\: std_logic;
38.         signal \1$N71\: std_logic;
39.         signal \1$N72\: std_logic;
40.         signal \1$N73\: std_logic;
41.         signal \1$N74\: std_logic;
42.         signal \1$N75\: std_logic;
43.         terminal \1$N78\: electrical;
44.         terminal WR: electrical;
45.         terminal CS: electrical;
46.         signal \1$N92\: std_logic;
47.   begin
48.         DataLatch_D: entity WORK.DataLatch_D(arch_DataLatch_D)
49.               port map (
50.                     Q1_D => IN1_D,
51.                     Q2_D => IN2_D,
52.                     Q3_D => IN3_D,
53.                     Q4_D => IN4_D,
54.                     Q5_D => IN5_D,
55.                     Q6_D => IN6_D,
56.                     Q7_D => IN7_D,
57.                     Q8_D => IN8_D,
58.                     Enable_D => \1$N92\,
59.                     R1_D => \1$N68\,
60.                     R2_D => \1$N69\,
61.                     R3_D => \1$N70\,
62.                     R4_D => \1$N71\,
63.                     R5_D => \1$N72\,
64.                     R6_D => \1$N73\,
65.                     R7_D => \1$N74\,
66.                     R8_D => \1$N75\
67.               );
68.         LadderR2R: entity WORK.LadderR2R(arch_LadderR2R)
69.               port map (
70.                     D1_D => \1$N68\,
71.                     D2_D => \1$N69\,
72.                     D3_D => \1$N70\,
73.                     D4_D => \1$N71\,
74.                     D5_D => \1$N72\,
75.                     D6_D => \1$N73\,
76.                     D7_D => \1$N74\,
77.                     D8_D => \1$N75\,
78.                     Out1 => \1$N78\
79.               );
80.         E_MULT: entity EDULIB.E_MULT
81.               port map (
82.                     IN1 => \1$N78\,
83.                     IN2 => VREF,
84.                     OUTPUT => OUT1
85.               );
86.         SNOR_D: entity WORK.SNOR_D(arch_SNOR_D)
87.               port map (
88.                     IN1_D => CS_D,
89.                     IN2_D => WR_D,
90.                     OUT_D => \1$N92\
```

```
91.             );
92.         E_SUM: entity EDULIB.E_SUM
93.             port map (
94.                   IN1 => \1$N78\,
95.                   IN2 => VREF,
96.                   OUTPUT => OUT2
97.             );
98.         E_Pulse: entity EDULIB.CLOCK_FREQ(IDEAL)
99.             generic map ( FREQ => 1.0 )
100.            port map ( CLK_OUT => In8_D );
101.
102.        E_Pulse1: entity EDULIB.CLOCK_FREQ(IDEAL)
103.            generic map ( FREQ => 0.5 )
104.            port map ( CLK_OUT => In7_D );
105.
106.        E_Pulse2: entity EDULIB.CLOCK_FREQ(IDEAL)
107.            generic map ( FREQ => 0.25 )
108.            port map ( CLK_OUT => In6_D );
109.
110.        E_Pulse3: entity EDULIB.CLOCK_FREQ(IDEAL)
111.            generic map ( FREQ => 0.125 )
112.            port map ( CLK_OUT => In5_D );
113.
114.        E_Pulse4: entity EDULIB.CLOCK_FREQ(IDEAL)
115.            generic map ( FREQ => 0.0625 )
116.            port map ( CLK_OUT => In4_D );
117.
118.        E_Pulse5: entity EDULIB.CLOCK_FREQ(IDEAL)
119.            generic map ( FREQ => 0.03125 )
120.            port map ( CLK_OUT => In3_D );
121.
122.        E_Pulse6: entity EDULIB.CLOCK_FREQ(IDEAL)
123.            generic map ( FREQ => 0.015625 )
124.            port map ( CLK_OUT => In2_D );
125.
126.        E_Pulse7: entity EDULIB.CLOCK_FREQ(IDEAL)
127.            generic map ( FREQ => 0.0078125 )
128.            port map ( CLK_OUT => In1_D );
129.
130.        V_CONT1: entity EDULIB.V_CONSTANT(IDEAL)
131.            generic map ( LEVEL => 0.0 )
132.            port map ( POS => CS,
133.                       NEG => ELECTRICAL_REF);
134.
135.        A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
136.             --generic map ( LEVEL => 1.0 )
137.             port map (A => CS,
138.                         D => CS_D);
139.
140.        V_CONT2: entity EDULIB.V_CONSTANT(IDEAL)
141.            generic map ( LEVEL => 0.0 )
142.            port map ( POS => WR,
143.                       NEG => ELECTRICAL_REF);
144.
145.        A2D_BIT2: entity EDULIB.A2D_BIT(IDEAL)
146.             --generic map ( LEVEL => 1.0 )
147.             port map (A => WR,
148.                         D => WR_D);
149.
150.        V_REF: entity EDULIB.V_CONSTANT(IDEAL)
151.            generic map ( LEVEL => 10.0 )
```

```
152.            port map ( POS => VREF,
153.                       NEG => ELECTRICAL_REF);
154.  end architecture arch_AD7524;
```

Apêndice C

A seguir é listado o código VHDL-AMS completo gerado pela ferramenta MS^{2}SV para o caso do conversor AD7528. Para esse modelo de conversor foi utilizada a mesma malha R2R descrita no Apêndice A. O subsistema *Data_Latch* e o blocos *LatchD* são exatamente os mesmo já descritos no Apêndice B.

- Control_Logic

```
1.    -- Digital Systems and Signals Processing Laboratory
2.    -- genhdl\..\ControlLogic_D.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Tue Sep 08 20:15:39 2009
5.
6.    library IEEE;
7.    use ieee.std_logic_1164.all;
8.    use ieee.electrical_systems.all;
9.    use ieee.mechanical_systems.all;
10.   use ieee.fluidic_systems.all;
11.   use ieee.thermal_systems.all;
12.   use ieee.radiant_systems.all;
13.   library EduLib;
14.   use WORK.all;
15.   library fundamentals_vda;
16.   library spice2vhd;
17.
18.   entity ControlLogic_D is
19.     port (
20.         signal DADB_D: in std_logic;
21.         signal WR_D: in std_logic;
22.         signal CS_D: in std_logic;
23.         signal EnA_D: out std_logic;
24.         signal EnB_D: out std_logic
25.     );
26.   end entity ControlLogic_D;
27.
28.   architecture arch_ControlLogic_D of ControlLogic_D is
29.         signal \1$N9\: std_logic;
30.         signal \1$N2\: std_logic;
31.         signal \1$N3\: std_logic;
32.         signal \1$N4\: std_logic;
33.   begin
34.         G_AND1: entity EDULIB.AND2
35.               port map (
36.                     IN1 => \1$N2\,
37.                     IN2 => \1$N3\,
38.                     OUTPUT => \1$N9\
39.               );
40.         G_AND2: entity EDULIB.AND2
41.               port map (
42.                     IN1 => \1$N4\,
43.                     IN2 => \1$N9\,
44.                     OUTPUT => EnA_D
```

```
45.             );
46.       G_AND3: entity EDULIB.AND2
47.             port map (
48.                   IN1 => DADB_D,
49.                   IN2 => \1$N9\,
50.                   OUTPUT => EnB_D
51.             );
52.       G_NOT1: entity EDULIB.INVERTER
53.             port map (
54.                   INPUT => WR_D,
55.                   OUTPUT => \1$N2\
56.             );
57.       G_NOT2: entity EDULIB.INVERTER
58.             port map (
59.                   INPUT => CS_D,
60.                   OUTPUT => \1$N3\
61.             );
62.       G_NOT3: entity EDULIB.INVERTER
63.             port map (
64.                   INPUT => DADB_D,
65.                   OUTPUT => \1$N4\
66.             );
67.  end architecture arch_ControlLogic_D;
```

♦ AD7528

```
1.   -- Digital Systems and Signals Processing Laboratory
2.   -- genhdl\..\AD7528.vhd
3.   -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.   -- File created Tue Sep 08 20:15:39 2009
5.
6.   library IEEE;
7.   use ieee.std_logic_1164.all;
8.   use ieee.electrical_systems.all;
9.   use ieee.mechanical_systems.all;
10.  use ieee.fluidic_systems.all;
11.  use ieee.thermal_systems.all;
12.  use ieee.radiant_systems.all;
13.  library EduLib;
14.  use WORK.all;
15.  library fundamentals_vda;
16.  library spice2vhd;
17.
18.  entity AD7528 is
19.  end entity AD7528;
20.
21.  architecture arch_AD7528 of AD7528 is
22.      signal IN1_D: std_logic;
23.        signal IN2_D: std_logic;
24.        signal IN3_D: std_logic;
25.        signal IN4_D: std_logic;
26.        signal IN5_D: std_logic;
27.        signal IN6_D: std_logic;
28.        signal IN7_D: std_logic;
29.        signal IN8_D: std_logic;
30.        signal DA_DB_D: std_logic;
31.        signal WR_D: std_logic;
32.        signal CS_D: std_logic;
33.        terminal CS: electrical;
34.        terminal WR: electrical;
35.        terminal DA_DB: electrical;
```

```
36.         terminal VrefA: electrical;
37.         terminal VrefB: electrical;
38.         terminal Out1A: electrical;
39.         terminal Out1B: electrical;
40.         signal \1$N204\: std_logic;
41.         signal \1$N205\: std_logic;
42.   begin
43.         ControlLogic_D: entity WORK.ControlLogic_D(arch_ControlLogic_D)
44.               port map (
45.                     DADB_D => DA_DB_D,
46.                     WR_D => WR_D,
47.                     CS_D => CS_D,
48.                     EnA_D => \1$N204\,
49.                     EnB_D => \1$N205\
50.               );
51.         DAC_A: entity WORK.DAC_A(arch_DAC_A)
52.               port map (
53.                     A1_D => IN1_D,
54.                     A2_D => IN2_D,
55.                     A3_D => IN3_D,
56.                     A4_D => IN4_D,
57.                     A5_D => IN5_D,
58.                     A6_D => IN6_D,
59.                     A7_D => IN7_D,
60.                     A8_D => IN8_D,
61.                     VrefA_I => VrefA,
62.                     EnA_D => \1$N204\,
63.                     OutA => Out1A
64.               );
65.         DAC_B: entity WORK.DAC_B(arch_DAC_B)
66.               port map (
67.                     B1_D => IN1_D,
68.                     B2_D => IN2_D,
69.                     B3_D => IN3_D,
70.                     B4_D => IN4_D,
71.                     B5_D => IN5_D,
72.                     B6_D => IN6_D,
73.                     B7_D => IN7_D,
74.                     B8_D => IN8_D,
75.                     VrefB_I => VrefB,
76.                     EnB_D => \1$N205\,
77.                     OutB => Out1B
78.               );
79.         E_Pulse: entity EDULIB.CLOCK_FREQ(IDEAL)
80.             generic map ( FREQ => 1.0 )
81.             port map ( CLK_OUT => In8_D );
82.
83.         E_Pulse1: entity EDULIB.CLOCK_FREQ(IDEAL)
84.             generic map ( FREQ => 0.5 )
85.             port map ( CLK_OUT => In7_D );
86.
87.         E_Pulse2: entity EDULIB.CLOCK_FREQ(IDEAL)
88.             generic map ( FREQ => 0.25 )
89.             port map ( CLK_OUT => In6_D );
90.
91.         E_Pulse3: entity EDULIB.CLOCK_FREQ(IDEAL)
92.             generic map ( FREQ => 0.125 )
93.             port map ( CLK_OUT => In5_D );
94.
95.         E_Pulse4: entity EDULIB.CLOCK_FREQ(IDEAL)
96.             generic map ( FREQ => 0.0625 )
```

```
97.             port map ( CLK_OUT => In4_D );
98.
99.         E_Pulse5: entity EDULIB.CLOCK_FREQ(IDEAL)
100.            generic map ( FREQ => 0.03125 )
101.            port map ( CLK_OUT => In3_D );
102.
103.        E_Pulse6: entity EDULIB.CLOCK_FREQ(IDEAL)
104.            generic map ( FREQ => 0.015625 )
105.            port map ( CLK_OUT => In2_D );
106.
107.        E_Pulse7: entity EDULIB.CLOCK_FREQ(IDEAL)
108.            generic map ( FREQ => 0.0078125 )
109.            port map ( CLK_OUT => In1_D );
110.
111.        E_Pulse8: entity EDULIB.CLOCK_FREQ(IDEAL)
112.            generic map ( FREQ => 1.0 )
113.            port map ( CLK_OUT => DA_DB_D );
114.
115.        V_CONST1: entity EDULIB.V_CONSTANT(IDEAL)
116.            generic map ( LEVEL => 0.0 )
117.            port map ( POS => CS,
118.                       NEG => ELECTRICAL_REF);
119.
120.        A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
121.              --generic map ( LEVEL => 1.0 )
122.              port map (A => CS,
123.                         D => CS_D);
124.
125.        V_CONST2: entity EDULIB.V_CONSTANT(IDEAL)
126.            generic map ( LEVEL => 0.0 )
127.            port map ( POS => WR,
128.                       NEG => ELECTRICAL_REF);
129.
130.        A2D_BIT2: entity EDULIB.A2D_BIT(IDEAL)
131.              --generic map ( LEVEL => 1.0 )
132.              port map (A => WR,
133.                         D => WR_D);
134.
135.        V_CONST3: entity EDULIB.V_CONSTANT(IDEAL)
136.            generic map ( LEVEL => 0.0 )
137.            port map ( POS => DA_DB,
138.                       NEG => ELECTRICAL_REF);
139.
140.        A2D_BIT3: entity EDULIB.A2D_BIT(IDEAL)
141.              --generic map ( THRES => 1.0 )
142.              port map (A => DA_DB,
143.                         D => DA_DB_D);
144.
145.        V_REFA: entity EDULIB.V_CONSTANT(IDEAL)
146.            generic map ( LEVEL => 16.0 )
147.            port map ( POS => VrefA,
148.                       NEG => ELECTRICAL_REF);
149.
150.        V_REFB: entity EDULIB.V_CONSTANT(IDEAL)
151.            generic map ( LEVEL => 10.0 )
152.            port map ( POS => VrefB,
153.                       NEG => ELECTRICAL_REF);
154.
155. end architecture arch_AD7528;
```

Apêndice D

A seguir é listado o código VHDL-AMS gerado pela ferramenta MS^{2}SV para o caso do conversor AD5450. Com exceção da malha R2R que é a mesma descrita no Apêndice A.

- Control_Latch

```
1.    -- Systems and Digital Signals Processing Laboratory
2.    -- genhdl\..\Control_Shift.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Wed Sep 09 08:14:38 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity CONTROL_SHIFT is
19.      port (signal PULSE: IN STD_LOGIC;
20.       signal C0: OUT STD_LOGIC;
21.       signal D0: IN STD_LOGIC;
22.       signal D1: IN STD_LOGIC;
23.       signal C1: OUT STD_LOGIC);
24.   end entity CONTROL_SHIFT;
25.
26.   architecture arch_CONTROL_SHIFT of CONTROL_SHIFT is
27.       signal Q1B: STD_LOGIC;
28.       signal Q0B: STD_LOGIC;
29.       signal RESET: STD_LOGIC;
30.       terminal \1$RESET\: ELECTRICAL;
31.   begin
32.
33.       DLATCH2: entity EDULIB.DLATCH
34.           port map (D => D1,
35.                CLK => PULSE,
36.                Q => C1,
37.                QN => Q1B,
38.                R => RESET);
39.
40.       DLATCH1: entity EDULIB.DLATCH
41.           port map (D => D0,
42.                CLK => PULSE,
43.                Q => C0,
44.                QN => Q0B,
45.                R => RESET);
46.
47.         V_CONST: entity EDULIB.V_CONSTANT(IDEAL)
48.             generic map ( LEVEL => 2.5 )
49.             port map ( POS => \1$RESET\,
```

```
50.                       NEG => ELECTRICAL_REF);
51.
52.        A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
53.              --generic map ( LEVEL => 1.0 )
54.              port map (A => \1$RESET\,
55.                        D => RESET);
56.
57.   end architecture arch_CONTROL_SHIFT;
```

♦ Shift16S8

```
1.    -- Systems and Digital Signals Processing Laboratory
2.    -- genhdl\..\SHIFT16S8.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Wed Sep 09 08:14:38 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity SHIFT16S8 is
19.   Port(
20.          signal C0 : OUT STD_LOGIC;
21.          signal C1 : OUT STD_LOGIC;
22.          signal INPUT : IN STD_LOGIC;
23.          signal CLOCK : IN STD_LOGIC;
24.          signal Q0 : OUT STD_LOGIC;
25.          signal Q1 : OUT STD_LOGIC;
26.          signal Q2 : OUT STD_LOGIC;
27.          signal Q3 : OUT STD_LOGIC;
28.          signal Q4 : OUT STD_LOGIC;
29.          signal Q5 : OUT STD_LOGIC;
30.          signal Q6 : OUT STD_LOGIC;
31.          signal Q7 : OUT STD_LOGIC);
32.   end entity SHIFT16S8;
33.
34.   architecture arch_SHIFT16S8 of SHIFT16S8 is
35.      signal Q15B: STD_LOGIC;
36.      signal Q14B: STD_LOGIC;
37.      signal Q13B: STD_LOGIC;
38.      signal Q12B: STD_LOGIC;
39.      signal Q11B: STD_LOGIC;
40.      signal Q10B: STD_LOGIC;
41.      signal Q9B: STD_LOGIC;
42.      signal Q8B: STD_LOGIC;
43.      signal Q7B: STD_LOGIC;
44.      signal Q6B: STD_LOGIC;
45.      signal Q5B: STD_LOGIC;
46.      signal Q4B: STD_LOGIC;
47.      signal Q3B: STD_LOGIC;
48.      signal Q2B: STD_LOGIC;
49.      signal Q1B: STD_LOGIC;
```

```
50.      signal Q0B: STD_LOGIC;
51.      signal \$1N26\: STD_LOGIC;
52.      signal \$1N28\: STD_LOGIC;
53.      signal \$1N18\: STD_LOGIC;
54.      signal \$1N20\: STD_LOGIC;
55.      signal \$1N22\: STD_LOGIC;
56.      signal \$1N24\: STD_LOGIC;
57.      signal \$BUF_Q0\: STD_LOGIC;
58.      signal \$BUF_Q1\: STD_LOGIC;
59.      signal \$BUF_Q2\: STD_LOGIC;
60.      signal \$BUF_Q3\: STD_LOGIC;
61.      signal \$BUF_Q4\: STD_LOGIC;
62.      signal \$BUF_Q5\: STD_LOGIC;
63.      signal \$BUF_Q6\: STD_LOGIC;
64.      signal \$BUF_Q7\: STD_LOGIC;
65.      signal \$BUF_C0\: STD_LOGIC;
66.      signal RESET : std_logic;
67.      terminal REF: electrical;
68.      --terminal ELECTRICAL_REF: electrical;
69.   begin
70.      Q0 <= \$BUF_Q0\;
71.      Q1 <= \$BUF_Q1\;
72.      Q2 <= \$BUF_Q2\;
73.      Q3 <= \$BUF_Q3\;
74.      Q4 <= \$BUF_Q4\;
75.      Q5 <= \$BUF_Q5\;
76.      Q6 <= \$BUF_Q6\;
77.      Q7 <= \$BUF_Q7\;
78.      C0 <= \$BUF_C0\;
79.
80.      DLATCH1 : entity EDULIB.DLATCH
81.         port map ( D => INPUT,
82.                    CLK => CLOCK,
83.                    Q => \$1N18\,
84.                    QN => Q0B,
85.                    R => RESET );
86.      DLATCH10 : entity EDULIB.DLATCH
87.         port map ( D => \$BUF_Q2\,
88.                    CLK => CLOCK,
89.                    Q => \$BUF_Q3\,
90.                    QN => Q9B,
91.                    R => RESET );
92.      DLATCH11 : entity EDULIB.DLATCH
93.         port map ( D => \$BUF_Q3\,
94.                    CLK => CLOCK,
95.                    Q => \$BUF_Q4\,
96.                    QN => Q10B,
97.                    R => RESET );
98.      DLATCH12 : entity EDULIB.DLATCH
99.         port map ( D => \$BUF_Q4\,
100.                   CLK => CLOCK,
101.                   Q => \$BUF_Q5\,
102.                   QN => Q11B,
103.                   R => RESET );
104.     DLATCH13 : entity EDULIB.DLATCH
105.        port map ( D => \$BUF_Q5\,
106.                   CLK => CLOCK,
107.                   Q => \$BUF_Q6\,
108.                   QN => Q12B,
109.                   R => RESET );
110.     DLATCH14 : entity EDULIB.DLATCH
```

```
111.       port map ( D => \$BUF_Q6\,
112.                  CLK => CLOCK,
113.                  Q => \$BUF_Q7\,
114.                  QN => Q13B,
115.                  R => RESET );
116.    DLATCH15 : entity EDULIB.DLATCH
117.       port map ( D => \$BUF_Q7\,
118.                  CLK => CLOCK,
119.                  Q => \$BUF_C0\,
120.                  QN => Q14B,
121.                  R => RESET );
122.    DLATCH16 : entity EDULIB.DLATCH
123.       port map ( D => \$BUF_C0\,
124.                  CLK => CLOCK,
125.                  Q => C1,
126.                  QN => Q15B,
127.                  R => RESET );
128.    DLATCH2 : entity EDULIB.DLATCH
129.       port map ( D => \$1N18\,
130.                  CLK => CLOCK,
131.                  Q => \$1N20\,
132.                  QN => Q1B,
133.                  R => RESET );
134.    DLATCH3 : entity EDULIB.DLATCH
135.       port map ( D => \$1N20\,
136.                  CLK => CLOCK,
137.                  Q => \$1N22\,
138.                  QN => Q2B,
139.                  R => RESET );
140.    DLATCH4 : entity EDULIB.DLATCH
141.       port map ( D => \$1N22\,
142.                  CLK => CLOCK,
143.                  Q => \$1N24\,
144.                  QN => Q3B,
145.                  R => RESET );
146.    DLATCH5 : entity EDULIB.DLATCH
147.       port map ( D => \$1N24\,
148.                  CLK => CLOCK,
149.                  Q => \$1N26\,
150.                  QN => Q4B,
151.                  R => RESET );
152.    DLATCH6 : entity EDULIB.DLATCH
153.       port map ( D => \$1N26\,
154.                  CLK => CLOCK,
155.                  Q => \$1N28\,
156.                  QN => Q5B,
157.                  R => RESET );
158.    DLATCH7 : entity EDULIB.DLATCH
159.       port map ( D => \$1N28\,
160.                  CLK => CLOCK,
161.                  Q => \$BUF_Q0\,
162.                  QN => Q6B,
163.                  R => RESET );
164.    DLATCH8 : entity EDULIB.DLATCH
165.       port map ( D => \$BUF_Q0\,
166.                  CLK => CLOCK,
167.                  Q => \$BUF_Q1\,
168.                  QN => Q7B,
169.                  R => RESET );
170.    DLATCH9 : entity EDULIB.DLATCH
171.       port map ( D => \$BUF_Q1\,
```

```
172.                  CLK => CLOCK,
173.                  Q => \$BUF_Q2\,
174.                  QN => Q8B,
175.                  R => RESET );
176.
177.    CONSTANT1: entity EDULIB.V_CONSTANT(IDEAL)
178.         generic map ( LEVEL => 2.5 )
179.         port map ( POS => REF,
180.                         NEG => ELECTRICAL_REF);
181.    A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
182.              --generic map ( LEVEL => 1.0 )
183.         port map ( A => REF,
184.                           D => RESET);
185.
186. end architecture arch_SHIFT16S8;
```

♦ Set_trigger

```
1.    -- Systems and Digital Signals Processing Laboratory
2.    -- genhdl\..\set_trigger.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Wed Sep 09 08:14:38 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity set_trigger is
19.   Port(
20.         signal CLK_D : out std_logic;
21.         signal SCLK_D : in std_logic;
22.         signal C1_D : in std_logic;
23.         signal C0_D : in std_logic);
24.   end entity set_trigger;
25.
26.   architecture arch_set_trigger of set_trigger is
27.     signal GNOT2 : std_logic;
28.     signal GNOT1 : std_logic;
29.     signal GNOT : std_logic;
30.     signal GAND1 : std_logic;
31.     signal GAND : std_logic;
32.   begin
33.     E_GNOT : entity EDULIB.INVERTER
34.       port map ( INPUT => C0_D,
35.                  OUTPUT => GNOT );
36.     E_GNOT1 : entity EDULIB.INVERTER
37.       port map ( INPUT => C1_D,
38.                  OUTPUT => GNOT1 );
39.     E_GAND : entity EDULIB.AND3
40.       port map ( IN1 => SCLK_D,
41.                  IN2 => GNOT,
42.                  IN3 => GNOT1,
43.                  OUTPUT => GAND );
```

```
44.    E_GNOT2 : entity EDULIB.INVERTER
45.      port map ( INPUT => SCLK_D,
46.                 OUTPUT => GNOT2 );
47.    E_GAND1 : entity EDULIB.AND3
48.      port map ( IN1 => C0_D,
49.                 IN2 => C1_D,
50.                 IN3 => GNOT2,
51.                 OUTPUT => GAND1 );
52.    E_GOR : entity EDULIB.OR2
53.      port map ( IN1 => GAND,
54.                 IN2 => GAND1,
55.                 OUTPUT => CLK_D );
56.  end architecture arch_set_trigger;
```

♦ Count_Pulse

```
1.   -- Systems and Digital Signals Processing Laboratory
2.   -- genhdl\..\Count_Pulse.vhd
3.   -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.   -- File created Wed Sep 09 08:14:38 2009
5.
6.   LIBRARY IEEE;
7.   USE ieee.std_logic_1164.all;
8.   USE ieee.electrical_systems.all;
9.   USE ieee.mechanical_systems.all;
10.  USE ieee.fluidic_systems.all;
11.  USE ieee.thermal_systems.all;
12.  USE ieee.radiant_systems.all;
13.  LIBRARY edulib;
14.  USE work.all;
15.  LIBRARY fundamentals_vda;
16.  LIBRARY spice2vhd;
17.
18.  entity COUNT_PULSE is
19.    port (signal LOAD: OUT STD_LOGIC;
20.          signal CLOCK: IN STD_LOGIC;
21.          signal RESET: IN STD_LOGIC);
22.  end entity COUNT_PULSE;
23.
24.  architecture arch_COUNT_PULSE of COUNT_PULSE is
25.       signal \$1N58\: STD_LOGIC;
26.       signal \$1N48\: STD_LOGIC;
27.       signal \$1N29\: STD_LOGIC;
28.       signal \$1N18\: STD_LOGIC;
29.       signal LIXO: STD_LOGIC;
30.       signal \$1N81\: STD_LOGIC;
31.       signal \$1N60\: STD_LOGIC;
32.       signal \$1N94\: STD_LOGIC;
33.       signal \$1N62\: STD_LOGIC;
34.       signal \$1N40\: STD_LOGIC;
35.       signal \$1N53\: STD_LOGIC;
36.       signal \$1N31\: STD_LOGIC;
37.       signal \$1N20\: STD_LOGIC;
38.       signal \$1N87\: STD_LOGIC;
39.       signal \$1N65\: STD_LOGIC;
40.       signal \$1N77\: STD_LOGIC;
41.       signal \$1N44\: STD_LOGIC;
42.       signal \$1N33\: STD_LOGIC;
43.       signal \$1N56\: STD_LOGIC;
44.       signal \$1N23\: STD_LOGIC;
45.       signal \$1N12\: STD_LOGIC;
```

```
46.       signal \$1N35\: STD_LOGIC;
47.   begin
48.
49.       DLATCH1: entity EDULIB.DLATCH
50.           port map (D => \$1N12\,
51.                 CLK => CLOCK,
52.                 Q => \$1N20\,
53.                 QN => \$1N12\,
54.                 R => RESET);
55.
56.       DLATCH2: entity EDULIB.DLATCH
57.           port map (D => \$1N23\,
58.                 CLK => CLOCK,
59.                 Q => \$1N18\,
60.                 QN => \$1N44\,
61.                 R => RESET);
62.
63.       DLATCH3: entity EDULIB.DLATCH
64.           port map (D => \$1N29\,
65.                 CLK => CLOCK,
66.                 Q => \$1N40\,
67.                 QN => \$1N48\,
68.                 R => RESET);
69.
70.       DLATCH4: entity EDULIB.DLATCH
71.           port map (D => \$1N62\,
72.                 CLK => CLOCK,
73.                 Q => LIXO,
74.                 QN => \$1N94\,
75.                 R => RESET);
76.
77.       DLATCH5: entity EDULIB.DLATCH
78.           port map (D => \$1N81\,
79.                 CLK => CLOCK,
80.                 Q => \$1N20\,
81.                 QN => \$1N12\,
82.                 R => RESET);
83.
84.       XOR2_1: entity EDULIB.XOR2
85.           port map (IN1 => \$1N20\,
86.                 IN2 => \$1N18\,
87.                 OUTPUT => \$1N23\);
88.
89.       AND2_1: entity EDULIB.AND2
90.           port map (IN1 => \$1N12\,
91.                 IN2 => \$1N65\,
92.                 OUTPUT => \$1N60\);
93.
94.       OR3_1: entity EDULIB.OR3
95.           port map (IN1 => \$1N31\,
96.                 IN2 => \$1N33\,
97.                 IN3 => \$1N35\,
98.                 OUTPUT => \$1N29\);
99.
100.      AND2_2: entity EDULIB.AND2
101.          port map (IN1 => \$1N40\,
102.                IN2 => \$1N12\,
103.                OUTPUT => \$1N31\);
104.
105.      AND2_3: entity EDULIB.AND2
106.          port map (IN1 => \$1N40\,
```

```
107.                IN2 => \$1N44\,
108.                OUTPUT => \$1N33\);
109.
110.      AND3_2: entity EDULIB.AND3
111.          port map (IN1 => \$1N20\,
112.                IN2 => \$1N18\,
113.                IN3 => \$1N48\,
114.                OUTPUT => \$1N35\);
115.
116.      AND4_1: entity EDULIB.AND4
117.          port map (IN1 => \$1N20\,
118.                IN2 => \$1N18\,
119.                IN3 => \$1N40\,
120.                IN4 => \$1N65\,
121.                OUTPUT => \$1N81\);
122.
123.      AND2_4: entity EDULIB.AND2
124.          port map (IN1 => \$1N44\,
125.                IN2 => \$1N65\,
126.                OUTPUT => \$1N58\);
127.
128.      AND2_5: entity EDULIB.AND2
129.          port map (IN1 => \$1N48\,
130.                IN2 => \$1N65\,
131.                OUTPUT => \$1N56\);
132.
133.      OR4_1: entity EDULIB.OR4
134.          port map (IN1 => \$1N60\,
135.                IN2 => \$1N58\,
136.                IN3 => \$1N56\,
137.                IN4 => \$1N53\,
138.                OUTPUT => \$1N62\);
139.
140.      AND4_2: entity EDULIB.AND4
141.          port map (IN1 => \$1N20\,
142.                IN2 => \$1N18\,
143.                IN3 => \$1N40\,
144.                IN4 => \$1N77\,
145.                OUTPUT => \$1N53\);
146.
147.      OR4_2: entity EDULIB.OR4
148.          port map (IN1 => \$1N94\,
149.                IN2 => \$1N20\,
150.                IN3 => \$1N18\,
151.                IN4 => \$1N40\,
152.                OUTPUT => \$1N87\);
153.
154.      OR2_1: entity EDULIB.OR2
155.          port map (IN1 => \$1N87\,
156.                IN2 => \$1N65\,
157.                OUTPUT => LOAD);
158.
159.  end architecture arch_COUNT_PULSE;
```

♦ Control_Load

```
1.    -- Systems and Digital Signals Processing Laboratory
2.    -- genhdl\..\Control_Load.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Wed Sep 09 08:14:38 2009
5.
```

```
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity CONTROL_LOAD is
19.      port (signal \!SYNC\: IN STD_LOGIC;
20.            signal C1: IN STD_LOGIC;
21.            signal CLOCK: OUT STD_LOGIC;
22.            signal C0: IN STD_LOGIC;
23.            signal PULSE: OUT STD_LOGIC;
24.            signal SCLK: IN STD_LOGIC);
25.   end entity CONTROL_LOAD;
26.
27.   architecture arch_CONTROL_LOAD of CONTROL_LOAD is
28.       signal \$BUF_PULSE\: STD_LOGIC;
29.       signal \$1N6\: STD_LOGIC;
30.       signal \$1N8\: STD_LOGIC;
31.       signal \$1N10\: STD_LOGIC;
32.   begin
33.
34.       PULSE <= \$BUF_PULSE\;
35.
36.       \$1|1\: entity WORK.SET_TRIGGER(ARCH_SET_TRIGGER)
37.            port map (C1_D => C0,
38.                  CLK_D => \$1N6\,
39.                  C0_D => C1,
40.                  SCLK_D => SCLK);
41.
42.       \$1|2\: entity WORK.COUNT_PULSE(ARCH_COUNT_PULSE)
43.            port map (CLOCK => \$1N10\,
44.                  LOAD => \$BUF_PULSE\,
45.                  RESET => \$1N8\);
46.
47.
48.       AND2_1: entity EDULIB.AND2
49.            port map (IN1 =>  \$1N10\,
50.                  IN2 => \$BUF_PULSE\,
51.                  OUTPUT => CLOCK);
52.
53.       INVERTER1: entity EDULIB.INVERTER
54.            port map (INPUT =>  \!SYNC\,
55.                  OUTPUT => \$1N8\);
56.
57.       OR2_1: entity EDULIB.OR2
58.            port map (IN1 =>  \$1N6\,
59.                  IN2 => \!SYNC\,
60.                  OUTPUT =>  \$1N10\);
61.
62.   end architecture arch_CONTROL_LOAD;
```

♦ DataLatch

```
1.    -- Systems and Digital Signals Processing Laboratory
```

```
2.    -- genhdl\..\DataLatch.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.    -- File created Wed Sep 09 08:14:38 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity DATALATCH is
19.      port( signal Q8_D : OUT STD_LOGIC;
20.            signal Q7_D : OUT STD_LOGIC;
21.            signal Q6_D : OUT STD_LOGIC;
22.            signal Q5_D : OUT STD_LOGIC;
23.            signal Q4_D : OUT STD_LOGIC;
24.            signal Q3_D : OUT STD_LOGIC;
25.            signal Q2_D : OUT STD_LOGIC;
26.            signal Q1_D : OUT STD_LOGIC;
27.            signal D8_D : IN STD_LOGIC;
28.            signal D7_D : IN STD_LOGIC;
29.            signal D6_D : IN STD_LOGIC;
30.            signal D5_D : IN STD_LOGIC;
31.            signal D4_D : IN STD_LOGIC;
32.            signal D3_D : IN STD_LOGIC;
33.            signal D2_D : IN STD_LOGIC;
34.            signal D1_D : IN STD_LOGIC;
35.            signal PULSE : IN STD_LOGIC );
36.   end entity DATALATCH;
37.
38.   architecture arch_DATALATCH of DATALATCH is
39.      signal ENABLE : STD_LOGIC;
40.      terminal \!EN\ : ELECTRICAL;
41.      signal NQ8 : STD_LOGIC;
42.      signal NQ7 : STD_LOGIC;
43.      signal NQ6 : STD_LOGIC;
44.      signal NQ5 : STD_LOGIC;
45.      signal NQ4 : STD_LOGIC;
46.      signal NQ3 : STD_LOGIC;
47.      signal NQ2 : STD_LOGIC;
48.      signal NQ1 : STD_LOGIC;
49.   begin
50.
51.      DLATCH1 : entity EDULIB.DLATCH
52.         port map ( D => D1_D,
53.                    CLK => PULSE,
54.                    Q => Q1_D,
55.                    QN => NQ1,
56.                    R => ENABLE );
57.
58.      DLATCH2 : entity EDULIB.DLATCH
59.         port map ( D => D2_D,
60.                    CLK => PULSE,
61.                    Q => Q2_D,
62.                    QN => NQ2,
```

```
63.                  R => ENABLE );
64.
65.     DLATCH3 : entity EDULIB.DLATCH
66.        port map ( D => D3_D,
67.                   CLK => PULSE,
68.                   Q => Q3_D,
69.                   QN => NQ3,
70.                   R => ENABLE );
71.
72.     DLATCH4 : entity EDULIB.DLATCH
73.        port map ( D => D4_D,
74.                   CLK => PULSE,
75.                   Q => Q4_D,
76.                   QN => NQ4,
77.                   R => ENABLE );
78.
79.     DLATCH5 : entity EDULIB.DLATCH
80.        port map ( D => D5_D,
81.                   CLK => PULSE,
82.                   Q => Q5_D,
83.                   QN => NQ5,
84.                   R => ENABLE );
85.
86.     DLATCH6 : entity EDULIB.DLATCH
87.        port map ( D => D6_D,
88.                   CLK => PULSE,
89.                   Q => Q6_D,
90.                   QN => NQ6,
91.                   R => ENABLE );
92.
93.     DLATCH7 : entity EDULIB.DLATCH
94.        port map ( D => D7_D,
95.                   CLK => PULSE,
96.                   Q => Q7_D,
97.                   QN => NQ7,
98.                   R => ENABLE );
99.
100.    DLATCH8 : entity EDULIB.DLATCH
101.       port map ( D => D8_D,
102.                  CLK => PULSE,
103.                  Q => Q8_D,
104.                  QN => NQ8,
105.                  R => ENABLE );
106.
107.    V_ENABLE: entity EDULIB.V_CONSTANT(IDEAL)
108.         generic map ( LEVEL => 5.0 )
109.         port map ( POS => \!EN\,
110.                    NEG => ELECTRICAL_REF);
111.
112.    A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
113.       --generic map ( LEVEL => 1.0 )
114.       port map (A => \!EN\,
115.                 D => ENABLE);
116.
117. end architecture arch_DATALATCH;
```

♦ DAC_Data_Latch

```
1.    -- Systems and Digital Signals Processing Laboratory
2.    -- genhdl\..\Data_Latch_Drives.vhd
3.    -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
```

```
4.    -- File created Wed Sep 09 08:14:38 2009
5.
6.    LIBRARY IEEE;
7.    USE ieee.std_logic_1164.all;
8.    USE ieee.electrical_systems.all;
9.    USE ieee.mechanical_systems.all;
10.   USE ieee.fluidic_systems.all;
11.   USE ieee.thermal_systems.all;
12.   USE ieee.radiant_systems.all;
13.   LIBRARY edulib;
14.   USE work.all;
15.   LIBRARY fundamentals_vda;
16.   LIBRARY spice2vhd;
17.
18.   entity DATA_LATCH_DRIVES is
19.       port (signal \!SYNC\: IN STD_LOGIC;
20.             signal D0: OUT STD_LOGIC;
21.             signal D1: OUT STD_LOGIC;
22.             signal D2: OUT STD_LOGIC;
23.             signal D3: OUT STD_LOGIC;
24.             signal D4: OUT STD_LOGIC;
25.             signal D5: OUT STD_LOGIC;
26.             signal D6: OUT STD_LOGIC;
27.             signal D7: OUT STD_LOGIC;
28.             signal INPUT: IN STD_LOGIC;
29.             signal SCLK: IN STD_LOGIC);
30.   end entity DATA_LATCH_DRIVES;
31.
32.   architecture arch_DATA_LATCH_DRIVES of DATA_LATCH_DRIVES is
33.        signal \$1N26\: STD_LOGIC;
34.        signal \$1N16\: STD_LOGIC;
35.        signal \$1N28\: STD_LOGIC;
36.        signal \$1N18\: STD_LOGIC;
37.        signal \$1N73\: STD_LOGIC;
38.        signal RELOGIO: STD_LOGIC;
39.        signal \$1N30\: STD_LOGIC;
40.        signal \$1N20\: STD_LOGIC;
41.        signal \$1N32\: STD_LOGIC;
42.        signal \$1N10\: STD_LOGIC;
43.        signal \$1N22\: STD_LOGIC;
44.        signal \$1N45\: STD_LOGIC;
45.        signal PULSO: STD_LOGIC;
46.        signal \$1N24\: STD_LOGIC;
47.   begin
48.
49.       \1|6\: entity WORK.CONTROL_LOAD(ARCH_CONTROL_LOAD)
50.            port map (\!SYNC\ => \!SYNC\,
51.                 C1 => \$1N73\,
52.                 CLOCK => RELOGIO,
53.                 C0 => \$1N45\,
54.                 PULSE => PULSO,
55.                 SCLK => SCLK);
56.
57.       \1|7\: entity WORK.DATALATCH(ARCH_DATALATCH)
58.             port map (D1_D => \$1N18\,
59.                 D2_D => \$1N20\,
60.                 D3_D => \$1N22\,
61.                 D4_D => \$1N24\,
62.                 D5_D => \$1N26\,
63.                 D6_D => \$1N28\,
64.                 D7_D => \$1N30\,
```

```
65.                 D8_D => \$1N32\,
66.                 PULSE => PULSO,
67.                 Q1_D => D0,
68.                 Q2_D => D1,
69.                 Q3_D => D2,
70.                 Q4_D => D3,
71.                 Q5_D => D4,
72.                 Q6_D => D5,
73.                 Q7_D => D6,
74.                 Q8_D => D7);
75.
76.      \1|8\: entity WORK.SHIFT16S8(ARCH_SHIFT16S8)
77.           port map (C0 => \$1N10\,
78.                 C1 => \$1N16\,
79.                 CLOCK => RELOGIO,
80.                 INPUT => INPUT,
81.                 Q0 => \$1N18\,
82.                 Q1 => \$1N20\,
83.                 Q2 => \$1N22\,
84.                 Q3 => \$1N24\,
85.                 Q4 => \$1N26\,
86.                 Q5 => \$1N28\,
87.                 Q6 => \$1N30\,
88.                 Q7 => \$1N32\);
89.
90.      \1|9\: entity WORK.CONTROL_SHIFT(ARCH_CONTROL_SHIFT)
91.           port map (C0 => \$1N73\,
92.                 C1 => \$1N45\,
93.                 D0 => \$1N10\,
94.                 D1 => \$1N16\,
95.                 PULSE => PULSO);
96.
97.  end architecture arch_DATA_LATCH_DRIVES;
```

♦ AD5450

```
1.   -- Digital Systems and Signals Processing Laboratory
2.   -- genhdl\..\AD5450.vhd
3.   -- Generated by MS2SV (Matlab / Simulink to SystemVision) version 1.7
4.   -- File created Wed Sep 09 08:14:38 2009
5.
6.   library IEEE;
7.   use ieee.std_logic_1164.all;
8.   use ieee.electrical_systems.all;
9.   use ieee.mechanical_systems.all;
10.  use ieee.fluidic_systems.all;
11.  use ieee.thermal_systems.all;
12.  use ieee.radiant_systems.all;
13.  library EduLib;
14.  use WORK.all;
15.  library fundamentals_vda;
16.  library spice2vhd;
17.
18.  entity AD5450 is
19.     port (
20.         signal SDIN: in std_logic;
21.         signal SCLK: IN std_logic;
22.         terminal VOUT1: electrical);
23.  end entity AD5450;
24.  architecture arch_AD5450 of AD5450 is
25.       terminal \$1N47\: ELECTRICAL;
```

```
26.       signal \$1N25\: STD_LOGIC;
27.       signal \$1N15\: STD_LOGIC;
28.       signal \$1N17\: STD_LOGIC;
29.       signal \$1N29\: STD_LOGIC;
30.       signal \$1N19\: STD_LOGIC;
31.       signal \$1N21\: STD_LOGIC;
32.       signal \$1N23\: STD_LOGIC;
33.       signal \$1N12\: STD_LOGIC;
34.       signal \!SYNC\: std_logic;
35.       terminal SYNC: electrical;
36.       terminal VREF: electrical;
37.  begin
38.        \$1|1\: entity WORK.LADDER_R2RC (ARCH_LADDER_R2RC)
39.             port map (IN1_D => \$1N12\,
40.                  IN2_D => \$1N15\,
41.                  IN3_D => \$1N17\,
42.                  IN4_D => \$1N19\,
43.                  IN5_D => \$1N21\,
44.                  IN6_D => \$1N23\,
45.                  IN7_D => \$1N25\,
46.                  IN8_D => \$1N29\,
47.                  OUT_E => \$1N47\);
48.
49.        \$1|2\: entity WORK.DATA_LATCH_DRIVES (ARCH_DATA_LATCH_DRIVES)
50.             port map (\!SYNC\ => \!SYNC\,
51.                  D0 => \$1N12\,
52.                  D1 => \$1N15\,
53.                  D2 => \$1N17\,
54.                  D3 => \$1N19\,
55.                  D4 => \$1N21\,
56.                  D5 => \$1N23\,
57.                  D6 => \$1N25\,
58.                  D7 => \$1N29\,
59.                  INPUT => SDIN,
60.                  SCLK => SCLK);
61.
62.        E_MULT1: entity EDULIB.E_MULT
63.             port map (IN1 => \$1N47\,
64.                  IN2 => VREF,
65.                  OUTPUT => VOUT1);
66.
67.        V_SYNC: entity EDULIB.V_CONSTANT(IDEAL)
68.             generic map ( LEVEL => 0.0 )
69.             port map ( POS => SYNC,
70.                        NEG => ELECTRICAL_REF);
71.
72.        A2D_BIT1: entity EDULIB.A2D_BIT(IDEAL)
73.             port map (A => SYNC,
74.                       D => \!SYNC\);
75.
76.        V_VREF: entity EDULIB.V_CONSTANT(IDEAL)
77.             generic map ( LEVEL => 10.0 )
78.             port map ( POS => VREF,
79.                        NEG => ELECTRICAL_REF);
80.
81.    end architecture arch_AD5450;
```