
UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRUNNA DE SOUZA ALBUQUERQUE

# DESCRIÇÃO DE UM CONVERSOR ANALÓGICO-DIGITAL DE 12 BITS

JOÃO PESSOA
2013

BRUNNA DE SOUZA ALBUQUERQUE

# DESCRIÇÃO DE UM CONVERSOR ANALÓGICO-DIGITAL DE 12 BITS

Monografia submetida ao Curso de Bacharelado em Ciência da Computação da Universidade Federal da Paraíba para a conclusão do curso e obtenção do título de Bacharel em Ciência da Computação.

**Orientador: Prof. Dr. José Antônio Gomes de Lima**

JOÃO PESSOA
2013

Catalogação na publicação
Universidade Federal da Paraíba
Biblioteca Setorial do CCEN

A345d Albuquerque, Brunna de Souza.

Descrição de um conversor analógico-digital de 12 Bits / Brunna de Souza Albuquerque. -- João Pessoa, 2013.

63p. : il.

Monografia (Graduação em Ciência da computação) - Universidade Federal da Paraíba.

Orientador: Prof. Dr. José Antônio Gomes de Lima.

BRUNNA DE SOUZA ALBUQUERQUE

DESCRIÇÃO DE UM CONVERSOR ANALÓGICO-DIGITAL DE 12 BITS

Monografia submetida ao Curso de Bacharelado em Ciência da Computação da Universidade Federal da Paraíba para a conclusão do curso e obtenção do título de Bacharel em Ciência da Computação.

Data de defesa: 19/04/13
Resultado: _____

BANCA EXAMINADORA

José Antônio Gomes de Lima Prof. Dr. _____
Universidade Federal da Paraíba

Hamilton Soares da Silva Prof. Me. _____
Universidade Federal da Paraíba

*A meus pais, que me ensinaram e continuam me ensinando que o estudo é a coisa mais importante que podemos ter.*

# AGRADECIMENTOS

A Deus acima de tudo, pois mesmo tendo minha família e amigos, quando estava só era a Ele que eu recorria, seja para me iluminar, para dar-me forças, ou para agradecer cada vitória alcançada.

A meus pais, Josinete e Francisco, que fizeram de tudo para que eu pudesse me graduar, mesmo estando tão longe de mim, sempre me apoiaram, me ajudaram, e me pediam para que eu ficasse mais um dia toda vez que eu viajava para passar um fim de semana com eles.

A meus tios, Ana Lúcia e Robério, que foram uma parte importante na minha graduação, e sempre me deram apoio e me acolheram de braços abertos. Sem eles, com certeza ela seria bem mais difícil.

A toda a minha família, meus tios, primos, que compartilharam de cada alegria e cada situação difícil por que passei, sempre me apoiando e torcendo por mim.

A minha irmã, Amanda, que sempre compartilhou de tudo o que eu fiz, e pelo companheirismo de todos os dias, seja pelo telefone ou pela internet.

A meus amigos, sempre presentes aonde quer que eu vá. Aos velhos amigos, que me deixavam com o coração apertado de saudade toda vez que perguntavam quando eu iria visitá-los e que nunca deixaram eu me esquecer de onde vim. Aos novos amigos, que me ajudaram a amaciar a saudade do que deixei em Pernambuco quando vim pra cá e me acompanharam durante esse período, pois muitos viraram tão queridos quanto os que tenho de longa data.

A meus professores de toda a graduação, que me ajudaram nessa busca do conhecimento, me compreenderam, me auxiliaram, me deram broncas de vez em quando e me adotaram. Sem vocês certamente eu não estaria terminando minha graduação.

Ao meu orientador, José Antônio, pela paciência desde o primeiro contato, no início do curso, e pelo infinito auxílio durante a construção desse trabalho. Muito obrigada, professor.

A todas as pessoas que simplesmente esqueci-me de mencionar, mas que passaram por minha vida e me deixaram algum aprendizado, podem ter certeza de que ele foi empregado aqui.

*“[...] a ênfase da arquitetura [de computadores] vai além das necessidades do usuário, assim como na engenharia a ênfase vai além das necessidades do fabricante.”*

F. P. Brooks, Jr.

RESUMO

O presente trabalho apresenta uma descrição detalhada de um sistema conversor analógico-digital. Utilizando componentes como um conversor de aproximações sucessivas, tem como principal objetivo fornecer uma descrição completa, simples, didática e objetiva dos processos que envolveram o seu desenvolvimento. A partir da constante observação, verificação e simulação do código, sempre buscando comparações com trabalhos similares, outros conversores A/D e materiais de apoio, a descrição foi tomando forma naturalmente, discorrendo sobre os módulos do sistema e como eles interagem para a obtenção do resultado desejado. Uma breve apresentação dos elementos necessários para obter um entendimento completo do sistema foi exposta, e assim, ele pode então mostrar-se sem empecilhos para o leitor. Considerações futuras, considerando, por exemplo, o modo como a captação dos dados a partir de sinais externos é feita, podem ser avaliadas e implementadas, atentando sempre para o melhoramento das técnicas empregadas, sem desprezar a tecnologia mais aceitável e eficaz para a realização da atualização do sistema.

**Palavras-chave**: Conversor Analógico-Digital, Descrição, Circuitos Integrados, Sistemas Digitais, VHDL, USB.

ABSTRACT

This work provides a detailed description of an analog-to-digital converter system. Using components like a successive approximation converter, its main goal is to report a complete, simple, didactic and objective description of the procedures that were part of its development. From the constant observation, checking and simulation of the source code, and always looking for comparisons with similar work, other A/D converters and supporting material, the description was naturally taking shape, discussing about the system's units, and how they interact to reach the expected results. A brief presentation of the elements required to obtain a complete understanding of it was reported, and then this understanding could be easier to readers. Future works can be evaluated and implemented, considering, for example, the way the data's capture from external signals is done, always paying attention to improve the used techniques, without forgetting the most acceptable and effective technology to update the system.

**Palavras-chave**: Analog-to-Digital Converter, Description, Integrated Circuits, Digital Systems, VHDL, USB.

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CE | Chip Enable Signal |
| CI | Circuito Integrado |
| CPLD | Complex Programmable Logic Device |
| $\overline{CS}$ | Chip Select Signal |
| DARPA | Defense Advanced Research Projects Agency – USA |
| DIP | Dual In-Line Package |
| EOC | End of Conversion Signal |
| FIFO | First In First Out |
| FPGA | Field Programmable Gate Array |
| FTDI | Future Technology Devices International |
| HDL | Hardware Description Language |
| IEEE | Institute of Electrical and Electronics Engineers |
| LSB | Least Significant Bit |
| MSB | Most Significant Bit |
| PLCC | Plastic Leader Chip Carrier |
| PWM | Pulse-Width Modulation |
| R/$\overline{C}$ | Read/Convert Signal |
| RCC | Ramp-Compare Converter |
| SAC | Successive Approximation Converter |
| SAR | Successive Approximation Register |
| SHA | Sample-and-Hold Amplifier |
| STS | Status Signal |
| USB | Universal Serial Bus |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

# 1 *INTRODUÇÃO*

O surgimento da Arquitetura de Computadores ainda é incerto, assim como o surgimento da própria Computação. A definição do termo "Arquitetura de Computadores" foi citada certa vez (pois se tratando de história é difícil afirmar com precisão as primeiras ocorrências de máquinas, ferramentas e nomenclaturas) por Frederick P. Brooks Jr., em "Planning a Computer System – Project Stretch", dizendo que

> [A] Arquitetura de Computadores, como qualquer arquitetura, é a arte de determinar as necessidades do usuário de uma estrutura e assim projetar para satisfazer essas necessidades tão efetivamente quanto possível, considerando as implicações econômicas e tecnológicas. (BUCHHOLZ, 1962, p. 5, tradução nossa).

A Arquitetura de Computadores engloba, portanto, a descrição do "comportamento funcional de um sistema computacional, do ponto de vista do programador" (MURDOCCA; HEURING, 1999, p. 3). Ela permite que mesmo uma pessoa encarregada de projetar o mais alto nível compreenda o que o computador pode fazer, em termos de níveis mais baixos. Essa definição estende-se ainda aos mais diversos tipos de arquitetura, por assim dizer. Ou seja, pode-se aplicar o termo à descrição dos conjuntos de instruções (mnemônicos), a arquiteturas de hardware (como sistemas distribuídos), à descrição dos requisitos (memória, periféricos, capacidade de processamento, etc.), ou ao conjunto de atributos do computador que um programador precisa compreender para efetuar a programação com sucesso.

Desde então, a arquitetura de computadores vem se aperfeiçoando à medida que surgem novas técnicas e releituras de sistemas, instruções e atributos de computadores, sempre identificando e melhorando aspectos referentes à qualidade, ao desempenho e à aplicação para a qual o computador foi designado. No início, havia apenas componentes eletrônicos: diodos, válvulas, transistores. Em 1958, Jack Kilby juntou os componentes num mesmo pedaço de germânio, manteve-os juntos por meio de um tipo de cera e assim, construiu o primeiro circuito integrado (CI), um oscilador de deslocamento de fase.

Iniciava-se, assim, uma fase de grande e rápida expansão da arquitetura de computadores e dos sistemas digitais. Os componentes foram diminuindo de tamanho, sua capacidade de processamento e armazenamento foi aumentando, problemas como ruído, erros de ganho e temperaturas altas foram sendo controlados, enquanto linguagens de descrição desses componentes surgiam e eram aperfeiçoadas.

Com essa expansão rápida do conhecimento acerca dos computadores, seus métodos de fabricação, componentes, capacidade de processamento e armazenamento, foram surgindo diversos tipos de dispositivos diferentes uns dos outros, mas que muitas vezes desempenhavam funções similares. Com isso veio a dificuldade de comunicação entre esses diversos componentes. Até a década de 1980, os fabricantes não conseguiam chegar a um acordo sobre normas de fabricação e descrição desses componentes, quando o Departamento de Defesa dos Estados Unidos (DARPA), convocou vários especialistas da área e conseguiu criar uma linguagem de descrição de hardware, a VHDL, a ser detalhada mais adiante na seção 2.1.

Antes, porém, do surgimento da VHDL, outra linguagem de descrição de hardware, a Verilog, foi projetada durante o inverno de 1983/84 (aproximadamente entre dezembro de 1083 e março de 1984) e hoje está padronizada como IEEE Std. 1364-1995. Desenvolvida como um produto de verificação/simulação, foram posteriormente adicionados a ela um analisador de tempo e um simulador de erros. Alguns módulos podem ser difíceis de separar, como a linguagem propriamente dita da ferramenta de simulação, porque alguns aspectos da linguagem são definidos pelo modo como o simulador trabalha.

Com essas duas principais linguagens de descrição de hardware, ficou mais fácil a comunicação entre as descrições dos componentes utilizados nos computadores. Estruturas frequentemente utilizadas foram sendo projetadas, como codificadores e decodificadores, clocks, latches, multiplexadores, blocos tri-state, divisores de frequência e conversores, entre outros, que são objetos de estudo de pesquisadores e fabricantes ao redor do mundo, tanto para aperfeiçoá-los, quanto para a utilização didática.

Entre os conversores, podemos citar os conversores de tipos internos à linguagem (*integer* para *std_logic_vector*, por exemplo), os conversores de formatos (binário para ASCII), conversores de frequências de clock, conversores de dados analógicos para digitais. Os tipos de conversão são inúmeros, e podem envolver quantidades diferentes de bits e o modo como o código é implementado.

Conversores analógico-digitais são tema de vários trabalhos de faculdades e universidades. A maneira como são feitos é objeto de pesquisa e aprendizado de estudantes de engenharia e eletrônica, mas a maneira como cada equipe desenvolvedora implementa, o objetivo a que é destinado e os componentes integrados ao sistema podem diferenciar o modo de ver de cada membro desenvolvedor e influenciar na implementação. Os conversores A/D (ou qualquer outro tipo de conversor) mais utilizados são os de 8 bits, principalmente por ser um número bastante razoável, nem muito trivial e nem muito complicado para o processo de aprendizagem da linguagem VHDL e de compreensão do assunto.

Os conversores A/D são, por diversas vezes tomados como exemplos para aprendizagem por causa do envolvimento e aplicação de vários conceitos e componentes básicos, mas importantes da arquitetura de computadores e dos sistemas digitais. Elementos como multiplexadores, blocos tri-state, amplificadores, contadores, divisores de frequência, registradores e clocks são empregados no projeto desses conversores. Estudando um conversor A/D, pode-se ver na prática como esses componentes interagem entre si e como funcionam.

Ferreira et al. (2003) descreve conversores A/D e D/A do tipo PWM (modulação por largura de pulso). Esses conversores, descritos em VHDL e implementados em dispositivos FPGAs, utilizam também conversores A/D de aproximações sucessivas. Dois fatores principais para seu uso é o tempo fixo de conversão e a fácil compreensão de seu funcionamento.

## 1.1 MOTIVAÇÃO

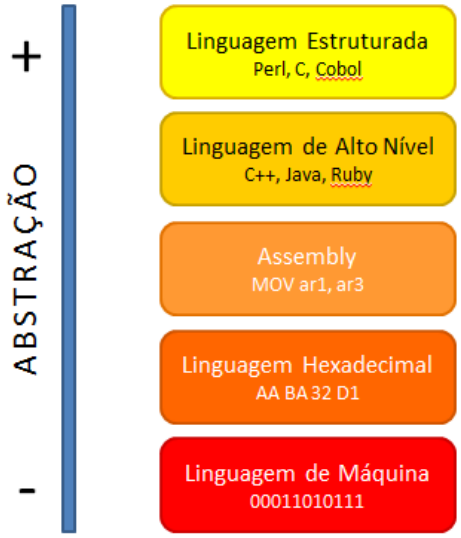
Os circuitos integrados são apenas pequenas peças de um quebra-cabeça imensamente maior. A partir de minúsculas estruturas, as portas lógicas, podem ser feitas infinitas combinações, a fim de conceber os mais variados sistemas para qualquer objetivo. A versatilidade da descrição dos sistemas, sendo ela comportamental, estrutural ou de fluxo de dados, faz com que eles sejam intuitivos e simples, mesmo que utilizem linguagens complexas, ou apresentem milhares de componentes.

Situados num nível mais baixo do que linguagens de programação orientadas a objeto, por exemplo, programas escritos em VHDL ou Verilog descrevem o funcionamento de estruturas que servirão de base para os níveis mais altos. Um programa escrito em linguagem de descrição de hardware é uma ponte entre os componentes físicos e os

componentes lógicos de um sistema, auxiliando na comunicação entre os diversos níveis, como a interação entre os níveis do modelo OSI ou TCP/IP.

Na figura 1, podemos observar a classificação das linguagens de programação, segundo o seu nível de abstração. As linguagens de descrição de hardware são responsáveis pela descrição formal de circuitos eletrônicos; se as compararmos com as linguagens de programação, elas se aproximariam do papel de fazer a comunicação entre a linguagem Assembly e as linguagens de alto nível; mas ao mesmo tempo temos que ter cuidado com essa analogia, pois alguns mecanismos como impressão na tela do console ou captação de variáveis a partir do teclado, típicas de linguagens de programação, não existem na VHDL. Ou seja, como ocorre nos níveis referentes aos modelos de redes de computadores, não existe comunicação efetiva sem que haja uma ponte que ligue ambos os lados para poder desempenhar seu papel com sucesso, garantindo o livre tráfego das informações.

**Figura 1 - Classificação das linguagens de programação, de acordo com o nível de abstração.**




**Fonte: Elaboração própria.**

A linguagem VHDL pode ser um pouco difícil de compreender no início, mas ao mesmo tempo, torna-se simples quando passamos a tomar conhecimento dos seus aspectos e comandos mais importantes. A VHDL possui semelhanças com a linguagem de programação Pascal, e, dessa maneira, pode tornar o aprendizado mais fácil, principalmente para quem já domina essa linguagem. Então, muitos projetos são implementados utilizando essa linguagem de descrição de hardware.

Projetos escritos em VHDL, em geral, tornam-se mais compreensíveis mais pelos componentes que utilizam na sua implementação (muitos deles didáticos e bastante

explorados), do que pela simples leitura e estudo do código fonte. Técnicas e algoritmos que são passados através de professores, artigos, aulas, palestras, e outros projetos são constantemente reciclados e reaproveitados em novos projetos. Com isso, a aprendizagem sobre um projeto específico pode levar, no futuro, ao entendimento quase que integral de projetos supostamente desconhecidos pelo pesquisador/projetista, mas que utilizam de conceitos anteriormente vistos em projetos antigos, levando o pesquisador/projetista à total compreensão do sistema sem maiores esforços.

## 1.2 OBJETIVOS

O trabalho corrente tem como objetivo geral descrever o mais clara e objetivamente possível o desenvolvimento do projeto do conversor A/D de 12 bits, contribuindo, assim, para o processo de aprendizagem de alunos que estão iniciando na área de sistemas digitais e circuitos integrados, através de um material detalhado sobre técnicas utilizadas e funcionamento do sistema.

Os objetivos específicos, por sua vez, podem ser assim descritos:

1. Fornecer uma descrição a um projeto de conversor A/D, visto que há uma pequena quantidade de materiais similares disponíveis para estudo;
2. Contribuir para o entendimento do sistema através de uma documentação clara e objetiva, que descreva em linguagem simples e direta os componentes do sistema;
3. Oferecer um material didático que auxilie os estudantes de sistemas digitais a compreender melhor o funcionamento de componentes vistos teoricamente e suas aplicações;
4. Permitir a trabalhos futuros uma compreensão rápida do sistema atual, adquirida com uma leitura rápida e dinâmica, baseada nas ilustrações e esquemas que o trabalho apresenta.

# 2 *FUNDAMENTAÇÃO TEÓRICA*

A fundamentação teórica foi elaborada principalmente com o auxílio de fontes mais didáticas do que científicas. O próprio tema do trabalho propõe a descrição do sistema conversor e assim torná-lo mais compreensível a quem constantemente lida com essa área e não dispõe de uma quantidade maior de tempo para estudar e analisar sistemas em geral.

O principal ponto de partida foi o artigo publicado pelos responsáveis pelo sistema, Costa et al. (2008). A partir dele, foi feito um estudo do sistema, bem como uma análise e compreensão do seu código fonte VHDL, e elaborados esquemas e descrições de seu funcionamento e objetivos. Com o aprofundamento de questões importantes para a implementação do sistema, como componentes externos, foi necessário adicionar à pesquisa fontes didáticas, que explicassem melhor esses elementos.

A partir da estrutura do sistema, módulos foram identificados e estudados separadamente, exigindo, para cada conceito e elemento utilizado, um estudo específico que é apresentado em cada seção.

## 2.1 A LINGUAGEM VHDL

A linguagem VHDL (ou VHSIC Hardware Description Language) foi desenvolvida para atender às necessidades do projeto VHSIC, do DARPA. No início da década de 1980, o DARPA promoveu um encontro de várias entidades e especialistas da área a fim de padronizar métodos de descrição de circuitos. Através da criação da linguagem VHDL, que foi sendo atualizada com o passar dos anos com a implementação de novos pacotes e versões, foi possível organizar a descrição de circuitos de modo que o intercâmbio de informações entre fabricantes, empresas e fornecedores se tornasse mais fácil, além de facilitar também a descrição de sistemas integrados mais complexos.

Padronizada pelo IEEE em 1986, a VHDL hoje utilizada é uma versão atualizada, cujo padrão mais amplamente utilizado é o IEEE Std. 1076-1993 juntamente com o IEEE Std. 1164-1993. Eles especificam mudanças relacionadas ao tratamento de arquivos e à definição

do pacote que traz consigo a implementação de tipos de dados importantes, como o *std_logic*, respectivamente.

Como toda linguagem (tanto de programação quanto de descrição de hardware), a VHDL apresenta algumas características importantes, descritas a seguir:

- Diferentes alternativas de descrição: a VHDL permite ao projetista particionar o sistema em diferentes níveis, como o nível de sistema, de transferência entre registradores, de circuito e o nível lógico, proporcionando assim uma melhor legibilidade;
- Rapidez no projeto: através de sentenças descrevendo o comportamento, a estrutura ou o aspecto físico do circuito, tal como numa linguagem de programação, permite que o tempo gasto com o projeto seja diminuído devido aos vários níveis de abstração;
- Verificação do comportamento do sistema digital: por meio da simulação, pode-se verificar se o sistema se comporta como o esperado;
- Portabilidade: o código pode ser utilizado com diversas tecnologias, devido à sua padronização;
- Descrição estruturada: o projeto pode ser composto por subprojetos, para depois serem conectados entre si;
- Simulação: antes da síntese do projeto, ele é simulado, diminuindo a quantidade de erros, e consequentemente reduzindo custos;
- Similaridade com linguagens de programação: funções podem ser especificadas de modos semelhantes a funções (ou procedimentos) de linguagens de programação, facilitando a compreensão.

Mesmo com alguns comandos bastante semelhantes a comandos de algumas linguagens de programação, a VHDL é complexa, e às vezes de difícil entendimento, por causa das várias opções de modelagem do comportamento de um circuito (D'AMORE, 2005, p. 3). Mas, por outro lado, com a compreensão de alguns poucos comandos da linguagem, pode-se obter um nível de entendimento suficientemente bom para a modelagem de estruturas complexas e compreensão de códigos.

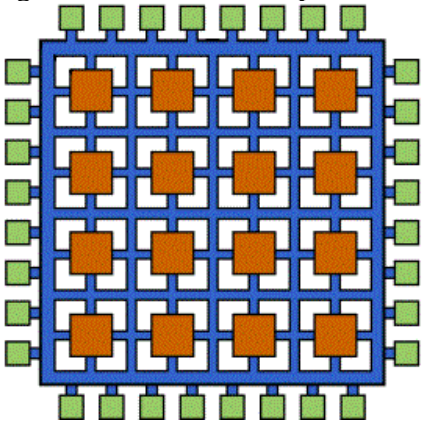
## 2.2 DISPOSITIVOS FPGAs

Com a exclusividade dos circuitos orientados à aplicação, seu alto custo de projeto e implementação e sua inflexibilidade – ainda que eles sejam eficientes, utilizem menos

recursos e seus resultados sejam satisfatórios – fazem os projetistas de hardware buscarem alternativas para a construção de seus circuitos. A computação reconfigurável admite a personalização, permitindo ao projetista a programação específica do dispositivo para cada objetivo a ser alcançado, combinado com o fato de poder ser reprogramado, quando forem necessárias mudanças nesse objetivo.

Os principais representantes da computação reconfigurável são os FPGAs (Field Programmable Gate Arrays) e os CPLDs (Complex Programmable Logic Devices). O FPGA é um conjunto de centenas (ou milhares) de unidades lógicas. Estas unidades lógicas podem ser vistas como componentes padrões que podem ser configurados independentemente e interconectados a partir de uma matriz de trilhas condutoras e switches programáveis (CARDOSO; FERNANDES, 2007, p. 1). Ou seja, os FPGAs são basicamente compostos pelas unidades lógicas, organizadas em arrays, e pelos switches, responsáveis por ligar cada uma dessas unidades.

**Figura 2 - Esquema de um FPGA. Os quadrados verdes representam os blocos de E/S, os quadrados laranjas representam as unidades lógicas, e as linhas azuis representam os switches.**



**Fonte: www.pldworld.com, adaptada.**

## 2.3 CONVERSORES DE APROXIMAÇÕES SUCESSIVAS (SACs)

Sabemos que conversores analógico-digitais (A/D) são dispositivos que transformam sinais analógicos em códigos digitais como saída. Existem vários tipos de conversores A/D, como os de rampa digital (RCC), de aproximações sucessivas (SAC) e flash, por exemplo. Nesta seção será descrito o conversor de aproximações sucessivas, tipo do AD574A, conversor escolhido para o projeto.

Basicamente, os conversores A/D apresentam a mesma estrutura, e seguem passos semelhantes, diferindo apenas na implementação das operações e nos dispositivos otimizadores, por exemplo. Numa descrição menos técnica, recebe-se uma voltagem como

entrada e é feita uma busca pelo número digital que mais se aproxime do original, filtrando e verificando a cada novo ciclo se esse número se encaixa nos requisitos determinados pelo conversor.

Segundo Tocci e Widmer (2000), o funcionamento desses conversores pode ser assim descrito: a conversão é iniciada com o pulso *START*, que faz com que o bloco de controle modifique continuamente os dados de entrada, de acordo com a frequência do clock, armazenando-os no registrador (SAR). Esses dados (digitais) são convertidos para uma tensão analógica pelo conversor D/A (interno ao conversor A/D), que será comparada com a entrada analógica original, para verificar se os valores desejados foram atingidos; ou seja, quando o valor que sai do conversor D/A (e consequentemente o valor no SAR também, já que é o equivalente digital do valor da saída do conversor D/A) estiver próximo o suficiente do valor analógico original, o bloco de controle sinaliza então que a conversão está completa, por meio do sinal *EOC*.

O SAC é um conversor bastante utilizado, e um dos motivos para isso é o fato do seu tempo de conversão ser relativamente baixo em relação ao RCC, que tem um tempo máximo de conversão de $(2^N - 1) \times C$[1] segundos, enquanto o SAC apresenta um tempo de $N \times C$ segundos (SATO, p. 14; TOCCI; WIDMER, 2000, p. 398). Por causa disso, o SAC é largamente utilizado em aplicações onde o dado analógico muda constantemente, permitindo que mais dados sejam coletados num intervalo de tempo menor. Além disso, esse tempo de conversão do SAC é sempre fixo independentemente do valor da entrada analógica.

Por outro lado, o SAC contém apenas um comparador. Isso significa que se houver algum erro de offset, isto irá interferir em todos os bits da conversão e então, a operação inteira estará comprometida. Corrigindo o erro de offset, ainda pode haver o erro de ganho, que é a diferença entre a saída real e a saída esperada.

Sua lógica de controle baseia-se em modificações contínuas bit a bit do valor de entrada até que o equivalente digital do SAR esteja dentro da resolução aceita pelo conversor, ou seja, num intervalo de aproximação suficientemente próximo do valor de entrada. Tomando um exemplo de Tocci e Widmer (2000), é considerado um valor de entrada ($V_A$) de 10,4 V. A tabela 1 e a figura 3 ilustram o funcionamento de um SAC de 4 bits.

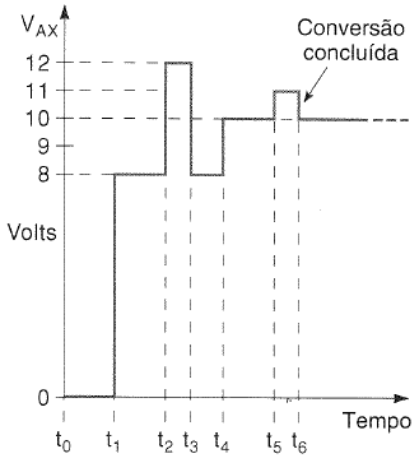
[1] N representa a quantidade de bits e C representa a frequência do clock.

**Tabela 1 – Valores dos dados analógicos e digitais no SAC durante a operação de verificação do conversor D/A.**

| Instante | Dados no SAR | Valor Digital ($V_{AX}$) | Sinal do Host |
|---|---|---|---|
| $t_0$ | 0000 | 0 ($V_{AX} < V_A$) | ALTO |
| $t_1$ | **1**000 | 8 V ($V_{AX} < V_A$) | ALTO |
| $t_2$ | 1**1**00 | 12 V ($V_{AX} > V_A$) | BAIXO |
| $t_3$ | 1**0**00 | 8 V | |
| $t_4$ | 10**1**0 | 10 V ($V_{AX} < V_A$) | ALTO |
| $t_5$ | 101**1** | 11 V ($V_{AX} > V_A$) | BAIXO |
| $t_6$ | 101**0** | 10 V | |

**Fonte: Elaboração própria.**

**Figura 3 - Ilustração passo a passo da operação de um SAC de quatro bits com valor de entrada 10,4 V e um degrau de 1 V.**




**Fonte: TOCCI; WIDMER, 2009, p. 397.**

Inicialmente, os dados do SAR são zerados para que a comparação bit a bit seja iniciada. A partir do bit mais significativo MSB, cada bit é verificado. Nesse exemplo, no primeiro intervalo de tempo, $t_0$, verifica-se que todos os bits estão zerados, e o sinal do comparador é então setado para ALTO, indicando que $V_{AX}$ (o valor contido no SAR) não excedeu $V_A$ (valor de entrada) e que, portanto, o SAR pode continuar com a verificação. Quando $V_{AX}$ excede $V_A$, então o sinal do comparador vai para BAIXO, indicando que é necessário mudar o valor do último bit verificado, de 1 para 0, a fim de atingir o valor mais próximo do valor de entrada. A figura 3 ilustra os passos seguidos até que o último bit seja verificado.

Quando a verificação termina, temos então o valor digital equivalente ao valor de entrada analógico. O valor digital, por sua vez, será sempre menor do que o valor analógico, característica do SAC (TOCCI; WIDMER, 2000, p. 398).

## 2.4 O CONVERSOR AD574A

O conversor AD574A, fabricado pela Analog Devices, foi o dispositivo escolhido para transformar os sinais analógicos em digitais. Ele é um conversor de aproximações sucessivas, apresentando, assim, um tempo fixo de conversão de dados de 35 µs, além de ter uma precisão satisfatória sem perda de dados decorrente da temperatura, por exemplo.

Um dos destaques do AD574A é que os buffers de saída podem conectar-se diretamente ao barramento de dados enquanto comandos de conversão e leitura são recebidos do barramento de controle. Esses e outros aspectos do conversor serão detalhados nesta seção.

### 2.4.1 ASPECTOS GERAIS

O AD574A é um bloco encapsulado de 28 pinos, revestido por um invólucro que pode ser de plástico ou de cerâmica, apresentado sob a forma de um DIP ou um PLCC, e que não necessita de componentes externos para garantir uma execução completa da função de conversão (ANALOG DEVICES, 1999, p. 1, tradução nossa). A partir do sinal de início da conversão, dado pelo bloco de controle, o clock é habilitado e o registrador SAR é resetado. Uma vez que o ciclo de conversão for iniciado, ele não pode ser reiniciado ou interrompido, nem os dados podem ficar disponíveis para os buffers.

A partir daí, o SAR irá realizar todo o processo de incrementação e verificação bit a bit dos dados que irão se tornar o equivalente digital do instante de onda que está sendo convertido, até obter um valor bastante próximo ao analógico, e por fim sinalizar que a conversão acabou. Dado o sinal do fim de conversão, o clock é desabilitado, o status da flag de saída é alterado para BAIXO e as funções de controle são habilitadas, de modo a permitir a execução de funções de leitura através de comandos externos.

“Qualquer carga externa ao AD574A deve permanecer constante durante a conversão”, de acordo com Analog Devices (1999, p. 6, tradução nossa). Para a conversão ser concluída sem que haja problemas com a temperatura do chip, a utilização do diodo Zener (conhecido também como diodo regulador de tensão, diodo de tensão constante, diodo de ruptura ou diodo de condução reversa) garante excelente estabilidade da tensão, tanto em relação ao tempo quanto à temperatura. O diodo Zener atua como um regulador de tensão, ou seja, garante que a tensão permaneça constante independentemente da corrente requisitada pela carga do circuito (SILVA, 2008, p. 1). A figura 4 auxilia a compreensão do seu comportamento.

**Figura 4 – Ilustração da tensão final do sistema, a partir do valor limite do diodo Zener e da tensão inicial.**

$$\text{Se } U_{in} < |V_{zener}|,\ U_{out} = U_{in}$$

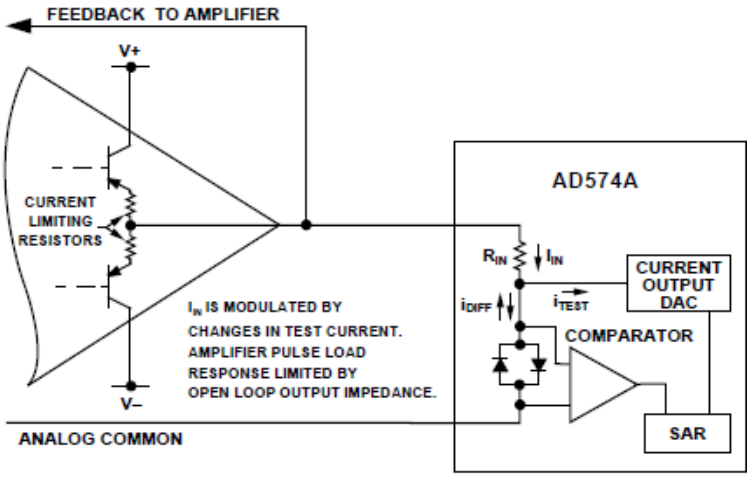
$$\text{Se } U_{in} > |V_{zener}|,\ U_{out} = V_{zener}$$

**Fonte: Elaboração própria.**

Já foi dito que as tensões das correntes que alimentam o AD574A devem permanecer constantes. O porquê disso é justamente impedir que as mudanças abruptas na voltagem da corrente interfiram no processo de conversão e atrapalhe o seu desempenho, diminuindo a precisão garantida nos seus 12 bits. Além do diodo Zener, alguns amplificadores são utilizados para impedir que isso aconteça.

Um desses amplificadores é o amplificador operacional, ou simplesmente amp op, como iremos referenciá-lo neste trabalho. O amp op é um amplificador de alto ganho que usa realimentação para seu próprio controle (LIRA, 2010, p. 1). Circuitos que o utilizam geralmente também utilizam a realimentação negativa, pois seu comportamento, devido ao alto ganho, é quase que totalmente determinado pelos elementos de realimentação. A realimentação negativa tem o objetivo de diminuir a saída do circuito. Desse jeito, o amp op recebe tensões com mudanças bruscas e frequentes, e ameniza-as de forma que estas não interfiram no processo de conversão.

**Figura 5 – Amp Op utilizado no conversor AD574A.**




**Fonte: ANALOG DEVICES, 1999, p. 6. É possível observar a seta na parte superior esquerda saindo dos resistores e retornando para dar o feedback ao amp op.**

Outro amplificador utilizado é o Sample-and-Hold, ou SHA. O SHA é um dispositivo que captura a voltagem de um sinal analógico que varia continuamente e bloqueia seu valor

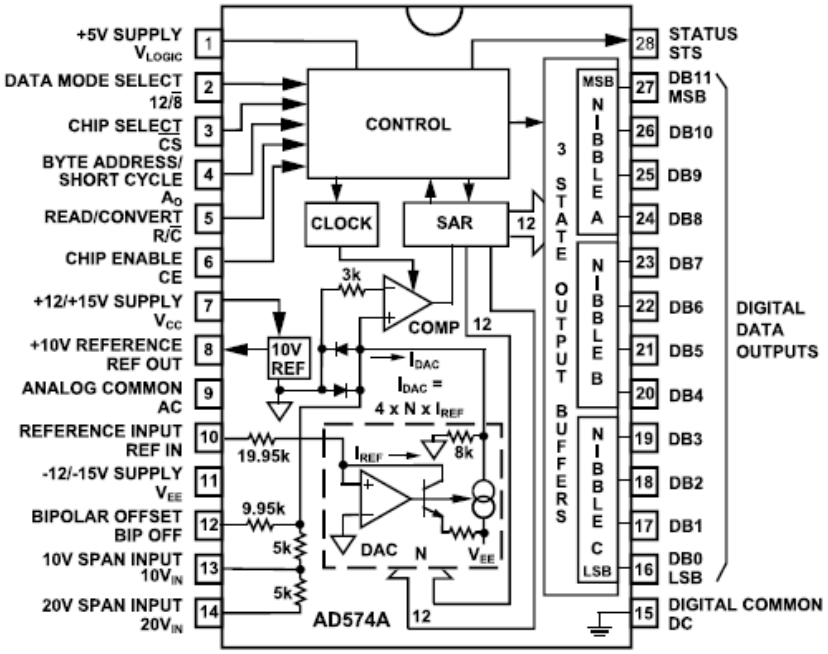
num nível constante por um período de tempo especificado, de acordo com Analog Devices (1999). SHAs são usados em conversores A/D para eliminar variações no sinal de entrada que podem comprometer o processo de conversão e garantir que isso seja feito rapidamente. O AD574A utiliza o AD585, também da Analog Devices, como o seu SHA. A frequência máxima de conversão, que antes era de 1,5 Hz passa a ser de 26 kHz, com a utilização do AD585.

Outro fator que pode prejudicar o sucesso da conversão é o ruído. Dispositivos como os capacitores de desacoplamento (ou capacitores bypass) auxiliam na diminuição do ruído, desviando-o de volta para a fonte de alimentação (CUNHA, 2009) e armazenando a corrente de modo que esta possa ser utilizada pelo circuito quando a demanda num dispositivo mude. Assim, esses capacitores filtram, regulam e diminuem o ruído de alta frequência que acompanha as fontes de alimentação do AD574A.

### 2.4.2 LÓGICA DE CONTROLE

Na seção 2.2 foi descrito o funcionamento do SAC, deixando em aberto apenas a descrição da lógica de controle, pois esta depende da implementação de cada fabricante. No AD574A essa lógica é simples, e isso é decorrente da nomenclatura de sinais comuns a alguns processadores, facilitando a compreensão de suas operações. A figura 6 ilustra o bloco do conversor AD574A, com a configuração dos seus pinos.

**Figura 6 - Diagrama do bloco do AD574A e a configuração dos pinos.**




**Fonte: ANALOG DEVICES, 1999, p. 1.**

Os principais sinais de controle de conversão são o *CE*, $\overline{CS}$ e $R/\overline{C}$. Quando *CE* e $\overline{CS}$ estão setados, ou seja, *CE* = 1 e $\overline{CS}$ = 0, $R/\overline{C}$ é que determina qual operação está em progresso: leitura ($R/\overline{C}$ = 1) ou conversão ($R/\overline{C}$ = 0). A0 e $12/\overline{8}$, sinais de entrada, controlam o formato e o comprimento dos dados, respectivamente. A0 referencia geralmente o LSB do barramento de endereços. Se uma conversão for iniciada com A0 setado como BAIXO, um ciclo completo de conversão de 12 bits é iniciado; do contrário, quando A0 for ALTO, o ciclo de conversão iniciado será de 8 bits.

Durante operações de leitura, A0 determina se os buffers tri-state contendo os 8 MSBs (A0 = 0) ou os 4 LSBs (A0 = 1) do resultado da conversão serão habilitados. O sinal $12/\overline{8}$ determina se os dados de saída serão organizados em duas palavras de 8 bits cada ou numa única palavra de 12 bits. Quando o modo selecionado for o de 8 bits, o byte endereçado quando A0 for ALTO contém os 4 LSBs da conversão, seguidos por quatro zeros. Analog Devices (1999, p. 9), afirma que não é recomendado que A0 mude de estado durante uma operação de leitura, pois sinais de habilita/desabilita em momentos errados podem causar danos ao AD574A.

Outro sinal de saída, o *STS*, indica o estado do conversor. Ele vai para ALTO no início da conversão e retorna para BAIXO quando o ciclo de conversão estiver completo. A tabela 2 representa a tabela verdade do AD574A, ilustrando o comportamento detalhado do conversor.

**Tabela 2 – Tabela verdade do conversor AD574A.**

| CE | $\overline{CS}$ | $R/\overline{C}$ | $12/\overline{8}$ | A0 | Operação |
|---|---|---|---|---|---|
| 0 | X | X | X | X | - |
| X | 1 | X | X | X | - |
| 1 | 0 | 0 | X | 0 | Inicia conversão de 12 bits |
| 1 | 0 | 0 | X | 1 | Inicia conversão de 8 bits |
| 1 | 0 | 1 | $V_{LOGIC}$ | X | Habilita saída paralela de 12 bits |
| 1 | 0 | 1 | Digital Common (*DC*) | 0 | Habilita 8 MSBs |
| 1 | 0 | 1 | Digital Common (*DC*) | 1 | Habilita 4 LSBs + 4 zeros |

**Fonte: ANALOG DEVICES, 1999, p. 8, adaptada.**

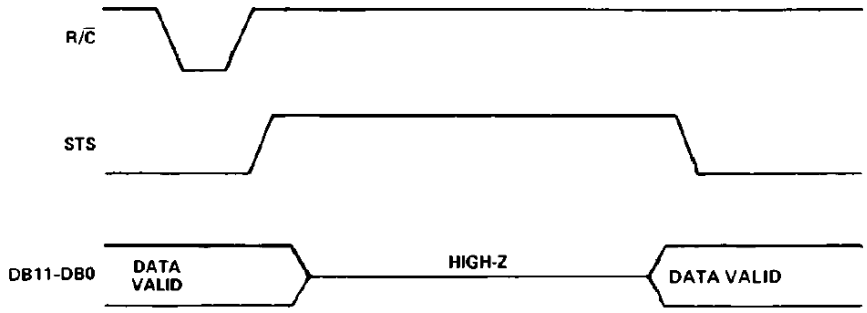
Em operações de conversão, $R/\overline{C}$ deve estar em BAIXO antes de ambos *CE* e $\overline{CS}$ serem setados, do contrário, uma operação de leitura irá ocorrer. *CE* ou $\overline{CS}$ podem ser utilizados para iniciar uma conversão, porém o uso do *CE* é recomendado desde que ele seja bem mais rápido e que apresente um atraso de propagação menor do que o do $\overline{CS}$. Quando uma conversão for iniciada e o *STS* for ALTO, os comandos de início de conversão serão ignorados até que o ciclo de conversão esteja completo.

Durante a leitura, o tempo da operação é medido quando ambos *CE* e $R/\overline{C}$ estiverem em ALTO (desde que $\overline{CS}$ já esteja em BAIXO). Se o $\overline{CS}$ for utilizado para habilitar a operação, o tempo de acesso é estendido em 100 ns.

No modo autônomo (usado principalmente em sistemas com pinos de entrada dedicados), *CE* e *12/*$\overline{8}$ devem estar em ALTO, $\overline{CS}$ e A0 em BAIXO e a conversão deve ser controlada por $R/\overline{C}$. Os buffers tri-state são habilitados quando $R/\overline{C}$ for ALTO e uma conversão é iniciada quando $R/\overline{C}$ for BAIXO. Temos então dois sinais de controle possíveis:

- Pulso baixo (representado na figura 7): o barramento de dados é forçado à alta impedância em resposta ao *falling edge* do $R/\overline{C}$ e retorna para os níveis normais depois que o ciclo de conversão terminar;

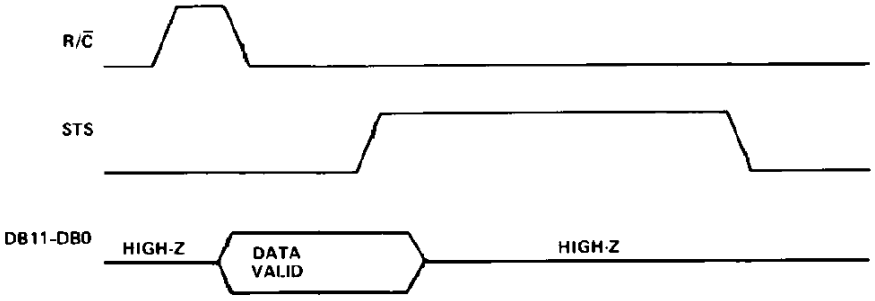
**Figura 7 – Pulso baixo para $R/\overline{C}$ - saídas habilitadas após conversão.**




**Fonte: ANALOG DEVICES, 1999, p. 9, adaptada.**

- Pulso alto (representado na figura 8): o barramento de dados é habilitado quando $R/\overline{C}$ for ALTO. O *falling edge* de $R/\overline{C}$ inicia-se após a próxima conversão, e o barramento de dados retorna à alta impedância (e permanece) até o próximo pulso de $R/\overline{C}$.

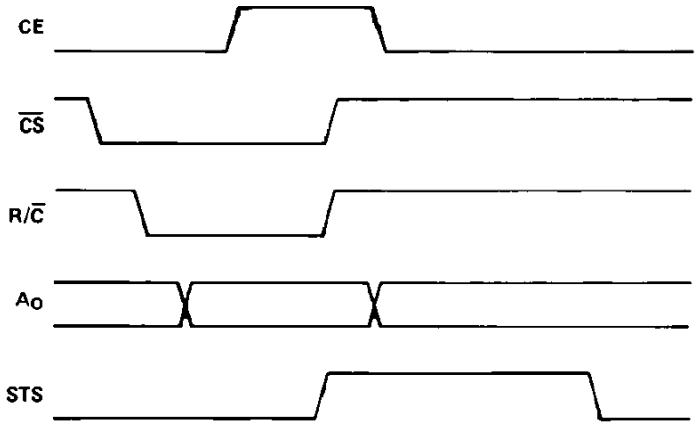
**Figura 8 – Pulso alto para $R/\overline{C}$ - saídas habilitadas enquanto $R/\overline{C}$ for ALTO, caso contrário, alta impedância.**




**Fonte: ANALOG DEVICES, 1999, p. 10, adaptada.**

A figura 9 representa o processo de conversão do AD574A. Pode-se notar que o sinal *CE* é utilizado para dar início à conversão. Percebe-se também que $\overline{CS}$ é setado primeiro, seguido por *R/*$\bar{C}$ e *CE*, mas sabe-se que, mesmo que R/$\bar{C}$ já tenha sido setado, é necessário que $\overline{CS}$ e *CE* também estejam. Como $\overline{CS}$ foi setado primeiro e *R/*$\bar{C}$ logo em seguida, a operação irá começar realmente quando *CE* for setado, havendo um pequeno atraso, até que *STS* sinalize que a conversão está em andamento.

**Figura 9 – Início da conversão.**




**Fonte: ANALOG DEVICES, 1999, p. 9, adaptada.**

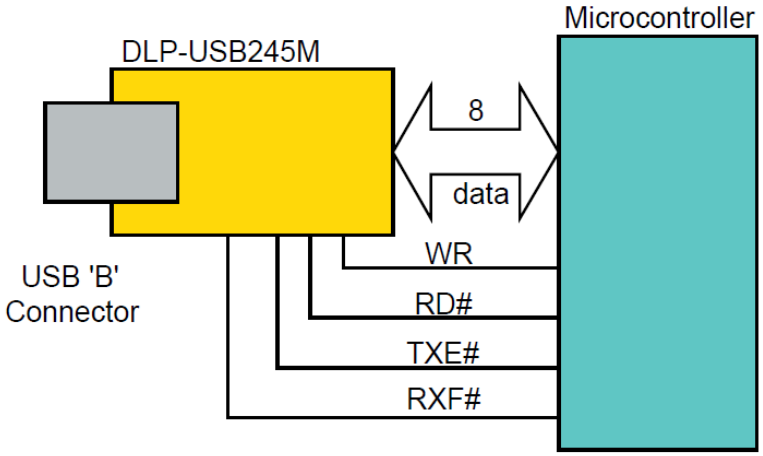
Assim, o AD574A mostra-se um conversor simples, mas que atende às necessidades do projeto, apresentando velocidade, precisão e níveis de confiabilidade satisfatórios.

## 2.5 O DISPOSITIVO DLP-USB245M

O módulo DLP-USB245M, fabricado pela DLP Design, foi escolhido para fazer a comunicação entre o computador (também referenciado aqui como host) e o conversor AD574A. Transferindo dados a uma taxa de 1 Mbps (DLP DESIGN, 2002, p. 2, tradução nossa), o DLP-USB245M é baseado numa FIFO de leitura e escrita.

O dispositivo tem quatro pinos principais: *TXE#*, *RXF#* (pinos de entrada), *WR* e *RD#* (pinos de saída). Através deles, as operações de leitura e escrita podem ser controladas e os dados transferidos do host para o conversor, e vice-versa. Além disso, o dispositivo também apresenta diversas características, como a utilização do dispositivo FTD245BM da FTDI (que elimina a necessidade de memórias para armazenar os dados a ser enviados para o host), um regulador de tensão integrado (de 3,3 V, evitando a utilização de um regulador externo), e a compatibilidade com vários sistemas operacionais.

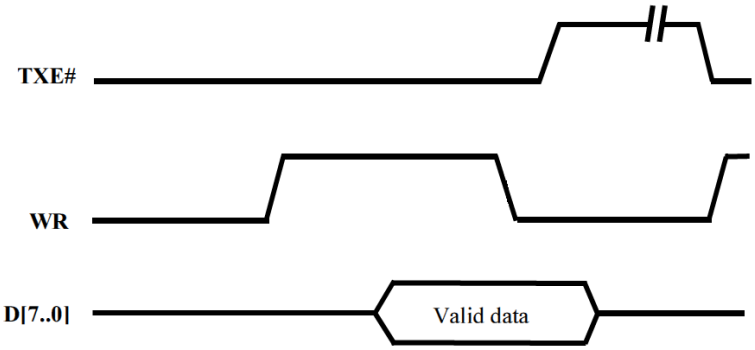
**Figura 10 – Conexão básica do DLP-USB245M a um microcontrolador.**




**Fonte: DLP DESIGN, 2002, p. 8.**

A lógica de controle do DLP-USB245M é simples: para enviar dados do conversor para o host, é necessário escrevê-los no módulo quando *TXE#* for BAIXO. Se o buffer de transmissão estiver cheio ou ocupado com o byte anteriormente escrito, o dispositivo configura *TXE#* como ALTO para impedir que dados sejam escritos até que os dados da FIFO sejam completamente transmitidos do USB para o host. A figura 11 ilustra o processo de escrita de dados.

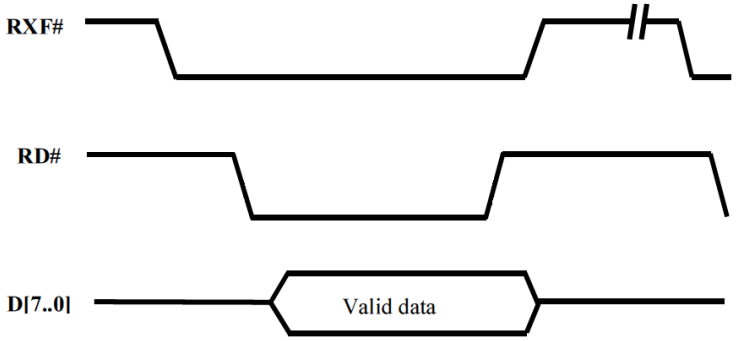
**Figura 11 – Ciclo de escrita da FIFO do DLP-USB245M.**




**Fonte: DLP DESIGN, 2002, p. 10, adaptada.**

Quando o host envia dados para o conversor através do barramento USB, o dispositivo configura *RXF#* como BAIXO para permitir que o conversor saiba que ao menos 1 byte dos dados está disponível. O conversor então lê os dados até que *RXF#* retorne para ALTO, indicando que nenhum dado está disponível para leitura. A figura 12 ilustra o processo de leitura dos dados.

**Figura 12 – Ciclo de leitura da FIFO do DLP-USB245M.**




**Fonte: DLP DESIGN, 2002, p. 10, adaptada.**

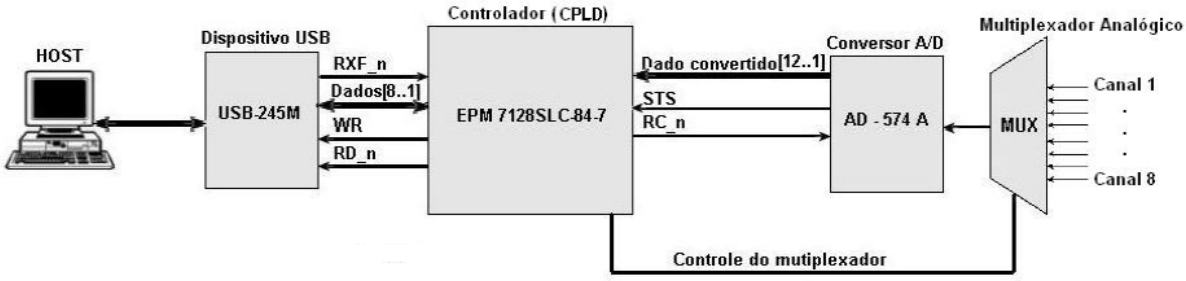
# 3 *DESCRIÇÃO GERAL DO PROJETO*

O sistema consiste principalmente de um grande bloco que acomoda os blocos internos. Nesta seção, cada um deles será detalhado, com ilustrações referentes aos seus processos, máquinas de estados e pinos.

## 3.1 VISÃO GERAL

Inicialmente, antes dos dados serem convertidos, é necessário saber a partir de onde eles foram amostrados, e a frequência com que foram amostrados. Em geral, as aplicações que captam sinais analógicos não são compostas apenas por uma fonte, mas sim por várias (COSTA et al., 2008, p. 4). Então, um multiplexador analógico foi implementado para selecionar os canais de entrada das frequências, suportando até oito canais. A figura 13 ilustra o sistema completo.

**Figura 13 - Arquitetura completa do sistema.**




**Fonte: COSTA et al., 2008, p. 3.**

Ao chegar ao multiplexador, os dados devem seguir por um único canal de frequência, escolhido através do sinal de controle, para depois serem encaminhados ao conversor. Mas, para ter certeza de que a frequência escolhida foi a que mais atende às necessidades do sinal amostrado, é preciso passar pelo contador de frequência, detalhado mais adiante na seção 3.4.

Os quatro sinais de entrada do sistema são:

- *clock_25MHz*, que é o sinal inicial da amostragem dos canais. A partir dele, é feita a divisão de frequências, ajustando o trabalho do multiplexador de acordo com a frequência do canal amostrado;

- *STS*, que se comunica com o bloco do controlador do conversor A/D e com o controlador do FPGA. É responsável por sinalizar quando o valor existente no barramento de saída do conversor é válido, liberando, então, o dado convertido para a leitura;
- *dados_in*, que são os dados, representados inicialmente por uma palavra analógica de 12 bits;
- *RXF_n*, como já foi visto (representado por RXF# na seção 2.4), controla o envio dos dados através do barramento USB, sendo configurado de acordo com a disponibilidade dos dados para leitura.

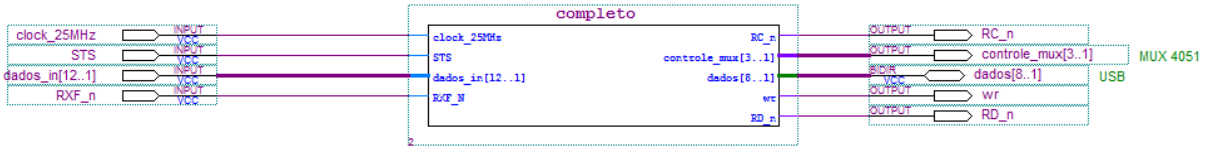
Após a passagem do sinal pelo multiplexador, o contador de frequências, a partir da contagem de pulsos do sinal amostrado, é encarregado de sincronizar as frequências e determinar o início da amostragem, enviando-a para o bloco controlador do AD574A, para dar início ao processo de conversão.

A próxima etapa é a conversão do sinal analógico em digital, feita pelo bloco controlador do conversor A/D. Através dos sinais STS e RC_n, é feito o controle da transmissão dos dados: os sinais são setados e resetados de acordo com a sua disponibilidade. Se a conversão já tiver sido finalizada, o sinal STS é o responsável por transmitir a mensagem ao controlador do FPGA. Mas antes do STS ser setado, é preciso haver o reset do RC_n, responsável transmitir ao controlador do conversor que a operação de leitura é desejada. Assim, o STS atua depois da conversão ter sido feita, que é desencadeada pelo requerimento de uma operação de leitura.

O caminho a seguir é o do circuito controlador do USB. A palavra de 12 bits, já convertida, é enviada para o bloco GERA_PULSO_WR, que basicamente é encarregado de dividi-la em duas palavras de 8 bits. Seu funcionamento é detalhado mais adiante.

Depois do gerador de pulso, os dados passam ainda pelo tri-state, que, através do sinal *habilita* podem ser transmitidos ou não ao controlador do USB. Esse controle é feito pelo próprio controlador do USB, que decidirá quando estará pronto para receber os dados. Uma representação gráfica dos blocos internos do sistema completo pode ser vista no anexo A.

**Figura 14 – Ilustração do bloco principal do sistema conversor.**




**Fonte: Elaboração própria.**

## 3.2 FUNCIONAMENTO DO GERA_CLOCK

O bloco GERA_CLOCK é basicamente um divisor de frequências. Ele recebe como entrada um pulso de 20 MHz, mas que pode ser modificado para receber pulsos de até 25 MHz de frequência, que é o valor máximo com o qual o conversor escolhido, o AD574A da Analog Devices, trabalha (COSTA et al., 2008, p. 4). Dentro desse bloco, o pulso de entrada é dividido em pulsos de 1MHz, 100KHz, 10KHz e 1KHz, que são enviados para o multiplexador.

O GERA_CLOCK é um divisor de frequências adaptado para trabalhar com o sinal de entrada de 20MHz. Quando o sinal é alimentado, ou seja, quando é enviado um pulso para o bloco, seu algoritmo entra em funcionamento, como descreve a figura 15. A partir do sinal de 20MHz, o bloco divide-o por 20 para resultar num sinal de 1MHz, que é a primeira saída. De modo similar, o sinal de 1MHz é dividido por 10 para gerar o sinal de 100KHz, e assim por diante para os sinais de 10KHz e 1KHz. Cada divisão é um processo distinto, com sua própria saída.

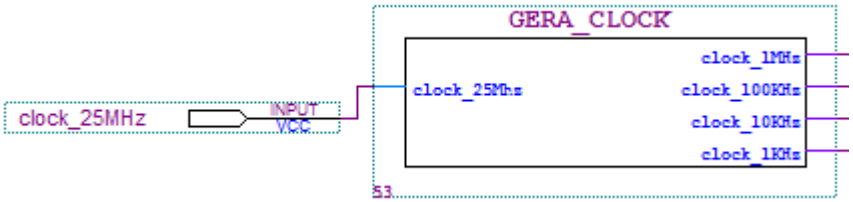
**Figura 15 – Trecho de código em VHDL descrevendo o que acontece quando há um pulso do clock.**

```
WAIT UNTIL clock_25Mhz'EVENT and clock_25Mhz = '1';
    IF count_1Mhz < 19 THEN
        count_1Mhz <= count_1Mhz + 1;
    ELSE
        count_1Mhz <= "00000";
    END IF;
```

**Fonte: Elaboração própria.**

Durante os processos, são alocados sinais para armazenar a quantidade de pulsos dos sinais dos clocks, e assim, não interferir nas variáveis iniciais, correspondentes às saídas do bloco. Esses sinais são constantemente atualizados e seus valores, atribuídos às variáveis de saída.

**Figura 16 – Ilustração do bloco GERA_CLOCK.**



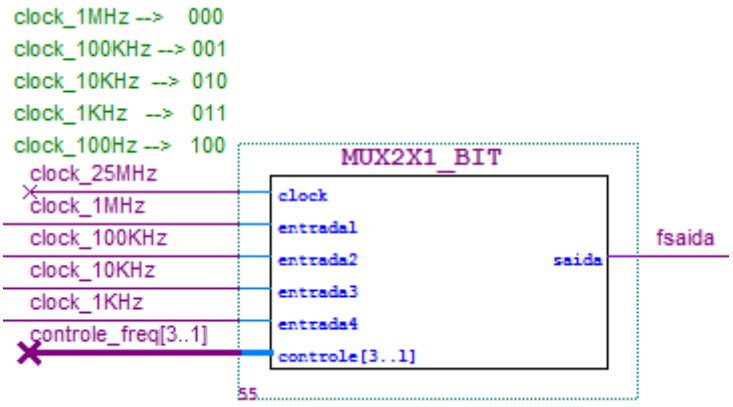

**Fonte: Elaboração própria.**

## 3.3 FUNCIONAMENTO DO MUX2X1_BIT

Os multiplexadores são estruturas que recebem como entrada um determinado número de sinais e retornam como saída apenas uma das entradas. Isso é feito por meio de sinais de controle, que determinam qual sinal deverá ser o escolhido para ser transmitido mais adiante. Seu mecanismo é simples: para N sinais de controle, há no máximo $2^N$ entradas; portanto, para um multiplexador de quatro entradas, são necessários dois sinais de controle (ou um vetor de dois bits). No caso do projeto em questão, há um vetor de três bits como sinal de controle. Dessa maneira, a possibilidade de trabalhar com maiores frequências está aberta, suportando até oito sinais de entrada.

O sinal inicial do clock de 20MHz, que é um sinal de entrada para o GERA_CLOCK, também é um sinal de entrada para o multiplexador. Isso é determinado para que o sinal do clock, quando ativado (ou seja, quando passa pelo *rising_edge*), permite que o multiplexador exerça sua função, baseado na entrada do sinal de controle. Já o sinal de controle vem do circuito controlador do USB, que será explicado detalhadamente mais adiante.

**Figura 17 – Ilustração do bloco MUX2X1_BIT.**




**Fonte: Elaboração própria.**

## 3.4 FUNCIONAMENTO DO CONTADOR_FREQUENCIA

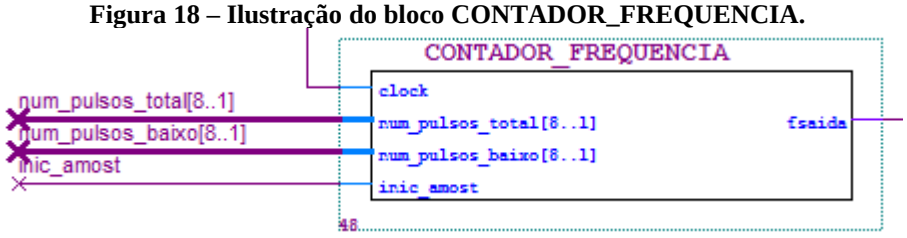
O contador de frequência é responsável, como o próprio nome do bloco já sugere, pela contagem da frequência do clock, que servirá de entrada para o controlador do conversor A/D. O contador de frequência recebe como parâmetros o clock, que é a saída do multiplexador, dois vetores contendo, respectivamente, o número total de pulsos e o número de pulsos baixos (*falling edge*), provenientes do controlador do USB, e o parâmetro que determina o momento em que será iniciada a amostragem do sinal, que também é proveniente do circuito controlador do USB.

A partir do sinal de início da amostragem, o contador de frequência segue alguns caminhos:

- Se o início da amostragem for resetado, ou seja, *inic_amost* = 0, não haverá amostragem e, portanto, o contador é zerado;
- Se o início da amostragem for setado, ou seja, *inic_amost* = 1, então há três caminhos a seguir, sempre considerando o *rising_edge* do clock de entrada:
  - Se *contador* < *num_pulsos_baixo*, então a saída será 0 (zero) e o contador será incrementado;
  - Se *contador* >= *num_pulsos_baixo* e *contador* < *num_pulsos_total*, então a saída será 1 (um) e o contador será incrementado;
  - Se *contador* = *num_pulsos_total*, então o contador será zerado.

Ou seja, enquanto o número de pulsos baixos não for atingido pelo contador, a saída do contador de frequências é 0 (zero). Essa saída é encaminhada então para o conversor, como o sinal *f_amost*, que trata a geração do pulso de amostragem, que auxiliará o processo de conversão, a ser visto adiante.

**Figura 18 – Ilustração do bloco CONTADOR_FREQUENCIA.**

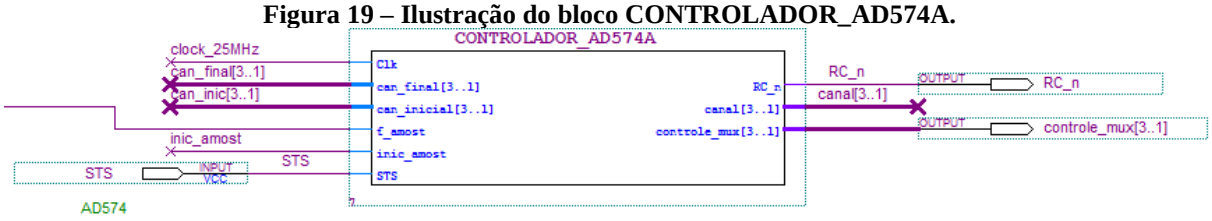



**Fonte: Elaboração própria.**

## 3.5 FUNCIONAMENTO DO CONTROLADOR_AD574A

A função principal do circuito controlador do conversor é gerar os sinais básicos de interface com o conversor analógico digital AD574A. O bloco tem seis pinos de entrada, dois de saída e um buffer. Os pinos de entrada são o clock, a frequência de amostragem (que é a saída do contador de frequência), o sinal de início da amostragem (que é o mesmo pino de entrada do contador de frequência, proveniente do controlador do USB), dois vetores de três bits cada, o *can_final* e o *can_inicial* (que irão originar o sinal do controle do multiplexador, que é um pino de saída) e o *STS*, que é uma entrada que alimenta tanto o controlador do conversor quando o controlador do USB, e é responsável por notificar quando o valor que está no barramento de saída é válido para leitura. Podemos observar, na figura 19, os pinos de entrada e saída do bloco.

**Figura 19 – Ilustração do bloco CONTROLADOR_AD574A.**




**Fonte: Elaboração própria.**

O controlador do conversor opera através de vários processos concorrentes, descritos a seguir:

- Processo para a geração do pulso de amostragem: a partir do *rising edge* do clock, controla o pulso da amostragem com o auxílio de uma flag, que, por sua vez, controla o funcionamento do sinal *conv_n_aux*, atuante na mudança de estados do conversor, como ilustrado na tabela 4, exposta mais adiante;
- Processo responsável pela mudança de estados do multiplexador: é onde a conversão ocorre. Enquanto o início da amostragem não é habilitado, a máquina fica em estado de espera até o sinal ser habilitado. Quando isso ocorre, há ainda dois estados responsáveis pela conversão. O primeiro estado de conversão é atingido quando a operação de leitura é requisitada, ou seja, quando o auxiliar do *RC_n* é zerado, assim, o estado atual passa do inicial para o primeiro estado de conversão (*conversao1*). Nesse estado há duas possibilidades: ir para o segundo estado de conversão (*conversao2*), que ocorre quando o auxiliar do *RC_n* é setado como BAIXO (*RC_n* = 0), ou seja, quando a operação de leitura é interrompida e quando o valor no barramento ainda não

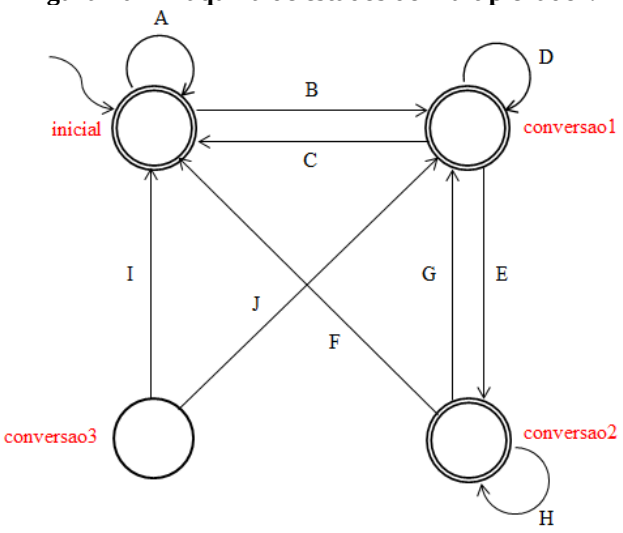
está pronto para ser lido (*STS* zerado); ou permanecer no estado *conversao1*, quando ocorre uma situação diferente da anterior. Uma vez que o estado atual passa a ser o *conversao2*, ele pode retornar ao *conversao1* se o auxiliar do *RC_n* for zerado, ou permanecer no *conversao2* se ocorrer outras situações. Se a qualquer momento o sinal de início da amostragem for zerado, o estado atual passa a ser o inicial, esperando ser setado novamente. Esse processo é ilustrado pela tabela 3 e pela máquina de estados na figura 20;

**Tabela 3 – Descrição dos estados durante o processo de mudança de estados do multiplexador.**

| Estado Atual | Próximo Estado | Condição | Indicador |
|---|---|---|---|
| inicial | inicial | (!inic_amost)+(inic_amost).(RC_n_int) | A |
| inicial | conversao1 | (inic_amost).(!RC_n_int) | B |
| conversao1 | inicial | (!inic_amost) | C |
| conversao1 | conversao1 | (inic_amost).(!process_1) | D |
| conversao1 | conversao2 | (inic_amost).(process_1[2]) | E |
| conversao2 | inicial | (!inic_amost) | F |
| conversao2 | conversao1 | (inic_amost).(!RC_n_int) | G |
| conversao2 | conversao2 | (inic_amost).(RC_n_int) | H |
| conversao3 | inicial | (!inic_amost) | I |
| conversao3 | conversao1 | (inic_amost) | J |

**Fonte: Elaboração própria.**

**Figura 20 – Máquina de estados do multiplexador.**



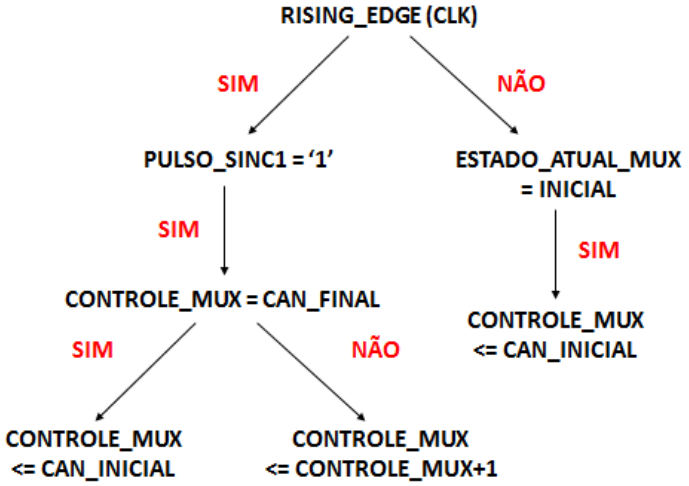

**Fonte: Elaboração própria.**

- Processo para a geração do sinal *pulso_sinc*: sincroniza os pulsos do clock com os estados do multiplexador, mais especificamente no *conversao1*;

[2] O *process_1* significa (RC_n_int = 1 and STS = 0), ou seja, quando ele for satisfeito, o estado mudará para o *conversao2*.

- Processo para a geração do sinal *pulso_sinc1*: similar ao processo anterior, sincroniza o clock com os estados do multiplexador, só que no estado *conversao2*;
- Processo para a geração do sinal *canal*: aqui é feita apenas a atribuição do valor contido em *controle_mux* para *canal*, a partir do *rising_edge* do *STS*, com o objetivo de eliminar leituras incorretas de canal;
- Processo para a geração do sinal *controle_mux*: o controle do mux é similar a um contador: enquanto seu valor não for igual ao do canal final, ele vai sendo incrementado até atingi-lo. Feito isso, o valor do canal inicial é atribuído ao controle do mux e começa a contagem novamente, exceto quando o estado atual do mux for o inicial, quando o valor do canal inicial é atribuído ao controle do mux, mesmo que este não seja igual ao valor do canal final. Ou seja, enquanto o dado está sendo convertido, o controle do mux está sendo incrementado. Esse processo é ilustrado na figura 21;

**Figura 21 - Esquema do funcionamento do processo para a geração do sinal *controle_mux*.**




**Fonte: Elaboração própria.**

- Processo responsável pela mudança de estados do conversor: aqui é ilustrada a mudança de estados do controlador do conversor. Se o sinal de conversão não estiver setado, então o processo entrará em espera até o valor ser alterado para 1 (um). Feito isso, o estado atual será modificado para *converte1*, que permanecerá nele mesmo, caso o sinal *STS* não tiver sido habilitado, ou irá para o estado *converte2*, caso o *STS* tiver sido habilitado. O estado atual

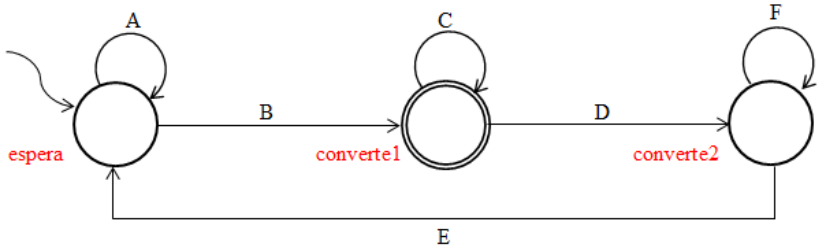
permanecerá como *converte2* até que o *STS* seja resetado. A máquina de estados deste processo está ilustrada na figura 22;

**Tabela 4 – Descrição dos estados durante o processo de mudança de estados do conversor.**

| Estado Atual | Próximo Estado | Condição | Indicador |
|---|---|---|---|
| espera | espera | (!conv_n_aux) | A |
| espera | converte1 | (conv_n_aux) | B |
| converte1 | converte1 | (!STS) | C |
| converte1 | converte2 | (STS) | D |
| converte2 | espera | (!STS) | E |
| converte2 | converte2 | (STS) | F |

**Fonte: Elaboração própria.**

**Figura 22 – Máquina de estados do conversor.**



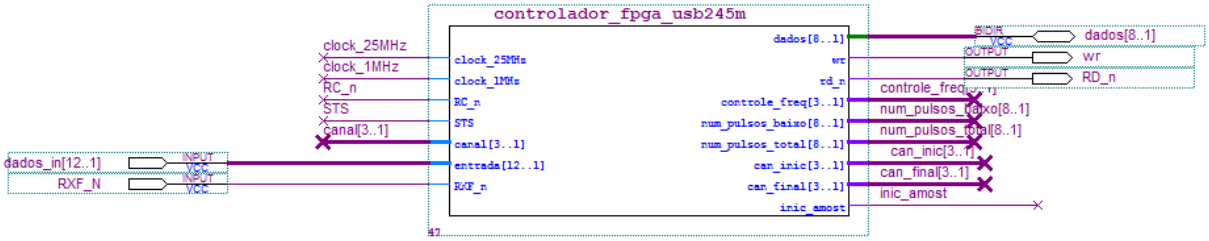

**Fonte: Elaboração própria.**

- Processo para a geração dos sinais de saída: aqui é determinada a saída do sinal *RC_n* (e o seu auxiliar), fazendo com que a operação de leitura (*RC_n* = 1) só seja requisitada quando o estado atual do conversor estiver no estado de espera, ou *converte2*. Quando o estado for o *converte1* e o início da amostragem tiver sido habilitado, a conversão será habilitada; caso contrário, a leitura será habilitada.

## 3.6 FUNCIONAMENTO DO GERA_PULSO_WR

O bloco GERA_PULSO_WR é o primeiro do circuito controlador do USB do sistema. O bloco maior CONTROLADOR_FPGA_USB245M resume o circuito que compreende três blocos, responsáveis pelo controle do USB e sua interface entre os sinais de conversão e captação dos dados convertidos. A figura a seguir ilustra o circuito controlador do USB, enquanto que no anexo B podemos ver os blocos internos ao CONTROLADOR_FPGA_USB245M.

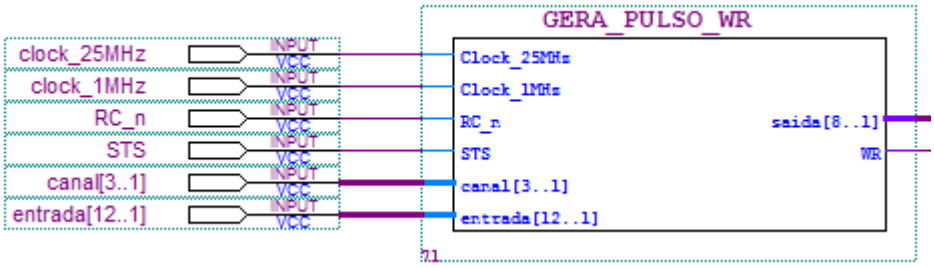
**Figura 23 – Ilustração do bloco CONTROLADOR_FPGA_USB245M.**




**Fonte: Elaboração própria.**

O bloco GERA_PULSO_WR é responsável por gerar os pulsos *WR* para o dispositivo DLP-USB245M, como o próprio nome já sugere. Através de um contador de 4 bits e uma máquina de estados, é feito o controle das operações de leitura e escrita, da liberação dos dados convertidos, feita pelo sinal *habilita*, e da transformação de cada palavra de 12 bits em duas de 8 bits.

**Figura 24 - Ilustração do bloco GERA_PULSO_WR.**




**Fonte: Elaboração própria.**

Cada uma destas funções é descrita a través de processos concorrentes, como em todo o projeto. A seguir há a breve descrição de cada um deles:

- Processo responsável pela geração do sinal *WR*: a principal função desse processo é sinalizar a flag, que servirá de parâmetro para a mudança de estados do sinal *habilita*. Se o sinal *habilita* estiver zerado, então a flag e o *contador* também serão zerados. Mas se o sinal *habilita* estiver setado como 1 (um), há dois casos a serem considerados: se o valor do *contador* for 1000, a flag é setada; se o valor do *contador* for diferente de 1000, então ele é incrementado até atingir esse valor. Quando o sinal *habilita* estiver zerado, a flag e o contador também são zerados;
- Processo responsável pela mudança de estados para a geração do sinal *habilita:* é responsável pelo controle dos sinais, que sinalizam que o dado está pronto para ser transmitido, depois de ter sido convertido. A flag, antes configurada no

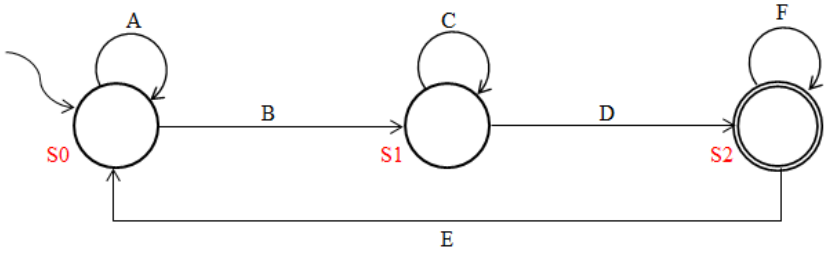
processo anterior, sinaliza quando um determinado valor do contador é atingido, influenciando, assim, a troca de estados do processo atual. Esse processo é ilustrado através da tabela 5 e da figura 25;

**Tabela 5 – Descrição dos estados do processo de mudança de estados para a geração do sinal *habilita*.**

| Estado Atual | Próximo Estado | Condição | Indicador |
|---|---|---|---|
| S0 | S0 | (!RC_n)+(RC_n).(!STS)+(RC_n).(STS).(flag) | A |
| S0 | S1 | (RC_n).(STS).(!flag) | B |
| S1 | S1 | (!RC_n)+(RC_n).(!STS).(flag)+(RC_n).(STS) | C |
| S1 | S2 | (RC_n).(!STS).(!flag) | D |
| S2 | S0 | (!STS).(flag) | E |
| S2 | S2 | (!STS).(!flag)+(STS) | F |

**Fonte: Elaboração própria.**

**Figura 25 - Máquina de estados para a geração do sinal habilita.**



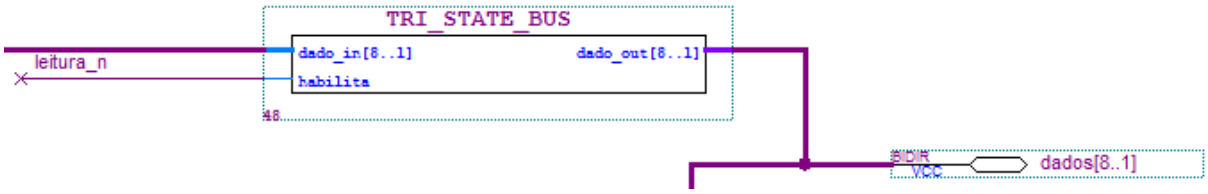

**Fonte: Elaboração própria.**

- Processo responsável pela geração do sinal *habilita*: apenas habilita a transmissão do dado convertido, ou seja, quando o estado atual for *S2*;
- Processo responsável pelo controle da palavra enviada ao dispositivo USB: encarrega-se de dividir a palavra de 12 bits em duas palavras de 8 bits cada. O contador de 4 bits é continuamente verificado e, até seu valor atingir 0100, a saída será 11111111 e o sinal *WR* é zerado. A partir do valor 0101, a divisão das palavras é iniciada. A primeira palavra é composta pela concatenação 0 (zero) + os três bits do *canal* + os quatro MSBs dos dados de entrada. O sinal *WR* recebe o valor do último bit do sinal *contador*. A segunda palavra é composta pelos oito bits restantes dos dados de entrada. Desse modo, a operação de divisão de palavras está concluída.

## 3.7 FUNCIONAMENTO DO TRI_STATE_BUS

O bloco tri-state é responsável por habilitar/desabilitar a transmissão dos dados provenientes do bloco gerador de pulso. Através de um pino de controle, o sinal *habilita*, o controlador do USB define quando os dados estão prontos para serem transmitidos adiante, habilitando a sua passagem quando o valor do sinal for 1 (um).

**Figura 26 - Ilustração do bloco TRI_STATE_BUS.**




**Fonte: Elaboração própria.**

## 3.8 FUNCIONAMENTO DO CONTROLADOR_USB245M_ALTERA_RECEPCAO

O último bloco do sistema, e também do circuito controlador do USB, é responsável por gerar os sinais básicos de recepção do dispositivo DLP-USB245M. A geração desses sinais é feita com o auxílio de uma máquina de estados responsável pela recepção dos dados e de um processo gerador dos sinais de saída do bloco, que servirão de entrada para os demais blocos do sistema, como o sinal de início da amostragem, por exemplo, que é uma entrada para os blocos contador de frequência e controlador do conversor.

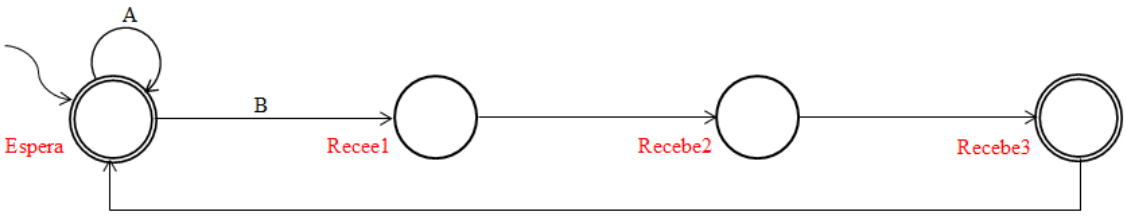
A máquina de estados é composta por quatro estados: um de espera, quando os dados ainda não estão prontos para serem recebidos, sinalizados por *RXF_n* (os dados estão prontos para serem recebidos no bloco se *RXF_n* = 0, como visto anteriormente na seção 2.4); e três de recepção. A máquina é ilustrada na figura 27.

**Tabela 6 – Descrição dos estados para a geração dos sinais de recepção do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO.**

| Estado Atual | Próximo Estado | Condição | Indicador |
|---|---|---|---|
| Espera | Espera | (RXF_n) | A |
| Espera | Recebe1 | (!RXF_n) | B |
| Recebe1 | Recebe2 | - | - |
| Recebe2 | Recebe3 | - | - |
| Recebe3 | Espera | - | - |

**Fonte: Elaboração própria.**

**Figura 27 - Máquina de estados para a geração de sinais de recepção do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO.**



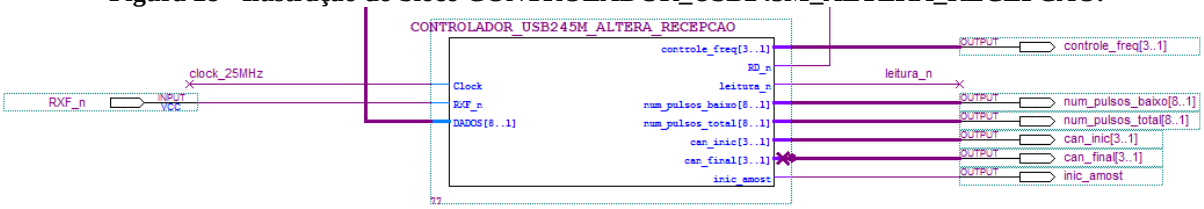

**Fonte: Elaboração própria.**

O processo responsável pela geração dos sinais de saída do bloco trabalha em cima da máquina de estados descrita anteriormente. Enquanto estiver em espera, é atribuído 1 (um) ao

sinal *RD_n* (que é um dos sinais de saída do sistema completo, como visto na figura 14, na seção 3.1) e verifica-se se os dados estão prontos para serem recebidos (*RXF_n* = 0): se sim, o sinal de leitura (*leitura_n*) é zerado, permitindo a leitura (e sendo enviado como uma entrada do bloco tri-state); se não, *leitura_n* é setado como 1. Quando o estado atual for o *Recebe1* ou o *Recebe2*, a leitura será permitida e o sinal *RD_n* será zerado. Quando o estado atual for o *Recebe3*, acontece a geração dos sinais *num_pulsos_baixo* e *num_pulsos_total* (que servirão de entrada para o contador de frequência), *can_inic* e *can_final* (que servirão de entrada para o controlador do conversor), o controle de frequência (que é o sinal de controle do multiplexador), e o sinal de início da amostragem. Assim, o controlador do USB define os

sinais de controle dos blocos do sistema.

**Figura 28 - Ilustração do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO.**




**Fonte: Elaboração própria.**

## 3.9 RESULTADOS OBTIDOS

Utilizando o Quartus II, um uma ferramenta de software da Altera que auxilia o programador na análise e síntese de projetos HDL, o projeto foi compilado e simulado para ser implementado num chip da Altera, o EPM7128SLC84-7, da família MAX7000S.

A partir da compilação e execução do sistema, pudemos ver os resultados da simulação, que podem ser observados no anexo C no final deste trabalho. Entre 0 e pouco mais de 1.760 microssegundos, aproximadamente, podemos observar o comportamento dos pinos de entrada e saída do sistema, os sinais de leitura, escrita e conversão de dados. No relatório de fluxo (Flow Summary), 111 macrocélulas das 128 típicas da família MAX7000 foram usadas, assim como 33 dos 68 pinos.

Analisando o relatório do tempo (Timing Analyzer Summary), podemos observar as seguintes estimativas:

- pior caso do tempo de configuração do clock de um pino de entrada para o pino de saída (Clock Setup Time, $t_{su}$): 6 ns. Esse caminho começa no pino

*RXF_n* e termina no pino *leitura_n*, do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO;

- pior caso do atraso do tempo decorrido da saída de um sinal de clock de um pino de entrada para um pino de saída através de um registrador (ou Clock-to-Output Time, $t_{co}$): 9,5 ns. Esse caminho começa no pino *leitura_n*, do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO, e termina no pino *dados[1]*;
- pior caso do tempo de pausa do clock (Clock Hold Time, $t_h$): -1 ns. Esse caminho começa no pino *RXF_n* e termina no pino *leitura_n*, do bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO.

Considerando os resultados obtidos, o sistema teve um desempenho suficientemente bom para o que se propõe a fazer, ou seja, a conversão analógico-digital de 12 bits. Para projetos futuros, pode-se cogitar aumentar seu poder de receber dados, aumentando o número de canais e o tempo para a captação destes.

# 4 *CONCLUSÃO*

Um dos principais fatores para o sucesso de um sistema é a sua facilidade de manutenção. Essa característica pode ser obtida de várias formas: com texto claro e objetivo, com documentação apropriadamente escrita, com o emprego de técnicas simples na sua implementação, sem deixar de lado a eficiência. Mas geralmente não há uma boa manutenibilidade quando esses fatores acima descritos estão em desacordo um com o outro.

Os desenvolvedores de um projeto ainda dão pouca importância à documentação de um sistema, por ela ser mais teórica do que o resto do projeto. É a etapa em que se deve basicamente relatar o que foi feito e as técnicas que foram empregadas, em termos simples, suficientes para que quem for fazer a manutenção do sistema no futuro possa compreendê-lo sem maiores esforços. É ainda desvalorizada por muitos programadores, fazendo com que muitos projetos possuam documentações precárias ou mesmo nenhuma. Mas, como visto ao decorrer do curso de Ciência da Computação, é uma etapa importantíssima, até para o cliente/usuário, abrangendo desde conceitos básicos até os mais técnicos.

A apresentação de conceitos e técnicas utilizadas neste trabalho de modo simples e claro fez com a descrição tomasse a forma de um texto didático, moldado para atender às necessidades de estudantes, de desenvolvedores, de gerentes de projeto, de pesquisadores, clientes e usuários. Com o auxílio de esquemas, gráficos, tabelas, máquinas de estados e demais ilustrações, a descrição do sistema ficou mais rica e fácil de entender.

Para trabalhos futuros, como já dito, documentações são sempre bem vindas, ainda mais quando são bem escritas, especificando componentes, linguagens, termos empregados, e até definições inicialmente desnecessárias, mas que se tornam relevantes quando uma ou outra dúvida surge. Sendo assim, pode até vir a tornar-se uma espécie de guia para trabalhos correlatos ou que empreguem algum elemento similar utilizado no sistema que precise de mais detalhes.

O sistema no seu modo atual consegue captar frequências de até oito canais diferentes, como citado na seção 3.1. Contudo, essa captação ocorre de maneira individual, ou seja, cada

sinal é captado, convertido e processado inteiramente para somente depois que todas essas operações forem feitas, começar a captação de outro sinal. Para trabalhos futuros, seria uma proposta interessante a captação de todos os canais de uma só vez, reduzindo assim, o tempo de atuação do sistema, processando mais dados em intervalos menores de tempo.

A descrição de um projeto, em qualquer área, é um exercício gratificante, pois soma experiência e conhecimento acerca de dispositivos utilizados para determinados fins. A aprendizagem é ampla, e ajuda-nos a compreender também a evolução das tecnologias, avaliando quais foram os marcos do passado, discutindo os motivos que levaram aos resultados obtidos. Além disso, força-nos também a olhar adiante, fazendo sempre pequenas "apostas" para tendências que possam vir a surgir com o advento de novos métodos e materiais.

# REFERÊNCIAS


ANALOG DEVICES. **Complete 12-Bit A/D Converter**. [S.I.], 1999. Disponível em: <http://www.analog.com/static/imported-files/data_sheets/AD574A.pdf>. Acesso em: 26 nov. 2012.

BUCHHOLZ, Werner (Ed.). **Planning a Computer System – Project Stretch.** New York: McGraw-Hill Book Company, 1962.

CARDOSO, Fabbryccio A. C. M.; FERNANDES, Marcelo Augusto Costa. **Aula 5: FPGA e Fluxo de Projeto.** [S. I.], 2007. Disponível em: <http://www.decom.fee.unicamp.br/~cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf>. Acesso em: 9 abr. 2013.

COSTA, Yuri Gonzaga Gonçalves da et al. **Sistema de Conversão Analógico Digital de 12 Bits**. Anais do XI Encontro de Iniciação à Docência. João Pessoa, PB: UFPB, 2008. Disponível em: <http://www.prac.ufpb.br/anais/xenex_xienid/xi_enid/monitoriapet/ANAIS/Area8/8CCENDIPET01.pdf>. Acesso em: 10 out. 2012.

CUNHA, Roberto. Eletrônica analógica: capacitores de desacoplamento. **Saber Eletrônica**, [S.I.], 29 jan. 2009. Disponível em: <http://www.sabereletronica.com.br/secoes/leitura/1147>. Acesso em: 7 dez. 2012.

D'AMORE, Roberto. **VHDL – Descrição e Síntese de Circuitos Digitais**. 1. ed. Rio de Janeiro: LTC, 2005.

DLP DESIGN. **DLP-USB245M User Manual**. [S.I.], 2002. Disponível em: <http://www.dlpdesign.com/usb/dlp-usb245mv15.pdf>. Acesso em: 26 nov. 2012.

FERREIRA, Eduardo dos Santos et al. Descrição e Síntese de Conversores A/D e D/A PWM. **Boletim Técnico da FATEC-SP**, São Paulo, n. 14, 2003. Disponível em: <http://bt.fatecsp.br/system/articles/42/original/2eduardo.pdf>. Acesso em: 25 fev. 2013.

LIRA, Jose Gutembergue de Assis. **Capítulo 7 – Amplificador Operacional**. [S.I.], 2010. Disponível em: <http://www.dee.ufcg.edu.br/~gutemb/AmpOp.pdf>. Acesso em: 5 dez. 2012.

MURDOCCA, Miles; HEURING, Vincent. **Introdução à Arquitetura de Computadores – Capítulo 1: Introdução**. [S.I.], 1999. Disponível em: <http://www.gta.ufrj.br/ensino/EEL580/apresentacoes/Parte1.pdf>. Acesso em: 24 fev. 2013.

SATO, Gilson Yukio. **Eletrônica Digital – Conversores A/D**. [S.I.], [2010?]. Disponível em: <http://pessoal.utfpr.edu.br/sato/arquivos/gs_0905_conv_ad_v0.pdf>. Acesso em: 19 dez. 2012.

SILVA, Anderson Gomes da. **Eletrônica – Diodo 02**. [S.I.], 2008. Disponível em: <http://www.cin.ufpe.br/~ags/eletr%f4nica/aula_03.pdf>. Acesso em: 13 dez. 2012.

TOCCI, Ronald J.; WIDMER, Neal S. **Sistemas Digitais – Princípios e Aplicações**. 7. ed. Rio de Janeiro: LTC, 2000.

# OBRAS CONSULTADAS

BAROT, Nila. **Successive Approximation Analog to Digital Converter**. San José, Califórnia, EUA: San José State University: 2010. Disponível em: <http://www.engr.sjsu.edu/ges/media/pdf/mse_prj_rpts/spring2010/Successive%20Approximation%20Analog%20to%20Digital%20Converter.pdf>. Acesso em: 20 dez. 2012.

BARROS, Pedro; DIAS, Ricardo. **Kit de Instrumentação Virtual**. Covilhã, Portugal: Universidade da Beira Interior, 2006. Disponível em: <http://webx.ubi.pt/~a13852/projecto/>. Acesso em: 27 dez. 2012.

CARVALHO, Carmen Maria Costa de. [S.I.], 2002. **Apostila para Disciplinas de Técnicas Digitais I & Circuitos Digitais.** Disponível em: <http://www.telecom.uff.br/tecdig1/arquivos/apostilas/apostila_td1_parte2.pdf>. Acesso em: 25 fev. 2013.

DIAS, Anfranserai Morais. [S.I.], 2004. **2 Arquitetura de Computadores.** Disponível em: <http://www.dca.ufrn.br/~xamd/dca0800/Cap02.pdf>. Acesso em: 20 mar. 2013.

LIMA, José Antônio Gomes de. [S. I.], [200-]. **Aula 16.** Disponível em: <http://www.di.ufpb.br/jose/aula16.ppt>. Acesso em: 14 abr. 2013.

LIMA JR, Herman P. **G15 – Eletrônica Digital para Instrumentação**. Rio de Janeiro: 2012. Disponível em: <http://mesonpi.cat.cbpf.br/e2012/arquivos/g15/G15-Aula2_VHDL.pdf>. Acesso em: 20 dez. 2012.

MARTINO, João Antonio. [S.I.], 2006. **Por Dentro do Circuito Integrado.** Disponível em: <http://www.lps.usp.br/lps/arquivos/conteudo/grad/dwnld/integrado.pdf>. Acesso em: 25 fev. 2013.

MARTINS, Carlos Augusto Paiva da Silva et al. **Computação reconfigurável: conceitos, tendências e aplicações.** [S. I.], [2003?]. Disponível em: <http://www.ppgee.pucminas.br/gsdc/papers/martins_eri02.pdf>. Acesso em: 9 abr. 2013.

NETO, José Monserrat. **Capítulo 3 – Linguagens de Programação.** [S.I.], 2005. Disponível em: <http://algol.dcc.ufla.br/~monserrat/icc/Capitulo3.html>. Acesso em: 25 fev. 2013.

SAITO, José Hiroki. **VHDL – Aula 2**. [S.I.], 2010. Disponível em: <http://www2.dc.ufscar.br/~saito/download/arquitetura-PIS/Aula2-VHDL.ppt>. Acesso em: 20 dez. 2012.

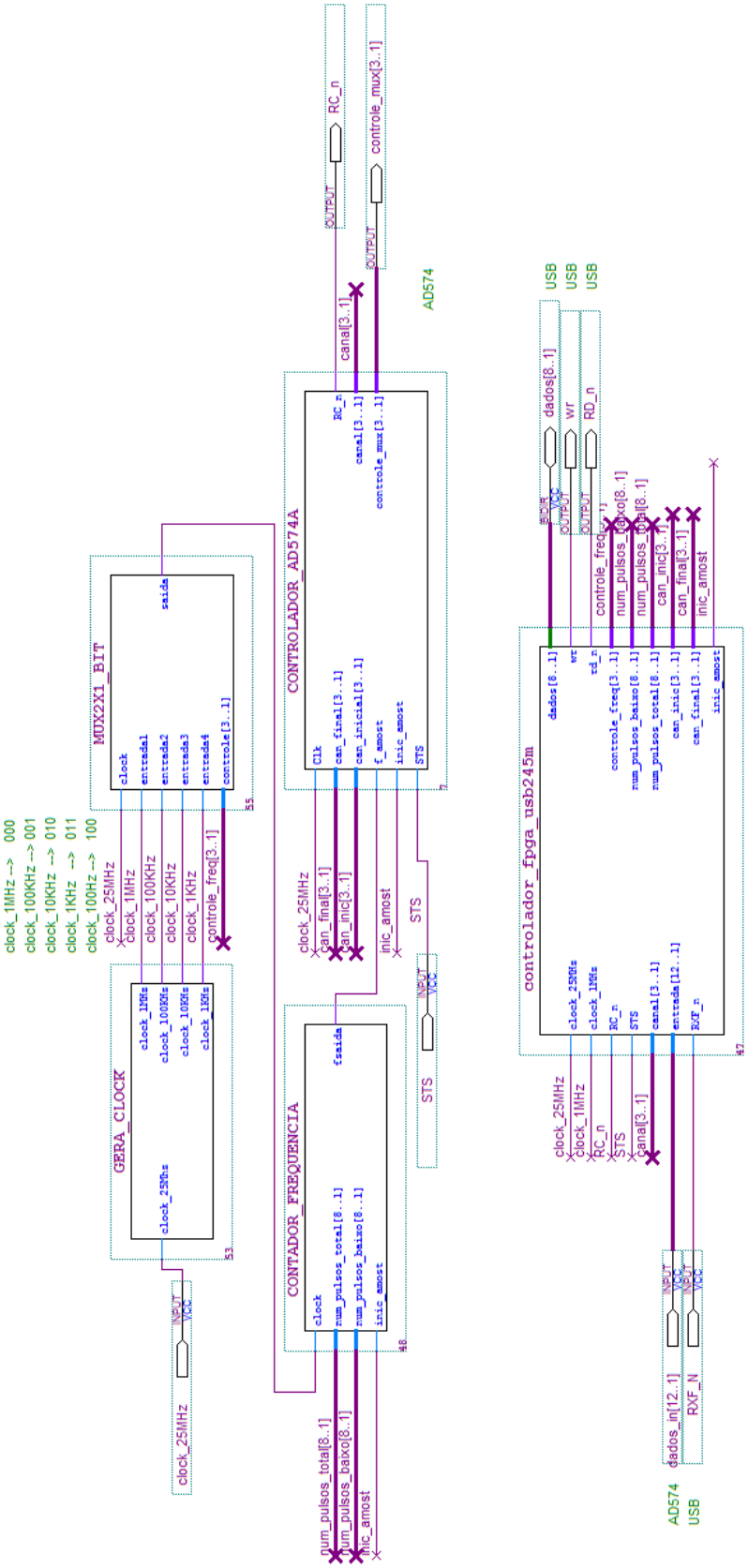
SIMÕES, Fábio da Costa. **Estudo da Relação Sinal/Ruído na Aquisição de Dados de Sensores de Alta Impedância**. Campinas, SP: Universidade Estadual de Campinas, 2008. Disponível em: <http://www.fem.unicamp.br/~lotavio/TGs/2008_Rela%C3%A7%C3%A3oSinalRuidoAquisi%C3%A7%C3%A3oDadosSensoresAltaImped%C3%A2ncia_TG_F%C3%A1bioSim%C3%B5es.pdf>. Acesso em: 19 dez. 2012.

SILVA, Gabriel P. **Introdução ao VHDL – Circuitos Lógicos – Parte 1**. [S.I.], 2006. Disponível em: <http://www.dcc.ufrj.br/~gabriel/circlog/VHDL-1.pdf>. Acesso em: 20 dez. 2012.

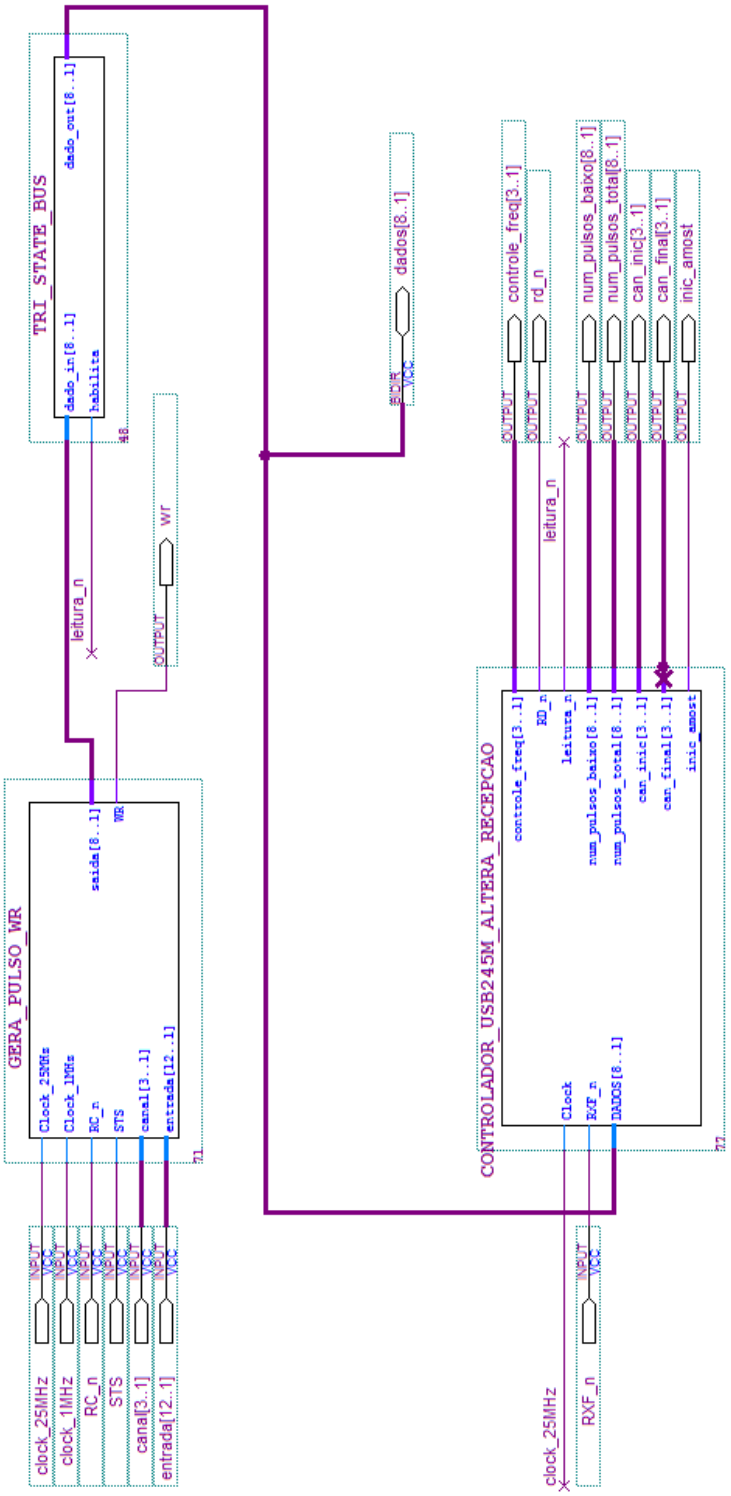
THOMAS, Donald E.; MOORBY, Philip R. **The Verilog® Hardware Description Language.** 5th ed. Dordrecht: Kluwer Academic Publishers, 2002.

WILLRICH, Roberto. **Introdução à Arquitetura de Computadores – Capítulo 3**. [S.I.], [2005?]. Disponível em: <http://algol.dcc.ufla.br/~monserrat/icc/Introducao_arq_computador.pdf>. Acesso em: 25 fev. 2013.

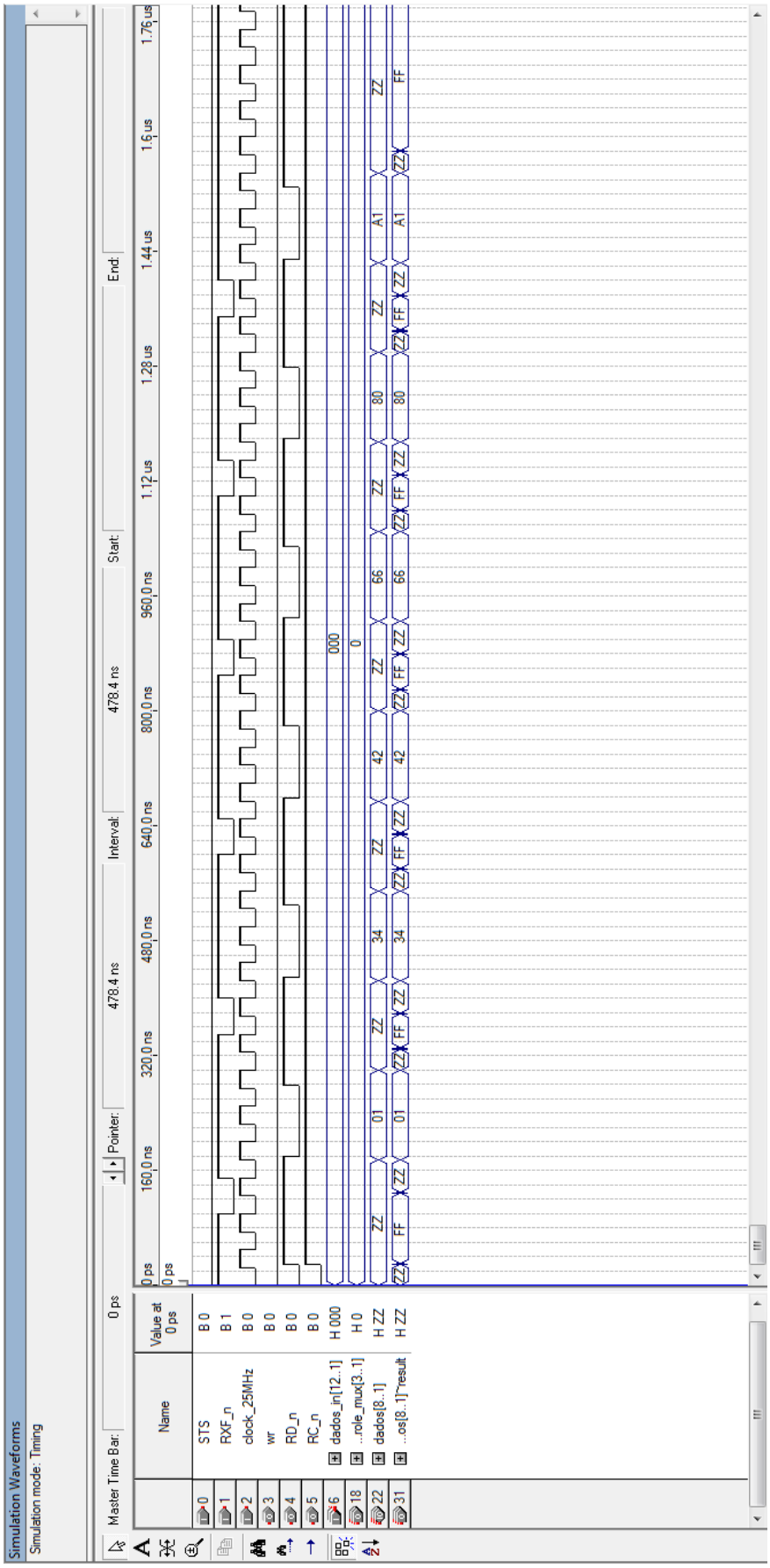
ANEXO A – Representação gráfica do sistema conversor completo, com seus blocos internos detalhados.



ANEXO B – Representação gráfica dos blocos internos ao bloco CONTROLADOR_FPGA_USB245M.



ANEXO C – Simulação do conversor analógico-digital.



ANEXO D – Código fonte do conversor analógico-digital.

**Bloco GERA_CLOCK:**

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY gera_clock IS
      PORT
      (
            clock_25Mhz                   : IN  STD_LOGIC;
            clock_1MHz                    : OUT STD_LOGIC;
            clock_100KHz                        : OUT STD_LOGIC;
            clock_10KHz                   : OUT STD_LOGIC;
            clock_1KHz                    : OUT STD_LOGIC);
END gera_clock;

ARCHITECTURE comportamental OF gera_clock IS
      SIGNAL      count_1Mhz: STD_LOGIC_VECTOR(4 DOWNTO 0);
      SIGNAL      count_100Khz,       count_10Khz,       count_1Khz       :
STD_LOGIC_VECTOR(2 DOWNTO 0);
      SIGNAL      count_100hz,  count_10hz,  count_1hz  :  STD_LOGIC_VECTOR(2
DOWNTO 0);
      SIGNAL        clock_1Mhz_int,    clock_100Khz_int,    clock_10Khz_int,
clock_1Khz_int: STD_LOGIC;
      SIGNAL      clock_100hz_int,    clock_10Hz_int,    clock_1Hz_int    :
STD_LOGIC;
BEGIN
      PROCESS
      BEGIN
-- Divide by 20
            WAIT UNTIL clock_25Mhz'EVENT and clock_25Mhz = '1';
            IF count_1Mhz < 19 THEN
                  count_1Mhz <= count_1Mhz + 1;
            ELSE
                  count_1Mhz <= "00000";
            END IF;
            IF count_1MHz < 10 THEN
                  clock_1Mhz_int <= '0';
            ELSE
                  clock_1Mhz_int <= '1';
            END IF;

            clock_1Mhz <= clock_1Mhz_int;
            clock_100Khz <= clock_100Khz_int;
            clock_10Khz <= clock_10Khz_int;
            clock_1Khz <= clock_1Khz_int;
      END PROCESS;
-- Divide by 10
      PROCESS
      BEGIN
            WAIT UNTIL clock_1Mhz_int'EVENT and clock_1Mhz_int = '1';
                  IF count_100Khz /= 4 THEN
                        count_100Khz <= count_100Khz + 1;
                  ELSE
                        count_100khz <= "000";
                        clock_100Khz_int <= NOT clock_100Khz_int;
```

```
                END IF;
      END PROCESS;
-- Divide by 10
      PROCESS
      BEGIN
            WAIT UNTIL clock_100Khz_int'EVENT and clock_100Khz_int = '1';
                IF count_10Khz /= 4 THEN
                      count_10Khz <= count_10Khz + 1;
                ELSE
                      count_10khz <= "000";
                      clock_10Khz_int <= NOT clock_10Khz_int;
                END IF;
      END PROCESS;
-- Divide by 10
      PROCESS
      BEGIN
            WAIT UNTIL clock_10Khz_int'EVENT and clock_10Khz_int = '1';
                IF count_1Khz /= 4 THEN
                      count_1Khz <= count_1Khz + 1;
                ELSE
                      count_1khz <= "000";
                      clock_1Khz_int <= NOT clock_1Khz_int;
                END IF;
      END PROCESS;
END comportamental;
```

**Bloco MUX2X1_BIT:**

```
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux2x1_bit is
 port (
        clock           : in std_logic;
        entrada1        : in std_logic;
        entrada2        : in std_logic;
        entrada3        : in std_logic;
        entrada4        : in std_logic;
        controle        : in std_logic_vector(3 downto 1);
        saida           : out std_logic
       );
end mux2x1_bit;

architecture comportamental of mux2x1_bit is
 begin
    process (clock)
      begin
      if rising_edge(clock) then -- teste
       case  controle  is
        when "000" =>
         saida <= entrada1;
        when "001" =>
         saida <= entrada2;
        when "010" =>
         saida <= entrada3;
        when "011" =>
         saida <= entrada4;
        when others => null;
       end case;
```

```
      end if;
      end process;
end comportamental;
```

**Bloco CONTADOR_FREQUENCIA:**

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY contador_frequencia IS
 PORT
 (
   clock            : IN STD_LOGIC;
   num_pulsos_total : IN STD_LOGIC_VECTOR (8 downto 1);
   num_pulsos_baixo : IN STD_LOGIC_VECTOR (8 DOWNTO 1);
   inic_amost       : IN STD_LOGIC;
   fsaida           : OUT STD_LOGIC
 );
END contador_frequencia;

ARCHITECTURE comportamental OF contador_frequencia IS
 signal contador : STD_LOGIC_VECTOR (8 DOWNTO 1);
 BEGIN
  PROCESS (clock, num_pulsos_baixo, num_pulsos_total, inic_amost)
   BEGIN
    IF inic_amost = '0' THEN
      contador <= "00000000";
     ELSIF RISING_EDGE(clock) THEN
      IF contador < num_pulsos_baixo THEN
        fsaida <= '0';
        contador <= contador + 1;
       ELSIF    (contador    >=    num_pulsos_baixo)    and    (contador    <
num_pulsos_total) THEN
        fsaida <= '1';
        contador <= contador + 1;
      ELSIF contador = num_pulsos_total THEN
       contador <= "00000000";
     END IF;
    END IF;
   END PROCESS;
END comportamental;
```

**Bloco CONTROLADOR_AD574A:**

```
library ieee;
use ieee.std_logic_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

entity controlador_AD574A is
 port
  (Clk          : in std_logic;
   can_final    : in std_logic_vector(3 downto 1);
   can_inicial  : in std_logic_vector(3 downto 1);
   f_amost      : in std_logic;
   inic_amost   : in std_logic;
   STS          : in std_logic;
   RC_n         : out std_logic;
```

```
   canal        : out std_logic_vector(3 downto 1);
   controle_mux : buffer std_logic_vector(3 downto 1)
  );
end controlador_AD574A;

architecture comportamental of controlador_AD574A is
  type estados is (Espera, Converte1, Converte2);
  type estados_mux is (inicial, conversao1, conversao2, conversao3);
  signal estado_atual_mux : estados_mux;
  signal estado_atual : estados;
  signal conv_n_aux : std_logic;
  signal flag : std_logic;
  signal RC_n_int : std_logic;
  signal pulso_sinc_anterior : std_logic;
  signal pulso_sinc_anterior1 : std_logic;
  signal pulso_sinc : std_logic;
  signal pulso_sinc1 : std_logic;

   begin
    process(Clk)  --  processo  responsável  pela  geração  do  pulso  de
amostragem
     begin
      if rising_edge(Clk) then
       if (inic_amost = '1' and flag = '0' and f_amost = '1') then
         conv_n_aux <= '1';
         flag <= '1';
        elsif flag = '1' then
         conv_n_aux <= '0';
       end if;
       if f_amost = '0' then
         flag <= '0';
       end if;
      end if;
     end process;

process(clk) -- processo responsável pela mudança dos estados da máquina
mux
 begin
  if rising_edge(clk) then
   case estado_atual_mux is
    when inicial => if inic_amost = '0' then estado_atual_mux <= inicial;
                     elsif inic_amost = '1' then
                      if(RC_n_int = '0') then
                       estado_atual_mux <= conversao1;
                      end if;
                     end if;
    when  conversao1  =>  if  inic_amost  =  '0'  then  estado_atual_mux  <=
inicial;
                        elsif inic_amost = '1' then
                         if(RC_n_int   =   '1'   and   STS   =   '0')   then
estado_atual_mux <= conversao2;
                          else estado_atual_mux <= conversao1;
                         end if;
                        end if;
   when conversao2 => if inic_amost = '0' then estado_atual_mux <= inicial;
                        elsif inic_amost = '1' then
                         if(RC_n_int   =   '0')   then   estado_atual_mux   <=
conversao1;
                          else estado_atual_mux <= conversao2;
                         end if;
                       end if;
```

```
    when conversao3 => if inic_amost = '0' then estado_atual_mux <=
inicial;
                        else estado_atual_mux <= conversao1;
                       end if;
   end case;
  end if;
 end process;
   process(clk) -- processo responsável pela geração do sinal "pulso_sinc"
    begin
     if rising_edge(clk) then
      if (estado_atual_mux = conversao1) then
       if (pulso_sinc_anterior = '0') then
         pulso_sinc <= '1';
         pulso_sinc_anterior <= '1';
        else
         pulso_sinc <= '0';
       end if;
      end if;
      if (estado_atual_mux = conversao2) then
       pulso_sinc_anterior <= '0';
      end if;
     end if;
   end process;
  process(clk) -- processo responsável pela geração do sinal "pulso_sinc1"
   begin
    if rising_edge(clk) then
     if (estado_atual_mux = conversao2) then
       if (pulso_sinc_anterior1 = '0') then
         pulso_sinc1 <= '1';
         pulso_sinc_anterior1 <= '1';
        else
         pulso_sinc1 <= '0';
       end if;
      end if;
      if (estado_atual_mux = conversao1) then
       pulso_sinc_anterior1 <= '0';
      end if;
     end if;
   end process;
 process(sts)  --  processo  responsável  pela  geração  do  sinal  "canal"
(substituindo o anterior
   begin      -- com o fim de eliminar leituras incorretas de canal)
    if rising_edge(sts) then
      canal <= controle_mux;
     end if;
   end process;
  process(clk) -- processo responsável pela geração do sinal "controle_mux"
   begin
    if rising_edge(clk) then
     if pulso_sinc1 = '1' then
      if controle_mux = can_final then
       controle_mux <= can_inicial;
       else
        controle_mux <= controle_mux + 1;
      end if;
     end if;
     if estado_atual_mux = inicial then
      controle_mux <= can_inicial;
     end if;
    end if;
   end process;
```

```
      process(Clk) -- processo responsável pela mudança dos estados
     begin
      if rising_edge(Clk) then
         case estado_atual is
          when Espera => if conv_n_aux = '0' then estado_atual <= Espera;
                         elsif  conv_n_aux  =  '1'  then  estado_atual  <=
Converte1;
                        end if;
          when Converte1 => if STS = '0' then estado_atual <= Converte1;
                            elsif   STS   =   '1'   then   estado_atual   <=
Converte2;
                           end if;
          when Converte2 => if STS = '1' then estado_atual <= Converte2;
                            elsif STS = '0' then estado_atual <= Espera;
                           end if;
         end case;
      end if;
     end process;
     process(estado_atual) -- processo responsável pela geração dos sinais
de saída
      begin
       if rising_edge(Clk) then
        case estado_atual is
         when Espera => RC_n <= '1';
                        RC_n_int <= '1';
         when Converte1 => if inic_amost = '1' then
                           RC_n <= '0';
                           RC_n_int <= '0';
                           else
                            RC_n <= '1';
                            RC_n_int <= '1';
                           end if;
         when Converte2 => RC_n <= '1';
                           RC_n_int <= '1';
        end case;
       end if;
      end process;
     end comportamental;
```

Início do circuito controlador do USB

**Bloco GERA_PULSO_WR:**

```
library ieee;
use ieee.std_logic_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

entity gera_pulso_wr is
 port
  (Clock_25MHz   : in std_logic;
   Clock_1MHz    : in std_logic;
   RC_n          : in std_logic;
   STS           : in std_logic;
   canal         : in std_logic_vector(3 downto 1);
   entrada       : in std_logic_vector(12 downto 1);
   saida         : out std_logic_vector(8 downto 1);
   WR            : out std_logic
  );
end gera_pulso_wr;
```

```
architecture comportamental of gera_pulso_wr is
type estados is (S0, S1, S2);
  signal estado_atual : estados;
  signal contador : std_logic_vector(4 downto 1);
  signal flag : std_logic;
  signal habilita : std_logic;
   begin
   process (Clock_1MHz, habilita, contador)  -- processo responsável pela
geração do sinal WR
    begin
     if rising_edge(Clock_1MHz) then
      if habilita = '1' then
       if contador = "1000" then
         contador <= "1000";
         flag <= '1';
        else contador <= contador + 1;
       end if;
      elsif habilita = '0' then
       contador <= "0000";
       flag <= '0';
      end if;
     end if;
    end process;
      process(Clock_25MHz)     -- processo responsável pela mudança dos
estados para geração do sinal habilita
     begin
      if rising_edge(Clock_25MHz) then
         case estado_atual is
          when S0 => if (RC_n = '1' and STS = '0' and flag = '0') then
                       estado_atual <= S0;
                      elsif (RC_n = '1' and STS = '1' and flag = '0') then
estado_atual <= S1;
                     end if;
          when S1 => if (RC_n = '1' and STS = '0' and flag = '0') then
                       estado_atual <= S2;
                      else estado_atual <= S1;
                     end if;
          when S2 => if (RC_n = '1' and STS = '0' and flag = '0') then
                       estado_atual <= S2;
                      elsif (RC_n = '1' and STS = '0' and flag = '1') then
                       estado_atual <= S0;
                      elsif (RC_n = '0' and STS ='0' and flag = '1') then
                       estado_atual <= S0;
                     end if;
         when others => null;
        end case;
       end if;
     end process;
     process(estado_atual,Clock_25MHz) -- processo responsável pela geração
do sinal habilita
      begin
      IF RISING_EDGE(Clock_25MHz) THEN
       case estado_atual is
        when S0 => habilita <= '0';
        when S1 => habilita <= '0';
        when S2 => habilita <= '1';
      end case;
     END IF;
    end process;
    process (clock_25MHz) -- responsável pelo controle da palavra enviada
ao dispositivo USB
```

```
      begin
      if rising_edge(clock_25MHz) then
       case  contador is
        when "0000" =>
         saida <= "11111111";
         wr <= '0';
        when "0001" =>
         saida <= "11111111";
        wr <= '0';
        when "0010" =>
         saida <= "11111111";
         wr <= '0';
        when "0011" =>
         saida <= "11111111";
         wr <= '0';
        when "0100" =>
         saida <= "11111111";
         wr <= '0';
        when "0101" =>
         saida <= '0' & canal (3 downto 1) & entrada (12 downto 9);
         wr <= contador(1);
        when "0110" =>
         saida <= '0' & canal (3 downto 1) & entrada (12 downto 9);
         wr <= contador(1);
        when "0111" =>
         saida <= entrada(8 downto 1);
         wr <= contador(1);
        when "1000" =>
         saida <= entrada(8 downto 1);
         wr <= '0';
        when others =>
         saida <= "00000000";
         wr <= '0';
       end case;
      end if;
      end process;
   end comportamental;
```

**Bloco TRI_STATE_BUS:**

```
library IEEE;
use  IEEE.STD_LOGIC_1164.all;
use  IEEE.STD_LOGIC_ARITH.all;
use  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY tri_state_bus IS
      PORT
      ( dado_in : in std_logic_vector(8 downto 1);
      habilita : in std_logic;
      dado_out : out std_logic_vector(8 downto 1)
    );
END tri_state_bus;

ARCHITECTURE comportamental OF tri_state_bus IS
 BEGIN
  process (dado_in, habilita)
   begin
    if habilita = '1' then dado_out <= dado_in;
     elsif habilita = '0' then dado_out <= "ZZZZZZZZ";
    end if;
   end process;
```

```
END comportamental;
```

**Bloco CONTROLADOR_USB245M_ALTERA_RECEPCAO:**

```
library ieee;
use ieee.std_logic_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

entity controlador_USB245M_altera_recepcao is
 port
  (Clock            : in std_logic;
   RXF_n            : in std_logic;
   DADOS            : in std_logic_vector(8 downto 1);
   controle_freq    : out std_logic_vector(3 downto 1);
   RD_n             : out std_logic;
   leitura_n        : out std_logic;
   num_pulsos_baixo : out std_logic_vector(8 downto 1);
   num_pulsos_total : out std_logic_vector(8 downto 1);
   can_inic         : out std_logic_vector(3 downto 1);
   can_final        : out std_logic_vector(3 downto 1);
   inic_amost       : out std_logic
  );
end controlador_USB245M_altera_recepcao;

architecture comportamental of controlador_USB245M_altera_recepcao is
  type estados is (Espera, Recebe1, Recebe2, Recebe3);
  signal estado_atual : estados;
   begin
      process(Clock) -- processo responsável pela mudança dos estados da
recepção (DLP USB245M -->FPGA )
     begin
      if rising_edge(Clock) then
         case estado_atual is
          when Espera => if RXF_n = '1' then estado_atual <= Espera;
                           elsif RXF_n = '0' then estado_atual <= Recebe1;
                         end if;
          when Recebe1 =>  estado_atual <= Recebe2;
          when Recebe2 => estado_atual <= Recebe3;
          when Recebe3 => estado_atual <= Espera;
         end case;
       end if;
     end process;
     process(Clock,estado_atual) -- processo responsável pela geração dos
sinais de saída da recepção
      begin
       if rising_edge(Clock) then
        case estado_atual is
         when Espera =>
                        RD_n <= '1';
                        if RXF_n = '0' then
                          leitura_n <= '0' ;
                         else leitura_n <= '1';
                        end if;
         when Recebe1 => leitura_n <= '0';
                         RD_n <= '0';
         when Recebe2 => leitura_n <= '0';
                         RD_n <= '0';
         when Recebe3 => leitura_n <= '0';
                         RD_n <= '1';
                         if DADOS(8 downto 6) = "000" then
```

```
                            num_pulsos_baixo(8 downto 5) <= DADOS(5 downto
2);
                            controle_freq(1) <= DADOS(1);
                           elsif DADOS(8 downto 6) = "001" then
                            num_pulsos_baixo(4 downto 1) <= DADOS(5 downto
2);
                            controle_freq(2) <= DADOS(1);
                           elsif DADOS(8 downto 6) = "010" then
                            num_pulsos_total(8 downto 5) <= DADOS(5 downto
2);
                            controle_freq(3) <= DADOS(1);
                           elsif DADOS(8 downto 6) = "011" then
                            num_pulsos_total(4 downto 1) <= DADOS(5 downto
2);
                           elsif DADOS(8 downto 6) = "100" then
                           can_inic <= DADOS(4 downto 2);
                           elsif DADOS(8 downto 6) = "101" then
                           can_final <= DADOS(4 downto 2);
                            inic_amost <= DADOS(1);
                         end if;
        end case;
       end if;
      end process;
     end comportamental;
```