



UNIVERSIDADE D
COIMBRA

Diogo Henrique de Castro Caires

**DESENVOLVIMENTO DE UMA PLATAFORMA DIGITAL DE
INTERFACE COM MATLAB/SIMULINK PARA CONTROLO
DE CONVERSORES DE POTÊNCIA**

Dissertação no âmbito do Mestrado integrado em Engenharia Eletrotécnica e de Computadores no ramo de Energia orientada pelo Professor Doutor André Manuel dos Santos Mendes e coorientada pelo Doutor Luís Miguel Antunes Caseiro apresentada ao Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Ciência e Tecnologia da Universidade de Coimbra.

Setembro de 2019

Desenvolvimento de uma plataforma digital de interface com Matlab/Simulink para controlo de conversores de potência



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA

Diogo Henrique de Castro Caires

Dissertação no âmbito do Mestrado Integrado em Engenharia Eletrotécnica e de Computadores, na área de Energia e apresentada à Faculdade de Ciências e Tecnologias da Universidade de Coimbra.

Orientador: Prof. Doutor André Manuel dos Santos Mendes

Coorientador: Doutor Luís Miguel Antunes Caseiro

Juri: Prof. Doutor Fernando José Teixeira Estêvão Ferreira

Prof. Doutor Tony Richard de Oliveira de Almeida

Prof. Doutor André Manuel dos Santos Mendes

Setembro 2019

Este trabalho insere-se no projeto SAICT-45-2017-POCI-01-0145-FEDER-029112 - PTDC/EEI-EEE/29112/2017, financiado pelo “Programa Operacional Temático Competitividade e Internacionalização” – FEDER e pela Fundação para a Ciência e a Tecnologia (FCT)—OE, e em parte pelo projeto UID/EEA/50008/2019, financiado pela FCT—OE.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR



*"Our greatest weakness lies in giving up. The most certain way to succeed is
always to try just one more time."*

- Thomas Edison

Agradecimentos

Começo por agradecer ao meu orientador, Professor Doutor André Manuel dos Santos Mendes e coorientador Luís Miguel Antunes Caseiro por todo o apoio, ajuda e acima de tudo confiança nas minhas capacidades.

Agradeço ao Instituto de Telecomunicações de Coimbra pelo material disponibilizado para a realização desta dissertação.

Agradeço aos meus amigos, que sempre me apoiaram e tornaram o meu percurso académico uma experiência única e enriquecedora.

Também agradeço aos meus colegas do laboratório de sistemas energéticos, em especial ao Tiago Oliveira e ao Válder Costa por todas as recomendações ao longo deste trabalho, bem como aos momentos de descontração.

O meu especial obrigado aos meus pais por todo o apoio ao longo do meu percurso, assim como pelos conselhos dados e todo o apoio desde sempre.

Quero agradecer a uma pessoa especial, à minha namorada Carolina, por todos os momentos partilhados e pelo incansável apoio ao longo desta jornada.

Por fim agradeço a todas as pessoas que têm vindo a me apoiar, pois sem o apoio de todos, certamente este percurso teria sido muito mais árduo.

Resumo

Com a evolução da tecnologia, os algoritmos de controlo passaram a ser cada vez mais avançados e complexos, o que suscitou um aumento das necessidades de processamento. Deste modo, as plataformas de controlo, a utilizar em sistemas de eletrónica de potência, têm de ter velocidades de processamento elevadas. Ao nível da investigação de sistemas de controlo e desenvolvimento de protótipos, podem ser escolhidos dois tipos de plataformas bastante distintos. Uma é o uso de plataformas para prototipagem rápida, que têm um elevado desempenho e uma programação simples, sendo que a sua desvantagem é o custo elevado. A outra solução é a utilização de placas de desenvolvimento que são uma solução económica, contudo a sua programação é complexa, ou então a sua capacidade de processamento é baixa.

O objetivo da tese é desenvolver uma solução que junte o baixo custo e o alto desempenho a uma programação fácil que recorra a ferramentas muito utilizadas. Com estes parâmetros, será possível apresentar uma solução de prototipagem com um custo-eficiência elevado, ideal para a investigação. Quanto à plataforma a utilizar, optou-se pela ZedBoard da *Digilent*, que apresenta uma capacidade de processamento alta e compatibilidade com a ferramenta *Matlab/Simulink*, muito utilizada a nível global.

O uso do *Matlab/Simulink* permite a implementação direta de código previamente desenvolvido, sem necessidade de conversão. Assim, é possível facilitar o processo de programação da plataforma e eliminar erros de transição que existem em plataformas configuradas diretamente com linguagem C/VHDL e não através do *Simulink*. Além do mais, o *Simulink* é um dos programas para desenvolvimento de algoritmos de controlo e simulação mais utilizado o que faz com que este seja o programa ideal para a solução a desenvolver.

Ao longo do trabalho, é pretendido aplicar na ZedBoard um algoritmo para controlo de um conversor de potência. Para isso, é necessário desenvolver ferramentas para programação da placa no *Simulink*, explorar as suas formas de processamento e garantir os requisitos do algoritmo. Após o desenvolvimento do modelo com o respetivo tipo de processamento, é realizada a programação da plataforma de controlo. Por último serão apresentadas as conclusões do trabalho bem como as propostas para trabalhos futuros.

Palavras-Chave: Plataformas Controlo, Conversores de Potência, Zedboard, Protótipos, Sistemas Embebidos, *Simulink HDL Coder*, *Simulink Embedded Coder*, *Simulink SoC Blockset*.

Abstract

With the evolution of technology, the control platform became a critical feature in electric power systems. This leads to increased research of electric power systems. Demand has guaranteed the evolution of control platforms, with a price reduction and increase of processing power. Even with the price reduction of control systems, solutions focused in rapid prototyping still have a high cost. Either it is possible to use barebone solution with lower cost, but in this case the processing power is slow or the programming procedure is complex and slow. Due to these problems, it does not exist an appropriate solution for use in research projects or small companies with low investments capacities.

The objective of this work is to create a solution, which fills a blank space in the control platform market. The solution needs to have a low cost and high processing capacity with an easy programming procedure. With these characteristics, the platform to be developed will be a prototyping solution with high cost-efficiency, ideal for use in research. The used platform is a ZedBoard from *Digilent*, which has a good processing capacity, is compact, and compatible with *Simulink*.

Using *Matlab/Simulink* ensures the direct implementation of code developed and tested, with no need for conversion. Due to the fact that the code doesn't need to be converted, as happens on platforms programmed in C/VHDL, conversion errors are eliminated. Furthermore, *Simulink* is one of the most used programs for developing control algorithms, testing and development, making this program ideal for programming the ZedBoard.

This thesis aims to develop a simple way to program a ZedBoard from Matlab/Simulink. The test model will be a power converter algorithm, and the used platform needs to guarantee all the control needs.

Keywords: Control Platforms, Power Converters, Zedboard, Prototypes, Embedded Systems, *Simulink HDL Coder*, *Simulink Embedded Coder*, *Simulink SoC Blockset*

Índice

Agradecimentos	vii
Resumo	ix
Abstract	xi
Lista de Figuras	xvii
Lista de Tabelas	xxi
Lista de Acrónimos	xxiii
1 Introdução	1
1.1 Motivação	1
1.2 Objetivos	2
1.3 Implementação	2
1.4 Estrutura do Trabalho	3
2 Estado da Arte	5
2.1 Sistema de Conversão de Energia Elétrica	5
2.1.1 Conversores de Eletrónica de Potência	7
2.1.1.1 Conversores Multinível	7
2.1.2 Estratégias de Controlo	8
2.2 Plataformas de Controlo	9
2.2.1 Programação pelo <i>Simulink</i>	10
2.2.2 Escolha da Plataforma	10
2.2.3 ZedBoard	11
2.3 Sistemas UPS	12
2.3.1 Conversores Multinível nas UPS	13
2.3.2 Controlo Preditivo	13
2.3.3 Retificador Ativo	14
2.3.4 Inversor	16
2.3.5 Requisitos do Sistema a Utilizar	17

3	Desenvolvimento da Plataforma	19
3.1	Escolha ADCs	20
3.1.1	Funcionamento Pmod AD1	20
3.2	Configuração da Plataforma	21
3.2.1	Módulos dos Pmod AD1	21
3.2.2	Verificação da aquisição de dados em simultâneo	22
3.2.3	Aquisição de Dados e Atualização de Saídas em Simultâneo	24
3.2.4	Velocidades de Transferência de Dados	24
3.2.5	Adição de ADC's ao IP CORE	26
4	Programação da Plataforma de Controlo	27
4.1	Processamento do Algoritmo	27
4.1.1	Processamento na FPGA	29
4.1.1.1	Conversão do Código para a ZedBoard	30
4.1.1.2	Aplicação do Algoritmo na ZedBoard	31
4.1.1.3	Resultados Obtidos	32
4.1.2	Processamento no ARM	33
4.1.2.1	Sincronismo Através de Esperas	34
4.1.2.2	Sincronismo Através de uma Verificação Periódica	36
4.1.3	Processamento Híbrido	36
4.1.3.1	Sincronismo com Múltiplas Esperas	37
4.2	Alternativa de Processamento	38
4.2.1	SoC Blockset	38
4.2.1.1	Desenvolvimento dos ADCs	39
4.2.1.2	Interrupções	41
5	Conclusões e Sugestões para Trabalhos Futuros	45
5.1	Conclusões	45
5.2	Sugestões para Trabalhos Futuros	46
	Referências	47
A	Manual Instalação	49
A.1	Instalação Matlab	49
A.2	Instalação Vivado	50
A.3	Conexção ao Hardware e testes	50
B	Manual Utilização	51
B.1	Leitura de Dados da FPGA no Matlab/Simulink	51
B.2	Gerar IP Core através do Simulink	59
B.3	Utilizar o "HDL Workflow Advisor"	61

B.4	Alteração do Projeto no <i>Vivado</i>	67
C	Código VHDL dos IP Cores utilizados	71
C.1	Código VHDL da <i>Digilent</i> para os Pmod AD1	71
C.2	Código VHDL para Flip Flop	74
C.3	Código VHDL para múltiplos Pmod AD1	75
D	Conversão Blocos Retificador Ativo	81
E	ZedBoard e Plataformas Utilizadas no Laboratório	87

Lista de Figuras

2.1	Esquema de um sistema de conversão de energia elétrica.	6
2.2	Esquema de um inversor NPC de 3 níveis.	8
2.3	Algumas das estratégias de controlo mais comuns.	9
2.4	Arquitetura de um SoC Zynq.	11
2.5	Modelo de um sistema UPS.	12
2.6	Retificador trifásico multinível do tipo NPC.	14
2.7	Diferentes vectores de tensão representados no plano $\alpha\beta$	15
2.8	Inversor trifásico multinível do tipo NPC.	16
2.9	Sistema <i>Back-to-Back</i> a utilizar.	17
3.1	Medidas a efetuar no sistema.	19
3.2	Pmod AD1 da <i>Digilent</i>	20
3.3	Funcionamento transferência de dados do Pmod AD1 da <i>Digilent</i>	20
3.4	Módulo Vivado Pmod AD1.	21
3.5	<i>Board Defenition</i> para 2 ADC PMOD AD1.	22
3.6	Valores adquiridos pelos 2 ADCs de um módulo, com uma frequência de amostragem de 8kHz.	23
3.7	Ampliação dos valores adquiridos pelos 2 ADCs de um módulo, com uma frequência de amostragem de 8kHz.	23
3.8	Módulo FF no <i>Vivado</i>	24
3.9	Simulação ao código VHDL do FF desenvolvido.	24
3.10	Esquema do envio de dados pelo Pmod AD1.	25
3.11	Forma de onda do sinal SCLK a 2 MHz e a 20 MHz.	25
3.12	Valor adquiridos pelos 4 ADC, a uma amostragem de 8kHz.	26
4.1	Fluxograma da programação ZedBoard pelo <i>Simulink</i>	28
4.2	Esquema de processamento de dados apenas pela FPGA.	29
4.3	Modelo do <i>Simulink</i> com os blocos do código do retificador incompatíveis com o <i>HDL Coder</i>	30
4.4	Algoritmo do bloco responsável por estimar os valores das variáveis no ciclo seguinte, após a sua conversão.	31

Lista de Figuras

4.5	Modelo <i>Vivado</i> a implementar na ZedBoard.	32
4.6	Esquema de processamento de dados apenas pelo ARM.	33
4.7	Fluxograma para sincronização de dados com recurso à espera ativa.	35
4.8	Variação do sinal de <i>clock</i> do ARM e da FPGA.	35
4.9	Esquema de processamento de dados híbrido.	36
4.10	Modelo <i>Simulink</i> , para verificação do sincronismo.	37
4.11	Exemplo da camada superior de um projeto do <i>SoC Blockset</i>	39
4.12	Esquema para implementação do ADC no <i>SoC Blockset</i>	40
4.13	Algoritmo do ADC desenvolvido no <i>Add-On Stateflow</i> do <i>Simulink</i>	40
4.14	Blocos do controlo e canal de memória.	41
4.15	Resultados de envio de dados pela memória RAM.	42
B.1	Gui para gerar IP Core para captura de dados.	52
B.2	Seleção “Board” para um novo projeto.	52
B.3	Painel de navegação no lado esquerdo com a opção “Create Block Design”.	53
B.4	Esquema básico da placa ZedBoard	53
B.5	“Design Sources” antes (a) e depois (b) de executar o HDL wrapper.	54
B.6	Exemplo de localização das pastas com os ficheiros VHDL dos IP Cores.	54
B.7	Exemplo da janela de adição de ficheiros VHDL.	55
B.8	Ferramenta "Create and Package IP Core".	55
B.9	Projeto temporário criado pela ferramenta " Create and Package New IP " do passo anterior.	56
B.10	Zedboard “block design” com os IP Cores “datacapture” e “pmodad1 _test”.	57
B.11	Ficheiro “.xdc” com definição da correspondência entre pinos da ZedBoard com o conector JA1.	58
B.12	Matlab Data Capture GUI.	58
B.13	Ficheiro da ZedBoard plugin _board.m	60
B.14	Ficheiro plugin _rd.m da ZedBoard.	61
B.15	Exemplo de um subsistema onde será executado o workflow advisor.	62
B.16	Ferramenta <i>HDL Workflow Advisor</i>	63
B.17	Janela do <i>HDL Workflow Advisor</i> durante o a escolha da plataforma.	63
B.18	Janela do <i>HDL Workflow Advisor</i> durante o ponto 1.2.	64
B.19	Janela do <i>HDL Workflow Advisor</i> durante o ponto 1.3.	64
B.20	Janela do <i>HDL Workflow Advisor</i> durante o ponto 1.3.	65
B.21	Janela do <i>HDL Workflow Advisor</i> no ponto para criar o projeto no <i>Vivado</i>	66
B.22	Janela do <i>HDL Workflow Advisor</i> após criação do projeto no <i>Vivado</i>	66
B.23	Janela do <i>Vivado</i> com o projeto criado a partir do <i>Workflow Advisor</i>	68
B.24	Ferramenta "create port" para adicionar portas de entrada e/ou saída no <i>Vivado</i>	69
D.1	Blocos do algoritmo incompatíveis com o <i>HDL Coder</i>	81

D.2 Interior de um dos blocos incompatíveis com o <i>HDL Coder</i>	81
D.3 Bloco responsável pela PLL existente no algoritmo.	82
D.4 Bloco que converte as tensões complexas nas tensões simples.	82
D.5 Bloco da transformada DQ para $\alpha\beta$ após conversão.	82
D.6 Bloco da transformada ABC para $\alpha\beta$ após conversão.	83
D.7 Bloco "rect_K2" o único bloco onde foi aplicado a conversão de variáveis para ponto fixo.	83
D.8 Bloco "rect _k 1" <i>apsconverso</i>	84
D.9 Bloco responsável pela PLL após conversão dos blocos incompatíveis.	85
E.1 Imagem da ZedBoard, e o seu tamanho comparado a uma caneta.	87
E.2 Módulos Pmod AD1 utilizados no projeto.	87
E.3 Comparação de tamanhos entre ZedBoard, <i>SB Rio</i> e <i>MicroLabBox</i>	88

Lista de Tabelas

2.1	Comparação entre Várias Soluções de Mercado e Placas de Controlo.	11
2.2	Diferentes estados de cada braço do retificador.	15
2.3	Entradas e saídas do algoritmo de controlo.	17
4.1	Valores de utilização ZedBoard para diferentes tipos de compilação.	33
B.1	Correspondência entre pinos Zedboard e conector Pmod JA1.	57
B.2	Correspondência entre os id dos pinos da ZedBoard com as suas portas PMOD. . .	69

Lista de Acrónimos

Acrónimos e Abreviaturas

AC	<i>Alternating Current</i>
ADC	Conversor Analógico para Digital
ARM	<i>Advanced RISC Machine</i>
AXI4	<i>Advanced eXtensible Interface 4</i>
CHB	<i>Cascade H-Bridge</i>
CS	<i>Chip Select</i>
DC	<i>Direct Current</i>
DMA	<i>Direct Memory Access</i>
DSPs	<i>Digital Signal Processor</i>
FCs	<i>Flying Capacitors</i>
FIFO	<i>First In First Out</i>
FOC	<i>Field Oriented Control</i>
FPGA	<i>Field-programmable Gate Array</i>
IGBT	<i>Insulated Gate Bipolar Transistor</i>
LED	<i>Light Emitting Diode</i>
LUT	<i>Look-up-Table</i>
NPC	<i>Neutral Point Clamped</i>
SCLK	<i>Serial Clock</i>
UPS	<i>Uninterruptible Power Supply</i>

Capítulo 1

Introdução

No primeiro capítulo serão enunciados os objetivos da dissertação, as motivações para a sua realização, e a estratégia de implementação a utilizar. Este ponto será finalizado com uma apresentação da estrutura da dissertação, com uma indicação das temáticas a serem abordadas em cada um dos capítulos.

A plataforma a desenvolver, insere-se no projeto "DRIFT" (*Datacenter Resilience Increase through Fault Tolerance in UPS systems*) que consiste no desenvolvimento de um protótipo de uma fonte de alimentação ininterrupta (UPS) de alta credibilidade para *datacenter* a decorrer no Instituto de Telecomunicações de Coimbra, no Laboratório de Sistemas de Energéticos. O projeto, deparou-se com a necessidade de, após desenvolvimento do sistema de controlo no *Simulink*, e a verificação do funcionamento do mesmo, escolher entre dois tipos de plataformas: Uma plataforma que permitisse uma programação simples, utilizando o *Simulink* para implementação do algoritmo, mas que apresentasse um custo elevado; Ou uma plataforma económica mas com um processo de programação complexo, devido à necessidade de conversão do algoritmo desenvolvido.

O objetivo do trabalho é desenvolver uma solução intermédia, que tenha um custo baixo mas que permita a implementação de algoritmos pelo *Simulink* de forma a utilizar o sistema de controlo já desenvolvido. Outro aspeto desta solução é ter dimensões reduzidas, de forma a ser facilmente incorporada no protótipo em desenvolvimento.

1.1 Motivação

O desenvolvimento de técnicas de controlo mais avançadas e complexas fez com que as plataformas de controlo precisassem de uma maior capacidade de processamento. O aumento dos requisitos destas plataformas levou a um grande desenvolvimento e à redução do custo das mesmas. As soluções para prototipagem rápida dispõem de uma programação simples e elevada velocidade de processamento. Contudo, estas plataformas ainda apresentam custos elevados. Em alternativa, existem placas de desenvolvimento que têm custos baixos, contudo apresentam uma programação complexa, ou então uma capacidade de processamento reduzida.

Ainda existem poucas plataformas adequadas para investigação, com boa capacidade de processamento e que permitam uma programação pelo *Simulink* a um baixo custo. Esta falha no mercado das plataformas digitais, motivou a procura de uma solução para satisfizer estas necessidades. Posto isto, o trabalho a realizar é uma mais valia para o grupo de investigação, pois providencia uma solução com custo-eficiência bastante apelativo.

1.2 Objetivos

O objetivo principal é o desenvolvimento de uma plataforma de controlo de baixo custo, para aplicação em protótipos de conversores de eletrónica de potência. A solução deverá ter uma capacidade de processamento elevada, embora com um preço reduzido. Esta capacidade de processamento e baixo custo, fará com que a solução tenha um custo-eficiência bastante elevado e apelativo.

Quanto à programação da plataforma, esta deve ser compatível com o *Matlab/Simulink* que permite o desenvolvimento de um algoritmo, simulação e posteriormente a implementação do mesmo. O *Simulink* apresenta vantagens não só a nível da implementação, uma vez que o uso do código é direto, mas também oferece uma programação simples (programação por blocos) e uma grande quantidade de informação devido aos inúmeros utilizadores do programa.

No decorrer do trabalho são desenvolvidas as ferramentas para programação da plataforma a partir do *Simulink* e são asseguradas as necessidades do sistema de controlo a implementar. Por último, pretende-se implementar um algoritmo na plataforma desenvolvida, bem como realizar testes e medições comparando a solução desenvolvida com uma já existente no mercado.

1.3 Implementação

A implementação pode ser dividida em duas partes distintas: Uma secção inicial onde será desenvolvido o modelo de *Simulink* para a plataforma, e uma segunda secção onde o algoritmo será implementado no modelo desenvolvido.

O desenvolvimento do modelo será realizado através das extensões (*Add-Ons*) "*Embedded Coder*" e "*HDL Coder*" que permitem a adaptação dos modelos do *Simulink* às plataformas digitais. Neste ponto, será verificado o funcionamento do modelo desenvolvido, e se o mesmo assegura os requisitos do algoritmo de controlo a implementar.

Após o desenvolvimento das ferramentas para programação da ZedBoard pelo *Simulink*, será efetuada a sua programação. Na implementação do algoritmo de controlo, é possível configurá-lo através de diversas formas de processamento. Como a plataforma a utilizar contém duas unidades de processamento, nomeadamente um ARM e uma FPGA, é possível executar o algoritmo apenas numa das unidades ou em ambas extraindo as vantagens de cada uma.

Para concluir, será ainda explorado o *SoC Blockset*, que é uma nova extensão do *Simulink* (adicionada na versão 2019a), que adiciona novas funcionalidades à programação de plataformas digitais, e sistemas embebidos.

Concluída a implementação do algoritmo na plataforma de controlo, será verificado o desempenho do sistema, por comparação com a plataforma *MicroLabBox* da *dSPACE*.

1.4 Estrutura do Trabalho

O presente documento está dividido em cinco capítulos distintos, sendo o primeiro capítulo composto por uma pequena introdução e pelas motivações que levaram à realização deste trabalho.

O capítulo 2 apresenta uma pequena introdução aos sistemas de conversão de energia elétrica. Aborda a temática das plataformas de controlo existentes e a escolha da plataforma a desenvolver. No fim deste capítulo será explicado os sistemas UPS a utilizar, bem como o seu sistema de controlo.

Quanto ao capítulo 3 é apresentado o trabalho realizado ao nível da plataforma, sendo explicado a adição dos ADCs, a verificação da aquisição de dados simultânea, e a verificação da actualização de dados em simultâneo com a aquisição.

No capítulo 4, o último referente ao trabalho desenvolvido, é exposto o modo como se realiza a programação da plataforma de controlo, e os diferentes modos de processamento do algoritmo na ZedBoard. Este capítulo será finalizado com a exploração da ferramenta *SoC Blockset* para programação da ZedBoard, e as diferenças que esta apresenta em relação ao *HDL Coder* e *Embedded Coder*.

Por fim no capítulo 5 serão apresentadas as conclusões do trabalho, bem como as sugestões para trabalhos futuros.

Capítulo 2

Estado da Arte

Este capítulo tem como principal intuito explicar qual a função da plataforma a desenvolver bem como o seu enquadramento nos sistemas de eletrónica de potência. Na primeira secção será explicado o funcionamento dos sistemas de conversão de energia eléctrica. Seguidamente na secção 2.2 será falado das plataformas de controlo, e como se procedeu à escolha da plataforma para o trabalho a desenvolver.

Verificado o enquadramento das plataformas de controlo e os sistemas onde estas são utilizadas, passar-se-à a um sistema concreto, nomeadamente, uma fonte de alimentação ininterrupta (*Uninterruptible Power Source* - UPS) e os seus constituintes.

2.1 Sistema de Conversão de Energia Eléctrica

Um sistema de conversão de energia eléctrica é constituído por três partes, representadas na figura 2.1. É constituído por uma componente de eletrónica de potência, que converte os níveis de corrente e tensão da fonte (entrada) para os valores pretendidos na carga (saída) e permite ainda o controlo do fluxo energético em função do que é necessário. Existe uma zona de medição que, através de sensores ligados à carga, permite obter os valores instantâneos das grandezas eléctricas necessárias, enviando-os para a placa de controlo. Finalmente a placa controladora que executa o algoritmo de controlo, enviando os pulsos de activação para os semicondutores do conversor de eletrónica de potência, permitindo variar o fluxo de energia consoante os parâmetros do algoritmo.

A zona de eletrónica de potência é constituída por todo o sistema eléctrico e o conversor de potência. O conversor de potência é responsável por interligar ambas as partes do sistema, permitindo que estas possam ter níveis e tipos de correntes e tensões diferentes. Existem quatro tipos de conversores de potência, distintos em função das formas de onda à entrada e saída do conversor. Contudo, todos têm em comum o facto de ser possível controlar o fluxo de energia entre os dois lados do conversor. Os diferentes conversores são:

AC-DC ou Retificador Recebe tensões e correntes alternadas, transformando em tensões e correntes contínuas.

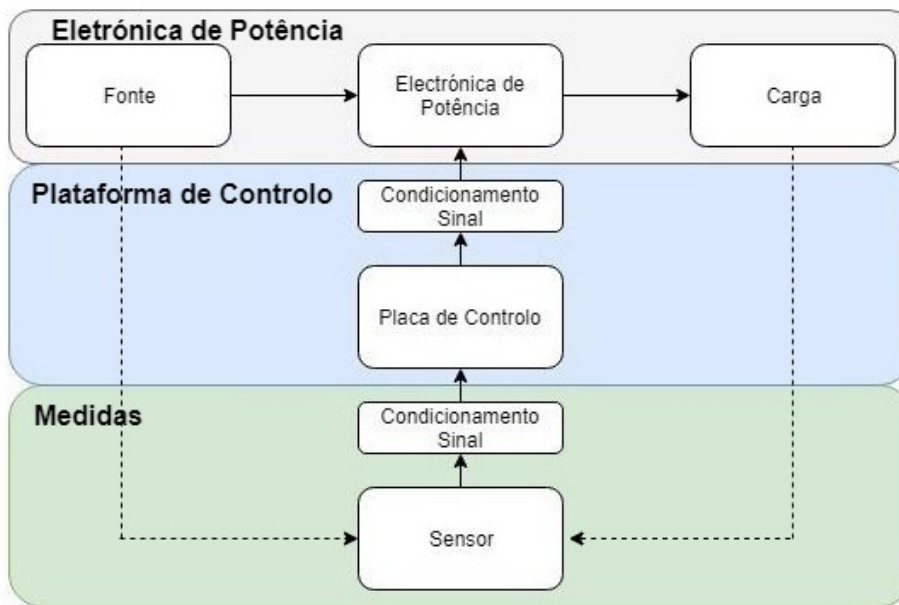


Figura 2.1 Esquema de um sistema de conversão de energia elétrica.

DC-AC ou Inversor Inversamente ao que acontece no retificador, o inversor converte tensões contínuas em tensões alternadas.

DC-DC Permite a alteração dos níveis de tensão entre os dois pontos do conversor, de forma a permitir a interligação de sistemas com diferentes tensões DC.

AC-AC Modifica os níveis de tensão alternada, possibilitando a ligação entre dois sistemas de tensões diferentes.

Na zona azul, da figura 2.1 está situada a placa de controle, ou plataforma digital, onde o algoritmo é executado. O algoritmo de controle envia os sinais para o conversor de eletrônica de potência em função dos parâmetros impostos no controle. Usualmente é necessário ajustar os sinais da placa de controle, denominado de condicionamento de sinal, pois os sinais recebidos pela eletrônica de potência têm valores de tensão superiores. Para que tudo isto se torne possível são necessárias medidas provenientes dos sensores.

Na parte de medidas, representada a verde, os sensores efetuam as medições de corrente e tensão, permitindo que a placa de controle atue corretamente no conversor de potência em função do algoritmo de controle. O condicionamento de sinal é necessário uma vez que os valores de tensão e corrente adquiridos pelos sensores podem ser diferentes dos valores aceites pela placa de controle, pelo que têm de ser ajustados para valores compatíveis com os ADC da plataforma de controle.

Durante a dissertação, apenas se atuará na plataforma de controle, desenvolvendo uma solução baseada na Zedboard, sendo que a sua escolha será justificada no ponto 2.2.3. Esta unidade de controle deve ser compatível com o *Matlab/Simulink* e tem como principal objetivo a substituição de placas de desenvolvimento como a ds1103, *MicroLabBox* da *dSPACE* e a TMS320 da *Texas Instruments*. As soluções da *dSpace* são multi propósitos pelo que contêm capacidades superiores

às necessárias, nomeadamente a existência de ADC e entradas e saídas digitais superiores às utilizadas, o que faz aumentar o seu custo. Estas soluções apresentam, como desvantagens para o seu uso em protótipos, elevado custo e elevada dimensão, o que dificulta a sua utilização em sistemas compactos. Já a solução da *Texas*, tem pequenas dimensões a um baixo custo, contudo esta plataforma carece de velocidade de processamento e um maior número de entradas e saídas. Posto isto, a solução a desenvolver deve conter as vantagens das diferentes soluções apresentadas. Isto é, ter um custo reduzido, e pequenas dimensões, sem comprometer excessivamente a capacidade de processamento.

2.1.1 Conversores de Eletrónica de Potência

Com o avanço da tecnologia e o aumento da automatização de processos, houve necessidade de controlo nos equipamentos eletrónicos, o que proporcionou o crescimento dos conversores de potência no mercado.

Os conversores de potência podem ser de vários tipos, com tensões de entrada e de saída alternada ou contínua. Devido a esta flexibilidade, os conversores de potência são utilizados nas mais diversas áreas desde controlo de motores, processos fabris automatizados, sistemas de armazenamento de energia, compensadores de energia reativa ou até mesmo nos computadores e eletrónica de consumo.

2.1.1.1 Conversores Multinível

Os conversores multinível são um dos maiores avanços na eletrónica de potência da última década. Estes conversores apresentam várias vantagens (em relação aos conversores convencionais ou de dois níveis) sendo que, a principal, é a possibilidade da tensão à saída do conversor poder apresentar um maior número de valores distintos (mais de 2 diferentes). Com este aumento do número de níveis é possível melhorar as formas de onda da tensão, aproximando-as a uma sinusóide, e com isto reduzir a distorção harmónica produzida pelos conversores [1]. Outra das vantagens destes conversores é a redução da tensão a que cada semicondutor está sujeito (no caso de um conversor de três níveis os semicondutores estão sujeitos a metade da tensão dos semicondutores de um conversor de dois níveis, quando ligados ao mesmo barramento DC). Esta redução da tensão a que os semicondutores estão sujeitos permite a utilização de conversores em níveis de tensão mais elevados (por ex. média tensão). A principal desvantagem destes sistemas é a necessidade de um maior número de semicondutores que incrementa a complexidade do sistema elétrico e do sistema de controlo. O aumento da complexidade no algoritmo de controlo, faz com que este tipo de conversor necessite de uma elevada velocidade de processamento.

Existem várias tipologias para os conversores multiníveis como é o caso da: *neutral point clamped* (NPC), *cascaded H-bridge* (CHB) e *flying capacitors* (FCs) [2]. Todas as soluções referidas têm o mesmo propósito, o aumento dos níveis de tensão intermédios, sendo que a solução NPC, apresentada na figura 2.2, é a mais frequente.

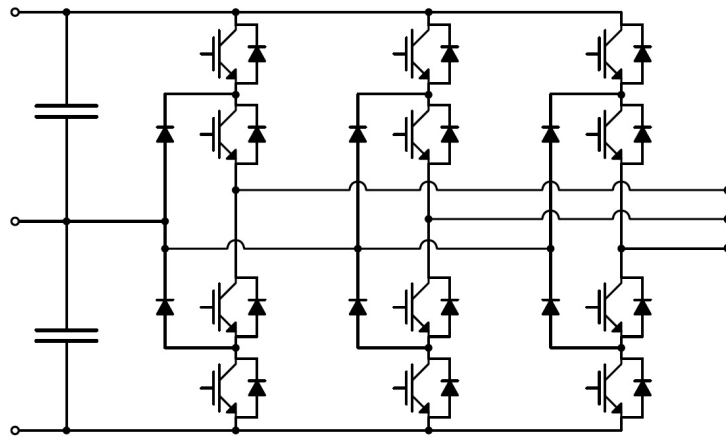


Figura 2.2 Esquema de um inversor NPC de 3 níveis.

O uso de uma montagem NPC permite a redução do número de condensadores necessários. Contudo, esta solução apresenta dificuldades no equilíbrio da tensão dos condensadores do barramento DC. Este problema é um dos desafios dos algoritmos de controlo quando aplicados em conversores multinível do tipo NPC.

2.1.2 Estratégias de Controlo

O controlo de conversores de potência pode ser efetuado de inúmeras formas distintas, sendo algumas delas muito precisas, mas com necessidades de processamento elevadas face à complexidade do seu algoritmo de cálculo. Posto isto, o controlo ideal varia de situação para situação consoante a precisão necessária, limitações da localização do equipamento, e dimensões pretendidas.

Face à evolução da indústria, são desenvolvidas novas estratégias de controlo sendo que os métodos mais comuns são apresentados na figura 2.3.

O controlo linear é um dos métodos mais frequentes do mercado, nomeadamente, o controlo FOC que orienta constantemente o vector do fluxo do motor. Existem outras estratégias lineares que atuam através da diferença entre o valor teórico da situação existente e o valor do sistema adquirido através dos sensores instalados [3, 4].

De forma distinta dos controladores lineares que atuam a uma frequência fixa, o controlo por histerese atua sempre que os valores saem dos limites pretendidos, o que faz com que este controlo não imponha nos semicondutores uma comutação de frequência constante. Os problemas destas oscilações de frequência é que as perdas de comutação deixam de ser previsíveis, pois variam com a frequência, e podem ainda ocorrer comutações em frequências que aumentam o desgaste dos semicondutores [3, 5].

Ainda existem técnicas de controlo preditivo, que como o próprio nome indica consistem num método que prevê o comportamento das variáveis do sistema. Para isso, o controlador calcula o comportamento do sistema para todos os estados de comutação possíveis, escolhendo o que produz o resultado mais próximo do desejado [6, 4].

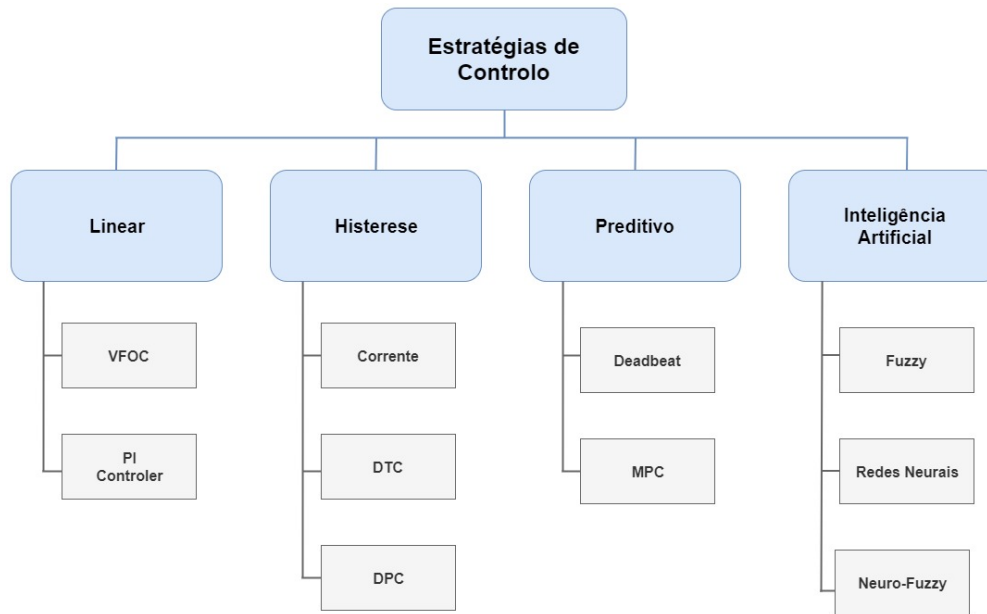


Figura 2.3 Algumas das estratégias de controlo mais comuns.

Os métodos Fuzzy e Neuro-Fuzzy surgiram devido à aplicação de conceitos de inteligência artificial aos algoritmos de controlo. Ambos os métodos consistem na lógica difusa que ao invés de utilizar um lógica binária com dois valores, utiliza valores intermédios [7].

2.2 Plataformas de Controlo

Relativamente às plataformas digitais, atualmente existem no mercado várias soluções de controlo, embora a maioria das soluções multipropósito sejam bastante caras. O facto de estas soluções abrangerem a utilização numa grande variedade de sistemas faz com contêm funcionalidades que acabam por ser desnecessárias para sistemas simples. Com isto, é necessário que unidades de investigação e pequenas empresas tenham de optar por plataformas de prototipagem rápida, que são caras, ou então optar por soluções simples e baratas que abdicam da capacidade de processamento ou da programação rápida e simples.

O projeto na qual esta dissertação se insere, consiste no desenvolvimento de um protótipo de uma UPS, onde é necessário o uso de uma plataforma compacta, com uma capacidade de processamento elevada e compatibilidade com o programa *Simulink*. Das soluções utilizadas pelo grupo, nomeadamente as *DS1103*, *Autobox* e *LabBox* da *dSPACE*, a *TMS320* da *Texas* e a *Compact Rio* e *SingleBoard Rio* da *National Instruments*, nenhuma preenchia as necessidades referidas. A plataforma *SB Rio* é a mais próxima da solução pretendida, contudo apenas pode ser programada através de *software* proprietário (*LabView*) da empresa que desenvolve o *Hardware*, o que implicava a conversão do sistema de controlo já desenvolvido no *Simulink* e a necessidade da compra da licença do *LabView*.

No caso das soluções sem compatibilidade com o *Simulink*, como é o caso da *SB Rio*, é sempre necessário adaptar o algoritmo de controlo desenvolvido. desta forma o código desenvolvido no *Simulink* que permitem a simulação e a verificação do funcionamento do algoritmo desenvolvido, tem de ser convertido para C (*LabView* no caso da *SB Rio*) e seguidamente implementado na plataforma de controlo. Em situações onde a plataforma de controlo contém uma FPGA para aumento do desempenho, é necessário transcrever o código para linguagem HDL (VHDL ou Verilog), que é uma linguagem complexa. Outras agravantes deste tipo de soluções é o tempo necessário para transcrição do código para C (ou HDL no caso de FPGAs) e a possibilidade de aparecimento de erros devido à transcrição do código.

Deste modo procurou-se desenvolver uma plataforma que satisfaça as necessidades do projeto, ou seja, uma plataforma compacta, económica com alta capacidade de processamento que permitisse a programação pelo *Simulink*

2.2.1 Programação pelo *Simulink*

A programação pelo *Simulink* é uma mais valia, pois oferece a possibilidade de implementação do código utilizado em simulação em plataformas de controlo compatíveis com o *Matlab/Simulink*. Esta vantagem é notória em ambiente de investigação pois o a funcionalidade dos algoritmos desenvolvidos tem de ser sempre verificada em ambiente de simulação antes de se proceder à sua implementação. Uma outra vantagem é a implementação direta já que traz facilidade na programação das plataformas, pois não é necessário realizar a conversão do código. Esta conversão é um dos maiores problemas das plataformas sem compatibilidade com *Simulink* (ou outro programa que permita a implementação após a simulação do algoritmo).

Embora o uso do *Simulink* acrescente os custos do programa para possibilitar a configuração da plataforma, as vantagens do programa fez com que se optasse por uma plataforma com compatibilidade com o mesmo. Na prática, os custos do *Simulink* acabam por ser necessários, pois ao nível da investigação este programa é utilizado para desenvolvimento e validação do funcionamento do algoritmo em simulação.

2.2.2 Escolha da Plataforma

Com base nos soluções referidas no ponto 2.2, procurou-se uma solução compacta, que tivesse compatibilidade com o *Simulink* e que fosse uma plataforma versátil, por forma a permitir uma maior possibilidade de expansão. Posto isto, as soluções mais apelativas para os sistema a desenvolver foram as placas Arduino, RaspberryPi e a ZedBoard, apresentadas na tabela 2.1.

Na seleção das soluções teve-se o cuidado de verificar plataformas compatíveis com o *Matlab*, e com custos inferiores às soluções de mercado.

Devido à necessidade de executar o algoritmo a cada $100\mu s$ (valor típico), fez com que a escolha da primeira plataforma a ser eliminada recaísse sobre o *Arduino*. O *RaspberryPi* não é desenvolvido para execução em tempo real, pelo que as entradas e saídas não seriam fáceis de

Tabela 2.1 Comparação entre Várias Soluções de Mercado e Placas de Controle.

Equipamento	Programação	FPGA	Dimensões	Processamento	Custos
ds1103	Simulink	Não	Elevadas	Muito elevado	€€€
LabBox	Simulink	Sim	Elevadas	Muito elevado	€€€
TMS320	Simulink	Não	Reduzidas	Baixo	€
Compact Rio	LabView	sim	Médias	Muito elevado	€€€
SingleBoard Rio	LabView	Sim	Reduzidas	Elevado	€€
ZedBoard	Simulink	Sim	Reduzidas	Elevado	€€
RaspberryPi	Simulink	Não	Reduzidas	Médio	€
Arduino	Simulink	Não	Reduzidas	Baixo	€

utilizar à frequência pretendida, ao contrário da ZedBoard que contém uma FPGA que permite a gestão de periféricos a tempo-real. Com esta vantagem da ZedBoard, seria fácil interligar os periféricos com uma temporização perfeita que não seria possível num processador.

2.2.3 ZedBoard

A ZedBoard da *Digilent* é baseada num *system on chip* (SoC) Zynq-7000 da *Xilinx* que é composto por um processador ARM A9 de 2 núcleos e uma FPGA Artix-7. Esta arquitetura (Zynq) é apresentada na figura 2.4, onde é possível verificar que há possibilidade de ligar vários periféricos a uma unidade de processamento, pelo que as mesmas têm de comunicar entre si para acederem a todos os recursos disponíveis.

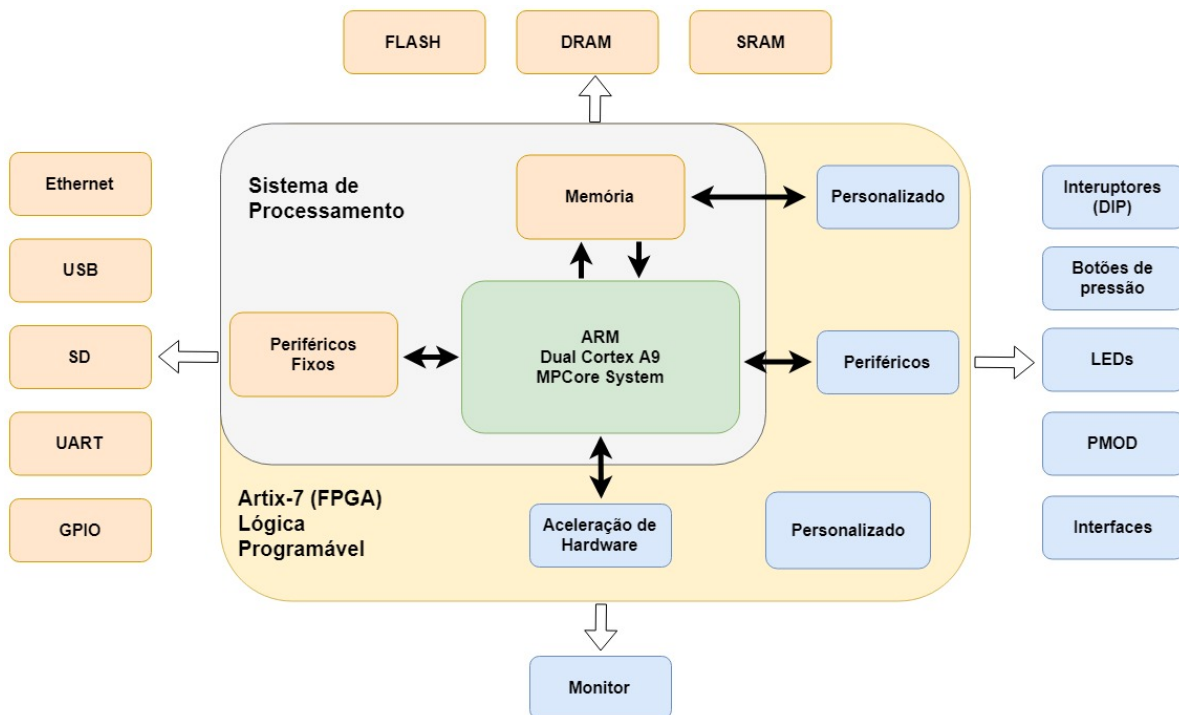


Figura 2.4 Arquitetura de um SoC Zynq.

A comunicação entre processador e a FPGA é realizada pelo protocolo de comunicação AXI4 permitindo que haja troca de informação. Outro ponto é o facto de que embora o processador tenha acesso às entradas e saídas pré estipuladas (USB, Internet, Cartão SD), as entradas e saídas (I/O) digitais configuráveis, apenas podem ser acedidas pela FPGA. Nestes pinos I/O tem-se 30 pinos configuráveis distribuídos por 5 portas PMOD, 8 Leds, 8 interruptores e 7 botões de pressão. Visto esta plataforma não conter ADCs, as soluções são individuais e dependem de projeto para projeto, sendo fundamental a adição de ADCs, na quantidade necessária.

2.3 Sistemas UPS

Devido à dissertação estar inserida no projeto "DRIF" que tem como objetivo o controlo de UPS, a plataforma a desenvolver deve ser focada no controlo deste tipo de sistemas. A nível de desenvolvimento iniciou-se pelo uso de apenas 1 dos conversores dada a complexidade destes sistemas, sendo que após verificação do funcionamento da plataforma será possível implementar todo o sistema.

A figura 2.5 representa o esquema de um sistema UPS, que tem um papel importante no mundo atual. Com o aumento das necessidades dos consumidores, é necessário manter diversos sistemas em funcionamento com elevada qualidade de energia mesmo em casos de falha da rede de energia elétrica. Esta imunidade às falhas pode ser realizada com geradores de socorro ou através de sistemas UPS que utilizam baterias para alimentar as cargas críticas durante falhas da rede elétrica.

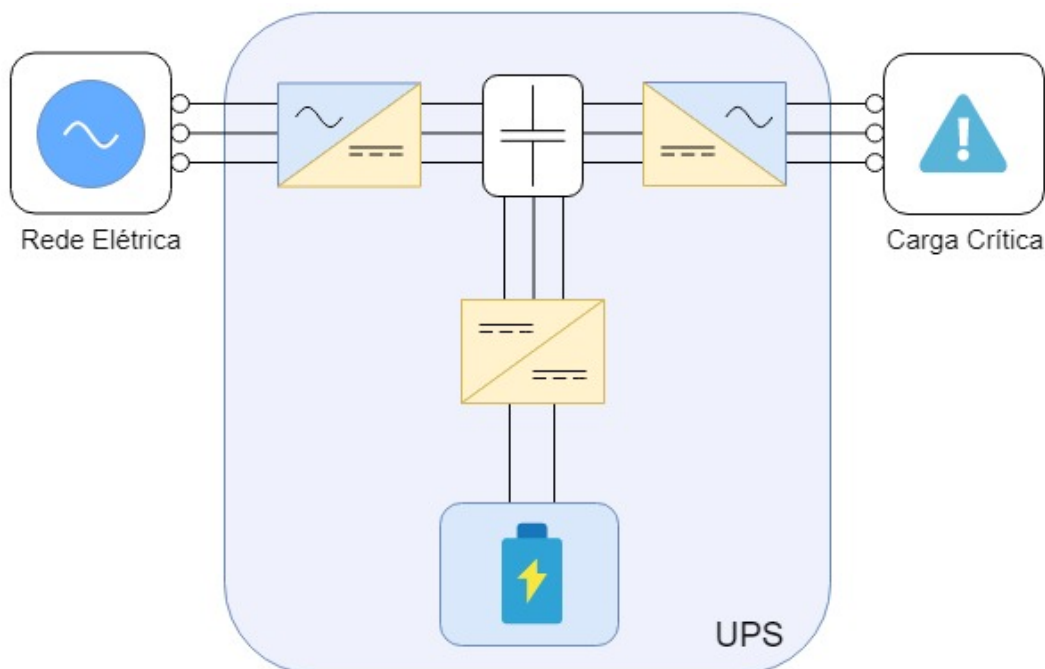


Figura 2.5 Modelo de um sistema UPS.

Os sistemas são compostos por 3 conversores interligados entre si pelo barramento DC (representado na figura 2.5 com um condensador). A UPS tem um retificador ativo que transforma a

energia AC da rede elétrica em DC. Existe ainda um conversor DC-DC que liga este barramento a um conjunto de baterias, responsável pelo ajuste da tensão do sistema de armazenamento de energia nas baterias. Por fim, existe o inversor que transfere a energia existente no barramento DC para as cargas críticas.

Na prática, o sistema absorve energia da rede suficiente para as cargas críticas e mantém as baterias carregadas. Em caso de falha da rede elétrica, o conversor DC-DC envia a energia armazenada nas baterias para o barramento DC. Quanto ao inversor, este mantém sempre o mesmo modo de funcionamento, garantindo que a carga crítica é continuamente alimentada por uma onda de tensão de elevada qualidade.

2.3.1 Conversores Multinível nas UPS

A utilização de conversores multinível em sistemas UPS proporciona várias vantagens [8, 9], que permitem a melhoria da qualidade da energia através da redução da distorção harmónica quer na corrente absorvida da rede, quer nas ondas de tensão na carga. Os conversores multinível apresentam melhor qualidade de onda quando comparados a conversores tradicionais (2 níveis) equipados com filtros idênticos, devido à menor distorção harmónica dos mesmos. Deste modo, é possível reduzir significativamente os elementos de filtragem. Esta redução dos filtros permite uma diminuição do peso e volume da UPS (mais de 50% de acordo com [10]).

Os custos dos conversores multinível tornam-se superiores aos custos dos conversores de 2 níveis, devido ao maior número de semicondutores existentes. Contudo, a maior eficiência dos multiníveis, assim como um menor consumo de energia, faz com que sejam uma solução mais apelativa, tendo em conta a vida útil da UPS [11].

As vantagens enunciadas são aplicáveis quer ao retificador quer ao inversor. Nos sistemas UPS é adotada uma configuração *Back-to-Back*.

2.3.2 Controlo Preditivo

Com o aumento de capacidade de processamento a nível de microprocessadores, DSPs (*digital signal processor*), e FPGAs (*field programmable gate array*), assim como a redução dos custos tornou o controlo preditivo cada vez mais apelativo. Dada a sua versatilidade na aplicação de sistemas de eletrónica de potência, o preditivo tem sido uma das escolhas mais atraentes e por isso muito utilizada para investigação [12–14].

O controlo preditivo a utilizar no sistema UPS do projeto onde a dissertação se insere é o controlo preditivo baseado em modelos de estados finitos. Esta técnica de controlo preditivo consiste na predição em cada período de amostragem do valor de variável a controlar para cada estado de comutação possível do conversor, escolhendo a mais favorável. Durante cada ciclo os valores das variáveis são calculados para todos os estados possíveis assim como a avaliação de uma função objetivo. O estado que minimiza o valor da função objetivo, e consequentemente minimiza o erro em relação às referências, é o estado aplicado ao conversor. A única grande desvantagem deste

método de controlo é a sua elevada carga computacional [15]. Hoje em dia, devido à evolução das unidades de processamento, é já possível utilizar este tipo de controlo em sistemas reais. Ainda assim, em sistemas complexos constituídos por conversores multinível, pode ser necessário utilizar frequências de amostragem menores, que têm um impacto direto no desempenho do sistema. Outra desvantagem deste método é a necessidade de um modelo matemático preciso, de forma a maximizar o desempenho do sistema, que variam a sua complexidade em função do sistema. Além do mais, caso não haja uma correta definição dos parâmetros ou caso estes variem ao longo do tempo faz com que ocorra uma redução no desempenho do sistema [16].

O sistema pode considerar uma única função objetivo g ou várias funções $g_1, g_2, g_3, \dots, g_n$. Neste segundo caso a função objetivo acaba por ser uma combinação das várias funções

$$g = W_1g_1 + W_2g_2 + W_3g_3 + \dots + W_n g_n$$

onde W_1, W_2, \dots, W_n são os pesos de cada função objetivo, e definem qual a importância relativa de cada uma das funções. Nos pontos seguintes serão expostos os objetivos do retificador e do inversor do sistema UPS, que por sua vez são aplicados numa função objetivo. Esta será uma breve exposição das variáveis a considerar, sendo que não terá foco no modelo matemático (visto não ser objetivo do trabalho a desenvolver).

2.3.3 Retificador Ativo

O retificador do sistema a utilizar, é um retificador trifásico de três níveis de tipologia NPC conforme está representado na figura 2.6. Os retificadores ativos têm como vantagens a redução da distorção harmónica e o aumento da eficácia quando comparados com os retificadores a díodos. A principal vantagem é a redução da distorção harmónica, sendo que o principal objetivo do sistema de controlo é garantir que a forma de onda da corrente do lado da rede é aproximadamente sinusoidal (com fator de potência unitário¹), mantendo a tensão desejada no barramento DC.

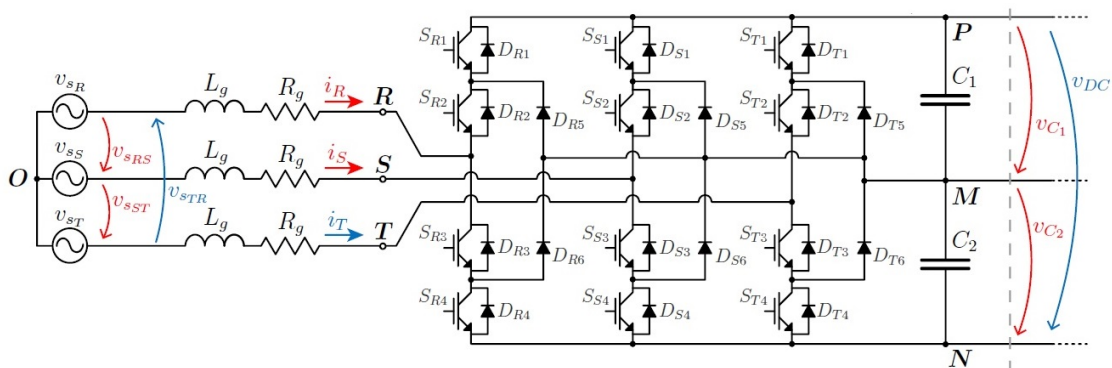


Figura 2.6 Retificador trifásico multinível do tipo NPC.

¹Com exceção do caso de ser pretendido utilizar o Retificador para compensar o fator de potência

Ao aplicar o algoritmo de controlo ao retificador ativo é necessário efetuar 6 medições assinaladas a vermelho na figura 2.6. Na figura, são também apresentados dados a azul, que são necessários para a aplicação do algoritmo, contudo podem ser calculados posteriormente, minimizando os custos com sensores e ADC. Isto apenas é possível devido a ser utilizado um sistema trifásico sem neutro, pelo que a soma das correntes e tensões é sempre nula. Com a medição dos pontos do sistema identificados é possível o controlo preditivo calcular os 27 estados possíveis, que produzem 19 tensões distintas. Os estados resultam da combinação dos 3 estados possíveis de cada um dos braços do conversor (correspondente a uma fase). Os estados estão representados na tabela 2.2 sendo definidos como 1, 0 e -1, e onde o valor x corresponde à fase.

Tabela 2.2 Diferentes estados de cada braço do retificador.

Estado	Condutor	Tensão Fase-Ponto Médio
1	S_{X_1} e S_{X_2}	$vc_1 (\approx v_{DC}/2)$
0	S_{X_2} e S_{X_3}	0
-1	S_{X_3} e S_{X_4}	$-vc_2 (\approx -v_{DC}/2)$

Das 3 possibilidades de cada braço surgem 27 estados de comutação ($3 \times 3 \times 3 = 27$). Contudo, alguns deles aplicam na saída o mesmo vector de tensão. Na figura 2.7, estão representado os 27 estados no plano $\alpha\beta$, onde apenas existem 19 vértices (19 estados distintos). Nesta figura é notória a redundância dos conversores NPC multinível, onde diferentes estados originam vectores de tensão iguais.

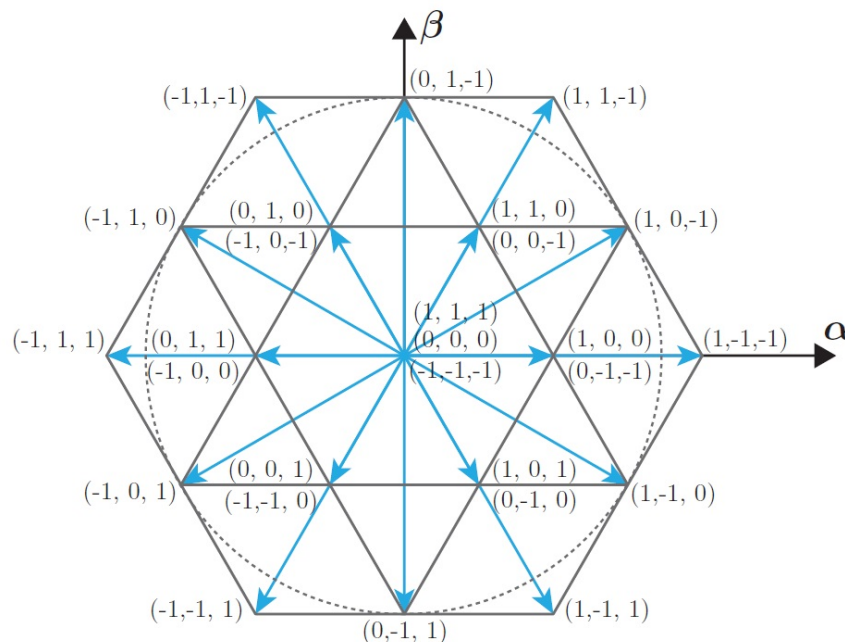


Figura 2.7 Diferentes vectores de tensão representados no plano $\alpha\beta$.

Após o cálculo dos valores esperados no sistema através do algoritmo em função do estado aplicado no retificador, é verificado qual o estado a aplicar consoante a função objetivo. No caso do retificador, a função objetivo tem como ideal os seguintes pressupostos:

- Minimizar o erro da corrente na rede;
- Minimizar o desequilíbrio de tensão entre os condensadores do barramento DC;
- Minimizar a frequência de comutação.

2.3.4 Inversor

O inversor presente no sistema UPS a utilizar é um inversor trifásico de três níveis de tecnologia NPC, assim como o retificador. No caso dos inversores e retificadores multinível, estes têm a mesma estrutura física, pelo que o algoritmo de controlo que atua nos semicondutores permite que o conversor funcione como retificador ou inversor. Contudo, as medidas necessárias para cada um dos sistemas são conforme ilustrado na figura 2.8. As variáveis assinaladas a vermelho são medidas por sensores, pelo que as azul são calculados pelo algoritmo de controlo.

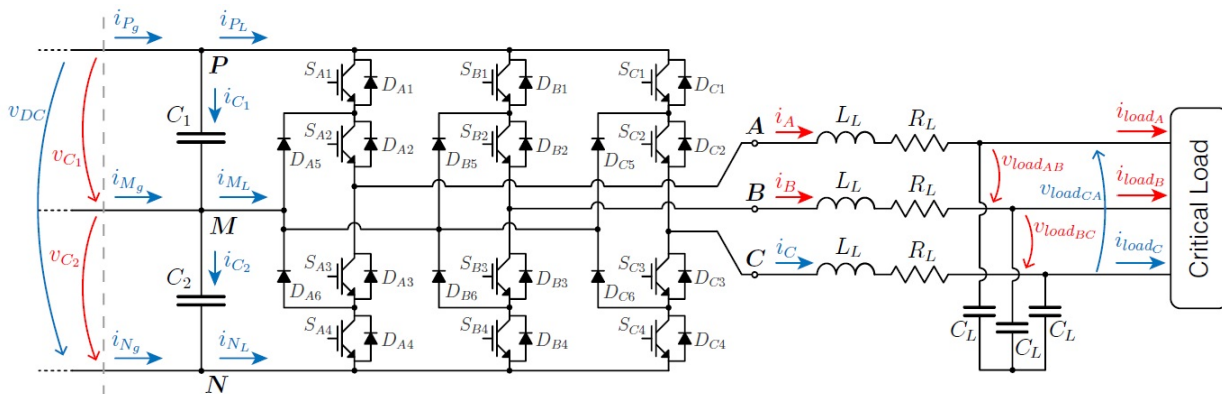


Figura 2.8 Inversor trifásico multinível do tipo NPC.

Assim como o retificador, o inversor NPC de 3 níveis também contem 27 estados, sendo que 19 deles são distintos. Como no caso anterior, o algoritmo de controlo a cada ciclo com base nos valores medidos permite o cálculo de diversos parâmetros. Seguidamente, com base na sua função objetivo verifica que estado minimiza o erro da função objetivo.

O inversor apresenta três objetivos principais:

- Minimizar o erro da tensão na carga;
- Minimizar o desequilíbrio de tensão entre os condensadores do barramento DC;
- Minimizar a frequência de comutação.

2.3.5 Requisitos do Sistema a Utilizar

O sistema a testar, já previamente desenvolvido, é composto por dois conversores numa montagem tipo *Back-to-Back*, projetado para utilização em sistemas UPS. O esquema do sistema previamente desenvolvido está presente na figura 2.9, sendo que ao longo dos testes ao sistema apenas será utilizado um conversor.

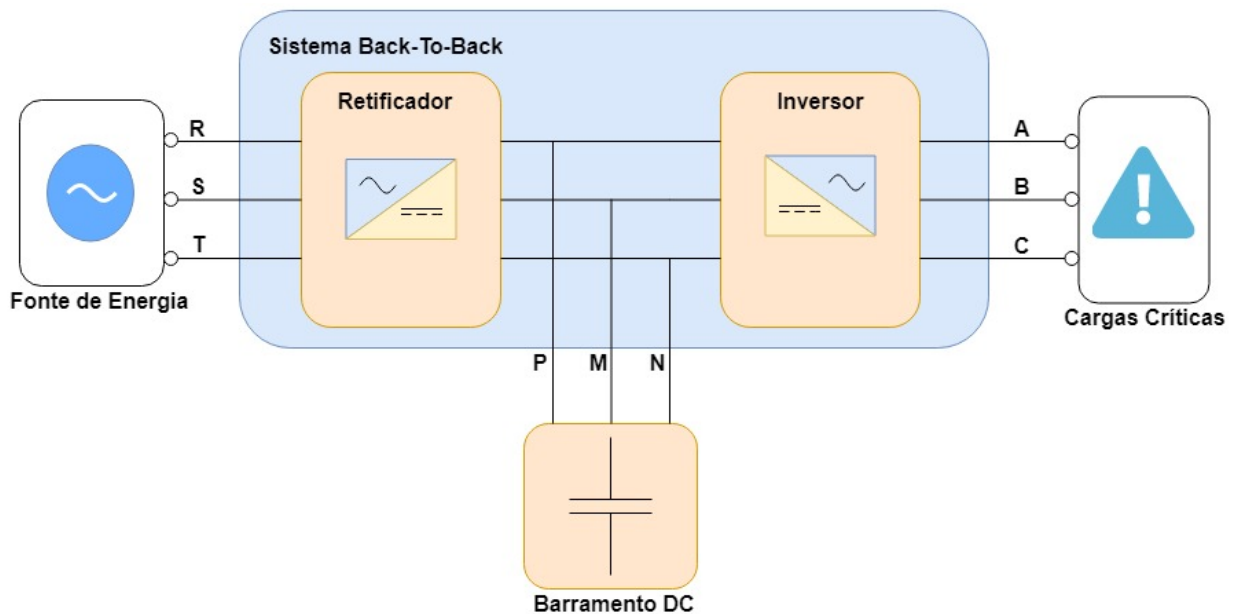


Figura 2.9 Sistema *Back-to-Back* a utilizar.

Esta utilização de apenas um conversor (retificador ou inversor) apenas é possível pois no sistema *Back-to-Back* todos os terminais estão acessíveis, embora estejam embutidos numa caixa. No caso do uso de apenas um conversor a funcionar como inversor ou retificador, é necessário ter em atenção para qual dos conversores são enviados os pulsos.

Para os testes com a plataforma de controlo a desenvolver optou-se por controlar apenas um dos conversores. Para aplicação do algoritmo de controlo, também já desenvolvido (no *Simulink*), é necessário que a ZedBoard possibilite a leitura dos sensores e envie os sinais de controlo necessários. As entradas e saídas necessárias estão representadas na tabela 2.3, sendo que é necessário deixar alguma margem nestas entradas e saídas de forma a possibilitar a adição de novos componentes.

Tabela 2.3 Entradas e saídas do algoritmo de controlo.

Conversor	Entradas	Saídas
Retificador	6(ADCs)	12(Sinais de controlo)
Inversor	8(ADCs)	12(Sinais de controlo)
UPS	12(ADCs) ²	24(Sinais de controlo)

²Pois nesta situação os 2 ADC do barramento DC, são utilizados pelo retificador e pelo inversor.

Estado da Arte

Para o ideal funcionamento do sistema é importante ter em conta a funcionalidade das entradas e saídas como também é necessário que a plataforma a desenvolver consiga:

- Adquirir os valores das entradas analógicas em simultâneo;
- Atualizar os valores das saídas no instante em que adquire os dados.

Estas restrições são essenciais para o funcionamento do algoritmo, isto porque, caso os valores não sejam adquiridos simultaneamente, os valores de corrente e de tensão calculados deixam de ser precisos. O segundo ponto, é um requisito do controlo preditivo, de forma a maximizar a sua eficiência. Caso o segundo ponto não seja garantido, o sistema consegue funcionar, contudo a qualidade do algoritmo de controlo será severamente prejudicada.

Capítulo 3

Desenvolvimento da Plataforma

Conforme foi referido anteriormente, para o desenvolvimento da plataforma digital, é necessário o conhecimento do sistema a controlar de forma a saber as entradas e as saídas necessárias. No presente caso foi escolhido um retificador ativo, baseado num conversor NPC de 3 níveis, com um controlo preditivo. De forma a ser possível aplicar o controlo é necessário medir as 4 tensões e as 2 correntes assinaladas na figura 3.1.

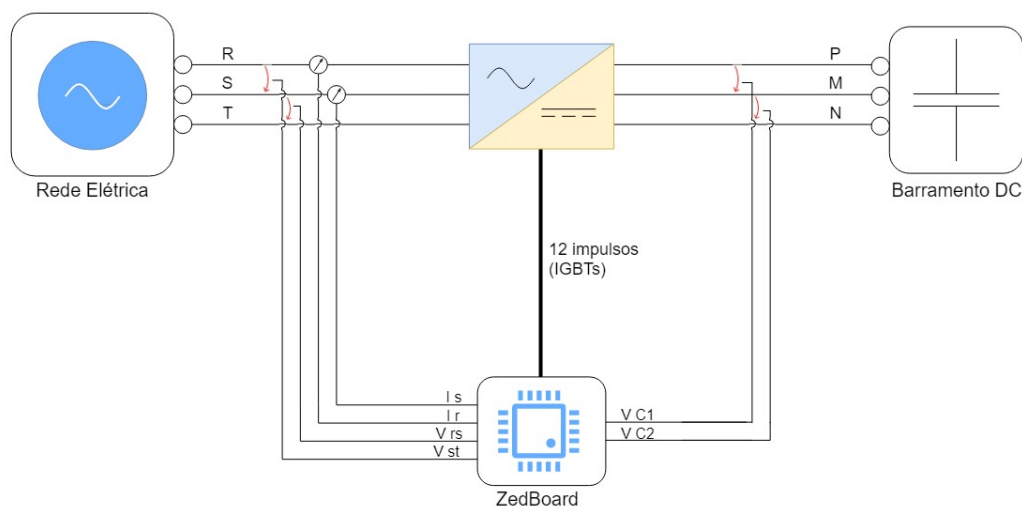


Figura 3.1 Medidas a efetuar no sistema.

Embora a tensão entre as fases TR e corrente da fase T sejam necessárias, estas podem ser calculadas através dos resultados das restantes fases, pois num sistema trifásico sem neutro a soma das 3 tensões compostas e das 3 correntes é nula. Quanto às saídas, são necessários 12 impulsos de comando, um para cada IGBT do retificador NPC de 3 níveis. Posto isto, é necessário que sejam adicionados 6 ADC à ZedBoard de forma a poder efetuar as medições referidas, mantendo 12 saídas disponíveis para emitirem os impulsos para os IGBTs.

3.1 Escolha ADCs

Para a escolha dos ADCs, procurou-se um módulo que fosse compatível com as entradas PMOD da ZedBoard para facilitar a ligação ao sistema. Desta forma, optou-se pelos módulos Pmod AD1 da *Digilent* compostos por 2 ADCs com 12 bits de resolução (valores entre 0 e 4095). O módulo tem 2 conectores conforme é possível verificar na figura 3.2, o conector J1 liga à ZedBoard e no J2 são ligados os sinais externos (2 sinais a medir e alimentação(Facultativo)).

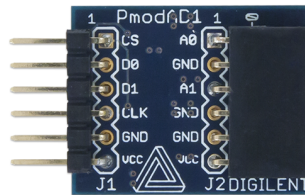


Figura 3.2 Pmod AD1 da *Digilent*.

Como o módulo é desenvolvido para ler sinais entre os 3,3V e 0V é possível ligá-lo diretamente à alimentação existente nas portas PMOD (no valor de 3,3V).

3.1.1 Funcionamento Pmod AD1

O ADC recebe os sinais CS e CLK vindos da ZedBoard, sendo que o primeiro (CS) indica a frequência de aquisição do sinal e o segundo (CLK) a frequência da passagem de dados. Quando o sinal CS faz a transição descendente, o ADC adquire o valor. Após a transição descendente (1 -> 0) é enviado imediatamente o primeiro bit. Seguidamente, a cada transição descendente do sinal de CLK são enviados os 16 bits de dados. Enviados todos os bits, o sinal CS pode retornar a "1" conforme é apresentado na figura 3.3. Caso o sinal CS retorne ao 1 lógico durante a passagem de dados, esta é considerada inválida.

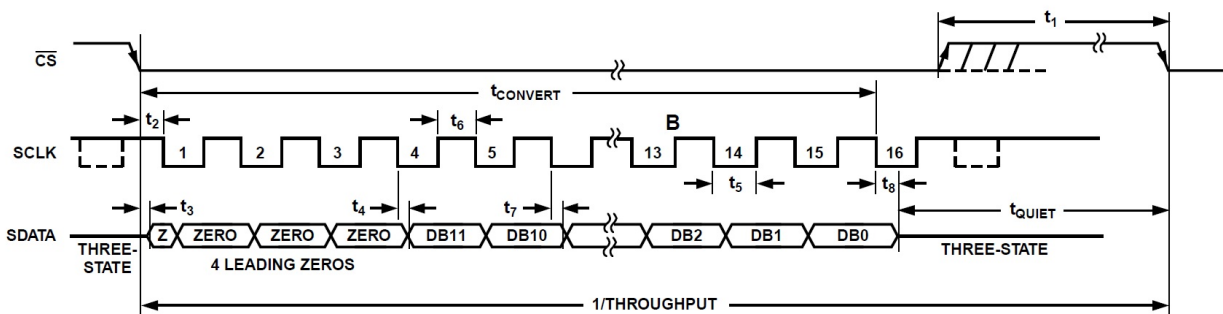


Figura 3.3 Funcionamento transferência de dados do Pmod AD1 da *Digilent*.

Embora o ADC seja de apenas 12 bits de resolução o sinal de dados tem o tamanho de 16 bits, isto porque, devido ao tempo de conversão do sinal os primeiros 4 bits são sempre 0. Quanto à velocidade máxima de aquisição de dados estipulada no *datasheet* é de 1MSPS, ou seja, uma amostra a cada 1µs. Para a velocidade de transferência de dados, é apresentada uma velocidade

máxima de 20MHz, ou seja 50ns, que nunca deve ser excedida. Após a passagem dos dados é necessário um tempo de repouso antes da conversão seguinte, representado na figura 3.1.1, como T_{Quiet} , que deve ter no mínimo 50ns.

3.2 Configuração da Plataforma

Conhecidas as necessidades para a plataforma a desenvolver, nomeadamente a adição de 6 ADCs, a aquisição de dados simultânea e a atualização das saídas no mesmo instante que a aquisição de dados, passou-se à configuração da plataforma. Para a configuração da ZedBoard foram utilizados os programas *Simulink* e *Vivado*. De forma a facilitar a instalação e utilização dos programas em conjunto documentou-se um manual de instalação e utilização presentes nos apêndices A e B.

Durante a configuração dos ADCs, procurou-se garantir os vários parâmetros, nomeadamente, aquisição simultânea dos diversos ADCs, a atualização de saídas em simultâneo com os ADCs e verificar qual a frequência máxima para a transferência de dados.

3.2.1 Módulos dos Pmod AD1

Para a configuração dos módulos Pmod AD1 da *Digilent*, utilizou-se o programa *Vivado*, de forma a que as entradas dos ADCs fossem adicionadas à placa. O procedimento utilizado foi o descrito no manual de utilização, presente no apêndice B. Embora o desenvolvimento de código em VHDL para adição dos módulos à Zedboard seja complexo devido à linguagem utilizada, a existência de uma biblioteca da *Digilent* com um código VHDL já elaborado para os módulos a utilizar presente no apêndice C.1, facilitou esta tarefa.

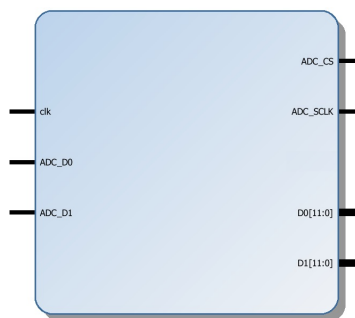


Figura 3.4 Módulo Vivado Pmod AD1.

Quanto ao funcionamento dos módulos em VHDL, estes recebem um sinal de CLK que é convertido nos sinais SCLK e CS, sendo divisores inteiros do sinal de CLK. A frequência do sinal de SCLK é o resultado da divisão da frequência do sinal CLK pelo tamanho do vetor ce_sr+1 . Já o sinal CS tem uma frequência igual à divisão da frequência do SCLK pelo tamanho do vetor $sequencer_shift_reg+1$. Estes dois sinais têm cada um deles uma função específica no código fornecido pela *Digilent*, o $sequencer_shift_reg$ é o número de bits enviados a cada interação e o ce_sr é o número de bits para tempo de repouso do ADC. É de salientar que, o número de bits enviados a

Desenvolvimento da Plataforma

cada interação contempla, os primeiros 4 bits a zero para espera da conversão do sinal, os 12 bits da conversão, os bits de tempo de repouso e por último bits inutilizados (estes bits embora não sejam utilizados são importantes para maximização da velocidade de transferência de dados sem alteração da frequência da aquisição como será demonstrado no ponto 3.2.4).

Após a adaptação dos módulos Pmod AD1 à ZedBoard no programa *Vivado* para os ADCs serem tratados como uma entrada da placa, foi criada uma *Board Definition*, conforme está explicado no apêndice B. Em seguida, foram realizadas as verificações de funcionamento do sistema.

3.2.2 Verificação da aquisição de dados em simultâneo

Para garantir que o sistema adquire os dados em simultâneo, desenvolveu-se uma configuração de placa (*Board Definition*), no programa *Vivado*, onde foram adicionados os blocos dos ADCs fornecidos pela biblioteca da *Digilent*. O esquema da figura 3.5 apresenta os ADCs e contém o bloco *datacapture* gerado pelo *Matlab* de forma a permitir o envio de dados da ZedBoard para o *Matlab/Simulink*.

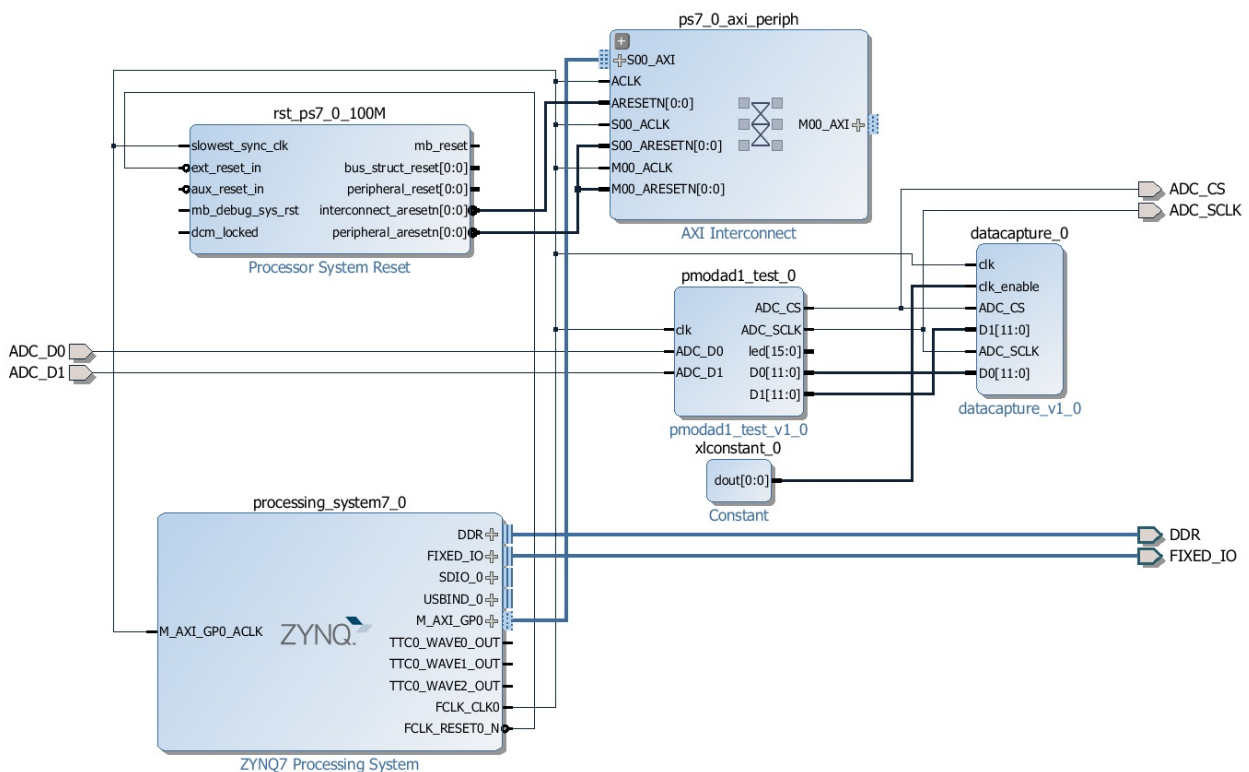


Figura 3.5 *Board Definition* para 2 ADC PMOD AD1.

O bloco *datacapture* é gerado através do comando *generateFPGADDataCaptureIP* na linha de comandos do *Matlab*, e posteriormente anexado na *Board Definition* no *Vivado*. Este bloco tem como função a aquisição de diversos dados na FPGA num buffer, posteriormente enviando-os para o processador (ARM) permitindo exportar os dados para o *Simulink*. Após gerar o código *bitstream* do modelo da figura 3.5, este foi implementado na Zedboard. O algoritmo em questão efetua a

leitura de 2 ADCs em simultâneo e guarda os valores no bloco de *datacapture*. De forma a adquirir os dados, abre-se o modelo de *Simulink* e executar durante o tempo pretendido de forma a adquirir os valores guardados no bloco *datacapture ip*.

A figura 3.6 mostra os dados adquiridos ao longo do intervalo de tempo. No presente teste, o mesmo sinal foi ligado aos 2 ADCs, pelo que todas as leituras deveriam apresentar os mesmos valores.

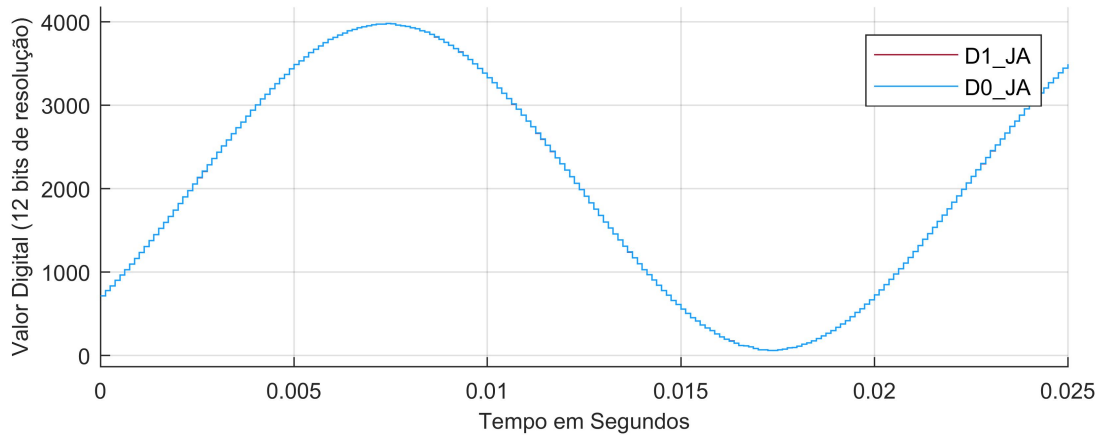


Figura 3.6 Valores adquiridos pelos 2 ADCs de um módulo, com uma frequência de amostragem de 8kHz.

Conforme esperado, os sinais da figura 3.6 estão sobrepostos o que indica que os 2 ADCs, 2 presentes nos módulos Pmod AD1 da *Digilent*, adquiriram simultaneamente as amostras. Ao ampliar para uma análise mais fina, é notório algumas variações em torno de 2 bits, embora pouco notórias na figura 3.7. Estas variações são provocadas essencialmente por ruído e variações de conversão.

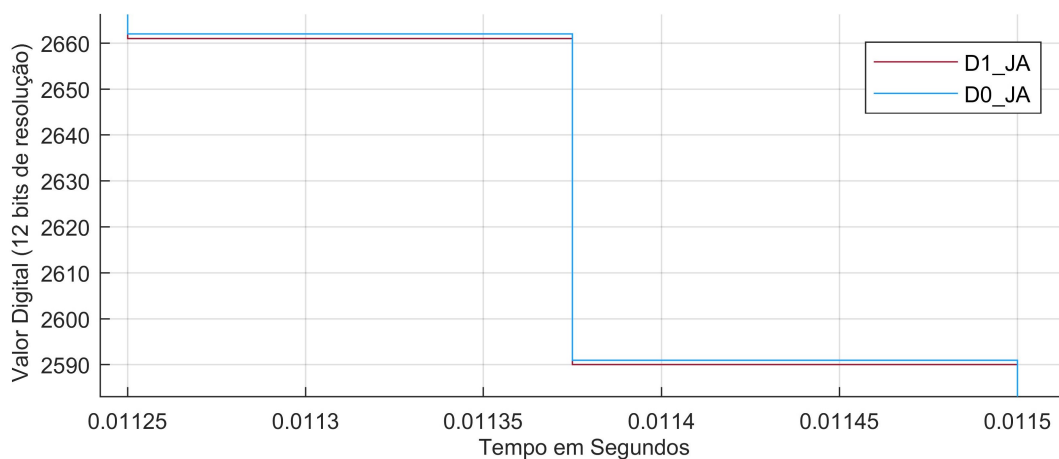


Figura 3.7 Ampliação dos valores adquiridos pelos 2 ADCs de um módulo, com uma frequência de amostragem de 8kHz.

Foi verificada a aquisição simultânea, que garante que os dados são adquiridos no mesmo instante, fundamental para o funcionamento correto do controlo preditivo a utilizar.

3.2.3 Aquisição de Dados e Atualização de Saídas em Simultâneo

De forma a garantir que as saídas são atualizadas em simultâneo com a aquisição de dados, caracterizada pela transição descendente do sinal CS, desenvolveu-se o código em VHDL para esta função (presente no apêndice C.2). O código é um simples flip-flop (FF) do tipo D com ativação na transição descendente, ou seja, sempre que acontece uma transição descendente o sinal à entrada do FF é passado para a saída. Desenvolvido o código, o mesmo foi adicionado ao *Vivado*, obtendo o módulo da figura 3.8.

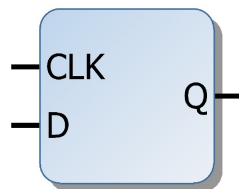


Figura 3.8 Módulo FF no *Vivado*.

Após a adição do módulo ao *Vivado* foi possível realizar a simulação, representada na figura 3.9, onde foi verificado que o código VHDL funcionava como pretendido. Na figura estão representados os sinais de *Clock* (CLK), entrada (D) e saída (Q). Durante a simulação no *Vivado* colocou-se um sinal aleatório na entrada e sempre que o sinal de CLK efetuava a transição negativa, a saída atualizava o seu valor para o existente na entradas. Confirmado, que o sistema funcionava conforme pretendido prosseguiu-se para os testes práticos que confirmaram os resultados obtidos em simulação.

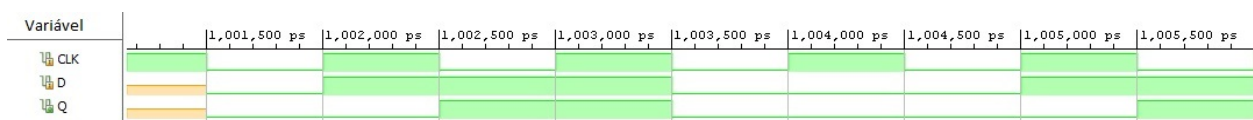


Figura 3.9 Simulação ao código VHDL do FF desenvolvido.

3.2.4 Velocidades de Transferência de Dados

A velocidade de transferência de dados é um ponto muito importante de verificar, pois permite aos ADCs minimizar o tempo que demoram a disponibilizar os 12bits correspondentes ao valor de leitura. Neste ponto é importante maximizar a velocidade, sem afectar o funcionamento dos ADCs. A cada ciclo (T_s) é pretendido efetuar uma nova leitura, processamento de dados e atualização de valores no fim do ciclo, conforme está representado na figura 3.10, pelo que quanto menor o tempo de envio de dados maior o tempo para cálculo do algoritmo. Para a redução deste tempo é necessário aumentar a frequência do sinal de envio de dados (SCLK), mantendo a frequência de

aquisição de dados (CS), sendo que para isso é necessário aumentar o número de bits inutilizados referidos no ponto 3.2.1.

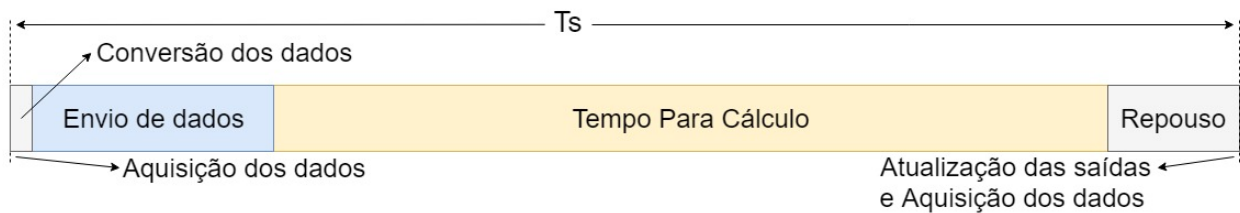


Figura 3.10 Esquema do envio de dados pelo Pmod AD1.

O incremento do vetor *sequencer_shift_reg* contribuiu para o aumento da diferença entre a frequência do SCLK e do CS. Após este incremento, verificou-se que o aumento da frequência reduzia o tempo de envio de dados. Contudo, ao atingir-se a gama dos 20 MHz (valor máximo apresentado no *datasheet*) o ADC apresentou problemas na conversão de dados apresentando apenas 11 bits de resolução em vez dos 12 bits de resolução máxima. Embora fosse expectável que este problema ocorresse devido aos ADCs não terem tempo suficiente para conversão dos dados, verificou-se que o problema situava-se nas capacidades da FPGA. Após medição do sinal de SCLK à frequência de 20 MHz verificou-se que o mesmo não tinha tempo de estabilizar, devido às limitações de frequência das portas de saída da FPGA. Este problema é apresentado na figura 3.11 e devido à forma de onda a 20 MHz ser pouco confiável reduziu-se o sinal para 2 MHz de forma a obter uma onda quadrada que garanta o correto funcionamento dos ADCs.

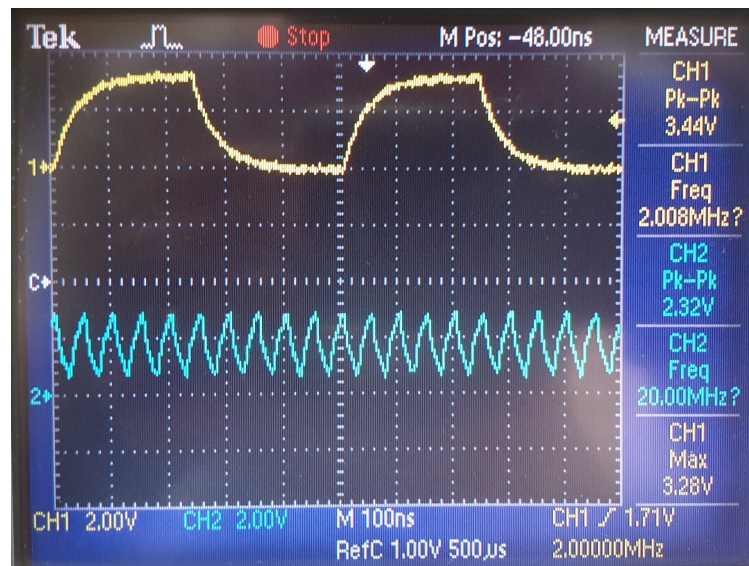


Figura 3.11 Forma de onda do sinal SCLK a 2 MHz e a 20 MHz.

Este problema do sinal enviado pela ZedBoard é notório na figura 3.11, onde a onda amarela a 2 MHz apresenta alguma deformação, mas tem tempo de atingir os valores de pico (0 V e 3,3V). Já no caso da onda a 20 MHz representada a azul, isto já não acontece. Nesta situação a excursão do sinal é apenas 2,3 V quando deveria ser 3,3 V. Por este motivo, os ADCs não conseguem detetar

corretamente o sinal de *clock*, prejudicando o seu funcionamento. Devido a esta limitação da frequência de saída das portas PMOD da ZedBoard, optou-se por uma velocidade de transferência de dados a 2 MHz pois foi uma das maiores velocidades que permitiam o funcionamento estável dos ADCs.

Esta limitação que implica utilizar a frequência do sinal de SCLK a 2 MHz em vez dos 20 MHz (frequência máxima apresentada no *Datasheet*) faz com que o tempo que o ADC leve a disponibilizar os dados suba de $0,8 \mu\text{s}$ para $8 \mu\text{s}$ (situação com o SCLK a 2 MHz). De forma a minimizar o tempo de disponibilização dos dados, pode ser utilizado um oscilador externo de forma a reduzir o tempo para os $0,8 \mu\text{s}$.

3.2.5 Adição de ADC's ao IP CORE

Os IP Cores são blocos de construção utilizados em projetos para programação de FPGA. No presente caso, estes blocos são utilizados no *Vivado*, e uma vez desenvolvidos, podem ser reutilizados em diferentes projetos.

Para adicionar mais ADCs ao IP core fornecido pela *Digilent* alterou-se os parâmetros indicados no apêndice C.3 para o número de ADCs necessários. Os ADCs foram adicionados dois a dois (número de ADC existentes em cada módulo), sendo que todos partilhavam o mesmo sinal para transferência de dados e aquisição (sinal SCLK e CS). Depois da adição de um módulo repetiu-se o procedimento da alínea 3.2.2, adquirindo os dados presentes na figura 3.12. Como esperado foram obtidos sinais sobrepostos o que garante que todos os ADCs adquirem em simultâneo os valores independentemente do número de ADCs colocados.

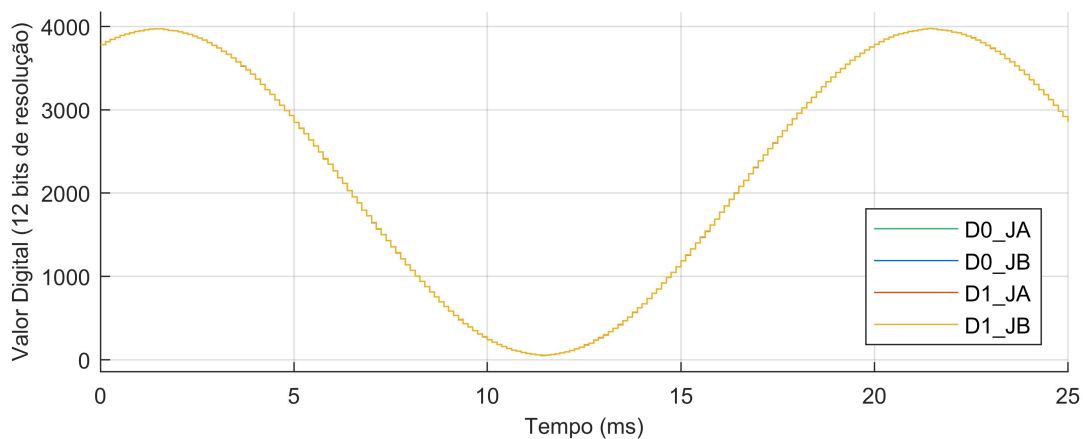


Figura 3.12 Valor adquiridos pelos 4 ADC, a uma amostragem de 8kHz.

Confirmado que as alterações ao código VHDL não provocaram qualquer interferência no funcionamento do código, prosseguiu-se com a adição de ADCs para o número necessário. Na utilização de múltiplos ADCs, caso estejam ligados diretamente nas portas PMOD da ZedBoard é necessário que cada porta PMOD tenha o sinal de SCLK e CS do seu Pmod AD1 ligado à saída do módulo de ADC no *Vivado*.

Capítulo 4

Programação da Plataforma de Controlo

Conforme já explicado, a ZedBoard consiste apenas numa unidade de processamento, pelo que ao ser utilizada como uma unidade de controlo necessita de funções das quais não dispõe. Ao serem adicionados periféricos, a ZedBoard consegue ultrapassar estas limitações. Contudo, é necessário saber qual o projeto a realizar de forma a estipular quais e quantos componentes expansíveis serão necessários adicionar à ZedBoard. No ponto 2.3, já foi identificado o sistema a controlar e no capítulo anterior foram identificados os componentes periféricos necessários.

Neste capítulo, será explicada a importância das proteções e de que forma são aplicadas na ZedBoard, bem como as diferentes formas de processar o algoritmo de controlo. Estas diferentes formas de processar o algoritmo apresentam características diferentes, sendo que cada uma delas tem as suas vantagens e, por isso, diferentes formas de aplicar o algoritmo, conforme será demonstrado ao longo deste capítulo.

4.1 Processamento do Algoritmo

Na ZedBoard, tal como em outros SoCs híbridos, o processamento de informação pode ser realizado de duas formas distintas, devido à existência de um processador (ARM) e de uma FPGA, sendo que cada uma delas tem as suas vantagens e desvantagens. A FPGA permite a execução do código num tempo de cálculo baixo, na ordem das dezenas a centenas de nanossegundos. As FPGAs têm como principal limitação a área programável limitada do dispositivo, o que faz com que algoritmos muito complexos não possam ser executados unicamente numa FPGA. Por outro lado, um processador consegue correr qualquer tipo de código independentemente da complexidade, sendo que o tempo de execução aumenta proporcionalmente.

Na estrutura da placa, apresentada no ponto 2.2.3, as entradas e saídas são sempre manipuladas pela FPGA, o que faz com que a gestão das mesmas seja mais rápida e permita uma temporização extremamente precisa dos eventos de leitura e escrita, assim como uma perfeita sincronização entre eles. Ao nível dos métodos de implementação estes podem ser divididos em:

- **Processamento apenas na FPGA**

- **Processamento apenas¹ no ARM**
- **Processamento Híbrido**

Ao longo do capítulo serão expostas as vantagens e desvantagens de cada uma destas formas de processamento, assim como as dificuldades de implementação de cada uma delas. Os programas utilizados para aplicação do algoritmo de controlo na plataforma são os *Add-Ons: Embedded Coder e HDL Coder* do *Simulink*. Estes *Add-Ons* interligam o modelo do *Simulink* ao *Vivado* que permite a sua implementação, quer no ARM quer na FPGA (duas unidades de processamento da ZedBoard).

A utilização do *Embedded Coder* e do *HDL Coder* permite converter o código desenvolvido no *Simulink* para um modelo equivalente no *Vivado*, de forma a permitir a configuração da FPGA presente na ZedBoard. Este método consiste no desenvolvimento de dois modelos, um modelo para a FPGA e outro para o processador. O processo de programação da ZedBoard é esquematicamente representado na figura 4.1. Do modelo desenvolvido inicialmente em *Simulink*, o *HDL Coder* gera dois modelos: Um modelo no *Vivado*, onde é produzido o ficheiro para configuração da FPGA (*Bitstream*), e um modelo de *Simulink* para o código a implementar no processador. O modelo do *Simulink* do processador é convertido no ficheiro para configuração do processador. Ambos os ficheiros são enviados para a ZedBoard de forma a programar a mesma com o código desejado. Primeiro deve ser programada a FPGA e só depois programar o processador (ARM). A comunicação entre os dois algoritmos é realizada através de registos do AXI4, que são manualmente atribuídos.

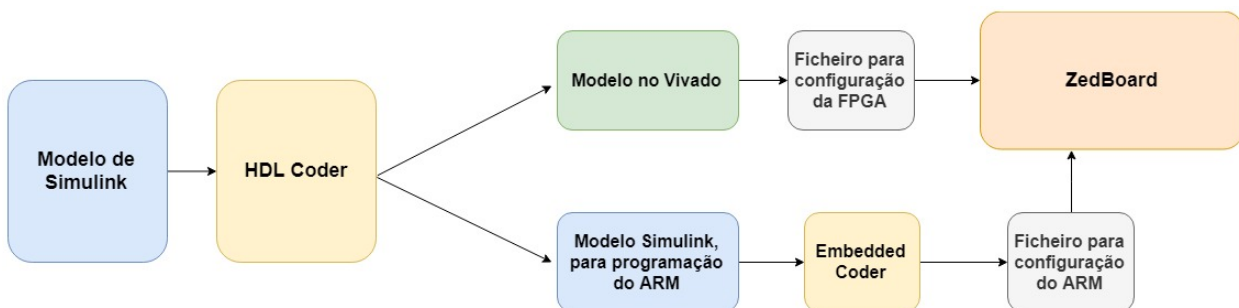


Figura 4.1 Fluxograma da programação ZedBoard pelo *Simulink*.

Nesta fase surgiu o primeiro problema pois o *HDL Coder* só permite criar um único "subsistema" (um único bloco) no *Simulink*. Mas isto traz um problema, pois o *software* considera que a comunicação entre a FPGA e o ARM é toda realizada num único instante. Após várias estratégias, foi possível ultrapassar esta limitação através da utilização de um sistema que incluía todos os subsistemas necessários, com as portas devidamente identificadas e através de mecanismos de sincronização manualmente desenvolvidos. Antes de ser gerado o ficheiro *Bitstream*, o modelo do *Vivado* deve ser ajustado para as entradas e saídas pretendidas. Concluído este passo é possível prosseguir com a configuração da FPGA da ZedBoard, conforme é identificado no apêndice B.

Seguidamente, no modelo de *Simulink* para configuração do ARM, é necessário interligar as entradas e saídas dos subsistemas aos locais corretos do algoritmo a implementar. Depois procede-se

¹Embora o processamento se realize apenas no ARM as entradas e saídas são sempre geridas pela FPGA

à compilação pelo *Embedded Coder* do modelo a implementar no ARM com o tempo de amostra (T_s) pretendido e uma duração explícita (infinito caso seja pretendido a execução contínua).

4.1.1 Processamento na FPGA

Com recurso aos *Add-Ons* referidos anteriormente, foi desenvolvido um método de processamento pela FPGA, considerado o processamento ideal, pois das três hipóteses apontadas é a que permite maior velocidades de processamento. Contudo, este método tem limitações na complexidade do código a utilizar, pois o algoritmo tem de ser convertido num circuito elétrico equivalente. Esta conversão utiliza os vários recursos da FPGA, nomeadamente *look up tables* (LUT), flip-flops (ou registos), processadores de sinais digitais (DSP), entre outros.

Embora o processamento da informação seja realizado pela FPGA, é necessário o utilizador escrever determinados parâmetros para o modelo e ler dados. Para isso recorreu-se ao modo externo que permite ao utilizador manipular variáveis do algoritmo executado na FPGA em tempo real, bem como consultar dados através do *Simulink*. Para esta manipulação de dados é necessário que estes sejam enviados para o processador, uma vez que este é o responsável pela gestão da comunicação através de *ethernet*, utilizada para comunicação com o *Simulink*, de forma a possibilitar o acesso do *Simulink* aos mesmos. Neste ponto, os dados essenciais a apresentar no modo externo são as medidas do sistema, de forma a ser possível calibrar os sensores. Esta calibração é efetuada através do envio do valor de *offset* pretendido para um registo lido pela FPGA. Isto implica que ocorram trocas de dados do ARM para a FPGA conforme é apresentado na figura 4.2. Neste caso, não é necessário assegurar a ordem e temporização das trocas de informação entre as duas unidades de processamento.

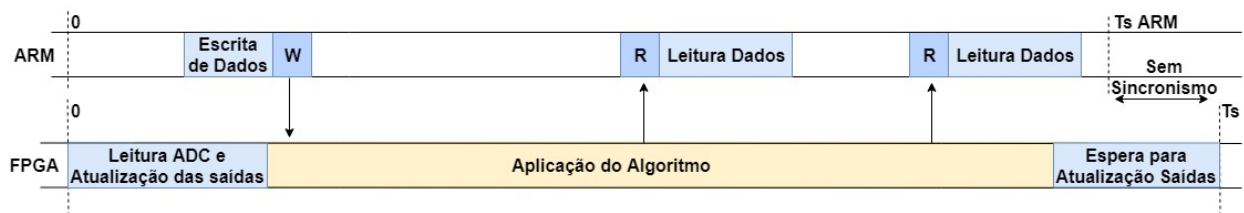


Figura 4.2 Esquema de processamento de dados apenas pela FPGA.

Conforme está indicado, no início de cada ciclo existe a leitura das medidas, que têm de ser convertidas pelos ADCs, o que demora poucos μs . Seguidamente é executado o algoritmo pela FPGA, resultado deste os pulsos a ser aplicados aos IGBTs do conversor. No entanto, estes pulsos não são aplicados imediatamente, sendo apenas atualizadas no final do ciclo. No fim do ciclo, as saídas são atualizadas em simultâneo com a aquisição de dados do ciclo seguinte, situação pretendida para o controlo preditivo a utilizar. As transferências de dados entre ARM e FPGA são simples uma vez que são leituras que podem ocorrer em qualquer instante do ciclo de processamento (T_s). Os dados são transferidos através dos registos da comunicação AXI4, que têm uma dimensão de 32 bits e uma velocidade de leitura em torno dos $5\mu s$.

4.1.1.1 Conversão do Código para a ZedBoard

No início da conversão do código surgiram alguns problemas, sendo que o principal estava relacionado com a necessidade de alterar a grande maioria dos blocos do código já desenvolvido (identificados na figura 4.3). Embora alguns blocos não fossem suportados pelo *HDL Coder*, estes foram substituídos por blocos alternativos da biblioteca do *Simulink* (*Simulink Library* ⇒ *HDL Coder*).

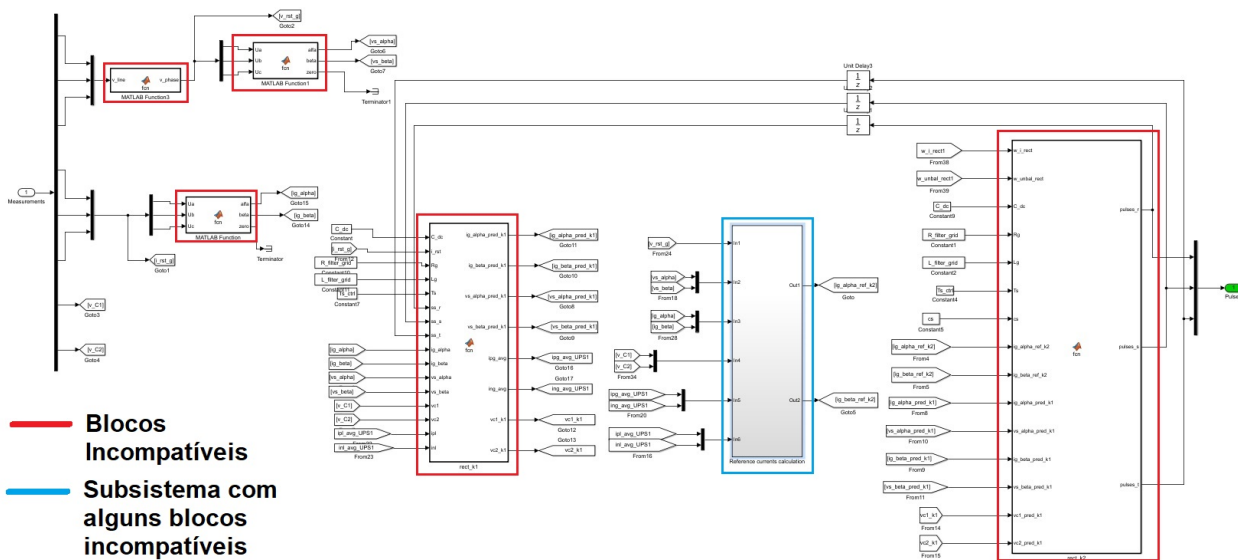


Figura 4.3 Modelo do *Simulink* com os blocos do código do retificador incompatíveis com o *HDL Coder*.

No caso dos blocos de funções do *Matlab* houve a necessidade de trabalhar com valores decimais, os quais não são diretamente suportados pelos mesmos quando utilizados pelo *HDL Coder* (não são suportadas variáveis de vírgula flutuante). A solução que possibilita a utilização das funções *MatLab* é a aplicação de variáveis de virgula fixa (*fixed-point*) correspondente, através da verificação da precisão e tamanho necessários a cada uma das variáveis da função. Esta solução permite a utilização direta do código contudo, é necessário verificar qual a precisão correta para cada variável em *fixed-point*, pois no caso do cálculo exceder o tamanho definido provocará valores incorretos devido a *overflow*. Por exemplo, a soma de 2 números com 4 bits para a parte inteira e 8 bits para a parte decimal origina um resultado com 5 bits na parte inteira e 8 na parte decimal. No caso da multiplicação dos valores referidos anteriormente o valor originará um resultado com 8 bits na parte inteira e 16 bits na parte decimal. Estes cálculos não podem ultrapassar 180 bits (parte inteira + parte decimal), que é o valor máximo suportado pelo *HDL Coder*, e o número de bits por variável deve ser limitado por forma a minimizar a utilização de recursos. Por isso mesmo, este processo é bastante complexo e deve ser efetuado com o máximo cuidado, uma vez, que uma incorreta definição da precisão das variáveis pode levar a falhas críticas no código. A alternativa a este processo consiste na transformação do código existente nas funções de *MatLab* num sistema equivalente composto pelos blocos da biblioteca própria para vírgula flutuante (*Simulink Library* ⇒

HDL Coder \Rightarrow *HDL Floating Point Operations*), utilizando sinais do tipo *single*. A vantagem do uso da vírgula flutuante em relação à vírgula fixa é o ajuste do número de bits para a parte inteira e para a parte decimal ao longo da propagação do sinal garantindo a máxima precisão. Contudo, devido às limitações dos blocos desta biblioteca, pode não ser possível converter o código para blocos do *Simulink* principalmente quando existem ciclos complexos no código. Assim, o método a utilizar depende do código a converter.

No caso do código do retificador ativo utilizado para os testes a este processamento foi necessário alterar vários blocos. Optou-se por alterar o código das funções para blocos do *Simulink*, tirando assim partido do uso da vírgula flutuante. Embora a conversão de alguns dos blocos fosse simples, como foi o caso das transformadas de Park, que consiste na aplicação de fórmulas matemáticas através de blocos o *Simulink*, houve um bloco mais trabalhoso, responsável por estimar os valores das variáveis no ciclo seguinte, representado na figura 4.4.

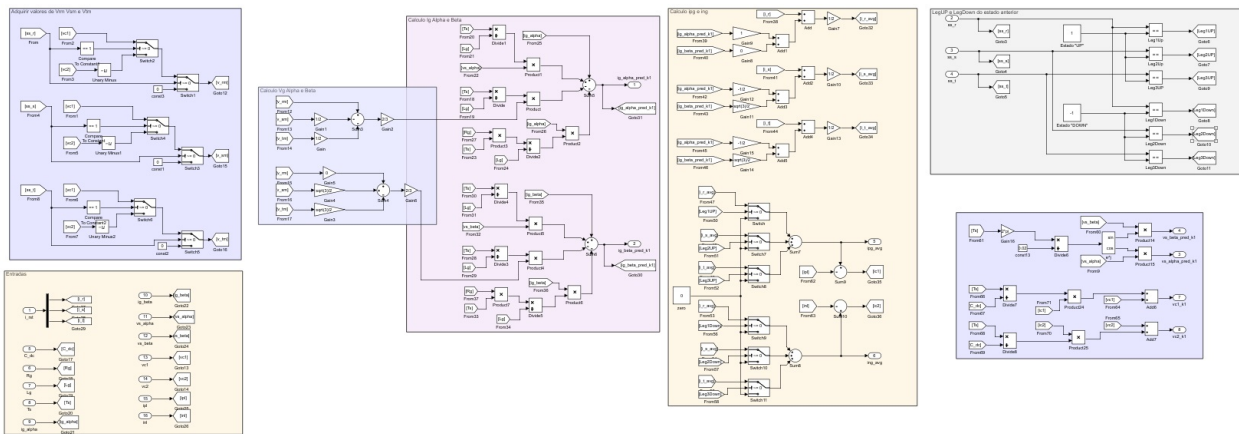


Figura 4.4 Algoritmo do bloco responsável por estimar os valores das variáveis no ciclo seguinte, após a sua conversão.

O único bloco em que foi utilizada a estratégia de vírgula fixa foi a verificação de qual o melhor dos 27 estados do conversor de 3 níveis, devido à dificuldade de implementação de ciclos com recurso à programação por blocos (com os blocos suportados pelo *HDL Coder*). Neste caso foi verificada a propagação das variáveis atribuindo a vírgula fixa ideal para cada uma delas, sendo que o resultado desta conversão está representado no apêndice D.

Na próxima versão do *Simulink* (2019b), que já deverá sair em outubro, já suportará a utilização de variáveis *single* nos ciclos quando utilizado o *HDL Coder*, pelo que a conversão do código deverá ser grandemente simplificada.

4.1.1.2 Aplicação do Algoritmo na ZedBoard

Para aplicação do algoritmo na ZedBoard é necessário começar pela definição das entradas e saídas a adicionar no código, bem como a posição das mesmas na placa. Foram configurados 3 interruptores, um para ligar o sistema, um para o retificador ativo e outro para ativar os *offset* (apenas deve ser ligado durante e após a configuração dos *offsets*). Foi também utilizado um botão de pressão

Programação da Plataforma de Controlo

para efetuar o *reset* das proteções, e 6 LEDs, sendo que, 1 representa o estado de funcionamento do sistema (ligar/desligar), outro o funcionamento do retificador e os quatro restantes cada um dos erros possíveis das proteções. Ainda foi necessária a utilização de 3 módulos Pmod AD1, de forma a possibilitar a existência de 6 ADC, e a configuração de 12 saídas correspondentes aos 12 impulsos para o retificador ativo.

Identificadas todas as entradas e saídas do sistema, estas foram adicionadas ao modelo de *Simulink*, convertido para utilização na ZedBoard. No último passo foi gerado o código *Bitstream* para configuração da FPGA a partir do *Simulink*, obtendo o esquema do *Vivado* presente na figura 4.5.

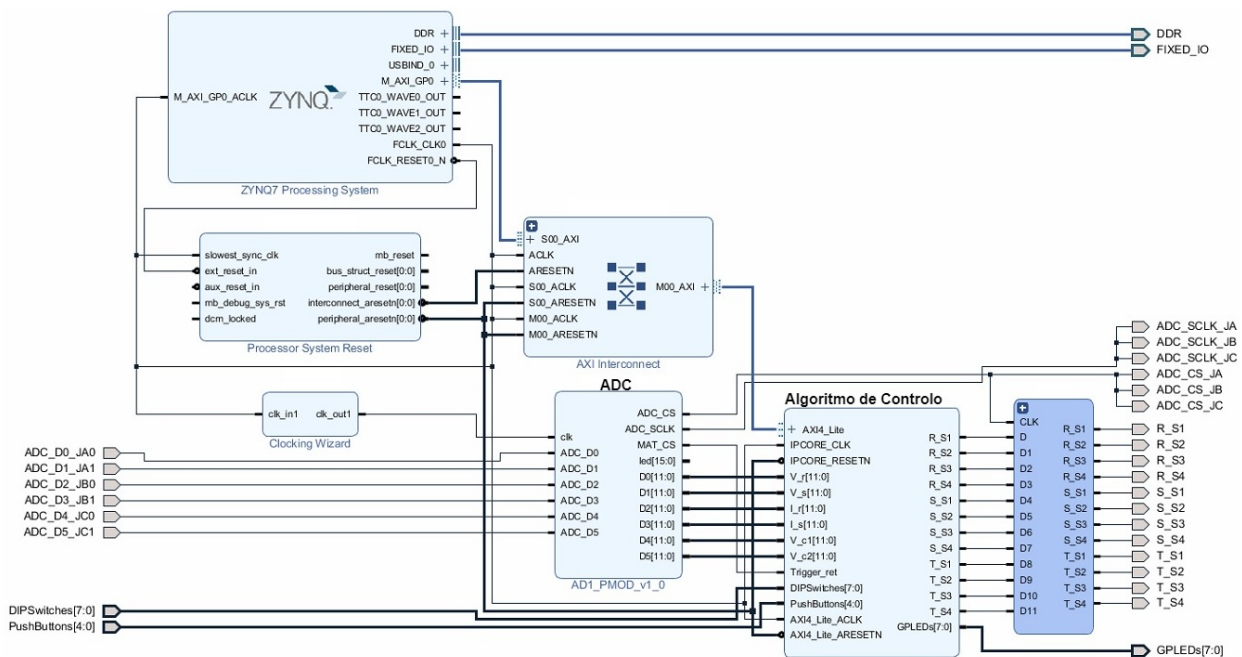


Figura 4.5 Modelo *Vivado* a implementar na ZedBoard.

Foi dada continuidade à configuração do sistema com a adição do mapeamento de pinos no *Vivado*. Esta atribuição de uma porta física às variáveis do sistema é realizada conforme apresentado no apêndice B, sendo que o id de cada porta da ZedBoard está representado na tabela B.2. Os ficheiros com a atribuição das variáveis às portas físicas estão presentes no apêndice D. A atribuição é idêntica à referida no início deste capítulo sendo que tudo o que não corresponde à interface com o utilizador, como é o caso dos interruptores, botão de pressão e LEDs são entradas e saídas atribuídas às portas PMOD da ZedBoard.

4.1.1.3 Resultados Obtidos

O processamento apenas pela FPGA seria a estratégia mais desejada devido à maior velocidade de processamento do algoritmo. Contudo, devido às suas limitações tornou-se numa opção impraticável. Utilizando apenas o código do retificador ativo para os testes, a Artix-7 presente na

ZedBoard não tinha recursos suficientes para as necessidades do algoritmo. A tabela 4.1, apresenta os recursos utilizados em diferentes estratégias de compilação do *Vivado*.

Tabela 4.1 Valores de utilização ZedBoard para diferentes tipos de compilação.

Tipo Compilação	LUT's	Slice Registers (FF)	DSP's	IO	MMCM	Duração
Sem Otimização	867362	15820	5559	36	1	7h15
Otimização Baixa	1503615	15750	166	36	1	27h42
Otimização Alta	1511632	15746	136	36	1	30h39
Valores Disponíveis	53200	106400	220	200	4	–

A escolha de estratégias focadas na utilização do menor número de recursos reduziu substancialmente o número de DSPs utilizadas, à custa de um aumento da utilização de LUTs. Uma das desvantagens deste tipo de implementação passa pela dificuldade de programação de algoritmos complexos na FPGA. Embora tenha sido apenas utilizado o código de um único conversor, os recursos da plataforma revelaram-se claramente insuficientes.

Na tabela 4.1, existe também uma coluna correspondente aos tempos de compilação, onde é notório um aumento do tempo em caso de otimização. Esta é outra das desvantagens deste tipo de implementação, pois a compilação de código para uma FPGA é extremamente demorada, pelo que qualquer alteração está sujeita a estes elevados tempos de compilação.

4.1.2 Processamento no ARM

Quanto ao processamento pelo ARM, ao ser um processador em vez de uma FPGA, deixa de existir o problema anterior de falta de espaço, uma vez que um processador pode executar um número ilimitado de instruções, ficando apenas limitado pelo tempo de processamento. No caso do processador, este consegue executar o algoritmo, contudo os tempos de processamento são tendencialmente elevados para algoritmos complexos. Apesar desta solução efetuar todo o processamento apenas no ARM, está sempre dependente da FPGA, pois as entradas e saídas do sistema são geridas por esta. A figura 4.6 representa o fluxo de informação que tem o início e o fim na FPGA devido à utilização de entradas e saídas (IO).

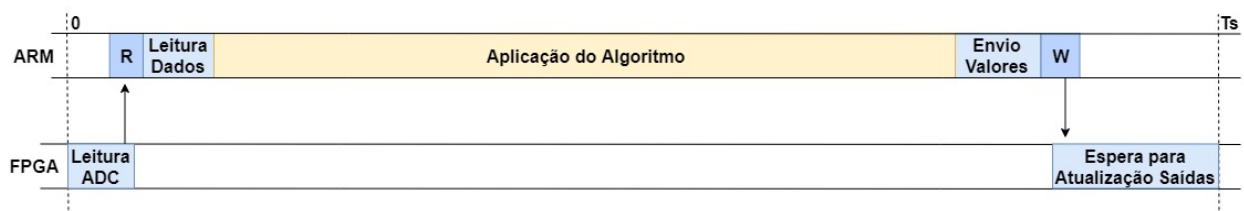


Figura 4.6 Esquema de processamento de dados apenas pelo ARM.

Neste tipo de implementação o ciclo tem início com a leitura dos ADCs na FPGA e envio dos valores para o ARM assim que a conversão seja concluída. Após receção dos valores no processador,

este executa o algoritmo de controlo e envia os resultados para a FPGA. Recebidos os valores, a FPGA aguarda que o ciclo termine para atualizar as saídas em simultâneo com a aquisição dos valores para o ciclo de cálculo seguinte.

Apesar deste método possibilitar uma fácil aplicação do algoritmo, o facto de os dados terem de fluir entre processador e FPGA implica que os mesmos estejam sincronizados (o início do ciclo de processamento do ARM deve ser despoletado pela FPGA). A existência de sincronismo entre processador e FPGA significa que ambas as unidades de processamento começam e acabam o ciclo de processamento no mesmo instante. Por outras palavras, o sincronismo garante que os dados são processados no ARM dentro do ciclo de cálculo estipulado pela FPGA.

Este método é vantajoso pela facilidade de implementação de algoritmos, pois não requer qualquer tipo de conversão (praticamente todos os blocos e tipo de dados são suportados pelo ARM). A sua grande dificuldade, sendo o maior problema existente no trabalho, consiste na sincronização das duas unidades de processamento. Quanto à sincronização entre as unidades de processamento, o *Matlab/Simulink* já tem incorporada uma solução para que o processador espere pela FPGA, no entanto não havia uma solução no *Simulink* para a FPGA aguardar pelo ARM. Devido a esta falta de modos de sincronismo, por parte do *Simulink*, foi necessário o desenvolvimento de um forma de sincronizar as unidades de processamento a partir dos blocos do *Simulink*. A solução ideal seria a utilização de uma interrupção gerada pela FPGA, que iniciaria cada nova tarefa no processador (ciclo de processamento). No entanto a inexistência de interrupções nas bibliotecas utilizadas tornou complexa a resolução deste problema. Assim, com base nos blocos do *Simulink*, estipulou-se a sincronização por duas estratégias distintas. A primeira foi o uso de um mecanismo de espera ativo por sinais de *enable*, que bloqueavam o código até o sinal de *enable* ser accionado. A segunda foi uma verificação periódica, que efetua a verificação para iniciar a execução do algoritmo uma vez por ciclo.

4.1.2.1 Sincronismo Através de Esperas

Este método consiste numa espera ativa até que o processador receba um sinal vindo da FPGA que permita a continuação do seu cálculo. O fluxograma da figura 4.7 ilustra como foi implementada a espera ativa. A verificação "Valor enviado para ARM" é realizada através do envio de uma variável pela FPGA no fim do ciclo de conversão de dados do ADC. Este sinal foi implementado no módulo dos Pmod AD1 no *Vivado* e nomeado de "Mat_CS". A alteração está identificada no ponto C.3, onde está presente o código VHDL do módulo com diversos ADC e o sinal de final de conversão.

Neste método ocorre um bloqueio do processador pois este fica num ciclo infinito até que seja verificado que a FPGA enviou o valor. Este bloqueio do processador faz com que não seja possível realizar outras tarefas necessárias ao sistema operativo da ZedBoard (Linux), impedindo a correta utilização do modo externo e causando perdas ocasionais de ciclos e/ou atrasos no início do ciclo. Assim sendo, este método também se torna limitado, pois ao sair do modo externo o código implementado no ARM deixa de executar. Por outro lado, sem o modo externo (execução

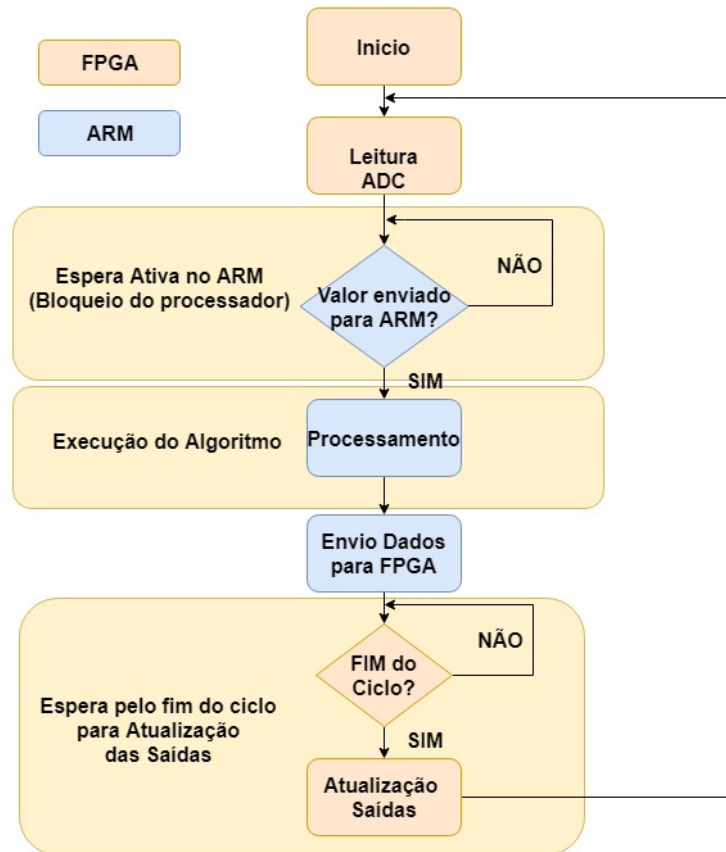


Figura 4.7 Fluxograma para sincronização de dados com recurso à espera ativa.

Standalone) não é possível proceder à verificação dos valores dos sensores nem ajustar os *offsets* dos mesmos.

Após a configuração da ZedBoard, verificou-se que o sincronismo não era garantido, pois o processador tinha um sinal de *clock* diferente da FPGA, com período variável. Embora o período do processador fosse muito próximo do definido ($100\ \mu\text{s}$), o mesmo tinha valores inferiores ao sinal da FPGA. O período menor por parte do ARM faz com que o seu início de ciclo "avance" progressivamente em relação à FPGA, conforme é apresentado na figura 4.8. Estes avanços comprometem o sincronismo entre estas duas unidades de processamento.

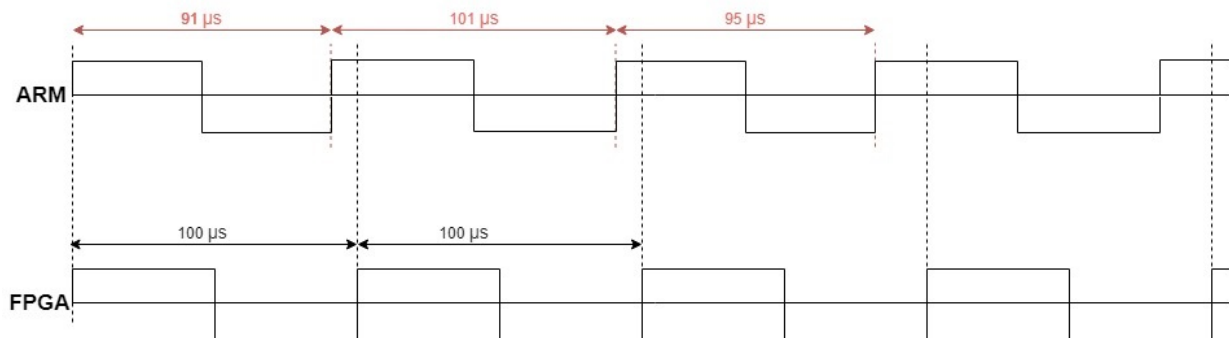


Figura 4.8 Variação do sinal de *clock* do ARM e da FPGA.

Devido a este problema de sincronismo, a execução do código não é confiável pelo que não pode ser utilizada para aplicação do algoritmo de controlo a aplicar. Devido ao problema estar no *clock* variável do ARM, procurou-se uma forma de resolver este problema através da utilização de um *clock* gerado pela FPGA. Todavia, após várias tentativas sem sucesso, procedeu-se com a sincronização das unidades de processamento através de esperas ativas deixando este método de parte.

4.1.2.2 Sincronismo Através de uma Verificação Periódica

Quanto ao segundo método supracitado, a verificação periódica, consiste numa verificação a uma frequência elevada se o sinal já foi enviado pela FPGA, sendo que caso seja verificado o ciclo prossegue, caso contrário acaba. Este método não bloqueia o processador numa verificação infinita, contudo tem como desvantagem o baixo tempo de ciclo, que tende a causar problemas de fiabilidade no Linux (com ciclos ignorados).

Em suma, o problema da verificação periódica é a incapacidade do sistema operativo lidar com ciclos tão curtos, tornando assim o modo externo de novo não confiável. Após contacto com a equipa da *Mathworks*, detetou-se que o linux existente na ZedBoard não era a versão real-time, pelo que os problemas causados seriam uma consequência do uso da versão regular em vez da versão para *Real-Time* (Linux RT).

4.1.3 Processamento Híbrido

No processamento híbrido o ideal é correr o ciclo no processador, mas fazer os cálculos mais intensivos na FPGA (no controlo preditivo, o ciclo das 27 hipóteses). Desta forma, é minimizado o tempo de execução em relação ao processamento no ARM, através do processamento da parte pesada do código na FPGA. Durante o ciclo de processamento ocorrem transferências de dados entre o processador e a FPGA. As transferências de dados podem implicar esperas do ARM pela FPGA ou vice-versa, mas também podem realizar cálculos em simultâneo. Na figura 4.9 é apresentado um esquema como exemplo deste processamento, onde se pode verificar que os tempos de execução entre as unidades de processamento devem estar sincronizados.

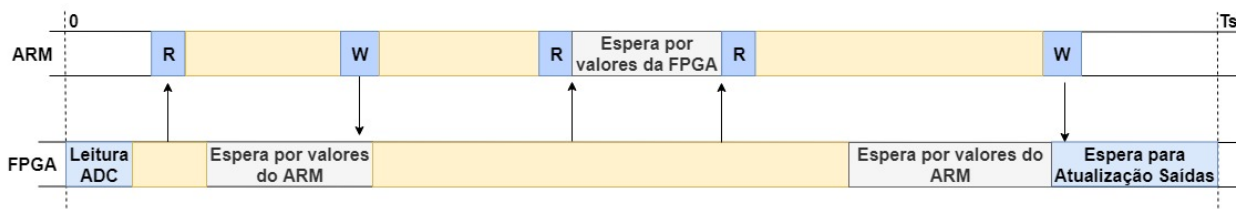


Figura 4.9 Esquema de processamento de dados híbrido.

A figura anterior apresenta um exemplo de um processamento de dados híbrido, que inicia o ciclo na FPGA com a leitura de dados dos ADCs. Seguidamente estes dados sofrem execução de parte do algoritmo e posteriormente são enviados para o ARM. Enviados os dados, a FPGA aguarda

que o ARM execute outra parte do algoritmo e retorne os resultados à FPGA. O ARM após enviar os dados necessários para a FPGA prossegue com a execução da sua parte do algoritmo até precisar dos dados devolvidos pela FPGA, esperando pelos mesmos. Estas tocas repetem-se, até que o ARM envie o resultado final para a FPGA, que aguarda até ao fim do ciclo para atualizar as saídas, e efetuar a leitura dos ADCs para o ciclo seguinte em simultâneo.

Com este fluxo de dados é importante determinar que partes devem correr na FPGA e que partes devem correr no ARM. É de salientar que no caso do processamento pela FPGA é necessário que o código seja convertido em blocos de *Simulink* compatíveis conforme apresentado no ponto 4.1.1.1. Quanto ao algoritmo de controlo processado pelo ARM, não são necessárias conversões pois os blocos do *Simulink* utilizados são todos suportados.

Assim como no método anterior a grande dificuldade deste método é o sincronismo entre as unidades de processamento, sendo que neste caso ainda acresce complexidade devido às trocas durante o ciclo de processamento. Neste caso utilizou-se apenas o método de espera ativa, visto ser das duas estratégias a que apresentou menos problemas no funcionamento com o modo externo.

4.1.3.1 Sincronismo com Múltiplas Esperas

O sistema de teste consiste numa comparação entre um cálculo realizado pelo ARM e FPGA em conjunto com os resultados obtidos por cada um deles individualmente. Este sistema é apresentado na figura 4.10, e o cálculo é efetuado através da equação $y = Valor_{ADC} \times 3 \div 5 \times 2$, onde $Valor_{ADC}$ é o valor de uma leitura de um ADC.

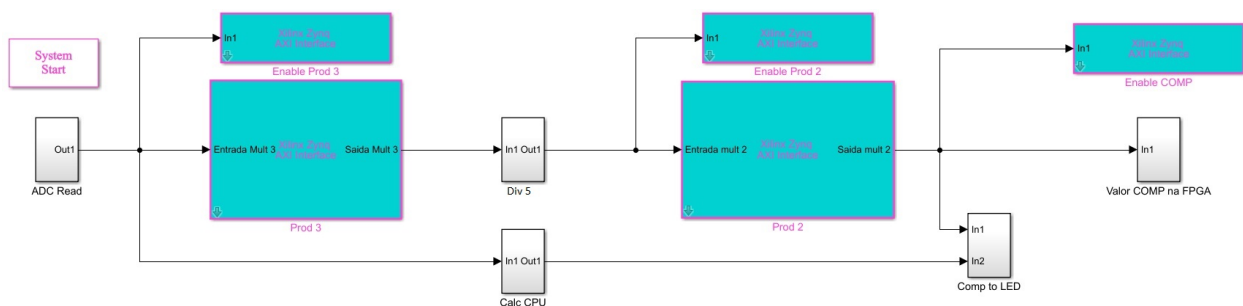


Figura 4.10 Modelo *Simulink*, para verificação do sincronismo.

O algoritmo representado na figura 4.10, efetua a leitura do valor de um ADC (bloco "ADC Read"). Este valor é enviado para FPGA, para esta efetuar a multiplicação por 3, também é enviado o sinal de enable de forma à FPGA executar o cálculo só depois de receber os valores enviados pelo ARM. Simultaneamente é aplicada a equação anterior ao valor do ADC no ARM e este valor aguarda pela finalização dos cálculos na FPGA para comparação. Após a FPGA concluir a multiplicação por 3, envia o valor para o ARM, que o divide por 5. Concluído o cálculo o ARM envia o resultado para a FPGA multiplicar por 2, assim como o sinal de enable. Por fim a FPGA envia ao ARM o valor do cálculo, que é comparado com o valor do processador e enviado o resultado para um led e para a FPGA. Na FPGA durante o envio de dados do ADC para o ARM, esta efetua o cálculo da equação, e aguarda pelo retorno do valor calculado pelo ARM e FPGA, comparando os

valores. Também foi colocado um sinal que varia o seu valor entre 0 e 1 a cada ciclo, de forma a verificar as frequências de funcionamento quer do ARM quer da FPGA.

Uma vez mais devido ao bloqueio do processador o modo externo não é confiável. Ainda assim este método apresenta problemas ainda maiores, sendo que a FPGA não consegue esperar pelos valores do ARM para efetuar a comparação. Desta feita o uso do *HDL Coder* e do *Embedded Coder* permite a implementação do algoritmo na ZedBoard, contudo as continuas falhas no modo externo e a execução não confiável, fez com que esta implementação não fosse utilizada.

4.2 Alternativa de Processamento

Tendo em conta as necessidades para a plataforma desenvolvida executar o algoritmo de controlo, na prática não foi possível configurar a ZedBoard da forma esperada com as extensões do *Simulink* descritas. A utilização apenas da FPGA encontra-se totalmente funcional, contudo, o facto de apenas permitir a execução de algoritmos simples faz com que não consiga executar o código pretendido. Devido às limitações do *HDL Coder* e do *Embedded Coder*, não existe um método de processamento que garanta inteiramente a sincronização das unidades de processamento. Recentemente, foi desenvolvida pela *Mathworks* uma nova ferramenta, o *SoC Blockset*, apresentado apenas na versão 2019a, que apresenta uma possível solução para o problema existente.

O *SoC Blockset*, ao contrário das ferramentas utilizadas anteriormente, agrupa o processamento efetuado no ARM e na FPGA num único modelo de *Simulink*. A novidade apresentada por este método é a inclusão de interrupções, o principal motivo de falha das ferramentas anteriores. Assim, adotou-se esta nova ferramenta para configurar a ZedBoard de forma a que esta pudesse executar o sistema de controlo de um conversor de eletrónica de potência.

4.2.1 SoC Blockset

O *SoC Blockset* tem uma metodologia diferente dos *Add-Ons* utilizados inicialmente. Este novo método de programação de sistemas embebidos, que inclui a ZedBoard, apresenta novas funcionalidades ao *Simulink*, como é o caso das interrupções. O acesso aos registos e à memória também foi melhorado, sendo que as desvantagens encontradas são uma maior dificuldade na implementação de código VHDL, bem como a adição e manipulação das portas de IO no *Vivado*.

Contrariamente ao que acontece na configuração pelo *Embedded Coder* e *HDL Coder* que utiliza dois modelos do *Simulink*, o *SoC Blockset* utiliza um projeto. A figura 4.11 é um exemplo de um projeto que utiliza esta ferramenta na sua camada superior. Esta camada indica o fluxo de dados entre processador e FPGA, pelo que representa de que modo estes são enviados, ou seja, se o envio é realizado pela memória RAM ou por registos.

Os blocos "FPGA" e "Processador" são apenas referências ao modelo a ser executado pela FPGA e processador. No caso de ser necessário enviar um valor para uma saída, o ARM tem de enviar o sinal para um registo (ou memória) e seguidamente a FPGA tem de ler esse registo.

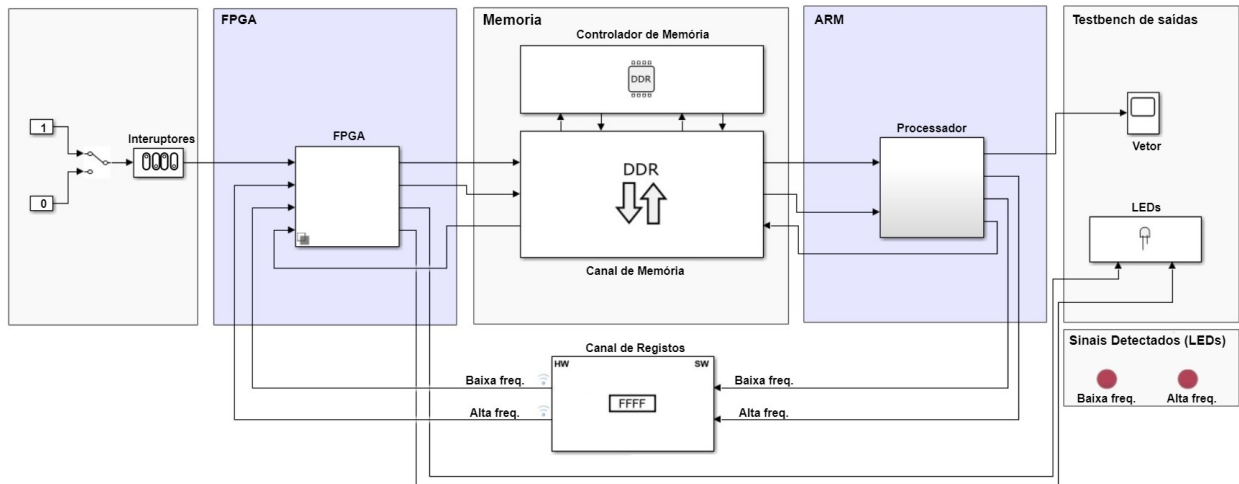


Figura 4.11 Exemplo da camada superior de um projeto do *SoC Blockset*.

Posteriormente a FPGA deve enviar o valor em questão para a saída correspondente (por exemplo um LED). No caso das leituras de valores de uma entrada, o processo é idêntico, a FPGA tem de ler o valor e colocá-lo num registo para possibilitar a leitura por parte do ARM. As regras de troca de dados mantêm-se iguais, embora o uso do *SoC Blockset* facilite o desenvolvimento dos projetos. Esta facilidade deve-se ao facto do envio de informação pelos registos do AXI4 serem automaticamente atribuídos e à existência de uma camada superior que facilita a visualização do fluxo de dados entre as unidades de processamento.

O grande problema desta nova ferramenta é a incompatibilidade com as utilizadas anteriormente. Assim sendo, todo o trabalho realizado nas ferramentas anteriores não poderia ser aproveitado, fazendo com que fosse necessário recomeçar as verificações das necessidades do sistema. Deste modo, foi necessário desenvolver novamente os ADCs, confirmar o seu funcionamento uma vez mais e verificar o funcionamento das interrupções no *SoC Blockset*.

4.2.1.1 Desenvolvimento dos ADCs

No caso desta nova biblioteca, a implementação de código VHDL não foi possível, pelo que foi necessário o desenvolvimento dos ADCs no *Simulink*. O modelo do *Simulink* para adição de ADCs ao *SoC Blockset* está representado na figura 4.12. Este modelo está desenvolvido para a adição de 6 ADCs. A geração dos sinais de SCLK e CS são realizados por um contador, as variáveis estão presentes no projeto do *SoC* e são denominadas por "ADCComTime" e "ADCSample". Também foi incluído um sinal de fim de conversão denominado por "EoC" (*End of Conversion*).

Realizada a conversão de dados, estes podem ser convertidos para valores do tipo "single" de forma a permitir a aplicação dos ganhos e a existência de casas decimais. Posteriormente, pode ser utilizado o bloco que escreve os bits de um "single" no registo de 32 bits da ZedBoard. Na leitura dos valores (no ARM) é necessário utilizar o mesmo bloco de forma a converter os 32bits numa variável do tipo "single". O código do ADC, que recebe os valores dos 12 bits, 1 a 1 à frequência de envio (SCLK) foi implementado com a ferramenta Stateflow do *Simulink*. Esta ferramenta

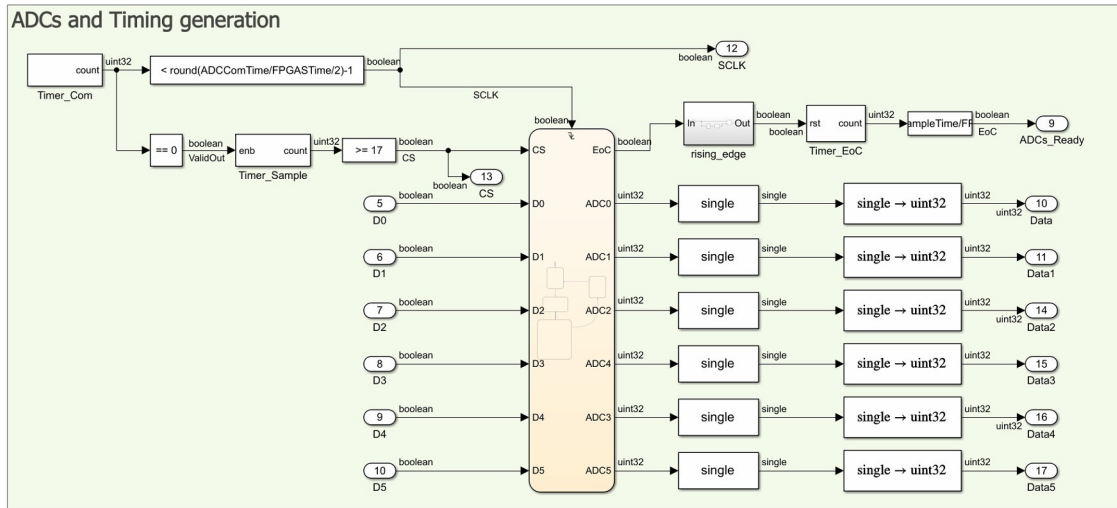


Figura 4.12 Esquema para implementação do ADC no *SoC Blockset*.

permite que seja desenvolvido um algoritmo num estilo de fluxograma e o desenvolvimento de código de forma intuitiva. O algoritmo é apresentado na figura 4.13, e a cada oscilação do SCLK é realizada uma interação. As condições para o passo seguinte são apresentadas entre parênteses retos (["condição"]) enquanto que as ações são apresentadas dentro de chavetas ({"ação"}). O sistema tem em atenção o funcionamento do ADC descrito no ponto 3.1.1. Em suma, após serem adquiridos novos dados (valor de CS vai a 0) os primeiros 4 bits são rejeitados e os 12 bits restantes são acumulados. No final da conversão, os 12 bits acumulados são disponibilizados na saída representando o valor resultante da leitura.

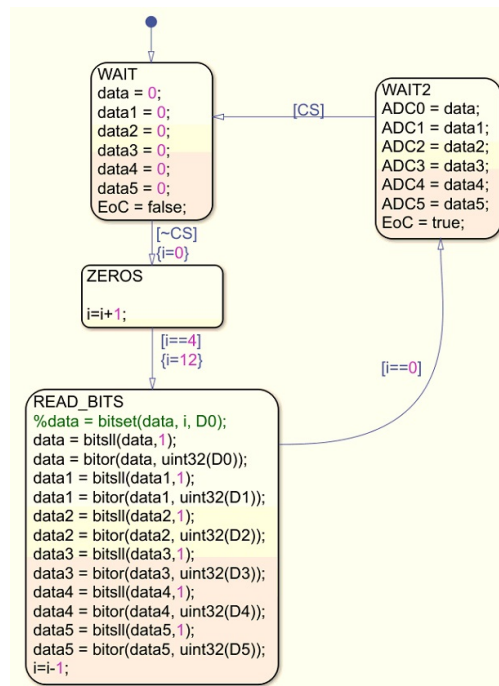


Figura 4.13 Algoritmo do ADC desenvolvido no *Add-On Stateflow do Simulink*.

Desenvolvidos os ADCs, verificou-se o seu funcionamento pela estratégia utilizada no método anterior. Este método consiste na aplicação do mesmo sinal nos vários ADCs, pelo que os valores adquiridos devem estar sobrepostos (caso as aquisições ocorram em simultâneo). Verificou-se que, quer a aquisição de dados quer os tempos de amostragem funcionavam conforme esperado. Superado o desafio de desenvolver os ADCs pelo *Simulink* passou-se à verificação do funcionamento das interrupções no *SoC Blockset*.

4.2.1.2 Interrupções

As interrupções permitem ao processador executar um código específico aquando da recepção de um sinal, ou seja, o código é assíncrono, sendo executado à frequência do sinal de interrupção podendo esta frequência ser constante ou variável. Uma vez que o *SoC Blockset* não suporta eventos relacionados com a escrita em registos, foi utilizada a escrita em memória RAM para gerar a interrupção desejada. A interrupção é gerada pelo canal de memória, apenas no modo de acesso direto à memória (DMA - *direct memory access*). Também tem de ser utilizado o bloco de controlo de memória de modo a permitir guardar os dados na memória RAM. Os dois blocos referidos estão representados na figura 4.14, onde o canal de memória corresponde ao bloco inferior e o controlo de memória ao bloco superior.

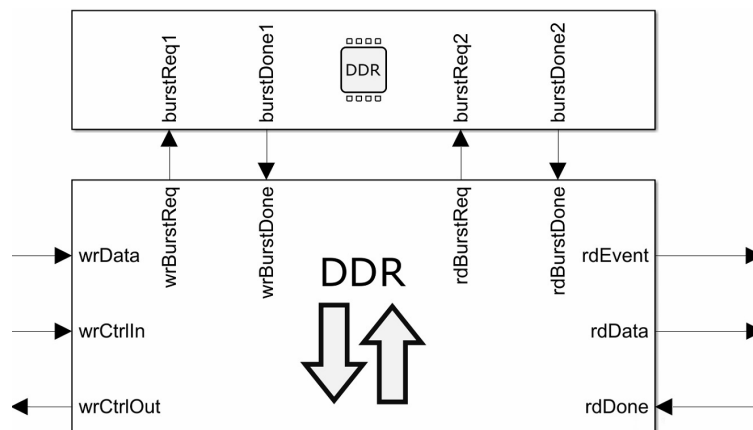


Figura 4.14 Blocos do controlo e canal de memória.

Assumindo uma troca de dados da FPGA para o ARM, a FPGA envia o valor pretendido para o "wrData" sendo que, no final do envio de todos os dados envia um "1" lógico para o "wrCtrlIn" de forma a validar a informação colocada na memória. Após esta sinalização, é gerado um evento ("rdEvent") pelo canal de memória, o qual desencadeia a interrupção (execução de um subsistema desencadeado por este sinal). O código a ser executado durante a interrupção efetua a leitura dos dados ("rdData"), sendo que no final da leitura é necessário enviar um "1" lógico para o "rdDone" com o intuito de confirmar a leitura de dados. Esta confirmação da leitura limpa o *buffer* do canal de memória permitindo a escrita de novos valores. Esta confirmação de leitura, não pode ser esquecida, pois devido à memória ser uma estrutura do tipo primeiro a entrar primeiro a sair (FIFO), caso não seja confirmada a leitura, não poderão ser escritos novos dados (não sendo geradas novas

Programação da Plataforma de Controlo

interrupções). O sinal "wrCtrlOut" indica se é possível escrever novos valores no canal de memória, ou seja, é um indicativo se o FIFO está disponível para escrita de novos valores.

Uma vez que os valores são enviados através de endereços de memória em vez de registos os dados são trocados a uma menor velocidade. Para verificar os tempos entre o sinal de interrupção e a leitura de dados foi desenvolvido um pequeno modelo *Simulink* que consiste num simples envio de um valor da FPGA para o ARM. Com o auxílio do osciloscópio foram medidos os sinais representados na figura 4.15 que permitiram a análise temporal pretendida. O sinal amarelo é um sinal que a cada ciclo da FPGA alterna o seu valor. O sinal azul é um sinal que no início de cada ciclo da FPGA é colocado a "1", e quando é iniciada a interrupção no ARM é colocado a "0". O sinal rosa é uma indicação do funcionamento do processador e o sinal verde é idêntico ao sinal 1, exceto no facto de alterar o seu valor a cada nova interrupção (no ARM).

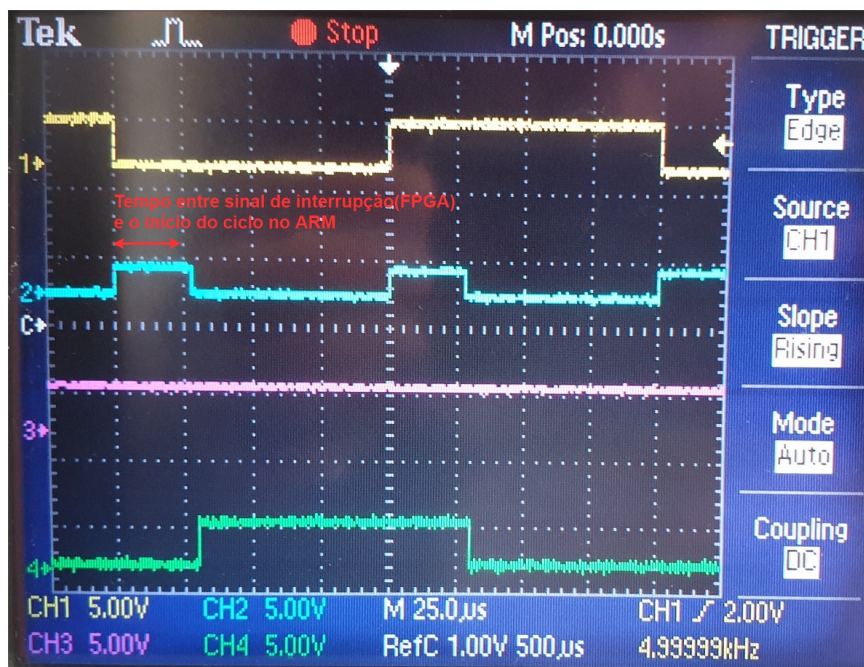


Figura 4.15 Resultados de envio de dados pela memória RAM.

Ao serem analisados os tempos de início de ciclo da interrupção e de envio dos dados verificou-se que estes eram superiores ao desejado. No caso da interrupção, esta demora $25 \mu\text{s}$ a ser desencadeada pelo canal de memória que, para os tempos de ciclo desejados, nomeadamente, $100 \mu\text{s}$ corresponde a $\frac{1}{4}$ de ciclo. Após a abertura da interrupção são necessários mais $25 \mu\text{s}$ para efetuar a leitura do valor guardado na memória. Com estes valores de envio de dados, este método torna-se pouco viável para os tempos de amostra desejados.

De modo a minimizar o tempo de código perdido em transferências de dados, passou-se a utilizar registos para o envio. Pela memória, apenas seria passado um valor sem influência para o algoritmo, cuja sua função seria simplesmente criar a interrupção no ARM. Com esta estratégia foi possível reduzir o tempo em que o ARM não executa código, resultando em apenas $25 \mu\text{s}$ de atraso no início do ciclo, correspondente ao tempo de abertura da interrupção.

Concluiu-se que o fraco desempenho das interrupções se deve ao facto do sistema operativo disponibilizado pela *Mathworks* para a *ZedBoard* não ser um sistema operativo *Real-Time* (distribuição de Linux *Non-Real-Time*). Assim, será necessário proceder a uma recompilação do Kernel Linux utilizado com uma nova versão *Real-Time* (Linux RT), por forma a melhorar o desempenho do sistema.

Apesar de os principais obstáculos à utilização da *ZedBoard* para prototipagem a partir do *Simulink* terem sido ultrapassados, não foi possível proceder ao teste do sistema de controlo completo antes do término desta dissertação. Este trabalho continuará, no entanto, a ser desenvolvido no âmbito do projeto em que esta dissertação se insere.

Capítulo 5

Conclusões e Sugestões para Trabalhos Futuros

5.1 Conclusões

Nesta dissertação foi desenvolvida uma plataforma para controlo de conversores de eletrónica de potência. O uso da plataforma destina-se ao projeto "DRIFT" (*Datacenter Resilience Increase through Fault Tolerance in UPS systems*) que necessita de uma plataforma de controlo programável pelo *Simulink* com um baixo custo e compacta, para ser embutida no protótipo. Foi estudada uma solução que utiliza a *ZedBoard* para execução do algoritmo de controlo já desenvolvido.

Inicialmente, foram utilizadas as ferramentas *HDL Coder* e *Embedded Coder*, que permitem interligar o *Simulink* ao *Vivado* de forma a permitir a programação da placa. Durante o uso destas ferramentas surgiram vários problemas, sendo os primeiros relacionados com a compatibilidade dos programas a utilizar (*Simulink* e *Vivado*). Ultrapassadas estas dificuldades foi possível adicionar ADCs à *ZedBoard* de forma a ser possível à plataforma efetuar a medição de valores analógicos. Conseguiu-se ainda garantir que a aquisição de dados e atualização de saídas ocorresse em simultâneo. Este ponto era fulcral para garantir que o sistema de controlo preditivo a utilizar funciona corretamente.

Na implementação do código na FPGA também foram encontrados alguns problemas de compatibilidade, entre as ferramentas e determinados blocos, sendo estes problemas superados com ajuste de blocos e conversões de código. Quanto ao processamento pela FPGA não foi possível utilizar pois esta apresentou recursos claramente insuficientes, para execução de todo algoritmo. Ao utilizar o ARM, para processamento da informação, foi necessário sincronizar as unidades de processamento (FPGA e ARM). Estes problemas de sincronismo foram parcialmente superados, pelo que a implementação com processamento apenas no ARM ou híbrido não é confiável, devido a existir ciclos de cálculo que não são efetuados.

Embora o método anterior não permitisse a plataforma funcionar corretamente, o aparecimento da ferramenta *SoC Blockset* na versão 2019a do *Simulink*, que já inclui interrupções, permitiu a hipótese do projeto a desenvolver funcionar conforme pretendido. O problema que surgiu ao

trocar de ferramenta para programação da ZedBoard, foi a perda de todo o trabalho realizado anteriormente, pois o *SoC Blockset* não é compatível com as ferramentas utilizadas no início da dissertação. Posto isto, recomeçou-se com o desenvolvimento dos ADCs, nesta nova ferramenta, que foi concluído com sucesso. Também se verificou que os mesmos adquiriam dados em simultâneo, assim como atualizavam as saídas no instante da aquisição. Também foi verificado o funcionamento das interrupções, funcionalidade nova apresentada nesta ferramenta que permitiu o sincronismo entre ARM e FPGA. Esta ferramenta garantiu a resolução dos problemas de sincronização, contudo ainda é necessário efetuar otimização para permitir uma execução no tempo de execução desejado.

Em suma, pode-se dizer que o ponto de desenvolvimento da utilização da ZedBoard como plataforma de controlo, é o cumular de um trabalho longo e extensivo. Para a plataforma ficar completamente funcional ainda são necessários alguns ajustes, contudo com a utilização do *SoC Blockset*, todas as verificações já foram realizadas. O trabalho ainda a ser realizado consiste em melhorias do sistema e no agrupamento de todas as soluções desenvolvidas para implementação do sistema de controlo na ZedBoard.

5.2 Sugestões para Trabalhos Futuros

Devido à dissertação estar inserida no projeto "DRIFT", o trabalho realizado será continuado, de forma a explorar os pontos que faltam para a sua conclusão.

O primeiro passo será a atualização do Kernel Linux existente na placa, específico para aplicações em *real-time*. Assim poderá ser minimizado os tempos para o ARM iniciar a interrupção, aumentando o tempo de cálculo disponível para o processador (sem alteração do tempo de cada ciclo de cálculo).

O segundo ponto será a divisão do algoritmo a executar, entre ARM e FPGA, atribuindo parte do código a cada unidade de processamento. De uma forma simples o ARM deverá executar a maior parte do código, sendo atribuído à FPGA a conversão dos valores dos ADCs para os valores reais, a gestão das entradas e saídas e o cálculo de qual dos 27 estados de um conversor deve ser aplicado. Este cálculo dos 27 estados é extremamente pesado para o ARM devido a incluir o cálculo de um modelo matemático complexo de forma cíclica (igual para todos os estados), podendo no entanto ser calculado de forma extremamente rápida na FPGA. Desta forma, o algoritmo é dividido de forma a utilizar as vantagens de cada uma das unidades de processamento, para execução do código no menor tempo possível.

Por fim, deve ser realizada a implementação do algoritmo e verificação do seu funcionamento. Ainda podem ser realizados testes comparando a plataforma com soluções existentes, como é o caso da *MicroLabBox* da *dSpace* de forma a comparar os desempenhos.

Referências

- [1] S. N. Rao, D. V. A. Kumar, and C. S. Babu, “New multilevel inverter topology with reduced number of switches using advanced modulation strategies,” in *2013 International Conference on Power, Energy and Control (ICPEC)*, pp. 693–699, Feb 2013.
- [2] J. Rodriguez, Jih-Sheng Lai, and Fang Zheng Peng, “Multilevel inverters: a survey of topologies, controls, and applications,” *IEEE Transactions on Industrial Electronics*, vol. 49, pp. 724–738, Aug 2002.
- [3] M. Malinowski, M. P. Kazmierkowski, and A. M. Trzynadlowski, “A comparative study of control techniques for pwm rectifiers in ac adjustable speed drives,” *IEEE Transactions on Power Electronics*, vol. 18, pp. 1390–1396, Nov 2003.
- [4] M. P. Kazmierkowski and L. Malesani, “Current control techniques for three-phase voltage-source pwm converters: a survey,” *IEEE Transactions on Industrial Electronics*, vol. 45, pp. 691–703, Oct 1998.
- [5] Wei Xie, Xiaocan Wang, G. Dajaku, D. Gerling, and R. Kennel, “Improvement and comparison of efficiency and low cost drive system based on dtc and dtc-svm,” in *2013 International Electric Machines Drives Conference*, pp. 1261–1266, May 2013.
- [6] Wen Duan, Feng Gao, Yuwei Xie, and Mengxing Chen, “Model predictive control of two-stage inverter with less switching losses and dc-link capacitor current,” in *2016 IEEE 8th International Power Electronics and Motion Control Conference (IPEMC-ECCE Asia)*, pp. 2085–2091, May 2016.
- [7] J. Ko, J. Choi, and D. Chung, “Hybrid artificial intelligent control for speed control of induction motor,” in *2006 SICE-ICASE International Joint Conference*, pp. 678–683, Oct 2006.
- [8] E. K. Sato, M. Kinoshita, Y. Yamamoto, and T. Amboh, “Redundant high-density high-efficiency double-conversion uninterruptible power system,” *IEEE Transactions on Industry Applications*, vol. 46, pp. 1525–1533, July 2010.
- [9] T. Lee, M. Kinoshita, and K. Sanada, “High-efficiency large-capacity uninterruptible power supply for 3-phase 4-wire power system,” in *Proceedings of The 7th International Power Electronics and Motion Control Conference*, vol. 2, pp. 1131–1136, June 2012.
- [10] T. Lee, M. Kinoshita, and K. Sanada, “High-efficiency large-capacity uninterruptible power supply using bidirectional-switch-based npc multilevel converter,” in *8th International Conference on Power Electronics - ECCE Asia*, pp. 2100–2105, May 2011.
- [11] F. Cammarota and S. Sinigallia, “High-efficiency on-line double-conversion ups,” in *INTELEC 07 - 29th International Telecommunications Energy Conference*, pp. 657–662, Sep. 2007.
- [12] J. Rodriguez, M. P. Kazmierkowski, J. R. Espinoza, P. Zanchetta, H. Abu-Rub, H. A. Young, and C. A. Rojas, “State of the art of finite control set model predictive control in power electronics,” *IEEE Transactions on Industrial Informatics*, vol. 9, pp. 1003–1016, May 2013.

Referências

- [13] P. Cortes, M. P. Kazmierkowski, R. M. Kennel, D. E. Quevedo, and J. Rodriguez, “Predictive control in power electronics and drives,” *IEEE Transactions on Industrial Electronics*, vol. 55, pp. 4312–4324, Dec 2008.
- [14] S. Vazquez, J. Rodriguez, M. Rivera, L. G. Franquelo, and M. Norambuena, “Model predictive control for power converters and drives: Advances and trends,” *IEEE Transactions on Industrial Electronics*, vol. 64, pp. 935–947, Feb 2017.
- [15] M. Norambuena, C. Garcia, and J. Rodriguez, “The challenges of predictive control to reach acceptance in the power electronics industry,” in *2016 7th Power Electronics and Drive Systems Technologies Conference (PEDSTC)*, pp. 636–640, Feb 2016.
- [16] H. A. Young, M. A. Perez, and J. Rodriguez, “Analysis of finite-control-set model predictive current control with model parameter mismatch in a three-phase inverter,” *IEEE Transactions on Industrial Electronics*, vol. 63, pp. 3100–3107, May 2016.

Apêndice A

Manual Instalação

Para utilização do Matlab/Simulink, para configuração da placa ZedBoard é necessário a instalação correta dos vários programas, pelo que é aconselhado o seguimento desde guia na integra. Desta forma é garantido o funcionamento correto do sistema, sendo que neste caso foram utilizadas as versões Matlab 2017a e Vivado 2016.2.

A.1 Instalação Matlab

Inicialmente é necessário instalar o programa Matlab, assim como instalar o pack de updates inerentes à versão instalada

Instalação Add-Ons

Após instalado o Matlab é necessária a instalação de 3 add-ons:

- “Embedded Coder Support Package for Xilinx Zynq-7000 Platform”
- “HDL Coder Support Package for Xilinx Zynq-7000 Platform”
- “MATLAB Support for MinGW-w64 C/C++ Compiler”

Se a versão instalada do Matlab foi a recomendada ou outra versão anterior a 2017b é necessário realizar, o seguinte hotfix: <https://www.mathworks.com/support/bugreports/1741173>.

Verificação da Versão Compatível do Vivado

Após instalado o Matlab é necessária, a abertura do setup do add-on “Embedded Coder Support Package for Xilinx Zynq-7000 Platform”. Para isso abra o Matlab e em Add-ons > Manage Add-Ons e no setup do add-on referido é necessário verificar qual a versão do Vivado compatível com a versão do Matlab.

A.2 Instalação Vivado

Instalar a versão do Vivado suportada pelo Matlab. Atenção que a versão a instalar tem de ser o “Vivado HL System Edition” devido à necessidade da função “System Generator for DSP”

Instalação Placas da Digilent

Após a instalação do vivado é necessário instalar as “boards” das placas da digilente utilizadas ao longo da dissertação, por isso e necessário adicionar à localização da instalação conforme está indicado no link: <https://github.com/Digilent/vivado-boards>.

Associar System Generator DSP ao Matlab

Devido a um problema de associação entre versões é necessário correr sempre este “hotfix” de forma a interligar os 2 programas, conforme o seguinte link:

<https://www.mathworks.com/matlabcentral/answers/359646-how-to-configure-xilinx-vivado-2017-2-system-generator-for-matlab2017b>

Seguidamente, é necessário abrir o Matlab através System Generator, e configurar os Add-ons através dos seus setup.

A.3 Conexão ao Hardware e testes

Após abrir o Matlab pelo system generator, é necessário ligar a placa ao pc pelo que é necessário a instalação do driver da cypress. (<https://www.cypress.com/documentation/software-and-drivers/microsoft-certified-usb-uart-driver>). Instalado os drivers, e com o Matlab aberto, testar a comunicação através dos comandos: “hdlsetuptoolpath (‘ToolName’,’xilinx Vivado’,’ToolPath’,’Localização instalação’)” A localização do ficheiro terminará em “.bat”, conforme a localização de exemplo: Vivado\2016.2\bin\vivado.bat.

Por Fim efetuando o comando “z = Zynq” que deve retornar o ip da Zedboard definido (169.254.1.10 ip por defeito para ligação direta ao pc). Garrantida a comunicação deve correr um dos exemplos como o Leb_blinking para verificar a intalação.

Nota : O *Matlab* deve ser aperto da aplicação *System Generator* do *Vivado* para funcionamento correto do *HDL Code Generator*.

Apêndice B

Manual Utilização

De forma a desenvolver um modelo para a placa ZedBoard funcionar como plataforma para controlo de potência é necessário seguir alguns passos de forma a configurar a placa de forma correta. Antes de mais, é necessário o uso de 2 programas, nomeadamente, o *Vivado* e o *Matlab*, pelo que é aconselhado seguir o manual de instalação.

B.1 Leitura de Dados da FPGA no Matlab/Simulink

Neste ponto será explicado como criar um IP core no *Matlab*, seguidamente associar a uma *board defenition* no *Vivado* de forma a monitorizar os dados de um ADC ligado a FPGA

Gerar IP Core para Captura de Dados

Baseado no guia: <https://www.mathworks.com/help/supportpkg/xilinxfpgaboards/examples/read-temperature-sensor-data-from-xilinx-fpga-board-using-fpga-data-capture.html>

Primeiro deve-se garantir que o *Xilinx Design Suite* está interligado ao *Matlab* através da execução do comando no Matlab (alterar o caminho para o local da instalação, no seu pc):

```
hdlsetuptoolpath ('ToolName','xilinx Vivado','ToolPath','C:\Apps\Vivado\2016.2\bin\nvivo.bat')
```

Criar uma nova pasta e colocar nessa mesma pasta o diretório do *Matlab* e executar Correr o comando “generateFPGADataCaptureIP” que deverá abrir uma GUI (Figura B.1)

Após aparecimento da GUI, deve-se indicar o nome dos sinais que é pretendido analisar bem como as suas dimensões. Posteriormente deve-se colocar a linguagem em VHDL e seleccionar o tamanho da captura e clicar no botão gerar. Após a geração ser concluída, aparecerá uma nova janela a confirmar a execução. Agora ao abrir a pasta “hdlsrc” gerada automaticamente existem vários ficheiros vhdl que têm de ser adicionados à “board defenition”, e um ficheiro .slx que é o ficheiro de Simulink a utilizar para análise dos valores lidos pelo ADC.

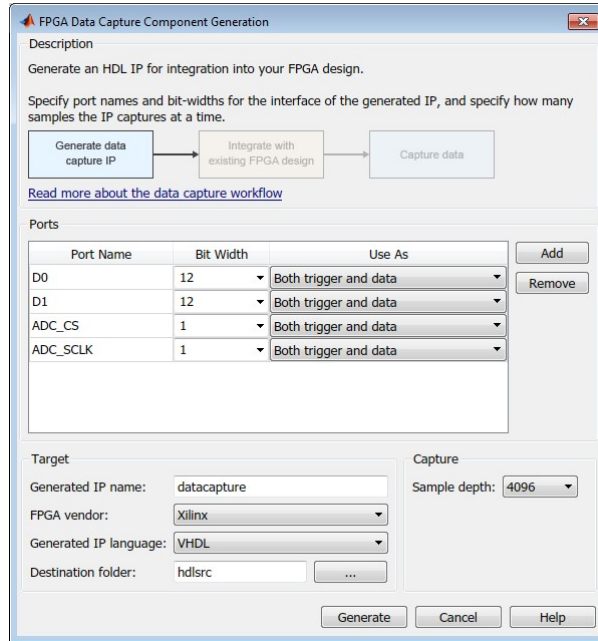


Figura B.1 Gui para gerar IP Core para captura de dados.

Criar uma Board Defenition no Vivado

Baseado no guia: <https://reference.digilentinc.com/learn/programmable-logic/tutorials/zedboard-getting-started-with-zynq/start>

Deve criar um novo projeto e selecionar a “Board” ZedBoard da *Digilent* conforme está representado na figura B.2.

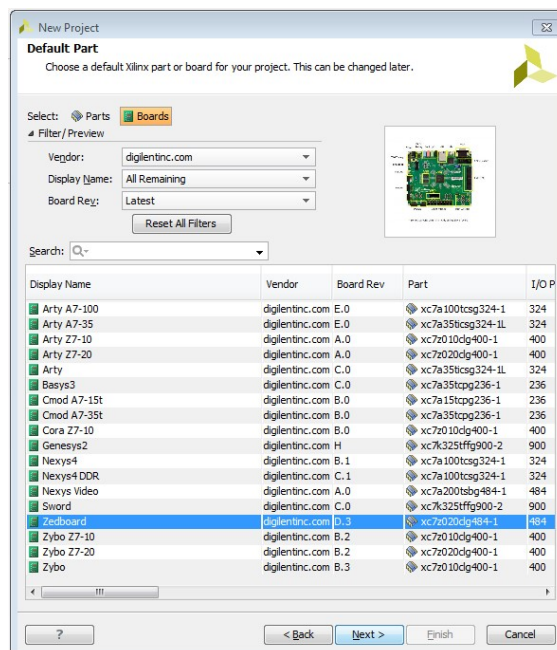


Figura B.2 Seleção “Board” para um novo projeto.

B.1 Leitura de Dados da FPGA no Matlab/Simulink

Seguidamente após estar o novo projeto aberto no Vivado, no painel de navegação, situado no lado esquerdo, clicar em “Create block design” e escolher o nome.

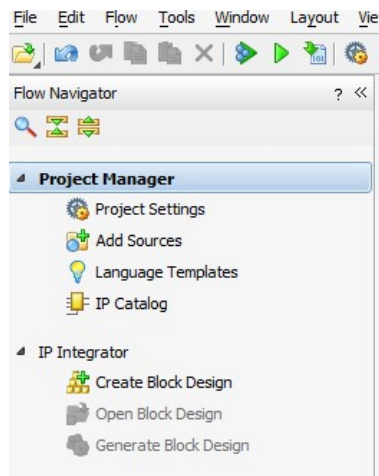



Figura B.3 Painel de navegação no lado esquerdo com a opção “Create Block Design”.

Criado o novo “block design” deve clicar no icon  e adicionar os seguintes 3 ip cores: ZYNQ7 Processing System, Processing system Reset e AXI Interconnect, no primeiro bloco é necessário clicar com o click direito do rato e correr o comando “Run Block Automation” Feito estes passos deve-se ligar os 3 blocos conforme está representado na figura B.4.

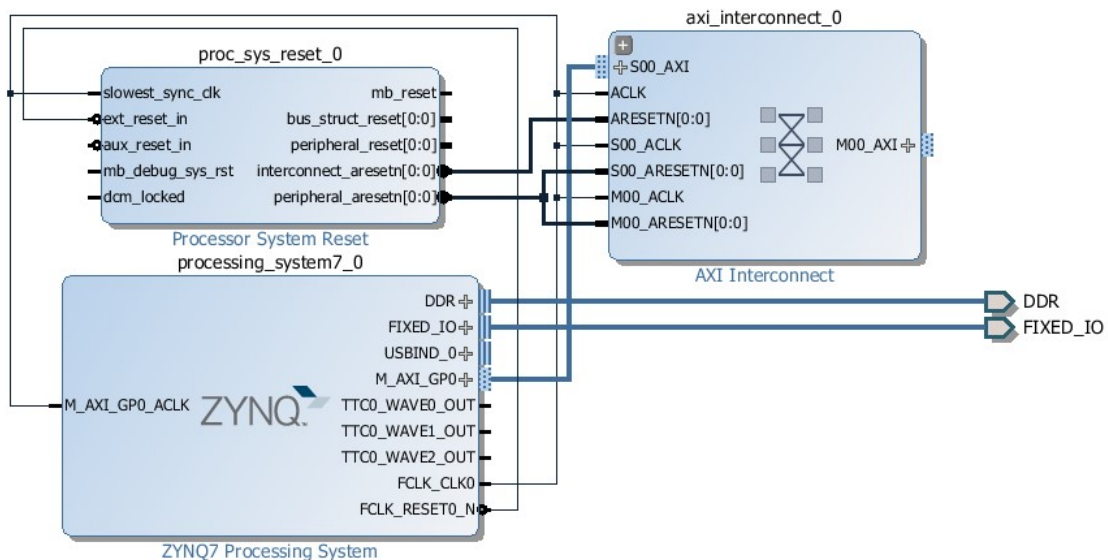




Figura B.4 Esquema básico da placa ZedBoard

Estando o diagrama todo interligado conforme é apresentado na figura 4, ao clicar no lado direito do rato é possível executar o comando “Regenerate Layout” ou clicando no ícone  no menu de atalhos rápidos. Após a organização do esquema de forma automática deve correr a análise ao sistema (Validate Design), que tem o seguinte ícone . Garantida a coerência do esquema

elaborado, passa-se ao último passo que é a geração de uma interligação VHDL entre os blocos, sendo que para isso é necessário criar um “HDL wrapper”.

Criar HDL wrapper: na janela “block design”, separador “Sources”, click direito do rato no ficheiro “block diagram”, “name.bd” e escolher a opção “Create HDL Wrapper”, conforme está referenciado no quinto passo do tutorial apresentado no início do presente capítulo. Isto Cria o modelo superior do esquema. A figura B.5 apresenta a hierarquia do esquema antes e após a execução deste comando.

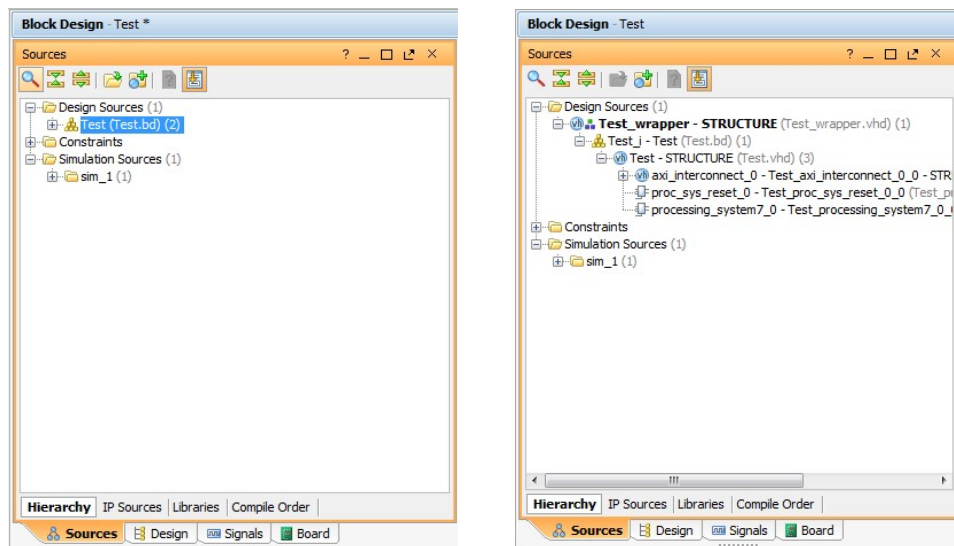


Figura B.5 “Design Sources” antes (a) e depois (b) de executar o HDL wrapper.

Integração de IP Cores no Vivado

A integração de IP Cores é um passo importante pois existem blocos que têm de ser desenvolvidos em VHDL ou blocos da biblioteca da Digilent no GitHub, têm de ser adicionados a cada novo projeto. Para isso cada IP core a adicionar deve ser colocado numa pasta com o nome pretendido na raiz do projeto conforme é apresentado na figura B.6.

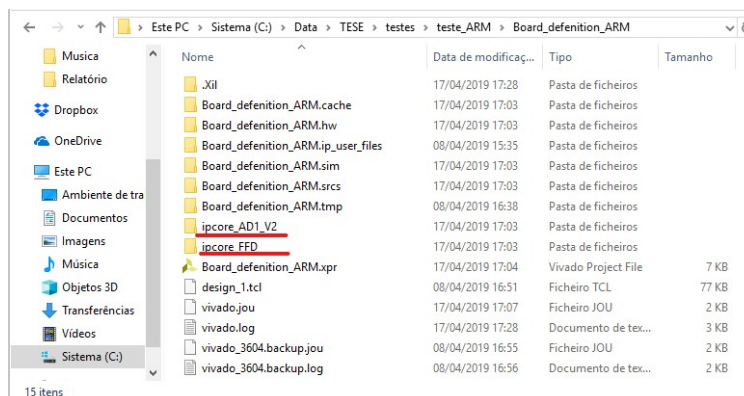


Figura B.6 Exemplo de localização das pastas com os ficheiros VHDL dos IP Cores.

B.1 Leitura de Dados da FPGA no Matlab/Simulink

Seguidamente no projeto a desenvolver é necessário adicionar os ficheiros VHDL (“Add Sources”). Clicar no botão direito do rato no nome do projeto e escolher “Add sources”. Abre uma janela para adição de ficheiros de qualquer tipo, para o caso de novos ficheiros VHDL selecione a opção “Add or create design sources”, na janela seguinte escolha “add files” e selecione todos os ficheiros, no caso de existirem várias pastas de IP Cores, pode escolher “add directory” e selecionar as várias pastas que contêm os ficheiros VHDL

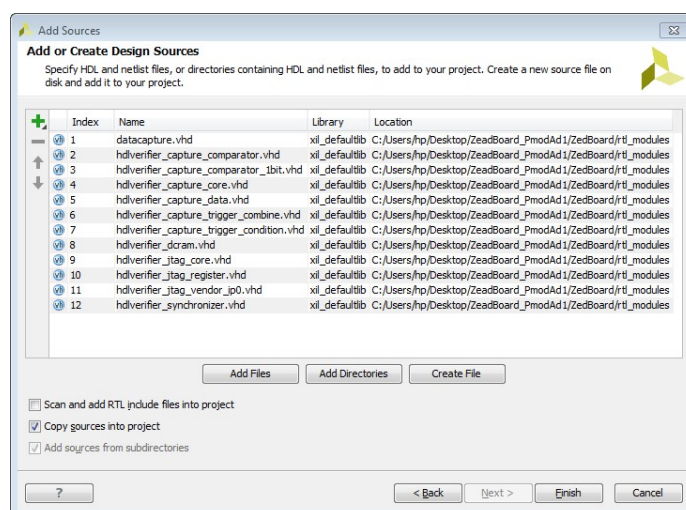


Figura B.7 Exemplo da janela de adição de ficheiros VHDL.

Adicionados os ficheiros VHDL é necessário transformar os ficheiros em blocos para seguidamente adicionar ao “block design” a desenvolver. Para isso é necessário realizar a função “Open IP Package”. Clicar em “Tools” → “Create and Package IP”. Na janela que abre, selecionar “Package a specified directory” conforme é apresentado na figura B.8.

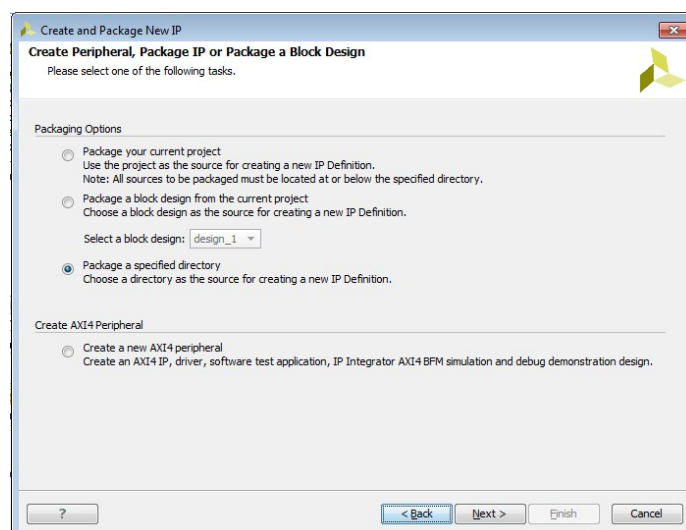


Figura B.8 Ferramenta "Create and Package IP Core".

No diretório escolher a pasta do IP Core, no caso de vários IP Cores é necessário repetir o processo para cada um e escolher a pasta relativa ao ip core que se quer adicionar. Continuando com a adição

dos ficheiros VHDL ao “block design” irá ser aberto um projeto temporário conforme é apresentado na figura B.9. Deve-se abrir a aba “Review and Package”, e verificar se as saídas e entradas do bloco correspondem às esperadas, se tudo estiver conforme pretendido deve clicar no botão “Package IP”.

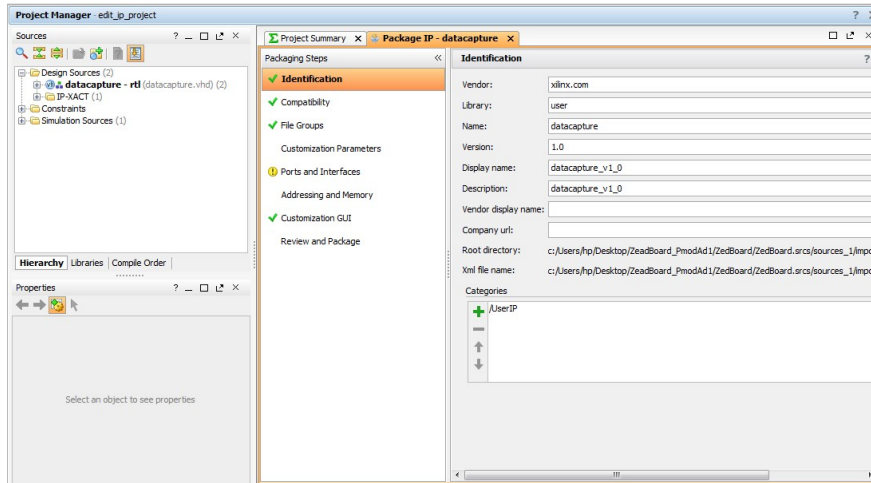




Figura B.9 Projeto temporário criado pela ferramenta " Create and Package New IP " do passo anterior.

Gerados os IP Cores no “block design” pode-se fechar o projeto temporário (é esperado que feche automaticamente após o passo anterior) e no “block design” a desenvolver deve-se procurar o bloco com o nome atribuído, visível na figura B.9. A figura B.10 apresenta o esquema já com os IP Cores adicionados, nomeadamente, o “datacapture _0” e o “pmodad1 _test _0”.

A adição dos blocos é idêntica a dos demais blocos, sendo que é importante ligar corretamente os mesmos, e executar o comando “Regenerate Layout” ou clicando no ícone  no menu de atalhos rápidos. Após a organização do esquema de forma automática deve correr a análise ao sistema (Validate Design), que tem o seguinte ícone . Isto de forma a garantir que o esquema está corretamente interligado.

No presente caso é necessário a adição de portas de entrada (ADC _D0 e ADC _D1) e de saída (ADC _CS e ADC _SCLK), que podem ser inseridas através do atalho ctrl+k. Gerado o esquema da placa a utilizar na FPGA é necessário fazer o mapeamento de portas, isto é, definir os sinais a que porta correspondem. Assumindo que o ADC pmod AD1 da *Digilent* será ligado ao conector Pmod JA1. Estas definições são colocadas num ficheiro “.xdc” conforme será exposto. Inicialmente é necessário identificar os pinos conectados à ficha pmod JA1 da zedboard, existente no “datasheet” da placa. Na tabela B.1 esta representada a correspondência entre os pinos da placa Zedboard e a porta do conector Pmod JA1.

A adição do ficheiro do tipo “.xdc” é idêntica a adição dos ficheiros VHDL. Assim sendo, deve-se clicar no lado direito do rato no nome do projeto → “Add Source files” → ”Add or create constrains” → Adicionar o ficheiro .xdc com o nome desejado. Neste exemplo será utilizado o nome “Zedboard.xdc” com o código apresentado na figura B.11.

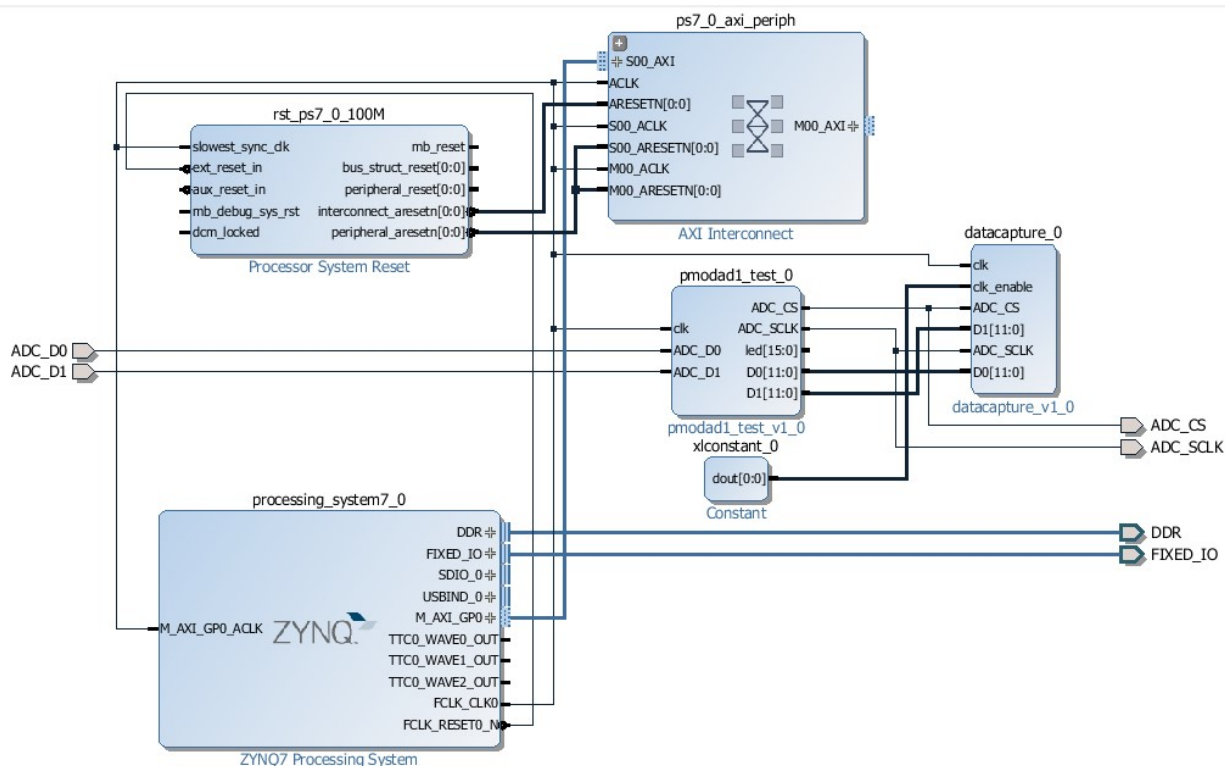


Figura B.10 Zedboard “block design” com os IP Cores “datacapture” e “pmodad1_test”.

Tabela B.1 Correspondência entre pinos Zedboard e conector Pmod JA1.

Sinal Pmod AD1	ZedBoard (I/O)	ZedBoard PinMap
CS (Clk para Amostras)	JA1	Y11
D0 (Sinal digital 1 ADC)	JA2	AA11
D1 (Sinal digital 2 ADC)	JA3	Y10
SCLK (Clk para Envio Dados)	JA4	AA9

Posto isto a definição da placa está concluída pelo que para finalizar apenas e preciso exportar o ficheiro “.tcl”, importante no passo de configurar a ZedBoard a partir do Matlab. Em “file” → Export → Export Block Design, é possível exportar o ficheiro “.tcl” para a raiz da localização do projeto.

Captura Dados no Matlab

Baseado no guia: <https://www.mathworks.com/help/supportpkg/xilinxfpgaboards/examples/read-temperature-sensor-data-from-xilinx-fpga-board-using-fpga-data-capture.html>

Deve-se abrir o Matlab na localização “hdlsrc”, onde no ponto “Gerar IP Core” foi gerado o IP Core anexado ao Vivado. Ao executar o comando “launchDataCaptureApp” no Matlab abre o GUI apresentado na figura 12.

```
3 # -----
4 # JA Pmod - Bank 13
5 # -----
6 set_property PACKAGE_PIN Y11 [get_ports {JA1}]; # "JA1"
7 set_property PACKAGE_PIN AA8 [get_ports {JA10}]; # "JA10"
8 set_property PACKAGE_PIN AA11 [get_ports {JA2}]; # "JA2"
9 set_property PACKAGE_PIN Y10 [get_ports {JA3}]; # "JA3"
10 set_property PACKAGE_PIN AA9 [get_ports {JA4}]; # "JA4"
11 set_property PACKAGE_PIN AB11 [get_ports {JA7}]; # "JA7"
12 set_property PACKAGE_PIN AB10 [get_ports {JA8}]; # "JA8"
13 set_property PACKAGE_PIN AB9 [get_ports {JA9}]; # "JA9"
14
15 |
16 # ADC
17 #-----
18 set_property -dict { PACKAGE_PIN Y11 IOSTANDARD LVCMOS33 } [get_ports { ADC_CS }];
19 set_property -dict { PACKAGE_PIN AA11 IOSTANDARD LVCMOS33 } [get_ports { ADC_D0 }];
20 set_property -dict { PACKAGE_PIN Y10 IOSTANDARD LVCMOS33 } [get_ports { ADC_D1 }];
21 set_property -dict { PACKAGE_PIN AA9 IOSTANDARD LVCMOS33 } [get_ports { ADC_SCLK }];
```

Figura B.11 Ficheiro “.xdc” com definição da correspondência entre pinos da ZedBoard com o conector JA1.

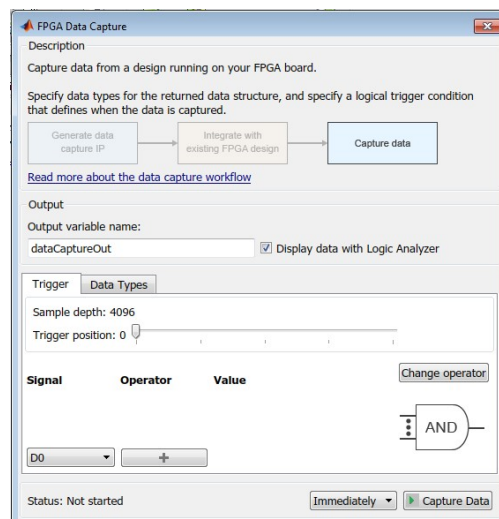


Figura B.12 Matlab Data Capture GUI.

Ao clicar em “Capture Data” é iniciado o processo de captura de dados de imediato. Outra hipótese é adicionar um trigger com base num ou mais sinais. É importante ter em atenção o número de amostras pois, estas são limitadas pelo “sample depth”. Os dados são guardados no “workspace” e apresentado no “logic analyser”.

No Simulink

É necessário abrir o ficheiro “datacapture _model.slx” existente na pasta “hdlsrc” gerada no passo “Gerar IP Core”. Abrir o modelo e clicar em “Run” faz com que a informação seja adquirida e apresentada no Simulink. O Tempo de simulação corresponde ao número de amostras retiradas, sendo que deve ser inferior ao número das “Sample depth” do bloco “datacapture” gerado anteriormente. No caso de ser necessário existir um trigger específico é necessário entrar no bloco “FPGA data capture” e indicar o trigger.

B.2 Gerar IP Core através do Simulink

A geração de IP Cores será dividida em 2 passos um primeiro que consiste na utilização de uma “board defenition” específica como a produzida no passo anterior que consiste numa Zedboard com um ADC anexado no Pmod JA1.

Anexar uma "Board Defenition" ao Matlab

Baseado no guia: <https://www.mathworks.com/help/hdlcoder/examples/define-and-register-custom-board-and-reference-design-for-zynq-workflow.html>

De forma a utilizar um esquema personalizado no Matlab é necessário cumprir uma série de regras de forma a que o Simulink consiga utilizar os ficheiros. Alguns exemplos destes esquemas podem ser encontrados em:

Matlab2017a\toolbox\hdlcoder\hdlcoderdemos\customboards

Estes esquemas são compostos por 2 componentes sendo estas: “the board files” e “the reference design files”, cada um com um ficheiro “customization.m” e um “plugin.m”.

The customization file - tem apenas como função indicar a localização do ficheiro plugin que o Matlab precisa de ler.

The plugin file – este é o ficheiro mais importante pois contem as definições para o Matlab e Simulink entenderem o esquema da placa ZedBoard. os ficheiros plugin são o plugin_board.m e o plugin_rd.m

Plugin_board.m file:

este ficheiro contem as informações da placa a utilizar, no presente caso, a ZedBoard e o mapeamento das entradas e saídas conforme é apresentado na figura B.13.

```

1  function hB = plugin_board()
2  % Board definition
3
4  % Copyright 2012-2014 The MathWorks, Inc.
5
6  % Construct board object
7  hB = hdlcoder.Board;
8
9  hB.BoardName = 'ZedBoard_datacapt';
10
11 % FPGA device information
12 hB.FPGAVendor = 'Xilinx';
13 hB.FPGAFamily = 'Zynq';
14 hB.FPGADevice = 'xc7z020';
15 hB.FPGAPackage = 'clg484';
16 hB.FPGASpeed = '-1';
17
18 % Tool information
19 hB.SupportedTool = {'Xilinx Vivado', 'Xilinx ISE'};

```

(a) Este ficheiro define o nome da placa (ZedBoard_datacapt) e como aparecerá no Simulink, as informações da FPGA e as ferramentas para compatibilidade com a FPGA.

```

24  %% Add interfaces
25  % Standard "External Port" interface
26 - hB.addExternalPortInterface( ...
27      'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
28
29  % Custom board external I/O interface
30 - hB.addExternalIOInterface( ...
31      'InterfaceID',    'LEDs General Purpose', ...
32      'InterfaceType',  'OUT', ...
33      'PortName',      'GPLEDs', ...
34      'PortWidth',     8, ...
35      'FPGAPin',       {'T22', 'T21', 'U22', 'U21', 'V22', 'W22', 'U19', 'U14'}, ...
36      'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
37
38 - hB.addExternalIOInterface( ...
39      'InterfaceID',    'DIP Switches', ...
40      'InterfaceType',  'IN', ...
41      'PortName',      'DIPSwitches', ...
42      'PortWidth',     8, ...
43      'FPGAPin',       {'F22', 'G22', 'H22', 'F21', 'H19', 'H18', 'H17', 'M15'}, ...
44      'IOPadConstraint', {'IOSTANDARD = LVCMOS18'});
45

```

(b) “addExternalIOInterface” é utilizado para definir as interfaces de E/S da ZedBoard, por exemplo os LEDs, interruptores DIP, Push Buttons, ...etc.

```

54 - hB.addExternalIOInterface( ...
55     'InterfaceID',    'Pmod Connector JA1', ...
56     'InterfaceType',  'INOUT', ...
57     'PortName',      'PmodJA1', ...
58     'PortWidth',     8, ...
59     'FPGAPin',       {'Y11', 'AA11', 'Y10', 'AA9', 'AB11', 'AB10', 'AB9', 'AA8'}, ...
60     'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
61
62 - hB.addExternalIOInterface( ...
63     'InterfaceID',    'Pmod Connector JB1', ...
64     'InterfaceType',  'INOUT', ...
65     'PortName',      'PmodJB1', ...
66     'PortWidth',     8, ...
67     'FPGAPin',       {'W12', 'W11', 'V10', 'W8', 'V12', 'W10', 'V9', 'V8'}, ...
68     'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
69
70 - hB.addExternalIOInterface( ...
71     'InterfaceID',    'Pmod Connector JC1', ...
72     'InterfaceType',  'INOUT', ...
73     'PortName',      'PmodJC1', ...
74     'PortWidth',     8, ...
75     'FPGAPin',       {'AB7', 'AB6', 'Y4', 'AA4', 'R6', 'T6', 'T4', 'U4'}, ...
76     'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

```

(c) Conectores Pmod são definidos são utilizados para interligar o PmodAD1 à board.

Figura B.13 Ficheiro da ZedBoard plugin _board.m

Plugin _rd.m file:

Este ficheiro indica quais os esquemas a implementar na FPGA, como tal tem de indicar 3 ficheiros conforme é indicado na figura B.14. Na linha 18 o ficheiro “.tcl”, que indica quais os blocos e de que forma os blocos são interligados. Os IP Cores a utilizar, no exemplo dado anteriormente seria o bloco “datacapture” e “Pmod _Ad1” E por ultimo é preciso indicar o ficheiro “.xdc” de forma ao Matlab saber qual a localização das entradas e saídas físicas da placa ZedBoard.

```
15 %% Add custom design files
16 % add custom Vivado design
17 - hRD.addCustomVivadoDesign( ...
18     'CustomBlockDesignTcl', 'design_1.tcl', ...
19     'VivadoBoardPart',      'digilentinc.com:zedboard:part0:1.0');
20 %'em.avnet.com:zed:part0:1.0');
21
22 %add the PmodAD1 custom core
23 - hRD.CustomFiles = {'ipcore'};
24
25 % Add constraint files
26 - hRD.CustomConstraints = {'ZedBoard.xdc'};
```

Figura B.14 Ficheiro plugin _rd.m da ZedBoard.

Nota Importante:

Qualquer novo IP Core adicionado ao esquema da ZedBoard no Vivado tem de ser adicionado também na pasta “ipcore” para o matlab conseguir aceder aos ficheiros VHDL adicionados. É importante que a pasta tenha este nome de forma ao matlab compreender o tipo de ficheiros. Após qualquer alteração no “block design” é necessário gerar o novo ficheiro “.tcl” no Vivado

B.3 Utilizar o "HDL Workflow Advisor"

Baseando em:

<https://es.mathworks.com/help/hdlcoder/examples/define-and-register-custom-board-and-reference-design-for-zynq-workflow.html>

<https://es.mathworks.com/help/hdlcoder/examples/getting-started-with-hardware-software-codesign-workflow-for-xilinx-zynq-platform.html>

Exemplo em video:

<https://es.mathworks.com/campaigns/products/partner/xilinx-zynq-design-with-simulink.html>

A ferramenta "HDL Workflow Advisor" do *Matlab*, permite converter para VHDL algoritmos desenvolvidos em funções do *Matlab* ou modelos do *Simulink*. Será utilizada esta ferramenta para transformar o código desenvolvido no *Simulink*, num IP Core, que permite a sua implementação na ZedBoard através do *Vivado*. Além da geração do IP Core o "Workflow Advisor" também contempla a elaboração do fluxo de bits (bitstream) para configurar a FPGA. Seguindo este guia é possível implementar diretamente na FPGA algoritmos desenvolvidos no *Simulink*.

Adição de uma nova "Board Defenition" ao Matlab

Para isso basta executar o seguinte comando na consola do *Matlab* (ZedBoard _datacapt deve ser alterado para o nome da *Board Defenition* desenvolvida):

```
addpath ( fullfile ( matlabroot , 'toolbox','hdlcoder' , 'hdlcoderdemos' , 'customboards' , 'ZedBoard_datacapt' ) ).
```

Executar o "Workflow Advisor"

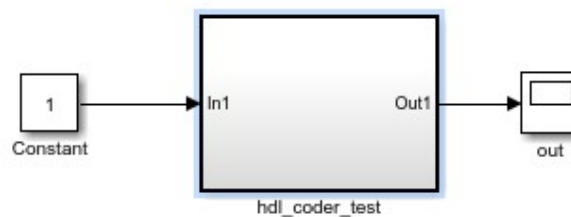



Figura B.15 Exemplo de um subsistema onde será executado o workflow advisor.

A figura B.15 representa um modelo básico, de um bloco de teste qualquer com uma entrada (constante) e uma saída (osciloscópio). A saída será ligada a um dos LED's da FPGA, enquanto a entrada será uma constante a "1" de forma a permitir acender o LED.

Para iniciar o "workflow advisor" deve-se clicar com o lado direito no rato no subsistema e escolher:

HDL code → Workflow advisor.

A figura anterior (B.16) representa a janela do "HDL Workflow Advisor". Este processor de conversão do modelo do *Simulink* num IP Core é composto por **quatro** passos, cada um com vários passos. Cada um dos passos tem de ser executado sem erros de forma a permitir a criação do IP Core e integração ao modelo do *Vivado* para implementação na ZedBoard. Cada passo realizado com sucesso é marcado com .

Passo 1 "Set Target"

Neste primeiro passo, é escolhida a placa para aplicar o algoritmo, e o caminho das entradas e saídas são definidos. Este é o passo mais crítico, que requer mais cuidado, pelo que a maioria dos restantes pode ser realizado automaticamente.

Passo 1.1 "Set Target Device and Synthesis Tool"

Na opção "*Target Platform*" representada na figura B.17 com um rectângulo a vermelho, deve ser escolhida a *Board Defenition* adicionada. Seguidamente os dados devem ser ajustados para a

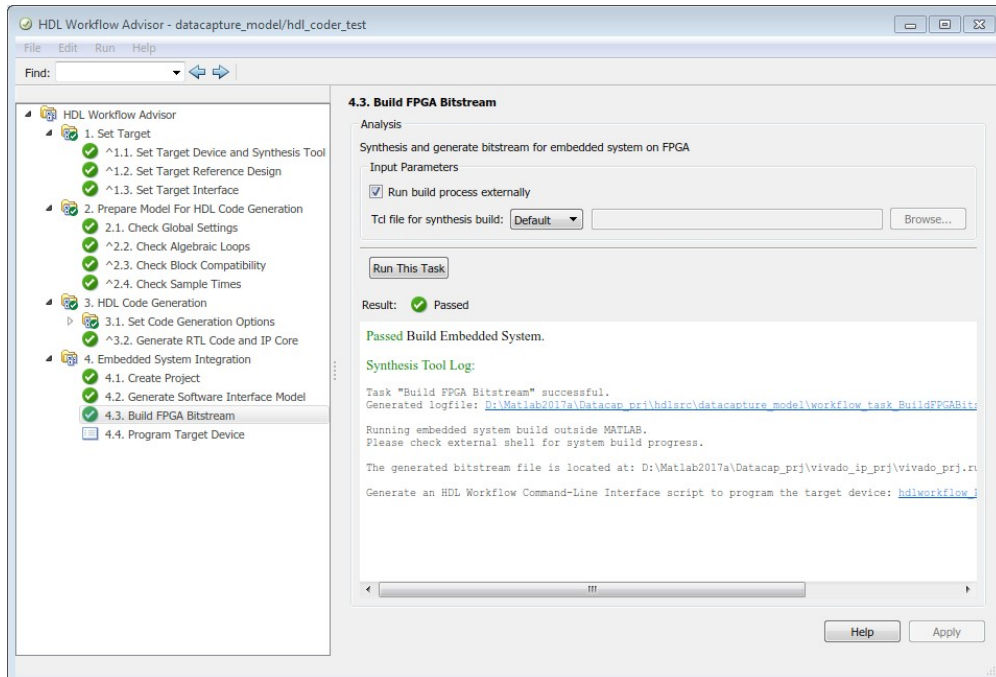


Figura B.16 Ferramenta *HDL Workflow Advisor*.

automaticamente para a ferramenta de síntese a utilizar (*Xilinx Vivado*), caso contrário deve ajustar manualmente.

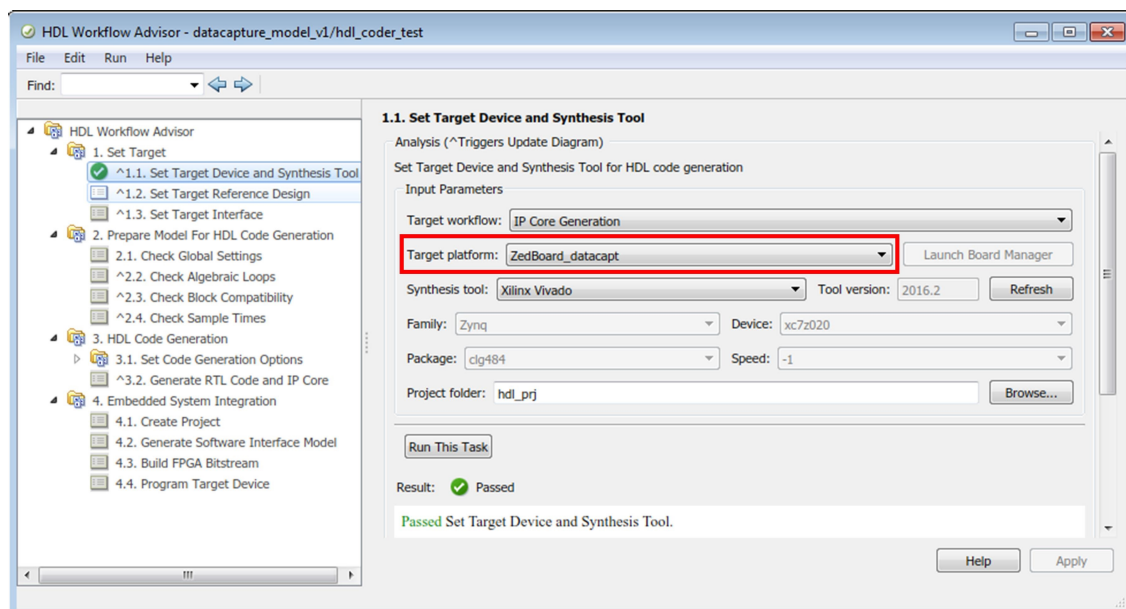


Figura B.17 Janela do *HDL Workflow Advisor* durante o a escolha da plataforma.

Passo 1.2 "Set Target Reference Design"

Como a ferramenta de síntese já foi definida no passo anterior, este passo não precisa de modificações pelo que é só executar. Para isso basta clicar em "Run Task".

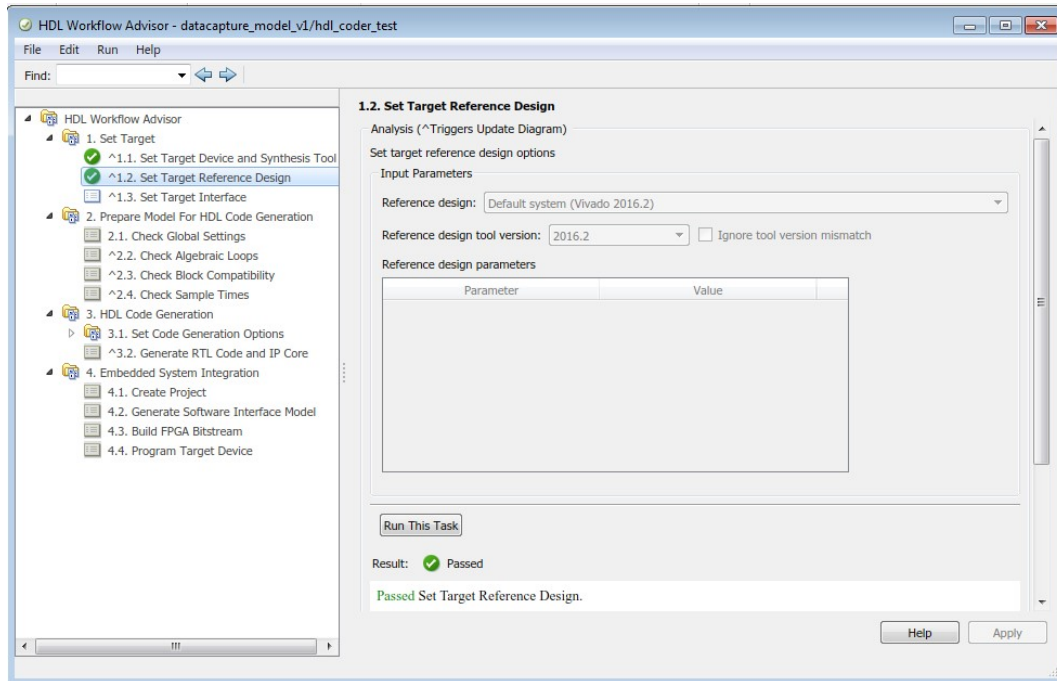


Figura B.18 Janela do *HDL Workflow Advisor* durante o ponto 1.2.

Passo 1.3 "Set Target Interface"

Neste ponto são identificadas as entradas e saídas do sistema, às entradas e saídas físicas da ZebBoard. Esta atribuição está representada na figura B.19.

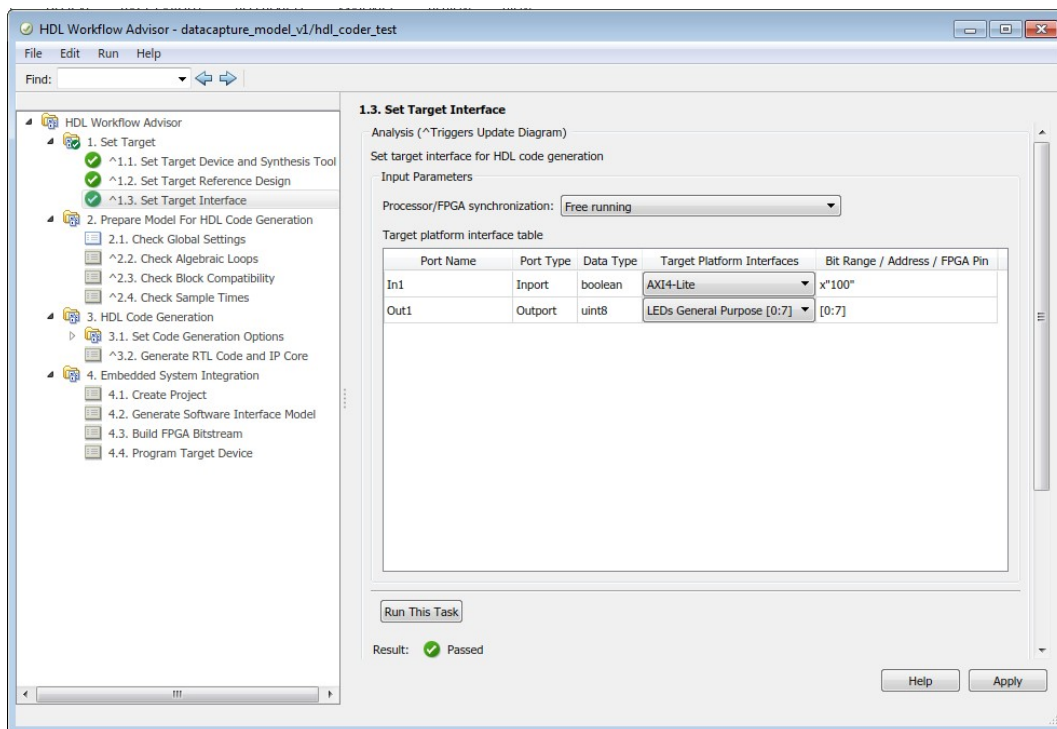


Figura B.19 Janela do *HDL Workflow Advisor* durante o ponto 1.3.

Quanto as interfaces disponíveis estas estão representadas nas figura B.20

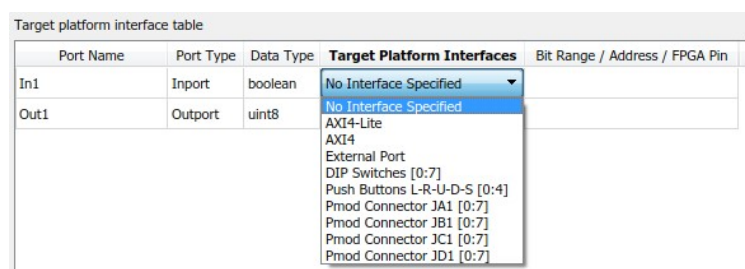


Figura B.20 Janela do *HDL Workflow Advisor* durante o ponto 1.3.

Se o sinal é um valor a receber ou a enviar para *Simulink*:

Nesta situação é necessário que o HDL gere um bloco que permita ao processador comunicar com a FPGA, através do barramento AXI4 que interliga as duas unidades de processamento. Nas opção da figura B.20, deve-se escolher o AXI4-Lite pois apresenta uma maior simplicidade de comunicação, e utiliza os registos desta ligação para um envio de dados mais rápido.

Se o sinal é uma entrada ou saída presente na FPGA:

No caso de entradas físicas da FPGA, estas podem ser LEDs, botões de pressão, interruptores ou as portas PMOD. Se for necessário especificar o pino na qual o sinal deve ser ligado, deve ser escolhida a opção "External Port", e após o ponto 4.1 do "workflow advisor" ajustar manualmente no *Vivado* o sinal para a porta física correspondente.

Passo 2 "Prepare Model For HDL Code Generation"

A função deste passo é verificar se o modelo desenvolvido é compatível com o "HDL Coder" para conversão do algoritmo para um IP Core. Deve-se deixar as opções padrão e eecutar todos os pontos deste passo.

Passo 3 "HDL Code Generation"

Este ponto é responsável pela geração do IP Core equivalente ao bloco de *Simulink* desenvolvido. Uma vez mais devem ser executado sem alteração das definições. No final deste ponto será aberto um relatório (ficheiro do tipo .html) onde é resumido o processo de geração do bloco, e os ficheiro produzidos pelo "HDL Coder".

Passo 4 "Embedded System Integration"

A ultima etapa consiste na integração do IP Core gerado, na "Board Defenition" seleccionada no ponto 1.1. Também neste ponto é desenvolvido o modelo que permite ao a FPGA comunicar com o processador, através dos registos AXI4, seleccionados no ponto 1.3.

Passo 4.1 "Create Project"

Devido aos passos anteriores permanecerem com as escolhas por defeito, e possível executar os vários passos, até o ponto 4.1 escolhendo a opção "Run To Select Task" (conforme a figura B.21)

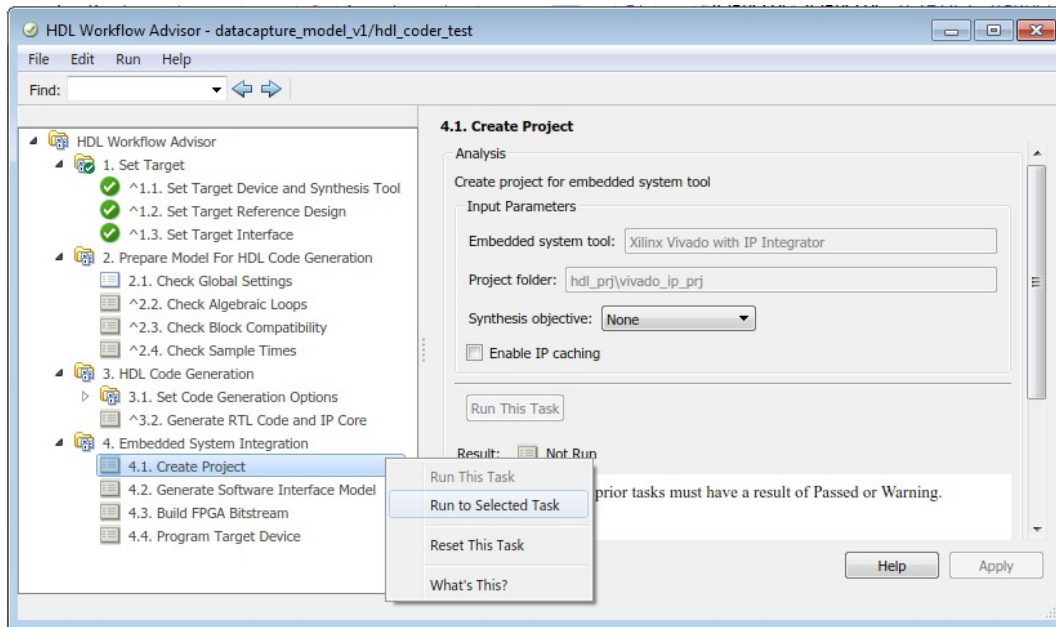


Figura B.21 Janela do *HDL Workflow Advisor* no ponto para criar o projeto no *Vivado*.

Após a geração do projeto no *Vivado* com o IP Core adicionado com sucesso, será apresentada uma janela idêntica à da figura B.22, que apresenta uma hiperligação para abrir o mesmo.

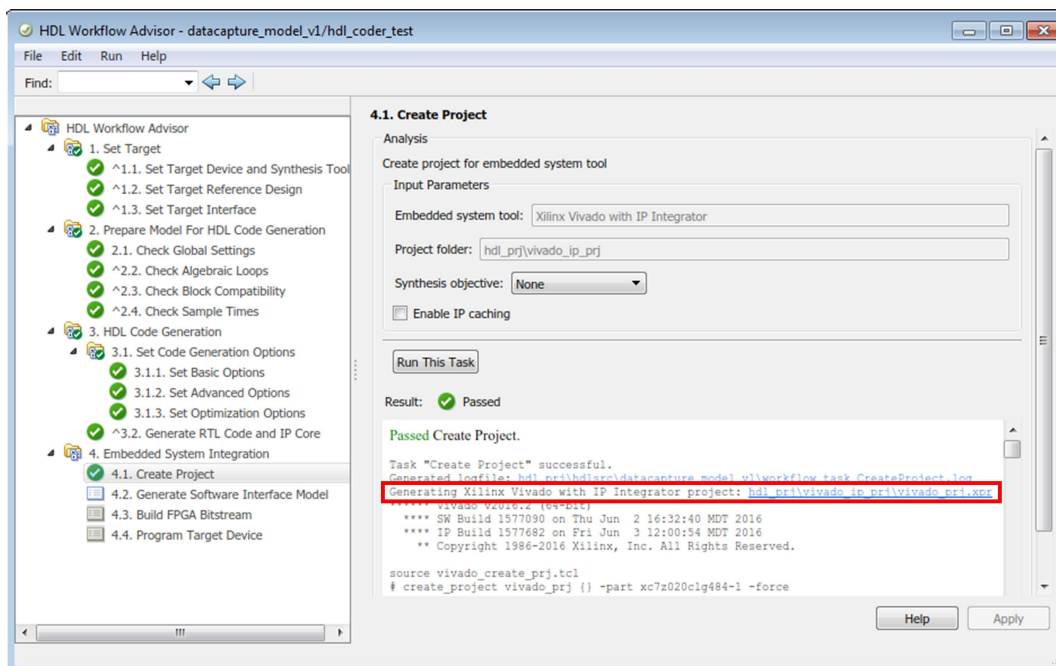


Figura B.22 Janela do *HDL Workflow Advisor* após criação do projeto no *Vivado*.

Nota Importante:

Caso sejam adicionadas entradas ou saídas no ponto 1.3 referenciadas a portas externas ("External Port") é necessário alterar manualmente o projecto no *Vivado* manualmente. Para realizar esta atribuição dos sinais às portas correspondentes é necessário abrir o projecto (hiperligação identificada a vermelho na figura B.22)

Se for necessária a atribuição manual deve saltar as seguintes etapas e passar para a próxima secção.

Passo 4.2 "Generate Software Interface model"

No final deste passo o *Simulink*, desenvolve um modelo que permite ao processador comunicar com a FPGA. O modelo consiste uma correspondência dos AXI4-lite a um modelo do *Simulink*, que permite ao processador enviar ou receber dados da FPGA pelos endereços defendidos no ponto 1.3. para melhor percepção deste ponto pode ver o seguinte vídeo:

<https://es.mathworks.com/campaigns/products/partner/xilinx-zynq-design-with-simulink.html>

No caso de apenas ser utilizada a FPGA, é possível saltar este passo, através da escolha da opção "Skip Task".

Passo 4.3 "Build FPGA Bitstream"

Este ponto simplesmente executa os comandos necessários para produção de um ficheiro "Bitstream" (ficheiro .bit) para programação da FPGA. É de salientar que em situações de algoritmos complexos, este passo pode chegar a demorar várias horas.

Passo 4.4 "Program Target Device"

Após todos os passos anteriores estarem concluídos, pode-se programar a FPGA. Esta programação é realizada através do envio do ficheiro .bit para a mesma. Deve-se seleccionar a opção "Download" de forma a enviar o código para o cartão de memória, e sempre que a ZedBoard seja ligado, o código seja executado.

B.4 Alteração do Projeto no Vivado

Conforme foi referido no passo 4.1 "Create Project", no caso de existirem entradas e saídas do sistema, referidas no "Worklow Advisor" como "External Port", é necessário atribuir manualmente

a porta física correspondente. Para isso é necessário entrar no projeto do *Vivado*, gerado no ponto 4.1, para isso deve-se abrir a hiperligação apresentada na figura B.22.

No *Vivado*, deve-se deparar com um projeto idêntico ao da figura B.23, onde está assinalado a vermelho o IP Core produzido no passo anterior.

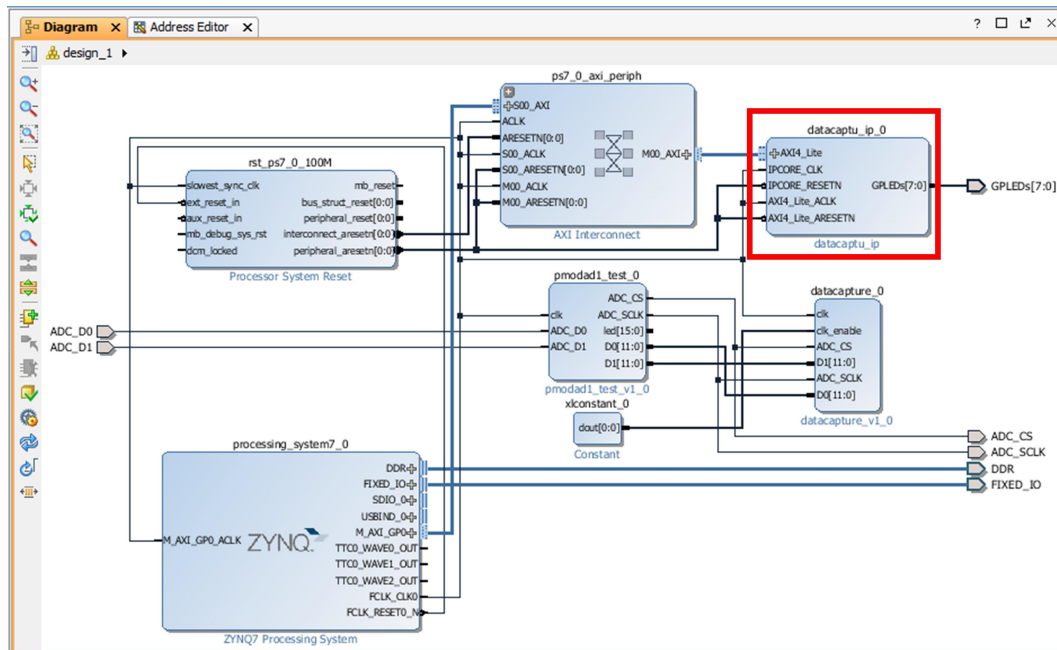


Figura B.23 Janela do *Vivado* com o projeto criado a partir do *Workflow Advisor*.

As entradas e saídas do IP Core que foram selecionadas no "HDL Workflow Advisor", etapa (1.3) estão correctamente ligadas, sendo que a entradas está ligada ao AXI4 (valor proveniente do processador), e a saída aos LED's.

No entanto as conexões podem ser manipuladas manualmente conforme seja desejado, as entradas ou saídas que sejam especificadas como portas externas, têm de ser sempre ajustadas manualmente conforme será exemplificado.. O módulo ADC, presente na figura B.23 ainda não tem os sinais ADC_CS e ADC_SCLK encaminhados para a porta PMOD correspondente, pelo que será os sinais e exemplo.

O primeiro passo é atribuir um sinal como entrada ou saída no *Vivado*. Através do atalho "Ctrl+K" é aberta a opção "Create Port", (conforme está na figura B.24), seguidamente deve ser colocado o nome da porta, e se o mesmo é uma entrada ou saída. Seguidamente esta porta de entrada ou saída deve ser interligada a porta do bloco pretendido.

O último passo para garantir o funcionamento da porta adicional é a atribuição do pino físico correspondente. Esta identificação é adicionada nas "constrains" (ficheiro .xdc) (igual ao representado na figura B.11). O comando para as portas PMOD é:

```
set_property -dict {PACKAGE_PIN Y11 IOSTANDARD LVCMOD33} [get_ports {ADC_CS}]
set_property -dict {PACKAGE_PIN A99 IOSTANDARD LVCMOD33} [get_ports {ADC_SCLK}]
```

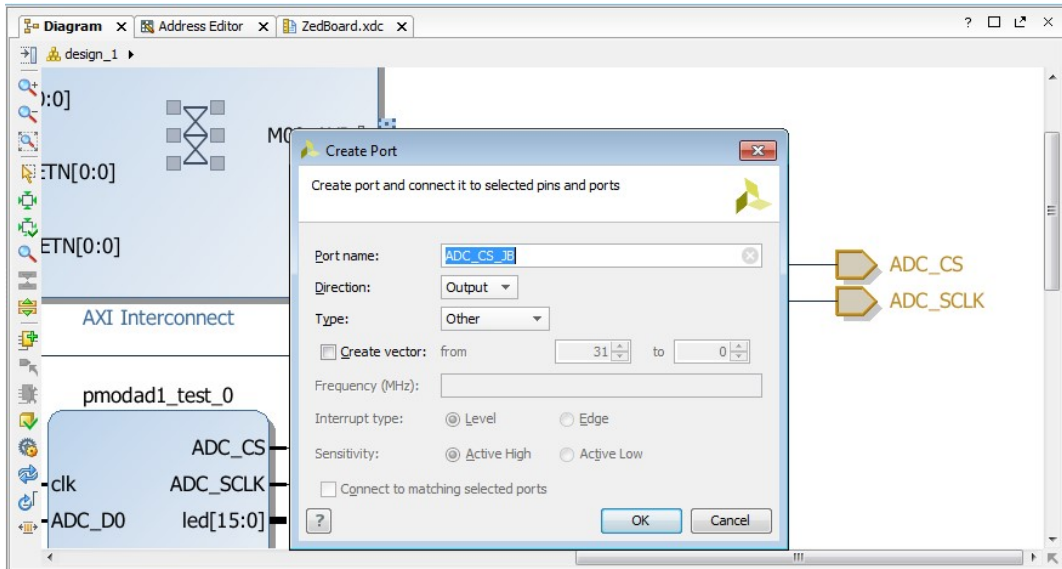


Figura B.24 Ferramenta "create port" para adicionar portas de entrada e/ou saída no Vivado.

O "PACKAGE_PIN" corresponde ao pino da ZedBoard, e o "get_port" ao sinal que a que o pino é atribuído. Para o caso das portas PMOD, a tabela B.2 contém a correspondência entre o id de cada pino e a porta correspondente de todos os PMODs da ZedBoard.

Tabela B.2 Correspondência entre os id dos pinos da ZedBoard com as suas portas PMOD.

Pmod	Signal Name	Zynq pin	Pmod	Signal Name	Zynq pin
JA1	JA1	Y11	JB1	JB1	W12
	JA2	AA11		JB2	W11
	JA3	Y10		JB3	V10
	JA4	AA9		JB4	W8
	JA7	AB11		JB7	V12
	JA8	AB10		JB8	W10
	JA9	AB9		JB9	V9
	JA10	AA8		JB10	V8

Pmod	Signal Name	Zynq pin	Pmod	Signal Name	Zynq pin	Pmod	Signal Name	Zynq pin	MIO
JC1 Differential	JC1_N	AB6	JD1 Differential	JD1_N	W7	JE1 MIO Pmod	JE1	A6	MIO13
	JC1_P	AB7		JD1_P	V7		JE2	G7	MIO10
	JC2_N	AA4		JD2_N	V4		JE3	B4	MIO11
	JC2_P	Y4		JD2_P	V5		JE4	C5	MIO12
	JC3_N	T6		JD3_N	W5		JE7	G6	MIO0
	JC3_P	R6		JD3_P	W6		JE8	C4	MIO9
	JC4_N	U4		JD4_N	U5		JE9	B6	MIO14
	JC4_P	T4		JD4_P	U6		JE10	E6	MIO15

Apêndice C

Código VHDL dos IP Cores utilizados

C.1 Código VHDL da *Digilent* para os Pmod AD1

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

entity pmodad1_test is
    port(
        clk      : in std_logic;    -- 100MHz clock
        ADC_CS   : out std_logic;   -- ADC chip select
        ADC_SCLK : out std_logic;   -- ADC serial clock
        ADC_D0   : in std_logic;    -- ADC Channel 0
        ADC_D1   : in std_logic;    -- ADC Channel 1
        led      : out std_logic_vector(15 downto 0);
        D0       : out std_logic_vector(11 downto 0);
        D1       : out std_logic_vector(11 downto 0)
    );
end pmodad1_test;

architecture Behavioral of pmodad1_test is
    signal data_0 : std_logic_vector(11 downto 0) := (others=>'0');
    signal data_1 : std_logic_vector(11 downto 0) := (others=>'0');

    -----
    -- You can control the sampling frequency with the length of
    -- sequencer_shift_reg and ce_sr.
    --
    -- F(sclk) =F(clk)/(2*(ce_sr'length+1))
end Behavioral;
```

Código VHDL dos IP Cores utilizados

```
--
-- Sampling frequency is F(sclk)/ (sequncer_shift_reg'length+1)
--
-- with 100MHz and ce_sr being four bits long SCLK is 10MHz.
-- with sequncer_shift_reg of 19 bits, that gives a sample rate
  of 0.5MHz
-----
signal ce_sr          : std_logic_vector(3 downto 0) := (
  others=>'X');
signal sequncer_shift_reg : std_logic_vector(19 downto 0) := (
  others=>'X');

signal clock_state      : std_logic := 'X';
signal clock_enable     : std_logic := 'X';
signal din0_shift_reg   : std_logic_vector(15 downto 0) := (
  others=>'X');
signal din1_shift_reg   : std_logic_vector(15 downto 0) := (
  others=>'X');
begin
  led <= std_logic_vector(data_1(11 downto 4)) & std_logic_vector(
    data_0(11 downto 4));
  D0 <= data_0(11 downto 0);
  D1 <= data_1(11 downto 0);
  -----
  -- Generate the clock_enable signal
  -- For the rest of the design.
  --
  -- Change the length of ce_sr to
  -- change the Serial clock speed
  -----
  clock_divide : process(CLK)
    begin
      if rising_edge(CLK) then
        -----
        -- Self-recovering in case of a glitch
        -----
        if unsigned(ce_sr) = 0 then
          ce_sr <= ce_sr(ce_sr'high-1 downto 0) & '1';
          clock_enable <= '1';
        else
          ce_sr <= ce_sr(ce_sr'high-1 downto 0) & '0';
          clock_enable <= '0';
        end if;
      end if;
    end process;
  end begin;
```

```

        end if;
    end if;
end process clock_divide;

main : process (CLK)
begin
    if rising_edge(CLK) then
        if clock_enable = '1' then
            if clock_state = '0' then
                -- Things to do on the rising edge of the clock.

                -----
                -Capture the bits coming in from the ADC
                -----
                if sequencer_shift_reg(16) = '1' then
                    data_0 <= din0_shift_reg(11 downto 0);
                    data_1 <= din1_shift_reg(11 downto 0);
                end if;
                din0_shift_reg <= din0_shift_reg(din0_shift_reg'
                    high-1 downto 0) & adc_d0;
                din1_shift_reg <= din1_shift_reg(din1_shift_reg'
                    high-1 downto 0) & adc_d1;

                -----
                -- And update the sequencing shift register
                -- Self-recovering in case of a glitch
                -----
                if unsigned(sequencer_shift_reg) = 0 then
                    sequencer_shift_reg <= sequencer_shift_reg(
                        sequencer_shift_reg'high-1 downto 0) & '1';
                else
                    sequencer_shift_reg <= sequencer_shift_reg(
                        sequencer_shift_reg'high-1 downto 0) & '0';
                end if;

                -----
                -- Output rising clock edge
                -----
                adc_sclk    <= '1';
                clock_state <= '1';
            else
                -----

```

```
        -- Output falling clock edge
        -----
        adc_sclk    <= '0';
        clock_state <= '0';
    end if;
end if;

-----
-- A special kludge to get CS to rise and fall while
SCLK
-- is high on the ADC. This ensures setup and hold
times are met.
-----
if ce_sr(ce_sr'length/2) = '1' and clock_state = '1'
then
    if sequencer_shift_reg(0) = '1' then
        adc_cs <= '1';
    elsif sequencer_shift_reg(1) = '1' then
        adc_cs <= '0';
    end if;
end if;
end if;
end process main;
end Behavioral;
```

C.2 Código VHDL para Flip Flop

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

entity FF_D is
    Port ( CLK : in STD_LOGIC;
          D   : in STD_LOGIC;
          Q   : out STD_LOGIC);
end FF_D;

architecture Behavioral of FF_D is

begin
```

```

process(Clk)
begin
    if(falling_edge(Clk)) then
        Q <= D;
    end if;
end process;
end Behavioral;

```

C.3 Código VHDL para múltiplos Pmod AD1

```

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

entity AD1_PMOD is
    port(
        clk      : in  std_logic;    -- sinal clock
        ADC_CS   : out std_logic;    -- Sinal de aquisicao de dados
        ADC_SCLK : out std_logic;    -- Sinal para envio de dados
        MAT_CS   : out std_logic;    -- Corfimacao do envio de
            dados
        ADC_D0   : in  std_logic;    -- ADC Canal 0
        ADC_D1   : in  std_logic;    -- ADC Canal 1
        --(Adicionar o n mero de adc pretendidos)
            ADC_D2   : in  std_logic;    -- ADC Canal 2
        ADC_D3   : in  std_logic;    -- ADC Canal 3
            ADC_D4   : in  std_logic;    -- ADC Canal 4
        ADC_D5   : in  std_logic;    -- ADC Canal 5
        led      : out std_logic_vector(15 downto 0);
        --(Adicionar o n mero de adc pretendidos)
        D0       : out std_logic_vector(11 downto 0);
        D1       : out std_logic_vector(11 downto 0);
            D2       : out std_logic_vector(11 downto 0);
        D3       : out std_logic_vector(11 downto 0);
            D4       : out std_logic_vector(11 downto 0);
        D5       : out std_logic_vector(11 downto 0);
    );
end AD1_PMOD;

```

Código VHDL dos IP Cores utilizados

```
architecture Behavioral of AD1_PMOD is
    signal data_0 : std_logic_vector(11 downto 0) := (others=>'0');
        --(Adicionar o n mero de adc pretendidos)
    signal data_1 : std_logic_vector(11 downto 0) := (others=>'0');
        signal data_2 : std_logic_vector(11 downto 0) := (others
=>'0');
    signal data_3 : std_logic_vector(11 downto 0) := (others=>'0');
        signal data_4 : std_logic_vector(11 downto 0) := (others
=>'0');
    signal data_5 : std_logic_vector(11 downto 0) := (others=>'0');

    -----
    -- A frequencia de aquisicao e envio controlado por:
    -- sequncer_shift_reg and ce_sr.

    -- F(sclk) =F(clk)/(2*(ce_sr'length+1))

    -- Frequencia de amostragem F(sclk)/ (sequncer_shift_reg'
length+1)
    -- Com um sinal de CLK de 20MHz e ce_sr com 4 bits comprimento o
SCLK 2MHz.

    -- Com sequncer_shift_reg of 198 bits, a frequencia de aquisicao
de 10kHz
    -----

        --mudar em funcao das frequencias desejadas
    signal ce_sr : std_logic_vector(3 downto 0) := (
others=>'X');
    signal sequncer_shift_reg : std_logic_vector(198 downto 0) := (
others=>'X');
        --
        ---
    signal matlab_clk : std_logic := 'X';
    signal clock_state : std_logic := 'X';
    signal clock_enable : std_logic := 'X';
    signal din0_shift_reg : std_logic_vector(15 downto 0) := (
others=>'X');
    signal din1_shift_reg : std_logic_vector(15 downto 0) := (
others=>'X');
```

```

        signal din2_shift_reg      : std_logic_vector(15 downto 0) :=
            (others=>'X');
    signal din3_shift_reg          : std_logic_vector(15 downto 0) := (
        others=>'X');
        signal din4_shift_reg      : std_logic_vector(15 downto 0) :=
            (others=>'X');
    signal din5_shift_reg          : std_logic_vector(15 downto 0) := (
        others=>'X');

begin
    led <= std_logic_vector(data_1(11 downto 4)) & std_logic_vector(
        data_0(11 downto 4));
    D0 <= data_0(11 downto 0);
    D1 <= data_1(11 downto 0);
        D2 <= data_2(11 downto 0);
    D3 <= data_3(11 downto 0);
        D4 <= data_4(11 downto 0);
    D5 <= data_5(11 downto 0);
    MAT_CS <= matlab_clk;

    -----
    -- Gerar sinais de CLK
    -----

    clock_divide : process(CLK)
        begin
            if rising_edge(CLK) then
                -----
                -- Recuperacao em caso de falha
                -----

                if unsigned(ce_sr) = 0 then
                    ce_sr <= ce_sr(ce_sr'high-1 downto 0) & '1';
                    clock_enable <= '1';
                else
                    ce_sr <= ce_sr(ce_sr'high-1 downto 0) & '0';
                    clock_enable <= '0';
                end if;
            end if;
        end process clock_divide;

    main : process (CLK)
        begin
            if rising_edge(CLK) then
                if clock_enable = '1' then

```

```
if clock_state = '0' then
  --
  -----
  -- Adquirir os bits de informacao dos ADCs
  --
  -----

if sequncer_shift_reg(16) = '1' then
  data_0 <= din0_shift_reg(11 downto 0);
  --(Adicionar o n mero de adc pretendidos)
  data_1 <= din1_shift_reg(11 downto 0);
  data_2 <=
    din2_shift_reg(11
    downto 0);
  data_3 <= din3_shift_reg(11 downto 0);
  data_4 <=
    din4_shift_reg(11
    downto 0);
  data_5 <= din5_shift_reg(11 downto 0);
    elsif sequncer_shift_reg(18)
      = '1' then
      matlab_clk <= '1';
    elsif sequncer_shift_reg(50)
      = '1' then

  matlab_clk <= '0';
end if;
din0_shift_reg <= din0_shift_reg(din0_shift_reg'
  high-1 downto 0) & adc_d0;
--(Adicionar o n mero de adc pretendidos)

din1_shift_reg <= din1_shift_reg(din1_shift_reg'
  high-1 downto 0) & adc_d1;
  din2_shift_reg <=
    din2_shift_reg(
    din2_shift_reg'high-1
    downto 0) & adc_d2;
din3_shift_reg <= din3_shift_reg(din3_shift_reg'
  high-1 downto 0) & adc_d3;
  din4_shift_reg <=
    din4_shift_reg(
```


C.3 Código VHDL para múltiplos Pmod AD1

```

                                din4_shift_reg'high-1
                                downto 0) & adc_d4;
din5_shift_reg <= din5_shift_reg(din5_shift_reg'
    high-1 downto 0) & adc_d5;

--
-----

-- Atualizacao do vetor sequencing shift
-- register
-- Recuperacao em caso de falha
--
-----

if unsigned(sequncer_shift_reg) = 0 then
    sequncer_shift_reg <= sequncer_shift_reg(
        sequncer_shift_reg'high-1 downto 0) & '1';
else
    sequncer_shift_reg <= sequncer_shift_reg(
        sequncer_shift_reg'high-1 downto 0) & '0';
end if;

-----
-- ADC sclk trasicao ascendente
-----

adc_sclk    <= '1';
clock_state <= '1';
else
    -----
    -- ADC sclk trasicao descendente
    -----

    adc_sclk    <= '0';
    clock_state <= '0';
end if;
end if;

--
-----

-- Verificacao para garantir que o CS s  efetua a
-- transicao ascendente
-- quando o sinal SCLK est  a 1 no ADC.
```

```
--  
-----  
  
if ce_sr(ce_sr'length/2) = '1' and clock_state = '1'  
  then  
    if sequncer_shift_reg(0) = '1' then  
      adc_cs <= '1';  
    elsif sequncer_shift_reg(1) = '1' then  
      adc_cs <= '0';  
    end if;  
  end if;  
end if;  
end process main;  
end Behavioral;
```

Apêndice D

Conversão Blocos Retificador Ativo

Neste ponto são apresentadas as figuras dos blocos convertidos, de forma a implementar o algoritmo da UPS na ZedBoard. Esta conversão é necessária devido ao algoritmo já desenvolvido, não ser compatível com o *HDL Coder* que permite configurar a FPGA da ZedBoard.

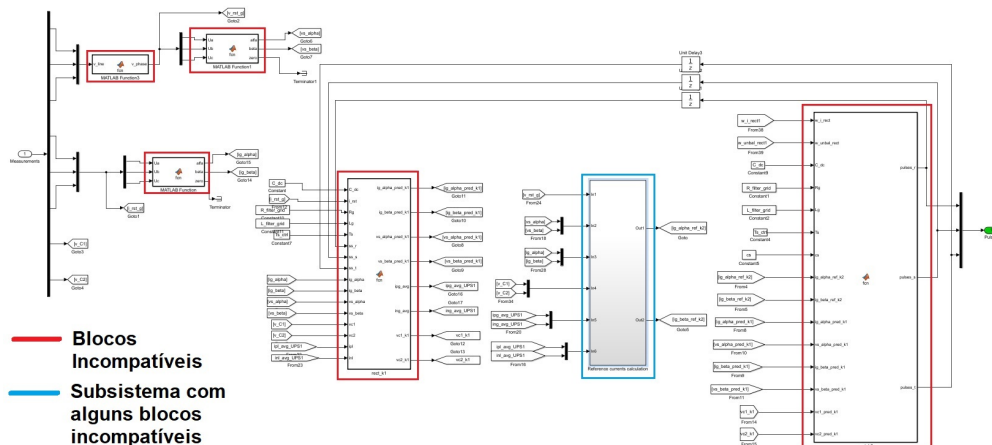


Figura D.1 Blocos do algoritmo incompatíveis com o *HDL Coder*.

O subsistema representado a azul na figura D.1, está representado na figura D.2, bem como os blocos no seu interior que precisam de ser alterados.

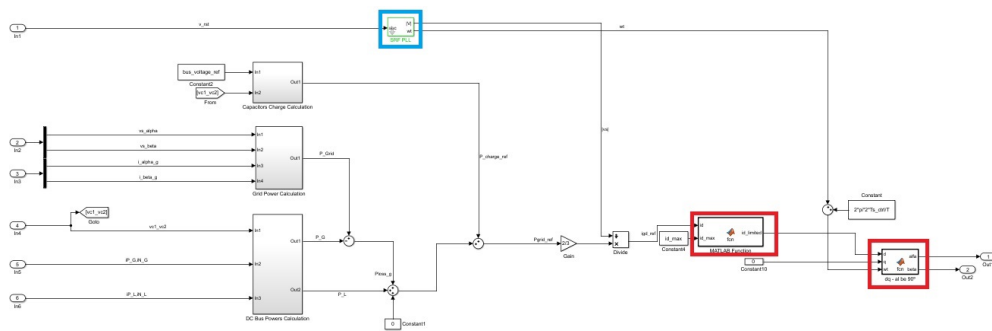


Figura D.2 Interior de um dos blocos incompatíveis com o *HDL Coder*.

Conversão Blocos Retificador Ativo

Assim como no caso anterior, a figura D.3, representa o interior do bloco com função de PLL representado a azul na figura anterior.

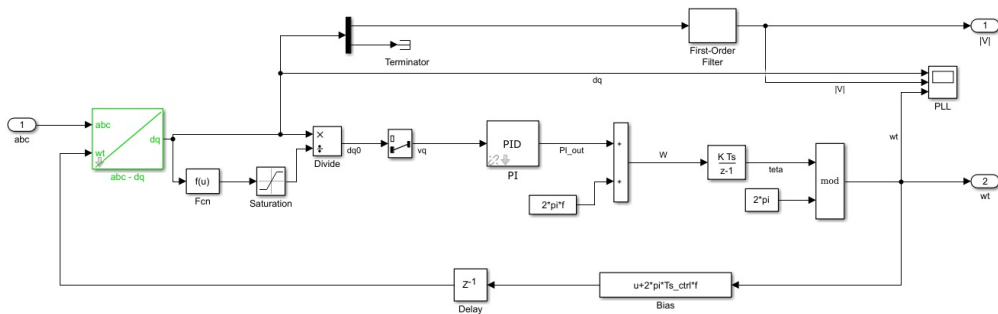


Figura D.3 Bloco responsável pela PLL existente no algoritmo.

Após conversão dos blocos identificados anteriormente obteve-se os representados nas figuras seguintes:

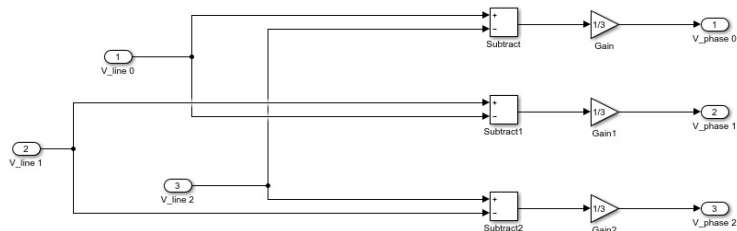


Figura D.4 Bloco que converte as tensões complexas nas tensões simples.

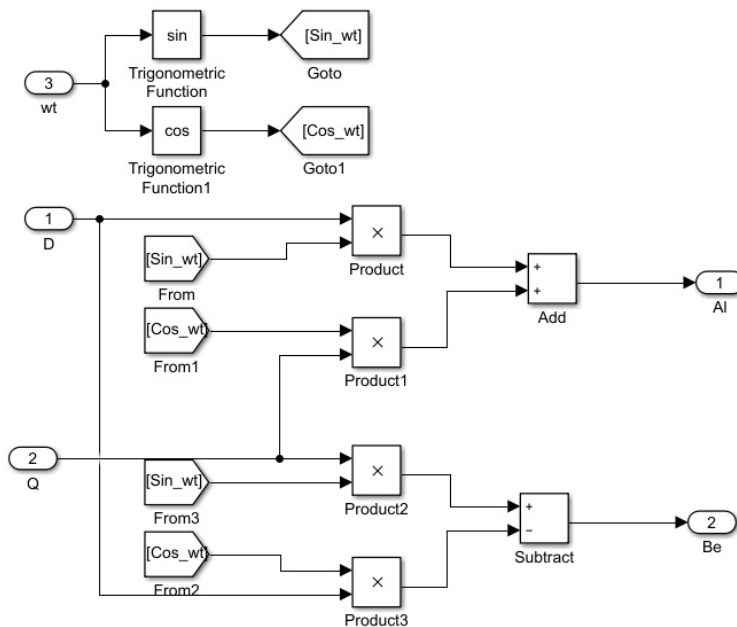


Figura D.5 Bloco da transformada DQ para $\alpha\beta$ após conversão.

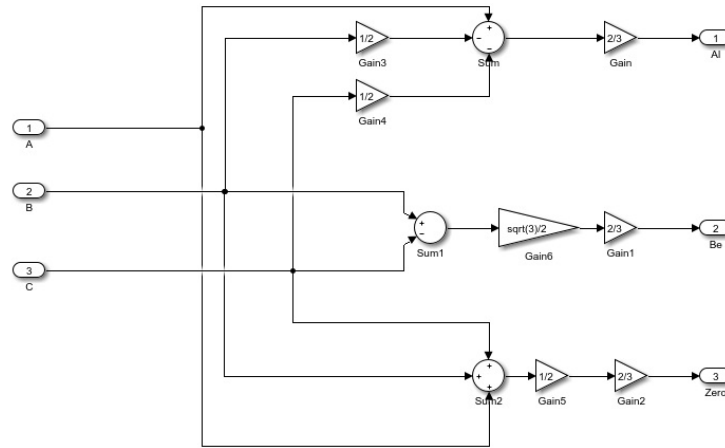


Figura D.6 Bloco da transformada ABC para $\alpha\beta$ após conversão.

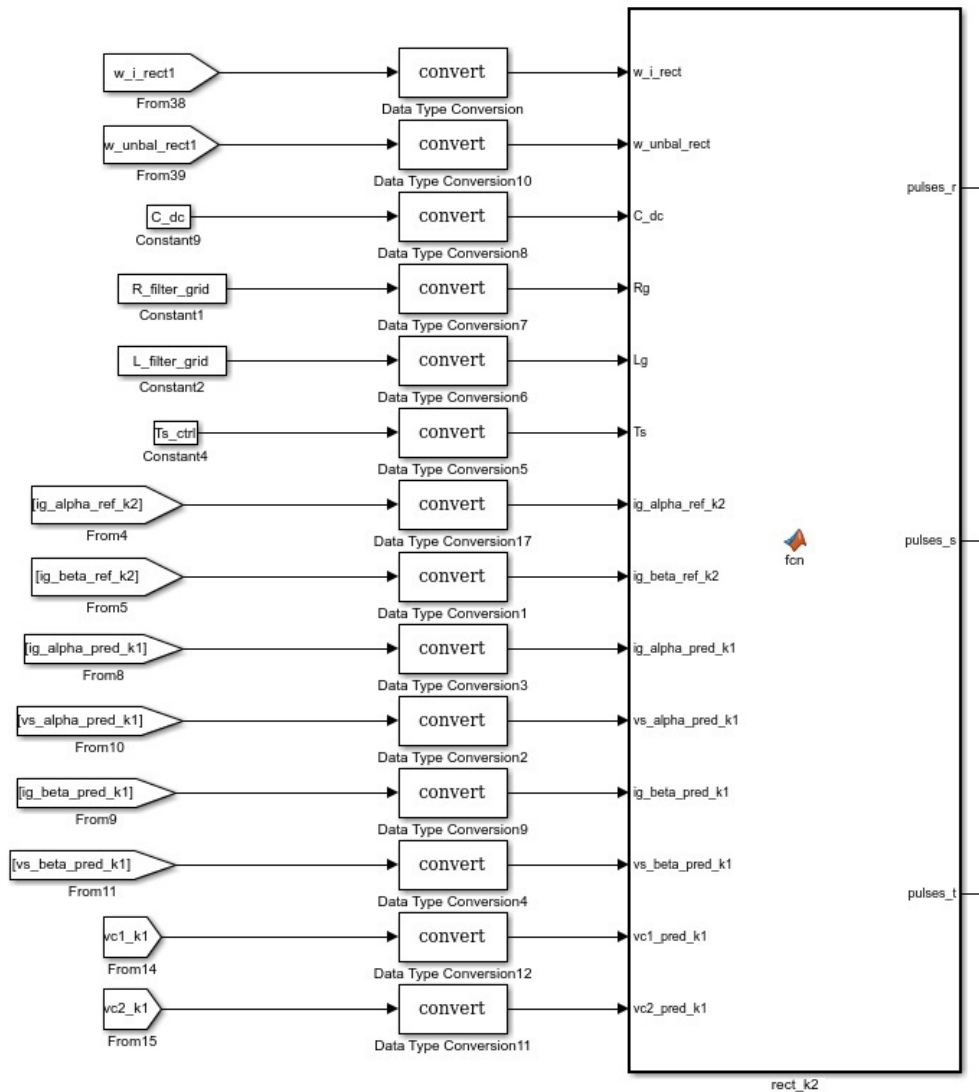


Figura D.7 Bloco "rect_k2" o único bloco onde foi aplicado a conversão de variáveis para ponto fixo.

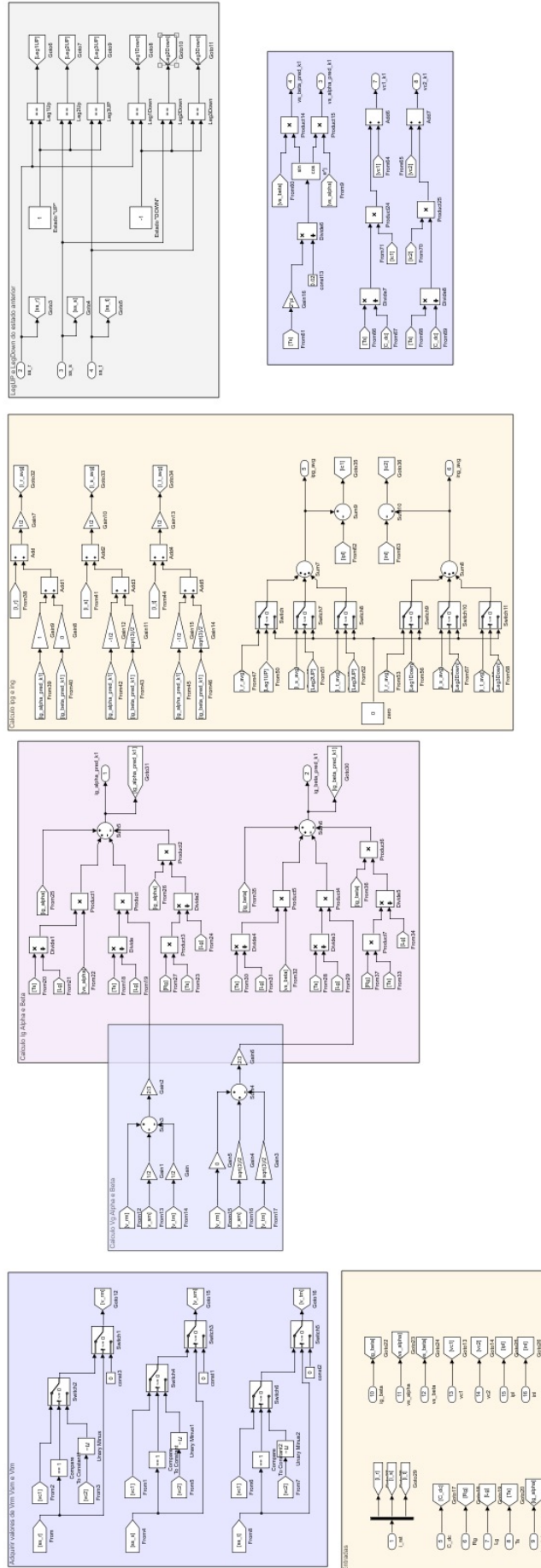


Figura D.8 Bloco "rect_k1" apscverso.

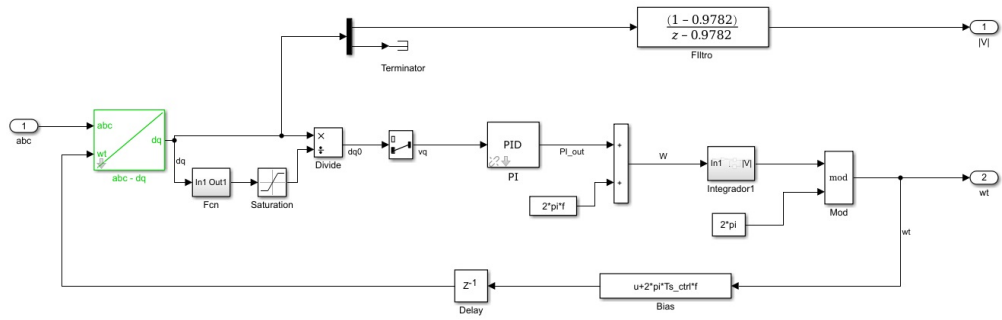


Figura D.9 Bloco responsável pela PLL após conversão dos blocos incompatíveis.

Apêndice E

ZedBoard e Plataformas Utilizadas no Laboratório

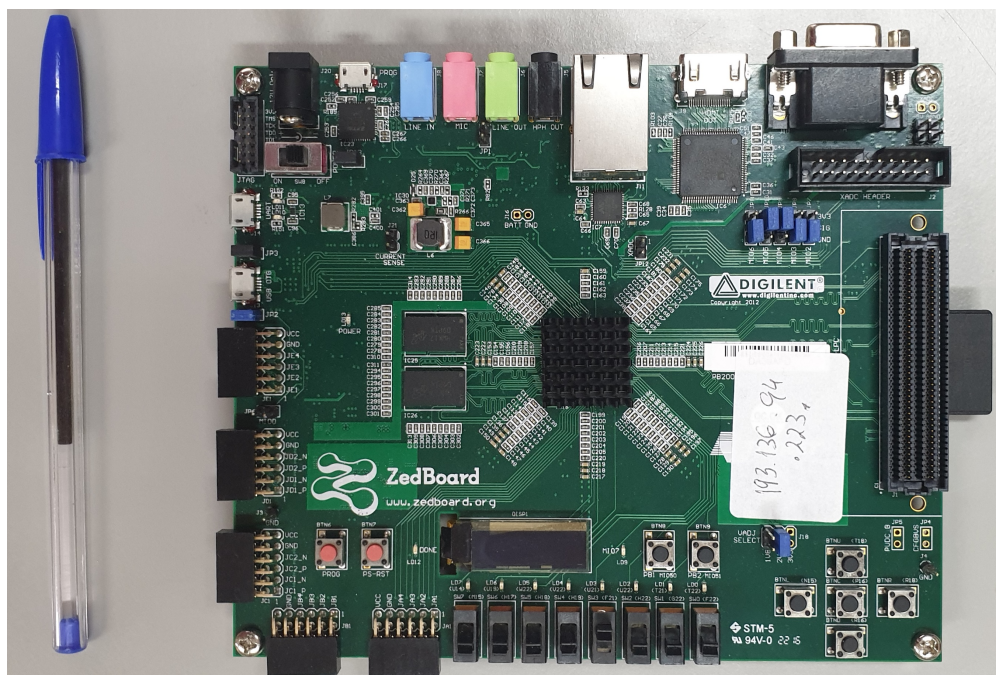


Figura E.1 Imagem da ZedBoard, e o seu tamanho comparado a uma caneta.

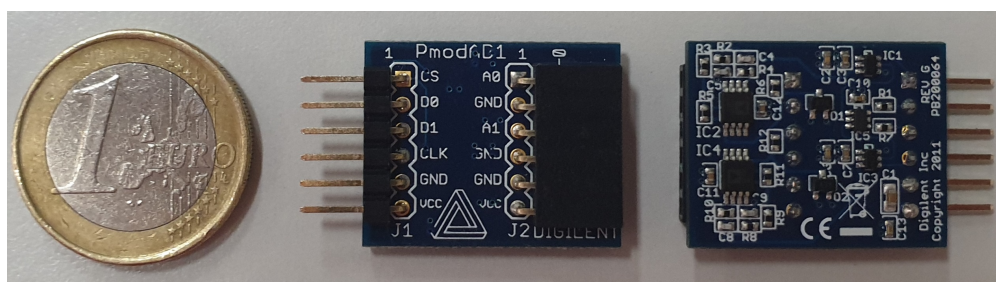


Figura E.2 Módulos Pmod AD1 utilizados no projeto.



Figura E.3 Comparação de tamanhos entre ZedBoard, *SB Rio* e *MicroLabBox*.