FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Sensor and Actuator integration in a FPGA-based hardware for Industry4.0 Digital Twin

Nuno Guterres Nogueira

PARA APRECIAÇÃO POR JÚRI

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Prof. Gil Manuel Magalhães de Andrade Gonçalves

Co-orientador: : Prof. João Pedro Correia dos Reis

12 de março de 2021

Resumo

A presente dissertação visa permitir o uso do máximo número de sensores configurados através da tecnologia FPGA para, posteriormente, ser possível controlar com *function blocks* pela plataforma Dinasore. Pretende-se assim, que exista uma abstracção do *hardware* e do *software* necessário para a implementação do sistema, facilitando e possibilitando o uso de *function blocks* para adicionar as respectivas funcionalidades. Esta implemnetação permite uma aquisição de sinal a frequências altas, o que adicionalmente à aceleração do FPGA pode potenciar uma próximo geração de PLCs industriais.

O sistema é desenhado inicialmente em *hardware* e utiliza ferramentas que facilitam a criação e modificação das características do mesmo. Numa segunda fase, são utilizadas ferramentas e bibliotecas providenciadas pelo fabricante, que permitem aceder às características do *hardware* através do *software*. Por fim, o processo de configuração do FPGA transita para o terminal, utilizando a linguagem de programação Python para a criação de *scripts* e ficheiros necessários à ferramenta Dinasore. Esta ferramenta permite a criação de sistemas ciber físicos de produção, que em paralelo com a tecnologia FPGA consegue garantir resultados em tempo real.

Abstract

The following dissertation aims the configuration of the maximum number of sensors through FPGA technology, and to, afterwards, be possible to control the system using Dinasore platform function blocks. Therefore, this implementation seeks to create an abstraction from hardware and software, allowing an easier use of function blocks to add the expected features to the system. This allows the existence of an high frequency signals acquisition, which according to the FPGA features, could create a new generation of industry PLCs.

The system is designed initially in hardware and applies tools that facilitate the creation and modification of the system' features. In a second phase, there are tools and libraries used which are given by the manufacturer and allows to access the hardware features through software. Lastly, the configuration process of the FPGA moves to the terminal, using Python as a programming language and to create mandatory files to the Dinasore platform. This platform, allows the creation of production cyber physical systems, which in parallel with the FPGA technology, provides reliable results in real-time.

Agradecimentos

Gostaria de deixar o meu agradecimento a todo o laboratório SYSTEC, o qual me recebeu da melhor maneira possível. Em especial, gostaria de agradecer ao co-orientador João Reis por me ter sempre ajudado ao longo da dissertação e ter contribuído com um apoio incomparável.

Do lado pessoal, quero deixar o meu agradecimento à minha família, principalmente aos meus pais Jorge e Elisabete, à minha irmã Cláudia e à minha namorada Bia. Sem eles, certamente que teria sido mais difícil.

Nuno Nogueira

Conteúdo

Lista de Figuras

Abreviaturas e Símbolos

ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuits
ASSP	Application Specific Standard Products
BSP	Board Support Package
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSP	Cyber-Physical System
DINASORE	Dynamic INtelligent Architecture for Software and MOdular REconfiguration
DMA	Direct Memory Access
DSP	Digital Signal Processors
EDS	Embedded Design Suite
FIFO	First-In-First-Out
FPGA	Field-programmable gate array
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Intellectual Property
JTAG	Joint Test Action Group
LDR	Light Dependent Resistor
LED	Light-Emitting Diode
MAV	Micro Air Vehicles
MM	Memory Mapped
NRE	Non-Recurring Engineering
PIO	Parallel I/O
PLC	Programmable Logic Controller
PLL	Phase Lock Loop
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SBT	Software Build Tools
SDI	Slave Data In
SDO	Slave Data Out
SDT	Software Development Tools
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
WSL	Windows Subsystem for Linux
XML	Extensible Markup Language

Capítulo 1

Introdução

1.1 Enquadramento

Esta dissertação intitulada Sensor and Actuator integration in a FPGA-based hardware for Industry 4.0 Digital Twin foi proposta pelo Centro de Investigação em Sistemas e Tecnologia (SYSTEC) alojado na FEUP. O projecto a ser desenvolvido nesta dissertação faz parte de um projecto maior já existente designado de Dynamic INtelligent Architecture for Software and MOdular REconfiguration (DINASORE). DINASORE [7] é uma plataforma distribuída que funciona numa camada intermédia de computação, permitindo o processamento de algoritmos de dados pré-programados em Function Blocks. Com esta plataforma é possível ter acesso a diferentes equipamentos dentro do mesmo sistema ciberfísico (CPS), com o intuito de os controlar e de os programar.

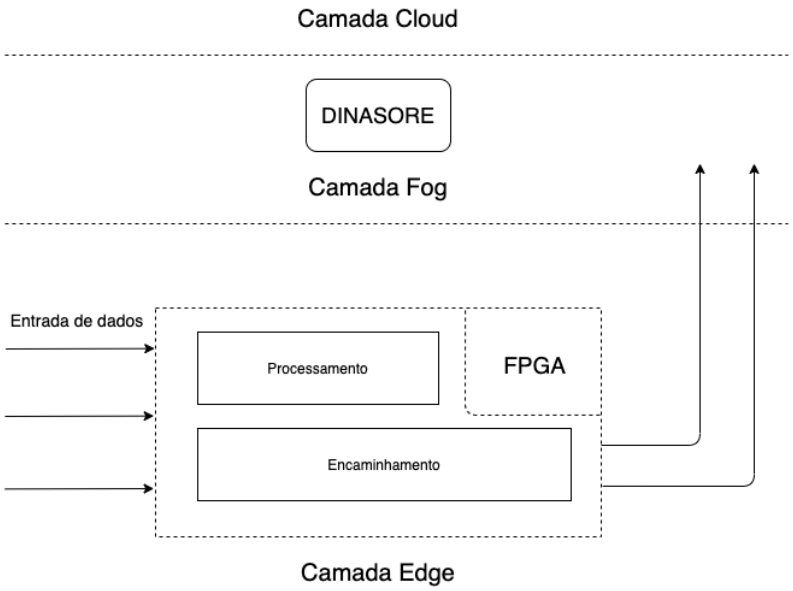

Figura 1.1: Estrutura por camadas na Indústria 4.0 e implementação do FPGA na mesma.

Dentro do sistema ciberfísico, sensores e atuadores são conectados a um FPGA, Terasic Cyclone V, que irá processar os dados provenientes dos sensores e em seguida encaminhar esses dados para uma camada a cima, onde estará a plataforma DINASORE a controlar o processo de processamento e encaminhamento, que pode ser alterado via interacção humana. Esta junção (figura 1.1) entre FPGAs e sensores/atuadores visa diminuir os tempos de processamento e de tomada de decisões.

O desenvolvimento do projecto é feito tendo em conta o conceito de Indústria 4.0 e o conceito de Internet Of Things, que visa reduzir os custos de equipamentos, manutenção e de mão de obra humana em paralelo com a mudança da produção e gestão para tempo-real, proporcionando uma tomada de decisões mais ampla.

Nos dias de hoje, com o aumento da popularidade dos FPGA, redução do custo, aumento de capacidade do mesmo e com o défice de investigação em FPGAs utilizados no contexto industrial, este trabalho é uma mais valia, não só para o desenvolvimento do programa DINASORE, mas também para o aumento de informação sobre FPGAs aplicados na aquisição de dados e respectivos resultados.

1.2 Objectivos

O objectivo desta dissertação é a implementação do máximo número de sensores e atuadores cumprindo os requisitos temporais críticos estabelecidos pela produção industrial na utilização de FPGA. O sistema deve ser capaz de processar e encaminhar dados, mantendo os requisitos de produção, de forma mais eficaz, mais eficiente e com menor custos do que a solução anterior.

Adicionalmente, o sistema deve cumprir o *standard* IEC 61499 através da plataforma Dinasore. Assim, o foco está assente desde o desenho de uma *pipeline* em 4DIAC até à orquestração da FPGA para fazer aquisição de sinal e redirecionamento da informação recolhida para o utilizador final.

No final da implementação, o sistema deve cumprir os requisitos em cima mencionados e garantir a flexibilidade de adaptação à plataforma DINASORE.

Para tal, o objectivo é utilizar o FPGA como controlador do ADC, que permitirá acelerar a conversão de sinais analógicos provenientes de sensores, em sinais digitais, através de características intrínsecas existentes na tecnologia FPGA.

1.3 Motivação

Esta dissertação assenta no tema de FPGAs utilizados na indústria e na óptica de uma grande aquisição de dados, um tema pouco popular na implementação de FPGAs até à data. Até 2015 [8], cerca de menos de 5% das publicações no IEEE tinham como foco a aquisição de dados e cerca de menos de 15% a engenharia de controlo, provando que esta é uma área onde existe um défice de trabalho de investigação e que tem espaço para poder ser bastante desenvolvida. Este trabalho

procura colmatar esse espaço, trazendo soluções que poderão ser adoptadas posteriormente nesta área e ainda contribuir para o seu desenvolvimento futuro.

1.4 Caracterização do problema

1.4.1 Definição do problema

No contexto industrial, a utilização de equipamentos obsoletos pode danificar uma produção, seja em termos qualitativos ou em termos quantitativos. Em termos qualitativos pode afectar a qualidade de produção por utilização de equipamentos que, em comparação ao mercado, não justifiquem o seu uso. Em termos quantitativos pode afectar o tempo de produção, produzindo em maior escala ou em menor escala, dependendo da qualidade de produção anteriormente estipulada. Sendo assim, é extremamente necessário que a indústria se vá actualizando em paralelo à evolução do mercado, para que consiga produzir quantidade com qualidade, competindo com a concorrência.

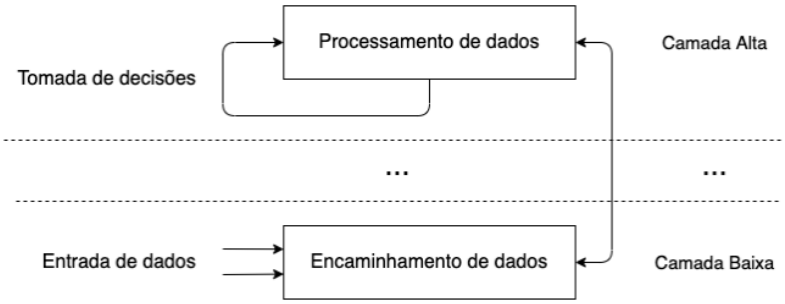

Figura 1.2: Esquema do problema encontrado

Com a constante evolução do mercado e a indústria a ser obrigada a acompanhar, o desperdício de tempo e custo monetário na pesquisa e implementação de vários equipamentos diferentes - às vezes de diferentes produtores - torna-se um obstáculo à inovação levando à estagnação tecnológica da indústria. O facto dos equipamentos utilizados não poderem ser reajustáveis depois da sua implementação, faz com que ao longo da evolução da tecnologia, se tornem obsoletos e não tragam valor de aplicação na indústria, obrigando à pesquisa, planificação, compra e aplicação de outros equipamentos que consigam responder às necessidades da industria actual, possível observar pela figura 1.2. Este obstáculo leva, também, a uma gestão precária dos equipamentos utilizados, trazendo uma grande latência na comunicação entre equipamentos e, posteriormente, a um atraso no processamento dos dados.

1.5 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais cinco capítulos. O segundo capítulo é composto pelo estado da arte, onde é efetuado uma revisão bibliográfica sobre a criação e o uso do FPGA, no mundo e na industria.

O terceiro capítulo visa explicar os fundamentos por de trás do uso da tecnologia do FPGA aplicado ao seu *workflow*. No mesmo capítulo é tambem referido fundamentos sobre o *software* aplicado ao *hardware*, utilizado ao longo da dissertação.

O quarto capítulo diz respeito à implementação dos fundamentos até então adquiridos e à criação do *design* nas ferramentas referidas em anteriores capítulos. A implementação passa pelo software Quartus Prime, pelo uso do IDE Eclipse com ferramentas de *software* especificas para o desenvolvimento do processador Nios II presente no FPGA, passa pelo uso do terminal de comandos para o desenvolvimento em ambiente de FPGA e a sua respectiva configuração, e , ainda passa pela implementação do sistema final na plataforma DINASORE em interface de *funtion blocks*.

No capítulo cinco, os resultados provenientes da implementação são abordados e representados com o uso de outras tecnologias.

Por fim, o capítulo seis, é responsável pela comparação dos resultados e exposição de um possível trabalho futuro.

Adicionalmente, no anexo A é exposto todo o código utilizado ao longo da dissertação e no anexo B é exposto alguns do problemas com o *software* utilizado ao longo do processo.

Capítulo 2

Revisão Bibliográfica

2.1 FPGA em expansão

Apesar da lógica de blocos não ser um conceito recente, os primeiros FPGAs só surgiram depois da fundação da Xilinx Inc. em 1984 [9]. A empresa americana lançou os primeiros equipamentos de lógica programável pelo usuário (user-programmable logic devices): os Xilinx XC 2000. Nessa altura, antes do aparecimento dos FPGAs, eram necessários vários circuitos integrados (por vezes centenas) numa placa de circuito para se chegar à mesma funcionalidade de hardware que um FPGA nos dias de hoje proporciona.

O uso de FPGA trouxe não só a flexibilidade de poder ser programado mesmo depois de produzido, mas também a possibilidade de não restringir o sistema ao uso de *application-specific integrated circuit* (ASIC) ou de *application-specific stantard products* (ASSP) como, por exemplo, microcontroladores [10]. Apesar das características vantajosas, na altura que surgiram os primeiros FPGAs, os ASICs tinham melhores especificações: mais rápidos, maior capacidade e com um menor custo.

Ainda assim, o FPGA tornou-se num competidor pelo mesmo mercado, retirando do mercado o custo pelo *non-recurring engineering* (NRE), ou seja, o custo que engloba a pesquisa, o projecto, o desenvolvimento e o teste de um novo produto.

Para o mesmo performance do ASIC, o FPGA saía mais caro por unidade, mas não incluiria o custo pelo NRE, o que faria que para poucas unidades compensasse frente ao ASIC, que cobra por NRE, como é possível verificar na figura 2.1.

Adicional à ausência de NRE, o uso de FPGA em relação ao ASIC elimina outro tipo de problemas como: *transistor-level design*, teste, integridade de sinais, *crosstalk*, design de I/O e distribuição de relógio [11].

Ao longos dos anos, e, adjacente à lei de Moore [12], o tamanho dos circuitos foi ficando cada vez mais pequeno permitindo ao FPGAs aumentar a sua capacidade. O facto de as empresas de FPGA optarem pelo modelo de negócio *fabless* - design e vendas feitas pela própria, mas recorrendo a terceiros para a fabricação do hardware - também influenciou o crescimento da produção de FPGAs. Mas, apesar do crescimento da produção e diminuição do tamanho dos transístores,

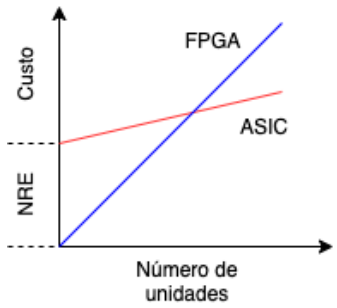

Figura 2.1: Relação entre custo e o número de unidades entre FPGA e ASIC.

o aumento de capacidade não era uma métrica suficiente para cativar as empresas à utilização de FPGAs. No início dos anos 2000, os FPGAs já eram comuns em sistemas digitais mas mantinham, ainda que menores, as desvantagens frente aos ASICs. Considerando um processo de fabrico de 90nm, o FPGA seria 3 a 4 vezes mais lento e consumia 10 vezes mais energia que um ASIC equivalente [13].

Face ao estado do mercado existiam melhores alternativas, pelo que os vendedores de FPGA tiveram que apostar numa produção *low-cost, low-performance and low-capacity* e na produção de bibliotecas de Intellectual Property (IP) para sistemas *high-end*. A mudança para um fabrico de FPGAs *low-cost* mudou a estratégia antiga assente sobre a capacidade e a sua quantidade de *gates*, passando a ter como foco principal os blocos de lógica dedicados, incluindo multiplicadores, blocos de RAM, *clock management*, encriptação para proteger as IPs do sistema e muitos outros.

2.2 Mudança de mercado alvo

Com a evolução dos FPGAs e a sua mudança de estratégia na arquitectura, o seu mercado alvo também evoluiu. Se antes os FPGAs competiam com ASICs com o intuito de eventualmente os substituírem, na década de 2000 eram implementados em infraestruturas de comunicações para processar grandes quantidades de dados. Companhias como a Cisco Systems utilizavam FPGAs para gerir grandes volumes de Internet e de dados de voz. A mudança do mercado alvo foi tão significativa que cerca de mais de metade das vendas das empresas de FPGAs, eram feitas para o mercado de comunicações.

A mudança de estratégia da arquitectura do FPGA e a implementação na área das comunicações, veio verificar-se uma mais valia pela facilidade do processamento de funções matemáticas em ambiente FPGA, podendo ser implementados algoritmos como Finite Impulse Response (FIR) e Fast Fourier Transform (FFT) [8]. Não tão comuns como a área de comunicações, mas também outros tipos de mercados começaram a surgir. O processamento de imagem foi um desses mercados e beneficiou da características do paralelismo que o FPGA proporciona. A engenharia de controlo beneficiou do facto do FPGA poder ser adoptado como controlador de um *hard real-time system* e do facto de poder ser reconfigurado em *run-time* dependendo dos requisitos do sistema.

O mercado de redes beneficiou da análise de tráfego em tempo real sem reduzir o desempenho da rede. A criptografia beneficiou do paralelismo que o FPGA proporciona e é usado tanto para encriptação como para desencriptação e, em alguns casos, usado para quebrar a encriptação de dados. A matemática, a computação neuronal e aquisição de dados também são outros mercados que utilizam FPGA, existindo ainda outros mais.

Os FPGAs passaram assim de alternativa para a utilização de ASICs num sistema, para serem considerados mecanismos essenciais de encaminhamento de grandes quantidades de dados.

2.3 Nos dias de hoje

Até 2015, ao longo de 30 anos desde a existência dos primeiros FPGAs, a capacidade dos mesmos já tinha aumentado em 10000 vezes, a velocidade em 100 vezes, reduzindo o custo e energia por operação em mais de 1000 vezes [11]. Actualmente os FPGA posicionam-se no mercado como tendo duas vantagens principais: reconfiguração em qualquer estado do sistema e grande capacidade de encaminhamento/processamento de dados em paralelo.

2.3.1 Reconfiguração e Aceleração

Com estas duas vantagens principais no uso moderno do FPGA, o mesmo começou a ser utilizado em campos mais abrangentes, como por exemplo, no uso para tecnologia militar. Em 2007, a implementação de FPGA em Micro-Air-Vehicles (MAV) foi estudada [14], resultando na possibilidade do processamento em paralelo de cerca de 245 sensores ópticos num equipamento com um tamanho entre 1 a 50 centímetros.

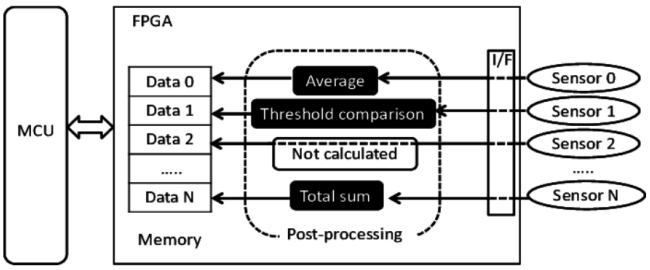

Figura 2.2: Processamento de dados de vários sensores à entrada do FPGA e envio dos dados processados para o microcontrolador. [1]

Outra vantagem que tem sido aproveitada no FPGA, é o facto de poder atenuar o trabalho pesado feito pelos microcontroladores, que acaba por ser uma vantagem consequente à capacidade do FPGA de processar dados em paralelo. Em 2014, desenvolveram um protótipo de um *wireless node* configurável [1], através da utilização de FPGA, microcontrolador e sensores/atuadores. Com este conceito (Figura 4) é possível processar os dados crus à entrada do FPGA e enviar apenas dados já processados para os microcontroladores, passando pela execução de funções matemáticas ou até *thresholds*. O microcontrolador é usado como mecanismo de tomada de decisão,

enquanto que o FPGA é usado como mecanismo de processamento e encaminhamento de dados - um acelerador. Outro exemplo da implementação conjunta de microcontroladores e FPGA [15], é quando o microcontrolador não consegue garantir rapidez necessária do próprio *timer* e tem de recorrer à aceleração do FGPA.

O facto de o FPGA garantir a reconfiguração do mesmo, tem vindo a criar popularidade no conceito de *Wireless Sensor Nodes*. A ideia passa por criar uma rede de nós capazes de comunicar entre si, mas que possam ser reconfigurados consoante os requisitos aplicados ao sistema. Para isso é utilizado o encaminhamento do FPGA para ler os valores dos sensores à entrada e depois é enviado a informação processada pelo FPGA para os nós requisitados pelo sistema.

2.3.2 Aquisição de dados

Existem vários tipos de sensores, sendo os sensores analógicos os mais comuns com cerca de 50% da quota do mercado [16].

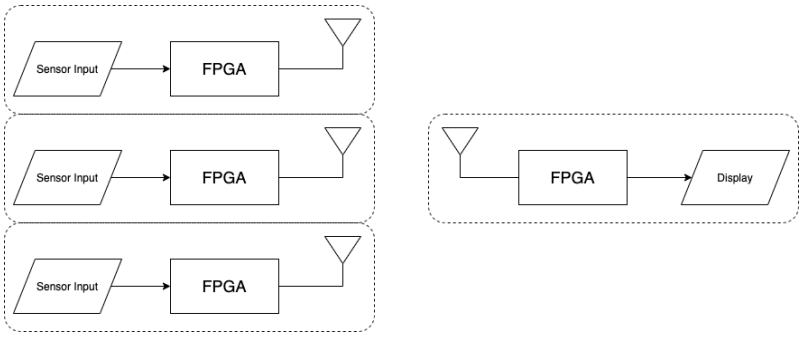

Figura 2.3: Exemplo de Wireless Sensor Nodes [1] [2]

Assim, em grande parte dos casos, a aquisição de dados é feita de maneira sequencial: sensor analógico, possível circuito com amplificador para aumentar a resolução e ligação final ao Analog-to-Digital (ADC) do microcontrolador.

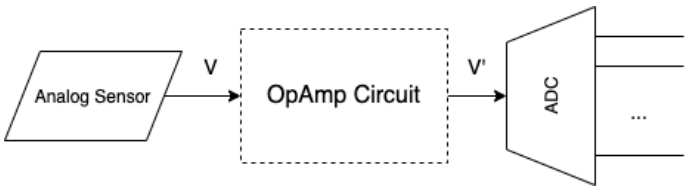

Figura 2.4: Esquema de medição sequencial de sensor analógico

Este tipo de aplicação é normalmente sugerido pelo fabricante no *datasheet* do sensor. Porém, a aquisição de dados pode dispensar da utilização de circuitos dispendiosos adicionais e de ADCs quando se trata de sensores com uma interface de ligação directa [17] [3], intitulados *smart sensors* [18]. Deste modo é possível criar uma interface com o mínimo de recursos, sem ADC e sem

circuitos eléctricos adicionais, e juntar às capacidades de aceleração do FPGA para criar uma aquisição de dados mais rápida e com menos gasto de recursos.

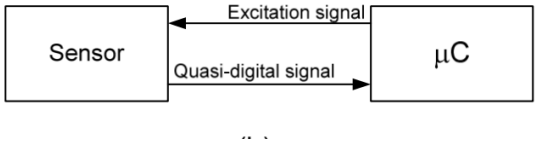

Figura 2.5: Exemplo de Interface directa [3].

2.4 FPGA na Indústria

Apesar de os FPGAs terem surgido com uma maior popularidade na área das telecomunicações, ao longo do tempo têm se dividido para outro tipo de áreas, criando uma aplicação da tecnologia mais abrangente. A área industrial é uma dessas aplicações. Num contexto industrial e tendo em conta equipamentos industriais, o seu planeamento em termos de controlo de hardware pode passar pela utilização de microcontroladores, Central Processing Unit (CPU), FPGA ou até ASICs [19].

Apesar da industria ser uma área dentro de muitas outras onde o FPGA está presente, acaba por ter características muito semelhantes ao do contexto geral de aplicação do FPGA. As aplicações baseadas em ASICs continuam a ser aquelas que fornecem o melhor desempenho, mas, em contrapartida, são também as que apresentam o maior custo, principalmente em contexto industrial, onde o número de máquinas de produção é relativamente baixo em comparação com outras áreas onde são utilizadas soluções baseadas em ASICs. Outra opção para a escolha de equipamento de controlo é a utilização de soluções baseadas em Digital Signal Processors (DSP), que incluí utilização de CPUs ou microcontroladores. Está solução é o contrário da utilização de soluções baseadas em ASICs, é uma tecnologia mais barata, mas que pode comprometer o desempenho do sistema por falta de velocidade e capacidade de processamento. Assim, o FPGA acaba por ser um meio termo entre soluções baseadas em ASICs e baseadas em DSPs, conseguindo transformar as desvantagens de cada uma delas em vantagens para o sistema.

Não só para controlo de equipamentos e para a aquisição de dados pode ser usado o FPGA, mas também para detecção de falhas em equipamentos industriais. Em 2010 [20], introduziram um novo conceito de detecção deste tipo de falhas, pela análise de vibração no domínio das frequências de um motor recorrendo à tecnologia do FPGA. Utilizando o FPGA e as suas capacidades de processamento em paralelo, foi possível a aquisição de dados contínua de 3 eixos de um acelerómetro, processamento de algoritmos matemáticos e tomada de decisão, produzindo um sistema de monitorização do motor em tempo-real.

Em relação ao uso do PLC em ambiente industrial, o FPGA compete diretamente com esse tipo de tecnologia mais popular na aplicação industrial. No entanto, a utilização de PLC garante limitações a nível de velocidade e flexibilidade, o que é responsável por um resultado negativo

em aplicações industriais de alta velocidade. [21]. Atualmente, pela procura de melhor relação custo/capacidades providenciadas, o PLC tem sido alvo de subtituição pelo FPGA [22], e, em caso de transição, o FPGA pode ser responsável por desenvolver o processor utilizado pelo PLC e não ocorrer uma substituição total da tecnologia anteriormente implementada. [23]

Capítulo 3

Hardware, Software e Workflow no desenvolvimento em FPGA

3.1 Placa C5G

Nesta dissertação é utilizada uma placa de desenvolvimento produzida pela Altera (posteriormente comprada pela Intel), a placa de desenvolvimento Cyclone V GX Starter Kit. Esta placa é alimentada a 12 V através de um adaptador incluído com a placa e incluí vários periféricos, tais como: 2x20 GPIO, Arduino Header, 7-Segment Display, Switches, Botões, LEDs, USB Blaster, UART para USB, entrada HDMI, MicroSD Card slot, LPDDR2, SRAM e ADC. Todos estes periféricos estão conectados com o FPGA, Cyclone V GX, sendo possível desenvolver controladores para aceder aos mesmos.

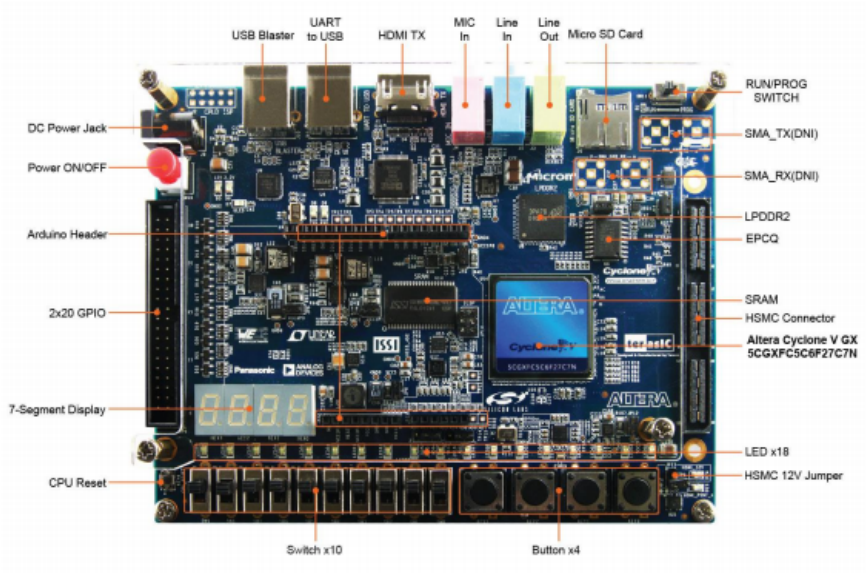

Figura 3.1: Vista de cima da placa de desenvolvimento Cyclone V GX Starter Kit

Para este tipo placa (figura 3.1), a Intel disponibiliza ao utilizador uma excelente documentação com manual de utilização, software de teste, *datasheets* dos periféricos e exemplos de sistemas a utilizar, tanto em hardware como em software.

3.1.1 Cyclone V

Cyclone V é uma família de FPGAs, existindo mais versões do que apenas a versão utilizada, a GX. A versão E, GX e GT, são versões pensadas para uma aplicação de baixo custo e de uso geral, enquanto que as versões SE, SX e ST, são pensadas para a utilização de um System on a Chip (SoC), visto que ambas têm um processador ARM Cortex-A9 junto do FPGA.

No entanto, ainda que, dentro da mesma família exista várias versões de Cyclone V, também em cada versão existe várias linhas de produção diferentes. Sendo assim, cada linha de produção acaba por ter especificações distintas.

Cyclone V GX FPGAs[1]				
5CGXC3	5CGXC4	5CGXC5	5CGXC7	5CGXC9
35.5	50	77	149.5	301
13,460	18,868	29,080	56,480	113,560
53,840	75,472	116,320	225,920	454,240
135	250	446	686	1,220
1,350	2,500	4,460	6,860	12,200
291	295	424	836	1,717
57	70	150	156	342
114	140	300	312	684
16	16	16	16	16
4	6	6	7	8

Figura 3.2: Linhas de produção de Cyclone V GX

Como é possível verificar na figura 3.2, existem cinco linhas de produção só para a a versão GX. A Cyclone V GX Starter Kit faz parte da linha de produção 5CGXC5 e conta com as características mais importantes a de 77 000 elementos lógicos, 4 460 000 bits de memória embarcada (tipo M10K), 424 000 bits de Memory logic array block (MLAB) e 6 PLLs. Somando os dois tipos de memória, no total, o FPGA possui 4884 Kilobits de memória embarcada (M10K + MLAB).

3.2 *Workflow*

O *workflow* em desenvolvimento de FPGA é bastante diferente do habitual em áreas de trabalho similares. Enquanto que o desenvolvimento em microcontroladores está mais relacionado com o software em sistemas embarcados, o desenvolvimento em FPGA está mais relacionado com o hardware em sistemas embarcados, daí a designação de hardware designer. O *workflow* assente em desenvolvimento de *software* para sistemas embarcados, é baseado, usualmente, numa linguagem escolhida para controlar o mesmo, desenvolvido através de qualquer Integrated Development

Environment (IDE), as ferramentas necessárias tem uma *footprint* baixa e a documentação existente é actualizada e acessível a qualquer *developer*. Por outro lado, o software utilizado para desenvolvimento em FPGA, é, usualmente, proprietário da placa de desenvolvimento utilizada e as ferramentas a utilizar têm uma *footprint* grande (por vezes a rondar os *gigabytes*). Em contexto de desenvolvimento de FPGA da Intel, o software a utilizar é o Quartus Prime. Com FPGAs da Xilinx é utilizado o software Vivado e com FPGA fabricados pela Lattice Semicondutor é utilizado softwares como o Lattice Diamond ou o Ice Cube 2, entre outros.

Com o facto da maior parte dos produtores de FPGA limitarem o uso dos FPGAs e as ferramentas ao seu *software* proprietário, faz com que o hardware designer sinta a necessidade de se adaptar ao software em questão e que aplique grande parte do seu tempo a estudar as ferramentas do software, algo não tão comum no desenvolvimento de software, fazendo com que o desenvolvimento em ambiente de FPGA seja extremamente dependente do software utilizado para o desenvolvimento do mesmo.

3.3 Quartus Prime Software

O software utilizado no desenvolvimento em ambiente de FPGA na placa de desenvolvimento previamente referida, é, sem surpresas, o Quartus Prime, visto que a placa é da Intel. O software Quartus Prime tem várias versões, para Windows ou Linux, e várias edições. No decorrer desta dissertação é utilizada a versão 20.1 (a mais recente à data de início da dissertação) e a edição Lite, em ambiente Windows. Existem, na versão 20.1, três edições. A edição Lite, utilizada na execução desta dissertação é a única em que não é necessário pagar uma licença à Intel. O facto de não necessitar de uma licença é uma vantagem, no entanto também existem desvantagens, como não poder usar todas as funcionalidades do processador Nios II, referido posteriormente.

A utilização do software Quartus Prime em Windows, trás algumas desvantagens na instalação em relação ao software em ambiente Linux. Depois instalação do Quartus Prime, é ainda necessário, que o utilizador instale a distribuição de Linux Ubuntu 18.04 LTS em Windows Subsystem for Linux (WSL) e o IDE Eclipse, obrigatório para o correto funcionamento das ferramentas e sistemas que utilizem o soft-processor Nios II.

O software Quartus Prime providencia inúmeras ferramentas para o desenvolvimento em FPGA e ferramentas necessárias para que seja possível ter o mesmo fim, partindo de meios diferentes. Sendo assim, ferramentas como o Platform Designer, Pin Planner e Programmer, são ferramentas que procuram facilitar o meio para chegar ao desenvolvimento em FPGA e, também ferramentas essenciais à execução desta dissertação.

3.3.1 Platform Designer

A ferramenta Platform Desginer providencia ao utilizador a opção de poder organizar os vários cores de IP a usar no sistema, como é possi´vel observar pela figura 3.3. No fundo, é uma interface gráfica onde se estabelece as ligações entre os vários cores com uma total abstracção de Hardware

Design Languages (HDL), como Verilog. É nesta ferramenta que o sistema deve mostrar os seus requisitos, sendo os mesmos definidos ao longo dos cores IP e dos seus parâmetros.

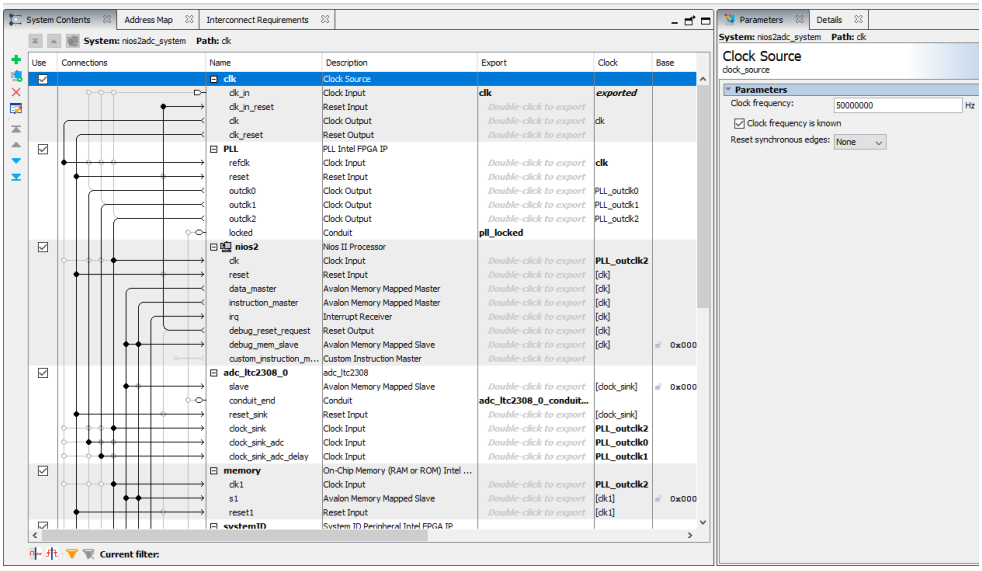

Figura 3.3: Exemplo da constituição de um sistema na ferramenta Platform Designer

Como previamente referido, o software Quartus Prime, permite o mesmo fim por vários meios, e, nesta ferramenta, não é excepção. A existência da ferramenta do Platform Designer não retira ao utilizador a possibilidade do mesmo ser feito sem ela, no entanto facilita a criação de todo um sistema de hardware em FPGA, partindo de cores providenciados pela Intel. Assim, a maior vantagem desta ferramenta, é a tamanha facilidade que disponibiliza ao utilizador para criar um sistema, seja ele em pequena ou em grande escala.

3.3.2 Pin Planner

A ferramenta Pin Planner, como o próprio nome indica, é responsável por atribuir às variáveis criadas uma ligação física ao FPGA. Geralmente, na criação de cada projecto, ao seleccionar a placa ou FPGA a utilizar, o Quartus Prime cria e atribui automaticamente um conjunto de variáveis ao hardware correspondente do FPGA.

Novamente, este processo pode ser feito manualmente, apesar da ferramenta o disponibilizar de forma a facilitar o desenvolvimento em FPGA.

3.3.3 Programmer

A ferramenta Programmer é responsável por permitir programar o FPGA. É uma interface gráfica que permite ao utilizador escolher qual a ligação física (por exemplo USB) a utilizar e qual o modo de programação, representado na figura 3.4.

O FPGA Cyclone V GX permite ser programado por JTAG ou por Active Serial. A vantagem da programação via JTAG é a facilidade de uso e a desvantagem é que o FPGA deixa de estar

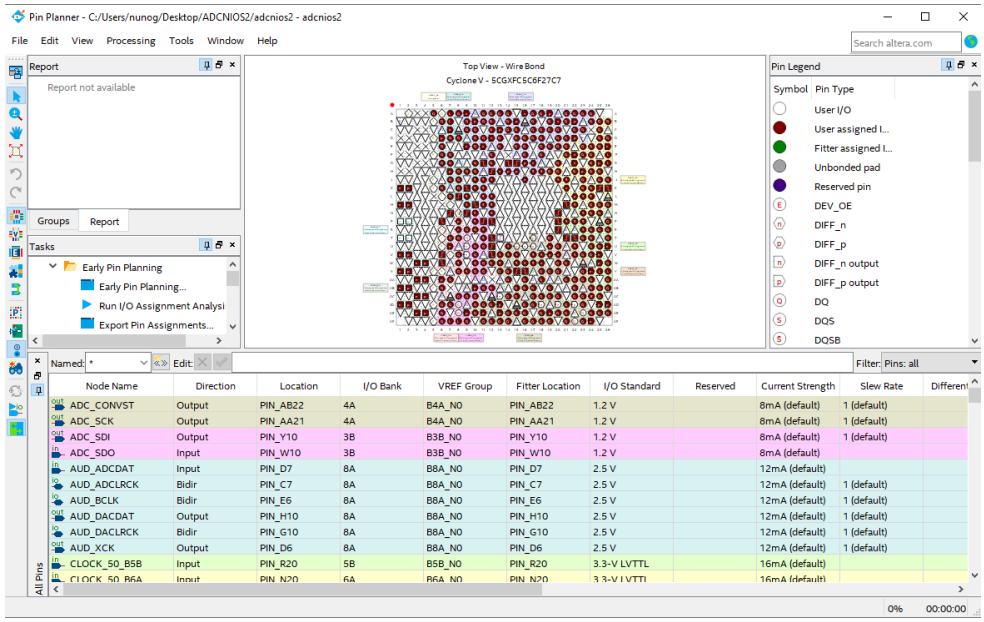

Figura 3.4: Exemplo da atribuição de variáveis aos pinos correspondentes do FPGA na ferramenta Pin Planner

programado assim que perder acesso à alimentação. Com a programação Active Serial, a vantagem é o contrário da desvantagem da programação via JTAG, ou seja, permite ao FPGA estar programado mesmo que tenha sido desligado. A desvantagem da programação via Active Serial é o *switching* necessário pré e pós programação, necessitando que o botão de programação esteja na posição "PROG"antes de ser programado e "RUN"depois de ser programado. Dito isto, no decorrer desta dissertação e, consequentemente, no desenvolvimento do FPGA é utilizado a programação via JTAG pela facilidade de utilização e pelo cumprimento dos requisitos para o sistema final.

3.4 Intel Avalon Interface

A fim de facilitar a criação, a conexão e o uso de componentes num FPGA, a Intel decidiu criar uma interface que fosse comum a todos os cores [4]. Este tipo de Interface, especificada como Avalon, é usado na ferramenta Platform Designer, anteriormente mencionada, e permite ao sistema estar constituído com uma transmissão de dados de alta velocidade, *registers* de escrita, leitura e memória, e, ainda, controlo de dispositivos fora da área do FPGA, como é o exemplo da figura 3.5.

Não só esta interface está presente em todos os cores IP providenciados pela Intel, como também pode ser utilizada como interface base na criação de cores personalizados. Assim, as Avalon Interfaces podem ser utilizadas para desenvolver e vender produtos, visto que correspondem ao padrão *open-source*.

Para cada tipo de interface, existe ainda *signal roles*, responsáveis por adicionar, retirar ou modificar funcionalidades segundo as necessidades do módulo. Cada Avalon Interface pode ter uma ou mais *signal roles*, e, cada *signal role* é definida com o tamanho (em bits), a direcção (entrada ou saída) e se é de uso obrigatório ou não.

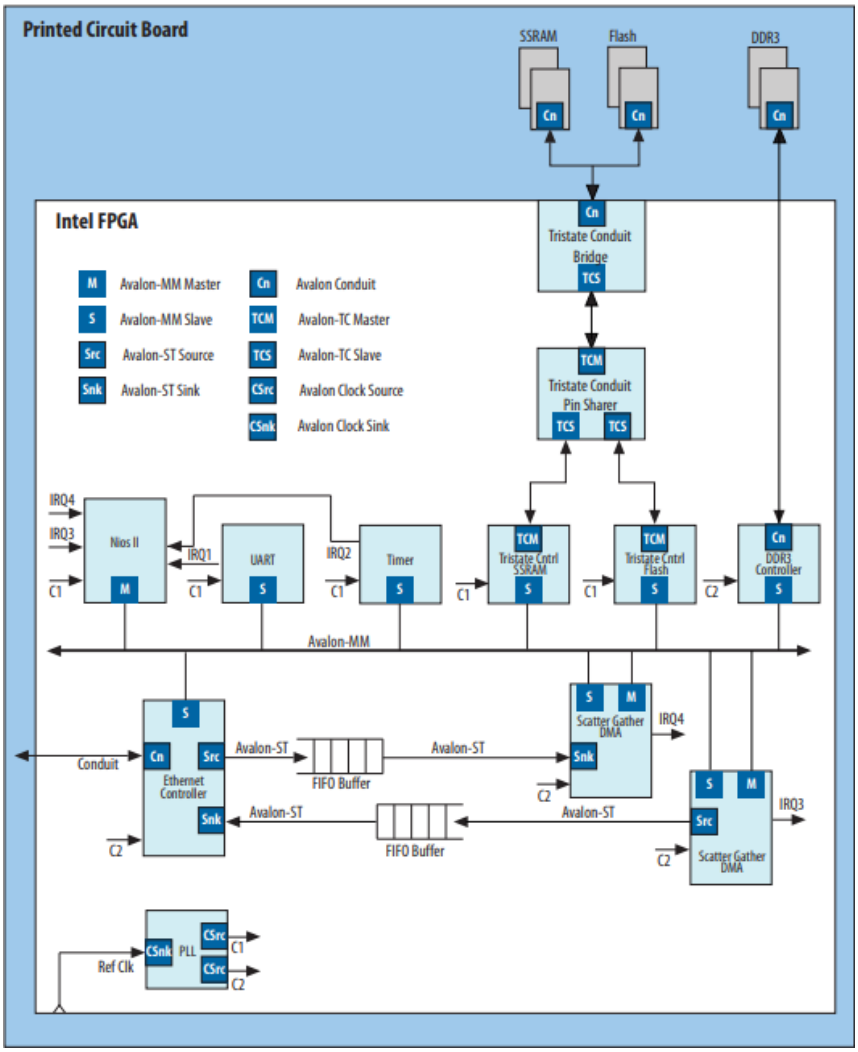

Figura 3.5: Design de um sistema utilizando diversos tipos de Avalon Interfaces [4]

3.4.1 Tipos de Interface

No total, existem sete tipos de Avalon Interfaces, todos eles com características e propósitos diferentes. No caso de criação de cores personalizados, segundo a Avalon Interface, o utilizador deve escolher qual a interface apropriada para o seu core, tendo em conta as suas características e comportamentos.

- Avalon Streaming Interface (Avalon-ST): Interface que suporta transmissão unilateral de dados. Normalmente utilizado em aplicações com *multiplexed streams*, pacotes de dados

e dados provenientes de Digital Signal Processing (DSP), como é possível observar pela figura 3.6.

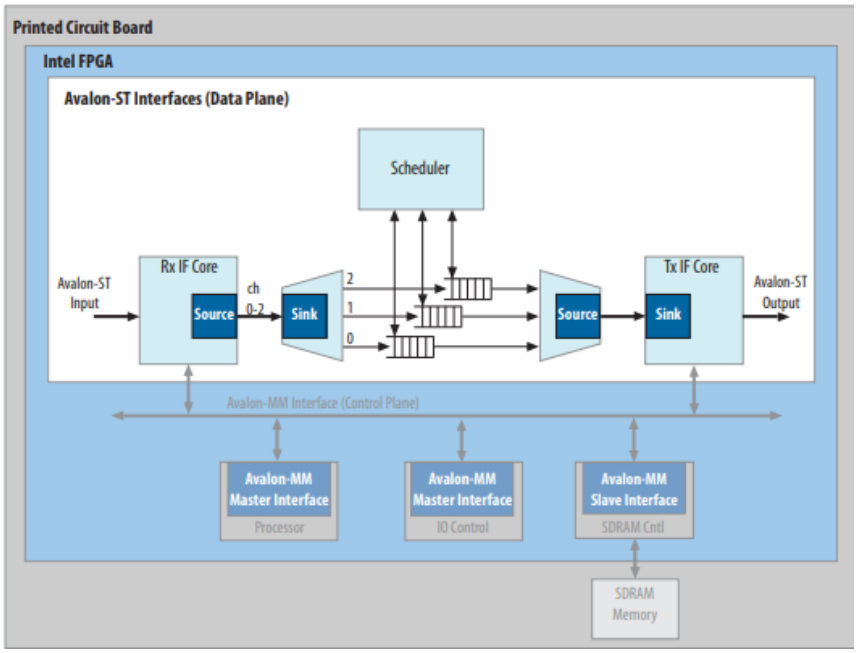

Figura 3.6: Exemplo de aplicação da Interface Avalon-ST [4]

- Avalon Memory Mapped Interface (Avalon-MM): Esta Interface é a interface mais comum dentro dos sete tipos. É uma interface baseada em operações de escrita e leitura através de um endereço, típico de conexões entre *master* e *slave*. Normalmente utilizada em componentes como, por exemplo, microprocessadores, memórias, UARTs, DMAs e Timers.
- Avalon Conduit Interface: Interface utilizada para exportar sinais utilizados dentro do Platform Designer de forma a conectar a outros módulos ou pinos do FPGA.
- Avalon Tri-State Conduit Interface (Avalon-TC): Interface que suporta conexões com periféricos fora da área interna do FPGA.
- Avalon Interrupt Interface: Como o próprio nome induz, esta Interface providência o alerta de eventos a outros componentes.
- Avalon Clock Interface: Interface utilizada para transmitir e receber *clocks*.
- Avalon Reset Interface: Interface que providência uma conexão à funcionalidade de *reset*.

3.4.2 Funcionamento

Componentes do FPGA podem ter entradas ou saídas de *clock*. Como exemplo, a Intel usa o componente *phase locked loop* (PLL) da figura 3.7.

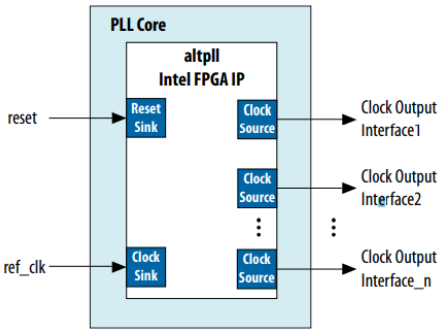

Figura 3.7: Representação do core PLL distribuído pela Intel [4]

Neste tipo de ambiente, o módulo PLL é geralmente utilizado para a criação de um ou vários *clocks* que visem cumprir os requisitos temporais dos componentes do sistema. A utilização do PLL é discutida posteriormente no capítulo da implementação.

Este tipo de *Avalon Clock and Reset Interfaces* tem, obviamente, *signal roles* associadas. Começando pelas entradas do módulo, tanto *Reset Sink* como *Clock Sink*, têm *signal roles* obrigatórias, como *reset* e *clk*, respectivamente. A *signal role clk* é, no fundo, um sinal de relógio à entrada do módulo que providência sincronização da lógica interna do módulo e para com outras interfaces. O módulo PLL pode ter diversas saídas, cada uma do tipo *Clock Source*, tendo obrigatoriamente uma *signal role* associada como *clk* - um sinal de relógio à saída do módulo.

Todas estas *signal roles* são depois modificadas na ferramenta Platform Designer, segundo os requisitos e restrições do sistema. Assim, as *signal roles* deixam de ser conceitos de protocolo e passam a ter hardware associado. Neste caso, *clk* na entrada (*Clock Sink*) seria associado para com o relógio interno do FPGA e *clk* externo (*Clock Source*) estaria disponível para ser associado a qualquer componente que precisasse de uma ligação *Clock Sink*.

Interfaces que usem apenas *clocks* ou *resets* são consideradas de baixa complexidade em comparação para com o resto das interfaces. Interfaces como Avalon Memory-Mapped (MM) são interfaces mais complexas e que involvem um maior número facultativo de *signal roles*. Um exemplo de uma Interface Avalon MM pode ser vista na figura 3.8.

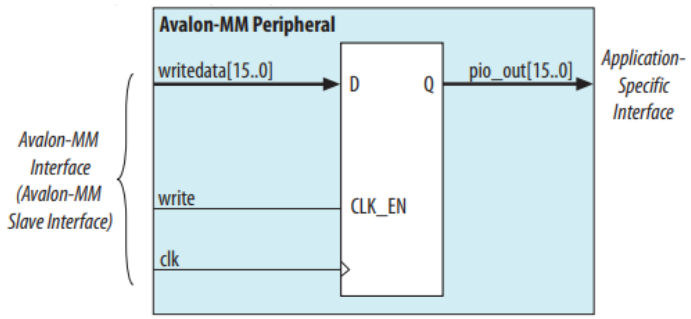

Figura 3.8: Exemplo de periférico que utiliza a Interface Avalon MM [4]

Este tipo de Interface passa a ter entradas e/ou saídas de dados definidas pelo tamanho e direcção da conexão. Sendo assim, do lado esquerdo do módulo, está representado as três *signal roles* que o caracterizam - writedata, write e clk. A *signal role clk*, como referido anteriormente, é um sinal de relógio à entrada do módulo. Adicional às Interfaces Clock e Reset, esta Interface MM possui a tarefa de receber, à entrada do módulo, dados provinientes de um módulo *master*. A *signal role writedata* é responsável por transmitir esses dados até ao módulo da figura 3.8. Segundo as especificações da Interface Avalon, *writedata* pode transmitir de 8 até 1024 bits, sendo que neste caso está definido para transmitir 16 bits. No entanto, a recepção destes dados está dependente de outra *signal role - write*. Quando definido, indica a transmissão de dados através da *signal role* writedata.

Observando a figura 3.8, é possível ainda verificar que a saída do módulo não faz parte da Interface Avalon, mas sim de uma aplicação especifica definida pelo *designer*.

Numa aplicação comum, um módulo que funciona segundo a Interface Avalon MM tem um endereço, permitindo identificar e ser identificado, dependente de ser *master* ou *slave*. Esse endereço pode ser de 1 até 64 bits, porém, geralmente é utilizado 32 bits em sistemas mais antigos e *low-cost*, e 64 bits em sistemas mais actuais, complexos e capazes. Os endereços utilizados nesta dissertação e aplicados ao sistema são de 32 bits. No software Quartus Prime, os endereços são utilizados em formato hexadecimal e, sendo 32 bits, são nomeados de *words*. Cada byte da *word* é referido como *offset* do endereço. Assim, o primeiro byte tem *offset* zero e o último tem *offset* de três.

A fim de facilitar a sincronização da transmissão de dados entre vários módulos com a Interface Avalon MM, para além das *signal roles address*, *write*, *writedata*, *read* e *readdata*, são utilizadas *signal roles* como: *byteenable*, *response* e *waitrequest*. Estas três últimas acrescentam complexidade à interface mas garantem o correto funcionamento, interno e externo, do módulo que as utiliza. A figura 3.9 demonstra o funcionamento de uma Interface Avalon MM que transmite e recebe dados, controlados pela *signal role waitrequest*.

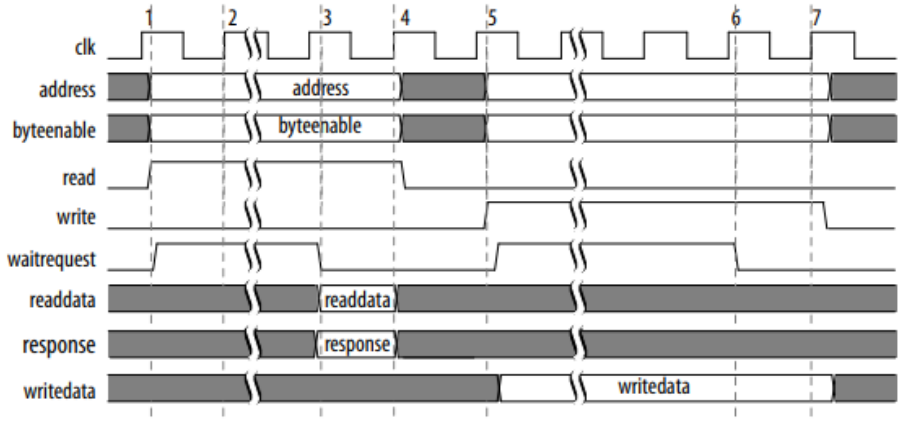

Figura 3.9: Transmissão e recepção de dados com recurso ao *waitrequest* [4]

Na primeira ascensão do sinal de *clk*, *address*, *byteenable* e *read* são declarados, informando o *slave* que o *master* pretende fazer uma leitura de dados. Logo em seguida, o *slave* declara *waitrequest*, ficando o sistema num estado de pausa até que *waitrequest* deixe de estar declarado. No caso da figura, à terceira ascensão do sinal de *clk*, *waitrequest* deixa de estar declarado, libertando o sistema e as restantes *signal roles* do estado de espera, e, em consequência, é enviado *readata* e *response* do *slave* para o *master*.

Numa nova comunicação, na quinta ascensão do sinal de *clk*, *signal roles* anteriormente utilizadas voltam a estar declaradas, como é o caso de *address* e *byteenable*. No entanto, visto que o *master* pretende, desta vez, enviar informação para o *slave*, adicionalmente às *signal roles* anteriores, também declara *write* e *writedata*. O *slave* volta a declarar *waitrequest* mantendo o sistema num estado de espera. Quando *waitrequest* deixa de estar declarado, o *slave* recebe a informação proveniente de *writedata*, acabando a comunicação entre os dois.

Para além deste tipo de comunicação entre *master* e *slave*, existem outros tipos de comunicação mais complexos, envolvendo outro tipo de *signal roles* e propriedades. Normalmente, ao adicionar complexidade, é necessário adicionar ferramentas que permitam controlar o aumento desta complexidade, como permissões e controlo de latência.

3.5 Nios II Soft-Processor

3.5.1 Arquitectura

O Nios II é um processador 32-bit de uso genérico com arquitectura *Reduced Instruction Set Computer* (RISC) [24]. A arquitectura utilizada no processador Nios II tem como característica a diminuição de ciclos por instrução a troca do número de instruções por programa, utilizando uma maior número de registos. Em comparação com a arquitectura contrária, *Complex Instruction Set Computer* (CISC), a utilização de RISC tende a ser ideal a este tipo de aplicação por ser *standardized* permitindo uma maior flexibilidade a camadas num mais alto nível, como é o caso da utilização da linguagem C/C++ para a programação do Nios II em toda a família de FPGAs da Intel.

Uma outra característica do processador Nios II é o facto de ser um processador *soft* e, por consequência, não estar fixado a silício. Esta característica permite que o FPGA tenha uma maior área disponível para o seu design e que a adição do processador seja facultativa.

No caso da adição do processador, o mesmo pode ainda ter as suas capacidades adaptadas aos requisitos e restrições do sistema, passando pela modificação das suas características na ferramenta Platform Designer.

3.5.2 Versões

A versão Lite do software Quartus Prime permite a utilização sem licença de uma das duas versões do Nios II, a versão Nios II/*e*. Esta versão é utilizada para uso genérico e numa utilização focada nos recursos disponíveis. A versão mais avançada, Nios II/*f*, é optimizada para o

desempenho do sistema, no entanto necessita de licença para a sua utilização. Um conjunto de características que diferenciam ambas as versões, pode ser observado na figura 3.10.

	Nios II/e	Nios II/f
Summary	Resource-optimized 32-bit RISC	Performance-optimized 32-bit RISC
Features	JTAG Debug ECC RAM Protection	JTAG Debug Hardware Multiply/Divide Instruction/Data Caches Tightly-Coupled Masters ECC RAM Protection External Interrupt Controller Shadow Register Sets MPU MMU
RAM Usage	2 + Options	2 + Options

Figura 3.10: Versões do *soft-processor* Nios II e as suas respectivas características

No decorrer desta dissertação, o *soft-processor* Nios II é utilizado como *master* do sistema e a versão escolhida é a versão Nios II/*e*. O poder computacional é comprometido a fim de garantir a continuidade do padrão *open-source*.

3.6 Embedded System Design

Num contexto abstracto, entre um microcontrolador e um FPGA, existe uma diferença: o microcontrolador possui de origem o seu hardware, enquanto que o FPGA não, o hardware tem de ser projectado e programado. Para que o FPGA consiga ter um fim similar ao de um microcontrolador, é fundamental que o FPGA tenha mais que apenas um processador, é necessário criar um sistema embarcado. Para tal, é essencial criar um meio que permita o acesso e a configuração do FPGA, a fim de gerar o hardware necessário [5]. Assim, é fundamental que o FPGA esteja devidamente alimentado electricamente e que exista uma interface JTAG que suporte a configuração de *hardware* e *software debugging*. A correta configuração de um interface JTAG no FPGA permite:

- Configuração do FPGA
- *Download* e *debug* de *software*
- Comunicação com o FPGA com uma *interface* similar à de uma *interface* UART, utilizando o módulo JTAG UART.
- *Debug* de *hardware* utilizando a ferramenta Signal Tap II do *software* de *design* Quartus Prime.
- Programar memoria *flash*

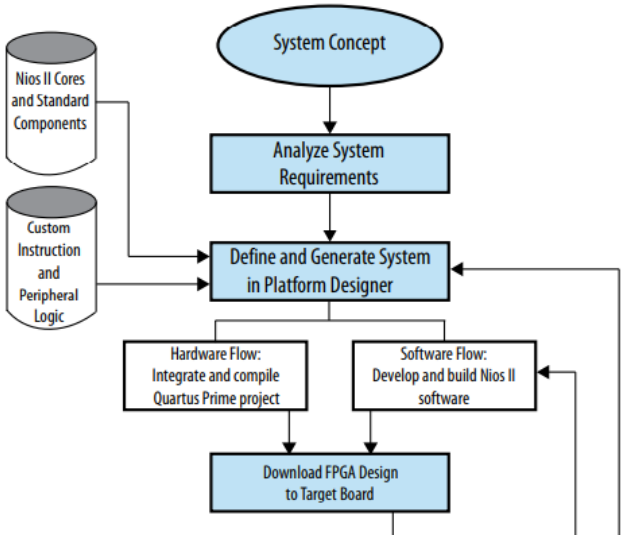

Figura 3.11: *General Nios II System Design Flow* [5]

Estas características permitem que, depois de providenciadas, seja possível obter um desenvolvimento em FPGA similar ao de um microcontrolador.

O desenvolvimento passa assim a contar com uma maior abstracção do *hardware*, permitindo o acesso a uma flexibilidade entre o mesmo e o software. Como é possível observar na figura 3.11, a ferramenta Platform Designer tende a ter um papel maioritário no controlo dos requisitos do sistema possibilitando dois tipos principais de desenvolvimento, a nível do *hardware* e a nível do *software*. É na camada de hardware que é possível integrar e compilar os projectos desenvolvidos no software Quartus Prime, no entanto, é também na camada de hardware que é possível responder aos requisitos necessários para que se consiga dividir o desenvolvimento do FPGA em hardware e software. Como referido anteriormente, é através da inclusão de um processador (Nios II) e respectiva configuração de uma interface JTAG, que é possível obter um ambiente de desenvolvimento muito similar ao de um microcontrolador.

Assim, o método de desenvolvimento em FPGA passa primeiro por garantir os requisitos em hardware, e, em seguida, garantir os mesmo em software. Por exemplo, no caso do sistema necessitar de medir o tempo de execução de uma função executada em software, inicialmente, o sistema necessita de ter o *hardware* capaz de satisfazer essa necessidade, um *timer*, e só depois, em software, o mesmo pode ser utilizado através de funções de drivers e Hardware Abstraction Layers (HAL).

Apesar de na ferramenta Platform Designer ser possível definir os requisitos em *hardware* para o *software* a utilizar, é na ferramenta Nios II Software Build Tools (SBT) for Eclipse onde acontece o desenvolvimento de software em cima do *hardware* definido.

3.7 Nios II Embedded Design Suite (EDS)

O Nios II Embedded Design Suite (EDS), é um pacote de ferramentas e bibliotecas de software fundamentais para o compilar e *debug* de aplicações para o processador Nios II. Observando a figura 3.12, o EDS é constituído por Nios II Software Build Tools (SBT), por um IDE (Eclipse) e software indicado para o uso do *hardware*, como drivers e bibliotecas HAL.

3.7.1 Nios II Software Build Tools (SBT)

As Software Build Tools (SBT) para o processador Nios II, são um conjunto de ferramentas que permitem o acesso ao processor Nios II e às suas funcionalidades. Como é possível verificar na figura 3.12, este conjunto de ferramentas, inclui ferramentas como compilador, *debugger*, *programmer*, editor de texto, entre outros.

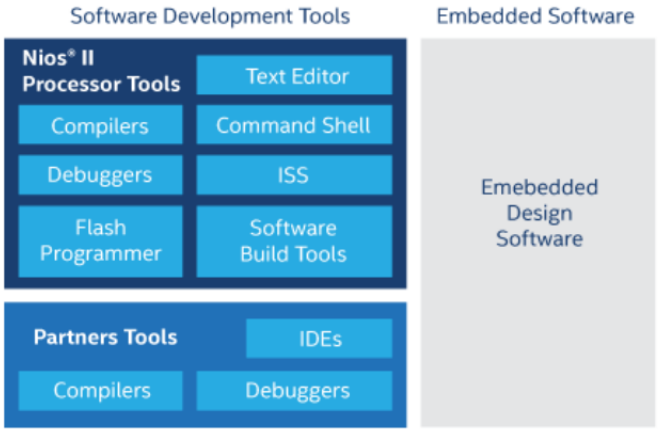

Figura 3.12: Nios II Embedded Design Suite (EDS)

Este conjunto de ferramentas tem um papel fundamental no desenrolar desta dissertação. São estas as ferramentas que permitem o acesso ao sistema com uma maior abstracção de hardware, permitindo, por exemplo, o desenvolvimento em FPGA utilizando linguagens de programação e não linguagens de design de hardware (HDL). Com este contributo, é possível transitar para um desenvolvimento mais comum com o desenvolvimento de software em sistemas embarcados, utilizando como linguagem de programação a linguagem C.

Para aceder às Software Build Tools, é possível fazê-lo através da Graphical User Interface (GUI) providenciada pelo Eclipse ou através da *shell* de comandos para o processador Nios II. Ambas as possibilidades estão incluídas na instalação do pacote do Nios II Embedded Design Suite. Ao longo da dissertação é utilizado a GUI do Eclipse em conjunto com as Software Build Tools (SBT) do Nios II para programar o FPGA de forma semelhante a um microcontrolador.

Em cada projecto criado a partir das SBT na GUI do Eclipse, é gerado uma pasta que agrupa a aplicação para o Nios II e uma pasta que agrupa a Board Support Package (BSP) para o Nios II [6].

Apesar de terem aplicações diferentes, estas duas pastas têm uma característica em comum - ambas possuem Makefile. As SBT criam dois tipos de Makefile. Do lado da aplicação, o Makefile constrói a aplicação. Do lado da BSP, o Makefile é mais complexo e é gerado em conformidade com os requisitos de hardware do sistema. Estes dois Makefiles são gerados pelas SBT apesar de o utilizador poder escolher usar um próprio. No decorrer da dissertação, os Makefiles utilizados são gerados pelas SBT para facilitar a integração com o resto do sistema e seguir a recomendação de uso do mesmo dada pelo fabricante.

Novamente, do lado da aplicação a organização do sistema é bem mais simples do que do lado da Board Support Package (BSP). Do lado da aplicação, a pasta que contém o projecto consiste no conjunto de código fonte com um ficheiro Makefile. Um dos ficheiros de código fonte possui a função main(). Este código fonte é responsável pelo controlo das funcionalidades do FPGA e por chamar bibliotecas definidas na parte de BSP, como por exemplo, as bibliotecas HAL. Depois de compilar o projecto, o Makefile da aplicação cria um ficheiro *Executable and Liking Formate File* (.elf) que é usado para operar o Nios II.

Do lado da Board Package Support (BSP), a organização do pacote é similar da figura 3.13.

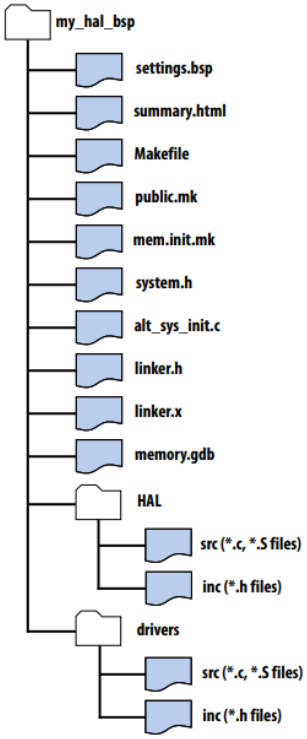

Figura 3.13: Exemplo da organização da pasta BSP. [6]

A BSP contém elementos como bibliotecas HAL, Biblioteca padrão C, Drivers, *Real-time operating system* e software adicional.

Adicionalmente a estes elementos, a BSP ainda contém elementos como o ficheiro system.h que possui dados sobre o *hardware* do sistema e o ficheiro settings.bsp que dispõe das configurações definidas para a BSP. Algumas definições da BSP devem ser modificadas a fim de usar determinadas funcionalidades das SBT.

3.7.2 Hardware Abstraction Layer (HAL)

Hardware Abstraction Layer (HAL) existe para permitir o uso das funcionalidades do sistema, omitindo partes mais complexas de camadas mais baixas (hardware) e assim facilitar o desenvolvimento e garantir a sua independência. Para o processador Nios II, a HAL providenciada tem o mesmo papel juntando as funcionalidades de uma biblioteca C padrão, como é possível observar na figura 3.14. Este tipo de junção, permite ao sistema que tenha acesso a funções comuns como printf(), fopen(), fwrite(), entre outras.

Na criação de um novo projecto, as Software Development Tools (SDT) extraem informação do SOPC Information File proveniente do sistema criado na ferramenta Platform Designer. Este ficheiro, .socinfo, é então utilizado para gerar uma BSP personalizada e consequente biblioteca HAL.

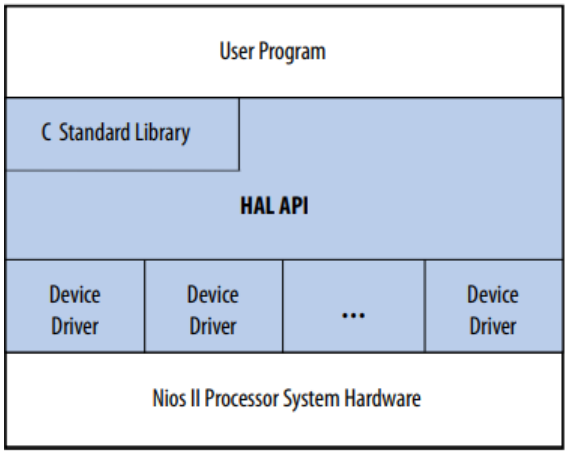

Figura 3.14: Camadas da BSP (a azul) [6]

A biblioteca HAL concede classes modelo para uso genérico, dependendo da utilização. A biblioteca HAL providencia classes modelo para:

- Dispositivos *character-mode* - Periféricos que recebem ou transmitem dados através de uma comunicação série, como é o caso de UART.
- Timers - Periféricos que contam ciclos de relógio e geram interrupções segundo pedidos.
- Subsistemas de ficheiros - Mecanismo para aceder a ficheiros guardados em dispositivos físicos.

- Dispositivos Ethernet - Dispositivos que concedem acesso a uma conexão Ethernet ou *network stack*.
- Dispositivos Direct Access Memory (DMA) - Periféricos que executam transmissões com grandes quantidades de dados.
- Dispositivos de memória *flash*

O uso de biblioteca HAL permite ao utilizador aceder a cada classe, independentemente de outras implementações. O acesso é feito sem que haja a necessidade do utilizador escrever rotinas de baixo nível. Por exemplo, no caso da classe de dispositivos *character-mode*, como UART, a comunicação pode ser estabelecida utilizando fopen() e a transmissão pode ser feita através de fprintf().

3.8 Analog to Digital Converter (ADC) LTC2308

Um dos principais objectivos do sistema é garantir que o mesmo consiga fazer uma leitura de dados, efectuar o processamento desses dados e, se possível, acelerar esse processamento. Para a obtenção de dados provenientes de um sensor, é utilizado o Analog to Digital Converter (ADC) presente na placa de desenvolvimento, o LTC2308 [25].

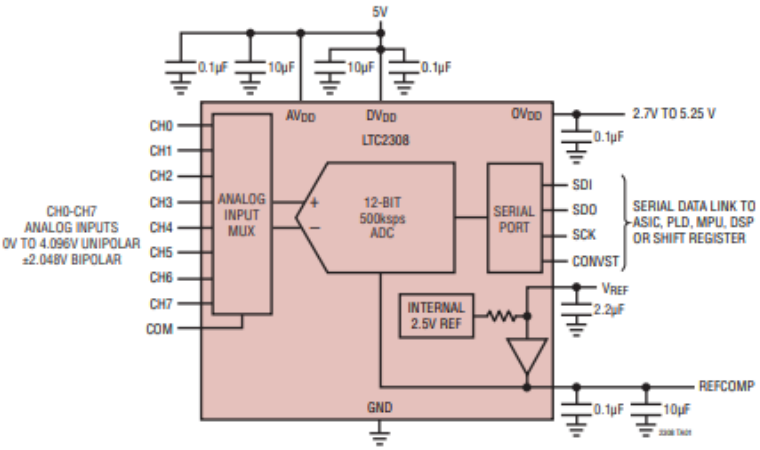

Figura 3.15: *Layout* do ADC LTC2308

Através da figura 3.15, é possível observar o *layout* do ADC LTC2308. O mesmo é constituído por 8 canais de entrada e tem como principais características a taxa de amostragem de 500kps e uma resolução de medida de 12 bits. Estas características são razoáveis tendo em conta o uso genérico da placa de desenvolvimento. Como comparação, é possível verificar que o ADC integrado no Arduino Uno, dentro do microcontrolador atmega328p, possui apenas uma taxa de amostragem de 77kps e 10 bits de resolução, valores inferiores ao LTC2308. Em contexto de tensão, pode dizer-se, que, a partir de uma tensão de referência de 4.096V, o ADC LTC2308 possui uma resolução de $V_{step} = \frac{4.096V}{1024bits} = 4mV$

Este ADC tem ainda outra característica: interface SPI. Esta interface permite que o ADC seja controlado através de um *bus* SPI e que passe a servir como *slave* na comunicação. Assim, o FPGA está conectado ao ADC pela interface SPI, necessitando de *hardware* que suporte a *interface* em questão.

O funcionamento do ADC tem como base a comunicação que se encontra estabelecida entre o FPGA e o ADC. É através de uma das ligações da *interface* SPI, a ligação Slave Data In (SDI), que o conversor é configurado. Do lado do FPGA, 6 bits são estabelecidos de acordo com a configuração requisitada e são inseridos na ligação Slave Data Out (SDO), que, por sua vez, está conectada à ligação SDI do lado do ADC. Agora do lado do ADC, cada bit na entrada do SDI é inserido a cada ciclo de relógio. Depois de 6 ciclos de relógio, o ADC encontra-se configurado para a próxima conversão.

S/D	O/S	S1	S0	UNI	SLP

S/D = SINGLE-ENDED/$\overline{\text{DIFFERENTIAL}}$ BIT
O/S = ODD/$\overline{\text{SIGN}}$ BIT
S1 = ADDRESS SELECT BIT 1
S0 = ADDRESS SELECT BIT 0
UNI = UNIPOLAR/$\overline{\text{BIPOLAR}}$ BIT
SLP = SLEEP MODE BIT

Figura 3.16: Os 6 bits de configuração do ADC.

Table 1. Channel Configuration

S/D	O/S	S1	S0	0	1	2	3	4	5	6	7	COM
0	0	0	0	+	−							
0	0	0	1			+	−					
0	0	1	0					+	−			
0	0	1	1							+	−	
0	1	0	0	−	+							
0	1	0	1			−	+					
0	1	1	0					−	+			
0	1	1	1							−	+	
1	0	0	0	+								−
1	0	0	1			+						−
1	0	1	0					+				−
1	0	1	1							+		−
1	1	0	0		+							−
1	1	0	1				+					−
1	1	1	0						+			−
1	1	1	1								+	−

Figura 3.17: Tabela de configuração do LTC2308

A figura 3.16 mostra o propósito de cada bit na configuração do ADC e a tabela da figura 3.17 mostra de que forma esses bits devem ser definidos a fim de obter os resultados expectáveis.

O ADC entra em funcionamento quando a ligação CONVST fica num estado lógico alto durante um pequeno intervalo de tempo e volta ao estado lógico baixo logo em seguida. Depois disso, existe um tempo de conversão, dependendo do tempo da mesma, até que o ADC insira os

dados dessa mesma conversão dentro da ligação SDO, prontos a serem enviados para o FPGA. Quando isso acontece, e visto que SDO guarda 12 bits a cada conversão, o ADC começa por receber ciclos de relógio pela ligação SCK. Quando 12 ciclos já tiverem passado e todos os dados tenham sido transmitidos por SDO, acaba a fase de conversão.

Voltando à configuração necessária do ADC, ao mesmo tempo que os dados da primeira conversão são inseridos e enviados por SDO, na ligação SDI estão a entrar os seis bits de configuração necessários para a próxima conversão de dados. Depois destes seis bits serem interpretados pelo ADC, está a existir, ainda, o envio dos dados da conversão para o FPGA através de SDO, faltando ainda seis ciclos de relógio até acabar a transmissão. No começo dos seis ciclos de relógio, o ADC começa a preparar uma nova conversão com a nova configuração, começando por adquirir o sinal à entrada do mesmo. Este funcionamento pode ser observado pela figura 3.18.

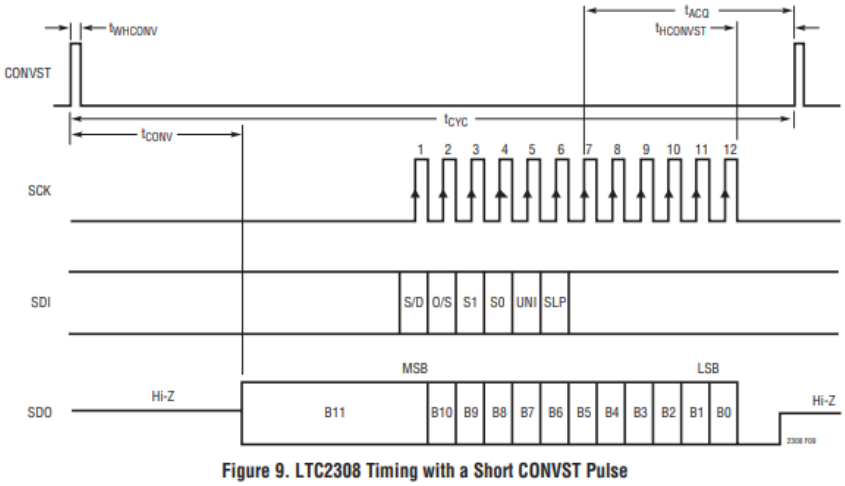

Figure 9. LTC2308 Timing with a Short CONVST Pulse

Figura 3.18: Funcionamento do ADC LTC2308 através da interface SPI

Capítulo 4

Implementação

4.1 Solução proposta

A solução para o problema na secção de Introdução pode ser dividido em duas resoluções: na perspectiva de hardware e na perspectiva da estratégia a utilizar.

4.1.1 Hardware

O uso de FPGA nos equipamentos industriais, vem retirar a impossibilidade da reconfiguração de um equipamento. A utilização de ASICs numa camada mais baixa de produção, atrasa a actualização da indústria pela impossibilidade de reconfiguração após a sua implementação. Com a utilização de FPGAs em ambiente industrial, é possível reprogramar o mesmo de maneira que cumpra os requisitos impostos pelo sistema. Esta solução (figura 4.1) retira a necessidade recorrente de pesquisa, de compra e de implementação, mantendo só os custos de compra unitária e de design do FPGA.

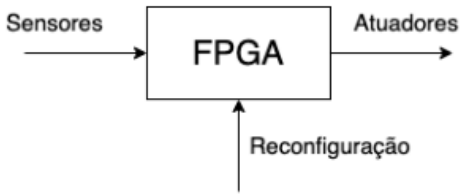

Figura 4.1: Solução a nível do hardware

4.1.2 Estratégia

Com a estagnação e obstáculos à evolução nas tecnologias usadas, o sistema industrial acaba por não estar organizado de maneira optimizada. Perto dos equipamentos é feita a aquisição de dados dos sensores e/ou atuadores, deixando a computação de dados para a camada mais alta

da hierarquia do sistema. Deste modo, o equipamento de aquisição de dados tem de esperar a recepção de informação da camada mais alta, gerando uma grande latência na comunicação entre os dois. Este problema seria facilmente resolvido, optimizando a estratégia industrial a utilizar e enquadrando o sistema dentro do conceito da Indústria 4.0 [26]. Esta estratégia, figura 4.2, passa por descentralizar a computação da aquisição de dados, atribuindo às camadas mais baixas do sistema a tarefa de processar certos dados, que só seriam processados em camadas mais altas, e a tarefa de reencaminhar os dados processados para uma camada intermédia do sistema, em vez da transmissão para uma camada alta do sistema, diminuindo assim os tempos de espera entre as comunicações e dando às camadas mais baixas uma maior autonomia.

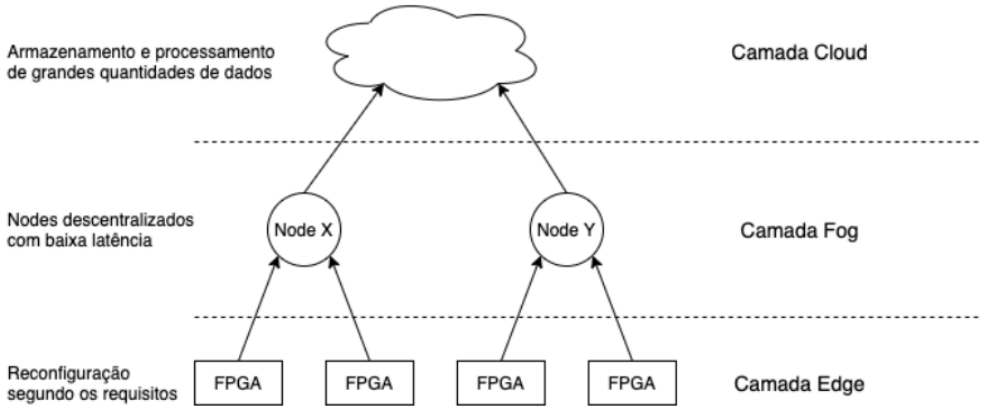

Figura 4.2: Esquema da solução encontrada.

Esta solução é representada na forma de plataforma, neste caso, a plataforma Dinasore.

4.2 Criação do sistema Qsys

Tal como referido no capítulo 3, a ferramenta Platform Designer é responsável por agrupar os módulos *hardware*, que, posteriormente, constituem o sistema final, permitindo o mesmo ser desenvolvido em *software*. É também na ferramenta Platform Designer que os diferentes parâmetros para os módulos necessários são definidos, podendo ser modificados a qualquer altura.

À criação de um sistema na ferramenta Platform Designer, o mesmo designa-se de *Qsys*, antigo nome da ferramenta Platform Designer.

4.2.1 Requisitos

Perante o objectivo citado no capítulo 2, o FPGA necessita estabelecer ligação ao ADC (LTC2308), garantir o correto funcionamento do sistema, ceder canais de comunicação entre o FPGA e terceiros, e ainda, permitir que o desenvolvimento do mesmo seja feito maioritariamente em *software*. Para tal, é necessário começar por estabelecer os requisitos em *hardware* antes de passar para a camada de desenvolvimento em *software*.

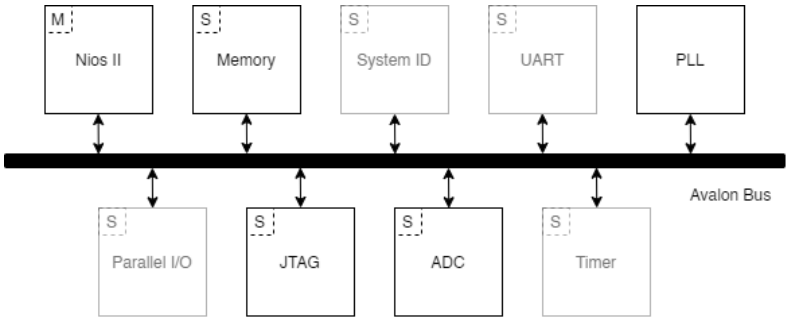

Figura 4.3: Representação dos módulos que constituem o sistema obrigatório

O sistema necessita de um módulo que actue como *master*, enviando os devidos comandos quando necessário, para que consiga controlar os restantes módulos. Como referido anteriormente, além de permitir o controlo do sistema, o processador Nios II também permite que o desenvolvimento do sistema seja feito em *software*, mostrando-se adequado para este tipo de aplicação. Os restantes módulos devem comportar-se como *slaves*, deixando o controlo do sistema da responsabilidade do processador Nios II, como é possível observar pela figura 4.3.

Para que o FPGA se comporte de forma similar a de um microcontrolador, necessita, além de um processador, de uma interface JTAG que permita a sua configuração, *debugging*, entre outros. Em seguida, para que o FPGA seja desenvolvido em *software*, outro requisito é a necessidade de memória de Random Access Memory (RAM), de maneira a que seja possível a alocação de variáveis e a sua respectiva utilização em *software*.

Estes três requisitos, mais o uso do ADC, constituem o sistema mínimo necessário para obter resultados do FPGA provenientes do ADC. No entanto, existe o requisito da existência de um modulo PLL, de forma a que o ADC seja operável.

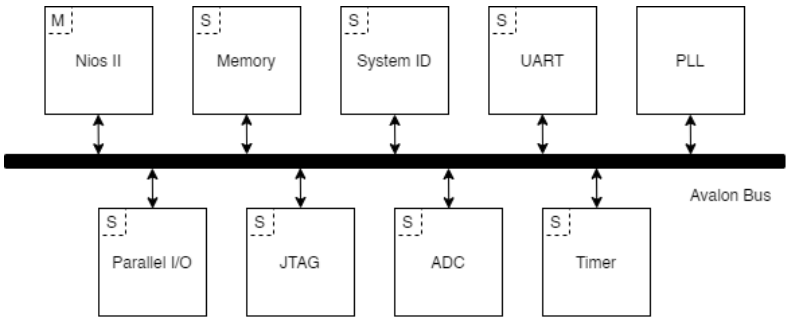

Figura 4.4: Representação dos módulos que constituem o sistema final

Para além destes cinco módulos já identificados, existem outros quarto opcionais que possuem um impacto positivo no sistema e são utilizados ao longo desta dissertação. A partir da figura 4.4 é possível observar os módulos constituintes do sistema final.

O uso de módulo System ID, permite que o sistema criado na ferramenta Platform Designer obtenha um identificador único. Esse identificador é depois utilizado para assegurar que o sistema

compilado e presente no FPGA, é o mesmo que está a ser desenvolvido em *software*.

A utilização do módulo de Parallel I/O permite ao FPGA o acesso a periféricos como LEDs, botões e *switches*. Útil para um *debugging* visual e para modificar fisicamente o canal a utilizar do ADC. Apesar de ter estas duas funções principais, o uso deste módulo pode ter outras utilizações dependendo dos requisitos do sistema e da decisão do *developer*.

O módulo UART possibilita o uso de uma comunicação série independente da comunicação de interface JTAG, permitindo ao FPGA comunicar com um computador ou outro tipo de dispositivo que também suporte a comunicação UART. Por fim, o módulo de Timer permite calcular o tempo de execução das funções em *software*.

4.3 Intellectual Property Cores

Apesar da Intel providenciar um conjunto de módulos IPs com o software Quartus Prime, alguns IPs são exclusivos da placa em uso. Assim, módulos que não se encontram disponíveis na ferramenta Platform Designer, precisam de ser importados para a mesma. Geralmente, dentro da documentação respectiva da placa em uso, existe uma secção com os módulos não disponíveis e prontos a serem importados.

No caso da placa de desenvolvimento usada nesta dissertação, o módulo correspondente ao ADC não está disponível inicialmente pela ferramenta Platform Designer, pelo que é necessário importa-lo para o poder usar juntamente com os outros módulos já presentes.

4.3.1 Parâmetros e Ligações

No capítulo 3 é referido que cada Interface Avalon tem as suas respectivas *signal roles*. Dentro da ferramenta Platform Designer, é possível aceder a algumas dessas *signal roles*, propriedades e modifica-los em forma de parâmetros perante as necessidades do sistema. Observando novamente o módulo PLL pela figura 4.5, desta vez através da ferramenta Platform Designer.

Perante este exemplo, é possível verificar semelhanças em relação à figura 3.7. À entrada do módulo PLL é utilizado um sinal de relógio igual ao do sistema, 50Mhz, e à saída é gerado três sinais de relógio, apesar de ser apenas possível visualizar o de 40MHz.

Este tipo de semelhanças é possível verificar em todos os módulos utilizados, alguns com mais parâmetros que outros. No entanto, o fundamental, para o correto funcionamento do sistema, é que esses parâmetros tenham correlação com o mesmo e que as ligações entre os módulos estejam corretas. Assim, os requisitos de cada módulo devem ser considerados antes de serem definidos o valores dos parâmetros.

Na figura 4.5, é considerado que o relógio de referência à entrada tem de ser o mesmo que o sinal de relógio à entrada do módulo. Visto que o sistema é alimentado por um sinal de relógio de 50MHz e conexão é feita do relógio para o módulo PLL, então o parâmetro está correto. Esta ligação pode ser observada através da figura 4.6, onde a saída do relógio do sistema está conectada à entrada do sinal de relógio do módulo PLL.

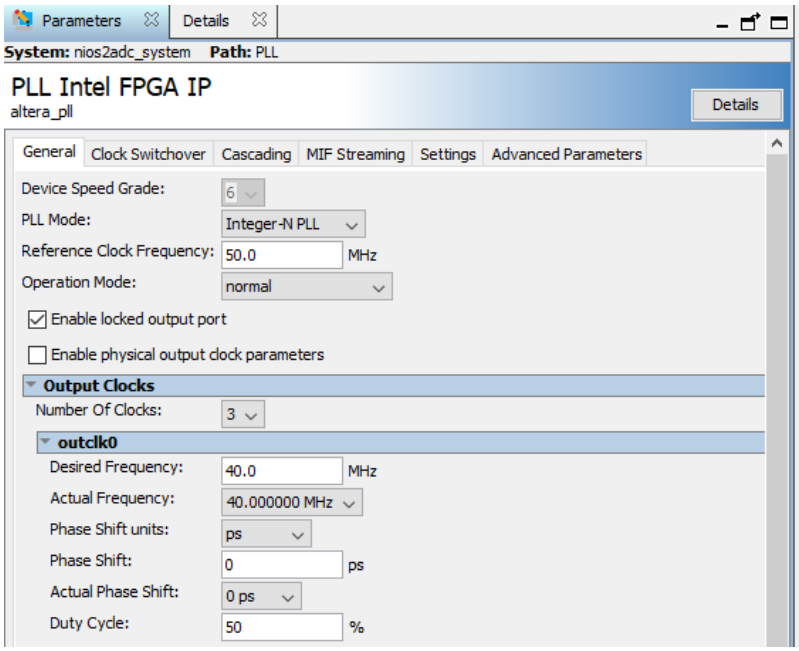

Figura 4.5: Parâmetros do módulo PLL na ferramenta Platform Designer

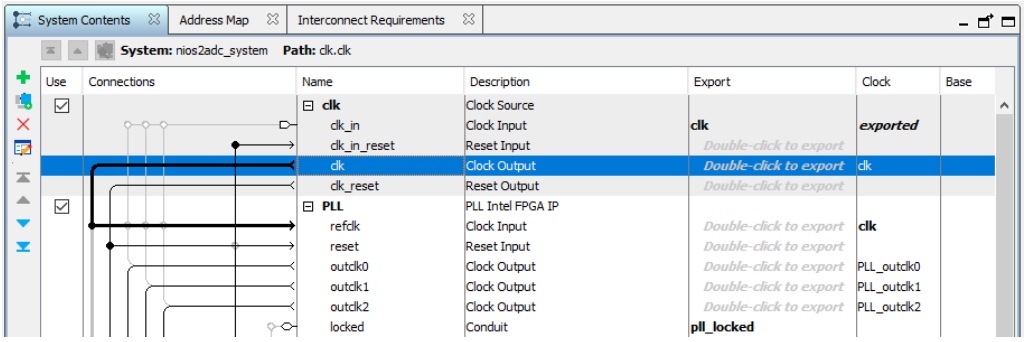

Figura 4.6: Ligação de *clk* com *refclk* do módulo PLL na ferramenta Platform Designer

Ainda na mesma figura, é possível também observar uma *signal role* adicional, o *reset*, tanto na entrada como na saída. Ao contrário das *signal roles* relativas aos sinais de relógio, o *reset* não faz parte dos parâmetros do módulo PLL, concluindo que nem todas as *signal roles* fazem parte dos parâmetros de um módulo.

4.3.2 On-Chip Memory (RAM or ROM)

O primeiro módulo a ser definido dentro dos essenciais da figura 4.3, é o módulo da memória RAM. Este módulo não depende de nenhum outro módulo para ser definido, no entanto, o módulo do Nios II depende dele.

O único parâmetro a ser modificado é o parâmetro da quantidade de bytes a alocar para a

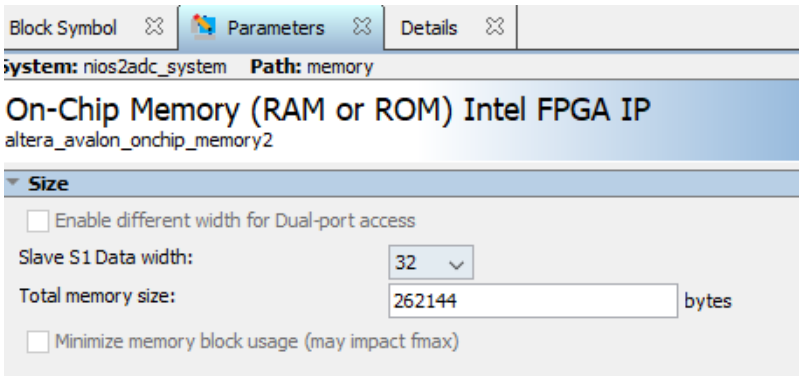

Figura 4.7: Parâmetros do módulo de memória RAM

memória, como é possível observar pela figura 4.7. Neste parâmetro deve ser considerado a quantidade de memoria embarcada que o FPGA possui para este tipo de aplicação. Como observado na figura 3.2, o FPGA possui um total de 4884 Kilobits de memória embarcada, ou seja, 610 500 bytes. O valor do parâmetro da memória não deve ser maior que este, no entanto deve ser alto o suficiente para garantir o correto funcionamento do *software*.

Em sistemas com memória RAM limitada, o *software* pode ainda ser adaptado para utilizar bibliotecas com uma *foot-print* mais pequena. Nesta dissertação, utiliza-se um valor para a memória RAM perto de metade do máximo da memória disponível no FPGA, garantido flexibilidade suficiente para o seu desenvolvimento.

4.3.3 Nios II Processor

Com os parâmetros definidos no módulo de memória RAM, já é possível definir os parâmetros do processador Nios II. O primeiro parâmetro a ser definido no módulo do processador Nios II é a versão do mesmo. Como referido no capítulo 3, existem duas versões distintas. Nos parâmetros do módulo Nios II é escolhida a versão Nios II/*e*, seleccionando a respectiva versão disponível na primeira secção dos parâmetros.

Na secção *Vectors* dos parâmetros do processador Nios II, é necessário escolher os parâmetros consoante a memória RAM definida. Para tal, é necessário definir os vectores segundo a ligação que contém a maior parte das *signal roles* do módulo de memória RAM, *Avalon Memory Mapped Slave* (s1).

Observando a figura 4.8, é possível identificar a ligação s1 do módulo de memória RAM e o seu endereço. Esse endereço é utilizado para definir os dois vectores pertencentes à secção de parâmetros *Vectors* do Nios II.

Em caso de partilha de um ou mais endereços com um ou mais módulos, é necessário redefinir os endereços para que não exista partilha dos mesmos entre os vários módulos que constituem o sistema. Para isso é necessário aceder a *System*, e, em seguida, a *Assign Base Addresses*, permitindo à ferramenta Platform Designer definir automaticamente novos endereços para os módulos existentes.

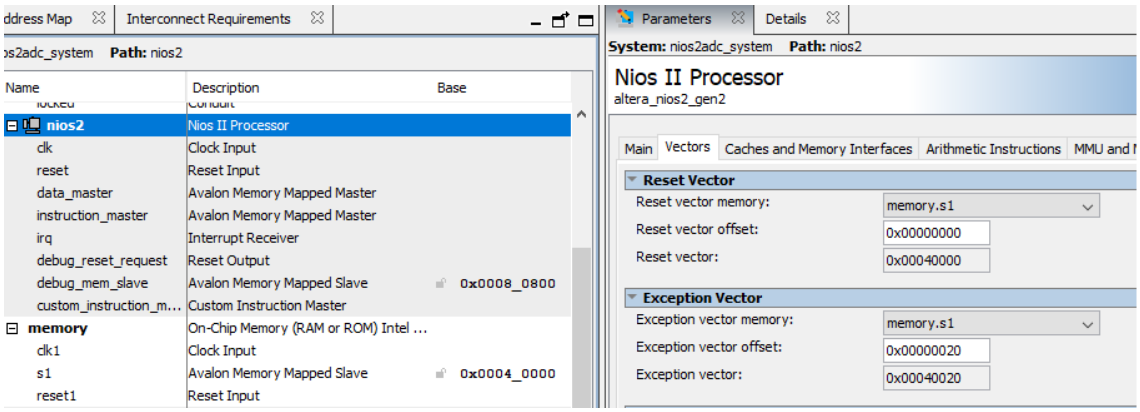

Figura 4.8: Escolha de vetores do Nios II segundo a memória RAM.

Ainda em relação ao módulo do processador Nios II, os restantes parâmetros estão relacionados com a versão Nios II/*f* e, por isso, devem permanecer como foram definidos no inicio da ferramenta Platform Designer.

4.3.4 JTAG UART Intel FPGA IP

O módulo JTAG UART não necessita de ser configurado depois de adicionado ao sistema. Para além de *clk*, *reset* e Avalon Memory Mapped Slave, este módulo possui um Interrupt Sender que deve estar conectado ao processador Nios II através de uma ligação até ao Interrupt Receiver, como mostra a figura 4.9.

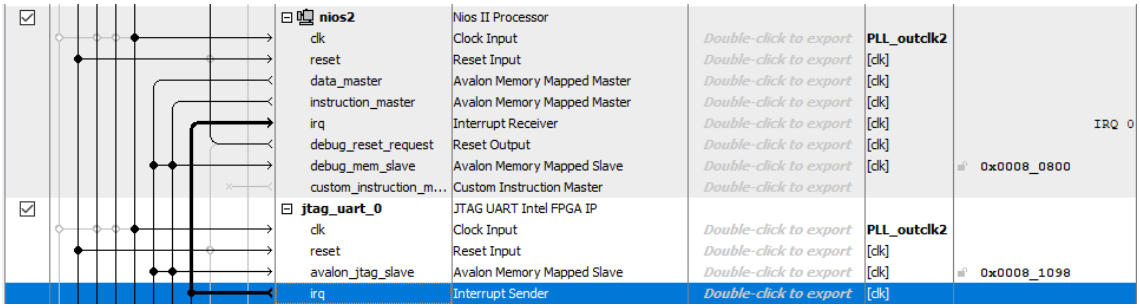

Figura 4.9: Ligação entre o módulo JTAG e o processador Nios II

4.3.5 Phase Lock Loop (PLL)

O PLL é usado neste sistema para criar três sinais de relógio distintos. Dois desses sinais de relógio servem directamente o funcionamento do ADC e o restante, alimenta o resto do sistema. Na figura 4.10 é possível observar os parâmetros dos sinais de relógio gerados.

Como referido anteriormente, o PLL recebe à entrada um sinal de relógio com uma frequência de 50MHz. A partir deste sinal de relógio, o mesmo gera três sinais de relógio para garantir o

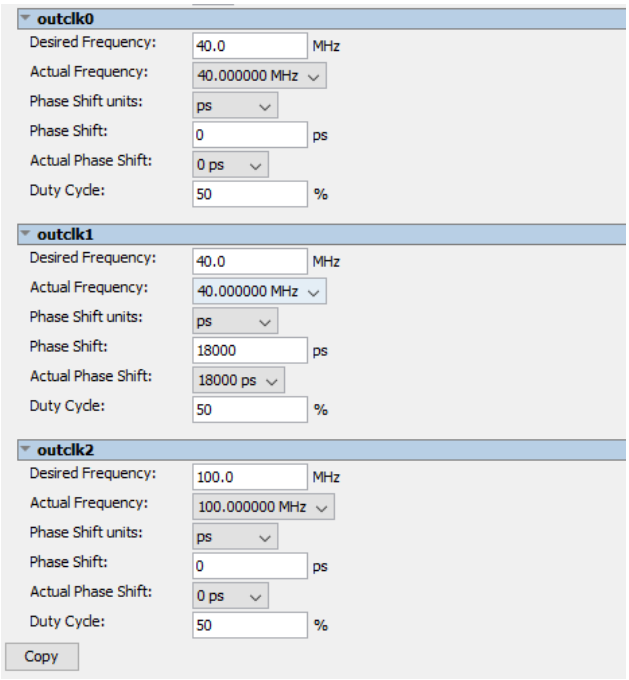

Figura 4.10: Parametros dos sinais de relógio gerados pelo módulo PLL

correto funcionamento do sistema. A primeira necessidade a ser atendida é a frequência de funcionamento do ADC, a qual pode ser entendida até 40MHz. No entanto, devido às características do módulo do ADC, é necessário criar um outro sinal de relógio com um desvio de fase, para corrigir o atraso que existe entre a escrita e a leitura no módulo do ADC. Por fim, o terceiro sinal de relógio é criado para aumentar a frequência de funcionamento do processador Nios II e respectivos periféricos. Assim, é este sinal de relógio, *outclk2*, que alimenta o resto do sistema a uma frequência de 100MHz.

Este módulo, possui ainda uma característica diferente em relação aos módulos já referidos: uma interface Conduit. Este tipo de interface permite ter acesso à variável exportada pela interface fora da ferramenta Platform Designer.

4.3.6 ADC_LTC2308

O módulo do ADC utilizado neste sistema é o único que é importado para a ferramenta Paltform Designer, e, como tal, não possui parâmetros. No entanto, possui outras características peculiares. Possui três entradas de sinal de relógio, sendo que duas delas estão referidas na descrição do módulo PLL: dois sinais de relógio de 40MHZ, sendo que umm deles está atrasado em relação ao outro. No entanto, a fim de acelerar a aquisição de dados a partir do ADC, o módulo também possui uma fila FIFO feita em hardware que permite o seu funcionamento a 100MHz, a frequência utilizada em quase todo o sistema.

4.3.7 System ID Peripheral Intel FPGA IP

O módulo do System ID, tal como o módulo do JTAG, não necessita de nenhuma alteração. No entanto, e dependendo da decisão do *developer*, é possível modificar o ID do sistema através do parâmetro do módulo, tendo como inicial o ID 0x00000000.

4.3.8 UART (RS-232 Serial Port) Intel FPGA IP

O módulo UART é constituído por parâmetros que constituem as configurações da comunicação UART, como *baud rate*, bits de paridade, stop bits, entre outros. O módulo é inicializado com o padrão 9600/8/N/1, ou seja, a comunicação é configurada a uma frequência de 9600 bits/s, com 8 *bits* de dados por trama, sem *bits* de paridade e com um *bit* de *stop*. No entanto, nesta dissertação, a frequência à qual a comunicação é feita, é alterada para o valor de 115200 bits/s.

4.3.9 PIO (Parallel I/O) Intel FPGA IP

O módulo PIO pode ser utilizado como input, output, ou ambos. Porém, neste sistema, é utilizado como input, no caso dos *switches*, e como output, no caso dos LEDs.

Inicialmente, nos parâmetros, define-se a direcção do módulo, alternando entre input ou output e escolhendo o tamanho do módulo em bits. Como a placa de desenvolvimento possui, entre outros, dez LEDS vermelhos e dez *switches*, então, para ambos os módulos, o tamanho a ser definido é de 10 bits, a fim de poder controlar cada um dos LEDs ou *switches* individualmente. Para além disso, no caso dos *switches*, o parâmetro do Interrupt deve ser activado, e deve ser do tipo LEVEL. Assim, o processador Nios II recebe uma interrupção sempre que o estado lógico de um switch for alterado.

Em relação às ligações, ambos os periféricos têm ligações entre o processador Nios II e o módulo em si, destacando-se o facto de terem uma interface Conduit que possibilita o acesso ao *hardware* dos periféricos fora da ferramenta Platform Designer.

4.3.10 Interval Timer Intel FPGA IP

O módulo Timer utiliza *ticks* à frequência do processador Nios II para providenciar um valor temporal instantâneo. Para não sobrecarregar o sistema e o processador, a interrupção gerada a cada ciclo do Timer, deve ser a mais demorada possível. Assim, para este sistema, foi definida a interrupção do Timer a cada um segundo contabilizado pelo mesmo.

4.3.11 Sistema Final Qsys

A partir da figura 4.11 é possível observar o sistema em hardware produzido a partir da ferramenta Platform Designer.

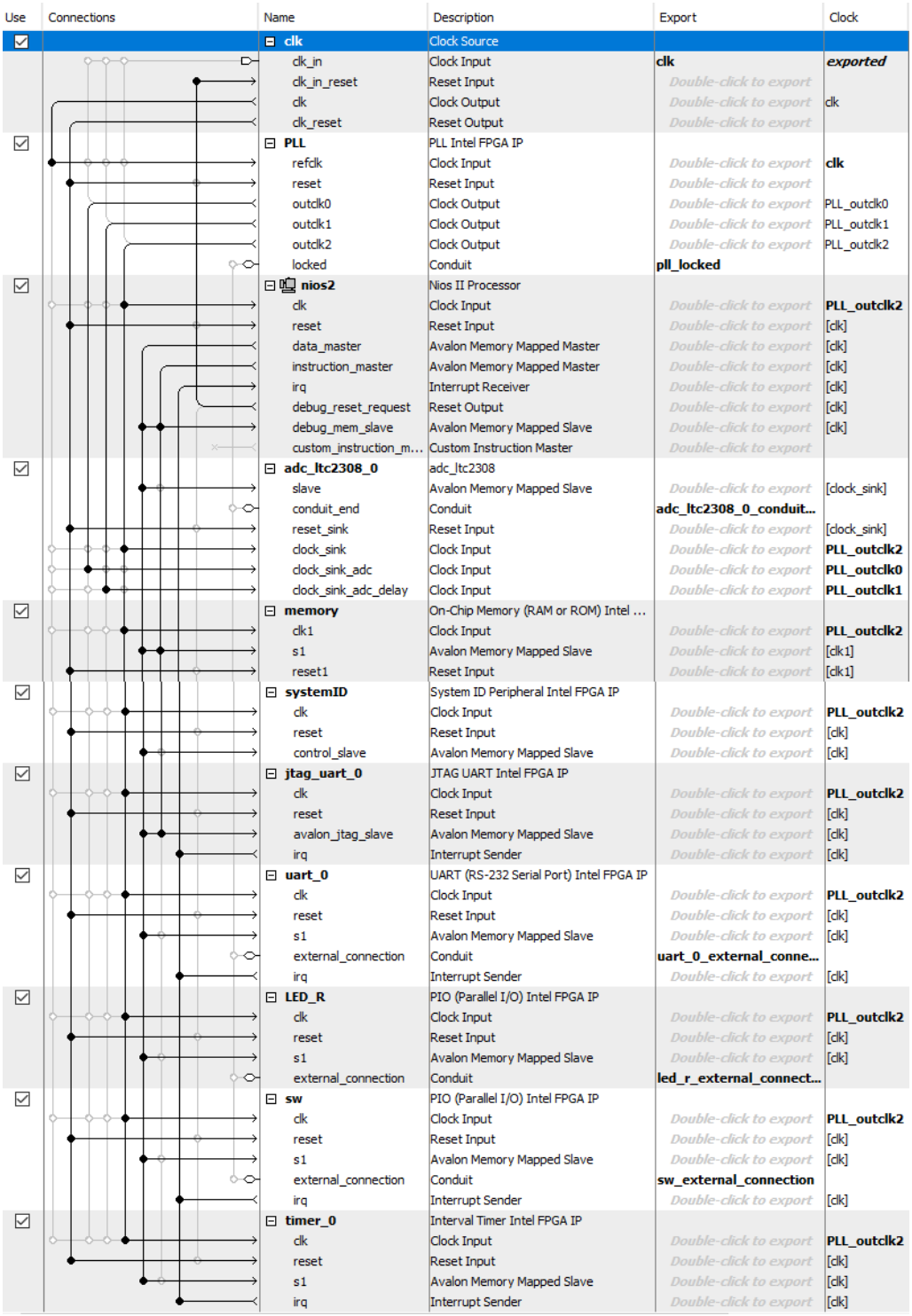

Figura 4.11: Todos os módulos e ligações utilizadas para a criação em hardware do sistema final

4.4 Hardware Abstraction

A fim de conseguir desenvolver em ambiente de FPGA de forma similar à de um microcontrolador, é necessário utilizar as ferramentas e bibliotecas presentes na Board Support Package (BSP). Depois de o sistema estar definido na parte de *hardware* e os requisitos estarem de acordo com o estabelecido, é necessário avançar para o desenvolvimento em *software*. Esse desenvolvimento é feito no IDE Eclispe em conjunto com as ferramentas providenciadas pela Embedded Design Suite (EDS).

Dentro da camada de desenvolvimento em *software* existe a camada da aplicação, que contém a lógica principal do sistema, e a camada de suporte, onde estão as ferramentas e bibliotecas necessárias para garantir o correcto funcionamento do desenvolvimento em software. Estas duas camadas posicionam-se em cima da camada de *hardware* num contexto hierárquico. A divisão das camadas de desenvolvimento é possível ser observada pela figura 4.12.

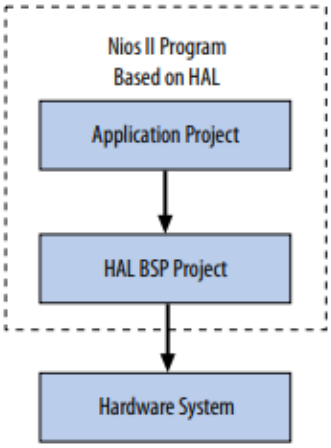

Figura 4.12: Representação do sistema em diversas camadas [6]

Sendo assim, depois da definição do sistema a nível da camada do *hardware*, é necessário verificar que ferramentas e/ou bibliotecas são necessárias da parte da camada de suporte, nomeadamente da Board Support Package (BSP).

4.4.1 Board Support Package (BSP)

Tal como referido no capítulo anterior, a BSP pode ser modificada de maneira a poder corresponder às necessidades do sistema. Em alguns casos, é a partir das configurações da BSP que é possível desbloquear o acesso a determinado *hardware* definido na ferramenta Platform Designer.

4.4.1.1 system.h

A primeira tarefa a fazer depois de ter o sistema de *hardware* feito, compilado e programado para o FPGA, é verificar se os dados do sistema em *software* são os mesmos que foram definidos em *hardware*. Dentro do ficheiro system.h estão todos os dados do módulos e respectivos

parâmetros definidos.

Assim, é possível verificar, por exemplo, as informações gerais provenientes da BSP em junção com o *hardware*:

```
#define ALT_DEVICE_FAMILY "Cyclone V"
#define ALT_ENHANCED_INTERRUPT_API_PRESENT
#define ALT_IRQ_BASE NULL
#define ALT_LOG_PORT "/dev/null"
#define ALT_LOG_PORT_BASE 0x0
#define ALT_LOG_PORT_DEV null
#define ALT_LOG_PORT_TYPE ""
#define ALT_NUM_EXTERNAL_INTERRUPT_CONTROLLERS 0
#define ALT_NUM_INTERNAL_INTERRUPT_CONTROLLERS 1
#define ALT_NUM_INTERRUPT_CONTROLLERS 1
#define ALT_STDERR "/dev/jtag_uart_0"
#define ALT_STDERR_BASE 0x81098
#define ALT_STDERR_DEV jtag_uart_0
#define ALT_STDERR_IS_JTAG_UART
#define ALT_STDERR_PRESENT
#define ALT_STDERR_TYPE "altera_avalon_jtag_uart"
#define ALT_STDIN "/dev/jtag_uart_0"
#define ALT_STDIN_BASE 0x81098
#define ALT_STDIN_DEV jtag_uart_0
#define ALT_STDIN_IS_JTAG_UART
#define ALT_STDIN_PRESENT
#define ALT_STDIN_TYPE "altera_avalon_jtag_uart"
#define ALT_STDOUT "/dev/jtag_uart_0"
#define ALT_STDOUT_BASE 0x81098
#define ALT_STDOUT_DEV jtag_uart_0
#define ALT_STDOUT_IS_JTAG_UART
#define ALT_STDOUT_PRESENT
#define ALT_STDOUT_TYPE "altera_avalon_jtag_uart"
#define ALT_SYSTEM_NAME "nios2adc_system"
```

Através deste excerto de código, é possível identificar o nome do sistema, a família do FPGA em uso, a porta série utilizada em *logs* (a qual está desactivada), o uso da interface JTAG no uso de Stdin, stdio e stderr, entre outros.

Utilizando agora o exemplo de um dos módulos de *hardwre* utilizados na criação do sistema na ferramenta Platform Designer, o módulo UART.

```
#define ALT_MODULE_CLASS_uart_0 altera_avalon_uart
#define UART_0_BASE 0x81040
```

```
#define UART_0_BAUD 115200
#define UART_0_DATA_BITS 8
#define UART_0_FIXED_BAUD 1
#define UART_0_FREQ 100000000
#define UART_0_IRQ 2
#define UART_0_IRQ_INTERRUPT_CONTROLLER_ID 0
#define UART_0_NAME "/dev/uart_0"
#define UART_0_PARITY 'N'
#define UART_0_SIM_CHAR_STREAM ""
#define UART_0_SIM_TRUE_BAUD 0
#define UART_0_SPAN 32
#define UART_0_STOP_BITS 1
#define UART_0_SYNC_REG_DEPTH 2
#define UART_0_TYPE "altera_avalon_uart"
#define UART_0_USE_CTS_RTS 0
#define UART_0_USE_EOP_REGISTER 0
```

Através das informações provenientes do excerto de código apresentado é possível obter o endereço do módulo (0x81040), o *baud rate* (115200), a frequência a que funciona (100MHz), o nome (/dev/uart_o), o número de bits de paridade, entre outros. É fundamental que todas estas definições estejam de acordo com o definido na camada de *hardware*, para todos os módulos.

4.4.1.2 io.h

O ficheiro io.h garante ao sistema uma forma abstrata de conseguir comandar a transmissão e recepção de dados através do endereço de cada módulo, caracteriza fundamental da *Interface Avalon Memory Mapped*. Para isso, esta biblioteca provediencia duas principais funções, `IORD`, para recepção, e `IOWR`, para transmissão.

```
#define IORD(BASE, REGNUM) \
__builtin_ldwio (__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM)))
#define IOWR(BASE, REGNUM, DATA) \
__builtin_stwio (__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM)), (DATA))
```

Como é possível observar pelo excerto de código, ambas recebem dois parâmetros em comum, BASE e REGNUM. BASE é o endereço do módulo e REGNUM é o *offset* referido em capítulos anteriores, na secção 3.4.2. No caso de IOWR, é enviado DATA como parâmetro, sendo os dados a transmitir.

4.4.1.3 altera_avalon_pio_regs.h

Com o intuito de facilitar o uso de Parallel I/O, a Intel desenvolveu um padrão a ser utilizado numa biblioteca com origem na utilização de io.h, e, cuja abstração é ainda maior.

Assim, é estabelecido que para o uso de *offset* 0, o uso da função IORD_ALTERA_AVALON_PIO_DATA, teria como objectivo o comando da transmissão e recepção de dados entre *Interfaces Avalon Memory Mapped*.

```
#define IORD_ALTERA_AVALON_PIO_DATA(base)             IORD(base, 0)
#define IOWR_ALTERA_AVALON_PIO_DATA(base, data)       IOWR(base, 0, data)
```

Em seguida, com o objectivo de modificar a direcção (input/output) da comunicação entre os módulos PIO, é usado a função IORD_ALTERA_AVALON_PIO_DIRECTION, sendo que neste caso o parâmetro *data*, tem como valores 0 para *input* e 1 para *output*.

```
#define IORD_ALTERA_AVALON_PIO_DIRECTION(base)        IORD(base, 1)
#define IOWR_ALTERA_AVALON_PIO_DIRECTION(base, data)  IOWR(base, 1, data)
```

Apesar destes dois tipos de padrão serem os mais utilizados, existem ainda mais quatro padrões providenciados pela biblioteca altera_avalon_pio_regs.h. E mesmo que estes padrões existam para ser utilizados, não quer dizer que todos os módulos os usem, como é referido mais adiante.

4.4.1.4 unistd.h

O uso da biblioteca unistd.h permite o acesso à Application Programming Interface (API) de sistema operativo UNIX, que faz parte do EDS.

4.4.1.5 stdio.h

Para além de adicionar tipos de variáveis ao sistema, stdio.h também adiciona algumas funções. No uso do sistema, o uso da função printf() só é possível através de stdio.

Por sua vez, stdio, stdout e stderr estão definidos pela interface JTAG do sistema. Esta configuração faz com que, na utilização da função printf(), inserida por stdio.h, existam uma simulação de entrada de dados no canal de comunicação e os dados sejam observados no terminal do IDE.

4.4.1.6 sys/alt_timestamp.h

A utilização das funcionalidades de alt_timestamp.h visam medições temporais com grande resolução. No entanto, a mesma tem de ser ativada através do editor de BSP, definindo o valor de hal.timestamp_timer com o nome dado ao módulo do timer, neste caso Timer_0, como é possível observar na figura 4.13.

4.4.2 LTC2308

Como referido anteriormente, o módulo não possui parâmetros na ferramenta Platform Designer, apesar de ter vários modos de operação, como é possível observar pela tabela 3.17. A fim de compreender a utilização das funções `IORD` ou `IOWR` no controlo do ADC LTC2308, é necessário compreender primeiro como funciona o seu hardware.

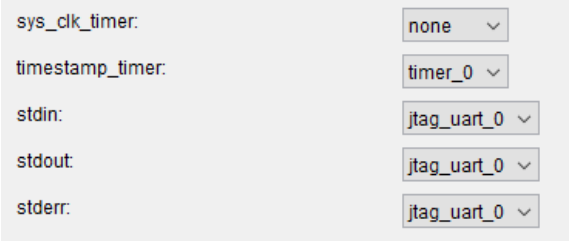

Figura 4.13: Configurações BSP

4.4.2.1 Transmissão

Dentro do funcionamento do ADC, para a transmissão, existem duas constantes, a de inicio de conversão e a de definição do número de conversões a guardar na fila FIFO.

```
‘define WRITE_REG_START_CH                    0
‘define WRITE_REG_MEASURE_NUM                 1

reg             measure_fifo_start;
reg [11:0]      measure_fifo_num;
reg [2:0]       measure_fifo_ch;
always @ (posedge slave_clk or negedge slave_reset_n)
begin
 if (~slave_reset_n)
  measure_fifo_start <= 1'b0;
 else if (~slave_chipselect_n && ~slave_wrtie_n && slave_addr ==
                                                ‘WRITE_REG_START_CH)
  {measure_fifo_ch, measure_fifo_start} <= slave_wriredata[3:0];
 else if (~slave_chipselect_n && ~slave_wrtie_n && slave_addr ==
                                                ‘WRITE_REG_MEASURE_NUM)
  measure_fifo_num <= slave_wriredata;
end
```

Estas duas constantes podem ser alcançadas através do *offset* do endereço. Assim, ao utilizar `IOWR(base, regnum, data)`, ambas as funcionalidades das constantes podem ser atingidas, modificando o valor de *offset*.

Para definir o número de conversões a que o ADC deverá guardar na fila FIFO, é utilizado `IOWR(base, 1, data)`, visto que 1 é a constante definida em WRITE_REG_MEASURE_NUM. Sendo assim, para dar inicio a uma conversão, o valor do *offset* deve ser 0, definido pela constante WRITE_REG_START_CH.

4.4.2.2 Recepção

A recepção possui uma lógica similar à da transmissão. Novamente existem duas funcionalidades distintas, uma interrupção que informa quando uma conversão é concluída e uma que permite obter os valores resultantes da conversão do ADC.

```
‘define READ_REG_MEASURE_DONE          0
‘define READ_REG_ADC_VALUE             1
wire slave_read_status;
wire slave_read_data;

assign slave_read_status = (~slave_chipselect_n && ~slave_read_n &&
slave_addr == ‘READ_REG_MEASURE_DONE) ? 1’b1 : 1’b0;
assign slave_read_data = (~slave_chipselect_n && ~slave_read_n &&
slave_addr == ‘READ_REG_ADC_VALUE) ? 1’b1 : 1’b0;

reg measure_fifo_done;
always @ (posedge slave_clk)
begin
 if (slave_read_status)
  slave_readdata <= {11’b0, measure_fifo_done};
 else if (slave_read_data)
  slave_readdata <= fifo_q;
end
```

Assim, para obter os valores correspondentes a uma conversão no ADC, os parâmetros a utilizar na função de leitura `IORD` devem corresponder a `IORD(base, 1)`, sendo 1 o valor correspondente à constante READ_REG_ADC_VALUE.

4.4.3 Aplicação

A aplicação pode ser criada de raiz ou a partir de um projecto padrão dado pelas Nios II SBT, por exemplo, hello_world.c. Existem outros exemplos que procuram outro tipo de implementações como RTOS, uso de *networks stacks* ou até redução da *foot-print* do sistema. No decorrer desta dissertação, é utilizado bibliotecas de uso genérico referidas anteriormente, como a biblioteca C providenciada pela EDS, bibliotecas HAL e drivers.

4.4.3.1 main()

O ficheiro que contém a função `main()` é onde se encontra toda a lógica pertencente à aplicação e é onde as bibliotecas referidas anteriormente são utilizadas.

Inicialmente, é necessário definir o canal de comunicação UART através do nome do módulo definido pelo *hardware* e observado no ficheiro system.h.

```
FILE* fp;
fp = fopen("/dev/uart_0","r++");
```

O código utilizado para o fazer é similar ao de uso comum no desenvolvimento em linguagem C e é este canal de comunicação que permitirá ao FPGA comunicar com equipamentos externos e enviar as conversões feitas pelo ADC.

Em seguida, é necessário definir o número de leituras/conversões que o ADC irá fazer e guardar na FIFO antes de começar a fazer as devidas leituras.

```
IOWR(ADC_LTC2308_0_BASE, WRITE_REG_MEASURE_NUM, nReadNum);
```

Esta configuração é conseguida utilizando a função `IOWR` para definir `nReadNum` como valor máximo de conversões guardadas na fila FIFO. Este valor não pode passar as 1024 conversões.

O sistema, de forma a obter medições a cada intervalo de tempo, é inserido num `while(1)`. Dentro deste ciclo infinito,

```
ch = IORD_ALTERA_AVALON_PIO_DATA(SW_BASE) & 0x0F;
```

`IORD_ALTERA_AVALON_PIO_DATA(base)` é responsável por receber o valor correspondente ao bit (ou bits) utilizado pelo PIO, que neste caso assume o papel de *switch*. Ainda neste uso de PIO, é utilizado funções definidas na biblioteca `altera_avalon_pio_regs.h`, referidas em secções anteriores, em vez do uso genérico de `IORD`.

Depois de definido o canal do ADc a utilizar para a conversão de valores analógicos à entrada do mesmo, é possível activar o começo da conversão no ADC. Referido em secções anteriores, a conversão no ADC LTC2308 começava quando a ligação CONVST tomava com um estado lógico alto durante um curto instante de tempo e logo a seguir assumia um estado lógico baixo. Para tal ser comandado a partir do *software* é necessário definir o estado lógico a partir da função `IOWR`, utilizando o endereço do ADC e o *offset* correspondente.

```
IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x00);
IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x01);
IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x00);
```

Utilizando como *offset* o mesmo valor da constante definida pelo *hardware* do módulo, é possível ativar a conversão do ADC.

Como o *hardware* proporciona uma *flag* que informa do estado da conversão (`READ_REG_MEASURE_DONE`), é possível utilizar essa funcionalidade para induzir o sistema num estado de espera até que o ADC tenha completado a conversão (ou conversões) para a qual foi configurado.

```
while ((IORD(ADC_LTC2308_0_BASE, READ_REG_MEASURE_DONE)
& 0x01) == 0x00);
```

No seguimento do término da conversão do ADC, é possível obter os dados provenientes dessa conversão. O número de vezes em que `IORD` com *offset* `READ_REG_ADC_VALUE` é invocado, é dependente do número de conversões definidas anteriormente por `nReadNum`.

```
for(int i = 0; i < nReadNum; i++)
 Value = IORD(ADC_LTC2308_0_BASE, READ_REG_ADC_VALUE);
```

4.5 Linha de comandos do Quartus Prime e Nios II

A fim de utilizar as capacidades de configuração e de programação do software Quartus Prime ou do IDE Eclipse na plataforma DINASORE, é necessário acabar com o desenvolvimento em GUI e passar a utilizar a linha de comandos para a configuração e programação do FPGA. As ferramentas que permitem esta configuração sem o uso de GUI já se encontram incluídas na instalação de ambos os softwares e, até existe a possibilidade de apenas e só utilizar a linha de comandos em todo o desenvolvimento do FPGA.

Tal como o desenvolvimento do FPGA nesta dissertação está dividido na camada de *hardware* e na camada de *software*, também o uso da linha de comandos se encontra limitada a essa separação. O uso das ferramentas do software Quartus Prime permitem o compilar e a programação do FPGA, mas não permitem o *build* nem *run* de aplicações para o processador Nios II. Do lado do software Quartus Prime, é possível utilizar as suas ferramentas para compilar e programar o FPGA. Do lado das Nios II SBT é possível dar *build* e *run* a aplicações/programas para o Nios II.

Na instalação do software Quartus Prime são definidas *Environment Variables* que definem o directório das principais ferramentas do Quartus Prime e do Nios 2 EDS, pelo que não é necessário usar o directório completo para aceder a uma desta ferramentas.

4.5.1 Quartus Shell

Começando pelas ferramentas do Quartus Prime, `quartus_sh` utiliza o Quartus Shell e permite compilar o *hardware* projectado até então.

```
quartus_sh --flow compile <top file>
```

O parâmetro `<top file>` é substituído pelo ficheiro principal na hierarquia do projecto.

4.5.2 Quartus Programmer

Depois de compilar o projecto, é possível programar o mesmo para o FPGA. Para isso, é necessário utilizar a ferramenta Quartus Programmer, a qual pode ser acedida através do executável `quartus_pgm`.

```
quartus_pgm -c USB-Blaster -m jtag -o p;<path to .sof file>
```

Como parâmetros, deve ser definido o tipo de ligação a utilizar, a interface e o ficheiro gerado pela compilação.

4.5.3 Nios II Command Shell

Para utilizar as ferramentas do Nios II EDS, os comandos são utilizados agora em ambiente Linux, através de Windows Subsystem for Linux (WSL). Sendo assim, diferente do uso anterior do Windows Powershell, é possível ter acesso a comandos utilizados em sistemas UNIX. No entanto, para aceder às ferramentas do Nios II EDS, é necessário fazê-lo a partir do executável Nios II Command Shell [27] [28].

Depois de compilar e programar o FPGA, a primeira tarefa a executar é actualizar a Board Support Package (BSP). É possível fazê-lo através do executável `nios2-bsp-generate-files.exe`.

```
Nios II Command Shell.bat nios2-bsp-generate-files.exe
--bsp-dir <path_to_bsp_files> --settings <path_to_settings.bsp>
```

Esta ferramenta tem como parâmetros o directório onde se encontra a pasta correspondente à BSP do projecto, e a localização do ficheiro com as configurações da BSP.

Depois de actualizar o pacote de suporte, é possível avançar com o *build* da aplicação através do comando comum `make`. Para tal, é preciso novamente invocar o Nios II Command Shell e, em seguida, definir o directório a executar o comando `make`.

```
Nios II Command Shell.bat make all –C <path to software project folder>
```

Por fim, na sucessão do sucesso da actualização da BSP e do *build* da aplicação, é exequível que o projecto leve *run* para o processador Nios II. Para tal, é necessário recorrer à ferramenta `nios2-download`.

```
Nios II Command Shell.bat nios2-download –g <path to .elf file>
```

O ficheiro .elf criado depois do projecto ser *build* é usado agora como parâmetro na ferramenta `nios2-download`. Adicionalmente, `-g` permite que o processador ative depois de ser programado.

4.6 Python *Scripts*

Parte do principio da plataforma DINASORE é baseada no uso de *scripts* feitos na linguagem de programação Python a fim fornecer funcionalidades aos *functions blocks* utilizados pela mesma. Assim, para transitar do processo de configuração e programação do FPGA para a plataforma, os comandos, anteriormente definidos, são envolvidos dentro de um ou mais *scripts*, para depois serem utilizados por *function blocks* com o intuito de usar o FPGA.

Para correr comandos no terminal usando a linguagem de programação Python, existem várias bibliotecas disponíveis. No entanto, a fim de transitar a configuração do FPGA para a automação via *script*, utiliza-se a biblioteca `subprocess`. Não só esta biblioteca é recente, como também é confiável a ponto de ser utilizada em ambiente Windows.

É possível correr qualquer comando no Power Shell do Windows através do excerto de código a baixo.

```
subprocess.run(['cmd'], shell = True)
```

O argumento `shell = True`, é obrigatório em ambiente Windows, caso contrário pode ser omitido.

Sendo assim, todo o processo de cada ferramenta referida na secção 4.5 pode ser facilmente transferido para um conjunto de linhas de código, utilizando a biblioteca `subprocess`.

```
import subprocess

## QUARTUS SIDE
# Compiles project
compileFPGA = subprocess.run([QUARTUS_SH, '--flow' ,
'compile', ADC_TOP_PATH], shell = True)

# Program FPGA
programFPGA = subprocess.run([QUARTUS_PGM, '-c' ,
'USB-Blaster', '-m', 'jtag', '-o', 'p;'+ ADC_SOF], shell = True)

## NIOS2 SIDE
# Update BSP and sent settings
update_bsp = subprocess.run([NIOS2SHELL,
'nios2-bsp-generate-files.exe',
'--bsp-dir', ADC_BSP, '--settings', SETTINGS_BSP], shell = True)

# Build Makefile
build = subprocess.run([NIOS2SHELL, 'make', 'all', '-C',
WSL_PATH + MAKEFILE], shell = True)

# Run
run = subprocess.run([NIOS2SHELL,'nios2-download','-g',
ADC_ELF], shell = True)
```

4.7 Dinasore

A plataforma Dinasore utiliza o IDE Eclipse para poder ter as suas funcionalidades representadas graficamente. O IDE Eclipse é uma das interfaces proporcionadas pelo 4diac, *Open Source PLC Framework for Industrial Automation & Control*, que é composto por mais 3 interfaces: ambiente de desenvolvimento (FORTE), bibliotecas de funções de blocos e projectos exemplos.

A plataforma Dinasore tem o papel de substituir a interface FORTE, responsável pelo funcionamento da ferramenta 4diac, e adicionar alguns *function blocks* à biblioteca já existente. Visto

que a plataforma Dinasore já se encontra em desenvolvimento, o objectivo passa por transformar os *scripts* feitos até então em *function blocks* que podem ser utilizados pela plataforma.

4.7.1 Function Blocks

Cada *function block* é composto por dois ficheiros: um em Python e outro em XML. O ficheiro em Python contém a lógica do *function block* tendo em conta a *framework* da ferramenta 4diac, e o ficheiro XML contém as entradas e saídas, e respectivas variáveis, que são responsáveis pelo aspecto gráfico do bloco. O processo de utilização de *function blocks* é possível ser observado através da figura 4.14.

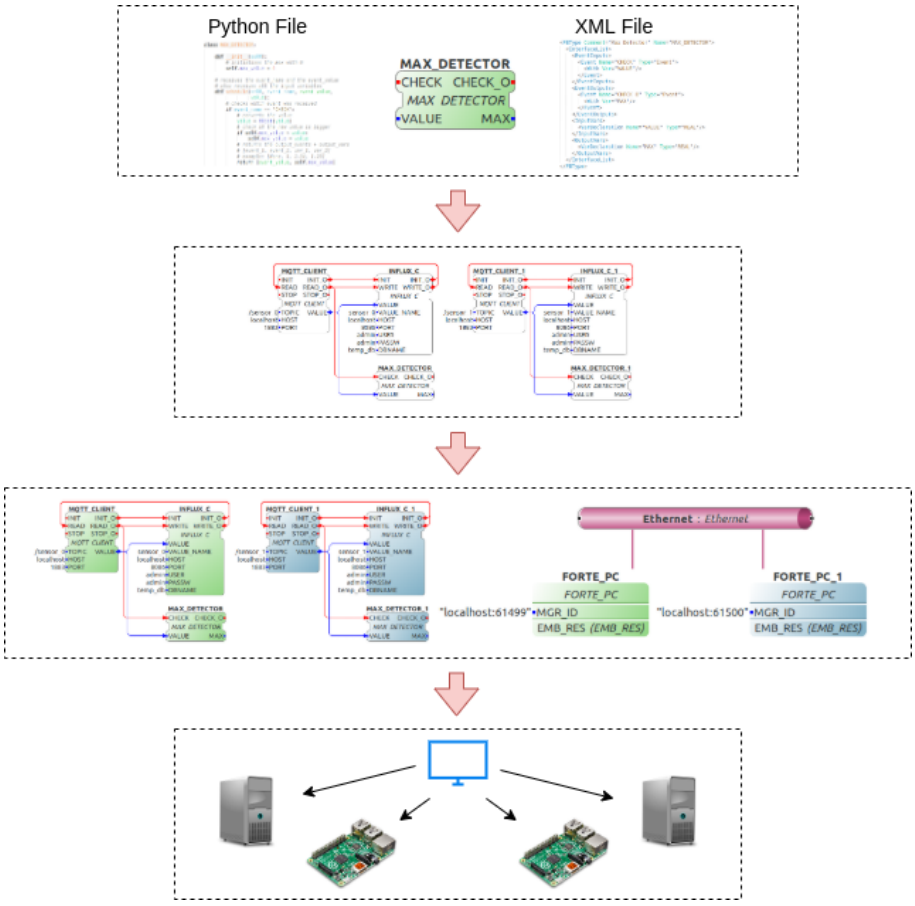

Figura 4.14: Processo de utilização de *function blocks*

Depois de ambos os ficheiros, Python e XML, estarem concluídos, é possível, através do IDE Eclipse, utilizá-los graficamente e inserir as devidas conexões entre *function blocks* existentes. Em seguida, após as conexões estarem devidamente estabelecidas, é necessário conectar o sistema a um dispositivo, por exemplo, um computador. Quando todas os últimos requisitos estiverem preenchidos e a funcionar correctamente, é plausível avançar para a activação do sistema.

Cada *function block* tem um tipo e, consequentemente, uma função. Na plataforma Dinasore existem quatro tipos diferente de *function block*. Um bloco do tipo `DEVICE.SENSOR` representa um sensor, um do tipo `SERVICE` simboliza um módulo de processamento, um do tipo `POINT.STARTPOINT` representa um protocolo que recebe informação, e, por fim, um bloco do tipo `POINT.ENDPOINT` representa um protocolo que envia informação.

4.7.2 Transição

Para efectuar a transição do desenvolvimento em FPGA via Quartus Prime para 4diac utilizando Dinasore, o sistema foi dividido em três fases que se podem representar por três *function blocks* (figura 4.15).

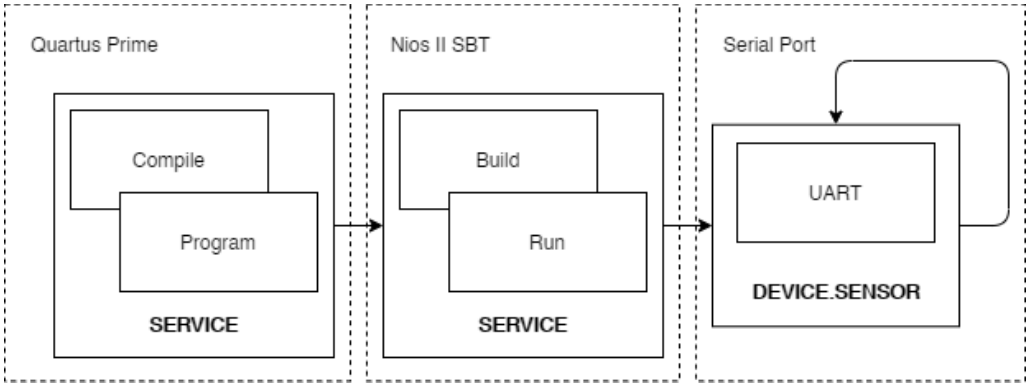

Figura 4.15: Divisão de tarefas em *function blocks*

Começando pelas ferramentas do Quartus Prime, o FPGA é compilado e programado. Em seguida, através das ferramentas do Nios II EDS o software é *build* e *run*. Estes dois *function blocks* são do tipo `SERVICE`, pelo que não se repetem. Sendo assim, no final do segundo bloco o FPGA encontra-se funcional e com a devida aplicação a executar. Visto que o *hardware* e o *software* estão configurados para a utilização da comunicação UART, um último bloco é adicionado ao sistema a fim de receber valores provenientes do FPGA com as conversões do ADC. Este bloco é do tipo `DEVICE.SENSOR` e é executado num ciclo infinito, estando conectado ao FPGA por uma ligação série. Sendo assim, a cada ciclo de execução do último bloco, está representado à saída o valor que corresponde à conversão do ADC controlado pelo FPGA.

4.7.3 Regras da framework

Para a criação dos *function blocks* é necessário seguir *guidelines* que permitem o funcionamento dos blocos dentro do sistema 4diac.

4.7.3.1 Python

Entre o ficheiro Python e o XML, ambos têm que ter a mesma identificação. No caso do ficheiro Python, a identificação é feita pela `class`, e no caso do ficheiro XML, é feito através

do parâmetro `Name`. Assim, no caso de identificar um novo bloco com o nome `NovoBloco`, o ficheiro Python teria que ter `class NovoBloco` e o ficheiro XML `Name=NovoBloco`. A estrutura obrigatório do ficheiro Python é constituída por uma função `schedule`. Cada variavél à entrada do tipo `Events` tem de constituir uma condição dentro da função `schedule`. Para alé de `self`, `event_input_name` e `event_input_value`, a função `schedule` recebe como parametro qualquer variável à entrada, independente do seu tipo. Um exemplo de um ficheiro Python constituinte de um *function block* pode ser observado no excerto de código seguinte:

```
class FB_TYPE_EXAMPLE:
 VAR2_EXAMPLE = 0
 def schedule(self, event_input_name, event_input_value,
CONST_EXAMPLE, VAR1_EXAMPLE):
  if event_input_name == 'INIT':
    self.VAR2_EXAMPLE = CONST_EXAMPLE
    return [event_input_value, None, self.VAR2_EXAMPLE]

  elif event_input_name == 'RUN':
    self.VAR2_EXAMPLE += VAR1_EXAMPLE
    return [None, event_input_value, self.VAR2_EXAMPLE]
```

4.7.4 Uso e implementação em 4diac

Os três *function blocks* anteriormente referidos são modificados de acordo com as regras e requisitos, a fim de fazerem parte da biblioteca de *function blocks* do 4diac e poderem ser usados para obter os valores das conversões do ADC. O código utilizado para a criação dos três *function blocks* está presente no anexo A.

Dentro do IDE Eclipse do 4diac, é possível, depois de importados, utilizar os novos *function blocks*. Depois de as conexões estarem feitas, o sistema terá uma representação similar à da figura 4.16.

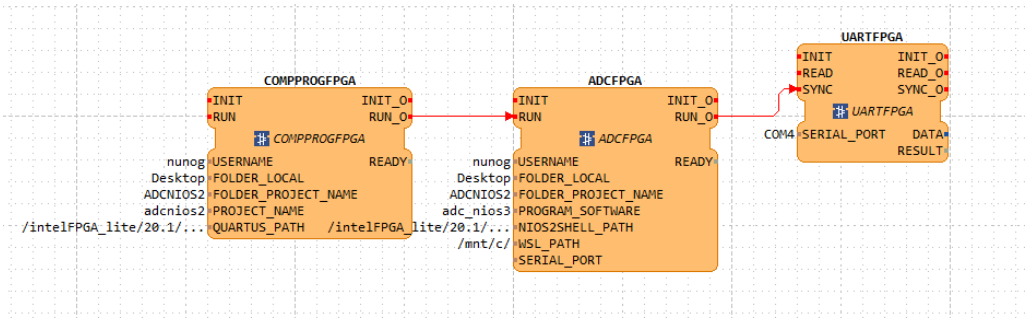

Figura 4.16: Representação do sistema em 4diac

Capítulo 5

Resultados

5.1 Aceleração de conversão de dados

Os primeiros dados relativos ao uso do FPGA para controlar e configurar o ADC, superam as expectativas. Devido à aceleração imposta pelo hardware formado pela fila FIFO, foi possível obter um tempo de conversão similar ao definido pelo *datasheet*. No *datasheet* do ADC LTC2308 é referido que o tempo máximo de conversão do ADC é de 1.6μs , enquanto que o tempo regular em que ele executa uma conversão é por volta dos 1.3μs.

Como é possível observar pelo gráfico da figura 5.1, o desempenho deste sistema é tanto maior, quanto maior for o número de conversões feitas e guardadas numa fila FIFO pelo ADC.

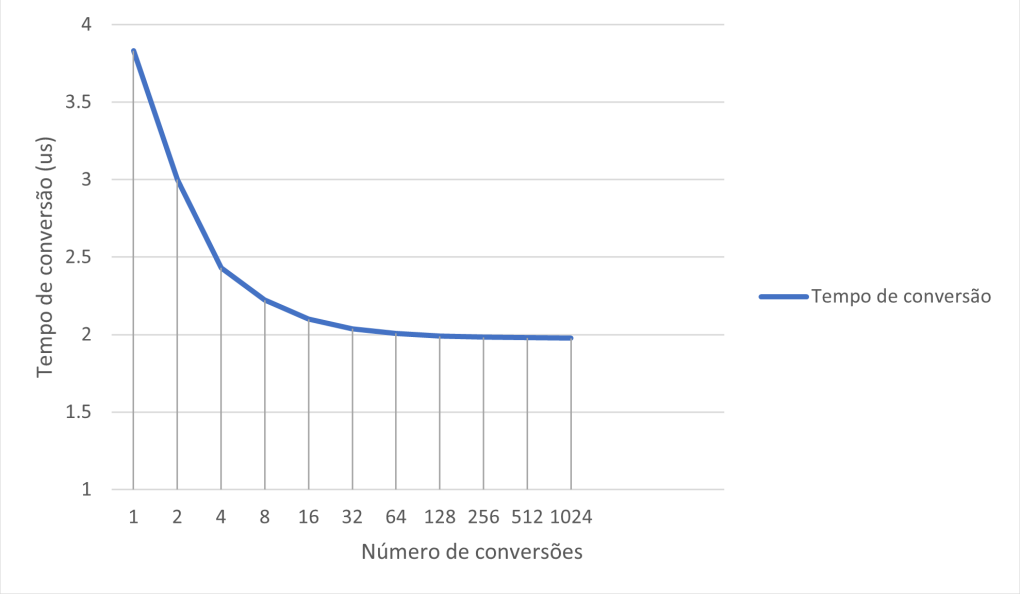

Figura 5.1: Resultado da aceleração de dados proveniente do uso de FIFO.

A partir das 64 até o máximo de 1024 conversões, o tempo de conversão tende a manter-se constante para o valor de 2μs, perto dos 1.3μs esperados. No entanto, para um número de

conversões menor que 64, os resultados não são constantes. Para o caso de uma só conversão, o tempo da mesma, 3.83μs, pode chegar a quase o triplo face ao resultado esperado de 1.3μs.

Sendo assim, conclui-se que a fila FIFO tem um papel fundamental para acelerar o processo de conversão de dados do ADC, atingindo o melhor desempenho na sua capacidade máxima de 1024 conversões.

Considerando que o desempenho máximo do sistema seria dado pelo tempo de conversão real igual ao esperado, pode dizer-se que para o uso do ADC sem *hardware* de aceleração e no caso de uma só conversão, o *design* teria um rendimento de $n = \frac{1.3}{3.83} * 100 = 33.94\%$, enquanto que com o uso do *hardware* de aceleração e no caso de 1024 conversões, o design teria um desempenho de $n = \frac{1.3}{1.976} * 100 = 65.7\%$. A aceleração proveniente do *hardware* proporciona ao *design* o dobro do desempenho em métricas temporais.

5.2 FPGA e microcontroladores populares no uso em IoT

A fim de obter outra perspectiva mais actual e focado no mundo real, o FPGA e a sua interface com o ADC, são testados face a placas de desenvolvimento populares no mundo do sistemas embarcados e da Internet Of Things (IoT), como é o caso da placa de desenvolvimento Arduino Uno e do NodeMCU. Nesta secção ADC LTC2308 é testado para o uso de entre um dos microcontroladores disponiveis, o Arduino Uno ou o NodeMCU, tendo como intermediário o FPGA. Neste caso, o FPGA apenas define as ligações entre o ADC LTC2308 e o exterior, a fim de ser possível conectar ao microcontrolador utilizado.

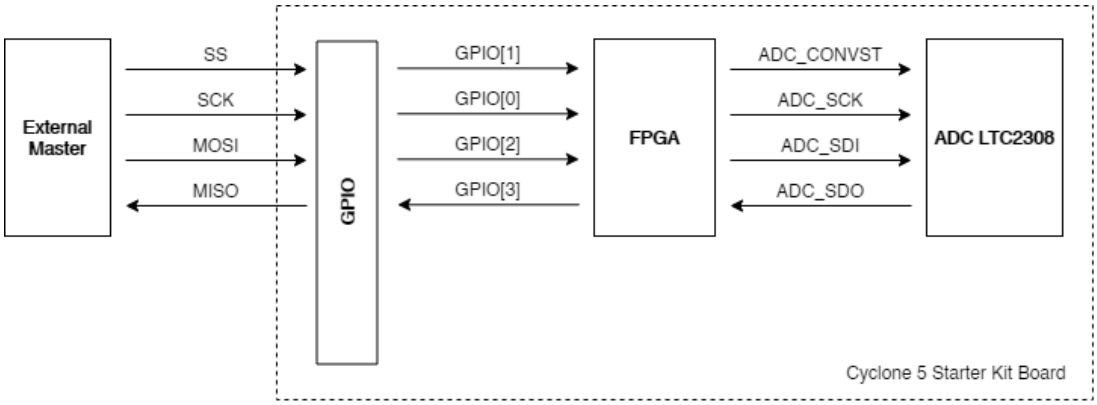

Figura 5.2: Cenário de teste

Como é possível observar pela figura 5.2, o microcontrolador utilizado tem o papel de *master* do sistema e utiliza a interface SPI comum directamente ligada aos pinos GPIO da placa de desenvolvimento do FPGA. Em seguida, o FPGA redirecciona os sinais nestes GPIO directamente para o ADC, sintetizando em *hardware* apenas as interligações necessárias para o efeito. Esta ligação directa entre os GPIO e o ADC é possível ser facilmente obtida através do seguinte excerto de HDL:

```
assign ADC_SCK = GPIO[0];
assign ADC_CONVST = GPIO[1];
assign ADC_SDI = GPIO[2];

assign GPIO[3] = ADC_SDO;
```

Este excerto deve ser compilado e programado para o FPGA antes configurar o microcontrolador.

Do lado do microcontrolador, ambos usam a *framework* Arduino, e, portanto, funções como `analogRead()` são comuns aos mesmos. Para testar o correto funcionamento do ADC, é conectado um sensor Light Dependent Resistor (LDR) em série com uma resistência de *pull-up*. O circuito divisor de tensão usa como fonte de tensão, como GND e como canal analógico, os pinos respectivos provenientes do *Arduino header* existente na placa Cyclone V Starter kit.

A frequência do sinal de relógio da interface SPI `SCK` é de 40Mhz e é utilizado em ambos os microcontroladores.

5.2.1 NodeMCU

Utilizando o ESP8266 e a biblioteca SPI providenciada pela *framework* Arduino é possível comunicar com o ADC. Para este escolha de microcontrolador, cada conversão demora cerca de $6\mu s$, ficando bastante a cima do valor estabelecido no *datasheet*, $1.3\mu s$.

5.2.2 Arduino Uno

Desta vez, utilizando o Arduino os resultados temporais foram menos constantes face aos obtidos no NodeMCU. Com o Arduino Uno, cada conversão é feita entre $4us$ e $8us$, ficando, igualmente longe do valor pretendido de $1.3us$.

Apesar do microcontrolador do NodeMCU funcionar a uma frequência padrão 10 vezes superior ao do microcontrolador do Arduino Uno, não existe uma correlação entre a velocidade de funcionamento e os resultados obtidos. Assim, e devido à semelhança dos valores obtidos, é possível afirmar que os requisitos temporais dependem maioritariamente da interface SPI e da restantes interligações feitas pelo FPGA.

5.3 LTC2308 e ADCs integrados

Na perspectiva da conveniência, muitos microcontroladores possuem um ADC presente dentro do *chip*. No entanto, este ADC não costuma ser tão preciso nem tão rápido face a um ADC externo. Face a esse conveniência, esta secção visa testar os ADCs presentes nas placas de desenvolvimento até agora referidas.

Em ambos os microcontroladores a função `analogRead` é utilizada e o tempo é medido em microsegundos.

5.3.1 ADC do ESP8266

No ESP8266, utilizando apenas a função `analogRead` no canal único do ADC, o tempo de conversão é entre $42\mu s$ e $47\mu s$.

5.3.2 ADC do Arduino

No Arduino, novamente utilizando apenas a função `analogRead` no canal 0 do ADC(A0), o tempo de conversão é entre $112\mu s$ e $116\mu s$.

5.4 Resultados finais

Observando a figura 5.3, é possível referir que a implementação do FPGA Cyclone V e do ADC LTC2308 foi a que obteve melhores resultados, mesmo sem a aceleração do *hardware*. Em seguida, ambos os microcontroladores usados para controlar e configurar o ADC LTC2308 também tiveram resultados similares, com especial atenção para o caso do ESP8266 que se manteve constante ou longo do tempo e dos testes efectuados.

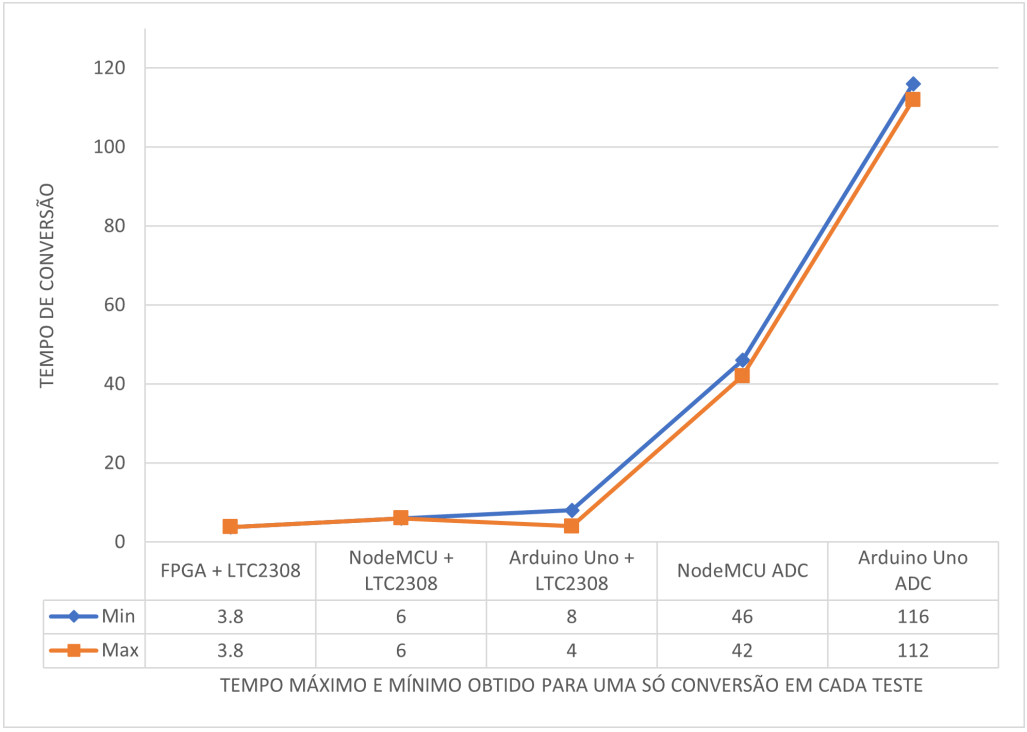

Figura 5.3: Representação dos resultados finais em μs

Por fim, é possível também observar, que, para uma utilização, *out-of-the-box*, o ADC do ESP8266 teria melhores resultados temporais face ao do Arduino Uno.

Capítulo 6

Conclusão

Esta dissertação tem como objectivo a implementação do maior número de sensores possível através da configuração e controlo do FPGA presente na placa de desenvolvimento usada. Com o correto uso do ADC LTC2308 e o correto design entre o mesmo e o FPGA, foi possível criar um sistema, o qual permite inserir qualquer tipo de sensor, desde que o mesmo tenha uma saída analógica. Assim, qualquer sensor pode ser utilizado, se o mesmo apresentar à saída um valor de tensão, ou esteja presente num circuito de divisor de tensão, como é o caso de sensores resistivos. Esse sensor é então conectado ao canal correspondente do ADC LTC2308 e os dados podem ser observados através da plataforma DINASORE.

A implementação do *design* feito para permitir o uso do FPGA com o ADC LTC2308, é adaptado à utilização a partir da linha de comandos usando Python e depois dividido em *function blocks* que são utilizados pela plataforma Dinasore. Todo o processo de transição entre os *scripts* para a sua devida utilização na plataforma Dinasore foi desenhado tendo em conta a independência dos mesmos blocos. Ou seja, os três *function blocks* estão desenhados de maneira a que possam ser utilizados para o seu propósito (compilar, programar, *build*, *run* e uso de UART) para qualquer que seja o projecto ou objectivo. Este desenho independente, permite que os blocos possam ser reutilizados em outro tipo de projectos que utilizem a mesma tecnologia e as mesmas ferramentas.

Ao longo da dissertação existiram vários problemas com o uso dos diversos *softwares*, como é o caso do Quartus Prime e do Eclipse com a Nios II EDS. Este problemas atrasaram significativamente a duração da dissertação e, consequentemente, o trabalho nela aplicado. Todos estes problemas são relevantes e, portanto, encontram-se detalhados no Anexo B.

6.1 Discussão de resultados

Com o objectivo inicial cumprido, é necessário comprovar se a utilização de um ADC através do controlo proveniente do FPGA é benéfico face às vantagens e desvantagens da tecnologia. Apesar do uso do ADC LTC2308 com qualquer microcontrolador ter uma fase de implementação mais directa e que consome menos tempo, os resultados apresentados no capítulo anterior mostram que existe um *tradeoff*. Ao depender mais do *software* acresce que o sistema tenha resultados

temporais mais fracos, face aos resultados apresentados pelo controlo do FPGA. Mesmo sem a aceleração do *hardware*, o FPGA mostrou-se ser o *master* com melhores resultados temporais. Obteve como resultado de uma só conversão, $3.81\mu s$, valor bastante satisfatório face aos $6\mu s$ e $8\mu s$ resultantes do controlo proveniente dos microcontroladores.

Numa utilização *out-of-the-box*, os resultados obtidos a partir dos microcontroldores são satisfatórios. Porém, para uma implementação que vise a sua aplicação na indústria ou outras áreas que necessitem de um grande processamento de dados, o FPGA mostra-se como o candidatado a ser utilizado pelas suas características e resultados provenientes da aceleração do *hardware* para a conversão de dados. Mesmo que os microcontroladores tenham uma frequência de funcionamento superior ao FPGA, é difícil que, usando apenas *software* consigam obter a mesma aceleração de processamento de dados que o FPGA obteve a partir de *hardware*.

6.2 Proposta de melhoria

O sistema cumpre o objectivo. No entanto, existem algumas propostas de melhoria que fariam o sistema ter um melhor desempenho ou cumprir um maior número de novos requisitos.

6.2.1 Controlo

O controlo do ADC apenas é feito, ou pelo FPGA por completo, ou pelos microcontroladores por completo. Um melhoria nos resultados apresentados, passaria pelo modificação das Nios II SBT pelo controlo *software* do Arduino Uno. Assim, o FPGA manteria o controlo do ADC através do *hardware*, mas a camada a cima, a de *software*, seria controlada via Arduino. Esta implementação poderia facilitar o controlo do ADC através do uso da famosa *framework* do Arduino e obter resultados similares aos resultados obtidos a partir do controlo executado pelo FPGA.

Anexo A

Código

A.1 *Design* do sistema

```
`define ENABLE_GPIO

module baseline_c5gx(

        //////// ADC ////////  1.2 V ///////
        output              ADC_CONVST,
        output              ADC_SCK,
        output              ADC_SDI,
        input               ADC_SDO,

        //////// AUD ////////  2.5 V ///////
        input               AUD_ADCDAT,
        inout               AUD_ADCLRCK,
        inout               AUD_BCLK,
        output              AUD_DACDAT,
        inout               AUD_DACLRCK,
        output              AUD_XCK,

        //////// CLOCK ////////
        input               CLOCK_125_p,  ///LVDS
        input               CLOCK_50_B5B,  ///3.3-V LVTTL
        input               CLOCK_50_B6A,
        input               CLOCK_50_B7A,  ///2.5 V
        input               CLOCK_50_B8A,

        //////// CPU ////////
```

```
      input              CPU_RESET_n, ///3.3V LVTTL

`ifdef ENABLE_DDR2LP
      //////// DDR2LP ///////// 1.2-V HSUL ///////
      output      [9:0]  DDR2LP_CA,
      output      [1:0]  DDR2LP_CKE,
      output             DDR2LP_CK_n, ///DIFFERENTIAL 1.2-V HSUL
      output             DDR2LP_CK_p, ///DIFFERENTIAL 1.2-V HSUL
      output      [1:0]  DDR2LP_CS_n,
      output      [3:0]  DDR2LP_DM,
      inout       [31:0] DDR2LP_DQ,
      inout       [3:0]  DDR2LP_DQS_n, ///DIFFERENTIAL 1.2-V HSUL
      inout       [3:0]  DDR2LP_DQS_p, ///DIFFERENTIAL 1.2-V HSUL
      input              DDR2LP_OCT_RZQ, ///1.2 V
`endif /*ENABLE_DDR2LP*/

`ifdef ENABLE_GPIO
      //////// GPIO ///////// 3.3-V LVTTL ///////
      inout       [35:0] GPIO,
`else
      //////// HEX2 ///////// 1.2 V ///////
      output      [6:0]  HEX2,

      //////// HEX3 ///////// 1.2 V ///////
      output      [6:0]  HEX3,


`endif /*ENABLE_GPIO*/

      //////// HDMI /////////
      output             HDMI_TX_CLK,
      output      [23:0] HDMI_TX_D,
      output             HDMI_TX_DE,
      output             HDMI_TX_HS,
      input              HDMI_TX_INT,
      output             HDMI_TX_VS,

      //////// HEX0 /////////
      output      [6:0]  HEX0,
```

```
      //////// HEX1 ////////
      output      [6:0]   HEX1,


      //////// HSMC ////////  2.5 V //////
      input               HSMC_CLKIN0,
      input       [2:1]   HSMC_CLKIN_n,
      input       [2:1]   HSMC_CLKIN_p,
      output              HSMC_CLKOUT0,
      output      [2:1]   HSMC_CLKOUT_n,
      output      [2:1]   HSMC_CLKOUT_p,
      inout       [3:0]   HSMC_D,
`ifdef ENABLE_HSMC_XCVR
      input       [3:0]   HSMC_GXB_RX_p,  /// 1.5-V PCML
      output      [3:0]   HSMC_GXB_TX_p,  /// 1.5-V PCML
`endif /*ENABLE_HSMC_XCVR*/
      inout       [16:0]  HSMC_RX_n,
      inout       [16:0]  HSMC_RX_p,
      inout       [16:0]  HSMC_TX_n,
      inout       [16:0]  HSMC_TX_p,


      //////// I2C ////////  2.5 V //////
      output              I2C_SCL,
      inout               I2C_SDA,

      //////// KEY ////////  1.2 V //////
      input       [3:0]   KEY,

      //////// LEDG ////////  2.5 V //////
      output      [7:0]   LEDG,

      //////// LEDR ////////  2.5 V //////
      output      [9:0]   LEDR,

`ifdef ENABLE_REFCLK
      //////// REFCLK ////////  1.5-V PCML //////
      input               REFCLK_p0,
      input               REFCLK_p1,
`endif /*ENABLE_REFCLK*/
```

```
      //////// SD ///////// 3.3-V LVTTL ///////
      output             SD_CLK,
      inout              SD_CMD,
      inout       [3:0]  SD_DAT,

`ifdef ENABLE_SMA
      //////// SMA ///////// 1.5-V PCML ///////
      input              SMA_GXB_RX_p,
      output             SMA_GXB_TX_p,
`endif /*ENABLE_SMA*/

      //////// SRAM ///////// 3.3-V LVTTL ///////
      output      [17:0] SRAM_A,
      output             SRAM_CE_n,
      inout       [15:0] SRAM_D,
      output             SRAM_LB_n,
      output             SRAM_OE_n,
      output             SRAM_UB_n,
      output             SRAM_WE_n,

      //////// SW ///////// 1.2 V ///////
      input       [9:0]  SW,

      //////// UART ///////// 2.5 V ///////
      input              UART_RX,
      output             UART_TX


);

wire ADC_SCK_DELAY;


nios2adc_system u0 (
 .clk_clk                              (CLOCK_50_B5B),
 .led_r_external_connection_export     (LEDR),
 .sw_external_connection_export        (SW),
 .adc_ltc2308_0_conduit_end_CONVST     (ADC_CONVST),
 .adc_ltc2308_0_conduit_end_SCK        (ADC_SCK),
```

```
  .adc_ltc2308_0_conduit_end_SDI          (ADC_SDI),
  .adc_ltc2308_0_conduit_end_SDO          (ADC_SDO),
  .adc_ltc2308_0_conduit_end_SCK_DELAY (ADC_SCK_DELAY),
  .uart_0_external_connection_rxd         (UART_RX),
  .uart_0_external_connection_txd         (UART_TX)
);

endmodule
```

A.2 Sistema em C

```
#include <stdio.h>
#include <io.h>
#include <unistd.h>
//#include <sys/alt_alarm.h>
//#include <sys/alt_timestamp.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"

#define MILI 1000
#define MICRO 1000000

// Write
#define WRITE_REG_START_CH 0
#define WRITE_REG_MEASURE_NUM 1

// Read
#define READ_REG_MEASURE_DONE 0
#define READ_REG_ADC_VALUE 1

//unsigned int time1;
//unsigned int time2;
unsigned int count;

int main()
{

        int ch;
```

```
const int nReadNum = 1024; // max 1024
int Value, nIndex=0;

FILE* fp;
fp = fopen("/dev/uart_0","r++");


if(fp)
        printf("fp opens!\r\n");
else
        printf("Error opening the UART!");

/*if (alt_timestamp_start() < 0)
{
        printf ("No timestamp device available\n");
} else {

}*/

IOWR(LED_R_BASE, 0, 0x02);

//Set measure number for ADC to convert
//This write is done with 4 bytes offset
//#define ADC_LTC2308_0_BASE 0x41020 + 0x04(1word 32 bit) = 0x41024
IOWR(ADC_LTC2308_0_BASE, WRITE_REG_MEASURE_NUM, nReadNum);

//NOTE: when using only the terminal, printf MUST be commented out.
//If its not, JTAG buffer increases, crashing the system.

while(1) {

 count = 0;
 //Read adc channel from switch value
 ch = IORD_ALTERA_AVALON_PIO_DATA(SW_BASE) & 0x0F;

 printf("====================== %d, ch=%d\r\n", nIndex++, ch);

 //Start measure
 //Writes 1 into base address
 //#define ADC_LTC2308_0_BASE 0x41020
```

```
            IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x00);
            IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x01);
            IOWR(ADC_LTC2308_0_BASE, WRITE_REG_START_CH, (ch << 1) | 0x00);

            //starts timer
            //time1 = alt_timestamp();
            // wait measure done
            // wait until it receives a one
            while ((IORD(ADC_LTC2308_0_BASE, READ_REG_MEASURE_DONE)
            & 0x01) == 0x00);

            //time2 = alt_timestamp();

            //time in seconds
            //double f = (double)(time2-time1)/alt_timestamp_freq();

            //Change MILI or MICRO
            //f = f * MICRO;
            //printf("Measurement time: %f at frequency of %u\r\n",
            //f, alt_timestamp_freq());

            //From -> IOWR(ADC_LTC2308_0_BASE, WRITE_REG_MEASURE_NUM,
            //nReadNum);
            //ADC measures are stored in a FIFO nReadNum times
            for(int i = 0; i < nReadNum; i++)
            {
             Value = IORD(ADC_LTC2308_0_BASE, READ_REG_ADC_VALUE);
             count += Value;
             printf("CH%d=%.2fV (%d)\r\n", ch, (float)Value/1000.0,
             Value);
            }

            fprintf(fp,"%d\n", count / nReadNum);

            usleep(1000*100);
        }
    }
```

A.3 *Script*

```
import subprocess

# INPUT
PROJECT_NAME = 'adcnios2'
FOLDER_PROJECT_NAME = 'ADCNIOS2'
FOLDER_LOCAL = 'Desktop'
USERNAME = 'nunog'
PROGRAM_SOFTWARE = 'adc_nios3'
QUARTUS_PATH = '/intelFPGA_lite/20.1/quartus/bin64/'
NIOS2SHELL_PATH = '/intelFPGA_lite/20.1/nios2eds/'
WSL_PATH = '/mnt/c/'

# PATHS
SYSTEM_PATH = 'C:/Users/' + USERNAME
PROGRAM_FILE_SOF = PROJECT_NAME + '.sof'
ADC_PROJECT_PATH = '/' + FOLDER_LOCAL + '/' + FOLDER_PROJECT_NAME
ADC_TOP_PATH = SYSTEM_PATH + '/' + ADC_PROJECT_PATH + '/' + PROJECT_NAME
ADC_SOF = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'output_files' +
'/' + PROGRAM_FILE_SOF
ADC_ELF = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'software' +
'/' + PROGRAM_SOFTWARE + '/' + PROGRAM_SOFTWARE + '.elf'
ADC_BSP = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'software' +
'/' + PROGRAM_SOFTWARE + '_bsp'
MAKEFILE = '/Users/' + USERNAME + ADC_PROJECT_PATH + '/' +
'software' + '/' + PROGRAM_SOFTWARE
SETTINGS_BSP = ADC_BSP + '/' + 'settings.bsp'

# COMMANDS
QUARTUS_SH = QUARTUS_PATH + 'quartus_sh'
QUARTUS_PGM = QUARTUS_PATH + 'quartus_pgm'
NIOS2SHELL = NIOS2SHELL_PATH + 'Nios␣II␣Command␣Shell.bat'
WSL = 'wsl'

## QUARTUS SIDE
# Compiles project
compileFPGA = subprocess.run([QUARTUS_SH, '--flow',
'compile', ADC_TOP_PATH], shell = True)
```

```
# Check Connections
#p2 = subprocess.run([QUARTUS_PGM, '-l' ], shell = True)

# Program FPGA
programFPGA = subprocess.run([QUARTUS_PGM, '-c' ,
'USB-Blaster', '-m', 'jtag', '-o', 'p;'+ ADC_SOF], shell = True)

## NIOS2 SIDE
# Update BSP and sent settings
update_bsp = subprocess.run([NIOS2SHELL,
'nios2-bsp-generate-files.exe', '--bsp-dir', ADC_BSP,
'--settings', SETTINGS_BSP], shell = True)

# Fix PATH problems
#fix_path = subprocess.run([WSL, 'export',
'WSLENV=PATH/l:${WSLENV}'], shell = True)

# Build Makefile
build = subprocess.run([NIOS2SHELL, 'make', 'all',
'-C', WSL_PATH + MAKEFILE], shell = True)

# Run
run = subprocess.run([NIOS2SHELL,'nios2-download',
'-g', ADC_ELF], shell = True)
```

A.4 Ferramenta 4diac

A.4.1 COMPPROGFPGA.py

Começando pelo primeiro bloco e utilizando os comandos anteriormente referidos na secção 4.6:

```
import subprocess

class COMPPROGFPGA:

 def schedule(self, event_input_name,
                    event_input_value,
                    USERNAME,
                    FOLDER_LOCAL,
                    FOLDER_PROJECT_NAME,
```

```
                    PROJECT_NAME,
                    QUARTUS_PATH):
  # PATHS
  SYSTEM_PATH = 'C:/Users/' + USERNAME
  PROGRAM_FILE_SOF = PROJECT_NAME + '.sof'
  ADC_PROJECT_PATH = '/' + FOLDER_LOCAL + '/' + FOLDER_PROJECT_NAME
  ADC_TOP_PATH = SYSTEM_PATH + '/' + ADC_PROJECT_PATH + '/' + PROJECT_NAME
  ADC_SOF = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'output_files' + '/' + PROGRA
  # COMMANDS
  QUARTUS_SH = QUARTUS_PATH + 'quartus_sh'
  QUARTUS_PGM = QUARTUS_PATH + 'quartus_pgm'
  def comp():
   compileFPGA = subprocess.run([QUARTUS_SH, '--flow' ,
   'compile', ADC_TOP_PATH], shell = True)
  def program():
   programFPGA = subprocess.run([QUARTUS_PGM, '-c' , 'USB-Blaster'
   , '-m', 'jtag', '-o', 'p;'+ ADC_SOF], shell = True)

  if event_input_name == 'INIT':
   return [event_input_value, None, False]

  elif event_input_name == 'RUN':
   comp()
   program()
   return [None, event_input_value, True]
```

A.4.2 COMPPROGFPGA.fbt

O ficheiro XML correspondente:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd">
<FBType Name="COMPPROGFPGA" OpcUa="SERVICE">
 <InterfaceList>
  <EventInputs>
   <Event Name="INIT" Type="Event"/>
   <Event Name="RUN" Type="Event"/>
  </EventInputs>
  <EventOutputs>
   <Event Name="INIT_O" Type="Event"/>
   <Event Name="RUN_O" Type="Event">
```

```
        <With Var="READY"/>
    </Event>
    </EventOutputs>
     <InputVars>
        <VarDeclaration Name="USERNAME" Type="STRING"
        OpcUa="Constant"/>
                <VarDeclaration Name="FOLDER_LOCAL" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="FOLDER_PROJECT_NAME" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="PROJECT_NAME" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="QUARTUS_PATH" Type="STRING"
                OpcUa="Constant"/>
    </InputVars>
    <OutputVars>
                <VarDeclaration Name="READY" Type="BOOL" OpcUa="Variable"/>
    </OutputVars>
 </InterfaceList>
</FBType>
```

A.4.3 ADCFPGA.py

Segundo Bloco:

```
import subprocess

class ADCFPGA:

    def schedule(self,  event_input_name,
                        event_input_value,
                        USERNAME,
                        FOLDER_LOCAL,
                        FOLDER_PROJECT_NAME,
                        PROGRAM_SOFTWARE,
                        NIOS2SHELL_PATH,
                        WSL_PATH,
                        SERIAL_PORT):

        # PATHS
        SYSTEM_PATH = 'C:/Users/' + USERNAME
```

```
ADC_PROJECT_PATH = '/' + FOLDER_LOCAL + '/' + FOLDER_PROJECT_NAME
ADC_ELF = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'software' + '/'
+ PROGRAM_SOFTWARE + '/' + PROGRAM_SOFTWARE + '.elf'
ADC_BSP = SYSTEM_PATH + ADC_PROJECT_PATH + '/' + 'software' + '/'
+ PROGRAM_SOFTWARE + '_bsp'
MAKEFILE = '/Users/' + USERNAME + ADC_PROJECT_PATH + '/' +
'software'
+ '/' + PROGRAM_SOFTWARE
SETTINGS_BSP = ADC_BSP + '/' + 'settings.bsp'

# COMMANDS
NIOS2SHELL = NIOS2SHELL_PATH + 'Nios␣II␣Command␣Shell.bat'
WSL = 'wsl'

def upd_bsp():
    update_bsp = subprocess.run([NIOS2SHELL,
    'nios2-bsp-generate-files.exe',
    '--bsp-dir', ADC_BSP, '--settings', SETTINGS_BSP],
    shell = True)

def fix_path():
    fix_path = subprocess.run([WSL, 'export',
    'WSLENV=PATH/l:${WSLENV}'], shell = True)

def build_makefile():
    build = subprocess.run([NIOS2SHELL, 'make' ,'-C',
    WSL_PATH + MAKEFILE], shell = True)

def run_c():
    run = subprocess.run([NIOS2SHELL,'nios2-download',
    '-g', ADC_ELF], shell = True)

if event_input_name == 'INIT':
    return [event_input_value, None, False]

elif event_input_name == 'RUN':

    upd_bsp()
    fix_path()
    build_makefile()
```

```
            run_c()
            return [None, event_input_value, True]
```

A.4.4 ADCFPGA.fbt

Correspondente ficheiro XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd">
<FBType Name="ADCFPGA" OpcUa="SERVICE">
<InterfaceList>
    <EventInputs>
        <Event Name="INIT" Type="Event"/>
        <Event Name="RUN" Type="Event"/>
    </EventInputs>
    <EventOutputs>
        <Event Name="INIT_O" Type="Event"/>
        <Event Name="RUN_O" Type="Event">
                        <With Var="READY"/>
                </Event>
    </EventOutputs>
    <InputVars>
        <VarDeclaration Name="USERNAME" Type="STRING"
        OpcUa="Constant"/>
                <VarDeclaration Name="FOLDER_LOCAL" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="FOLDER_PROJECT_NAME" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="PROGRAM_SOFTWARE" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="NIOS2SHELL_PATH" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="WSL_PATH" Type="STRING"
                OpcUa="Constant"/>
                <VarDeclaration Name="SERIAL_PORT" Type="STRING"
                OpcUa="Constant"/>
    </InputVars>
    <OutputVars>
                <VarDeclaration Name="READY" Type="BOOL"
                OpcUa="Variable"/>
```

```
        </OutputVars>
</InterfaceList>
</FBType>
```

A.4.5 UARTFPGA.py

Ficheiro Python do terceiro bloco:

```
import serial

class UARTFPGA:

 def schedule(self, event_input_name, event_input_value, SERIAL_PORT):

    def set_serial():
        self.ser = serial.Serial(SERIAL_PORT, 115200, timeout=1)

        if event_input_name == 'INIT':

            self.sync = False
            self.data = 0

            return [event_input_value, None, None, self.data, self.sync]

        elif event_input_name == 'READ':

            if (self.sync == False):
                 self.data = 0

            elif (self.sync == True):
                self.data = self.ser.readline().decode('utf-8')

            return [None, event_input_value, None, self.data, self.sync]

        elif event_input_name == 'SYNC':

            self.sync = True
            set_serial()
            return [None, None, event_input_value, self.data, self.sync]
```

A.4.6 UARTFPGA.fbt

Ficheiro XML correspondente ao último e terceiro bloco:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd">
<FBType Name="UARTFPGA" OpcUa="DEVICE.SENSOR">
<InterfaceList>
    <EventInputs>
        <Event Name="INIT" Type="Event"/>
        <Event Name="READ" Type="Event"/>
                <Event Name="SYNC" Type="Event"/>
    </EventInputs>
    <EventOutputs>
        <Event Name="INIT_O" Type="Event"/>
        <Event Name="READ_O" Type="Event">
                        <With Var="DATA"/>
                </Event>
                <Event Name="SYNC_O" Type="Event">
                        <With Var="RESULT"/>
                </Event>
    </EventOutputs>
    <InputVars>
                <VarDeclaration Name="SERIAL_PORT" Type="STRING"
                OpcUa="Constant"/>
    </InputVars>
    <OutputVars>
                <VarDeclaration Name="DATA" Type="INT"
                OpcUa="Variable"/>
                <VarDeclaration Name="RESULT" Type="BOOL"
                OpcUa="Variable"/>
    </OutputVars>
</InterfaceList>
</FBType>
```

Anexo B

Troubleshooting

B.1 WSL

É necessário ter o WSL instalado para poder aceder às ferramentas do Nios II. O WSL tem de ser instalado à parte da Instalação do Quartus Peime ou de qualquer ferramenta do Nios II.

A Intel recomenda o uso da distribuição Ubuntu 18.04 LTS para WSL, a qual pode ser instalada a partir da Windows Store. Adicionalmente, o utilizador deve correr estes três comandos seguintes:

```
sudo apt install wsl
sudo apt install dos2unix
sudo apt install make
```

B.1.1 apt-get

Depois da instalação da distribuição Ubuntu, a mesma pode não garantir que comandos como `apt-get` funcionem correctamente. Para resolver este problema basta correr este comando seguinte no terminal WSL:

```
sudo apt update
```

B.2 Nios® II SBT for Eclipse

O conjunto de ferramentas SBT para o Nios II não se encontram no mesmo pacote de instalação do Quartus Prime desde a versão 19.1. Assim, é necessário procurar por Eclipse C/C++ IDE for Mars.2 Nios II e seguir as recomendações dadas pela Intel.

B.2.1 .elf not generated

Dentro do Eclipse para o Nios II, na ocorrência do erro

`wslpath: hal_bsp: No such file or directory`, é necessário fazer umas alterações no Makefile. Dentro do Makeifile, onde está referido

`APP_LDFLAGS += -msys-lib=$(call adjust-path-mixed,$(SYS_LIB))`, a frase deve ser mudada para `APP_LDFLAGS += -msys-lib=hal_bsp`. O ficheiro deve ser guardado e o projecto compilado.

B.2.2 .elf.srec not found

No caso de ocorrer o erro `can't disassemble for architecture UNKNOWN!`, é necessário abrir novamente o ficheiro Makefile e modificar `BUILD_PRE_PROCESS` para:

```
BUILD_PRE_PROCESS := touch $(ELF).srec
```

Novamente, o ficheiro deve ser guardado e compilado.

B.3 Drivers USB-Blaster

Um erro que pode ocorrer esporadicamente é quando a interface USB-Blaster não é reconhecida. Este erro pode ocorrer quando o Windows sofre uma actualização de *software* e deixa de conter as drivers para interpretar a ligação USB-Blaster. No entanto, o principal problema surge porque as drivers do USB-Blaster que se encontram no directório de instalação do Qaurtus Prime, não possuem uma assinatura válida e o Windows não reconhece a sua instalação. Uma das maneiras de resolver o problema é reiniciar o computador através de definições avançadas e permitir no seu inicio, a instalação de drivers não assinadas.

Referências

[1] T. Miyazaki, S. Yamaguchi, K. Kobayashi, J. Kitamichi, Song Guo, T. Tsukahara, e T. Hayashi. A software defined wireless sensor network. Em *2014 International Conference on Computing, Networking and Communications (ICNC)*, páginas 847–852, 2014. doi:10.1109/ICCNC.2014.6785448.

[2] G. Chalivendra, R. Srinivasan, e N. S. Murthy. Fpga based re-configurable wireless sensor network protocol. Em *2008 International Conference on Electronic Design*, páginas 1–4, 2008. doi:10.1109/ICED.2008.4786652.

[3] Ferran Reverter. The art of directly interfacing sensors to microcontrollers. *Journal of Low Power Electronics and Applications*, 2(4):265–281, 2012. URL: https://www.mdpi.com/2079-9268/2/4/265, doi:10.3390/jlpea2040265.

[4] Avalon® interface specifications. Relatório técnico, Intel, Dezembro 2020.

[5] Embedded design handbook. Relatório técnico, Intel, 2020.

[6] Nios® ii software developer handbook. Relatório técnico, Intel, Setembro 2020.

[7] Dinasore - dynamic intelligent architecture for software and modular reconfiguration. https://github.com/DIGI2-FEUP/dinasore.

[8] Johannes Romoth e Mario Porrmann. Survey of fpga applications in the period 2000 – 2015. 03 2017. doi:10.13140/RG.2.2.16364.56960.

[9] David Romano. A brief history of fpga, 2019. https://makezine.com/2019/10/11/a-brief-history-of-fpga/.

[10] Andrew Moore e Ron Wilson. *FPGA for dummies*. John Wiley Sons, Inc, Second edição, 2017.

[11] S. M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015. doi:10.1109/JPROC.2015.2392104.

[12] Moore's law definition. https://www.investopedia.com/terms/m/mooreslaw.asp.

[13] Daniel Ziener Dirk Koch, Frank Hannig. *FPGAs for Software Programmers*. Springer International Publishing, First edição, 2016.

[14] Aubepart Fabrice e Nicolas Franceschini. Bio-inspired optic flow sensors based on fpga: Application to micro-air-vehicles. *Microprocessors and Microsystems*, 31:408–419, 09 2007. doi:10.1016/j.micpro.2007.02.004.

[15] A.H.G. Al-Dhaher. Development of microcontroller/fpga-based systems. *International Journal of Engineering Education*, 20:52–60, 01 2004.

[16] S.Y Yurish. High performance digital sensors design: How to make it smarter?,, 2014. http://www.iaria.org/conferences2014/filesSENSORCOMM14/Yurish_Tutorial_2014.pdf.

[17] Implementing ohmmeter/temperature sensor. http://ww1.microchip.com/downloads/en/AppNotes/00512e.pdf.

[18] Peter J. Hesketh Chung-Chiun Liu Gary W. Hunter, Joseph R. Stetter. Smart sensor systems. 2010.

[19] R. Joost e R. Salomon. Advantages of fpga-based multiprocessor systems in industrial applications. volume 2005, página 6 pp., 12 2005. doi:10.1109/IECON.2005.1568946.

[20] Luis Contreras-Medina, René Romero-Troncoso, Eduardo Cabal-Yepez, José Rangel-Magdaleno, e Jesus Millan-Almaraz. Fpga-based multiple-channel vibration analyzer for industrial applications in induction motor failure detection. *IEEE Transactions on Instrumentation and Measurement*, 59:63–72, 01 2010. doi:10.1109/TIM.2009.2021642.

[21] Dhruv M. Patel e Ankit K. Shah. Fpga-plc-based multi-channel position measurement system. *ISA Transactions*, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0019057821000148, doi:https://doi.org/10.1016/j.isatra.2021.01.012.

[22] JUNBEOM YOO, JONG-HOON LEE, e JANG-SOO LEE. A research on seamless platform change of reactor protection system from plc to fpga. *Nuclear Engineering and Technology*, 45(4):477–488, 2013. URL: https://www.sciencedirect.com/science/article/pii/S1738573315300334, doi:https://doi.org/10.5516/NET.04.2012.078.

[23] M. Chmiel, W. Kloska, D. Polok, e J. Mocha. Fpga-based two-processor cpu for plc. Em *2016 International Conference on Signals and Electronic Systems (ICSES)*, páginas 247–252, 2016. doi:10.1109/ICSES.2016.7593860.

[24] Nios® ii processor reference guide. Relatório técnico, Intel, 2020.

[25] Ltc2308 - analog devices. Relatório técnico, Linear Technology, 2007.

[26] Cloud computing vs fog computing vs. edge computing na era da internet das coisas industrial. http://www.ccg.pt/cloud-computing-vs-fog-computing-vs-edge-computing-na-internet-das-coisas-indust

[27] Quartus ii scripting reference manual for command-line operation tool command language (tcl) scripting. Relatório técnico, Intel, 2013.

[28] Intel® quartus® prime standard edition user guide scripting. Relatório técnico, Intel, 2018.