Fernando Marcio Barcellos de Sousa

Lógica programável aplicada à aquisição e transmissão serial de dados em alta velocidade via barramento PCI Express

Rio de Janeiro – RJ

Setembro / 2012

Fernando Marcio Barcellos de Sousa

Lógica programável aplicada à aquisição e transmissão serial de dados em alta velocidade via barramento PCI Express

Dissertação apresentada à Coordenação de Formação Científica do Centro Brasileiro de Pesquisas Físicas para a obtenção do título de Mestre em Física com ênfase em Instrumentação Científica.

Orientador:
Herman Pessoa Lima Junior

Co-orientador:
Pablo Diniz Batista

Ministério da Ciência, Tecnologia e Inovação
Centro Brasileiro de Pesquisas Físicas
Mestrado Profissional em Física

Rio de Janeiro – RJ

Setembro / 2012

A minha belíssima esposa Renata e as minhas filhas queridas, Fernanda e Juliana.

Agradecimentos

Em primeiro lugar, agradeço a Deus por conceder-me o dom da vida e uma visão singular da sua eterna graça e misericórdia.

Agradeço a minha esposa pelo incentivo nos momentos difíceis, por se alegrar comigo nas conquistas e por ter aturado todos os meus maus humores e "stresses" durante o período final do mestrado. Santa Renata!!! A minha filhotinha Fernanda, agradeço por ter me convencido algumas vezes a sair da frente do computador e ir brincar com ela.

Aos meus pais agradeço pelo exemplo de caráter ético e por não terem poupado esforços em investir na minha educação básica. A minha mãe em particular, agradeço por ter me incentivado a concluir esse trabalho, mas sempre dizendo para não descuidar da saúde. Ao meu pai Fernandão, que já partiu dessa vida, agradeço por me apoiar nos momentos difíceis, mas nunca sendo complacente com meus erros. Isso me fez crescer muito como ser humano. Agradeço também ao seu Fernandão por ter me incentivado a entrar no curso técnico do CEFET e, graças a isso, eu sustento a minha família até hoje.

Ao meu orientador Herman, agradeço por idealizar o projeto e acreditar que eu seria capaz de desenvolvê-lo. Agradeço a ele também pela oportunidade do convívio profissional, o que me proporcionou um enorme ganho em conhecimento técnico e científico.

Agradeço ao meu co-orientador Pablo pela grande colaboração na parte de programação do projeto. Sem a ajuda dele, o desenvolvimento dos programas seria muito mais demorado. Ainda bem que a corda sempre arrebentou!

Ao pesquisador André Massafferri pelo apoio e colaboração no financiamento

do projeto. Agradeço também a todos os professores do mestrado profissional pela dedicação e transferência de conhecimento, em especial aos professores Marcio Portes de Albuquerque e ao saudoso professor Laudo, pelos preciosos conselhos profissionais. Gostaria de agradecer também aos colaboradores do LSD, Luciano, PC e Rafael Nóbrega pelas valiosas opiniões, ainda que informais, sobre o projeto. Ao Leonardo Lessa, agradeço por disponibilizar a 1ª versão do driver PCIe para Linux.

Ao coordenador da CFC, professor Ivan Oliveira, pela compreensão de alguns contratempos que surgiram na etapa final do projeto. Ao professor Gilvan Augusto, meu coordenador no Lafex, pela compreensão de ter no laboratório do departamento, um colaborador e aluno do mestrado profissional ao mesmo tempo . A louríssima Sônia, sempre prestativa e disposta a me ajudar.

Aos amigos, Marcelo Giovani e Ednardo Miranda, muito obrigado pelos momentos de incentivo, descontração e degustação cervejeira. Aos companheiros Rafael Gama, Vitor baiano e Gabi, agradeço pelas rodadas de café na livraria acompanhadas de um besteirol sem igual. Ao Rafael Gama, em especial, agradeço por ter me ajudado em vários momentos com seu vasto conhecimento de lógica programável.

Ao meu grande professor do CEFET, Arídio, que reencontrei no CBPF depois de 20 anos, agradeço por ter me esclarecido alguns detalhes técnicos que poderiam ter passado em branco.

Agradeço ao CBPF por dar a oportunidade de crescimento acadêmico e profissional a centenas de estudantes como eu. A CAPES, RENAFAE e demais instituições que contribuíram para o desenvolvimento desse trabalho.

Gostaria de finalizar agradecendo ao restante da família e amigos, apesar de não ser possível citá-los aqui, pelo apoio e incentivo que certamente contribuíram de alguma forma para que eu seguisse em frente.

Resumo

Este documento descreve o desenvolvimento de um módulo eletrônico programável com aplicações em instrumentação científica e controle de processos. O principal destaque do ponto de vista tecnológico está na utilização do protocolo PCI *Express* como interface de comunicação entre o módulo e um computador pessoal. Este dispositivo será capaz de realizar aquisição de dados e controle de experimentos, análise espectral, processamento digital de sinais, controle e monitoração de processos industriais, entre outras aplicações. Serão apresentadas as contribuições dadas ao projeto Neutrinos Angra, através de algumas medidas realizadas com tubos foto-multiplicadores (*Photomultiplier tubes* - PMT) utilizados no detector principal. Serão também apresentados os resultados obtidos em medidas de impedância utilizando a técnica de Espectroscopia por Impedância Elétrica (EIE).

Abstract

This document describes the development of a programmable electronic module with applications in scientific instrumentation and process control. The main technological highlight is the use of the PCI Express protocol as the communication interface between the module and a personal computer. This device will be able to perform data acquisition and experiment control, spectral analysis, digital signal processing, control and monitoring of industrial processes, among other applications. Contributions to the Neutrinos Angra project will be presented with measurements performed with the photomultiplier tubes employed in the targed detector. It will also be presented results obtained from impedance measurements by using the Electrical Impedance Spectroscopy technique.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

PMT	*Photomultiplier tubes*,	p. iv
EIE	Espectroscopia por Impedância Elétrica,	p. iv
ADC	*Analog-to-digital converter*,	p. 1
LSD	Laboratório de Sistemas de Detecção,	p. 1
Lafex	Laboratório de Altas Energias e Física Experimental,	p. 1
LHCb	*Large Hadron Collider beauty*,	p. 2
LHC	*Large Hadron Collider*,	p. 2
ECS	*Experiment control system*,	p. 2
PCIe	PCI *Express*,	p. 2
MOPI	Módulo programável para instrumentação científica e controle de processos,	p. 2
TDC	*Time-to-digital converter*,	p. 4
FPGA	*Field Programmable Gate Array*,	p. 4
ISA	*Industry Standard Architecture*,	p. 6
PCI-SIG	*Peripheral Component Interconnect Special Interest Group*,	p. 7
PCI	*Peripheral Component Interconnect*,	p. 7
AGP	*Accelerated Graphics Port*,	p. 7
USB	*Universal Serial Bus*,	p. 10
CRC	*Cyclic Redundancy Check*,	p. 13
TLP	*Transaction Layer Packet*,	p. 15
DLLP	*Data Link Layer Packet*,	p. 16
JTAG	*Joint Test Action Group*,	p. 20
IR	*Instruction Register*,	p. 20

BSR	*Boundary Scan Register*,	p. 20
SPI	*Serial Peripheral Interface*,	p. 22
I^2C	*Inter-Integrated Circuit*,	p. 24
bdf	*block design file*,	p. 26
SOPC	*System on a programmable-chip*,	p. 27
FIFO	*First-in First-out*,	p. 31
SPS	*Samples per second*,	p. 35
DDS	*Direct Digital Synthesis*,	p. 36
LVTTL	*Low Voltage Transistor-Transistor Logic*,	p. 36
FSK	*Frequency Shift-Keying*,	p. 36
PSK	*Phase Shift-Keying*,	p. 36
DAC	*Digital-to-analog converter*,	p. 37
VCXO	*Voltage-controlled crystal oscillator*,	p. 38
XO	*Crystal oscillator*,	p. 39
DNL	*Differential non-linearity*,	p. 52
SPE	*Single photon electron*,	p. 54
DUT	*Device Under Test*,	p. 70
PLL	*Phase-locked loop*,	p. 75
DMA	*Direct memory access*,	p. 88

Introdução

Apresentação

A evolução da ciência, e consequentemente do conhecimento humano, através dos séculos sempre esteve ligada ao desenvolvimento de instrumentos que pudessem ampliar o horizonte de observação dos pesquisadores. Em outras palavras, a medida que as teorias eram confirmadas através de experimentos, novos experimentos eram criados com o objetivo de ir mais além no conhecimento. Com o advento da eletrônica digital, uma técnica em particular, vem sendo praticada com bastante frequência no intuito de estudar eventos e comportamentos de partículas sub-atômicas: a digitalização de sinais. Essa técnica gira em torno de um dispositivo eletrônico chamado conversor analógico-digital (*Analog-to-digital converter* - ADC) . Esse dispositivo converte um sinal analógico em digital (bits), utilizando técnicas de amostragem de sinais e codificação binária. O sinal digitalizado pode ser armazenado em memórias e ser submetido aos mais diversos tipos de processamento e análises computacionais. No Centro Brasileiro de Pesquisas Físicas, em particular no Laboratório de Sistemas de Detecção (LSD), podemos observar uma significativa evolução na produção de módulos eletrônicos capazes de digitalizar e processar dados.

Atualmente, e como tema dessa dissertação, o LSD em parceria com o Laboratório de Altas Energias e Física Experimental (Lafex), está realizando os testes iniciais em um novo módulo eletrônico que será capaz de realizar aquisição de dados e transferí-los para um computador a uma taxa próxima de 1 GBytes/s.

Motivação e objetivos

Com o objetivo de melhorar a estatística, o LHCb (*Large Hadron Collider beauty*) [1] aumentará a luminosidade para $10^{33}\ cm^{-2}\ s^{-1}$ a partir de 2017. Visando otimizar o tratamento de um maior volume de dados gerados, está prevista uma atualização dos sistemas de aquisição e transmissão de dados. Os dados serão adquiridos a uma frequência de 40 MHz, limite natural dado pela frequência de colisões do LHC (*Large Hadron Collider*), enquanto que a taxa de transmissão de dados, hoje em 1 Gbits/s, aumentará para 10 Gbits/s devido a troca dos módulos de transmissão de dados. O CBPF (Centro Brasileiro de Pesquisas Físicas) contribuirá com essa atualização desenvolvendo o sistema de controle do novo módulo de aquisição, parte do ECS (*Experiment Control System*) [2]. A principal modificação nesta unidade de controle consiste na incorporação do protocolo de alta velocidade PCI *Express* [3].

Motivado com o desenvolvimento desta tecnologia, o CBPF, através do LSD e do Lafex, propôs desenvolver, em paralelo à atualização do ECS, um módulo eletrônico programável para instrumentação científica e controle de processos (MOPI). Dentre outras características, este módulo será capaz de realizar aquisição de dados, fazer análise espectral de sinais contínuos, processamento digital de sinais, controle de experimentos e controle e monitoração de processos industriais. Utilizando o protocolo PCI *Express*, todas essas tarefas poderão ser realizadas com alta capacidade de processamento e alcançando elevada velocidade nas medidas. Desta forma, espera-se contribuir com a instrumentação científica, com os laboratórios do CBPF e com pesquisadores de outras instituições de pesquisa. É importante destacar que estas atividades também irão contribuir para o desenvolvimento da tecnologia nacional uma vez que algumas ferramentas auxiliares, que serão apresentadas ao longo do trabalho, foram desenvolvidas durante a pesquisa e poderão ser utilizadas em outros projetos do CBPF.

Aplicações

Projeto Neutrinos Angra

O Projeto Neutrinos Angra visa detectar o fluxo de antineutrinos emitidos pelo reator nuclear a fim de monitorar parâmetros relacionados à atividade do reator, como a composição isotópica do combustível e a potência térmica instantânea liberada por ele. Esse projeto é coordenado pelo CBPF e possui a participação de algumas instituições do país.

Como a potência produzida em um reator é diretamente proporcional ao número de fissões por unidade de tempo, a taxa de produção de antineutrinos é diretamente proporcional à potência térmica do reator. No caso do reator de Angra II, por exemplo, a taxa de produção, em condições nominais de operação, pode chegar a 10^{20} antineutrinos por segundo. Os antineutrinos são partículas com probabilidade muita baixa de interação com a matéria. No entanto, eles podem ser detectados indiretamente através da reação de decaimento *beta* inverso [4], onde um pósitron e um nêutron são produzidos quando o antineutrino [5] interage, eventualmente, com um próton.

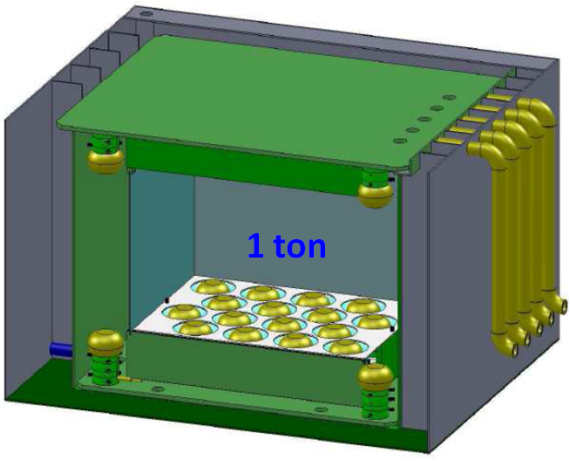


Figura 0.1: Detector principal do projeto Neutrinos Angra.

Atualmente, o detector do projeto está em fase de construção. O detector principal, onde é esperada a interação dos antineutrinos pela reação de decaimento *beta*

inverso, é constituído por um volume cúbico preenchido com uma tonelada de água dopada com Gadolínio. Nessa solução, e devido ao efeito Cherenkov, a interação produz fótons com um intervalo de tempo característico ($\approx$ 30 μs), que podem ser detectados por 40 tubos fotomultiplicadores [6] dispostos ao redor do volume, como mostra a Figura 0.1. Os PMT's estão conectados a circuitos de *front-end*, que otimizam o sinal antes de enviá-lo ao sistema de aquisição de dados.

Com o objetivo de minimizar os efeitos de um dos principais tipos de ruído denominado pulso de corrente de escuro [6], foi desenvolvido um TDC (*Time to digital converter*) sintetizado em uma FPGA (*Field Programmable Gate Array*) com o objetivo de investigar o comportamento desses eventos em alguns dos PMT's utilizados no detector principal do projeto Neutrinos Angra.

Espectroscopia de impedância elétrica - EIE

Esse tipo de espectroscopia consiste em submeter o sistema em análise a um sinal elétrico alternado, com amplitude constante e frequência variável [7]. O resultado obtido é um espectro de impedância em função da frequência aplicada. Essa informação é relevante, pois o conhecimento do comportamento elétrico dos materiais tem grande importância para caracterização dos mesmos.

A EIE possibilita o entendimento dos processos condutivos e suas causas, uma vez que a impedância está relacionada com diversas características do material como composição química, pureza, grau de homogeneidade, tamanhos de grãos etc. A título de exemplo destaca-se o estudo de células cancerígenas [8], onde é possível determinar o quanto uma célula pode estar comprometida, em relação a uma célula considerada normal, por meio da comparação dos espectros de impedância.

Com o objetivo de contribuir com futuras pesquisas no CBPF envolvendo EIE, o sistema desenvolvido foi utilizado como elemento principal na realização de algumas medidas preliminares utilizando essa técnica.

Visita guiada ao documento

No primeiro capítulo, é mostrada a evolução tecnológica dos barramentos de comunicação utilizados em computadores pessoais, finalizando com a apresentação do barramento PCI *Express*, utilizado no MOPI.

O capítulo 2 mostra as ferramentas de desenvolvimento utilizadas no projeto do módulo eletrônico.

O terceiro capítulo traz, de forma detalhada, os circuitos do *hardware* e o *firmware* sintetizado no FPGA.

No quarto capítulo são apresentadas as medidas experimentais realizadas. Na primeira seção é mostrada uma contribuição ao projeto Neutrinos Angra e, na segunda, as medidas de Espectroscopia de Impedância Elétrica. Será mostrado todo o processo de calibração da instrumentação até a realização das medidas de interesse.

No capítulo 5, são apresentadas as conclusões e as expectativas futuras em relação a utilização do MOPI no CBPF.

Os esquemáticos elétricos, assim como toda a parte de software (incluindo os fluxogramas) são apresentados nos Apêndices e Anexos.

1 Barramentos de comunicação

Desde o surgimento dos primeiros computadores pessoais muitos tipos de barramentos de entrada e saída foram criados. Isto aconteceu, principalmente, devido ao desenvolvimento de processadores cada vez mais rápidos, softwares mais complexos, e um grande aumento na utilização dos recursos multimídia. Para acompanhar esse avanço tecnológico, era imperativo que os barramentos de entrada e saída fossem tão rápidos o quanto possível. Novos barramentos de entrada e saída tem um aparecimento lento principalmente pela necessidade da manutenção da compatibilidade com padrões anteriores. A compatibilidade é essencial, pois não permite que milhares de módulos de entrada e saída fiquem obsoletos como consequência do surgimento de um novo barramento no mercado.

1.1 Barramentos paralelos e seriais para computador

1.1.1 Barramentos paralelos

O barramento ISA (*Industry Standard Architecture*) [9] foi introduzido no IBM PC original em 1981, na versão de 8 bits que operava a uma velocidade de 4,77 MHz. Fisicamente, este conector possui 62 contatos com 8 linhas de dados e 20 linhas de endereços podendo gerenciar 1 MBytes de memória. Com o lançamento do PC/AT em 1984, equipado com o processador 286 e um barramento de dados de 16 bits, foi possível estabelecer a comunicação de dados entre o processador, a placa-mãe e a memória com palavras de 16 bits ao invés dos 8 bits anteriores. A compatibilidade foi mantida e o novo sistema suportava os barramentos de 8 ou 16 bits. O PC/AT

foi introduzido no mercado com conectores de expansão nos quais se podia conectar placas de 8 bits na primeira parte do conector ou placas de 16 bits em ambas as partes. Ficou estabelecido pelos fabricantes que 8,33 MHz seria a velocidade máxima para ambas as versões do barramento ISA. O conector de 16 bits incorporava mais 36 contatos, que seriam necessários para transportar os sinais adicionais.

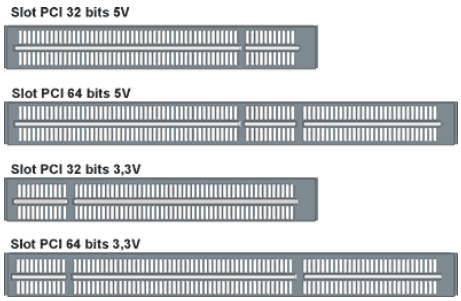


Figura 1.1: Conectores utilizados no barramento PCI.

No início de 1992, a Intel promoveu a criação do comitê PCI-SIG (*Peripheral Component Interconnect Special Interest Group*) com o objetivo de especificar um novo barramento de comunicação de dados que superasse as dificuldades técnicas impostas pelos barramentos utilizados na época. O novo barramento, chamado PCI (*Peripheral Component Interconnect*) [10], foi lançado em junho de 1992 redesenhando a arquitetura do PC ao inserir um barramento local que fazia uma ponte entre a CPU e o barramento de entrada e saída. Dessa forma, evitou-se a ligação de dispositivos diretamente ao barramento do processador e todos os problemas elétricos e de temporização que isso acarretava. A versão mais utilizada foi a de 32 bits, tensão de alimentação 5 Volts e frequência de 33MHz. A Figura 1.1 mostra as interfaces físicas de algumas versões do barramento PCI.

O barramento AGP (*Accelerated Graphics Port*) [11] foi lançado em julho de 1996, e conforme o próprio nome indica, foi projetado especificamente para conectar placas gráficas. Esse barramento é uma extensão da tecnologia PCI na sua versão de 66 MHz, mas com apenas 32 bits. Foi criado com dois grandes objetivos: liberar o barramento PCI dos dados relativos à placa gráfica, permitindo que este ficasse

com mais recursos para realizar outras tarefas, e proporcionar uma largura de banda maior para a placa gráfica, melhorando o seu desempenho.

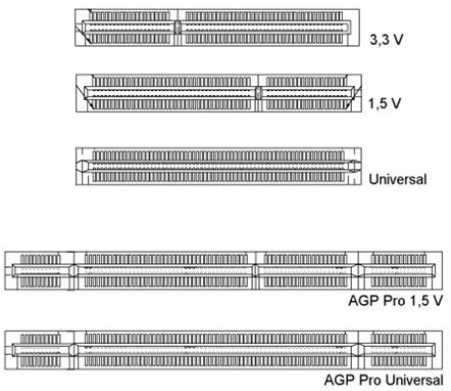


Figura 1.2: Conectores utilizados no barramento AGP.

O barramento AGP, inicialmente, operava em dois modos: x1 e x2. Esses modos de operação dizem respeito a quantidade de dados que são transferidos por pulso de *clock*. O AGP x1 é capaz de transferir apenas um dado por pulso de *clock*. Como o barramento AGP opera a 66 MHz, transferindo 32 bits de dados por vez, a taxa de transferência máxima do barramento AGP x1 é de aproximadamente 266 MBytes por segundo. O barramento AGP x2 trabalhava transferindo dois dados por pulso de *clock* o que resulta em uma taxa de transferência de aproximadamente 533 MBytes por segundo.

A segunda versão do barramento AGP introduziu o modo de operação x4, que permite transferir quatro dados por pulso de *clock*, obtendo assim uma taxa de transferência de 1 GByte/s. Algum tempo depois surgiu o AGP 3.0, que operava com tensão de alimentação igual a 0,8 V e modo de operação de x8, correspondendo a uma taxa de transferência de 2 GBytes/s. Devido as várias versões desse barramento, também existem variações nos tipos de conectores, como ilustra a Figura 1.2.

1.1.2 Barramentos seriais

Devido a vários fatores que limitam o desempenho dos barramentos paralelos, os investimentos tecnológicos, atualmente, tem se direcionado ao desenvolvimento dos barramentos seriais. Dentre esses fatores limitadores, pode-se destacar a temporização, como um dos mais críticos. O barramento paralelo deve sincronizar a chegada de vários sinais com o sinal de *clock* para que as transferências de dados ocorram. Variações comuns nas características físicas do circuito, tais como impedância, a carga do sinal, e o roteamento das trilhas, causam distorções entre os sinais no barramento paralelo. É possível controlar de maneira rigorosa essas características físicas a fim de minimizar o atraso, a distorção, e a temporização do *clock* até que todos os sinais cheguem de maneira confiável ao destino. No entanto, basta aumentar a velocidade do *clock* para que esses parâmetros comecem a falhar novamente.

À medida que a frequência de operação aumenta, as dificuldades em controlar as características físicas do circuito também aumentam. Dessa forma, a única maneira para otimizar a largura de banda do barramento paralelo, sem aumentar a frequência de *clock*, é aumentando o número de linhas. Mas esse aumento de linhas agrava o problema da distorção e gera um outro problema: o aumento do número de pinos de entrada e saída que o circuito integrado da *interface* do barramento deve possuir para se conectar. Atualmente, adicionar pinos em circuitos integrados tem um custo elevado. Desta forma, o aumento da largura do barramento em função da inclusão de mais pinos nas interfaces levará os barramentos paralelos a tornarem-se comercialmente inviáveis.

A maioria das limitações dos barramentos paralelos são evitadas num contexto de utilização do barramento serial. Embora tais estruturas prometam um melhor desempenho, projetá-las é um trabalho bem mais complexo do que projetar barramentos paralelos. O grande segredo dos barramentos seriais é a utilização do *clock* auto-sincronizado, ou *self clocking* [12]. Esse tipo de *clock* fica embutido no sinal sendo decodificado posteriormente, não havendo a necessidade de um sinal de *clock* em linhas independentes. Portanto, não há atraso entre as linhas de dados e de

clock para limitar a velocidade do barramento serial. Devido a essa característica, o barramento serial pode conseguir uma maior largura de banda, transmitindo blocos de dados, ao invés de bits individuais, como fazem os barramentos paralelos. Barramentos seriais são altamente escalonáveis e permitem aumentar a largura de banda agregando-se múltiplos barramentos organizados em vias de dados, ou *lanes*, entre os dispositivos. A *interface* do barramento divide o bloco de dados (normalmente em bytes) entre os *lanes* para o transporte, e os reagrupa quando chegam ao destino na outra extremidade. Usar blocos de dados maiores para aumentar o rendimento significa simplesmente adicionar mais *lanes*.

A interface do barramento aguarda até que todos os *lanes* tenham entregado a sua parte do bloco de dados para poder remontá-lo. A remontagem é normalmente concluída com a transmissão de um caractere de alinhamento para todos os *lanes* simultaneamente. O *self clocking* e a remontagem inteligente em barramentos seriais simplificam o projeto de *layout*, porque os barramentos acomodam as distorções e atrasos através das trilhas durante o processo. Restrições de roteamento ocorrem apenas quando sinais diferenciais possuem *clock* elevado [12]. Apesar disso, o roteamento de apenas duas trilhas diferenciais mantém as deformações do sinal sob controle. Os sinais em outros *lanes* podem seguir caminhos diferentes.

Os três barramentos seriais listados a seguir destacam-se por atingirem taxas de transmissão de dados superiores a 100 MBytes/s:

- IEEE 1394 (*FireWire*) [13].
- USB (*Universal Serial Bus*).
- PCI *Express*.

A seguir serão vistas algumas características do barramento USB e na próxima seção será destacado o barramento PCI *Express*.

Lançado em 1998, o barramento USB [14] surgiu com o principal objetivo de facilitar a instalação de periféricos de entrada e saída em um PC, sem a necessidade

de ter que abrir o equipamento. Inicialmente, foram lançadas as versões 1.0 e 1.1 com largura de banda de 1,5 MBytes/s e 12 MBytes/s, respectivamente. No ano de 2000, atendendo a crescente necessidade de velocidade nos dispositivos USB, foi lançada a versão 2.0, que possui uma largura de banda de 480 MBytes/s. Atualmente, o padrão USB encontra-se na versão 3.0 e possui uma largura de banda de 5 Gbits/s.

O sistema USB consiste de um dispositivo central (*hub*-raiz) conectado ao barramento local do PC que fornece canais de entrada ou saída, através conectores, para diversos dispositivos.

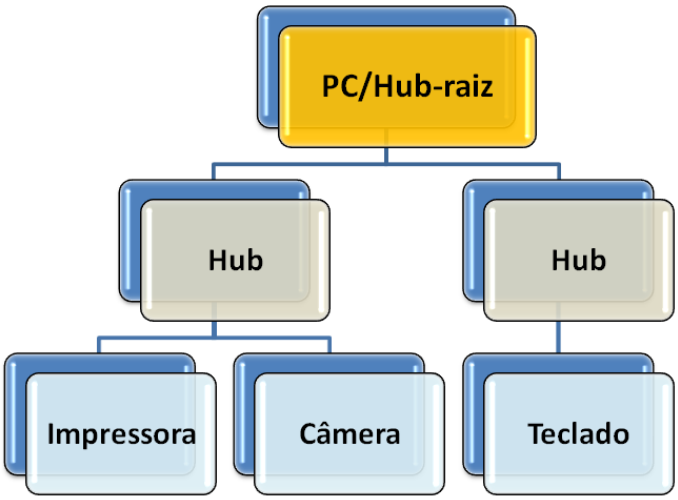


Figura 1.3: Topologia do protocolo USB.

Em termos lógicos, o sistema USB pode ser visto como um conjunto de ramificações que saem do *hub*-raiz para os dispositivos de entrada e saída, como mostra a Figura 1.3. Cada dispositivo pode subdividir sua própria ramificação em até dezesseis ramos secundários para diferentes tipos de dados. Dentro de cada ramo secundário os dados fluem de maneira bidirecional entre o *hub*-raiz e o dispositivo. Não há tráfego de informações entre dispositivos de entrada e saída.

O controle da comunicação é baseado na transmissão de blocos de informação pelo *hub*-raiz para o dispositivo. A cada intervalo de 1 ms, o *hub*-raiz transmite um novo bloco, que consiste de um ou mais pacotes de informação, para manter todos os dispositivos sincronizados no tempo, mesmo que não haja tarefas a executar.

Existem quatro tipos de pacotes, relacionados a seguir:

- Permissão.
- Dados.
- Apresentação.
- Especial.

Os pacotes de permissão são enviados pelo *hub*-raiz para um dispositivo e servem para controle do sistema. O pacote de permissão SOF (*Start of Frame* - Início de Quadro) é o primeiro de cada bloco e também define o início do mesmo. Caso não haja nenhum trabalho a realizar, o pacote SOF é único no quadro. O pacote de permissão "IN" é uma verificação e solicita ao dispositivo que retorne dados específicos.

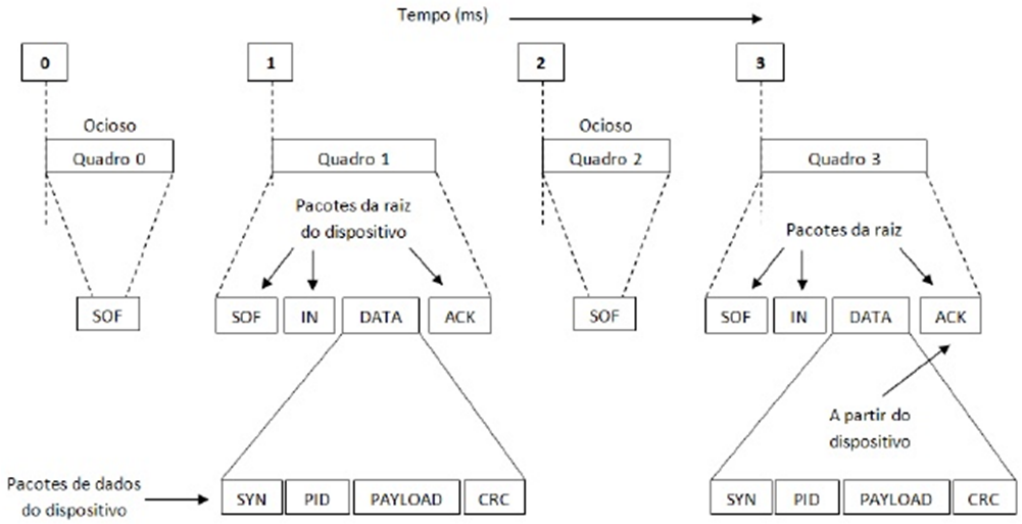


Figura 1.4: Exemplo de pacotes de dados em uma comunicação USB.

O pacote de permissão "OUT" informa ao dispositivo que serão enviados dados para ele. Um quarto tipo de pacote de permissão, chamado SETUP, é usado para configuração. Em relação ao pacote de dados mostrado na Figura 1.4, pode-se dizer que o 1º byte é uma informação de sincronização (SYN), seguido do segundo byte de

identificação da operação (PID). O quarto e o quinto bytes correspondem aos dados (*payload*), e os dois últimos bytes são utilizados para detecção de erros, segundo a técnica CRC (*Cyclic Redundancy Check*) [15].

Quanto aos pacotes de apresentação, podemos destacar os seguintes: O pacote "ACK" informa que o pacote de dados anterior foi recebido corretamente, o "NAK" informa que foi detectado um erro CRC e o pacote "STALL" é uma sinalização de ocupado.

Os pacotes especiais são responsáveis por algumas informações como mudança de velocidade ou erros de transmissão e verificar se algum dispositivo está ocupado.

1.2 O barramento PCI *Express*

Lançado pela Intel em 2004, o barramento[1] PCIe foi desenvolvido com o objetivo de resolver vários problemas que vinham atrapalhando o avanço tecnológico na área de transmissão de dados em alta velocidade e como consequência substituir os padrões AGP e PCI. O protocolo PCIe mantém a compatibilidade com o modelo de endereçamento do barramento PCI, de forma a garantir que todos os aplicativos e drivers existentes funcionem normalmente [3]. Entretanto, é importante ressaltar que apesar dessa compatibilidade, o PCIe é um barramento serial, ao passo que o PCI é paralelo, não sendo possível utilizar uma placa PCI em um barramento PCIe e vice-versa.

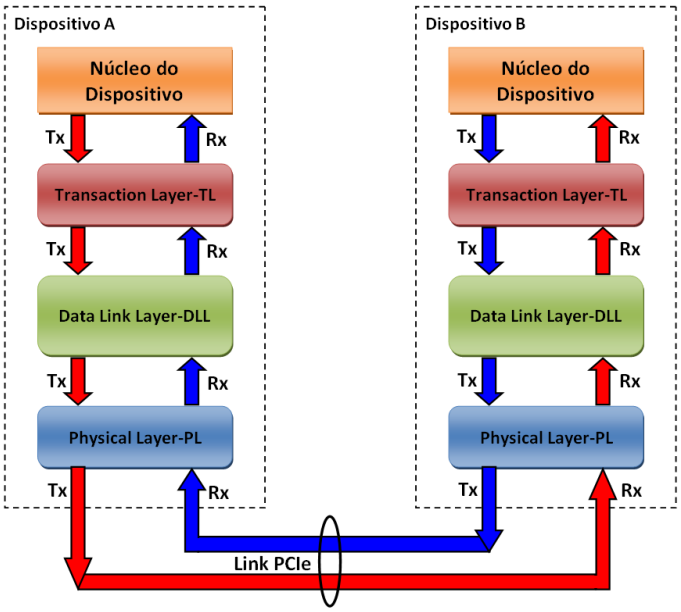


Figura 1.5: Diagrama ilustrativo de um *link* PCIe e as camadas do protocolo.

O protocolo PCIe baseia-se na conexão física ponto a ponto entre dois dispositivos, onde cada dispositivo periférico possui um canal exclusivo de comunicação com

[1]Há contradições quanto a forma de se referir ao PCIe como sendo um barramento, uma vez que o termo "barramento" surgiu para descrever um canal de comunicação compartilhado por vários dispositivos ou periféricos. No entanto, em toda documentação pesquisada é utilizado o termo "PCI *Express bus*" para mencioná-lo. Talvez isso ocorra, pois da mesma forma que a tecnologia evolui, os termos técnicos também estejam sujeitos a isso.

o dispositivo PCIe principal. Essa conexão física é chamada de *link*. Cada *link* pode conter até 32 *lanes*, e cada *lane* possui dois pares de sinais diferenciais [16], um para transmissão e outro para recepção. A transmissão dos dados é efetuada utilizando um protocolo baseado em pacotes de informação. Esses pacotes são enviados e recebidos serialmente, e distribuídos em bytes, através dos *lanes* disponíveis no *link*. Quanto mais *lanes* são implementados no *link*, mais rápido o pacote é transmitido e maior é a largura de banda do *link*. A arquitetura do protocolo PCIe é baseada na operação de três camadas, como mostra a Figura 1.5. O PCIe pode chegar a uma velocidade de 2,5 Gbits por segundo por direção (2,5 Gbits/s/d), resultando em 250 MBytes/s por cada *lane*, o que é aproximadamente duas vezes o valor da taxa de dados do PCI clássico. Ao longo desta seção serão apresentados mais detalhes sobre cada camada do protocolo PCIe.

1.2.1 *Transaction Layer* (TL)

A TL é a 1^a camada da arquitetura do protocolo PCIe, interagindo com o dispositivo PCIe e com a DLL (*Data Link Layer*). A TL é responsável por iniciar o processo de transmissão de dados transformando uma requisição ou um pacote de dados, provenientes de um dispositivo, em um TLP (*Transaction Layer Packet*). Um TLP é um pacote de informação composto por um cabeçalho codificado para identificar o tipo de operação a ser realizada, e opcionalmente por dados, conforme mostra a Figura 1.6.

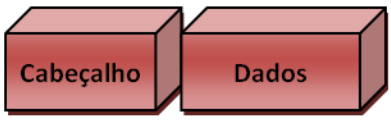


Figura 1.6: Pacote PCIe com os blocos formados pela 1^a camada TL.

O TLP é enviado da TL de um dispositivo transmissor para a TL de outro dispositivo receptor. Como exemplo, pode-se pensar em uma requisição de leitura de dados da memória por um dispositivo. Ao receber essa requisição, a TL transmissora codifica essa operação em um TLP e o repassa através do *link*. A TL do dispositivo

receptor decodifica essa operação e a repassa para as camadas de *software* para ser processada. Quando os dados solicitados são disponibilizados, a TL receptora organiza-os em um novo TLP de resposta e os encaminha à TL requisitante. Cada TLP tem uma identificação individual permitindo que sejam direcionados para o solicitante correto.

1.2.2 *Data Link Layer* (DLL)

A DLL é a 2^a camada da arquitetura do protocolo PCIe, interagindo com a TL e com a PL (*Physical Layer*). O principal objetivo dessa camada é garantir a integridade dos dados que circulam através do *link* PCIe. Para tanto, a DLL adiciona uma sequência numérica no início do pacote e uma codificação para checagem de erro no final do mesmo, conforme ilustra a Figura 1.7.

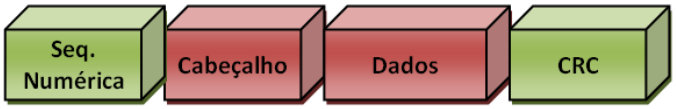


Figura 1.7: Pacote PCIe : TL + DLL.

Utiliza-se a técnica CRC para a detecção de erros. Pode-se descrever, resumidamente, esse processo da seguinte forma: no dispositivo transmissor, a DLL recebe os TLP's encaminhados pela TL, faz a codificação e os envia para a PL. No dispositivo receptor, a DLL recebe o pacote da PL, faz a checagem de erro, e se tudo estiver correto, direciona o pacote para a TL receptora. Caso a DLL encontre algum erro no pacote recebido, esse pacote não é enviado para a TL receptora. Nesse caso, a DLL cria um pacote chamado DLLP (*Data Link Layer Packet*) que é devolvido para a DLL transmissora. Esse pacote possui informações específicas, tais como gerenciamento de hierarquia, notificação de erro, controle de fluxo de dados do *link* etc. De volta ao dispositivo transmissor, o pacote, agora sob a forma de um DLLP, é verificado pela DLL original. Dependendo do tipo de erro detectado, o pacote pode ser reenviado pela DLL para o dispositivo receptor para que seja efetuada uma nova tentativa de decodificação, ou simplesmente descartado. Em caso de descarte, a TL transmissora é informada e faz uma nova requisição. É importante perceber que,

excluindo este último caso, o controle de erros é totalmente gerenciado pelas DLL's e transparente para as TL's.

1.2.3 *Physical Layer* (PL)

A PL é a 3^a e última camada da arquitetura do protocolo PCIe, sendo responsável efetivamente pelo envio e recebimento de todos os dados através do PCIe *link*. A PL interage com a DLL e com o *link* físico PCIe (fios, cabos, fibra óptica, conectores) [17]. Essa camada possui todos os circuitos para operação da interface: *buffer's* de entrada e saída, conversores paralelo-serial e serial-paralelo, PLL's e circuitos de "casamento" de impedâncias. Também possui uma lógica para inicialização da interface e para mantê-la ativa.

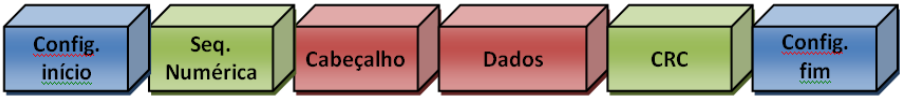


Figura 1.8: Pacote PCIe completo: TL + DLL + PL.

No bloco transmissor, a PL recebe a informação da DLL e a converte em um formato serial próprio, no qual é inserido um sistema de codificação chamado 8 bits / 10 bits, onde são incluídos dois bits adicionais para cada byte de dados transmitidos (*self-clocking*). Estes bits adicionais eliminam a necessidade de linhas para enviar o sinal de *clock*, simplificando bastante o projeto, melhorando a confiabilidade e contribuindo para alcançar as altas taxas de transmissão de dados. Atualmente, esta taxa está em 2,5 Gbits/s/d (Giga bits por segundo por direção) conforme citado no início desta seção, porém com os avanços das tecnologias em silício, é esperado um aumento para 10 Gbits/s/d . Também é adicionado um conjunto de caracteres em uma configuração que indica o início e o fim do pacote, conforme mostra Figura 1.8.

O *link* PCIe mais simples é composto por dois pares diferenciais (1 *lane*) de baixa tensão. Um par é utilizado para transmissão e outro para recepção de dados. A largura de banda de um *link* PCIe é dimensionada adicionando pares de sinais

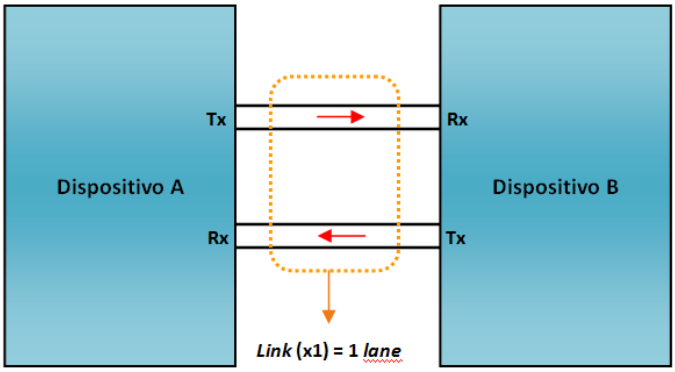


Figura 1.9: *Link* PCIe com 1 lane.

de maneira a formar múltiplos *lanes*. A camada física suporta *links* com 1, 2, 4, 8, 12, 16, ou 32 *lanes* de largura que podem ser representados da seguinte forma: (x1), (x2), (x4), (x8), (x12), (x16), (x32). As Figuras 1.9 e 1.10 ilustram um *link* PCIe (x1) e um *link* PCIe (x4), respectivamente.

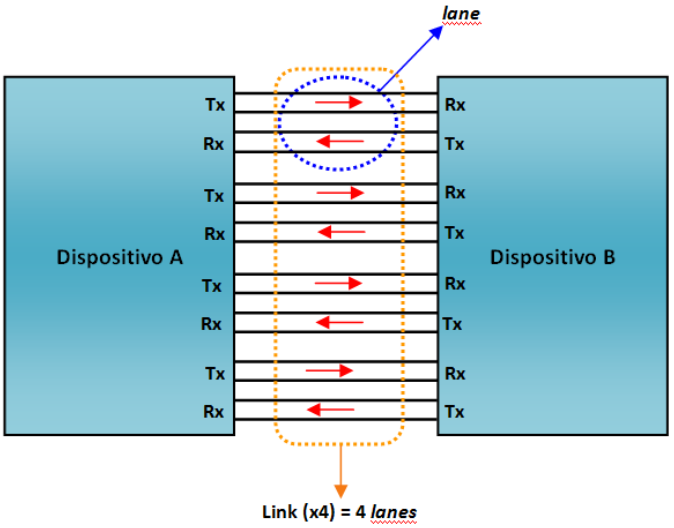


Figura 1.10: *Link* PCIe com 4 lanes.

Durante a transmissão a PL divide os dados em bytes através dos *lanes*. Essa arquitetura de distribuição dos bytes de dados é transparente para as outras camadas. Durante a inicialização, cada *link* PCIe se ajusta de maneira a buscar uma melhor configuração de velocidade de operação entre os dispositivos localizados nas extremidades do *link*. Nenhum *firmware* ou sistema operacional são envolvidos nesse

processo. No dispositivo receptor, a PL recebe os dados seriais do *link* físico PCIe, os transforma novamente em um pacote de dados e os repassa para a DLL receptora. O processo de decodificação dos dados é basicamente o inverso do que ocorreu no dispositivo transmissor.

A Figura 1.11 mostra as interfaces mecânicas (conectores) disponíveis na maioria dos PC's para *links* com 1, 4, 8 ou 16 *lanes*.

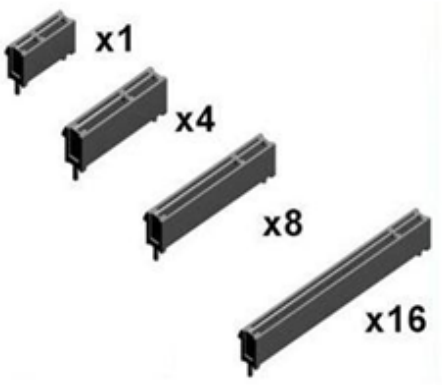


Figura 1.11: Conectores PCIe.

Em todos os formatos, os conectores são divididos em duas seções. A primeira contém os contatos de alimentação elétrica e é igual em todos os conectores, enquanto a segunda inclui os contatos de dados, que aumentam em número de acordo com a quantidade de *lanes*. Essa organização foi desenvolvida para que exista compatibilidade entre placas com um número menor de *lanes* e os conectores maiores. Ao conectar uma placa com apenas 1 *lane* em um conectores (x16), por exemplo, os *lanes* não usados são desativados pelo protocolo.

1.3 Protocolos de comunicação entre dispositivos

1.3.1 JTAG

Em 1985, foi criado um grupo internacional chamado JTAG (*Joint Test Action Group*), com o objetivo de criar um padrão para testar circuitos eletrônicos, pois os pontos de testes se tornavam cada vez menos acessíveis devido ao aumento da integração circuitos, a utilização de placas multicamadas e a uma miniaturização de componentes dificultando cada vez mais o acesso das pontas de prova de instrumentos de medição. Em 1990, o *Joint Test Action Group* publicou o padrão IEEE Std 1149.11990: IEEE *Standard Test Access Port and Boundary Scan Architecture* [18], que definiu uma *interface* de testes para módulos eletrônicos e circuitos integrados, também conhecida como JTAG. O padrão utiliza componentes de *hardware* incorporados aos circuitos integrados, permitindo o seu controle por meio de *software*. Mais recentemente o grupo JTAG desenvolveu o padrão IEEE Std 15322001: IEEE *Standard for System Configuration of Programmable Devices* [19] que padroniza a configuração de dispositivos lógicos programáveis através da *interface* JTAG.

O objetivo da *interface* JTAG é realizar testes no dispositivo que a contém. Para isso, utiliza quatro pinos e um conjunto de registradores de controle e de dados, como mostra a Figura 1.12. Dentre os registradores existentes, destacam-se o registrador de instruções IR (*Instruction Register*), responsável por selecionar o tipo de teste que será executado pela *interface*, e o registrador BSR (*Boundary Scan Register*) que possibilita a monitoração dos níveis e da lógica interna durante a operação do dispositivo, devido a cada pino do dispositivo estar conectado a um elemento BSR.

Os pinos utilizados são:

- TCK: Sinal de *clock* (*Test clock*).
- TMS: Seleção do modo de teste (*Test mode select*).
- TDI: Entrada de dados (*Test data in*).
- TDO: Saída de dados (*Test data out*).

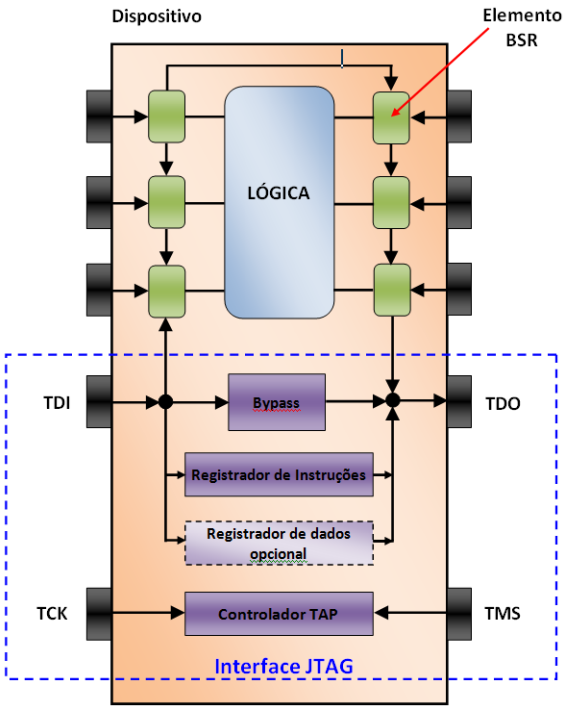


Figura 1.12: *Interface* JTAG.

O sinal TMS controla uma máquina de estados chamada controlador TAP (*Test Access Port*) responsável pela execução dos testes. A Figura 1.13 ilustra a diagrama de transição de estados do controlador TAP.

O gerenciamento e controle dos testes nos dispositivos que possuem a interface JTAG incorporada são realizados pelo controlador JTAG *master*. Esse controlador, normalmente, é um processador, ou microcontrolador, que envia sinais aos quatro pinos da *interface* JTAG do dispositivo para configurá-lo e realizar os testes. O controlador de JTAG é utilizado para testar as conexões internas do dispositivo e as conexões externas com outros dispositivos no módulo eletrônico. Ele também pode ser utilizado para inserir dados de configuração, tal como o *firmware* de um FPGA, no dispositivo lógico programável através da *interface*.

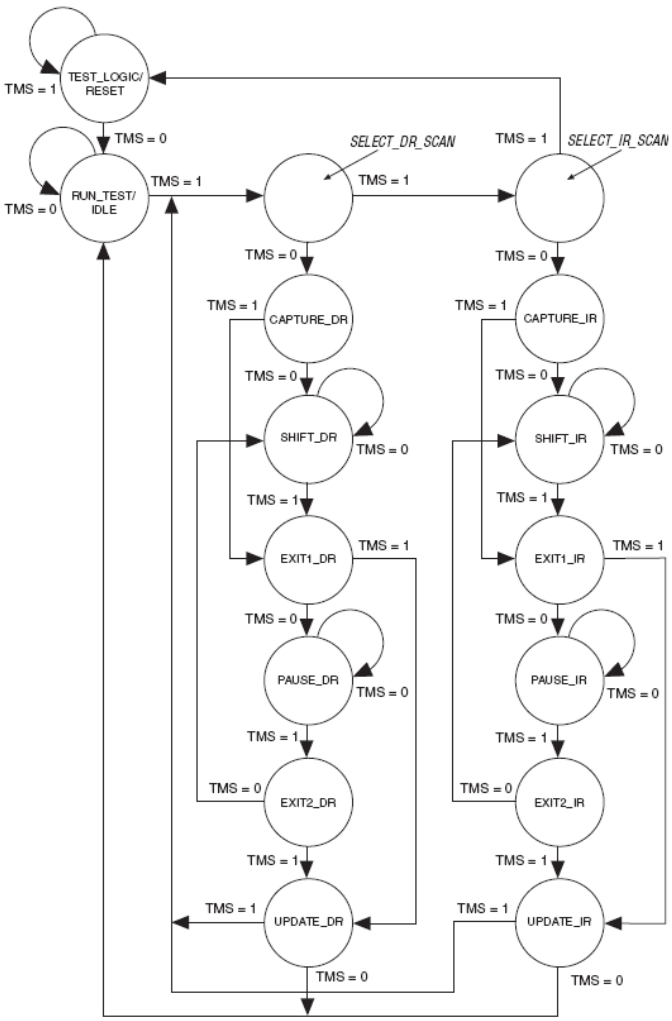


Figura 1.13: Máquina de estados do controlador JTAG.

1.3.2 SPI

A tecnologia de comunicação SPI (*Serial Peripheral Interface*) [20] foi desenvolvida pela Motorola para a linha de processadores da família MC68K. O SPI é um protocolo síncrono composto pelos seguintes sinais:

- SCLK: Sinal de *clock*.
- SDIO: Entrada ou saída de dados seriais.
- CSB: Seleção de dispositivo.

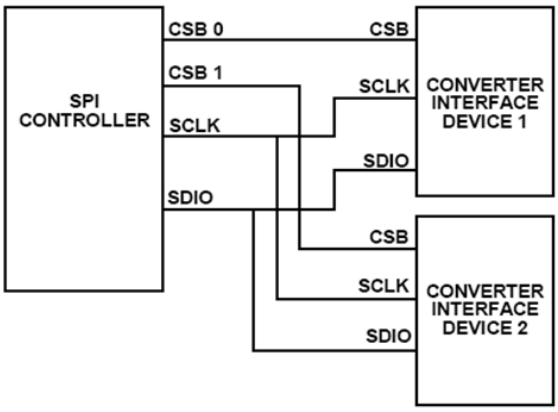


Figura 1.14: Exemplo de controlador SPI com dois dispositivos *slaves*.

O *clock* é utilizado para sincronizar o processo de leitura ou escrita na *interface*. Os dados de entrada são registrados na borda de subida do *clock*, e os dados de saída são registrados na borda de descida. O que determina se o sinal SDIO será um sinal de entrada ou saída, é o nível lógico do 1º bit de configuração que é enviado no início da comunicação. Se a solicitação for de escrita, o nível lógico do 1º bit deve ser alto, pois caracteriza entrada de dados. Da mesma forma, se a solicitação for de leitura, o nível lógico desse bit deverá ser baixo, caracterizando uma saída de dados. Na Figura 1.15 é possível observar essa situação. O sinal CSB é responsável por habilitar o funcionamento de um dispositivo. Quando o CSB está em nível baixo, os sinais SCLK e SDIO são processados pelo dispositivo, e quando o CSB está em nível alto o dispositivo ignora o SCLK e o SDIO. A Figura 1.14 mostra um controlador SPI com duas saídas CSB.

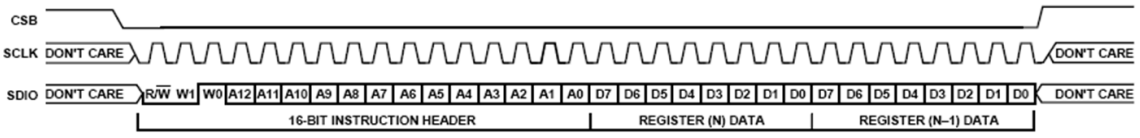


Figura 1.15: Diagrama de tempo do protocolo SPI.

Os dois primeiros bytes transmitidos são de configuração e endereçamento, onde o 1º bit define a leitura ou escrita, como dito anteriormente, os bits W1 e W0 determinam a quantidade de bytes de dados a serem transmitidos e os outros treze bits (A12 a A0) representam o endereço inicial do primeiro byte de dados. Na Figura 1.15 é possível visualizar essa representação.

Segundo a Tabela 1.1, a combinação dos bits W1 e W0 define a quantidade de bytes de dados que serão transmitidos.

W1 W0	**Qtd de bytes de dados transmitidos**
00	1
01	2
10	3
11	4 ou mais

Tabela 1.1: Codificação dos bits W1 e W0 do protocolo SPI.

No caso de quatro ou mais bytes de dados selecionados através de W1 e W0, quem definirá o término da transmissão de dados será o CSB, ou seja, enquanto CSB for igual a zero, o dispositivo continuará aceitando novos dados.

Além dos protocolos citados nesta seção, existem outros protocolos destinados a comunicação de dados entre dispositivos eletrônicos. A escolha do protocolo a ser utilizado vai depender das características de cada aplicação. Como outro exemplo, pode ser citado o protocolo I^2C (*Inter-Integrated Circuit*) [21], muito utilizado em sistemas de baixa velocidade.

2 Ferramentas de desenvolvimento

Todo o desenvolvimento do projeto foi realizado utilizando-se basicamente duas ferramentas comercialmente disponíveis:

- O kit de desenvolvimento Arria II GX [22] em conjunto com o *software* Quartus II [23].
- O programa Altium [24], utilizado para criação dos esquemáticos e do layout da placa de circuito impresso.

2.1 Kit de desenvolvimento Arria II GX

Este kit comercial, visualizado na Figura 2.1, representa uma plataforma de hardware para o desenvolvimento e prototipagem de projetos de baixa potência e alto desempenho. Essa ferramenta utiliza uma FPGA modelo Arria II GX que possui blocos de hardware dedicados ao desenvolvimento de projetos com protocolo PCI *Express*, permitindo maior desempenho do sistema. A escolha deste kit levou em consideração os custos envolvidos e as características funcionais similares ao FPGA Cyclone IV [25], utilizado no módulo MOPI.

A partir desse kit é possível projetar e verificar o funcionamento do protocolo PCIe com até 8 lanes. O kit também fornece periféricos e interfaces de memória para facilitar o desenvolvimento e testes de projetos. Duas portas de alta velocidade estão disponíveis para aplicações externas. Estas últimas foram utilizadas na realização das medidas experimentais como interface de entrada e saída, apresentadas

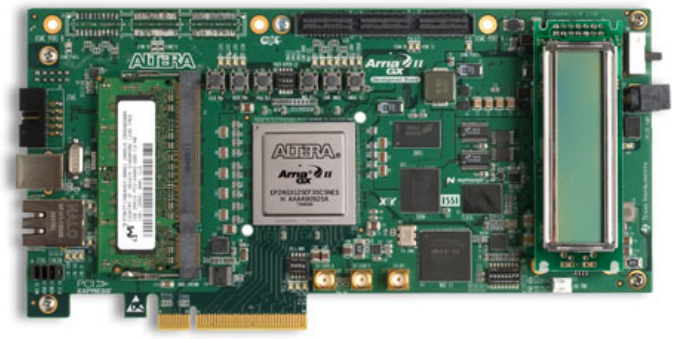


Figura 2.1: Kit de desenvolvimento Arria II GX.

no capítulo 4. O kit pode ser instalado no PC, diretamente no barramento PCIe da placa mãe.

2.2 A ferramenta Quartus II

O Quartus II foi utilizado em todas as fases do projeto. Com ele, foi possível desenvolver, em VHDL [26], componentes específicos para o projeto, utilizar componentes disponibilizados pelo fabricante, também conhecidos como *megafunctions*, realizar testes, e executar as etapas de *synthesis*, *placement* e *routing* do projeto lógico sintetizado no FPGA.

O programa possui aplicativos destinados ao desenvolvimento das diversas etapas de um projeto. Como exemplo, pode ser citado o editor de blocos, que proporciona um ambiente gráfico, no qual pode-se desenvolver todo o projeto inserindo blocos e símbolos primitivos, ou que representem uma lógica descrita, em VHDL, pelo projetista. Também é possível editar o projeto gráfico utilizando este aplicativo. Todas as informações do ambiente gráfico ficam armazenadas em um arquivo tipo bdf (*block design file*). A Figura 2.2 mostra o painel do Quartus II e, através de uma captura de tela, parte do projeto MOPI, desenvolvido em ambiente gráfico.

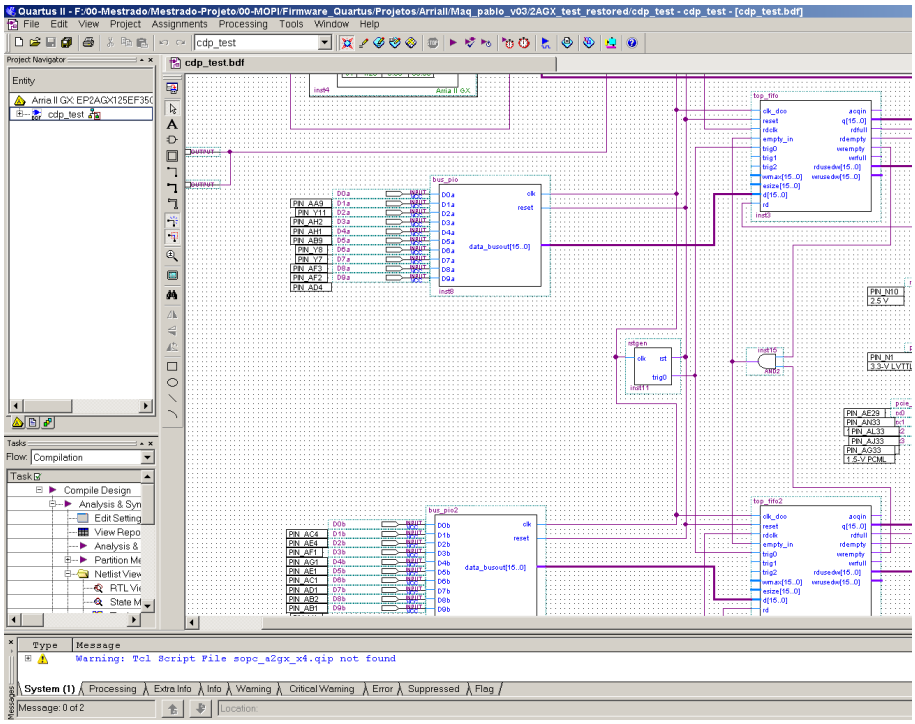


Figura 2.2: Parte do projeto MOPI, em ambiente gráfico, no Quartus II.

2.2.1 O SOPC *Builder*

O SOPC (*System on a programmable-chip*) *Builder* [27] é uma ferramenta de desenvolvimento incorporada ao Quartus II que interconecta, de maneira automatiza, os componentes do hardware. Essa ferramenta é indicada no desenvolvimento de sistemas que exigem interconexões mais complexas entre os diversos barramentos de um projeto. No Projeto MOPI optou-se por utilizar o SOPC para implementar com maior confiabilidade a integração entre o PCI *Express*, o processador NIOS II e uma memória RAM compartilhada. A Figura 2.3 mostra as interconexões e o ambiente de trabalho do SOPC. É possível observar os canais de compartilhamento, s1 e s2, da memória RAM compartilhada (Mem_Dual) e, a interconexão entre o PCI *Express* (pcie_compiler_0) e o NIOS II, através dos sinais *Control_Register_Access* e *data_master*, respectivamente.

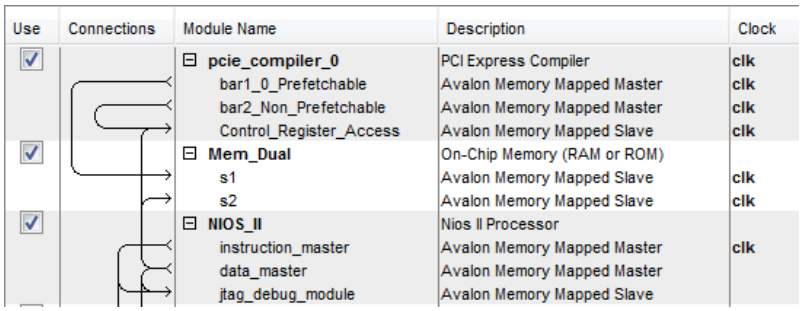


Figura 2.3: Integração entre os dispositivos no SOPC.

O compilador PCI *Express*

Esse compilador, disponibilizado pelo fabricante, permite integrar *megafunctions* [28] PCI *Express* em componentes sintetizados no FPGA. Dessa forma, é possível desenvolver componentes do tipo *root* ou *endpoint* com o protocolo PCI *Express* incorporado. No MOPI, o bloco PCI *Express* está configurado como *endpoint*.

As *megafunctions* PCI *Express* são configuráveis e implementadas com todas as características exigidas pela especificação, para a composição das camadas do protocolo. Uma visualização do Compilador PCI *Express* é apresentada na Figura 2.4.

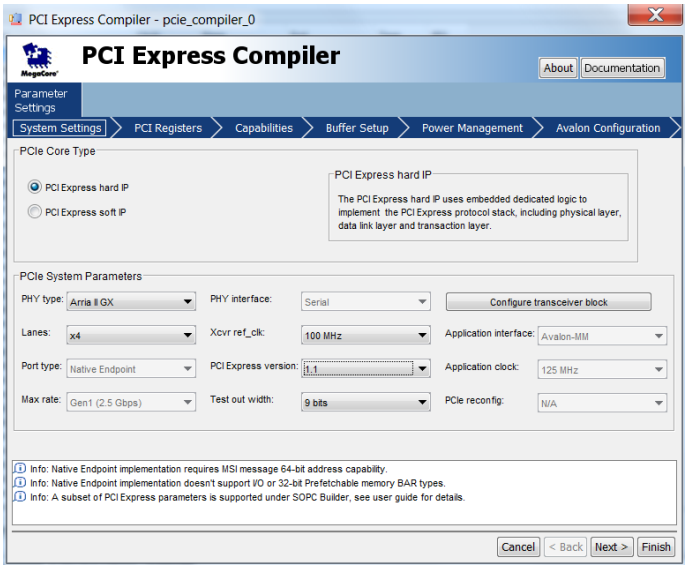


Figura 2.4: Compilador PCI *Express*.

O processador NIOS II

O sistema de processamento NIOS II [29], é sintetizado no FPGA e, funciona de forma equivalente a um microcontrolador. A diferença entre microcontroladores tradicionais e o NIOS II reside no fato, de que este último é configurável. Portanto, é possível adicionar, remover, ou editar recursos do sistema de maneira a atender as metas de desempenho e custo do projeto. O sistema NIOS II é chamado *soft*, pois o núcleo do processador, assim como os periféricos, não são estáticos (fixos) na pastilha de silício, podendo ser direcionados para qualquer família de FPGA's do mesmo fabricante. O NIOS II, assim como qualquer microcontrolador, pode ser programado em linguagem C. A ferramenta utilizada para programação e compilação do NIOS II chama-se Eclipse [30].

O Nios II é um processador RISC genérico com as seguintes características principais:

- Conjunto de instruções, dados e espaço de endereçamento de 32 bits;
- 32 registradores de propósito geral;
- 32 fontes de interrupção externa;
- Instruções simples de multiplicação e divisão em 32 bits;
- Instruções dedicadas para processamento de produtos de multiplicação de 64 bits e 128 bits;
- Instruções simples de deslocamento;
- Acesso a uma variedade de periféricos on-chip, interfaces para memória e periféricos off-chip;
- Módulo de depuração por hardware assistido, habilitando o processador a iniciar, parar, e executar passo a passo sob controle de um ambiente de desenvolvimento;

- Ambiente de desenvolvimento de software baseado na ferramenta GNUC/C^{++} e Eclipse IDE;
- Desempenho em torno de 250 DMIPS (*Dhrystone Millions of Instruction per Second*).

O *core* do processador inclui apenas os circuitos necessários para implementar a arquitetura do NIOS II. Na Figura 2.5 são apresentados os principais componentes internos do processador NIOS II. A arquitetura do NIOS II define as seguintes unidades funcionais visíveis ao usuário: registradores, ULA, interfaces para instruções lógicas personalizadas, controlador de exceção, controlador de interrupção, barramento de instruções, barramento de dados, memória cache de instruções e dados, interfaces de memória acopladas para instruções e dados e módulo de depuração JTAG.

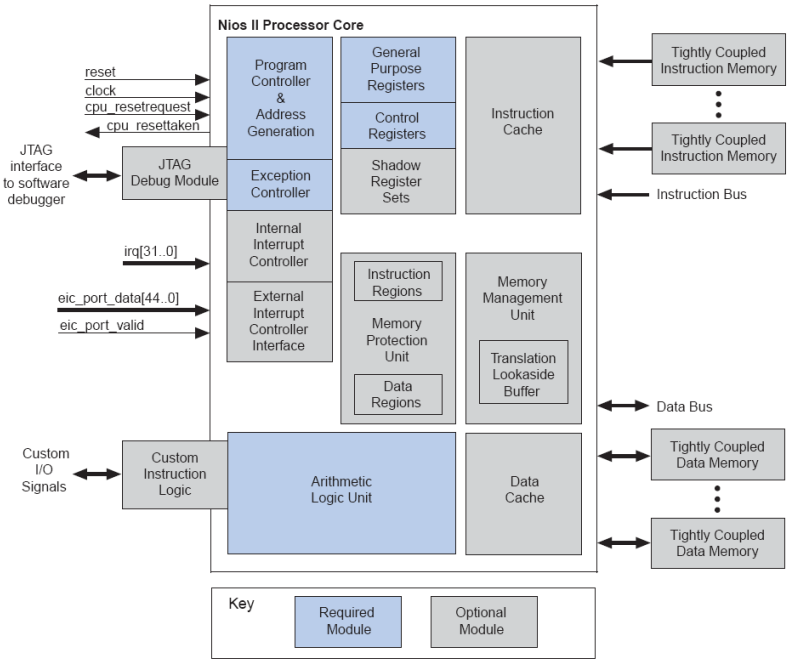


Figura 2.5: Estrutura interna do processador NIOS II.

Optou-se pela utilização do NIOS II nesse trabalho pela facilidade de implementação, em vista do tempo reduzido, e pelo aprendizado deste recurso flexível e interessante do FPGA.

A memória RAM compartilhada

A principal característica desse tipo de memória é permitir que dois dispositivos tenham acesso a leitura e a escrita de dados. Para que esse processo seja realizado de forma eficiente, cada canal da memória deve estar sincronizado com o *clock* do respectivo dispositivo. Devido a essa característica, a memória RAM foi utilizada como área comunicação e de transferência de dados, de modo bidirecional, entre o bloco PCI *Express* e o NIOS II.

Também chamado de memória dual, este componente é disponibilizado nas bibliotecas do Quartus II. Tanto o comprimento da palavra de dados, quanto o tamanho total da memória são configuráveis.

A memória FIFO compartilhada

A memória FIFO (*First-in First-out*) compartilhada utilizada no MOPI, da mesma forma que a memória RAM dual, está disponível nas bibliotecas do Quartus II. Na FIFO, o fluxo de dados é unidirecional. O dispositivo de escrita (fonte de dados da aplicação) não pode ler dados, assim como o dispositivo de leitura (NIOS II) não pode escrever na memória. Tanto o canal de leitura, quanto o de escrita possuem entradas de *clock* que devem ser conectadas aos respectivos dispositivos. A FIFO possui *flags* que indicam a condição de memória cheia ou vazia, permitindo o controle do fluxo de dados.

3 Módulo programável para instrumentação científica - MOPI

O MOPI foi desenvolvido para os seguintes propósitos: aquisição de dados, análise espectral, processamento digital de sinais, controle de experimentos, controle e monitoração de processos industriais, entre outros. Este módulo pode ser visto como um dispositivo de processamento que reúne as seguintes funcionalidades:

- Dois canais de conversão analógico-digital.
- Dois canais de conversão digital-analógico.
- Um gerador senoidal de alta frequência.
- Quatro canais digitais de entrada ou saída *single-ended*.
- Quatro canais digitais de entrada ou saída diferenciais.

O módulo é baseado em um dispositivo lógico programável do tipo FPGA e utiliza componentes eletrônicos disponíveis comercialmente. O MOPI e o seu diagrama em blocos são apresentados nas Figuras 3.1 e 3.2, respectivamente.

Dois modos de comunicação com o módulo são possíveis:

1. O modo PCIe, no qual o módulo é instalado no barramento PCIe de um PC, o qual fornece a alimentação necessária para seu funcionamento. O protocolo PCIe foi implementado com quatro *lanes* para comunicação de dados.

2. O modo USB, onde o módulo não é instalado no PC, mas se comunica com ele através de um dispositivo USB [31]. Neste caso, o módulo deve ser alimentado por uma fonte externa (5V @ 2,5A).

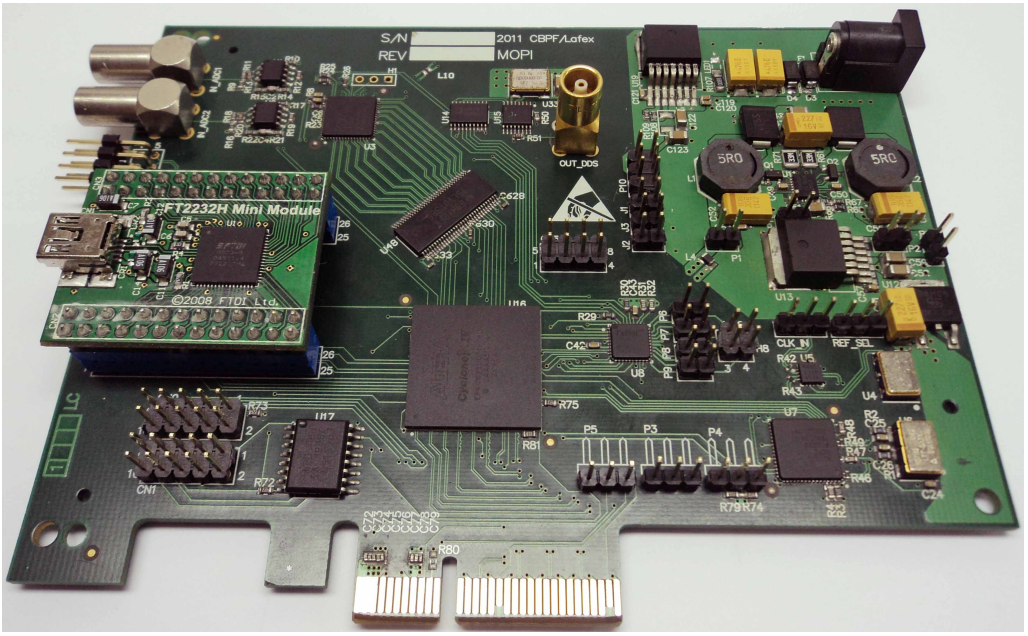


Figura 3.1: Módulo para instrumentação científica - MOPI.

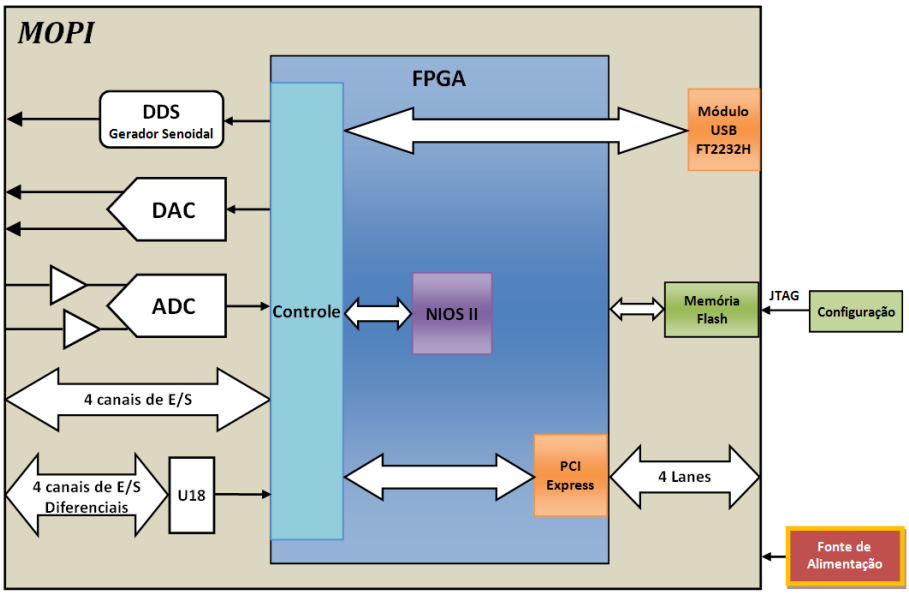


Figura 3.2: Diagrama em blocos do MOPI.

O MOPI possui um microprocessador sintetizado no FPGA chamado NIOS II. Sua função é estabelecer e gerenciar todo o funcionamento do módulo a partir das informações que chegam através dos barramentos PCIe ou USB. A comunicação de dados entre o NIOS II e os dispositivos de entrada e saída do módulo é realizada via protocolo SPI.

3.1 Desenvolvimento do *hardware*

3.1.1 Circuito conversor analógico-digital

A Figura 3.3 mostra o diagrama em blocos do circuito de aquisição de dados. O sinal analógico passa por amplificadores operacionais, com ganho 1 e saídas diferenciais, antes de ser convertido pelo ADC [32]. O principal motivo da conversão do sinal para o modo diferencial é o cancelamento dos transientes de corrente presentes na entrada do ADC, através da rejeição em modo comum [33]. Os transientes normalmente são gerados pela utilização da técnica *sample-and-hold* de chaveamento de capacitores durante a amostragem do sinal.

O conversor analógico-digital utilizado é o AD9627. Este dispositivo apresenta arquitetura multi-estágio *pipeline* [32] e seus dois canais de conversão são capazes de operar simultaneamente a uma taxa de 125 MSPS (*Mega Samples per Second*). O sinal analógico é convertido com uma resolução de 12 bits. A faixa dinâmica pode ser configurada através de um *jumper* no módulo, em 1 Vpp ou 2 Vpp, definindo, respectivamente, resoluções de conversão iguais a 0,24 mV ou 0,48 mV.

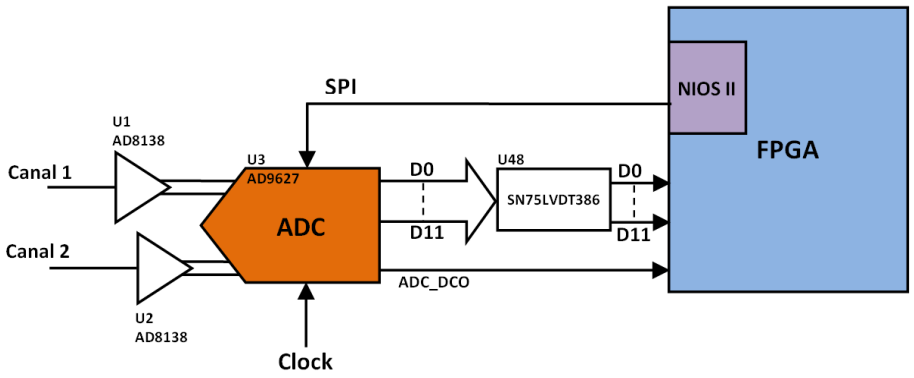


Figura 3.3: Diagrama em blocos do circuito de digitalização de sinais.

O ADC opera em modo *free running* [32], de maneira que o sinal é amostrado ininterruptamente na frequência de *clock* de entrada. O dado resultante da conversão fica disponível em sincronismo com um sinal de *clock* de 125 MHz (*adc_dco*), gerado pelo próprio ADC. Os dados do primeiro canal são síncronos com a borda de subida do *clock adc_dco*, e os dados do segundo canal são síncronos com as bordas de descida. Tal característica permite a transferência síncrona dos dados convertidos pelo ADC para o circuito integrado SN75LVDT386, que converte os 12 bits diferenciais para o modo *single-ended*, devido a indisponibilidade de canais diferenciais do FPGA. A impedância de entrada de ambos os canais de digitalização é fixada em 50 Ω através de resistores de precisão. O *clock* do ADC é diferencial, sendo fornecido por um cristal oscilador e um circuito integrado distribuidor.

3.1.2 Sintetizador digital de sinais

Este bloco do MOPI permite gerar um sinal senoidal com frequência configurável através do FPGA. Uma aplicação prevista, no CBPF, é a utilização do módulo

em um sistema de Espectroscopia por Impedância Elétrica, do qual este gerador será parte integrante. Para realizar esta tarefa é utilizado o DDS (*Direct Digital Synthesis*) AD9835 [34, 35, 36], que pode gerar frequências com precisão na ordem de 0,01 Hz.

A Figura 3.4 apresenta o diagrama em blocos do circuito. O DDS possui um circuito de *clock* exclusivo operando a 50 MHz. Esse valor garante a geração de ondas senoidais com frequências de até 25 MHz sem ocorrência de fenômenos de *aliasing* [37]. Algumas versões desse dispositivo podem gerar até 300 MHz. O circuito integrado ADG3300 é um adaptador bidirecional de nível lógico, pois o DDS opera com uma tensão de alimentação de 5 V, enquanto que os sinais enviados pelo FPGA estão no padrão LVTTL (*Low Voltage Transistor-Transistor Logic*), ou seja, 3,3 V.

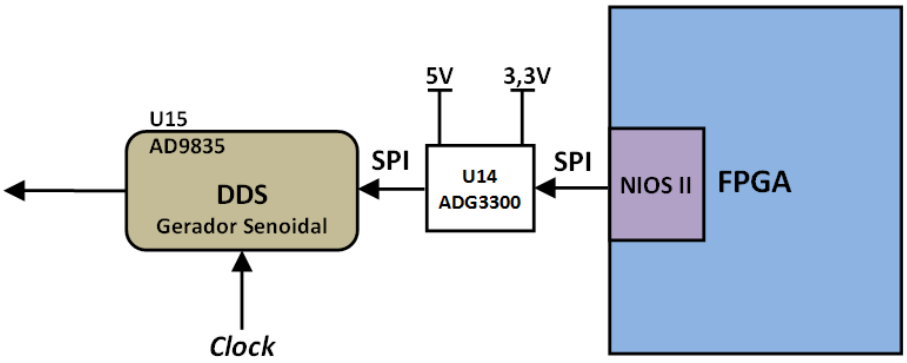


Figura 3.4: Diagrama em blocos do circuito do DDS.

O sinal senoidal será observado na saída do DDS a partir da configuração de registradores internos. Esta configuração se dá através de dados enviados pelo NIOS II, utilizando o protocolo SPI. Poucos componentes são utilizados no circuito do sintetizador, devido ao alto nível de integração do dispositivo. O DDS é constituído internamente por vários circuitos, tais como: oscilador controlado digitalmente, moduladores de fase e frequência e conversor digital-analógico. É importante ressaltar que o DDS também encontra utilização em outras aplicações, como por exemplo, nas técnicas de modulação FSK (*Frequency Shift-Keying*), PSK (*Phase Shift-Keying*) e suas variações.

3.1.3 Circuito conversor digital-analógico

O objetivo desta funcionalidade do MOPI é atuar como elemento de comando em uma cadeia de controle de processos, sejam estes experimentos científicos ou industriais. Como exemplo, o FPGA pode, através deste circuito, interagir no processo de maneira a compensar erros detectados por sensores conectados as entradas diferenciais do MOPI. O circuito integrado utilizado possui duas entradas de *clock*: CLKIN para a parte analógica e DCLKIN para a parte digital.

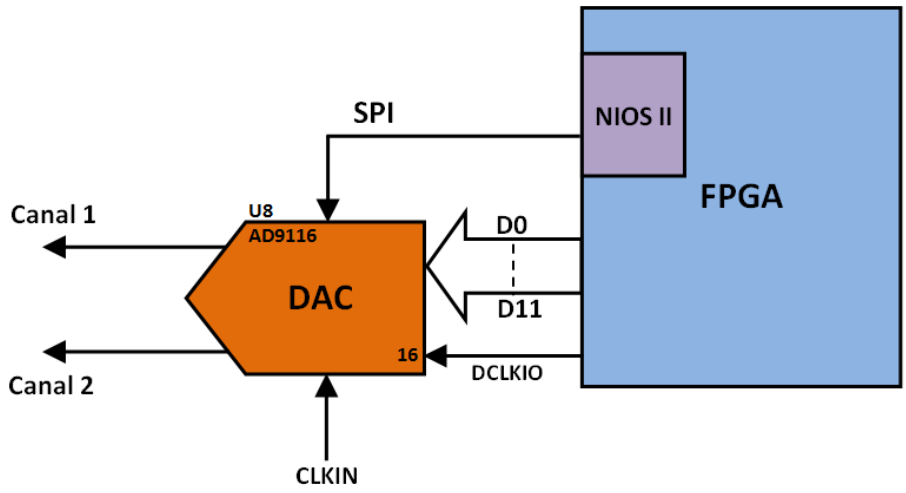


Figura 3.5: Diagrama em blocos do circuito do DAC.

A Figura 3.5 mostra o diagrama em blocos com as principais conexões entre o FPGA e o DAC (*Digital-to-analog converter*)AD9116 [38]. É possível observar que os pinos de entrada de dados do DAC estão conectados diretamente nos pinos de saída do FPGA. O sincronismo para aquisição de dados pelo DAC é feito através do sinal de *clock* (DCLKIO) fornecido pelo FPGA, de modo que os dados sejam amostrados de maneira síncrona pelo DAC. Os dados do canal 1 podem ser armazenados nas bordas de subida do *clock* e os dados do canal 2 na borda de descida, ou vice-versa. A frequência de amostragem é de 125 MHz. Por padrão, os dois canais de saída analógica são disponibilizados em corrente diferencial, porém dependendo da aplicação, podem ser convertidos em corrente *single-ended*, tensão diferencial ou tensão *single-ended*. Os dois últimos modos são obtidos utilizando um resistor de

carga. As correntes de fundo de escala de cada canal possuem uma faixa entre 4 mA a 20 mA, que pode ser ajustada através de resistores externos. Para o valor máximo de fundo de escala utiliza-se um resistor de 8 kΩ. Os dois canais de saída do DAC estão conectados diretamente as saídas físicas do MOPI.

3.1.4 Circuito distribuidor de *clock*

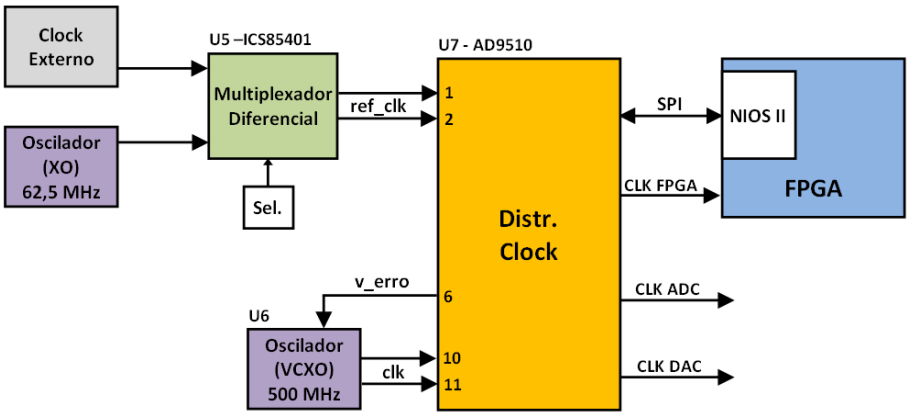


Figura 3.6: Diagrama em blocos do circuito de distribuição de *clock*.

Como o MOPI é um módulo multi-funcional, possuindo vários circuitos operando em altas frequências, é necessário que possua um controle de *clock* confiável e robusto. A Figura 3.6 ilustra o diagrama em blocos do circuito de geração e distribuição de *clock* do MOPI. O circuito integrado AD9510 é o principal componente desse circuito, sendo escolhido por reunir características adequadas ao projeto e pelo seu bom desempenho em projetos anteriores desenvolvidos pelo LSD.

O AD9510 recebe, nos pinos 1 e 2, uma referência de *clock* (ref_clk) e, nos pinos 10 e 11, o sinal de *clock* (clk) de 500 MHz enviado pelo oscilador a cristal controlado por tensão (VCXO). Internamente, esses dois sinais são devidamente ajustados em divisores antes de serem injetados em um detector de fase e frequência. Este último faz a comparação entre ambos e gera uma tensão (v_erro) no pino 6, proporcional ao erro encontrado. A tensão *v_erro* é enviada ao oscilador VCXO para que a frequência seja corrigida. A Figura 3.7 ilustra esse processo.

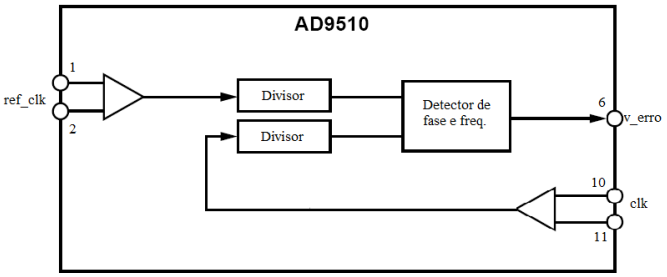


Figura 3.7: Diagrama interno do AD9510 responsável pelo ajuste da frequência.

O AD9510 possui oito saídas de *clock* com divisores de frequência independentes, o que torna possível o envio de diferentes valores de *clock* para os diversos dispositivos existentes no MOPI. A configuração dos registradores do distribuidor é realizada pelo NIOS II através do protocolo SPI. A referência de *clock* é enviada para o AD9510 pelo circuito integrado ICS85401. Tal dispositivo é um multiplexador diferencial de 2 canais. Um canal recebe uma referência *clock* de 62,5 MHz do oscilador a cristal (XO), enquanto que o outro canal está disponível para receber uma referência de *clock* externa. A seleção dos canais é realizada através de um *jumper* de configuração disponível no módulo.

3.1.5 Projeto da placa de circuito impresso

O projeto de layout da placa de circuito impresso (PCI) do MOPI foi finalizado em meados do ano de 2011. Com exceção de alguns ajustes realizados no CBPF, todo o layout foi desenvolvido por uma empresa especializada nessa área. Uma visão da PCI produzida a partir do projeto é mostrada na Figura 3.8. A PCI apresenta quatro camadas elétricas, sendo duas internas para os planos de GND e distribuição de alimentação e duas externas para trilhas de sinais e comunicação entre dispositivos.

Como ponto crítico do projeto de layout pode-se destacar o dimensionamento e o roteamento das trilhas diferencias dos circuitos do ADC e do barramento PCIe, devido a alta frequência dos sinais. As trilhas estão na configuração *microstrip* e possuem dimensões específicas para apresentarem uma impedância de 100 Ω [39].

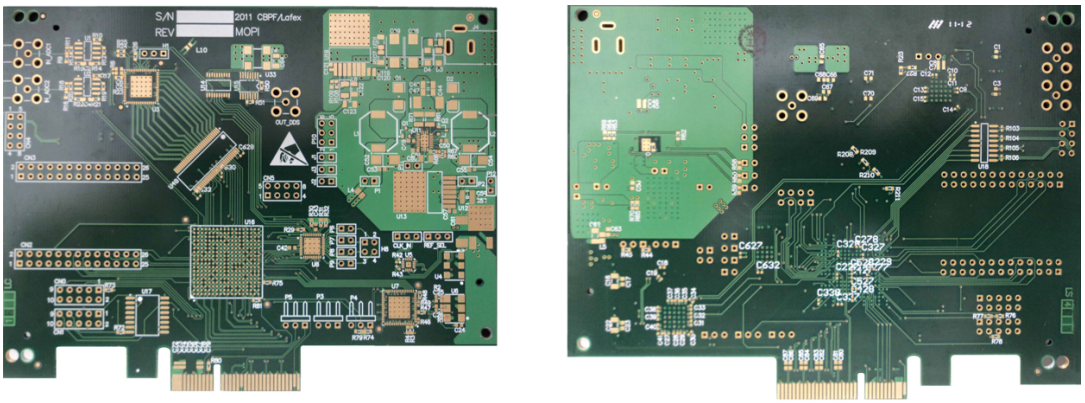


Figura 3.8: Placa de circuito impresso produzida para o MOPI.

As variáveis para o cálculo da impedância são: a largura das trilhas (W), a distância entre as trilhas (S) e a distância entre a camada das trilhas e a camada inferior (H), conforme ilustra a Figura 3.9.

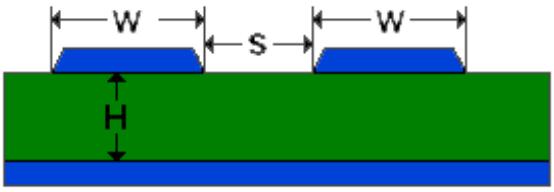


Figura 3.9: Dimensões para determinar a impedância das trilhas.

Para efetuar os cálculos do dimensionamento das trilhas com uma melhor precisão, optou-se por utilizar uma ferramenta dedicada a essa tarefa [40].

Outro aspecto relevante do projeto foi a fonte de alimentação, devido à utilização de reguladores de tensão com potência relativamente elevada. Para minimizar os efeitos térmicos foi embutido na própria placa, sob a região da fonte, um plano de dissipação de potência para esses componentes, como mostra a Figura 3.10.

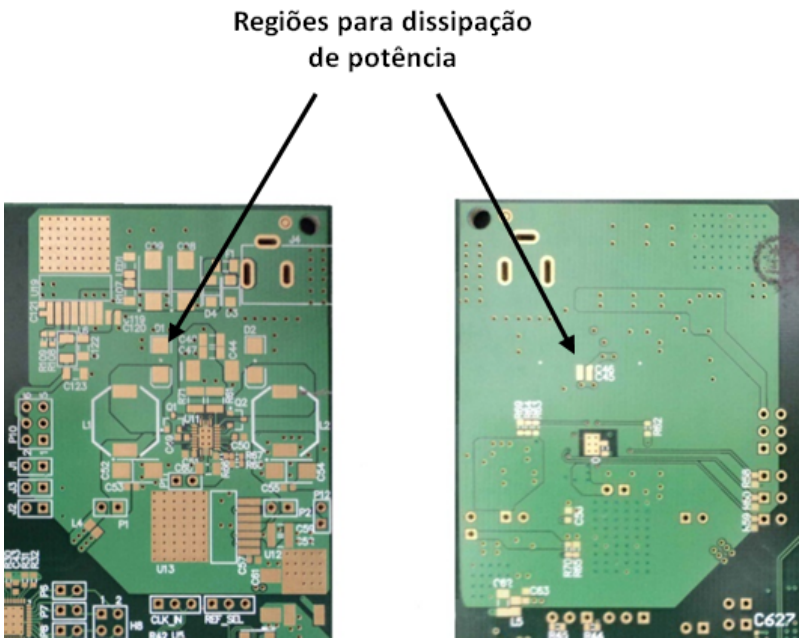


Figura 3.10: Regiões destinadas à dissipação de potência.

3.2 *Firmware* de controle e aquisição de dados

O *firmware* possui componentes, desenvolvidos em VHDL e exclusivos ao projeto, trabalhando em conjunto com alguns componentes disponibilizados no Quartus II, pelo fabricante. Estes últimos, chamados *megafunctions*, foram configuradas de acordo com as especificidades do projeto.

O programa desenvolvido para gerenciar as funcionalidades do MOPI, como aquisição de dados, acionamento do DDS e do TDC se encontra no Anexo A.

3.2.1 Aquisição de dados

O processamento dos dados convertidos pelos ADC é realizado pelo bloco *top_fifo*, apresentado na Figura 3.11. O bloco *top_fifo* é capaz de processar apenas um canal do ADC, portanto, existem dois desses blocos sintetizados no FPGA. Um bloco é sincronizado com a borda de subida do sinal *adc_dco* e o outro com a borda de descida do mesmo sinal.

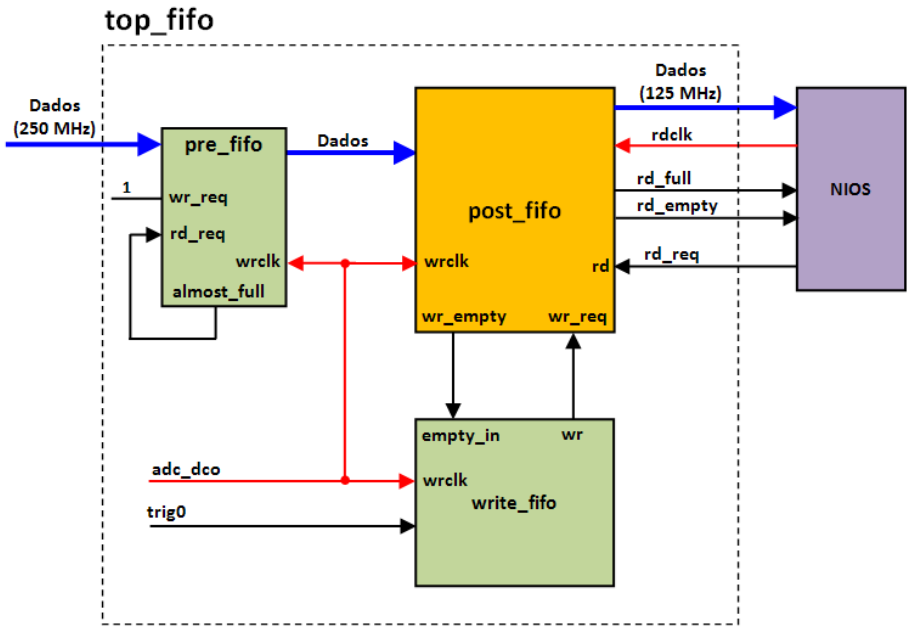


Figura 3.11: Bloco *top_fifo* sintetizado na FPGA.

Os dados convertidos pelo ADC são transferidos ininterruptamente para uma primeira camada de memória do tipo FIFO, identificada como *pre_fifo*, cujo sinal de permissão de escrita *wr_req* está permanentemente ativado (nível lógico 1). A leitura desta última é ativada pelo sinal *rd_req* a partir de sua própria *flag* *almost_full*, sinal indicativo de que a memória quase atingiu o seu limite de armazenamento (128 bytes).

Os dados lidos da memória podem ser armazenados na próxima camada de memória FIFO (*post_fifo*), desde que esta esteja vazia, obedecendo ao controle gerado pelo componente *write_fifo*. Quando este último componente recebe um sinal de disparo (*trig0*), a escrita na *post_fifo* é habilitada através do sinal *wr_req* por um período dado pelo produto Namt x Tdco, onde Namt é o número de amostras que constituem uma forma de onda e Tdco é o período do sinal *adc_dco*. O tamanho máximo, em bytes, da *post_fifo* é 65.536. Isso significa que, para um Tdco = 8ns, o período máximo de uma forma de onda é 524,288 μs. O parâmetro Namt pode ser configurado, através de um registrador, em potências de 2^n ($4 < n < 16$).

Para que não ocorram erros nos dados armazenados na memória *post_fifo* por informações espúrias armazenada anteriormente, optou-se por permitir que a escrita nessa memória seja possível somente quando esta estiver completamente vazia. A memória *post_fifo* informa sobre essa condição através do sinal *wr_empty* conectado à entrada *empty_in* do controlador *write_fifo*.

Pode-se concluir que a *pre_fifo* permite que um determinado número de amostras convertidas seja armazenado, antes de um sinal de disparo. Quando um disparo ocorre, habilita-se a escrita na *post_fifo* por tempo suficiente para que as amostras na FIFO anterior sejam transferidas, assim como o restante das amostras que compõem uma forma de onda.

Quando a memória está cheia, o NIOS II é informado através do sinal *rd_full*, e pode fazer a leitura dos dados escritos na *post_fifo*. Ao receber um comando de leitura, o NIOS II ativa o sinal *rd_req* para habilitar a memória e começa a fazer a leitura de dados em sincronismo com o *clock* de leitura (*rdclk*) gerado pelo próprio NIOS II. Quando a memória retorna à condição de vazia, o NIOS II é informado através do sinal *rd_empty* e uma nova aquisição de dados pode ser realizada.

3.2.2 Circuito para medida de tempo entre pulsos

O TDC [41, 42] implementado no FPGA utiliza as entradas *single-ended* do MOPI. Os pulsos elétricos capturados são injetados no componente *tpulse* através da entrada *trig_in*, conforme mostra a Figura 3.12.

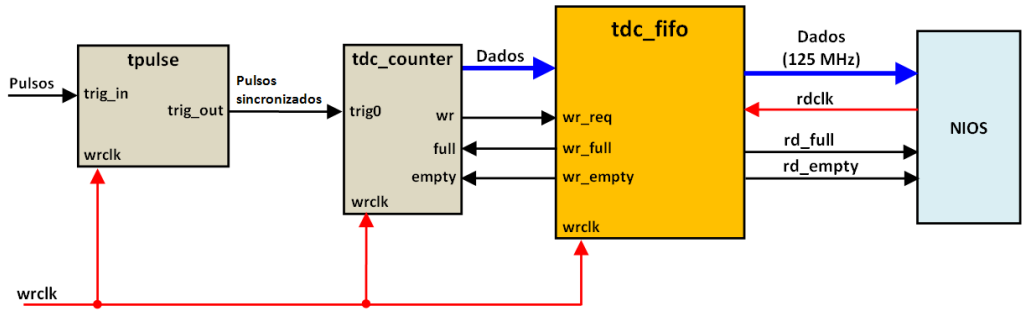


Figura 3.12: TDC sintetizado no FPGA.

Esse componente formata e sincroniza os pulsos de entrada com o sinal de clock, de maneira que na saída *trig_out* os pulsos são apresentados com um período de 8 ns e em fase com o *clock*. Em seguida, os pulsos são aplicados no componente *tdc_counter* para que seja determinado o intervalo de tempo eles. O tempo é determinado de maneira indireta, ou seja, o *tdc_counter* faz a contagem dos ciclos do *clock* de escrita (*wrclk*) que acontecem entre os pulsos de entrada. O *tdc_counter* também faz o controle do envio de dados para uma memória FIFO (*tdc_fifo*). Tal controle é realizado através dos sinais *wr_empty* e *wr_full* enviados pela *tdc_fifo* para o *tdc_counter*, indicando sua condição de vazia ou cheia, respectivamente. Se a memória estiver indicando que está cheia, o *tdc_counter* continua contando, mas não escreve na memória, pois mantém a saída *wr* desativada. Nessa condição o *tdc_counter* fica aguardando que o NIOS II faça a leitura das 65.536 posições da *tdc_fifo*. Quando a memória está totalmente vazia, o *tdc_counter* volta a escrever na memória e o NIOS II fica aguardando que esta última retorne à condição de cheia novamente. O NIOS II faz a leitura, baseado na informação dos sinais *rd_empty* ou *rd_full*.

Para o projeto Neutrinos Angra foi desenvolvido um componente chamado *coinc*, utilizado na medida de coincidência temporal entre pulsos. Para tanto, foi sintetizada uma porta lógica AND com quatro entradas. Quando este componente é utilizado, sua saída (*trigger*)é conectada à entrada *trig_in* do componente *tpulse*. A Figura 3.13 apresenta este componente.

Figura 3.13: Componente utilizado nas medidas de coincidência de pulsos.

3.2.3 Transferência de dados para o computador

A transferência de dados, assim como a troca de informações de controle entre o MOPI e o PC é realizada pelo bloco *control*, desenvolvido em ambiente SOPC e sintetizado no FPGA. Essa comunicação pode ser dividida em duas etapas: a primeira é a interconexão do PC com o bloco *control* através do *driver* PCIe-Linux, desenvolvido no CBPF [43].

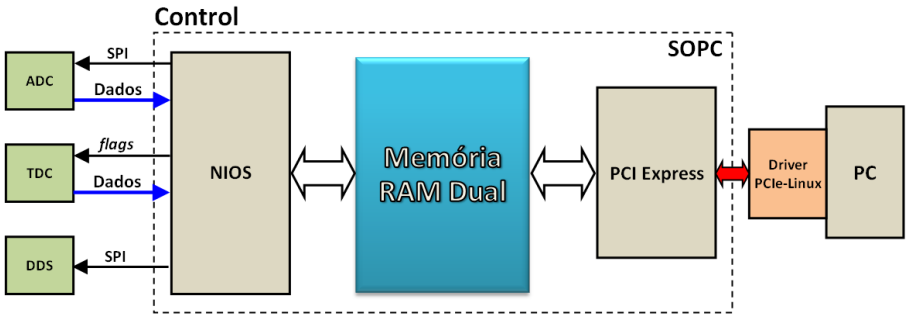


Figura 3.14: Interconexão entre o bloco *control* e o barramento PCI Express do computador.

Este *driver* estabelece a comunicação entre o barramento PCIe do PC com a *megafunction* PCIe do bloco *control*. A segunda etapa é o compartilhamento de uma memória RAM dual entre o NIOS II e a *Megafunction* PCIe, de modo a estabelecer uma comunicação bidirecional dentro do bloco *control*.

A memória dual está dividida em duas regiões[1], de forma que a primeira está reservada para os registradores de instrução, enquanto que a segunda para o armazenamento de dados. A primeira região de memória encontra-se subdividida da seguinte forma:

- Área 1: O PCIe escreve as instruções e o NIOS II lê.
- Área 2: O NIOS II escreve as instruções e o PCIe lê.
- Área 3: Reservada a aplicações futuras.

[1]Os endereços de início e fim das regiões de memória estão representados na Figura 3.15 na notação decimal.

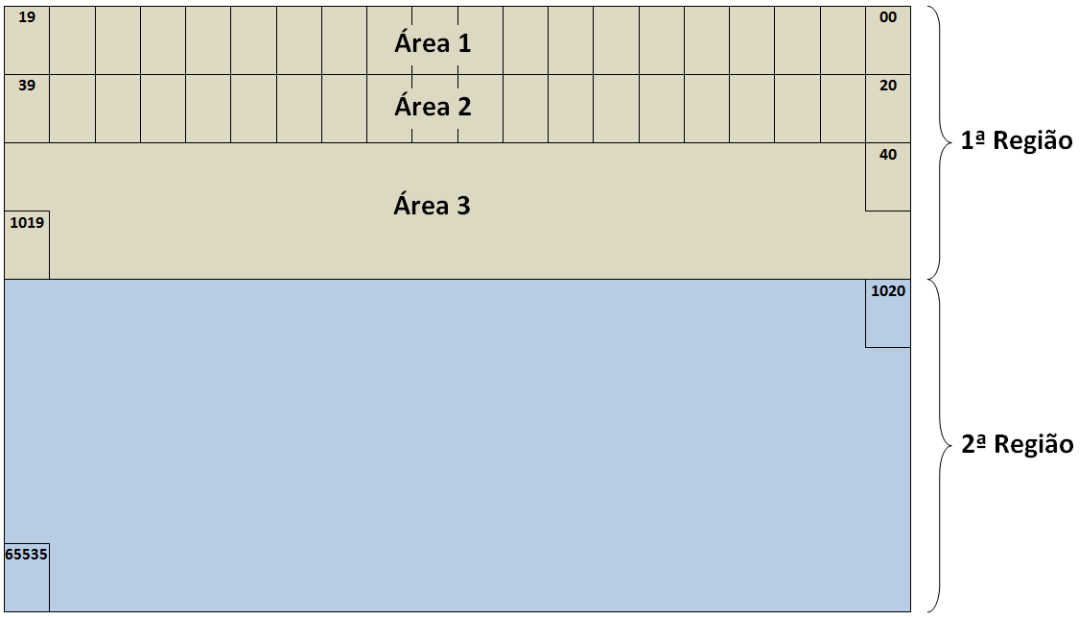


Figura 3.15: Mapa da memória RAM dual.

A memória dual possui 64 kB, dos quais os primeiros 40 bytes estão reservados para as áreas 1 e 2. A área 3 ocupa 980 bytes, e o restante da memória é destinada a região de transferência de dados. O mapa da memória dual pode ser visualizado na Figura 3.15.

Cada *frame* de dados está dividido em 4 palavras com 4 bytes, de tal forma que cada palavra possui uma informação específica dentro do protocolo interno. As Tabelas 3.1 e 3.2 mostram como é organizado esse protocolo e a codificação das palavras dentro dos *frames* enviados pelo PCIe e pelo NIOS II, respectivamente.

***Frame* enviado pelo PCIe para o NIOS II**		
Endereço na mem dual	Função da palavra	Código/Informação
00	*status*	AA(*frame* enviado)
04	descrição do comando	1,2,3,4,5,...
08	bytes enviados	n
12	*checksum*	somatório

Tabela 3.1: Descrição do *frame* enviado pelo PCIe para o NIOS II.

***Frame* enviado do NIOS II para o PCIe**		
Endereço na mem dual	Função da palavra	Código/Informação
20	*status*	AA(pronto para receber*frame*)
24	descrição do comando	Sucesso ou falha (AA ou CC)
28	bytes enviados	n
32	*checksum*	somatório

Tabela 3.2: Descrição do *frame* enviado pelo NIOS II para o PCIe.

Para que o PCIe inicie o envio de dados para o NIOS II, este deve informar que está pronto, escrevendo o código AA no endereço 20 da memória dual (área 2). Após isso, e considerando como exemplo de *frame* enviado, a sequência *1F 02 01 AA*, pode-se descrever que: a primeira palavra informa, através do código AA, que um *frame* foi enviado. A segunda palavra solicita a execução do comando 1, a terceira indica a quantidade de bytes (enviados ou recebidos) e a quarta palavra contém o valor do somatório dos dados.

Como exemplo de *frame* de resposta do NIOS II ao PCIe, pode ser considerada a sequência *1F 02 AA AA*, na qual a primeira palavra indica que o NIOS II está pronto para receber outro *frame*, a segunda palavra informa que houve sucesso (AA) no recebimento do *frame* anterior, a terceira mostra a quantidade de bytes (recebidos ou enviados) e a quarta palavra contém o valor do somatório dos dados. Esse é um exemplo em que não ocorreram erros na transmissão ou recepção dos dados. Supondo que ocorresse um erro, a segunda palavra do *frame* de resposta do NIOS II indicaria o código CC.

A implementação dessa comunicação necessita de três programas. O primeiro é o *driver* PCIe desenvolvido para o sistema operacional Linux. Acessando o *driver*, tem-se um conjunto de funções que implementam o protocolo acima (ver Anexo B). Finalmente, o programa implementado no NIOS II encontra-se no Apêndice B.

4 Medidas experimentais

Neste capítulo são apresentados os resultados das medidas realizadas neste trabalho. A plataforma de hardware utilizada foi um kit comercial equipado com uma FPGA Arria II GX, uma vez que não houve tempo hábil para testes completos com o módulo projetado (MOPI). Porém, é importante ressaltar que o *firmware* e o *software* utilizados no kit comercial são projetados para serem compatíveis com o MOPI. O circuito de geração de sinal senoidal (DDS) utilizado também faz parte do circuito do MOPI.

Na primeira seção deste capítulo são apresentadas medidas realizadas para determinar o espectro e a taxa de ocorrência de eventos aleatórios decorrentes de corrente de escuro de tubos fotomultiplicadores. Também foram determinadas as taxas de ocorrência deste tipo de evento para dois e quatro tubos em coincidência. Para a realização dessas medidas foram utilizados, como instrumentação auxiliar, o módulo NDAQ [44] (que possui o mesmo circuito digitalizar do MOPI) e quatro pré-amplificadores do laboratório de detecção.

A segunda seção apresenta medidas de Espectroscopia por Impedância Elétrica. Nesta aplicação, o objetivo é determinar uma curva de impedância em função da freqüência de um sinal aplicado, utilizando filtros RC [45]. É realizada uma varredura automatizada de frequências, geradas pelo DDS, e determinada a impedância de um dispositivo desconhecido. O circuito do DDS foi montado em um *protoboard*.

A Tabela 4.1 relaciona toda instrumentação utilizada nas calibrações e medidas ao longo desse capítulo.

Instrumento	Modelo	Fabricante
Fonte HV	1458	Universal Voltronics
Fonte DC	PS280	Tektronix
Fonte DC	E3648A	Agilent
Osciloscópio	DSO-X 3034A	Agilent
NDAQ	SN11	CBPF/LSD
Pré-amplificadores	—	CBPF/LSD

Tabela 4.1: Instrumentos utilizados nas calibrações.

4.1 Caracterização dos tubos fotomultiplicadores

O detector principal do Projeto Neutrinos Angra possui 40 tubos fotomultiplicadores, também chamados de PMT's. Tais dispositivos funcionam a vácuo e são capazes de converter luz em corrente elétrica. Para que essa conversão ocorra, a superfície de detecção (janela) do PMT é revestida internamente com uma fina camada de um material denominado fotocatodo, que tem a função de liberar elétrons quando interage com um fóton. Quando um fóton atravessa a janela da PMT, há uma probabilidade de que seja liberado um elétron após a interação.

Após o fotocatodo são colocados dinodos revestidos de material metálico. O primeiro fotoelétron emitido pelo fotocatodo é acelerado contra o primeiro dinodo devido a uma diferença de potencial, de tal forma que, na colisão são arrancados outros elétrons do dinodo. Entre o primeiro dinodo e os seguintes também existe uma diferença de potencial, que acelera agora, um grupo de elétrons gerando um processo de colisão semelhante, até o último dinodo. Cada um dos dinodos proporciona um pequeno ganho ao número de elétrons, porém o ganho total entre o primeiro e o último dinodo pode ser superior a 10^6. Os elétrons emitidos pelo último dinodo são coletados por um anodo, que disponibiliza uma corrente elétrica com a informação desejada.

A seguir serão descritas as calibrações da instrumentação utilizada e as medidas realizadas para caracterização dos PMT's.

4.1.1 Calibração do NDAQ

O módulo NDAQ possui 8 canais para digitalização de sinais analógicos utilizando o mesmo ADC do módulo MOPI. As características deste circuito integrado foram vistas na seção 3.1.1. O estágio de entrada analógica de cada canal é composto por um amplificador operacional com saída diferencial e pelo ADC (AD9627), conforme ilustra a Figura 4.1. A calibração do NDAQ se faz necessária para que os erros desses dispositivos sejam compensados posteriormente no *software* de aquisição. A calibração foi realizada somente no canal utilizado nas medidas apresentadas ao longo desse capítulo.

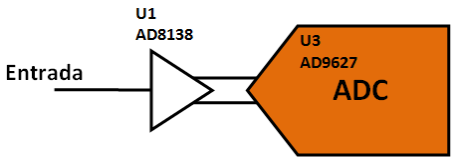


Figura 4.1: Estágio de entrada dos canais de aquisição de dados do NDAQ.

O aparato de calibração utilizado pode ser visualizado na Figura 4.2.

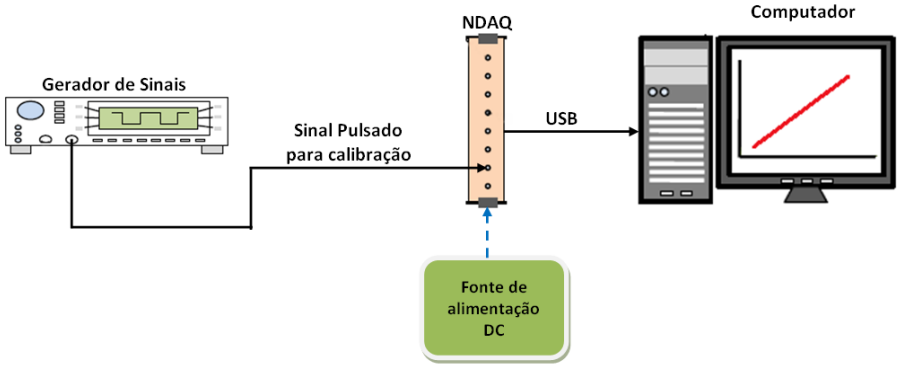


Figura 4.2: Aparato para calibração do NDAQ.

Metodologia

Utilizando um gerador de sinais, foi injetado no NDAQ um sinal pulsado com largura de 160 ns e 5 kHz de frequência. A tensão de referência do ADC foi ajustada em 2 V, o que permitiu variar o sinal do gerador de -1 V a 1 V, com intervalos

de 100 mV. Como utiliza-se efetivamente 10 bits de conversão do ADC, a resolução teórica é de aproximadamente 2 mV, resultado de 2 V/2^{10}. Para cada valor de amplitude injetado no NDAQ foram realizadas 10.168 medidas e calculadas as respectivas médias aritméticas e desvio padrão. Os dados são transferidos para o computador via interface USB para análise.

Resultados

O gráfico da Figura 4.3 apresenta os resultados obtidos da calibração do canal 2 do módulo. Os valores médios de tensão de entrada estão no eixo horizontal e as saídas digitais geradas pelo ADC no eixo vertical. Em vermelho, ajustou-se uma reta para verificar a linearidade da função de transferência. As barras de erro estão multiplicadas por 100 para melhor visualização.

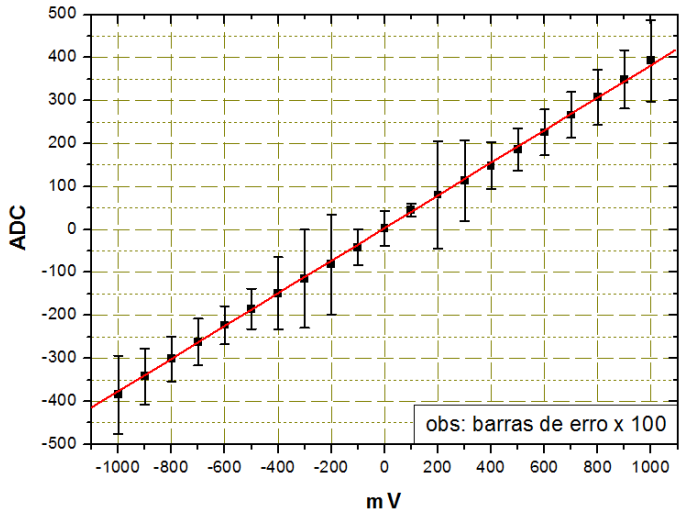


Figura 4.3: Função transferência de calibração do ADC do NDAQ.

É possível observar uma variação no erro encontrado nas medidas. Entre os principais fatores que contribuem para que isso ocorra, podem-se destacar:

- O erro aleatório produzido pelo ruído [46] e pela flutuação de cada dispositivo envolvido na medida.
- A precisão do gerador de funções.

- O erro sistemático gerado pelo estágio de entrada do NDAQ. Nesse caso, o amplificador operacional AD8138 contribui com erros de *off-set* e o ADC com erros de DNL (*Differential non-linearity*) [32].

4.1.2 Calibração dos pré-amplificadores

O objetivo dessa calibração é garantir que a carga injetada na entrada do pré-amplificador tenha correspondência linear e proporcional com a tensão de saída. A calibração dos pré-amplificadores é realizada aplicando-se pulsos com diferentes amplitudes, com um valor de carga conhecida, na entrada do circuito. A função transferência do pré-amplificador é obtida através da relação entre os valores de tensão observados na saída e os valores de carga que foram aplicados na entrada. A carga do sinal na entrada do pré-amplificador é dada pela equação 4.1, demonstrada a seguir.

$$i = \frac{dQ}{dt} \Rightarrow dQ = i.dt$$

$$Q = \int_{to}^{t1} i.dt$$

$$Q = \int_{to}^{t1} \frac{V(t)}{Z} dt \tag{4.1}$$

A Figura 4.4 mostra o circuito equivalente de entrada do pré-amplificador, e que pode ser representado pela equação 4.1. A influência da capacitância de entrada C é desprezível.

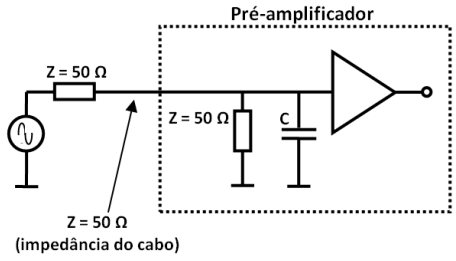


Figura 4.4: Circuito equivalente da entrada do pré-amplificador.

Como exemplo, pode-se considerar um pulso com 5 mV de amplitude e Δt ($t_1 - t_0$) igual a 50 ns. Supondo uma impedância de entrada igual a 50 Ω, obtém-se um sinal que corresponde a uma carga de 5 pC como mostra a equação 4.2.

$$Q = \frac{V}{Z}(t_1 - t_0) = \frac{5\ mV}{50\ \Omega}(50\ ns) = 5\ pC \tag{4.2}$$

A Figura 4.5 mostra o diagrama em blocos do aparato utilizado na calibração dos pré-amplificadores.

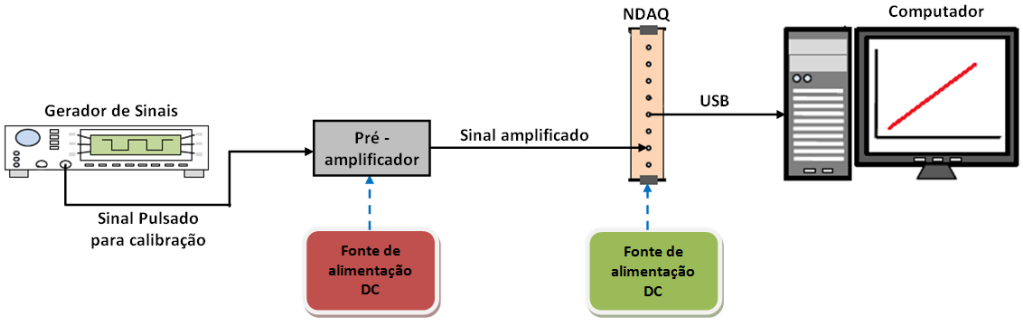


Figura 4.5: Aparato para calibração dos pré-amplificadores.

Metodologia

Através do gerador de sinais são injetados na entrada do pré-amplificador pulsos com amplitudes iguais a 5 mV, 10 mV, 15 mV e 20 mV, largura fixa de 50 ns e frequência 1 kHz. Como a impedância de entrada do pré-amplificador é uma resistência de 50 Ω, os valores de tensão injetados na entrada são proporcionais as cargas 5 pC, 10 pC, 15 pC e 20 pC, conforme mostra a equação 4.2.

O pré-amplificador é um circuito com três estágios, cada um com uma função bem definida, como mostra a Figura 4.6. O primeiro estágio é um circuito integrador, responsável por apresentar na saída um sinal elétrico com amplitude proporcional a carga injetada na entrada. O segundo estágio é basicamente um filtro RC, capaz de alterar a forma do sinal através da constante de tempo dada pelo produto R_4 x C_{14} e, portanto, tornar possível a amostragem correta de sinais na ordem de nano-segundos. No terceiro e último estágio o sinal é amplificado por um ganho constante, dado por R_5 e R_6.

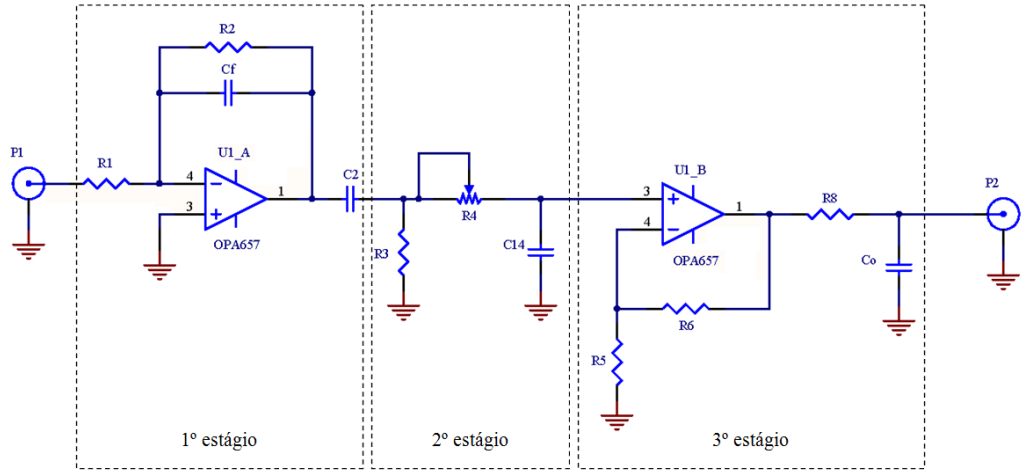


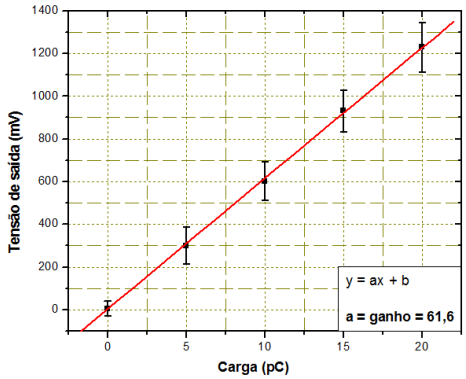
Figura 4.6: Esquemático do pré-amplificador.

Ao sair do pré-amplificador, o sinal é enviado ao módulo NDAQ, responsável por fazer a digitalização do sinal. Nesta calibração, o módulo é configurado para fazer 10.000 aquisições e calcular o valor médio. Os dados são enviados para o computador via interface USB. Foram calibrados quatro pré-amplificadores para serem utilizados na obtenção do espectro do fotoelétron único (*Single photon electron*) [6], e posteriormente nas medidas de pulsos de corrente de escuro das PMT's.

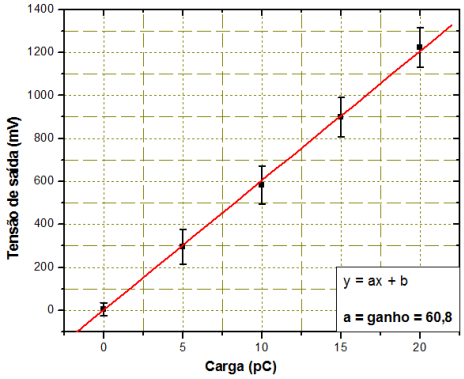
Resultados

A função transferência dos pré-amplificadores é apresentada na Figura 4.7. O coeficiente angular da inversa da função transferência fornece a relação entre carga e tensão. Dessa forma, é possível determinar a carga de saída para cada valor de tensão de entrada.

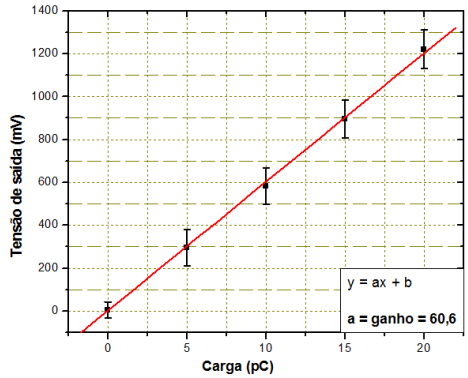
Como possíveis fontes de erro temos, a precisão do gerador de funções, o ruído e desvios de *off-set* do próprio pré-amplificador e erros de DNL do ADC do NDAQ. Para uma melhor visualização, as barras de erro foram multiplicadas por 10.



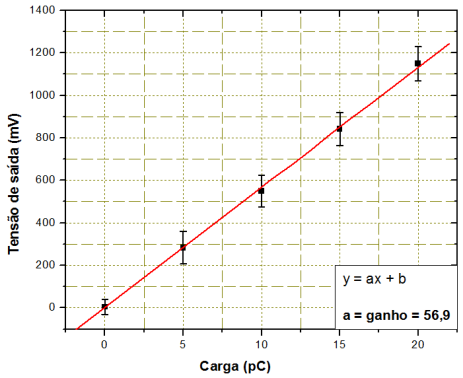
(a) Função Transferência do Pré-amplificador 1.



(b) Função Transferência do Pré-amplificador 5.



(c) Função Transferência do Pré-amplificador 6.



(d) Função Transferência do Pré-amplificador 7.

Figura 4.7: Gráficos da função transferência dos pré-amplificadores.

4.1.3 Verificação do espectro do fotoelétron único - SPE

O objetivo em determinar o espectro do SPE é verificar o comportamento de cada PMT do projeto Neutrinos Angra. O espectro do SPE permite, principalmente, verificar:

- Flutuações de linha de base da PMT e ruído eletrônico.
- Ganho do PMT.
- Resolução em energia.

A aquisição será realizada utilizando o módulo NDAQ. Para realizar corretamente esta medida e garantir que o espectro adquirido seja predominantemente do SPE, a quantidade de luz que deve atingir o fotocatodo é reduzida a nível de poucos fótons. A estatística de Poisson [47] é utilizada para descrever matematicamente a produção de fotoelétrons em pequenas quantidades no fotocatodo do PMT. Na equação 4.3, observa-se a probabilidade *P(n)*, para geração de n fotoelétrons quando o fotocatodo é atingido por fótons.

$$P(n) = \mu^n . \frac{e^{-\mu}}{n!} \tag{4.3}$$

μ: *média da distribuição.*

De acordo com os estudos [48, 49], pode-se considerar que a probabilidade de ocorrência de 2 fotoelétrons (2PE) varia entre 5 % e 10 % da probabilidade de 1 fotoelétron. Substituindo esses referidos valores percentuais na variável x da equação 4.4, são encontrados $\mu = 0,1$ e $\mu = 0,2$ para $x = 5\%$ e $x = 10\%$, respectivamente.

$$x.P(1) = P(2)$$

$$x.\mu^1 . \frac{e^{-\mu}}{1!} = \mu^2 . \frac{e^{-\mu}}{2!} \tag{4.4}$$

Calculando as probabilidades para zero, um e dois fotoelétrons com $\mu = 0,1$ obtém-se:

$$P(0) = 0,1^0 . \frac{e^{-0,1}}{0!} = 90,48\% \tag{4.5}$$

$$P(1) = 0,1^1 . \frac{e^{-0,1}}{1!} = 9,05\% \tag{4.6}$$

$$P(2) = 0,1^2 . \frac{e^{-0,1}}{2!} = 0,45\% \tag{4.7}$$

Refazendo os cálculos com $\mu = 0,2$ encontra-se:

$$P(0) = 0,2^0.\frac{e^{-0,2}}{0!} = 82\% \tag{4.8}$$

$$P(1) = 0,2^1.\frac{e^{-0,2}}{1!} = 16,4\% \tag{4.9}$$

$$P(2) = 0,2^2.\frac{e^{-0,2}}{2!} = 1,64\% \tag{4.10}$$

Para uma quantidade maior de fotoelétrons a probabilidade será insignificante.

Com base nos estudos [48, 49], mas utilizando a relação percentual de 5 % a 10 % para os valores máximos de contagem dos eventos de 2 fotoelétrons e 1 fotoelétron, ajusta-se o gerador.

A Figura 4.8 ilustra o aparato montado para a medida do espectro do SPE.

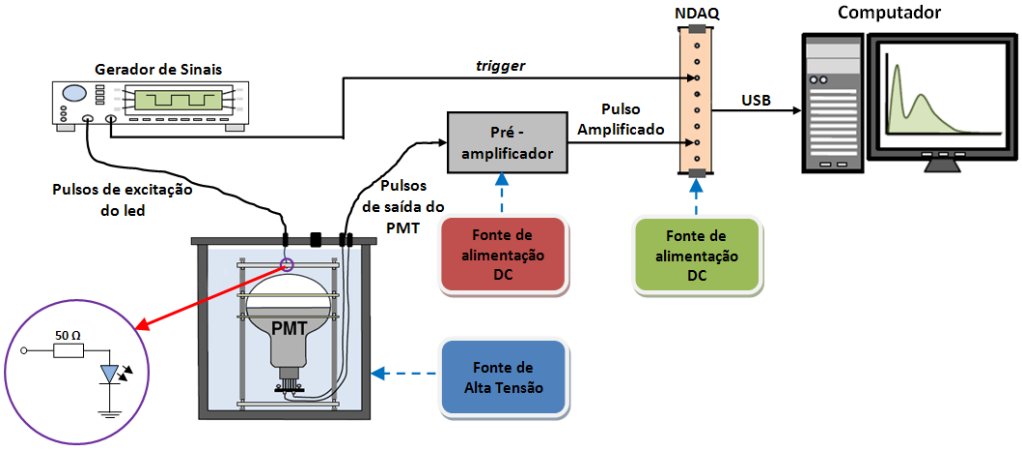


Figura 4.8: Aparato para obtenção do espectro do SPE.

Metodologia

O sinal adequado para excitar o LED foi obtido ajustando-se o gerador de sinais com um pulso de 8 ns de largura, 1 V de amplitude e frequência 1 kHz. O LED foi instalado em uma tampa escura confeccionada especialmente para medidas desse tipo, e colocado sobre o PMT, de modo a cobrir toda a superfície do fotocatodo, conforme mostra a Figura 4.9.

Figura 4.9: Dispositivo para instalação do LED.

O PMT e o LED foram colocados em uma câmara escura com capacidade para acondicionar quatro PMT's, como mostra a Figura 4.10.

Figura 4.10: Câmara escura para acondicionamento dos PMT's.

Os PMT's SD2845, SD2846, SD2782 e SD2785 foram energizados com as respectivas tensões de alimentação 1600 V, 1510 V, 1400 V e 1350 V, segundo indicação do fabricante.

O sinal de saída do PMT passa pelo pré-amplificador e segue para o módulo NDAQ, onde será digitalizado. A aquisição de dados foi sincronizada com a excitação do LED através da conexão da saída de sincronismo do gerador de sinais à entrada de *trigger* do módulo NDAQ. Isso significa que, para cada medida de amplitude, o LED emitiu fótons que, eventualmente, podem ser convertidos em fotoelétrons pelo PMT. O programa de aquisição do NDAQ gera uma tabela de dados com os resultados, que podem ser analisados por *software* destinado à análise de dados.

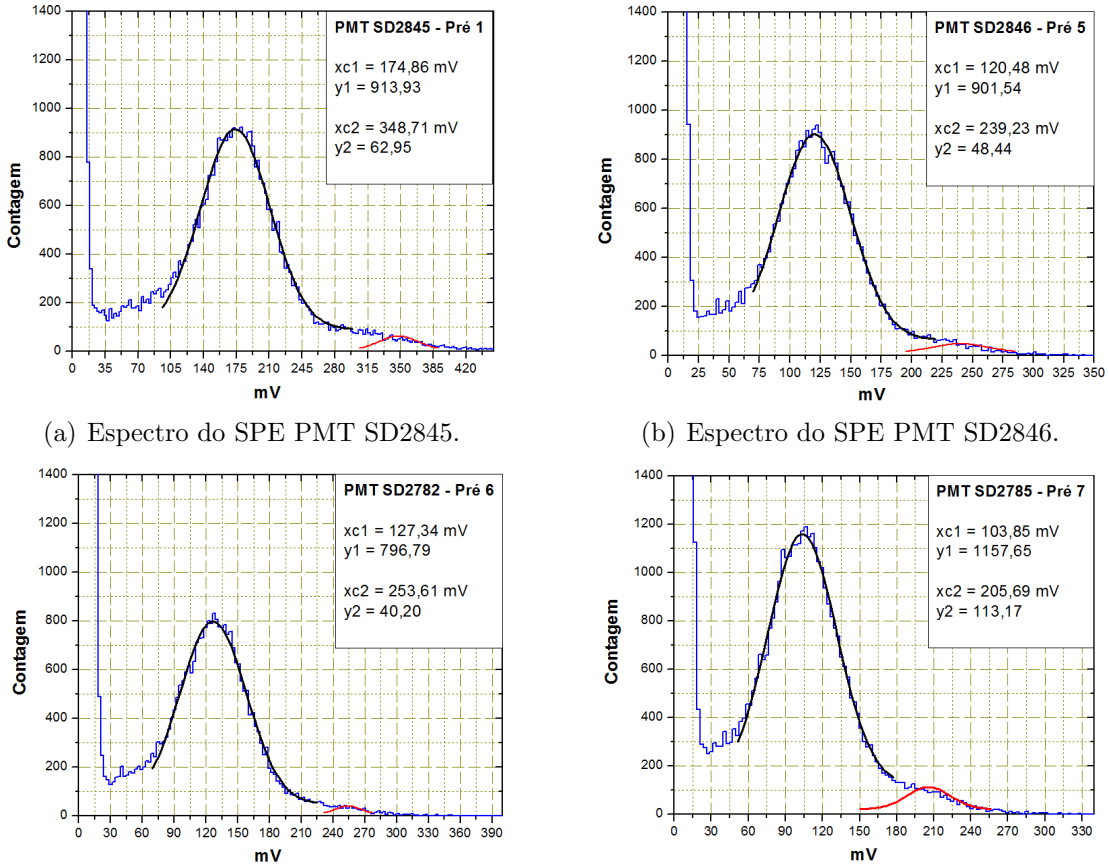
Resultados

A Figura 4.11 mostra os espectros do SPE obtidos para cada PMT. O pico que vai de 0 mV a aproximadamente 20 mV é decorrente do ruído do sistema de medida e alguma flutuação da linha de base do PMT.

Observando as gaussianas ajustadas nas figuras e os pontos centrais xc_1 e xc_2, nota-se que existe uma relação de carga coerente entre os espectros do SPE e 2PE. O pico central xc_2 possui um valor bem próximo do dobro do valor de xc1, caracterizando a ocorrência de 2 fotoelétrons.

Outra informação importante é o percentual de ocorrência de 2PE em relação a probabilidade do SPE, conforme visto no início dessa seção. Esse valor pode variar entre 5% e 10%. Os dados utilizados para efetuar esse cálculo são os valores de y_1 e y_2, observados na Figura 4.11. Esses parâmetros representam os valores máximos de contagem obtidos para o SPE e 2PE, respectivamente. Os valores encontrados foram:

- 6,88% para o PMT SD2845.
- 5,37% para o PMT SD2846.
- 5,04% para o PMT SD2782.
- 9,77% para o PMT SD2785.



(a) Espectro do SPE PMT SD2845.

(b) Espectro do SPE PMT SD2846.

(c) Espectro do SPE PMT SD2782.

(d) Espectro do SPE PMT SD2785.

Figura 4.11: Gráficos dos espectros obtidos de SPE.

Determinação do ganho dos PMT's

Segundo o fabricante dos PMT's, quando estes são alimentados com a tensão nominal, possuem um ganho da ordem de 10^7. Portanto, a carga teórica Q do SPE na saída dos PMT's deve ser:

$$Q = 1{,}6\ .\ 10^{-19}\ C\ .\ 10^7 = 1{,}6\ pC$$

Considerando que os pré-amplificadores foram calibrados em carga, ou seja, os valores de tensão de saída estão correlacionados com os valores da carga na entrada (ver equação 4.2), é possível determinar o valor da carga obtida para 1 fotoelétron na entrada do pré-amplificador (saída do PMT) dividindo a tensão de saída xc_1

(Figura 4.11) pelo ganho dos pré-amplificadores apresentados na Figura 4.7. Os valores encontrados para os PMT's SD2845, SD2846, SD2782 e SD2785 foram 2,83 pC, 1,98 pC, 2,1 pC e 1,82 pC, respectivamente. O ganho dos PMT's é determinado pela relação entre estes últimos valores citados e o valor teórico de 1,6 pC. A Tabela 4.2 mostra os valores dos ganhos obtidos.

PMT	**SD2845**	**SD2846**	**SD2782**	**SD2785**
Ganho do PMT	1,76 x 10^7	1,23 x 10^7	1,30 x 10^7	1,13 x 10^7

Tabela 4.2: Ganhos dos PMT's.

Ajuste da tensão de discriminação

Na saída do pré-amplificador, além dos pulsos de interesse, também estão presentes diversos pulsos de baixa amplitude. Muitos deles, são ruídos gerados pelo próprio pré-amplificador. Para evitar que esses pulsos sigam para o TDC e sejam contados indevidamente, é necessário utilizar um circuito discriminador para selecionar, por amplitude, os pulsos desejados. O circuito discriminador também é importante, pois formata os pulsos no padrão LVTTL, compatível com a entrada do FPGA. O discriminador utilizado está incorporado no módulo amplificador e seu esquemático pode ser visualizado na Figura 4.12.

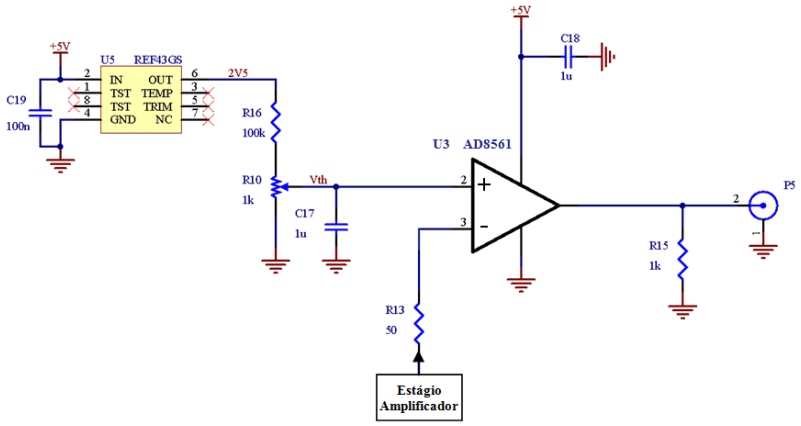


Figura 4.12: Circuito discriminador de tensão.

De acordo com [6], para determinação do espectro de corrente de escuro, a tensão de discriminação deve ser ajustada em torno de 1/4 da amplitude do valor máximo do espectro para o SPE. Através do divisor resistivo entre R_{16} e R_{10} (Figura 4.12) é possível ajustar, na entrada não-inversora do comparador, o valor de tensão de discriminação desejado. Dessa forma, toda vez que o sinal na saída do amplificador ultrapassar o valor de discriminação ajustado, o discriminador gera um pulso em sua saída (pino 8 de U3) no padrão LVTTL, compatível com a entrada da FPGA.

Na Figura 4.13 pode-se observar uma imagem retirada do osciloscópio que mostra no canal 1 (amarelo) três pulsos de corrente de escuro gerados por um PMT após passar pelo pré-amplificador. No canal 2 (verde) pode-se observar os pulsos discriminados e preparados para serem processados pela FPGA.

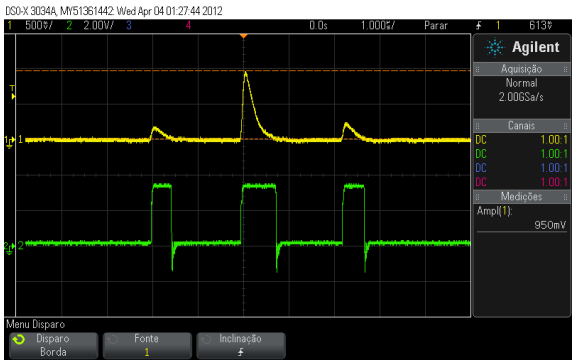


Figura 4.13: Pulsos amplificados (amarelo) e discriminados (verde).

Os valores das tensões de discriminação foram ajustados em 23,5 mV, 23 mV, 24,5 mV e 28 mV para os PMT's SD2845, SD2846, SD2782 e SD2785, respectivamente. Esses valores foram ajustados um pouco abaixo do valor teórico, porém ainda na região de vale do espectro do SPE.

4.1.4 Caracterização do TDC

Como visto na seção 3.12, um TDC mede o intervalo de tempo entre dois pulsos elétricos, convertendo esse valor em um dado digital. O TDC desenvolvido neste trabalho será utilizado para medir o tempo entre os pulsos de corrente de escuro

enviados pelo discriminador. Esses dados servirão como base para análise do comportamento dos PMT's utilizados no detector do projeto Neutrinos Angra.

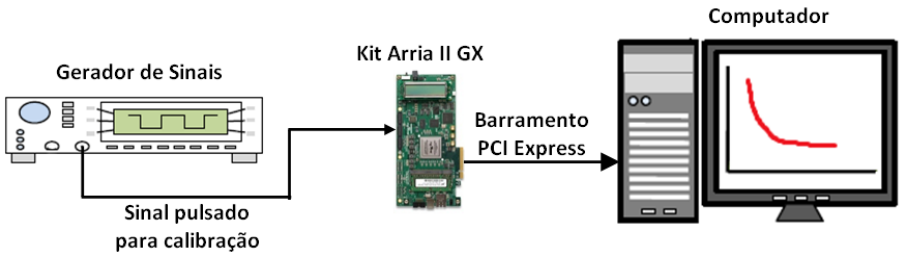


Figura 4.14: Aparato utilizado para calibração do TDC.

Metodologia

Para caracterizar o TDC em questão, são injetados pulsos de tensão com frequências conhecidas. O TDC faz a contagem dos pulsos de *clock* (40 MHz), no qual está operando, durante o intervalo de tempo entre os pulsos do sinal de entrada. Os resultados obtidos são multiplicados pelo período do referido *clock* (25 ns), obtendo-se o tempo entre cada pulso de entrada, ou seja, o período da frequência de entrada.

Foram injetados sinais com largura de pulso de 500 ns e frequência variando entre 2 kHz a 20 kHz, com passos de 2 kHz. O objetivo foi verificar toda a faixa de probabilidade de ocorrência de pulsos de corrente de escuro. Para cada valor de frequência injetado foram realizadas 655.360 medidas (10 x FIFO de 64 kB), calculadas as médias e o desvio padrão. Os dados são enviados para o computador via barramento PCIe do Kit Arria II GX. A Figura 4.14 ilustra o aparato utilizado para esta medida.

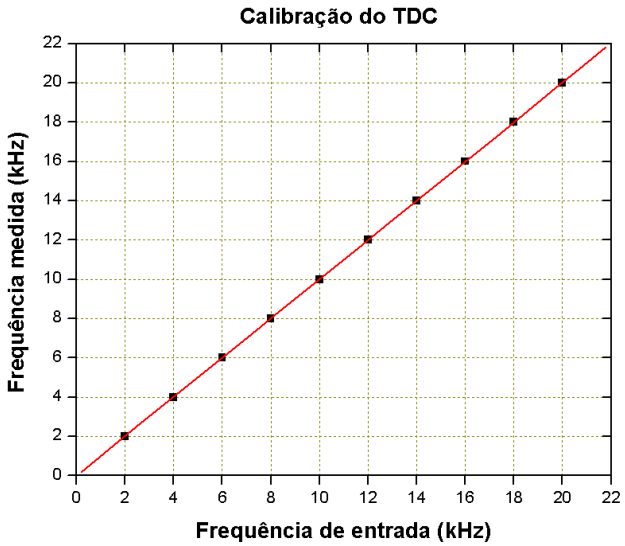


Figura 4.15: Curva de calibração do TDC.

Resultados

A Figura 4.15 relaciona os valores teóricos das frequências injetadas no TDC (eixo horizontal) com os valores das frequências calculadas a partir das médias dos períodos medidos pelo TDC (eixo vertical). Para todos os valores verificados o erro relativo é menor que 1%.

4.1.5 Verificação do espectro dos pulsos de corrente de escuro dos PMT's

Sabe-se que todo PMT, mesmo estando em um ambiente sem presença alguma de luz, apresenta pulsos de corrente em seu anodo, também conhecidos como *pulsos de corrente de escuro*. Este fenômeno, de natureza aleatória, pode ser causado por emissão termiônica do catodo, efeitos de campo ou corrente de fuga no circuito da base [6]. A verificação da distribuição dos pulsos de corrente de escuro no tempo e, da taxa de ocorrência desses eventos aleatórios são imprescindíveis para os primeiros testes a serem realizados no detector do projeto Neutrinos Angra. Para os tubos testados nesse trabalho, a taxa desses eventos está entre 3 kHz e 8 kHz, segundo o fabricante.

A Figura 4.16 ilustra o aparato, no qual foram realizadas as medidas.

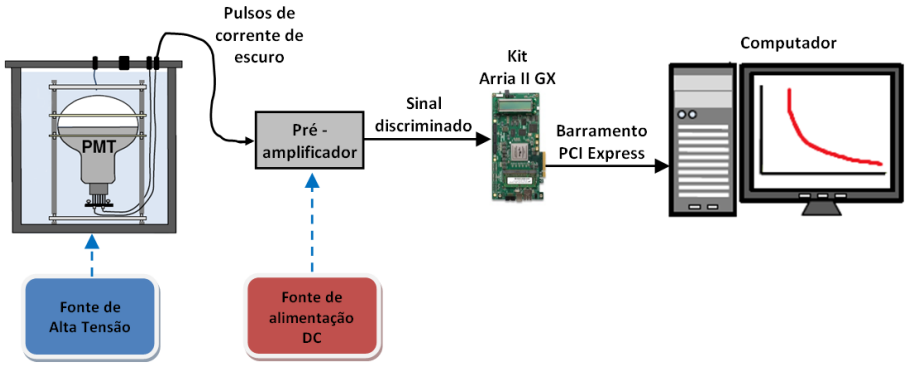


Figura 4.16: Aparato para medidas de corrente de escuro.

Metodologia

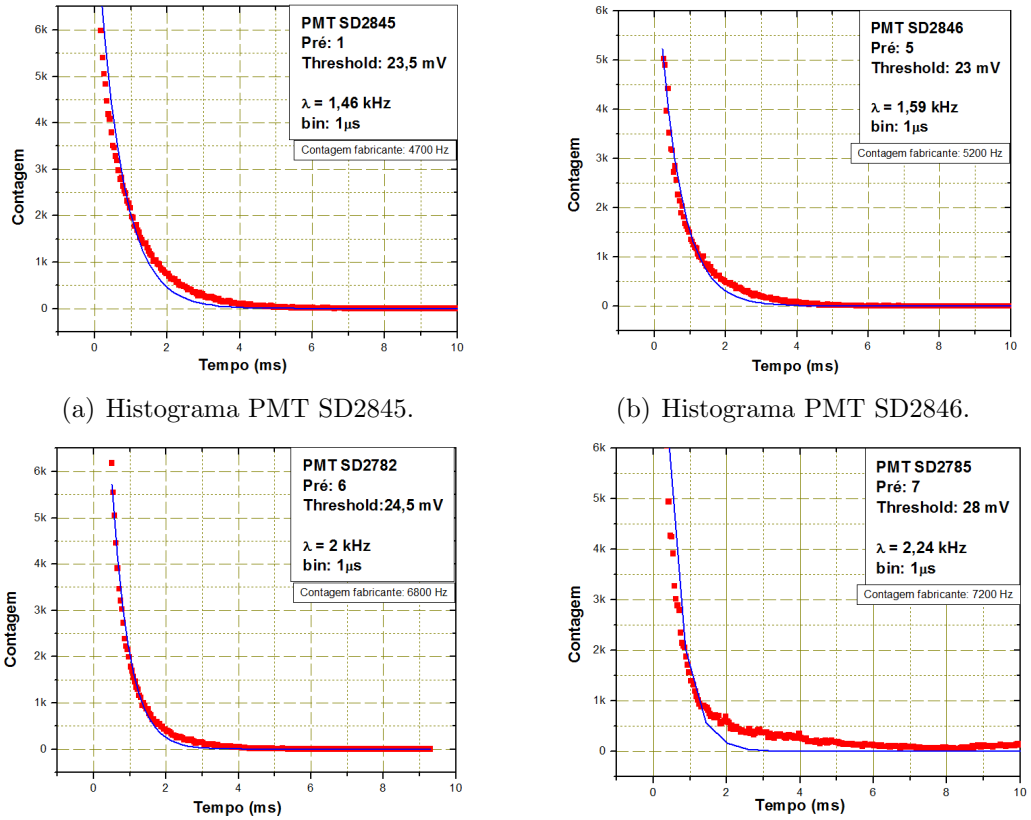
Os PMT's foram energizados com a tensão nominal indicada pelo fabricante e confinados na câmara escura. A fonte luminosa não foi utilizada, pois espera-se observar os pulsos gerados aleatoriamente pelo PMT, sem a presença de luz. O sinal discriminado segue para o Kit Arria II GX, onde será realizada a contagem dos evento casuais pelo TDC. Após isso, os dados são enviados ao computador via barramento PCIe para análise. A aquisição de dados foi realizada individualmente para os seguintes conjuntos de PMT + pré-amplificador:

- SD2845 e pré-amplificador 1.
- SD2846 e pré-amplificador 5.
- SD2782 e pré-amplificador 6.
- SD2785 e pré-amplificador 7.

Foram efetuadas 20 aquisições (20 x FIFO de 64 kB) para cada PMT.

Resultados

A Figura 4.17 apresenta os histogramas obtidos nas medidas realizadas com os PMT's.



(a) Histograma PMT SD2845.

(b) Histograma PMT SD2846.

(c) Histograma PMT SD2782.

(d) Histograma PMT SD2785.

Figura 4.17: Histogramas obtidos.

Pode-se observar que a distribuição possui uma forma de acordo com o esperado (Poisson) e a maioria das contagens se concentra na faixa de alguns kHz, conforme anuncia o fabricante dos PMT's. A taxa média (λ) de contagem, ajustada pela ferramenta de plotagem, é apresentada no gráfico. Os valores obtidos para o parâmetro λ, nos 4 PMT's, representam 30 % dos valores apresentadas pelo fabricante. Uma provável causa, pode ter sido a influência do estágio integrador dos pré-amplificadores, pois a ocorrência de vários pulsos em um intervalo tempo menor do que a constante de integração, acarretaria na integração dos mesmos em um único pulso. A Figura 4.18 ilustra este fenômeno.

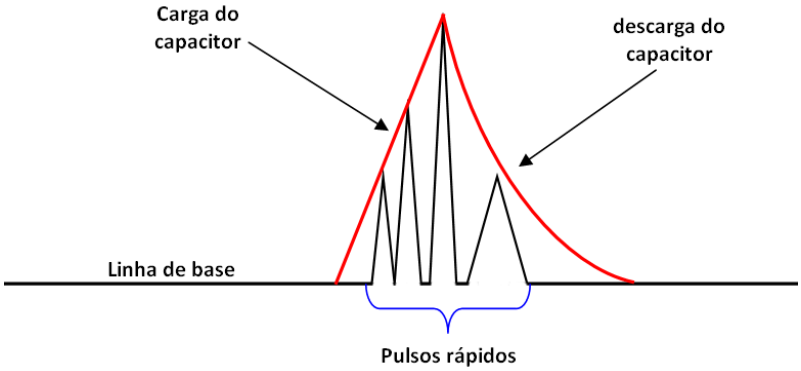


Figura 4.18: Integração de pulsos mais rápidos do que a constante de integração.

A indisponibilidade de um pré-amplificador sem o estágio integrador inviabilizou a realização de novas medidas, mas já se encontram em andamento estudos para investigar essa questão. É importante ressaltar que a instrumentação desenvolvida no MOPI para contagem dos pulsos de corrente de escuro contribuiu para a observação desse fato e é considerada confiável devido aos resultados de calibração.

4.1.6 Verificação do espectro dos pulsos de corrente de escuro em coincidência temporal

O detector do projeto Neutrinos Angra possui, na sua versão atual, 40 PMT's. Os eventos de interesse serão observados em várias PMT's, enquanto que os eventos de corrente de escuro são descorrelacionados. A contribuição dada ao projeto com essa medida foi verificar, em laboratório, o comportamento dos eventos casuais quando observados em grupo. Desta forma, considerou-se como um evento, a ocorrência de pulsos de corrente de escuro em coincidência temporal entre 2 PMT's ou 4 PMT's. A partir desses dados será possível fazer uma estimativa do número mínimo de PMT's que devem apresentar pulsos coincidentes correlacionados a um evento físico. A Figura 4.19 representa o aparato utilizado para essa medida.

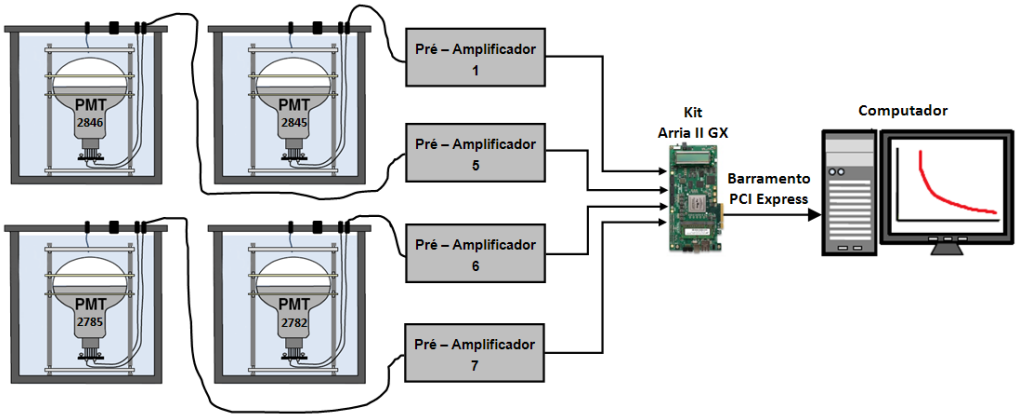


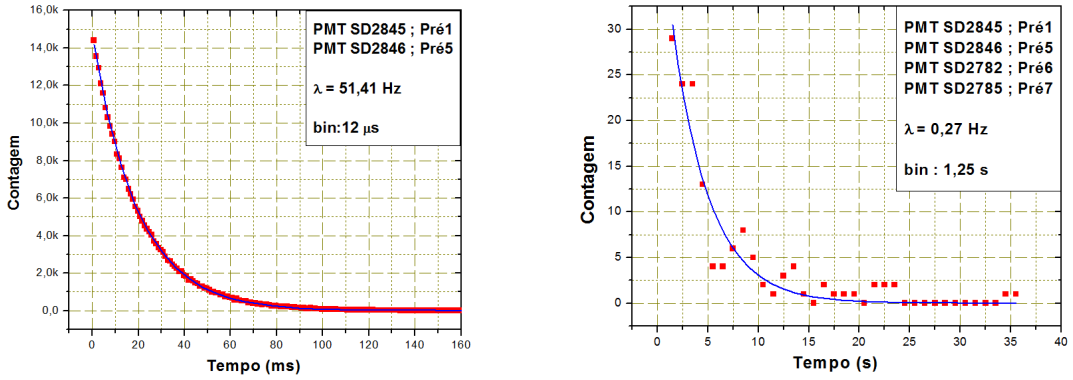
Figura 4.19: Visão simplificada do aparato para a medida de coincidência.

Metodologia

A metodologia aplicada nessa medida é a mesma aplicada na medida para um PMT. Os PMT's ficam confinados na câmara escura sob as mesmas condições. A diferença é que o TDC contará um evento somente quando ocorrerem pulsos em coincidência temporal.

Resultados

A Figura 4.20 mostra os resultados obtidos nas medidas realizadas no laboratório.



(a) Coincidência entre duas PMT's. (b) Coincidência entre quatro PMT's.

Figura 4.20: Taxas de coincidência obtidas para 2 PMT's e 4 PMT's.

Nota-se, através dos histogramas, que a região de contagem de pulsos de corrente de escuro, para 2 PMT's, se encontra na faixa de dezenas de Hertz, enquanto que para 4 PMT's em coincidência, a região de contagem de eventos é da ordem de 10^{-2} Hz. Esses resultados demonstram que as taxas de contagem diminuíram significativamente conforme esperado, porém ainda continuam acima dos valores de λ calculados em [50]. Esses cálculos probabilísticos consideram um λ máximo de 10 kHz para cada PMT e a ocorrência dos eventos casuais dentro de uma janela de tempo $\Delta t = 100\ ns$. A taxa de contagem para que m PMT's apresentem eventos casuais na mesma janela de observação Δt é dada pela expressão matemática:

$$\lambda^{(m)} = \lambda_1 \lambda_2 ... \lambda_m (\Delta t)^{m-1} \tag{4.11}$$

Para 2 PMT's e 4 PMT's os resultados dos cálculos foram 10 Hz e 10^{-5} Hz, respectivamente.

As diferenças encontradas entre esses valores teóricos e os resultados experimentais podem ter sua causa em dois aspectos a serem levados em consideração:

- Os valores de tensão de discriminação poderiam estar ajustados em uma região crítica, o que poderia acarretar em uma contagem eventual de pulsos de ruído eletrônico.

- Nas medidas realizadas neste trabalho não utilizou-se janelas temporais bem definidas para contagem de eventos casuais em coincidência. O processo baseou-se na contagem dos pulsos de saída de uma porta *and*, sintetizada no FPGA.

4.2 Espectroscopia de impedância elétrica

Nesta segunda aplicação do trabalho desenvolvido, é descrito um processo para a determinação da impedância complexa de uma amostra desconhecida em função da frequência.

Para determinar a impedância complexa (Zx) de um elemento desconhecido, um método possível é o descrito em [7]. Neste método, o elemento desconhecido (DUT - *Device Under Test*) é conectado serialmente a um resistor de referência, com uma resistência R conhecida, conforme ilustra a Figura 4.21.

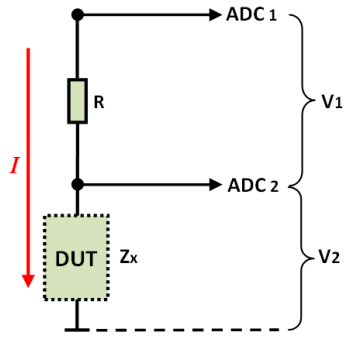


Figura 4.21: DUT interconectado ao resistor de referência para a determinação da impedância.

$$Zx = \frac{V_2}{I}\ (cos\theta + j\ sen\theta) = \frac{V_2}{I}\ e^{j\theta} \tag{4.12}$$

A impedância Zx, mostrada na equação 4.12, depende da tensão V_2, medida sobre o DUT, da corrente I e de θ. A corrente I é determinada a partir de R, segundo a Lei de Ohm (equação 4.13), uma vez que o resistor e o DUT estão em série.

$$I = \frac{V_1}{R} \tag{4.13}$$

Substituindo o valor de I encontrado em 4.13 na equação 4.12, a impedância desconhecida pode ser reescrita de acordo com a equação 4.14.

$$Zx = R\ \frac{V_2}{V_1}\ e^{j\theta} \tag{4.14}$$

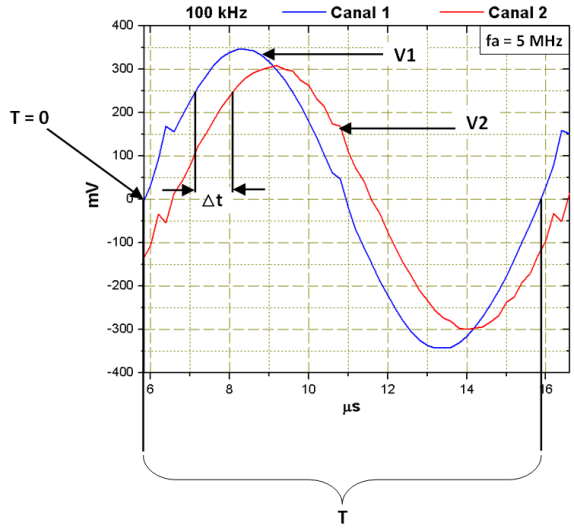


Figura 4.22: Defasagem entre V_1 e V_2.

Considerando-se os sinais V_1 e V_2 na Figura 4.22, θ pode ser determinado através da relação:

$$\frac{\theta}{\Delta t} = \frac{2\pi}{T} \ \Rightarrow\ \theta = 2\pi f \Delta t \tag{4.15}$$

onde T é o período de ambos os sinais, V_1 e V_2.

Portanto, adotando as equações 4.14 e 4.15, para determinar a impedância complexa do elemento desconhecido é necessário somente determinar a relação de amplitude e a diferença de fase entre os sinais V_1 e V_2.

4.2.1 Caracterização do DDS

O objetivo desta caracterização é demonstrar a precisão do DDS e verificar se ele possui os requisitos necessários para a aplicação. Duas características foram rele-

vantes para a utilização desse dispositivo: a integridade do sinal gerado e a *interface* de comunicação SPI. A Figura 4.23 ilustra o aparato utilizado na caracterização do DDS.

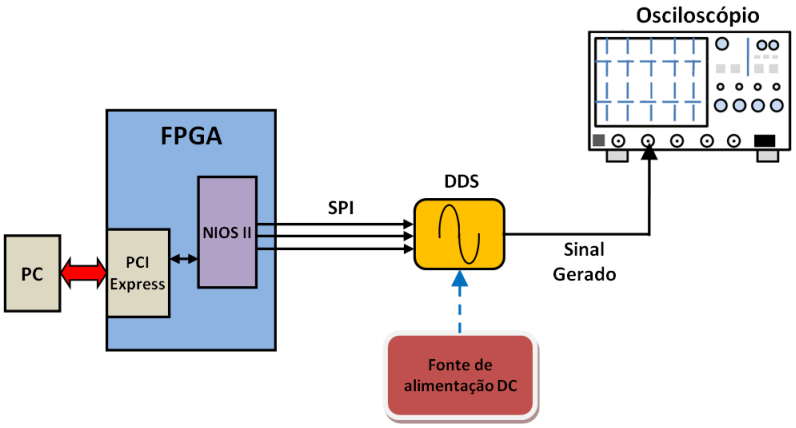


Figura 4.23: Aparato de caracterização do DDS.

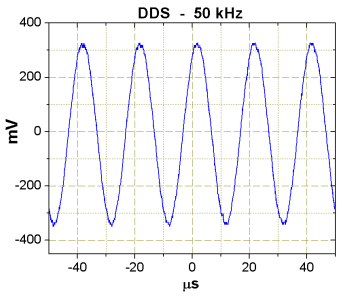
Metodologia

A frequência do DDS pode ser ajustada através de um registrador interno via protocolo SPI. A faixa de frequência verificada foi de 5 kHz a 500 kHz com passos de 1 kHz.

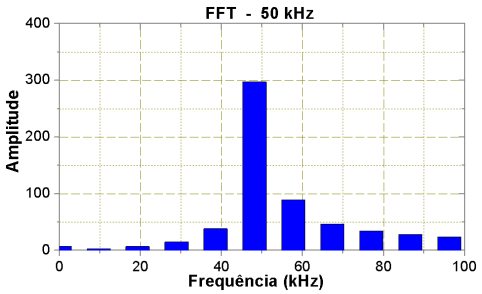
Resultados

A título de exemplo, nas Figuras 4.24 e 4.25 são apresentados quatro sinais com frequências diferentes. O osciloscópio[1] foi ajustado de maneira que possam ser visualizados 5 ciclos dos sinais gerados pelo DDS. Através da transformada de Fourier, é possível verificar que a maior parte da potência do sinal está na frequência principal, mais pronunciada no centro do espectro. A existência de outras componentes de frequência pode ser explicada pelo efeito de janelamento (*windowing effect*) do sinal. O uso de somente uma janela de tempo do sinal para o cálculo da FFT, ao invés de um número infinito de amostras, produz componentes artificiais (*side lobes*) que na verdade não estão presentes no sinal original [51].

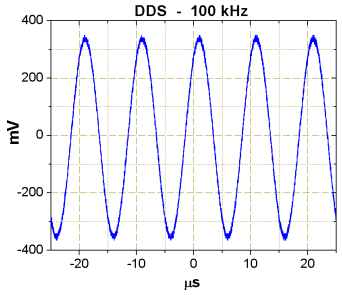
[1]Fabricante Agilent, modelo DSO-X 3034A (Tabela 4.1).



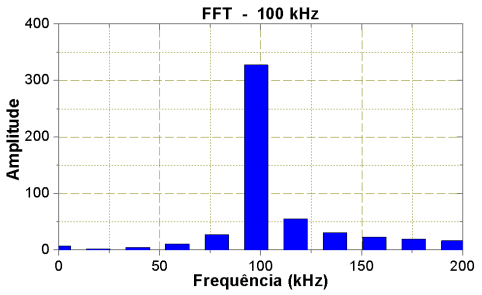
(a) 50 kHz



(b) FFT 50 kHz

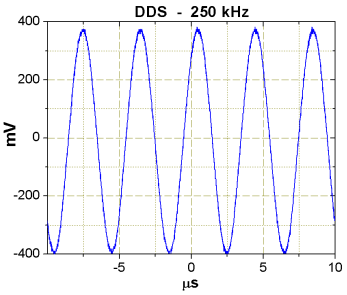


(c) 100 kHz

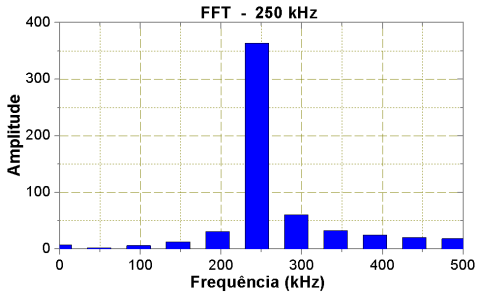


(d) FFT 100 kHz

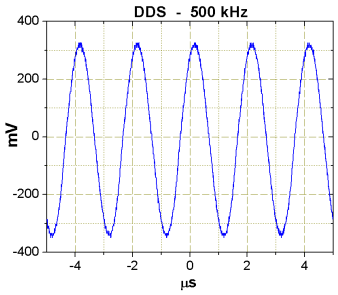
Figura 4.24: Sinais gerados pelo DDS e as respectivas FFT's.



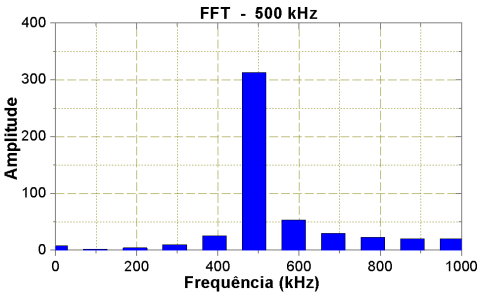
(a) 250 kHz



(b) FFT 250 kHz



(c) 500 kHz



(d) FFT 500 kHz

Figura 4.25: Sinais gerados pelo DDS e as respectivas FFT's.

Como pode ser observado, os resultados apresentados mostram que o DDS é capaz de gerar o sinal senoidal dentro da faixa de interesse. O erro do DDS, anunciado pelo fabricante, é de 1,5%, enquanto que o do osciloscópio é de 0,0025%.

4.2.2 Caracterização do conversor analógico-digital

Este ADC faz parte de um módulo desenvolvido no CBPF [52] para aquisição de dados em um experimento com raios cósmicos. Foram utilizados dois módulos iguais para se medir V_1 e V_2, ver Figura 4.21, devido à necessidade de dois canais de digitalização para as medidas de EIE. Optou-se pela utilização dos módulos em conjunto com o kit Arria II GX, devido a indisponibilidade do MOPI no momento das medidas.

Para garantir a integridade dos dados adquiridos digitalmente nas medidas realizadas, os ADC's utilizados foram submetidos a uma caracterização. O ADC utilizado foi o ADS828 [53], fabricado pela *Texas Instruments*. Este dispositivo é alimentado com + 5 V, possui resolução de 10 bits, e pode operar com frequência de amostragem de até 75 MHz. A faixa dinâmica da tensão de entrada pode ser programada para 1 Vpp ou 2 Vpp. Sua arquitetura do tipo *pipeline* permite que o resultado das digitalizações seja disponibilizado continuamente no barramento de saída. A Figura 4.26 ilustra o aparato de calibração.

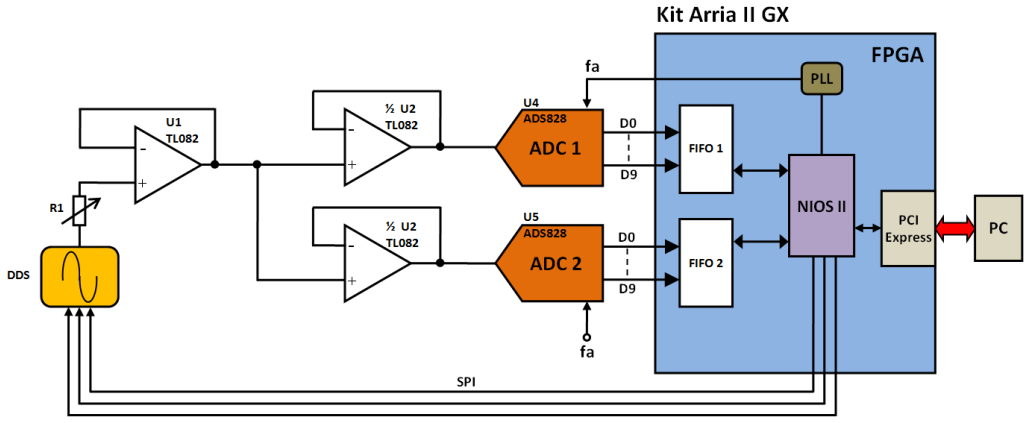


Figura 4.26: Aparato utilizado para a calibração dos ADC's.

Metodologia

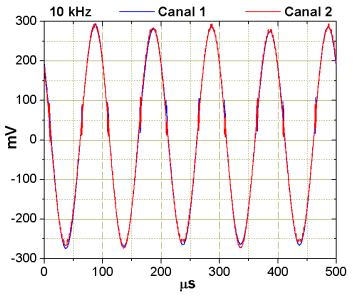
O DDS é configurado pelo NIOS II, de modo a fazer uma varredura das frequências na faixa situada entre 5 kHz e 500 kHz. Para cada frequência gerada, é realizada uma aquisição (1 x FIFO de 65.536 kB). A amplitude dos sinais gerados pelo DDS foi ajustada em 650 mVpp por meio do resistor variável R_1, enquanto que os ADC's estão configurados para capturar sinais com até 1 Vpp. Cada canal de digitalização está conectado a uma memória FIFO sintetizada no FPGA. A frequência de amostragem (f_a) dos ADC's é ajustada pelo NIOS II, por meio de uma PLL (*Phase-locked loop*), em 5 MHz, ou seja, dez vezes a frequência máxima gerada pelo DDS.

Para casamento de impedância entre os dispositivos foram utilizados três amplificadores operacionais TL082 [54].

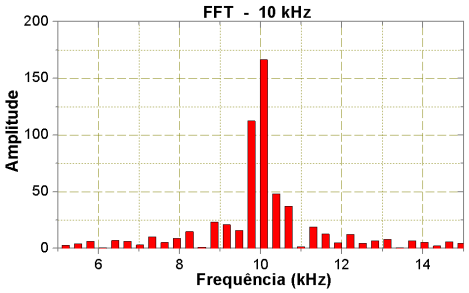
Resultados

Nas Figuras 4.27 e 4.28 são apresentados os resultados obtidos para as frequências de 10 kHz, 100 kHz, 200 kHz e 400 kHz. Os sinais dos dois canais estão sobrepostos para que seja visualizada a ausência de defasagem entre eles e a homogeneidade entre os canais de digitalização, conforme esperado.

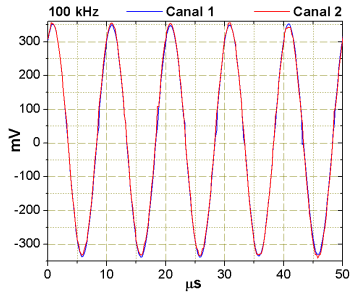
Da mesma forma que nas Figuras 4.24 e 4.25, os gráficos da transformada de Fourier apresentam a frequência principal do sinal (ao centro) com maior potência. A presença de componentes de outras frequências pode ser explicada pelo efeito de janelamento (*windowing effect*) do sinal. O uso de somente uma janela de tempo do sinal para o cálculo da FFT, ao invés de um número infinito de amostras, produz componentes artificiais (*side lobes*) que na verdade não estão presentes no sinal original.



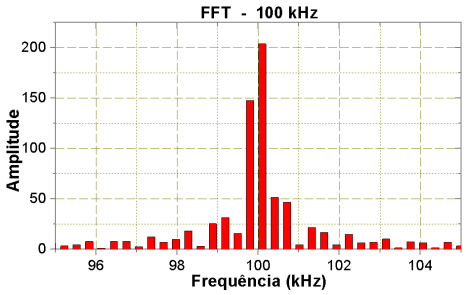
(a) 10 kHz



(b) FFT 10 kHz

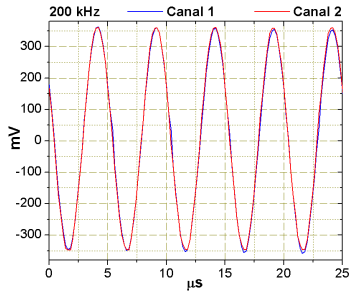


(c) 100 kHz

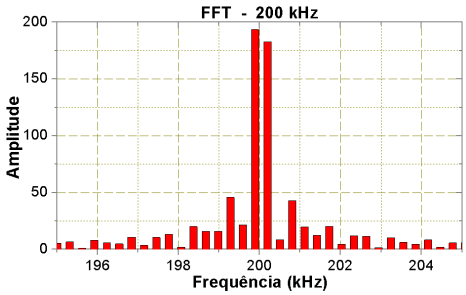


(d) FFT 100 kHz

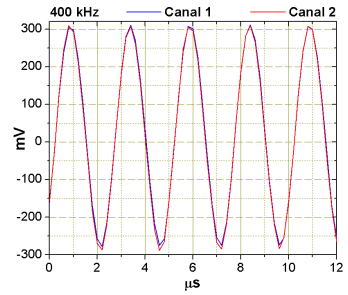
Figura 4.27: Sinais adquiridos pelos ADC's e as respectivas FFT's.



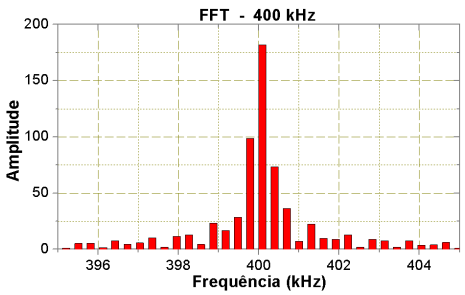
(a) 200 kHz



(b) FFT 200 kHz



(c) 400 kHz



(d) FFT 400 kHz

Figura 4.28: Sinais adquiridos pelos ADC's e as respectivas FFT's.

4.2.3 Verificação do ganho de tensão e da defasagem em filtros RC

Para verificar a capacidade do sistema em detectar diferenças de fase e amplitude entre sinais senoidais, foram analisados três filtros RC de 1ª ordem [45], com frequências de corte (f_c) em 32 kHz, 68 kHz e 96 kHz.

A Figura 4.29 apresenta os elementos utilizados. A esquerda, montados sobre *protoboard*, estão os circuitos do DDS, os amplificadores operacionais e o filtro RC. No centro da Figura estão os dois módulos com os ADC's utilizados e à direita o Kit Arria II GX.

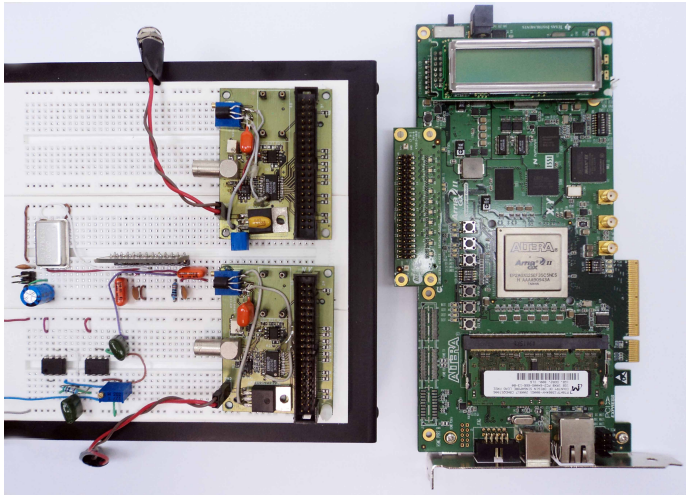


Figura 4.29: Elementos utilizados nas medidas de EIE.

Metodologia

O sinal gerado pelo DDS é injetado na entrada do filtro e adquirido pelo ADC 1, enquanto que o sinal de saída (sobre o capacitor) é digitalizado pelo ADC 2, como ilustra a Figura 4.30.

Os dados convertidos são escritos nas FIFO's e posteriormente são transferidos, por meio do barramento PCIe, para o PC.

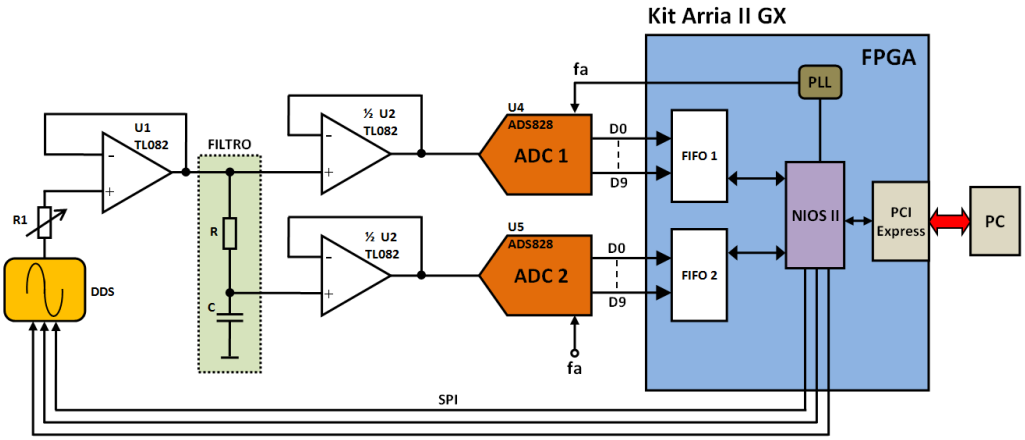


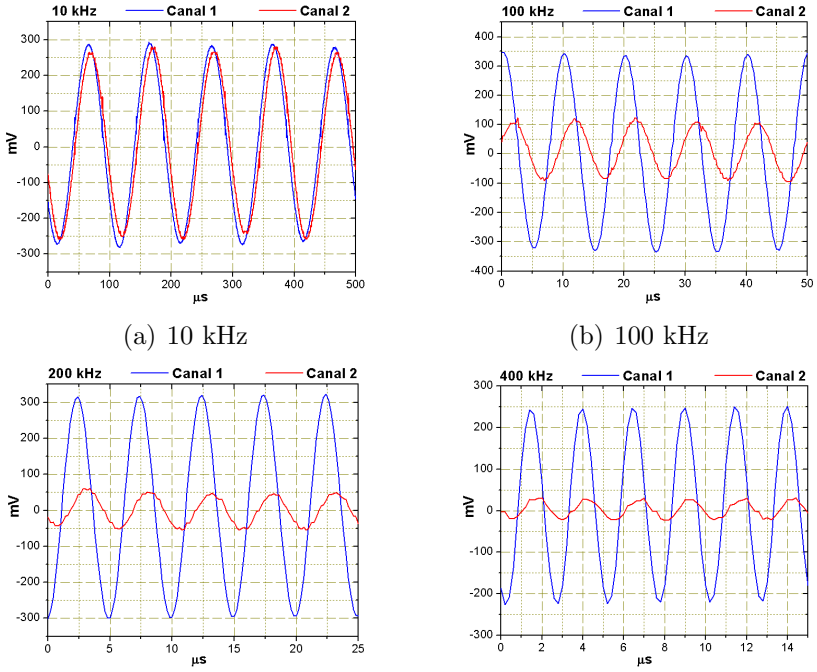
Figura 4.30: Circuito montado para a determinação do ganho de tensão e da defasagem dos filtro RC.

Resultados

As Figuras 4.31, 4.32 e 4.33 apresentam as medidas obtidas. O canal 1, em azul, é o sinal injetado na entrada dos filtros, enquanto que, o canal 2, em vermelho, é o sinal de saída.

São apresentados resultados para sinais com frequência de 10 kHz, 100 kHz, 200 kHz e 400 kHz, para os três filtros utilizados. A frequência de amostragem (f_a) foi ajustada em 5 MHz em todos os casos.

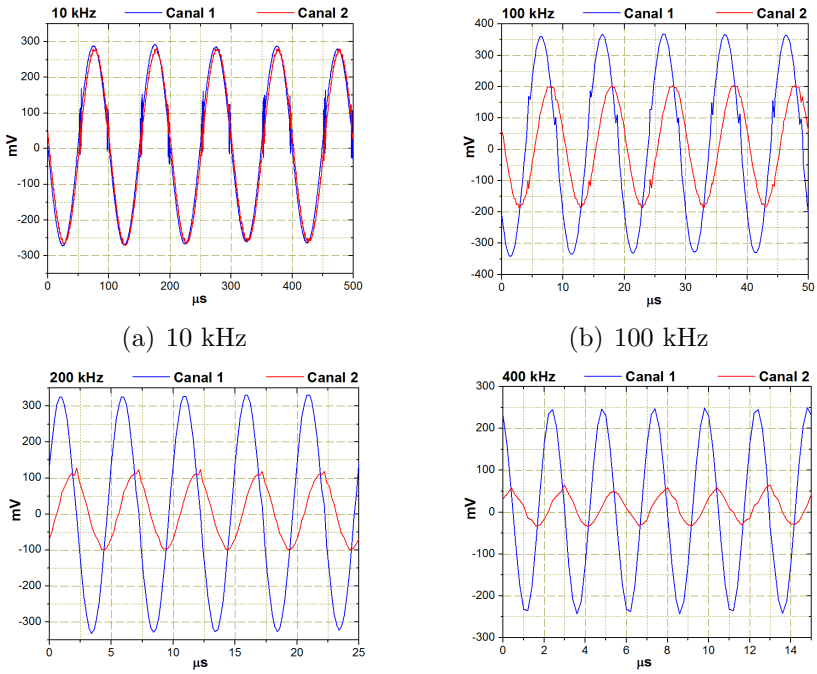
É possível observar que o deslocamento de fase e a redução da amplitude do sinal de saída são mais pronunciados à medida que a frequência de entrada se aproxima da frequência de corte, conforme esperado.



(a) 10 kHz (b) 100 kHz

(c) 200 kHz (d) 400 kHz

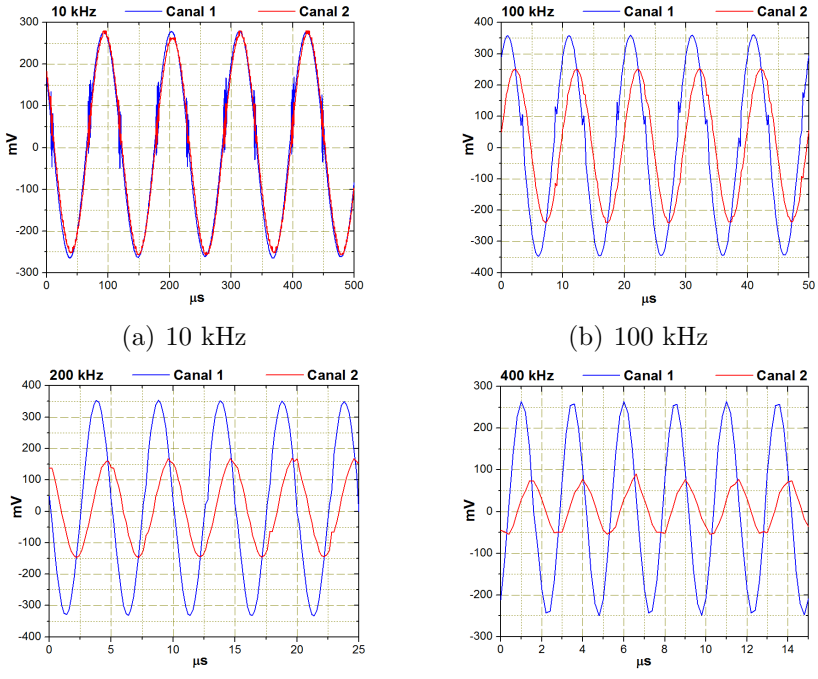
Figura 4.31: Filtro com $f_c = 32$ kHz (R = 50 Ω e C = 100 ηF na Figura 4.30).



(a) 10 kHz (b) 100 kHz

(c) 200 kHz (d) 400 kHz

Figura 4.32: Filtro com $f_c = 68$ kHz (R = 50 Ω e C = 47 ηF na Figura 4.30).



(a) 10 kHz (b) 100 kHz

(c) 200 kHz (d) 400 kHz

Figura 4.33: Filtro com f_c = 96 kHz (R = 50 Ω e C = 33 ηF na Figura 4.30).

Para análise dos dados obtidos, foi desenvolvido um aplicativo, em linguagem C^{++}, capaz de determinar o ganho de tensão e a diferença de fase dos filtros RC em função da frequência. Teoricamente, na frequência de corte (f_c) o ganho de tensão é atenuado em aproximadamente 30% e o ângulo de defasagem é de 45º [45]. As figuras 4.34, 4.35 e 4.36 apresentam os resultados desta análise.

Observando os gráficos é possível perceber a existência de uma oscilação em torno do valor médio (ajustado em vermelho), tanto nos resultados do ganho, quanto nos de fase. Dentre as possíveis causas, destaca-se um aumento do nível de ruído, em parte pelo fato de o sistema estar montado sobre um *protoboard*.

Os maiores erros de fase ocorrem nas frequências mais elevadas e podem estar associados aos baixos níveis de amplitude nessa região.

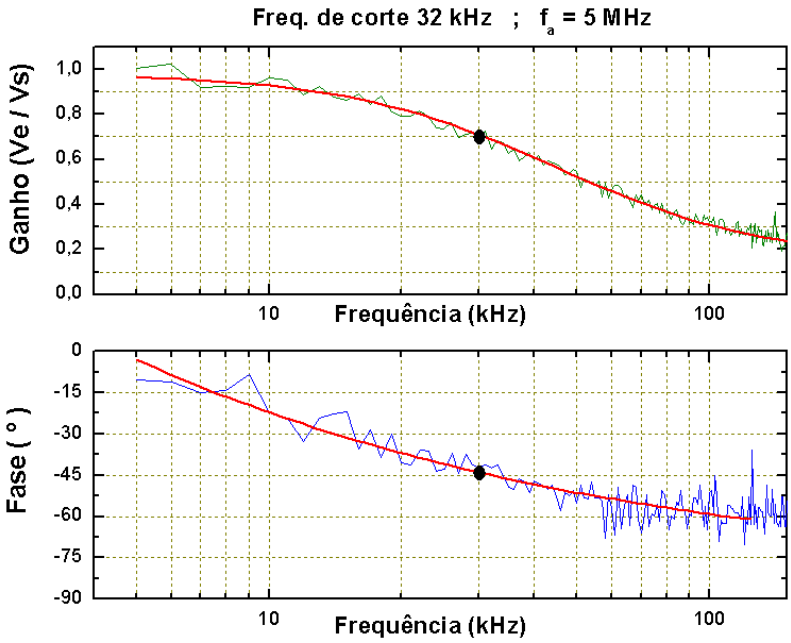


Figura 4.34: Ganho e fase com frequência de corte sintonizada em 32 kHz (R = 50 Ω e C = 100 ηF).

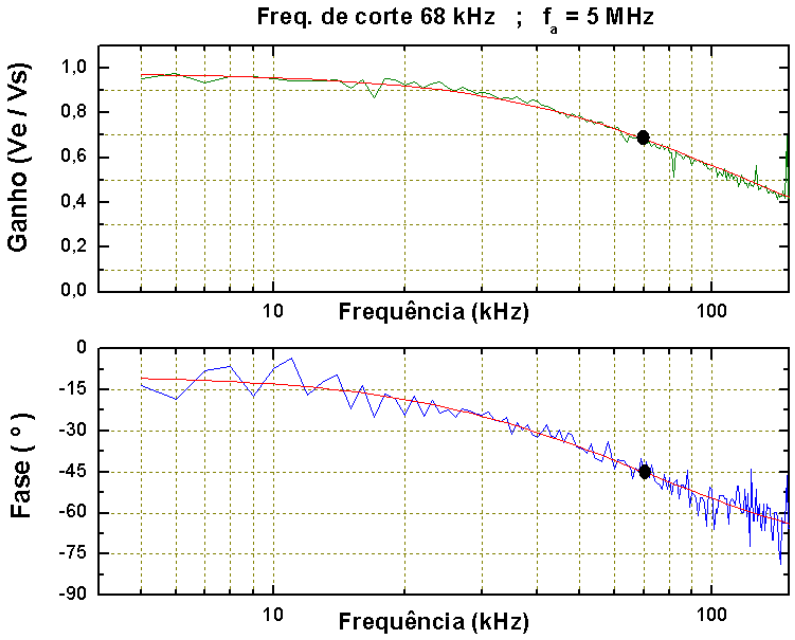


Figura 4.35: Ganho e fase com frequência de corte sintonizada em 68 kHz (R = 50 Ω e C = 47 ηF).

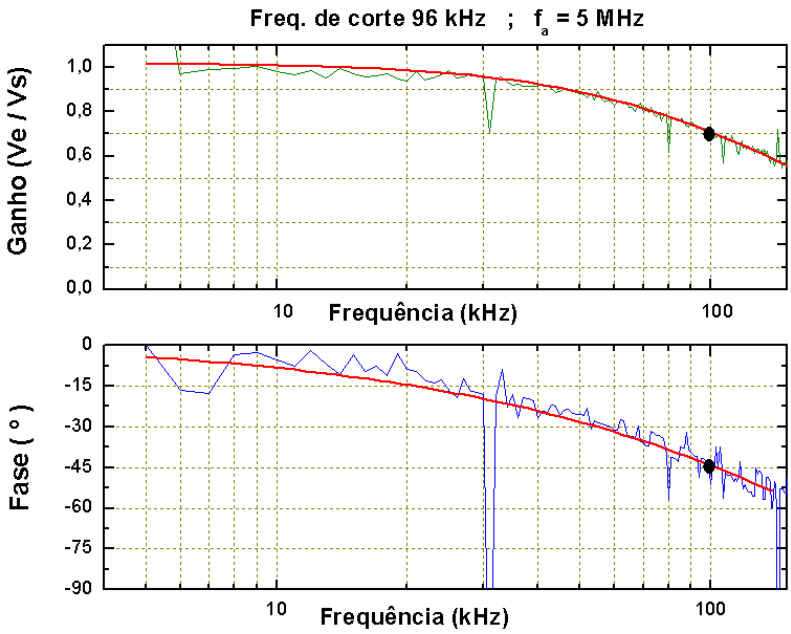


Figura 4.36: Ganho e fase com frequência de corte sintonizada em 96 kHz (R = 50 Ω e C = 33 ηF).

Apesar dessas características, observa-se que o ganho de tensão, assim como a fase, na frequência de corte, estão coerentes com os valores esperados. Para uma melhor visualização, esses pontos estão destacados com um ponto sobre o ajuste das curvas.

4.2.4 Determinação do espectro de impedância

De uma maneira geral, a técnica de espectroscopia de impedância elétrica consiste em aplicar um sinal senoidal, com amplitude constante e frequência variável, a uma amostra de material sob investigação. A partir do sinal obtido sobre a amostra é possível determinar o espectro de impedância, que revela características peculiares do material, como por exemplo a pureza e o grau de homogeneidade.

O espectro da impedância complexa é a representação gráfica das partes real e imaginária da impedância em função da frequência. A parte real refere-se ao resistor e a parte imaginária ao elemento capacitivo ou indutivo.

Foram utilizadas nesse trabalho quatro amostras com o espectro de impedância conhecido para que se pudesse comparar os resultados obtidos com os valores teóricos. As amostras utilizadas foram: um capacitor, um circuito RC em série, um circuito RC em paralelo e um circuito RC misto, conforme ilustra a Figura 4.37.

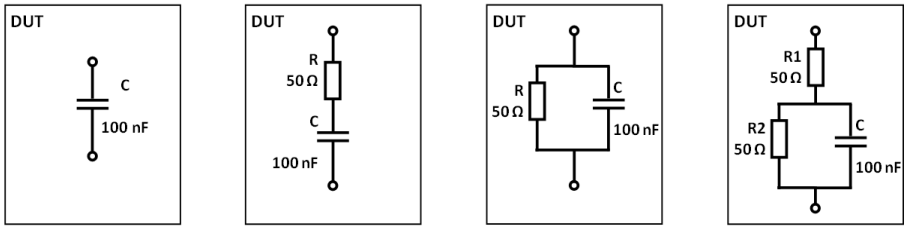


Figura 4.37: Amostras utilizadas no experimento.

Metodologia

Para determinar a impedância complexa desconhecida de uma amostra (DUT), é necessário interconectá-la em série com um resistor de referência (REF). A partir das quedas de tensão sobre esses elementos é possível obter as diferenças de amplitude e fase, que possibilitam determinar a impedância complexa do DUT. A Figura 4.38 apresenta o diagrama em blocos do aparato utilizado.

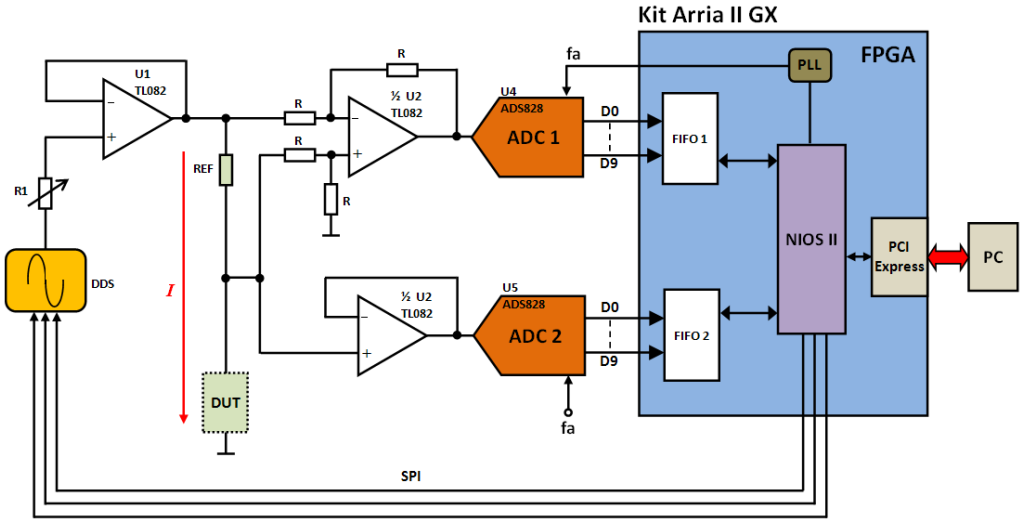


Figura 4.38: Circuito para realizar medidas de impedância de amostras desconhecidas.

O sinal injetado pelo DDS no circuito faz com que circule uma corrente i através do resistor de referência (REF) e da amostra (DUT). O sinal sobre o resistor REF é adquirido em modo diferencial por meio de um amplificador operacional configurado para este fim, enquanto que o sinal sobre o DUT é adquirido em relação ao terra do circuito.

Resultados

As medidas foram analisadas por meio de um aplicativo específico desenvolvido no CBPF. A interface gráfica deste aplicativo é apresentada na Figura 4.39.

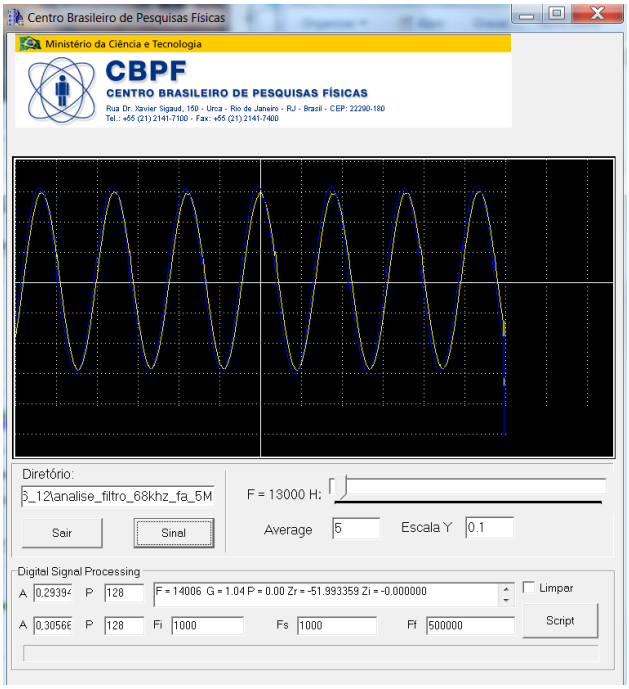


Figura 4.39: Interface gráfica do aplicativo utilizado para análise de impedância complexa.

Observando a Figura 4.40 é possível perceber, por meio do ajuste das curvas (em magenta para a parte real e em vermelho para parte imaginária), que de uma maneira geral, os resultados obtidos estão de acordo com os valores esperados. Nota-se que, apenas a parte imaginária do circuito misto apresenta uma dispersão maior em torno do valor ajustado. Uma possível causa para esse efeito pode estar associada à redução da tensão no circuito RC paralelo, ocasionada pela adição do resistor R_1

no circuito. Esse fato pode fazer com que o aplicativo de análise perca precisão nessa região.

Com o auxílio das expressões matemáticas para impedância complexa (Z), é possível fazer uma análise mais detalhada de cada caso. A impedância complexa dos circuitos analisados é descrita matematicamente da seguinte forma:

- Circuito puramente capacitivo: $Z = -\frac{1}{j\omega C}$.

  A parte real da impedância complexa de um capacitor é nula, enquanto que, a parte imaginária possui um comportamento inversamente proporcional à variação da frequência. A Figura 4.40(a) apresenta esse comportamento.

- Circuito RC em série: $Z = R - \frac{1}{j\omega C}$.

  Nesse circuito, é possível perceber que a parte real assume valores constantes em torno de 50 Ω em todo o espectro. A parte imaginária apresenta um comportamento inversamente proporcional à variação da frequência, conforme mostra a Figura 4.40(b).

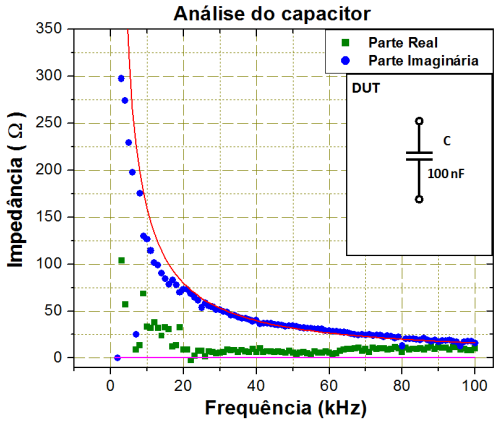
- Circuito RC em paralelo: $Z = \frac{R}{1+(\omega RC)^2} - \frac{\omega R^2 C}{1+(\omega RC)^2} j$.

  No circuito RC em paralelo, para frequências baixas há predomínio da parte real em torno de 50 Ω enquanto que a parte imaginária tende a zero. À medida que a frequência aumenta a parte real diminui rapidamente e nota-se um rápido aumento da parte imaginária. Na região das frequências intermediárias, percebe-se pela Figura 4.40(c) uma proximidade entre os valores assumidos pelas duas parcelas da impedância complexa. Na região de frequências altas, a parte real tende a zero, ao passo que a parte imaginária também diminui, porém não de forma abrupta.

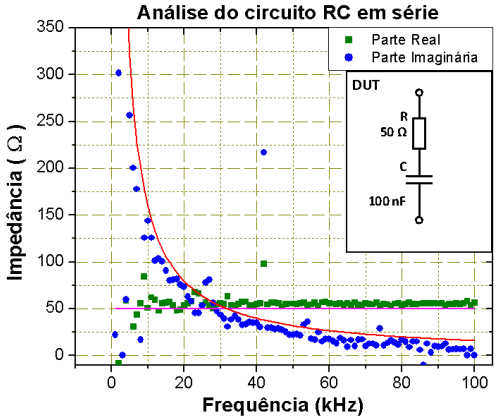
- Circuito RC misto: $Z = R_2 + \frac{R_1}{1+(\omega R_1 C)^2} - \frac{\omega R_1^2 C}{1+(\omega R_1 C)^2} j$.

  Nessa associação, a parte real da impedância assume valores mais elevados em todo o espectro. Na região de baixas frequências, a parte real atinge valores próximos de 100 Ω, ou seja, como nessa região a parte imaginária assume valores muito elevados, o valor da resistência equivalente da associação em

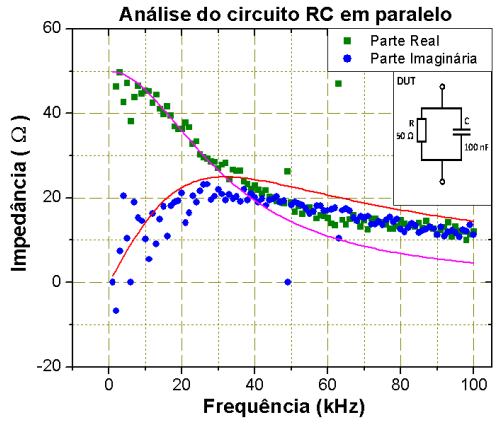
paralelo entre R_2 e C tende a 50 Ω. Dessa forma, predomina uma associação em série entre R_1 e R_2 que resulta em 100 Ω. A medida que a frequência aumenta a parte imaginária tende a zero suavemente e a parte real vai em direção a 50 Ω. Isso ocorre devido ao fato do capacitor se comportar como um curto-circuito nas frequências elevadas.



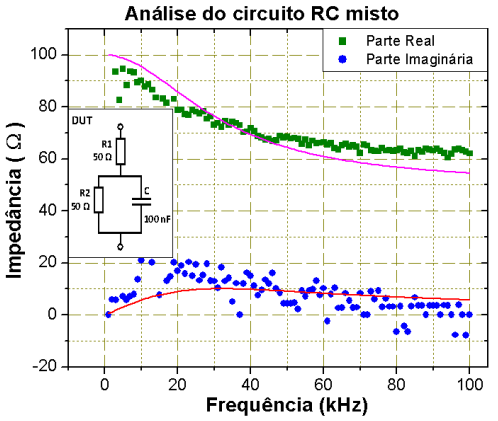
(a) Impedância complexa de uma amostra puramente capacitiva.



(b) Impedância complexa de um circuito RC em série.



(c) Impedância complexa de um circuito RC em paralelo.



(d) Impedância complexa de um circuito RC misto.

Figura 4.40: Impedância complexa de circuitos RC tomados como amostras desconhecidas.

5 Conclusão e perspectivas

Conforme proposto no início desse trabalho, foi desenvolvido um módulo eletrônico de alta velocidade capaz de contribuir com a instrumentação científica nas instituições de pesquisa. Todos os esquemáticos do módulo foram desenvolvidos no CBPF. O projeto do *layout*, a produção da placa de circuito impresso e a montagem dos componentes foram realizadas por empresas especializadas. O *hardware* encontra-se em fase de testes no laboratório do Lafex, enquanto que o *firmware* foi utilizado com sucesso, através de um kit de desenvolvimento comercial, em duas aplicações: no Projeto Neutrinos Angra e nas medidas de Espectroscopia de Impedância Elétrica.

Para o Projeto Neutrinos Angra, o TDC implementado foi utilizado na análise dos possíveis efeitos dos pulsos de corrente de escuro no funcionamento do detector principal de anti-neutrinos. Nessa aplicação foi possível determinar os espectros dos pulsos de corrente de escuro de alguns PMT's individualmente e em coincidência temporal. Entre as medidas realizadas, destaca-se a caracterização do TDC, onde foram obtidos erros inferiores a 1%.

Nas medidas de Espectroscopia de Impedância Elétrica, o objetivo foi verificar os circuitos do sintetizador de sinais e de aquisição de dados. A implementação do sistema de EIE foi bastante interessante, pois além de abrir possibilidades de utilização dessa técnica no CBPF, verificou-se que o MOPI poderá ser utilizado como elemento principal nesse tipo de experimento. Nas medidas realizadas, a performance do DDS destacou-se pela sua precisão. Na caracterização desse dispositivo com um osciloscópio, o erro obtido foi inferior a 1,5%. Tanto o *hardware* como o aplicativo desenvolvidos para a determinação da impedância complexa de amostras

desconhecidas apresentaram os resultados esperados, demonstrando que é possível avançar nessa direção e obter melhores resultados através da otimização do sistema.

Foi utilizada uma versão preliminar do *driver* PCI *Express* desenvolvido para o sistema operacional Linux [43]. A taxa de transmissão de dados alcançada entre o computador e o bloco PCIe do *firmware* com esta versão do driver foi da ordem de 5 MBytes/s. Entretanto, já se encontra em fase de testes - sendo tema de outra dissertação de mestrado - uma nova versão de *driver*, baseada na transferência de dados via DMA (*Direct memory access*), o que possibilitará atingir as taxas de transferência de dados especificadas pelo protocolo.

Como perspectivas futuras de utilização dessa nova ferramenta, é possível destacar:

- Realização das mesmas medidas agora com o MOPI, para revalidação do *firmware* e comparação do desempenho.
- Através da transferência de dados em alta velocidade, via DMA, ler e visualizar os dados convertidos pelo ADC quase em tempo real.
- Implementação do *firmware* para controle e leitura do conversor digital-analógico, permitindo o uso do MOPI no controle de experimentos ou de processos industriais.
- Utilização no processamento digital de sinais, auxiliando experimentos em andamento no CBPF.
- Desenvolvimento de novas versões mais otimizadas e robustas.

É importante ressaltar que o MOPI, apesar de ser um protótipo, é um produto tecnológico desenvolvido no CBPF e possui um grande potencial de aprimoramento.

Referências Bibliográficas

[1] LHCb. *Public web pages*. Http://lhcb.web.cern.ch/lhcb/.

[2] NóBREGA, R. A. *Development of the Experiment Control System and Performance Study of the Muon Chambers for the LHCb Experiment*. [S.l.]: Universita di Roma, 2010.

[3] BUDRUK, R.; ANDERSON, D.; SHANLEY, T. *PCI Express System Architecture*. 2nd. ed. [S.l.]: MindShare, Inc, 2003.

[4] GRIFFITHS, D. *Introduction to Elementary Particles*. 2. ed. [S.l.]: Wiley-VCH, 2009.

[5] SUTTON, C. *Spaceship Neutrino*. [S.l.]: Cambridge University Press, 1992.

[6] PHILIPS Photonics. *Photomultiplier tubes - principles and applications*. [S.l.: s.n.], 1994.

[7] CHINAGLIA, D. et al. Espectroscopia de impedância no laboratório de ensino. *Revista Brasileira de Ensino de Física*, n. 4, 2008.

[8] BROWN, B. H. et al. *Tretrapolar measurement of cervical tissue structure using impedance spectroscopy*. *IEEE Proccedigns of the 20th Annual International Conference on Biomedical Engineering*, v. 6, 1998.

[9] SHANLEY, T.; ANDERSON, D. *ISA System Architecture*. 3rd. ed. [S.l.]: MindShare, Inc, 1995.

[10] SHANLEY, T.; ANDERSON, D. *PCI System Architecture*. 4rd. ed. [S.l.]: MindShare, Inc, 1999.

[11] SHANLEY, D. D. . T. *AGP System Architecture*. [S.l.]: MindShare, 1999.

[12] RIDGEWAY, C. A.; SANGHA, H. S. *Making the move to serial buses*. Http://www.edn.com.

[13] ANDERSON, D. *FireWire System Architecture - IEEE 1394*. [S.l.]: MindShare, 1998.

[14] AXELSON, J. *USB Complete*. 3. ed. [S.l.]: Lakeview Research LLC, 2006.

[15] PRESS, W. H. *Numerical Recipes:The art of Scientific Computing*. 3rd. ed. [S.l.]: Cambridge University Press, 2007.

[16] JOHNSON, H.; GRAHAM, M. *High-Speed Digital Design, A Handbook of Black Magic*. [S.l.]: Prentice Hall, 1993.

[17] WILEN, A.; SCHADE, J.; THORNBURG, R. *Introduction to PCI Express - A Hardware and Software Developer's Guide*. 1. ed. [S.l.]: Intel Press, 2003.

[18] IEEE, Inc. *IEEE Std 1149.1 - Standard Test Access Port and Boundary Scan Architecture*. [S.l.]: IEEE, Inc.

[19] IEEE, Inc. *IEEE Std 1532 - Standard for In-System Configuration of Programmable Devices*. [S.l.]: IEEE, Inc.

[20] ANALOG Devices. *Intefacing High Speed ADCs via SPI /AN877*. [S.l.].

[21] PHILIPS Semiconductors. *The I^2C-bus specification - version 2.1*. [S.l.]: Philips, 2000.

[22] ALTERA Corporation. *Arria II GX FPGA Development Kit - User Guide*. [S.l.], 2009.

[23] ALTERA Corporation. *Quartus II Handbook Version 11.0*. [S.l.], may 2011.

[24] ALTIUM Ltd. *Altium Designer*. Http://www.altium.com.

[25] ALTERA Corporation. *Cyclone IV Device Handbook*. [S.l.], 2011.

[26] BROWN, S.; VRANESIC, Z. *Fundamentals of Digital Logic with VHDL Design*. [S.l.]: McGraw-Hill, 2000.

[27] ALTERA Corporation. *SOPC Builder - User Guide*. [S.l.], 2010.

[28] ALTERA Corporation. *PCI Express Compiler - User Guide*. [S.l.], 2011.

[29] ALTERA Corporation. *Nios II Processor Reference Handbook*. [S.l.], 2011.

[30] ALTERA Corporation. *Nios II Software Developer's Handbook*. [S.l.], 2011.

[31] FTDI Ltd. [S.l.], 2009.

[32] JR., D. F. H. *Analog-to-Digital and Digital-to-Analog Conversion Techniques*. [S.l.]: John Wiley & Sons Inc, 1994.

[33] ANALOG Devices. *Interfacing to data converters*. Http://www.analog.com.

[34] ANALOG Devices. *Complete DDS*. [S.l.], 2011.

[35] SANTOS, G. N.; BATISTA, P. D. Caracterização de um dispositivo sintetizador digital de sinais (DDS) como um gerador de sinais. *Nota Técnica/CBPF*, 2011.

[36] ANALOG Devices. *A Technical Tutorial on Digital Signal Synthesis*. [S.l.], 1999.

[37] HAYKIN, S.; VEEN, B. V. *Sinais e Sistemas*. [S.l.]: Bookman, 2001.

[38] ANALOG Devices. *TxDAC Digital-to-Analog Converters AD9116*. [S.l.], 2011.

[39] PHILIPS Eletronics. *PCI Express PHY PCB Layout Guideline*. [S.l.], 2005.

[40] SATURN PCB Design, Inc. *Saturn PCB*. Http://www.saturnpcb.com.

[41] BAYER, E.; TRAXLER, M. *A High-Resolution (< 10 ps RMS) 48-Channel Time-to-Digital Converter (TDC) Implemented in a Field Programmable Gate Array (FPGA). IEEE Transactions on nuclear science*, 2011.

[42] PARK, M. *Low power digital PLL based TDC using low rate clocks. Electronics Letters*, 2011.

[43] LESSA, L. H. P. Desenvolvimento de um barramento PCI Express para a comunicação entre um processador e uma FPGA. *Programa de Capacitação Institucional do MCT/CBPF*, 2010.

[44] GAMA, R. G. Desenvolvimento de instrumentação baseada em lógica programável para aquisição de dados no Projeto Neutrinos Angra. *Dissertação de Mestrado/CBPF*, 2011.

[45] HOROWITZ, P.; HILL, W. *The Art of electronics*. 2. ed. [S.l.]: Cambridge University Press, 1989.

[46] OTT, H. W. *Noise Reduction Techniques in Electronic Systems*. 2. ed. [S.l.]: Wiley-Interscience, 1988.

[47] FORBES, C. et al. *Statistical Distributions*. 4. ed. [S.l.]: Wiley, 2010.

[48] BUENO, A. *Characterization of large area photomultipliers and its application to dark matter search with noble liquid detectors*. 2007. ArXiv:0711.3592v2.

[49] HANHARDT, M. *Hamamatsu R7081 Photomultiplier Tube Characterization for LUX Dark Matter Search*. 2011.

[50] BARBOSA, A. F.; ABRAHAO, T. Um estudo da taxa de eventos casuais no detector central. *AngraNote 18*, 2011.

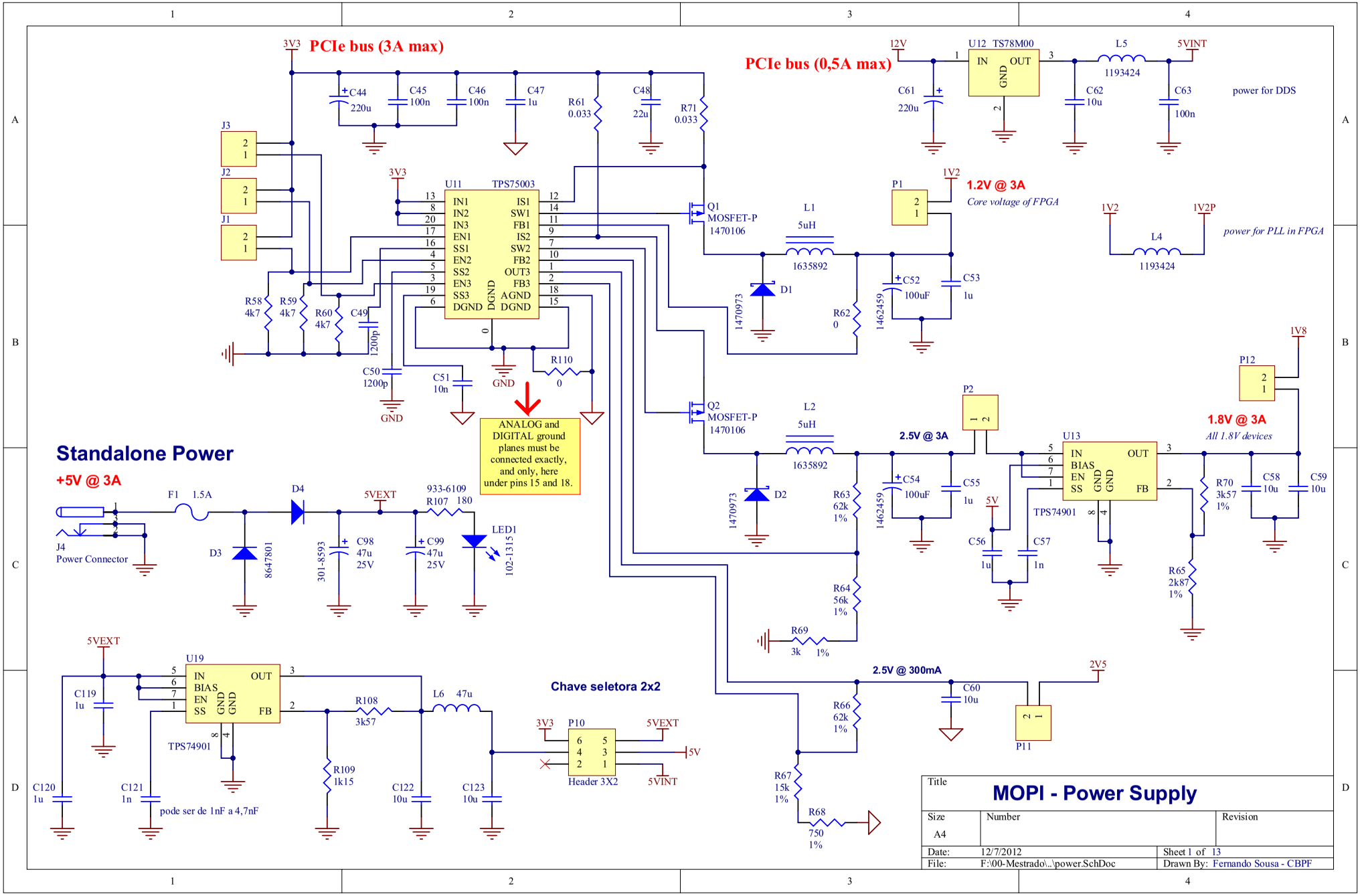
[51] OPPENHEIM, A. V. *Discrete-Time Signal Processing*. 3. ed. [S.l.]: Pearson, 2009.

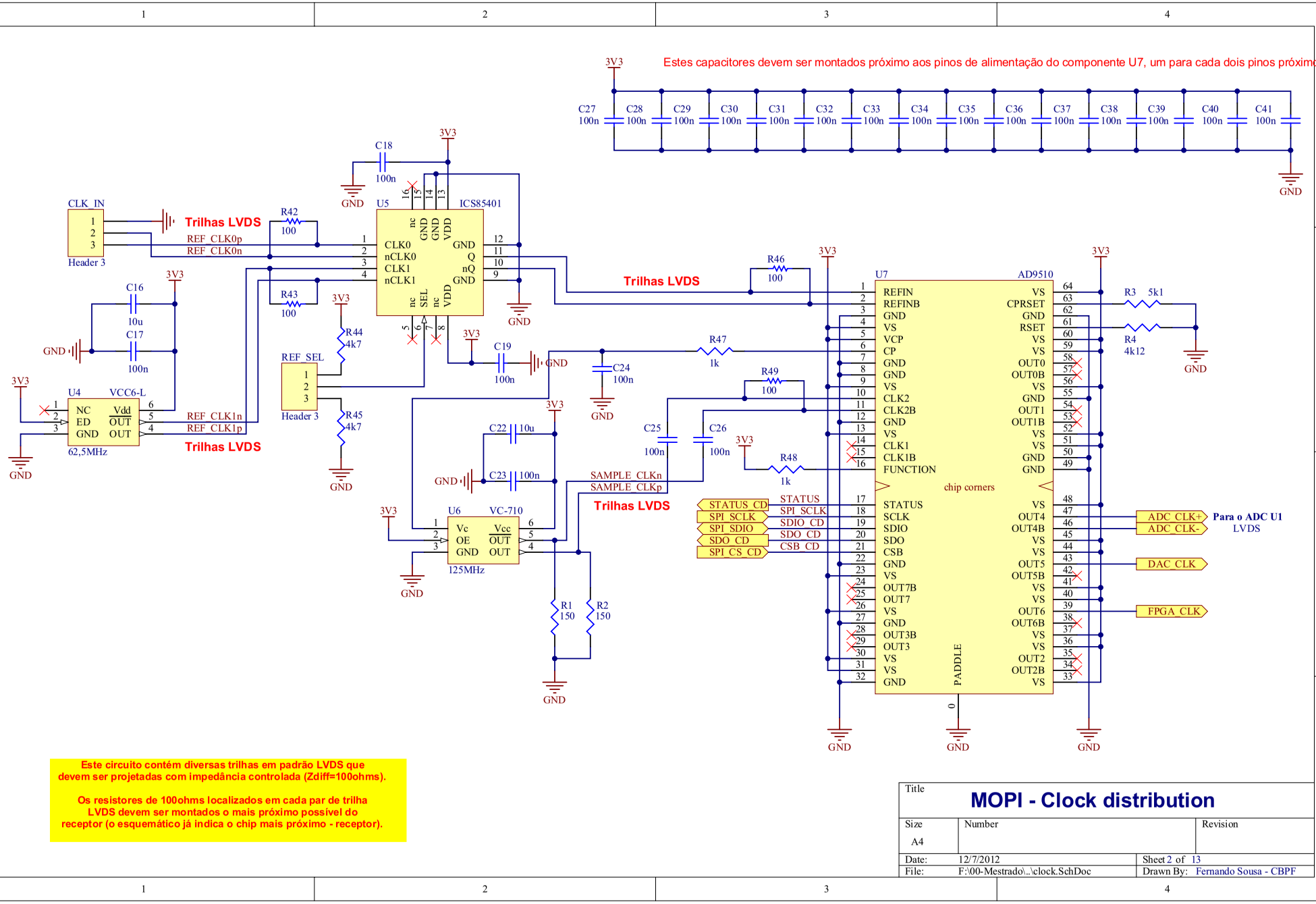
[52] SILVA, R. M. da. Digitalizador de forma de onda e aquisição de dados em um detector Cerenkov em água. *Dissertação de Mestrado/CBPF*, 2005.

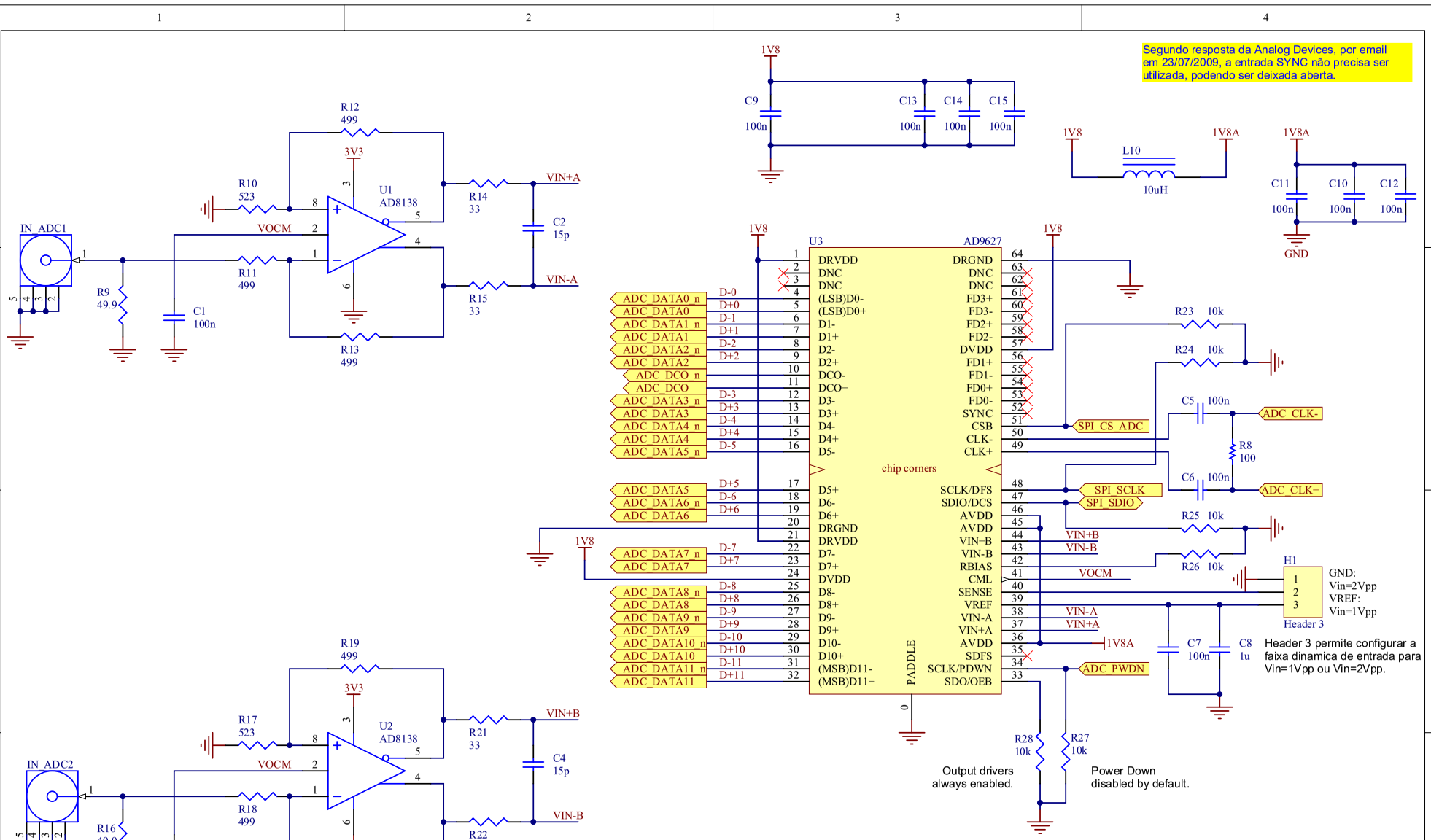
[53] TEXAS Instruments. *ADS828 - Datasheet*. Http://www.ti.com.

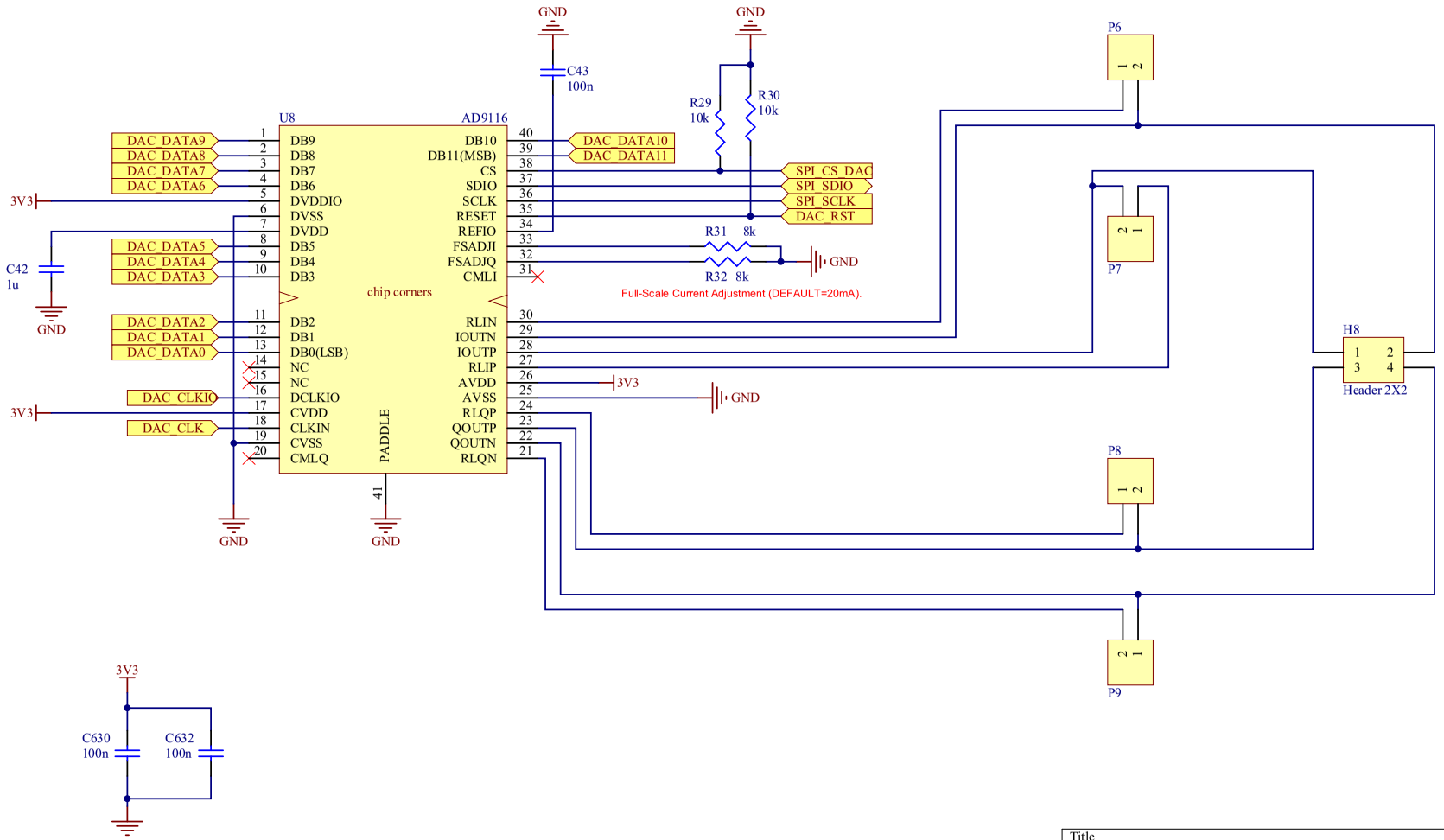
[54] TEXAS Instruments. *TL082 - Datasheet*. Http://www.ti.com.

APÊNDICE A -- Esquemáticos MOPI

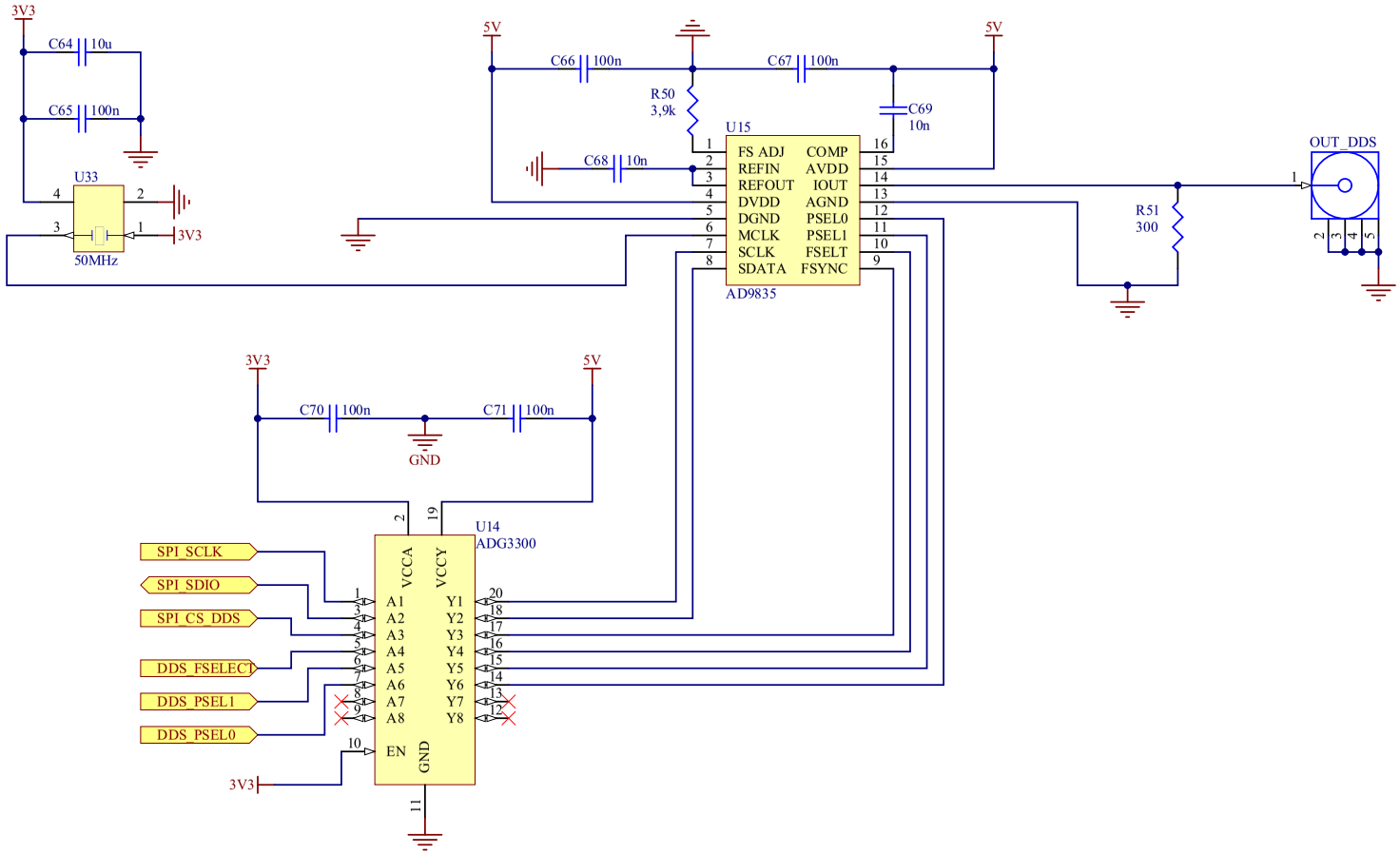

PCIe bus (3A max)
PCIe bus (0,5A max)
Standalone Power
+5V @ 3A
1.2V @ 3A
Core voltage of FPGA
power for DDS
power for PLL in FPGA
2.5V @ 3A
1.8V @ 3A
All 1.8V devices
2.5V @ 300mA
ANALOG and DIGITAL ground planes must be connected exactly, and only, here under pins 15 and 18.
Chave seletora 2x2
Header 3X2
pode ser de 1nF a 4,7nF
TPS74901
TPS75003
MOPI - Power Supply
Size A4
Number
Revision
Date: 12/7/2012
Sheet 1 of 13
File: F:\00-Mestrado\..\power.SchDoc
Drawn By: Fernando Sousa - CBPF


Estes capacitores devem ser montados próximo aos pinos de alimentação do componente U7, um para cada dois pinos próximos.
3V3
C27 100n
C28 100n
C29 100n
C30 100n
C31 100n
C32 100n
C33 100n
C34 100n
C35 100n
C36 100n
C37 100n
C38 100n
C39 100n
C40 100n
C41 100n
GND
CLK_IN
Header 3
Trilhas LVDS
REF_CLK0p
REF_CLK0n
R42 100
R43 100
C18 100n
U5 ICS85401
CLK0
nCLK0
CLK1
nCLK1
GND
Q
nQ
VDD
nc
SEL
C16 10u
C17 100n
R44 4k7
REF_SEL
Header 3
R45 4k7
C19 100n
U4 VCC6-L
NC
ED
Vdd
OUT
62,5MHz
REF_CLK1n
REF_CLK1p
Trilhas LVDS
C22 10u
C23 100n
U6 VC-710
Vc
OE
Vcc
125MHz
SAMPLE_CLKn
SAMPLE_CLKp
Trilhas LVDS
R1 150
R2 150
C24 100n
C25 100n
C26 100n
R46 100
R47 1k
R49 100
R48 1k
U7 AD9510
REFIN
REFINB
VS
VCP
CP
CLK2
CLK2B
CLK1
CLK1B
FUNCTION
STATUS
SCLK
SDIO
SDO
CSB
OUT7B
OUT7
OUT3B
OUT3
PADDLE
chip corners
CPRSET
RSET
OUT0
OUT0B
OUT1
OUT1B
OUT4
OUT4B
OUT5
OUT5B
OUT6
OUT6B
OUT2
OUT2B
STATUS_CD
SPI_SCLK
SPI_SDIO
SDO_CD
SPI_CS_CD
SDIO_CD
CSB_CD
R3 5k1
R4 4k12
ADC_CLK+
ADC_CLK-
Para o ADC U1
LVDS
DAC_CLK
FPGA_CLK
0
Este circuito contém diversas trilhas em padrão LVDS que devem ser projetadas com impedância controlada (Zdiff=100ohms).
Os resistores de 100ohms localizados em cada par de trilha LVDS devem ser montados o mais próximo possível do receptor (o esquemático já indica o chip mais próximo - receptor).
Title
MOPI - Clock distribution
Size
A4
Number
Revision
Date: 12/7/2012
Sheet 2 of 13
File: F:\00-Mestrado\..\clock.SchDoc
Drawn By: Fernando Sousa - CBPF

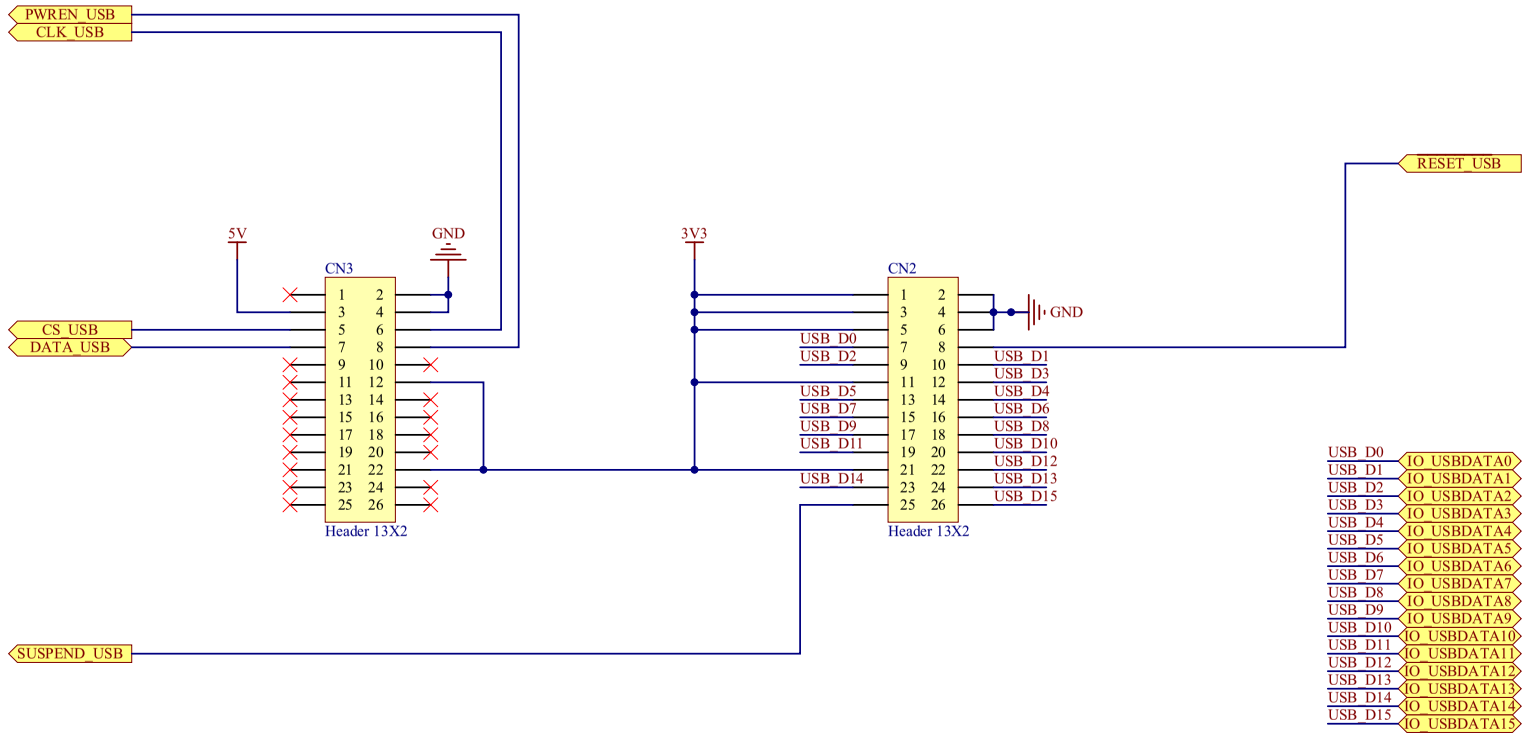

Segundo resposta da Analog Devices, por email em 23/07/2009, a entrada SYNC não precisa ser utilizada, podendo ser deixada aberta.
IN_ADC1
IN_ADC2
U1 AD8138
U2 AD8138
U3 AD9627
VOCM
3V3
1V8
1V8A
VIN+A
VIN-A
VIN+B
VIN-B
R9 49.9
R16 49.9
R10 523
R17 523
R11 499
R12 499
R13 499
R18 499
R19 499
R20 499
R14 33
R15 33
R21 33
R22 33
C1 100n
C3 100n
C2 15p
C4 15p
C9 100n
C13 100n
C14 100n
C15 100n
C11 100n
C10 100n
C12 100n
L10 10uH
GND
chip corners
PADDLE
SPI_CS_ADC
SPI_SCLK
SPI_SDIO
ADC_CLK-
ADC_CLK+
ADC_PWDN
R23 10k
R24 10k
R25 10k
R26 10k
R27 10k
R28 10k
C5 100n
C6 100n
R8 100
C7 100n
C8 1u
H1
Header 3
GND:
Vin=2Vpp
VREF:
Vin=1Vpp
Header 3 permite configurar a faixa dinamica de entrada para Vin=1Vpp ou Vin=2Vpp.
Output drivers always enabled.
Power Down disabled by default.
Title
MOPI - Analog to Digital Converter
Size
A4
Number
Revision
Date: 12/7/2012
Sheet 3 of 13
File: F:\00-Mestrado\..\adc.SchDoc
Drawn By: Fernando Sousa - CBPF



Title	**MOPI - Digital to Analog Converter**	
Size: A4	Number	Revision
Date: 12/7/2012		Sheet 4 of 13
File: F:\00-Mestrado\..\dac.SchDoc		Drawn By: Fernando Sousa - CBPF

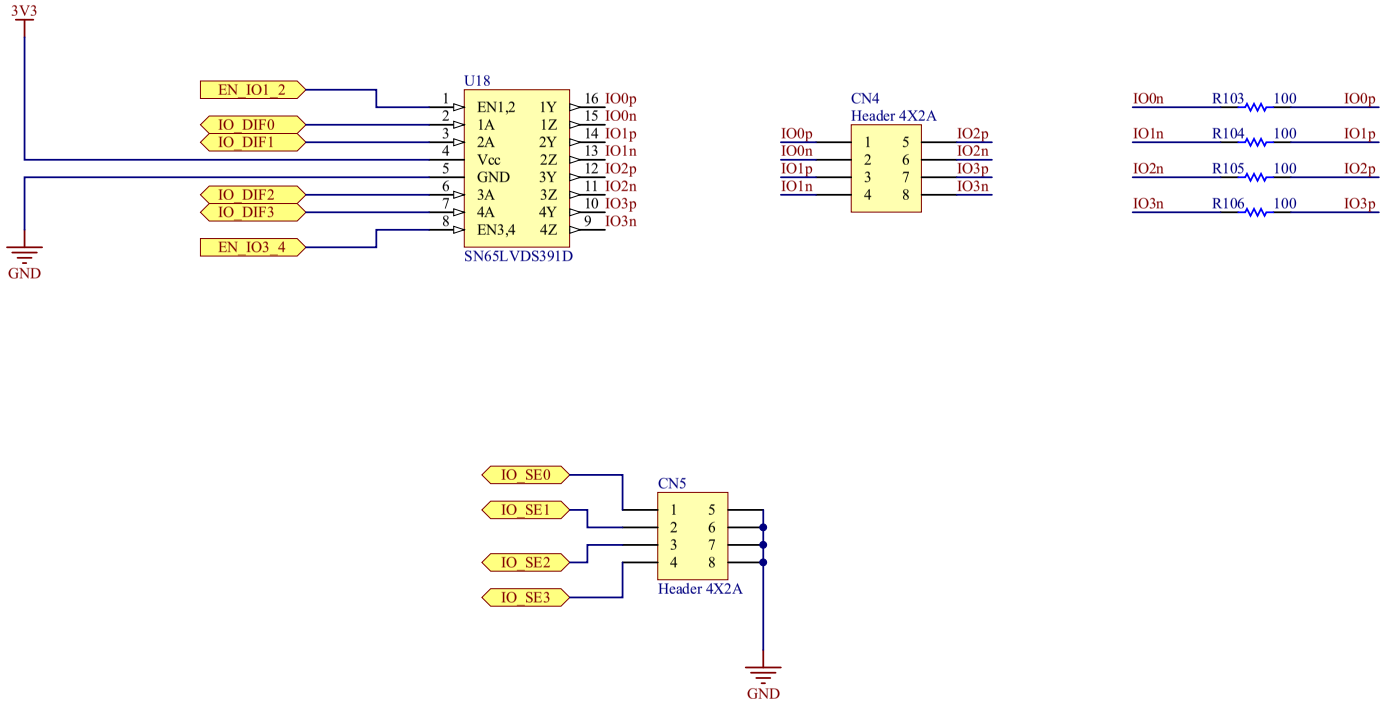


Title	**MOPI - Complete DDS**	
Size: A4	Number	Revision
Date: 12/7/2012		Sheet 5 of 13
File: F:\00-Mestrado\..\dds.SchDoc		Drawn By: Fernando Sousa - CBPF

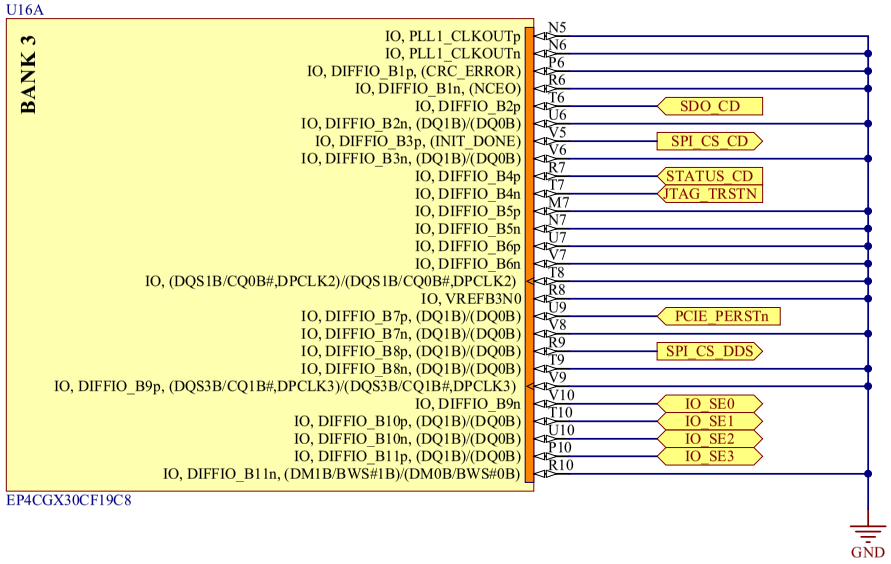


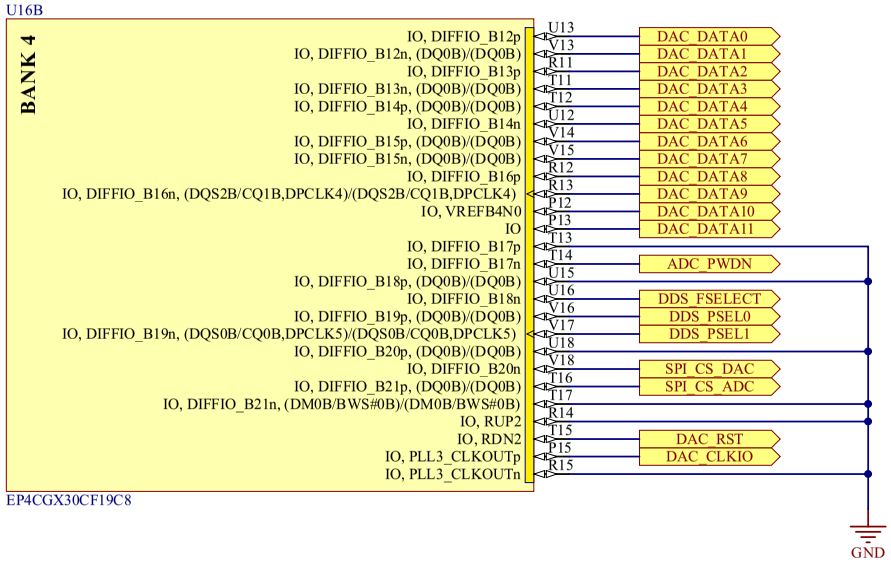
Title	**MOPI - USB Interface**		
Size: A4	Number:		Revision:
Date:	12/7/2012	Sheet 6 of 13	
File:	F:\00-Mestrado\..\usb_interface.SchDoc	Drawn By:	Fernando Sousa - CBPF

As trilhas de U18 para CN4 têm que ser pares diferenciais com Zdiff=100ohms



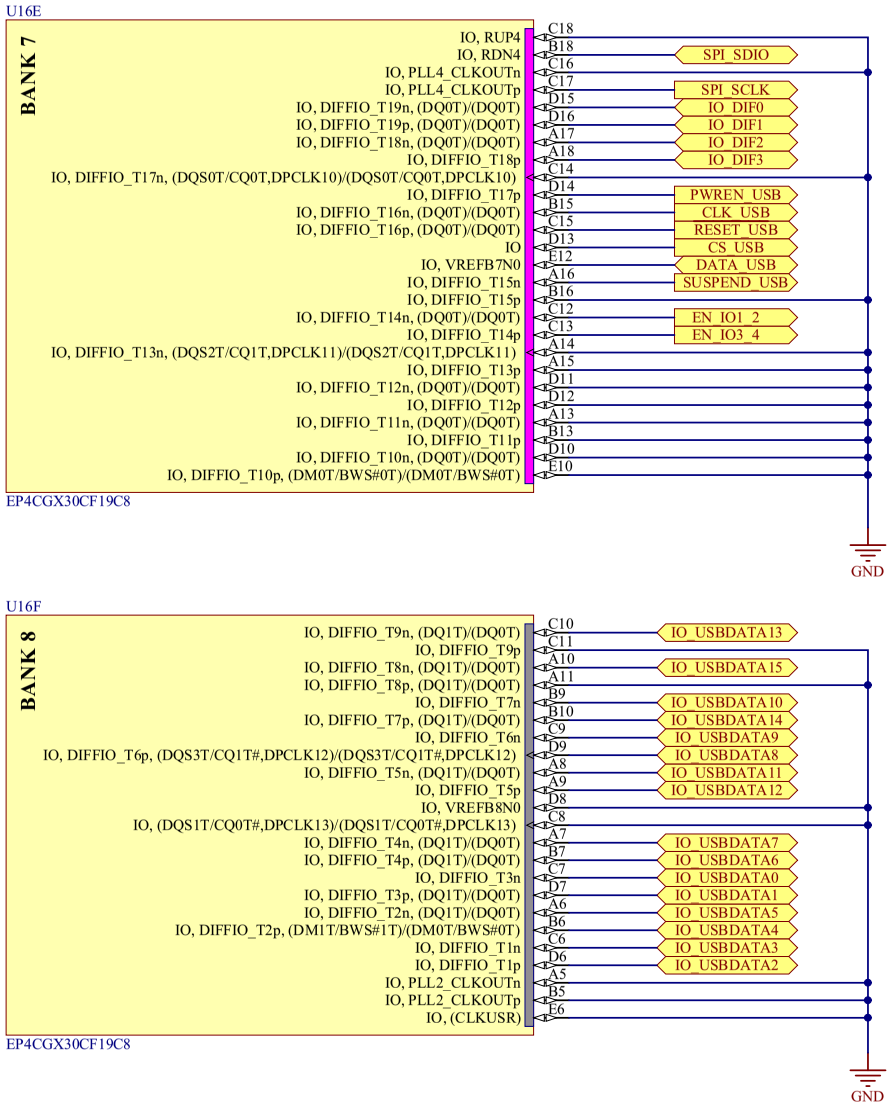
Title	**MOPI - Inputs and Outputs**	
Size: A4	Number:	Revision:
Date: 12/7/2012		Sheet 7 of 13
File: F:\00-Mestrado\..\IOs.SchDoc		Drawn By: Fernando Sousa - CBPF

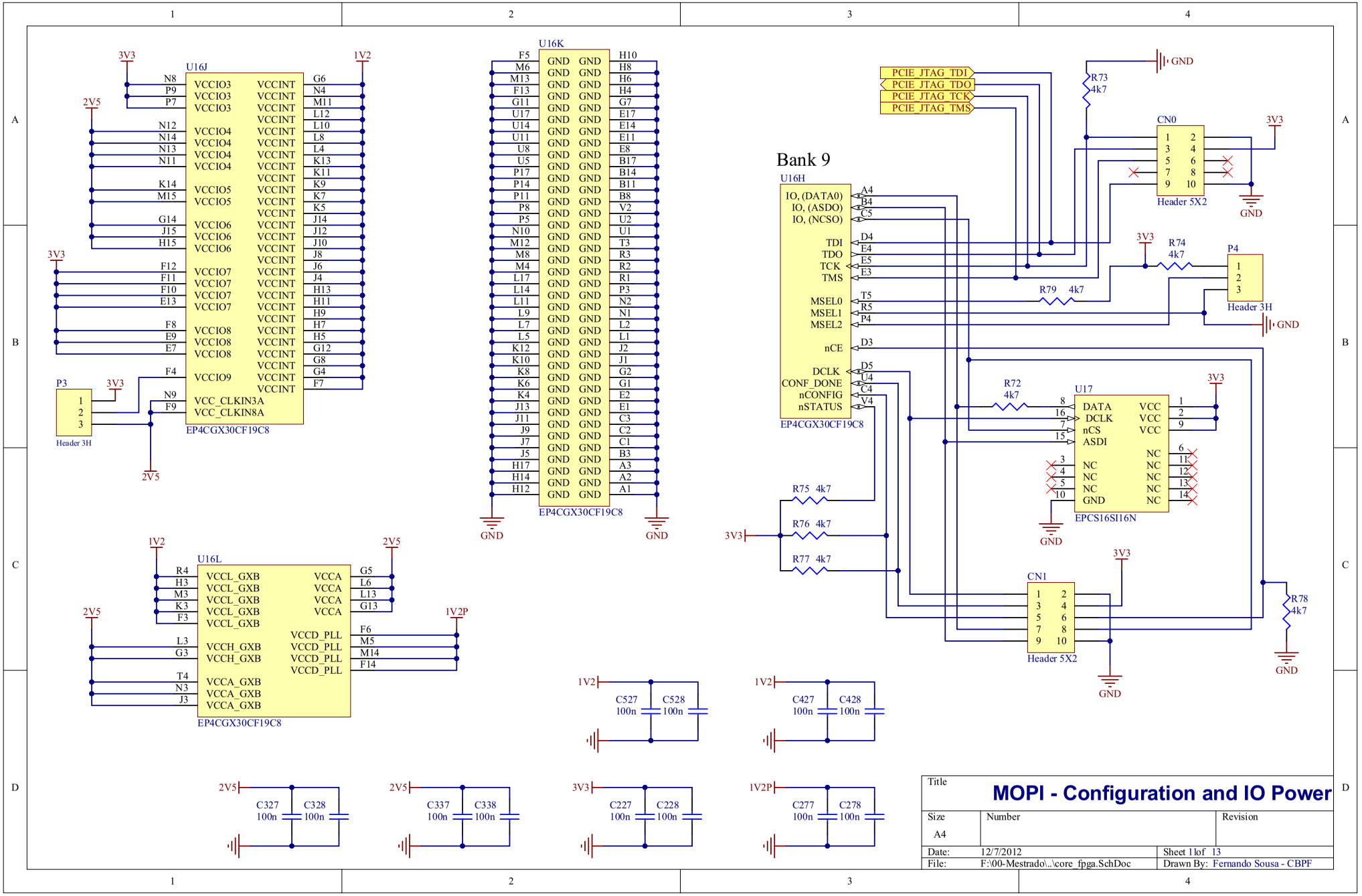


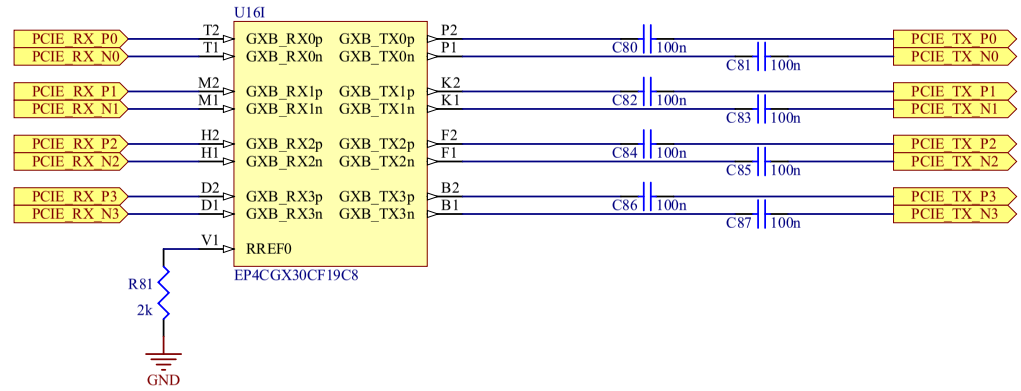


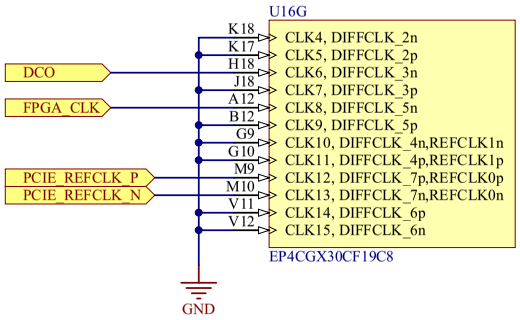
Title	**MOPI - FPGA 1**		
Size: A4	Number:		Revision:
Date:	12/7/2012	Sheet 8 of 13	
File:	F:\00-Mestrado\..\fpga_1.SchDoc	Drawn By:	Fernando Sousa - CBPF


U16C
BANK 5
IO, RUP3
IO, RDN3
IO, DIFFIO_R14n, (DQ1R)/(DQ0R)
IO, DIFFIO_R14p
IO, DIFFIO_R13n, (DQ1R)/(DQ0R)
IO, DIFFIO_R13p, (DQ1R)/(DQ0R)
IO, DIFFIO_R12n, (DQ1R)/(DQ0R)
IO, DIFFIO_R12p, (DQS3R/CQ1R#,DPCLK6)/(DQS3R/CQ1R#,DPCLK6)
IO, VREFB5N0
IO, (DQS1R/CQ0R#,DPCLK7)/(DQS1R/CQ0R#,DPCLK7)
IO, DIFFIO_R11n
IO, DIFFIO_R11p
IO, DIFFIO_R10n, (DQ1R)/(DQ0R)
IO, DIFFIO_R10p, (DQ1R)/(DQ0R)
IO, DIFFIO_R9n, (DQ1R)/(DQ0R)
IO, DIFFIO_R9p, (DQ1R)/(DQ0R)
IO, DIFFIO_R8n, (DQ1R)/(DQ0R)
IO, DIFFIO_R8p, (DM1R/BWS#1R)/(DM0R/BWS#0R)
N15
N16
P16
R16
T18
R17
P18
R18
L16
L15
M17
M16
N18
N17
K16
K15
L18
M18
DATA11p
DATA10p
DATA7p
DATA8p
DATA9p
EP4CGX30CF19C8
GND
U16D
BANK 6
IO, DIFFIO_R7n, (DQ0R)/(DQ0R)
IO, DIFFIO_R7p, (DQ0R)/(DQ0R)
IO, DIFFIO_R6n, (DEV_OE)
IO, DIFFIO_R6p, (DQS0R/CQ0R,DPCLK8)/(DQS0R/CQ0R,DPCLK8)
IO, DIFFIO_R5n, (DQ0R)/(DQ0R)
IO, DIFFIO_R5p, (DQ0R)/(DQ0R)
IO, VREFB6N0
IO
IO, DIFFIO_R4n, (DEV_CLRn)
IO, DIFFIO_R4p, (DQ0R)/(DQ0R)
IO, DIFFIO_R3n, (DQS2R/CQ1R,DPCLK9)/(DQS2R/CQ1R,DPCLK9)
IO, DIFFIO_R3p, (DQ0R)/(DQ0R)
IO, DIFFIO_R2n, (DQ0R)/(DQ0R)
IO, DIFFIO_R2p, (DQ0R)/(DQ0R)
IO, DIFFIO_R1n, (DQ0R)/(DQ0R)
IO, DIFFIO_R1p, (DM0R/BWS#0R)/(DM0R/BWS#0R)
J17
J16
G18
F17
G17
G16
G15
H16
E18
F18
D18
D17
E16
F16
E15
F15
DATA6p
DATA4p
DATA5p
DATA2p
DATA3p
DATA0p
DATA1p
EP4CGX30CF19C8
GND
U48
ADC_DATA11
ADC_DATA11_n
ADC_DATA10
ADC_DATA10_n
ADC_DATA9
ADC_DATA9_n
ADC_DATA8
ADC_DATA8_n
ADC_DATA7
ADC_DATA7_n
ADC_DATA6
ADC_DATA6_n
ADC_DATA5
ADC_DATA5_n
ADC_DATA4
ADC_DATA4_n
ADC_DATA3
ADC_DATA3_n
ADC_DCO
ADC_DCO_n
ADC_DATA2
ADC_DATA2_n
ADC_DATA1
ADC_DATA1_n
ADC_DATA0
ADC_DATA0_n
A1A
A1B
A2A
A2B
A3A
A3B
A4A
A4B
B1A
B1B
B2A
B2B
B3A
B3B
B4A
B4B
C1A
C1B
C2A
C2B
C3A
C3B
C4A
C4B
D1A
D1B
D2A
D2B
D3A
D3B
D4A
D4B
GND
VCC
VCC
GND
ENA
A1Y
A2Y
A3Y
A4Y
ENB
B1Y
B2Y
B3Y
B4Y
GND
VCC
VCC
GND
C1Y
C2Y
C3Y
C4Y
ENC
D1Y
D2Y
D3Y
D4Y
END
GND
VCC
VCC
GND
SN75LVDT386
3V3
R208 4k7
R209 4k7
R210 4k7
R211 4k7
DATA11p
DATA10p
DATA9p
DATA8p
DATA7p
DATA6p
DATA5p
DATA4p
DATA3p
DCO
DATA2p
DATA1p
DATA0p
C627 100n
C628 100n
C633 100n
Title
MOPI - FPGA 2
Size
A4
Number
Revision
Date: 12/7/2012
Sheet 9 of 13
File: F:\00-Mestrado\..\fpga_2.SchDoc
Drawn By: Fernando Sousa - CBPF

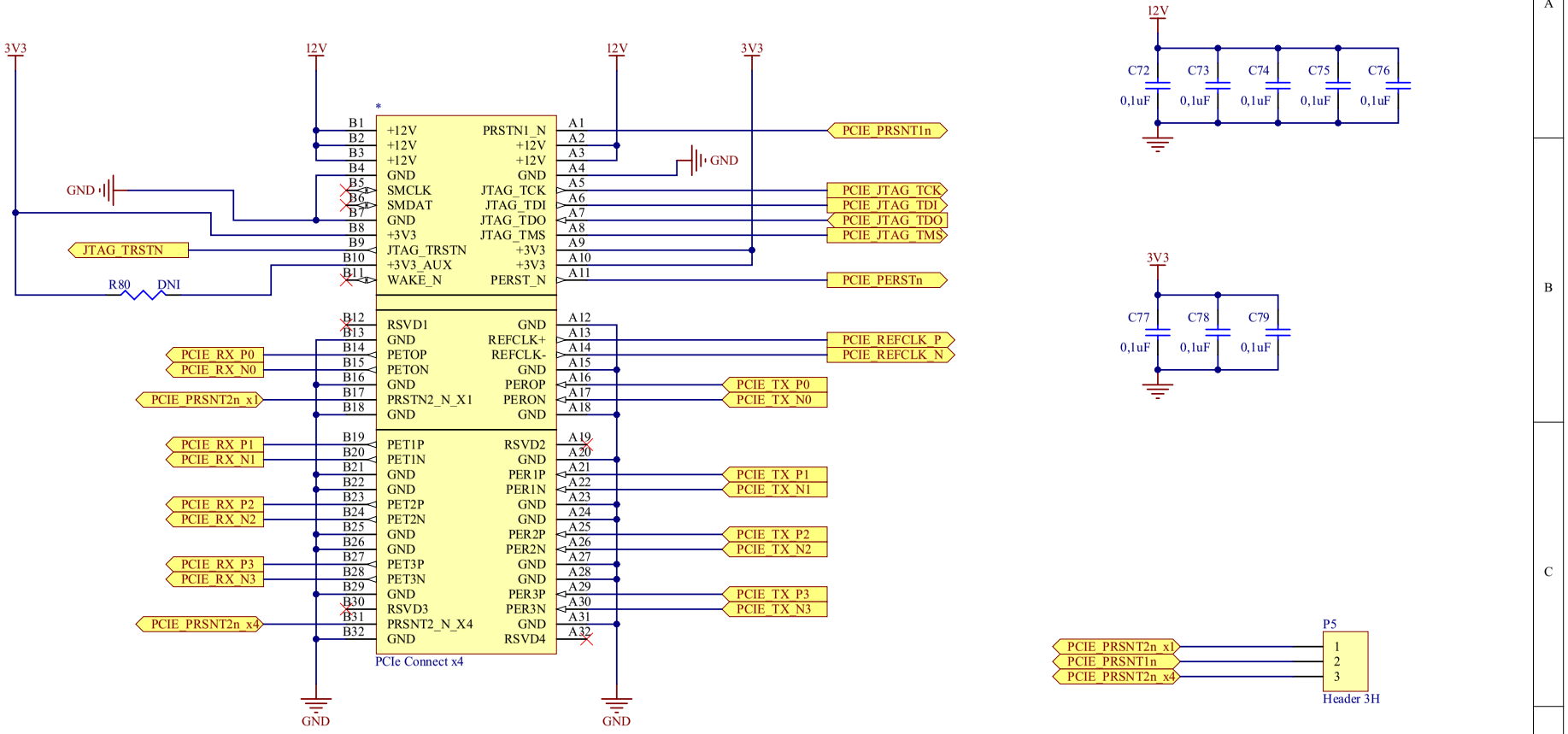

U16E
BANK 7
IO, RUP4
IO, RDN4
IO, PLL4_CLKOUTn
IO, PLL4_CLKOUTp
IO, DIFFIO_T19n, (DQ0T)/(DQ0T)
IO, DIFFIO_T19p, (DQ0T)/(DQ0T)
IO, DIFFIO_T18n, (DQ0T)/(DQ0T)
IO, DIFFIO_T18p
IO, DIFFIO_T17n, (DQS0T/CQ0T,DPCLK10)/(DQS0T/CQ0T,DPCLK10)
IO, DIFFIO_T17p
IO, DIFFIO_T16n, (DQ0T)/(DQ0T)
IO, DIFFIO_T16p, (DQ0T)/(DQ0T)
IO
IO, VREFB7N0
IO, DIFFIO_T15n
IO, DIFFIO_T15p
IO, DIFFIO_T14n, (DQ0T)/(DQ0T)
IO, DIFFIO_T14p
IO, DIFFIO_T13n, (DQS2T/CQ1T,DPCLK11)/(DQS2T/CQ1T,DPCLK11)
IO, DIFFIO_T13p
IO, DIFFIO_T12n, (DQ0T)/(DQ0T)
IO, DIFFIO_T12p
IO, DIFFIO_T11n, (DQ0T)/(DQ0T)
IO, DIFFIO_T11p
IO, DIFFIO_T10n, (DQ0T)/(DQ0T)
IO, DIFFIO_T10p, (DM0T/BWS#0T)/(DM0T/BWS#0T)
C18
B18
C16
C17
D15
D16
A17
A18
C14
D14
B15
C15
D13
E12
A16
B16
C12
C13
A14
A15
D11
D12
A13
B13
D10
E10
SPI_SDIO
SPI_SCLK
IO_DIF0
IO_DIF1
IO_DIF2
IO_DIF3
PWREN_USB
CLK_USB
RESET_USB
CS_USB
DATA_USB
SUSPEND_USB
EN_IO1_2
EN_IO3_4
EP4CGX30CF19C8
GND
U16F
BANK 8
IO, DIFFIO_T9n, (DQ1T)/(DQ0T)
IO, DIFFIO_T9p
IO, DIFFIO_T8n, (DQ1T)/(DQ0T)
IO, DIFFIO_T8p, (DQ1T)/(DQ0T)
IO, DIFFIO_T7n
IO, DIFFIO_T7p, (DQ1T)/(DQ0T)
IO, DIFFIO_T6n
IO, DIFFIO_T6p, (DQS3T/CQ1T#,DPCLK12)/(DQS3T/CQ1T#,DPCLK12)
IO, DIFFIO_T5n, (DQ1T)/(DQ0T)
IO, DIFFIO_T5p
IO, VREFB8N0
IO, (DQS1T/CQ0T#,DPCLK13)/(DQS1T/CQ0T#,DPCLK13)
IO, DIFFIO_T4n, (DQ1T)/(DQ0T)
IO, DIFFIO_T4p, (DQ1T)/(DQ0T)
IO, DIFFIO_T3n
IO, DIFFIO_T3p, (DQ1T)/(DQ0T)
IO, DIFFIO_T2n, (DQ1T)/(DQ0T)
IO, DIFFIO_T2p, (DM1T/BWS#1T)/(DM0T/BWS#0T)
IO, DIFFIO_T1n
IO, DIFFIO_T1p
IO, PLL2_CLKOUTn
IO, PLL2_CLKOUTp
IO, (CLKUSR)
C10
C11
A10
A11
B9
B10
C9
D9
A8
A9
D8
C8
A7
B7
C7
D7
A6
B6
C6
D6
A5
B5
E6
IO_USBDATA13
IO_USBDATA15
IO_USBDATA10
IO_USBDATA14
IO_USBDATA9
IO_USBDATA8
IO_USBDATA11
IO_USBDATA12
IO_USBDATA7
IO_USBDATA6
IO_USBDATA0
IO_USBDATA1
IO_USBDATA5
IO_USBDATA4
IO_USBDATA3
IO_USBDATA2
EP4CGX30CF19C8
GND
Title
MOPI - FPGA 3
Size
A4
Number
Revision
Date: 12/7/2012
Sheet 10 of 13
File: F:\00-Mestrado\..\fpga_3.SchDoc
Drawn By: Fernando Sousa - CBPF


U16J
EP4CGX30CF19C8
U16K
EP4CGX30CF19C8
U16L
EP4CGX30CF19C8
Bank 9
U16H
EP4CGX30CF19C8
PCIE_JTAG_TDI
PCIE_JTAG_TDO
PCIE_JTAG_TCK
PCIE_JTAG_TMS
CN0
Header 5X2
P4
Header 3H
P3
Header 3H
U17
EPCS16SI16N
CN1
Header 5X2
R73 4k7
R74 4k7
R79 4k7
R72 4k7
R75 4k7
R76 4k7
R77 4k7
R78 4k7
C527 100n
C528 100n
C427 100n
C428 100n
C327 100n
C328 100n
C337 100n
C338 100n
C227 100n
C228 100n
C277 100n
C278 100n
3V3
2V5
1V2
1V2P
GND
Title
MOPI - Configuration and IO Power
Size
A4
Number
Revision
Date: 12/7/2012
Sheet 1lof 13
File: F:\00-Mestrado\..\core_fpga.SchDoc
Drawn By: Fernando Sousa - CBPF

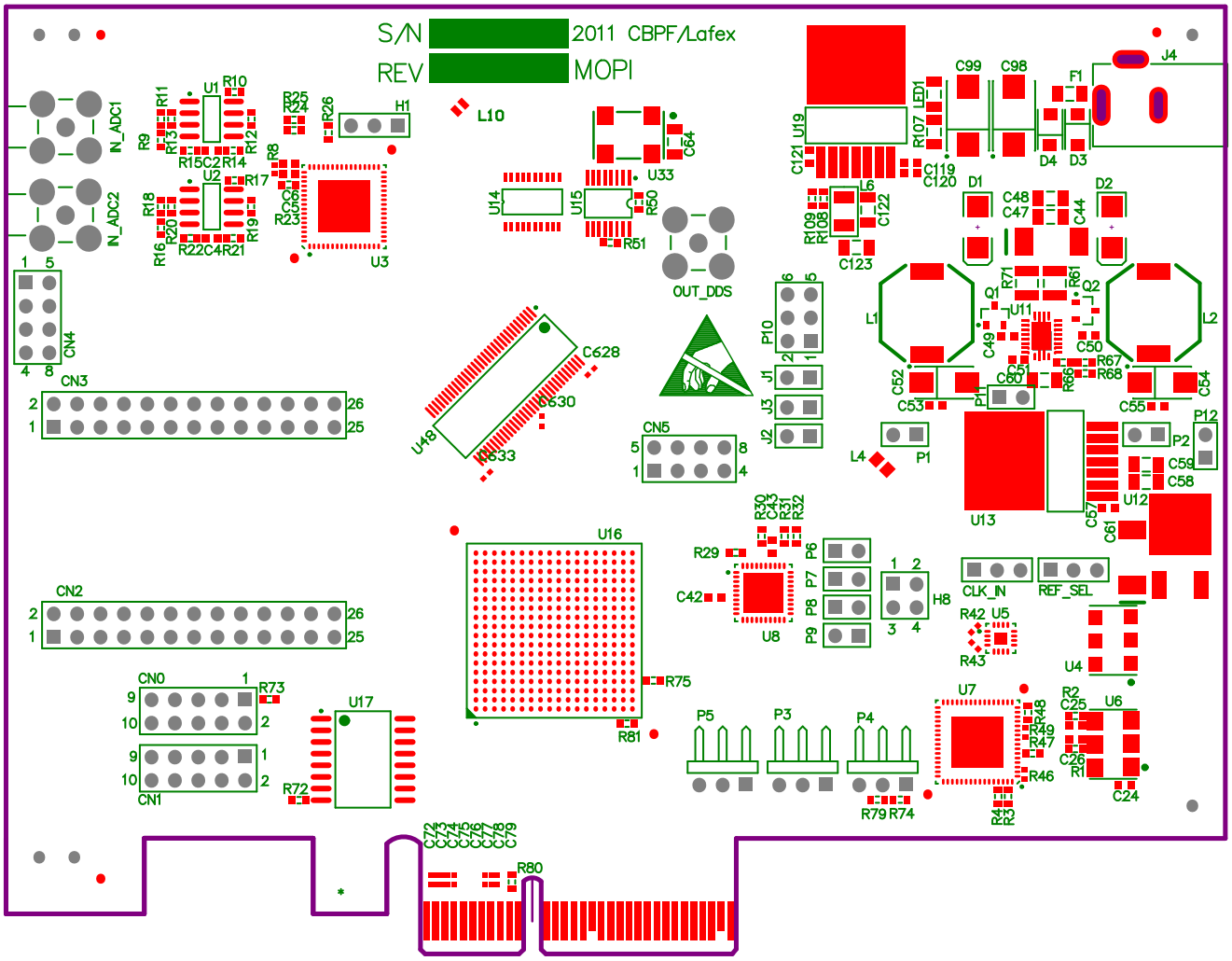




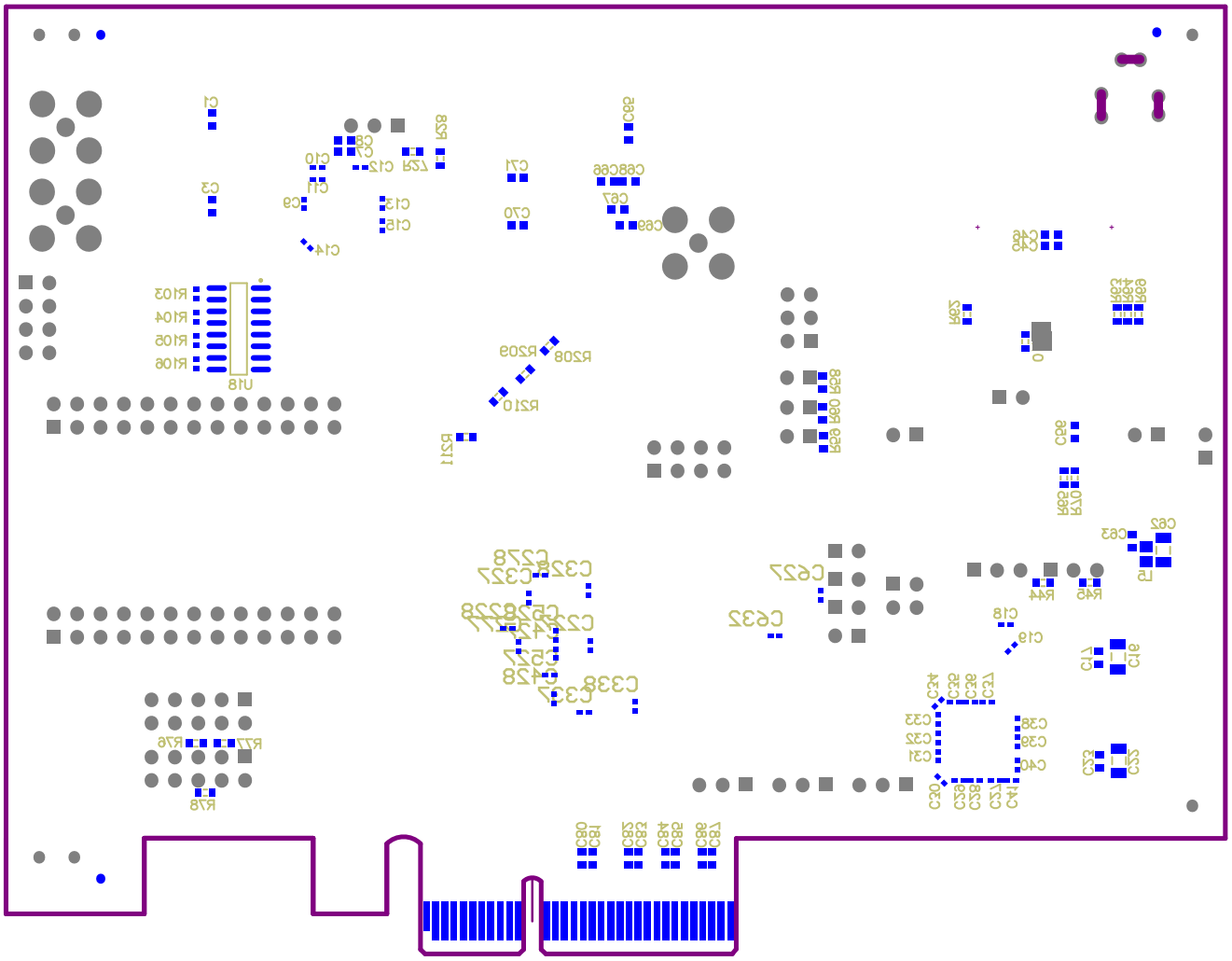
Title	**MOPI - Transceivers PCI Express**	
Size: A4	Number:	Revision:
Date: 12/7/2012	Sheet 12 of 13	
File: F:\00-Mestrado\..\trans_pcie.SchDoc	Drawn By: Fernando Sousa - CBPF	



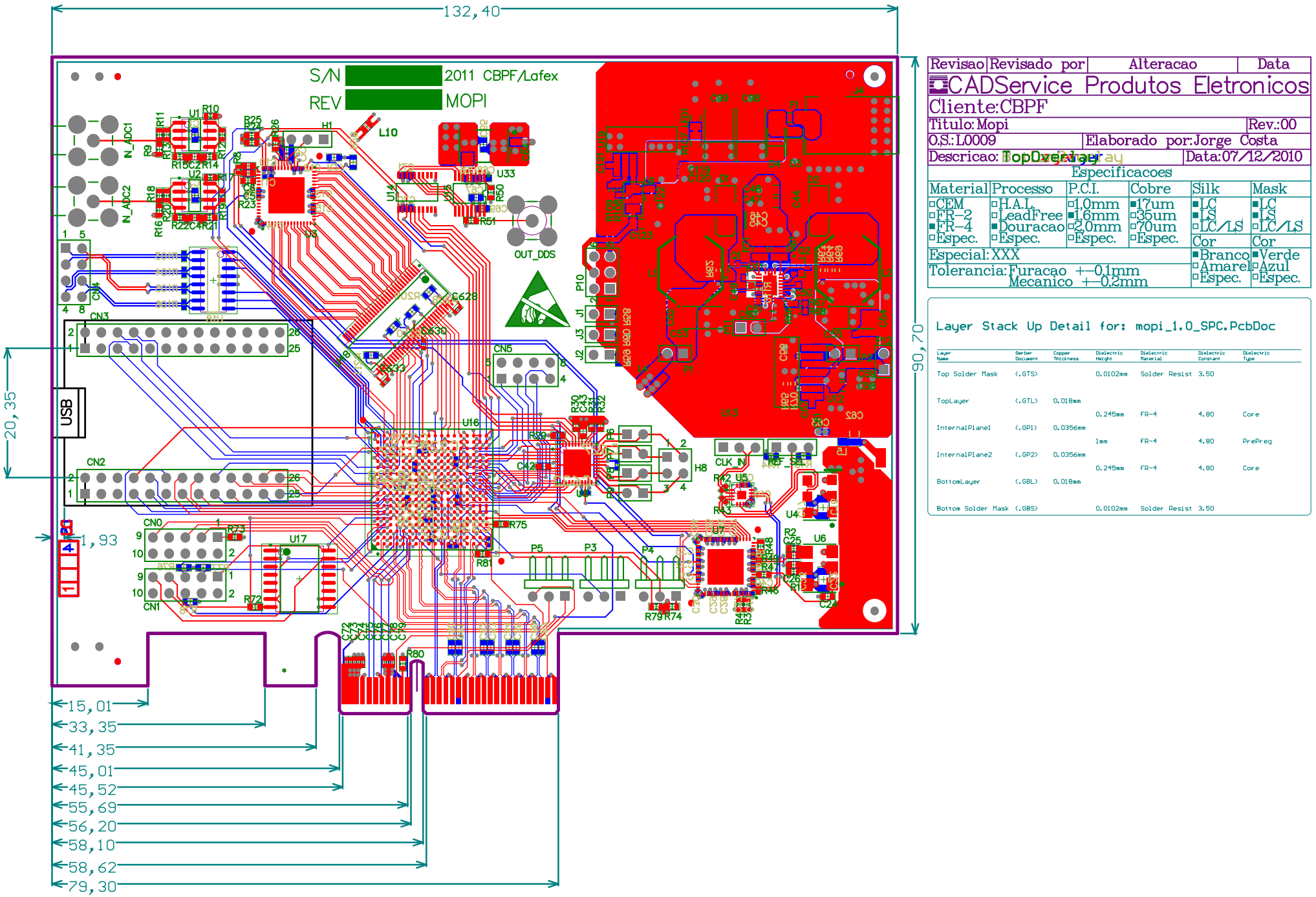
Title	**MOPI - PCI Express Edge Connector**	
Size: A4	Number	Revision
Date: 12/7/2012		Sheet 13 of 13
File: F:\00-Mestrado\..\pcie_conn.SchDoc		Drawn By: Fernando Sousa - CBPF



Revisao	Revisado por	Alteracao	Data
CADService Produtos Eletronicos			
Cliente: CBPF			
Titulo: Mopi			Rev.: 00
O.S.: L0009		Elaborado por: Jorge Costa	
Descricao: TopOverlay			Data: 07/12/2010



Revisao	Revisado por	Alteracao	Data
CADService Produtos Eletronicos			
Cliente: CBPF			
Titulo: Mopi			Rev.: 00
O.S.: L0009		Elaborado por: Jorge Costa	
Descricao: BottomOverlay			Data: 07/12/2010



Revisao	Revisado por	Alteracao	Data

CADService Produtos Eletronicos

Cliente: CBPF

Titulo: Mopi — Rev.: 00

O.S.: L0009 — Elaborado por: Jorge Costa

Descricao: TopOverlay — Data: 07/12/2010

Especificacoes

Material	Processo	P.C.I.	Cobre	Silk	Mask
☐ CEM	☐ H.A.L.	☐ 1,0mm	■ 17um	■ LC	■ LC
☐ FR-2	☐ LeadFree	■ 1,6mm	☐ 35um	■ LS	■ LS
■ FR-4	■ Douracao	☐ 2,0mm	☐ 70um	☐ LC/LS	☐ LC/LS
☐ Espec.	☐ Espec.	☐ Espec.	☐ Espec.	Cor	Cor

Especial: XXX — ■ Branco ☐ Amarel ☐ Espec. — ■ Verde ☐ Azul ☐ Espec.

Tolerancia: Furacao +-0,1mm
Mecanico +-0,2mm

Layer Stack Up Detail for: mopi_1.0_SPC.PcbDoc

Layer Name	Gerber Document	Copper Thickness	Dielectric Height	Dielectric Material	Dielectric Constant	Dielectric Type
Top Solder Mask	(.GTS)		0.0102mm	Solder Resist	3.50	
TopLayer	(.GTL)	0.018mm				
			0.245mm	FR-4	4.80	Core
InternalPlane1	(.GP1)	0.0356mm				
			1mm	FR-4	4.80	PrePreg
InternalPlane2	(.GP2)	0.0356mm				
			0.245mm	FR-4	4.80	Core
BottomLayer	(.GBL)	0.018mm				
Bottom Solder Mask	(.GBS)		0.0102mm	Solder Resist	3.50	

APÊNDICE B -- Programa do NIOS II

```
1 //   --------------------------------------------------------------------
2 //            BEGIN, batista@cbpf.br / fsousa@.cbpf.br
3 //   --------------------------------------------------------------------
4 /*   Begin of comments:
5
6    Centro Brasileiro de Pesquisas Fisicas.
7    20 de janeiro de 2011.
8    Pablo Diniz Batista / Fernando Sousa.
9    batista@cbpf.br    / fsousa@cbpf.br.
10
11   Objetivos:
12
13   1 - Integração do NIOS com o PC via PCI Express.
14     a) Isso é realizado compartilhando uma memória RAM interna.
15     b) Futuramente isso pode ser testando com uma memória externa.
16     c) Controla o fluxo de dados de uma memória FIFO externa ao SOPC para
17        o PC via PCI Express.
18    End of comments.
19 */
20
21 #define JTAG_COMPILER 0
22
23 #include "sys/alt_stdio.h"
24 #include "stdio.h"
25 #include "altera_avalon_pio_regs.h"
26 #include "altera_avalon_spi.h"
27
28    //Endereço dos periférico construído internamente.
29
30 #define PIO_DADOS_FIFO1         0x00045000
31 #define PIO_FIFO_FULL           0x00045060
32 #define PIO_RDCLK               0x00045040
33 #define PIO_RDCLK2              0x00044120
34 #define PIO_FIFO_RDUSEDW        0x00045080
35 #define PIO_FIFO_RESET          0x000450A0
36 #define PIO_FIFO_EMPTY          0x000450C0
37 #define PIO_MUX_CLK             0x000450E0
38 #define PIO_TDCFIFO_EMPTY       0x00044000
39 #define PIO_TDCFIFO_FULL        0x00044020
40 #define PIO_TDCFIFO_RDREQ       0x00044040
41 #define PIO_TDCFIFO_CLK         0x00044060
42 #define PIO_TDCFIFO_Q           0x00044080
43 #define PIO_DADOS_FIFO2         0x000440A0
44 #define PIO_FIFO_FULL2          0x000440C0
45 #define PIO_FIFO_RDUSEDW2       0x000440E0
46 #define PIO_FIFO_EMPTY2         0x00044100
47 #define PIO_RD_FIFO1            0x00044180
48 #define PIO_RD_FIFO2            0x000441A0
49 #define SPI_DDS                 0x00044140
50
51 #define MEMDUAL_BASE            0x00000000
52 #define PCIE_COMPILER_0_BASE    0x00040000
53
54 #define OFFSET_INST_NIOS        0x0000
55 #define OFFSET_INST_PCIE        0x000A
56 #define OFFSET_DATA             0x00FF
```

```
 57
 58#define MAX_MEMORIA             0xFFFF
 59
 60unsigned int *mem;
 61
 62//-----------------------------------------------
 63//   Definições do Protocolo de Comunicação:
 64//-----------------------------------------------
 65
 66#define ACK   0xAA
 67#define NACK  0xCC
 68#define WAIT  0xEE
 69
 70#define ADDRESS_NIOS_STATUS    OFFSET_INST_NIOS + 0 //0xAA + 0
 71#define ADDRESS_NIOS_COMMAND   OFFSET_INST_NIOS + 1 //0x00 + 1
 72#define ADDRESS_NIOS_SIZE      OFFSET_INST_NIOS + 2
 73#define ADDRESS_NIOS_CHEKSUM   OFFSET_INST_NIOS + 3
 74
 75#define ADDRESS_PCIE_STATUS    OFFSET_INST_PCIE + 0
 76#define ADDRESS_PCIE_COMMAND   OFFSET_INST_PCIE + 1
 77#define ADDRESS_PCIE_SIZE      OFFSET_INST_PCIE + 2
 78#define ADDRESS_PCIE_CHEKSUM   OFFSET_INST_PCIE + 3
 79
 80////////////////////////////////////////////////////////////
 81
 82char RunCommand_EXE01(unsigned char option);
 83char RunCommand_EXE02(void);
 84char RunCommand_EXE03(void);
 85char RunCommand_EXE04(void);
 86char RunCommand_EXE05(void);
 87char RunCommand_EXE06(void);
 88char RunCommand_EXE07(void);
 89int  CheckCommand(void);
 90void Testa_Memoria(void);
 91void ProcessMenu (void);
 92void delay(int a,int b);
 93
 94alt_u8 Send_Byte_SPI(alt_u8 byte1, alt_u8 byte2);
 95
 96/************************************************
 97 ************************************************
 98   ||\\     //||      //\\      ||  ||\\   ||
 99   || \\   // ||     //  \\     ||  || \\  ||
100   ||  \\ //  ||    //    \\    ||  ||  \\ ||
101   ||   \\/   ||   //======\\   ||  ||   \\||
102   ||         ||  //        \\  ||  ||    \||
103 ************************************************
104 ************************************************/
105
106//-------------------------------------------------------------------------------
107
108int  Config_DDS(unsigned char a,unsigned char b,unsigned char c,unsigned char d)
109{
110     int i;
111     const int M = 1000;
112     int delay,tempo;
```

```
113     unsigned char  data[4];
114     alt_u8 byte1[] = {0x30,0x21,0x32,0x23};
115
116     data[0] = a;
117     data[1] = b;
118     data[2] = c;
119     data[3] = d;
120
121     // Send_Byte_SPI(0xF8,0x00); LIGA DDS
122     //-------------------------
123     delay = 0;
124     while( delay < M )
125     {
126            tempo = 0;
127            while ( tempo < M )
128                   tempo++;
129            delay++;
130     }
131     //-------------------------
132     for ( i = 0; i < 4 ; i  ++ )
133     {
134         Send_Byte_SPI(byte1[i],data[i]);
135
136         delay = 0;
137         while( delay < M )
138         {
139                tempo = 0;
140                while ( tempo < M )
141                       tempo++;
142                delay++;
143         }
144     }
145     Send_Byte_SPI(0x90,   0x00);
146     //-------------------------
147     delay = 0;
148     while( delay < M )
149     {
150            tempo = 0;
151            while ( tempo < M )
152                   tempo++;
153            delay++;
154     }
155     //-------------------------
156    //  Send_Byte_SPI(0xC0,   0x00); DESLIGA DDS
157    return 1;
158}
159//---------------------------------------------------------------------------
160//---------------------------------------------------------------------------
161
162void Ajusta_Fre_DDS(unsigned long int frequency)
163{
164     unsigned char a,b,c,d;
165     unsigned long int temp = 0;
166
167     temp = frequency *(0xFFFFFFFF/ 20000000.0);
168     d = ((temp & 0xFF000000) >> 24);
```

```
169     c = ((temp & 0x00FF0000) >> 16);
170     b = ((temp & 0x0000FF00) >> 8);
171     a = ((temp & 0x000000FF));
172
173     Config_DDS(a,b,c,d);
174 }
175
176 //*************************************************************************************
    ***********
177 //*************************************************************************************
    ***********
178
179 #define NIOS_SPI_BASE 0x00044140
180
181 alt_u8 Send_Byte_SPI(unsigned char byte1, unsigned char byte2)
182 {
183
184     alt_u32 base  = NIOS_SPI_BASE;
185     alt_u32 slave = 0x00;
186     alt_u32 write_length = 2;
187     alt_u8  write_data[2];
188     alt_u32 read_length  = 0;
189     alt_u8  read_data = 0x00;
190
191
192     write_data[0] = byte1;
193     write_data[1] = byte2;
194     alt_u32 flags = 0;       // if flags == 0, force chipselect
195     alt_avalon_spi_command(base, slave,
    write_length,&write_data[0],read_length,&read_data,flags);
196     return read_data;
197 }
198
199
200
201 //*************************************************************************************
    ***********
202 //*************************************************************************************
    ***********
203
204 int main(void)
205 {
206     #ifdef JTAG_COMPILER
207         alt_putstr("Centro Brasileiro de Pesquisas Físicas\n");
208         alt_putstr("        Version 1.0 10/10/2011 \n");
209         alt_putstr("<batista@cbpf.br> / <fsousa@cbpf.br>\n");
210         alt_putstr("\n");
211         alt_putstr("\n");
212         alt_putstr("1 - Iniciando!\n");
213     #endif
214
215     //Teste_DDS();
216     //Teste_PIO_SPI();
217
218     mem = (unsigned int*)MEMDUAL_BASE;
219
```

```
220     mem[ADDRESS_PCIE_STATUS] = WAIT;  // WAIT para o PCIExpress.
221
222     //Testa_Memoria();
223
224     mem[ADDRESS_PCIE_STATUS] = ACK;   // Pronto Para receber comando do PCI
225     mem[ADDRESS_NIOS_STATUS] = NACK;  // Nenhum comando para o NIOS.
226
227     #ifdef JTAG_COMPILER
228         printf("NIOS\n");
229         printf("0x%x 0x%x \n",ADDRESS_NIOS_STATUS , mem[ADDRESS_NIOS_STATUS]   );
230         printf("0x%x 0x%x \n",ADDRESS_NIOS_COMMAND, mem[ADDRESS_NIOS_COMMAND]  );
231         printf("0x%x 0x%x \n",ADDRESS_NIOS_SIZE   , mem[ADDRESS_NIOS_SIZE]     );
232         printf("0x%x 0x%x \n",ADDRESS_NIOS_CHEKSUM, mem[ADDRESS_NIOS_CHEKSUM]  );
233
234         printf("PCIE\n");
235
236         printf("0x%x 0x%x \n",ADDRESS_PCIE_STATUS , mem[ADDRESS_PCIE_STATUS]  );
237         printf("0x%x 0x%x \n",ADDRESS_PCIE_COMMAND, mem[ADDRESS_PCIE_COMMAND]  );
238         printf("0x%x 0x%x \n",ADDRESS_PCIE_SIZE   , mem[ADDRESS_PCIE_SIZE]  );
239         printf("0x%x 0x%x \n",ADDRESS_PCIE_CHEKSUM, mem[ADDRESS_PCIE_CHEKSUM]  );
240     #endif
241
242
243     Ajusta_Fre_DDS(1000);
244
245     RunCommand_EXE06();
246
247     IOWR_ALTERA_AVALON_PIO_DATA(PIO_MUX_CLK,0);
248     do
249         {
250             ProcessMenu ();
251             delay(0x00FF,0x00FF);
252         }
253     while(1);
254 }
255
256
257 ////////////////////////////////////////////////////
258
259 int  CheckCommand(void)
260 {
261      unsigned int data;
262
263      data = mem[ADDRESS_NIOS_STATUS];
264
265      if (  data == ACK )
266      {
267         mem[ADDRESS_NIOS_STATUS] =  NACK; // Limpa Comando
268         mem[ADDRESS_PCIE_STATUS] =  WAIT; // COMANDO VAI SER PROCESSADO
269         return 1;
270      }
271      else
272      {
273         return 0;
274      }
275
```

```
276}
277
278//////////////////////////////////////////////////////
279
280void ProcessMenu (void)
281{
282    unsigned char byte, option ;
283    char r = 0;
284
285    if ( CheckCommand() == 1 )
286    {
287         byte = mem[ADDRESS_NIOS_COMMAND];
288
289         switch ( byte )
290         {
291              case 1:
292                        #ifdef JTAG_COMPILER
293                          alt_putstr("Executando comando ... 01 -- FIFO 1");
294                        #endif
295                        r = RunCommand_EXE01(1);
296                        break;
297              case 2:
298                        #ifdef JTAG_COMPILER
299                          alt_putstr("Executando comando ... 02 -- FIFO 2");
300                        #endif
301                        r = RunCommand_EXE01(0);
302                        break;
303              case 3:
304                        #ifdef JTAG_COMPILER
305                          alt_putstr("Executando comando ... 03 - F Amostragem");
306                        #endif
307                        r = RunCommand_EXE03();
308                        break;
309              case 4:
310                        #ifdef JTAG_COMPILER
311                          alt_putstr("Executando comando ... 04 - TDC");
312                        #endif
313                        r = RunCommand_EXE04();
314                        break;
315              case 5:
316                        #ifdef JTAG_COMPILER
317                          alt_putstr("Executando comando ... 05 - Ajsut Freq");
318                        #endif
319                        r = RunCommand_EXE05();
320                        break;
321              case 6:
322                        #ifdef JTAG_COMPILER
323                          alt_putstr("Executando comando ... 06 - LIGA DDS");
324                        #endif
325                        r = RunCommand_EXE06();
326                        break;
327              case 7:
328                        #ifdef JTAG_COMPILER
329                          alt_putstr("Executando comando ... 07 - DESLIGA DDS");
330                        #endif
331                        r = RunCommand_EXE07();
```

```
332                         break;
333
334         }
335         if ( r == 1 )
336         {
337            #ifdef JTAG_COMPILER
338             alt_putstr("ACK\n\n");
339            #endif
340            mem[ADDRESS_PCIE_COMMAND] = ACK;
341         }
342         else //RETORNA -1
343         {
344            #ifdef JTAG_COMPILER
345             alt_putstr("NACK\n\n");
346            #endif
347            mem[ADDRESS_PCIE_COMMAND] = NACK;
348         }
349         mem[ADDRESS_PCIE_STATUS] =  ACK;
350     }
351}
352
353//************************************************
354/* Executa o comando 01
355
356  1 - Lê 64KB da FIFO.
357  2 - Transfere para MEM_DUAL.
358 */
359//************************************************
360
361void CLK_FIFO(unsigned int RDCLK)
362{
363     char i;
364
365     for ( i = 0; i  < 6 ; i  ++ )
366     {
367         IOWR_ALTERA_AVALON_PIO_DATA(RDCLK,1);
368         delay(0x00FF,0x00FF);
369         IOWR_ALTERA_AVALON_PIO_DATA(RDCLK,0);
370         delay(0x00FF,0x00FF);
371     }
372}
373
374char RunCommand_EXE01(unsigned char option)
375{
376     int             i,size;
377     unsigned int    data_port = 0x00;
378     unsigned int   *ptr;
379     unsigned char   b;
380     unsigned char   checksum;
381     unsigned int    address;
382
383     size     = mem[ADDRESS_NIOS_SIZE];
384     address  = OFFSET_DATA;
385
386     unsigned int FIFO_FULL  ;
387     unsigned int FIFO_EMPTY ;
```

```
388     unsigned int RDCLK      ;
389     unsigned int FIFO_DADOS ;
390     unsigned int FIFO_READ ;
391
392     //---------------------------------------
393     if ( option == 1 )
394     {
395         // Ler FIFO 1
396         //-------------------------------
397         FIFO_FULL  = PIO_FIFO_FULL;
398         FIFO_EMPTY = PIO_FIFO_EMPTY;
399         RDCLK      = PIO_RDCLK;
400         FIFO_DADOS = PIO_DADOS_FIFO1;
401         FIFO_READ  = PIO_RD_FIFO1;
402     }
403     else
404     {
405         // Ler FIFO 2
406         //-------------------------------
407         FIFO_FULL  = PIO_FIFO_FULL2;
408         FIFO_EMPTY = PIO_FIFO_EMPTY2;
409         RDCLK      = PIO_RDCLK2;
410         FIFO_DADOS = PIO_DADOS_FIFO2;
411         FIFO_READ  = PIO_RD_FIFO2;
412     }
413     //---------------------------------------
414     if ( size <= MAX_MEMORIA )
415     {
416         IOWR_ALTERA_AVALON_PIO_DATA(FIFO_READ,0);
417         delay(0x00FF,0x00FF);
418
419         CLK_FIFO(RDCLK);
420
421         if (IORD_ALTERA_AVALON_PIO_DATA(FIFO_FULL) == 1)
422         {
423             CLK_FIFO(RDCLK);
424             if (IORD_ALTERA_AVALON_PIO_DATA(FIFO_EMPTY) == 0)
425             {
426                 IOWR_ALTERA_AVALON_PIO_DATA(FIFO_READ,1);
427                 delay(0x00FF,0x00FF);
428                 checksum = 0;
429                 while (IORD_ALTERA_AVALON_PIO_DATA(FIFO_EMPTY) != 1)
430                 {
431                        IOWR_ALTERA_AVALON_PIO_DATA(RDCLK,1);
432                        IOWR_ALTERA_AVALON_PIO_DATA(RDCLK,0);
433                        data_port = IORD_ALTERA_AVALON_PIO_DATA(FIFO_DADOS);
434                        mem[address++] = data_port;
435                        checksum      += data_port;
436                 };
437             }
438         }
439         else
440         {
441             return -1;
442         }
443         // Informa ao PCI o número de bytes disponíveis
```

```
444         mem[ADDRESS_PCIE_SIZE]    = size;
445
446         // Informa ao PCI o checksum dos bytes
447         mem[ADDRESS_PCIE_CHEKSUM] = checksum;
448         return 1;
449     }
450     return -1;
451 }
452 //   ---------------------------------------------------------------------
453 //   ---------------------------------------------------------------------
454 char RunCommand_EXE03(void)
455 {
456      unsigned int scale;
457
458      scale = mem[OFFSET_DATA];
459      if ( scale >= 0 && scale < 7 )
460      {
461           IOWR_ALTERA_AVALON_PIO_DATA(PIO_MUX_CLK,scale);
462           return 1;
463      }
464      else
465           return -1;
466 }
467 //   ---------------------------------------------------------------------
468 //   ---------------------------------------------------------------------
469
470 void CLK_TDCFIFO(int time, int n)
471 {
472      int i;
473
474      for ( i = 0; i < n; i ++)
475      {
476            IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,1);
477            delay(time,time);
478            IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,0);
479            delay(time,time);
480      }
481 }
482
483 char RunCommand_EXE04(void)
484 {
485     int             i, size;
486     char            status;
487     unsigned int    *ptr, address, contador=0, cont_bloco=0, t;
488     unsigned char   b, checksum, cheksum;
489
490     size     = mem[ADDRESS_NIOS_SIZE];
491     contador = 0;
492     address  = OFFSET_DATA;
493
494     status = -1;
495     if ( size <= MAX_MEMORIA )
496     {
497             IOWR_ALTERA_AVALON_PIO_DATA(PIO_FIFO_RESET,0);  delay(0x00FF,0x00FF);
498             IOWR_ALTERA_AVALON_PIO_DATA(PIO_FIFO_RESET,1);  delay(0x00FF,0x00FF);
499             IOWR_ALTERA_AVALON_PIO_DATA(PIO_FIFO_RESET,0);  delay(0x00FF,0x00FF);
```

```
500
501             #ifdef JTAG_COMPILER
502                 printf("\n A quatidade de pulsos é: ....\n\n",size);
503             #endif
504
505             if (IORD_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_FULL) == 1)//FIFO CHEIA
506             {
507                 #ifdef JTAG_COMPILER
508                     alt_putstr("Fifo Full !!!!\n");
509                 #endif
510                 //CLK_TDCFIFO(0x0FF,3);
511
512                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,1);
513                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,0);
514                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,1);
515                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,0);
516                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,1);
517                 IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,0);
518
519                 if (IORD_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_EMPTY) == 0)
520                 {
521                       #ifdef JTAG_COMPILER
522                         alt_putstr("Rdempty = 0 !!!!\n");
523                       #endif
524                       do{
525                            IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,1);
526                            IOWR_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_CLK,0);
527
528                            mem[address] = IORD_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_Q);
529                            checksum+= mem[address++];
530                       } while (IORD_ALTERA_AVALON_PIO_DATA(PIO_TDCFIFO_EMPTY) != 1);
531
532                       #ifdef JTAG_COMPILER
533                          alt_putstr("Rdempty = 1 !!!!\n");
534                       #endif
535                       status = 1;
536                  }
537                  mem[ADDRESS_PCIE_SIZE]    = size;
538                  mem[ADDRESS_PCIE_CHEKSUM] = cheksum;
539                  #ifdef JTAG_COMPILER
540                     for ( i = 0; i < 10; i++)
541                        printf("%d\n",mem[OFFSET_DATA + i] );
542                  #endif
543           }
544     }
545     return status;
546}
547
548//---------------------------------------------------------------------
549//---------------------------------------------------------------------
550//Altera a frequencia do DDS
551//
552//
553char RunCommand_EXE05(void)
554{
555      unsigned int f = mem[OFFSET_DATA];
```

```
556
557     Ajusta_Fre_DDS(f);
558
559     return 1;
560}
561//---------------------------------------------------------------------
562//---------------------------------------------------------------------
563//Altera a frequencia do DDS
564//
565//
566char RunCommand_EXE06(void)
567{
568     Send_Byte_SPI(0xC0,0x00);
569
570     return 1;
571}
572//---------------------------------------------------------------------
573//---------------------------------------------------------------------
574//Altera a frequencia do DDS
575//
576//
577char RunCommand_EXE07(void)
578{
579     Send_Byte_SPI(0xF8,0x00);
580
581     return 1;
582}
583//   ---------------------------------------------------------------------
584//   ---------------------------------------------------------------------
585void delay(int a,int b)
586{
587    int i,j;
588
589    for ( i = 0; i < a; i ++)
590    for ( j = 0; j < b; j++);
591
592}
593//   ---------------------------------------------------------------------
594//   ---------------------------------------------------------------------
595void Testa_Memoria(void)
596{
597    int  i,cont_ram=0,cont_comp=0;
598    unsigned char data,cont;
599
600    // ---------------------------------------------
601    // Primeiro passo:
602    // Realiza um teste na Memória compartilhada.
603    // ---------------------------------------------
604
605    #ifdef JTAG_COMPILER
606        alt_putstr("2 - Teste da memória RAM compartilhada .... ");
607    #endif
608
609        cont = 0;
610        for ( i = 0; i < 100+OFFSET_DATA; i++)
611        {
```

```
612             mem[i] = (unsigned char) cont++;
613         }
614         cont = 0;
615 
616         for ( i = 0; i < 100+OFFSET_DATA; i++)
617         {
618             data = (unsigned char) mem[i];
619             if ( data != cont++ )
620                  cont_comp++;
621             mem[i]=0;
622         }
623 
624         #ifdef JTAG_COMPILER
625             if ( cont_comp == 0)
626                 alt_putstr("OK\n");
627             else
628                 printf("bytes %d errados\n\n",cont_comp);
629         #endif
630 
631         //------------------------------------------------------
632         // PICExpress pode enviar um comando.
633         //------------------------------------------------------
634 
635     #ifdef JTAG_COMPILER
636         alt_putstr("3 - Esperando por comandos via PCI Express. \n\n");
637     #endif
638 
639         //------------------------------------------------------
640         //  Libera a comunicação com o PCIExpress
641         // Ou seja, coloca um ACK para que o PCI possa ler.
642         //------------------------------------------------------
643 }
644 //   ---------------------------------------------------------------------
645 //   End of file, batista@cbpf.br / fsousa@cbpf.br
646 //   ---------------------------------------------------------------------
647 
648 
649 
650 
```

ANEXO A -- Aplicativo de Controle

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

//  Centro Brasileiro de Pesquisas Físicas
//  Rio de janeiro, 2011
//  LHCb

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------


//===========================================================================

//#######################################
//      Conversor Analógico Digital
//#######################################

void ADC(void)
{
        int           i,cont,n,status,r, option;
        const int     size_inst = 10*TIPO_MEMORIA;
        unsigned char buffer_reg[size_inst];
        char          path[1000],nome[255];
//***************************************************************************

    Initialization();
    Full_Memo_Data(fd,0);
    Show_Register_Map (fd);

    printf("\nDigite a opção para frequência de amostragem ADC\n");
    scanf ("%d",&option);
    r = Send_to_NIOS_Command_Exe02(fd,option);
    printf("r = %d\n",r);

    printf("\nDigite a opção para frequencia DDS\n");
    scanf ("%d",&option);
    r = DDS_Freq(option);
    printf("r = %d\n",r);

    printf("Digite o nome do arquivo\n");
    scanf ("%s",nome);

    printf("Digite o numero de amostragem\n");
    scanf ("%d",&n);

    cont = 0;

    do
    {
            Read_Register_PCIE(fd,buffer_reg);

            if ( buffer_reg [ADDRESS_PCIE_STATUS-OFFSET_INST_PCIE] == ACK )
        {

             sprintf(path,"%s%d.txt",nome,cont++);

             status = Send_to_NIOS_Command_Exe01(fd,1,1024*10,nome);

             if ( status != 1 )
            {
```

```
 61                printf("Erro ao ler comando NIOS\n\n");
 62                break;
 63             }
 64             else
 65                  printf ("Salvando arquivo %s\n",path);
 66
 67         }
 68         else
 69         {
 70             printf("NIOS nÃ£o responde \r");
 71         }
 72     }while ( cont < n );
 73
 74     close(fd);
 75 }
 76
 77 //=========================================================================
 78
 79 //################################################
 80 //             Medida de Impedância
 81 //################################################
 82
 83 int  DDS_Turn_ON(void)
 84 {
 85      int           status;
 86      const int     size_inst = 10*TIPO_MEMORIA;
 87      unsigned char buffer_reg[size_inst];
 88 //*********************************************************
 89
 90      Read_Register_PCIE(fd,buffer_reg);
 91
 92      if ( buffer_reg [ADDRESS_PCIE_STATUS-OFFSET_INST_PCIE] == ACK )
 93      {
 94           status = Send_to_NIOS_Command_Exe06(fd);
 95           if ( status == 1 )
 96       {
 97            return 1;
 98       }
 99       else
100       {
101            return -1;
102       }
103     }
104     else
105     {
106          return 0;
107     }
108 }
109
110 //=========================================================================
111
112 int  DDS_Freq(unsigned int f)
113 {
114      int           status;
115      const int     size_inst = 10*TIPO_MEMORIA;
116      unsigned char buffer_reg[size_inst];
117 //*********************************************************
118
119      Read_Register_PCIE(fd,buffer_reg);
120
```

```
121     if ( buffer_reg [ADDRESS_PCIE_STATUS-OFFSET_INST_PCIE] == ACK )
122     {
123          status = Send_to_NIOS_Command_Exe05(fd,f);
124          if ( status == 1 )
125      {
126           return 1;
127      }
128      else
129      {
130           return -1;
131      }
132    }
133    else
134    {
135          return 0;
136    }
137}
138
139//=========================================================================
140
141int ReadADC(char *nome)
142{
143        int           status;
144        const int     size_inst = 10*TIPO_MEMORIA;
145        unsigned char buffer_reg[size_inst];
146        FILE         *file;
147//************************************************************
148
149    Read_Register_PCIE(fd,buffer_reg);
150
151        if ( buffer_reg [ADDRESS_PCIE_STATUS-OFFSET_INST_PCIE] == ACK )
152    {
153         status = Send_to_NIOS_Command_Exe01(fd,1,1024*10,nome);
154
155         if ( status != 1 )
156              return -1;
157         else
158          return  1;
159    }
160    else
161    {
162            return 0;
163    }
164}
165
166//=========================================================================
167
168void Impedance(unsigned int F1, unsigned int FS, unsigned int FN)
169{
170    unsigned int TAB_FA [5] = { 20E3, 200E3, 1E6, 2E6, 5E6};
171
172    int  i,r,status,cont,dds = 0;
173    int  option;
174    char nome[255];
175//*****************************************************************
176
177    Initialization();
178    Full_Memo_Data(fd,0);
179    Show_Register_Map (fd);
180
```

```
181     printf("\nDigite a opção para frequência de amostragem ADC\n");
182     scanf ("%d",&option);
183     r = Send_to_NIOS_Command_Exe02(fd,option);
184
185     unsigned int FA = 0;
186     unsigned int F  = 0;
187
188     F = F1;
189     cont = 0;
190     do
191     {
192         FA     = F * 4;
193         status = DDS_Freq (F);
194         delay(0x0FFF,0xFFFF);
195         r      = DDS_Turn_ON ();
196         delay(0x0FFF,0xFFFF);
197
198         if (  status ==  1 )
199         {
200               do{
201                   r = ReadADC(nome);
202                   if ( r == 1 )
203                   {
204                    cont++;
205                    if ( cont == 2 )
206                    {
207                     sprintf(nome,"ADC_F_%d_Hz.txt",F);
208                     printf("Salvando arquivo = %s\n",nome);
209                     F    = F + FS;
210                     cont = 0;
211                     break;
212                    }
213                   }
214                   else
215                 {
216                    printf("Esperando FIFO\n");
217                 }
218              }while(1);
219         }
220         else
221         {
222               printf("Erro ao configurar o DDS %d %d\n",F,status);
223               break;
224         }
225     }while ( F <= FN );
226 }
227
228
229 //=========================================================================
230
231 //##########################################
232 //      Medida de Coincidência - TDC
233 //##########################################
234
235 int TDC(void)
236 {
237         const int     size_inst = 10*TIPO_MEMORIA;
238         unsigned char buffer_reg[size_inst];
239
240         const int     size_word = 1024*64 ;  // numero de bytes a serem lidos da
```

```
   FIFO = 64 Kbytes
    int           data[size_word];

    int           i,n,m,cont,max;
    char          nome[255];
    FILE  *file,*File_temp;
//*****************************************************************

    Initialization();
    Clear_Command_NIOS(fd);
    Full_Memo_Data(fd,0);
    Show_Register_Map (fd);

    printf("Digite o nome do arquivo\n");
    scanf ("%s",nome);

    printf("Digite o numero de amostragem\n");
    scanf ("%d",&n);

    printf("Digite a opção para o CLK do TDC < 0 a 7> \n");
    scanf ("%d",&m);

    if ( Send_to_NIOS_Command_Exe02(fd,m)  == 1 )
    {
         printf("CLK selecionado \n");
    }else
    {
         printf("Erro CLK\n");
    }

        File_temp = fopen   ("DADOS_FIFO_GERAL.txt","w+t");

    int *h  = (int*) malloc(sizeof(int) * 1000000 );

    if ( h == NULL )
        {
            printf("Erro ao alocar memória");
            return 1;
        }
    for ( i = 0; i  < 1000000 ; i ++)
        {
            h[i] = 0;
        }

    cont    = 0;
    do
    {
            Read_Register_PCIE(fd,buffer_reg);

            if ( buffer_reg [ADDRESS_PCIE_STATUS-OFFSET_INST_PCIE] == ACK )
        {
            if ( Send_to_NIOS_Command_Exe03(fd,data,size_word,File_temp) == 1 )
            {
                max = 0;

                histograma(h, data, size_word );

                printf("Adquirindo dados %d\n",cont++);
            }
        }
```

```
300     }while ( cont < n );
301
302         printf ("Salvando arquivo %s\n",nome);
303     file = fopen   (nome,"w+t");
304     if ( file != NULL && h != NULL )
305         {
306                 save_histograma(file,h,1000000);
307             fclose(file);
308             close(fd);
309         }
310     free(h);
311     close(fd);
312     return 1;
313 }
314
315 //=========================================================================
316
317 //##########################################
318 //    Análise da performance do PCIExpress
319 //##########################################
320
321 float  Medida_Tempo_Morto(int n)
322 {
323       float t,tx;
324       unsigned int i,j,a,b,m;
325       const int CLOCKS_PER_SEC = 1000000;
326
327 //************************************************************************
328
329       m = 30;
330       //-------------------------------------------------------------
331       a    = clock();
332       for ( j = 0; j < m ; j++)
333       {
334             Send_to_NIOS_Command_Exe01(fd,0,n,NULL);
335       }
336       //-------------------------------------------------------------
337       b   = clock();
338
339       t   = (float) (b - a )/ (float) CLOCKS_PER_SEC;
340
341       tx  = m * n * 2 / t;
342
343       return tx;
344 }
345
346 //=========================================================================
347 //=========================================================================
348
349 void Run_Tempo_Morto(void)
350 {
351     int   i;
352     float TX;
353     int   n_word[] =
    {2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536};
354
355   //*********************************************************************
356
357     FILE *file;
358
```

```
359     file = fopen ("Tempo_Morto.txt","w+t");
360
361     for ( i = 0; i < 16 ; i ++)
362     {
363         TX =  Medida_Tempo_Morto( n_word [i]) / 1E6;
364
365         printf("bloco = %d (words) Tx = %f (Mb/s) \n", n_word [i] , TX);
366
367         fprintf(file,"%d %f\n", n_word [i] , TX);
368     }
369     fclose(file);
370
371 }
372
373 //=========================================================================
374 //=========================================================================
375
376 float Medida_Tx_PCI(FILE *file,unsigned char option,unsigned int m, unsigned p,
    unsigned int size_data )
377
378 {
379       const int CLOCKS_PER_SEC = 1000000;
380       const int size_word = 1024*64 ;
381       unsigned char byte[size_word*TIPO_MEMORIA];
382
383       unsigned int i,j,n,a,b;
384       float t,tx;
385
386  //************************************************************************
387
388       for ( i = 0; i < m ; i ++ )
389       {
390            n    = p*(i+1);
391        //-------------------------------------------------------------
392
393            a    = clock();
394        for ( j = 0; j < n ; j++)
395            {
396               RunRead_Memory_PCI (&byte[0],OFFSET_DATA,size_data,fd);
397        }
398        //-------------------------------------------------------------
399        b   = clock();
400
401        t   = (float) (b - a )/ (float) CLOCKS_PER_SEC;
402
403        tx  = size_data * n / t;
404
405        if ( option )
406            {
407              printf("N  = %d  T = %f (s) Tx = %f (MB/s) Size = %d\n", n, t ,
    tx/1E6,size_data);
408        }
409        fprintf(file,"%d\t %f\n",n,t);
410       }
411       return tx;
412 }
413
414 //=========================================================================
415 //=========================================================================
416
```

```
417 void Testa_Hardware(void)
418 {
419     int   i,size;
420     float TX;
421     char  path[1000];
422     int   bloco[] = {   2,   4,   8,  16,  32,  64, 128,
    256,512,1024,2048,4096,8192,16384,32768,65536,131072};
423     int   vezes[] = {1000,1000,1000,1000,1000,1000,1000,1000,500, 100,  50,  10,
   10,   10,   10,   10,    10};
424
425  //************************************************************************
426
427     FILE *file_01,*file_02;
428
429     file_02 = fopen ("resultado.txt","w+t");
430
431     do
432     {
433         size = bloco[i];
434
435         if ( size <= 131072 )
436         {
437             sprintf(path,"bloco%d.txt",size);
438
439             file_01 = fopen (path,"w+t");
440             TX = Medida_Tx_PCI(file_01,0,10,vezes[i], size) / 1E6;
441             fclose(file_01);
442
443             printf("bloco = %d (bytes) Tx = %f (Mb/s) \n", size, TX);
444             fprintf(file_02,"%d %f\n", size , TX);
445         }
446         else
447             break;
448
449         i++;
450
451     }while(1);
452     fclose(file_02);
453 }
454
455 //=========================================================================
456 //=========================================================================
457
```

ANEXO B -- Programa de Comunicação com o NIOS II

```
 1//---------------------------------------------------------------------
 2//---------------------------------------------------------------------
 3
 4//  Centro Brasileiro de Pesquisas Físicas
 5//  Rio de janeiro, 2011
 6//  LHCb
 7
 8//---------------------------------------------------------------------
 9//---------------------------------------------------------------------
10
11#include <stdio.h>
12#include <math.h>
13#include <stdlib.h>
14#include <unistd.h>
15#include <string.h>
16
17#include <sys/types.h>
18#include <sys/times.h>
19#include <sys/stat.h>
20#include <fcntl.h>
21
22#include "write_pci_V01.h"
23
24#define ACK   (unsigned char )0xAA
25#define NACK  (unsigned char )0xCC
26#define WAIT  (unsigned char )0xEE
27
28#define OFFSET_INST_NIOS       (unsigned int) 0x0000
29#define OFFSET_INST_PCIE       (unsigned int) 0x0028
30#define OFFSET_DATA        (unsigned int) 0x03FC
31
32#define TIPO_MEMORIA           sizeof ( int )
33
34#define ADDRESS_PCIE_STATUS    (unsigned int) OFFSET_INST_PCIE + 0*TIPO_MEMORIA
35#define ADDRESS_PCIE_COMMAND   (unsigned int) OFFSET_INST_PCIE + 1*TIPO_MEMORIA
36#define ADDRESS_PCIE_SIZE      (unsigned int) OFFSET_INST_PCIE + 2*TIPO_MEMORIA
37#define ADDRESS_PCIE_CHECKSUM  (unsigned int) OFFSET_INST_PCIE + 3*TIPO_MEMORIA
38
39//---------------------------------------------------------------------------------
40//---------------------------------------------------------------------------------
41//                             Funções de baixo nível
42//---------------------------------------------------------------------------------
43//---------------------------------------------------------------------------------
44int   fd = 0;
45/*
46      Abre um canal de comunicação com o driver para o PCI Express
47*/
48void  OpenCommunication(void)
49{
50      if ((fd = open("/dev/ecs", O_RDWR)) == -1)
51      {
52        printf("Erro ao abrir aquivo com driver make\n");
53      }
54}
55/*
56      Funções para escrever dados na memória da FPGA por meio do PCI Express
```

```
57 */
58 void  RunWrite_Memory_PCI(unsigned char *data,int offset,int size,int fd)
59 {
60       lseek(fd, offset, SEEK_SET);
61       write(fd, data, size);
62 }
63 /*
64       Funções para ler dados da memória da FPGA por meio do PCI Express
65 */
66 void  RunRead_Memory_PCI (unsigned char *data,int offset,int size,int fd)
67 {
68       lseek(fd, offset, SEEK_SET);
69       read(fd, data, size);
70 }
71 /*
72       Funções para receber informações do NIOS
73 */
74 void  Read_Register_PCIE(int fd,unsigned char *buffer_reg)
75 {
76       const int size_inst = 10*TIPO_MEMORIA;
77
78       RunRead_Memory_PCI (buffer_reg,OFFSET_INST_PCIE,size_inst,fd);
79 }
80 /*
81       Funções para escrever enviar informações para o NIOS
82 */
83 void  Read_Register_NIOS(int fd,unsigned char *buffer_reg)
84 {
85       const int size_inst = 10*TIPO_MEMORIA;
86
87       RunRead_Memory_PCI (buffer_reg,OFFSET_INST_NIOS,size_inst,fd);
88 }
89 /*
90       Funções para acessar a memória de dados
91 */
92 void Full_Memo_Data(int fd,int option)
93 {
94      union{
95         unsigned char byte[150*TIPO_MEMORIA];
96         unsigned int  word[150];
97      }temp;
98
99      int       i;
100     unsigned  char      data;
101
102     for ( i = 0 ; i < 150; i ++ )
103           temp.word[i] = i;
104
105     for ( i = 0; i  < 150 ; i ++)
106     {
107       switch ( option ) {
108         case 0: data = 0x00;
109                         break;
110         case 1: data = (unsigned char) i ;
111                         break;
112         case 2: data = (unsigned char) 150 - i;
```

```
113                         break;
114          default:
115                         data = 0x00;
116        }
117        temp.word[i] = data;
118      }
119      RunWrite_Memory_PCI(&temp.byte[0],OFFSET_DATA,150*TIPO_MEMORIA,fd);
120 }
121 
122 //-------------------------------------------------------------------------------
123 //-------------------------------------------------------------------------------
124 //                             Funções de médio nível
125 //-------------------------------------------------------------------------------
126 //-------------------------------------------------------------------------------
127 
128 void delay(unsigned int a,unsigned int b)
129 {
130      unsigned int i,j;
131 
132      for ( i = 0; i  < a; i ++)
133         for ( j = 0; j < b; j ++);
134 }
135 
136 /*
137      Limpa o registrador de comandos para o NIOS
138 */
139 
140 void Clear_Command_NIOS(int fd)
141 {
142      int i;
143      const int      size_inst = 10*TIPO_MEMORIA;
144      unsigned char buffer_reg_w_nios[size_inst];
145 
146      //-----------------------------------**********----------
147      for ( i = 0; i < size_inst; i ++)
148        buffer_reg_w_nios[i] = 0x00;
149      //-------------------------------------------------------------
150 
151      RunWrite_Memory_PCI(buffer_reg_w_nios,OFFSET_INST_PCIE,size_inst,fd);
152 }
153 
154 /*-------------------------------------------------------------------------*/
155 /*-------------------------------------------------------------------------*/
156 /*
157    Essa função apresenta o mapa de registradores na memória interna da FPGA
158    que está sendo compartilhada entre o NIOS e o PCIe.
159 /*-------------------------------------------------------------------------*/
160 /*-------------------------------------------------------------------------*/
161 
162 void Show_Register_Map (int fd)
163 {
164     int i,j,cont;
165     const int size_inst = 10;
166     unsigned char buffer_inst[size_inst*TIPO_MEMORIA];
167 
168     const int size_data = 100;
```

```
169     unsigned char buffer_data[size_data];
170
171     /* Lê os N bytes dos registradores utilizados              */
172     /* para enviar comandos do PC para o NIOS via PCIExpress    */
173
174
    printf("=========================================================================\n");
175     printf("|                  MAP_REGISTER MEM_DUAL PCIE and NIOS
    |\n");
176     printf("|                              < > C B P F < >                    pablo
    |\n");
177
    printf("=========================================================================\n");
178
179     RunRead_Memory_PCI (buffer_inst,OFFSET_INST_NIOS,size_inst*4,fd);
180
181     printf("\n");
182
183     for ( i = 0; i < 4; i ++)
184     {
185             printf("REG NIOS : Address 0x%.4x to 0x%.4x :...",OFFSET_INST_NIOS
     +i*10,OFFSET_INST_NIOS+(i+1)*10-1);
186
187         for ( j = 0; j  < size_inst ; j ++)
188
189            printf("_%.2X",buffer_inst[ (i*10) + j ]);
190
191         printf("\n");
192     }
193
194     /* Lê os N bytes dos registradores utilizados              */
195     /* para enviar comandos do NIOS para o PC via PCIEpress    */
196
197     printf("\n");
198
199     RunRead_Memory_PCI (buffer_inst,OFFSET_INST_PCIE,size_inst*4,fd);
200
201     for ( i = 0; i < 4 ; i ++)
202     {
203             printf("REG PCIE : Address 0x%.4x to 0x%.4x :...",OFFSET_INST_PCIE
     +i*10,OFFSET_INST_PCIE+(i+1)*10-1);
204         for ( j = 0; j  < size_inst ; j ++)
205
206            printf("_%.2X",buffer_inst[ (i*10) + j ]);
207
208         printf("\n");
209     }
210
211       printf("\n\n");
212
213
214     /* Lê os 100 primeiro bytes da memória compartilhada
215        utilizados para transferência de dados.
216     */
217
218     RunRead_Memory_PCI (buffer_data,OFFSET_DATA,size_data,fd);
```

```
219
220     for ( i = 0; i  < size_data/10 ; i ++)
221     {
222       printf("REG DATA : Address 0x%.4x to 0x%.4x :...",OFFSET_DATA+i*10,OFFSET_DATA+
    (i+1)*10-1);
223       for ( j = 0; j < 10 ; j ++)
224       {
225             printf("_%.2X",buffer_data[10*(i+1)-1 - j]);
226       }
227       printf("\n");
228     }
229     printf("\n");
230
    printf("=========================================================================\n");
231     printf("|
    ********************************************************|\n");
232
    printf("=========================================================================\n");
233 }
234
235 /*-------------------------------------------------------------------------*/
236 /*-------------------------------------------------------------------------*/
237 /*
238    Essa função apresenta um comando + dados para o NIOS
239    É preciso acresentar o checksum aos dados.
240 */
241 /*-------------------------------------------------------------------------*/
242 /*-------------------------------------------------------------------------*/
243
244 int  Send_Command(unsigned char *data,unsigned char command, int n, int  m,int fd)
245 {
246
247     const int     size_inst = 10*TIPO_MEMORIA;
248     unsigned char buffer_reg_w_nios[size_inst];
249     unsigned char buffer_reg_r_pcie[size_inst];
250
251     int i,k;
252     unsigned char checksum = 0;
253
254     union {
255        unsigned char  byte[TIPO_MEMORIA];
256        unsigned int   word;
257     }temp;
258
259     temp.word = n;
260
261 //-------------------------------------------------------------------------
262     for ( i = 0; i < size_inst;  i ++)
263       buffer_reg_w_nios[i] = 0x00;
264 /-------------------------------------------------------------------------
265
266     checksum = 0;
267
268     buffer_reg_w_nios[0*TIPO_MEMORIA+0] = 0xAA;
269     buffer_reg_w_nios[1*TIPO_MEMORIA+0] = command;
270
```

```
271     buffer_reg_w_nios[2*TIPO_MEMORIA+0] = temp.byte[0];
272     buffer_reg_w_nios[2*TIPO_MEMORIA+1] = temp.byte[1];
273
274     buffer_reg_w_nios[3*TIPO_MEMORIA+0] = checksum;
275
276//-----------------------------------------------------------------
277     // Salva os dados na memória compartilhada referente
278
279     RunWrite_Memory_PCI(&data[0], OFFSET_DATA, m,fd);
280
281     // Salva o comando na memória compartilhada referente aos
282     // registradores.
283
284     RunWrite_Memory_PCI(buffer_reg_w_nios,OFFSET_INST_NIOS,size_inst,fd);
285     delay(0x0FFF,0xFFFF);
286     Read_Register_PCIE (fd,buffer_reg_r_pcie);
287
288     if ( buffer_reg_r_pcie [ADDRESS_PCIE_COMMAND-OFFSET_INST_PCIE] == ACK )
289    {
290           return  1;
291    }
292     else
293    {
294       return -1;
295    }
296}
297
298//-------------------------------------------------------------------------------
299//-------------------------------------------------------------------------------
300//                            Funções de alto nível
301//-------------------------------------------------------------------------------
302//-------------------------------------------------------------------------------
303
304int  Send_to_NIOS_Command_Exe01(int fd,int option,int n_read_word,char *path)
305{
306     // option, se igual 1, salva os dados.
307     // n_read_word, número de dados a serem salvos se option igual a 1.
308
309    const unsigned char command_Exe01[2] = {0x01,0x02};
310     const int size_word = 1024*64 ;
311     int i,j,erro,cont=0;
312
313     union{
314        unsigned char  byte[size_word*TIPO_MEMORIA];
315        unsigned int   word[size_word];
316     } data [2];
317
318     FILE *file;
319
320     for ( i = 0 ; i < size_word ; i ++ )
321    {
322            data[0].word[i] = 0;
323            data[1].word[i] = 0;
324    }
325
326     // size É o número de bytes a serem lidos.
```

```
327     // No máximo será de 1024*64 words, sendo cada word de 2 bytes
328     //------------------------------------------------------------------------------
    -------------
329     //------------------------------------------------------------------------------
    -------------
330
331     erro = 0;
332
333     for ( j = 0 ; j < 2; j ++)
334    {
335         if  ( Send_Command( &data[j].byte[0], command_Exe01[j], n_read_word , 0 , fd)
    == 1 )
336         {
337           delay(0x0FFF,0xFFFF);
338               RunRead_Memory_PCI
    (&data[j].byte[0],OFFSET_DATA,n_read_word*TIPO_MEMORIA,fd);
339           delay(0x0FFF,0xFFFF);
340             }
341             else
342         {
343           erro = 1;
344           break;
345         }
346    }
347
348//------------------------------------------------------------------------------
    --------
349//------------------------------------------------------------------------------
    --------
350
351     if ( erro == 0 )
352     {
353      if ( option )
354      {
355          file = fopen (path,"w+t");
356
357          for ( i = 0; i < n_read_word ; i ++)
358            fprintf(file,"%d\t %d\t %d\n",i, data[0].word[i], data[1].word[i]);
359
360          fclose(file);
361      }
362      return  1;
363     }
364     else
365     {
366          return -1;
367     }
368
369}
370
371//=============================================================================
372//=============================================================================
373
374int  Send_to_NIOS_Command_Exe02(int fd,int option)
375{
376    // option, se igual 1, salva os dados.
```

```
377     //
378     // n_read_word, número de dados a serem salvos se option igual a 1.
379
380      const unsigned char command_Exe02 = 0x03   ;
381      const int size_word = 1024*64 ;
382      int i,j,m,cont=0;
383
384      union{
385         unsigned char byte[size_word*TIPO_MEMORIA];
386         unsigned int  word[size_word];
387      } data;
388
389
390      for ( i = 0 ; i < size_word ; i ++ )
391            data.word[i] = 0;
392
393      m = 4;
394      data.word[0] = option;
395
396      // size É o número de bytes a serem escritos.
397
398      if  ( Send_Command( &data.byte[0], command_Exe02, 0 , m, fd) == 1 )
399     {
400         return  1;
401     }
402     else
403         return -1;
404 }
405
406 //=============================================================================
407 //=============================================================================
408
409 int  Send_to_NIOS_Command_Exe03(int fd,int *data,int n,FILE *file)
410 {
411     // n_read_word, número de dados a serem salvos se option igual a 1
412
413      const unsigned char command_Exe03 = 0x03;
414      const int size_word = 1024*64 ;
415      unsigned int i,address,pos;
416      int m;
417
418      union{
419         unsigned char byte[size_word * 4];
420         unsigned int  word[size_word];
421      } temp ;
422
423
424      for ( i = 0 ; i < size_word * 4 ; i ++ )
425            temp.byte[i] = 0;
426
427      m = 0;
428      if  ( Send_Command( &temp.byte[0], command_Exe03, 0 , m , fd) == 1 )
429      {
430         pos     = 0;
431         address = OFFSET_DATA;
432         for ( i = 0; i < 255 ; i  ++ )
```

```
433         {
434               RunRead_Memory_PCI ( &temp.byte[pos], address, 1024 , fd);
435
436               address += 1024;
437               pos     += 1024;
438         }
439
440         for ( i = 0; i < n ; i++)
441               data[i] = temp.word[i];
442
443         if  ( file != NULL )
444         {
445             n = 1024 * 255 / 4;
446             for ( i = 0; i < n ; i++)
447             {
448                   fprintf(file,"%d\t %d\n",i,data[i]);
449             }
450         }
451         return  1;
452        }
453        else
454        {
455         return -1;
456        }
457
458}
459
460//=============================================================================
461//============================================================================
462
463int  Send_to_NIOS_Command_Exe05(int fd,unsigned int option)
464{
465     const unsigned char command_Exe05 = 0x05;
466     const int size_word = 1024*64 ;
467     int i,j,m,cont=0;
468
469     union{
470        unsigned char byte[size_word*TIPO_MEMORIA];
471        unsigned int  word[size_word];
472     } data;
473
474
475     for ( i = 0 ; i < size_word ; i ++ )
476           data.word[i] = 0;
477
478     m = 4;
479     data.word[0] = option;
480
481
482
483     if  ( Send_Command( &data.byte[0], command_Exe05, 0 , m, fd) == 1 )
484    {
485       return  1;
486    }
487    else
488       return -1;
```

```
489 }
490
491 //=====================================================================
492 //==========================================================================
493
494 int  Send_to_NIOS_Command_Exe06(int fd)
495 {
496      const unsigned char command_Exe06 = 0x06;
497      const int size_word = 1024*64 ;
498      int i,j,m,cont=0;
499
500      union{
501         unsigned char byte[size_word*TIPO_MEMORIA];
502         unsigned int  word[size_word];
503      } data;
504
505
506      for ( i = 0 ; i < size_word ; i ++ )
507            data.word[i] = 0;
508
509      m = 0;
510      // size É o número de bytes a serem escritos.
511
512      if  ( Send_Command( &data.byte[0], command_Exe06, 0 , m, fd) == 1 )
513     {
514        return  1;
515     }
516     else
517        return -1;
518 }
519
520 //===================================================================================
521 //=================================================================================
522
523 void  Initialization(void)
524 {
525       OpenCommunication();
526
527       if ( fd != -1 )
528     {
529     }
530       else
531     {
532               printf ("Erro ao abrir o driver\n");
533               exit(-1);
534     }
535 }
536
537 //===================================================================================
538 //===================================================================================
539
540
541
```

ANEXO C -- Fluxogramas dos programas desenvolvidos

Este anexo apresenta os fluxogramas dos programas desenvolvidos para o TDC, utilizado nas medidas do projeto Neutrinos Angra, e para o DDS e os ADC's, utilizados nas medidas de EIE.

O programa representado na Figura C.1 foi desenvolvido com o objetivo de realizar um controle em alto nível do funcionamento do TDC. O usuário determina o nome do arquivo de armazenamento de dados, a quantidade de amostragens que será realizada e a frequência de operação do TDC (*clock*). Após isso, o sistema cria o arquivo, aloca a memória necessária e começa a fazer a contagem de tempo entre os eventos de entrada. Quando a quantidade determinada de aquisições é atingida, o arquivo é fechado e o sistema fica aguardando o início de uma nova operação de contagem

A Figura C.2 mostra o fluxograma do programa desenvolvido para realização de medidas de espectroscopia de impedância elétrica. O usuário entra com 4 parâmetros; frequências inicial (fi) e final (ff) de varredura do DDS, passo de incremento da frequência de varredura (Δf) e o valor da frequência de amostragem (fa). Posteriormente, o sistema liga o DDS, ajustado na fi, e inicia a aquisição de dados até completar a capacidade de armazenamento da memória FIFO. Na sequência, os dados são gravados em um arquivo e a frequência do DDS é incrementada de Δf. Caso a nova frequência não seja maior que a frequência final, o sistema inicia uma nova aquisição de dados. O processo é finalizado quando a frequência do DDS atinge o valor de ff.

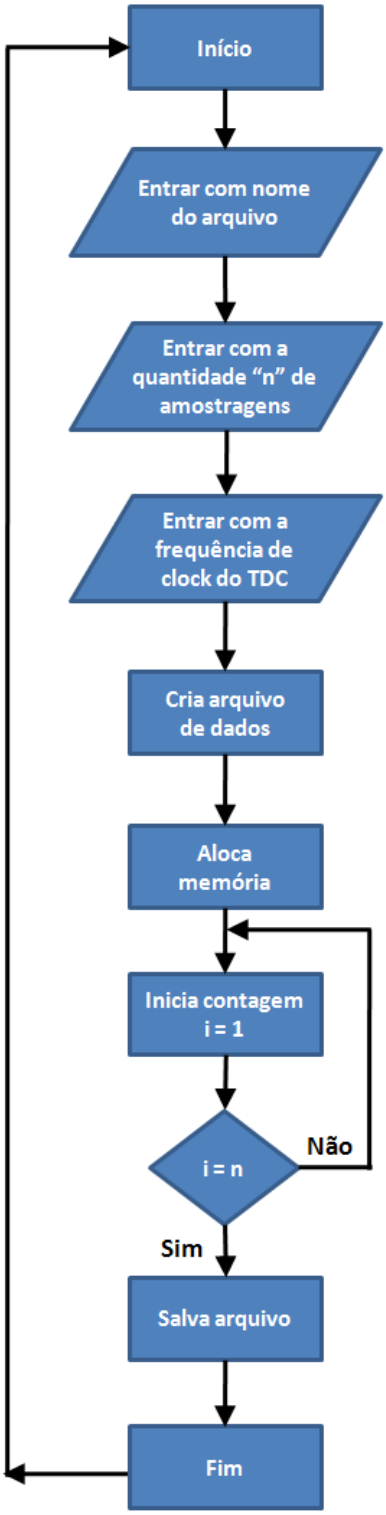


Figura C.1: Fluxograma de controle do TDC.

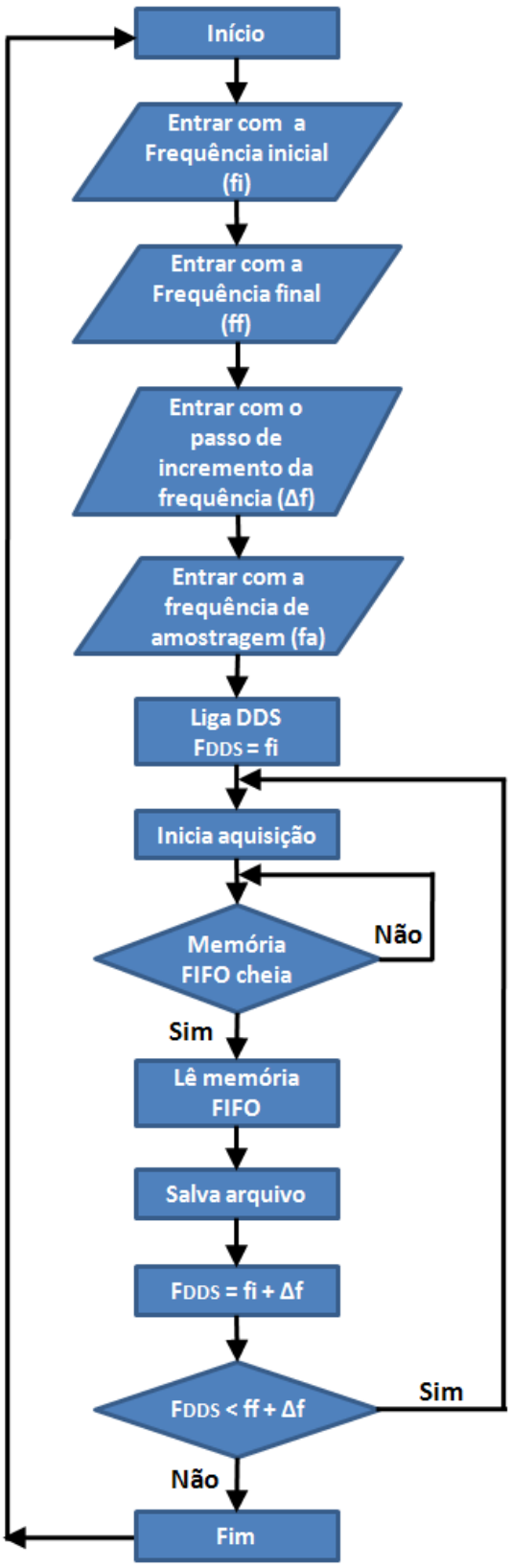


Figura C.2: Fluxograma para medida de impedância