Universidade do Minho
Escola de Engenharia

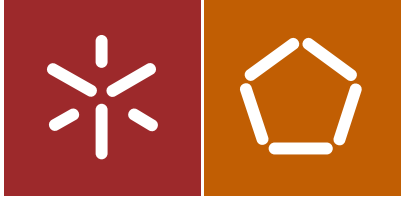
João Filipe Cerqueira Gonçalves

Middleware para Sistemas Embebidos baseados em Linux Acelerados em Hardware

julho de 2016

João Filipe Cerqueira Gonçalves Middleware para Sistemas Embebidos baseados em Linux Acelerados em Hardware

UMinho | 2016



Universidade do Minho
Escola de Engenharia

João Filipe Cerqueira Gonçalves

Middleware para Sistemas Embebidos baseados em Linux Acelerados em Hardware

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Adriano José da Conceição Tavares

julho de 2016

Agradecimentos

“Em primeiro lugar gostaria de agradecer aos meus pais por toda a dedicação e pelo esforço que fizeram para me proporcionarem a oportunidade de frequentar um curso de ensino superior. Gostaria também de agradecer à minha irmã e restante família pela presença, compreensão e todo o apoio emocional.

Agradeço ao meu orientador Adriano Tavares pela visão tecnológica fornecida ao longo da especialização e do projeto. Expresso um agradecimento especial ao Engenheiro Vitor Silva pela orientação, protagonizando um papel essencial não só na realização desta dissertação mas também na fase final da minha carreira académica.

Aos meus amigos, colegas de curso e colegas de laboratório um grande obrigado por toda a companhia, todo o apoio emocional e profissional e toda a alegria proporcionada nos momentos de descompressão.

Por fim gostaria de deixar um agradecimento especial à minha prima Ana pelo apoio nesta reta final e pela revisão ortográfica deste documento.

A todos o meu muito sincero obrigado.”

Resumo

Com o avanço tecnológico as Field-Programmable Gate Array (FPGA) aumentaram as suas capacidades computacionais podendo executar paralelamente tarefas complexas, processando grandes quantidades de dados. Os mais recentes chips FPGA podem conter, para além de uma grande quantidade de lógica programável, vários poderosos processadores. Nestes processadores pode ser executado um sistema operativo sofisticado com inúmeras vantagens, como o Linux, enquanto que na lógica programável podem ser instanciados co-processadores dedicados que podem otimizar um processo de computação intensiva ou com requisitos temporais exigentes. Este tipo de aplicações híbridas (CPU+FPGA) resulta de um processo avançado de *Hardware/Software Co-Design*, onde se exige o domínio das várias tecnologias integrantes.

De modo a facilitar este processo e torná-lo mais eficiente têm sido desenvolvidas na comunidade científica ferramentas que o auxiliam. Nesta dissertação pretende-se desenvolver um middleware que presta serviços à aplicação no controlo e comunicação com os aceleradores hardware, promovendo o desenvolvimento de aplicações híbridas (ou aceleração em hardware de aplicações software) no ambiente de sistema operativo Linux.

O MiLHA (***Mi****ddleware for* ***L****inux* ***H****ybrid* ***A****pplications*) é o serviço de suporte à aceleração em hardware de aplicações baseadas em Linux desenvolvido no âmbito desta dissertação. O MiLHA fornece um modelo de programação paralelo que integra processamento em software através de *threads* e processamento em hardware através de aceleradores por via de interface direta entre eles, abstraindo o programador de toda a complexidade que isso implica. Para além disso fornece um ambiente que auxilia a integração do acelerador no SoC.

Abstract

With the current technological progress, the FPGA computing demands increased so they can process complex parallel tasks, processing large amounts of data.

The newest FPGA chips may contain, in addition to a large quantity of programmable logic, several powerful processors. These processors may run sophisticated operating systems, such as Linux, with numerous advantages, while dedicated co-processors, instantiated in the programmable logic, can optimize processes with compute-intensive demands or hard time requirements. This hybrid applications (CPU + FPGA) results from an advanced process of Hardware/Software Co-Design, in which the designer must have advanced skills covering all technologies.

In order to simplify and improve this process, several tools have been developed and reported in the literature. This aim of this thesis is to develop a middleware that provides services in order to control and communicate with hardware accelerators. Moreover, we intended to promote the design of hybrid and hardware accelerated applications on the Linux OS environment.

The MiLHA (**Mi**ddleware for **L**inux **H**ybrid **A**pplications) is the service developed under the scope of this dissertation that supports the hardware acceleration of Linux Applications. The MiLHA provides a parallel programming model that integrates both software and hardware processing under threads and accelerators, respectively. MiLHA automatizes the interface between them, avoiding the complexity that it implies. Furthermore, MiLHA provides an environment that assists the integration of the hardware accelerator in the SoC.

Conteúdo

1 Introdução **1**

1.1 Contextualização 1
1.2 Motivação e Objetivos 2
1.3 Contribuições 3
1.4 Organização da Dissertação 3

2 Estado da Arte **5**

2.1 Sistemas embebidos 5
2.1.1 Elementos de um sistema embebido 6
2.1.2 Unidades de processamento 8
2.2 Sistemas Operativos 10
2.2.1 Linux e Sistemas Embebidos 12
2.3 Aceleração de aplicações em Hardware 13
2.3.1 FPGA 14
2.3.2 Linguagens de descrição de hardware 16
2.3.3 Design Flow 18
2.4 *Middleware* 21
2.5 Publish Subscribe Pattern 23
2.6 ReconOS 24

3 Ambiente de Desenvolvimento **27**

3.1 Plataforma Xilinx XUPV2P 27
3.1.1 Unidade de processamento 28
3.2 Xilinx Embedded Development Kit 30
3.3 Buildroot 32

4 Middleware for Linux Hybrid Applications **33**

4.1 Estrutura do MiLHA 34
4.2 Modelo de programação no MiLHA 35

4.2.1 Objeto principal MiLHA 35
4.2.2 Tasks 36
4.2.3 Comunicação entre Tasks 39
4.3 Interface MiLHA e Aceleradores 43
4.3.1 Hardware Interface API 44
4.3.2 MiLHA Kernel 49
4.4 Integração do Acelerador no MiLHA 56
4.4.1 Modelo genérico de Acelerador 56
4.4.2 Lógica Computacional do Acelerador 60
4.4.3 Comunicação com Acelerador 61

5 Cenário de aplicação **67**
5.1 Aplicação Software-only 68
5.2 Paralelização da aplicação 72
5.3 Aceleração da aplicação em hardware 75
5.4 Performance e resultados 78

6 Conclusão **81**
6.1 Trabalho desenvolvido 81
6.2 Trabalho Futuro 83

Bibliografia **85**

A Template de função de thread **89**

B Processos de leitura e escrita num acelerador **91**
B.1 Processo de Escrita 91
B.2 Processo de Leitura 92

C Template de User Logic Wrapper **95**

Lista de Figuras

2.1 Diagrama de blocos de um sistema embebido genérico [31] 8
2.2 Tipos de unidades de processamento 9
2.3 Diagrama de blocos de um SoC [38] 9
2.4 Estrutura interna da FPGA [21] 15
2.5 Diagrama de blocos do Zynq UltraScale+ EV MPSoC [39] 16
2.6 Metodologia aceleração de uma aplicação em *hardware* 19
2.7 *Middleware* em modelos de Sistemas Embebidos [30] 22
2.8 Exemplo padrão *publish-subscribe topic-based* [27] 23
2.9 Visão geral dos componentes do ReconOS [25] 25

3.1 Plataforma XUPV2P [40] 29
3.2 Diagrama de blocos do PowerPC 405 [17] 30
3.3 Ciclo de desenvolvimento do Xilinx EDK [13] 31
3.4 Menu de configuração do buildroot 32

4.1 Estrutura do MiLHA 34
4.2 Diagrama de classes do modelo programação MiLHA 36
4.3 Os dois tipos de task existentes. 37
4.4 Diagrama de classes UML para CThread 38
4.5 Exemplo interação entre duas `Tasks` 39
4.6 Diagrama UML para a classe CTopic 40
4.7 Exemplo de publicação de dados num tópico subscrito pelos diferentes tipos de `Tasks` 42
4.8 Ligações para comunicação entre `Hardware Tasks` 42
4.9 Esquema geral da camada de interface com o hardware do MiLHA . 43
4.10 Diagrama de classes Hardware Interface API 44
4.11 Sequencia de criação de um Proxy Device 45
4.12 Sequência da execução do processo de escrita 48
4.13 Sequência de execução do processo de leitura 49

4.14 Exemplo de um nó MiLHA Interface na *device-tree* 53
4.15 Execução de pedido de escrita 54
4.16 Execução de pedido de leitura 54
4.17 Esquema *buffer* 55
4.18 Mecanismos de integração do acelerador 56
4.19 Esquema do User Logic Wrapper 57
4.20 Possíveis ligações entre *wrappers* 58
4.21 Máquina de estados de receção de dados 59
4.22 Máquina de estados de envio de dados 59
4.23 Top Level do modelo de acelerador hardware 60
4.24 Máquina de estados do modelo de acelerador 60
4.25 Sinais lógicos do modelo de acelerador 60
4.26 Esquema MiLHA Interface IP 61
4.27 Parâmetros do MiLHA Interface IP no XPS 63
4.28 Protocolo comunicação MiLHA Interface - acelerador 63
4.29 *Top Level* das FIFO 64
4.30 Máquina de estados de controlo da escrita na Input FIFO 64
4.31 Máquina de estados de controlo de leituras na Output FIFO 64
4.32 Máquina de estados de controlo da leitura da Input FIFO 65
4.33 Máquina de estados de controlo de escritas na Output FIFO 65

5.1 Esquema de um Filtro Ativo de Potência 68
5.2 Diagrama de classes da aplicação de demonstração 69
5.3 *Taskgraph* do algoritmo de cálculo de correntes de compensação . . 70
5.4 Circuito de potência do FAP 71
5.5 Resultados da simulação do FAP 72
5.6 Esquema de Tasks e Tópicos da aplicação paralelizada em software 73
5.7 Código de criação da aplicação software 74
5.8 Aplicação gráfica na plataforma *target* 75
5.9 Top-Level dos aceleradores hardware da aplicação de demonstração 76
5.10 Configuração das ligações entre aceleradores em *array* 77
5.11 Código de criação da aplicação software com o *array* de aceleradores 77
5.12 Configuração das ligações entre aceleradores em ADC + *array* . . . 78
5.13 Código de criação da aplicação software ADC + *array* 78

B.1 Processamento de um pedido de escrita 91
B.2 Processamento de um pedido de leitura 92

Lista de Tabelas

3.1 Caracteristicas do chip FPGA Xilinx XC2VP30 [40] 28

5.1 Resultado da medições do tempo de execução do algorimo de cálculo das correntes de compensação 79

5.2 Medições de tempos de execução das várias funcionalidades do MiLHA 80

Lista de Acrónimos

ADC Analog-to-Digital Converter.

ASIC Application-Specific Integrated Circuit.

ASIP Application-Specific Instruction set Processor.

DAC (Digital-to-Analog Converter.

DLL Dynamic-link library.

DSP Digital Signal Processor.

FAP Filtro Ativo Paralelo.

FPGA Field-Programmable Gate Array.

GPIO General Purpose Input/Output.

GPL General Public License.

GPP General-Purpose Processor.

HDL Hardware Description Language.

I2C (Inter-Integrated Circuit.

IP Intellectual Property.

JTAG Joint Test Action Group.

LUT LookUp Tables.

MPSoC Multiprocessor System-on-Chip.

PWM Pulse-Width Modulation.

RAM Random Access Memory.

SoC Sistem-on-Chip.

SPI (Serial Peripheral Interface.

USB Universal Serial Bus.

Capítulo 1

Introdução

Este capítulo inicia com uma contextualização do âmbito desta dissertação, são depois apresentados os seus objetivos e a motivação do autor. Seguem-se as contribuições deste trabalho e finaliza-se o capítulo com a organização deste documento.

1.1 Contextualização

Os *designers* de sistemas embebidos estão continuamente a ser desafiados para fornecerem mais poder computacional e a cumprirem requisitos cada vez mais exigentes. Para que isso seja possível o esforço pelo desenvolvimento de novas e mais avançadas tecnologias é constante. Na busca de maior performance surgem processadores cada vez mais poderosos e complexos. Os General-Purpose Processor (GPP) providenciam boa performance numa larga variedade de programas, no entanto, pelo seu carácter genérico, podem muitas vezes não ser suficientemente rápidos para aplicações mais especificas. Neste sentido surgiram processadores especializados, como por exemplo os DSP, cujas instruções especializadas permitem o aumento de performance em aplicações no domínio de processamento de sinais, contudo, em aplicações fora desse domínio, a sua performance pode não ser suficiente.

Aí surgem, com o recente desenvolvimento da tecnologia FPGA, os *chips* híbridos. Estes podem conter no seu interior um, ou até vários, processadores genéricos e ainda uma vasta área de lógica programável. Isto permite a execução de aplicações sofisticadas nas suas unidades de processamento genérico, enquanto partes

da aplicação, de processamento intensivo ou com requisitos de tempo real, podem ser migradas para aceleradores dedicados instanciados na lógica programável, aproveitando as capacidades de processamento paralelo do hardware. Este tipo de abordagem permite executar um sistema operativo complexo como o Linux com as suas inúmeras vantagens, entre as quais, gestão total dos recursos hardware da plataforma, potenciação da reutilização de código, fornecimento de um grande conjunto de serviços e bibliotecas, acesso a mecanismos de interface com o hardware (uma vasta quantidade de *device drivers*), acesso a bibliotecas e *frameworks* de programação gráfica. No entanto, a utilização de um sistema operativo compromete as necessidades de tempo real de uma aplicação já que acrescenta imprevisibilidade e *overhead* de execução, o que muitas vezes torna impossível satisfazer características necessárias como o determinismo e baixas latências. Com o chip híbrido (CPU+FPGA), o núcleo de computação intensiva ou com requisitos *real time* da aplicação não é comprometido já que executa na lógica programável, paralelamente ao sistema operativo.

O *design* deste tipo de aplicações requer, para além de um grande esforço, conhecimentos específicos avançados como por exemplo programar linguagens de descrição de *hardware* e conhecimento de arquiteturas de microcontroladores que as deixa fora do alcance de grande parte dos programadores. Para contornar essa situação é necessário criar ferramentas, modelos de programação e ambientes de execução para aplicações híbridas (*software/hardware*), à semelhança dos criados para aplicações *software only*. Assim é promovida uma metodologia de design eficiente que é acompanhada por ferramentas que auxiliam e aceleram o processo de *design*.

1.2 Motivação e Objetivos

O processo de aceleração de uma aplicação em hardware é um processo complexo, moroso e muitas vezes difícil de concretizar mantendo a versão inicial da aplicação com o mínimo de alterações. Este processo envolve um conjunto de tarefas onde o programador desenvolve um elevado esforço na programação em software e na programação de hardware, através de linguagens de HDL.

À semelhança das aplicações software para as quais foram desenvolvidas ferramentas como bibliotecas, *frameworks*, sistemas operativos, etc., que auxiliam o seu desenvolvimento, têm sido desenvolvidas outras com suporte a aplicações híbridas e a aceleração de aplicações. O objetivo é desenvolver um serviço de suporte

à aceleração de aplicações baseado em middleware que presta serviços à camada da aplicação para comunicação e controlo com o aceleradores hardware. Pretende-se promover o desenvolvimento de aplicações híbridas e a aceleração de aplicações em ambientes de sistema operativo Linux. O conjunto de serviços desenvolvidos pretende tornar o processo de paralelização mais eficiente, mais acessível, promovendo uma metodologia de *design* e diminuindo o *time to market*.

1.3 Contribuições

Esta dissertação envolve áreas como sistemas embebidos, especialmente nos campos de Linux e FPGA, e alguma eletrónica de potência.

O Linux é o sistema operativo que maior quantidade de arquiteturas suporta, para além de uma vasta gama de dispositivos hardware, no entanto o suporte à aceleração de aplicações ainda não se encontra intrinsecamente presente no Linux. O trabalho desenvolvido contribui nesse sentido, da inclusão do suporte à aceleração de aplicações em hardware no Linux.

A tecnologia FPGA, com o seu grande potencial, encontra-se numa fase de grande evolução, prevendo-se que num futuro próximo grande parte dos computadores pessoais a possuam na sua unidade de processamento lógica programável. Esta dissertação pode, eventualmente, dar um pequeno contributo nesse sentido.

1.4 Organização da Dissertação

Este documento apresenta o desenvolvimento de um serviço de suporte à aceleração de aplicações em hardware. Para além deste primeiro capítulo introdutório é composto mais cinco capítulos.

No segundo capítulo são introduzidas ao leitor tecnologias, conceitos teóricos e metodologias que se inserem no contexto desta dissertação. É apresentada uma introdução sobre Sistemas Embebidos, Sistemas Operativos e Aceleração de aplicações em hardware, alguns conceitos utilizados e por fim alguns serviços já com propósito semelhante ao desenvolvido.

O terceiro capítulo apresenta o ambiente tecnológico em que esta dissertação foi realizada. É apresentada a plataforma hardware utilizada bem como as ferramentas

utilizadas para a sua manipulação.

No quarto capítulo é apresentado detalhadamente o serviço de suporte à aceleração de aplicações em hardware desenvolvido. São apresentadas as suas funcionalidades, a forma como está modelado, com alguns detalhes sobre a sua implementação e como utiliza-lo.

No quinto capítulo é apresentado um cenário de aplicação para o sistema desenvolvido. É criada uma aplicação exemplo, onde depois é aplicado o serviço desenvolvido.

O sexto e último capítulo apresenta uma breve conclusão sobre o serviço desenvolvido, evidenciando as suas vantagens de desvantagens. São também apresentadas algumas propostas para uma futura continuação do desenvolvimento deste serviço.

Capítulo 2

Estado da Arte

Neste capítulo é apresentada uma breve descrição das tecnologias utilizadas no desenvolvimento desta dissertação. São apresentadas algumas considerações sobre sistemas embebidos, sistemas operativos e sobre o processo de aceleração de aplicações em hardware. São apresentados alguns conceitos utilizados no projeto desta dissertação e o capítulo é finalizado com a apresentação de uma ferramenta que surgiu no mesmo âmbito desta dissertação.

2.1 Sistemas embebidos

Um sistema embebido é a combinação de hardware e software e, possivelmente, partes mecânicas desenhada para executar um tarefa específica [5].

Embora muitas vezes passem despercebidos os sistemas embebidos estão presentes um pouco por todo lado e são utilizados em diversas tarefas do quotidiano. São de variados tamanhos e complexidades, desde o pequeno relógio digital ou leitor MP3 até grandes eletrodomésticos, impressoras, televisões, etc. Normalmente têm interface com o mundo exterior, através de sensores (*input*) e atuadores *output*. Sistemas embebidos são, normalmente autónomos, alguns até sem nenhuma configurabilidade, no entanto também podem possuir interface com o utilizador, através de, por exemplo, teclados, leds, ecrã tátil, etc.

Um sistema embebido tem um propósito específico, este é desenvolvido e otimizado para esse propósito. Por exemplo um computador pessoal, apesar de ser também um conjunto de software e hardware, não é considerado um sistema embebido pois

o seu propósito é genérico. Pelo seu propósito específico, o seu desenvolvimento é orientado à sua aplicação, desse modo os engenheiros que desenvolvem estes tipos de sistemas devem criá-lo com os recursos estritamente necessários para a execução da tarefa proposta, de modo a otimizar variáveis como custo, consumo energético, peso e performance. No entanto, como é natural no desenvolvimento de projetos, o desenvolvimento de um sistema embebido está sempre sujeito a requisitos e restrições,que têm de ser respeitadas, deste modo é necessário um balanço e uma gestão dos recursos utilizados. Um caso especial dos sistemas embebidos são os sistemas embebidos com restrições temporais exigentes, estes podem ser considerados *soft real-time*, se a consequência de uma *deadline* ultrapassado não for fatal, como por exemplo aplicações de *streaming*, onde são permitidos atrasos ou perdas de dados. Por outro lado num sistema *hard real-time*, o incumprimento de uma *deadline* poderá ter uma consequência catastrófica, como por exemplo um sistema de controlo da trajetória de numa nave espacial.

Sistemas embebidos mais simples são usualmente codificados diretamente sobre a unidade de processamento utilizada (*bare metal*). No entanto com o aumenta de complexidade dos sistemas a utilização de um sistema operativo de suporte torna-se imprescindível. Para além disso metodologias de *design* e de eficazes, como o *watterfall model* facilitam o desenvolvimento de sistemas complexos.

2.1.1 Elementos de um sistema embebido

A nível arquitetural, o sistema embebido é representado pela interação de elementos de software e hardware cujos detalhes de implementação são abstraídos, mantendo-se apenas informação a nível comportamental e relacional. Estes elementos podem estar integrados internamente no dispositivo embebido, ou existirem externamente interagindo com os elementos internos assim como com o ambiente circundante [30].

Um sistema embebido é geralmente constituído pelos elementos básicos necessários à execução de um código, periféricos internos, interfaces de comunicação e o software associado. De forma genérica um sistema embebido possui os seguintes elementos:

- **Unidade de processamento central:** Responsável pela execução do fluxo de código, realizando as operações lógicas, de controlo e de entrada/saída;

- **Memória Random Access Memory (RAM):** É uma memória volátil de acesso rápido que é utilizado para armazenamento temporário das variáveis necessárias durante a execução do fluxo de código;
- **Memória Flash:** É uma memória não volátil, de acesso mais lento que a RAM, utilizada para guardar dados permanentemente. Pode conter o código para arranque do sistema embebido, o código do sistema operativo, programas, e sistema de ficheiros;
- **Periféricos de comunicação:** Um sistema embebido utiliza frequentemente protocolos de comunicação como o Universal Serial Bus (USB), RS232, Ethernet, para os quais existem periféricos dedicados;
- **Dispositivos de entrada/saída:** Como já referido um sistema operativo tem interface com o mundo exterior, podendo ou não ter também com o utilizador. Alguns periféricos como Analog-to-Digital Converter (ADC), controladores de áudio, controladores de General Purpose Input/Output (GPIO), entre outros, estão também geralmente presentes num sistema embebido.

A Figura 2.1 apresenta uma possível combinação de hardware para uma plataforma de desenvolvimento de sistemas embebidos. Estas plataformas contêm uma unidade de processamento central e, complementarmente, uma série de componentes, como memórias, áudio (Digital-to-Analog Converter (DAC), controladores de vídeo, expansores de GPIO, e também uma série de interfaces, como (Inter-Integrated Circuit (I2C), (Serial Peripheral Interface (SPI), Ethernet, Joint Test Action Group (JTAG). Estas plataformas de desenvolvimento são uma solução prática para prototipar um sistema embebido. Dado que não é prática a espera pela finalização do desenvolvimento do hardware para se iniciar o desenvolvimento do software e combinar o desenvolvimento do software e hardware pode tornar muito difícil a deteção e identificação das falhas. Depois de desenvolvido e testado o protótipo pode ser criada uma plataforma com apenas o hardware estritamente necessário à aplicação.

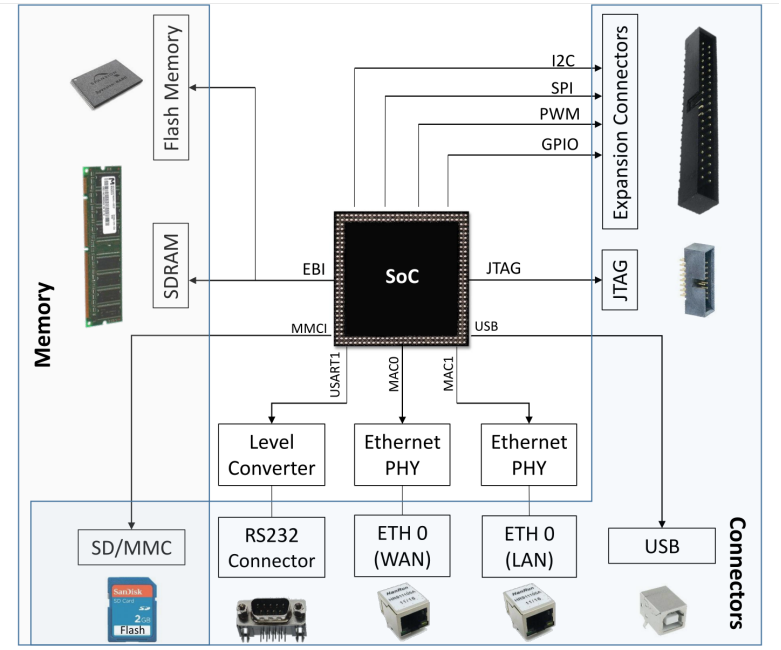


Figura 2.1: Diagrama de blocos de um sistema embebido genérico [31]

2.1.2 Unidades de processamento

O componente principal de um sistema embebido é a unidade de processamento computacional. A Figura 2.2 apresenta os diferentes tipos de unidades de processamento e relaciona a sua flexibilidade com a performance e eficiência energética. Os GPP apesar de apresentarem um maior consumo energético possuem grande flexibilidade proporcionada pelo vasto conjunto de instruções fornecido. Embora possua grande flexibilidade, este processador pode não suportar muitos dos cenários de aplicação de um sistema embebido com requisitos mais específicos. Nestes cenários são utilizados outros tipos de processamento mais especializados.

Os Application-Specific Instruction set Processor (ASIP) são processadores especializados num tipo de aplicação. São processadores flexíveis e possuem componentes *hardwired* específicos à aplicação, adicionais à unidade processamento básico, bem como instruções especializadas. Alguns exemplos de ASIP são os Digital Signal Processor (DSP) e os Sistem-on-Chip (SoC).

Os DSP é um tipo de processador especializado em aplicações orientadas em *data stream* que inclui instruções e periféricos específicos para esse efeito, como por exemplo o *floting-point unit*.

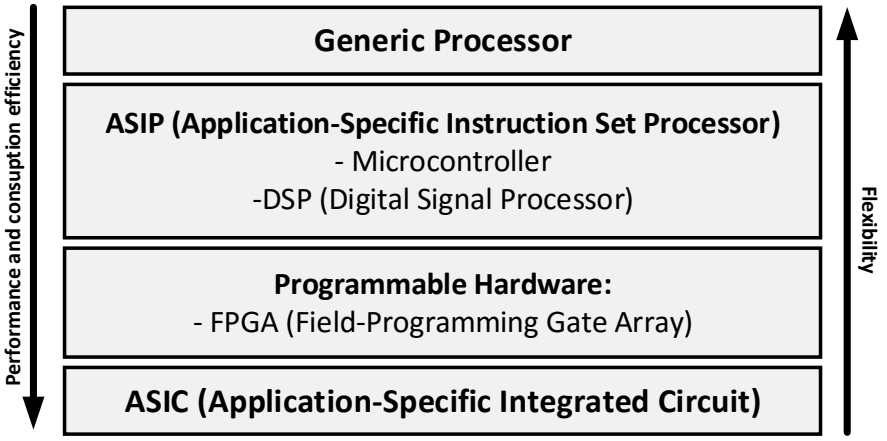


Figura 2.2: Tipos de unidades de processamento

O SoC é um circuito integrado que engloba vários componentes num único chip. Como apresentado na Figura 2.3, um SoC pode ser constituído pelos elementos presentes num microcontrolador: uma unidade processamento central (CPU); unidades de memória (ROM, FLASH,etc) e seus controladores; interfaces de comunicação (USB, ethernet, etc) e por periféricos específicos do SoC (não presentes num microcontrolador), como por exemplo controladores de áudio e vídeo.

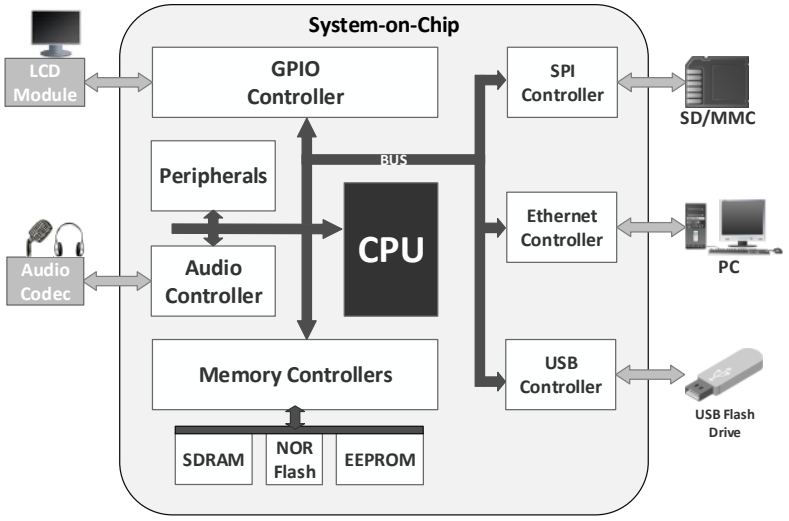


Figura 2.3: Diagrama de blocos de um SoC [38]

Por outro lado os Application-Specific Integrated Circuit (ASIC) são circuito integrados customizados para suportar uma aplicação especifica. A sua funcionalidade está *hardwired* nos seus circuitos e não é programável. Por ser desenvolvido especificamente para um efeito este consegue obter a maior performance e o menor consumo energético. No entanto os custos para o seu desenvolvimento são muito

elevados, e seria apenas rentável uma produção e venda em massa. A economia em escala atual, onde as alterações e *updates* são constantes, é muitas vezes incompatível com esta solução.

As FPGA providenciam uma alternativa aos ASIC de menor custo e risco, uma vez que podem ser reprogramadas. Esta tecnologia é frequentemente utilizada para prototipar e depurar os próprios ASIC. Como produto final apresentam uma solução com menor performance, no entanto têm um risco e um custo de desenvolvimento menor. A tecnologia FPGA está muitas vezes presente em unidades SoC e permite a criação de periféricos customizados pelo utilizador.

2.2 Sistemas Operativos

Nos anos 60, a definição de um sistema operativo resumia-se a "o software que controla o hardware" mas desde então os sistemas computacionais evoluíram significativamente. Atualmente o hardware executa uma grande variedade de aplicações e, de maneira a otimizar a utilização dos recursos hardware, partilhando-o entre aplicações, existe uma camada intermédia de software, o sistema operativo [12].

O objetivo é fornecer ao utilizador de uma forma simples de acesso e partilha dos recursos hardware disponíveis, como por exemplo processadores, memória e dispositivos de entrada/saída, e ao mesmo tempo gerir e otimizar a sua utilização. O software que contém as componentes de interface com o hardware é denominado *Kernel*, este executa código que pode comprometer o sistema. De um modo geral o sistema operativo torna um conjunto de hardware numa máquina mais facilmente utilizável, criando um ambiente virtual com diversos serviços que facilitam a utilização do hardware da máquina [23], como por exemplo:

- **Gestão de tarefas:** É das mais importantes funcionalidades de um sistema operativo. Num sistema operativo existe o conceito de tarefa, estas são porções de código cridas para realizar uma determinada função. Num sistema coexistem várias tarefas e o sistema operativo é o responsável pela escalonação da sua execução no processador. Isto cria a ilusão que várias tarefas estão a ser executadas simultaneamente, mesmo existindo apenas um único processador. O tempo e a sequência de execução de cada tarefa é determinado pela política de escalonamento utilizada. O *multitasking* torna o

sistema muito mais escalável, modular e permite ao utilizador ter controlo da sua execução;

- **Gestão de interfaces I/O:** As funcionalidades do hardware básico podem ser extremamente complexas e requerer programação sofisticada de forma a poderem ser utilizadas. O sistema operativo cria uma abstração sobre esta complexidade e fornece ao utilizador opções simples de utilização do hardware;

- **Gestão de memória:** O sistema operativo faz a gestão da memória disponível na máquina e fornece ao utilizador mecanismos simples de alocação e libertação da mesma. Muitos sistemas operativos criam um espaço de memória virtual onde o utilizador observa toda a memória homogénea e disponível, quando na realidade esta pode ser proveniente de diferentes espaços físicos (exemplo RAM ou Cache);

- **Sistema de ficheiros:** Muitos sistemas operativos incluem um sistema de ficheiros para armazenamento não volátil de programas de dados. Os dados do sistema de ficheiros estão presentes nos espaços de memória das memórias não voláteis da máquina (exemplo disco rígido ou memória Flash) e o sistema operativo permite ao utilizador aceder a essa informação através de nomes simbólicos (*path*) em vez de providenciar detalhes sobre a sua localização física;

- **Comunicação entre processos:** O sistema operativo fornece um conjunto de serviços de interação entre programas, como por exemplo *pipes*, *sockets*, etc;

- **Proteção e tratamento de erros:** Um erro numa pequena parte do sistema pode comprometer o sistema como um todo. Para evitar este problema os sistemas operativos monitorizam as tarefas para deteção de erros. Na deteção de erros são tomadas ações, como por exemplo a terminação da tarefa, que protegem o restante sistema.

Alguns exemplos de sistemas operativos são o Windows OS da Microsoft, presente na maioria dos computadores pessoais, o seu concorrente direto MacOS da Apple, o Linux, um sistema operativo de código livre, entre outros.

Em sistemas embebidos o sistema operativo mais utilizado é o Linux devido a um conjunto de factos que seguidamente irão ser abordados.

2.2.1 Linux e Sistemas Embebidos

Linux é um sistema operativo de código aberto, licenciado sobre os termos da General Public License (GPL). Em 1991, como *hobby*, Linus Torvalds, estudante universitário de ciências de computação, criou o primeiro *Kernel* para o sistema operativo que o próprio viria a batizar de Linux. Com o crescimento da Internet no início dos anos 90, entusiastas criaram comunidades online com o objetivo de desenvolver o Linux. Desde aí, o Linux evoluiu cada vez com mais funcionalidades e suporte a uma vasta gama de arquiteturas e hardware [43].

Linux é atualmente o sistema operativo mais utilizado no mundo [9], ele está presente em mais de 97% dos supercomputadores existentes no mundo, mais de 80% dos *smartphones*, muitos milhões de computadores pessoais, 70% dos servidores Web, um grande número de *tablets*, e outros variados dispositivos [9].

Várias características do Linux foram responsáveis pela sua adoção e difusão pelos programadores [43]:

- **Modularidade e estrutura:** As diferentes funcionalidades estão implementadas em módulos distintos. Funções complexas são divididas num número adequado de funções mais simples, facilitando a compreensão do código;
- **Legibilidade:** O código encontra-se legível, o que facilita o trabalho dos programadores na compreensão da estrutura interna do Linux;
- **Extensibilidade:** Facilidade em adicionar dinamicamente funcionalidades ao Linux;
- **Configurabilidade:** Na criação de um sistema Linux é possível selecionar as funcionalidades desejadas na aplicação final. Este aspeto é muito interessante para o desenvolvimento de sistemas operativos onde os recursos devem ser os estritamente necessários à aplicação;
- **Previsibilidade:** Em execução o comportamento dos programas é coerente com a sua implementação;
- **Longevidade:** Os programas executam sem assistência durante longos períodos de tempo, conservando a sua integridade. Exatamente uma das características base de um sistema embebido: funcionar corretamente durante longos períodos de tempo;

- **Recuperação de erros:** Na ocorrência de erros o Linux regista, alerta e toma medidas para recuperar do problema.

O modelo de desenvolvimento *open-source*, permitiu que a comunidade de programadores contribuísse para o desenvolvimento do Linux. Com o seu código aberto, a identificação de problemas, discussão sobre as suas soluções e a sua correção tornou-se muito mais rápida. A maioria dos programadores concorda que o Linux é um sistema operativo de qualidade e de confiança [43]. Com a contribuição de um grande número de programadores o suporte fornecido pelo Linux aumentou rapidamente que outros sistemas operativos desenvolvidos apenas pelo conjunto de programadores da empresa responsável.

Algumas características que distinguiram o Linux de outros sistemas operativos foram [37]:

- É o sistema operativos com o mais alargado suporte a dispositivos hardware. Suporta um grande número de arquiteturas e possui uma vasta quantidade de *device-drivers* de suporte ao mais variado hardware;
- Suporta uma variedade de protocolos de ligação em rede, e também protocolos de comunicação (USB, SPI, etc), muito úteis na interação com outros sistemas informáticos;
- Pode ser gratuitamente distribuído em produtos, tornando os produtos baseados em Linux comercialmente mais competitivos;
- Grandes fabricantes do mercado, como IBM, Texas Instruments, Atmel, Samsung dão suporte ao sistema operativo Linux;
- A grande comunidade ativa que participa no desenvolvimento do Linux acelera o suporte a novas arquiteturas e dispositivos hardware. Com um maior suporte é potenciada a portabilidade das aplicações baseadas em Linux [38].

2.3 Aceleração de aplicações em Hardware

As exigências computacionais das aplicações superam as capacidades dos processadores convencionais [11]. A aceleração em hardware aumenta a performance de computação através de co-processadores dedicados específicos à aplicação. Deste modo, quando o poder de processamento em software não é suficiente para cumprir

os requisitos de um determinado processo, este, ou parte dele, é implementado em hardware dedicado, sob a forma de um acelerador.

A tecnologia FPGA, com a sua capacidade de criar hardware reprogramável aliada à boa combinação de características como custo, performance, facilidade de uso e uma boa eficiência energética tornou-se a base da aceleração em hardware.

Seguidamente serão apresentadas algumas características da tecnologia FPGA, bem como as linguagens Hardware Description Language (HDL) que as programam e, finalmente, explicado o processo de aceleração adotado nesta dissertação.

2.3.1 FPGA

As FPGA são dispositivos semicondutores que contêm blocos lógicos configuráveis pré-construídos, conectados por interligações programáveis. Deste modo é possível criar um bloco de processamento em hardware, semelhante a um ASIC, no entanto este pode ser posteriormente reprogramado. Da flexibilidade proporcionada pelas FPGA advém uma redução na sua performance em relação aos ASIC, a FPGA requer entre 20 a 35 vezes mais área e é 3 a 4 vezes mais lenta do que o seu equivalente em ASIC [21].

A matriz de blocos programáveis presentes na FPGA inclui blocos do tipo lógica geral, memória e multiplicadores [21], por suas vez, os blocos de lógica geral podem ser *flip-flops* ou LookUp Tables (LUT). Os diferentes blocos são organizados numa matriz com interconexões entre eles que podem ser ativadas ou não, implementando deste modo diferentes comportamentos e consequentemente circuitos digitais mais complexos. A Figura 2.4 apresenta o diagrama interno de uma FPGA.

Os blocos especializados apresentam melhor performance que os criados pela interligação de blocos básicos, deste modo, muitos chips FPGA incluem elementos *hardwired* que implementam funções frequentemente usadas como multiplicadores, memórias e blocos DSP. A configuração presente na FPGA é volátil, necessitando assim de ser reprogramada cada vez que é ligada. As FPGA modernas incluem uma memória não volátil que guarda o ficheiro de configuração da FPGA (*bitstream*), a partir da qual a FPGA é reprogramada automaticamente, quando selecionada esta opção.

Os avanços na tecnologia FPGA proporcionam a sua crescente utilização em aplicações específicas. Por exemplo, nas atuais *smart TV*, é frequentemente utilizado

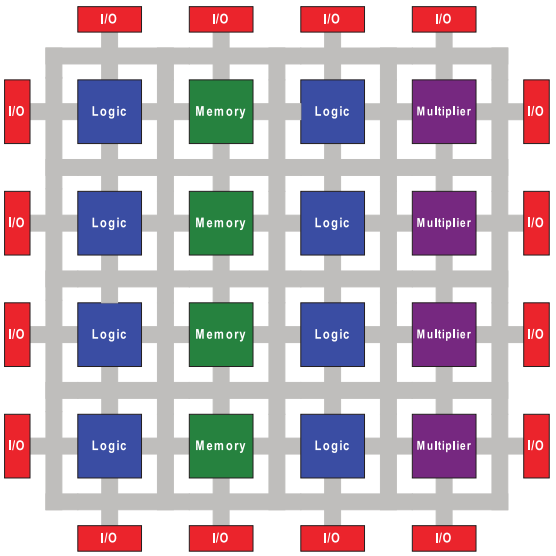


Figura 2.4: Estrutura interna da FPGA [21]

um chip FPGA pois, mantendo a mesma plataforma, este consegue ser atualizado e melhorado para o desenvolvimento dos novos modelos. Neste tipo de aplicações o bloco FPGA implementa um SoC que, para além de uma unidade de processamento genérico, implementa periféricos especializados na aplicação, como controladores gráficos. A unidade de processamento pode estar implementada na lógica programável do chip FPGA (*softcore processor*) ou pode estar embutida no chip FPGA (*hardwired processor*) que apresenta melhor performance, no entanto não pode ser reconfigurado. Chips FPGA modernos integram uma vasta quantidade de lógica programável, para além de uma enorme oferta de periféricos e unidades processamento *hardwired* no próprio chip. Isto alia o processamento de vários poderosos processadores à flexibilidade da lógica programável da FPGA. Na Figura 2.5 é apresentado um Multiprocessor System-on-Chip (MPSoC), o Zynq UltraScale+ EV da Xilinx, que integra, para além de *multicore processors* e um vasto leque de outros periféricos, *codecs* de vídeo para alta resolução (4k). Este tipo de SoC são também utilizados em sistemas com requisitos de tempo real e para aceleração de aplicações em hardware, onde o processamento crítico é transferido do processador para co-processadores dedicados, implementados na lógica programável da FPGA, que executam paralelamente ao fluxo de código nos processadores.

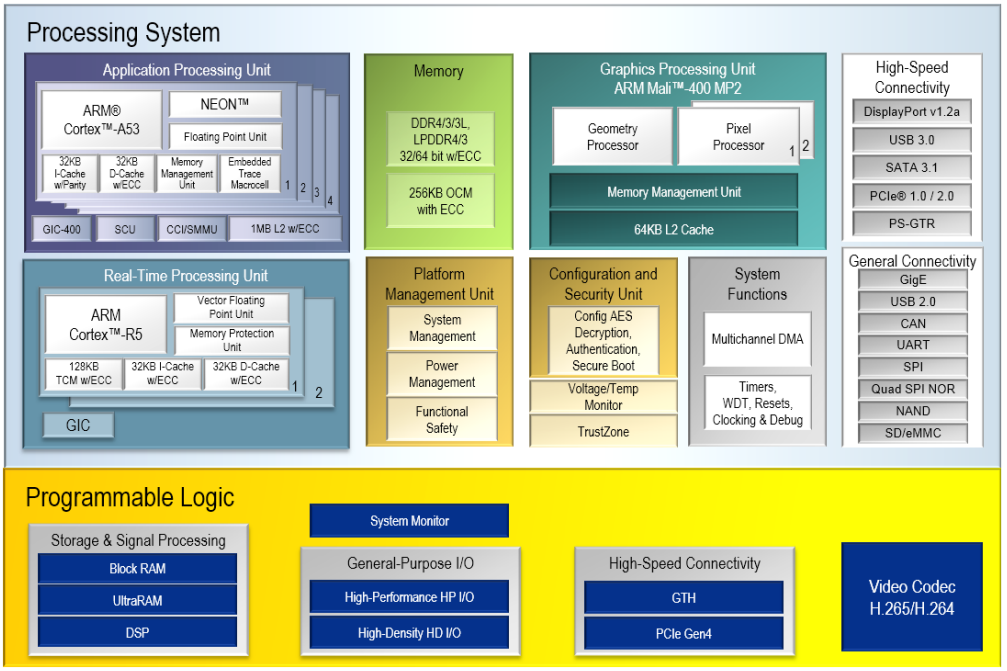


Figura 2.5: Diagrama de blocos do Zynq UltraScale+ EV MPSoC [39]

2.3.2 Linguagens de descrição de hardware

As linguagens de descrição de hardware ou HDLs são linguagens de programação usadas para descrever circuitos eletrónicos digitais. O código HDL que descreve o circuito pode posteriormente ser compilado por ferramentas de síntese, ou simulado por ferramentas de simulação. Da compilação resulta a implementação física do circuito descrito, que pode ser posteriormente sintetizado num chip FPGA ou até mesmo aplicado num ASIC.

Devido à natureza paralela do hardware, a execução das HDL é diferente das linguagens de programação de software, onde o código é sequencial. Esta situação requer uma mudança do paradigma de pensamento do programador que está habituado à programação em software.

As HDL permitem a descrição do hardware em diferentes níveis de abstração: a nível comportamental, o de maior abstração e menor detalhe, onde o programador define o comportamento do circuito, deixando os detalhes de implementação para o compilador; a nível estrutural, um nível intermédio onde o programador define a estrutura do sistema; e por fim, o nível físico, sem abstração e com maior detalhe, onde o programador define os componentes físicos do sistema, podendo chegar a nível da porta lógica ou do transístor [14].

A simulação do código HDL é uma ferramenta de validação e depuração muito importante no desenvolvimento de circuitos digitais. Do mesmo modo que os sistemas podem ser descritos em vários níveis de abstração, a simulação também pode ser executada em vários níveis de detalhe. Dependendo da ferramenta de simulação, o código HDL pode ser simulado desde o nível comportamental até à pós-síntese, em que o sistema é simulado posteriormente à ação do compilador.

Um circuito especificado em HDL é, normalmente, sub-dividido em vários módulos menos complexos. Cada módulo é codificado individualmente e depois utilizado como um *blackbox* onde são abstraídos os detalhes de implementação e é apenas observado como um conjunto de entradas e saídas sobre as quais é executada a função que lhe foi destinada. Durante a simulação, para a validação do módulo, as estradas são estimuladas e é verificados se os resultados obtidos nas saídas estão de acordo com o previsto. Estes estímulos estão geralmente associados a um outro módulo onde é incluído o módulo em teste, este é denominado o *testbench*.

Existem várias linguagens de descrição de hardware, no entanto Verilog e VHDL são as mais utilizadas e com maior suporte.

A VHDL deriva e tem sintaxe semelhante à linguagem de programação ADA, é do tipo *strongly-typed* e no seu *stardard* contém um maior número de tipos de dados não triviais. Por ser *strongly-typed* requer um código mais extenso para as conversões entre tipos de dados, no entanto também facilita a leitura do código e tem como objetivo tornar a linguagem *self-documented*. O facto de ser *strongly-typed*, o que baixa o grau de liberdade do programador, aumenta a rigidez da linguagem, que aumenta a deteção de erros em fases iniciais do projeto [4].

Já o Verilog tem sintaxe semelhante à linguagem de programação C, uma linguagem muito conhecida e utilizada por engenheiros que facilita introdução da HDL. É uma linguagem *weakly-typed*, reconhece que todos os tipos de dados têm uma representação a nível do *bit*. Deste modo é mais flexível e permite o desenvolvimento de código mais rapidamente, no entanto não serão prevenidos tantos erros em fases iniciais do projeto, que podem atrasar o desenvolvimento em fases finais [4].

Em suma, os *designers* do VHDL queriam uma linguagem segura que detetasse o maior número possível de erros em fases iniciais do projeto. Isto com um maior custo de desenvolvimento e com ferramentas de síntese mais lentas, devido à maior verificação do código. O *designers* do Verilog queriam uma linguagem que permi-

tisse aos programadores desenvolver módulos mais rapidamente. No entanto estas duas linguagens não são mutuamente exclusivas, as ferramentas de síntese atuais permitem *designs* com a mistura de ambas.

No projeto desta dissertação, foi utilizada maioritariamente a linguagem Verilog, com a exceção de um requisito da ferramenta que utilizava um módulo VHDL.

2.3.3 Design Flow

O processo de aceleração de uma aplicação em *hardware* segue uma metodologia. Primeiramente o algoritmo é codificado numa linguagem de *software*, desta maneira o algoritmo desenvolvido é depurado e validado. Seguidamente são identificados núcleos de processamento com poucas dependências de dados que podem ser implementados em *threads*, utilizando um modelo de programação *multi-threaded* e, desta maneira, paralelizar a aplicação *software*. A aplicação é depois analisada de forma a identificar as componentes de código de computação intensiva, as quais serão as candidatas à migração para *hardware*. Selecionadas as componentes a migrar, estas são implementadas em aceleradores *hardware* usando linguagens HDL. Os aceleradores serão integrados na restante aplicação prosseguindo-se para a validação do sistema completo. Isto é um processo iterativo, pelo que, se as métricas requeridas não forem satisfeitas, o desenvolvimento volta para fases anteriores até que todas as métricas sejam satisfeitas.

A Figura 2.6 mostra as várias fases do processo de aceleração de uma aplicação em *hardware*, que serão descritas mais detalhadamente em seguida.

Modelação da aplicação em software

Nesta fase a aplicação é implementada numa linguagem de programação software funcional e flexível. Linguagens como o C/C++ são as escolhas de eleição na programação de software embebido, através das quais é possível criar código genérico e, posteriormente, portá-lo para diferentes plataformas através da *cross-compilation*.

O algoritmo é rapidamente validado na plataforma *host*, onde podem ser facilmente utilizadas ferramentas de depuração como o GDB, e até interfaces gráficos sobre essas ferramentas como o GNU Data Display Debugger (GNU DDD).

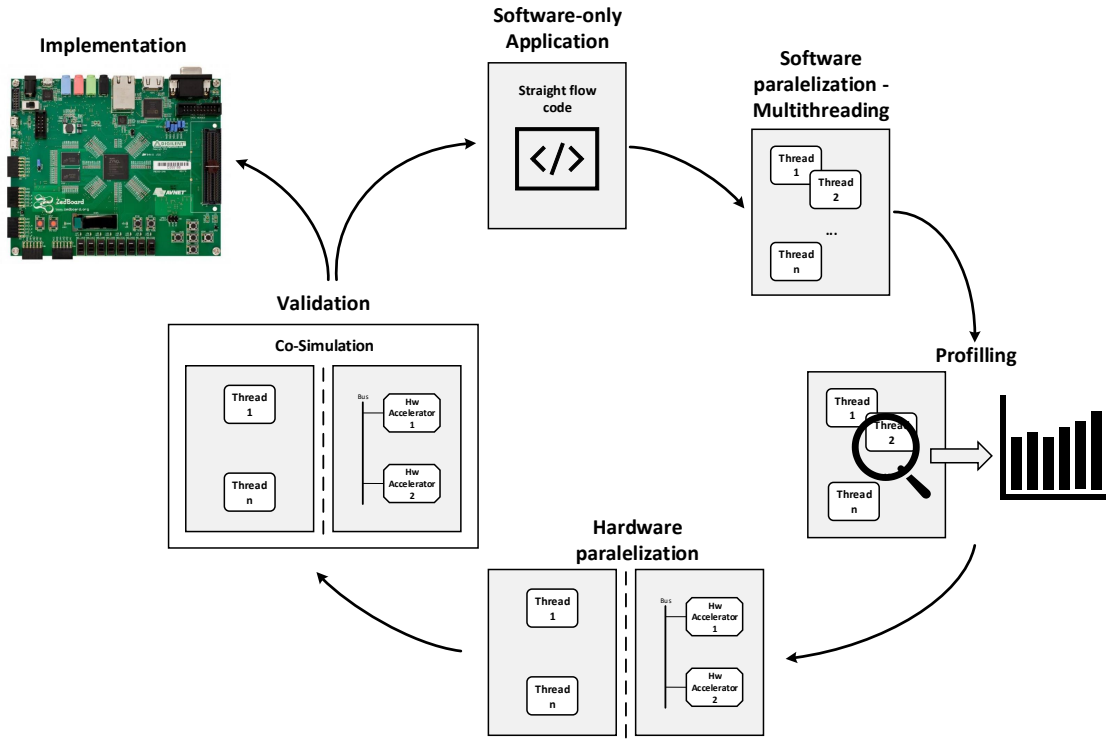


Figura 2.6: Metodologia aceleração de uma aplicação em *hardware*

Paralelização em Software

Depois de o algoritmo estar validado, este deve ser paralelizado em software, isto é, dividido em várias tarefas e implementado em várias *threads* segundo um modelo *multi-threaded*, usando por exemplo a POSIX API, vastamente utilizada em sistemas Linux. As *threads* devem ser porções de código de computação intensiva e, de preferência, com poucas dependências de dados. Isto para minimizar a partilha e, consequentemente, minimizar a necessidade de utilização dos inevitáveis mecanismos de sincronização entre *threads*.

Em sistemas *singlecore* esta paralelização é virtual, visto que apenas uma *thread* pode ser executada no mesmo instante. Já em sistemas *multicore* várias *threads* podem executar simultaneamente, em vários *cores*.

Profiling

Profiling é o acto de analisar a execução de uma aplicação de forma a caracterizá-la. O resultado de um *profiling* é uma análise estatística da execução do programa, onde se pede observar, por exemplo, o número de vezes que uma função foi execu-

tada e a percentagem de processamento que esta utilizou. Desta forma é possível escolher as *threads* potencialmente candidatas a serem migradas para aceleradores *hardware*. Normalmente o processo de *profiling* é feito na plataforma *host*, visto que a diferença nos resultados entre *host* e o *target* não é suficientemente relevante para exigir a necessidade de um *profiling* na plataforma *target*.

Paralelização em Hardware

Selecionada a *thread*, ou as várias *threads* a serem migradas para hardware estas são implementadas como co-processadores dedicados com o auxílio de linguagens HDL, geralmente Verilog ou VHDL. Ferramentas como o Xilinx ISE fornecem um ambiente amigável para o design de hardware e integram ferramentas para síntese e *routing* do código HDL. Xilinx ISE integra um repositório de Intellectual Property (IP) que podem ser utilizados no design do acelerador, integra também um simulador de HDL, o ISE Simulator. Simuladores de HDL, como o ISE Simulator da Xilinx ou ModelSim da MentorGraphics permitem simular o código HDL a nível comportamental, ou até mesmo a nível RTL e mostram-se muito úteis na validação e depuração do código.

Codificado e validado o acelerador é necessário integrá-lo com a restante aplicação. Isto traduz-se, num design CPU+FPGA, na integração do acelerador no restante SoC e na adaptação da aplicação inicial *software-only*. Para integrar o acelerador no SoC este é ligado ao barramento do sistema, desta forma fica um periférico do CPU. Nesta fase o Xilinx Platform Studio (XPS) torna-se útil pois auxilia o hardware designer criar, configurar e conectar sistemas baseados em CPU+FPGA. Para concluir o processo de integração do acelerador a aplicação inicial é adaptada para interagir com o acelerador, o processamento que a *thread* acelerada executava em software é agora delegado para o respetivo co-processador hardware.

Validação

A validação do sistema é indispensável, quanto melhor e mais incisiva for a validação mais problemas irão ser encontrados e resolvidos nesta fase, em vez de serem encontrados na fase de implementação, onde as consequências podem ser devastadoras.

Neste tipo de sistemas, que integram domínios heterogéneos, nomeadamente o

software e o hardware, o processo de validação torna-se difícil e demorado. A utilização de uma metodologia de co-simulação acelera este processo e permite validar integralmente o sistema. Num ambiente de co-simulação cada domínio é simulado na ferramenta especializada. As diferentes ferramentas interagem entre si de modo a simular as interações entre os diferentes domínios. O QEMU (Quick EMUlator) é uma ferramenta de emulação e virtualização que permite simular não só a execução de um determinado código mas também a plataforma onde este está a ser executado. Um exemplo de co-simulação pode ser a utilização do QEMU para simular a plataforma *target*, com o código a executar sobre ela enquanto o ModelSim simula o hardware sintetizado, as ferramentas comunicam entre si nas transições entre domínios.

Implementação

Esta é a fase final do processo de aceleração da aplicação. Consiste na implementação e teste da aplicação na plataforma *target*. Aqui é verificado se todos os requisitos foram satisfeitos, finalizando o processo de aceleração. Caso não sejam verificadas todas as métricas o processo volta para fases anteriores, que torna isto num processo iterativo. A fase de retorno do processo depende das alterações que serão necessárias fazer, por exemplo pode ser necessária a alteração do algoritmo inicial, voltando para a fase de modelação da aplicação software ou apenas alterar os processos a migrar para hardware, voltando para a fase de paralelização em hardware.

2.4 *Middleware*

Em termos gerais, *middleware* é qualquer software que não faz parte do Sistema Operativo, *device drivers* ou software aplicacional. Num sistema embebido, um *middleware* pode atuar sobre *device drivers*, sobre o Sistema Operativo ou até integrar o próprio Sistema Operativo [30], como pode ser observado na Figura 2.7.

Middleware assiste uma aplicação a interagir com outras aplicações, redes, hardware e/ou sistemas operativos. O *middleware* assiste programadores abstraindo-os da complexidade que pode advir das várias ligações entre os sistemas. Providencia ferramentas para melhorar a qualidade do serviço, segurança, troca de dados, serviços de controlo, etc., que podem ser invisíveis ao utilizador [6].

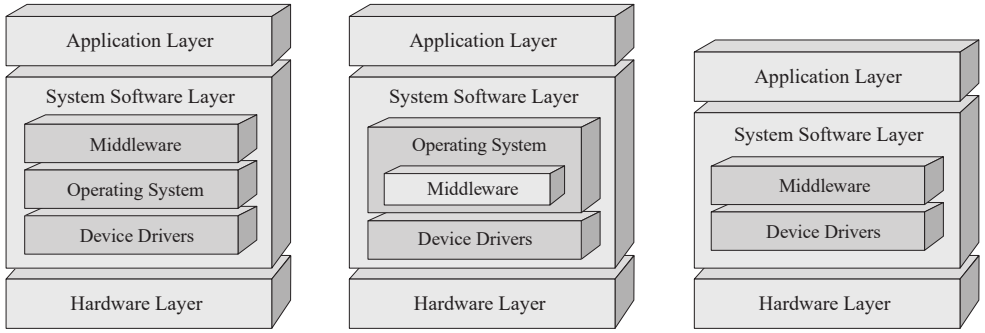


Figura 2.7: *Middleware* em modelos de Sistemas Embebidos [30]

Umas das vantagens na utilização de um *middleware* é a redução da complexidade das aplicações através da centralização de software que normalmente estaria replicado em várias componentes da aplicação. Como por exemplo, o software de interface com um determinado elemento da aplicação existe apenas uma vez, centralizado no *middleware*, ao invés de cada componente da aplicação possuir uma réplica de software para interação com cada um dos restantes componentes. No entanto a inclusão de um *middleware* acrescenta *overhead* de execução que pode ter impacto na performance da aplicação final.

Os diferentes tipos de *middleware* são: *message oriented middleware* (MOM), *object request brokers* (ORBs), *remote procedure calls* (RPCs), database acess e protocolos de rede sobre a camada *device driver* e sob a camada da aplicação do modelo OSI. No entanto os tipos de *middleware* generalizam-se em duas categorias: *general-purpose*, que são concebidos para uma grande variedade de dispositivos, como protocolos de rede, e algumas máquinas virtuais (por exemplo a JVM); e *market-specific*, que são concebidos para uma família particular de sistemas embebidos, como por exemplo o software de uma TV que opera sobre um sistema operativo ou uma JVM [30].

Nesta dissertação foi desenvolvido um *middleware* do tipo Message Oriented, específico para aplicações aceleradas em hardware. Este generaliza a troca de dados entre entidades de processamento em software (*threads*) e em hardware (aceleradores).

2.5 Publish Subscribe Pattern

As comunicações síncronas e ponto a ponto tendem a criar aplicações rígidas, estáticas, dificilmente manipuláveis e adaptáveis em outras situações para além da que foram especificamente implementadas. Em modelos de interação *loosely coupled*, como o paradigma *publish-subcribe*, em que os seus componentes estão aptos a interagir mesmo sem conhecendo informações uns sobre os outros, mostram-se assim bastante versáveis e portáveis, facilitando a adaptação de aplicações em outros ambientes ou na criação de modelos transparentes de programação [24].

O padrão *publish-subcribe* é um mecanismo de troca de mensagens onde há dois atores, os *publishers*, que enviam dados, e os *subscribers*, que recebem dados. A principal característica deste padrão é a maneira como os dados fluem entre os atores. Os *publishers* apenas publicam dados para uma unidade central. Os *subscribers* fazem subscrições a essa unidade central, usando um dos vários modelos de subscrição, e, quando há uma correspondência, estes podem, ou não, ser notificados e têm acesso aos dados. Um dos modelos de subscrição é o *topic-based*, onde as mensagens são associadas a um tópico e os *subscribers* subscrevem esse tópico. Outro modelo é o *content-based*, em que o *subscriber* recebe mensagens que respeitem as condições definidas por ele, como por exemplo restrições de tamanho ou de primeira letra.

A Figura 2.8 representa um exemplo de um sistema *publish-subscribe* baseado em tópicos (*topic-based*), como se pode observar os *publishers* publicam para os tópicos, sem restrições, um *publisher* pode publicar para vários tópicos, os *subscribers* subscrevem um ou mais tópicos, recebendo os dados presentes nesses tópicos.

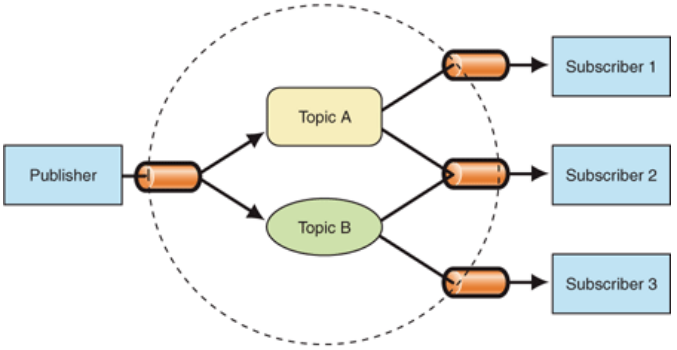


Figura 2.8: Exemplo padrão *publish-subscribe topic-based* [27]

2.6 ReconOS

O ReconOS integra um sistema operativo, um modelo de programação e uma arquitetura de sistema que fornece um conjunto de serviços funções a serem executadas em software e hardware e uma interface *stardard* para inclusão de aceleradores hardware dedicados [34].

A sua primeira versão foi desenvolvida por Enno Lübbers no âmbito da sua tese de doutoramento, publicada em 2010. Desde então o ReconOS tem sido melhorado e encontra-se atualmente na versão 3.1 [34].

ReconOS usa o modelo de programação *multithreading* e extende o sistema operativo *host* suportando hardware *threads*. Deste modo permite a interação entre hardware e software *threads* usando os mesmos mecanismos *standard* do sistema operativo, como por exemplo semáforos, *mutex*, *message queues*.

Enno Lübbers desenvolveu um modelo de programação *multithreading* unificado para software e hardware *threads* associado a um modelo de implementação de *threads* em hardware. Interagindo com os sistemas operativos suportados, o eCos e o Linux, estes modelos permitem aos módulos hardware o acesso aos mesmos serviços de comunicação e sincronização que os em software. ReconOs especifica uma metodologia de desenho de *hardware threads*, fornece um módulo em hardware de interface com o sistema operativo (o OSIF) e uma biblioteca em software que permite a sincronização e comunicação com as hardware *threads* do modo semelhante às *threads* em software. As *hardware threads* são representadas em software por uma *software thread* de controlo, a *delegate thread*, que atua como *proxy* executando as *system calls* em prol da *hardware thread* que representa [25].

A Figura 2.9 apresenta um esquema geral do ReconOS, onde podem ser observados os seus vários constituintes.

Hardware thread

No ReconOS, aos módulos hardware é acrescentada uma máquina de estados (FSM), esta implementa funções em VHDL que podem ser usadas para interagir com objetos do sistema operativo como semáforos, *mutexes* ou memória partilhada. Esta FSM está conectada ao módulo OSIF e pode, transparentemente, iniciar processos de comunicação com a respetiva *delegate thread* e, consequentemente, com

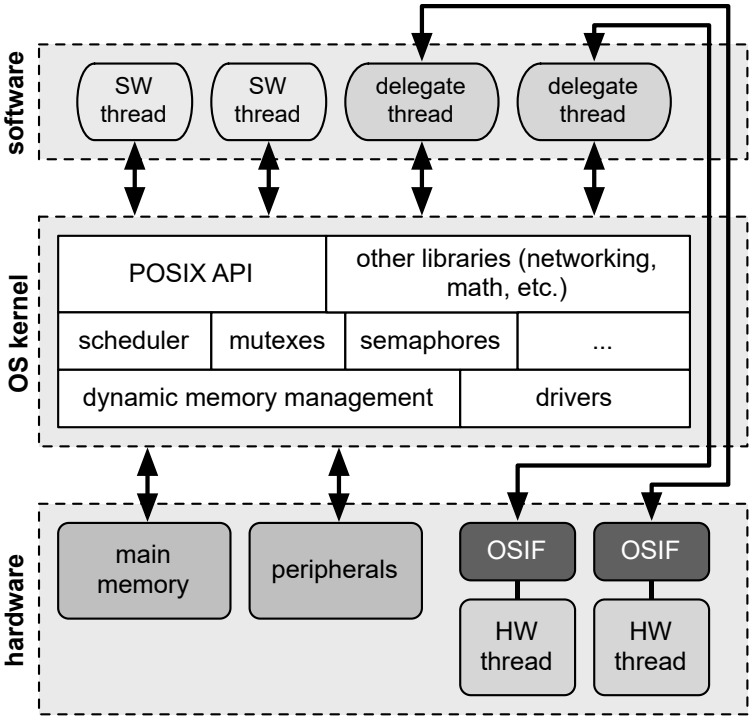


Figura 2.9: Visão geral dos componentes do ReconOS [25]

o sistema operativo. Juntamente com o OSIF, esta FSM permite o controlo sobre a execução da *hardware thread* [25].

Operating System Interface (OSIF)

Para que as *hardware threads* tenham, do mesmo modo que as *software threads*, acesso aos objetos do sistema operativo existe no ReconOS o módulo OSIF. O OSIF é um módulo hardware que executa as *system calls* das *hardware threads*, quer executando-as, se possível, diretamente em hardware ou delegando-as para o CPU. Também é responsável pela interface de controlo entre o OS e o hardware [25].

Delegate Threads

Para cada *hardware thread* é criada uma *delegate thread*, esta é uma *thread* em software que executa as *system calls* do sistema operativo em prol da sua correspondente *hardware thread*. A *delegate thread* permite a execução em software de uma *system call* iniciada em hardware e contribui também para a transparência do

sistema, uma vez que o sistema operativo vê a *hardware thread* como uma *software thread* regular, não distinguindo as *system calls* iniciadas em software ou hardware [25].

Capítulo 3

Ambiente de Desenvolvimento

Neste capítulo irá ser apresentada a plataforma *hardware* utilizada no desenvolvimento desta dissertação.

Este trabalho visa a criação de um serviço de suporte ao *design* de aplicações híbridas (CPU+FPGA), baseadas em Linux, deste modo era mandatório a seleção de uma plataforma que suportasse estes tipos de sistemas: seria necessário um chip híbrido com área programável suficiente e com pelo menos um processador, *harwired* preferencialmente.

Seguidamente irá ser apresentada detalhadamente a plataforma escolhida bem como as várias ferramentas necessárias para lidar com esta tecnologia, desde a criação do SoC, à inclusão do sistema operativo Linux e sua *toolchain*.

3.1 Plataforma Xilinx XUPV2P

A plataforma XUPV2P, apresentada na Figura 3.1, é uma plataforma de *hardware* avançada, incluída no programa da Digilent, uma corporação que visa providenciar, com um baixo custo, tecnologias avançavas de engenharia eletrónica, para fins educativos [40].

A plataforma integra uma vasta quantidade de periféricos *hardware*, tais como:

- **Xilinx XC2VP30:** É um chip FPGA da familia Virtex-2 Pro com uma vasta gama de recursos e grande quantidade de área programável, como é mostrado na Tabela 3.1, e ainda dois processadores *hard-core* PowerPC 405.

- **Porto para memória DDR SDRAM:** Onde podem ser aplicadas memórias com capacidade de até 2GB;
- **Portas série:** Três interfaces de porta série, uma RS-232 conectada a uma ficha stardard DB-9, que pode ser utilizada para comunicação com um pc *host*. E ainda duas interfaces PS/2, que permitem a ligação de um rato e um teclado.
- **Compact Flash:** Permite a utilização de uma memória não volátil do tipo CF, onde pode ser armazenada o *bitstream* que programa a lógica FPGA, a imagem binária de código a correr e ainda ser utilizado como sistema de ficheiros do Linux. A sua interface é feita através do controlador Xilinx System ACE.
- **Interface Ethernet:** A plataforma contem um *transceiver* compatível com o protocolo IEEE Fast Ethernet, que suporta aplicações que user os padrões 100BASE-TX e 10BASE-T.
- **Saída XSGA:** É incluido na plataforma um DAC de vídeo e um conector de 15 pins D-sub, que permite a interface com monitores VGA.
- **Outros:** A plataforma possui ainda uma porta USB, com interface JTAG, vários portos de expansão, leds, *switches*, um codec de aúdio AC97, entre outros.

Tabela 3.1: Caracteristicas do chip FPGA Xilinx XC2VP30 [40]

Caracteristica	**XC2VP30**
Células Lógicas	30 816
Slices	16 696
Blocks RAM (kb)	2448
Blocos multiplicadores 18-bit	136
DCM	8
PowerPC 405 cores	2
Multi-Gigabit Transceivers	8

3.1.1 Unidade de processamento

Como foi referido o chip FPGA XC2VP30 integra dois processadores PowerPC 405 *hard-wired*. O PowerPc 405 é um processador de 32 bits, com arquitetura RISK, com alta performance e baixo consumo desenvolvido pela IBM [17]. O

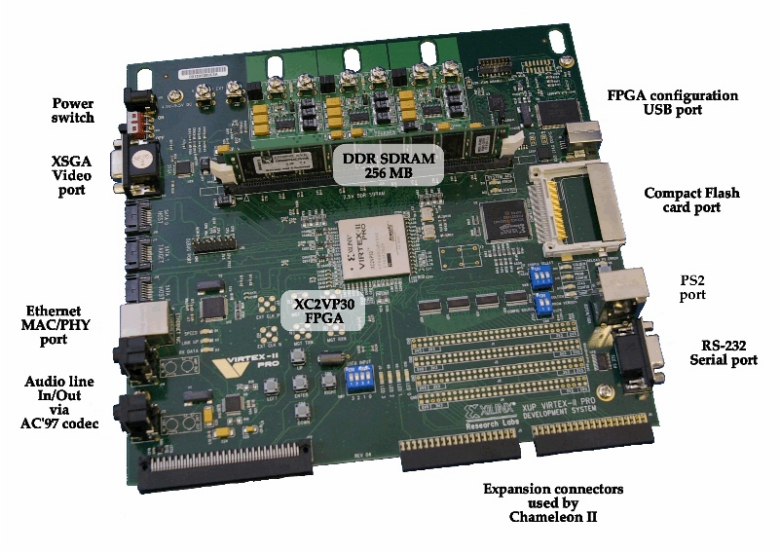


Figura 3.1: Plataforma XUPV2P [40]

processador consiste num *pipeline* de cinco estágios, com unidades de *cache* de instruções e data separadas, com uma Memory Management Unit(MMU), entre outras funcionalidades. Na Figura 3.2 podem-se observar os vários componentes do processador:

- **Unidades cache:** Caches de de instruções (ICU) e de dados (DCU) separadas permitem acesso concurrente, minimizando os *pipeline stalls*. Estas unidades podem ter capacidades de 0KB até 32KB.

- **Memory Management Unit:** Esta unidade providencia traduções de endereços lógicos até 4GB em endereços físicos. Os Tranlation Lookaside Buffers (TLB) são o recurso *hardware* de controlo da translação e proteção. A MMU divide a memória lógica em páginas (de 1KB a 16MB) e estas são representadas como entradas nos TLB.

- **Timers:** O processador possui um contador base de 64 bits incrementado prlo *clock* do CPU ou por um *clock externo*. Possui ainda 3 unidades de temporização: o *programmable interval timber* (PIT), um *timer* de 32 bit com o propósito de gerar atrasos; o *fixed interval timer* (FIT), com o propósito de gerar interrupções periódicas e o *watchdog timer* que pode gerar *resets* ao sistema.

- **Barramentos:** O processador possui interface com o Processor Local Bus (PLB), com 32 ou 64 bits no barramento de endereços e 32, 64 ou 128 bits no barramento de dados. Possui ainda interface com o Device Control Register

Bus (DCR), com 10 bits no barramento de endereços e 32 bits no de dados. Este é acedido com as instruções **mfdcr** e **mtdcr**.

- **Interrupções**: Interface com um controlador de interrupções externo.

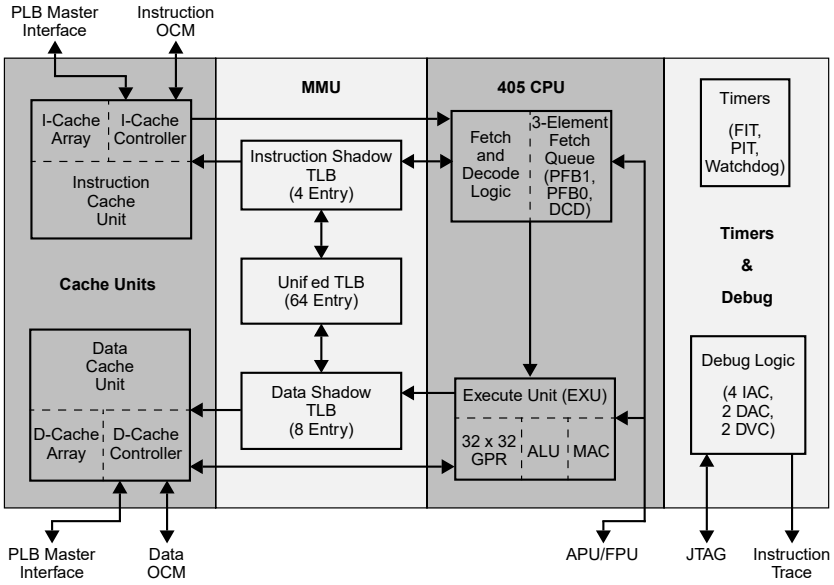


Figura 3.2: Diagrama de blocos do PowerPC 405 [17]

3.2 Xilinx Embedded Development Kit

O Xilinx Embedded Development Kit (EDK) é um conjunto de ferramentas e IP que suporta o design de sistemas embebidos baseados nos chips FPGA da Xilinx. Engloba ferramentas, documentação, biblioteca de IP reutilizáveis para projetar sistemas com um processador embebido suportado pela plataforma. Para além disso integra ambientes para desenvolvimento e síntese de software e hardware para a plataforma alvo [42].

A Figura 3.3 apresenta o ciclo de desenvolvimento do Xilinx EDK que engloba o desenvolvimento de software, hardware, bem como a gestão dos componentes que constituem o sistema e as respetivas interligações. Podem ser observadas as três ferramentas principais:

- **Xilinx Platform Studio (XPS):** Ferramenta gráfica que permite ao programador desenhar, conectar e configurar sistemas com um processador integrado. Incorpora o Base System Builder (BSB) que cria um SoC básico com os componentes essenciais. O programador pode depois customizar este

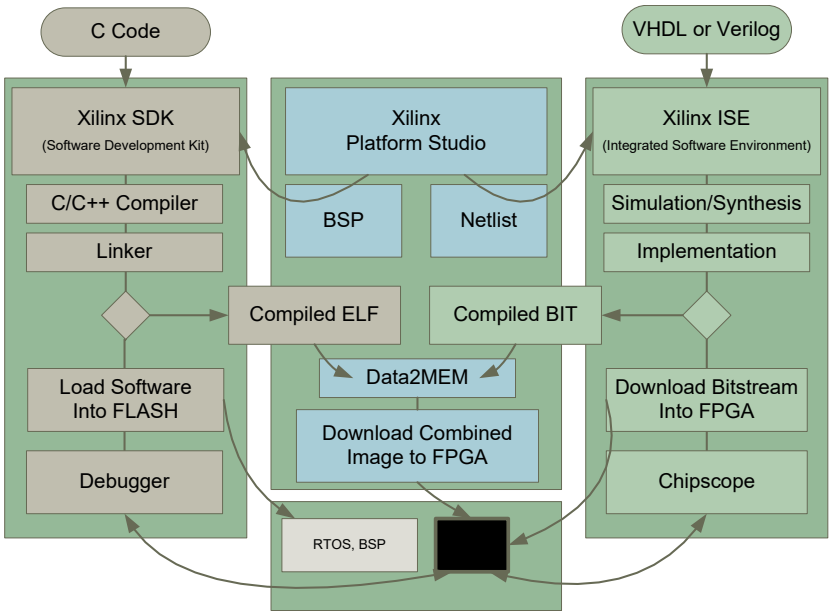


Figura 3.3: Ciclo de desenvolvimento do Xilinx EDK [13]

SoC adicionando os periféricos necessários à sua aplicação. No projeto desta dissertação esta ferramenta foi utilizada para construir o SoC baseado no processador PowerPC 405, ao qual foram adicionados os periféricos customizados ligados ao processador através do barramento do sistema;

- **Xilinx Integrated Software Environment (ISE):** Esta ferramenta inclui um IDE para programação em linguagens de descrição de hardware. Inclui ferramentas para síntese de código HDL e também uma ferramenta de simulação, o ISE Simulator. Possui também a possibilidade de interface com outras ferramentas de simulação, como o ModelSim. Integra a ferramenta para fazer o download do *bitstream* criado para a respetiva plataforma FPGA bem como interface com a ferramenta de validação Chipscope. Nesta dissertação, o Xilinx ISE foi utilizado para criar e validar os vários componentes criados em hardware.

- **Xilinx Software Development Kit (SDK):** Esta ferramenta fornece um ambiente para desenvolver software para aplicações baseadas nas plataformas Xilinx. Inclui ferramentas para compilação e download do binário para a plataforma bem como ferramentas para debug do código código executado. No projeto desta dissertação esta ferramenta foi utilizada apenas para a criação de código de teste de alguns componentes hardware, visto que o código da aplicação era desenvolvido sobre o ambiente Linux.

3.3 Buildroot

Buildroot é uma ferramenta open-source que automatiza o processo de construção de um sistema Linux para um sistema embebido utilizando *cross-compilation* [7]. É constituído por um conjunto de Makefiles e *patches* que automaticamente fazem download, extraem e compilam os vários integrantes de um sistema Linux:

- A *toolchain* necessária para a *cross-compilation*;
- O sistema de ficheiros;
- A imagem do kernel Linux;
- O bootloader para a plataforma *target*.

O Buildroot contém várias pré-configurações para uma vasta quantidade de plataformas, no entanto também permite configurar mais refinadamente o sistema utilizando o sistema de configuração Kconfig, semelhante ao utilizado pelo Linux. A imagem do Linux compilada e o sistema de ficheiros criado são uma versão base, no entanto, através das configurações, é possível adicionar suporte a aplicações de rede, *codecs* de áudio e de vídeo, *frameworks* de programação gráfica, etc. A Figura 3.4 mostra uma das possíveis configurações gráficas do Buildroot, o `menuconfig`. No projeto desta dissertação foi utilizado para construir a *toolchain* a imagem do Linux OS e o sistema de ficheiros.

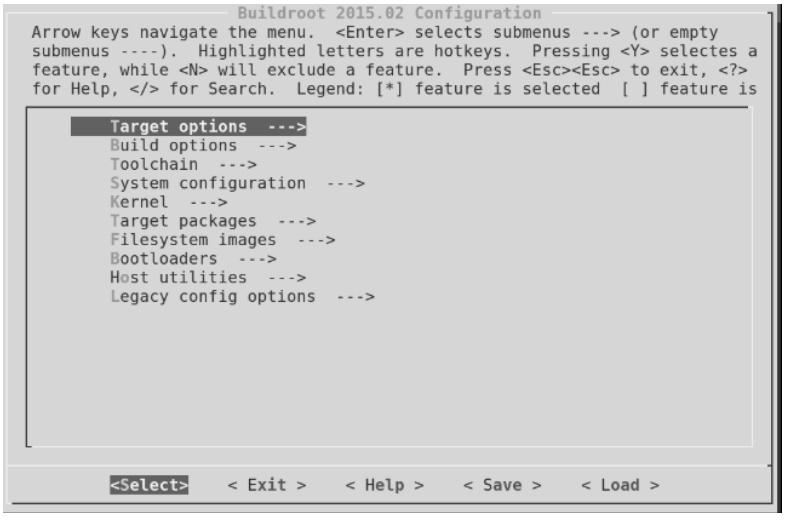


Figura 3.4: Menu de configuração do buildroot

Capítulo 4

Middleware for Linux Hybrid Applications

O MiLHA (***Mi**ddleware for **L**inux **H**ybrid **A**pplications*), é o serviço de suporte à aceleração de aplicações baseadas em Linux em hardware desenvolvido no âmbito desta dissertação. O MiLHA fornece um modelo de programação paralelo que integra software e aceleradores através de mecanismos de interface direta entre eles, abstraindo o programador de toda a complexidade que isso implica.

O modelo de programação proposto baseia-se num mecanismo de *Inter-Thread Communication* integrado baseado no *Publish Subscribe Pattern*, em que entidades comunicam através de publicações e subscrições em tópicos (Secção 2.5). Este modelo de programação dá suporte à integração de aceleradores em *hardware* na aplicação, minimizando as alterações necessárias a fazer na aplicação *software-only* quando esta é acelerada em *hardware*.

Neste capítulo será apresentada a estrutura geral do MiLHA e, em seguida, explicado detalhadamente cada uma das três partes distintas que o compõem. Os conteúdos serão apresentados segundo uma abordagem *top-down*: iniciando na API em *user level*, seguindo com os serviços implementados em *kernel level* e terminando com o *hardware* sintetizado na FPGA.

4.1 Estrutura do MiLHA

O MiLHA é composto por três componentes fundamentais. Na Figura 4.1 está representada a sua estrutura, onde podem ser observadas os seguintes componentes:

- **MiLHA API:** Implementa o modelo de programação paralela do MiLHA. É a API utilizada pelo programador para criar a aplicação;

- **Camada de interface com o hardware**: Presta serviços de comunicação a nível do Kernel Linux entre o software o os aceleradores;

- **Camada de integração do acelerador:** Tem como função auxiliar a integração do acelerador hardware, desenvolvido pelo utilizador, num ambiente suportado pelo MiLHA.

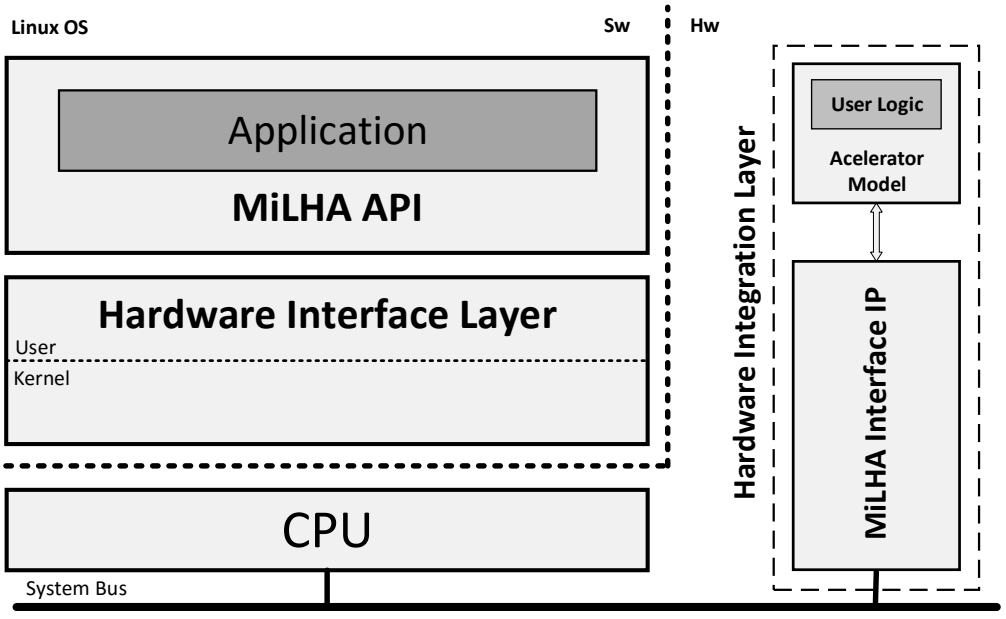


Figura 4.1: Estrutura do MiLHA

A MiLHA API encaminha os dados para os aceleradores hardware através de pedidos à `Hardware Interface Layer`, que os executa através de escritas `MiLHA Interface IP` ligado ao barramento do sistema. Os dados são então encaminhados para o acelerador.

4.2 Modelo de programação no MiLHA

No final da fase de paralelização com aceleração em hardware de uma aplicação em hardware, concluída a integração dos aceleradores no SoC, é necessária a adaptação da aplicação inicial (*software-only*) para fazer a interface com os aceleradores. Com o propósito de auxiliar esta fase o MiLHA fornece um modelo de programação *multithreading* que suporta a interface com os aceleradores, criando uma interface transparente entre as diferentes formas de processamento: software (*threads*) ou hardware (aceleradores).

Ao utilizar o MiLHA e seguindo o *design-flow* apresentado na Secção 2.3.3, na fase de paralelização da aplicação em software esta é implementada utilizando o modelo de programação *multithreading* fornecido. Seguidamente é feito o *profiling*, é selecionada a *thread* ou as *threads* a migrar para hardware, estas devem ser implementadas segundo o modelo de acelerador definido pelo MiLHA e integradas no SoC utilizando a camada de integração do hardware do MiLHA. Finalmente, no momento da adaptação da aplicação inicial para a inclusão dos aceleradores, o modelo de programação utilizado irá suportar os aceleradores, minimizando as alterações necessárias na aplicação inicial para, no limite, uma: a alteração da unidade de processamento de *thread* para acelerador.

Seguidamente será apresentado o MiLHA *main object*, que representa a entidade máxima do MiLHA e que inclui a API de interface com o programador; o objeto `Task`, que representa de forma transparente as unidades de processamento (hardware ou software) e, por fim, o mecanismo de comunicação entre `Tasks` fornecido pelo MiLHA.

4.2.1 Objeto principal MiLHA

Os serviços disponibilizados pelo MiLHA estão concentrados num objeto principal que servirá de interface com o programador e concentra toda a informação sobre as entidades presentes na aplicação. O MiLHA é representado através da classe `CMiLHA` que implementa os métodos de criação das entidades básicas do modelo de programação no MiLHA, nomeadamente:

- Criação das unidades de processamento: `Tasks`;
- Criação das unidades básicas de comunicação entre Tasks: `Topics`.

No diagrama de classes da Figura 4.2 é apresentada a classe `CMiLHA` juntamente com as classes `CTask` e `CTopic`, que representam as `Tasks` e os `Topics`, respetivamente.

`CMiLHA` segue o padrão *singleton*, tornando-o assim único e facilmente acessível nos vários *scopes* do programa. Contém, para além dos métodos referentes à criação das entidades do modelo de programação, métodos para a utilização independente da camada de interface com o hardware e guarda internamente referências para todos os objetos relativos ao MiLHA através de três listas ligadas: `taskList`, `topicList` e `hwDevList`.

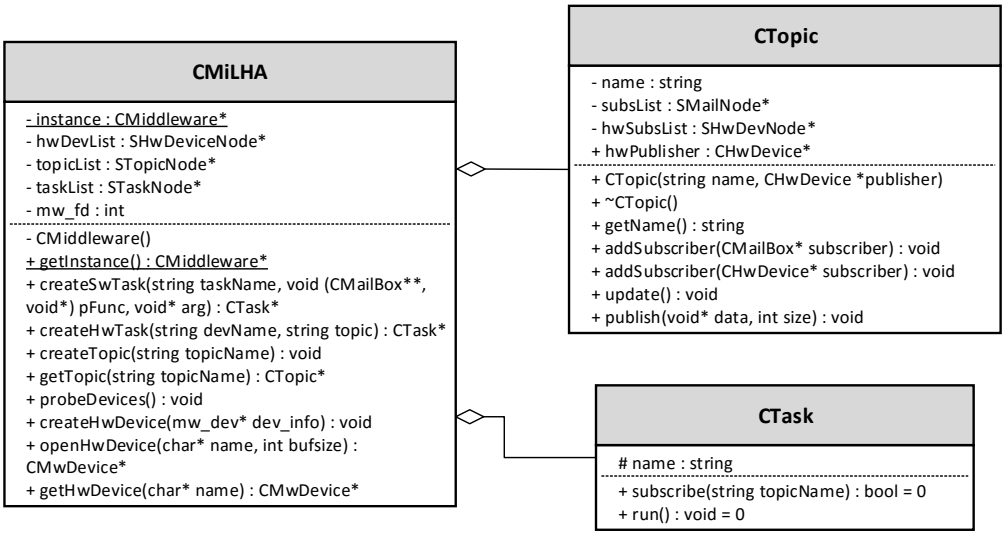


Figura 4.2: Diagrama de classes do modelo programação MiLHA

4.2.2 Tasks

No ambiente MiLHA, as `Tasks` são a representação das unidades de processamento de dados. Um mesmo tipo de objeto representa, de forma transparente ao programador e à aplicação, os dois tipos tipos de processamento existentes:

- **Processamento em software:** *MiLHA Thread* (`Software Task`), que executa no paralelismo virtual do CPU;
- **Processamento em hardware:** hardware *device* (`Hardware Task`), que executa no paralelismo real do acelerador FPGA;

As `Tasks` são representadas pela classe `CTask`, esta é uma classe abstrata da qual derivam duas outras classes: a sua especialização em software traduz-se numa

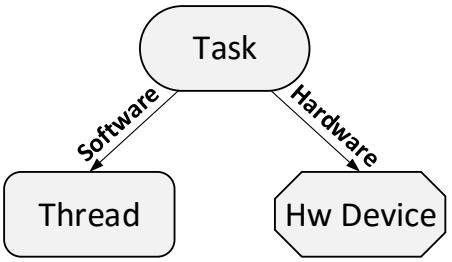


Figura 4.3: Os dois tipos de task existentes.

thread, e é representada pela classe `CThread`; a sua especialização em hardware traduz-se num acelerador hardware, representado pela classe `CHwDevice`. A classe `CTask` contém dois métodos (`run` e `subscribe`) que serão implementados por cada uma das classes filhas. O método `run` dará início à execução da `Task` e o `subscribe` fará a subscrição de um tópico para de comunicação entre `Tasks`.

Nesta Secção será apresentado o processo para a criação de `Tasks` e na Secção 4.2.3 será apresentada interface para o mecanismo de comunicação entre `Tasks`.

Software Task - *MiLHA Thread*

A `MiLHA Thread` (`Software Task`) é a representação da entidade de processamento em software do MiLHA. Esta unidade de processamento resultará numa POSIX *thread* que executa no paralelismo virtual do processador. Num sistema MiLHA cada *thread* será representada por um objeto que incluirá os mecanismos de gestão da *thread* e os mecanismos de comunicação entre *threads*.

A Figura 4.4 apresenta o diagrama de classes relativo ao objeto *Thread*, este é representado pela classe `CThread`, classe filha de `CTask`. Cada *Thread* irá conter um array de `Mailboxes`, representadas pela classe `CMailBox`. As *Mailboxes* pertencem ao mecanismo de comunicação entre *threads* e é através deste objeto que a *thread* terá acesso aos dados que lhe são destinados. Os dados presentes na *Mailbox* estão armazenados num contentor do tipo *buffer* circular, representado pela classe `CRingBuffer`.

A criação de uma `MiLHA Thread` deverá ser realizada através do objeto principal MiLHA (classe `CMiLHA`), através do método `createSwTask`. A função recebe como argumentos o nome da *thread*, a função a executar e um argumento opcional que poderá ser usado para passagem de parâmetros.

A função a executar terá de ser do tipo:

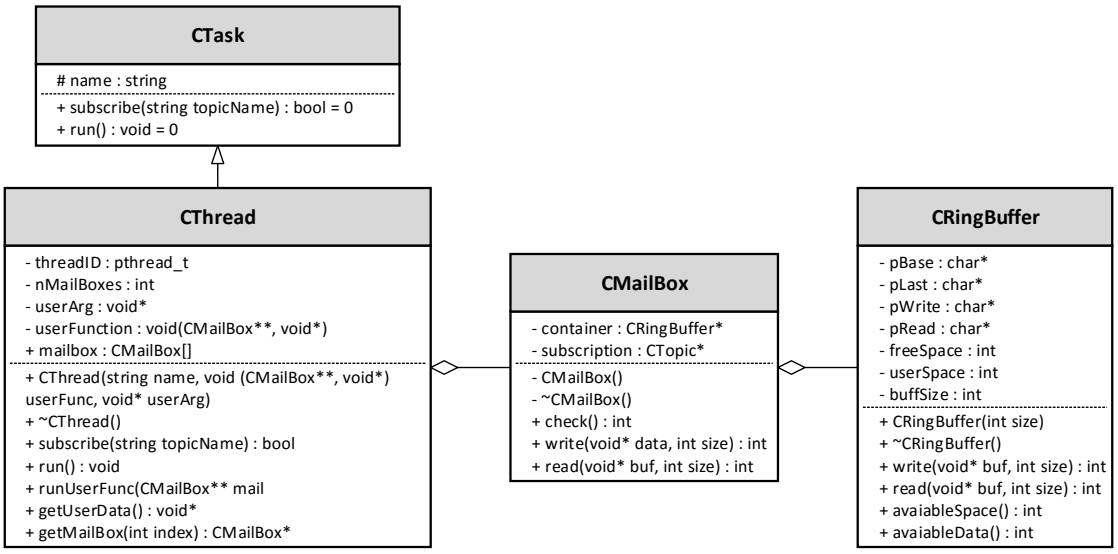


Figura 4.4: Diagrama de classes UML para CThread

```
void UserFunction(CMailBox* mailBox[], void* userArg){...}
```

Esta função será executada pela *thread de software*, para a criação da função o programador deve usar o *template* disponível na classe CThread (Anexo A).

Dentro da função o programador terá acesso ao *array* de `Mailboxes` da respetiva `Thread`. Cada `Mailbox` irá conter os dados respetivos ao tópico subscrito. O segundo argumento da função será o parâmetro opcional fornecido pelo programador. No final, a função `createSwTask` irá retornar um objeto do tipo `CTask` e, através do seu método `run()`, se poderá iniciar a execução da *thread*.

Hardware Task - HwDevice

Num ambiente MiLHA o processamento em hardware é representado por uma `Hardware Task`.

O modelo de programação apresentado utiliza diretamente a camada de interface com o hardware do MiLHA. Este fornece acesso ao acelerador através de um `Hardware Device Proxy`, representado por um objeto da classe `CHwDevice` (apresentado em 4.3.1).

Para a criação de uma `Hardware Task` deverá ser utilizado o objeto principal MiLHA (classe `CMiLHA`), através da função `createHwTask`. A função recebe dois argumentos:

- ***Path* para o Linux device:** A camada de interface com o hardware do MiLHA publica cada acelerador como um Linux *character device*, o primeiro argumento deverá ser uma *string* com o caminho para esse *character device*, por exemplo "/dev/pqhardware";

- **Tópico de publicação:** O segundo argumento é o nome do Tópico onde todos os dados provenientes do acelerador serão publicados. Pode ser um tópico já existente ou um novo tópico que será criado automaticamente pelo MiLHA.

A função retornará um objeto do tipo `CTask`. Este poderá ser utilizado para configurar interações com outras Task, que o MiLHA se encarrega de executar.

4.2.3 Comunicação entre Tasks

Nesta secção irá ser apresentado o modelo de comunicação entre `Tasks` disponibilizado pelo MiLHA. O MiLHA propõe ao programador que, após a criação das diferentes `Tasks` que compõem a sua aplicação, este defina as relações de troca de dados entre elas através do modelo proposto. Após as relações serem definidas e aplicação iniciar a sua execução o MiLHA encarregar-se-à de fazer chegar os dados às diferentes `Tasks`, independentemente de estas serem `Hardware Tasks` ou `Software Tasks`.

O modelo que permite definir as dependências de dados entre `Tasks` é baseado no *Publish Subscribe Pattern*, introduzido na Secção 2.5. Na Figura 4.5 pode ser observado um simples exemplo em que duas `Tasks`, uma produtora e uma consumidora, comunicam através de um tópico. A `Task` consumidora irá subscrever o tópico fazendo com que todos os dados publicados nesse tópico sejam reencaminhados para ela. Neste caso os dados publicados pela `Task 1` irão ser encaminhados para a `Task 2`.

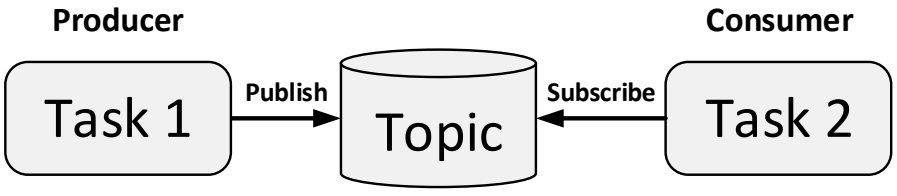


Figura 4.5: Exemplo interação entre duas `Tasks`

O Tópico é a entidade intermediária da comunicação entre `Tasks`, no MiLHA ele

é representado pela classe `CTopic`, apresentada da Figura 4.6. Para a criação de tópicos deve ser utilizado o objeto principal MiLHA (Figura 4.2), através da função membro `createTopic`, que recebe como argumento o nome de um tópico. Para se obter um objeto tópico previamente criado deve ser utilizada a função `getTopic`, que devolve o objeto tópico especificado pelo parâmetro nome no argumento da função.

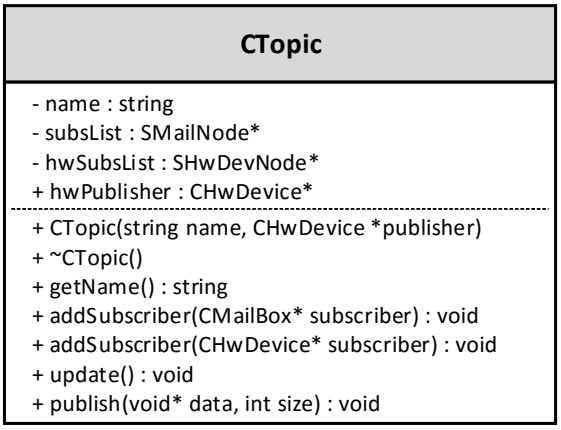


Figura 4.6: Diagrama UML para a classe CTopic

Publicação num Tópico

Uma publicação num tópico traduz-se no envio dos dados publicados para todos os subscritores do tópico.

No software as publicações podem ser feitas através do método `publish` da classe `CTopic`. O objeto tópico é obtido através do método `getTopic()` do objeto principal `CMiLHA`.

Para o caso das `Hardware Tasks` todos os dados provenientes do acelerador serão publicados no tópico indicado na criação da `Hardware Task`, que, neste caso, é sinalizado como contendo um hardware *publisher*. Quando uma `Software Task` subscritora verificar (`check()`) um tópico sinalizado irá ser verificada existência de dados no `Hardware Device Proxy` associado, e caso existam, estes serão encaminhados para todos os subscritores do tópico. Isto porque a publicação em tópicos por parte das `Hardware Tasks` só faz sentido se houver pelo menos uma `Software Task` subscritora.

Subscrição de um Tópico

As `Tasks`, independentemente da sua natureza, pode subscrever tópicos, indicando ao MiLHA que pretendem receber todos os dados lá publicados. A subscrição de tópicos é feita pelo objeto `CTask`, o objeto abstrato que representa os dois tipos de processamento. Deste modo, para o programador, a alteração do tipo de processamento de uma `Task` não irá afetar a relação entre o processamento e o destino dos dados.

No caso das `Software Tasks`, sempre que é feita uma subscrição de um tópico, é criada uma *Mailbox* à qual o tópico é associado e essa *mailbox* é inserida na lista de *Mailboxes* subscritoras do tópico. Cada `Software Taks` poderá ter até dez subscrições e dentro da função de *thread* o programador terá acesso ao *array* de *mailboxes*. Este encontra-se ordenado por ordem de subscrição, sendo os dados da primeira subscrição obtidos na `mailBox[0]`, da segunda subscrição na `mailBox[1]` e assim sucessivamente. O programador pode verificar se existem dados disponíveis nas *Mailboxes*, através do método `check()` e poderá aceder aos dados através do método `read(...)`.

No caso das `Hardware Tasks`, sempre que é feita uma subscrição o respetivo `Hardware Device Proxy`, (objeto `CHwDevice`) será inserido na lista de hardware *devices* subscritores do tópico.

No momento em que a `Task` produtora publica os dados no tópico estes irão ser encaminhados para todos os subscritores. Nas `Software Tasks` subscritoras é utilizada a respetiva `Mailbox`, nas `Hardware Tasks` os dados são enviados para o acelerador através da camada de acesso ao hardware do MiLHA.

A Figura 4.7 apresenta o percurso dos dados publicados num tópico subscrito por duas `Tasks`, uma em software, outra em hardware. Na `Software Task` os dados publicados no tópico são encaminhados para a *mailbox* da `Task`, a função *thread* ao fazer o *check* à *mailbox* verifica a existência de dados e retira-os (`read`), os dados são então processados e posteriormente publicados no tópico seguinte. Na `Hardware Task` quando ocorre a publicação no tópico é utilizada a camada de interface com o hardware do MiLHA e é feito um pedido de escrita no acelerador; os dados são então enviados para o IP em hardware `MiLHA Interface` que os encaminha para o acelerador, são processados e o resultado é enviado para o software novamente.

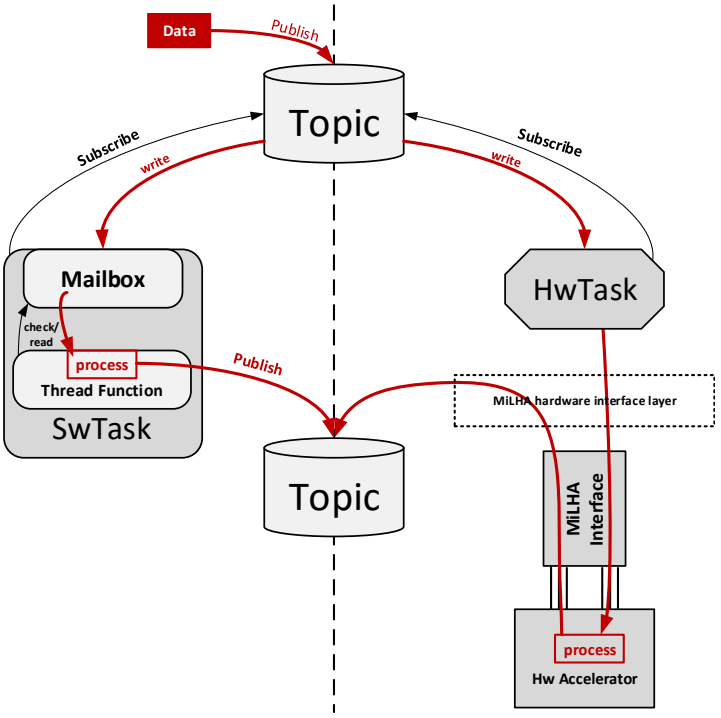


Figura 4.7: Exemplo de publicação de dados num tópico subscrito pelos diferentes tipos de `Tasks`

Comunicação entre `Hardware Tasks`

O modelo de comunicação *Publish Subscribe* é aplicado às relações entre `Taks` do tipo software-software, software-hardware e hardware-software. Para a comunicação hardware-hardware é necessário fazer as ligações entre os aceleradores (envolvidos pelo `User Logic Wrapper`, apresentado em 4.4.1) de modo a ligar a saída de dados de um à entrada de dados do próximo. A Figura 4.8 apresenta um exemplo de ligações entre duas `Hardware Tasks`, neste caso os dados produzidos pela unidade `User Wrapper 1` serão enviados para a unidade `User Wrapper 2`. Com este tipo de ligação os dois aceleradores são ligados em *pipeline* e para o software passam a ser interpretados como um só.

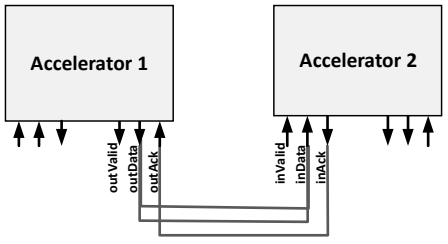


Figura 4.8: Ligações para comunicação entre `Hardware Tasks`

4.3 Interface MiLHA e Aceleradores

Durante aceleração de uma aplicação em hardware, depois de desenvolvidos os aceleradores hardware é necessário desenvolver mecanismos de comunicação entre eles e a restante aplicação. Neste sentido o MiLHA inclui, num ambiente Linux, uma camada de interface direto entre aplicações software e os aceleradores hardware, mapeados em memória. Esta camada está preparada para fazer a interface com o MiLHA Interface IP, no entanto pode fazer interface com um outro qualquer acelerador, desde que este cumpra com os requisitos de interface.

Como se pode observar na Figura 4.9, a camada de interface com o hardware está dividida em 2 partes: um módulo a nível do Kernel, o `MiLHA Kernel`, e uma API a nível da aplicação, a `Hardware Interface API`.

Na `Hardware Interface API` existem objetos que representam os aceleradores hardware, os `Hardware Device Proxy`. No `MiLHA Kernel` existe um Linux *device node* principal (`/dev/mw_main`), um Linux *device node* por cada um dos aceleradores, duas kernel *threads*, uma para executar pedidos de leitura outra para executar pedidos de escrita, e um *handler* de interrupções.

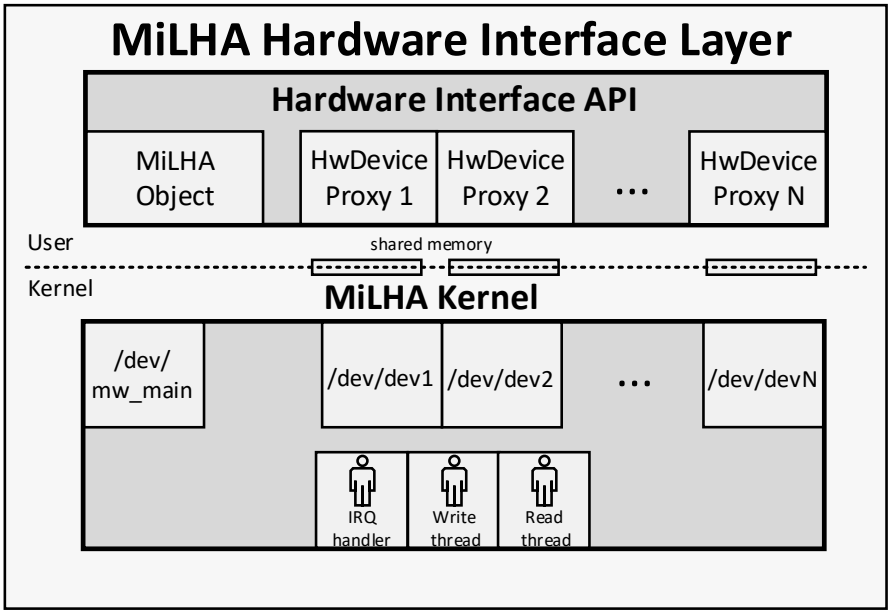


Figura 4.9: Esquema geral da camada de interface com o hardware do MiLHA

4.3.1 Hardware Interface API

O MiLHA inclui na sua API métodos para a utilização independente da camada de interface com o hardware. Deste modo, o programador pode, se assim o desejar, utilizar apenas os serviços de interface com os aceleradores. Isto pode-se mostrar útil por exemplo para testar os aceleradores desenvolvidos antes da conclusão da aplicação software.

À semelhança do modelo de programação, o objeto principal MiLHA concentra toda a informação do resto do sistema. Na Figura 4.10 são apresentadas as classes relativas a esta camada do MiLHA, na classe `CMiLHA` também implementa os métodos do modelo de programação, no entanto nesta figura apenas são apresentados os métodos relativos a esta camada. Existem dois tipos de objetos: o objeto principal MiLHA, que irá permitir criar o segundo tipo de objetos, os `Hardware Device Proxy`, estes são a representação dos aceleradores a nível de aplicação.

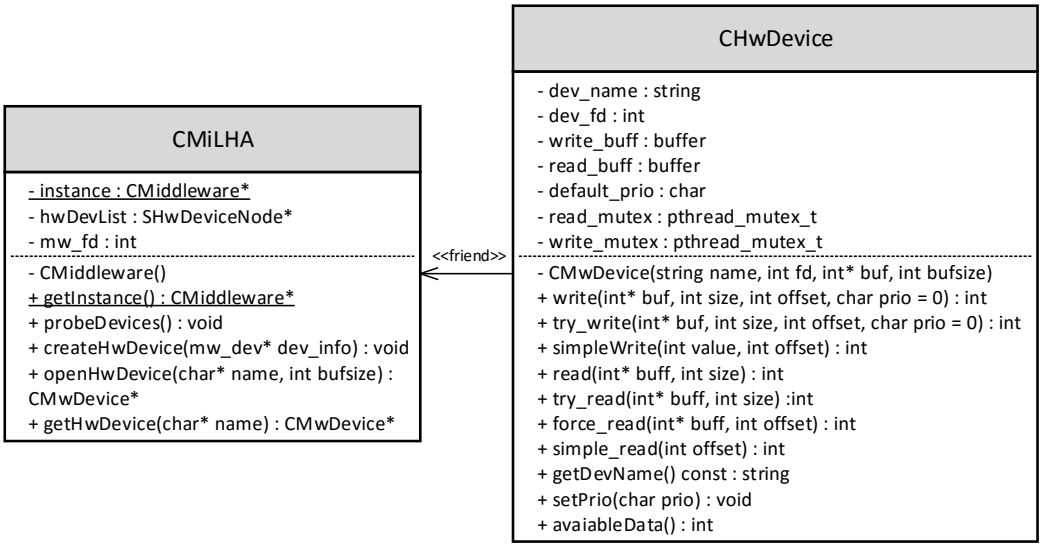


Figura 4.10: Diagrama de classes Hardware Interface API

Hardware Device Proxy

O `Hardware Device Proxy` é um objeto que representa o acelerador hardware a nível da aplicação. É representado pela classe `CHwDevice`, que disponibiliza um conjunto de funções de interface entre o programador e os aceleradores, deverá existir uma instância deste objeto por cada acelerador hardware presente. O `Hardware`

`Device Proxy` é responsável pela:

- **Gestão da comunicação com as aceleradores:** interface com a representação dos aceleradores a nível do *kernel*;
- **Gestão da transferência de dados:** interface com os *buffers* em kernel dos aceleradores e a gestão das variáveis de sincronização;
- **Controlo da concorrência:** proteger contra múltiplos acessos ao mesmo Proxy Device.

A criação deste objeto é feita recorrendo ao `MiLHA Main Object`. Neste, deve ser executado o método `openHwDevice`. Este método recebe como parâmetros caminho para o*device node* do que representa o acelerador e o tamanho pretendido do *buffer* de memória. O `MiLHA Main Object` responde lançando o processo de criação do `Proxy Device`, apresentado na Figura 4.11, que finalizará com o retorno de uma referência para o objeto criado.

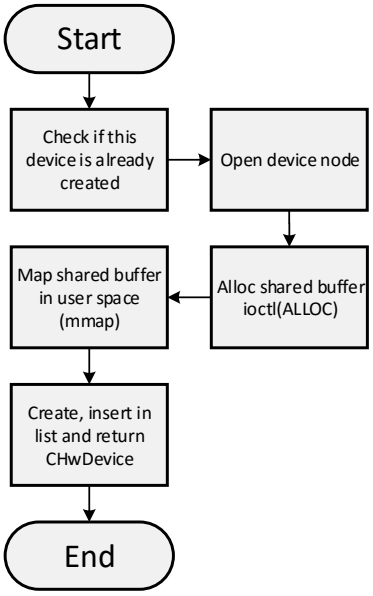


Figura 4.11: Sequencia de criação de um Proxy Device

O primeiro estágio do fluxograma da Figura 4.11 verifica se o Proxy Device com o *device node* selecionado já se encontra aberto (o MiLHA Main Object mantém uma lista com os Proxy Devices criados), uma vez que a sequência de abertura de um `Hardware Proxy Device` só deve ser executada uma única vez. Segue-se abertura do respetivo ficheiro *device node*, é executado o comando para a alocação do *buffer* (`ioctl(MW_IOC_ALLOC)`) com o tamanho especificado pelo programador através do argumento da função, o *buffer* é mapeado no espaço de endereçamento da aplicação

através da *system call* `mmap`. Encontrado o endereço do *buffer* (endereço base + *offset*) é então criado o objeto `CHwDevice`, que posteriormente é guardado na lista e retornado.

Sobre objeto `Proxy Device` criado o programador pode executar um conjunto de operações que representam os serviços disponibilizados pela camada de acesso ao hardware do MiLHA ao nível do *kernel* do Linux. Estes serviços baseiam-se num conjunto de acessos de leitura e escrita no acelerador e são disponibilizados pelos seguintes métodos da classe `CHwDevice`:

- **`write:`** Transfere para o acelerador um conjunto, de tamanho especificado, de dados fornecidos pela aplicação. Esta transferência é do tipo *blocking* pelo que, não sendo possível a sua execução no momento da chamada, a operação será adiada até que a transferência possa retomar. Isto pode acontecer se, por exemplo, não existir espaço suficiente do *buffer* devido à existência de dados de uma outra escrita pendente;

- **`try_write:`** O mesmo serviço que `write` mas do tipo *non blocking*. Caso a transferência não possa ser executada a função retornará erro;

- **`atomic_write:`** Executa uma escrita única de um valor inteiro (32 bits) diretamente no acelerador. Utiliza a *system call* `write` disponibilizada pela driver do *device node* associado ao acelerador. Recebe como argumento um *offset* para escrita em endereços posteriores ao base de escrita. A melhor opção para o controlo do acelerador;

- **`read:`** Transfere para a aplicação um conjunto de dados, de tamanho especificado, presentes no *buffer* do acelerador em kernel. É do tipo *blocking*, pelo que, se não existir no *buffer* a quantidade de dados requisitados, a operação é adiada até que o acelerador os disponibilize;

- **`try_read:`** O mesmo serviço que `read` mas do tipo *non blocking*. Caso não exista a quantidade de dados requisitada a função retornará erro;

- **`atomic_read:`** Executa uma leitura única de um valor inteiro (32 bits) diretamente no acelerador. Recebe como argumento um *offset* para leitura em endereços posteriores ao base de leitura. A melhor opção para leitura do estado do acelerador;

- **`force_read:`** Força uma leitura no endereço do acelerador, mesmo que este não tenha indicado a presença de dados.

Processo de escrita

O processo de escrita inicia-se com a execução sobre o objeto `CHwDevice` de uma das funções membro de escrita no acelerador: `write` ou `try_write`. Estas funções recebem como argumentos:

- Um *buffer* de dados: onde estão os dados a transferir para o acelerador;
- A quantidade de dados a transferir;
- O *offset* de escrita em relação ao endereço base de escrita do acelerador;
- A prioridade do pedido de escrita.

Os dois últimos são opcionais e, caso não sejam especificados, a transferência é efetuada para o endereço base e com a prioridade por defeito do acelerador.

As funções executam essencialmente quatro ações: transferem os dados do *buffer* da aplicação para o *buffer* partilhado; preenchem a estrutura de dados que representa um pedido de escrita com os parâmetros do pedido; notificam o `MiLHA Kernel`; e protegem contra acessos múltiplos no mesmo objeto, permitindo assim a reentrância na função. No caso da função `write` é também executa a *system call* de suspensão da *caller thread*, se o pedido não puder ser efetuado. No Anexo B são apresentados mais detalhes sobre a implementação destas funções.

A Figura 4.12 apresenta o diagrama de sequência de execução de um pedido de escrita. O processo inicia-se na aplicação, que, utilizando a API, faz um pedido de escrita. Ainda na API os dados são transferidos para o *buffer* partilhado, é preenchida a estrutura do pedido e enviada para o `MiLHA Kernel`. Neste momento a função retorna, e aplicação pode continuar a sua execução. O pedido é depois executado por uma *kernel thread* no `MiLHA Kernel`. Esta é a sequência de execução com sucesso de um pedido de escrita. Se no momento do pedido não for possível executá-lo a *thread* da aplicação será suspensa, esta é colocada numa lista de espera em *kernel* e é ativa quando for possível proceder a execução do pedido.

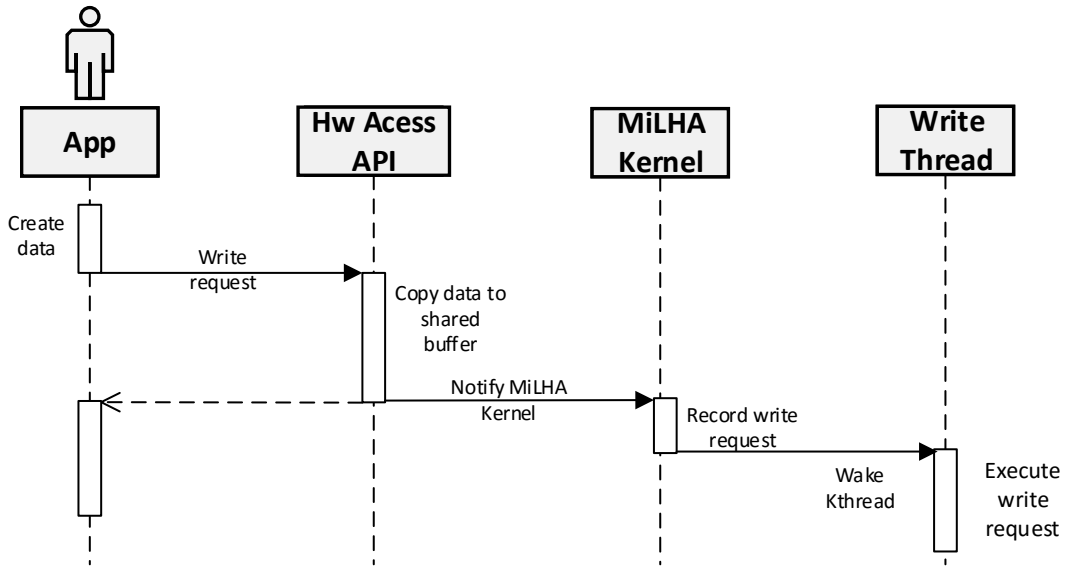


Figura 4.12: Sequência da execução do processo de escrita

Processo de Leitura

O processo de leitura inicia-se com a execução de uma das funções de leitura do acelerador: `read` ou `try_read` sobre o objeto `CHwDevice`. Cada uma destas funções irá receber como argumentos:

- Um *buffer* de dados: o *buffer* da aplicação para onde são transferidos os dados;
- A quantidade de dados a transferir.

As funções verificam a existência de dados no *buffer* partilhado através das variáveis de sincronização do *buffer*. Aí, se não estiver disponível a quantidade de dados especificada é retornado erro (*non blocking*) ou a *thread* da aplicação é suspensa. Se existirem dados, estes são transferidos para o *buffer* da aplicação e as variáveis de sincronização são atualizadas. Estas funções também estão protegidas contra múltiplas execuções, permitindo assim a sua reentrância. Mais detalhes sobre a sua implementação são apresentados no Anexo B.

Para uma melhor perceção do processo de leitura é apresentado na Figura 4.13 o diagrama de sequência dos processos que o compõem. O processo de leitura inicia-se com a sinalização, por parte do acelerador, da existência de dados válidos, através de uma interrupção. Em `MilHA Kenel`, na rotina de atendimento à interrupção é

registado um pedido de leitura. Este é posteriormente processado por uma *kernel thread*, que transfere os dados do acelerador para o *buffer* partilhado. No lado da aplicação é feito um pedido de leitura, através do `Hardware Device Proxy`, caso não exista a quantidade requisitada de dados, a *thread* aplicação é colocada numa lista de espera e suspensa. A *thread* suspensa é posteriormente ativada quando a quantidade especificada de dados estiver disponível.

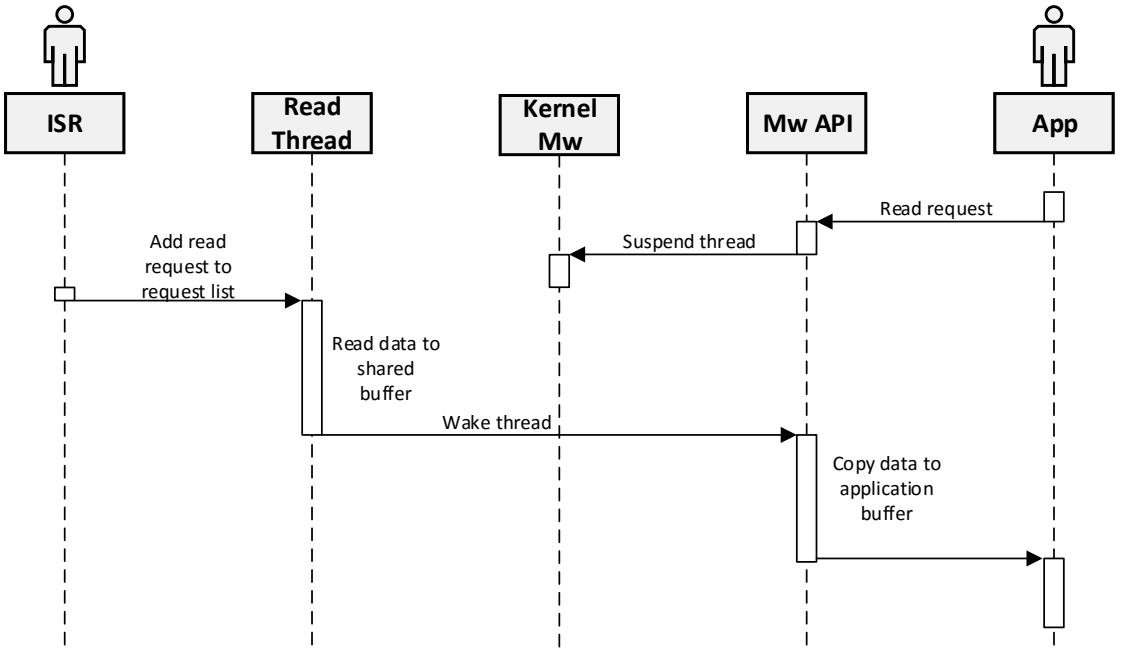


Figura 4.13: Sequência de execução do processo de leitura

4.3.2 MiLHA Kernel

O `MiLHA Kernel` consiste num Linux Kernel Object (.ko) onde estão implementados os serviços privilegiados da camada de interface com o hardware do `MiLHA`. A invocação destes serviços é controlada pela `Hardware Interface API` no espaço do utilizador, que por sua vez entra no espaço privilegiado através de *system calls*.

Os vários componentes do `MiLHA Kernel` podem ser observados na Figura 4.9 , estes são:

- **Um Linux *device node* principal:** Este é criado quando o Kernel Object é inserido no Linux Kernel, é responsável pela criação dos `Hardware Accelerator Nodes`. Pode ser encontrado em "`/dev/mw_main`";

- **`Hardware Accelerador Nodes:`** Estes são os representantes dos aceleradores a nível do Linux Kernel. Implementam o *device driver* que fornece serviços de comunicação com os aceleradores;

- ***Buffers* de dados:** Cada `Hardware Accelerator Node` estará associado a dois *buffers* de dados (leitura e escrita). Estes são criados a nível do Linux Kernel e são partilhados com a aplicação;

- **Duas *threads*:** Existem duas *threads* no espaço *kernel*. Estas são as responsáveis pela execução dos pedidos, uma para pedidos de leitura outra para os de escrita;

- **Rotina de atendimento a interrupções:** Os aceleradores indicam ao `MiLHA Kernel` a presença de dados válidos e prontos para leitura através de uma interrupção. Esta rotina atende essa interrupção e cria um pedido de leitura ao respetivo acelerador. Este é posteriormente processado pela *thread* de leitura que lê os dados do acelerador.

De seguida, cada uma destas entidades será apresentada mais detalhadamente.

Hardware Acelerator Node

Num ambiente MiLHA cada acelerador será publicado como um Linux *device node*, na directoria `/dev/`. Este, juntamente com o respetivo *device driver*, será o meio de acesso aos aceleradores hardware. Para além da interface com o acelerador, esta unidade inclui o conjunto de parâmetros necessários para definir o acelerador:

- **Nome do acelerador:** O nome com que será publicado, ou seja, cada acelerador poderá ser encontrado com o caminho `/dev/(nome_do_acelerador)`;

- **Endereço de escrita:** Endereço físico base de destino dos pedidos de escrita no acelerador;

- **Endereço de leitura:** Endereço físico base de leitura de dados do acelerador;

- **Prioridade:** Prioridade no escalonamento das operações de leitura/escrita no acelerador. Pedidos de aceleradores com maior prioridade irão ser executados primeiro, podem inclusive interromper execuções de pedidos com menos prioridade.

- ***Threshold* para leituras:** Este parâmetro indica o número de dados presentes no acelerador quando este ativa a interrupção. Irá ser o número de leituras a executar quando ocorre a interrupção;

- **Número de interrupção:** Interrupção associada ao acelerador, quando ocorre a interrupção significa que o acelerador tem dados válidos prontos para serem lidos.

Estes parâmetros irão ser guardados na estrutura `struct mw_device` que irá conter também as restantes estruturas de suporte, nomeadamente `wait queues`, `mutexes` e `buffers`.

O *device driver* do `Hardware Accelerator Node` será a interface com a camada da aplicação dos sistema através da `Hardware Intarface API`. O *device driver* implementa as seguintes funcionalidades:

- **Leitura atómica:** A *system call* `read` executa uma leitura única (32 bits) no endereço base de leitura do acelerador com um *offset* dado pelo argumento `count` da *system call*. Com esta funcionalidade, em que uma leitura é executada diretamente, é possível ler de forma simples e rápida um registo de controlo ou de estado do acelerador;

- **Escrita atómica:** Em resposta à *system call* `write` é executada uma escrita única (32 bits) no endereço base de escrita do acelerador com o *offset* dado pelo argumento `count` da *system call*. Com esta leitura simples, é possível, por exemplo, escrever de forma rápida num registo de controlo de um acelerador;

- **Suspender *thread* para leitura:** Em resposta à *system call* `ioctl` `MW_IOC_WAIT_READ` a *caller thread* é adormecida e colocada numa *wait queue* até que exista a quantidade de dados requisitada;

- **suspender *thread* para escrita:** Em resposta à *system call* `ioctl` `MW_IOC_WAIT_WRITE` a *caller thread* é adormecida e colada numa *wait queues* até que exista espaço suficiente para executar a escrita requisitada;

- **Notificação de escrita:** O device é notificado, através da *system call* `ioctl` `MW_IOC_WRITE_NOTIFY`, que foram escritos dados no *buffer de escrita*. É criado um novo pedido pedido e adicionado à lista de pedidos de escrita, que posteriormente é executado pela *write thread*;

- **Forçar Leitura:** Em resposta à *system call* `ioctl` `MW_IOC_FORCE_READ` é possível forçar a execução de um pedido de leitura, mesmo que o acelerador não tenha sinalizado a existência de dados válidos. Esta funcionalidade cria um novo pedido de leitura e adiciona-o à lista de pedidos de leitura, para posteriormente ser executado pela `thread` de leitura;

- **Alocação de buffers:** Em resposta à *system call* `iotcl` `MW_IOC_ALLOC`, são alocados, a nível do kernel, os *buffers* de escrita e de leitura do *device*. É recebido como argumento o tamanho, em número de inteiros, a alocar para cada *buffer* e estes são alocados em endereços de memória consecutivos;

- **Mapeamento de buffers:** Esta funcionalidade permite, através da *system call* `mmap`, mapear os endereços dos *buffers* do *device* em *user level*, de modo a que os *buffers* possam ser acedidos diretamente a nível da aplicação. O mapeamento é feito a nível de páginas de memória, o que faz com que o endereço retornado pelo `mmap` seja o endereço base da página de memória. O endereço do *buffer* irá ter um *offset* em relação ao endereço base da página, esse *offset* pode ser retornado pela *system call* `ioctl` `MW_IOC_OFFSET`;

Parsing do Device-Tree

A maneira aconselhada de criar os nós *devices* é através da interpretação da *device-tree*. Quando utilizada a ferramenta Xilinx XPS esta gera a *device-tree source* (DTS), que contém informação sobre o hardware presente na plataforma. A DTS é compilada no *device-tree blob* (DTB) e este pode ser interpretada pelo Linux kernel durante o arranque, onde se auto configura em função da plataforma hardware e dos aceleradores existentes.

O `MiLHA kernel` utiliza a infraestrutura `of_platform`, em que a *device-tree* é interpretada e é registada uma `struct of_device` para cada nó encontrado. Por outro lado, a *device driver* regista uma `struct of_platform_driver` e, na correspondência do campo `compatible` entre o nó e a *driver*, é chamada uma função registada pela *driver*. No MiLHA kernel essa função lê a informação sobre o acelerador existente no FPGA-Fabric é criado um novo `Hardware Accelerator Node`. Para simplificação foi criada a função `read_mwdev_info`, que recebe como parâmetro uma `struct device_node`, onde apenas é interpretada a informação do nó *device-tree*. Esta está preparada para interpretar nós do periférico `MiLHA Interface IP` e teria de ser alterada para dar suporte a um outro periférico, caso

os campos fossem diferentes. Na Figura 4.14 observa-se um exemplo de um nó `MiLHA Interface` numa *device-tree source*, de onde podem ser retirados os seguintes parâmetros do acelerador:

- Nome do acelerador do campo `"xlnx,mw-dev-name"`;
- Endereço e espaço de memória do campo `"reg"`;
- Número e tipo de interrupção do campo `"interrupts"`;
- Prioridade do acelerador do campo `"xlnx,mw-dev-priority"`;
- Número de leituras ao acelerador do campo `"xlnx,read-threshold"`;

Estes são os parâmetros utilizados para a criação do novo `Hardware Accelerator Node`.

```
milha_interface_pq: milha-interface@CE400000 {
                    compatible = "xlnx,milha-interface-1.00.a";
                    interrupt-parent = <&xps_intc_0>;
                    interrupts = < 2 0 >;
                    reg = < 0xCE400000 0x10000 >;
                    xlnx,include-dphase-timer = <0x0>;
                    xlnx,mw-dev-name = "pq";
                    xlnx,mw-dev-priority = <0x64>;
                    xlnx,read-threshold = <0x28>;
} ;
```

Figura 4.14: Exemplo de um nó MiLHA Interface na *device-tree*

Threads do espaço Kernel

O acesso aos periféricos, uma vez que são do tipo *memory-mapped I/O*, é feito através de acessos à memória, nos endereços onde estarão mapeados os aceleradores. No MiLHA, a execução final dos pedidos, ou seja, o acesso ao hardware é da responsabilidade de duas *kernel threads*: a `Read Thread` e a `Write Thread`.

Existem duas listas de pedidos, uma para pedidos de escrita outra para pedidos de leitura, estas encontram-se ordenadas por prioridade, ou seja, o pedido com maior prioridade encontrar-se-á sempre no primeiro nó da lista (a cabeça). Um pedido de execução corresponde ao registo na respetiva lista (escrita ou leitura) de uma estrutura que contém informação sobre: o *device* associado, o endereço atual de escrita/leitura no *buffer*, o endereço atual de escrita/leitura no acelerador, a

quantidade restante de dados a copiar e a prioridade do pedido. Pedidos de escrita podem ser registados através da *system call* `ioctl(MW_IOC_WRITE_NOTIFY)`, pedidos de leitura podem ser registados pelo *handler* de interrupção do *device* e pela *system call* `ioctl(MW_IOC_FORCE_READ)`. O registo de pedidos irá ativar a respetiva *thread* caso esta esteja suspensa.

As duas *threads* seguem o mesmo padrão de execução. Cada *thread* irá verificar a respetiva lista e executar o pedido presente no topo, caso não existam pedidos, a *thread* será suspensa através uma `wait queue`. A execução dos pedidos está apresentada sob a forma de fluxogramas nas Figuras 4.15 (escrita) e 4.16 (leitura). A execução traduz-se numa cópia de dados do *buffer* para o endereço de escrita do acelerador (escrita) ou do endereço de leitura do acelerador para o *buffer* (leitura), à razão de 4 bytes por operação. A cada iteração as *threads* verificam os limites dos *buffers* e atualizam as suas variáveis de sincronização. No fim de cada iteração é selecionado novamente o pedido presente na cabeça da lista, assim, se durante a execução de um pedido ocorrer um novo com maior prioridade, esse será executado.

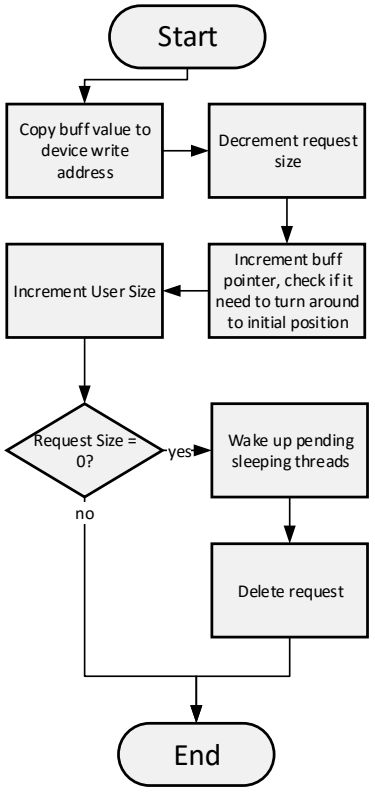


Figura 4.15: Execução de pedido de escrita

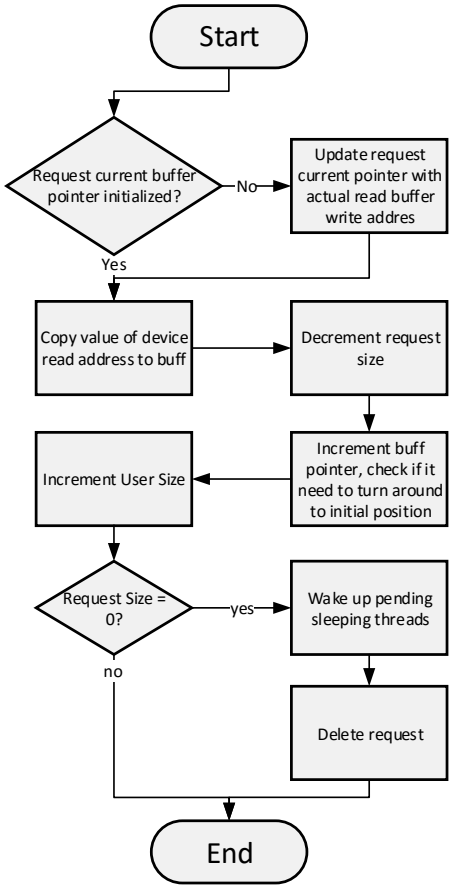


Figura 4.16: Execução de pedido de leitura

Buffers

A troca de dados entre a aplicação e os *devices* é feito através de dois *buffers*: um para o processo de escrita e outro para o processo de leitura. Quando a aplicação efetua um pedido de escrita os dados são copiados para o *buffer* de escrita, onde se mantêm até o pedido ser executado. Quando o acelerador indica a presença de dados válidos estes são lidos e guardados no *buffer* de leitura, onde se mantêm até que a aplicação software execute o pedido leitura.

Para permitir o acesso partilhado aos *buffers* entre o *user* e o *kernel spaces* recorre-se a uma técnica de mapeamento de memória em que os endereços virtuais a nível da aplicação são mapeados nos endereços lógicos a nível do kernel. Deste modo é possível o acesso às mesmas posições de memória pelos dois espaços de endereçamento. Os *buffers* são de acesso circular, o que exige que nos momentos de escrita ou leitura sejam conhecidos os endereços atuais de escrita e leitura. Para esse efeito, durante a alocação dos *buffers*, são alocados, para além do espaço de memória destinado ao armazenamento de dados, dois inteiros para cada *buffer*, o `User Offset` e o `User Size`. Assim sendo, cada *buffer* terá o formato mostrado na Figura 4.17.

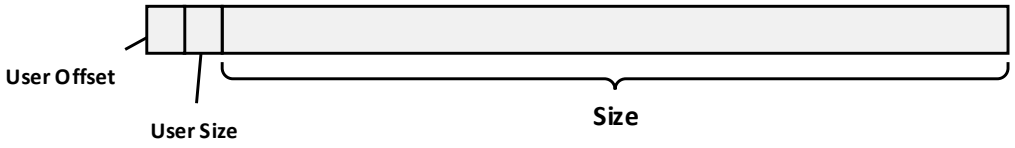


Figura 4.17: Esquema *buffer*

Os dois inteiros iniciais não representam especificamente os índices de leitura e de escrita, têm uma interpretação diferente dependendo se se trata do *buffer* de escrita ou do de leitura. No *buffer* de escrita, o `User Offset` representa o índice de escrita a nível da aplicação enquanto que o campo `User Size` representa o espaço disponível (em número de inteiros) no *buffer*. Já no *buffer* de leitura o `User Offset` representa o índice de leitura a nível da aplicação enquanto o campo `User Size` representa a quantidade de dados válidos disponíveis (um número de inteiros) no momento.

4.4 Integração do Acelerador no MiLHA

O *offloading* de processamento para hardware pode ser utilizado quando o poder de processamento em software não é suficiente para cumprir os requisitos de um determinado processo. Esse processo, ou parte dele, é então implementado em hardware dedicado, sob a forma de um acelerador. O acelerador executa então a função do processo software tirando partido do processamento paralelo do hardware, retirando carga ao processador.

No processo de aceleração de uma aplicação, apresentada na Secção 2.3.3, durante a fase de paralelização em hardware, depois de codificar a lógica que o acelerador implementa em HDL é necessário integrá-lo no SoC.

O MiLHA propõe um modelo de acelerador hardware, e fornece um ambiente de suporte para esse modelo. Foram desenvolvidos mecanismos em hardware de interface com o MiLHA que foram implementados consoante as necessidades da aplicação de demonstração desta dissertação, apresentada no Capítulo 5. Estes focam a transferência de dados e a ligação entre vários aceleradores.

Os mecanismos de interface desenvolvidos foram: um IP de interface com a camada de software do MiLHA que liga ao barramento do sistema, o `MiLHA Interface IP`; e um modelo de acelerador que inclui um *wrapper* para a lógica do utilizador (*User Logic*) que implementa a interface com o `MiLHA Interface IP`. Estes estão apresentados na Figura 4.18 e seguidamente serão abordados com mais detalhe.

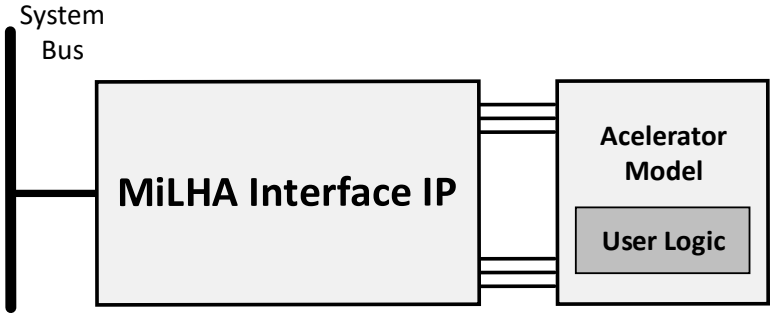


Figura 4.18: Mecanismos de integração do acelerador

4.4.1 Modelo genérico de Acelerador

O MiLHA utiliza um modelo genérico de acelerador. A utilização deste modelo permite unificar as interfaces e desenvolver mecanismos genéricos que podem ser

facilmente reutilizados. Neste modelo, apresentado na Figura 4.19, existem duas entidades: a `User Logic` e a `User Logic Wrapper`.

A `User Logic` é a lógica desenvolvida pelo programador que implementa o processamento específico do acelerador, esta deve também seguir o padrão apresentado em 4.4.2.

O `User Logic Wrapper` é fornecido pelo MiLHA e implementa as interfaces de troca de dados com o IP de interface com o MiLHA e, também, a troca de dados entre aceleradores.

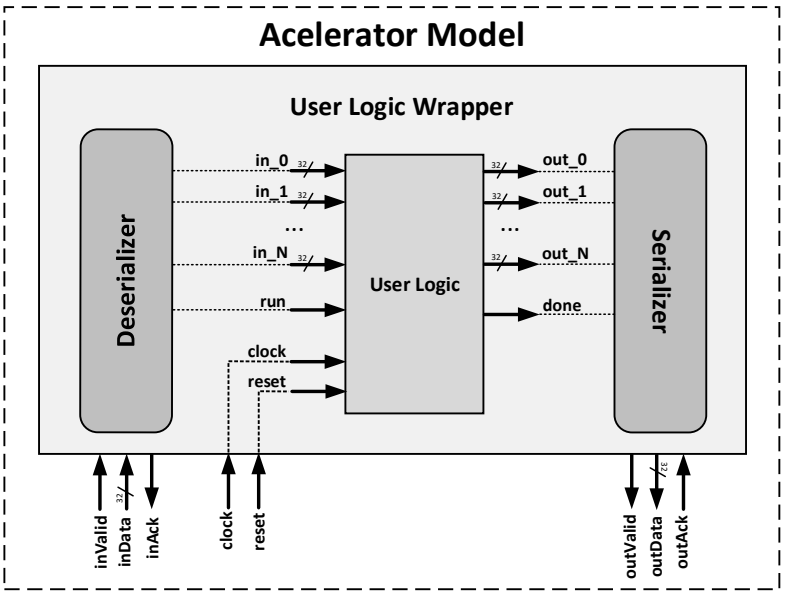


Figura 4.19: Esquema do User Logic Wrapper

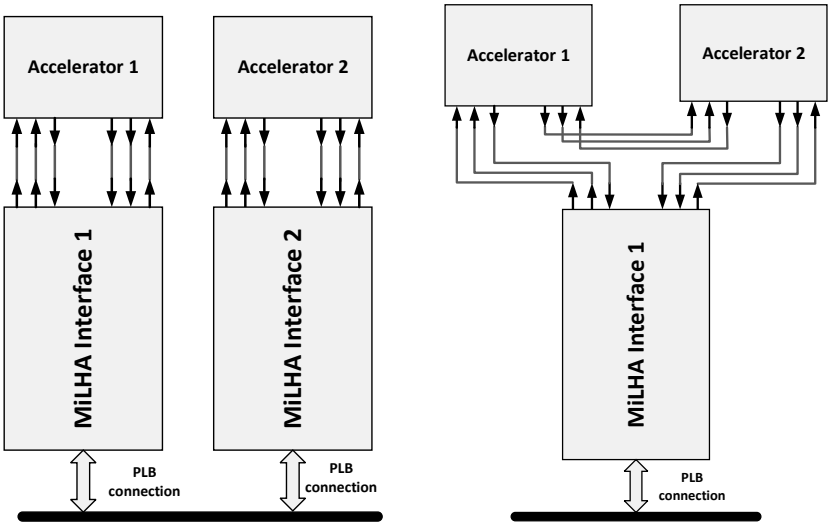
User Logic Wrapper

O User Logic Wrapper é um template de código Verilog criado com o propósito de envolver o acelerador hardware.

Este *wrapper* implementa funcionalidades de interface entre o acelerador e IP MiLHA Interface. Implementa lógica para receber dados em série, com o mesmo protocolo de comunicação usado pelo MiLHA Interface, e dispô-los ao acelerador de forma paralela (`Deserializer`). Da mesma maneira envia em série os dados criados pelo acelerador (`Serializer`).

Para além de interface entre o acelerador e o MiLHA Interface IP este *wrapper* permite fazer a ligação entre vários aceleradores, transformando-os num *pipeline*

de aceleradores, a Figura 4.20 mostra as duas possíveis ligações.



(a) Ligação simples

(b) Ligação em array de aceleradores

Figura 4.20: Possíveis ligações entre *wrappers*

Para auxilio ao programador é fornecido no Anexo C um template com um exemplo de ligação. Para incluir o acelerador neste wrapper deve-se:

1. Codificar o acelerador segundo o modelo de acelerador especificado em 4.4.2;
2. Indicar o número de dados de entrada e de saída do acelerador nos parâmetros `N_INPUT_PARAMETERS` e `N_OUTPUT_PARAMETERS`;
3. Fazer a ligação dos sinais `process_run` e `process_done` aos respetivos `run` e `done` do acelerador;
4. Fazer as ligações dos sinais `inputs[i]`, com `i` de `0` até `N_INPUT_PARAMETERS`, às respetivas entradas do acelerador.
5. Fazer as ligações dos sinais `outputs[i]`, com `i` de `0` até `N_OUTPUT_PARAMETERS`, às respetivas saídas do acelerador.

Desta maneira, o wrapper tratará da interface de entrada e saída de dados e dos sinais de sincronização `run` e `done` através de duas máquinas de estados: uma para receção de dados e outra para envio de dados.

A Figura 4.21 apresenta a máquina de estados de receção de dados: num estado inicial, `index` é inicializado a zero e espera-se pelo sinal de presença de dados válidos (`in_valid`). Na ocorrência deste sinal inicia-se a receção de dados. Os

dados são guardados no endereço `index` do banco de registos `inputs`, a cada leitura efetuada com sucesso é incrementado o `index`. Quando o sinal `enough_pop` for ativado, indicando que já se receberam o número de parâmetros especificados pelo programador finaliza-se a receção de dados e ativa-se o acelerador através do sinal `process_run`. Quando o acelerador iniciar o seu processamento, indicando pela transição negativa do sinal `process_done`, fecha-se o ciclo de receção de dados. Isto permite que se esteja a receber novos dados enquanto o acelerador processa os anteriores.

A Figura 4.22 apresenta a máquina de estados de envio de dados. Num estado inicial, um outro índice é inicializado a zero e espera-se pela transição ascendente do sinal `process_done`, que indicará que o acelerador terminou o processamento e as suas saídas têm dados válidos. Seguidamente, as saídas do acelerador são copiadas para um banco de registos, para que, no caso de o acelerador iniciar imediatamente a seguir um novo ciclo de processamento, não seja necessário manter as suas saídas válidas com os dados do ciclo anterior. Inicia-se então o protocolo de envio de dados, enviando os dados guardados no banco de registo, até o sinal `enough_push` ser ativado, que indica que foram enviados o número de dados indicados pelo programador.

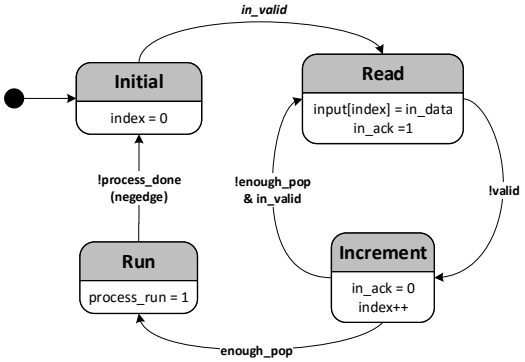


Figura 4.21: Máquina de estados de receção de dados

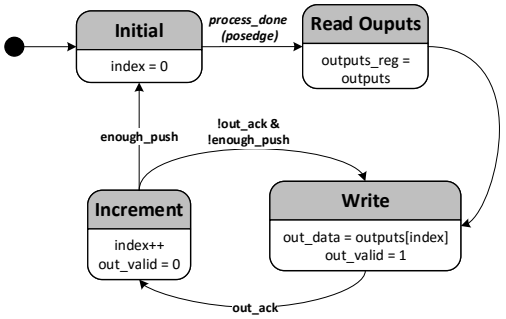


Figura 4.22: Máquina de estados de envio de dados

4.4.2 Lógica Computacional do Acelerador

O MiLHA propõe o modelo simplista de acelerador hardware apresentado na Figura 4.23.

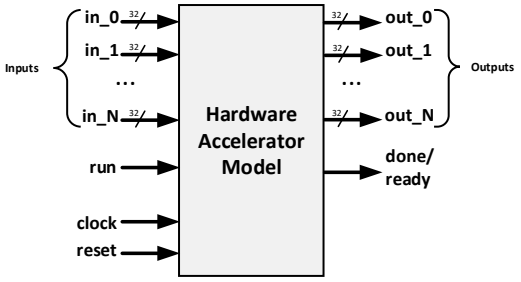


Figura 4.23: Top Level do modelo de acelerador hardware

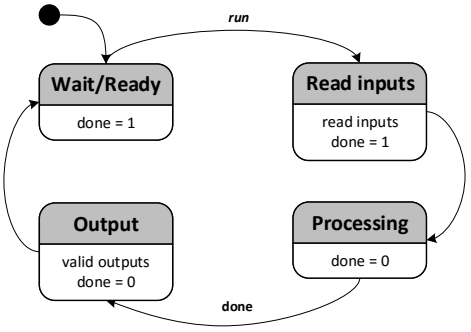


Figura 4.24: Máquina de estados do modelo de acelerador

O modelo composto pelas entradas básicas de um sistema hardware, `clock` e `reset`, pelas entradas de dados, `in_[0-N]`, de 32 bits cada, pelas saídas de dados `out_[0-N]`, o sinal de `run` e sinal de `done`. O sinal de entrada `run` indica ao acelerador que os dados nas entradas estão válidos e que pode iniciar a execução. O sinal de saída `done` é ativado quando o acelerador finaliza a execução e mantém-se ativo até o acelerador iniciar uma nova execução. O processo do acelerador, apresentado na Figura 4.24, inicia-se com o sinal `done` do acelerador ativo, indicando que o acelerador está pronto para uma nova interação de processamento. Quando o sinal `run` for ativado o acelerador deve então iniciar a execução, mantendo `done` desativado. Ao terminar a execução, o acelerador coloca os resultados nas nas saídas e ativa novamente o sinal de `done`, que sinaliza o fim de uma iteração de processamento. Na Figura 4.25 pode ser observado um exemplo do comportamento dos sinais durante o processamento execução de um acelerador.

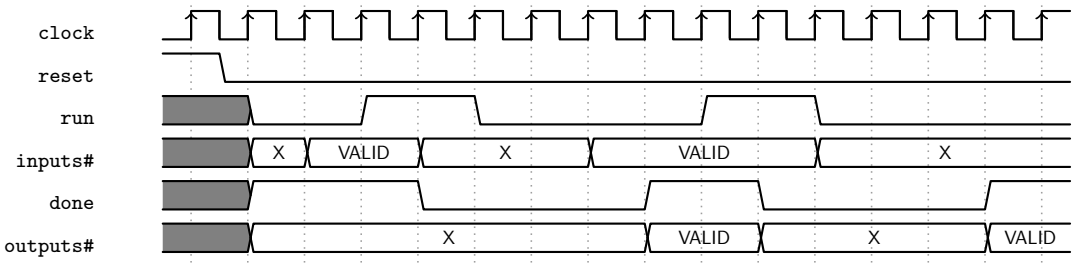


Figura 4.25: Sinais lógicos do modelo de acelerador

4.4.3 Comunicação com Acelerador

O hardware MiLHA Interface é um IP de interface entre o acelerador hardware criado pelo programador e a componente software do MiLHA. É compatível e automaticamente reconhecido pela componente de interface com o hardware do MiLHA, assim sendo, está pronto para ser utilizado na aplicação sem necessidade de qualquer alteração nas configurações da camada de interface com o hardware.

A Figura 4.26 mostra o esquema do MiLHA Interface IP, este contém:

- Interface com o barramento;
- FIFO de armazenamento;
- Sinais de comunicação com o acelerador;
- Sinal de interrupção.

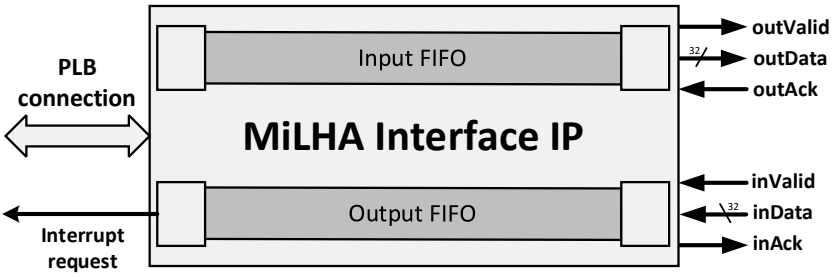


Figura 4.26: Esquema MiLHA Interface IP

Para poder ser acedido pelo CPU o IP encontra-se ligado ao barramento do sistema. O IP inclui interface com o PLB (Processor Local Bus) que é o barramento utilizado pela arquitetura do processador usado nesta dissertação, o PowerPC. Este é o único componente do MiLHA específico à plataforma utilizada. Numa eventual utilização do MiLHA noutra plataforma, com um diferente tipo de barramento haverá necessidade de adaptar este IP de forma a suportar o novo barramento.

O IP contém também duas unidades de armazenamento do tipo FIFO (First In First Out): Input FIFO para comunicação no sentido CPU acelerador, e a Output FIFO no sentido acelerador CPU, ambas com capacidades de 1024 amostras de 32bits. O objetivo deste armazenamento temporário é a dessincronização da comunicação entre o CPU e o acelerador hardware.

Para comunicação com o acelerador o IP utiliza duas instâncias de um mecanismo

de comunicação simples de três sinais: `data, valid` e `acknowledge`, uma para enviar e outra para receber dados do acelerador. Este será explicado posteriormente.

Por fim o IP contém lógica para ativação de um sinal de interrupção, este destina-se a desencadear uma interrupção que indicará ao processador a existência de dados na unidade de armazenamento.

Parâmetros de interface com o MiLHA

O MiLHA Interface IP inclui três parâmetros que podem ser definidos pelo programador para definir o comportamento do IP e para interface com o software:

- **C_MW_DEV_NAME :** Nome do acelerador hardware, nome com que irá ser publicado um *device node* em Linux. Por exemplo: este IP é incluído com o nome `"magic"`, na execução do MiLHA irá ser criado o *device node*, `"/dev/magic"`;
- **C_MW_DEV_PRIORITY :** Prioridade do acelerador hardware. Pedidos de leitura provenientes de aceleradores com maior prioridade irão ser executados primeiro, podendo até interromper a execução de pedidos de prioridade mais baixa;
- **C_READ_THRESHOLD :** Número de dados presentes no armazenador de saída de dados necessário para ser desencadeada a interrupção de leitura. Existe para diminuir a cadência da interrupção de leitura (aumentando o numero de dados lidos a cada interação) e é utilizado também pelo software para saber o número de leituras a executar.

O `MiLHA interface IP` pode ser importado para um projeto do Xilinx Platform Studio (XPS) e, como se pode observar na Figura 4.27, é possível configurar os parâmetros de interface com a camada de interface com o hardware do MiLHA. No exemplo da figura o IP foi configurado com o nome "pq", com uma prioridade de 100 e com ativação de leituras a cada 40 dados presentes na unidade de armazenamento FIFO.

Comunicação com o acelerador hardware

Para troca de dados entre o `MiLHA Interface IP` e o acelerador é utilizado um mecanismo simples de comunicação. Este é baseados em três sinais: `Data`, um

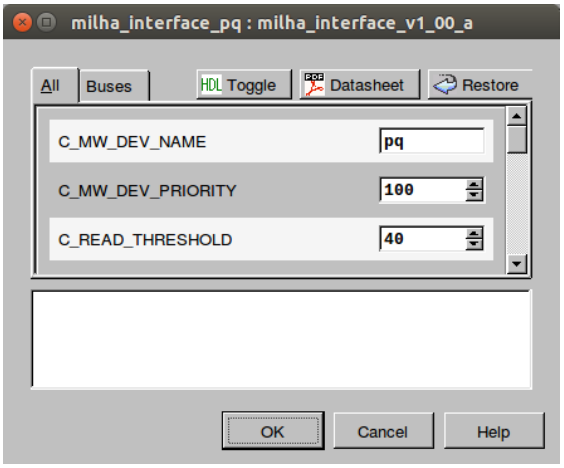


Figura 4.27: Parâmetros do MiLHA Interface IP no XPS

barramento de 32 bits onde está a data; `Valid`, um sinal que indica que o barramento de dados tem dados válidos, pronto a serem lidos, e o sinal `Ack`, que indica ao transmissor que o recetor leu com sucesso os dados do barramento. Esta comunicação é do tipo assíncrona e não utiliza qualquer tipo de sinal de *clock*, deste modo não é necessário que o acelerador e o MiLHA Interface possuam o mesmo sinal de *clock*, nem que os seus sinais *clock* estejam em fase.

A Figura 4.28 representa o comportamento dos sinais intervenientes numa transferência de dados usando este protocolo.

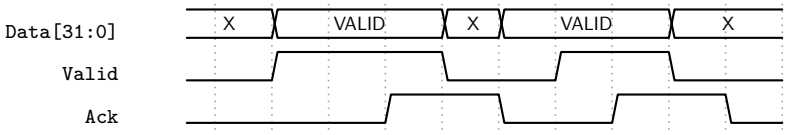


Figura 4.28: Protocolo comunicação MiLHA Interface - acelerador

Gestão das FIFO de armazenamento temporário

O MiLHA Interface contém duas unidades de armazenamento do tipo FIFO, a `Input FIFO` e a `Output FIFO`. Estas foram criadas com a ferramenta IP Core Generator da Xilinx e apresentam o *top level* apresentado na Figura 4.29.

A `Input FIFO` serve de armazenamento temporário para transferências de dados no sentido barramento acelerador. Escritas no endereço base deste IP irão ser traduzidas em escritas na `Input FIFO`, os dados presentes nesta unidade de armazenamento serão despachados para o acelerador pela interface de saída de dados

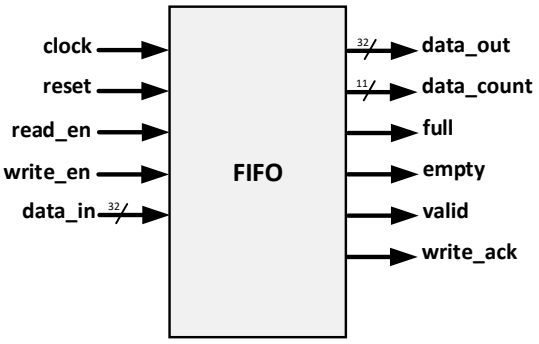


Figura 4.29: *Top Level* das FIFO

(sinais `outValid, outData, outAck`). Complementarmente a `Output FIFO` suportará as transferências de dados no sentido acelerador barramento. Os dados provenientes da interface de entrada de dados serão armazenados na FIFO, posteriormente as leituras no endereço base deste IP são trazidas em leituras nesta FIFO.

As Figuras 4.30 e 4.31 apresentam as máquinas de estado de controlo da interface FIFO-barramento, mais especificamente as escritas na `Input FIFO` e leituras na `Output FIFO`. As ações iniciam-se com as escritas/leituras no endereço base do IP, que se traduzem na ativação dos sinais `write_trigger` e `read_trigger`, respetivamente. Posteriormente a lógica encarrega-se da ativação dos sinais de escrita/leitura nas FIFO.

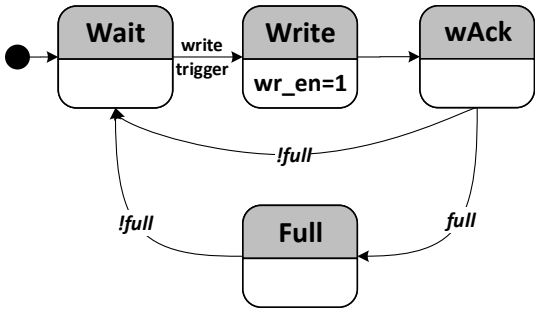


Figura 4.30: Máquina de estados de controlo da escrita na Input FIFO

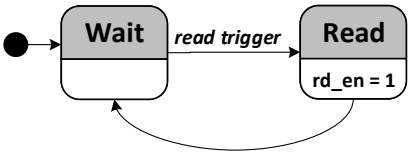


Figura 4.31: Máquina de estados de controlo de leituras na Output FIFO

Do outro lado existe a lógica para gestão da interface FIFO-acelerador. A Figura 4.32 apresenta sequência de transferência de dados no sentido FIFO-acelerador, que se traduz em leituras nas `Input FIFO`: na existência de dados válidos na saída da FIFO (`fifo_valid`) e estando o recetor pronto para receber (`!out_ack`) é indicado ao recetor, pelo sinal `out_valid`, a existência de dados válidos, quando este completar a leitura (`out_ack`) é ativado o sinal de leitura na FIFO (`rd_en`) para

avançar para os próximos dados. A sequência de transferência de dados no sentido acelerador-FIFO, que se traduz em escritas na `Output FIFO`, é apresentada na Figura 4.33. A escrita inicia-se com a sinalização, por parte do acelerador, de dados válidos no barramento (`in_valid`), é ativado o sinal de escrita na FIFO (`rd_en`) durante um ciclo e, depois, ativado o sinal de *aknowlage* (`in_ack`), para indicar ao acelerador que o dado foi guardado com sucesso. Seguidamente espera que o sinal `in_valid` seja desativado e verifica se a FIFO atingiu o limite de capacidade de armazenamento.

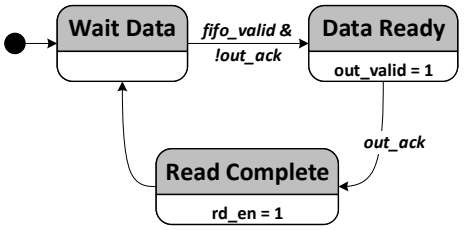


Figura 4.32: Máquina de estados de controlo da leitura da Input FIFO

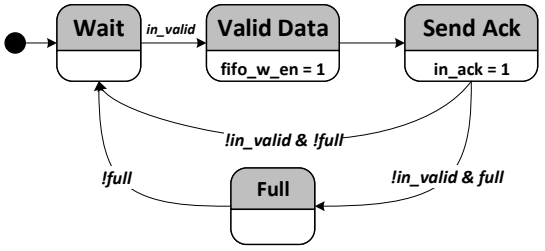


Figura 4.33: Máquina de estados de controlo de escritas na Output FIFO

Capítulo 5

Cenário de aplicação

Para efeitos de teste e de demonstração foi criada uma aplicação onde foi aplicado o processo de aceleração em hardware recorrendo aos serviços MiLHA. A aplicação escolhida foi o sistema de controlo de um Filtro Ativo Paralelo (FAP), esta escolha deve-se à participação na construção de um FAP, no âmbito da tese de douramento do Engenheiro Vitor Silva, do qual resultou o artigo "Linux- and FPGA-based Accelerated Single-Phase Shunt Active Power Filter" aceite para publicação na 42nd Annual Conference of IEEE Industrial Electronics Society.

Cargas não lineares, cada vez mais presentes na nossa rede elétrica, consomem correntes distorcidas, com conteúdo harmónico. O consumo dessas componentes harmónicas é nocivo para a rede elétrica, afetando a qualidade da energia elétrica. Um FAP é ligado à rede elétrica paralelamente à carga, o filtro calcula e produz localmente o conteúdo harmónico que quando subtraído à corrente na carga resulta na componente sinusoidal fundamental. Deste modo a fonte apenas fornecerá à carga a componente constante da corrente, ou seja, corrente sinusoidal, equilibrada e em fase com a tensão.

A Figura 5.1 apresenta um esquema de ligação de um FAP onde se podem observar as três correntes distintas: I_S - corrente da fonte; I_L - corrente de linha e I_F - corrente de filtro. A corrente da fonte irá ser a corrente que a carga consumir menos a que for fornecida pelo filtro ($I_S = I_L - I_F$). Portanto, apesar de a carga consumir corrente com conteúdo harmónico esse será fornecido pelo filtro mantendo do lado da fonte uma corrente sinusoidal, equilibrada e em fase com a tensão.

A aplicação de demonstração desenvolvida calcula, através das correntes e tensões na carga, a corrente que o filtro deverá de fornecer a cada instante.

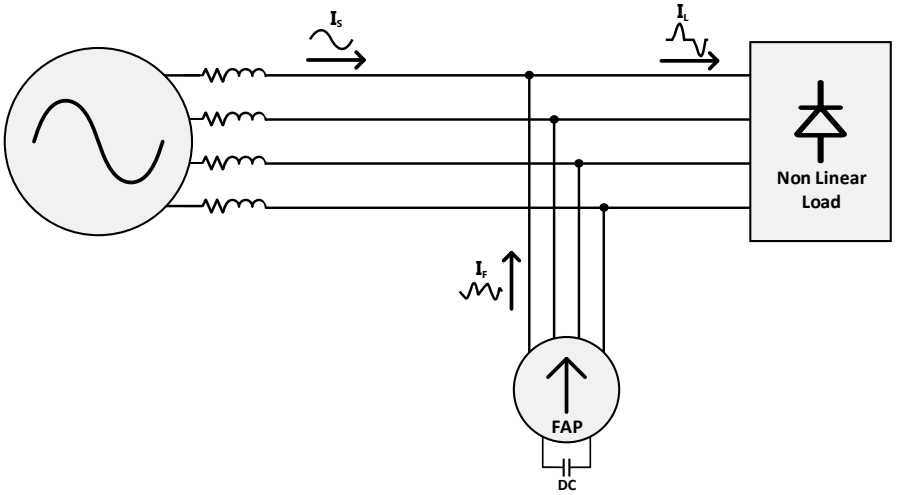


Figura 5.1: Esquema de um Filtro Ativo de Potência

A aplicação foi desenvolvida segundo o *design flow* apresentado em 2.3.3, ou seja, criado uma aplicação inicial *software-only*, que posteriormente foi paralelizada e a computação crítica foi migrada para aceleradores no FPGA.

5.1 Aplicação Software-only

A primeira etapa de *design flow* é a modelação da aplicação em software. Aqui a aplicação foi desenvolvida sem nenhuma paralelização, os processamento é totalmente sequencial. com o propósito de validação funcional.

A Figura 5.2 mostra o diagrama de classes da aplicação desenvolvida. A classe `ADC` e responsável pela aquisição dos valores das grandezas físicas de tensões de correntes que são usadas como entradas no algoritmo. A aquisição é *offline*, os valores estão presentes numa *cache* obtida através da simulação de um sistema trifásico. A cada aquisição (função `acquire`) será retornado um objeto do tipo `gridSample`, que para além de conter os valores das tensões e correntes da rede, contém também o valor `vcc`, que se refere à tensão do barramento de tensão DC, que funciona como fonte de energia usada pelo inversor para gerar a corrente de saída. Este valor é importante para o algoritmo visto que o objetivo deste é também regular essa tensão, mantendo-a em níveis estáveis.

A classe `pqTheory` é onde está implementado o algoritmo de controlo. O algoritmo é baseado na Teoria da Potência Ativa e Reativa Instantânea [1] (**Teoria p-q**) e está dividido em várias funções, apresentadas na Figura 5.3. Os dados de entrada

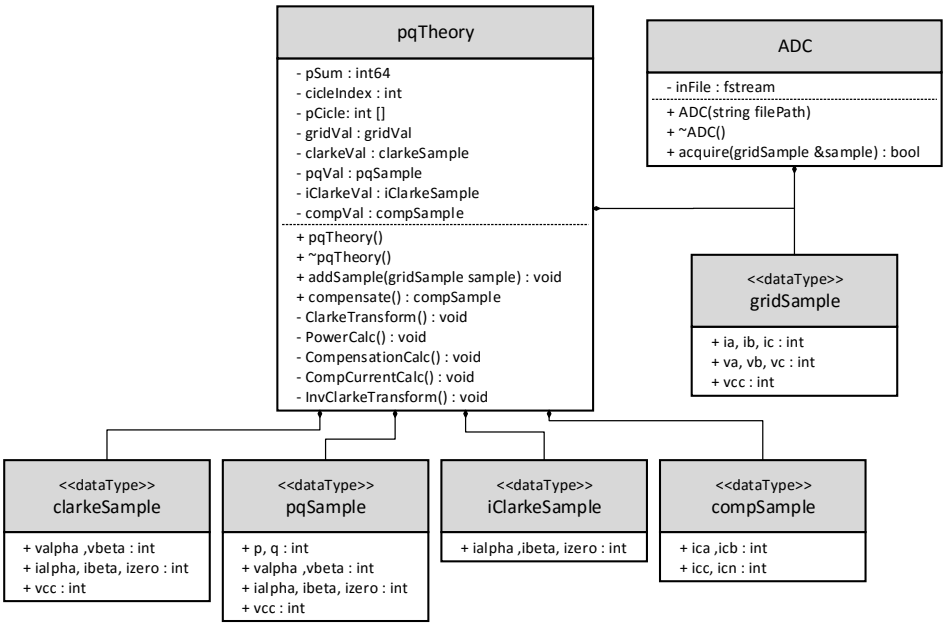


Figura 5.2: Diagrama de classes da aplicação de demonstração

são provenientes da classe `ADC`, que implementa a função membro `addSample`, e são executadas sequencialmente as seguintes funções:

- **Clarke Transform:** Aqui, usando a transformada de Clarke, os valores das componentes de tensão e corrente são transformados do sistema *abc* para o sistema $\alpha\beta 0$. O resultado é um objeto do tipo `clarkeSample`;

- **Power Calculation:** Apartir das componentes $\alpha\beta 0$ são calculadas as componentes de potência instantâneas - `p`, `q`. O resultado é um objeto do tipo `pqSample`, que contem as componentes de tensão e corrente em $\alpha\beta 0$, as potências instantâneas calculadas `p` e `q`, e o valor instantâneo da tensão barramento CC (`vcc`);

- **Compensation Calculation:** Aqui são calculadas os valores das potências de compensação. Na estratégia de potência constante na fonte são compensadas as componentes de potencia pulsantes e toda a energia reativa. Assim sendo é compensada a parte alternada da componente `p` e toda a componente `q`. É também compensada a energia gasta no barramento CC, de modo a manter a sua tensão dentro dos limites estabelecidos. O resultado é colado num objeto `pqSample`;

- **Compensation Current Calculation:** Aqui as componentes de potência

a compensar são transformadas nas relações de tensão e correntes em $\alpha\beta 0$ equivalentes. Ou seja, são calculadas as correntes $\alpha\beta 0$ que geram as potências de compensação com base nas valores atuais da tensão. O resultado é colocado num objeto do tipo `iClarkeSample`;

- **Inverse Clarke Transform:** Finalmente, as componentes $\alpha\beta 0$ das correntes de compensação são transformadas em componentes *abc*. Num sistema real esses valores são as referências de corrente para o inversor. O resultado é um objeto do tipo `compSample`;

Em suma, é calculada a potência instantânea presente no sistema, sobre essas potências são isoladas as componentes prejudiciais (as que o filtro deve gerar) e, por fim, são calculadas as correntes de geram essa potência.

De modo a tornar o algoritmo mais rápido e eficiente foi utilizada na sua implementação algoritmia de inteiros, em que nos cálculos são sempre utilizadas variáveis do tipo inteiro. A precisão de casas decimais é obtida através da multiplicação dos números por uma potência de base 2.

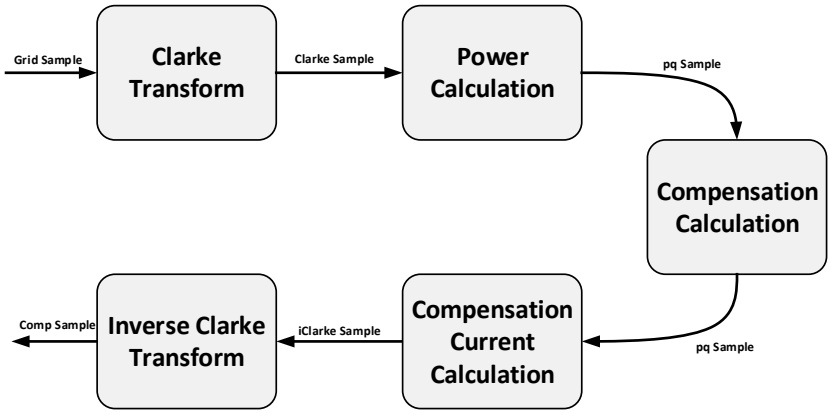


Figura 5.3: *Taskgraph* do algoritmo de cálculo de correntes de compensação

O algoritmo desenvolvido foi validado através da simulação de um FAP utilizando a ferramenta de simulação de circuitos de potência PSIM. Na Figura 5.4 pode ser observado o circuito de potência construido no software de simulação. No topo encontra-se a fonte trifásica, as respetivas impedâncias de linha, os sensores de medição das correntes/tensões e as cargas. No fundo da imagem pode ser observado o inversor que funcionará com a topologia fonte de corrente com controlo de tensão. Este transforma as referências de corrente fornecidas pelo algoritmo de controlo nas grandezas reais equivalentes.

O algoritmo de controlo é incluído na simulação através de uma Dynamic-link library (DLL), em que o seu código é executado a cada passo de integração da simulação. Como apresentado anteriormente este recebe as correntes/tensões da linha e devolve os valores de referência das correntes de compensação, estas são encaminhadas para o circuito de controlo do inversor (gatedrive) que gera o Pulse-Width Modulation (PWM) para as *gates* dos Mosfet do inversor.

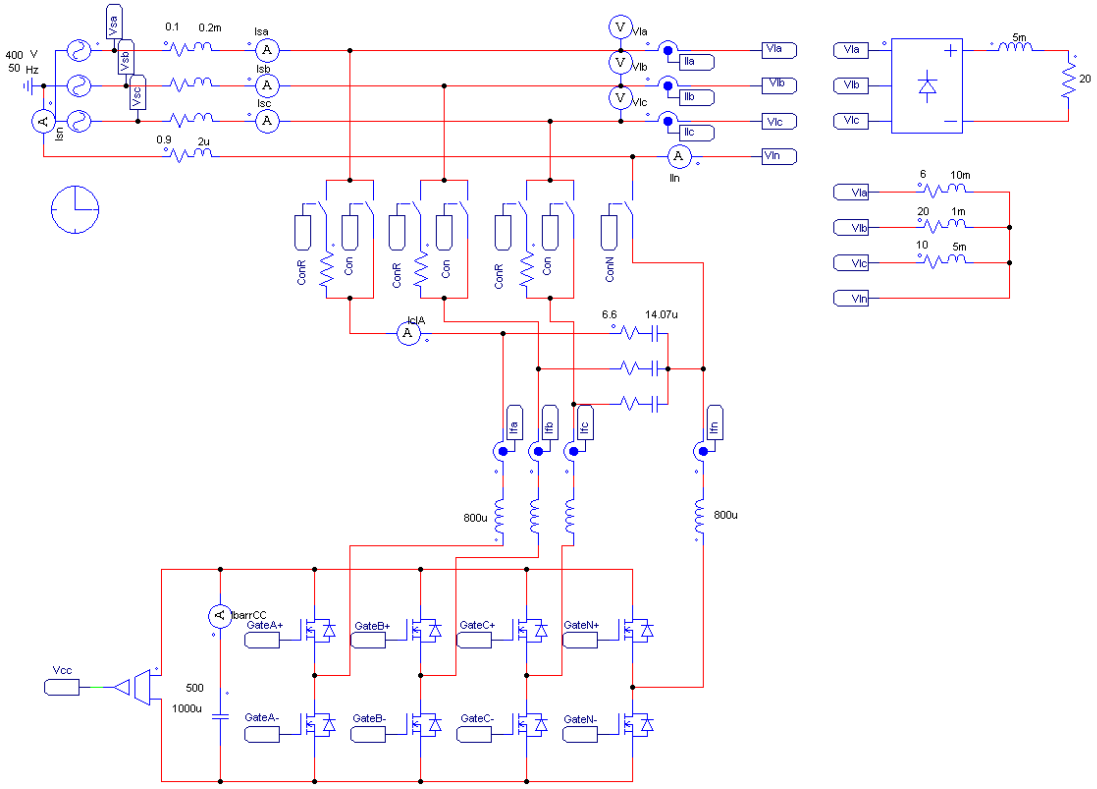


Figura 5.4: Circuito de potência do FAP

A Figura 5.5 apresenta os resultados da simulação do comportamento do FAP quando a rede alimenta simultaneamente duas cargas: um retificador trifásico com uma carga RL e uma carga RL desequilibrada. O FAP inicia, a compensação após um ciclo de rede (20 milissegundos),ou seja, aos 0.02 segundos de simulação. Este é o tempo necessário para o algoritmo de controlo obter os valores corretos das potências médias.

No gráfico `a)` são apresentadas as tensões da linha, aqui pode ser verificado que, antes da compensação, as tensões apresentam uns pequenos cortes. Estes cortes devem-se à variações rápidas das correntes consumidas pela carga que provocam uma queda de tensão elevada nas impedâncias de linha que contrariam a tensão da fonte. É verificado que, após a compensação, esses cortes são praticamente

suprimidos, isto porque o filtro elimina as variações rápidas de corrente na fonte.

O gráfico `b)` apresenta as correntes da fonte, aqui pode ser observado que estas, antes da compensação, se encontram distorcidas (com conteúdo harmónico) e desequilibradas (devido à presenta de uma carga desequilibrada). Também se verifica a existência de corrente no neutro, também devido à carga desequilibrada. Após o início da compensação as correntes da fonte ficam sinusoidais (sem conteúdo harmónico), equilibradas e em fase com a tensão. A corrente de neutro é também eliminada.

No gráfico `c)` são apresentadas as correntes da linha, onde não são verificadas alterações, o que significa que a carga não nota a presença do filtro.

No gráfico `d)` são apresentadas as correntes de saída do filtro. Antes do inicio de compensação verifica-se a existência de uma pequena corrente que se refere ao carregamento do condensador do barramento CC através dos díodos dos Mosfet. Após o início da compensação o filtro está a gerar as correntes de compensação, fornecidas pelo algoritmo de controlo.

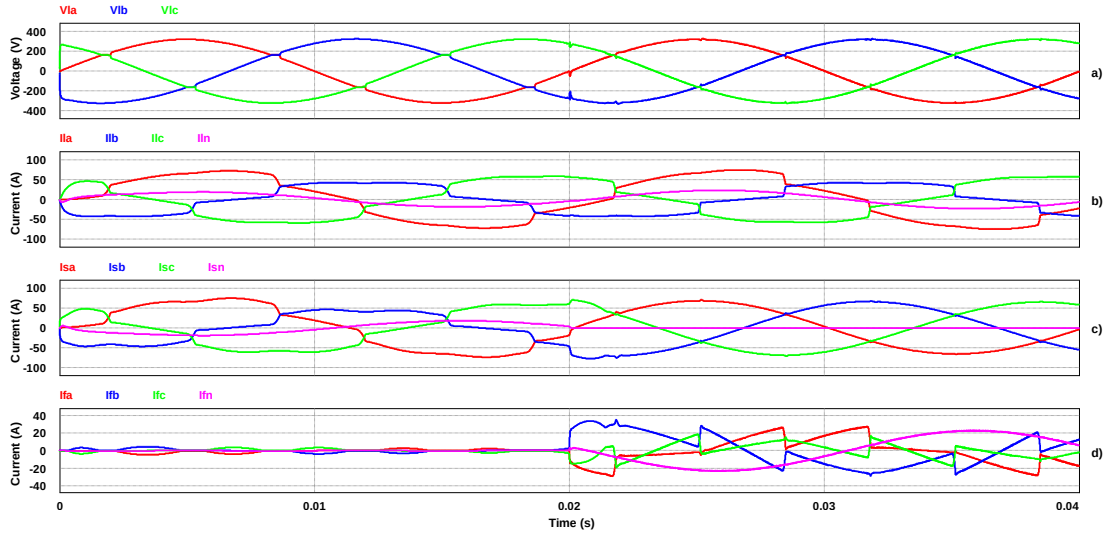


Figura 5.5: Resultados da simulação do FAP

5.2 Paralelização da aplicação

Depois de criada e validada a aplicação esta foi paralelizada. Para isso recorreu-se ao modelo de programação *multithreading* fornecido pelo MiLHA.

Cada uma das funções referidas na secção anterior foi isolada, foram verificadas as dependências de dados entre elas e foi adaptada para uma MiLHA `Software Task`. Esta adaptação com o auxilio do template fornecido que está disponível no Anexo A.

Para além das `MiLHA Tasks` que implementam o algoritmo fora criadas duas `MiLHA Tasks` adicionais, uma para o ADC *offline* e outra para mostrar os resultados através de um gráfico. Foram criados também diversos Tópicos para comunicação entre as `MiLHA Tasks`, tudo resultou no sistema apresentado na Figura 5.6. Na Figura 5.7 é apresentado o código de criação da aplicação.

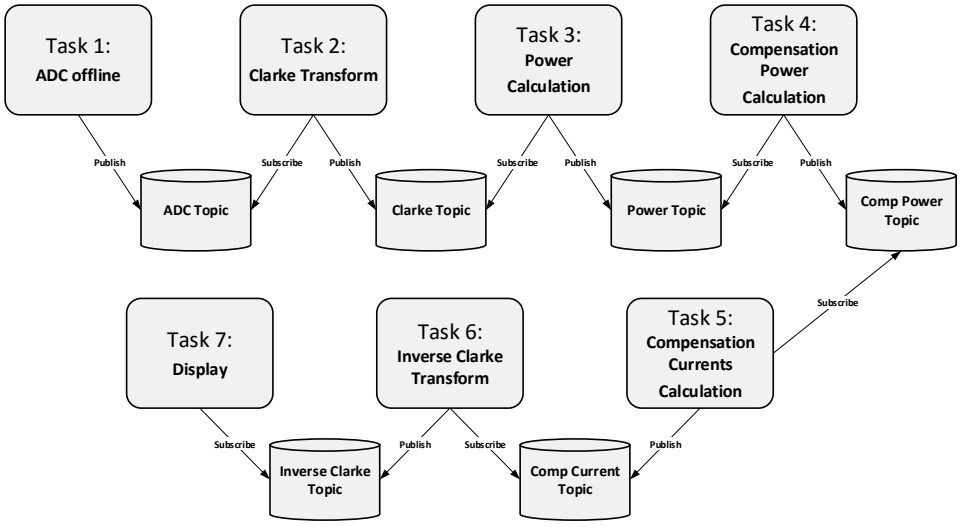


Figura 5.6: Esquema de Tasks e Tópicos da aplicação paralelizada em software

A aplicação foi posteriormente incluída num ambiente gráfico, criado com a *Qt Framework*, onde é criado um gráfico com o resultado final da aplicação, as quatro correntes de compensação. Para isso foi adicionado suporte a um monitor VGA na plataforma de desenvolvimento (Xilinx XUPV2P), e incluído o suporte à *Qt Framework* na imagem do sistema Linux. No monitor VGA é desenhado um gráfico com os dados recebidos pela `Task Display`, este gráfico é atualizado a cada 5 segundos. Na Figura 5.8 é possível observar a aplicação a ser executada na plataforma *target*.

Mais tarde (no momento da paralelização em hardware) a `Software Task` ADC foi substituída para uma `Hardware Task`. Para isso a única alteração necessária na aplicação foi a alteração da linha de código da criação da `Software Task` ADC para criar uma `Hardware Task`:

```
adcThread = mw->createHwTask("/dev/adc", "ADC");
```

```
CMiLHA *mw = CMiLHA::getInstance();
CTask *adcThread, *clarkThread, *powerThread;
CTask *compThread, *compCurrThread, *invClkThread, *displayThread;

mw->createTopic("ADC");
mw->createTopic("Clark");
mw->createTopic("Power");
mw->createTopic("CompCalc");
mw->createTopic("CurrentCalc");
mw->createTopic("InvClark");

adcThread        = mw->createSwTask("ADCT", ADC);
clarkThread      = mw->createSwTask("ClarkT", ClarkTransform);
powerThread      = mw->createSwTask("PowerCalc", PowerCalc);
compThread       = mw->createSwTask("CompCalc",CompensationCalc);
compCurrThread   = mw->createSwTask("CCurrentCalc", CCurrentCalc);
invClkThread     = mw->createSwTask("InvClarkT", InvClarkT);
displayThread    = mw->createSwTask("Display", display, w);

clarkThread->subscribe("ADC");
powerThread->subscribe("Clark");
compThread->subscribe("Power");
compCurrThread->subscribe("CompCalc");
invClkThread->subscribe("CurrentCalc");
displayThread->subscribe("InvClark");
```

Figura 5.7: Código de criação da aplicação software

Assim, a `Task` ADC foi migrada para hardware sem necessidade de qualquer outra alteração no resto da aplicação.

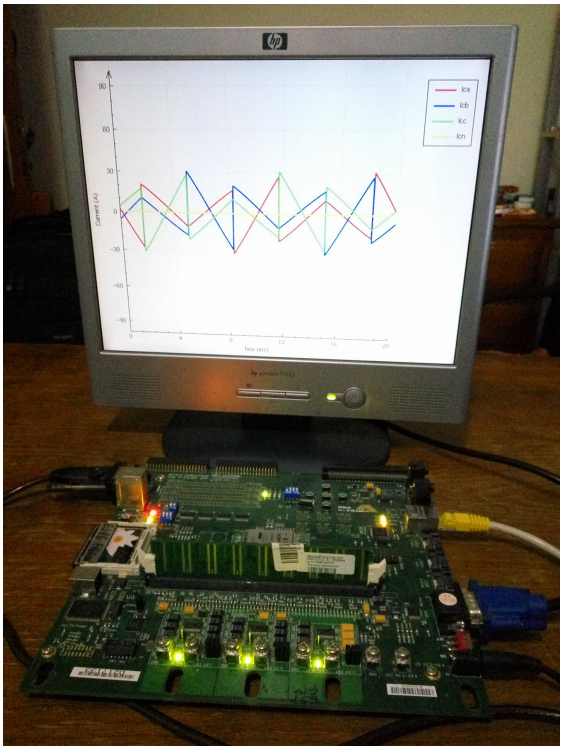


Figura 5.8: Aplicação gráfica na plataforma *target*

5.3 Aceleração da aplicação em hardware

Depois de desenhada e implementada em software, paralelizada com recurso aos serviços MiLHA, a aplicação foi implementada em hardware. Cada uma das `Software Tasks` foi implementada como um acelerador hardware. Para poderem ser integrados no MiLHA os aceleradores seguem o modelo de acelerador proposto, apresentado em 4.4.2. Para além das cinco `Tasks` que compõe o algoritmo de cálculo das correntes de compensação foi criado também um acelerador que irá simular um ADC, neste este acelerador inclui várias ROM pré-preenchidas com valores práticos de tensões e correntes de uma simulação, neste caso o ADC está a fornecer dados simulando a presença de um retificador trifásico com carga RL na rede.

A Figura 5.9 apresenta o *top-level* dos aceleradores desenvolvidos.

Posteriormente, cada acelerador foi incluído num *User Logic Wrapper* do MiLHA, disponível no Anexo C. Deste modo, cada acelerador passa a estar habilitado a fazer interface com um MiLHA Interface IP ou com um outro acelerador. Para

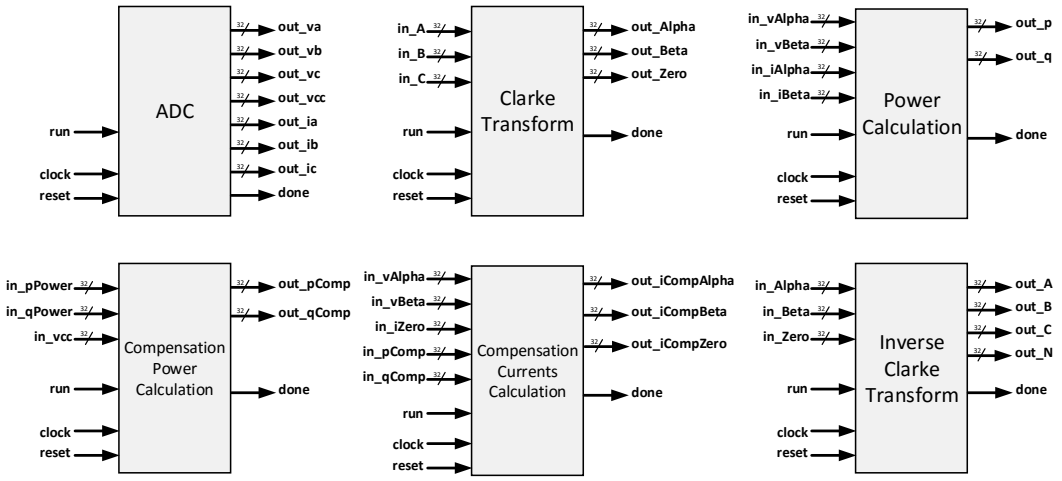


Figura 5.9: Top-Level dos aceleradores hardware da aplicação de demonstração

haver compatibilidade entre as `Software Tasks` e `Hardware Tasks` é necessário que ambas as interfaces de dados sejam compatíveis, isto é, recebam/enviem os dados nos mesmos formatos, nas mesmas quantidades e na mesma ordem. Para isso, nos *User Logic Wrapper* os dados que não sejam utilizados pelo acelerador mas que precisem de ser enviados para o seguinte acelerador são guardados temporariamente em registos enquanto os aceleradores processam. É o caso `Task Compensation Power Calculation` recebe dados do tipo `pqSample` e, apesar de não utilizar todos os dados contidos nessa estrutura de dados terá de os enviar para a seguinte `Task`.

Finalizada a integração do acelerador no *User Logic Wrapper* estes foram incluídos no SoC e criadas as ligações entre eles. Foram criadas duas configurações entre aceleradores distintas.

Na primeira configuração os seis (ADC e as cinco do algoritmo) aceleradores estão ligados em *array* e com apenas um módulo de interface com o MiLHA (Figura 5.10), o módulo foi configurado com o nome "pq", prioridade base 100 e para ativar a interrupção de leitura quando estiver presentes 50 amostras de dados (parâmetro `C_READ_THRESHOLD`). Nesta configuração, na aplicação software apenas irão ser recebidas os resultados finais do algoritmo (as correntes de compensação). A Figura 5.11 apresenta o código da criação da aplicação software com esta configuração. Nesta aplicação existe apenas uma `Hardware Task`, cujo *Proxy device* é o *"/dev/pq"* e que publica as correntes de compensação no tópico "InvClark". Este tópico é depois subscrito pela `Sofware Task` *display*.

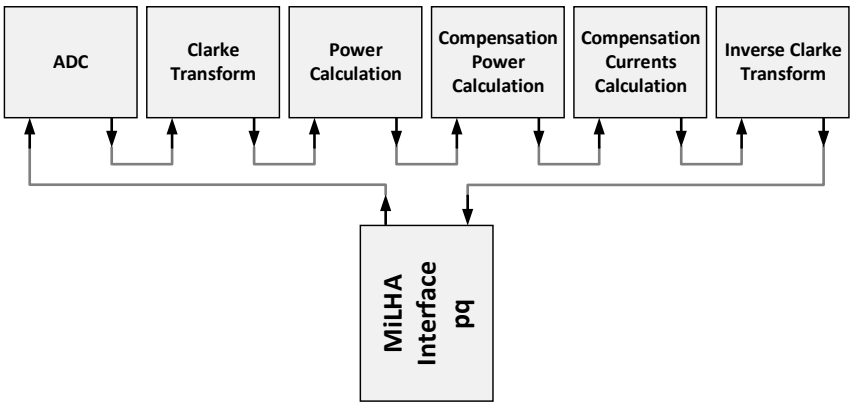


Figura 5.10: Configuração das ligações entre aceleradores em *array*

```
CMiLHA *mw = CMiLHA::getInstance();
CTask *pqHw, *displayT;

pqHw        = mw->createHwTask("/dev/pq","InvClark");
displayT    = mw->createSwTask("Display", display, w);
displayT->subscribe("InvClark");
displayT->run();
```

Figura 5.11: Código de criação da aplicação software com o *array* de aceleradores

Na segunda configuração o acelerador ADC foi isolado e ligado a um módulo de interface com o MiLHA, os cinco aceleradores de implementação do algoritmo foram ligados em *array*, que por sua vez se liga a um módulo de interface com o MiLHA (Figura 5.12). Esta configuração permite a utilização de mais funcionalidades do MiLHA, uma vez que na aplicação software é possível: receber os dados do ADC, enviar dados para o *array* de processamento do algoritmo de cálculo das correntes de compensação e receber as correntes de compensação.

Na Figura 5.13 é apresentado o código da criação da aplicação software nesta configuração. Existem duas `Hardware Tasks`: a ADC (*"/dev/adc"*), que publica os dados do ADC no tópico "ADC"; e a Hwpq, que representa o array de aceleradores, publicada em *"/dev/pq"*, subscreve o tópico "ADC" para receber os dados do ADC e publica as correntes de compensação nono tópico "compSamples". É criada uma `Software Taks ghostADC`, que subscreve também o tópico ADC apenas para consumir os dados nos *buffers* de dados, uma vez que o MiLHA não está preparado a troca de dados entre `Hardware Tasks` em software e prevê que para isso seja utilizado o mecanismo de troca de dados entre aceleradores. Por fim é criada a `Sofware Taks` *display* que subscreve o tópico "compSamples" para obter

as correntes de compensação.

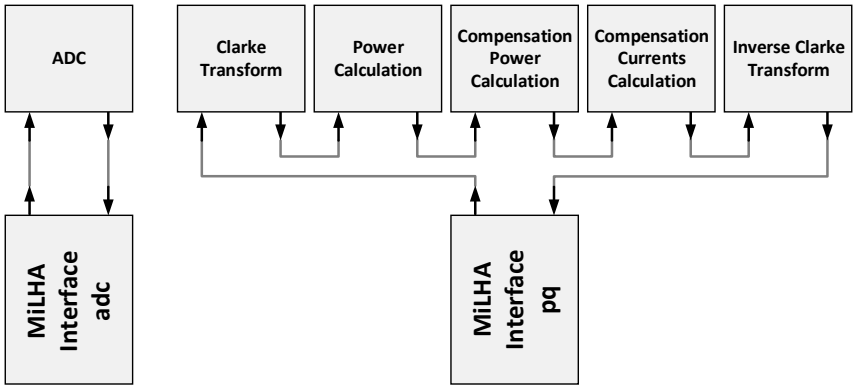


Figura 5.12: Configuração das ligações entre aceleradores em ADC + *array*

```
CMiLHA *mw = CMiLHA::getInstance();
CTask *hwADC, *swADC, *hwpq, *displayT;

mw->createTopic("ADC");
mw->createTopic("compSamples");

hwADC    = mw->createHwTask("/dev/adc", "ADC");
hwpq     = mw->createHwTask("/dev/pq", "compSamples");
swADC    = mw->createSwTask("ghostADC", ghostADC);
displayT = mw->createSwTask("Display", display, w);

swADC->subscribe("ADC");
hwpq->subscribe("ADC");
displayT->subscribe("compSamples");

swADC->run();
displayT->run();
```

Figura 5.13: Código de criação da aplicação software ADC + *array*

5.4 Performance e resultados

Sobre aplicação de demonstração foram efetuadas algumas medições de modo a se poder obter algumas noções sobre a performance do `MiLHA` e da aceleração em hardware.

Para isso foi implementado um periférico contador dedicado, que foi ligado ao SoC através do barramento PLB. Este periférico contém dois registos: um registo de controlo e um registo de contagem. O registo de contagem é incrementado à

frequência do barramento PLB desde que o bit menos significativo do registo de controlo seja ativado. Ambos os registos podem ser lidos e escritos.

Posteriormente os endereços físicos dos registos do contador foram mapeados em *user* e *kernel space* e foram realizadas algumas medições dos tempos de processamento das funcionalidades do MiLHA. A natureza indeterminística do sistema operativo Linux alastra-se as medições efetuadas, visto que a porção de código que está a ser avaliada pode ser interrompida, a *thread* pode ser suspensa, o que altera drasticamente os resultados obtidos. De modo a contrariar essa natureza cada medição foi efetuada milhares de vezes e calculada a média desses resultados.

A Tabela 5.1 apresenta os resultados das medições sobre o tempo de execução do algoritmo de cálculo das correntes de compensação. No processamento em software os resultados traduzem o tempo desde que os dados chegam à primeira *Software Task* (*ClarkeTransform*) até serem obtidos os resultados na última (*Display*). As medições do processamento em hardware foram efetuadas na aplicação com a configuração *ADC + pq*, os resultados traduzem o tempo desde a receção dos dados do acelerador *ADC*, até à receção dos resultados do acelerador *pq*. Isto inclui o envio dos dados para o acelerador *pq*, o processamento e o envio dos resultados de volta para o software. Como resultados só são recebidos a cada 50 amostras, isto porque o IP `MiLHA Interface pq` acumula as 50 amostras na FIFO até ativar a interrupção de leitura, portanto o tempo medido traduz o tempo de cálculo de 50 amostras. Verifica-se que na aplicação software cada amostra demora, em média, cerca de 95 mil ciclos de clock do CPU a ser executada, enquanto que em hardware 50 amostras demoram 26 mil ciclos, o que corresponde a cerca de 530 ciclos por amostra, que se traduz num aumento de velocidade de cerca de 182 vezes.

Tabela 5.1: Resultado da medições do tempo de execução do algorimo de cálculo das correntes de compensação

	CPU Clocks	**Time (μs)**
Cálculo em software	95068	2852,04
Cálculo em hardware	26540	796,19

De modo caracterizar um pouco o sistema MiLHA foram efetuadas algumas medições sobre a execução das suas funcionalidades básicas. Numa aplicação construida com o sistema MiLHA estas funcionalidades são constantemente executadas, portanto influenciam muito a performance da aplicação. Na Tabela 5.2 estão apresentados os resultados das medições efetuadas, utilizando as aplicações de demonstração em que todas as trocas de dados se referiam a uma amostra do *ADC*

com o tamanho de 7 inteiros (28 bytes).

A tabela inicia-se com as medições das funcionalidades sobre os tópicos, primeiro com *publishers/subscribers* em software e depois em hardware, aqui destaca-se com um maior tempo de execução a publicação num tópico com um subscritor em hardware, que se traduz numa escrita num acelerador. Esta escrita engloba a cópia dos dados para o *buffer* partilhado e a poterior *system call* de notificação de escrita, por sua vez esta *system call* irá ativar a *thread* de execução, neste ponto de escalonamento a *thread* da aplicação poderá ser suspensa.

Na segunda parte da tabela estão apresentadas algumas medições sobre as atividades do `Kernel MiLHA`. É apresentado o tempo processamento dos pedidos (7 inteiros) de escrita/leitura pelas respetivas *threads*, os valores referem-se ao tempo desde o inicio da execução do pedido até à sua finalização. É também apresentado o tempo total de processamento dos pedidos desde que o `MiLHA` é notificado (o pedido é criado), através da *system call write notify*no caso das escritas ou da interrupção no caso das leituras, até à sua conclusão. Isto inclui a criação do pedido, a ativação da *thread* de execução, a execução e a eliminação do pedido.

Através das medições efetuadas verifica-se que a utilização de serviços `MiLHA` inclui um *overhead* de execução significativo, este prevê-se que possa diminuir significativamente com a utilização do *real-time patch* para o Linux.

Tabela 5.2: Medições de tempos de execução das várias funcionalidades do MiLHA

	CPU Clocks	**Time (μs)**
Check de um tópico com SW publisher	76	2,27
Ler de um tópico com SW publisher	84	2,51
Publicação num tópico com SW subscriber	144	4,32
Check de um tópico com HW publisher	115	3,45
Leitura de um tópico com HW publisher	175	5,24
Publicação num tópico com HW subscriber	16163	484,89
Execução de pedido de escrita	3619	108,56
Execução de pedido de leitura	3790	113,71
Processamento de pedido de escrita (desde notificação)	6476	194,28
Processamento de pedido de leitura (desde interrupção)	22809	684,26

Capítulo 6

Conclusão

Neste capítulo são abordadas as principais conclusões acerca do trabalho desenvolvido no âmbito da presente dissertação, assim como apresentadas propostas para uma futura continuação do seu desenvolvimento.

Dada a ambição e abrangência temática deste projeto, durante o seu desenvolvimento foram aplicados conhecimentos sobre áreas como *design* de sistemas embebidos; compilação e integração do Linux numa plataforma; desenvolvimento de software em Linux (incluindo *device drivers*); *design*, simulação e síntese de co-processadores em hardware, bem como alguns conhecimentos em eletrónica de potência, durante a simulação do FAP. Estes conhecimentos foram adquiridos ao longo do curso e aprofundados durante do desenvolvimento deste projeto.

Para além das competências técnicas adquiridas, foram desenvolvidas novas competências na resolução das diferentes tarefas propostas. A pesquisa bibliográfica e série de tentativas na procura e otimização de soluções proporcionaram a aprendizagem, melhoria e a conclusão deste projeto.

6.1 Trabalho desenvolvido

Esta dissertação descreve as considerações e as soluções de implementação sobre o serviço MiLHA. MiLHA é um serviço ao programador de suporte à aceleração em hardware de aplicações baseadas em Linux. MiLHA cria um modelo de programação que suporta a utilização de aceleradores em hardware como unidades de processamento.

Numa fase inicial do projeto foi elaborado um estudo sobre as tecnologias que envolvem este tipo de aplicações, nomeadamente, Linux, ferramentas de programação em hardware, plataformas híbridas, entre outras. Foram analisadas outras soluções existentes na comunidade científica, de onde surgiram ideias para a conceção do MiLHA. Este foi idealizado de acordo com o âmbito da dissertação, os recursos e o tempo útil disponíveis.

Como já referido, o MiLHA pode ser dividido em três camadas; uma que cria um modelo de programação paralelo, outra de interface com os aceleradores e outra de integração dos aceleradores no SoC.

Inicialmente foi criada a camada central do MiLHA, a camada que implementa a interface com os aceleradores. Nesta camada foram desenvolvidos mecanismos para identificar os aceleradores customizados presentes na plataforma (através da *device-tree*), incluí-los no sistema operativo Linux e fornecer uma interface a nível da aplicação de comunicação direta com os aceleradores.

Sobre essa camada foi desenvolvido o modelo de programação do MILHA. Aqui foi criado o conceito de `MiLHA Task`, uma entidade que representa de forma transparente os dois tipos de processamento (hardware e software). `MiLHA Software Task` corresponde ao processamento em software e corresponde a um modelo de criação de uma *thread*. Por outro lado, `MiLHA Hardware Task` representa o processamento em hardware sob a forma de um acelerador. No modelo de programação é possível criar uma relação de dependência de dados entre `MiLHA Tasks`, baseada no *Publish Subscribe Pattern*. Após definida a relação entre `MiLHA Tasks`, o MiLHA encaminha automaticamente os dados entre as diferentes *threads* ou utiliza a camada de interface com os aceleradores para encaminhar e receber dados dos aceleradores.

A camada de integração dos aceleradores no SoC foi criada para unificar a interface dos aceleradores com o MiLHA e para facilitar a integração do acelerador, tendo sido definido um modelo simples de acelerador em hardware. Sobre o acelerador codificado sobre esse modelo deve ser aplicado um *wrapper* fornecido pelo MiLHA que permite ligação ao módulo de interface com o MiLHA. Este módulo liga-se ao barramento do sistema e serve de ponte entre o software e o acelerador.

Finalmente foi criada uma aplicação de demonstração, a qual implementa o controlo de um Filtro Ativo Paralelo. Essa aplicação foi acelerada em hardware, segundo o *design flow* adotado onde foram aplicados os serviços MiLHA. Na cons-

trução da aplicação o MiLHA verificou-se bastante útil dado que facilitou a integração dos aceleradores; permitiu alterar facilmente as configurações de ligações entre aceleradores; as alterações na configuração da aplicação utilizando o modelo de programação fornecido foram efetuadas de uma maneira rápida e simples, escolhendo facilmente quais as `MiLHA Tasks` que devem ser executadas em software e em hardware, quase sem alterações na aplicação inicial.

6.2 Trabalho Futuro

Durante o desenvolvimento desta dissertação, e mesmo durante a fase final em modo de retrospetiva, surgiram várias ideias de melhoramento e sugestões de exploração de outras topologias. Essas sugestões ficam aqui registadas para a possibilidade de uma futura continuação deste projeto.

Na API do MiLHA, onde é implementado o modelo de programação paralelo e a interface com o programador sugere-se:

- Utilização da técnica de *template metaprogramminig* para gerir a variabilidade em tempo de compilação, aumentando a performance do programa;
- Melhorar o conceito de `MiLHA software Task`, fornecendo mais mecanismos de sincronização entre as `threads`.
- Exploração do conceito de `thread` híbrida, uma *thread* executada simultaneamente em hardware e em software, aumentando ainda mais o seu *throughput*.

Na camada central do MiLHA, a camada de interface com os aceleradores, sugerem-se melhorias na execução das trocas de dados, nomeadamente, executar as trocas de dados com recurso ao hardware, através de um periférico *master* do barramento ou de *DMA*, libertando o processador deste processamento intensivo.

Na camada de integração do hardware sugere-se a adaptação do hardware desenvolvido para plataformas mais recentes, para, por exemplo, suportar o barramento AXI. Sugere-se também a melhoria do modelo de acelerador, acrescentando novas funcionalidades como por exemplo registos de controlo intrínsecos ao modelo de acelerador.

De forma geral, o MiLHA fornece serviços na etapa de paralelização em hardware do *design flow* de aceleração de aplicações em hardware. Sugere-se a exploração da

possibilidade de adicionar serviços sobre outras etapas do processo de paralelização em hardware, como por exemplo no *profiling* da aplicação software.

Bibliografia

[1] H. Akagi, E. H. Watanabe, and M. Aredes, *Instantaneous Power Theory and Applications to Power Conditioning*. Wiley-Interscience, 2007.

[2] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, and E. Komp, "Programming Models for Hybrid FPGA-CPU computational Componetns: A Missing Link," *Ieee Micro*, pp. 42–53, 2004.

[3] B. Arzhanov, "Kernel de Tempo Real assisitido por Hardware," Master Thesis, University of Minho, 2013.

[4] S. Bailey, "Comparison of VHDL, Verilog and SystemVerilog," 2003.

[5] M. Barr, *Programming Embedded Systems in C and C ++*, 1999, no. January.

[6] T. Bishop and R. Karne, "A survey of middleware," *Proceedings of the ISCA 18th International Conference Computers and Their Applications, Honolulu, Hawaii, USA, March 26-28, 2003*, pp. 254–258, 2003.

[7] Buildroot, "The Buildroot user manual," p. 131, 2015. [Online]. Available: http://buildroot.uclibc.org/downloads/manual/manual.pdf

[8] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," *2008 Symposium on Application Specific Processors, SASP 2008*, pp. 101–107, 2008.

[9] P. Cobbaut, *Linux Fundamentals*, 2015.

[10] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly, 2005.

[11] A. Corporation, "White Paper Accelerating High-Performance Computing With FPGAs," *Cluster Computing*, no. October, pp. 1–8, 2007.

[12] H. M. Deitel, P. J. Deitel, and D. R. Choffnes, *Operating Systems*, 3rd ed. Prentice Hall, 2004.

[13] Florida Internation University, "PowerPC based embedded design in FPGA using Xilinx EDK." [Online]. Available: http://web.eng.fiu.edu/$\sim$gaquan/teaching/ccli/EDK.pdf

[14] I. Grout, *Digital Systems Design with FPGAS*. Newnes, 2008, vol. 1.

[15] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," *Computer*, vol. 40, pp. 50–57, 2007.

[16] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani, and M. Chiu, "Computing models for FPGA-based accelerators," *Computing in Science and Engineering*, vol. 10, pp. 35–45, 2008.

[17] IBM, "PowerPC 405 User's Manual," 2001.

[18] R. Jidin, "Extending the Thread Programming Model Across CPU and FPGA Hybrid Architectures," Ph.D. dissertation, University of Kansas, 2005.

[19] A. Journal and O. F. Basicapplied, "Design and Implementation of FPGA-Based Systems -A Review," no. October 2015, 2009.

[20] R. Kirchgessner, A. D. George, and G. Stitt, "Low-Overhead FPGA Middleware for Application Portability and Productivity," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 21:1—-21:22, 2015.

[21] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007.

[22] G. Likely and J. Boyer, "A Symphony of Flavours: Using the device tree to describe embedded hardware," vol. 2, pp. 27–37, 2008.

[23] Lister and Andrew, *Fundamentals of operating systems*. Springer Science & Business Media, 2013.

[24] Y. Liu and B. Plale, "Survey of Publish Subscribe Event Systems," *Indiana University Department of Computer Science*, 2003.

[25] E. Lübbers, "Multithreaded Programming and Execution Models for Reconfigurable Hardware," Ph.D. dissertation, University of Paderborn, 2010.

[26] C. Maxfield, *FPGAs: World Class Designs*. Newnes, 2009.

[27] Microsoft, "Publish/Subscribe." [Online]. Available: https://msdn.microsoft.com/en-us/library/ff649664.aspx

[28] C. Monteiro, "Framework compatível com Repositório IP-XACT para o Domínio Específico de Aplicações," Master Thesis, University of Minho, 2015.

[29] N. Naia, "Real-Time Linux and Hardware Accelerated Systems on QEMU and Computers Declaração," Master Thesis, University of Minho, 2015.

[30] T. Noergaard, *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers*, 2005.

[31] P. Oliveira, "Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux," Master Thesis, University of Minho, 2013.

[32] S. Panneerselvam and M. Swift, "Operating systems should manage accelerators," *Whtp*, pp. 1–7, 2012. [Online]. Available: https://www.usenix.org/system/files/hotpar12-final40.pdf

[33] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A Computational Model for Reconfigurable Devices," *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1–4, 2006.

[34] ReconOS, "ReconOS." [Online]. Available: http://www.reconos.de/

[35] D. C. Schmidt, "Middleware techniques and optimizations for real-time, embedded systems," *System Synthesis, 1999. Proceedings. 12th International Symposium on*, pp. 12–16, 1999.

[36] ——, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, no. 6, pp. 43–48, 2002.

[37] V. Silva, "Sistema de Aquisição de Sados Tempo Real baseado em Linux," Master Thesis, University of Minho, 2011.

[38] M. Sousa, "Hybrid Linux-Based Real-Time Acquisition System," Master Thesis, University of Minho, 2015.

[39] Xilinx, "Zynq UltraScale+ EV." [Online]. Available: http://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-ev-block.PNG

[40] ——, "Virtex-II Pro Hardware Reference Manual," pp. 1–142, 2005.

[41] ——, "PowerPC 405 Processor Block Reference Guide," *ReVision*, vol. 018, p. 252, 2010.

[42] ——, "EDK Concepts, Tools, and Techniques A Hands-On Guide to Effective Emedded System Design," 2013.

[43] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building Embedded Linux Systems*, 2nd ed. O'Reilly, 2008, vol. 53, no. 9.

Apêndice A

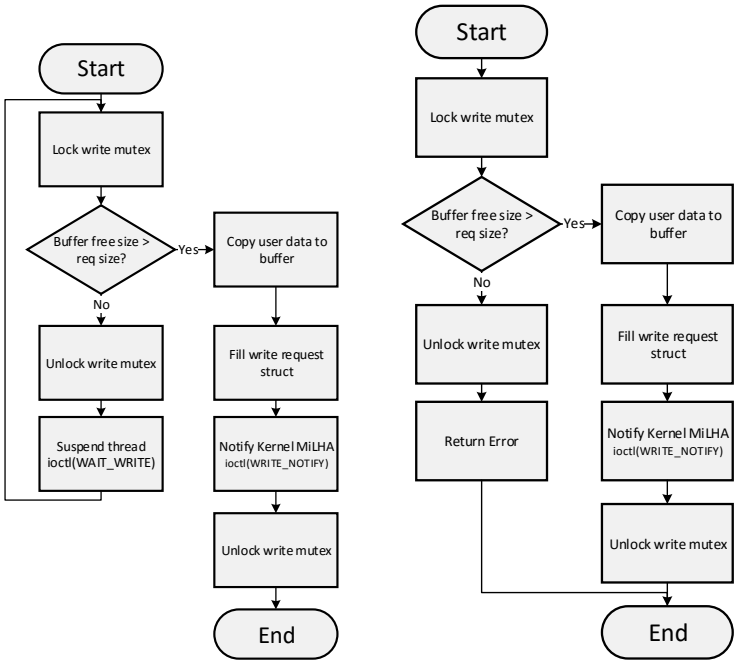
Template de função de thread

```
void ThreadFunction(CMailBox* mailBox[], void* arg)
{
        CMiLHA *inst     = CMiLHA::getInstance();
        CTopic *topic    = inst->getTopic("TopicName");

        IN_DATA_TYPE     in_data;
        OUT_DATA_TYPE    out_data;

        unsigned int     check;

        while(1)
        {
                check = mailBox[0]->check();     //check the amount
                if(check >= IN_DATA_SIZE)        //of data avaiable
                {
                        mailBox[0]->read(&in_data, DATA_SIZE);
                        /*
                                Do something with in_data
                        */
                        topic->publish(&out_data, OUT_DATA_SIZE);
                }
                else
                {
                        sched_yield();
                }
        }
}
```

Apêndice B

Processos de leitura e escrita num acelerador

B.1 Processo de Escrita



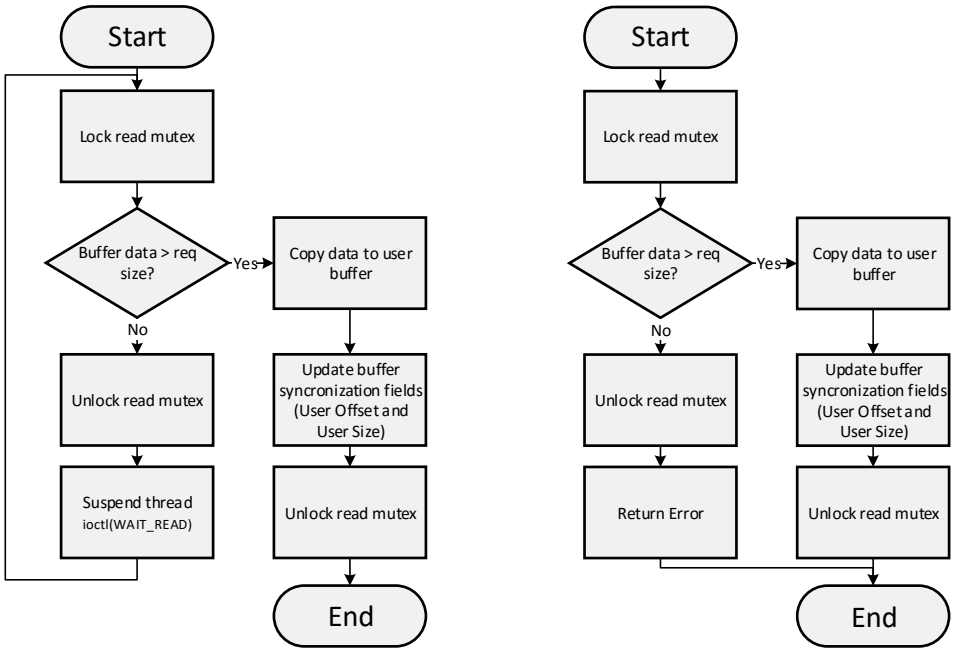
(a) Pedido *blocking* (`write`) (b) Pedido não *blocking* (`try_write`)

Figura B.1: Processamento de um pedido de escrita

O processo inicia-se com a aquisição de um mutex associado à escrita do respe-

tivo Device Proxy, seguidamente é verificado se existe espaço livre suficiente no *buffer* de escrita para a quantidade de dados requisitada. Se a condição falar (não existir espaço suficiente) o *mutex* é libertado e a `thread` é suspensa no caso do serviço *blocking* (`write`) ou é retornado erro, no caso do serviço *non-blocking*. Se existir espaço suficiente no *buffer* o processo pode prosseguir, os dados são copiados para o *buffer* partilhado, é preenchida uma estrutura com alguma informação sobre o pedido, nomeadamente o tamanho, a prioridade e o *offset* de escrita, que seguidamente é encaminhada para o MiLHA Kernel onde será registado o pedido.

B.2 Processo de Leitura



(a) Pedido *blocking* (`read`) (b) Pedido não *blocking* (`try_read`)

Figura B.2: Processamento de um pedido de leitura

O procedimento para a transferência de dados para a aplicação inicia-se com a aquisição do mutex associado, é verificado se existe a quantidade de dados necessária no *buffer* partilhado, através do campo `User Size`. Se não houver dados suficientes o `mutex` é libertado e a *thread* é suspensa (*blocking*) ou é retornado erro (*non-blocking*). No caso da existência da quantidade de dados requisitada estes são copiados para o *buffer* da aplicação, as váriáveis de sincronização do *buffer*

partilhado (`User Offset` e `User Size` são atualizadas e, finalmente, o mutex é libertado.

Apêndice C

Template de User Logic Wrapper

```
module milha_user_template(

input clk,
input rst,
input in_valid,
input [31:0] in_data,
output in_ack,
output out_valid,
output [31:0] out_data,
input out_ack
);

//*** Instantiate user modules here  ****//
wire instance_run;
wire instance_done;

wire [31:0]      in_0;
wire [31:0]      in_1;
wire [31:0]      in_2;

wire [31:0]      out_0;
wire [31:0]      out_1;

module_name instance_name(
        .clock(clk),
        .reset(rst),

        .run(instance_run),
        .done(instance_done),
```

```
        .in_0(in_0),
        .in_1(in_1),
        .in_2(in_2),
        .out_0(out_0),
        .out_1(out_1)
);

/***********************************
*       INPUT / OUTPUT INTERFACE        *
***********************************/

//** Specify number of input and output parameters  **//
parameter N_INPUT_PARAMETERS = 3;
parameter N_OUTPUT_PARAMETERS = 2;

//inputs/outputs words in the same sequence that software

reg     [31:0]  inputs  [N_INPUT_PARAMETERS-1:0]; // Input data is
    here, assign this to your user logic inputs
wire    [31:0]  outputs [N_OUTPUT_PARAMETERS-1:0]; // assign this to
    your outputs

reg process_run;        // Signal user logic to run
wire process_done;      // User logic done complete  signal

/**** Assign  triggers, inputs and outputs signals here ****/
assign instance_run = process_run;
assign process_done = instance_done;

assign in_0 = inputs[0];
assign in_1 = inputs[1];
assign in_2 = inputs[2];

assign out_0 = outputs[0];
assign out_1 = outputs[1];

/***** DO NOT EDIT BELOW THIS LINE *****/

/***********************************
*               INPUT DATA LOGIC
***********************************/
reg prev_process_done;
```

```
always @(posedge clk)
        if (rst)
                prev_process_done <= 1;
        else
                prev_process_done <= process_done;

assign posedge_process_done = process_done & !prev_process_done;
assign negedge_process_done = !process_done & prev_process_done;

reg     [3:0]   pop_index;
wire enough_pop = (pop_index == N_INPUT_PARAMETERS);

localparam pop_initial = 2'd0, pop_reading = 2'd1,
        pop_increment = 2'd2, pop_run = 2'd3;

reg [1:0]pop_state;

reg in_ack_reg;
assign in_ack = in_ack_reg;

always @(posedge clk)
begin
        if(rst)
        begin
                pop_state <= pop_initial;
                pop_index <= 0;
                process_run <= 0;
                in_ack_reg <= 0;
        end
        else
        begin
                if(pop_state == pop_initial)
                begin

                        if(in_valid)
                        begin
                                pop_state <= pop_reading;
                                inputs[pop_index] <= in_data;
                                in_ack_reg <= 1;
                        end
                end

                if(pop_state == pop_reading)
                begin
```

```
                        if(!in_valid)
                        begin
                                pop_state <= pop_increment;
                                pop_index <= pop_index +1;
                                in_ack_reg <= 1;
                        end
                end

                if(pop_state == pop_increment)
                begin
                        if(in_valid && !enough_pop)
                        begin
                                pop_state <= pop_reading;
                                inputs[pop_index] <= in_data;
                                in_ack_reg <= 1;
                        end

                        if(enough_pop)
                        begin
                                pop_state <= pop_run;
                                process_run <= 1;
                        end
                end
                if(pop_state == pop_run)
                begin
                        if(negedge_process_done)
                        begin
                                pop_index <= 0;
                                pop_state <= pop_initial;
                                process_run <= 0;
                        end

                end

        end
end

/*************************************
*               OUTPUT DATA LOGIC
*************************************/
reg [31:0]      outputs_reg [N_OUTPUT_PARAMETERS-1:0];
reg [4:0]push_index;
wire enough_push = (push_index == N_OUTPUT_PARAMETERS);
```

```
localparam push_initial = 2'd0, push_writing = 2'd1, push_increment =
    2'd2;
reg [1:0]push_state;

reg out_valid_reg;
assign out_valid = out_valid_reg;

reg [4:0]index;

assign out_data = outputs[push_index];

always @(posedge clk)
begin
        if(rst)
        begin
                push_state <= push_initial;
                push_index <= 0;
                out_valid_reg <= 0;
        end
        else
        begin
                if(push_state == push_initial)
                begin
                        if(posedge_process_done)
                        begin
                                push_state <= push_writing;
                                out_valid_reg <= 1;

                                for (index=0; index <
                                   N_OUTPUT_PARAMETERS; index=index
                                   +1)
                                        outputs_reg[index] <= outputs
                                           [index];
                        end
                end

                if(push_state == push_writing)
                begin
                        if(out_ack)
                        begin
                                push_state <= push_increment;
                                push_index <= push_index + 1;
                                out_valid_reg <= 0;
                        end
```

```
                end

                if(push_state == push_increment)
                begin
                        if(!enough_push)
                        begin
                                push_state <= push_writing;
                                out_valid_reg <= 1;
                        end

                        if(enough_push)
                        begin
                                push_state <= push_initial;
                                push_index <= 0;
                        end
                end
        end
end
endmodule
```