Universidade Federal do Rio de Janeiro

Escola Politécnica

Departamento de Eletrônica e de Computação

Lógica de Aquisição de Dados para Detecção de Raios Cósmicos

Autor:

_____

Pedro Oliveira Quitete de Lima

Orientador:

_____

Prof. Mário Vaz da Silva

Examinador:

_____

Prof. José Paulo Brafman

Examinador:

_____

Prof. Carlos José Ribas D'Avila

DEL

Agosto de 2013

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica – Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro – RJ CEP 21949-900

AGRADECIMENTO

Agradeço primeiramente aos meus pais, Angela e Guilherme, que me deram toda a base e apoio físico, monetário e emocional para que fosse possível sair de minha cidade natal e conseguir concluir minha graduação numa das melhores universidades do país.

Faço um agradecimento especial a minha namorada Ana, cujo apoio e carinho nesses últimos (aproximadamente) 3 anos foram insubstituíveis para que eu conseguisse ter o equilíbrio necessário para terminar as matérias, fazer as mudanças necessárias e conseguir terminar este projeto.

Outro agradecimento especial para meus amigos: os que me acompanham desde o inicio da faculdade, Hugo, Lui e Renato e os que surgiram no caminho, Henrique, Bean, Diego e Menezes, esses e mais outros sumidos que me aguentaram em bares e festas; todos os colegas de GECOM que não me deixaram enfartar, em especial o Sr. Saraiva e Laura; aos meus praticamente irmãos campistas, Nilson, Jonathan, Sky e Rafael; os santos amigos que me ajudaram durante as partes mais complexas do curso, Fernanda e Marlon; assim como todos os mais recentes que me aguentam diariamente, ao vivo ou online.

Finalmente agradeço ao meu orientador Prof. Mario Vaz que desde 2009 me acompanhou na iniciação científica, me indicou para estágio e me orientou neste projeto. Também agradeço aos Prof. Carlos José e Prof. Brafman, que se disponibilizaram como avaliadores e cujas ajudas tanto para o GECOM quando eu me responsabilizava pelo grupo, quanto pessoalmente foram insubstituíveis.

RESUMO

Este documento trata do desenvolvimento de um sistema digital de aquisição de dados para detectores raios cósmicos [1]. O protótipo resultante deste trabalho se destina ao apoio no desenvolvimento de sistemas de veto de múons cósmicos para aquisição de dados do detector Neutrinos-Angra [2] e o detector de raios cósmicos para ensino de física de 2º e 3º graus no Laboratório Didático do CBPF [3].

Palavras-Chave: raios cósmicos, detecção, aquisição.

ABSTRACT

This document deals with the development of a digital data acquisition system for cosmic ray detectors [1]. The resulting prototype of this work is intended to support the development of cosmic muon veto systems for data acquisition of Neutrinos-Angra detector [2] and for a detector to teach cosmic rays physics at 2nd and 3rd degree in CBPF's Laboratory Guided [3].

Key-words: cosmic rays, detection, acquisition.

SIGLAS

UFRJ – Universidade Federal do Rio de Janeiro

CBPF – Centro Brasileiro de Pesquisas Físicas

LAFEX – Laboratório de Física Experimental de Altas Energias do CBPF

FPGA – Field-Programmable gate array

CERN – Conseil Européen pour la Recherche Nucléaire

LHCb – Large Hadron Collider beauty

PMT – Photomultiplier Tube

MAPMT – Multianode PMT

LVDS – Low-Voltage Differential Signaling

DCM – Digital Clock Manager

DAC – Digital to Analog Converter

CI – Circuito Integrado

DC – Direct Current

ADC – Analog to Digital Converter

FIFO – First In First Out

USB – Universal Serial Bus

SPI – Serial Peripheral Interface

SCK – Serial Clock

MOSI – Master Output Slave Input

MISO – Master Input Slave Output

CS – Chip Select

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 – Tema

Neste projeto para instrumentação eletrônica, desenvolvemos um sistema digital de aquisição de dados para detectores de partículas, em especial detectores de raios cósmicos que utilizam cintiladores plásticos e tubos fotomultiplicadores. O sistema digital foi implementado em um dispositivo programável do tipo FPGA. O sistema além de realizar a aquisição de dados do detector, também faz a comunicação com um computador pessoal, para fins de controle e configuração da aquisição de dados.

1.2 – Delimitação

Este projeto teve origem no CBPF, no desenvolvimento de detectores de raios cósmicos para fins de ensino em diversos níveis, desde o 2º grau até a pós-graduação, bem como no desenvolvimento de um detector de partículas como neutrinos produzidos na usina nuclear Angra II. Consiste em um sistema eletrônico digital que foi integrado a um detector de múons composto de cintiladores, MAPMT e amplificadores, e de um circuito desenvolvido em projeto de graduação anteriormente apresentado ao DEL.

1.3 – Justificativa

A importância no desenvolvimento deste equipamento se dá na instrumentação eletrônica em geral, na física experimental e no ensino de física de partículas. No futuro, poderá ser estendido para aplicações nas áreas de instrumentação médica e de análise de materiais.

1.4 – Objetivos

Este projeto tem dois objetivos principais. O primeiro é o aprendizado e ganho de experiência prática no desenvolvimento e produção de sistemas digitais em

dispositivos programáveis. O outro é o desenvolvimento de um produto tecnológico para a área de pesquisa e ensino, para compor um sistema de aquisição de dados de detectores de partículas ou sistemas equivalentes.

1.5 – Metodologia

O desenvolvimento do sistema foi feito em tópicos separados, integrados ao longo do trabalho.

O primeiro tópico abordado foi a detecção dos raios cósmicos, tal como está sendo feita no LAFEX, com observações através de instrumentos. Este tópico será apresentado sucintamente, para introduzir a noção dos sinais elétricos gerados desde a detecção das partículas até a apresentação ou armazenamento do resultado da observação ou medida nos eventos observados.

O segundo tópico foi a análise dos sinais referentes à detecção, a transformação, visualização e armazenamento dos dados dos eventos observados. Este tópico foi dividido em três partes: o *condicionamento do sinal*, que consiste em transformar o sinal gerado no fotodetector de forma conveniente para processamento da informação de interesse, separando-a do ruído; o *back-end*, onde se determina a relevância da informação desejada contida no sinal, através da lógica de coincidência e da filtragem dos eventos; e o *front-end*, onde é feita a visualização e armazenamento dos dados recebidos do back-end, a configuração, monitoração e o controle do sistema pelo usuário.

Um resumo do desenvolvimento do sistema pode ser observado na Fig. 1.1.

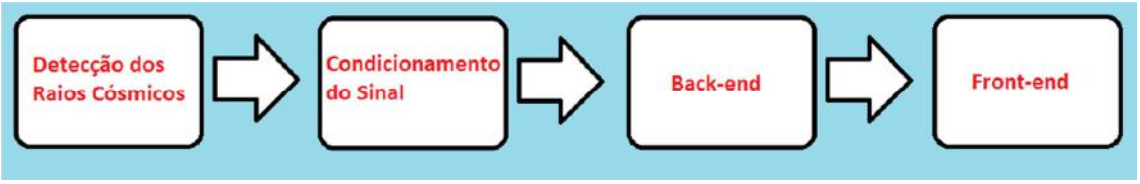


Figura 1.1 – Tópicos de desenvolvimento

1.6 – Descrição

No capítulo 2 serão abordados os temas referentes ao primeiro tópico, a detecção dos raios cósmicos. Neste capítulo, apresenta-se a metodologia utilizada na detecção de raios cósmicos, como ela foi implementada, como foi especificado, projetado e montado o equipamento para realizar esta operação e como se dá a saída de dados para fins de

armazenamento e visualização. Para completar o capítulo, também será abordado como o sinal foi condicionado para que fosse possível manipulá-lo pelo back-end. Será definido como deve ser a saída do sinal conformado, baseando-se na entrada recebida pelos detectores. Também será demonstrado como a conformação foi implementada e o que foi necessário para completar a operação.

Em seguida no Capítulo 3, será detalhado o back-end desenvolvido. Serão indicados como será recebido o sinal conformado na entrada, quais dispositivos irão utilizá-lo e qual será a lógica de controle. Será demonstrado como cada parte do dispositivo foi escolhida e montada para realizar as funções necessárias. Neste capítulo, teremos exemplos implementados antes da integração que demonstrarão cada etapa implementada, utilizando simulações e testes de bancada.

Após, no Capítulo 4, o front-end será detalhado. Será demonstrado como os sinais vindos do back-end serão recebidos, tratados e mostrados para o usuário e como ele poderá trabalhar com eles. Além disso, serão descritas as opções de configuração e como elas funcionam, mostrando como o usuário irá operar o sistema e como as mudanças que ele realizar irão afetar o sistema. O final deste capítulo irá descrever o final do desenvolvimento, mostrando como foi realizada a integração do sistema, como todas as etapas foram reunidas e os testes realizados, deixando claro todos os resultados obtidos.

Finalmente no Capítulo 5, teremos a conclusão do projeto, com as considerações finais, contendo o que foi aprendido e o que poderá ser aplicado no futuro.

Capítulo 2

Detecção dos Raios Cósmicos e Condicionamento dos Sinais

2.1 – Da detecção ao sinal a ser condicionado

Os raios cósmicos detectados ao nível do mar são partículas resultantes da interação com os átomos da alta atmosfera de partículas altamente energéticas provenientes do espaço sideral. Os raios cósmicos são gerados na forma de chuveiros de partículas, sendo compostas principalmente de neutrinos e múons, e em menor quantidade, de píons, elétrons e pósitrons. Um estudo de suas incidências pode ser observado na Fig. 2.1. As partículas que observamos foram os múons, partículas eletricamente carregadas.

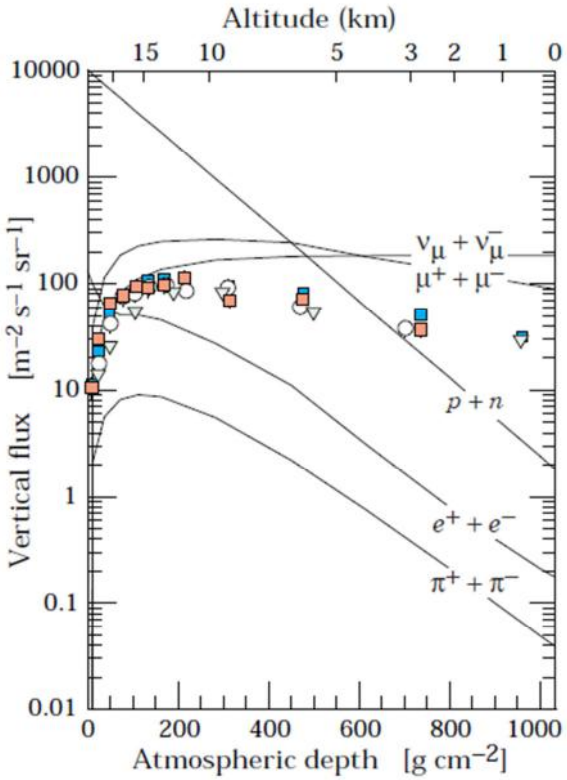


Figura 2.1 - Fluxo e composição de raios cósmicos secundários em função da altitude. Os pontos indicam múons negativos medidos em diversos experimentos. Fonte: [4, figura 24.3]

A maior taxa de partículas eletricamente carregadas é de múons. Um detector de múons colocado na horizontal recebe uma taxa de cerca de 1 múon/cm^2/s, sendo que essa taxa varia segundo uma função de quadrado de cosseno do ângulo da incidência com o zênite. Isto é suficiente para afetar experimentos de detecção de partículas, tal como detecção de neutrinos.

A detecção que este projeto aborda, resulta da interação do múon com um plástico cintilador, onde luz é gerada por luminescência, ou fluorescência, forma mais simples e menos onerosa, muito efetiva na cobertura de grandes áreas. Um cintilador é um material com as propriedades de radioluminescência, que emite fótons quando excitado por radiação ionizante (fluxo de partículas carregadas eletricamente tais como múons, elétrons e íons) ou partículas de carga neutra (por ex. fótons e nêutrons) com energia suficiente para excitar os átomos do material. O processo é chamado fluorescência quando a reemissão é imediata e os elétrons excitados retornam ao estado inicial rapidamente, em menos de 10ns. Porém, quando os elétrons ficam em um estado excitado metaestável por um tempo que pode durar entre poucos microssegundos a horas, dependendo do material, ocorre a fosforescência, com uma reemissão demorada de fótons (after-glow) [5]. Um diagrama de energia destes processos pode ser observado na Fig. 2.2.

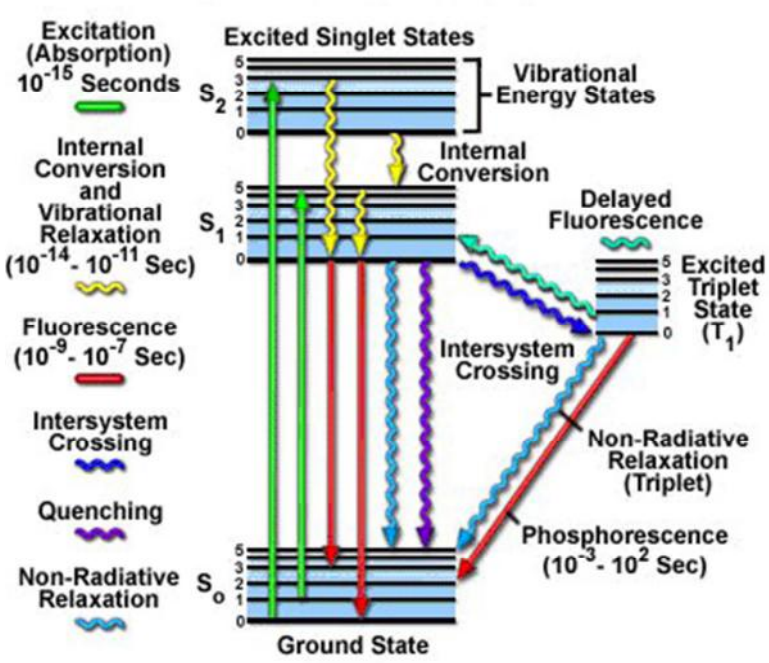


Figura 2.2 - Diagrama de energia para luminescência (fluorescência e fosforescência). Fonte: [5]

As placas cintiladoras utilizadas tem dimensões de 160x5x1cm^3 e ao longo do seu comprimento possuem um orifício central para receber a fibra cintiladora WLS. Esta fibra tem como função coletar a luz gerada em qualquer ponto da placa e dirigir esta luz para um fotocatodo de um tubo fotomultiplicador ou fotodiodo. Ao mesmo tempo a fibra WLS amplifica a luz ao se excitar com fótons de energia correspondente à luz do cintilador, entre o azul e o ultravioleta, diminuindo a excitação, gerando fótons com energia correspondente à luz à qual o fotodetector é mais sensível, na faixa do verde, menos energéticos. A diferença de energia faz com que o número de fótons gerados seja superior ao de fótons incidentes, em um processo amplificador de luz que compensa as perdas de coleta, geração e transmissão dos fótons. Este processo pode ser observado pela Fig. 2.3, que relaciona a sensibilidade e eficiência do tubo fotomultiplicador com o comprimento de onda de fótons.

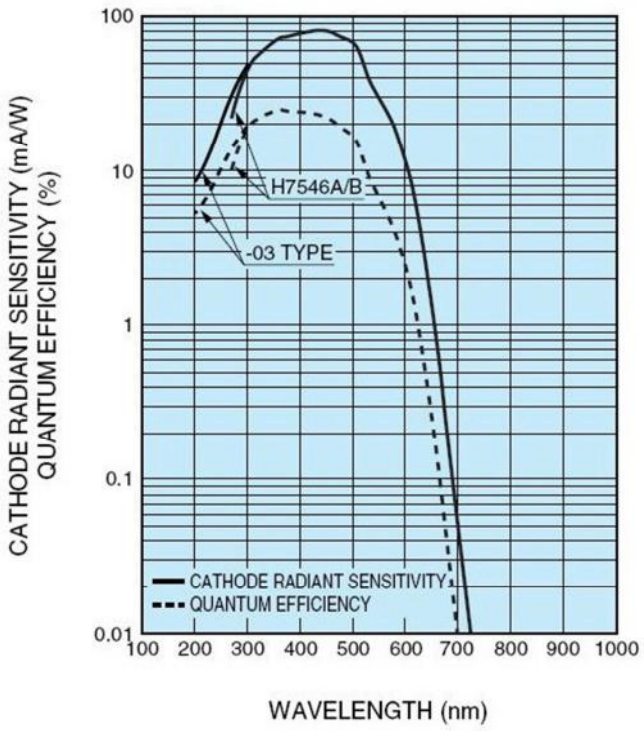


Figura 2.3 - Eficiência quântica da MAPMT H7546A/B e sensibilidade à energia radiante em função do comprimento de onda do fóton. Fonte: [8, Fig. 1]

O tubo fotomultiplicador ou simplesmente fotomultiplicadora, é uma válvula de vidro hermeticamente fechada, com pressão interna típica em torno de 10^{-4} Pa (vácuo) que realiza a detecção de luz por meio do efeito fotoelétrico. Consequentemente, a luz será detectada em termos de partículas (fótons), com energia definida em função do comprimento de onda, segundo a dualizada onda-partícula. Como mostra a Fig. 2.4, em

seu interior há um cátodo sensível à luz (fotocatodo), eletrodo de foco, um ânodo e elementos multiplicadores de elétrons (dinodos).

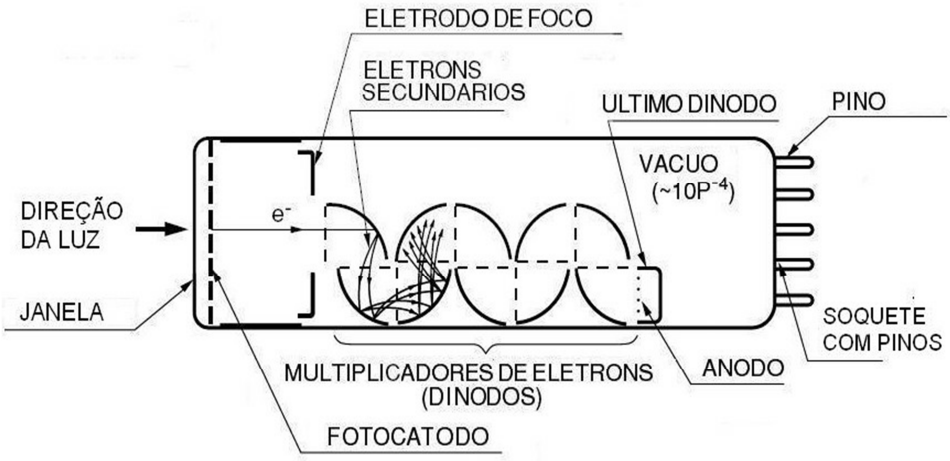


Figura 2.4 - Esquemático genérico de um tubo fotomultiplicador. Fonte: [7]

A luz atravessa a janela, a qual é feita com material transparente à radiação a ser observada e, ao atingir o fotocatodo, os fótons transferem energia (E=hv) aos elétrons que ocupam a banda de valência do material do fotocatodo. Pelo efeito fotoelétrico, se a energia fornecida ao material excita elétrons com energia superior à energia de trabalho do material, permitindo aos elétrons passarem para o vácuo, haverá emissão destes. Os elétrons neste processo são chamados de fotoelétrons sendo acelerados no vácuo pelo eletrodo de foco em direção ao primeiro dinodo pela diferença de potencial elétrico entre ele e o fotocatodo. Ao atingir o primeiro dinodo, cada fotoelétron pode provocar uma emissão secundária, com desprendimento de um ou mais elétrons. Estes elétrons serão acelerados por diferença de potencial entre o primeiro e o segundo dinodo, incidindo neste e provocando outra emissão de elétrons, em maior número que o dos incidentes, este processo irá se repetir nos dinodos seguintes, produzindo uma multiplicação de elétrons.

Do último dinodo os elétrons emitidos são coletados no ânodo, passando deste para o pino externo ao tubo, gerando uma corrente elétrica, aproximadamente proporcional ao número de fótons incidentes no fotocatodo. Todo este processo é estocástico e gera um ruído considerável, tanto maior quanto maior a diferença de potencial entre os elementos geradores de elétrons, ou seja, no anodo de um PMT se tem pulsos de corrente que são gerados internamente por efeito térmico e não fotoelétrico, sem correspondência com a detecção de luz.

Para operar com fibras WLS existem PMTs com múltiplos anodos e respectivos fotocatodos com área de 2 mm x 2 mm, mais compacta e de mais baixo custo por fotocatodo, como o Hamamatsu MAPMT H7546A, cujas características podem ser observadas na Fig. 2.5.

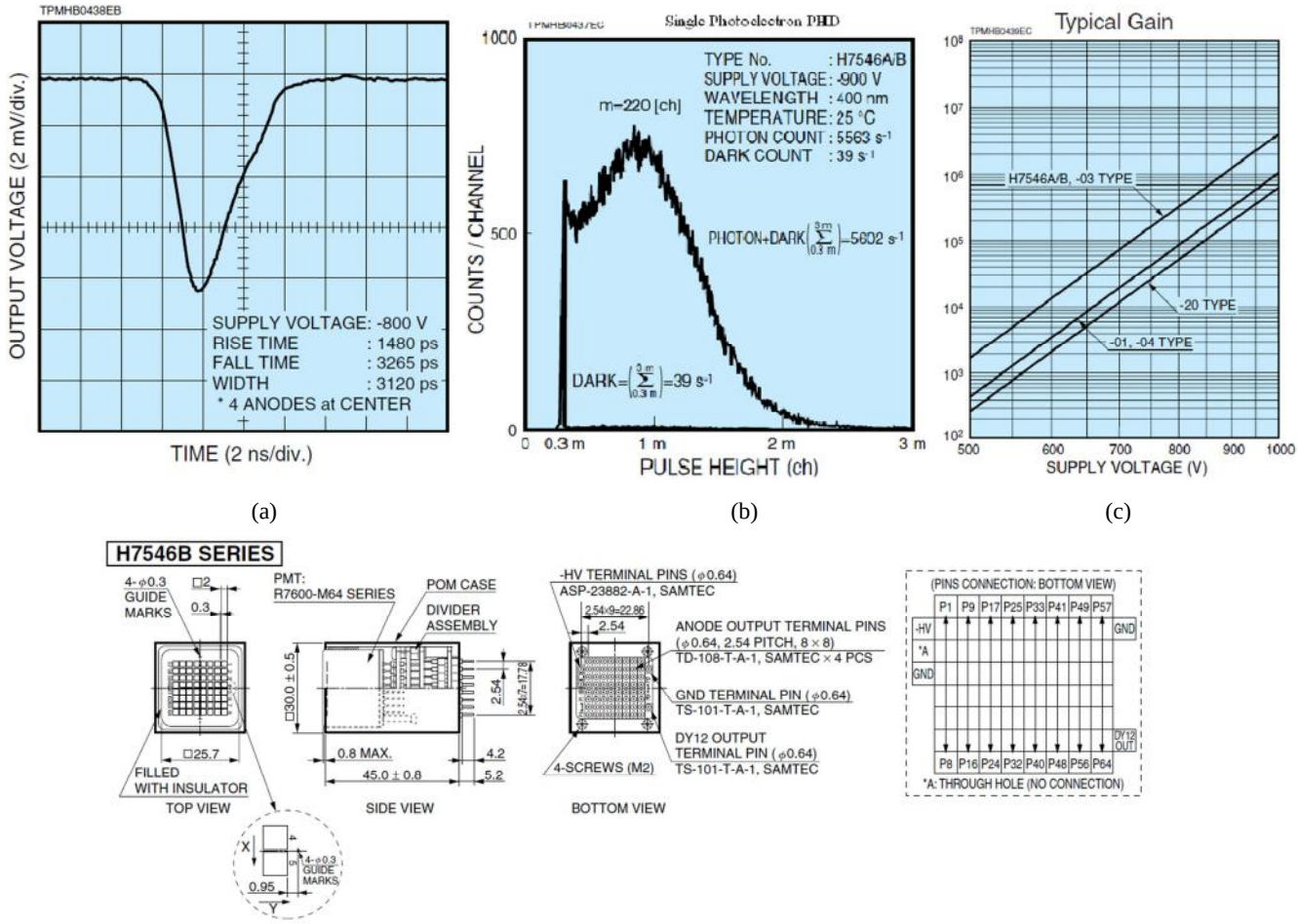


Figura 2.5 - Características dadas pela Hamamatsu para o MAPMT H7546A: (a) tempo de resposta, (b) altura de pulso ,(c) ganho da multiplicação de elétrons pela tensão de polarização do MAPMT e (d) características mecânicas. Fonte [8, Figs. 2, 3, 4 e 8]

Com esses componentes foi montado o sistema de detecção de raios cósmicos que usamos neste trabalho. As placas cintiladoras foram posicionadas de modo a que pudéssemos ter coincidência de sinais para reduzir ruído e incerteza na detecção, além de determinar qual a parte do detector foi atingida pela partícula. Com essa finalidade, foram então utilizadas fibras de comprimento de 4m, cada uma passando por duas placas cintiladoras, estas superpostas 4 a 4, como mostra a Fig. 2.6. Desta forma, cada fibra produz sinais de luz em suas extremidades defasados de acordo com a posição da incidência do múon. A velocidade da partícula na fibra será de 0.6c, onde c é a

velocidade da luz no vácuo (299 792 458 m/s), relaciona-se posição a atraso por 1/0.6c = 50ps/cm. Por exemplo, para ter a precisão igual à largura das placas, 5 cm, a precisão correspondente na determinação do tempo deve ser de 250ps. A taxa esperada de eventos de detecção de múons neste sistema pode chegar a 1 evento/min.cm² vezes a área horizontal dos cintiladores (2x160cmx5cm = 1600cm²) portanto 1600 eventos/min, logo 27 eventos/s.

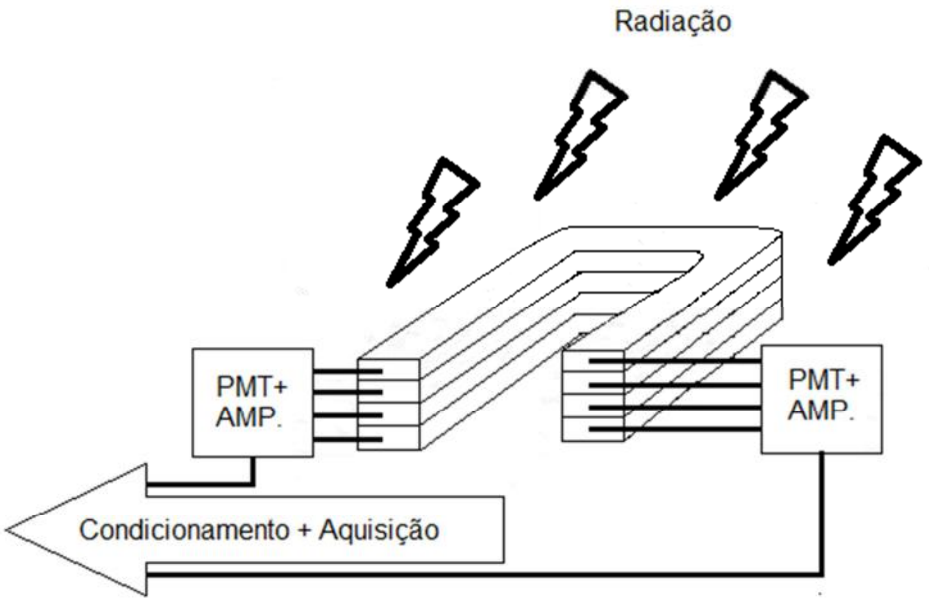
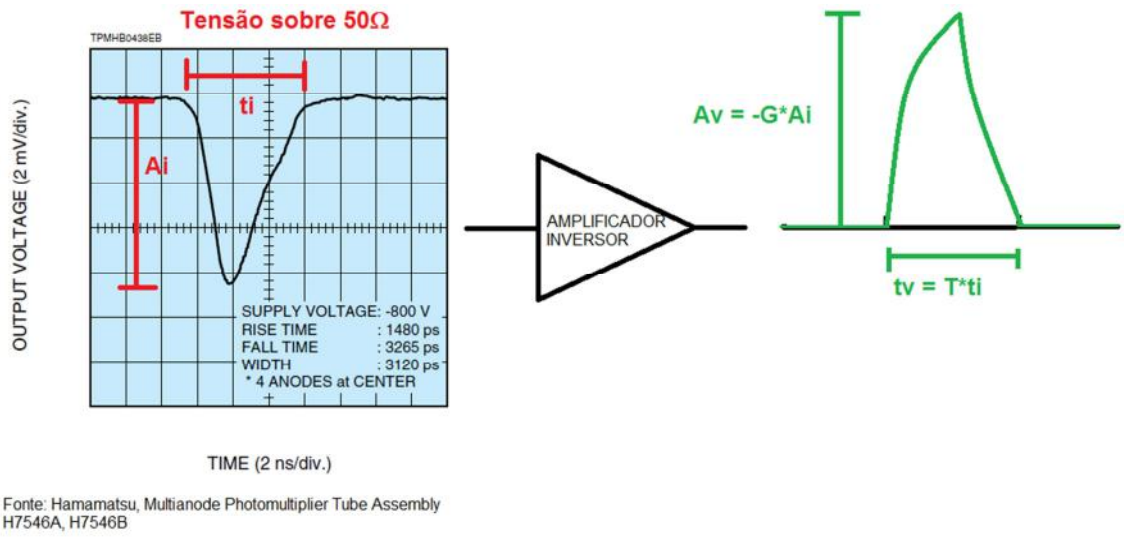


Figura 2.6 - Esquema de montagem das placas e fibras cintilantes

2.2 – Condicionamento do Sinal

Do anodo na saída do PMT teremos uma corrente elétrica detectável, com a forma aproximadamente triangular. Para tratar este sinal, se converte corrente em tensão de nível alto o suficiente para ser processado diretamente por uma FPGA com amplificadores modeladores da forma de onda, que preservem as características de transição inicial do pulso de corrente, importantes para a medida de tempo, mas que apresentam uma transição final lenta, de modo a integrar e reduzir o ruído, melhorando o processamento digital pela FPGA. O pulso de tensão na saída destes amplificadores, portanto é invertido em relação ao sinal de entrada e mais longo, amplificado de modo a facilitar a detecção pela FPGA, como mostrado na Fig. 2.7, sendo esta montagem já testada e aprovada [15].



Fonte: Hamamatsu, Multianode Photomultiplier Tube Assembly H7546A, H7546B

Figura 2.7 - Saída dos fotomultiplicadores e passagem pelos amplificadores, conversão da corrente anódica da MAPMT em pulsos triangulares de tensão

Os sinais amplificados são enviados para o módulo de coincidência do back-end. O operador do sistema pode selecionar um deles tanto para conversão analógica digital e envio da forma de onda digitalizada para o computador, quanto para observação por osciloscópio através de circuito que o apresente via conector BNC.

Para o primeiro módulo, temos 8 sinais de tensão como entrada, sendo detectadas coincidências entre eles, entretanto, é necessário se evitar sinais indesejados. Assim, por discriminação, estabelecendo uma tensão de referência VREF, cujo valor define quais dos sinais recebidos serão válidos e armazenados para processamento, os demais são considerados nulos.

Esperamos fazer esta discriminação pela FPGA, usando módulos de entrada em padrão LVDS, diferencial, já existentes, onde uma entrada recebe a tensão de referência de discriminação e o outro o sinal amplificado. A discriminação de sinais assim é feita em alta velocidade característica do padrão LVDS, 600Mbps.

Já a seleção de sinais a serem convertidos, será uma operação de multiplexação analógica dos 8 canais, onde um multiplexador 74HC4051 irá selecionar um dentre os sinais de tensão vindos do amplificador. O sinal selecionado irá para o módulo de conversão analógica-digital do back-end e enviado para o front-end, de modo que o operador possa observar seu comportamento. A operação pode ser observada na Fig. 2.8.

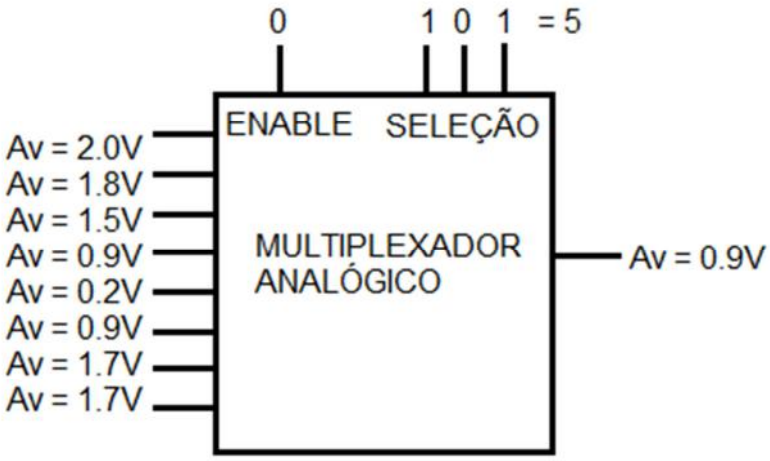


Figura 2.8 - Operação do multiplexador analógico

No exemplo de operação dado na figura acima, o multiplexador recebe o código de seleção e um sinal habilitador. No caso a entrada 5, representado por um sinal de amplitude máxima Av=0.9V será selecionado.

Finalmente, ao final destes blocos, os sinais condicionados e selecionados são usados na lógica do back-end de modo que posteriormente será usada pelo operador, o qual será responsável por configurar o sistema.

Capítulo 3

Back-end: configuração, coincidência e comunicação

3.1 – Configuração do Condicionamento

A primeira função do back-end é responsável por executar a configuração do condicionamento dos sinais definida pelo operador. Desta forma, teremos um módulo do back-end para definir a tensão de referência da operação de discriminação executada pela porta LVDS, definindo quais dos sinais serão relevantes, e por definir quais dos sinais serão selecionados pelo multiplexador, para serem convertidos e monitorados pelo operador.

3.1.1 – Tensão de referência do discriminador

Para se definir o nível de tensão de referência, será necessário realizar uma conversão do sinal digital originado na FPGA para analógico. Isso se deve a necessidade do sinal ser variável entre a alimentação e o valor mínimo (terra), isto é, deve ser possível assumir qualquer valor entre 0V e 3,3V.

Desta forma, foi utilizado um DAC controlado pela FPGA. Assim, da FPGA sairá um sinal contendo o valor que será utilizado por uma função configurada dentro do DAC, definindo um valor constante de tensão na saída, no intervalo entre 0V e 3.3V. Além disso, era necessário que o DAC possibilitasse até 8 saídas de tensão, pois será determinada uma tensão de referência para cada uma dos sinais vindos do amplificador.

Com estes parâmetros, decidiu se utilizar o AD5308, que permite realizar uma comunicação com a FPGA utilizando protocolo serial próprio com 16 bits, cuja alimentação pode variar de 2,5V até 5,5V, permitindo uma configuração onde a saída depende da expressão (3.1), onde VDD é a alimentação e D o valor inteiro enviado pela FPGA.

$$V_{OUT} = \frac{VDD \times D}{2^8} \qquad (3.1)$$

Para definir esta configuração e realizar a comunicação entre a FPGA e o DAC, era necessário entender seu funcionamento, de modo que ao se escrever o código em VHDL fosse possível implementar o algoritmo corretamente.

O primeiro passo era verificar a pinagem do DAC, determinando a função de cada um e seu papel no protocolo de comunicação, para que se conheça quais seriam conectados a FPGA e a forma como seriam enviados os sinais. Esta pinagem do CI pode ser observada na Fig. 3.1.

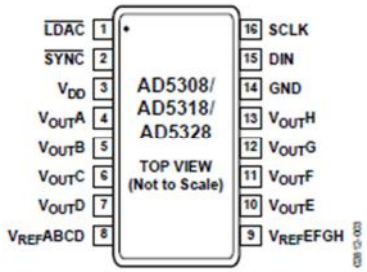


Figura 3.1 - Pinagem do AD5308

Fonte: [9, Fig. 3]

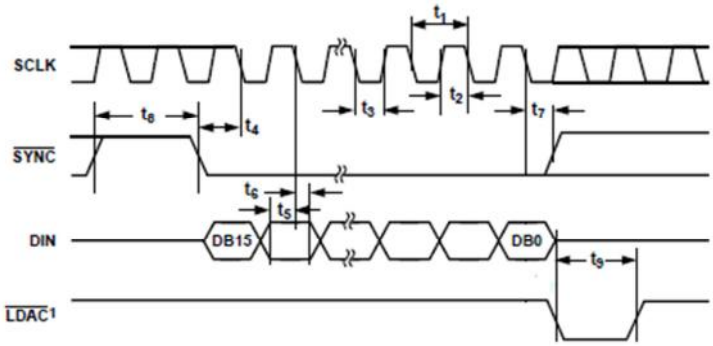


Figura 3.2 - Diagrama de tempo da comunicação do AD5308

Fonte: [9, Fig. 2]

Os pinos importantes para a comunicação serão o pino 2, SYNC, responsável pela sincronização, funcionando como o sinal de habilitação da comunicação. Além deste, o pino 16, SCLK, que receberá o clock definindo a operação de leitura na sua subida (variação de 0 para 1). Se o CI estiver habilitado (SYNC em 0), um bit será lido da entrada serial, que se localiza no pino 15, DIN. Ao final, é necessário haver uma variação de 1 para 0 no pino 1, LDAC, de modo que os dados recebidos serialmente sejam gravados nos registradores internos do CI, para serem utilizados na definição das

saídas de tensão analógicas. Esta operação pode ser observada na Fig. 3.2, com o diagrama de tempo do DAC.

O restante dos pinos ou serão de alimentação e terra, pinos 3 e 14, respectivamente, ou serão saídas: pinos 4 à 7, saídas A à D, sendo o pino 8 utilizado para referência no cálculo da saída analógica e conectado na alimentação. Semelhante será para os pinos 10 à 14, saídas de E à H, com a referência no pino 9, também conectado à alimentação.

Conhecendo a pinagem e suas funções, foi necessário verificar as possíveis operações realizadas pelo CI, de modo a se saber quais códigos serão enviados e a ordem destes. Existem dois modos de operação: o modo Write, cujo objetivo é escrever um vetor de bits num dos registradores internos, cada um conectado a um dos pinos de saída e com um endereço que é enviado junto aos dados; e o modo Control que determina quatro operações diferentes dependendo do vetor recebido. As operações do modo Control utilizados são:

- Reset, que limpa os dados salvos nos registradores internos;
- LDAC, que determina o valor da lógica do LDAC, que será controlado pelo pino 1 e pela configuração deste modo, cujas opções são deixar sempre em 0 de modo que a escrita esteja sempre habilitada ou deixar sempre em 1 de modo que seja completamente controlada pelo hardware;
- Reference, que determina se a referência para cálculo do valor de tensão analógica da saída será externa, vinda dos pinos 8 e 9 ou se será interna, vinda da alimentação.

Estas configurações devem ser realizadas antes de todas as escritas realizadas. A diferença entre os modos será dada pelo bit mais significativo, o primeiro a ser recebido pela entrada serial, se for 0, será o modo Write, se for 1, será o modo Control. Então são enviados três vetores de bits de modo a definir o controle de escrita nos registradores internos sendo realizado pelo hardware, limpar os registros, definir a referência para cálculo da saída como sendo a alimentação, para então enviar um último vetor contendo o endereço do pino e os dados de modo que segundo a expressão (3.1) se possua a saída desejada. Os passos podem ser observados na Tabela 3.1.

Passo	Modo	Comando	Vetor de bits
1	Control	LDAC	1010000000000001
2	Control	Reset	1110000000000000
3	Control	Reference	1000000000000011
4	Write	Write	0 A2A1A0 D7D6D5D4D3D2D1D0 0000

Tabela 3.1 - Vetores de bits enviados ao DAC. Fonte:[9]

Como pode se observar na Tabela 1, apenas o comando de escrita no DAC permitirá um vetor variável. Então neste é enviado o endereço da saída pelos bits A2A1A0 e os dados pelos bits D7D6D5D4D3D2D1D0, com relação como vista na Tabela 3.2.

A2 (Bit 14)	A1 (Bit 13)	A0 (Bit 12)	Saída
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

Tabela 3.2 - Endereço das saídas do DAC. Fonte: [9]

Com essas definições, foi possível escrever um código em VHDL que permite a comunicação com o DAC de modo a testar os comandos e a escrita. Este funciona a partir de uma máquina de estados, onde no primeiro estado, tem se um estado de repouso, com todas as saídas com seus valores iniciais, até se receber o sinal externo de start. No próximo estado, os códigos dos comandos a serem utilizados são armazenados e o vetor de dados a ser salvo nos registradores internos é montado. Em seguida, teremos os estados dos comandos e da escrita, sendo estes, sempre separados por um

atraso, de modo que a comunicação possa ser encerrada e reiniciada. Deste modo, todos os vetores serão enviados, retornando, então, para o repouso. O funcionamento pode se observado no diagrama de estados da Fig. 3.3.

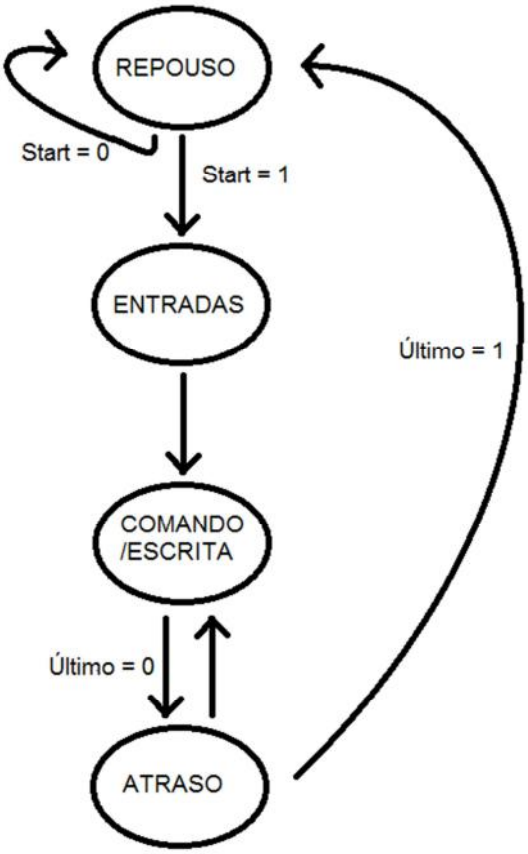


Figura 3.3 - Máquina de estados para comunicação com o DAC

Com essa montagem, o código de testes envia os comandos LDAC, Reset e Reference, para em seguida se escrever o vetor no registrador interno referente a um dos pinos. Para verificação do algoritmo, primeiro foi feita a simulação, sendo o resultado mostrado na Fig.3.4. Em seguida, foi analisada a resposta na FPGA, observando os sinais em um osciloscópio, com as imagens obtidas no equipamento vistas na Fig. 3.5. É possível observar que tanto na simulação, quanto para os sinais vistos pelo osciloscópio, foram obtidos resultados desejados. Finalmente, foi montado numa placa o CI ligando-o através de fios paralelos na FPGA para realizar a comunicação e a alimentação, como mostrado na Fig. 3.6. Verificou-se uma resposta correta, pois ao se enviar um vetor correspondente a 165 (10100101), obteve-se na saída uma tensão de 2.13V (medida com um multímetro), o resultado esperado.

Para se manter a lógica do sistema, se separou o código em dois outros, um que escreve nos registradores internos após uma configuração fixa e outro que limpa todos os dados. Ambos foram testados separadamente, de modo que o operador tenha controle tanto na definição da saída ao escrever, quanto na limpeza dos dados indesejados. Estes códigos podem ser observados no Apêndice A.

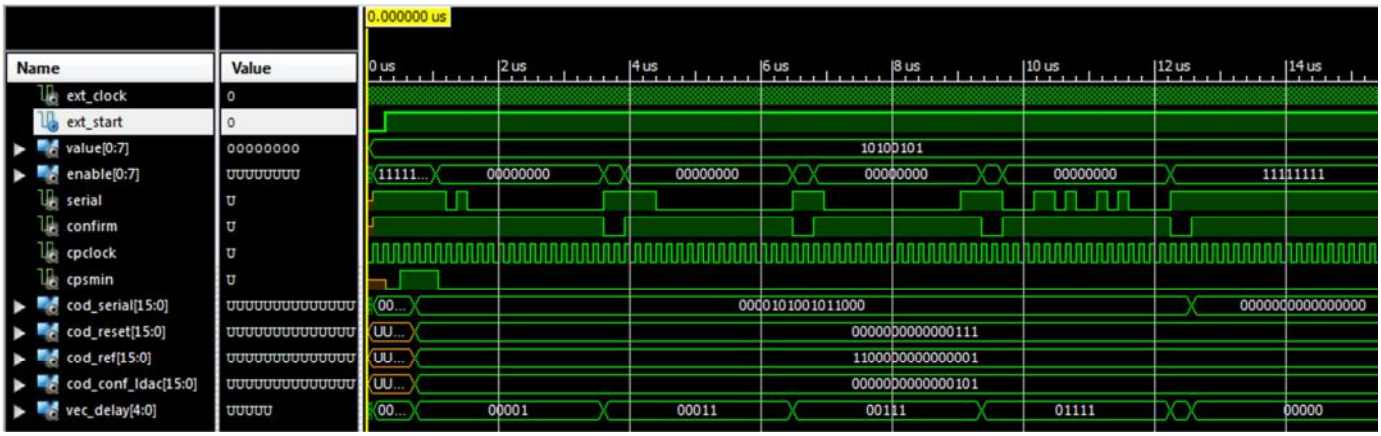
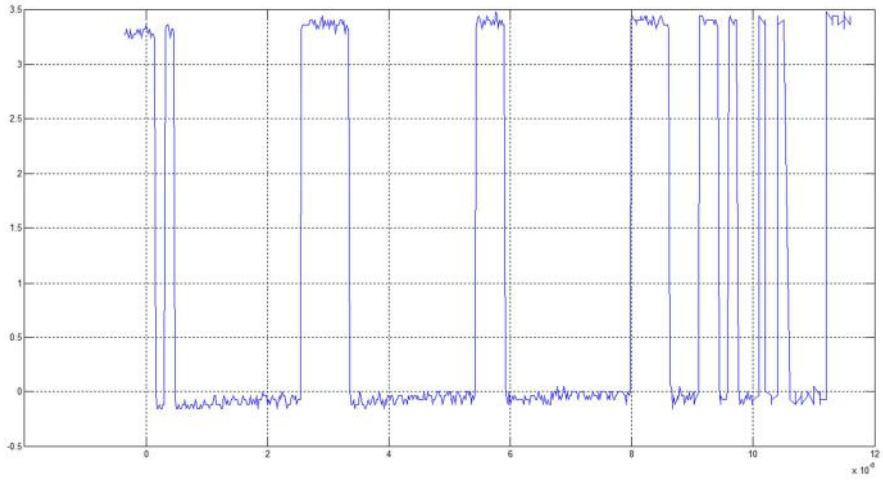
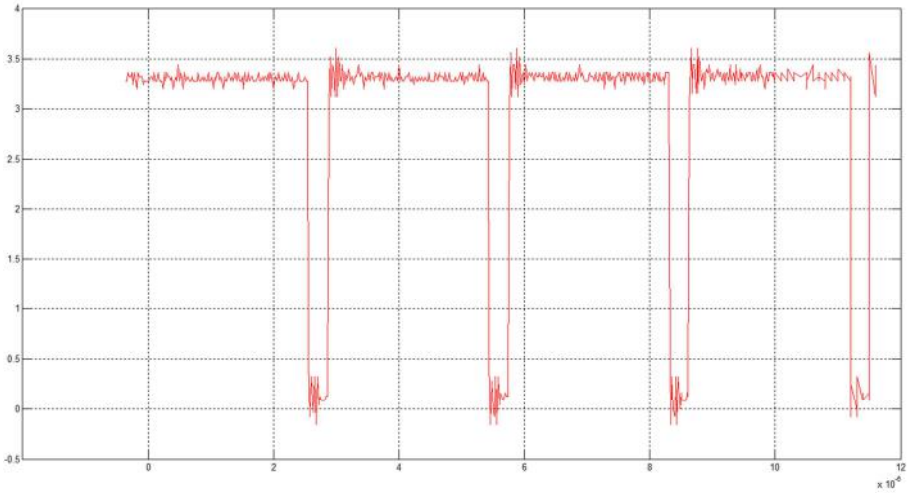


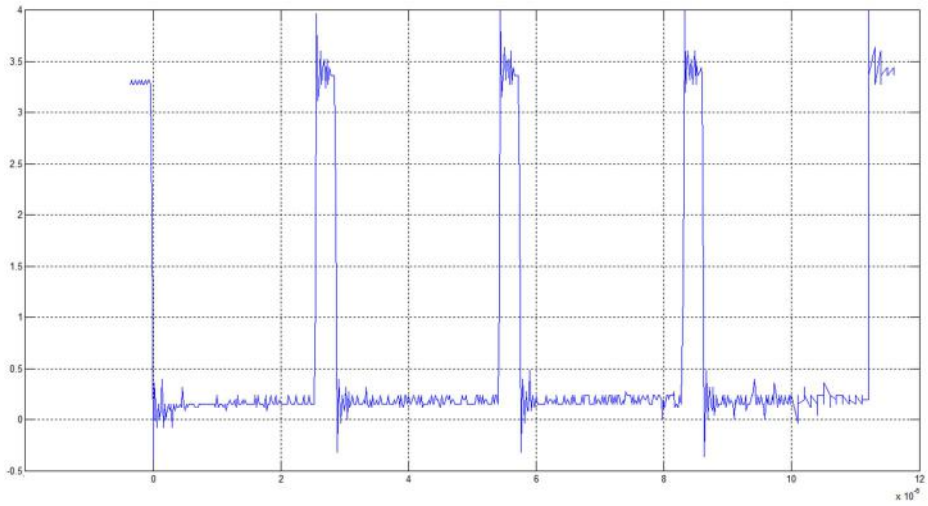
Figura 3.4 - Simulação de teste da comunicação entre FPGA e DAC



(a)



(b)



(c)

Figura 3.5 - Resposta da FPGA para teste da comunicação com o DAC : (a)din; (b)ldac; (c)sync

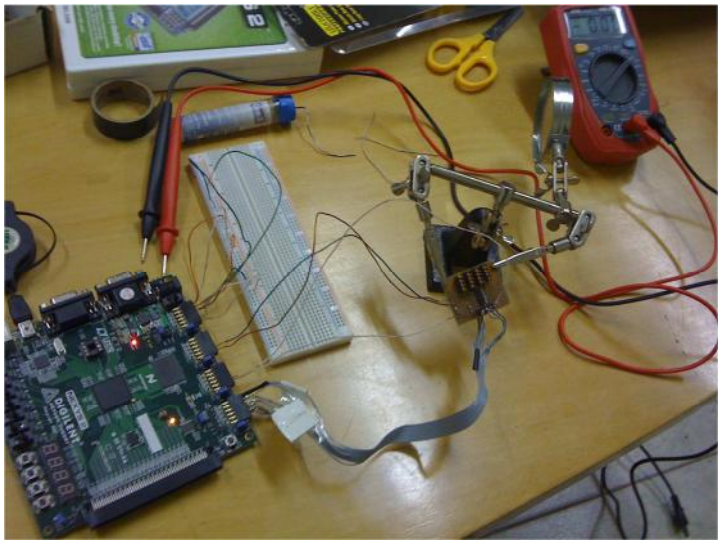


Figura 3.6 - Montagem de teste da comunicação entre FPGA e DAC

3.1.2 – Seleção de sinal e Conversão Analógica Digital

A seleção do sinal vindo dos amplificadores será dada pelo multiplexador analógico previamente explicado. Para controlá-lo foi preciso enviar um código que definisse qual das entradas será selecionada, esta parte será realizada por um módulo da FPGA escrito em VHDL.

Ao implementar o controle do multiplexador analógico 74HCT4051, o CI escolhido para a operação, foi necessário enviar, além do sinal de controle, um sinal de habilitação, de forma que este execute a seleção apenas quando habilitado. Para realizar o teste do código, não foi necessária a montagem de uma placa, diferentemente do

DAC. Pois, neste caso, não se trabalha com sinais rápidos, ou seja, que realizassem rápida transição de nível, algo afetado pelas capacitâncias parasitas presentes numa protoboard, de modo que foi possível utilizá-la para os testes. Assim, para este código, que se encontra no Apêndice A, foi preparada a montagem que pode ser observada na Fig. 3.7. Verificou-se o correto funcionamento do controle do CI, através de medidas utilizando multímetro, foi visto o sinal desejado ser selecionado.

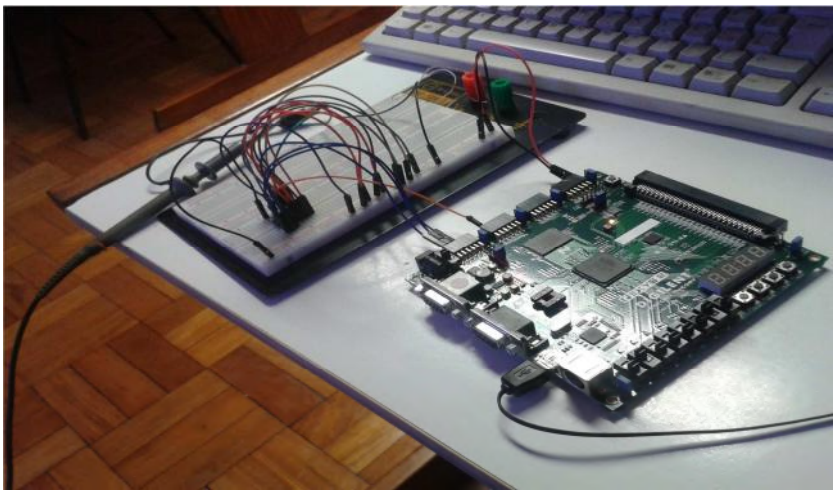


Figura 3.7 - Montagem para teste do controle do multiplexador analógico

Após o sinal ser selecionado, ele deverá ser convertido de analógico para digital, de forma que os dados possam ser enviados pelo front-end. Será utilizada uma placa desenvolvida anteriormente, que contém um ADC AD9214 da Analog Devices, o qual irá receber o sinal analógico selecionado e amostrá-lo numa frequência definida pelo clock da FPGA, no nosso caso, de 50MHz. Assim teremos uma amostra de 8 bits de resolução a cada 20ns, sendo a amostragem dos sinais executada ininterruptamente, com os dados salvos em uma fila circular.

Os dados de um sinal selecionado, se não forem encaminhados para visualização com instrumentos, serão salvos numa memória de dados amostrados com um tamanho fixo de 1Kbyte (1024 bytes de 8 bits), quando o sinal selecionado for relevante. Esta memória será acessível ao operador através de comandos. Desta maneira, os dados armazenados na fila, serão salvos nesta memória fixa para serem lidos quando um sinal de trigger for dado pela lógica de coincidência, explicado no próximo tópico. Este desenvolvimento, no entanto, não foi o foco principal desta fase do projeto do detector de raios cósmicos.

Por se tratar de um código anterior já pronto, se bastou definir as saídas dos códigos implementados a serem utilizadas e que a integração de códigos será realizada posteriormente ao final desta fase do projeto. O caminho dos dados pode ser observado na Fig. 3.8, sendo esta montagem previamente testada e aprovada [10].

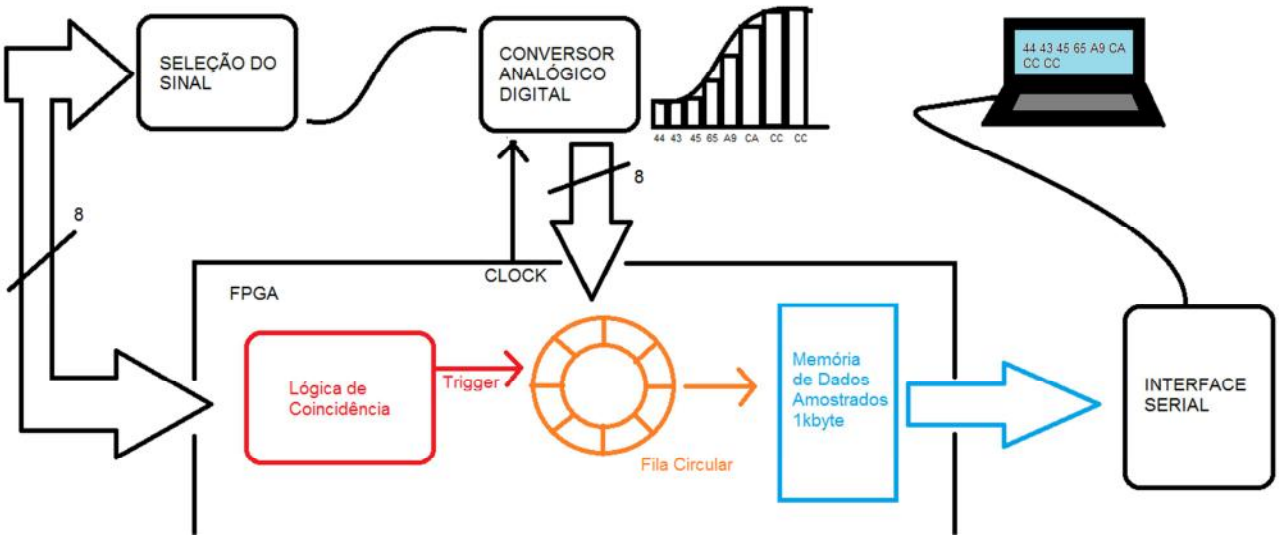


Figura 3.8 - Caminho do sinal selecionado até o front-end

3.2 – Coincidência de Sinais

Para a definição de sinais coincidentes, teremos duas etapas desenvolvidas. Na primeira, ao se detectar um primeiro sinal, o qual se denominou primeiro evento, por um tempo determinado serão aguardados outros eventos, isto é, a chegada de outros sinais, de modo que ao final do intervalo se obtenha um vetor de eventos ocorridos. Esta é denominada a etapa da aquisição numa janela de tempo. Além disso, com o início desta janela de tempo, o sinal selecionado digitalizado será armazenado na memória de dados amostrados, definindo o sinal de trigger explicado no tópico anterior.

Em seguida, este vetor de eventos ocorridos deve passar por uma filtragem realizada por uma lógica combinacional, sendo, em seguida, armazenado de maneira cronológica numa memória interna do sistema, de modo que, quando o operador desejar, ele tenha acesso a eles. Esta é a etapa da memória interna.

Além desses pontos realizados, entre os eventos é necessário verificar os atrasos. Isso se deve ao fato de que parte dos eventos é proveniente de uma terminação da fibra cintilante e se propagará até a outra terminação. Desta forma, a cada detecção de incidência de múon, são gerados dois eventos. Como a velocidade da partícula na fibra é

conhecida (visto no Capítulo 2), será possível determinar a posição em que a partícula atingiu a fibra ao se verificar os atrasos. Devido à complexidade da lógica para se realizar esta função, além da necessidade de alta frequência por se utilizar um clock de 400MHz, este desenvolvimento será realizado após esta etapa do projeto, não entrando no escopo atual.

Para os testes da coincidência, será necessário uma velocidade de clock, pelo menos 4 vezes maior que a velocidade de aquisição, de forma a simular a chegada de sinais rápidos que representem os eventos detectados. Para isso, serão utilizados DCMs nativos da FPGA, de modo a gerar um clock mais veloz que o dado pelo oscilador da placa utilizada para o desenvolvimento, e um contador para a definição de sinais que simulem a incidência de múons nas fibras, atuando, assim, como um gerador de sinais interno da FPGA. Os códigos VHDL que realizam estas operações podem ser observados no Apêndice A, de maneira que seja possível ver mais a fundo o funcionamento dos módulos.

3.2.1 – Aquisição numa Janela de Tempo

Na etapa da aquisição, inicialmente se deve esperar por um primeiro evento. Existirão até 8 eventos, sendo cada um deles proveniente de uma terminação de uma fibra, separando a aquisição para cada uma das terminações. Logo, ao se receber um evento de uma terminação em um módulo de aquisição, o evento é propagado até a outra terminação e recebido em outro módulo, definindo, assim, para cada módulo uma entrada de 4 bits.

Os 4 bits de entrada do módulo devem ser monitorados, de maneira idêntica, para a detecção do primeiro evento. Se for detectado em qualquer uma delas deve se armazenar o evento e iniciar uma janela de tempo em que os eventos serão considerados relevantes.

Este módulo deverá trabalhar de maneira assíncrona, isto é, não dependerá de um sinal de clock no tempo, afinal a ocorrência de um evento não estará relacionada a nenhum dos sinais do sistema, ele pode ocorrer a qualquer instante. Apesar disso, é necessário realizar uma janela de tempo para se receber sinais relevantes em que se verifiquem coincidências, sendo necessário um contador que implemente esta etapa, o único síncrono, pois irá determinar a duração da janela baseado nas transições do clock.

Foi realizado, então, um monitoramento assíncrono das 4 entradas com uma porta lógica OU (1 na saída se houver uma ou mais entradas em 1), de modo que, quando a saída da porta estiver em 1, se inicia a operação do contador síncrono. A janela de tempo irá durar até o fim da contagem, sendo sua largura dependente do número máximo da contagem, $CNT_{MÁX}$, e do período do clock , T_{CLK}, como pode ser observada na expressão (3.2).

$$JanelaContador = T_{CLK} \star (CNT_{MÁX} - 1) \quad (3.2)$$

O primeiro sinal e os que ocorrerem durante a janela de tempo, deverão ser salvos numa memória temporária interna que foi implementada com flip-flops, onde cada um funcionará como uma célula de memória. Assim, se o flip-flop estiver habilitado, ao ocorrer um evento, se define 1 na saída deste, senão permanecerá em 0.

Ao final da janela, um sinal para a memória interna do sistema é enviado de modo que os conteúdos dos flip-flops sejam lidos e, em seguida, retornem para seus valores iniciais, permitindo que se possa aguardar novos eventos a serem salvos. A montagem dos módulos pode ser observada na Fig. 3.9 e as simulações para cada um dos módulos separados e em conjunto na Fig. 3.10.

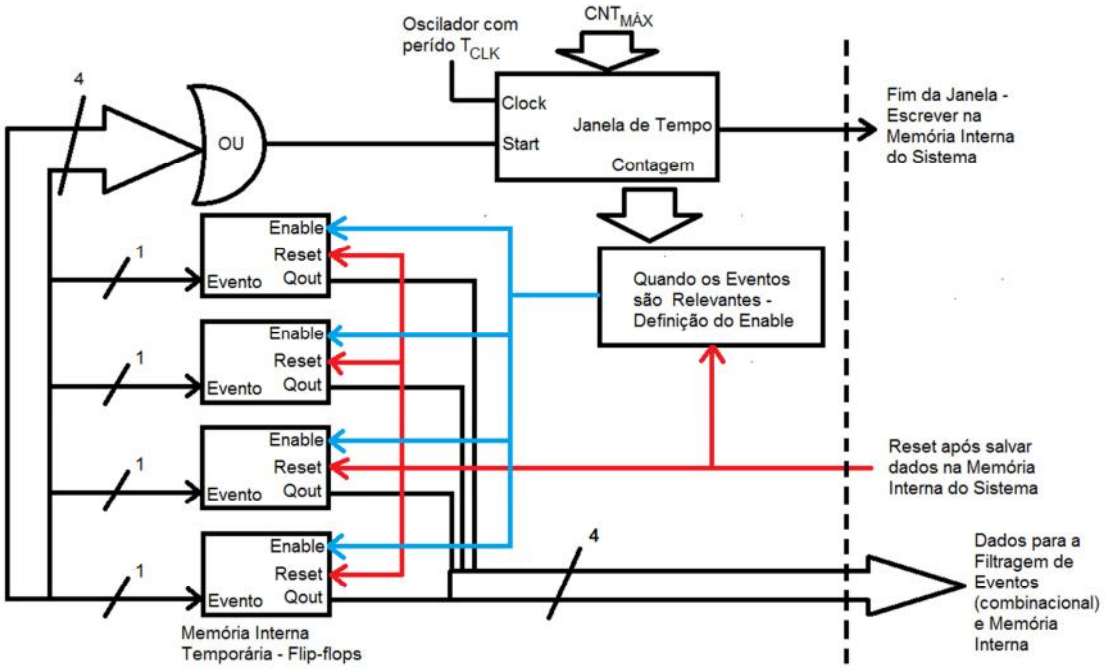
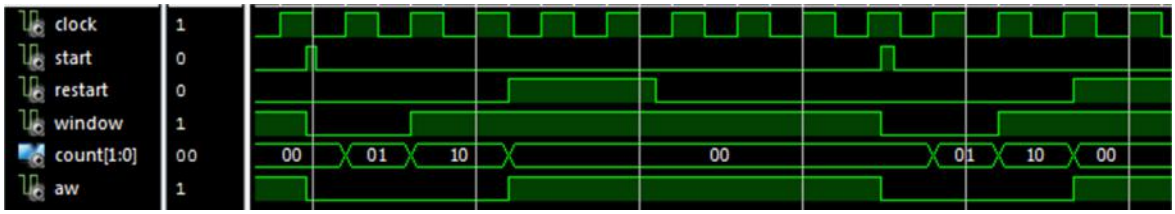
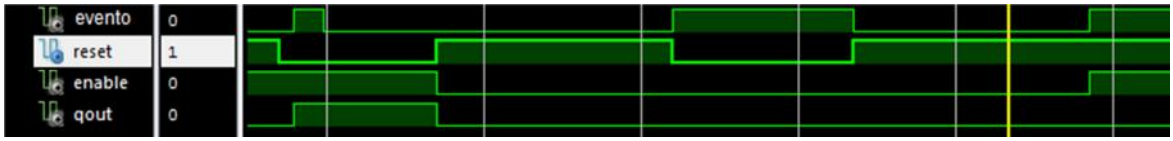


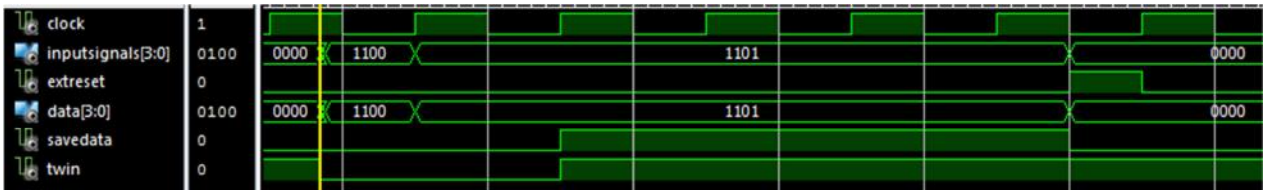
Figura 3.9 - Módulos ligados para aquisição de eventos relevantes



(a)



(b)



(c)

Figura 3.10 - (a) Janela de Tempo; (b) Flip-flop; (c)Aquisição de eventos relevantes

A partir de dados levantados em laboratório, sabe-se que uma janela de tempo com 20ns de largura é suficiente para realizar uma aquisição. Como o clock utilizado foi de 50MHz, tem-se um T_{CLK} de 20ns, definindo um $CNT_{MÁX}$ igual a 2, para se obter o tamanho necessário da janela de tempo. Ao observar a simulação da janela de tempo na Fig.3.10(a), nota-se que a janela possui uma largura maior do que a calculada. Isso se deve a combinação do monitoramento assíncrono com o contador síncrono. Como o primeiro evento pode ocorrer a qualquer instante e o contador só irá alterar o valor com o clock, o tempo depois da detecção do primeiro evento e antes da primeira transição com o contador na janela iniciada, o qual se denominou Δt, irá ser acrescido.

Modifica-se, então, a expressão (3.2), que apenas determinará o tamanho mínimo da janela, obtendo-se a expressão (3.3), com o valor completo da largura. O valor de Δt será aleatório e limitado entre 0ns, imediatamente antes da primeira transição do clock após o inicio da janela de tempo, e 20ns, imediatamente após a última transição do clock fora da janela de tempo, desta forma, a largura da janela será limitada pelo intervalo dado em (3.4).

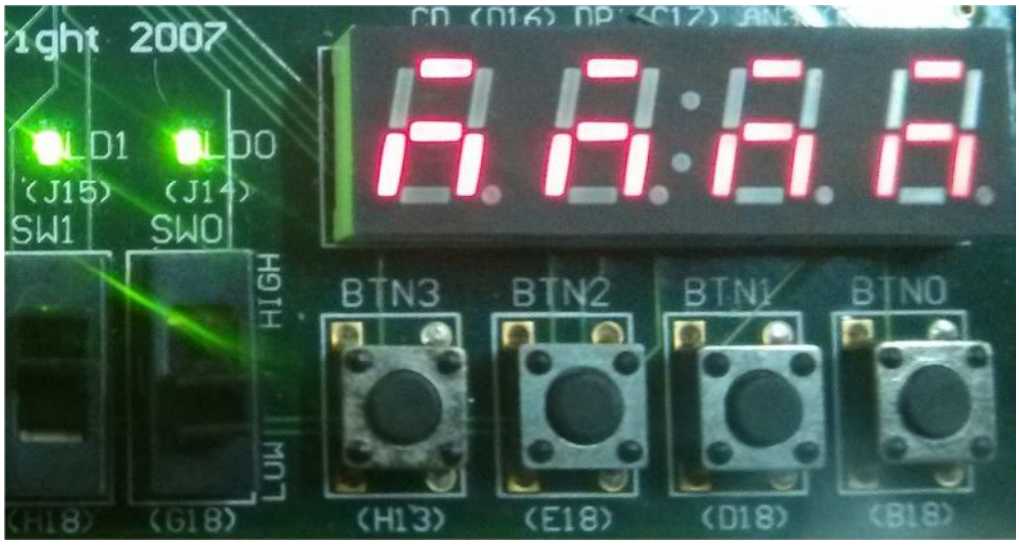
$$Janela = [T_{CLK} \star (CNT_{MÁX} - 1)] + \Delta t \quad (3.3)$$

$$20ns \leq Janela \leq (20ns + \Delta t) \leq 40ns \quad (3.4)$$

Na Fig.3.10(b) pode se ver como o flip-flop dependerá tanto do sinal de habilitar (enable), quanto do sinal de limpar a saída (reset), para se salvar um evento. Deste modo, sabendo as ligações vistas na Fig. 3.9, observa-se na Fig. 3.10(c), como apenas os eventos relevantes serão salvos, a duração da janela de tempo até que a informação após passagem pela filtragem de eventos seja salva na memória interna e ao final o retorno para o monitoramento.

Com as simulações realizadas, utilizou-se o gerador de sinais simulado internamente na FPGA para se testar o módulo. Desta forma, 3 eventos seriam recebidos para cada módulo com 4 entradas, sendo apenas dois destes relevantes. Assim, na saída teríamos um vetor de 8 bits formado por dois de 4 equivalentes, de modo a simular os eventos capturados por um módulo e os mesmos propagados para a outra terminação, sendo também capturados. Para a visualização, se utilizou o display de 7 segmentos presente na placa de desenvolvimento e leds de modo a visualizar alguns sinais relevantes, assim como um botão para a entrada do sinal que limpa os flip-flops. O gerador definiria os sinais relevantes os bits 2 e 4 da entrada, e como irrelevante o bit 3, obtendo assim na saída o vetor 0xAA (10101010). Como pode ser observado na Fig. 3.11(a), a saída obtida foi a esperada, além de se observar o sinal para se salvar os dados em 1, como também era desejado. Em seguida, se apertou o botão de reset, observando o retorno para os valores iniciais, como visto na Fig. 3.11(b).

Este vetor passará, então, pela lógica combinacional de filtragem de eventos, para que então, os dados de coincidência simples, dupla, tripla e quádrupla sejam salvos na memória.



(a)

(b)

Figura 3.11 - Display e LD0: saída ; LD1: salvar; BTN0: reset; (a) resultado obtido; (b) após a limpeza do resultado

3.2.2 – Memória Interna

A memória interna precisa ser organizada de maneira que a ordem de chegada dos vetores de eventos seja mantida na retirada destes, ou seja, o primeiro evento a ser salvo na escrita da memória pelo módulo de aquisição na janela de tempo, deverá ser o primeiro a ser lido pelo operador, quando este desejar. Assim, a memória interna deverá funcionar como uma fila do tipo FIFO.

Para a FIFO, decidiu-se utilizar um algoritmo de fila circular, já conhecido, com verificação de fila cheia e fila vazia. Desta maneira, ao se decidir escrever na fila, só poderá executar a operação se ela não estiver cheia, além disso, ao se tentar ler quando estiver vazia, também não será realizada a operação. Estas informações devem ser enviadas ao operador, de modo que ele possa agir da maneira correta para a situação, isto é, se desejar salvar algum novo evento e a fila se encontrar cheia, deverá ler os dados que estão salvos de modo a liberar espaço. Além disso, como a fila será circular, ao se atingir o endereço de maior valor, o seguinte, a ser escrito, será o inicial. Desta forma, a FIFO se comportará como pode ser observa na Fig. 3.12.

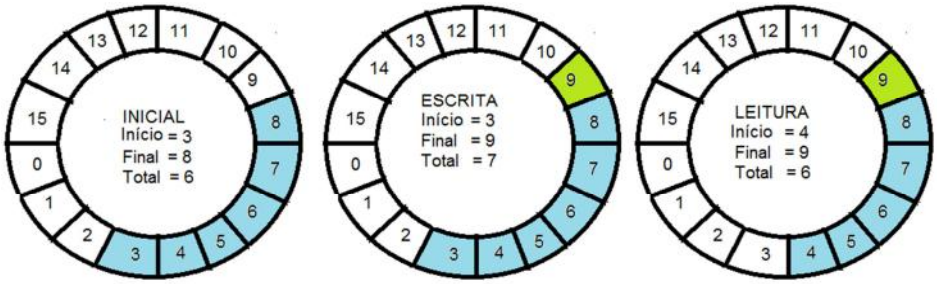


Figura 3.12 - Funcionamento do algoritmo da FIFO

Pode se observar que numa operação de escrita, se acrescenta uma posição no final da fila e incrementa o total de endereços ocupados. Já na leitura se retira o dado que está no início da fila e decrementa o total de endereços ocupados. Este valor definirá se a fila está cheia, quando for igual a 16, ou vazia quando for igual a 0. Como dito anteriormente, pela fila ser circular, se o final dela for o endereço 15 e ocorrer uma operação de escrita, o próximo preenchido será o endereço 0, similarmente ocorrerá para a leitura.

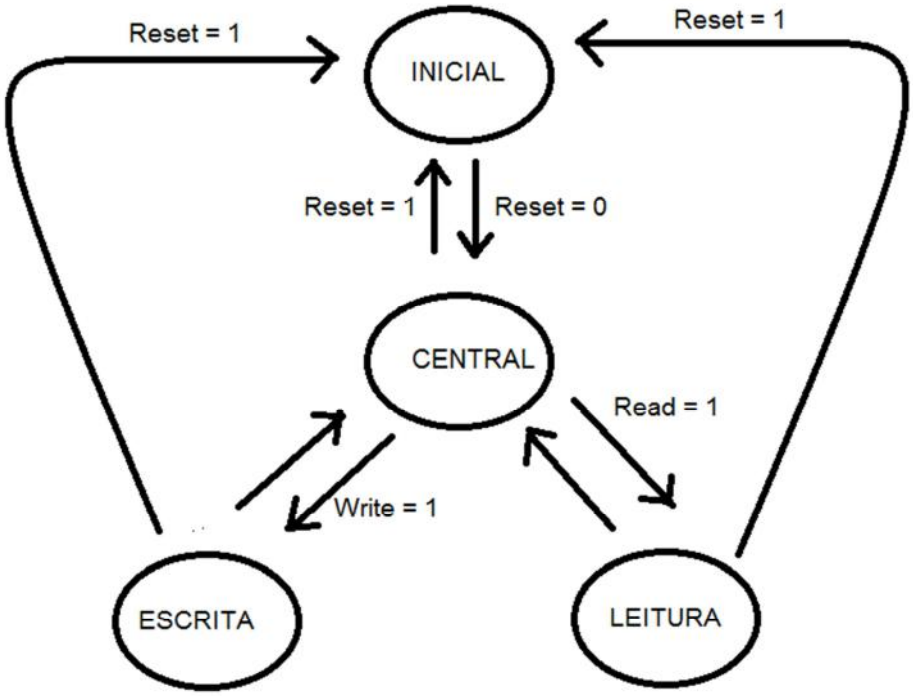


Figura 3. 13 - Máquina de estados da FIFO

Para implementar este algoritmo, se definiu uma máquina de estados. Nela, tem-se o estado inicial, onde são definidos os valores iniciais da fila, caso haja um sinal de Reset, a qualquer momento, leva-se a FIFO para este estado. Em seguida, vamos para o estado central, que fica aguardando os comandos, definindo se irá para escrita, caso

recebe um sinal de Write e a fila não esteja cheia, ou se irá para leitura, caso recebe um sinal de Read e a fila não esteja vazia. Independente de qual operação for executada, logo após a execução desta, o estado volta ao central, a menos que um Reset ocorra. A definição da máquina pode ser vista na Fig. 3.13.

Tanto a leitura quanto a escrita terão a duração de um pulso de clock, de modo que nesse instante sua entrada é lida e salva no próximo endereço após o final da fila, caso seja uma escrita, ou o início da fila é lido e seu valor altera a saída, caso seja leitura. Durante estes eventos, um sinal de operação Done, indica que está ocorrendo uma, isto é, muda seu valor até que chegue ao final a operação, indicando que ela foi executada. O funcionamento com um teste pode ser observado na simulação presente na Fig. 3.14.

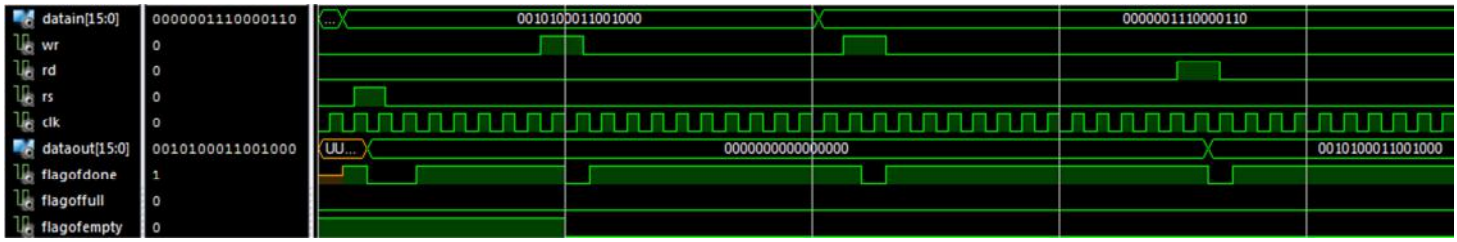


Figura 3.14 -Simulação do funcionamento da FIFO

Para a verificação da FIFO na placa, vários testes foram realizados. Assim como para a aquisição numa janela de tempo, foi utilizado o gerador simulado, porém para a memória interna, ele foi utilizado para definir vetores de entrada e o sinal de escrita e de leitura, de modo que somente se observasse a saída e os flags utilizando o display e os leds. Assim os seguintes testes foram realizados:

a. Teste de leitura e escritas únicas: um sinal de escrita e/ou um sinal de leitura;
b. Teste de escritas e leituras compleatas: escrever e/ou ler até atingir o número máximo e/ou mínimo de dados presentes na fila;
c. Teste de escrita e leituras alternadas: visualizar o comportamento para operações fora de ordem

Para todos os testes, os resultados obtidos foram positivos, sendo um dos Testes c observado na Fig.3.15, demonstrando que após várias escritas e leituras, ainda há dados na fila não completa, sendo visto o último dado lido que foi 0xE0 (11100000).

Figura 3.15 - Display e LD0: saída; LD1: flag de cheio; LD2: flag de vazio

3.2.3 – Lógica de Coincidência

A lógica de coincidência reunirá os dois módulos demonstrados acima, de modo que sejam armazenados apenas os eventos relevantes, passando-os pela lógica combinatória e os salvando na memória interna do tipo FIFO, de maneira sincronizada. As ligações podem ser observadas na Fig. 3.16.

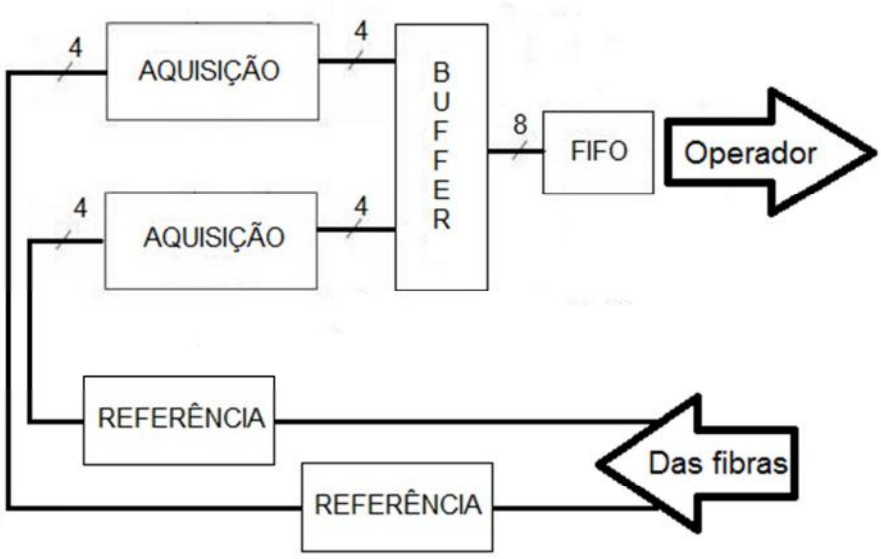


Figura 3.16 - Lógica de Coincidência

Para a sincronização dos módulos, foi implementada uma máquina de estados cujas passagens irão depender dos sinais de escrita e de leitura da FIFO. Estes por suas vezes dependerão, respectivamente, do final da aquisição dos dados relevantes e de um sinal externo oriundo da comunicação com o front-end. Além disso, será necessário ao final da aquisição realizar a limpeza dos flip-flops, liberando estes para novo

monitoramento, reunir todos os dados dos dois módulos de aquisição e salvá-los na FIFO, sendo utilizado, com este intuito, o buffer intermediário que também irá conter a lógica combinatória para verificação das coincidências, como visto na figura acima.

A máquina de estados irá definir a ordem e forma como os sinais de controle dos módulos irão atuar, sincronizando a operação. Esta pode ser observada na Fig. 3.17.

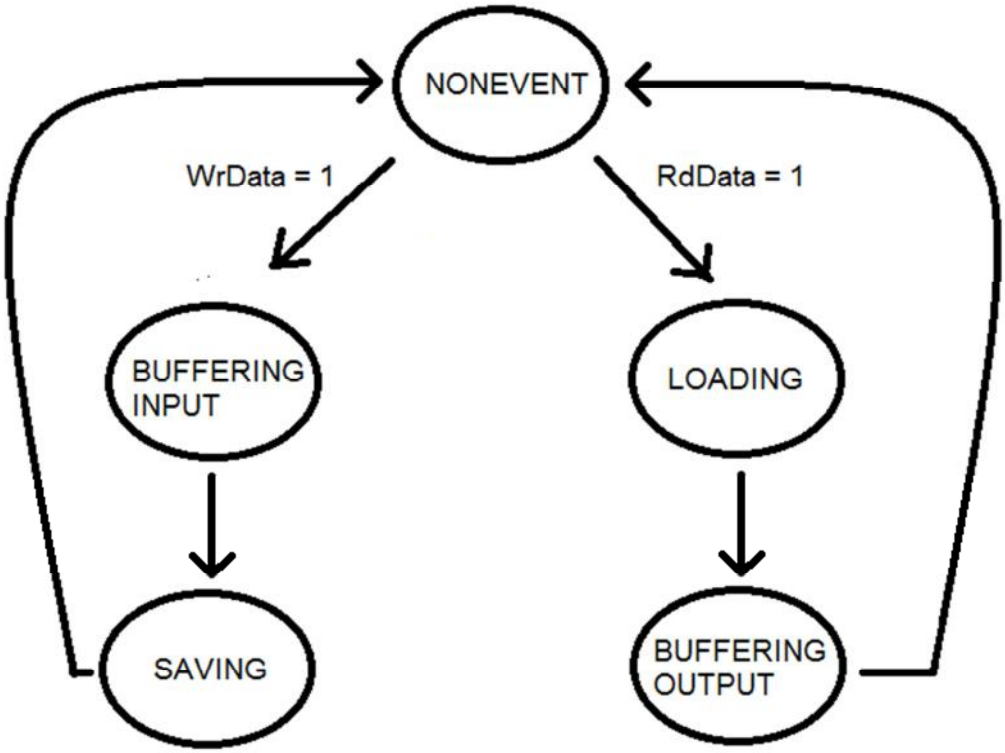
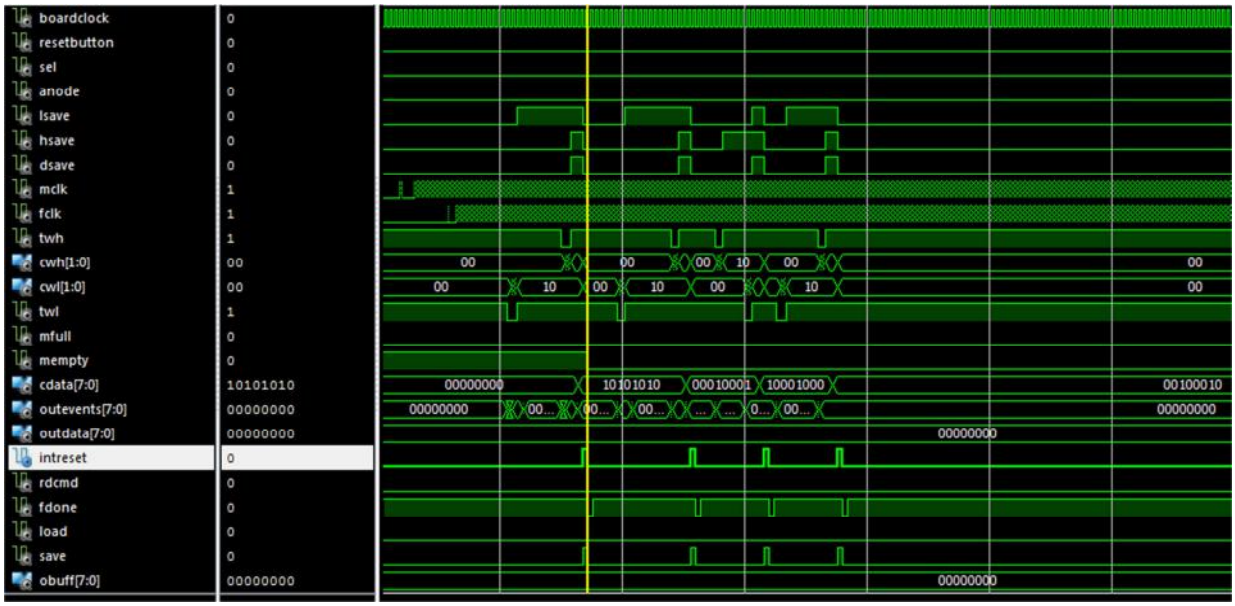


Figura 3.17 - Máquina de estados para sincronização entre a aquisição e a FIFO

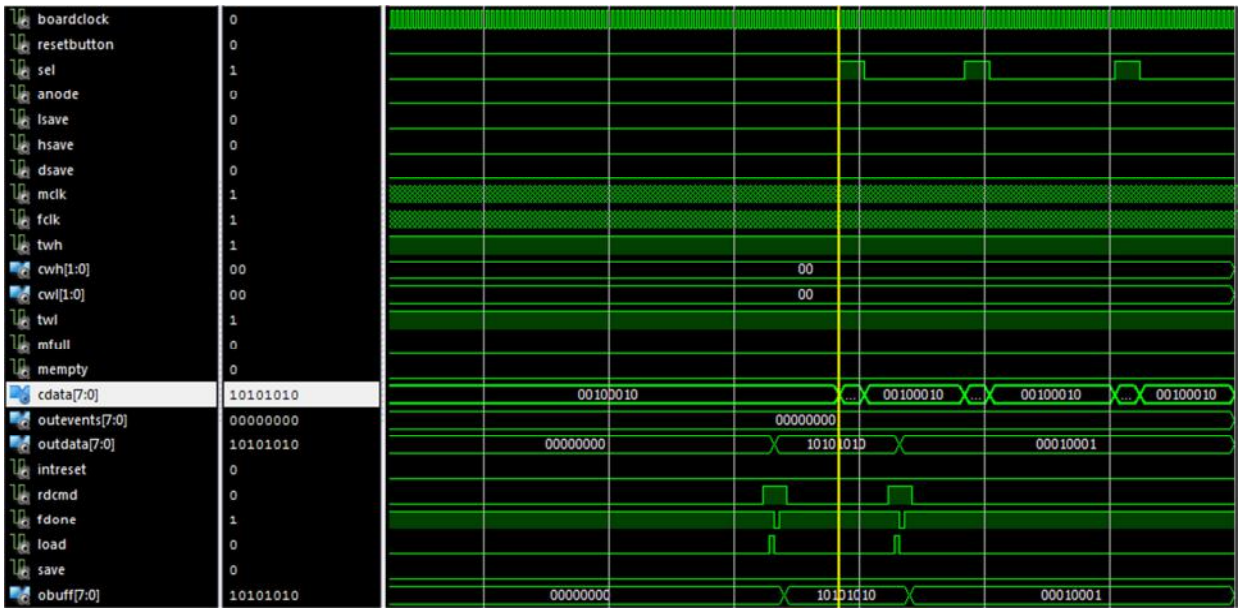
No estado inicial, se aguarda os sinais de comando. O sinal de escrita dependerá do final da aquisição dos dois módulos, ou seja, apenas quando ambos tiverem feito a aquisição os dados serão salvos no buffer passando pela filtragem dos eventos e os flip-flops limpos para um novo monitoramento, sendo, em seguida, salvos na FIFO, sem perda de dados. Já o sinal de leitura será externo, vindo da interface serial, explicada no próximo tópico. Isto se deve ao fato que a operação de leitura da memória interna será muito mais lenta, necessitando aguardar o envio completo dos dados para o operador antes de retornar para o estado inicial. Este será o grande gargalo de toda a operação, que limitará a taxa de recebimento da combinação de eventos.

Com a implementação da máquina de estados, era preciso realizar uma simulação para verificar o funcionamento. Para isso foram utilizados os DCMs e um

contador para simular o gerador de sinais, como explicado anteriormente. Assim, o resultado das simulações pode ser visto na Fig. 3.18.



(a)

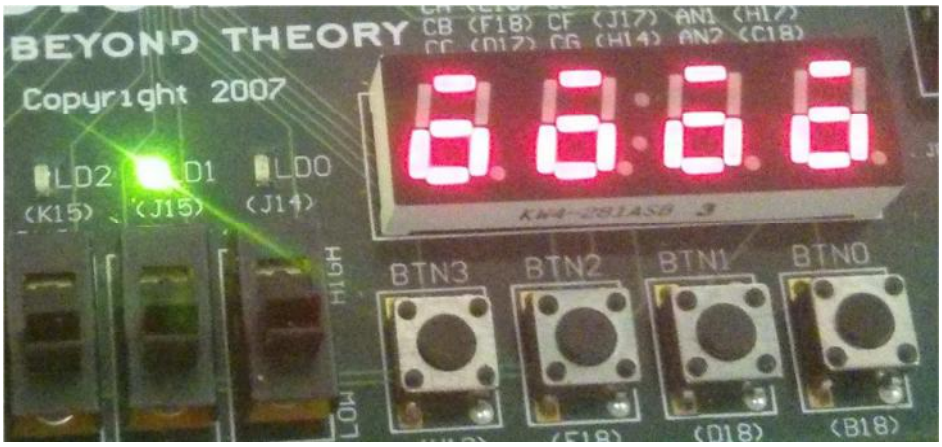


(b)

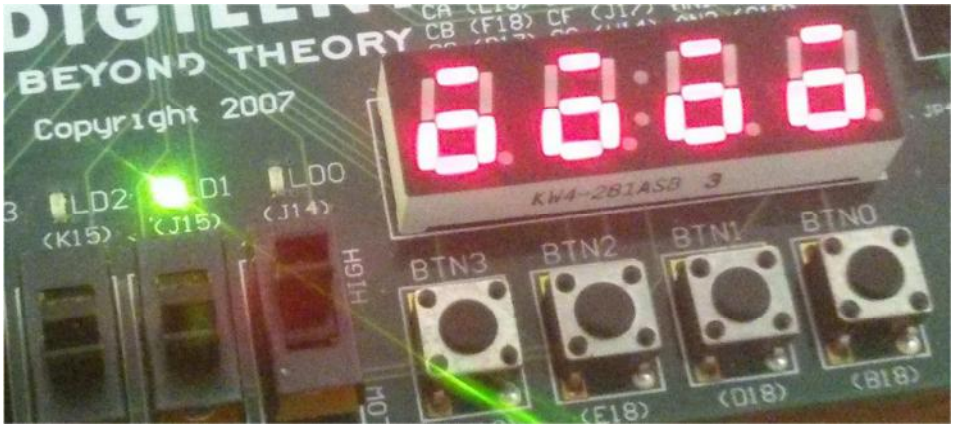
Figura 3.18 - (a) Escrita na FIFO de 4 aquisições; (b) Leitura da FIFO de duas aquisições

Obtendo o resultado positivo com as simulações, foram feitos os testes na placa de desenvolvimento. Novamente foi utilizado o gerador simulado na FPGA, utilizando os displays de 7 segmentos e leds presentes na placa, de modo que fosse possível ver o vetor de bits recebido na janela de tempo e lido da memória interna, além dos flags desta, que definiriam que a operação foi executada corretamente. Neste teste, se simula

a aquisição de 4 eventos e a leitura destas 4 entradas, visualizando o valor do buffer e na saída da FIFO, tendo que estas devem ser as mesmas. Foi utilizada uma chave para selecionar qual dos valores será mostrado no display, desta forma, o resultado obtido e visualizado, foi o último vetor de eventos recebido 0x22 (00100010). Assim, na Fig. 3.19, se observa tanto o buffer, quanto a saída da FIFO com os valores corretos, assim como os flags esperados.



(a)



(b)

Figura 3.19 - LD1: flag de vazio; LD2: flag de cheio; (a)Chave seletora em 0: Buffer no display e LD0; (b)Chave seletora em 1: Saída no display e LED0

3.3 – Comunicação com o Front-End

Para que o operador possa definir as configurações e ler os dados da memória interna do sistema, foi necessário implementar uma interface de comunicação entre o back-end e o front-end.

A primeira definição da comunicação foi o meio físico, sendo escolhido um cabo USB, cujas portas são comuns em todos os computadores atuais. Em seguida, era preciso definir como seria a comunicação da FPGA com o cabo USB, a interface serial. Para isso, verificou-se que a ligação direta seria complexa e não confiável, de modo que se tornou necessário um intermediário para realizar esta interface entre os dois. Foi utilizado, então, um microcontrolador que receba e envie os dados. Assim, entre o microcontrolador e a FPGA se utilizou um protocolo SPI, simples e confiável, de maneira a economizar ligações, com as conversões serial/paralelo para configuração e paralelo/serial para devolução do dado requisitado, serem executadas dentro de módulos da FPGA, para, em seguida, o microcontrolador enviar os dados de acordo com o protocolo USB.

A comunicação deve ocorrer de acordo com o protocolo SPI, desta forma, serão necessários quatro sinais entre a FPGA e o microcontrolador:

- SCK, clock de comunicação, deve sair de quem comanda o inicio e o fim da comunicação (mestre) para quem é comandado (escravo);
- MOSI, linha serial de comunicação, onde o mestre escreve e o escravo lê;
- MISO, linha serial de comunicação, onde o mestre lê e o escravo escreve;
- CS, habilitador da comunicação definido pelo mestre, sempre ativo no nível 0.

Como o microcontrolador será o responsável por estar em comunicação direta com o operador através do computador conectado ao cabo USB, este será o mestre na comunicação e a FPGA o escravo, definindo um esquema que pode ser observado na Fig. 3.20.

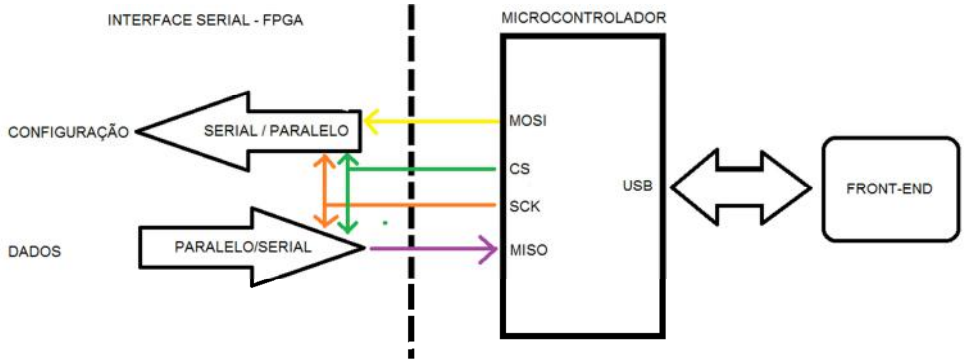


Figura 3.20 - Esquema de comunicação da FPGA para o micorontrolador e para o front-end

Outro ponto do protocolo SPI será a configuração deste, definida no mestre, mas que deve ser conhecida pelo escravo, de modo que atue na mesma. Existem quatro modos de operação, dependendo de duas características: a polaridade do clock de comunicação, que define o valor base; e a fase do clock de comunicação, que define em qual transição do clock, de 0 para 1 ou de 1 para 0, os dados se propagam e em qual se capturam. Os modos podem ser observados na Tabela 3.3.

Modos	Polaridade	Fase	Captura o dado	Propaga o dado	Base
0	0	0	subida	descida	0
1	0	1	descida	subida	0
2	1	0	descida	subida	1
3	1	1	subida	descida	1

Tabela 3.3 - Modos de operação do protocolo SPI. Fonte: [14]

Definindo que o modo de operação utilizado será o 0, foi possível definir a interface serial da FPGA, que deve executar então dois tipos de comandos oriundos do mestre. Para que a interface defina as operações necessárias automaticamente, ela foi implementada de acordo com uma máquina de estados que sempre irá executar duas comunicações consecutivas de 8 bits, logo 2 bytes.

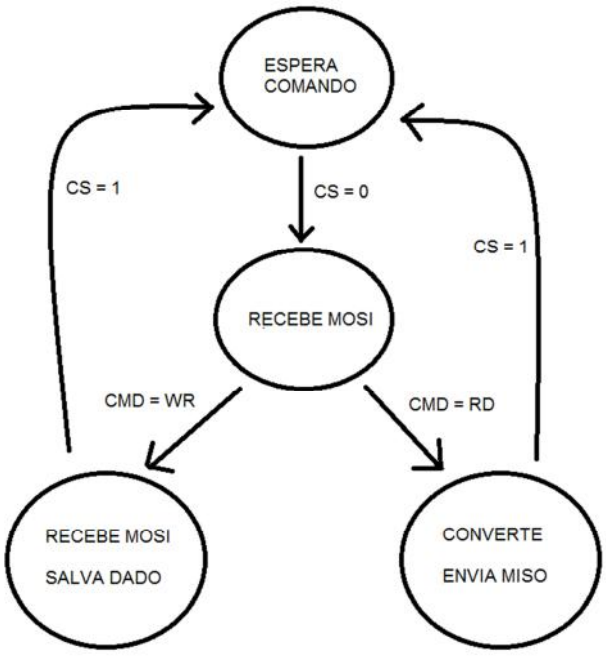


Figura 3.21 - Máquina de estados da interface serial

O primeiro byte irá definir se a operação será uma escrita, onde irá receber um dado pela porta MOSI, convertendo de serial para paralelo de acordo com a subida de SCK e salvando o byte para ser utilizado na configuração do sistema, ou uma leitura, onde se lê um byte salvo das operações passada, convertendo-o de paralelo para serial de acordo com as descidas de SCK e o envia pela porta MISO. O seu comportamento pode ser verificado de acordo com máquina de estados da Fig. 3.19 e pela simulação na Fig. 3.20. Resta demonstrar o funcionamento do microcontrolador e os testes realizados, sendo apresentado no próximo capítulo.

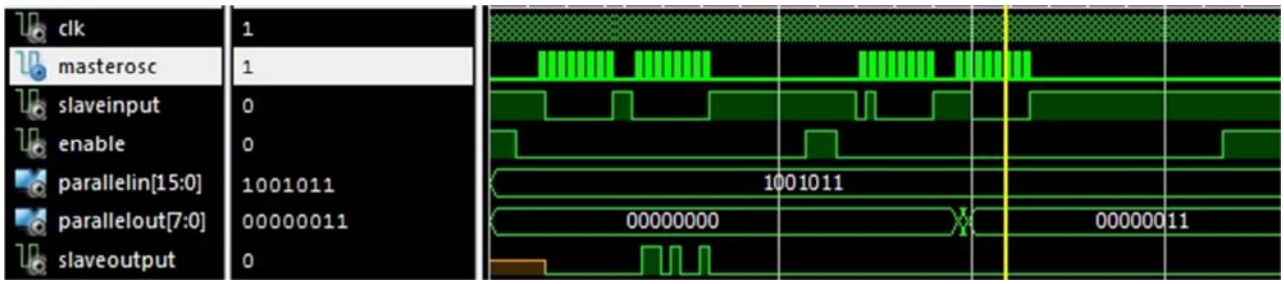


Figura 3.22 - Simulação do funcionamento da interface serial

Capítulo 4
Front-end e Integração: comandos e controle

4.1 – Comunicação com o Back-End

O primeiro ponto de definição do front-end será dado pela comunicação com o back-end. Como dito anteriormente, esta se dará por meio de um microcontrolador conectado a uma interface serial da FPGA, comunicando-se pelo protocolo SPI e com o computador utilizando cabo USB.

A escolha para este microcontrolador levou em conta estes aspectos, além do que fosse necessário ser de fácil acesso e manuseio. Com isso, escolheu-se por utilizar uma placa Arduino Duemilanove, contendo um microcontrolador ATmega328P da Atmel.

O Arduino já possui seus próprios comandos para ler e escrever numa porta USB, assim como para ler e escrever seguindo o protocolo SPI, além de ter sua própria janela para monitorar e controlar a comunicação. Com esta ferramenta em mãos, um primeiro firmware para se comunicar o computador com a FPGA foi preparado, tanto no Arduino, quanto na FPGA.

Neste teste, o operador envia dois tipos de comandos pela porta USB. Um destes será o de leitura, onde se lê um byte enviado serialmente pela FPGA, da mesma forma como um dado salvo na memória interna do sistema, será lido. O outro comando será de escrita, se enviando e salvando dentro de um registro de 8 bits a serem usados para configuração do sistema, usando displays de 7 segmentos para observar qual vetor foi recebido.

O Arduino nos possibilita definir as configurações já explicadas para a comunicação SPI além de definir o clock da comunicação, sendo o primeiro passo a configuração desta comunicação. Em seguida, cada comando digitado pelo operador é quebrado em caracteres onde se avaliam seus valores ASCII, que irão definir qual foi a

entrada do operador. Com esta definição o comando é enviado e, ou se espera para ler o que será enviado pela FPGA, ou se escreve novamente o vetor de bits que será mostrado no display.

Para completar a construção da comunicação entre o Arduino e a FPGA, são necessários 3 divisores resistivos formados por resistores de 2,2kΩ e 3,3kΩ entre os sinais que irão do Arduino para a FPGA. Isto se deve, pois apesar das duas placas serem alimentadas por cabos USBs conectados ao computador, o Arduino utiliza uma tensão de alimentação de 5V e a FPGA uma de 3,3V, de modo que é necessário baixar o nível de tensão vindo do Arduino para que seja possível ajustar os níveis lógicos de tensão para a comunicação sem exceder a tensão máxima de entrada da FPGA.

Montada uma placa de testes para os divisores, foi testada a comunicação, obtendo o resultado mostrado na Fig. 4.1, onde se tem a entrada dos comandos com os resultados na tela do computador através da ferramenta disponibilizada pelo ambiente de desenvolvimento do Arduino e na Fig. 4.2 com a foto demonstrando os displays escritos pela comunicação.

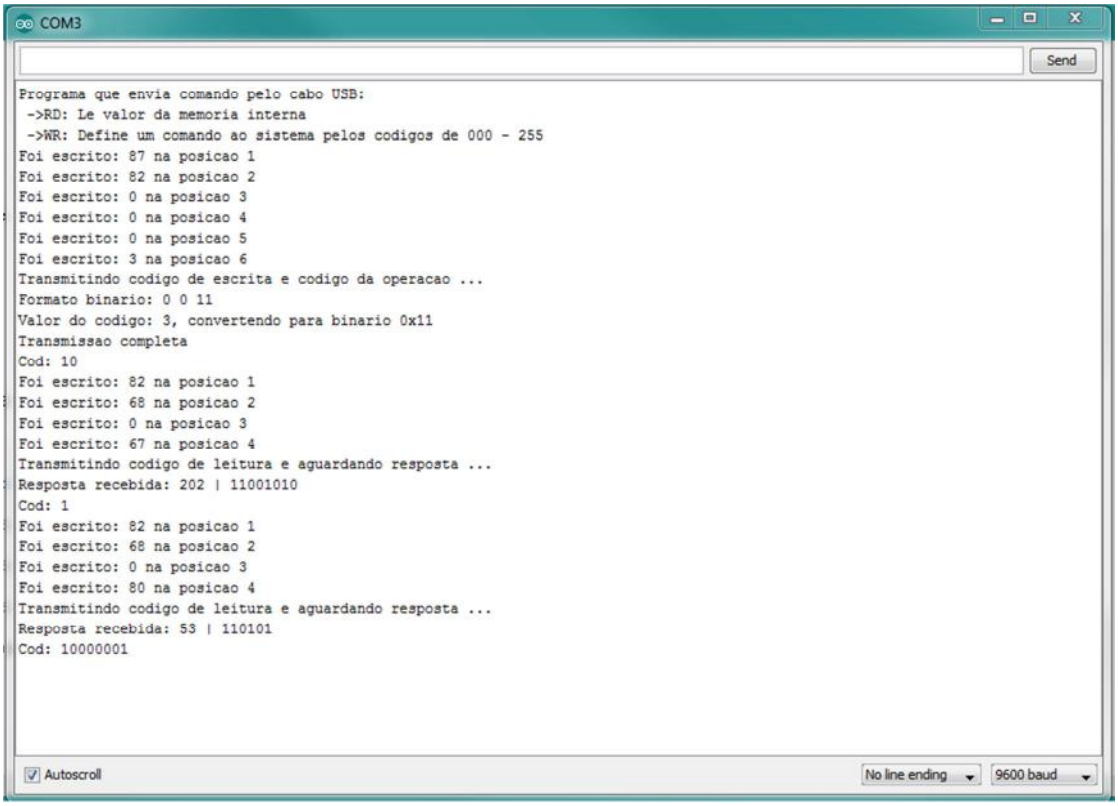


Figura 4.1 - Monitoramento da comunicação entre front-end e back-end

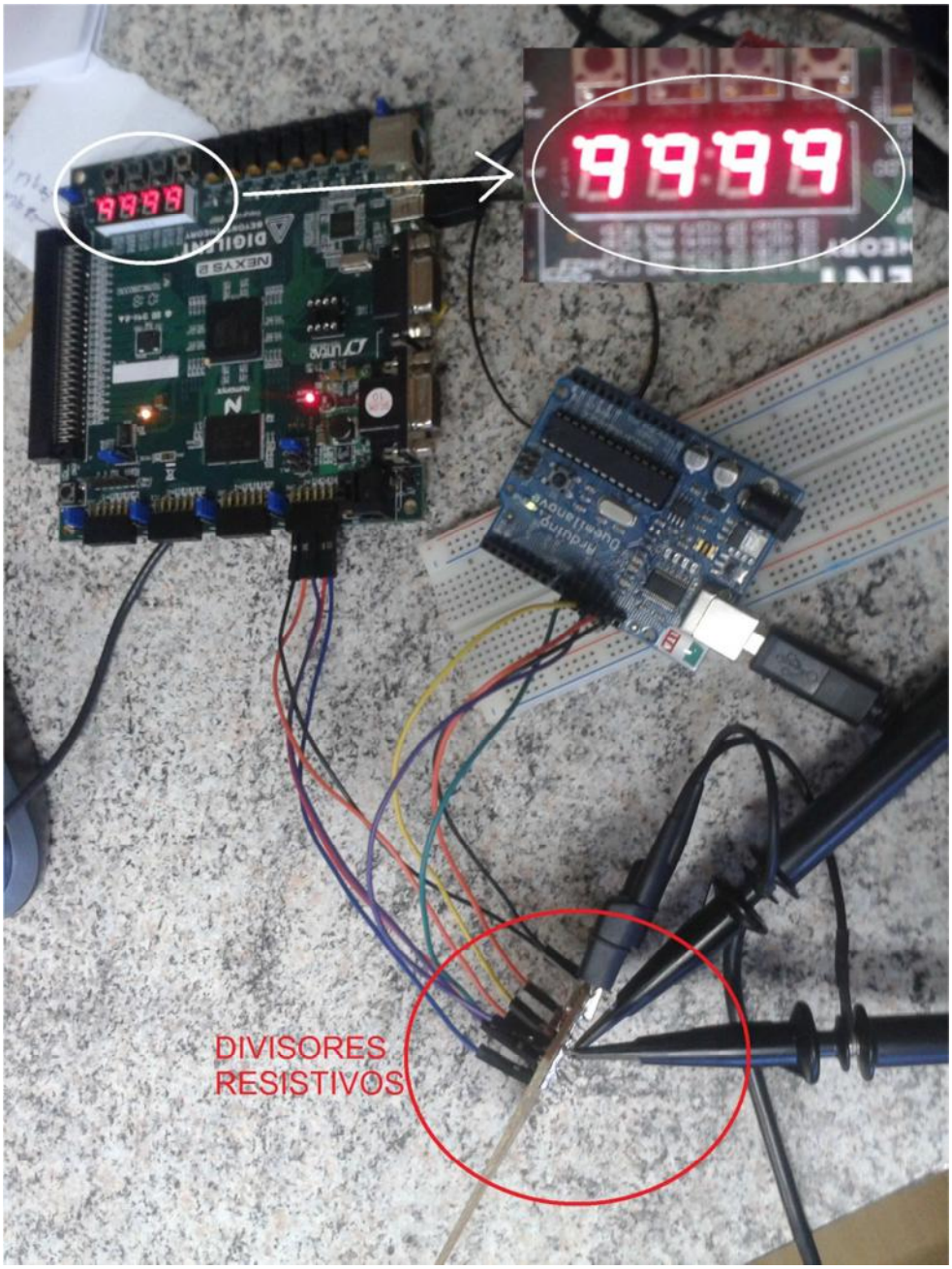


Figura 4.2 - Montagem para o teste da comunicação

Pelas figuras pode se observar que os testes foram bem sucedidos, pois quando o Arduino enviou o valor 0x03 em hexadecimal para FPGA, a interface serial deveria definir este valor para o display de modo a apagar os dois leds correspondentes. Além disso na leitura, que é executada duas vezes, o primeiro byte a ser lido foi o 0xCA (11001010) e o segundo byte o 0x35 (00110101). Com este algoritmo para a comunicação completo, para o front-end resta a definição dos comandos e de como estes serão executados, sendo feito durante a integração do sistema.

4.2 – O sistema integrado

Com os módulos da lógica de coincidência, definição da referência pelo controle do DAC, seleção de um sinal analógico pelo multiplexador analógico (MUX) e a interface serial de comunicação implementados, resta realizar a primeira integração destes para um único sistema a ser futuramente expandido pela integração dos códigos de controle do conversor ADC e da contagem de pulsos determinando a posição de incidência do múon. Desta forma, este primeiro módulo integrado, funcionará de acordo com o mostrado na Fig. 4.3.

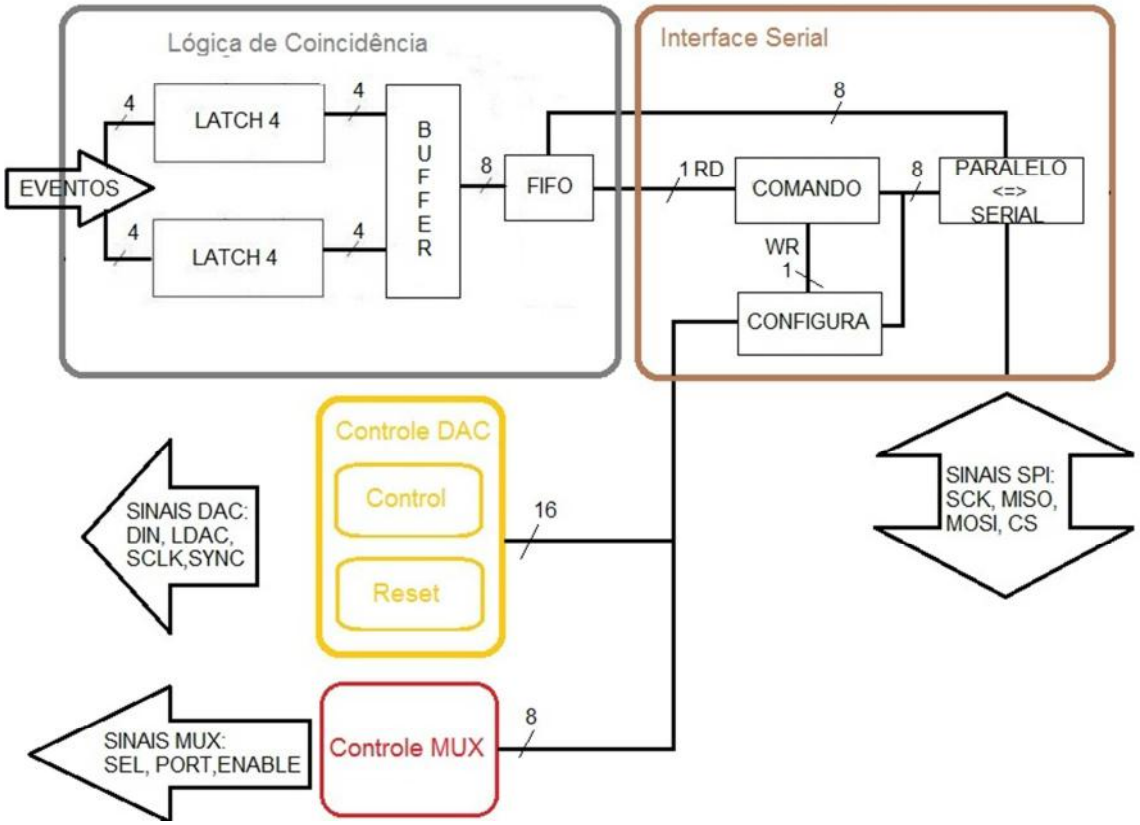
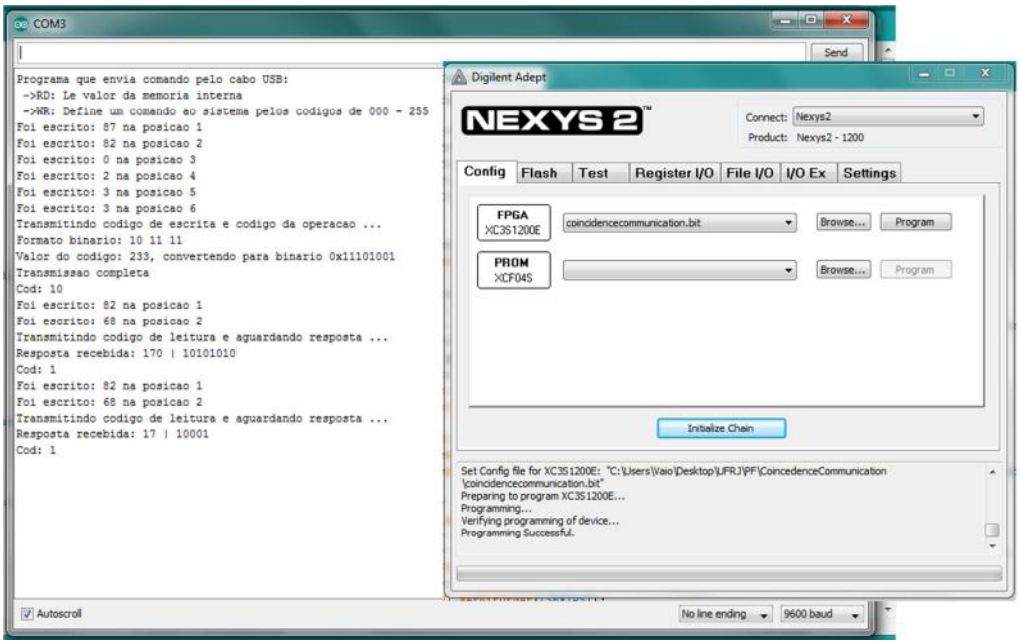


Figura 4.3 - Montagem do sistema integrado

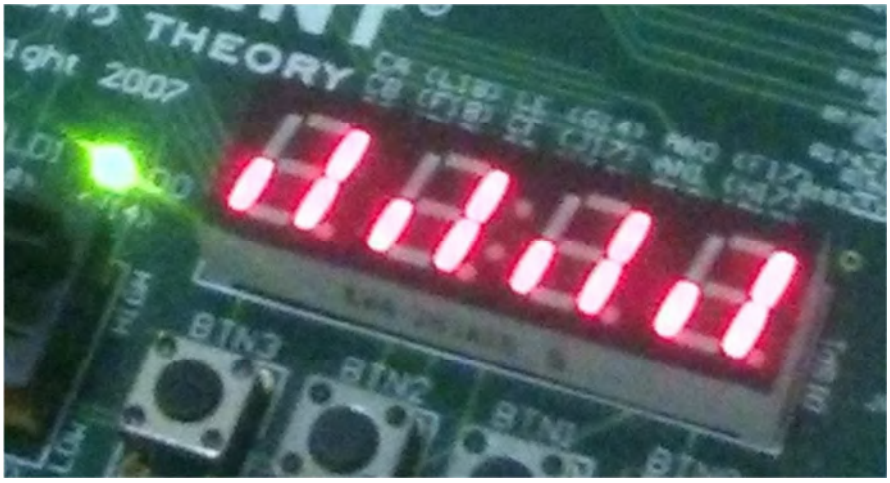
Para a ligação entre a lógica de coincidência e a interface serial, os dois módulos foram unidos como componentes de uma entidade de maior hierarquia e os sinais conectados de forma que os dados fossem lidos no momento em que o sinal de leitura fosse recebido. Enquanto este estiver ativo, o dado da memória interna será convertido e enviado para o operador através da interface e o microcontrolador. Ao final desta operação, um novo comando pode ser enviado pelo operador e novos eventos serem

recebidos, durante ou depois dessa comunicação com o operador, poderão ser salvos na memória.

Para o teste desta montagem contendo apenas a coincidência e a interface, os dados escritos pelo operador serão simplesmente mostrados no display de 7 segmentos de modo a verificar seu correto funcionamento. No teste, será enviado 0xE9 (11101001) e com o gerador simulado, dois vetores de eventos serão recebidos e o operador irá lê-los, limpando a FIFO, sendo os dados esperados 0xAA (10101010) e 0x11 (00010001), respectivamente nesta ordem. Como pode ser observado na Fig. 4.4, estes foram os resultados obtidos.



(a)



(b)

Figura 4.4 - (a) Interface do operador mostrando o histórico de operações, junto com a imagem do programa que síntese na FPGA; (b) Resultado da operação de escrita nos displays e led

Para a inclusão dos códigos que realizam o controle do MUX e do DAC, foi preciso utilizar uma nova máquina de estados que sincronizasse seu funcionamento e definisse qual a configuração e a operação realizada: a definição de referência ou a seleção de um sinal.

Primeiro se recebe o vetor de configuração que definirá a próxima operação. Para isso é necessário aguardar o fim da escrita deste valor, para somente em seguida observá-lo e definir qual operação será executada, se passa, então, pelo estado de espera inicial, até o de se salvar o comando recebido e o de decisão. Esta definição dependerá dos dois bits menos significativos deste vetor, os outros bits já serão informações a serem utilizadas na configuração do módulo, os quais serão a seleção do MUX (MS) e do DAC (DS) para habilitação dos CIs, a seleção do sinal de entrada analógico do MUX (MA) e da porta de saída do DAC (DP).

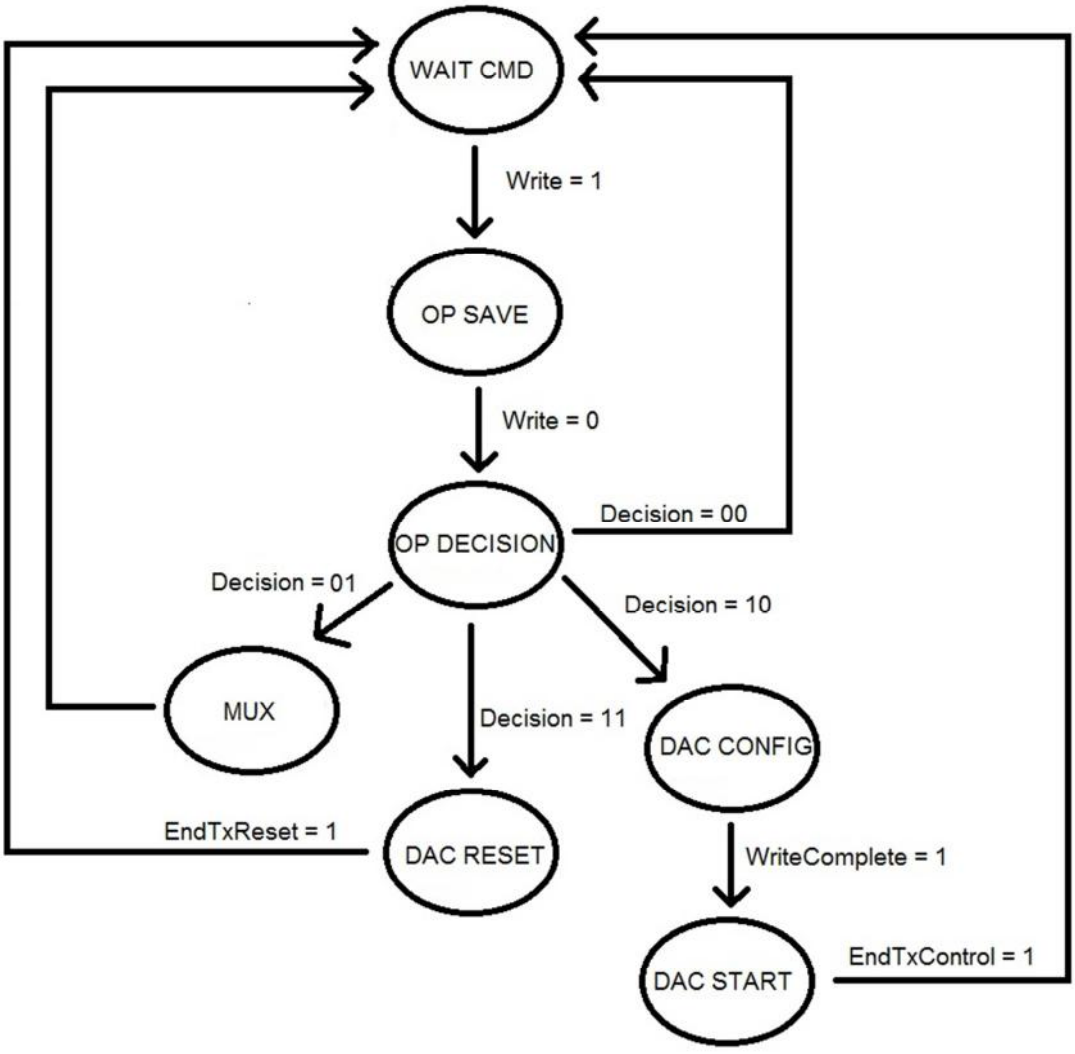


Figura 4.5 - Máquina de estados para o bloco de configuração do MUX e do DAC

O estado seguinte se utiliza destes bits mais significativos para definir a operação executada pelo módulo e depois retornar para a espera de um novo comando. A única operação que utilizará os 8 bits recebidos do operador será a escrita do valor que definirá a tensão de referência no DAC. Este será o único modo em que o operador necessitará enviar dois vetores distintos, isto é, duas escritas para dois estados, se aguardando o fim da segunda para se passar ao próximo estado de espera por um novo comando, devido à limitação do tamanho da palavra transmitida.

Na Tabela 4.1 podem ser observados os valores assumidos dos comandos para cada operação desejada, na Fig.4.5 a máquina de estados que implementa a configuração dos módulos e na Fig. 4.6 uma simulação do funcionamento.

Operação	Vetor de Bits
Controle do MUX	MS2MS1MS0 MA2MA1MA0 01
Controle do DAC - Config	DS2DS1DS0 DP2DP1DP0 10
Controle do DAC - Start	DV7DV6DV5DV4DV3DV2DV1DV0
Controle do DAC - Reset	000 DS2DS1DS0 11

Tabela 4.1 - Comandos para configuração

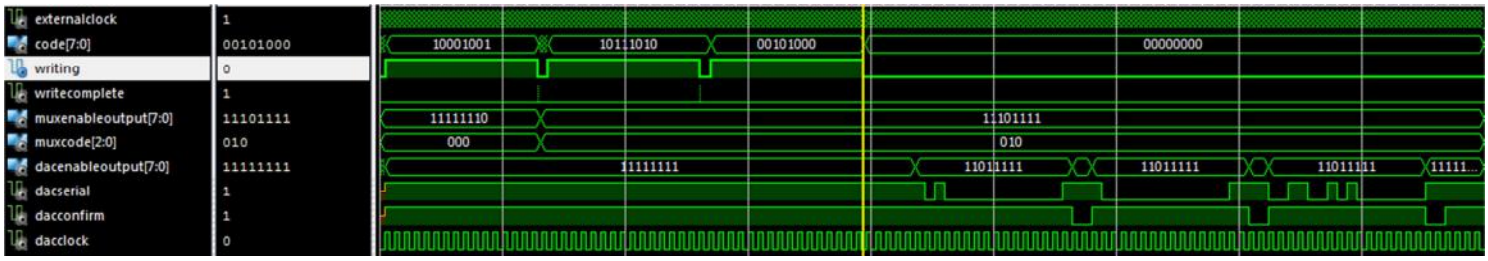


Figura 4.6 - Simulação do funcionamento da máquina de estados para os módulos MUX e DAC

Depois de realizada a simulação, o código foi integrado ao anterior que reunirá o módulo da lógica de coincidência e a interface serial, de modo que a integração estivesse completa e os testes finais pudessem ser realizados. Para isso, a montagem anterior foi repetida, com a modificação de que a palavra de bits enviados não iriam aparecer no display, mas sim ser utilizada para configurar o MUX ou enviar os dados serialmente que alterariam as saídas utilizadas como referências do DAC. Foi utilizado o CI do MUX para teste e as saídas referentes ao DAC observadas através de osciloscópio, de modo a repetir o funcionamento que anteriormente havia definido as

tensões de referência corretamente. Assim, com a montagem vista na Fig. 4.7, verificou-se com o auxílio de um multímetro a entrada do MUX corretamente selecionados e na Fig. 4.8 a tela do front-end com as imagens retiradas do osciloscópio, mostrando a correta forma dos sinais.

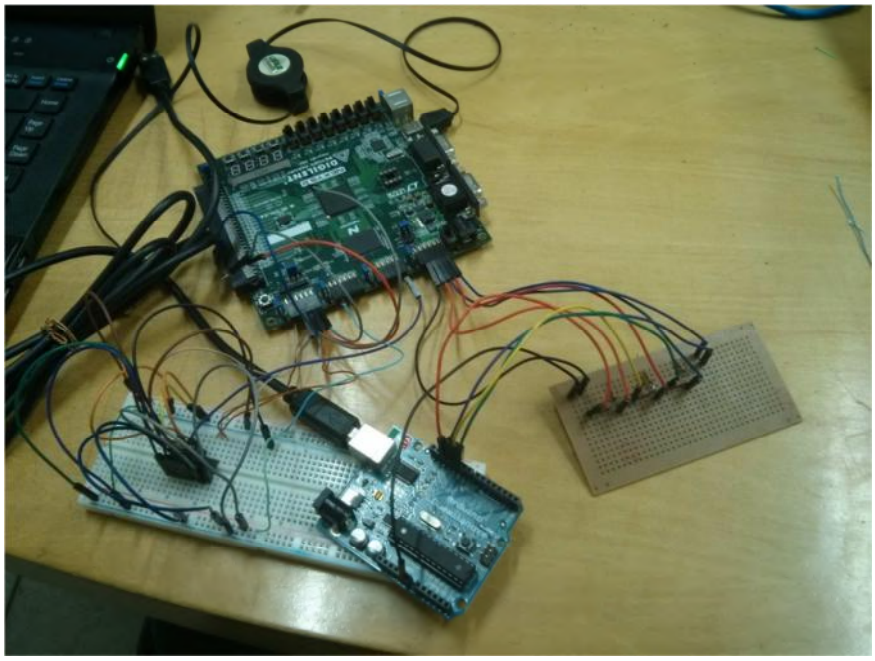
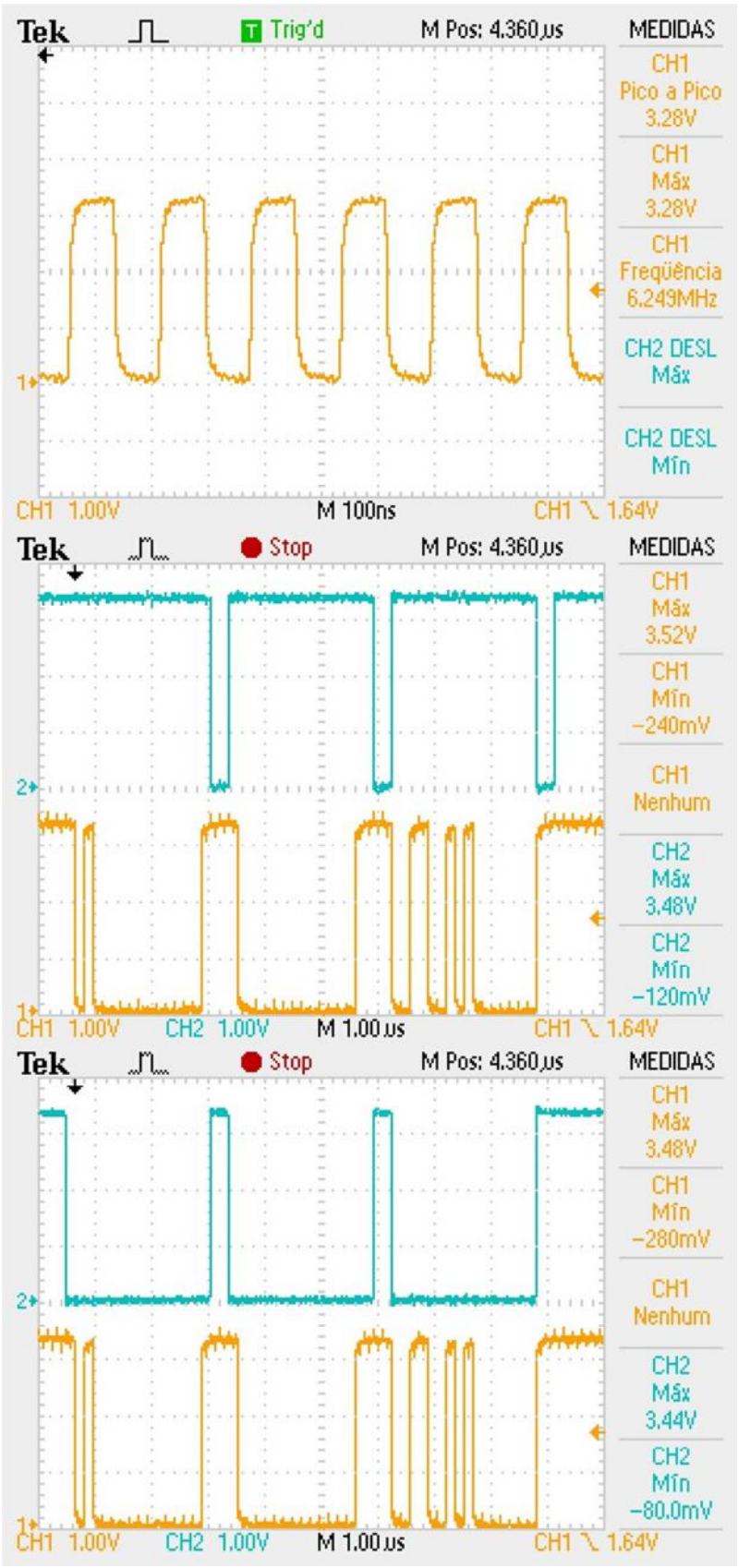


Figura 4.7 - Montagem para testes do DAC e MUX

```
Foi escrito: 87 na posicao 1
Foi escrito: 82 na posicao 2
Foi escrito: 0 na posicao 3
Foi escrito: 1 na posicao 4
Foi escrito: 8 na posicao 5
Foi escrito: 6 na posicao 6
Transmitindo codigo de escrita e codigo da operacao ...
Formato binario: 1 1000 110
Valor do codigo: 186, convertendo para binario 0x10111010
Transmissao completa
Cod: 10
Foi escrito: 87 na posicao 1
Foi escrito: 82 na posicao 2
Foi escrito: 0 na posicao 3
Foi escrito: 0 na posicao 4
Foi escrito: 4 na posicao 5
Foi escrito: 0 na posicao 6
Transmitindo codigo de escrita e codigo da operacao ...
Formato binario: 0 100 0
Valor do codigo: 40, convertendo para binario 0x101000
Transmissao completa
Cod: 10
Foi escrito: 82 na posicao 1
Transmitindo codigo de leitura e aguardando resposta ...
Resposta recebida: 17 | 10001
Cod: 1
```

(a)



(b)

Figura 4.8 - (a) Front-end exemplo de configuração e escrita do DAC; (b) Telas do osciloscópio: SCK, LDAC/DIN, SYNC/DIN

Finalmente com o sistema integrado, foi possível realizar o cálculo da taxa de transmissão dos eventos detectados para a visualização pelo operador. Para este verificamos os atrasos relacionados aos módulos e os sensores.

Com relação aos sensores, teremos um atraso relacionado a propagação da incidência do múon de uma terminação a outra, cujo valor máximo será de 100ns. Para o circuito da MAPMT e dos amplificadores seus atrasos poderão ser descartados, sendo muito menores do que os relacionados a esta propagação pela fibra.

Com relação aos módulos, podemos descartar os valores de atrasos referentes a aquisição e memória interna, levando em consideração apenas o relacionado ao envio desta para o operador, cujo valor dependerá do clock utilizado na transmissão. Como o clock utilizado no caso foi um de 200kHz e teremos uma transmissão de 8 bits de forma a completar todo o vetor de eventos, isso nos dará um tempo de 40μs por transmissão.

Reunindo os tempos, teremos um total de 40.1μs por transmissão, nos dando a taxa máxima de aquisição de aproximadamente 25000 bits/s recebidos, sendo esta suficiente para a taxa de aquisição verificada no Capítulo 2 de 27 eventos/s de modo a ter espaço para quase 900 bits por evento.

Capítulo 5

Conclusão

Ao final do desenvolvimento descrito, obtivemos um sistema que realiza a aquisição de sinais, controlado através de comandos dados por um operador, obtendo no processo um aprendizado na área, principalmente com os erros e problemas detectados, alcançando desta forma, os objetivos propostos.

Apesar disso, alguns aspectos do projeto poderiam ser mais desenvolvidos, como, por exemplo, os testes conectando a aquisição de dados ao detector já em funcionamento no laboratório do CBPF. Este teste com fibras, amplificadores e fotomultiplicadoras, além de avaliações de desempenho e robustez da lógica serão realizados em trabalhos futuros.

Grande parte dos problemas encontrados no desenvolvimento, se deve ao fato de trabalharmos com velocidades de transmissão que impossibilitasse o uso de montagens simples. Isso foi percebido tanto para a comunicação com o CI do DAC, onde foi necessário a utilização de fios paralelos para evitar erros e maus contatos; quanto na comunicação com o Arduino, pois devido aos valores de resistores utilizados foi preciso reduzir a velocidade de transmissão. No entanto, como este resultado não impactou na taxa máxima de aquisição, foi possível manter a utilização destes.

Também foi aprendida a importância da utilização das máquinas de estado para rápidas transições, como também, para sincronização de módulos, de maneira que o sistema trabalhe conforme o desejado.

Embora o objetivo tiver sido alcançado, foi possível detectar possíveis melhorias para o sistema ao longo do desenvolvimento. É possível perceber a importância de melhorar a velocidade de comunicação entre o operador e o sistema, de modo a dar uma maior margem para a taxa de transmissão, evitando erros devidos às diferenças de alimentação. Também é possível melhorar a interface com o operador, de modo a facilitar o controle do sistema, tornando-o mais amigável ao usuário.

Além das melhorias e trabalhos futuros para este módulo a ser incorporado no detector de raios cósmicos, o outro módulo desenvolvido previamente que realiza a transformação analógica para digital do sinal selecionado, de modo a visualizar a forma de onda na saída dos amplificadores, também precisa ser adicionado. Assim como o módulo de contagem de atrasos (TDC) que determinará a posição de incidência da partícula na fibra cintiladora, será futuramente desenvolvido e incluído ao detector.

Finalmente, com essas considerações, é possível dizer que apesar dos objetivos alcançados, para finalizar o desenvolvimento do sistema de instrumentação eletrônica para detecção de raios cósmicos ainda é preciso realizar testes e melhorias no sistema de aquisição descrito acima; inclusão de módulos já desenvolvidos de modo que trabalhem sincronizados; e o desenvolvimento do módulo que implementará o TDC.

Bibliografia

[1] Raios Cósmicos http://pt.wikipedia.org/wiki/Raio_c%C3%B3smico

[2] Grupo de Física de Neutrinos de Reatores
http://portal.cbpf.br/index.php?page=GruposPesquisa.Apresentacao&grupo=40

[3] LABDID Laboratório Didático do CBPF
http://portal.cbpf.br/index.php?page=FormacaoCientifica.labdid

[4] K.Nakamura et al. (Particle Data Group), The Review of Particle Physics, Chap. 24
Cosmic Rays, J. Phys. G 37, 075021, 2010.

[5] Olympus Co. - Theory of Confocal Microscopy - Fluorescence Excitation and EmissionFundamentals
http://www.olympusfluoview.com/theory/fluoroexciteemit.html

[6] A. Dyshkant et alli, About NICADD Extruded Scintillating Strips, FERMILAB-PUB-05-
010-E, 2005.

[7] HAMAMATSU PHOTONICS K. K. Electron Tube Division, Editorial Committee,
Photomultipliers Tubes – Basics and Applications, February 2006.

[8] Hamamatsu, Multianode Photomultiplier Tube Assembly H7546A, H7546B

[9] Analog Devices, 2.5 V to 5.5 V Octal Voltage Output 8-/10-/12-Bit DACs in 16-Lead TSSOP AD5308/AD5318/AD5328

[10] Costa, R., *Sistemas Digitais Aplicados à Aquisição e Processamento de Sinais*. B.Sc. project, Departamento de Eletrônica e Computação, Universidade Federal do Rio de Janeiro, Maio de 2010

[11] Digilent Nexys2 Board Reference Manual

[12] 74HC4051; 74HCT4051 8-channel analog multiplexer/demultiplexer

[13] Arduino Duemilanove http://arduino.cc/en/Main/arduinoBoardDuemilanove

[14] SPI Library for Arduino http://arduino.cc/en/Reference/SPI

[15] Ferreira, W. R., *Teste de Componentes do Sistema de Veto de Múons do Detector Nrutrinos Angra*, M.Sc. dissertation, Centro Brasileiro de Pesquisas Físicas, Setembro de 2010

Apêndice A

Códigos VHDL

- **DAC Control**

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

entity DACControl is
generic (datamin: integer := 0;
          datamax: integer := 16;
          dclk: integer := 7);
   Port (  ext_clock : in  STD_LOGIC;
           ext_start : in  STD_LOGIC;

           value : in STD_LOGIC_VECTOR (7 downto 0);
           dac_port : in STD_LOGIC_VECTOR (2 downto 0);
           dac_sel : in STD_LOGIC_VECTOR (2 downto 0);
           finish_tx : out STD_LOGIC;
           enable : out STD_LOGIC_VECTOR (7 downto 0);
           serial : out  STD_LOGIC;
           confirm : out  STD_LOGIC;
           cpclock : out STD_LOGIC);
end DACControl;

architecture serial of DACControl is
signal dly1,dly2,dly3,dly4: STD_LOGIC;
signal smin : STD_LOGIC;

signal cstart : integer range 0 to 50;
signal clock: STD_LOGIC;

signal cd_tx: STD_LOGIC_VECTOR (15 downto 0);
signal cd_reference, cd_ldac: STD_LOGIC_VECTOR (15 downto 0);
```

```
signal cd_sync: STD_LOGIC_VECTOR (7 downto 0);

signal datacount: integer range datamin to datamax;
signal cclk: integer range 0 to dclk;
type STATE is (rest, d16_ldac, aftertr_ldac, d16_ref, aftertr_ref, entries, d16_write, aftertr);
signal sttr: STATE;

begin
        process(ext_clock)
        begin
                if (ext_clock'EVENT and ext_clock = '1') then
                        if (cclk /= dclk) then cclk <= cclk+1;
                        else cclk <= 0;
                        end if;
                        if (cclk < 4) then clock <= '0';
                        else clock <= '1';
                        end if;
                end if;
        end process;

        process (ext_start, ext_clock)
        begin
                if (ext_start = '0') then cstart <= 0;
                        elsif(ext_clock'EVENT and ext_clock = '1') then
                                if (cstart /= 50) then cstart <= cstart + 1;
                                else cstart <= 50;
                                end if;
                                if (cstart > 10 and cstart < 40) then smin <= '1';
                                else smin <= '0';
                                end if;
                        end if;
        end process;

        process (clock)
        begin
                if (clock'EVENT and clock = '1') then
                        case sttr is
                                when rest =>
                                        if (smin = '1') then sttr <= entries;
                                        else sttr <= rest;
```

```
                        end if;
                when entries =>
                        if (dly1 = '1') then sttr <= d16_ldac;
                        else sttr <= entries;
                        end if;
                when d16_ldac =>
                        if (datacount = datamax-1) then sttr <= aftertr_ldac;
                        else sttr <= d16_ldac;
                        end if;
                when aftertr_ldac =>
                        if (dly2 = '1') then sttr <= d16_ref;
                        else sttr <= aftertr_ldac;
                        end if;
                when d16_ref =>
                        if (datacount = datamax-1) then sttr <= aftertr_ref;
                        else sttr <= d16_ref;
                        end if;
                when aftertr_ref =>
                        if (dly3 = '1') then sttr <= d16_write;
                        else sttr <= aftertr_ref;
                        end if;
                when d16_write =>
                        if (datacount = datamax-1) then sttr <= aftertr;
                        else sttr <= d16_write;
                        end if;
                when aftertr =>
                        if (dly4 = '1') then sttr <= rest;
                        else sttr <= aftertr;
                        end if;
        end case;
    end if;
end process;

process (clock,sttr)
begin
    if (clock'EVENT and clock = '1') then
        case sttr is
                when rest =>
                        serial <= '1';
                        confirm <= '1';
```

```
                dly1 <= '0';
                dly2 <= '0';
                dly3 <= '0';
                dly4 <= '0';
                finish_tx <= '0';
                cd_tx <= "0000000000000000";
                cd_sync <= "11111111";
                enable <= "11111111";
            when entries =>
                cd_ldac <= "1000000000000101";
                cd_reference <= "1100000000000001";
                cd_tx(1) <= dac_port(0);
                cd_tx(2) <= dac_port(1);
                cd_tx(3) <= dac_port(2);
                cd_tx(4) <= value(7);
                cd_tx(5) <= value(6);
                cd_tx(6) <= value(5);
                cd_tx(7) <= value(4);
                cd_tx(8) <= value(3);
                cd_tx(9) <= value(2);
                cd_tx(10) <= value(1);
                cd_tx(11) <= value(0);
                if(dac_sel = "000") then cd_sync <= "11111110";
                elsif (dac_sel = "001") then cd_sync <=
"11111101";
                elsif (dac_sel = "010") then cd_sync <=
"11111011";
                elsif (dac_sel = "011") then cd_sync <=
"11110111";
                elsif (dac_sel = "100") then cd_sync <=
"11101111";
                elsif (dac_sel = "101") then cd_sync <=
"11011111";
                elsif (dac_sel = "110") then cd_sync <=
"10111111";
                elsif (dac_sel = "111") then cd_sync <=
"01111111";
                else cd_sync <= "11111111";
                end if;
                dly1 <= '1';
```

```
when d16_ldac =>
        enable <= cd_sync;
        if (datacount /= datamax) then
                serial <= cd_ldac (datacount);
                datacount <= datacount + 1;
        else datacount <= datamax;
        end if;
        confirm <= '1';
when aftertr_ldac =>
        enable <= "11111111";
        serial <= '1';
        confirm <= '0';
        datacount <= datamin;
        dly2 <= '1';
when d16_ref =>
        enable <= cd_sync;
        if (datacount /= datamax) then
                serial <= cd_reference (datacount);
                datacount <= datacount + 1;
        else datacount <= datamax;
        end if;
        confirm <= '1';
when aftertr_ref =>
        enable <= "11111111";
        serial <= '1';
        confirm <= '0';
        datacount <= datamin;
        dly3 <= '1';
when d16_write =>
        enable <= cd_sync;
        if (datacount /= datamax) then
                serial <= cd_tx (datacount);
                datacount <= datacount + 1;
        else datacount <= datamax;
        end if;
        confirm <= '1';
when aftertr =>
        enable <= "11111111";
        serial <= '1';
        confirm <= '0';
```

```
                                        datacount <= datamin;
                                        dly4 <= '1';
                                        finish_tx <= '1';
                                end case;
                        end if;
                end process;

                process(clock)
                begin
                        cpclock <= clock;
                end process;

        end serial;
```

- **DAC Reset**

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

entity DACReset is
generic (datamin: integer := 0;
          datamax: integer := 16;
          dclk: integer := 7);
   Port (  ext_clock : in  STD_LOGIC;
          ext_start : in  STD_LOGIC;
          dac_sel : in STD_LOGIC_VECTOR (2 downto 0);
          finish_tx : out STD_LOGIC;
          enable : out STD_LOGIC_VECTOR (7 downto 0);
          serial : out  STD_LOGIC;
          confirm : out  STD_LOGIC;
          cpclock : out STD_LOGIC);
end DACReset;

architecture serial of DACReset is
signal dly,dly1 : STD_LOGIC;
signal smin : STD_LOGIC;

signal cstart : integer range 0 to 50;
```

```
signal clock: STD_LOGIC;

signal cd_reset: STD_LOGIC_VECTOR (15 downto 0);
signal cd_sync: STD_LOGIC_VECTOR (7 downto 0);

signal datacount: integer range datamin to datamax;
signal cclk: integer range 0 to dclk;
type STATE is (rest, entries, d16_reset, aftertr_reset);
signal sttr: STATE;

begin

	process(ext_clock)
	begin
		if (ext_clock'EVENT and ext_clock = '1') then
			if (cclk /= dclk) then cclk <= cclk+1;
			else cclk <= 0;
			end if;
			if (cclk < 4) then clock <= '0';
			else clock <= '1';
			end if;
		end if;
	end process;

	process (ext_start, ext_clock)
	begin
		if (ext_start = '0') then cstart <= 0;
			elsif(ext_clock'EVENT and ext_clock = '1') then
				if (cstart /= 50) then cstart <= cstart + 1;
				else cstart <= 50;
				end if;
				if (cstart > 10 and cstart < 40) then smin <= '1';
				else smin <= '0';
				end if;
			end if;
	end process;

	process (clock)
	begin
```

```
        if (clock'EVENT and clock = '1') then
                case sttr is
                        when rest =>
                                if (smin = '1') then sttr <= entries;
                                else sttr <= rest;
                                end if;
                        when entries =>
                                if (dly = '1') then sttr <= d16_reset;
                                else sttr <= entries;
                                end if;
                        when d16_reset =>
                                if (datacount = datamax-1) then sttr <= aftertr_reset;
                                else sttr <= d16_reset;
                                end if;
                        when aftertr_reset =>
                                if (dly1 = '1') then sttr <= rest;
                                else sttr <= aftertr_reset;
                                end if;
                end case;
        end if;
end process;

process (clock,sttr)
begin
        if (clock'EVENT and clock = '1') then
                case sttr is
                        when rest =>
                                serial <= '1';
                                confirm <= '1';
                                dly <= '0';
                                dly1 <= '0';
                                finish_tx <= '0';
                                cd_sync <= "11111111";
                                enable <= "11111111";
                        when entries =>
                                cd_reset <= "0000000000000111";
                                if(dac_sel = "000") then cd_sync <= "11111110";
                                elsif (dac_sel = "001") then cd_sync <= "11111101";
                                elsif (dac_sel = "010") then cd_sync <= "11111011";
                                elsif (dac_sel = "011") then cd_sync <= "11110111";
```

```
                elsif (dac_sel = "100") then cd_sync <= "11101111";
                elsif (dac_sel = "101") then cd_sync <= "11011111";
                elsif (dac_sel = "110") then cd_sync <= "10111111";
                elsif (dac_sel = "111") then cd_sync <= "01111111";
                else cd_sync <= "11111111";
                end if;

                dly <= '1';
            when d16_reset =>
                enable <= cd_sync;
                if (datacount /= datamax) then
                    serial <= cd_reset (datacount);
                    datacount <= datacount + 1;
                else datacount <= datamax;
                end if;
                confirm <= '1';
            when aftertr_reset =>
                enable <= "11111111";
                serial <= '1';
                confirm <= '0';
                datacount <= datamin;
                dly1 <= '1';
                finish_tx <= '1';
        end case;
    end if;
end process;

process(clock)
begin
    cpclock <= clock;
end process;

end serial;
```

- **MUX Control**

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

```
entity MuxControl is
   Port (address : in  STD_LOGIC_VECTOR (2 downto 0);
         sel_mux : in  STD_LOGIC_VECTOR (2 downto 0);
         en_mux : out  STD_LOGIC_VECTOR (7 downto 0);
         sel_code : out  STD_LOGIC_VECTOR (2 downto 0));
end MuxControl;

architecture mux_entries of MuxControl is

begin

process(sel_mux)
begin
            if (sel_mux = "000") then  en_mux <= "11111110";
            elsif (sel_mux = "001") then  en_mux <= "11111101";
            elsif (sel_mux = "010") then  en_mux <= "11111011";
            elsif (sel_mux = "011") then  en_mux <= "11110111";
            elsif (sel_mux = "100") then  en_mux <= "11101111";
            elsif (sel_mux = "101") then  en_mux <= "11011111";
            elsif (sel_mux = "110") then  en_mux <= "10111111";
            elsif (sel_mux = "111") then  en_mux <= "01111111";
            end if;
end process;

sel_code <= address;
end mux_entries;
```

- **MuxDAC**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MuxDAC is
   Port (
            ExternalClock : in  STD_LOGIC;
            Code : in STD_LOGIC_VECTOR (7 downto 0);
            Writing : in STD_LOGIC;
            WriteComplete : in STD_LOGIC;
            MUXEnableOutput : out STD_LOGIC_VECTOR (7 downto 0);
```

```
            MUXCode : out STD_LOGIC_VECTOR (2 downto 0);
            DACEnableOutput : out STD_LOGIC_VECTOR (7 downto 0);
            DACSerial : out STD_LOGIC;
            DACConfirm : out STD_LOGIC;
            DACCLock : out STD_LOGIC );
end MuxDAC;

architecture ControlMUXDAC of MuxDAC is

signal DACValue   : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
signal DACPort    : STD_LOGIC_VECTOR (2 downto 0) := "000";
signal DACSelect  : STD_LOGIC_VECTOR (2 downto 0) := "000";
signal MUXAddress : STD_LOGIC_VECTOR (2 downto 0) := "000";
signal MUXSelect  : STD_LOGIC_VECTOR (2 downto 0) := "000";

signal CodeCmd : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
signal Decision : STD_LOGIC_VECTOR(1 downto 0) := "00";
signal StartTxReset, EndTxReset, StartTxControl, EndTxControl : STD_LOGIC := '0';

type operation is (wait_cmd, op_save, op_decision, mux, dac_config, dac_start, dac_reset);
signal cmd : operation;

signal DACCSer, DACCConf, DACCClk, DACRSer, DACRConf, DACRClk : STD_LOGIC;
signal DACCEnbOut, DACREnbOut : STD_LOGIC_VECTOR (7 downto 0);
signal DACOp : STD_LOGIC := '0';

component DACControl is
generic (datamin: integer := 0;
          datamax: integer := 16;
          dclk: integer := 7);
   Port (  ext_clock : in  STD_LOGIC;
           ext_start : in  STD_LOGIC;
           value : in STD_LOGIC_VECTOR (7 downto 0);
           dac_port : in STD_LOGIC_VECTOR (2 downto 0);
           dac_sel : in STD_LOGIC_VECTOR (2 downto 0);
           finish_tx : out STD_LOGIC;
           enable : out STD_LOGIC_VECTOR (7 downto 0);
           serial : out  STD_LOGIC;
           confirm : out  STD_LOGIC;
```

```
            cpclock : out STD_LOGIC);
end component;

component MuxControl is
   Port ( address : in  STD_LOGIC_VECTOR (2 downto 0);
          sel_mux : in  STD_LOGIC_VECTOR (2 downto 0);
          en_mux : out  STD_LOGIC_VECTOR (7 downto 0);
          sel_code : out  STD_LOGIC_VECTOR (2 downto 0));
end component;

component DACReset is
generic (datamin: integer := 0;
              datamax: integer := 16;
              dclk: integer := 7);
   Port (  ext_clock : in  STD_LOGIC;
           ext_start : in  STD_LOGIC;
           dac_sel : in STD_LOGIC_VECTOR (2 downto 0);
           finish_tx : out STD_LOGIC;
           enable : out STD_LOGIC_VECTOR (7 downto 0);
           serial : out  STD_LOGIC;
           confirm : out  STD_LOGIC;
           cpclock : out STD_LOGIC);
end component;

begin
dacc: DACControl port map (ext_clock => ExternalClock, ext_start => StartTxControl, value => DACValue, dac_port => DACPort, dac_sel => DACSelect, finish_tx => EndTxControl, enable => DACCEnbOut, serial => DACCSer, confirm => DACCConf, cpclock => DACCClk);

dacr: DACReset port map (ext_clock => ExternalClock, ext_start => StartTxReset, dac_sel => DACSelect, enable => DACREnbOut,  finish_tx => EndTxReset, serial => DACRSer, confirm => DACRConf, cpclock => DACRClk);

muxc: MUXControl port map (address => MUXAddress, sel_mux => MUXSelect, en_mux => MUXEnableOutput, sel_code => MUXCode);

process(ExternalClock, Writing, CodeCmd, EndTxControl, EndTxReset, Decision)
begin
          if(ExternalClock'EVENT and ExternalClock = '1') then
```

```
                case cmd is
                        when wait_cmd =>
                                if(Writing = '1') then cmd <= op_save;
                                else cmd <= wait_cmd;
                                end if;
                        when op_save =>
                                if(Writing = '0') then cmd <= op_decision;
                                else cmd <= op_save;
                                end if;
                        when op_decision =>
                                if(Decision = "01") then cmd <= mux;
                                elsif(Decision = "10") then cmd <= dac_config;
                                elsif(Decision = "11") then cmd <= dac_reset;
                                else cmd <= wait_cmd;
                                end if;
                        when mux =>
                                cmd <= wait_cmd;
                        when dac_config =>
                                if(WriteComplete = '1') then cmd <= dac_start;
                                else cmd <= dac_config;
                                end if;
                        when dac_start =>
                                if(EndTxControl = '1') then cmd <= wait_cmd;
                                else cmd <= dac_start;
                                end if;
                        when dac_reset =>
                                if(EndTxReset = '1') then cmd <= wait_cmd;
                                else cmd <= dac_start;
                                end if;
                end case;
        end if;
end process;

process(ExternalClock)
begin
        if(ExternalClock'EVENT and ExternalClock = '0') then
                case cmd is
                        when wait_cmd =>
                                CodeCmd <= "00000000";
                                Decision <= "00";
```

```
                    DACOp <= '0';
                    DACPort <= "000";
                    DACSelect <= "000";
                    DACValue <= "00000000";
                    StartTxControl <= '0';
                    StartTxReset <= '0';
                    MuxAddress <= MuxAddress;
                    MuxSelect <= MuxSelect;
                when op_save =>
                    CodeCmd <= Code;
                when op_decision =>
                    Decision(0) <= CodeCmd(0);
                    Decision(1) <= CodeCmd(1);
                when mux =>
                    MuxAddress(0) <= CodeCmd(2);
                    MuxAddress(1) <= CodeCmd(3);
                    MuxAddress(2) <= CodeCmd(4);
                    MuxSelect(0) <= CodeCmd(5);
                    MuxSelect(1) <= CodeCmd(6);
                    MuxSelect(2) <= CodeCmd(7);
                when dac_config =>
                    DACPort(0) <= CodeCmd(2);
                    DACPort(1) <= CodeCmd(3);
                    DACPort(2) <= CodeCmd(4);
                    DACSelect(0) <= CodeCmd(5);
                    DACSelect(1) <= CodeCmd(6);
                    DACSelect(2) <= CodeCmd(7);
                    DACValue <= Code;
                when dac_start =>
                    StartTxControl <= '1';
                    DACOp <= '0';
                when dac_reset =>
                    DACSelect(0) <= CodeCmd(5);
                    DACSelect(1) <= CodeCmd(6);
                    DACSelect(2) <= CodeCmd(7);
                    StartTxReset <= '1';
                    DACOp <= '1';
            end case;
        end if;
    end process;
```

```
process(DACOp,DACCSer, DACCConf, DACCClk, DACRSer, DACRConf, DACRClk, DACCEnbOut,
DACREnbOut)
begin
        if(DACOp = '0') then
                DACSerial <= DACCSer;
                DACEnableOutput <= DACCEnbOut;
                DACClock <= DACCClk;
                DACConfirm <= DACCConf;
        else
                DACSerial <= DACRSer;
                DACEnableOutput <= DACREnbOut;
                DACClock <= DACRClk;
                DACConfirm <= DACRConf;
        end if;
end process;

end ControlMUXDAC;
```

- **FIFO**

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

entity Fifo is
   Port (   DataIn : in  STD_LOGIC_VECTOR (7 downto 0);
            WR : in  STD_LOGIC;
            RD : in  STD_LOGIC;
            RS : in STD_LOGIC;
            Clk : in  STD_LOGIC;
             DataOut : out  STD_LOGIC_VECTOR (7 downto 0);
            FlagOfDone : out STD_LOGIC;
            FlagOfFull : out  STD_LOGIC;
            FlagOfEmpty : out STD_LOGIC );
end Fifo;

architecture data of Fifo is
-------------------------------------------------------------------------------------------------------------
```

```
type queue is array (0 to 15) of STD_LOGIC_VECTOR (7 downto 0);
signal regs: queue;
-------------------------------------------------------------------------------------------------------------------
type state is (initial,central,append,serve);
signal queue_state: state;
-------------------------------------------------------------------------------------------------------------------
signal head: integer range 0 to 15 := 0;
signal tail: integer range 0 to 15 := 0;
signal cnt: integer range 0 to 16 := 0;
signal operation, empty, full: STD_LOGIC;
-------------------------------------------------------------------------------------------------------------------
begin


process (Clk,WR,RD,RS,empty,full,operation)
begin
        if (Clk'EVENT and Clk = '1') then
                case queue_state is
                        when initial =>
                                if (RS = '0') then queue_state <= central;
                                else queue_state <= initial;
                                end if;
                        when central =>
                                if (RS = '1') then queue_state <= initial;
                                elsif (empty = '0' and RD = '1') then queue_state <= serve;
                                elsif (full = '0' and WR = '1') then queue_state <= append;
                                else queue_state <= central;
                                end if;
                        when append =>
                                if (RS = '1') then queue_state <= initial;
                                elsif (operation = '1') then queue_state <= central;
                                else queue_state <= append;
                                end if;
                        when serve =>
                                if (RS = '1') then queue_state <= initial;
                                elsif (operation = '1') then queue_state <= central;
                                else queue_state <= serve;
                                end if;
                end case;
        end if;
end process;
```

```
---------------------------------------------------------------------------------------------------------------------
process (queue_state,Clk)
begin
        if (Clk'EVENT and Clk = '0') then
                case queue_state is
                        when initial =>
                                head <= 0;
                                tail <= 0;
                                cnt <= 0;
                                regs (0) <= "00000000";
                                regs (1) <= "00000000";
                                regs (2) <= "00000000";
                                regs (3) <= "00000000";
                                regs (4) <= "00000000";
                                regs (5) <= "00000000";
                                regs (6) <= "00000000";
                                regs (7) <= "00000000";
                                regs (8) <= "00000000";
                                regs (9) <= "00000000";
                                regs (10) <= "00000000";
                                regs (11) <= "00000000";
                                regs (12) <= "00000000";
                                regs (13) <= "00000000";
                                regs (14) <= "00000000";
                                regs (15) <= "00000000";
                                FlagOfDone <= '0';
                                DataOut <= "00000000";
                        when central =>
                                operation <= '0';
                                FlagOfDone <= '1';
                        when append =>
                                operation <= not operation;
                                if(cnt < 16) then cnt <= cnt + 1;
                                else cnt <= 16;
                                end if;
                                regs (tail) <= DataIn;
                                if (tail /= 15) then tail <= tail +1;
                                else tail <= 0;
                                end if;
                                FlagOfDone <= '0';
```

```
                    when serve =>
                        operation <= not operation;
                        if (cnt > 0) then cnt <= cnt - 1;
                        else cnt <= 0;
                        end if;
                        DataOut <= regs (head);
                        if (head /= 15) then head <= head +1;
                        else head <= 0;
                        end if;
                        FlagOfDone <= '0';
                end case;
        end if;
end process;
-------------------------------------------------------------------------------------------------------------------
--Verify the counter  to see if the queue is empty or full
process (cnt,empty,full)
begin
        if (cnt = 0) then empty <= '1';
        else empty <= '0';
        end if;

        if (cnt = 16) then full <= '1';
        else full <='0';
        end if;

        FlagOfEmpty <= empty;
        FlagOfFull <= full;
end process;

end data;
```

- **Aquisição na Janela de Tempo**

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

entity AsyncTimeWindow is
    Port (Clock : in  STD_LOGIC;
```

```
        InputSignals : in  STD_LOGIC_VECTOR (3 downto 0);
        ExtReset : in  STD_LOGIC;                                                    --
        Data : out  STD_LOGIC_VECTOR (3 downto 0);
        SaveData : out  STD_LOGIC);
end AsyncTimeWindow;

architecture temp of AsyncTimeWindow is
signal Passing: STD_LOGIC;
signal TimeWindow: STD_LOGIC;
signal enb : STD_LOGIC;
signal newWindow : STD_LOGIC := '0';
signal windowCnt : STD_LOGIC_VECTOR(1 downto 0);
--------------------------------------------------------------------------------------------------------------------
component FlipFlopControl is
    Port ( Clock : in  STD_LOGIC;
           Enable : in STD_LOGIC;
           Reset : in  STD_LOGIC;
           QOut : out  STD_LOGIC);
end component;
--------------------------------------------------------------------------------------------------------------------
component WindowDefinition is
    Port ( Clock : in  STD_LOGIC;
             Start : in  STD_LOGIC;
             Restart : in STD_LOGIC;
             Count : out STD_LOGIC_VECTOR (1 downto 0);
             Window : out  STD_LOGIC);
end component;
--------------------------------------------------------------------------------------------------------------------
begin
atw: WindowDefinition port map (Clock => Clock, Start => Passing, Restart => ExtReset,
                                        Count => windowCnt, Window => TimeWindow);
ff0: FlipFlopControl port map (Clock => InputSignals(0), Reset => ExtReset,
                                        Enable => enb, QOut => Data(0));
ff1: FlipFlopControl port map (Clock => InputSignals(1), Reset => ExtReset,
                                         Enable => enb, QOut => Data(1));
ff2: FlipFlopControl port map (Clock => InputSignals(2), Reset => ExtReset,
                                        Enable => enb, QOut => Data(2));
ff3: FlipFlopControl port map (Clock => InputSignals(3), Reset => ExtReset,
                                         Enable => enb, QOut => Data(3));
process (InputSignals)
```

```
begin
        Passing <= InputSignals(0) or InputSignals(1) or InputSignals(2) or InputSignals(3);
end process;

process (ExtReset,TimeWindow,Passing,newWindow)
begin
        if (ExtReset = '1') then newWindow <= '0';
        elsif (Passing'EVENT and Passing = '1') then newWindow <= '1';
        end if;
        if(newWindow = '1' and TimeWindow = '1') then SaveData <= '1';
        else SaveData <= '0';
        end if;
end process;
process(TimeWindow,ExtReset,windowCnt)
begin
        if(ExtReset = '1') then enb <= '0';
        elsif (TimeWindow = '0') then enb <= '1';
        elsif (windowCnt = "00") then enb <= '1';
        else enb <= '0';
        end if;
end process;
end temp;
```

- **Definição da janela de tempo**

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

entity WindowDefinition is
   Port ( Clock : in  STD_LOGIC;
          Start : in  STD_LOGIC;
          Restart : in STD_LOGIC;
          Count : out STD_LOGIC_VECTOR (1 downto 0);
          Window : out  STD_LOGIC);
end WindowDefinition;

architecture WinTime of WindowDefinition is
-------------------------------------------------------------------------------------------------------------------
```

```
type state is (waiting, inside);
signal timewindow : state;
signal asyncWindow: STD_LOGIC := '1';
signal Entry: STD_LOGIC := '0';
signal Cnt : STD_LOGIC_VECTOR (1 downto 0);
-----------------------------------------------------------------------------------------------------------------------
begin
process (Entry,Restart,timewindow)
begin
	case timewindow is
		when waiting =>
			if (Entry = '1') then timewindow <= inside;
			else timewindow <= waiting;
			end if;
		when inside =>
			if (Restart = '1') then timewindow <= waiting;
			else timewindow <= inside;
			end if;
	end case;
end process;

process (timewindow,asyncWindow)
begin
	case timewindow is
		when waiting =>
			asyncWindow <= '1';
		when inside =>
		  asyncWindow <= '0';
	end case;
end process;

process(Clock,asyncWindow)
begin
	if(asyncWindow = '0') then
		if(Clock'EVENT and Clock = '1') then
			if(Cnt /= "10") then
				Cnt <= Cnt + 1;
			else
				Cnt <= "10";
			end if;
```

```
                end if;
        else
                Cnt <= "00";
        end if;
end process;

process (Cnt,Start)
begin
        if (Cnt > "00") then Entry <= '0';
        elsif (Start'EVENT and Start = '1') then Entry <= '1';
        end if;
        Count <= Cnt;
end process;

process(Cnt,asyncWindow)--,syncWindow)
begin
        if(Cnt = "00") then
                if (asyncWindow = '0') then Window <= '0';
                else Window <= '1';
                end if;
        elsif(Cnt = "01") then Window <= '0';
        else
                Window <= '1';
        end if;
end process;

end WinTime;
```

- **Flip Flop**

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

entity FlipFlopControl is
   Port ( Clock : in  STD_LOGIC;
        Enable : in STD_LOGIC;
        Reset : in  STD_LOGIC;
        QOut : out  STD_LOGIC);
```

```
end FlipFlopControl;
architecture FlipFlop of FlipFlopControl is
signal buff : STD_LOGIC := '0';
begin
process (Clock,Reset)
begin
        if (Reset = '1') then  buff <= '0';
        elsif (Clock'EVENT and Clock = '1') then
                if (Enable ='1') then buff <= '1';
                else buff <= buff;
                end if;
        end if;
end process;
process(buff)
begin
        QOut <= buff;
end process;
end FlipFlop;
```

- **Lógica de Coincidência**

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

entity TesteCoincidencia is
   Port ( BoardClock : in  STD_LOGIC;
        ResetButton : in  STD_LOGIC;
                        Events : in STD_LOGIC_VECTOR (7 downto 0);
                        RdData : in STD_LOGIC;
                        MFull : out STD_LOGIC;
                        MEmpty: out STD_LOGIC;
                        CData : out  STD_LOGIC_VECTOR (7 downto 0)
                        );
end TesteCoincidencia;

architecture Behavioral of TesteCoincidencia is

--signal Cnt : integer range 0 to 2500;
```

```
signal MData, TData : STD_LOGIC_VECTOR (7 downto 0); --, Events, OData : STD_LOGIC_VECTOR (7 downto 0);

signal TrigBuf : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
--signal OutBuf : STD_LOGIC_VECTOR (7 downto 0) := "00000000";

signal MSave, MLoad : STD_LOGIC;

--signal TimeReset : STD_LOGIC_VECTOR (1 downto 0) := "00";
--signal TimeRead : STD_LOGIC_VECTOR (1 downto 0) := "00";

--signal FastClock, SlowClock, MediumClock, Locked, DCMRst : STD_LOGIC;

signal EventHigh, EventLow, DataHigh, DataLow : STD_LOGIC_VECTOR (3 downto 0);

signal WrData, HighSave, LowSave, MDone, IMReset : STD_LOGIC;
type coincidence_data is (nonevent,buffering_input,saving,loading,buffering_output);
signal actual : coincidence_data;

component AsyncTimeWindow is
    Port (Clock : in  STD_LOGIC;
            -- Clock signal: 50MHz
          InputSignals : in  STD_LOGIC_VECTOR (3 downto 0);  -- Signals coming from amplifiers
          ExtReset : in  STD_LOGIC;                                                    --
External reset of the internal temporary memory
          Data : out  STD_LOGIC_VECTOR (3 downto 0);                    -- Send data saved from
acquisition
--                              tWin : out STD_LOGIC;
--                              wCnt : out STD_LOGIC_VECTOR(1 downto 0);
          SaveData : out  STD_LOGIC);
            -- Signal to write data on an external register
end component;

component Fifo is
    Port ( DataIn : in  STD_LOGIC_VECTOR (7 downto 0);          --Data Input to be saved on the
FIFO
          WR : in  STD_LOGIC;
                    --Signal to make write operation
```

```
        RD : in  STD_LOGIC;
                --Signal to make read operation
                        RS : in STD_LOGIC;
                                --Signal to reset the FIFO
                        Clk : in  STD_LOGIC;
                                --FIFO Clock Signal: 200MHz
        DataOut : out  STD_LOGIC_VECTOR (7 downto 0);          --Data Output to read from the
FIFO
                        FlagOfDone : out STD_LOGIC;
              --Flag to define when an operation is over
                        FlagOfFull : out  STD_LOGIC;
                --Flag to view if the FIFO is full
                        FlagOfEmpty : out STD_LOGIC );
                       --Flag to view if the FIFO is empty
end component;

begin
-------------------------------------------------------------------------------------------------------------------------
-----------------------------
atwh : AsyncTimeWindow  port map (Clock => BoardClock, InputSignals => EventHigh, ExtReset =>
IMReset,
                                                        Data  =>  DataHigh,  --
tWin => Twh, wCnt => Cwh,
                                                        SaveData => HighSave);


atwl : AsyncTimeWindow  port map (Clock => BoardClock, InputSignals => EventLow, ExtReset =>
IMReset,
                                                        Data  =>  DataLow,  --
tWin => Twl, wCnt => Cwl,
                                                        SaveData => LowSave);


memf : Fifo port map (DataIn => MData, WR => MSave, RD => MLoad, RS => ResetButton, Clk =>
BoardClock, DataOut => CData,
                                        FlagOfDone  =>  MDone,  FlagOfFull  =>  MFull,
FlagOfEmpty => MEmpty);
```

```
-------------------------------------------------------------------------------------------------------------------------------
-----------------------------
process(Events,LowSave,HighSave,DataLow,DataHigh)
begin
        EventLow(0) <= Events(0);
        EventLow(1) <= Events(1);
        EventLow(2) <= Events(2);
        EventLow(3) <= Events(3);

        EventHigh(0) <= Events(4);
        EventHigh(1) <= Events(5);
        EventHigh(2) <= Events(6);
        EventHigh(3) <= Events(7);

        TData(0) <= DataLow(0) or DataLow(1) or DataLow(2) or DataLow(3);

        TData(1) <= (DataLow(0) and DataLow(1)) or (DataLow(0) and DataLow(2)) or
                                (DataLow(0) and DataLow(3)) or (DataLow(1) and
DataLow(2)) or
                                (DataLow(1) and DataLow(3)) or (DataLow(2) and
DataLow(3));

        TData(2) <= (DataLow(0) and DataLow(1) and DataLow(2)) or (DataLow(0) and DataLow(1)
and DataLow(3)) or
                                (DataLow(0) and DataLow(2) and DataLow(3)) or
(DataLow(1) and DataLow(2) and DataLow(3));

        TData(3) <= DataLow(0) and DataLow(1) and DataLow(2) and DataLow(3);

        TData(4) <= DataHigh(0) or DataHigh(1) or DataHigh(2) or DataHigh(3);

        TData(5) <= (DataHigh(0) and DataHigh(1)) or (DataHigh(0) and DataHigh(2)) or
                                (DataHigh(0) and DataHigh(3)) or (DataHigh(1) and
DataHigh(2)) or
                                (DataHigh(1) and DataHigh(3)) or (DataHigh(2) and
DataHigh(3));
```

```
        TData(6) <= (DataHigh(0) and DataHigh(1) and DataHigh(2)) or (DataHigh(0) and DataHigh(1) and DataHigh(3)) or
                                        (DataHigh(0) and DataHigh(2) and DataHigh(3)) or (DataHigh(1) and DataHigh(2) and DataHigh(3));

        TData(7) <= DataHigh(0) and DataHigh(1) and DataHigh(2) and DataHigh(3);

        WrData <= LowSave and HighSave;

end process;

--Synchronizing modules: states changes
process(BoardClock, WrData, RdData)--, TimeReset)--, TimeRead)
begin
        if(BoardClock'EVENT and BoardClock = '1') then
                case actual is
                        when nonevent =>
                                if(WrData = '1') then actual <= buffering_input;
                                elsif(RdData = '1') then actual <= loading;
                                else actual <= nonevent;
                                end if;
                        when buffering_input =>
                                actual <= saving;
                        when saving =>
                                actual <= nonevent;
                        when loading =>
                                actual <= buffering_output;
                        when buffering_output =>
                                if(RdData = '0') then actual <= nonevent;

                                else actual <= buffering_output;
                                end if;
                end case;
        end if;
end process;

--Synchronizing modules: states actions
process(BoardClock, MSave, MLoad, MData, TData, IMReset)--, TimeReset)--, TimeRead)
begin
```

```
        if(BoardClock'EVENT and BoardClock = '0') then
                case actual is
                        when nonevent =>
                                TrigBuf <= "00000000";--TrigBuf;

                                IMReset <= '0';

                                MSave <= '0';
                                MLoad <= '0';
                                MData <= "00000000";

                        when buffering_input =>
                                TrigBuf <= TData;
                        when saving =>
                                TrigBuf <= TrigBuf;

                                IMReset <= '1';

                                MSave <= '1';
                                MLoad <= '0';
                                MData <= TrigBuf;

                        when loading =>
                                IMReset <= '0';

                                MSave <= '0';
                                MLoad <= '1';
                                MData <= "00000000";

                        when buffering_output =>
                                MLoad <= '0';
                end case;
        end if;
end process;



end Behavioral;
```

- **Interface Serial**

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

entity SPICommunication is
   Port ( Clk : in  STD_LOGIC;
          MasterOsc : in STD_LOGIC;
          SlaveInput : in  STD_LOGIC;
          Enable : in  STD_LOGIC;

          ParallelIn : in  STD_LOGIC_VECTOR (7 downto 0);
          ParallelOut : out STD_LOGIC_VECTOR (7 downto 0);
          Reading : out STD_LOGIC;
          WriteActive : out STD_LOGIC;
          WriteFinish : out STD_LOGIC;
          SlaveOutput : out  STD_LOGIC);
end SPICommunication;

architecture com of SPICommunication is
type state is (waiting,reccomand,definition,rdslave,wrslave);

signal spi: state;
signal RDFlag, WRFlag : STD_LOGIC;
signal Comand, CMDSaved : STD_LOGIC_VECTOR (7 downto 0);
signal CMDWR, CMDRD, CMDRS : STD_LOGIC;
signal CMDEnd, CMDEnable, RDEnd, WREnd : STD_LOGIC;

component RegBuffer is
   Port ( BufIn : in  STD_LOGIC_VECTOR (7 downto 0);
          BufOut : out STD_LOGIC_VECTOR (7 downto 0);
          BufWR : in  STD_LOGIC;
          BufRD : in  STD_LOGIC;
          BufRS : in STD_LOGIC;
          BufClk : in  STD_LOGIC);
end component;

component SerialParallel is
```

```
   Port ( Clk : in  STD_LOGIC;
          Enable : in STD_LOGIC;
          SignalInput : in  STD_LOGIC;
           Finish : out STD_LOGIC;
          ParallelOutput : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component ParallelSerial is
   Port ( Clk : in  STD_LOGIC;
           Enable : in STD_LOGIC;
           SignalOutput : out  STD_LOGIC;
           Finish : out STD_LOGIC;
          ParallelInputByte : in  STD_LOGIC_VECTOR (7 downto 0));
end component;

begin

spc : SerialParallel PORT MAP (Clk => MasterOsc, Enable => CMDEnable, SignalInput => SlaveInput,
Finish => CMDEnd, ParallelOutput => Comand);

cmd : RegBuffer PORT MAP (BufIn => Comand, BufOut => CMDSaved, BufWR => CMDWR, BufRD
=> CMDRD, BufRS => CMDRS, BufClk => Clk);

srd : ParallelSerial PORT MAP (Clk => MasterOsc, Enable => RDFlag, SignalOutput => SlaveOutput,
Finish => RDEnd, --ByteSelect => DataSelect, ParallelInputByte => ParallelIn);

swr : SerialParallel PORT MAP (Clk => MasterOsc, Enable => WRFlag, SignalInput => SlaveInput,
Finish => WREnd, ParallelOutput => ParallelOut);

Reading <= RDFlag;
WriteActive <= WRFlag;
WriteFinish <= WREnd;

process(Clk,Enable,RDFlag,WRFlag,CMDEnd,RDEnd,WREnd)
begin
        if(Clk'EVENT and Clk = '1') then
                case spi is
                        when waiting =>
                                if(Enable = '0') then spi <= reccomand;
```

```
                        else spi <= waiting;
                        end if;
                when reccomand =>
                        if(Enable = '0' and CMDEnd = '1') then spi <= definition;
                        else spi <= reccomand;
                        end if;
                when definition =>
                        if(Enable = '0' and RDFlag = '1') then spi <= rdslave;
                        elsif (Enable = '0' and WRFlag = '1') then spi <= wrslave;
                        else spi <= definition;
                        end if;
                when rdslave =>
                        if(Enable = '1' and RDEnd = '1') then spi <= waiting;
                        else spi <= rdslave;
                        end if;
                when wrslave =>
                        if(Enable = '1' and WREnd = '1') then spi <= waiting;
                        else spi <= wrslave;
                        end if;
        end case;
    end if;
end process;
process(Clk,SlaveInput,ParallelIn,CMDSaved,CMDEnable,CMDEnd,CMDWR,CMDRD,RDFlag,
RDEnd,WRFlag,WREnd)--,DataSelect)
begin
    if(Clk'EVENT and Clk = '1') then
        case spi is
                when waiting =>
                        CMDWR <= '0';
                        CMDRD <= '0';
                        CMDRS <= '1';
                        CMDEnable <= '0';
                        WRFlag <= '0';
                        RDFlag <= '0';
                when reccomand =>
                        CMDWR <= '1';
                        CMDRD <= '0';
                        CMDRS <= '0';
                        CMDEnable <= '1';
                when definition =>
```

```
                                        CMDWR <= '0';
                                        CMDRD <= '1';
                                        CMDRS <= '0';
                                        CMDEnable <= '0';
                                        if (CMDSaved = "00000001") then
                                                RDFlag <= '1';
                                        elsif (CMDSaved = "00000010") then
                                                WRFlag <= '1';
                                        else
                                                RDFlag <= '0';
                                                WRFlag <= '0';
                                        end if;
                                when rdslave =>
                                        CMDWR <= '0';
                                        CMDRD <= '0';
                                        CMDRS <= '0';
                                        RDFlag <= '1';
                                        CMDEnable <= '0';
                                when wrslave =>
                                        CMDWR <= '0';
                                        CMDRD <= '0';
                                        CMDRS <= '0';
                                        WRFlag <= '1';
                                        CMDEnable <= '0';
                        end case;
                end if;
        end process;


        end com;
```

- **Buffer de comando**

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


        entity RegBuffer is
           Port ( BufIn : in  STD_LOGIC_VECTOR (7 downto 0);
                  BufOut : out  STD_LOGIC_VECTOR (7 downto 0);
                  BufWR : in  STD_LOGIC;
```

```
        BufRD : in  STD_LOGIC;
         BufRS : in STD_LOGIC;
        BufClk : in  STD_LOGIC);
end RegBuffer;

architecture reg of RegBuffer is

signal data : STD_LOGIC_VECTOR (7 downto 0) := "00000000";

begin

process(BufClk,BufWR,BufRD)
begin
        if(BufClk'EVENT and BufClk = '1') then
                if (BufRS = '1') then
                        BufOut <= "00000000";
                        data <= "00000000";
                elsif(BufWR = '0' and BufRD = '0') then data <= data;
                elsif(BufWR = '1' and BufRD = '0') then data <= BufIn;
                elsif(BufWR = '0' and BufRD = '1') then BufOut <= data;
                else data <= data;
                end if;
        end if;
end process;

end reg;
```

- **Conversão Paralelo - Serial**

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

entity ParallelSerial is
   Port ( Clk : in  STD_LOGIC;
           Enable : in STD_LOGIC;
          SignalOutput : out  STD_LOGIC;
           Finish : out STD_LOGIC;
            ParallelInputByte : in  STD_LOGIC_VECTOR (7 downto 0));
```

```
end ParallelSerial;

architecture shift_reg of ParallelSerial is
signal Cnt : integer range 0 to 8 := 0;
begin
process(Clk,Cnt,Enable)
begin
        if (Enable = '0') then
                        Cnt <= 0;
                        SignalOutput <= '0';
        elsif(Clk'EVENT and Clk = '0') then
                if(Cnt < 8) then
                        SignalOutput <= ParallelInputByte(Cnt);
                        Cnt <= Cnt + 1;
                else
                        Cnt <= 8;
                        SignalOutput <= '0';
                end if;
        end if;
end process;

process(Cnt)
begin
        if(Cnt = 8 ) then Finish <= '1';--and Enable = '1') then Finish <= '1';
        else Finish <= '0';
        end if;
end process;
end shift_reg;
```

- **Conversão Serial - Paralelo**

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

entity SerialParallel is
   Port ( Clk : in  STD_LOGIC;
          Enable : in STD_LOGIC;
          SignalInput : in  STD_LOGIC;
```

```
        Finish : out STD_LOGIC;
        ParallelOutput : out  STD_LOGIC_VECTOR (7 downto 0));
end SerialParallel;

architecture shift_reg of SerialParallel is

signal Cnt : integer range 0 to 8 := 0;
signal Data : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
begin

process(Clk,Cnt,Data)
begin
        if(Clk'EVENT and Clk = '1') then
                if (Enable = '0') then
                        Cnt <= 0;
                        Data <= Data;--"00000000";
                elsif(Cnt < 8) then
                        Data(Cnt) <= SignalInput;
                        Cnt <= Cnt + 1;
                else
                        Cnt <= 8;
                        Data <= Data;
                end if;
        end if;
        ParallelOutput <= Data;
end process;

process(Cnt,Enable)
begin
        if(Cnt = 8 and Enable = '1') then Finish <= '1';
        else Finish <= '0';
        end if;
end process;
end shift_reg;
```

- **Controle do Sistema Integrado**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SystemControl is
    Port ( BoardClk : in  STD_LOGIC;
           FReset : in STD_LOGIC;
           ArduinoOsc : in  STD_LOGIC;
           ArduinoIn : in  STD_LOGIC;
           ArduinoEnb : in  STD_LOGIC;
           ArduinoOut : out  STD_LOGIC;
           FFull : out  STD_LOGIC;
           FEmpty : out  STD_LOGIC;
           MUXEnable : out STD_LOGIC_VECTOR (7 downto 0);
           MUXSelect : out STD_LOGIC_VECTOR (2 downto 0);
           DACEnable : out STD_LOGIC_VECTOR (7 downto 0);
           DACLine : out STD_LOGIC;
           DACConf: out STD_LOGIC;
           DACClk : out STD_LOGIC);
end SystemControl;

architecture Behavioral of SystemControl is

signal Cnt : integer range 0 to 500;
signal FastClock, SlowClock, MediumClock, Locked, DCMRst : STD_LOGIC;
signal TEvents : STD_LOGIC_VECTOR (7 downto 0);
signal DataTx : STD_LOGIC_VECTOR (7 downto 0);
signal Sending, Receiving, RcvDone : STD_LOGIC;
signal ReceivedCode     : STD_LOGIC_VECTOR (7 downto 0);

component DCMFreq is
  port ( U1_CLKIN_IN       : in    std_logic;
         U1_RST_IN         : in    std_logic;
         U1_CLKIN_IBUFG_OUT : out   std_logic;
         U1_CLK0_OUT       : out   std_logic;
         U1_CLK2X_OUT      : out   std_logic;
         U2_CLK0_OUT       : out   std_logic;
         U2_CLK2X_OUT      : out   std_logic;
         U2_LOCKED_OUT     : out   std_logic);
end component;
```

```
component SPICommunication is
   Port (                    Clk : in  STD_LOGIC;
                             MasterOsc : in STD_LOGIC;
                             SlaveInput : in  STD_LOGIC;
                             Enable : in  STD_LOGIC;

                             ParallelIn : in  STD_LOGIC_VECTOR (7 downto 0);

                             ParallelOut : out STD_LOGIC_VECTOR (7 downto 0);
                             Reading : out STD_LOGIC;
                             WriteActive : out STD_LOGIC;
                             WriteFinish : out STD_LOGIC;
                             SlaveOutput : out  STD_LOGIC);
end component;

component TesteCoincidencia is
   Port ( BoardClock : in  STD_LOGIC;
          ResetButton : in  STD_LOGIC;
                             Events : in STD_LOGIC_VECTOR (7 downto 0);
                             RdData : in STD_LOGIC;
                             MFull : out STD_LOGIC;
                             MEmpty: out STD_LOGIC;
                             CData : out  STD_LOGIC_VECTOR (7 downto 0)
                             );
end component;

component MuxDAC is
   Port (
                             ExternalClock : in  STD_LOGIC;
                             Code : in STD_LOGIC_VECTOR (7 downto 0);
                             Writing : in STD_LOGIC;
                             WriteComplete : in STD_LOGIC;
                             MUXEnableOutput : out STD_LOGIC_VECTOR (7 downto 0);
                             MUXCode : out STD_LOGIC_VECTOR (2 downto 0);
                             DACEnableOutput : out STD_LOGIC_VECTOR (7 downto 0);

                             DACSerial : out STD_LOGIC;
                             DACConfirm : out STD_LOGIC;
                             DACCLock : out STD_LOGIC
```

```
                                );
end component;

begin

spi : SPICommunication port map (Clk => SlowClock, MasterOsc => ArduinoOsc, SlaveInput =>
                                    ArduinoIn, Enable => ArduinoEnb, ParallelIn => DataTx,
                                    ParallelOut => ReceivedCode, Reading => Sending,
                                    WriteActive => Receiving, WriteFinish => RcvDone,
                                    SlaveOutput => ArduinoOut);

coi : TesteCoincidencia port map (BoardClock => SlowClock, ResetButton => FReset, Events =>
TEvents,
                                RdData => Sending, MFull => FFull, MEmpty => FEmpty, CData =>
DataTx);

cie : MuxDAC port map (ExternalClock => SlowClock, Code => ReceivedCode,
                                Writing => Receiving, WriteComplete => RcvDone,
                                MUXEnableOutput => MUXEnable, MUXCode => MUXSelect,
                                DACEnableOutput => DACEnable,          DACSerial => DACLine,
                                DACConfirm => DACConf, DACClock => DACClk);

mfqh : DCMFreq  port map (U1_CLKIN_IN   => BoardClk,
        U1_RST_IN         => DCMRst,
        U1_CLKIN_IBUFG_OUT => open,
        U1_CLK0_OUT       => SlowClock,
        U1_CLK2X_OUT      => MediumClock,
        U2_CLK0_OUT       => open,
        U2_CLK2X_OUT      => FastClock,
        U2_LOCKED_OUT     => Locked);
process(FastClock, Locked, Cnt)
begin
--          Anode <= '0';
            DCMRst <= '0';

            if(Locked = '1') then
                    if(FastClock'EVENT and FastClock = '1') then
                            if (Cnt /= 500) then
                                    Cnt <= Cnt + 1;
```

```
                else
                        Cnt <= 500;
                end if;
        end if;
        if(Cnt < 30) then TEvents <= "00000000";
        --(1)----------------------------------------
        elsif(Cnt = 34) then TEvents <= "00000010";
        elsif(Cnt = 37) then TEvents <= "00001000";
        elsif(Cnt = 47) then TEvents <= "00000100";
        ----------------------------------------
        elsif(Cnt = 78) then TEvents <= "00100000";
        elsif(Cnt = 81) then TEvents <= "10000000";
        elsif(Cnt = 91) then TEvents <= "01000000";
        ----------------------------------------
        --(2)----------------------------------------
        elsif(Cnt = 124) then TEvents <= "00000001";
        elsif(Cnt = 136) then TEvents <= "00001000";
        ----------------------------------------
        elsif(Cnt = 168) then TEvents <= "00010000";
        elsif(Cnt = 180) then TEvents <= "10000000";
        ----------------------------------------
        --(3)----------------------------------------
        elsif(Cnt = 204) then TEvents <= "10000000";
        ----------------------------------------
        elsif(Cnt = 228) then TEvents <= "00001000";
        ----------------------------------------
        --(4)----------------------------------------
        elsif(Cnt = 264) then TEvents <= "00000010";
        ----------------------------------------
        elsif(Cnt = 298) then TEvents <= "00100000";
        ----------------------------------------
        else TEvents <= "00000000";
        end if;
    else
        Cnt <= 0;
        TEvents <= "00000000";
    end if;
end process;

end Behavioral;
```

Apêndice B

Códigos para Arduino

- **Interface SPI USB**

```
#include <SPI.h>

/* SPI Pins:
 * MOSI: pin 11
 * MISO: pin 12
 * SCK: pin 13
 * CS: pin 10
 */

// set pin 10 as the slave select for the digital pot:
const int slaveSelectPin = 10;

//Formatdo das intruções:
//1) Leitura de dados salvos na memória interna: R
//2) Escrever comando para próxima operação do sistema (configuração ou aquisição extra de dados): WR OPCODE
//OPCODE: 0x00 - 0xFF (000 - 255)
int pos;
//unsigned int rcv;
byte rcv;
const int SIZECMD = 6;

void setup(){
  Serial.begin(9600);
  Serial.println("Programa que envia comando pelo cabo USB:\n ->R: Le valor da memoria interna\n ->WR: Define um comando ao sistema pelos codigos de 000 - 255");

  // set the slaveSelectPin as an output:
  pinMode (slaveSelectPin, OUTPUT);
  // configure SPI:
  // SPI_MODE0:clock base is 0 and data is captured on the clock's rising edge (low→high transition) and data is propagated on a falling edge (high→low clock transition)
  SPI.setDataMode(SPI_MODE0);
```

```
  //LSBFIRST: data shifted in Least Significant Bit (LSB) first
  SPI.setBitOrder(LSBFIRST);
  SPI.setClockDivider(10);
  // initialize SPI:
  SPI.begin();

  pos = 1;
  rcv = 0;
}

void loop(){
  if(Serial.available()>0){
    digitalWrite(slaveSelectPin,HIGH);

    int cmdChar = Serial.read();
    byte sendCmd = 0x00;
    byte opCode = 0x00;
    int comand[SIZECMD];

    comand[pos] = formatingComand(cmdChar);

    Serial.print("Foi escrito: ");
    Serial.print(comand[pos]);
    Serial.print(" na posicao ");
    Serial.println(pos);
    pos++;

      if(comand[1] == 82)
      {
        Serial.println("Transmitindo codigo de leitura e aguardando resposta ...");
        sendCmd = 0x01;           //read operation
        rcv = readOperation(sendCmd);
        Serial.print("Resposta recebida: ");
        Serial.print(rcv);
        Serial.print(" | ");
        Serial.println(rcv,BIN);

        Serial.print("Cod: ");
        Serial.println(sendCmd,BIN);
```

```
      pos = 1;
    }
    else if (pos == SIZECMD+1)
    {
      Serial.println("Transmitindo codigo de escrita e codigo da operacao ...");
      sendCmd = 0x02;           //write operation
      int iopc = comand[4]*100 + comand[5]*10 + comand[6];
      opCode = iopc & 0xFF;

      Serial.print("Formato binario: ");
      Serial.print(comand[4],BIN);
      Serial.print(" ");
      Serial.print(comand[5],BIN);
      Serial.print(" ");
      Serial.print(comand[6],BIN);

      Serial.print("\nValor do codigo: ");
      Serial.print(iopc);
      Serial.print(", convertendo para binario 0x");
      Serial.print(opCode,BIN);
      Serial.println(" ");

      writeOperation(sendCmd,opCode);
      Serial.println("Transmissao completa");

      Serial.print("Cod: ");
      Serial.println(sendCmd,BIN);

      pos = 1;
    }
  }
}

int formatingComand(int cmd)
{
  int cch = cmd;

  if(isDigit(cch)) {
```

```
    cch = cch  - 48;
  }
  else if(!isUpperCase(cch)) {
    cch  = cch  - 32;
  }

  return cch;
}

byte readOperation(byte cmd)
{
  byte response;

  digitalWrite(slaveSelectPin,LOW);
  //send the comand
  SPI.transfer(cmd);
  delay(0.005);
    response = SPI.transfer(0x00);
  digitalWrite(slaveSelectPin,HIGH);

  return response;
}

void writeOperation(byte cmd, byte opc)
{
  digitalWrite(slaveSelectPin,LOW);
  SPI.transfer(cmd);
  delay(0.005);
  SPI.transfer(opc);
  digitalWrite(slaveSelectPin,HIGH);
}
```