Universidade do Minho
Escola de Engenharia

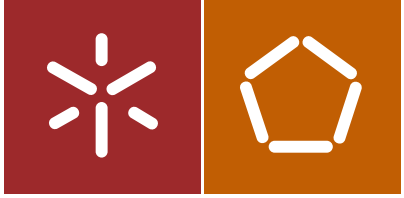
João Fernando da Silva Martins

Desenvolvimento de um System-on-Chip baseado em Microblaze para aplicações automóveis

Novembro de 2014

João Fernando da Silva Martins Desenvolvimento de um System-on-Chip baseado em Microblaze para aplicações automóveis

UMinho | 2014



Universidade do Minho
Escola de Engenharia

João Fernando da Silva Martins

Desenvolvimento de um System-on-Chip baseado em Microblaze para aplicações automóveis

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 Conceição Tavares

Novembro de 2014

AGRADECIMENTOS

Quero aproveitar este espaço e tempo para agradecer a todas as pessoas que me ajudaram a ser quem sou, especialmente a quem contribui para a realização deste trabalho.

Em primeiro lugar gostaria de agradecer aos meus pais, João Fernando de Freitas Martins e Maria José Freitas da Silva, por todo o apoio prestado ao longo do percurso académico, sem o qual não seria possível conclui-lo. A eles o meu muito obrigado por todo o apoio.

Ao meu orientador Professor Doutor Adriano Tavares pelo apoio prestado, confiança depositada e conhecimento transmitido ao longo do desenvolvimento desta dissertação.

Ao ESRG (*Embedded Systems Research Group*) do departamento de Eletrónica Industrial da Universidade do Minho por proporcionar todas as condições necessárias para o desenvolvimento da presente dissertação de mestrado e ao aluno de doutoramento Tiago Gomes pela ajuda prestada na construção do relatório desta dissertação.

Aos colegas que me acompanharam ao longo do curso, em especial aos meus colegas de laboratório Pedro Matos, Tiago Vasconcelos, Rui Machado, Eurico Esteves, Davide Guimarães, Filipe Alves e Vasco Lima pela entreajuda, companheirismo e momentos de descontração.

Por fim, mas não menos importante, à minha espetacular namorada Diana Neto.

RESUMO

Hoje em dia deseja-se implementar num *chip* o maior número de funções possíveis, o que faz diminuir o número de microcontroladores necessários para uma determinada aplicação e assim a consequente diminuição de custos. O aparecimento das FPGAs de baixo custo nos últimos anos, levou à implementação de sistemas baseados em plataformas reconfiguráveis uma vez que as suas características permitem uma rápida prototipagem de diferentes implementações facilitando o desenvolvimento de vários projetos. A sua flexibilidade permite aos *designers* criar módulos customizáveis e específicos à aplicação. As FPGAs permitem a implementação de SoCs dedicados a aplicações onde métricas como desempenho, determinismo e *time-to-market* são muito importantes em sistemas de tempo real. A implementação de um SoC numa FPGA oferece um bom equilíbrio entre a flexibilidade de implementação e um rápido *time-to-market*.

Esta dissertação passa por desenvolver um SoC orientado para aplicações automóveis. O SoC está dotado de um controlador de interrupções baseado no NVIC da ARM que permite o atendimento a interrupções com uma latência muito baixa e um *array* de *timers* baseado nos *timers* presentes no microcontrolador 32-bit TriCore™. A implementação destes periféricos permite a utilização deste SoC em aplicações automóveis devido ao determinismo que o mesmo oferece, bem como o seu desempenho.

O processador do SoC desenvolvido é baseado no *Microblaze*, que segue uma arquitetura de processadores RISC como o DLX, um processador muito utilizado para o ensino ao longo dos anos. O processador implementa um *datapath* de cinco estágios de *pipeline*, de forma a aumentar o número de instruções executadas por unidade de tempo, possui uma *hazard unit* para resolver os problemas inerentes a uma implementação *pipelined* e um barramento para fazer a comunicação com os seus periféricos.

O desenvolvimento desta dissertação é feito em paralelo com uma outra, onde foi desenvolvido o compilador que dá suporte ao SoC desenvolvido nesta dissertação. Várias decisões como o ISA foram tomadas em conjunto pelos dois responsáveis das duas dissertações.

Palavras-chave: *Pipeline*; *System-on-Chip*; *Microblaze*; DLX; FPGA; NVIC; Tricore *timers*;

ABSTRACT

Implementing a chip with a wide number of features reduces the number of microcontrollers required for a particular application and thus the project cost is reduced too. The advent of low cost FPGAs in recent years has led to the implementation of systems based on reconfigurable platforms since their features allow rapid prototyping of different implementations facilitating the development of various projects. FPGA's flexibility allows designers to create customizable and specific modules for an application. FPGAs allow the implementation of application-specific SoCs where metrics such as performance, determinism and time-to-market have a key-role in real-time systems. The implementation of a SoC on a FPGA offers a good trade-off between implementation's flexibility and fast time-to-market.

This dissertation presents a SoC developed targeting automotive applications. The SoC features an interrupt controller based on the ARM NVIC, which allows the service of interrupts with a very low latency and an array of timers based on the timers present in the 32-bit Tricore ™ microcontroller. The implementation of these peripherals allows the use of the SoC for automotive applications due to the determinism that it offers, as well as its performance and priority space unification capability.

The SoC's processor is based on the *Microblaze* which follows a RISC architecture, like the DLX processor, a processor widely used for teaching over the years. The processor implements a five pipeline stages datapath, in order to increase the number of instructions executed in a unit of time, a hazard unit to solve the problems inherent to a pipelined implementation, and a bus to communicate with the peripherals.

The development of this work was done in parallel with another, where it was developed the compiler that supports the SoC developed in this dissertation. Several decisions as the ISA were taken together on both dissertations.

Keywords: *Pipeline*; *System-on-Chip*; *Microblaze*; DLX; FPGA; NVIC; Tricore *timers*;

ÍNDICE GERAL

ABREVIATURAS E SIGLAS

ADC – Analog-to-Digital Converter

ALU – Arithmetic Logic Unit

ASIC – Application Specific Integrated Circuit

ASSP – Application Specific Standard Product

AXI4 – Advanced eXtensible Interface 4

BRAM – Block RAM

CLB – Configurable Logic Block

CPU – Central Processing Unit

DAC – Digital-to-Analog Converter

DSP – Digital Signal Processor

ECU – Electronic Control Unit

EEC – Error Correction Code

FIFO – First In, First Out

FPGA – Field Programmable Gate Array

FPU – Float Point Unit

FSL – Fast Simplex Link

GPU – Graphics Processing Unit

HDL – *Hardware* Description Language

I^2C – Inter-Integrated Circuit

IC – Integrated Circuit

IP – Intellectual Property

ISA – Instruction Set Architecture

LMB – Local Memory Bus

LTE – Long Term Evolution

MIPS – Microprocessor without Interlocked *Pipeline* Stages

MMU – Memory Management Unit

MPEG – Moving Picture Experts Group

OPB – On-chip Peripheral Bus

PC – Program Counter

PLB – Processor Local Bus

RISC – Reduced Instruction Set Computer

RTOS – Real Time Operating System

SCMP – Single Chip Multiple Processor

SoC – System on Chip

SoI – Silicon on Insulator

TLB – Translation Lookaside buffer

UART – Universal Asynchronous Receiver Transmitter

VGA – Video Graphics Array

VHDL – Very High Speed Integrated Circuit HDL

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Capítulo 1

INTRODUÇÃO

Neste primeiro capítulo é apresentada uma breve contextualização relativamente ao tema da dissertação. De seguida são apresentados os objetivos da dissertação, terminando com uma descrição da organização da mesma.

1.1. Contextualização

Atualmente, procuram-se processadores que tenham um desempenho o mais alto possível e um tamanho e consumo reduzidos. Um processador que executa uma instrução de cada vez faz um uso ineficiente dos recursos disponíveis, o que faz com que tenha um baixo desempenho. Para contrariar este baixo desempenho apareceram várias técnicas ao longo dos anos, como por exemplo, *pipelining*, *out-of-order execution* e *superscalar*.

Com os SoCs é possível criar IC que possuem um poder de processamento e memórias maiores do que um computador *desktop* de 10 anos atrás. Com esta tecnologia é possível possuir um computador com todo o tipo de funcionalidades na nossa mão, nomeadamente através de *smartphones* e *tablets* [1].

Nesta dissertação trabalhou-se com FPGA, o que é uma mais-valia, pois hoje em dia, *designers* industriais são incentivados a lançarem os produtos no mercado o mais rápido possível para maximizar o rendimento e o *time-to-market* [2]. Para melhorar o *time-to-market* os *designers* industriais procuram uma solução rápida, flexível, fiável e fácil de projetar. As FPGAs, devido às suas características mencionadas anteriormente, fizeram com que fossem a opção mais conveniente para qualquer tipo de *design* [3]. Em adição, podem ser facilmente reprogramadas, seja para adicionar funcionalidades ou corrigir *bugs*, sendo também possíveis novas lógicas programáveis que proporcionam velocidades maiores em portos I/O mais rápidos e a preços menores, permitindo que as FPGAs sejam usadas em aplicações embebidas que antes eram abordadas por *Application Specific Integrated Circuit* (ASIC) ou *Application Specific Standard Product* (ASSP) [2]. O aumento do uso de FPGAs deve-se também à quantidade disponível de IPs que podem ser facilmente programados, os quais incluem funcionalidades amplamente usadas em aplicações industriais e o fato de os *designers* conseguirem criar soluções sem os custos NRE ou os atrasos de fabricação e assemblagem geralmente associados aos ASICs [2].

1.2. Objetivos

Nesta dissertação o objetivo principal é implementar um SoC com um processador baseado no *softcore Microblaze* e orientado para o domínio automóvel. Este objetivo pode ser divido em várias partes:

- Desenvolvimento de um SoC com um processador baseado no *Microblaze*: Este objetivo passa por desenvolver um SoC com um processador *pipelined* para tornar o seu processamento mais rápido;
- Implementar uma *hazard unit*: Num *pipeline* existem contrapartidas, tais como os *hazards* (*hazards* de dados, *hazards* estruturais e *hazards* de controlo). A *hazard unit* tem como função resolver esses *hazards* para que o programa execute corretamente;
- Inserção de periféricos de suporte à aplicação automóvel, sendo estes o controlador de interrupção programável e *arrays* de temporizadores;

Todo este projeto foi desenvolvido em paralelo com um outro projeto, um compilador. Decisões como o ISA, os tipos de instruções, entre outros, foram decididos em conjunto com o responsável pelo compilador. Todos os testes feitos foram validados tanto pelo SoC como pelo compilador ao longo do desenvolvimento dos mesmos.

1.3. Organização da Dissertação

A presente dissertação encontra-se dividida em sete capítulos, nomeadamente: introdução; estado de arte; ambiente de desenvolvimento; especificações do sistema; implementação; testes e resultados; conclusão. Seguidamente é apresentado um pequeno resumo sobre o conteúdo de cada um dos capítulos referidos.

No primeiro capítulo é apresentada uma breve contextualização seguida dos principais objetivos desta dissertação e da estruturação da organização da mesma.

O segundo capítulo mostra o estado das tecnologias atualmente utilizadas no desenvolvimento desta dissertação. Começa-se com uma explicação sobre o que é um SoC, seguido da apresentação de vários SoC existentes. Também é dada uma breve explicação sobre os processadores DLX e *Microblaze*. No fim deste capítulo são identificadas possíveis FPGAs que podem ser usadas para o desenvolvimento do SoC proposto neste trabalho.

No terceiro capítulo encontra-se o ambiente de desenvolvimento. Neste capítulo aborda-se qual a placa de desenvolvimento usada bem como o *software*.

No quarto capítulo são abordadas todas as especificações do sistema a implementar. O capítulo começa com a justificação da escolha do *softcore* e quais os requisitos para que este

SoC esteja preparado para a indústria automóvel. Há uma divisão do sistema em vários módulos neste capítulo para ser possível identificar as várias tarefas que foram realizadas, estando esta divisão presente nos próximos capítulos.

No quinto capítulo é apresentada a implementação do sistema, ou seja, são apresentados todos os passos efetuados no processo de desenvolvimento do SoC. Este capítulo descreve desde a criação de um simples processador capaz de executar apenas operações aritméticas e logicas até um processador *pipelined* com *hazard unit* e os seus periféricos implementados.

No sexto capítulo são apresentados os testes realizados e os resultados experimentais obtidos. São feitos testes com o objetivo de determinar se as instruções executam corretamente ao longo do *pipeline*, se a *hazard unit* é capaz de identificar e resolver os *hazards* existentes e se os periféricos funcionam como é suposto.

O documento termina com uma conclusão no sexto capítulo. Para além das principais conclusões são feitas algumas sugestões de trabalho futuro, visando melhorar/aumentar as funcionalidades do trabalho desenvolvido.

Capítulo 2

ESTADO DE ARTE

Este capítulo começa com uma explicação sobre o que é um SoC e uma descrição dos processadores DLX e o *Microblaze*. De seguida é dada uma breve descrição de vários SoC existentes baseados no *Microblaze*. É descrita a forma geral de funcionamento de um processador *pipelined*, o que é uma FPGA e quais as famílias existentes. No final deste capítulo é explicado o processo de *desgin flow* de uma FPGA e são apresentados alguns periféricos que podem integrar um SoC.

2.1. System on Chip

Ao longo do tempo tem sido possível aumentar o número de elementos lógico que podem ser implementadas num único chip. Este aumento permitiu a implementação de várias funcionalidades, como memórias, CPU, DSP, MPEG *decoder*/*encoder* e muito mais num único chip [4]. Estes, chamados de SoC, podem ser considerados como um computador completo, visto que, além de um CPU, podem conter uma GPU, memórias, controlador USB, circuitos de *power-management* e comunicação *wireless* (Wifi, 4G LTE, etc.) num único *chip*. Consequentemente poderão mesmo correr versões *desktop* de Windows e Linux além de trazerem várias vantagens em relação aos sistemas que substituem tais como:

- Tamanho reduzido;
- Menor consumo de energia;
- Menor custo;
- Maior fiabilidade.

Um dos problemas de um SoC é a sua falta de flexibilidade, enquanto com um computador pessoal é possível trocar ou adicionar componentes de modo a melhorá-lo o que é impossível num SoC.

Existem muitas razões para a diminuição do número de *Electronic Control Unit* (ECU) num automóvel, mas existem diversos obstáculos para tal. Para responder a este problema, devido a mudanças significantes na arquitetura, na diminuição de dissipação de energia entre outros, são usados os SoCs [5]. Estes estão a evoluir em paralelo com outras tecnologias, nomeadamente o *Silicon on Insulator* (SoI) [6], o qual pode fornecer uma maior velocidade de sinal de relógio e reduzir a energia consumida.

Os SoCs são cada vez mais utilizados na indústria automóvel para as mais diversas aplicações. Existem SoCs criados com o propósito de fornecer soluções a problemas da indústria automóvel, tal como o EyeQ desenvolvido pela Mobileye [7]. Um outro exemplo do uso de SoCs na indústria automóvel verifica-se nos *airbags*, onde existe um aumento do seu número presente num automóvel. Estes necessitam de cumprir regras de segurança mais estritas, mas também é necessário arranjar uma maneira de reduzir os seus custos [8], o que é possível com SoCs.

2.1. DLX

O DLX [9], igualmente conhecido como Deluxe, é um processador *Reduced Instruction Set Computer* (RISC) criado por John L. Hennessy e David A. Patterson. Eles foram os principais criadores do Stanford MIPS e dos Berkeley RISC *designs*.

O DLX é uma versão melhorada de um CPU MIPS, tendo uma arquitetura *load/store* de 32 bits. Este processador foi criado para o ensino, sendo bastante utilizado por várias universidades, principalmente nas unidades curriculares ligadas à arquitetura de computadores.

Existem implementações deste processador, entre elas uma implementação de uma versão do DLX assíncrona [10] e uma implementação que possui o barramento *Wishbone* [11].

Tabela 1 – DLX

DLX	
*Design*er	John L. Hennessy e David A. Patterson
Architecture	32-bit
Version	1.0
Design	RISC
Type	*Register-Register* & *Load-Store*
Encoding	*Fixed*
Branching	*Condition register*
Endianness	*Bi-endian*
Open	*Yes*
General purpose registers	31 (R0 = 0)
Floating point	32

2.2. *Microblaze*

O *Microblaze* [12] é um *soft processor core*, projetado e otimizado para a implementação nas FPGA desenvolvidas pela Xilinx. O seu ISA segue um *design* RISC [13], que é uma linha de arquitetura de processadores. Recorre a um pequeno conjunto de instruções, altamente otimizado e que não tem micro-programação, sendo executadas diretamente pelo *hardware*.

O acesso à memória é realizado através de uma arquitetura de *load*/*store*, onde a memória é acedida através de instruções específicas. Por conseguinte, o *Microblaze* consegue manter um *single-cycle throughput* na maioria das circunstâncias.

Apresenta um sistema de interconexão versátil para suportar uma variabilidade de aplicações. O barramento primário I/O, embutido no *Microblaze*, é o barramento PLB CoreConnect, que se caracteriza por ser um sistema de transação de memória mapeada, onde há uma relação de comunicação de *master*/*slave*. Para o acesso às memórias locais presentes na FPGA (BRAM) utiliza-se o barramento dedicado LMB, que reduz a carga dos restantes barramentos. A comunicação com os coprocessadores é feita através de uma conexão dedicada do tipo FIFO a que se dá o nome de FSL. A interface dos coprocessadores pode acelerar os algoritmos de computação intensiva, transferindo parte ou a totalidade da computação para um módulo de *hardware* especificamente projetado para a função a desempenhar. Muitos aspetos do *Microblaze* podem ser configurados pelo programador, tais como:

- Tamanho da *cache*;
- Número de estágios do *pipeline* (processo usado em processadores RISC, como é o caso do *Microblaze*);
- Periféricos integrados;
- Unidade de gestão de memória;
- Barramento de interface.

Esta personalização que está inerente ao microprocessador possibilita ao programador fazer alterações para *designs* apropriados para um conjunto específico de aplicações de *hardware* e *software*. Com a MMU o microprocessador é capaz de incorporar sistemas operativos que requerem memória virtual baseada em *hardware*. Associado ao *Microblaze* está também o recurso opcional à *cache* de instrução ou à *cache* de dados, que tendem a melhorar o desempenho do microprocessador, quando o código de execução e dados se encontram fora da faixa de endereços de memória LMB.

Concluindo, numa visão geral, o *Microblaze* possui as seguintes características:

- Arquitetura 32-bit RISC Harvard
- 32x32-bit *general purpose registers*
- PLB *interface*
- MMU
- *cache* de instruções e de dados
- *Pipeline depth* configurável
- FPU

Tabela 2 – *Microblaze*

Microblaze	
*Design*er	Xilinx
Architecture	32-bit
Version	8.50.b
Design	RISC
Encoding	*Fixed*
Endianness	*Big*/*Little*
Open	*No*
Registers	32 x 32 bits

2.2.1. OpenFIRE

O OpenFIRE [14] é um processador *binary-compatible* com o Xilinx *Microblaze* 32-bit RISC *soft processor*, disponibilizado sob licença MIT *Open Source*. Esta característica permite que um ficheiro binário compilado para um sistema embebido com *Microblaze*, irá correr num OpenFIRE, que seja colocado no mesmo sistema embebido em vez do *Microblaze*.

O OpenFIRE foi criado por Stephen Craven, um estudante de doutoramento na Virgínia Tech, e o último melhoramento foi feito por Alex Marschner, um estudante de mestrado da mesma universidade. Foi concebido para ser totalmente configurável, suportar um *datapath* de 16-bit e, ao contrário do *Microblaze* que precisa no mínimo de quatro BRAMs para permitir *byte-steering*, o OpenFIRE requere apenas uma BRAM.

O OpenFIRE é pequeno e possui um *feature-set* ajustável, o que o tornou um bom candidato para pesquisa *Single Chip Multiple Processor* (SCMP). Para este tipo de pesquisa foi necessário

tornar o OpenFIRE independente, pois antes dependia de um processador *Microblaze* como *host* para aceder à memória externa ou periféricos [15]. A solução passou pela adição de um bus OPB. Foi levada em consideração a área, por isso, apenas o mínimo de interfaces I/O foram implementados, em especial portos FSL para comunicações ponto-a-ponto, já que têm baixa latência na comunicação e um *throughtput* alto.

2.2.2. SecretBlaze

O SecretBlaze [16][17][18] é um processador com uma arquitetura baseada no DLX de John Hennessy e David Patterson, desenvolvido usando a linguagem VHDL. É compatível com o ISA do *Microblaze*, possui um *datapath de* 32-bit RISC que tira proveito do paralelismo existente entre as instruções com um *pipeline* de cinco estágios. Possui um global *hazard controller* que fornece sinais de *stall* e *flush* sincronizados para os estágios apropriados do *pipeline*, os quais servem para lidar com *hazards*.

A combinação dos métodos distribuídos e centralizados na implementação da unidade de controlo foi levada a cabo para tornar a qualidade do *design* e eficiência do *datapath* mais equilibrado.

Apesar de não estar otimizado para FPGAs, os resultados de implementações confirmam que o SecretBlaze tem um desempenho bastante próximo do *Microblaze*, oferecendo assim uma boa alternativa *open-source*.

2.2.3. LatticeMico32

O LatticeMico32 [18][19] é um processador RISC *softcore* de 32-bit com uma arquitetura Harvard. Este *softcore*, devido a um *datapath* de 32-bit, instruções de 32-bit e 32 registos *general purpose*, faz com que o desempenho por ele oferecido seja adequado para uma grande variedade de mercados. Consegue lidar até 32 interrupções externas, suporta instruções opcionais, *caches* de dados e vários periféricos podem ser integrados, desde que sejam compatíveis com Wishbone, para acelerar o desenvolvimento de sistemas de microprocessadores. O sistema LatticeMico é baseado nas ferramentas Eclipse C/C++ *Development Tools Environment*.

2.2.4. LEON4

O LEON4 [18][20] é um processador de 32-bit baseado na arquitetura do SPARC V8, altamente configurável e particularmente adequado para SoC. Segue uma arquitetura Harvard, usa o AMBA *Advanced High-performance Bus* e é concebido e mantido por Gaisler Research.

Implementa um *pipeline* de sete estágios, *caches* de dados e instruções separadas e o número de registos é configurável dentro do limite padrão da SPARC. Uma implementação por defeito deste processador tem oito registos globais e oito conjuntos de registos, em que cada conjunto consiste em 24 registos. Sendo assim, um programa irá ter a cada instante 32 registos disponíveis, havendo um total de 200 registos. Esta versão, em relação à anterior, adicionou uma *static branch prediciton*, uma *cache* L2 configurável e o AMBA AHB pode ter uma *path* de 64-bit ou 128-bit. Possui também uma interface de *debug* que permite o *debugging* de *hardware* não-intrusivo e proporciona acesso a todos os registos e memória

2.2.5. OpenRISC1200

A partir do número presente no nome do processador, neste caso 1200, podemos saber várias características dele. O primeiro dígito identifica a que família pertence, neste caso sendo a família OpenRISC 1000, o segundo dígito identifica quais as características e como foram implementadas e os últimos dois dígitos definem como é feita a sua configuração para ser implementado [18].

O OpenRISC1200 [21] é um processador RISC de 32-bit com uma arquitetura Harvard. Possui um *pipeline* de cinco estágios, MMU, capacidades básicas de DSP, instruções de 32-bit e pode operar com dados de 32-bit ou 64-bit. Por defeito, tem-se tanto a *cache* de dados como a *cache* de instruções 1-way *direct-mapped* 8KB, tendo cada uma linha de 16-byte. A nível da MMU, ambos os *Translation Lookaside buffer* (TLB) de dados e instruções são constituídos por 1-way *direct-mapped* 64-entry *hash based*.

Este processador consegue competir com os mais recentes processadores *scalar* 32-bit RISC e consegue correr muitos dos sistemas operativos modernos.

2.2.6. aeMB

O aeMB [18][22][23] é uma implementação compatível com o *software Microblaze* core EDK3.2. É compatível a nível de instruções para a maior parte dos comandos de *software*, mas não substitui o *Microblaze*. Usa um barramento Wishbone tanto para memória de dados como de instruções e o CPU é capaz de mover e manipular dados para e da memória. Apesar de não possuir periféricos ou controladores de interrupções consegue dar suporte a interrupções externas e caso sejam implementados, tanto eles como os respetivos registos podem ser mapeados para a memória de dados.

Este processador segue uma arquitetura Harvard com barramentos 32-bit separados para dados e instruções e possui um *pipeline* de três estágios que é capaz de executar uma instrução por ciclo de relógio. Suporta também um multiplicador e *barrel shifter* em *hardware*.

2.3. *Pipelining*

A técnica utilizada para o desenvolvimento de um processador *pipelined* é chamada de *pipelining* [24]. Neste tipo de implementação, múltiplas instruções são sobrepostas em execução, possibilitando tirar-se vantagem do paralelismo que existe entre as ações necessárias para executar uma instrução [9]. Na Figura 1 é possível ver uma execução normal de um processador *pipelined*, onde a cada ciclo de relógio uma nova instrução é iniciada.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figura 1 – *RISC pipeline* [9]

Um *pipeline* pode ser associado a uma linha de produção fabril, onde cada passo para a construção de um produto é realizado em paralelo aos restantes passos de outros produtos. O mesmo acontece com o *pipeline*, cada estágio decorre em paralelo com os restantes. Assim, como numa linha de produção fabril, cada estágio de um *pipeline* completa uma parte da instrução.

O primeiro estágio está conectado ao segundo, o segundo ao terceiro, o terceiro ao quarto e o N-estágio está conectado ao N+1-estágio, em que N é o estágio atual e N+1 é o estágio seguinte. Como todos estão ligados, só é possível avançar quando todos os estágios já completaram o seu trabalho. O tempo necessário para mover uma instrução de um estágio para outro é chamado *processor cycle*.

Por conseguinte, pode-se deduzir que a duração de um *processor cycle* vai ser dada pela duração do estágio mais moroso. Geralmente, a duração deste *processor cycle* é igual a um ciclo de relógio ou, no máximo, dois ciclos de relógio. O mais importante num *pipeline* é o *throughput* que é, no caso do *pipeline*, a frequência com que uma instrução é completada, tendo

como seu equivalente, numa linha de produção fabril, o número de produtos fabricados por unidade de tempo.

Assumindo condições ideais pode-se dizer que o ganho em termos de velocidade que se obtém num processador com *pipeline* é dada por [9]:

$$\frac{Tempo\ por\ instrução\ num\ processador\ sem\ pipeline}{Número\ de\ estágios}$$

Assim sendo, o aumento da velocidade é proporcional ao número de estágios, ou seja, é N vezes mais rápido, onde N é o número de estágios. Tudo dito anteriormente apenas é verdade num mundo ideal, pois os estágios nunca demoram o mesmo tempo a executar e o uso de *pipelining* envolve *overhead* como *pipeline register delay* devido à adição de registos, *hazards* e aumento de lógica entre os estágios e *clock skew*, pois o sinal de relógio precisa de ser encaminhado para mais sítios. Consequentemente, a velocidade do processador não aumentará N vezes, mas não andará muito longe desse valor.

Um *pipeline*, tal como foi referido, pode ter N-estágios, mas o primeiro estágio é igual para todos, sendo este o *Instruction Fetch*. O último estágio geralmente é o *Write back*. O número de estágios pode variar, desde um *pipeline* de 2-estágios até um de 7, 10, 20 ou até mesmo 31 como nos "Pentium 4 cores" "Prescott", "Cedar Mill" e nos derivados dos "Pentium D". Um dos modelos de implementação mais conhecido é o "*classic RISC pipeline*" que tem 5-estágios. Estes estágios são o *Instruction Fetch*, *Instruction Decode*/*Register Fetch*, *Execute*, *Memory access* e *Write back*.

2.4. Field Programmable Gate Array

FPGA são dispositivos semicondutores programáveis, que se baseiam numa matriz de *Configurable Logic Block* (CLB), conectados por interconexões também programáveis [25]. OS CLBs podem ser configurados para realizar funções complexas ou simples, como OR e NAND. Na maioria das FPGAs os CLBs podem incluir memórias, que podem ser simples *flip-flops* ou um bloco de memória mais completo [26].

As FPGAs são utilizadas para implementar funcionalidades de lógica digital personalizada (*Glue Logic*) ou um SoC. Comparado com os ASICs, onde o dispositivo é feito especialmente para um único *design*, as FPGAs podem ser programadas várias vezes [25]. São feitas para poderem ser configuradas pelo utilizador, podendo isso acontecer após o seu fabrico, daí o nome

field-programmable. Existe também a possibilidade de fazer essa reprogramação remotamente, eliminando custos.

Existem FPGAs com funções analógicas em adição às funções digitais. A mais comum é o *slew rate* e *drive strength* em cada um dos pinos de saída. Sendo assim, o programador consegue definir taxas lentas em pinos com pouca carga e taxas mais fortes e rápidas em pinos com uma carga grande. Outra função analógica são os comparadores diferenciais nos pinos de entrada, que servem para estarem ligados a canais de sinais diferenciais. Algumas *mixed signal* FPGAs têm conversores *Analog-to-Digital Converter* (ADC) e *Digital-to-Analog Converter* (DAC), com blocos de acondicionamento de sinais analógicos, o que os permite funcionar como SoCs [27].

2.4.1. Virtex family

As FGPA Virtex [28][29][30] tem integrada características como FIFO, logica EEC, blocos DSP, Ethernet MAC, controladores PCI-Express e *transceivers* de alta velocidade. Inclui também *hardware* para funções que são usadas frequentemente, como multiplicadores, memorias, *transceivers* série e *cores* de microprocessador. Todas estas características são uteis para aplicações, como equipamentos de infraestrutura com ou sem fios, equipamento médico e sistemas de defesa.

Com o aparecimento da Virtex-5, veio também a alteração de LUTs de quatro entradas para LUTs de seis entradas. Com o aumento da complexidade exigida pelos *designs* de SoC, o número de LUTs de quatro entradas requeridas é tão grande que se tornou um problema a nível de desempenho e roteamento. Com as LUTs de seis entradas, passou a haver uma solução melhor para funções combinacionais cada vez mais complexas, mas havendo uma redução no número total de LUTs por dispositivo.

2.4.2. Kintex family

A família de FPGA Kintex-7 [31][32] são, segundo a Xilinx, capazes de ter o mesmo desempenho que a família da Virtex-6 a um preço inferior a metade da mesma e consumir menos 50% de energia. A família Kintex inclui conexão série de alto desempenho a 12.5 Gbit/s ou baixo consumo a 6.5 Gbit/s, memórias, desempenho requerido para aplicações em equipamento de comunicações e proporciona um equilíbrio de desempenho de processamento de sinal, consumo de energia e custo para suportar a implementação de redes sem fios LTE. Possui também flexibilidade e poupança de tempo a nível de programação e *Targeted Design*

Platforms, para tempos e custos de desenvolvimento menores e arquitetura unificada com AMBA AXI4 para IP *plug-and-play* que facilitam a portabilidade e reutilização.

2.4.3. Artix family

A família Artix-7 [33][34] é baseada na arquitetura unificada da série Virtex, conseguindo obter um consumo 50% menor em relação às gerações anteriores. Segundo a Xilinx, as FPGA Artix-7 possuem uma arquitetura escalável otimizada para uma migração rápida de *design*, uma *footprint* pequena e conseguem proporcionar o desempenho necessário para mercados de grande volume, os quais tinham como resposta ASSPs, ASICs e FPGAs de baixo custo. Foram criadas para equipamentos de ultrassom portáveis alimentados a baterias, controlo de lentes de câmaras digitais e equipamentos militares de aviação e comunicação.

2.4.4. Spartan family

A família Spartan proporciona aos *design*ers metodologias mais simples para criar SoCs baseados em FPGA. A série Spartan é a resposta para aplicações com baixo consumo, grande dependência do custo e grandes volumes, como por exemplo *displays, set-top boxes* entre outros [35][36]. Permite o uso de *open standards*, metodologias de *design* comuns, fazendo com que seja possível reduzir o tempo utilizado para desenvolver a infraestrutura de uma aplicação, havendo assim mais tempo para desenvolver características diferenciadoras.

2.5. Periféricos

Existem diversos periféricos que podem ser implementados num SoC, tais como VGA, I²C, *timers*, ou seja, qualquer sistema de hardware digital. Neste sistema, uma vez que é dedicado para aplicações automóvel, os periféricos principais que serão implementados serão um controlador de interrupções programável e um *array* de *timers*. Além destes pode ser implementado um porto I/O, uma UART, entre outros.

2.5.1. Porto *Input*/*Output*

Este periférico serve para fazer o interface com o exterior, seja um humano ou um outro sistema. *Inputs* são sinais recebidos pelo sistema e *outputs* são sinais que são enviados do sistema para o exterior.

2.5.2. Controlador de interrupções programável

A família de processadores ARM Cortex™-M [37] são possuem uma boa razão entre energia-eficiência, sendo também fáceis de usar o que ajuda os utilizadores a ir de encontro às

necessidades do amanhã. Estas necessidades incluem o aumento de recursos a um preço mais baixo, uma maior conectividade, código mais fácil de reutilizar e uma utilização de energia eficiente. Nesta família existe um controlador de interrupções bastante vantajoso para indústria automóvel – NVIC – que devido ao fato de se encontrar integrado no processador (*tightly-coupled*) possui uma baixa latência durante a gestão das interrupções. Sendo programável possibilita a unificação do espaço de prioridades através de um modelo de programação com tarefas regulares da aplicação como ISRs.

2.5.3. *Timers*

Os microcontroladores TriCore™ [38] são utilizados em sistemas de gestão de motores a gasolina e diesel, atendendo às crescentes demandas do mercado para emissões mais baixas e níveis de eficiência mais elevados. Produtos que usam este microcontrolador também oferecem a versatilidade necessária para o setor industrial, destacando-se em aplicações de controlo de motores otimizados e processamento de sinais.

2.5.3.1. *System timer*

O *System Timer* (SYSTICK) é um *timer* usado, normalmente, para providenciar um relógio para auxiliar na comutação de contexto de um OS/RTOS. A sua função é gerar uma interrupção de *x* tempo em *x* tempo, em que *x* é um valor definido pelo programador, e executar o código que geralmente corresponde ao escalonador de tarefas.

2.5.4. Comunicação série

A comunicação série é um meio de comunicação entre dispositivos [39]. Ao contrário da comunicação paralela, em que os bits são enviados ao mesmo tempo, na comunicação sequencial são enviados um de cada vez através de um canal de comunicação. Entre a comunicação paralela e sequencial, a sequencial permite maiores distâncias, pois segundo a norma IEEE 488 as especificações para comunicação paralela especificam que a cablagem entre os equipamentos não pode ter mais de 20 metros e não pode haver uma distância superior a dois metros entre os aparelhos, enquanto a sequencial pode ir até aos 1200 metros [40].

2.1. *Design Flow*

2.1.1. FPGA

O *design flow* são os passos requeridos entre as ideias iniciais e o *hardware* final [41]. Após todos os requisitos serem especificados, tem de se passar por uma fase de *design*, em que a

mesma não tem qualquer tipo de ajuda das ferramentas dos vendedores de FPGA. Cada vendedor de FPGA tem o seu próprio *design flow* e ferramentas para o desenvolvimento, mas são todas semelhantes. Pode-se ver na Figura 2 o *design flow* seguido pelo ISE da Xilinx.

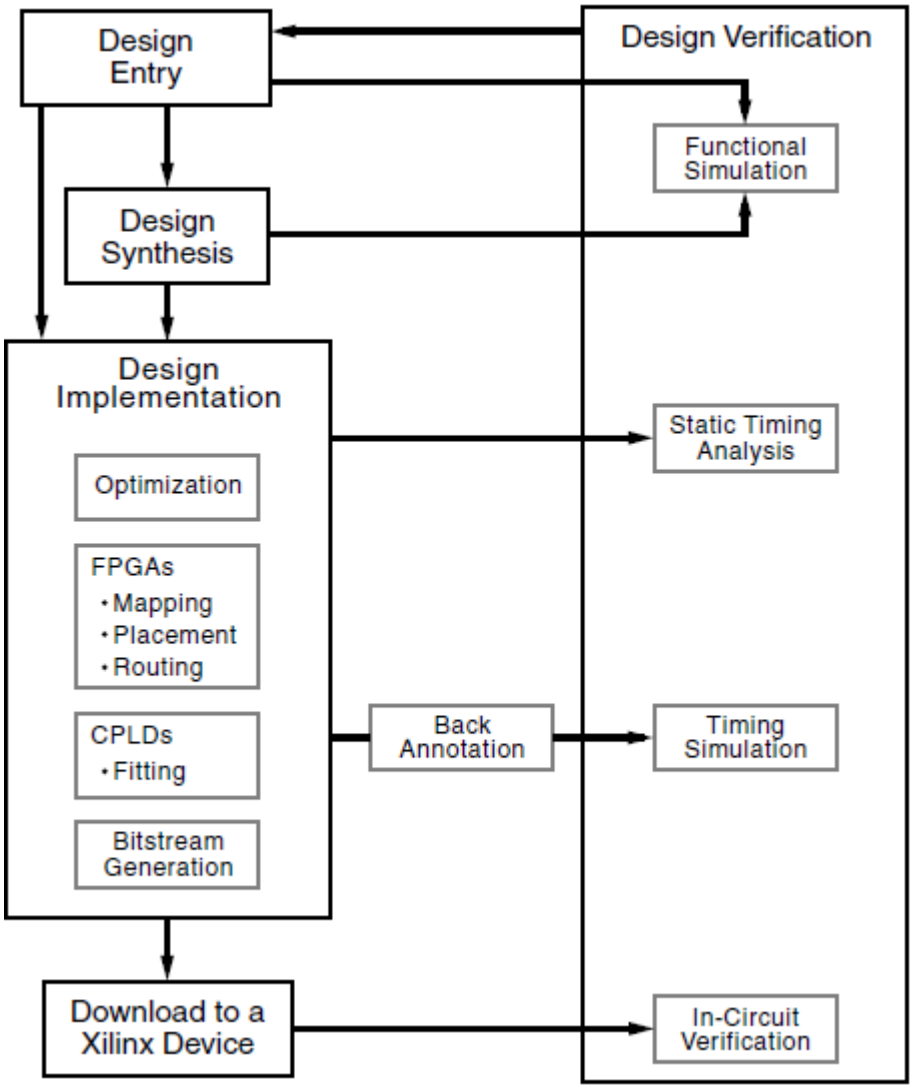


Figura 2 – Design flow Xilinx [42]

As fases mais utilizadas no *design flow* são *design*, síntese, *place* & *route* e geração do *bitstream* [41]. A fase de *design* consiste em transformar um *design* numa representação lógica do sistema, recorrendo a uma HDL. As HDLs mais comuns são VHDL e Verilog e é nesta fase que é exigido um maior trabalho humano, pois nas fases seguintes passamos a ter o suporte das ferramentas dos fabricantes de FPGA. Após o *design* ser definido passa-se para a síntese. Nesta fase, após escolhida a FPGA e a família a que pertence a mesma, é gerada uma *netlist* que usa as primitivas fornecidas pelo fabricante para satisfazer o comportamento especificado nos

ficheiros HDL. Existem alguns passos adicionais executados pela maioria das ferramentas de síntese, como otimização lógica, entre outras, para melhorar o desempenho temporal do sistema, para deste modo a *netlist* obtida seja uma implementação do HDL o mais eficiente possível. A terceira fase tem o nome de *place* & *route*, podendo se dividir em duas sub-fases, *place* e *route*. A sub-fase *place* consiste em usar a *netlist* criada e escolher a posição no *chip* onde cada primitiva vai ser colocada. A sub-fase *route* tem como tarefa ligar todas as primitivas de forma a satisfazer as restrições de tempo colocadas. A última fase consiste na geração de um ficheiro *bitstream*. É neste ficheiro, usado para programar a *flash* da FPGA alvo, que se encontram as definições para a configuração de cada elemento na FPGA.

Capítulo 3

AMBIENTE DE DESENVOLVIMENTO

Neste capítulo é apresentada a plataforma de desenvolvimento utilizada ao longo desta dissertação bem como todo o seu ciclo de desenvolvimento.

O objetivo desta dissertação é o desenvolvimento de um SoC com periféricos que deem suporte a uma aplicação automóvel. As tarefas mais importantes passam pelo desenvolvimento de um processador *pipelined*, um controlador de interrupções e um *array* de *timers*. Após todos os módulos estarem implementados e validados, deverão estar todos integrados no sistema final.

A escolha da tecnologia FPGA para implementar a dissertação deve-se a vários fatores, dentre os quais os mais importantes o fato de esta oferecer um maior desempenho em relação aos tradicionais DSPs e um curto *time-to-market*, permitindo aos projetistas testar novas ideias e conceitos diretamente no *hardware* sem que para isso tenham de passar pelo longo processo de fabricação e prototipagem que caracteriza a tecnologia ASIC. Também existe uma diminuição no custo de produção e por ser uma tecnologia confiável permite a manutenção a longo prazo. A placa de desenvolvimento utilizada para este projeto foi uma XUPV5-LX110T do distribuidor Xilinx, a qual pode ser observada na Figura 3.

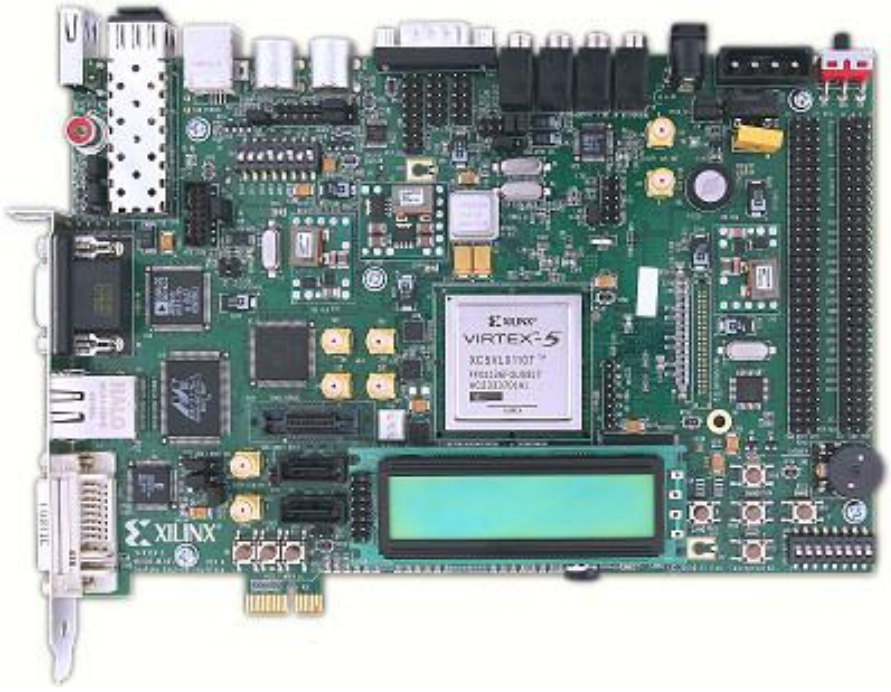


Figura 3 – Virtex 5 XC5VLX110T [43]

3.1. Plataforma de Desenvolvimento

A plataforma de desenvolvimento utilizada é a FPGA de alto desempenho denominada XUPV5-LX110T pertencente à família Virtex-5 e do fabricante Xilinx [44]. Esta placa encontra-se abrangida pela Xilinx *University Program*, podendo ser utilizada em projetos de investigação avançados, cobrindo assim um abrangente domínio de aplicações possíveis.

A placa possui bastantes recursos, tem um amplo conjunto de periféricos que podem ser usados na construção de sistemas variados e complexos [45]. Pode-se configurar e programar a mesma através da *Compact Flash Card* ou então através do cabo de programação USB/JTAG. A nível de memórias, a placa possui uma SDRAM, ZBT SRAM, *compact flash card*, *strataflash* e uma *flash* não volátil. O diagrama de blocos apresentado na Figura 4 apresenta a constituição da plataforma de desenvolvimento.

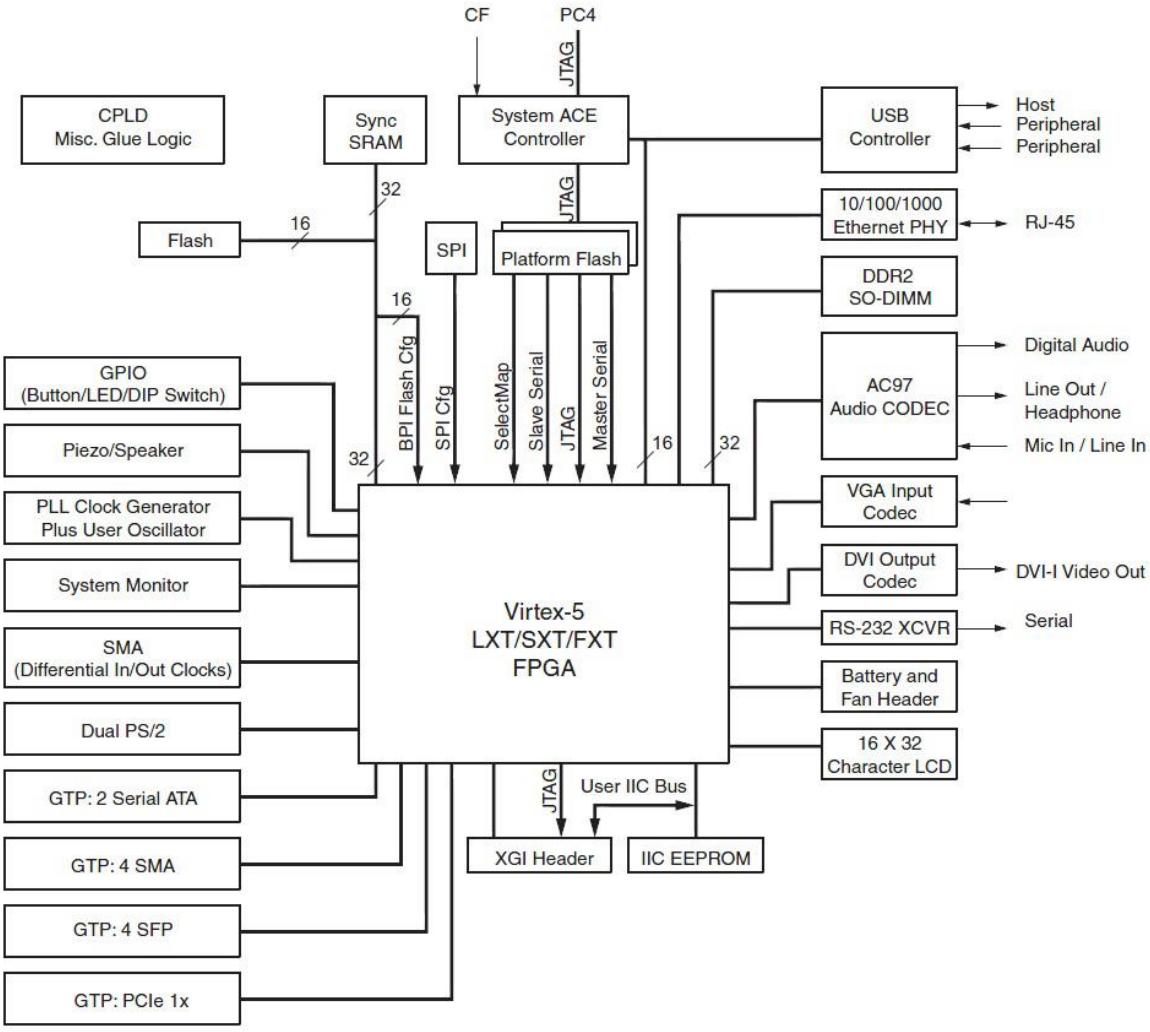


Figura 4 – Diagrama de blocos da XC5VLX110T

Quanto ao dispositivo FPGA XC5VLX110T propriamente dito, este está integrado em um package ff1136 e possui um *speed grade* de -1. A constituição interna do dispositivo referente às quantidades dos determinados recursos básicos está apresentada na Tabela 3.

Tabela 3 – Características XC5VLX110T

Virtex-5 FPGA - XC5VLX110T	
CLBs	
Array I/O (*Row* x *Col*)	160 x 54
Virtex-5 *Slices*	17,280
Max Distributed RAM (Kb)	1,120
DSP48E *Slices*	64
BRAM *Blocks*	
18 Kb	296
36 Kb	148
Max (Kb)	5,328
Clock Management Tile	6
Endpoint Blocks for PCI Express	1
Ethernet MACs	4
Max Rocket IO Transceivers GTP	16
Total I/O Banks	20
Max User IO	680

Cada *slice* possui quatro LUTs e quatro *flip-flops*. *Slices* do tipo DSP48E contém vários multiplicadores, um somador e um acumulador. Finalmente tem-se as BRAM com um tamanho de 36Kb podendo ser usadas como dois blocos independentes de 18Kb.

3.2. Xilinx Embedded Development Kit

As ferramentas que dão suporte à placa de desenvolvimento apresentada anteriormente são denominadas de Xilinx Embedded Development Kit.

De seguida serão apresentas todas as ferramentas utilizadas no desenvolvimento do SoC proposto durante a dissertação. A Figura 5 descreve o ciclo de desenvolvimento de sistemas embebidos usando o EDK. O desenvolvimento de projetos baseados em tecnologia FPGA

compreende a integração de *software*, *hardware* bem como a gestão das respetivas interligações.

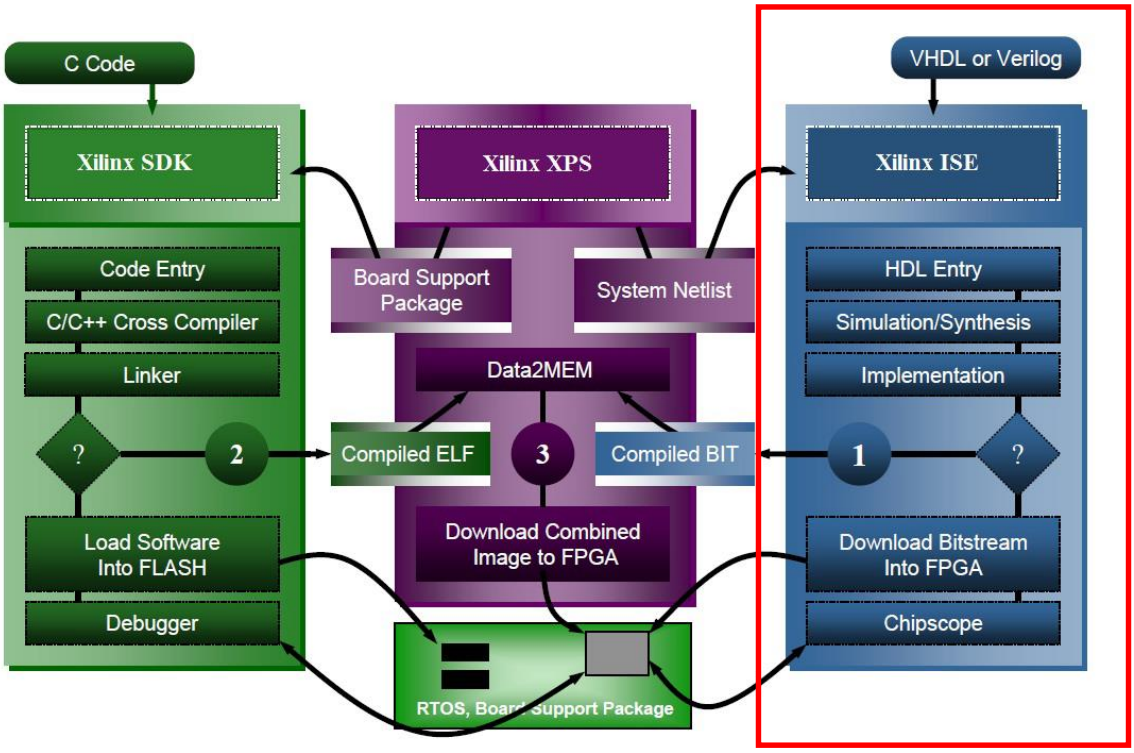


Figura 5 – Ciclo de desenvolvimento [43]

No diagrama destacam-se as três ferramentas fundamentais que constituem o EDK, sendo que para este projeto apenas foi utilizada a ferramenta Xilinx ISE. De seguida é dada uma explicação sobre a ferramenta utilizada:

• ***Integrated Software Environment*** **(ISE)** – este *software* é um IDE gráfico, sendo essencialmente um editor de código que aceita ficheiros em HDL, dentre os mais conhecidos o VHDL e Verilog, sendo este último a linguagem usada neste projeto.

Os projetistas de HDL têm a liberdade de criar IPs, usufruindo das ferramentas fornecidas por este IDE para realizar a síntese, implementação, verificação e depuração.

Os ficheiros constituintes dos IPs numa primeira fase passam pelo processo de síntese onde são convertidos para ficheiros do tipo *netlist*. Na segunda fase, implementação, são convertidos para um formato próprio pronto para configurar o dispositivo FPGA.

Depois da síntese e implementação, é boa prática passar pela etapa de verificação do projeto. Nesta fase os projetistas usufruem de um *software* de simulação para verificar a funcionalidade e os timings do projeto. O simulador interpreta os ficheiros HDL do projeto, para circuito

funcional e apresenta os resultados funcionais em ordem para determinar o correto funcionamento do circuito.

Para terminar, é possível colocar na FPGA o ficheiro final produzido pela segunda fase, chamado de *bitstream*. Também é possível depurar em ambiente *runtime* o projeto IP concebido utilizando a ferramenta de depuração em *hardware*, fornecida pela Xilinx, chamada de *Chipscope Pro Analyser*.

Capítulo 4

ESPECIFICAÇÃO DO SISTEMA

No presente capítulo serão abordadas todas as etapas de análise e *design* realizadas, em que os resultados das mesmas serão usadas como guia para o processo de implementação do SoC. Em primeiro lugar é apresentado um diagrama de blocos com a descrição do sistema completo, seguido das funcionalidades e restrições do sistema. De seguida tem-se como deve funcionar um processador *pipelined*, estando explicado detalhadamente como deve funcionar cada estágio, como funciona a *hazard unit* e o barramento. Por fim, serão apresentados os periféricos, como devem funcionar, como interagem com o resto do sistema e como o programador deve interagir com os mesmos.

4.1. Visão geral do sistema

Na Figura 6 pode-se ver o diagrama de blocos do sistema completo. O processador para comunicar com todos os periféricos utiliza o barramento, o qual se encontra dividido em *encoder* e *decoder*. Os periféricos *timer*, SYSTICK, controlador I/O e DVIC comunicam diretamente com o processador para enviar determinadas informações, tais como o endereço de interrupção enviado pelo DVIC, valor lido nos pinos de entrada encaminhado pelo controlador I/O ou o valor atual de contagem do SYSTICK ou *timer*. As linhas de interrupção dos periféricos *timer*, UART, SYSTICK e controlador I/O estão ligados ao controlador de interrupções DVIC, responsável por gerir a prioridade no atendimento das interrupções dos periféricos.

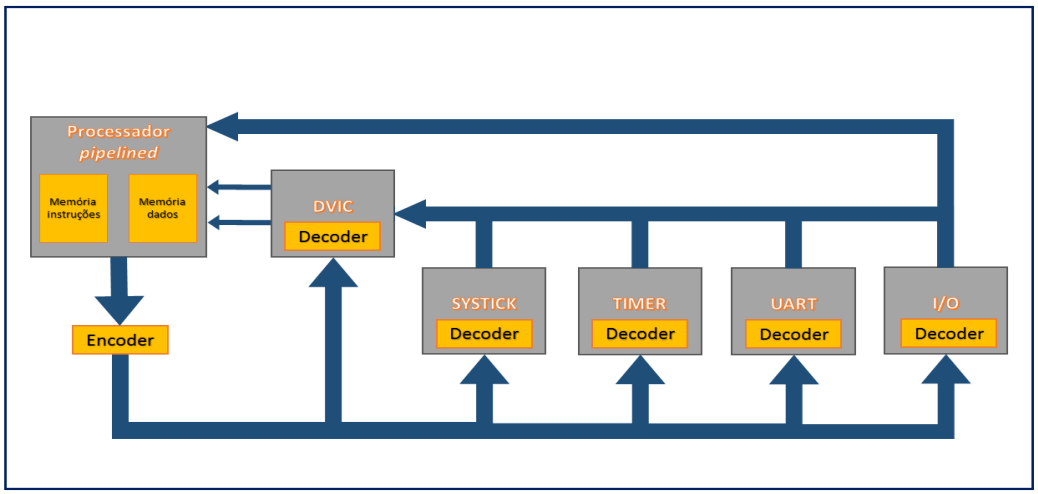


Figura 6 – Diagrama de blocos

Este SoC é dotado de processador RISC com cinco estágios de *pipeline*, uma arquitetura de memória Harvard, ou seja, possui memória de dados e de instruções separadas, possui instruções de 32 bits e 32 registos de 32 bits para uso genérico.

4.2. Funcionalidades e Restrições

Como mencionado no capítulo introdutório, o projeto proposto nesta dissertação visa desenvolver um processador *pipelined* com uma *hazard unit*. O SoC será composto por este processador e um conjunto de periféricos.

O processador escolhido para analisar é baseado no *Microblaze*, uma vez que este se baseia na arquitetura do DLX, que é um processador muito utilizado para o ensino ao longo dos anos.

De modo a que este SoC tenha aplicabilidade na indústria automóvel, é necessário a implementação de alguns periféricos. Estes são um controlador de interrupções e um *array* de *timers*, baseados nos periféricos presentes nos microcontroladores ARM Cortex™-M3 e TriCore™, respetivamente. Estes periféricos, como estão implementados em *hardware*, são bastantes importantes, pois tornam o SoC determinístico e capaz de fazer um uso dos seus recursos e os dos automóveis de uma maneira mais eficiente. O controlador de interrupções é capaz de fornecer a cada ciclo a interrupção com maior prioridade e os *timers* são altamente configuráveis, programáveis e, como supracitado, são implementados em *hardware* o que faz com que o tempo contado por eles sejam exatos.

4.3. *Instruction Set Architecture*

O ISA deste SoC segue uma arquitetura RISC *load-store* de 3 operandos, com 32 registos genéricos e instruções com um tamanho fixo de 32 bits, a mesma que o *Microblaze*.

Na Figura 7 encontram-se os diferentes formatos das instruções. O tipo A e tipo B são para operações aritméticas/lógicas e acessos à memória. Caso seja o primeiro, é realizada uma operação entre o conteúdo em *Ra* e *Rb*/*Imm* sendo o resultado guardado no registo *Rd*. Em operações de acessos à memória os conteúdos em *Ra* e *Rb*/*Imm* são usados para endereçar a memória. Numa instrução *load*, o registo *Rd* irá guardar o valor obtido da memória. Numa instrução *store*, o registo *Rd* contém o valor a guardar na memória.

O tipo C são instruções de salto e o tipo D são instruções que necessitam apenas do *opcode* para as identificar, como por exemplo a instrução de retorno de uma função RET.

Instruction	ENCODING																											
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Type A	Opcode						Rd					Ra					Rb					x									
Type B	Opcode						Rd					Ra					imm														
Type C	Opcode						imm																								
Type D	Opcode						x																								

Figura 7 – *Instructions layout*

No início deste projeto foi necessário definir um ISA para facilitar o desenvolvimento. Como este SoC possui um processador baseado no *Microblaze*, o seu ISA é bastante semelhante, havendo poucas mudanças. Como já foi referido, quase todas as instruções tem o mesmo formato que as instruções do *Microblaze* mas existe uma exceção, a instrução *PERACC*. Esta instrução foi criada para aceder aos periféricos através do barramento implementado neste projeto. O ISA final deste projeto pode ser encontrado na Tabela 10 no Anexo A.

4.4. Processador *pipelined*

O processador terá de conseguir com que a cada ciclo de relógio uma instrução seja iniciada e uma outra concluída. Sendo um processador com *pipeline* de cinco estágios, cada instrução terá uma duração de pelo menos cinco ciclos, à exceção de saltos, pois estas necessitam de ser executadas o mais rápido possível para não desperdiçar ciclos de relógio. Os cinco estágios do *pipeline* tem o nome de *fetch*, *decode*, *execute*, *memory access* e *write back*, havendo sempre um *buffer* a ligar um estágio ao seguinte. Na Figura 8 é apresentado o *datapath* do processador deste projeto.

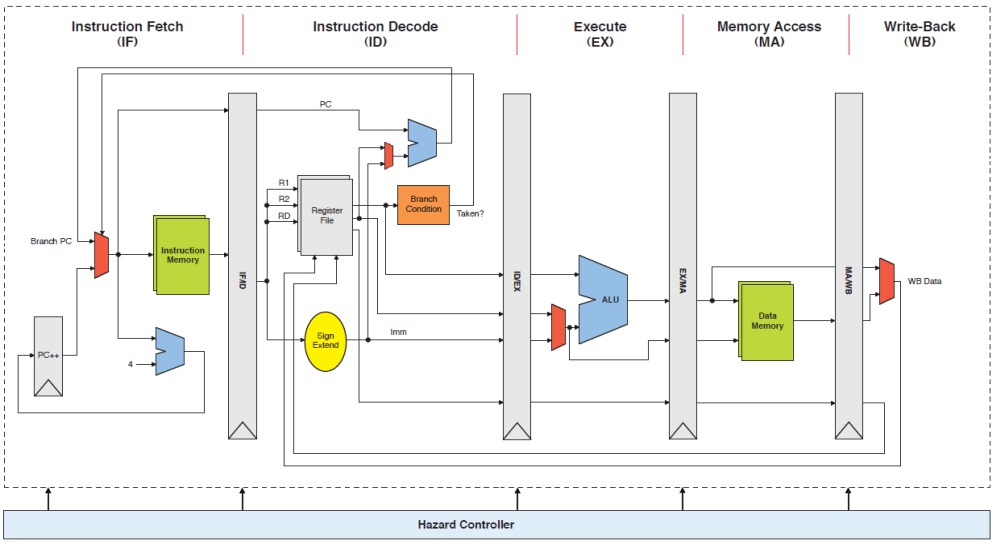


Figura 8 – *Datapath* processador D_Blaze_SoC

O primeiro estágio é leitura de instrução (*Instruction Fetch*). Neste ciclo é feito o *fetch* da instrução da memória de instruções. O *Program Counter* (PC) é o registo responsável por conter o endereço da instrução atual, a qual é guardada num registo para posteriormente ser utilizada em outros estágios. Neste estágio, após o *fetch* da instrução, o *PC* é incrementado para passar a conter o endereço da próxima instrução ou o endereço de salto/interrupção. Atualmente existem algoritmos – *branch prediction* e *branch target prediction* – que ajudam a prever qual o valor que o *PC* deve tomar. Neste projeto não foi utilizado nenhum algoritmo para previsão dos saltos

O segundo estágio é descodificação de instrução (*Instruction Decode*). Neste estágio é realizado o *decoding* da instrução e são feitos os acessos ao *register file*. É no *decoding* da instrução que se identificam os registos que devem ser lidos do *register file*. Caso a instrução seja um salto, o endereço de salto é calculado ao mesmo tempo que se lê do *register file*. O valor do *PC* só é alterado para este novo endereço se a condição de salto for verdadeira. O valor *immediate* é *sign-extended* e guardado num registo temporário chamado *Imm*, para ser usado posteriormente.

Ainda neste estágio é dada a resposta a instruções de controlo. As instruções de controlo podem ser classificadas como saltos condicionais ou saltos absolutos. Para os saltos absolutos basta fornecer o valor que deve ser adicionado ao *PC* ou o valor que o *PC* deve tomar. Para saltos condicionais existe a necessidade de a instrução de salto condicional ser precedida por uma instrução de comparação. Dependendo do resultado desta comparação, a instrução de salto irá se realizar ou não. Caso se realize, o valor do registo *Rb* ou do campo *Imm* é adicionado ao *PC* para se obter o endereço de salto.

O terceiro estágio é execução (*Execute*). Neste estágio encontra-se a *Arithmetic Logic Unit* (ALU), um *barrel shifter* e uma unidade de comparações. O *barrel shifter*, tal como o nome indica, é responsável por fazer *shifts* e rotações. A unidade de comparações faz as comparações necessárias para verificar se os saltos condicionais se concretizam ou não.

A ALU é responsável por executar operações booleanas – AND, OR, NOT, NAND, NOR, XOR e XNOR – e operações de adição e subtração com inteiros. Pode realizar uma de quatro funções possíveis sobre os operandos preparados no estágio anterior, dependendo do tipo de instrução a ser executada. Estes tipos podem ser:

- Formar endereço de acesso à memória: A ALU adiciona os operandos para formar o endereço efetivo e coloca o resultado num registo.
- Instrução da ALU do tipo registo-registo: A ALU executa a operação especificada pelo *opcode* nos registos *Ra* e *Rb*, sendo o resultado guardado num registo.

- Instrução da ALU do tipo registo-*imm*: Semelhante ao anterior, só que em vez de a operação ser sobre os registos A e B, é sobre o registo A e o conteúdo em *Imm* que foi obtido no estágio anterior.

O quarto estágio é acesso à memória (*Memory access*). Qualquer acesso à memória é feito neste estágio. Caso seja um *load*, acede-se ao endereço calculado anteriormente e guarda-se o valor no registo indicado pela instrução.

$$Rd \leftarrow \text{Mem[ALU\_output]}$$

Caso seja um *store*, o valor no registo *Rd* é guardado no endereço calculado.

$$\text{Mem[ALU\_output]} \leftarrow Rd$$

O quinto e último estágio é atualização do register file (Write back). Neste estágio os registos são atualizados de acordo com os valores obtidos na instrução executada.

4.5. *Hazard unit*

Com a introdução de *pipeline* irão ocorrer eventualmente *hazards* ao longo da execução de um programa. Os *hazards* induzem resultados incorretos na execução de um programa. Existem três tipos de *hazards*, os quais são referidos por *hazard* estrutural, *hazard* de dados e *hazard* de controlo. Os *hazards* estruturais ocorrem quando existe conflito a nível dos recursos disponíveis, os *hazards* de dados acontecem quando uma instrução depende do resultado de uma instrução anterior ainda presente no *pipeline* e os *hazards* de controlo ocorrem quando há instruções que alterem o valor do *PC*.

4.6. Barramento

Este barramento é utilizado pelo CPU para comunicar com os periféricos existentes no sistema. Neste barramento são enviados comandos para os periféricos que podem ser de leitura ou escrita. Pode-se ver na Figura 9 a interface de programação dos periféricos:

- O *Opcode* tem o mesmo propósito que nas outras instruções;
- O Registo indica qual o registo para o qual se vai escrever caso seja uma leitura, ou caso seja uma escrita, é o valor deste registo que é enviado para o periférico;
- O Periférico indica qual o periférico que vai ser utilizado;
- O bit R/W indica se é uma escrita ou leitura dependo do seu valor, sendo que o valor 1 indica uma leitura e o valor 0 indica uma escrita;
- O Comando serve para indicar qual a operação que irá ser realizada no periférico.

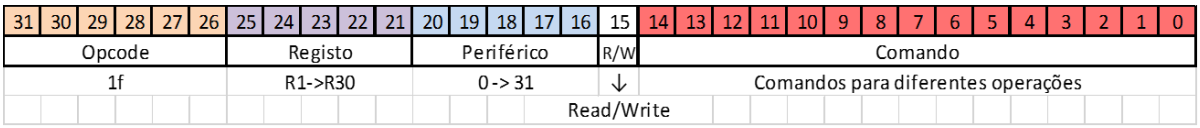


Figura 9 – Formato da instrução de acesso aos periféricos

4.7. Periféricos

Este SoC é dotado de cinco periféricos, sendo eles:

- Porto I/O;
- *Timers*;
- DVIC;
- UART;
- SYSTICK;

Entre eles existe um que comunica com todos, pois é o responsável pelas interrupções, nomeadamente o DVIC. Todos os outros periféricos necessitam de comunicar com o DVIC para indicar que precisam do processador para atendar a uma interrupção despoletada por eles.

O barramento usa três sinais para comunicar com os periféricos:

- A palavra de controlo (*ctrl*) – é nesta palavra que se encontra o que deve ser feito no periférico, como por exemplo qual o registo que se deve modificar. Também contém qual o periférico onde vai ser guardado o valor caso seja uma leitura e o bit que indica se é uma leitura ou escrita;
- A palavra de dados (*data*) – contêm o valor que irá ser usado para configurar algum registo, como por exemplo o valor de contagem de um *timer*;
- Sinal seleção (*sel*) – este sinal indica qual o periférico que está a ser acedido.

Sempre que o programador pretende alterar as configurações de algum periférico, o sinal de seleção necessita de estar a um e a palavra de dados e controlo têm de ter os valores correspondentes ao que ser quer fazer no periférico.

4.7.1. Porto I/O

Este periférico está limitado de momentos aos recursos disponíveis na placa, logo apenas é capaz de ler um valor de 8 bits dado por um *switch* e escrever um valor de 8 bits para os oitos *leds* disponíveis. Mais a frente é possível ver o modelo de programação, onde se pode ver que tem apenas um comando, que serve para ler ou escrever dependendo do bit de R/W.

4.7.1.1. Interface do I/O com todo o sistema

Na Figura 10 está apresentada a interação do periférico I/O com o resto do sistema. Ao todo tem 10 ligações, sendo sete delas de entrada e três de saída.

Podemos dividir os sinais de entrada em três grupos. Os sinais a amarelo são os sinais comuns a todos os periféricos, sinal de relógio e sinal de *reset*. A vermelho temos os sinais provenientes do barramento e a verde os sinais que vêm do exterior do sistema.

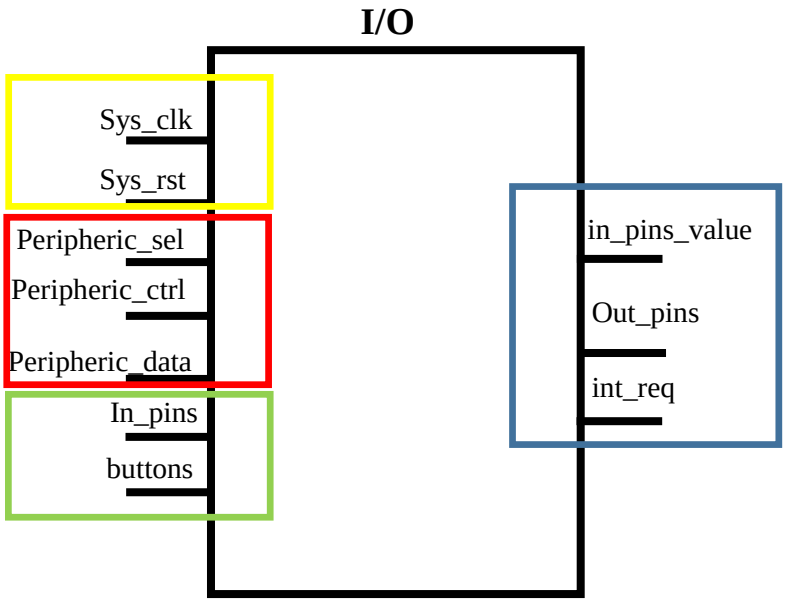


Figura 10 – Interface do I/O com o sistema

Os sinais de saída são enviados todos para sítios diferentes. O sinal *in_pins_value* é enviado para o processador, o sinal *out_pins* é enviado para o exterior do sistema e o sinal *int_req* é enviado para um outro periférico, o qual será explicado posteriormente neste capítulo.

4.7.1.2. Sinais de entrada/saída

Do exterior

Os sinais *in_pins* e *buttons* estão ligados a um *switch* e cinco botões, respetivamente, na placa de desenvolvimento. Estes sinais têm sempre o valor apresentado tanto nos *switchs* como nos botões. O *switch* serve para enviar um valor de 8 bits para o periférico, enquanto os cinco botões servem para despoletar cinco interrupções distintas.

Para o processador

Deste módulo apenas um sinal é enviado para o processador, sendo ele o *in_pins_value*. Este sinal, sempre que for feita uma leitura do periférico I/O, carregará o valor presente no *switch*. Este valor será escrito num registo à escolha do utilizador para futuro uso.

Para o exterior

Sendo um periférico I/O é necessário, além do sinal de entrada, um sinal de saída que neste caso é o *out_pins*. Este sinal está ligado aos oito *leds* da placa de desenvolvimento. Sempre que for feita uma escrita para o periférico, este sinal irá tomar o valor enviado, sendo o mesmo refletido nos oito *leds*.

Para o DVIC

O sinal *int_req* tem o tamanho de cinco bits, pois é a quantidade de interrupções suportadas pelo periférico. Está ligada ao controlador de interrupções e o valor que envia varia dependendo do botão pressionado.

4.7.1.3. Modelo de programação do I/O

As opções de programação para este periférico consistem em, tal como referido anteriormente, ler um valor do *switch* presente na placa de desenvolvimento ou escrever um valor para os oitos *leds*. Para estas duas opções é necessário apenas um comando, como se pode ver na Figura 11. Alterando apenas o bit de R/W é possível ler ou escrever deste periférico.

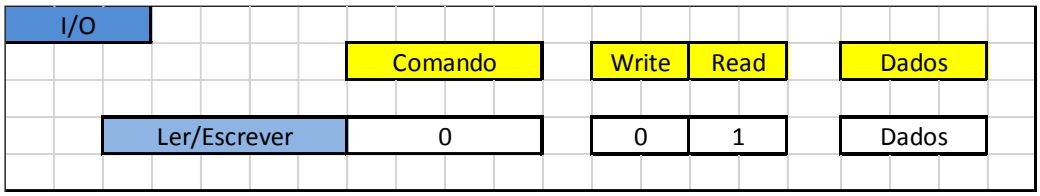


Figura 11 – Modelo de programação do I/O

4.7.2. *Timers*

Os *timers* são configuráveis e programáveis por *software*. O modelo de *timer* escolhido foi baseado nos *timers* do Tricore da Infineon, nomeadamente um *timer* de 32 bits, que conforme a configuração pode ser subdividido para criar novos *timers*. Este *timer*:

- Permite contagem ascendente e descendente;
- Permite funcionar como temporizador ou contador;
- Efetua o *reload* do *timer* em *overflow* e *underflow*;

- Dividindo o *timer* principal podem ser obtidas várias configurações desde que o número de bits total dos *timers* não ultrapasse os 32 bits. Seguem-se as possíveis combinações de *timers*:
  - 4 *Timers* de 8 bits;
  - 2 *Timers* de 8 bits + 1 *Timer* de 16 bits;
  - 1 *Timer* de 24 bits + 1 *Timer* de 8 bits;
  - 2 *Timers* de 16 bits;
  - 1 *Timer* de 32 bits.

Para se obter todas estas configurações é necesário configurar dois registos, TIMCONFA e TIMCONFB. Com o registo TIMCONFA (Figura 12) decide-se qual das diferentes configurações se irá escolher, bem como quais dos *subtimers* irão estar ativos.

TIMCONFA

c2	c1	c0	e3	e2	e1	e0

E3	E2	E1	E0	
0	0	0	0	Timers disable
1	1	1	1	Timers enable

C2	C1	C0	
0	0	0	4 Timers 8-bits
0	0	1	1 Timers 16-bits & 2 Timers 8-bits
0	1	0	2 Timers 16-bits
0	1	1	1 Timers 24-bits & 1 Timers 8-bits
1	0	0	1 Timer 32-bits

Figura 12 – Modelo de programação do registo TIMCONFA

O registo TIMCONFB (Figura 13) é um registo que existe para cada um dos quatro *subtimers* de 8 bits. Com ele define-se se a contagem será ascendente ou descendente, qual a origem do sinal de relógio e se o valor é atualizado na transição positiva, negativa ou ambas.

TIMCONFB

Mode	CLK1	CLK0

Mode	
0	Ascendente
1	Descendente

CLK1	CLK0	
0	0	Clock interno
0	1	Clock ext posedge
1	0	Clock ext negedge
1	1	Clock ext edge

Figura 13 – Modelo de programação do registo TIMCONFB

Na Figura 14 é possível ver como funciona o bloco de *timers*. O sinal de relógio não entra diretamente nos *timers* dois, três e quatro. O *tick* destes pode ser a *flag* de *overflow* do *timer* anterior ou do sinal de relógio, dependendo da sua configuração. É por esta razão que todas as saídas de *overflow*, com exceção da saída do último *timer*, vão estar ligadas a *multiplexers*. Dependendo do modo de configuração estas saídas vão indicar a interrupção de *overflow* do *timer* ou dar o *tick* ao *timer* seguinte.

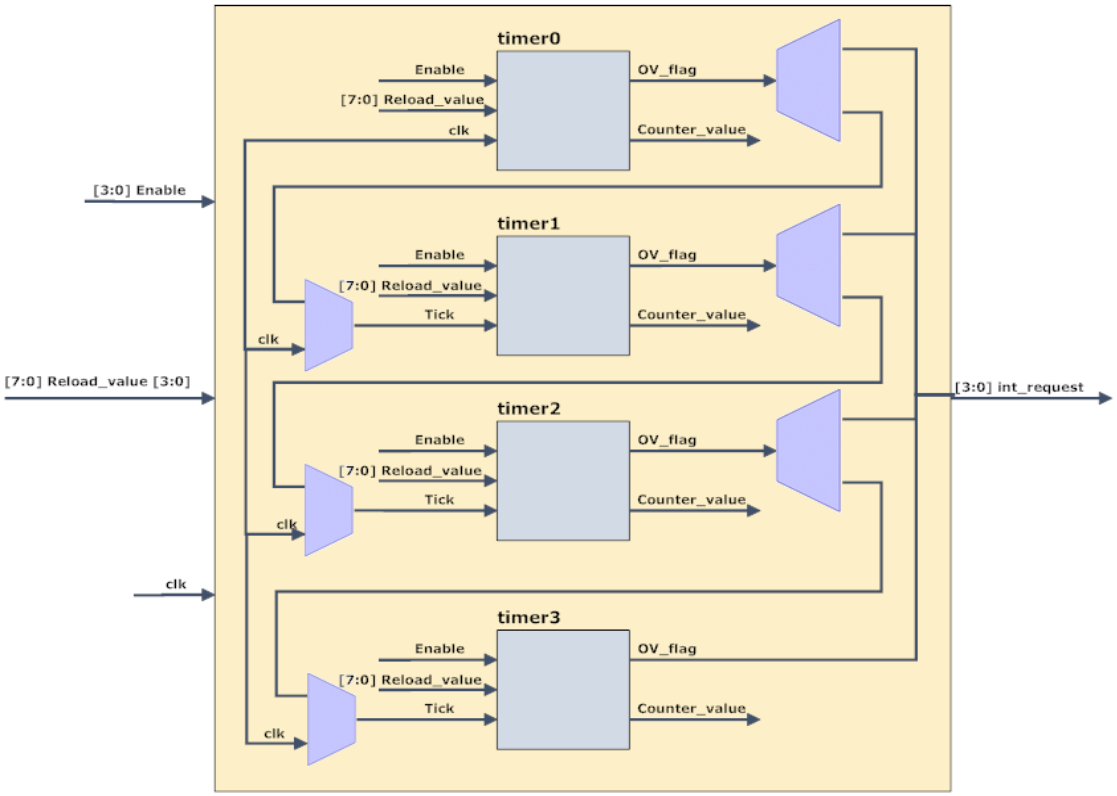


Figura 14 – Bloco de *Timers*

De seguida são apresentados os diferentes modos de funcionamento dos *timers* recorrendo a diagramas de blocos, seguidos de uma breve explicação.

Funcionamento no modo de 4 *timers* 8 bits

No modo de funcionamento com quatro *timers* de 8 bits, o *tick* de contagem é o sinal de relógio para todos os *timers* e as saídas de *overflow* estão todas ligadas à saída *int_request*. O funcionamento dos *timers* é independente e cada um pode ter o seu valor próprio de *reload*.

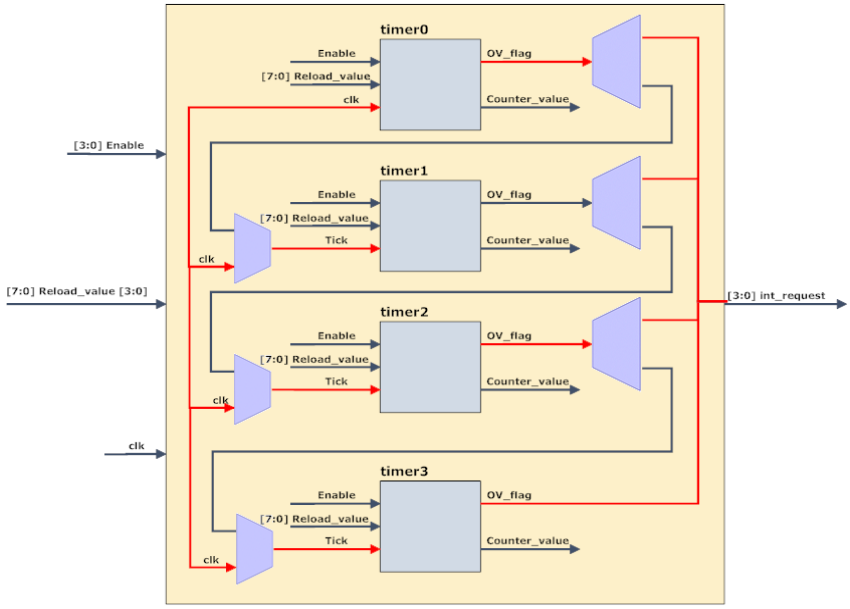


Figura 15 – 4 *Timers* 8 bits

Funcionamento no modo de 2 *timers* 8 bits e 1 *timer* 16 bits

No modo de funcionamento de dois *timers* de 8 bits e um de 16 bits, são utilizados dois dos quatro *timers* em conjunto para formar o *timer* de 16 bits. Para formar o *timer* de 16 bits é ligada a saída *overflow* do primeiro *timer* de 8 bits ao *tick* do *timer* seguinte. Esta saída deixa de estar ligada à saída de interrupções e só a saída de *overflow* do segundo *timer* serve como sinal de interrupção. Os outros dois *timers* de 8 bits continuam a operar de forma independente.

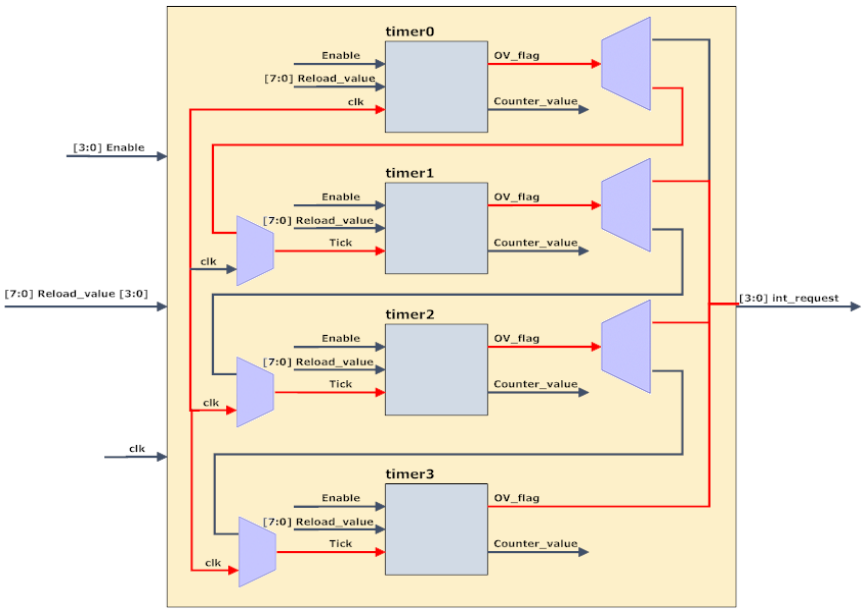


Figura 16 – 1 *Timer* 16 bits + 2 *Timers* 8 bits

Funcionamento de no modo 2 *timers* 16 bits

No modo de funcionamento dois *timers* de 16 bits estes são formados da mesma forma que o único *timer* de 16 bits no caso anterior. A única diferença é que neste caso todos os *timers* de 8 bits são agrupados aos pares para formar os dois *timers* de 16 bits. Cada *timer* de 16 bits funciona de forma independente com o seu valor próprio de contagem e com o sinal de relógio a dar o *tick* a um dos *timers* de 8 bits que o compõem.

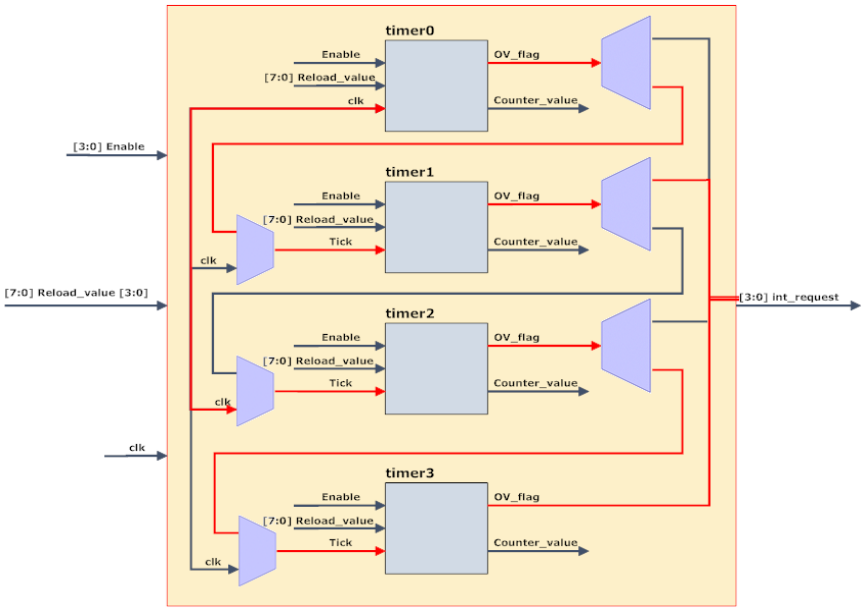


Figura 17 – 2 *Timers* 16 bits

Funcionamento no modo de 1 *timer* 8 bits e 1 *timer* de 24 bits

No modo de funcionamento de um *timer* de 8 bits e um *timer* de 24 bits três *timers* de 8 bits são usados para formar o *timer* de vinte e quatro. Para isso o *overflow* do primeiro *timer* dá o *tick* ao segundo, que por sua vez é responsável por dar o *tick* ao terceiro *timer* quando atinge o seu valor máximo de contagem. O primeiro *timer* recebe o *tick* do sinal de relógio. O último *timer* de 8 bits funciona de forma independente.

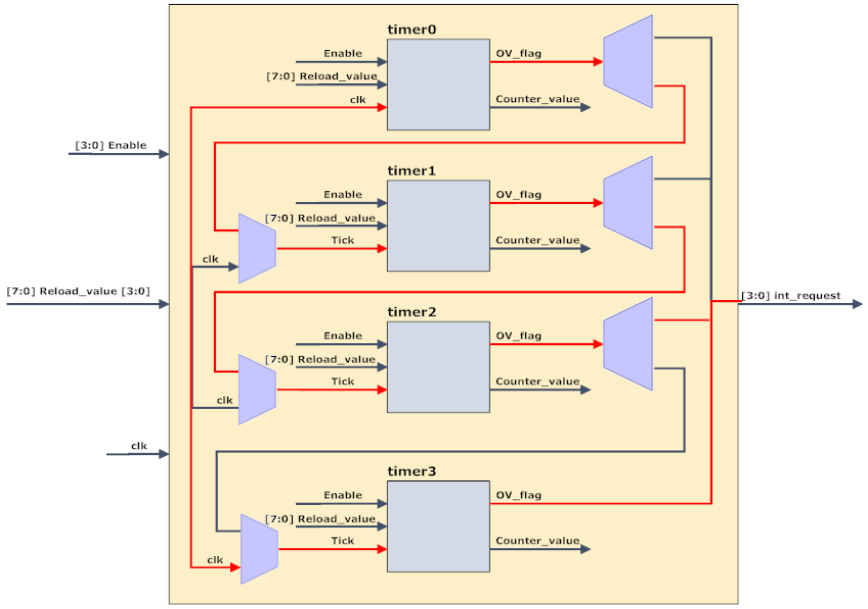


Figura 18 – 1 *Timer* 24 bits + 1 *Timer* 8 bits

Funcionamento de no modo 1 *timer* 32 bits

No modo de funcionamento de um *timer* de 32 bits todos os quatro *timers* são utilizados para formar o de 32 bits. Tal como nos casos anteriores, o *overflow* do *timer* um é o responsável pelo *tick* do segundo *timer*, que por sua vez é responsável pelo *tick* do *timer* três e este pelo *tick* do *timer* quatro. O primeiro *timer* é que recebe o sinal de relógio como *tick*. Neste caso apenas a *flag* de interrupção do *timer* quatro está ativa.

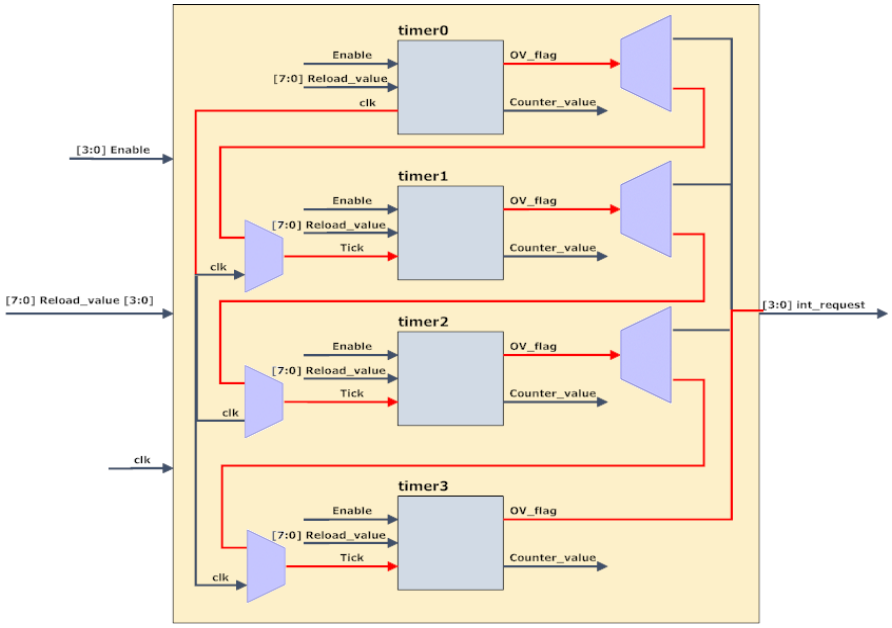


Figura 19 – 1 *Timer* 32 bits

4.7.2.1. Interface dos *Timers* com todo o sistema

Este módulo possui cinco sinais de entrada e dois sinais de saída. Dos cinco sinais temos a amarelo os dois comuns a todos os periféricos, bem como os sinais provenientes do barramento, que também são comuns a todos os periféricos.

A azul tem-se os sinais de saída, onde um deles é enviado para o controlador de interrupções e o outro para o processador.

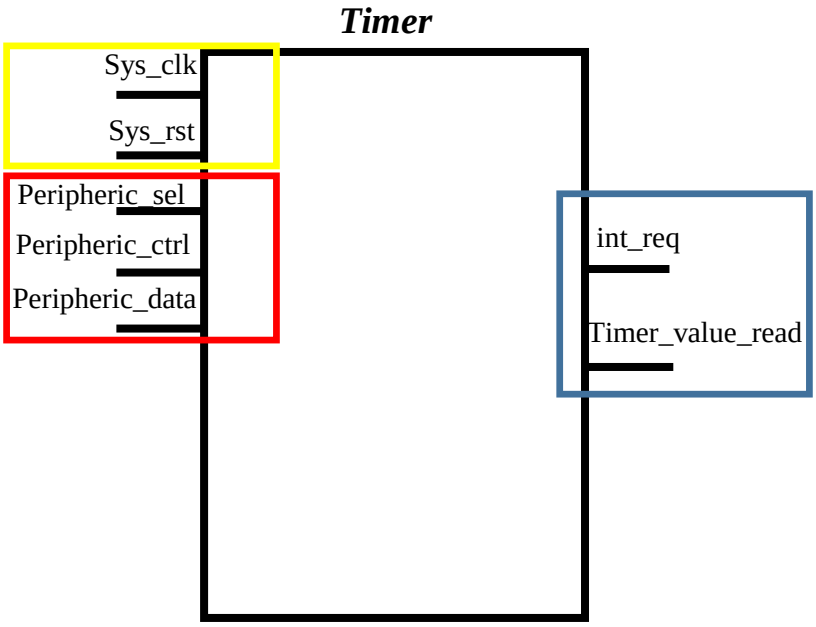


Figura 20 – Interface do *Timer* com o

4.7.2.2. Sinais de entrada/saída

Para o processador

O sinal enviado para o processador é o *timer_value_read*. Este sinal, sempre que for feita uma leitura do periférico *timer*, irá conter o valor do registo TIMCNT# do *subtimer* escolhido. Este valor será escrito num registo à escolha do utilizador para futuro uso.

Para o DVIC

O sinal *int_req* tem o tamanho de 4 bits, pois é a quantidade de interrupções que o periférico pode despoletar no máximo. Está ligado ao controlador de interrupções e o valor que envia varia conforme o *timer* em que ocorrer o *overflow* ou *underflow*.

4.7.2.3. Modelo de programação dos *Timers*

Para configurar o bloco de *timers* é necessário configurar cada *subtimer*. Como já foi referido anteriormente, é possível configurar três registos para cada *subtimer* e um registo comum a todos os registos. Na Figura 21 é possível ver o modelo de programação deste periférico.

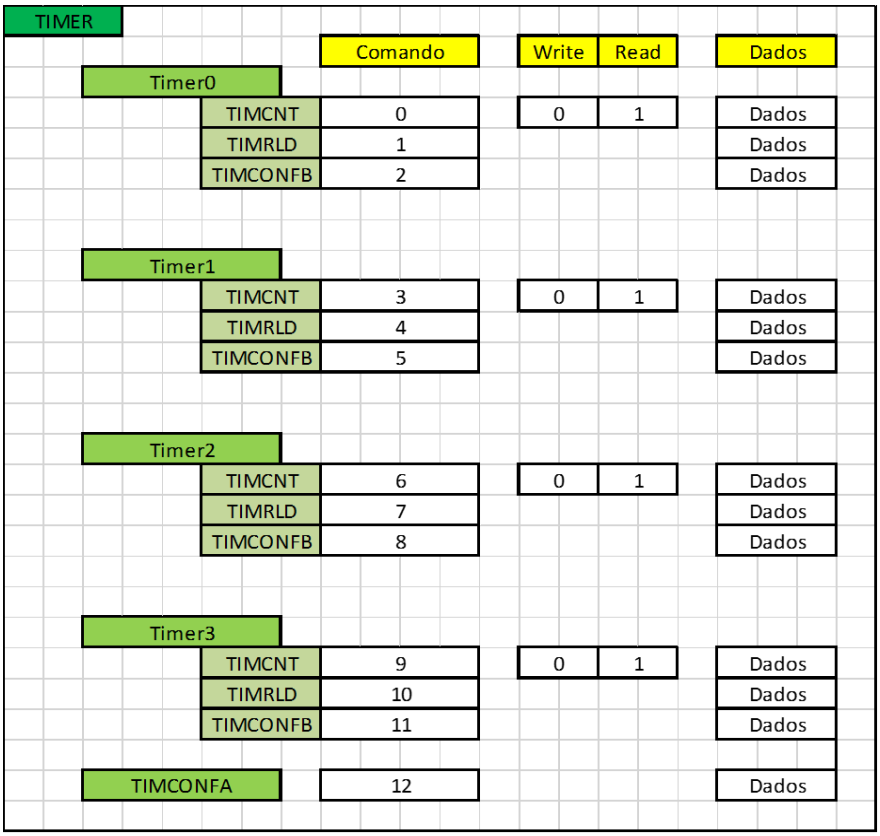


Figura 21 – Modelo de programação do *Timer*

4.7.3. UART

A placa de desenvolvimento é dotada de duas COMs, mas apenas uma possui um conetor. Este conetor é um RS232 DB9 macho estando ligado à COM1 como DCE *host*, ou seja, para comunicar com um computador é necessário um cabo *null modem*. Este porto de comunicação série pode operar até um *baudrate* de 115200, mas apenas possui alguns pinos conectados, sendo eles o TX e o RX.

A comunicação série é feita segundo o *standard* RS-232. Uma comunicação segundo este standard pode ser *half-duplex* ou *full-duplex*, ter *hardware/software flow control* e pode ter várias configurações quanto a trama enviada. A UART é *half-duplex*, não possui *flow control* de qualquer género e em relação às possíveis configurações, apenas possui uma, que é *baudrate* fixo, oito bits de dados, um stop bit e sem paridade. Na Figura 22 pode-se ver o formato da transmissão de um byte.

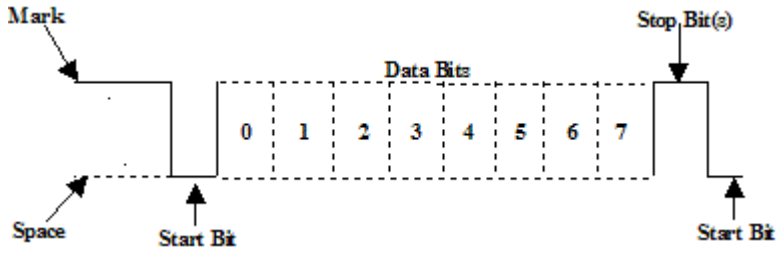


Figura 22 – Diagrama de transmissão de um byte [46]

Sempre que se iniciar o envio/receção de um caractere será necessário começar com o *start* bit, seguido dos oito bits de dados e acabando com um *stop* bit.

4.7.3.1. Interface da UART com todo o sistema

Este periférico possui cinco sinais de entrada e dois sinais de saída. Os sinais a amarelo como sendo os sinais de relógio e *reset* e os sinais a vermelho como sendo os sinais provenientes do barramento. A verde tem-se o sinal RX que está ligado a um pino físico da placa de desenvolvimento e a azul tem-se os dois sinais de saída, um deles sendo o TX que também está ligado a um pino físico e o *int_req*, que tal como nos outros periféricos, está ligado ao DVIC.

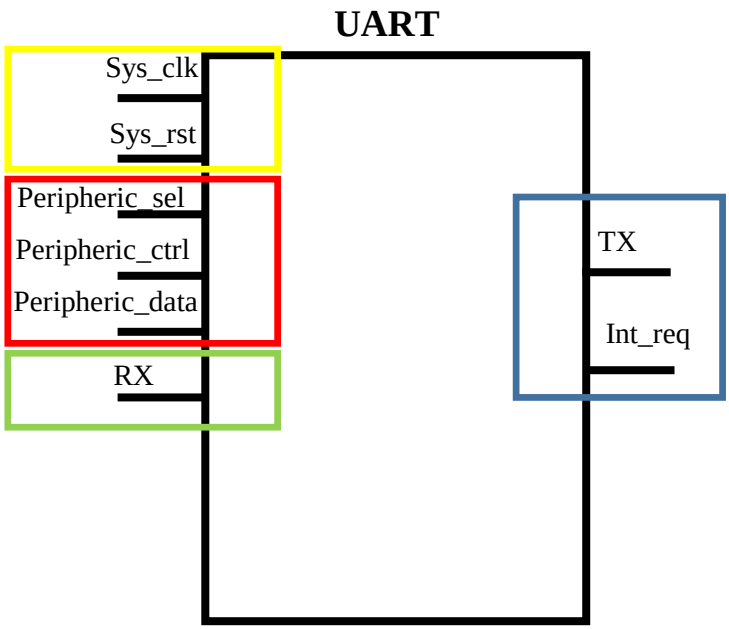


Figura 23 – Interface da UART com o sistema

4.7.3.2. Sinais de entrada/saída

<u>Do exterior</u>

O sinal RX está ligado ao pino físico da porta série da placa de desenvolvimento, ou seja, é através deste sinal que são recebidas as tramas enviadas por um terminal exterior.

Para o exterior

O sinal TX, que também está ligado ao pino físico da porta série, é o responsável por enviar as tramas para um terminal exterior.

Para o DVIC

Neste periférico o sinal *int_req* possui um tamanho de dois bits, para sinalizar a receção ou envio de um caractere.

4.7.3.3. Modelo de programação da UART

Para programar este periférico basta seguir o modelo de programação presente na Figura 24. Pode-se ver que não é necessário muitos passos para utilizar, basta habilitar a UART e depois enviar ou ler o caractere.

UART	Comando	Dados
Send	0	Dados
Enable	1	Dados
Receive	2	Dados

Figura 24 – Modelo de programação da UART

4.7.4. SYSTICK

O SYSTICK não é nada mais que um *timer*, logo o modelo de programação é igual ao do periférico *timers* anteriormente mostrado. Sendo assim, as possíveis configurações são semelhante ao *timers*.

4.7.4.1. Interface do SYSTICK com todo o sistema

Para este periférico bastam os cinco sinais comuns a todos os periféricos, ou seja, o sinal de relógio e *reset* rodeados a amarelo e os sinais provenientes do barramento rodeados a vermelho. A azul estão os sinais de saída.

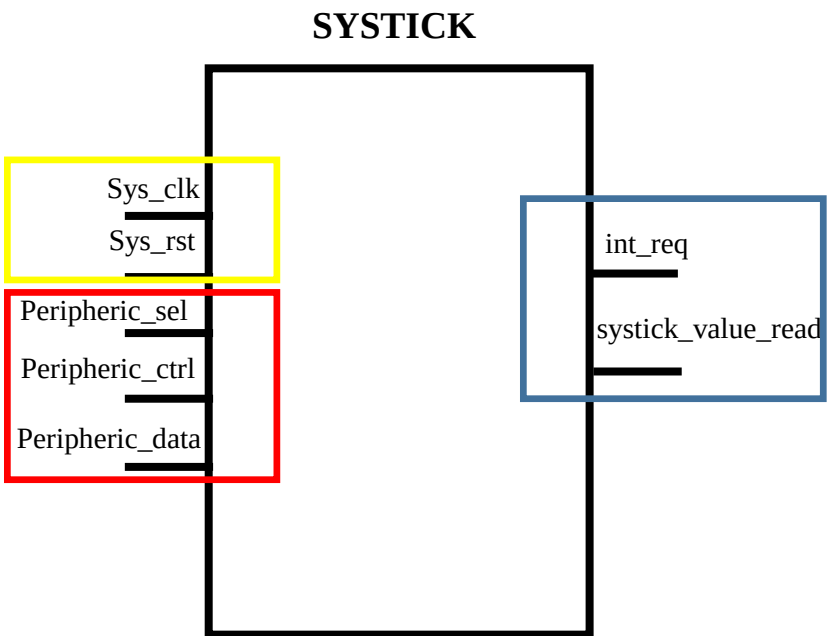


Figura 25 – Interface do SYSTICK com o

4.7.4.2. Sinais de entrada/saída

Para o processador

Dos dois sinais de saída deste periférico apenas um deles é enviado para o processador, sendo esse o *systick_value_read*. Este sinal serve para, caso seja feita uma leitura do periférico, escrever o valor atual de contagem num registo a escolha do programador.

Para o DVIC

Neste periférico o sinal *int_req* tem o tamanho de 1bit. Como este periférico não passa de um *timer* de 32 bits apenas é necessário 1bit para indicar que houve o *overflow*.

4.7.4.3. Modelo de programação do SYSTICK

Na Figura 26 pode-se ver o modelo de programação deste periférico. Para o programar bastam três comandos, um para dar o valor de contagem, outro para dar o valor de recarga e outro para habilitar/desabilitar o periférico. É possível ler o valor atual de contagem através do valor do bit R/W usando o comando zero.

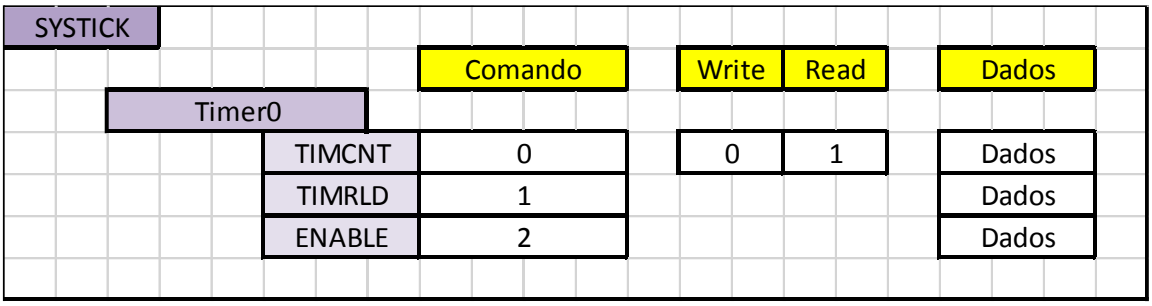


Figura 26 – Modelo de programação do SYSTICK

4.7.5. Controlador de interrupções programável

Neste tópico será feita uma abordagem geral sobre o controlador de interrupções, denominado de *DVIC*. Posteriormente irão ser definidos quais os componentes necessários para o seu correto funcionamento e também qual o comportamento das interrupções.

4.7.5.1. Vista Geral

- Suporta 12 interrupções configuráveis e o *RESET* não configurável;
- Interrupções por *hardware* ou *software*:
  - SYSTICK;
  - Botão *center*
  - *Timer* 0;
  - *Timer* 1;
  - *Timer* 2;
  - *Timer* 3;
  - Botão *north*;
  - Botão *east*;
  - Botão *west*;
  - Botão *south*;
  - UART_Tx;
  - UART_Rx;
- Quando uma interrupção ocorre é guardado na pilha os seguintes registos:
  - *Program Counter*;
  - Registos que forem usados na rotina de serviço à interrupção.

4.7.5.2. Comportamento das interrupções

O comportamento das interrupções deverá ser o seguinte sob as circunstâncias mais críticas:

Pedido de interrupção único

No caso do pedido de interrupção único, quando um pedido de interrupção é ativo este terá de ser atendido. Assim, ao entrar na interrupção é necessário executar o *PUSH* ao *PC* e aos registos e quando esta terminar haverá o *POP*. Como não há mais pedidos o processador continua no seu modo normal.

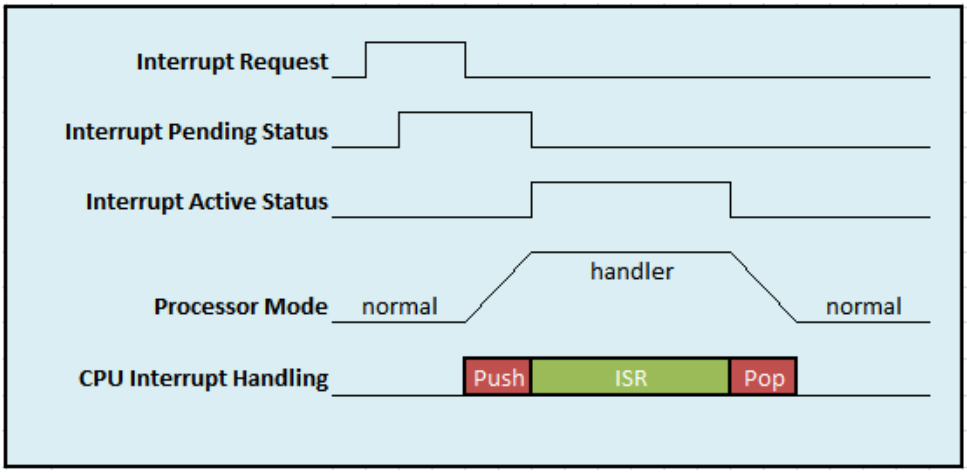


Figura 27 – Pedido de interrupção único

Pedido de interrupção continuo

Neste caso, acontece um constante pedido de interrupção. Como a interrupção é a mesma, depois de ser processada a primeira vez ela irá ser atendida uma vez mais. Entre cada *ISR* haverá sempre o *PUSH* e o *POP* dos registos mais importantes.

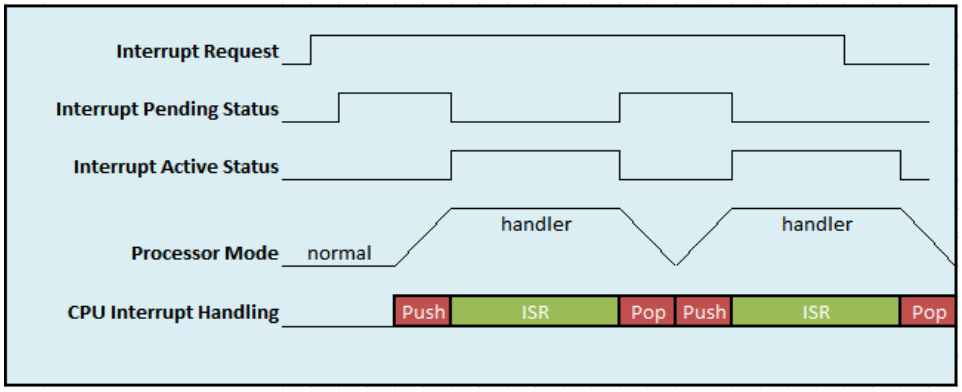


Figura 28 – Pedido de interrupção contínuo

Pedido de interrupção único durante o *handler*

Neste caso haverá um pedido de interrupção quando a mesma interrupção está a processar o *handler*. Quando a primeira interrupção acabar o processador terá de atender mais uma vez a mesma interrupção, pois, existe um pedido pendente da mesma.

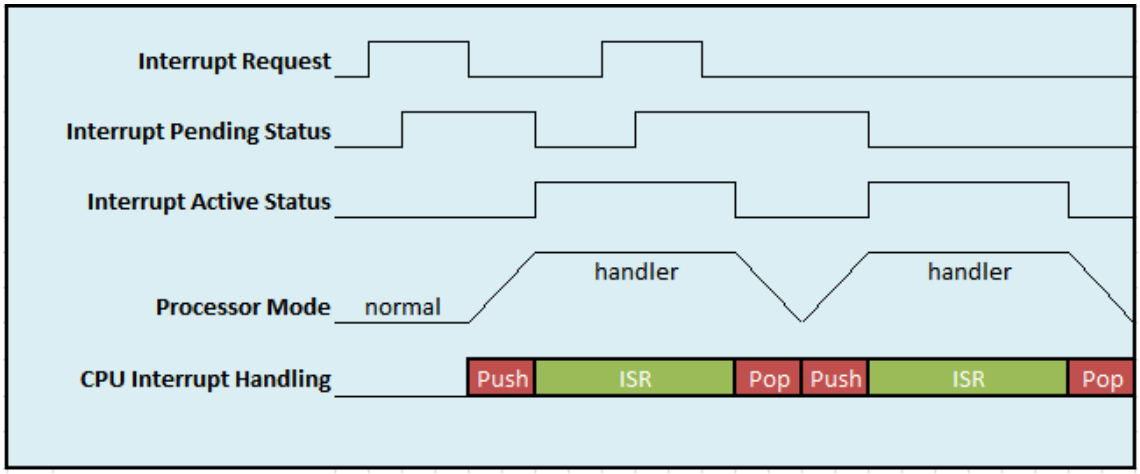


Figura 29 – Pedido de interrupção único durante o *handler*

Dois pedidos de Interrupção com diferentes prioridades ao mesmo tempo

Neste caso existem duas interrupções com prioridades diferentes. A interrupção com maior prioridade terá de ser atendida primeiro e quando esta acabar irá ser atendida a próxima. Como ocorreram ao mesmo tempo a *ISR* de menor prioridade ficará em estado pendente.

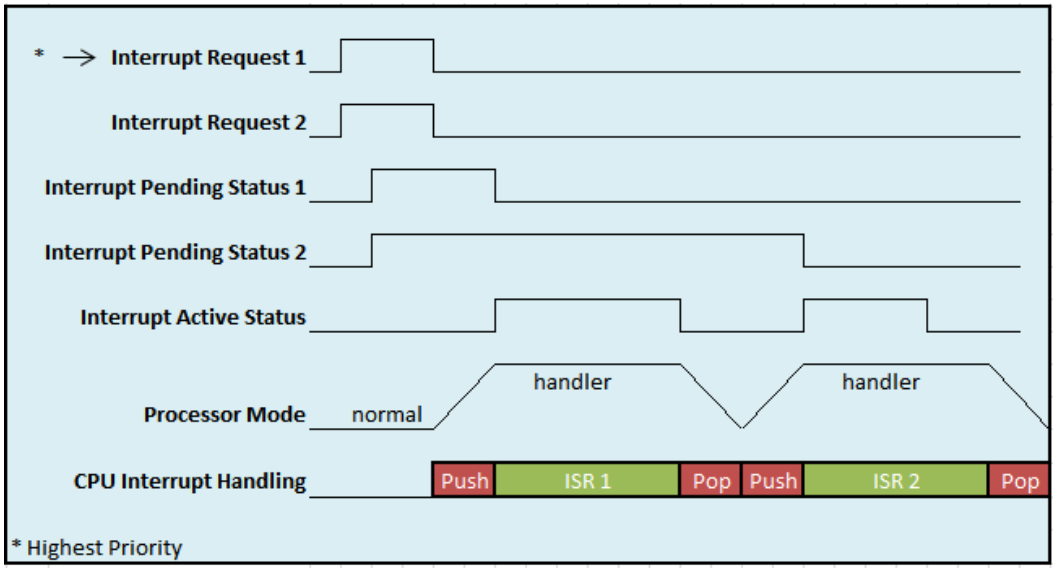


Figura 30 – Dois pedidos de Interrupção com diferentes prioridades ao mesmo tempo

Aparecimento de uma interrupção com maior prioridade

Neste caso acontecem dois pedidos de interrupção de prioridades diferentes em alturas diferentes mas antes do processador atender qualquer interrupção. O *DVIC* deverá atender primeiro a interrupção de maior prioridade e só depois de esta terminar deverá atender a de menor prioridade.

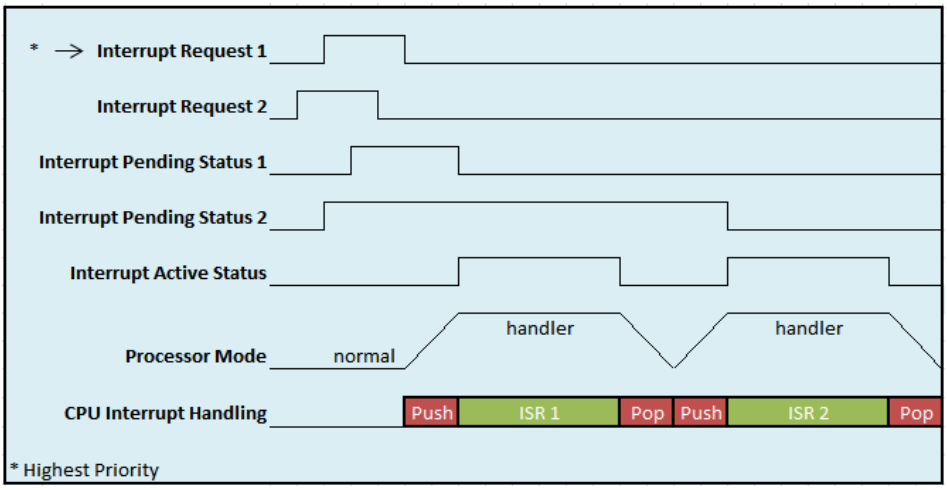


Figura 31 – Aparecimento de uma interrupção com maior prioridade

Ocorrência de uma interrupção de maior prioridade durante o *handler*

Neste caso, uma interrupção de maior prioridade acontece quando uma de menor está a ser processada. O *DVIC* deve colocar a interrupção atual no estado pendente e tratar a interrupção de maior prioridade. Quando esta terminar deve voltar para a de menor até esta terminar.

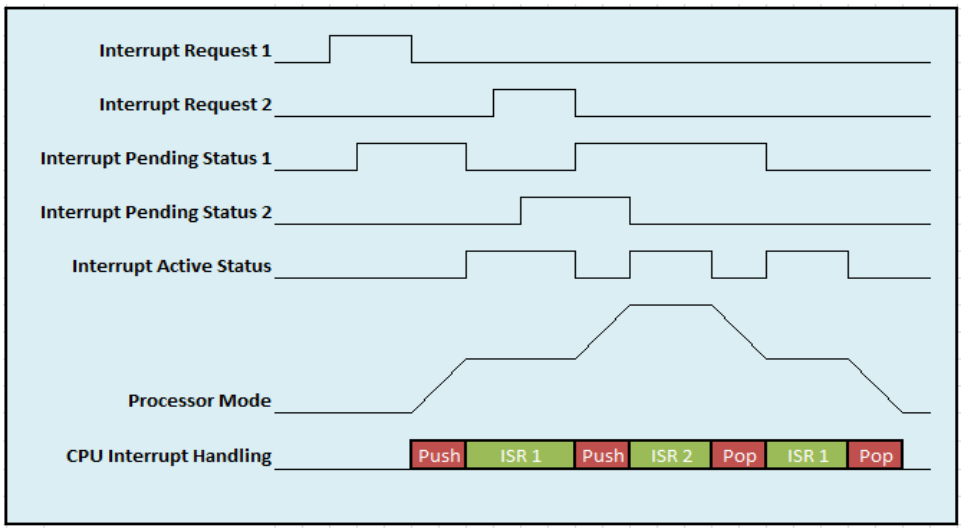


Figura 32 – Ocorrência de uma interrupção de maior prioridade durante o *handler*

4.7.5.3. Registos de estado das interrupções

Para alojar toda a informação relativa às interrupções houve a necessidade de criar um registo para cada uma. Será nesses registos que se guardará o estado da interrupção, ou seja, se está pendente, ativa e habilitada. Para além do estado da interrupção este também guardará a prioridade de cada uma, para posteriormente saber qual a interrupção com maior prioridade. Pode-se ver o formato de cada um dos registos na Figura 33.

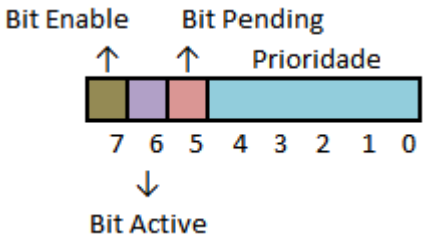


Figura 33 – Formato do registo de interrupção

Na Tabela 4 é possível ver as 16 interrupções, onde apenas 12 delas estão ativas mais o *RESET*, sendo as restantes para futuras expansões. Todas as interrupções à exceção do *RESET* são programáveis, sendo possível definir a prioridade das mesmas e alterar o estado delas. Quando houver um *RESET*, por defeito, terá de eliminar todos os dados de cada registo relativo a cada interrupção.

Tabela 4 – Interrupções

Número	Nome	Prioridade	Periférico
0	*Reset*	Não programável	*RESET*
1	IRQ#0	Programável	SYSTICK
2	IRQ#1	Programável	Botão *center*
3	IRQ#2	Programável	Timer0
4	IRQ#3	Programável	Timer1
5	IRQ#4	Programável	Timer2
6	IRQ#5	Programável	Timer3
7	IRQ#6	Programável	Botão *north*
8	IRQ#7	Programável	Botão *east*
9	IRQ#8	Programável	Botão *west*

10	IRQ#9	Programável	Botão *south*
11	IRQ#10	Programável	UART_Tx
12	IRQ#11	Programável	UART_Rx
13	IRQ#12	Programável	Unused_4
14	IRQ#13	Programável	Unused_5
15	IRQ#14	Programável	Unused_6
16	IRQ#15	Programável	Unused_7

Atualização de um registo

Como foi dito anteriormente, cada entrada da Tabela 4 terá três estados:

- *Enable*;
- *Enable* & *Pendent*;
- *Enable* & *Active*.

Cada estado pode ser provocado pelo programador, para além do estado pendente que também é provocado por eventos externos. Na Figura 34, está representado um esquema para facilitar o entendimento desta pequena máquina de estados.

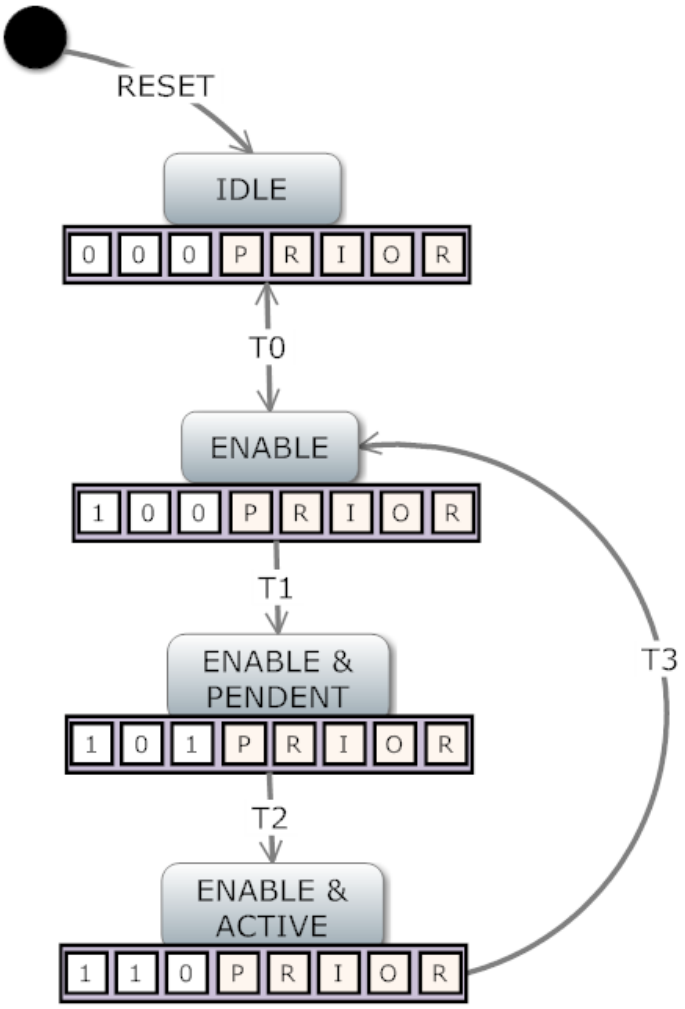


Figura 34 – Máquina de estados da configuração das interrupções.

Aquando do *reset* todas as interrupções passarão para o estado *IDLE*. No momento T0 o *bit enable* é programado e o seu campo (*MSB*) no registo de interrupção é colocado a um, estando a partir deste instante num estado *ENABLE*. Do estado *ENABLE* apenas será possível passar para o estado *ENABLE & PENDENT* (transição T1), quando a fonte de interrupção externa relativa a essa interrupção for acionada ou então quando o programador a despoletar por *software*. O *DVIC*

através dos mecanismos de seleção compara todas as interrupções e se houver interrupções no estado *ENABLE & PENDENT*, a que tiver a maior prioridade irá no ciclo de relógio seguinte (transição T2) passar para o estado *ENABLE & ACTIVE*. Quando a interrupção está neste estado, quer dizer que nesse instante está a ser executada. No momento T3 o estado da interrupção a ser executada deverá voltar para o estado *ENABLE* e deverá comportar-se desta forma até que eventualmente receba uma instrução que desative esta interrupção.

4.7.5.4. Pilha Dedicada

Surgiu a necessidade de criar este componente para auxiliar a máquina de estados aquando da atualização dos registos de estado ao lidar com "aninhamentos" de interrupções. Vai-se analisar o seguinte exemplo presente na Figura 35.

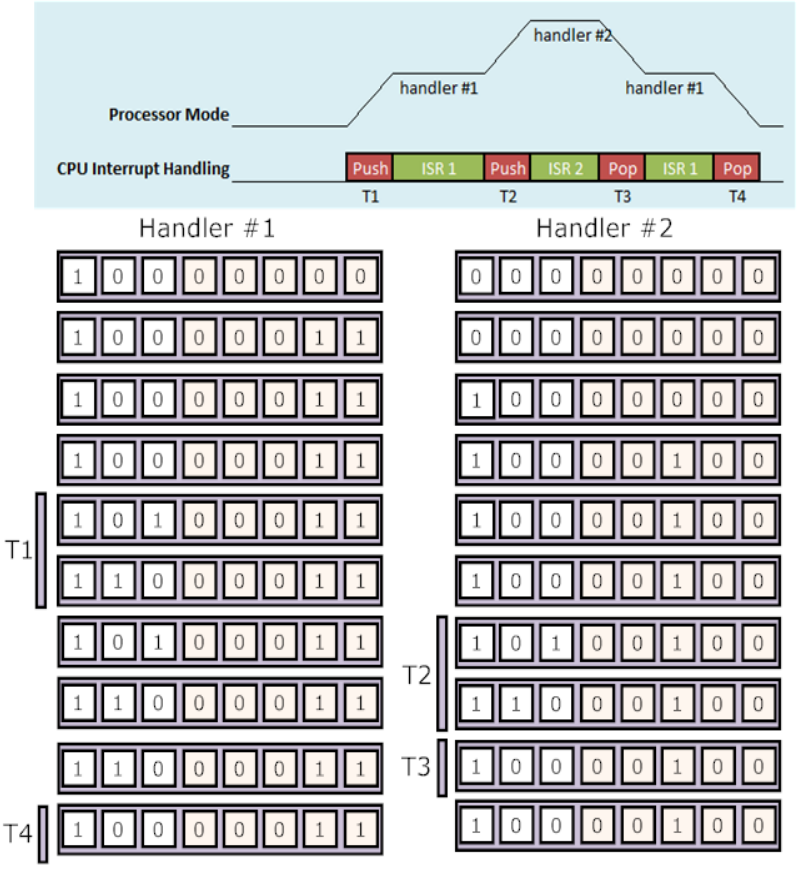


Figura 35 – Exemplo de um "aninhamento".

Ao executar um programa reparou-se que no momento em que o *handler* #1 passava do estado *ENABLE & ACTIVE* para *ENABLE* (transição T4), o *DVIC* "perdia" o endereço do respetivo *handler*, que era necessário para zerar a *flag active* (sexto bit mais significante do registo de interrupção). De uma forma simplista o *DVIC* deixava de saber qual o registo de estado que deveria alterar para manter tudo coerente.

Foi então criada uma pequena pilha interna ao *DVIC* apenas para guardar o endereço da *ISR* que entra em execução, para nunca perder o contexto sempre que lida com "aninhamentos". Desta forma pode-se ter várias *ISR* aninhadas, e nunca perder a conformidade dos registos de estado. Para tal, a pilha interna deverá comportar-se da seguinte forma nos seguintes momentos:

- Momento do *RESET*:

Tabela 5 – Exemplo da pilha *DVIC* no *reset*.

RESET		
	Endereço	**Conteúdo**
	0xF	
	…	
	0x5	
	0x4	
	0x3	
	0x2	
	0x1	
SP -->	0x0	

Neste momento a pilha deverá estar completamente vazia, todas as posições devem estar a zero, e o apontador da pilha deverá apontar para o endereço 0x0.

- Momento T0:

Tabela 6 – Exemplo da pilha *DVIC* no momento T0.

T0		
	Endereço	**Conteúdo**
	0xF	
	…	
	0x5	
	0x4	
	0x3	
	0x2	
SP -->	0x1	
	0x0	0x01

Este momento assinala a entrada de uma *ISR.* O processador dá início a execução de uma *ISR*, sendo guardado o endereço da *ISR* no início da pilha e o apontador passará a apontar para a posição adjacente 0x01. Sendo assim, sempre que uma *ISR* dá entrada o seu endereço é guardado na pilha e de seguida o apontador da pilha deve ser incrementado uma posição.

- Momento T1:

Tabela 7 – Exemplo da pilha *DVIC* no momento T1.

T1		
	Endereço	**Conteúdo**
	0xF	
	…	
	0x5	
	0x4	
	0x3	
SP -->	0x2	
	0x1	0x02
	0x0	0x01

Este cenário é similar ao anterior, uma *ISR* entra no estado *ENABLE & ACTIVE*, o seu endereço 0x02 é guardado na pilha, e de seguida o apontador incrementado. A particularidade que se deve referir, é que a *ISR* anterior não deixou o estado *ENABLE & ACTIVE*, então a pilha deve ter o valor do seu endereço guardado para posterior atualização.

- Momento T2:

Tabela 8 – Exemplo da pilha *DVIC* no momento T2.

T2		
	Endereço	**Conteúdo**
	0xF	
	…	
	0x5	
	0x4	
	0x3	
	0x2	
SP -->	0x1	0x02
	0x0	0x01

Nesta situação, a *ISR* com o endereço 0x02 terminou a execução, ou seja, na perspetiva do *DVIC*, recebeu um sinal de *RETI*. Esta *ISR* passou de *ENABLE & ACTIVE* apenas para *ENABLE*. Nestas circunstâncias, em primeiro lugar, deve-se decrementar o valor do apontador

da pilha e de seguida usar o conteúdo do apontador para atualizar a entrada da *ISR* e consequentemente alterar o estado para *ENABLE*.

- Momento T3:

Tabela 9 – Exemplo da pilha *DVIC* no momento T3.

T3		
	Endereço	**Conteúdo**
	0xF	
	…	
	0x5	
	0x4	
	0x3	
	0x2	
	0x1	0x02
SP -->	0x0	0x01

Este momento é muito semelhante ao anterior, só que neste caso quem termina a execução é a *ISR* cujo endereço tem o valor 0x01. É necessário alterar o estado da *ISR* apenas para *ENABLE*, e para tal, decrementa-se o apontador e de seguida usa-se o seu conteúdo para colocar o bit *active* a zero do registo de estado em questão.

Volta-se a reforçar que sem este componente não seria possível lidar com "aninhamentos". Este tipo de sistema permite lidar bastante bem com situações que possam necessitar até 16 "aninhamentos". Desta forma consegue-se manter a conformidade entre os três diferentes estados das 12 interrupções programáveis deste sistema.

4.7.5.5. Árvore binária de decisão

Este componente será responsável por selecionar a *ISR* com maior prioridade. Toda a sua arquitetura deverá ser puramente combinacional e deverá executar em apenas um ciclo de relógio. Por outras palavras, este componente irá a cada ciclo de relógio selecionar a *ISR* habilitada de maior prioridade.

A árvore binária de decisão deverá selecionar de entre as 12 *ISR*s a mais prioritária, comparando as *ISR*s duas a duas até obter no final a *ISR* com maior prioridade. O esquema da Figura 36 ilustra o formato da árvore de decisão.

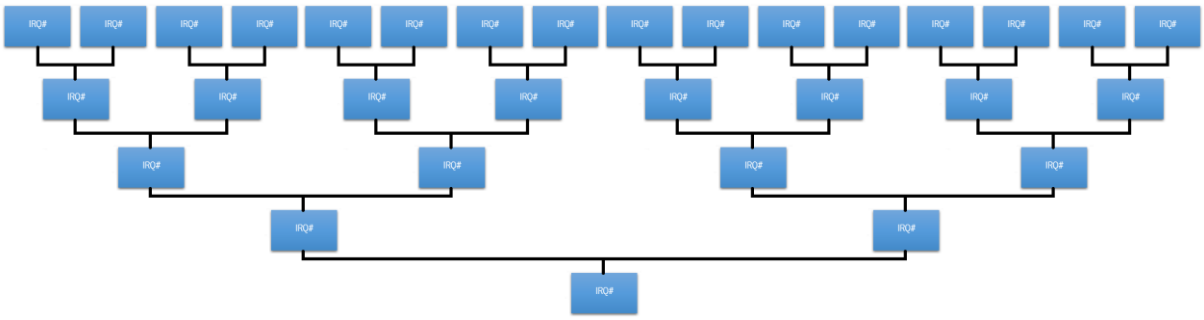


Figura 36 – Árvore binária de decisão

Em cada nó de decisão deverá ser feita a comparação *bit* a *bit*, e colocada na saída do nó a *ISR* com maior prioridade. A imagem da Figura 37 ilustra como deverá funcionar esse sistema.

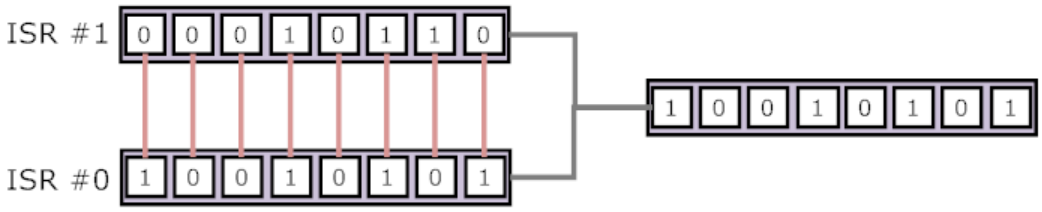


Figura 37 – Exemplo da comparação entre duas interrupções.

Esta estratégia para selecionar a *ISR* com maior prioridade foi escolhida por ser simples de implementar e ao mesmo tempo não haver atrasos no seu cálculo, pois obedece a um tipo de arquitetura puramente combinacional.

4.7.5.6. Interface do *DVIC* com todo o sistema

Nesta parte será explicada de que forma o *DVIC* será incorporado no sistema final – o SoC.

Este componente terá um total de 11 ligações, nove delas são sinais de entrada e irá apenas codificar dois sinais de saída.

Como indica a Figura 38, existem quatro grupos de sinais de entrada. A amarelo os sinais de relógio e reset, a vermelho

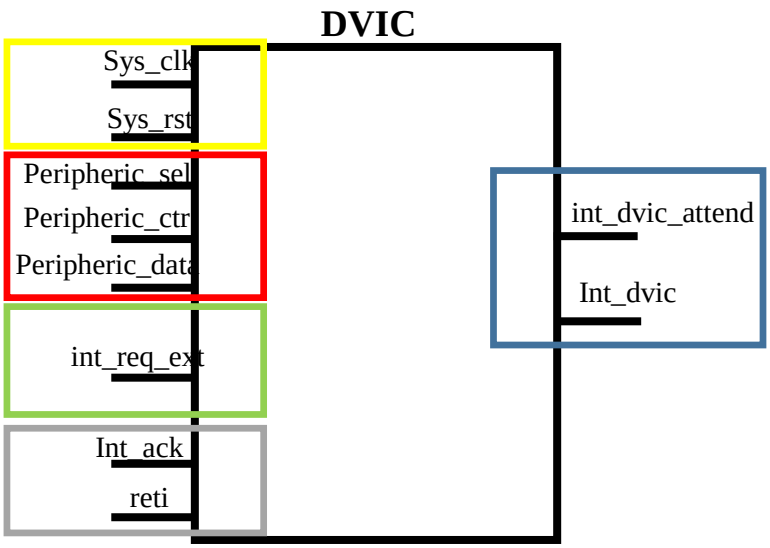


Figura 38 – Interface do DVIC com o

estão os sinais provenientes do barramento, a verde sinais originários dos vários periféricos do sistema e a cinzento sinais vindos do processador. Os sinais de saída, a azul, são todos direcionados ao processador.

4.7.5.7. Sinais de entrada/saída

Dos periféricos

Para este caso específico, o que notifica o *DVIC* que necessita de atender interrupções é apenas uma palavra de 16 bits, *int_req_ext*. A esta palavra estão ligados todos os periféricos implementados, o bloco de *timers*, o controlador *I/O*, a UART e o SYSTICK que contêm cinco fontes, quatro fontes, duas fontes e uma fonte de interrupção, respetivamente. Este sinal está normalmente com o valor 16'b1111111111111111, pelo que quando algum destes bits varia de um para zero, conforme o índice no registo, o bit pendente da *ISR* em questão é colocado a um, o que significa que a *ISR* fica no estado pendente.

Do processador

O processador apenas envia dois sinais ao *DVIC* que tem o propósito de notificar o *DVIC* em dois cenários distintos:

- Início da execução duma *ISR*;
- Fim da execução duma *ISR*.

O sinal responsável pela notificação do início da execução duma *ISR* é o *int_ack*, enquanto o sinal que notifica o final é apelidado por *reti*. O sinal *int_ack* irá alterar o *bit* pendente duma *ISR* de um para zero e o *bit* ativo de zero para um. O sinal *reti* apenas terá de colocar a zero o bit *active* da *ISR* que terminou de executar baseado-se na pilha interna do *DVIC*.

Para o processador

Este módulo, apenas tem um sinal de saída, o qual liga diretamente ao processador. Este sinal irá servir para notificar o processador qual o *handler* da *ISR* que deverá atender naquele instante.

4.7.5.8. Modelo de programação do *DVIC*

Relativamente à programação do *DVIC*, este irá obedecer aos mesmos critérios dos outros periféricos.

Este periférico deverá permitir programar as seguintes funcionalidades:

- Ativar ou desativar interrupções no sistema.

- Manipular a prioridade de cada *ISR* individualmente.
- Provocar interrupções por *software*.

Então para programar estas três características são usados três comandos distintos. Como existem 16 interrupções será preciso quatro bits para especificar qual a entrada que se pretende configurar.

Na Figura 39 está esquematizado a forma como o programador deverá proceder para interagir com o *DVIC*.

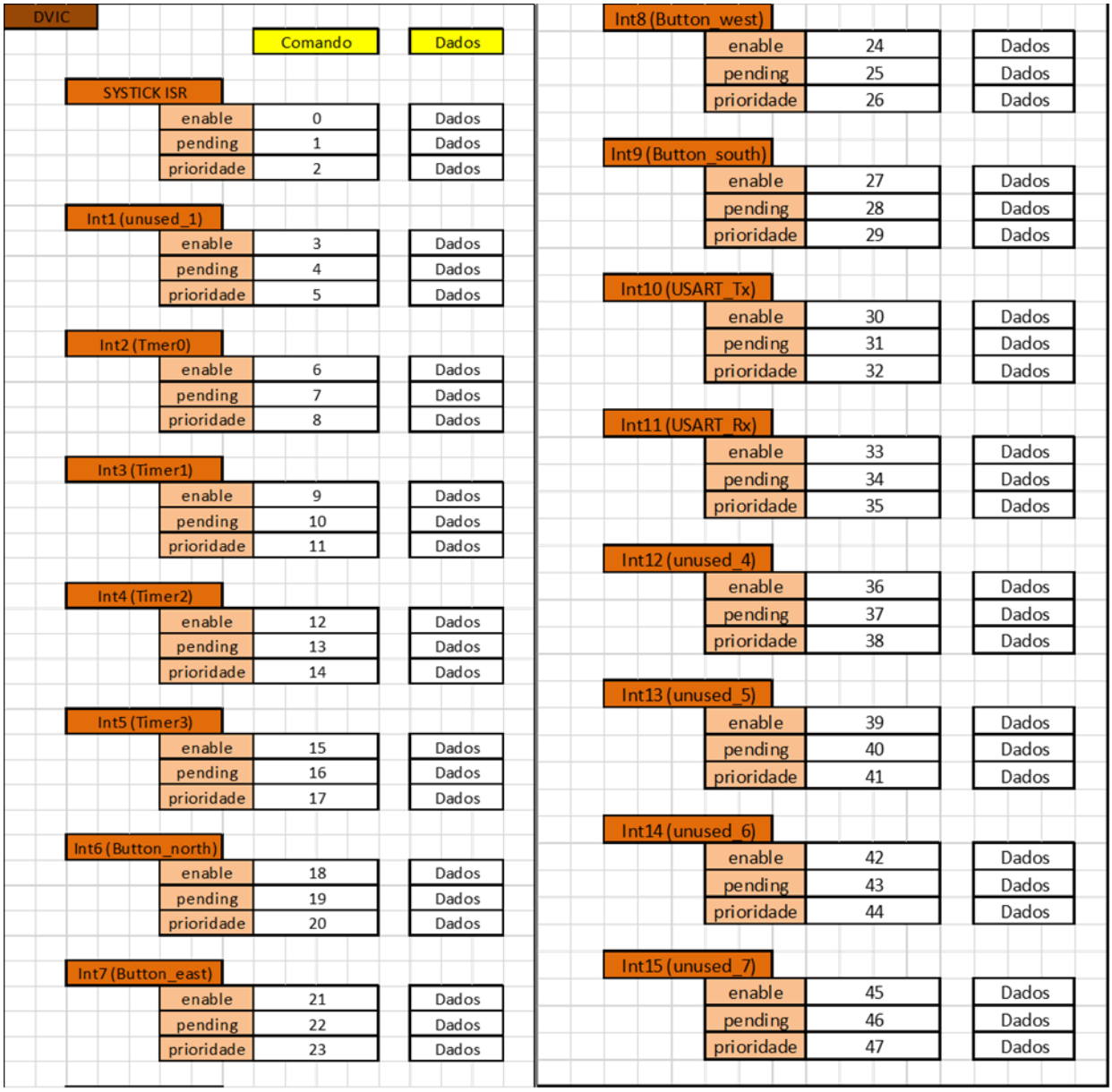


Figura 39 – Modelo de programação do DVIC

Capítulo 5

IMPLEMENTAÇÃO

Neste capítulo são apresentadas as fases da implementação. É explicado como foi implementado o módulo principal do SoC, o processador, a *hazard unit* e o barramento, sendo dada um explicação sobre como estão implementados. No fim, é apresentada a implementação dos vários periféricos integrados no SoC.

5.1. Implementação do Processador

5.1.1. Módulo hierarquicamente superior

Neste módulo estão presentes as instanciações do CPU, do barramento, dos periféricos e todas as variáveis de entrada e saída do SoC, como por exemplo, os pinos de entrada e de saída.

```
//////////////////////////////////////////
module D_Blaze_SoC(
   //inputs
    input                   sys_clk,
    input                   rst,
    input       [7:0]       in_pins,
    input       [4:0]       buttons,
    input                   Rx,
    //outputs
    output      [7:0]       out_pins,
    output                  Tx
);
//////////////////////////////////////////
```

Figura 40 – Entradas/Saídas do Módulo hierarquicamente superior

A placa de desenvolvimento utilizada é a FPGA da Xilinx XUPV5-LX110T. Nesta placa o sinal de *reset* por *hardware* tem lógica inversa. Para evitar que seja necessário mudar a variável de *reset*, foi criado um wire que toma o valor do *reset*, podendo ele estar negado para casos como a Virtex 5.

```
//////////////////////////////////////////
// Development Board specific variables
//////////////////////////////////////////
wire sys_rst;            // sys_rst signal for CPU
assign sys_rst = !rst;   // Virtex 5 sys_rst is always 1
//////////////////////////////////////////
```

Figura 41 – Sinal de *reset*

5.1.2. CPU

Como o processador desenvolvido encontra-se *pipelined* em cinco estágios, foi necessário criar um módulo para cada estágio, vários módulos para os *buffers* e um módulo para a *hazard unit*. As instanciações destes módulos são feitas no módulo CPU.

5.1.3. Estágio de Leitura de instrução

Neste estágio é onde se encontra a memória de instruções e é feita a atualização do valor do *PC*. A memória é um *ipcore* disponível na ferramenta da Xilinx. É uma memória ROM, uma vez que apenas é necessário ler os valores presentes na mesma, sendo estes as diferentes instruções que devem ser executadas.

A cada ciclo de relógio, o valor presente na posição de memória indicada pelo *PC* é passado para o *wire* IR, que por sua vez passa o seu valor, através de um *assign* para o *wire* instruction. O valor contido no *wire* instruction é a próxima instrução a ser executada, ou seja, tem que ser enviada para os estágios seguintes para ser descodificada e executada.

Na Figura 42 pode-se ver como o *PC* é atualizado a cada ciclo. Dependendo das instruções executadas, o valor do *PC* pode apenas passar a apontar para a próxima posição na memória, ou ter o seu valor alterado por uma instrução.

```
////////////////////////////////////////////////////////////
always@(posedge sys_clk)
    if(sys_rst)
        begin
            PC <= 32'b0;
        End
    else if(jump_signal == 2'b01 | jump_signal == 2'b11 )
            begin
                PC <= ((PC + jump_value) - 2'b10);
            End
    else if(jump_signal == 2'b10)
            begin
                PC <= jump_value;
            End
    else if(pc_restore != 0)
            begin
                PC <= pc_restore;
            end
    else
        begin
            PC <= PC + 1'b1;
        end
////////////////////////////////////////////////////////////
```

Figura 42 – Atualização do *PC*

Existem três condições para a atualização do valor do *PC*, sendo elas o *reset*, instruções de salto e execução normal. O caso de *reset* serve para o *PC* passar a apontar para o início da memória de instruções caso seja feito um *reset* na placa. Execução normal é quando não existem instruções que alterem o valor do *PC*, por isso ele apenas é incrementado para passar a apontar para a posição seguinte. Nas instruções de salto pode-se ter saltos relativos ou saltos absolutos, onde os primeiros são obtidos através da adição de um valor ao *PC*, enquanto nos outros o *PC* toma um valor dado por uma instrução.

5.1.4. Estágio de Descodificação de Instrução

Neste estágio é feita a decodificação das várias instruções presentes na memória de instruções. Também se resolvem neste estágio os diferentes *hazards* que ocorrem no *pipeline*, usando as técnicas de *stalls*, *bubbles* ou *data forwarding*. Neste estágio também se encontra o *register file* onde se realiza a leitura/escrita dos registos.

Como se pode ver na Figura 43, neste estágio é descodificado o *opcode* da instrução a executar, quais os registos a usar e valor imediado se necessário. Também se encontra a variavel *barrel_dir* que serve para informar ao módulo *barrel shifter* qual a direção do shift.

```
////////////////////////////////////////////////////////////////////////////////
// Decoding ////////////////////////////////////////////////////////////////////
assign opcode_aux = (ignore)                        ? 6'b0:
                                                      instruction[31:26];

assign Rd_aux   =   (ignore)                        ? 5'b0:
                                                      instruction[25:21];

assign Ra   =       (opcode == `RETI)               ? 5'd31:
                    (opcode == `PERACC)             ? 5'b0:
                    (opcode == `CALL)               ? 5'd31:
                    (ignore)                        ? 5'b0:
                    (opcode == `RET)                ? 5'd31:
                    (opcode == `PUSH)               ? 5'd31:
                    (opcode == `POP)                ? 5'd31:
                                                      instruction[20:16];

assign Rb   =   (ignore)                            ? 5'b0:
                (opcode == `PERACC)                 ? 5'b0:
                                                      instruction[15:11];

assign Imm = ((opcode == `ADDI | (…)) & !imm_signal) ?  {16'b0,instruction[15:0]}:
             ((opcode == `LWI | (…))  & !imm_signal) ?  {16'b0,instruction[15:0]}:
             ((opcode == `ADDI | (…)) & imm_signal)  ?  {imm_temp,instruction[15:0]}:
             ((opcode == `LWI | (…))  & imm_signal)  ?  {imm_temp,instruction[15:0]}:
                                                        32'b0;

assign barrel_dir = ((opcode == `BS) & (instruction[10] == 1'b1)) ? 1'b1 : 1'b0;
```

Figura 43 – *Decoding* de instrução

Na Figura 44 está apresentada a resolução dos saltos, tal como *hazards* envolvendo acessos à memória e mudanças do valor do apontador da pilha.

```
////////////////////////////////////////////////////////////////////
/// Branches
////////////////////////////////////////////////////////////////////
assign jump_signal = ((opcode == `BR | opcode == `BRI) & Ra == 5'b00000)          ? 2'b01:
                     ((opcode == `BR | opcode == `BRI) & Ra == 5'b01000)          ? 2'b10:
                     ((opcode == `BEQ) & (Rd == 5'b00000) & (RF_out_a == 2'b00)) ? 2'b11:
                     ((opcode == `BEQ) & (…) | (RF_out_a == 2'b01)))              ? 2'b11:
                     ((opcode == `BEQ) & (…) | (RF_out_a == 2'b00)))              ? 2'b11:
                     ((opcode == `BEQ) & (Rd == 5'b00100) & (RF_out_a == 2'b10)) ? 2'b11:
                     ((opcode == `BEQ) & (…) | (RF_out_a == 2'b01)))              ? 2'b11:
                     ((opcode == `BEQ) & (Rd == 5'b00010) & (RF_out_a == 2'b01)) ? 2'b11:
                     (Forward_ctrl[7:6] == 2'b01 | Forward_ctrl[7:6] == 2'b11)   ? 2'b01:
                     (opcode == `CALL)                                           ? 2'b10:
                     (Forward_ctrl[9:8] == 2'b01 | Forward_ctrl[9:8] == 2'b10)   ? 2'b01:
                                                                                   2'b00;

assign jump_value = (opcode == `BR | opcode == `BEQ)        ? RF_out_b:
                    (opcode == `BRI & !jump_side)           ? Imm:
                    (opcode == `BRI & jump_side)            ? (~Imm + 1'b1):
                    (opcode == `CALL)                       ? {6'b0, instruction[25:0]}:
                    (Forward_ctrl[9:8])                     ? 32'b1:
                                                              32'b0;

assign jump_side = ((opcode == `BRI) & (Rd[4] == 1'b1)) ? 1'b1 : 1'b0;
////////////////////////////////////////////////////////////////////
```

Figura 44 – Resolver saltos

Sempre que é feito um salto, existe a necessidade de ignorar as duas próximas instruções que estão no pipeline. Quando uma instrução avança para o próximo estágio, na transição ascendente do ciclo de relógio, ou seja, quando a instrução de salto está no estágio de *decode* já foi feito o *fetch* a uma outra instrução. Quando se verifica que o salto tem que ser executado, são enviados dois sinais, um de controlo e outro de dados, para o estágio de *fetch* para que o valor do *PC* seja atualizado, o que só acontece no ciclo de relógio seguinte, havendo assim uma outra instrução que foi anteriormente carregada para execução. Na Figura 45 pode-se ver que sempre que há um sinal de salto – *jump_signal* encontra-se a 1 – a variavel *ignore* passa a ter um valor diferente de zero para que quando estiver a ser feito o *decoding* da instrução, sejam introduzidas duas *bubbles* no *pipeline*.

```
////////////////////////////////////////////////////////////////////////
// Branches auiliary
////////////////////////////////////////////////////////////////////////
always@(posedge sys_clk)
begin
    if(sys_rst)
        begin
            ignore <= 3'b000;
        end
    else if (jump_signal)
        begin
            ignore <= 3'b010;
        end
    else if (opcode == `RET | opcode == `RETI)
        begin
            ignore <= 3'b100;
        end
    else if(ignore != 0)
        begin
            ignore <= ignore - 1'b1;
        end
end
////////////////////////////////////////////////////////////////////////
```

Figura 45 – Auxilio nos saltos

Neste processador existe o suporte para o prefixo IMM. Esta instrução faz com que seja possivel usar valores imediatos de 32 bits em instruções do tipo B. Como estas tem um campo para um valor imediato de 16 bits apenas, se for necessário carregar um valor de 32 bits estas instruções terão de ser precedidas por um prefixo IMM. O campo de 16 bits do prefixo IMM irá formar a parte mais significativa da palavra de 32 bits e o campo de 16 bits da instrução do tipo B irá formar a parte menos significativa da palavra de 32 bits.

Register file

A Figura 46 mostra a implementação do *register-file* onde são feitas as leituras/escritas dos diferentes registos. Tem-se que o R0 terá sempre o valor zero e o R31 está *hardwired* como o apontador da pilha. Caso se verifique um sinal de *reset* todo o *register file* fica com o valor zero, à excepção do registo 31 que toma o valor do topo da pilha.

```
always@(posedge sys_clk)
begin
    if(sys_rst)
        begin
            for (i = 0; i < 31; i = i + 1)
                 register_file[i] <= 0;
            register_file[31] <= 32'hFFF;    // Top of stack in RAM
        end
    if(wr_en)
        begin
            register_file[addr_write] <= data_ina;  // Write
        end
    if(stack_signal)
        begin
            register_file[31] <= data_in_stack; // Write stack
        end
end

assign data_outa = register_file[addra];
assign data_outb = register_file[addrb];
assign data_outd = register_file[addrd];
```

Figura 46 – Atualização do *register file*

5.1.5. Estágio de Execução

Neste estágio encontram-se o *barrel_shifter*, o *comparator*, o *multiplier* e a ALU, onde são feitas todas as operações lógicas e aritméticas. Na Figura 47 encontra-se o código referente aos *inputs* e que tipo de operação irá ser realizada na ALU.

```
////////////////////////////////////////////////////////////////////////////////
// ALU inputs
////////////////////////////////////////////////////////////////////////////////
assign ALU_input0 =      (opcode == `BR | opcode == `BEQ | opcode == `BS |
                          opcode == `BRI)                              ? 32'b0:

                         (opcode == `MOV | opcode == `MOVI)           ? 32'b0:
                         (opcode == `NOT)                             ? RF_out_d:
                         (opcode == `CMP | opcode == `MUL)            ? 32'b0:
                                                                        RF_out_a;


assign ALU_input1 =     (opcode == `BR   | (…))                       ? 32'b0:
                        (opcode == `NOT)                              ? 32'b0:
                        (opcode == `ADDI | (…))                       ? Imm:
                        (opcode == `SWI  | (…))                       ? Imm:
                        (opcode == `MOV)                              ? RF_out_a:
                        (opcode == `MOVI)                             ? Imm:
                        (opcode == `CMP | opcode == `MUL)             ? 32'b0:
                                                                        RF_out_b;

assign operation =  (opcode == `ADD | opcode == `ADDI)        ? `ALU_ADD:
                                  (…)
                    (opcode == `NOT)                          ? `ALU_LOGIC_NOT:
                                                                 ALU_NOP;
////////////////////////////////////////////////////////////////////////////////
```

Figura 47 – Atribuição dos valores para a ALU

A ALU contém dois *inputs* de dados e outro *input* com a operação que irá ser efectuada sobre os mesmos. Caso as instruções sejam acessos à memória, neste estágio é feita a adição do registo A com o registo B/*Imm* para calcular o endereço de memória a ser acedido.

As variáveis na Figura 48 servem para auxiliar nas instruções de leitura da memória. Não são enviadas pelo *buffer* que liga o estágio de execução ao estágio de acesso à memória, mas sim por uma ligação direta. Seguiu-se esta abordagem pois, para ler um posição da memória de dados é necessário fornecer o endereço, mas apenas no ciclo de relógio seguinte é que o valor é atualizado na variável de saída da memória.

```
////////////////////////////////////////////////////////////////////
// Load auxiliary
////////////////////////////////////////////////////////////////////
assign load_signal = (opcode == `LW | opcode == `LWI | opcode == `LWBI) ? 1'b1:
                                                                          1'b0;

assign ret_signal = (opcode == `RETI | opcode == `RET | opcode == `POP) ? 1'b1:
                                                                          1'b0;

assign forward_execute_output = (opcode == `RETI)        ? execute_output:
                                (opcode == `LW)          ? execute_output:
                                (opcode == `LWI)         ? execute_output:
                                (opcode == `LWBI)        ? execute_output:
                                (opcode == `RET)         ? execute_output:
                                (opcode == `POP)         ? execute_output:
                                                           32'b0;
////////////////////////////////////////////////////////////////////
```

Figura 48 – Auxilio em instruções de *Load*

Desta maneira evita-se perder dois ciclos para ler um valor da memória de dados. Existe um problema com esta implementação, que se verifica sempre que existe uma instrução de escrita para a memória de dados no ciclo anterior a uma instrução de leitura. Este problema é resolvido pela *hazard unit*.

5.1.6. Estágio de Acesso à Memória

Neste estágio são feitos todos os acessos à memória, sejam eles para escrita ou leitura. Na Figura 49 encontram-se as variáveis necessárias para aceder à memória de dados. A variável *wea_data_mem* pode ter o valor um ou zero e serve para dar permissão para escrita na memória. A variavel *addr_data_mem* diz qual o endereço se deve aceder, quer para leitura ou escrita. Por fim tem-se a variavel *dina_data_in* que possui o valor que irá ser escrito.

```
////////////////////////////////////////////////////////////////////////////////
// Data memory variables ///////////////////////////////////////////////////////
assign   wea_data_mem = (opcode == `SW | opcode == `SWI | opcode == `SWBI) ? 4'b1111:
                        (opcode == `CALL | opcode == `PUSH)                 ? 4'b1111:
                                                                              4'b0;

assign  addr_data_mem = (ret_signal)              ?       (forward_execute_output):

                        (opcode == `SW | opcode == `SWI | opcode == `SWBI) ?
                                                    execute_output[`INST_ADDR-1:0]:

                        (load_signal)             ?    forward_execute_output[11:0]:

                        (opcode == `CALL | opcode == `PUSH) ?
                                                    execute_output[`INST_ADDR-1:0]:
                                                    5'b0;

assign  dina_data_in = (opcode == `SW | opcode == `SWI | opcode == `SWBI) ? RF_out_d:
                       (opcode == `CALL | opcode == `PUSH)                ? RF_out_d:
                                                                            32'b0;

assign pc_restore = (opcode == `RET | opcode == `RETI) ? data_mem_out[`INST_ADDR-1:0]:
                                                         14'b0;
////////////////////////////////////////////////////////////////////////////////
```

Figura 49 – Variáveis de acesso à memória

5.1.7. Estágio de Atualização do *Register File*

Neste estágio é feita a atualização dos valores no *register file*. Não existe um módulo para este estágio, é apenas efectuada a ligação entre o *buffer* que conecta o estágio *memory access* ao estágio de *decode* para efectuar a actualização do *register file*.

```
////////////////////////////////////////////////////////////////////////////////
// Write Back values ///////////////////////////////////////////////////////////
assign RF_addr =    (opcode == `ADD  | (…) )                  ? Rd:
                    (opcode == `ADDI | (…) )                  ? Rd:
                    (opcode == `LW   | (…) )                  ? Rd:
                    (opcode == `BS)                           ? Rd:
                    (opcode == `MOV)                          ? Rd:
                    (opcode == `MOVI)                         ? Rd:
                    (opcode == `POP)                          ? Rd:
                    (opcode == `PERACC)                       ? Rd:
                                                                5'b0;

assign RF_data =   (opcode == `ADD  | (…) )             ? execute_output:
                   (opcode == `ADDI | (…) )             ? execute_output:
                   (opcode == `MOV)                     ? execute_output:
                   (opcode == `MOVI)                    ? execute_output:
                   (opcode == `LW   | (…) )             ? data_mem_out:
                   (opcode == `BS)                      ? execute_output:
                   (opcode == `CMP)                     ? {30'b0, CMP_output}:
                   (opcode == `POP)                     ? data_mem_out:
                   (opcode == `PERACC)                  ? execute_output:
                                                          32'b0;
////////////////////////////////////////////////////////////////////////////////
```

Figura 50 – Estágio *write back*

5.1.8. *Buffers*

São responsaveis por passar a informação de um estágio para o seguinte. Existem quatro *buffers*, em que a instanciação dos mesmos é igual para todos, havendo apenas diferenças nos valores que são passados para a variavel *in* e o tamanho das variaveis *in* e *out*.

Na Figura 51 tem-se o módulo do *buffer*. Este módulo comporta-se como um *flip-flop* normal, ou seja, a cada ciclo positivo de relógio o valor da entrada é passado para a saida do módulo.

```
////////////////////////////////
module D_Blaze_buf#(
    input sys_clk,
    input sys_rst,
    input      [`BUF#_SIZE-1:0] in,
    output reg [`BUF#_SIZE-1:0] out
     );

always@(posedge sys_clk)
begin
    if(sys_rst)
        out <= 0;
    else
        out <= in;
end

endmodule
////////////////////////////////
```

Figura 51 – *Buffer #*

5.2. Implementação da *Hazard unit*

Neste módulo é necessário saber quais os registos que estão a ser utilizados em cada estágio, à exceção do estágio *fetch*, para se poder detetar os *hazards*. É no estágio de *decode* que é feito o acesso ao *register file*, logo é onde se pode verificar se existem *hazards* de dados. É verificado se uma instrução requer o valor de um registo que ainda não foi atualizado pelas instruções anteriores, pois tipicamente numa implementação *pipeline* o *register file* só é atualizado no estágio *write back*. Para resolver estes *hazards* recorre-se a *data forwarding*, ou seja, o valor correto é encaminhado dos estágios seguintes para o estágio de *decode*. Existe um caso em que se a instrução necessitar de um valor obtido de um *load*, é necessário fazer *stall* ao *pipeline*, pois o valor só será lido da memória no estágio *memory access*.

De seguida estão apresentados alguns casos em que ocorrem *hazards* bem como o SoC procede para os resolver. O *datapath* utilizado é uma simplificação do *datapath* do sistema final.

***Hazard* de dados**

Este provavelmente é o *hazard* mais comum num *pipeline*. De seguida vai ser demonstrado como o *pipeline* deve reagir a casos onde existem este tipo de *hazards*.

Na Figura 52 está um trecho de código simples. Como se pode ver existe um *hazard* entra a instrução LW e ADD, pois esta última necessita de um valor que está a ser usado pela instrução anterior.

```
0x0017    MOVI   r3,0x03
0x0018    MOVI   r2,0x05
0x0019    MOVI   r5,0x07
0x0020    LW     r1,r2,r3
0x0021    ADD    r4,r1,r5
```

Figura 52 – Código com *hazard* de dados

Na Figura 53 pode-se ver o *pipeline* quando todas as instruções se encontram em execução. Nesta altura a instrução ADD ainda se encontra no estágio *fetch*, logo ainda não necessita do registo R1.

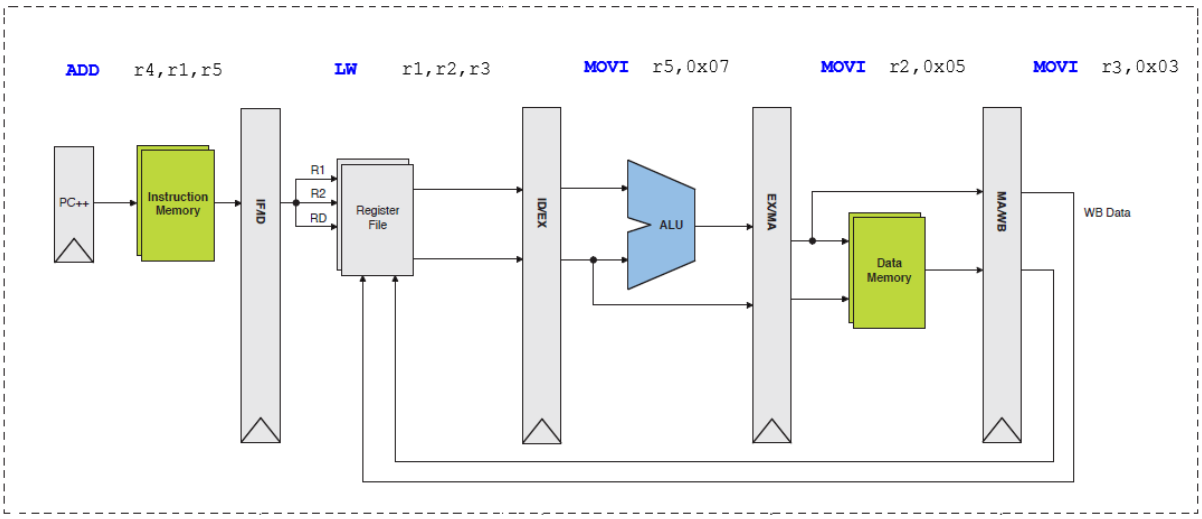


Figura 53 – *Datapath* com *hazard* de dados (1)

O problema surge agora quando a instrução ADD se encontra no estágio *decode*, pois agora necessita do registo R1, que ainda não foi atualizado porque a instrução de LW ainda não executou. Na Figura 54 encontra-se o *pipeline* na situação mencionada.

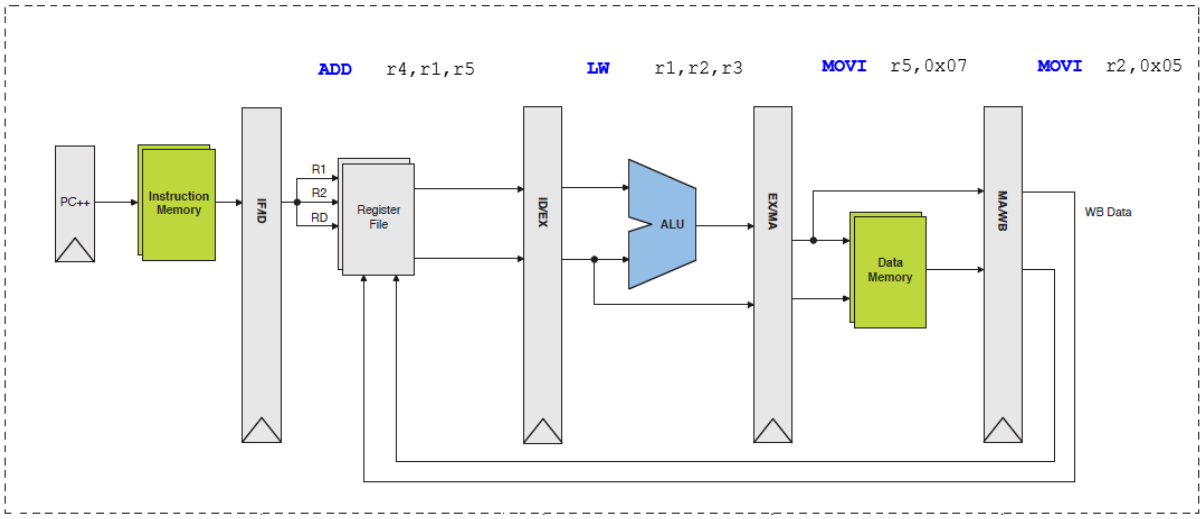


Figura 54 – *Datapath* com *hazard* de dados (2)

A resolução deste *hazard* passa por fazer o *stall* ao *pipeline* e o *data forward*, como se pode ver na Figura 55. Para este caso basta esperar um ciclo de relógio para a instrução de LW leia o valor que o registo R1 deve tomar e fazer o *data forward* para evitar fazer o *stall* mais um ciclo de relógio.

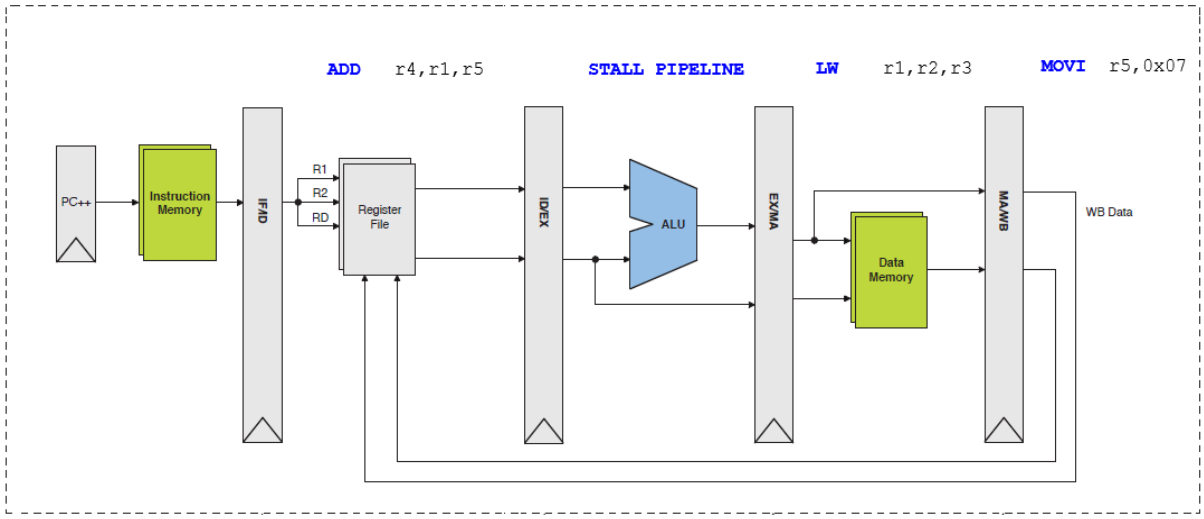


Figura 55 – *Datapath* com *hazard* de dados(3)

***Hazard* de controlo**

Este *hazard* verifica-se sempre que existem saltos, sejam eles saltos condicionais ou absolutos. No trecho de código apresentado na Figura 56 existe um *hazard* de controlo na posição 0x0016, a instrução de CALL. Anteriormente foi dito que sempre que há um salto é necessário ignorar as duas próximas instruções.

```
0x0015      MOVI   r3,0x03
0x0016      CALL   0x0022
0x0017      MOVI   r5,0x07
0x0018      MOVI   r3,0x03
0x0019      ADD    r4,r1,r5
0x0020      OR     r6,r4,r3
0x0021      AND    r8,r1,r2
0x0022      XOR    r9,r1,r6
```

Figura 56 – Código com *hazard* de controlo

A Figura 57 apresenta o momento em que a instrução CALL é obtida da memória de instruções. Os saltos são apenas tratados no estágio de *decode*, onde a *hazard unit* irá detetar o *hazard* de controlo.

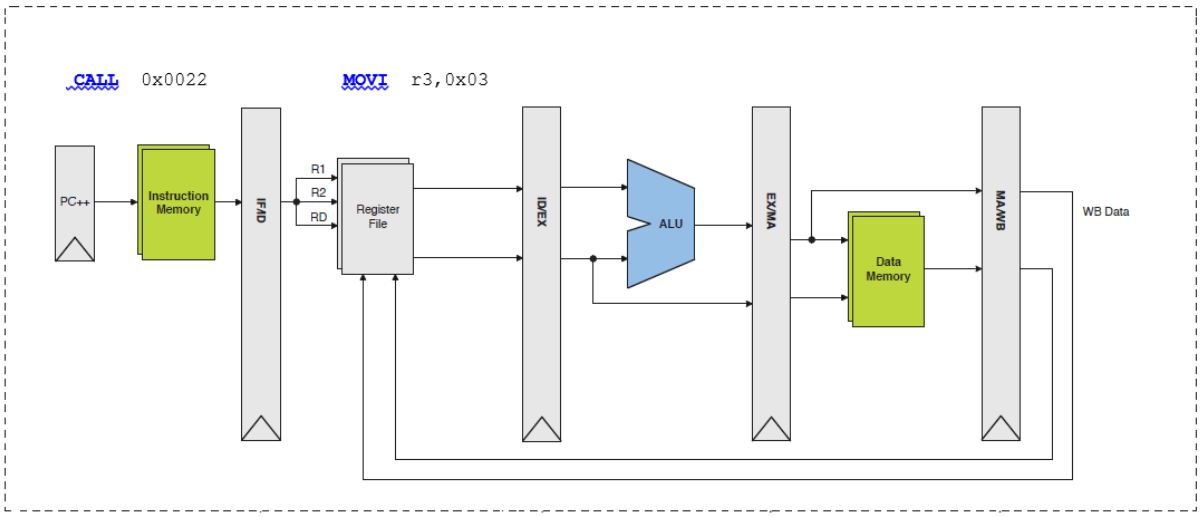


Figura 57 – *Datapath* com *hazard* de controlo (1)

Na Figura 58 a *hazard unit* já entrou em ação ao inserir duas *bubbles* após o salto. Desta maneira o processador descarta as duas instruções que se encontram nas posições 0x0017 e 0x0018.

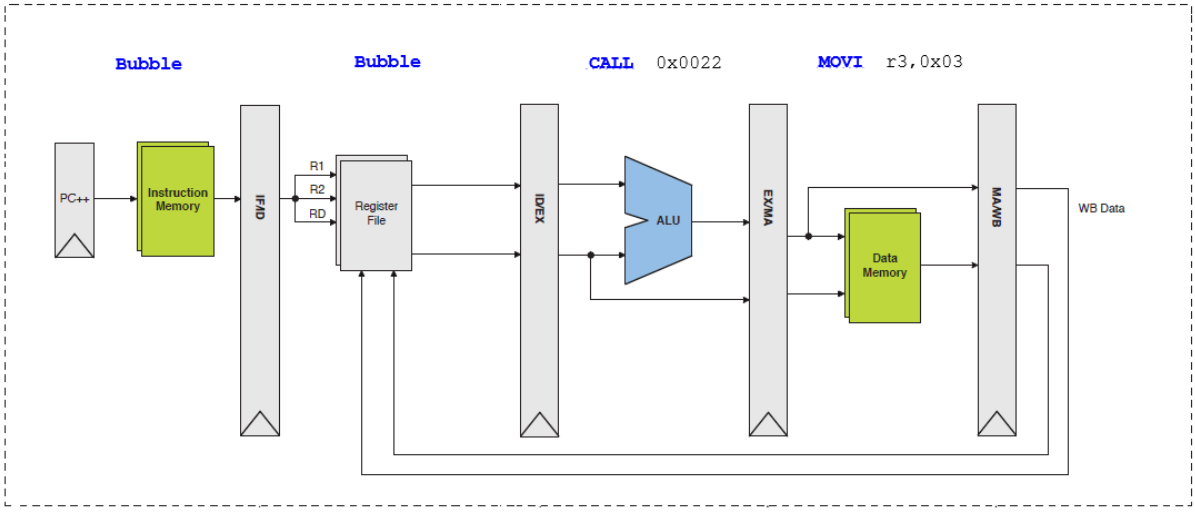


Figura 58 – *Datapath* com *hazard* de controlo (2)

Na Figura 59 verifica-se que a próxima instrução no *pipeline* é a instrução da posição para a qual foi feito o CALL.

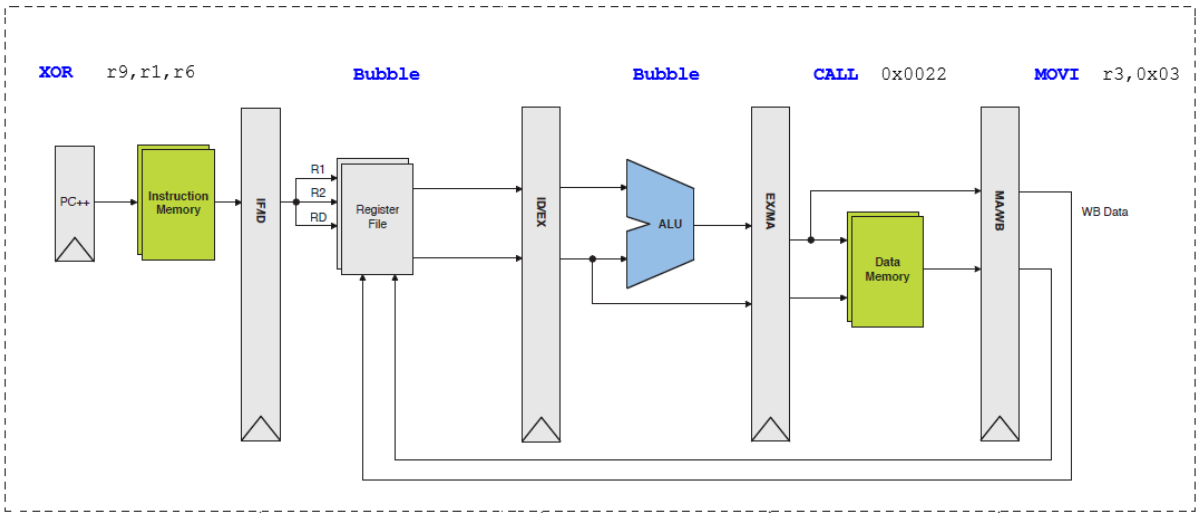


Figura 59 – *Datapath* com *hazard* de controlo (3)

***Hazard* estrutural**

Este *hazard*, como já referido anteriormente, acontece quando existe concorrência no acesso aos recursos disponíveis. Nesta implementação não acontecem este tipo de *hazards*, uma vez que foram resolvidos no momento de *design* do processador.

5.3. Implementação do barramento

O barramento é constituído por dois módulos, um *encoder* e um *decoder*. O *encoder* trata de recolher a informação enviada pelo CPU e codifica-la. O *decoder* presente em todos os periféricos usa essa informação codificada e descodifica-a para ser utilizada para as diferentes operações dentro dos periféricos.

5.3.1. *Encoder*

O *encoder* é responsável por verificar qual o periférico que se quer aceder. É feita a codificação da palavra de controlo e de dados para posterior uso dentro dos periféricos. Na Figura 60 é possivel ver a codificação da palavra de controlo, da palavra de dados e a ativação do sinal *select* correto conforme o perifério que está a ser acedido.

```
////////////////////////////////////////////////////////////////////////////////
// Local variables ////////////////////////////////////////////////////////////
wire    [4:0]       peripheral;                     // peripheral
wire                peripheral_access_aux;          // Access to a peripheral

////////////////////////////////////////////////////////////////////////////////
assign peripheral_access_aux = peripheral_access[26];
assign peripheral = peripheral_access[20:16];

////////////////////////////////////////////////////////////////////////////////
assign peripheral_select[0] = (peripheral_access_aux & peripheral == 0);
assign peripheral_select[1] = (peripheral_access_aux & peripheral == 1);
assign peripheral_select[2] = (peripheral_access_aux & peripheral == 2);
                                (…)
assign peripheral_select[30] = (peripheral_access_aux & peripheral == 30);
assign peripheral_select[31] = (peripheral_access_aux & peripheral == 31);

////////////////////////////////////////////////////////////////////////////////

assign peripheral_ctrl= {6'b0,peripheral_access[25:21]/*Register*/,
                         peripheral_access[20:16]/*peripheral*/,
                         peripheral_access[15:0]/*Control word*/};

assign peripheral_data = peripheral_reg;
////////////////////////////////////////////////////////////////////////////////
```

Figura 60 – *Encoder*

5.3.2. *Decoder*

Existe um *decoder* em todos os periféricos, pois é necessário descodificar a informação enviada pelo *encoder* do barramento. Na Figura 61 pode-se ver a parte do módulo que trata da descodificação. Como isto é um barramento simples, a quantidade de sinais que necessitam de ser interpretados são reduzidos.

```
////////////////////////////////////////////////
// Decoder
////////////////////////////////////////////////
assign command = peripheral_ctrl[14:0];
assign data = peripheral_data;
assign read_write = peripheral_ctrl[15];
////////////////////////////////////////////////
```

Figura 61 – *Decoder*

5.4. Implementação dos periféricos

De seguida será explicado como cada periférico foi implementado, bem como as partes mais importantes de cada um.

5.4.1. Porto I/O

Este módulo permite aceder aos pinos de entrada, bem como os *leds* presentes na placa de desenvolvimento. Tal como se pode ver na Figura 62 este periférico apenas possui duas opções, que são a leitura do *switch* e a escrita nos pinos de *output* onde estão mapeados os *leds*, ambos de 8 bits.

```
////////////////////////////////////////////////////////////////////
assign in_pins_value =  (sys_rst)                          ? 8'b0:
                        (peripheral_select & command == 4'd0 &
                         read_write)                       ? in_pins:
                                                             8'b0;

assign out_pins_aux =   (sys_rst)                          ?   8'b0:
                        (peripheral_select & command == 4'd0 &
                         !read_write)                    ? data[7:0]:
                                                           out_pins;
////////////////////////////////////////////////////////////////////
```

Figura 62 – Parte do módulo I/O

Neste módulo é verificado se os cincos botões que originam interrupções são pressionados. Na Figura 63 é possível ver que a variável *int_req* tomará diferentes valores em função do botão pressionado.

```
////////////////////////////////////////////////////////////////////
assign int_req =  (sys_rst)                                  ?  5'b00000:
                  (button_center & ~button_center_aux)       ?  5'b10000:
                  (button_east & ~button_east_aux)           ?  5'b00010:
                  (button_north & ~button_north_aux)         ?  5'b00001:
                  (button_south & ~button_south_aux)         ?  5'b01000:
                  (button_west & ~button_west_aux)           ?  5'b00100:
                                                                5'b00000;
////////////////////////////////////////////////////////////////////
```

Figura 63 – Interrupções I/O

5.4.1.1. Programação do I/O

De seguida, na Figura 64, são apresentadas as duas funções usadas para programar o periférico.

```
/*****************************************/
readBus(num,parioId,ler);
writeBus(num,parioId,escrever);

/*****************************************/
```

Figura 64 – Trecho de código do I/O

Neste código encontra-se um ciclo infinito onde é lido o valor de entrada dos *switchs* e o mesmo é refletido nos *leds*. Como se pode ver, para interagir com este periférico basta fazer uso de duas instruções simples.

5.4.2. DVIC

Este periférico é responsável por gerir as várias interrupções despoletadas por todos os outros periféricos. Deve cumprir alguns requisitos básicos, tais como:

- Caso ocorra uma interrupção de maior prioridade, o processador deverá pausar a de menos prioridade e atender a de prioridade superior;
- Deve permitir a alteração das prioridades das interrupções dinamicamente.
- Despoletar interrupções por *hardware*/*software*

5.4.2.1. Registos de estado da interrupção

Os registos criados para alojar toda a informação relativa a cada interrupção podem ser observados na Figura 65. Foi explicado anteriormente que é necessário guardar a prioridade e o estado de cada interrupção. O formato de cada um dos registos, também foi anteriormente definido.

```
reg [7:0]     ISR_zero_SYSTICK;          //registo individual SYSTICK
reg [7:0]     ISR_one_PAR_IO;            //registo individual I/O Center
reg [7:0]     ISR_two_timer0;            //registo individual timer0
reg [7:0]     ISR_three_timer1;          //registo individual timer1
reg [7:0]     ISR_four_timer2;           //registo individual timer2
reg [7:0]     ISR_five_timer3;           //registo individual timer3
reg [7:0]     ISR_six_PAR_IO;            //registo individual I/O North
reg [7:0]     ISR_seven_PAR_IO;          //registo individual I/O East
reg [7:0]     ISR_eight_PAR_IO;          //registo individual I/O West
reg [7:0]     ISR_nine_PAR_IO;           //registo individual I/O South
reg [7:0]     ISR_ten_USART_Tx;          //registo individual UART_Tx
reg [7:0]     ISR_eleven_USART_Rx;       //registo individual UART_Rx
reg [7:0]     ISR_twelve_unused_4;       //registo individual unused
reg [7:0]     ISR_thirteen_unused_5;     //registo individual unused
reg [7:0]     ISR_fourteen_unused_6;     //registo individual unused
reg [7:0]     ISR_fifteen_unused_7;      //registo individual unused
```

Figura 65 – Registos para cada uma das interrupções

A atualização efetuada para estes registos é feita usando um *multiplexer*. Na Figura 66 está representado como deverá ser definido o *multiplexer* para cada uma das entradas.

```
assign ISR_zero_SYSTICK_next =
            (sys_rst)                                 ? 8'b0
            (peripheral_select & (command == 15'd0))  ? {data[0],ISR_zero_SYSTICK[6:0]}:
            (peripheral_select & (command == 15'd1))  ? ISR_zero_SYSTICK[7:6],data[0],ISR_zero_SYSTICK[4:0]}:
            (peripheral_select & (command == 15'd2))  ? {ISR_zero_SYSTICK[7:5],data[4:0]}:
            (int_req_ext[0])                          ? {ISR_zero_SYSTICK[7:6],1'b1,ISR_zero_SYSTICK[4:0]}:
            ((max_priority_irq==4'b0000) & int_ack)   ? {ISR_zero_SYSTICK[7],2'b10,ISR_zero_SYSTICK[4:0]}:
            (reti & ~master_reti & (…))               ? {ISR_zero_SYSTICK[7],1'b0,ISR_zero_SYSTICK[5:0]}:
                                                        ISR_zero_SYSTICK;
```

Figura 66 – *Multiplexer* responsável pela atualização do registo de uma interrupção

A atualização de cada registo é efetuada sempre que acontecem as seguintes situações:

- *Peripheral_select & (command == 15'd0)* – este cenário refere-se ao momento que o utilizador ativa ou desativa uma interrupção;
- *Peripheral_select & (command == 15'd1)* – este cenário refere-se ao momento que o utilizador provoca uma interrupção por *software*;
- *Peripheral_select & (command == 15'd2)* – este cenário refere-se ao momento que o utilizador define uma prioridade para uma interrupção;
- *int_req_ext[0]* – este cenário refere-se ao momento em que uma fonte externa de interrupção provoca uma interrupção;
- *(max_priority_irq==4'b0000 & int_ack)* - este cenário refere-se ao momento em que uma interrupção entra em execução e passa do estado *ENABLE & PENDENT* para *ENABLE & ACTIVE*;

- *reti & ~master_reti (irq_stack[irq_stack_pointer-1]==4'b0000)* - este cenário refere-se ao momento em que uma interrupção termina de executar e precisa de alterar o estado de *ENABLE & ACTIVE* apenas para *ENABLE*.

Será desta forma que a atualização de cada um dos registos será implementada.

5.4.2.2. Pilha Dedicada

Neste subcapítulo irá ser descrita de que forma a pilha interna ao *DVIC* está implementada. Primeiro começa-se por definir a pilha como uma pequena memória *LIFO* com 16 posições e um registo simples que será o apontador para a pilha.

```
reg     [3:0]           irq_stack[15:0];
reg     [3:0]           irq_stack_pointer;
```

Figura 67 – Pilha e apontador para a mesma.

Para saber o momento em que se deve guardar valores na pilha, foi usado um pequeno *multiplexer*. Quando uma interrupção for colocada a correr, o sinal *int_dvic* ficará com o valor um e *irq_stack_next* tomará o valor de *max_priority_irq* que contém o endereço da interrupção gerada. Caso contrário o seu conteúdo é inalterado, indexado pelo apontador da pilha, como se pode ver na Figura 68.

```
assign irq_stack_next = (int_dvic)  ?  max_priority_irq:
                                       irq_stack[irq_stack_pointer];
```

Figura 68 – *Multiplexer* responsável por guardar o conteúdo da interrupção a correr

Agora, basta mostrar de que forma o apontador da pilha é atualizado, ou seja, os momentos em que é incrementado e decrementado.

```
if(int_ack)
    irq_stack_pointer <= irq_stack_pointer + 1;
else if(reti & master_reti_next)
    irq_stack_pointer <= irq_stack_pointer - 1;
```

Figura 69 – Actualização da pilha

Quando no módulo *DVIC* se recebe um sinal de *int_ack*, que é o mesmo que dizer quando o processador diz ao *DVIC* que uma interrupção entrou em execução, o apontador da pilha deverá ser incrementado. Para decrementar o valor do apontador da pilha, o módulo deverá

receber um sinal de *RETI*, que é quando o processador diz ao módulo que a interrupção libertou o processador.

5.4.2.3. Árvore binária de decisão

Este componente foi dividido em quatro níveis, em que o primeiro nível, guarda os resultados da comparação dos dezasseis registos de estado das interrupções, o segundo nível guarda os resultados das comparações no nível um, o terceiro nível guarda os resultados das comparações do nível dois e o último nível apenas guarda o endereço da interrupção mais prioritária, resultado da comparação no nível três. A Figura 70 apresenta a declaração dos quatro níveis.

```
wire    [11:0]      first_level[7:0];
wire    [11:0]      second_level[3:0];
wire    [11:0]      third_level[1:0];
wire    [3:0]       fourth_level;
```

Figura 70 – Declaração dos níveis da árvore de decisão

De seguida irá ser apresentada a codificação do primeiro nível, que está presente na Figura 71. Neste nível, são comparados os registos de estado das interrupções dois a dois, e é guardada na ligação *first_level[x]* tanto o seu endereço como o seu conteúdo, para não se perder informação no seguimento da árvore binária de decisão.

```
//primeiro nivel
assign first_level[0] = ((…) >= (…))  ? {4'b0000,ISR_zero_SYSTICK}:
                        ((…) <  (…))  ? {4'b0001,ISR_one_PAR_IO}:
                                        12'b0;

assign first_level[1] = ((…) >= (…))  ? {4'b0010,ISR_two_timer0}:
                        ((…) <  (…))  ? {4'b0011,ISR_three_timer1}:
                                        12'b0;

                              (…)

assign first_level[6] = ((…) >= (…))  ? {4'b1100,ISR_twelve_unused_4}:
                        ((…) <  (…))  ? {4'b1101,ISR_thirteen_unused_5}:
                                        12'b0;

assign first_level[7] = ((…) >= (…))  ? {4'b1110,ISR_fourteen_unused_6}:
                        ((…) <  (…))  ? {4'b1111,ISR_fifteen_unused_7}:
                                        12'b0;
```

Figura 71 – Codificação do primeiro nível

No segundo nível são comparados as entradas de ligação do *first_level* sendo guardado a sua informação no *second_level*[x].

```
//segundo nivel
assign second_level[0] = ((…) >= (…))    ?   {first_level[0]}:
                         ((…) <  (…))    ?   {first_level[1]}:
                                             12'b0;

assign second_level[1] = ((…) >= (…))    ?   {first_level[2]}:
                         ((…) <  (…))    ?   {first_level[3]}:
                                             12'b0;

assign second_level[2] = ((…) >= (…))    ?   {first_level[4]}:
                         ((…) <  (…))    ?   {first_level[5]}:
                                             12'b0;

assign second_level[3] = ((…) >= (…))    ?   {first_level[6]}:
                         ((…) <  (…))    ?   {first_level[7]}:
                                             12'b0;
```

Figura 72 – Codificação do segundo nível

À semelhança do nível anterior, agora serão comparadas as entradas da ligação *second_level*, e também será guardado o endereço e o conteúdo na ligação *third_level[x]*.

```
//terceiro nivel
assign third_level[0] = ((…) >= (…)) ?   {second_level[0]}:
                        ((…) <  (…)) ?   {second_level[1]}:
                                         12'b0;

assign third_level[1] = ((…) >= (…)) ?   {second_level[2]}:
                        ((…) <  (…)) ?   {second_level[3]}:
                                         12'b0;
```

Figura 73 – Codificação do terceiro nível

Por fim, no último nível será guardado apenas o endereço da interrupção mais prioritária. Este resultado é obtido através da comparação da ligação *thrid_level[0]* com o *third_level[1]*.

```
//quarto nivel
assign fourth_level =  ((…) >= (…))  ? {third_level[0][11:8]}:
                       ((…) <  (…))  ? {third_level[1][11:8]}:
                                         4'b0;
```

Figura 74 – Codificação do último nível da árvore

Concluindo, é desta forma que este componente está estruturado. Como é um componente arquitetado com lógica puramente combinacional tem-se a cada momento o endereço da interrupção com maior prioridade.

5.4.2.4. Programação do *DVIC*

Irá ser demonstrado como o programador deverá proceder para programar o SoC, agora dotado de um controlador de interrupções com base em prioridades.

Para cada interrupção pode-se definir a interrupção como habilitada ou desabilitada, a prioridade da mesma e também é possível provocar uma interrupção por *software* ativando o respetivo *bit pending*. Desta forma, é possível programar as interrupções, tanto por *hardware* como por *software*. Para explicar de que forma é feita a programação do *DVIC* serão dados exemplos práticos de forma a facilitar a compreensão do mesmo. A Figura 75 apresenta a programação de duas IRQs do *timer* 0 e do *timer* 2. Neste exemplo são habilitadas as interrupções e definidas as suas prioridades.

```
/*******************************************/
/******** Configurar DVIC ******************/
output=0x01;
writeBus(output,dvicID,IRQ_TIMER0_enable);
output=0x03;
writeBus(output,dvicID,IRQ_TIMER0_prio);

output=0x01;
writeBus(output,dvicID,IRQ_TIMER2_enable);
output=0x05;
writeBus(output,dvicID,IRQ_TIMER2_prio);
/*******************************************/
```

Figura 75 – Código DVIC: configurar duas interrupções

Após programar o DVIC é necessário implementar as rotinas de atendimento para as duas interrupções. A Figura 76 apresenta o que é feito dentro de cada ISR. Na ISR do *timer* 0 é atribuído o valor 0xAA a uma variável que será usada depois para escrever no porto I/O. Na ISR do *timer* 2 o valor passado para a variável que vai ser escrita no porto I/O é o 0x55.

```
/*******************************************/
/************ Interrupções *****************/
void TIMER0_ISR_vect(void)
{
    out = 0xAA;
    writeBus(out,parioId,escrever);
}

void TIMER2_ISR_vect(void)
{
    out = 0x55;
    writeBus(out,parioId,escrever);
}
/*******************************************/
```

Figura 76 – Código DVIC: corpo das interrupções

No trecho de código, apresentado na Figura 77, estão na mesma a ser programadas duas interrupções, mas está a ser despoletada a interrupção do *timer* 0 por *software* através do *bit pending*. É na última linha que se encontra a função que despoleta a interrupção por *software*.

```
/*******************************************/
/******** Configurar DVIC ******************/
output=0x01;
writeBus(output,dvicID,IRQ_TIMER0_enable);
output=0x03;
writeBus(output,dvicID,IRQ_TIMER0_prio);

output=0x01;
writeBus(output,dvicID,IRQ_TIMER2_enable);
output=0x05;
writeBus(output,dvicID,IRQ_TIMER2_prio);

output=0x01;
writeBus(output, dvicID, IRQ_TIMER0_pending);
/*******************************************/
```

Figura 77 – Trecho de código do DVIC

5.4.3. *Timer*

Para implementar o *timer* recorreu-se a três blocos. O primeiro bloco, *Timer*.v, é responsável por estabelecer a comunicação com o módulo *D_Blaze_SoC*, guardando em registos os valores iniciais de contagem e a configuração de cada *timer*. Estes valores são depois passados ao módulo inferior, *timer*_block.v, que é responsável por gerir o relógio para cada um dos *timers*, de acordo com a configuração definida para cada um e por determinar quais das *flags* de *overflow* estão ativas. É neste módulo que se deteta as transições do pino de contagem externa. Por fim tem-se quatro módulos base, timer0, timer1, timer2 e timer3, que são responsáveis por

incrementar um registo de contagem sempre que recebem um sinal de relógio do módulo superior.

5.4.3.1. Módulo *Timer*

Como foi supracitado, o módulo *timer*.v é responsável por receber as tramas do barramento e guardar estes valores nos registos corretos recorrendo a um conjunto de *assigns* que determinam para qual dos *timers*, 0, 1, 2 ou 3 é a configuração presente no barramento, para depois passar ao módulo *timer*_block.v. Se a nova configuração ou o novo valor de contagem é para o *timer* 0, todos os restantes *timers* vão manter o antigo valor de configuração e o *timer* 0 é atualizado com a nova configuração. O mesmo acontece para os restantes *timers*. A Figura 78 apresenta o código responsável por produzir *hardware* para realizar este processo.

```
////////////////////////////////////////////////////////////////////////////////
// TIMCNT
assign count_value0_last   = (peripheral_select & command == 15'd0)      ?  data[7:0]:
                                                                            count_value0;

assign count_value1_last   = (peripheral_select & command == 15'd3)      ?  data[7:0]:
                                                                            count_value1;

assign count_value2_last   = (peripheral_select & command == 15'd6)      ?  data[7:0]:
                                                                            count_value2;

assign count_value3_last   = (peripheral_select & command == 15'd9)      ?  data[7:0]:
                                                                            count_value3;
// TIMRLD
assign reload_value0_last  = (peripheral_select & command == 15'd1)      ?  data[7:0]:
                                                                            reload_value0;

assign reload_value1_last  = (peripheral_select & command == 15'd4)      ?  data[7:0]:
                                                                            reload_value1;

assign reload_value2_last  = (peripheral_select & command == 15'd7)      ?  data[7:0]:
                                                                            reload_value2;

assign reload_value3_last  = (peripheral_select & command == 15'd10)     ?  data[7:0]:
                                                                            reload_value3;
// TIMCONFB
assign config_value_lastb  = (peripheral_select & command == 15'd2)      ?  data[7:0]:
                                                                            config_valueb;

assign config_value_lastb1 = (peripheral_select & command == 15'd5)      ?  data[7:0]:
                                                                            config_valueb;

assign config_value_lastb2 = (peripheral_select & command == 15'd8)      ?  data[7:0]:
                                                                            config_valueb;

assign config_value_lastb3 = (peripheral_select & command == 15'd11)     ?  data[7:0]:
                                                                            config_valueb;

// TIMCONFA
assign config_value_last = (peripheral_select) & (command == 15'd12)     ?  data[7:0]:
                                                                            config_value;

assign read_value =      (command == 15'd0 & read_write)                  ? 4'b0001:
                         (command == 15'd3 & read_write)                  ? 4'b0010:
                         (command == 15'd6 & read_write)                  ? 4'b0100:
                         (command == 15'd9 & read_write)                  ? 4'b1000:
                                                                            4'd0;
////////////////////////////////////////////////////////////////////////////////
```

Figura 78 – Descodificação dos valores de configuração dos *timers*

5.4.3.2. Módulo *Timer_block*

Estes valores são então passados ao módulo inferior, responsável por interpretá-los e gerar, a partir deles, o relógio correto para cada um dos *timers*, bem como as *flags* de *overflow* que devem estar ativas. Em primeiro lugar o módulo *timer_block.v* deteta quando ocorre uma transição positiva, negativa ou ambas no pino configurado como contador externo. Na Figura 79 é possível ver como é feito através dos *wires pos_edge*, *neg_edge* e *both_edge*. Baseando-se no valor atual do pino, no registo *state*, no registo *prev_state*, consegue-se determinar as transições no pino. Estes sinais de transição são depois passados aos *timers* como sinais de relógio, se os *timers* estiverem configurados como contadores. Caso estes estejam configurados como temporizadores, o sinal de relógio será sempre o relógio interno do *D_Blaze_SoC* ou a *flag* de *overflow* de um dos outros *timers*, no caso em que se queira um *timer* superior a 8 bits. Este processo de determinação do sinal de relógio para cada *timer* através do valor da configuração definida é ilustrado na Figura 79.

```
////////////////////////////////////////////////////////////////////////////////
assign pos_edge = (state & ~prev_state)                 ? 1'b1 : 1'b0;

assign neg_edge = (~state & prev_state)                 ? 1'b1 : 1'b0;

assign both_edge    =   (pos_edge & ~prev_state)        ? 1'b1:
                        (neg_edge & prev_state)         ? 1'b1:
                                                          1'b0;
////////////////////////////////////////////////////////////////////////////////
////// sys_clk signal ////////////////////////////////////////////////////////
assign connect0 = (~configurationb[1] & ~configurationb[0])                 ? sys_clk:
                  (~configurationb[1] & configurationb[0])                  ? pos_edge:
                  (configurationb[1] & ~configurationb[0])                  ? neg_edge:
                  (configurationb[1] & configurationb[0])                   ? both_edge:
                                                                              1'b0;

assign connect1 = (configuration[4] | configuration[5] | configuration[6])  ? ovflag1:
                  (~configurationb1[1] & ~configurationb1[0])               ? sys_clk:
                  (~configurationb1[1] & configurationb1[0])                ? pos_edge:
                  (configurationb1[1] & ~configurationb1[0])                ? neg_edge:
                  (configurationb1[1] & configurationb1[0])                 ? both_edge:
                                                                              1'b0;

assign connect2 = ((configuration[4] & configuration[5]) | configuration[6]) ? ovflag2:
                  (~configurationb2[1] & ~configurationb2[0])               ? sys_clk:
                  (~configurationb2[1] & configurationb2[0])                ? pos_edge:
                  (configurationb2[1] & ~configurationb2[0])                ? neg_edge:
                  (configurationb2[1] & configurationb2[0])                 ? both_edge:
                                                                              1'b0;

assign connect3 = (configuration[5] | configuration[6])                     ? ovflag3:
                  (~configurationb3[1] & ~configurationb3[0])               ? sys_clk:
                  (~configurationb3[1] & configurationb3[0])                ? pos_edge:
                  (configurationb3[1] & ~configurationb3[0])                ? neg_edge:
                  (configurationb3[1] & configurationb3[0])                 ? both_edge:
                                                                              1'b0;
////////////////////////////////////////////////////////////////////////////////
```

Figura 79 – *Multiplexers* para o sinal de relógio dos *timers*

Este mesmo módulo tem a responsabilidade de determinar quais a *flags* de *overflow* que devem despoletar uma interrupção, através do valor da configuração passada aos *timers*. O valor de parte da configuração dos *timers* é passado para o *wire mux_out*, responsável por decidir qual das *flags* vão estar ligadas ao DVIC, por forma a gerar uma interrupção.

Existem dois modos de temporização/contagem, a crescente e a decrescente. Se os dois modos fossem implementados nos módulos base dos *timers*, seria necessário implementar dois mecanismos de contagem por *timer*, um para contagem crescente e o outro para contagem decrescente. Para reduzir a lógica necessária e manter o módulo base do *timer* inalterado, ou seja, o módulo base apenas incrementa um registo a cada *tick* de relógio, sem fazer qualquer tipo de processamento relativo ao sentido de contagem, foi decidido determinar o sentido de contagem recorrendo ao valor inicial de contagem passado ao *timer*. Desta forma, quando configurado para contagem ascendente, o módulo *timer_block* passa o valor inicial de contagem diretamente para o módulo base do *timer*. Por outro lado, quando o *timer* é configurado para contagem decrescente, é feito o complemento para dois do valor inicial de contagem. Desta forma, o *timer* conta o tempo estipulado pelo programador de forma ascendente e gera o *overflow* normalmente. A Figura 80 apresenta o processo de determinação do valor inicial de contagem de acordo com a configuração definida.

```
////////////////////////////////////////////////////////////////////////////////
//// Contagem ascendente/descendente ///////////////////////////////////////////
//TIMCNT
assign value0 = configurationb[2]  ? ((~count_value0) + 8'b00000001) : (count_value0);
assign value1 = configurationb1[2] ? ((~count_value1) + 8'b00000001) : (count_value1);
assign value2 = configurationb2[2] ? ((~count_value2) + 8'b00000001) : (count_value2);
assign value3 = configurationb3[2] ? ((~count_value3) + 8'b00000001) : (count_value3);
//TIMRLD
assign rld_value0 = configurationb[2]  ? ((~reload_value0) + 8'b00000001) : (reload_value0);
assign rld_value1 = configurationb1[2] ? ((~reload_value1) + 8'b00000001) : (reload_value1);
assign rld_value2 = configurationb2[2] ? ((~reload_value2) + 8'b00000001) : (reload_value2);
assign rld_value3 = configurationb3[2] ? ((~reload_value3) + 8'b00000001) : (reload_value3);
////////////////////////////////////////////////////////////////////////////////
```

Figura 80 – Definir valor de contagem/recarga

5.4.3.3. Módulo *Timer#*

Por fim, falta apenas referir o módulo base do *timer*. Existem quatro módulos base, todos eles idênticos e todos eles instanciados no módulo *timer_block*. Este módulo é responsável por incrementar a cada sinal de relógio um *bit* no valor de contagem. A *flag* de *overflow* é colocada a 1 no ciclo assim que o valor de contagem atinja o valor 0xFF. Como é possível visualizar na Figura 81, o valor de *counting_next* é constantemente atualizado através da operação de *assign*.

No entanto o valor de *counting*, registo de contagem, só é atualizado com o valor de *counting_next* à transição ascendente do sinal de relógio.

```
////////////////////////////////////////////////////////////////////////////////

assign counting_next =  (sys_rst)                          ? 8'b0:
                        (enable_counter & ~enable_count)   ? (count_value):
                        (ovflag)                           ? reload_value:
                        (~counting[0] & enable_count)      ? {counting[7:1],1'b1}:
                        (~counting[1] & enable_count)      ? {counting[7:2],1'b1,1'b0}:
                        (~counting[2] & enable_count)      ? {counting[7:3],1'b1,2'b0}:
                        (~counting[3] & enable_count)      ? {counting[7:4],1'b1,3'b0}:
                        (~counting[4] & enable_count)      ? {counting[7:5],1'b1,4'b0}:
                        (~counting[5] & enable_count)      ? {counting[7:6],1'b1,5'b0}:
                        (~counting[6] & enable_count)      ? {counting[7],1'b1,6'b0}:
                        (~counting[7] & enable_count)      ? {1'b1,7'b0}:
                                                             counting[7:0];

assign ovflag =   (sys_rst)                    ? 1'b0:
                  (!enable_counter)            ? 1'b0:
                  (!enable_count)              ? 1'b0:
                  (counting == 8'b11111111)    ? 1'b1:
                                                 1'b0;

assign value_read = (read_value)    ? counting:
                                      32'b0;
////////////////////////////////////////////////////////////////////////////////
always @( posedge tick )
begin
    if(sys_rst)
        begin
            counting <= 8'b00000000;
        end
    else
        begin
            counting <= counting_next;
            enable_count <= enable_counter;
        end
end
```

Figura 81 - Módulo dos *subtimers*

5.4.3.4. Programação do *Timer*

Será agora apresentado como se deve proceder para programar este periférico. Na Figura 82 pode-se ver a configuração de dois *timers*, o *timer* 1 e o *timer* 2.

Neste exemplo são configurados no início os registos TIMCONFB de ambos os *timers*. São configurados para contagem descendente. De seguida tem-se a configuração dos valores de contagem e recarga de cada *timer*. Por fim é configurado o registo TIMCONFA, que habilita os *timers* e inicia a contagem.

```
/*****************************************/
/******** Configurar Timer ***************/
output=4;
writeBus(output,timerID,TIMCONFB0);
writeBus(output,timerID,TIMCONFB2);

// Configurar valores de contagem e recarga
output=0xC8;
writeBus(output,timerID,TIMCNT0);
writeBus(output,timerID,TIMRLD0);

output=0x05;
writeBus(output,timerID,TIMCNT2);
writeBus(output,timerID,TIMRLD2);

// Inicializa a contagem e habilitar timers
output=0x37;
writeBus(output,timerID,TIMCONFA);
/*****************************************/
```

Figura 82 – Código de programação do *timer*

5.4.4. UART

Para uma UART é necessário um gerador de *baudrate*, um *transmitter* e um *receiver*. O gerador de *baudrate* é responsável pela sincronização entre a UART e o terminal externo, o *transmitter* por enviar caracteres para o terminal e o *receiver* por receber os caracteres enviados pelo terminal.

5.4.4.1. *Baudrate*

A placa de desenvolvimento suporta um *baudrate* até 115200, mas como esta opera normalmente a 9600, foi decidido usar este valor para *baudrate*. É possível ver na Figura 83 a implementação do gerador de *baudrate* para esta UART.

Para gerar o *baudtick* é necessário dividir o relógio fornecido pela placa pelo *baudrate* desejado. Tal como foi dito anteriormente, foi escolhido um *baudrate* de 9600 e sendo fornecido um relógio de 100Mhz tem-se que o *baudtick* deve dar um *tick* aproximadamente a cada 104µs.

```
//////////////////////////////////////////////////////////////////////
// Baud rate generator
//////////////////////////////////////////////////////////////////////
`define Boardsys_clk        100000000
`define BaudRate            9600

`define NumberTicksTX  ((`Boardsys_clk)/(`BaudRate))
//////////////////////////////////////////////////////////////////////
// Local variables
//////////////////////////////////////////////////////////////////////

reg[16:0] count2=0;

//////////////////////////////////////////////////////////////////////
always@(posedge sys_clk)
    begin
        if(count2 == `NumberTicksTX)
            begin
                count2 <= 0;
            end
        else
            begin
                count2 <=count2 +1;
            end
    end

wire tickTx;

assign tickTx = (count2 == `NumberTicksTX)  ?   1 : 0;
```

Figura 83 – Gerador de *baudrate*

5.4.4.2. *Transmitter*

Para esta parte foi implementada uma máquina de estados, composta por quatro estados, sendo estes *Wait*, *Available*, *Transmitting* e *Done*. Pode-se ver na Figura 84 a implementação da máquina de estados.

O primeiro estado é o estado *Wait*, onde fica a aguardar que a UART seja habilitada e seja enviado um caractere. Quando estas condições se verificam passa para o estado *Available*. Neste estado apenas se realizam duas operações de atribuição, uma do valor que deve ser enviado e outra é a reinicialização de uma variável de contagem, passando logo de seguida para o próximo estado.

No estado *Transmitting*, tal como o nome indica, são transmitidos os dados pela UART. É enviado um bit sempre que se verificar um *baudtick* e ao fim de ter sido enviado o caractere todo passa-se para o último estado. Neste último estado a variável *ti_flag* toma o valor 1, que indica que a interrupção associada ao TX da UART deve ser despoletada e passa-se novamente para o estado *Wait*.

```
////////////////////////////////////////////////////////////////////////
// UART TX
////////////////////////////////////////////////////////////////////////
always @(posedge sys_clk)
begin
    if(sys_rst)
    begin
        countBits_TX <= 0;
        state_TX <= `Wait_TX;
        txReg<=1;
    end
    else
    begin
        case(state_TX)
        `Available_TX:
            begin
                state_TX <= `Transmitting_TX;
                dataSent = tbuf;
                countBits_TX <= 0;

            end
        `Transmitting_TX:
            begin
                if(countBits_TX == 4'd10)
                    begin
                    state_TX <= `Done_TX;
                    end
                else
                begin
                    if (tickTx)
                        begin
                            txReg <= dataSent[countBits];
                            countBits_TX <= countBits_TX +1;
                        end
                end
            end
        `Wait_TX:
            begin
            if(ten)
                begin
                    if(peripheral_select & command == 15'd0)
                        state_TX <= `Available_TX;
                end
            end
        `Done_TX:
            begin
                state_TX <= `Wait_TX;
            end
        endcase
    end
end
////////////////////////////////////////////////////////////////////////
```

Figura 84 – Máquina de estados do UART *transmitter*

5.4.4.3. *Receiver*

Esta parte também foi implementada recorrendo a uma máquina de estados, desta vez composta por três estados, *Wait*, *Receiving* e *Done*. A implementação da máquina de estados está apresentada na Figura 85.

O primeiro estado é o estado *Wait*, onde fica a aguardar que o sinal RX fique com o valor 0. Quando isto acontece passa-se para o estado *Receiving*, espera-se metade do tempo de um *baudtick* e verifica-se novamente se o RX continua a 0. Caso continue é iniciada a aquisição do caractere. Finalizada a aquisição passa-se para o estado *Done* onde é guardado o caractere recebido, a variável *ri_flag* toma o valor 1 e passa-se novamente para o estado *Wait*.

```
////////////////////////////////////////////////////////////////////////
// UART RX
////////////////////////////////////////////////////////////////////////
always @(posedge sys_clk)
begin
      if(sys_rst)
        begin
            countBits_RX <= 0;
            state_RX <= `Wait_RX;
        end
      else
        begin
            case(state_RX)
            `Wait_RX:
                  begin
                   if(RX == 0 & ten & tickHalf)
                        state_RX <= `Receiving_RX;
                  end
            `Receiving_RX:
                  begin
                        if(countBits_RX == 4'd8 & (RX == 1 & tickRx))
                              begin
                                    state_RX <= `Done_RX;
                              end
                        else
                        begin
                              if (tickRx)
                                    begin
                                          dataReceived[countBits_RX] <= RX;
                                          countBits_RX <= countBits_RX +1;
                                    end
                        end
                  end
            `Done_RX:
                  begin
                        countBits_RX <= 0;
                        state_RX <= `Wait_RX;
                        dataReceived <= 8'b0;
                  end
            endcase
      end
end
////////////////////////////////////////////////////////////////////////
```

Figura 85 – Máquina de estados do UART *receiver*

5.4.4.4. Programação da UART

Para programar este periférico são necessários alguns passos. Como se pode ver na Figura 86 é necessário configurar a interrupção da UART_TX no periférico DVIC, criar uma função para enviar os caracteres pela UART e configurar o que deve ser feito na interrupção quando esta é despoletada.

```
/*********************************************/
/*************** Funções *********************/
void usartConfig(void)
{
	int var=1;
	writeBus(var,dvicID,IRQ_USART_SEND_enable);
	writeBus(var,dvicID,IRQ_USART_SEND_prio);
	writeBus(var,usartID,USART_enable);
}

void println(char *buffer)
{
	char *ptr,*ptr2;
	ptr=buffer;
	ptr2=&txBuffer[0];

	while(txBusy){}

	while(*ptr != 0)
	{
		*ptr2=*ptr;
		ptr++;
		ptr2++;
	}
	*ptr2='\r';
	ptr2++;
	*ptr2='\n';
	ptr2++;
	*ptr2=0;

	writeBus(txBuffer[txIter],usartID,USART_send);
	txIter=1;
	txBusy=1;
}


void USART_TX_ISR_vect(void)
{
	if(txBuffer[txIter] != 0)
	{

	writeBus(txBuffer[txIter],usartID,USART_send);
		txIter++;
	}
	else
	{
		txIter=0;
		txBusy=0;
	}
}
```

Figura 86 – Funções auxiliares da UART

Na Figura 87 encontra-se o programa principal, a *main*. Está apresentada a função de configuração da interrupção que é chamada no início, sendo de seguida enviada pela UART, usando a função *println*, a *string* "UART…".

```
/*******************************************/
/*************** MAIN **********************/
int main()
{
      usartConfig();

      char str[]={"UART..."};
      char *addr;

      addr=&str[0];
      println(addr);

return 0;
}
/*******************************************/
```

Figura 87 – Programação da UART

5.4.5. SYSTICK

A implementação deste periférico é bastante semelhante ao periférico de temporizadores. A Figura 88 apresenta as diferentes opções de programação. Pode-se ver que os valores de contagem e de recarga recebem o complemento para dois do valor enviado pelo utilizador, pois este periférico apenas permite contagem descendente. As outras opções de programação, tal como no periférico *timers*, permitem ler o valor de contagem atual e habilitar/desabilitar o periférico.

```
////////////////////////////////////////////////////////////////////
assign systick_count_value_last = (peripheral_select & command == 15'd0 &
                                  !read_write)  ?   ((~data) + 8'b00000001):
                                                    systick_count_value;

assign systick_value_read = (peripheral_select & command == 15'd0 &
                                   read_write)  ?   counting:
                                                    32'b0;

assign systick_reload_value_last = (peripheral_select & command == 15'd1)
                                                 ?   ((~data) + 8'b00000001):
                                                    systick_reload_value;

assign enable_counter = (peripheral_select & command == 15'd2)
                                                 ?   data:
                                                    enable_count;
////////////////////////////////////////////////////////////////////
```

Figura 88 – Variáveis de configuração do SYSTICK

Como se pode ver na Figura 89 este periférico apenas tem uma fonte de interrupção, que é ativa quando a *flag* de *overflow* tem o valor um. Este valor é enviado para o controlador de interrupções onde é processado pelo mesmo de acordo com a programação.

```
assign int_req = ovflag ? 1'b1: 1'b0;
```

Figura 89 – Interrupção do SYSTICK

5.4.5.1. Programação do SYSTICK

Para programar o SYSTICK é necessário configurar a sua interrupção no DVIC e depois basta configurar o valor de contagem inicial, o valor de recarga e habilitar o periférico para iniciar a contagem. Na Figura 90 está apresentada a função *sysTickConfig*, que configura a interrupção e de seguida tem-se um excerto da *main* em que é feita a configuração dos valores de contagem, recarga e é habilitado o periférico.

```
/*********************************************/
/****** Configurar interrupção SYSTICK *******/
void sysTickConfig(void)
{
      int val=1;
      writeBus(val,dvicID,IRQ_SYSTICK_enable);
      val=32;
      writeBus(val,dvicID,IRQ_SYSTICK_prio);


}
/*********************************************/


/*********************************************/
/******** Configurar SYSTICK *****************/
output=1000;
output *= 100;
writeBus(output,systickID,SYSTICK_count);
writeBus(output,systickID,SYSTICK_reload);
output=1;
writeBus(output,systickID,SYSTICK_enable);
/*********************************************/
```

Figura 90 – Configurar interrupções SYSTICK

Capítulo 6

TESTES E RESULTADOS

O capítulo seis encontra-se dividido em dois subcapítulos. O primeiro refere-se aos testes de unidade, onde são testados os diferentes componentes do SoC separadamente. No outro subcapítulo são apresentados os testes de integração, que são os testes referentes ao sistema com o processador e os periféricos integrados.

Todos os testes apresentados foram realizados utilizando código gerado pelo compilador e *assembler* desenhados em paralelo com este SoC.

6.1. Testes de unidade

Estes testes destinam-se a validar as diferentes unidades que constituem o SoC. Numa primeira parte estão apresentados os testes sobre o processador propriamente dito, tal como a sua *hazard unit*. De seguida são apresentados os testes referentes aos periféricos.

6.1.1. Processador

Como já foi dito em capítulos anteriores, este processador possui uma arquitetura *load*/*store*, logo os testes mais importantes passam por verificar se estas duas instruções são executadas corretamente. Além destas também vou ser testadas outras instruções que acedem à memória como *push* e *pop*, instruções genéricas como operações aritméticas/lógicas e saltos.

Store

Neste primeiro caso vai-se analisar um teste sobre uma instrução de *store*. Na Figura 91 encontra-se o excerto de código que vai ser mostrado na simulação. Pode-se ver que da posição 0x0036 à 0x003A são executadas instruções de *Store Word Immediate*, ou seja, o endereço onde vão ser guardados os valores é dado pela soma do valor presente no registo R14 e o valor *Imm*.

```
0x0021      MOVI        r14,32
  (...)
0x0031      MOVI        r1,85
0x0032      MOVI        r2,65
0x0033      MOVI        r3,82
0x0034      MOVI        r4,84
0x0035      MOVI        r5,32
0x0036      SWI         r1,r14,24
0x0037      SWI         r2,r14,25
0x0038      SWI         r3,r14,26
0x0039      SWI         r4,r14,27
0x003A      SWI         r5,r14,28
```

Figura 91 – Processador *store* código

Tendo em consideração este código espera-se que o processador execute as cinco instruções de seguida, guardando os diferentes registos em diferentes posições, mas todas consecutivas. A simulação do processador a executar este código é visível na Figura 92. Os sinais correspondes aos acessos à memória encontram-se a cinzento. Como se pode ver a partir dos 350ns começam a ser feitos os acessos à memória, onde vão sendo guardados a cada ciclo de relógio os valores dos diferentes registos. Os valores guardados são aqueles guardados nos registos R1-R5, que na simulação se encontram na base hexadecimal.

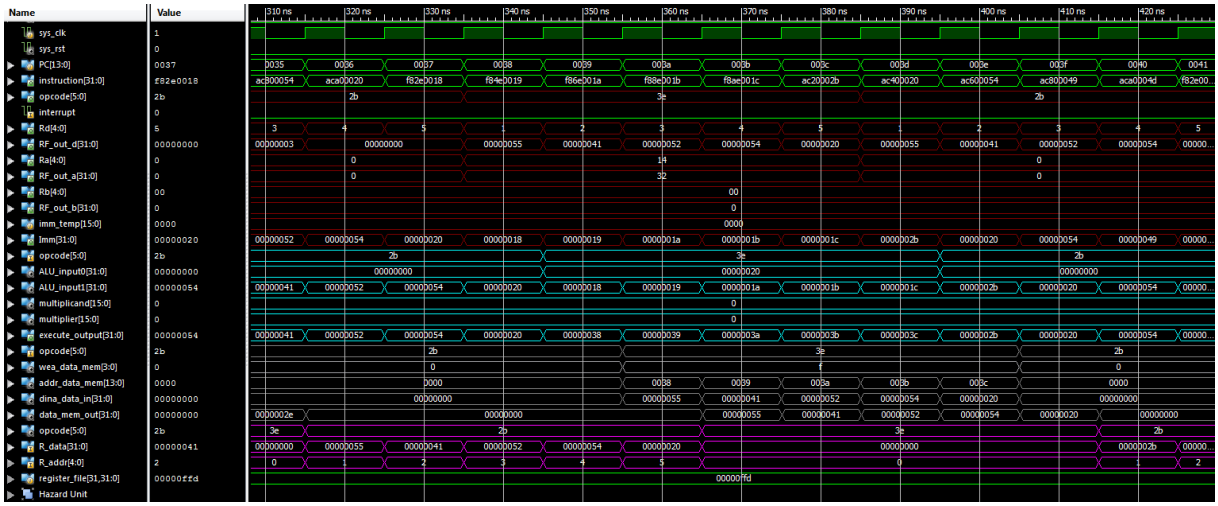


Figura 92 – Processador *store* simulação

Após execução de todas as operações de *store* espera-se que nas posições 56-60 estejam os valores presentes nos registos. A Figura 93 apresenta esse pedaço da memória de dados, onde se pode ver que os valores se encontram nas posições de memória que deviam estar.

55	0
56	85
57	65
58	82
59	84
60	32
61	0
62	0

Figura 93 – Processador *store* memória de dados

Load

Neste teste verifica-se se as instruções de *load* são executadas corretamente. Como se pode ver pelo código apresentado na Figura 94 são executadas três instruções de *load*. No final deste trecho de código os registos R2, R3 e R4 vão conter os valores lidos da memória de dados.

```
0x0024      MOV       r16,r31
  (…)
0x0035      LWBI      r3,r16,-3
0x0036      LW        r2,r3,r0
0x0037      LWBI      r4,r16,-0
```

Figura 94 – Processador *load* código

Através da simulação presente na Figura 95 pode-se ver quais os valores que são lidos da memória. No estágio *write back* a roxo consegue-se ver os valores lidos a serem guardados nos registos respetivos.

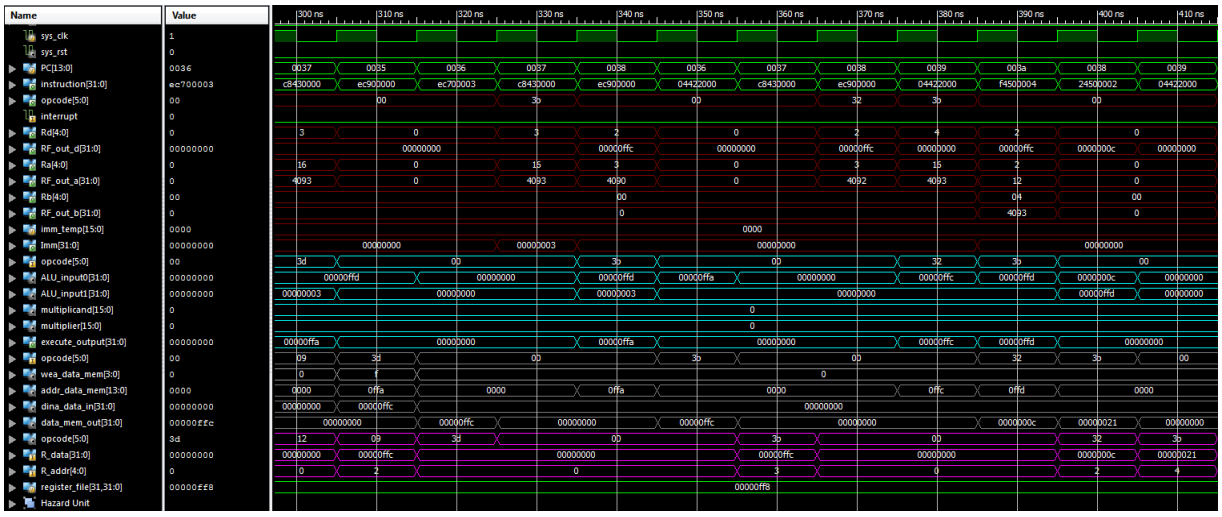


Figura 95 – Processador *load* simulação

As posições lidas foram 0xFFA, 0xFFC e 0xFFD. Na Figura 96 é possível ver os valores contidos nessas posições na memória de dados, bem como os registos para onde foram escritos esses valores.

0xFF8	00000000
0xFF9	00000000
0xFFA	00000FFC
0xFFB	00000000
0xFFC	0000000C
0xFFD	00000021
0xFFE	00000000
0xFFF	00000026

0x5	00000000
0x4	00000021
0x3	00000FFC
0x2	0000000C

Figura 96 – Esquerda – memória de dados; Direita – *register file*

Push

Para este teste analisa-se o excerto de código presente na Figura 97, onde se pode ver a ser feito o *push* a quatro registos diferentes. Estes valores irão ser guardados na pilha, que começa no topo da memória de dados.

```
0x0031      MOVI      r5,0
0x0032      PUSH      r5
0x0033      PUSH      r1
0x0034      PUSH      r0
0x0035      PUSH      r16
0x0036      MOV       r2,r31
0x0037      LWBI      r4,r16,-0
```

Figura 97 – Processador *push* código

Na simulação presente na Figura 98 deve-se analisar o estágio *memory access* a cinzento. A partir dos 280ns são feitas escritas na memória de dados nas posições que correspondem à pilha dos valores presentes nos registos.

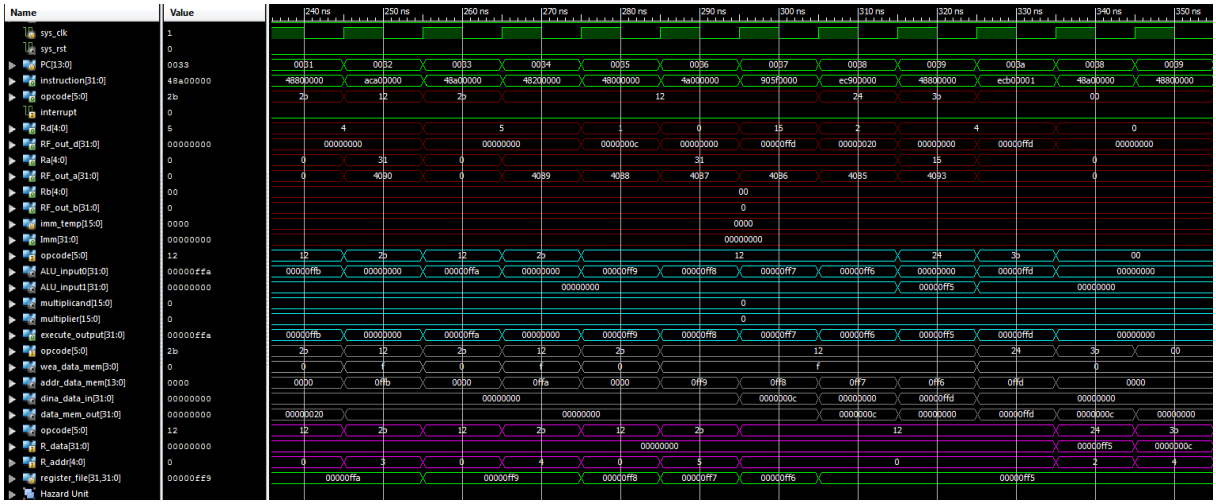


Figura 98 – Processador *push* simulação

Assim, espera-se que na memória de dados estejam os diferentes valores dos registos na pilha. Na Figura 99 é possível ver que tal acontece, da posição 0xFF9 até à 0xFF6 estão os valores que foram *push*.

Figura 99 – Processador *push* memória de dados

Pop

Para este teste vai-se analisar o trecho de código presente na Figura 100, onde se podem ver sete instruções de *pop*.

```
0x0117      SWI         r1,r14,0
0x0118      MOVI        r2,0
0x0119      SWI         r2,r14,1
0x011A      POP         r1
0x011B      POP         r2
0x011C      POP         r3
0x011D      POP         r4
0x011E      POP         r5
0x011F      POP         r6
0x0120      POP         r7
0x0121      RETI
```

Figura 100 – Processador *pop* código

Sendo assim espera-se que sejam feitos sete acessos à memória a posições consecutivas para se obter os valores que foram *push* numa parte anterior deste código. Na Figura 101 é possível ver a simulação deste trecho de código. Aqui também se deve analisar o estágio *de* acesso à memória a cinzento, onde se pode ver a partir 6,250,885ns as setes instruções. Além desse estágio também se deve analisar o estágio de atualização do *register file* que se encontra a roxo. Aqui pode-se ver os valores que foram lidos da memória de dados a serem escritos no registos correspondentes.

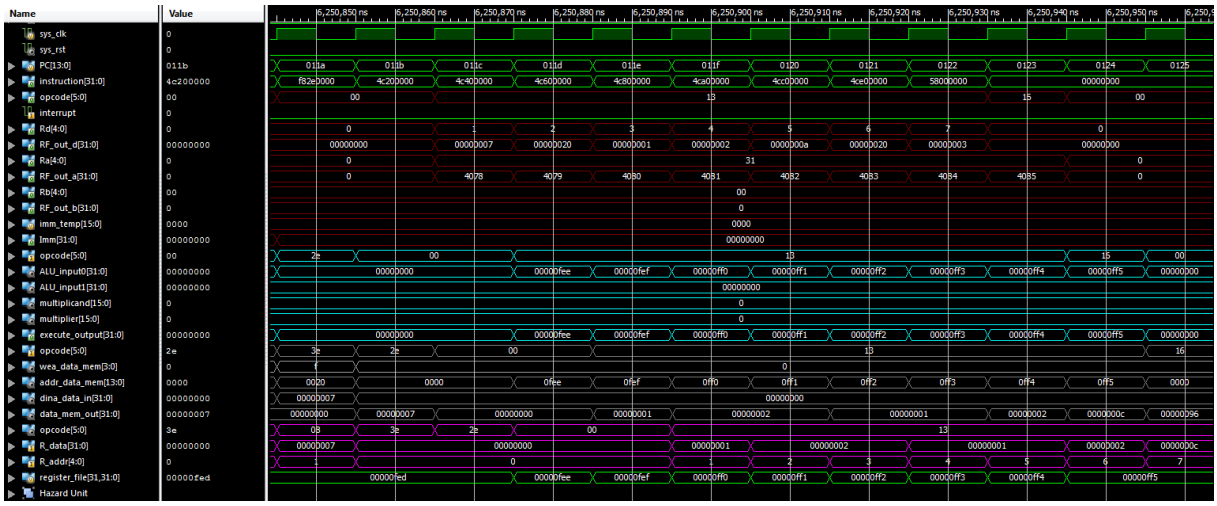


Figura 101 – Processador *pop* simulação

Espera-se então que em sete posições consecutivas se encontrem os valores que vão ser escritos para os registos. Na Figura 102 pode-se ver as posições de memória que continham os valores a serem lidos.

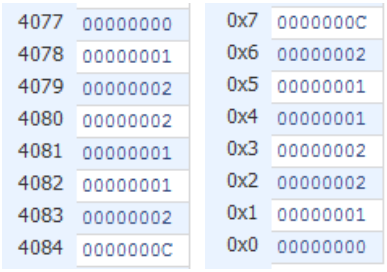


Figura 102 – Processador *pop*; Esquerda - memória de dados; Direita – *register file*

Tendo em consideração os valores presentes na memória, os valores dos registos R1-R7 devem ter esses mesmos valores. Como se pode ver o *register file* na Figura 102, os registos contêm os valores que estavam na memória de dados.

Instruções aritméticas

Neste teste verifica-se se as instruções aritméticas funcionam corretamente. Apenas são executadas algumas como mostra o trecho de código na Figura 103. Espera-se com este teste que os valores finais nos registos R3, R4, R6 e R8 estejam corretos de acordo com as operações feitas sobre os operandos.

```
0x0021      MOVI      r1,3
0x0022      MOVI      r2,7
0x0023      ADD       r3,r1,r2
0x0024      SUB       r4,r3,r2
0x0025      MOVI      r5,5
0x0026      OR        r6,r5,r3
0x0027      MOVI      r7,4
0x0028      AND       r8,r7,r1
```

Figura 103 – Processador instruções aritméticas/lógicas código

Na simulação (Figura 104) é possível ver as quatro instruções a serem executadas e guardadas nos resistos ao longo do *pipeline*. Para estas instruções deve-se estar mais atento ao estágio *execute* onde se encontra a ALU, a azul na simulação.

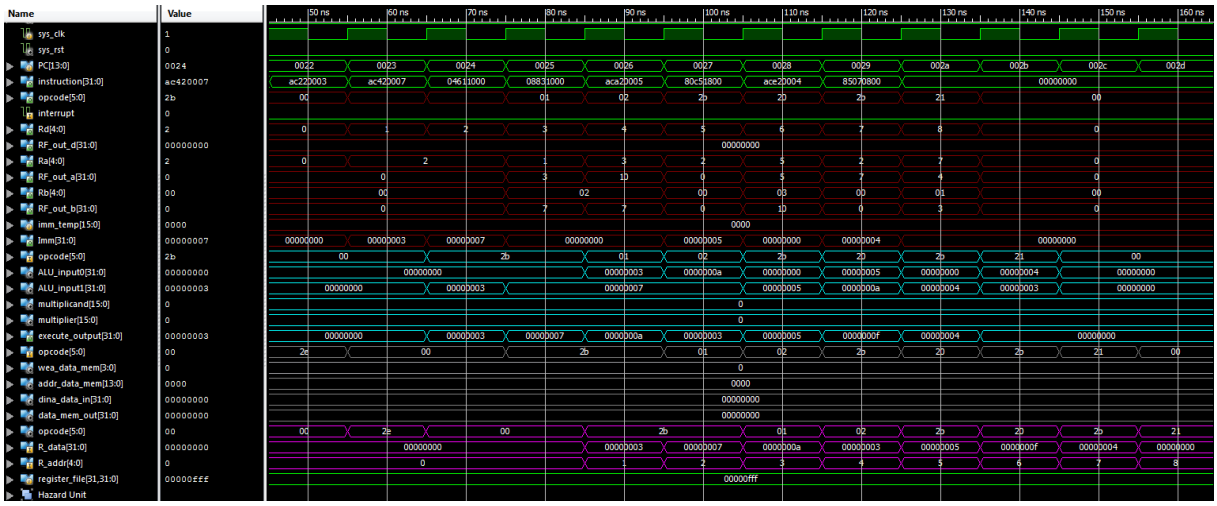


Figura 104 – Processador instruções aritméticas/lógicas simulação

Os resultados esperados nos registos R3, R4, R6 e R8 verificam-se como corretos. Na Figura 105 é possível ver os valores presentes no *register file* após a execução de todas as instruções.

0x8	00000000
0x7	00000004
0x6	0000000F
0x5	00000005
0x4	00000003
0x3	0000000A
0x2	00000007
0x1	00000003
0x0	00000000

Figura 105 – Processador instruções aritméticas/lógicas *register file*

Instruções de salto

Como já foi explicado em capítulos anteriores, as instruções de salto condicionais necessitam de uma instrução de CMP antes. O trecho de código a ser analisado encontra-se na Figura 106, onde se pode ver três instruções, duas das quais são para o salto condicional e a terceira é um salto absoluto, mas que só executa caso a condição de salto condicional falhe.

```
0x003A      CMP       r4,r3,r0
0x003B      BEQ       r4,r5
0x003C      BRI       0x4
```

Figura 106 – Processador instruções de salto

Na simulação da Figura 107 pode-se ver a instrução de salto condicional BEQ precedida por uma instrução CMP. Neste caso a instrução de salto não se verificou, logo não houve o salto. Sendo assim, foi executado o salto absoluto que faz um salto de quatro.

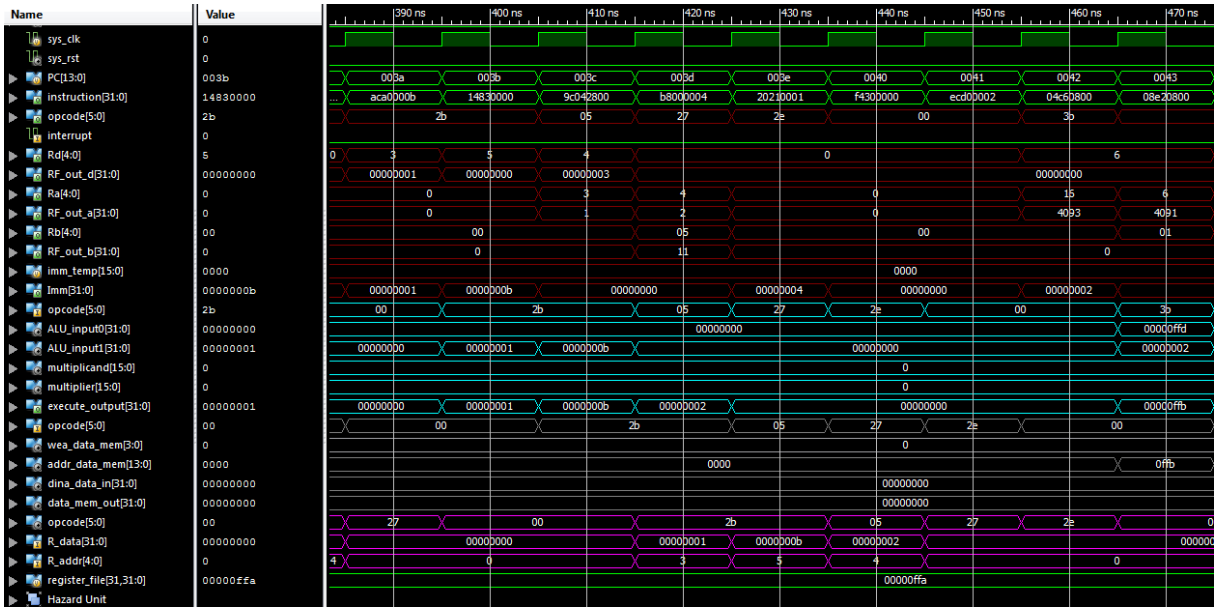


Figura 107 – Simulação instruções de salto

6.1.2. *Hazard unit*

Data forward

Existem os *data forward* sempre que uma instrução necessita de um valor que é calculado por uma instrução anterior, mas só é possível o reencaminhamento desse valor caso a instrução anterior já o tenha calculado. Na Figura 108 encontra-se um trecho de código onde é possível

ver um *hazard* entre as instruções de adição e *store*. A instrução de *store* necessita do registo R1, que irá ser calculado pela instrução de adição.

```
0x0112      LWI       r1,r14,0
0x0113      ADDI      r1,r1,1
0x0114      SWI       r1,r14,0
0x0115      BRI       0x5
```

Figura 108 – *Hazard unit* código *data forward*

A resolução deste *hazard* pode ser vista na Figura 109. Nos 1,042,325ns um sinal que indica que existe um *hazard* denominado de *Rd_eq_Rd_exec* fica com o valor 1, o que quer dizer que a instrução no estágio de *decode* necessita de um valor que está a ser usado pela instrução que se encontra no estágio *execute*. Para resolver este *hazard*, já que o valor que o *store* precisa já se encontra calculado pela instrução de adição, é reencaminhado o seu valor através da variável *data_forward_exec* do estágio *execute* para o estágio *decode* para a instrução de *store*.

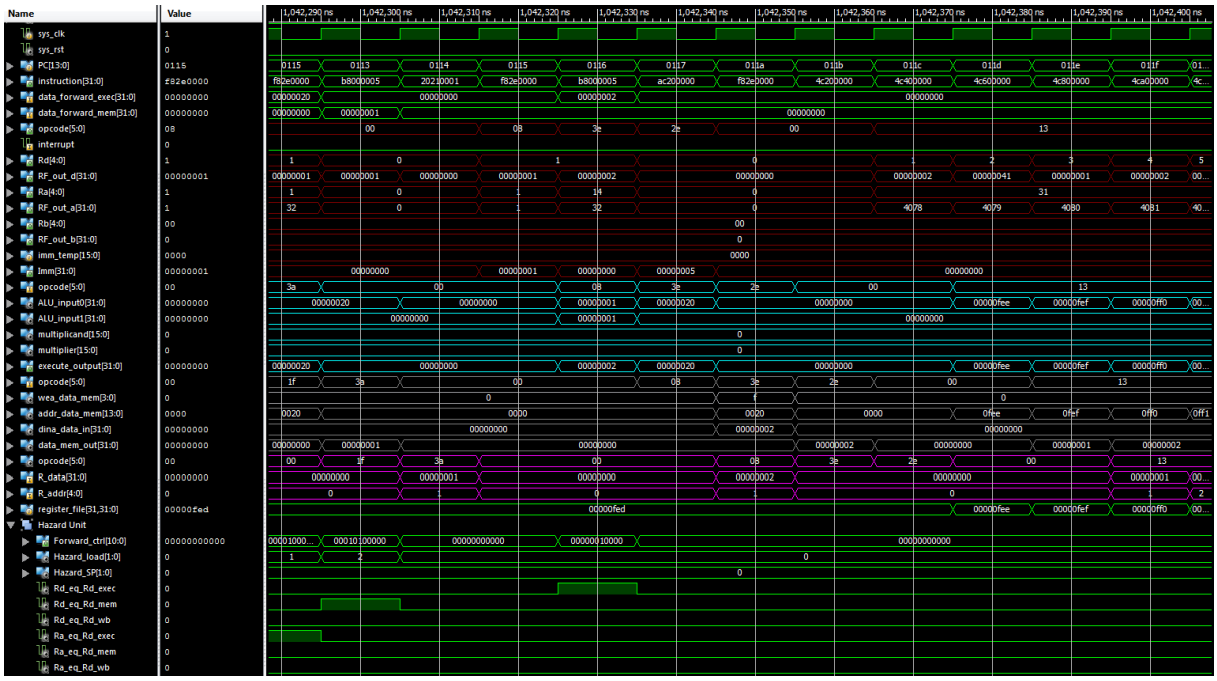


Figura 109 – *Hazard unit* simulação *data forward*

Stall

Neste teste vai-se ver o processador a realizar um *stall* devido à forma como o código (Figura 110) se encontra. Pode-se ver que nas posição 0x00C2 se encontra uma instrução que faz uso do registo R2, o qual é utilizado pela instrução anterior LW. Como esta instrução é um

load, o processador só saberá o valor que o registo R2 irá conter no estágio *memory access*, logo existe a necessidade de fazer um *stall* ao processador. O estágio de *memory access* é o quarto estágio neste *pipeline* de cinco estágios, por isso há a necessidade de fazer o *stall* durante quatro ciclos de relógio. Ao fim desses quatro ciclos já se sabe o valor que R2 irá ter, por isso é feito o *data forward* do valor para evitar esperar pela atualização do *register file*.

```
0x00C0      ADD         r2,r2,r3
0x00C1      LW          r2,r14,r2
0x00C2      PERW        r2,4,0
0x00C3      MOVI        r1,1
0x00C4      SWI         r1,r14,0
```

Figura 110 – *Hazard unit* código *stall*

Na Figura 111 é possível ver que aos 14,845ns o sinal *hazard_load* toma o valor 1. Este valor indica que uma instrução posterior ao *load* requer o valor que se irá ser lido da memória, ou seja, que é necessário o *stall*, seguido de um *data forward*. Nos 14,875ns vê-se que a instrução que se segue ao *load* já utiliza o valor lido da memória, 0x55.

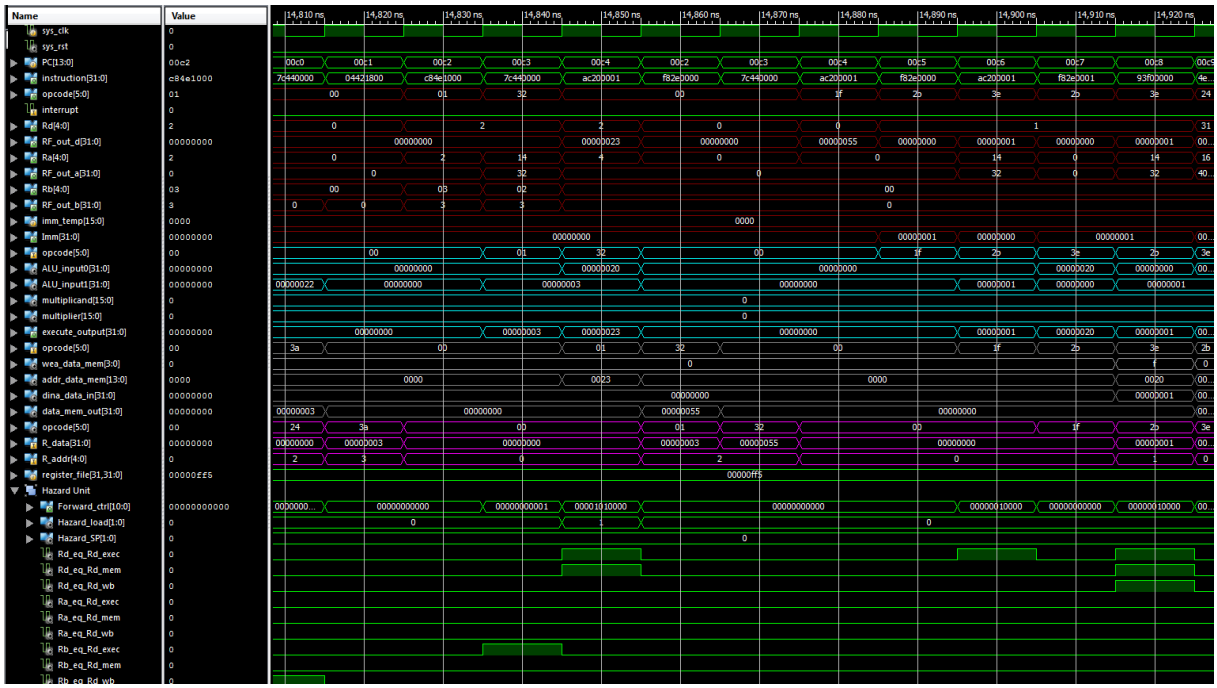


Figura 111 – *Hazard unit* simulação *stall*

Bubble

As *bubbles* são sempre necessários quando existem saltos. Neste processador sempre que há um salto existe a necessidade de duas *bubbles*. O código que vai ser apresentado na

simulação encontra-se na Figura 112, onde se pode ver uma instrução de CALL. Neste caso apenas era necessário uma *bubble* já que o salto é de 0x25 para 0x27, mas o processador usa na mesma duas *bubbles* para todos os saltos.

```
0x0024      MOV       r16,r31
0x0025      CALL      0x27
0x0026      BRI       -0x0
0x0027      POP       r1
0x0028      SWI       r1,r16,2
```

Figura 112 – *Hazard unit* código *bubble*

Dito isto, na Figura 113 encontra-se a simulação onde se pode ver um salto a ser feito, bem como as *bubbles* inseridas no *pipeline*. Pode-se ver que o processador insere duas *bubbles* no *pipeline* após o salto, sendo a próxima instrução a ser executada aquela para onde foi feito o salto.

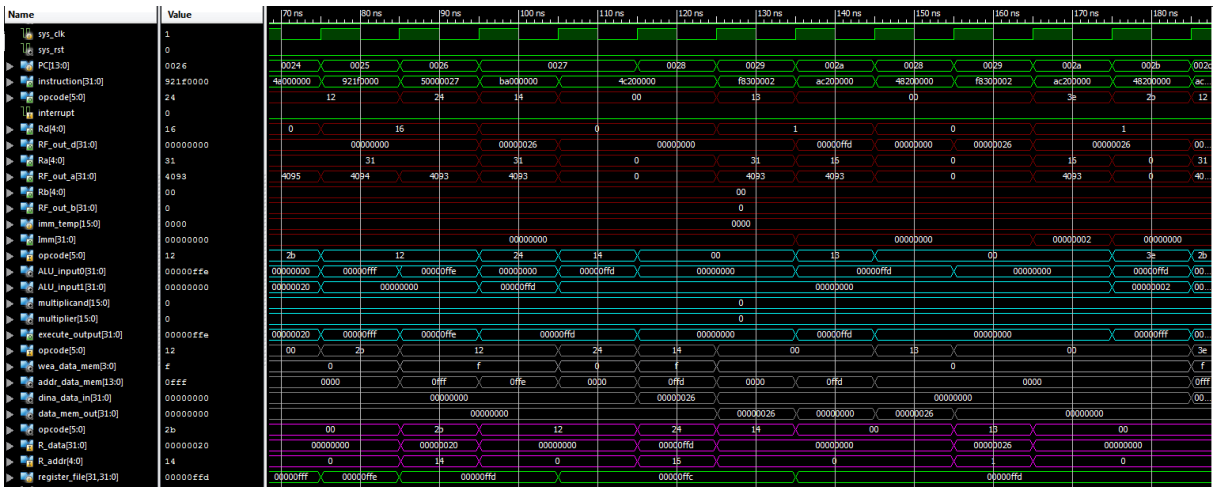


Figura 113 – *Hazard unit* simulação *bubble*

6.1.3. Periféricos

Vão ser apresentados os testes feitos para validar os diferentes periféricos. O periférico DVIC é validado através dos testes feitos ao periféricos, tais como gerar interrupções pelo I/O ou pelos *timers*.

6.1.3.1. Porto I/O

Este periférico é constituído, como já foi dito anteriormente, por um oito pinos de entrada, oito pinos de saída e cinco botões. Para validar este periférico são necessários dois testes, um

para testar os pinos de entrada/saída e outro para testar os botões, que são as fontes de interrupção.

Neste primeiro teste foi executado o código apresentado na Figura 114. Este código faz uso das duas funções que são usadas para programar o controlador I/O. Estão dentro de um ciclo *while* infinito, onde a primeira função lê o valor presente nos pinos de entrada e a segunda escreve esse mesmo valor nos pinos de saída.

```
/*******************************************/
/*************** MAIN **********************/
int main()
{
    while(1)
    {
            readBus(num,parioId,ler);
            writeBus(num,parioId,escrever);

    }
return 0;
}
/*******************************************/
```

Figura 114 – Código de teste do controlador I/O

Como os pinos de entrada estão mapeados em um dos *switchs* presentes na placa de desenvolvimento e os pinos de saída estão mapeados nos *leds*, tem-se que verificar se o valor que for introduzido no *switch* irá se refletir nos *leds*. Como se pode ver na Figura 115 o controlador I/O executa como esperado, pois no *switch* encontra-se o valor 0xF0 que é o mesmo apresentado pelos *leds*.

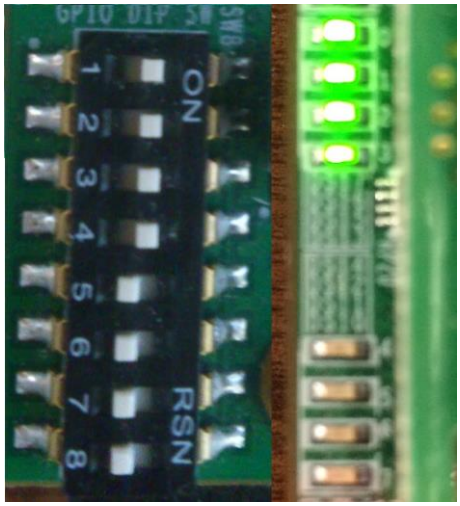


Figura 115 – Botões/Leds 0xF0

De seguida alterou-se o valor no *switch* para 0xAA e pode-se ver que os *leds* refletem essa mudança passando a apresentar também o valor 0xAA. O resultado desta alteração encontra-se na Figura 116.

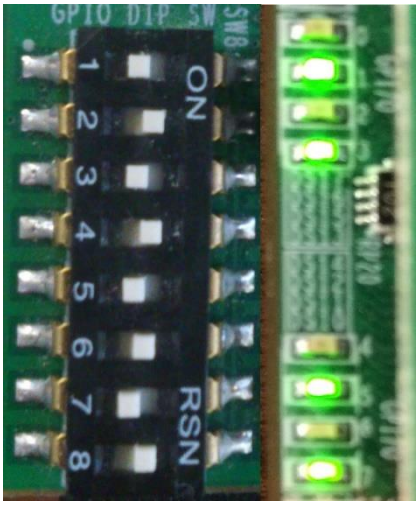


Figura 116 – Botões/Leds 0xAA

Neste segundo teste vai-se executar um código que valide as interrupções do controlador I/O, provando assim que o DVIC e o processador são capazes de responder às interrupções despoletadas. O código executado encontra-se na Figura 117. As interrupções são despoletadas por *hardware* recorrendo aos cinco botões presentes na placa de desenvolvimento. No código apresentado tem-se que ao premir os diferentes botões são apresentados diferentes valores nos *leds*. No início deste código é configurado o DVIC para habilitar as cinco interrupções e é dada a mesma prioridade a todas. Dentro de cada ISR estão os valores que devem ser enviados para os *leds*.

```
/************************************/
/************* MAIN *****************/
int main()
{
    int output;



/************************************/
/****** Configurar DVIC *************/
output=0x01;

writeBus(output,dvicID,IRQ0_enable);

writeBus(output,dvicID,IRQ1_enable);

writeBus(output,dvicID,IRQ2_enable);

writeBus(output,dvicID,IRQ3_enable);

writeBus(output,dvicID,IRQ4_enable);

    output=0x03;
    writeBus(output,dvicID,IRQ0_prio);
    writeBus(output,dvicID,IRQ1_prio);
    writeBus(output,dvicID,IRQ2_prio);
    writeBus(output,dvicID,IRQ3_prio);
    writeBus(output,dvicID,IRQ4_prio);

/************************************/


    while(1){}

return 0;
}
```

```
/************************************/
/********* Interrupções *************/
void IRQ0_ISR_vect(void)
{
    out = 0x01;
    writeBus(out,parioId,escrever);
}

void IRQ1_ISR_vect(void)
{
    out = 0x02;
    writeBus(out,parioId,escrever);
}

void IRQ2_ISR_vect(void)
{
    out = 0x03;
    writeBus(out,parioId,escrever);
}

void IRQ3_ISR_vect(void)
{
    out = 0x04;
    writeBus(out,parioId,escrever);
}

void IRQ4_ISR_vect(void)
{
    out = 0x05;
    writeBus(out,parioId,escrever);
}
/************************************/
```

Figura 117 – Configuração de 5 interrupções do I/O

Espera-se então que quando forem pressionado o botão *north*, *east*, *west*, *south* e *center* irá ser posto nos *leds* os valores 0x01, 0x02, 0x03, 0x04 e 0x05, respetivamente. Na Figura 118 encontra-se a prova de que quando pressionados os botões, os valores que os *leds* tomam correspondem aos valores presentes em cada interrupção.

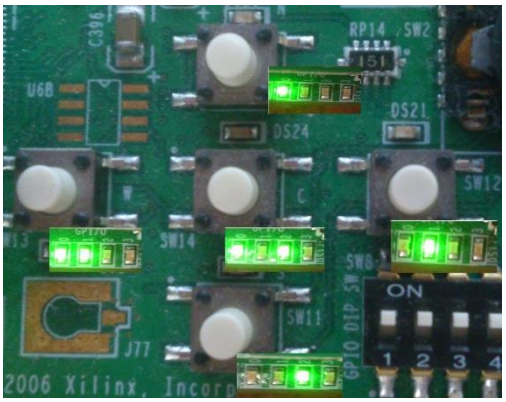


Figura 118 – Teste às interrupções do I/O

6.1.3.2. *Timers*

Este periférico é capaz de despoletar quatro interrupções, uma para cada *subtimer*. Para o validar é necessário configurar *timers*, o controlador DVIC e o que deve ser feito em cada interrupção.

Com o teste presente na Figura 119 tem-se que quando forem despoletadas as interrupções dos *timers* configurados, os *leds* devem ter valores diferentes. Quando for despoletada a interrupção do *timer* 0 os *leds* devem ter o valor 0x0F e quando for despoletada a interrupção do *timer* 2 os *leds* devem ter o valor 0xF0.

```
/*****************************************/
/*************** MAIN ********************/
int main()
{
    int output;


/*****************************************/
/****** Configurar DVIC ******************/
output=0x01;
writeBus(output,dvicID,IRQ_TIMER0_enable);

output=0x03;
writeBus(output,dvicID,IRQ_TIMER0_prio);

output=0x01;
writeBus(output,dvicID,IRQ_TIMER2_enable);

output=0x05;
writeBus(output,dvicID,IRQ_TIMER2_prio);
/*****************************************/


/*****************************************/
/******** Configurar Timer ***************/
    output=4;
    writeBus(output,timerID,TIMCONFB0);

    output=0xC8;
    writeBus(output,timerID,TIMCNT0);
    writeBus(output,timerID,TIMRLD0);

    output=0x64;
    writeBus(output,timerID,TIMCNT2);
    output=0xC8;
    writeBus(output,timerID,TIMRLD2);

    output=0x37;
    writeBus(output,timerID,TIMCONFA);

/*****************************************/

    while(1){}

return 0;
}
```

```
/*****************************************/
/************ Interrupções ***************/
void TIMER0_ISR_vect(void)
{
    out = 0x0F;
    writeBus(out,parioId,escrever);
}

void TIMER2_ISR_vect(void)
{
    out = 0xF0;
    writeBus(out,parioId,escrever);

/*************************************
```

Figura 119 – Configuração de interrupções dos *timers*

Os tempos com que foram configurados os *timers* fazem com os *leds* fiquem 50% do tempo com o valor 0x0F e os outros 50% com o valor 0xF0. Como se pode ver na Figura 120 o resultado esperado é o que acontece, pois os valores apresentados pelos *leds* varia entre 0xF0 e 0x0F.

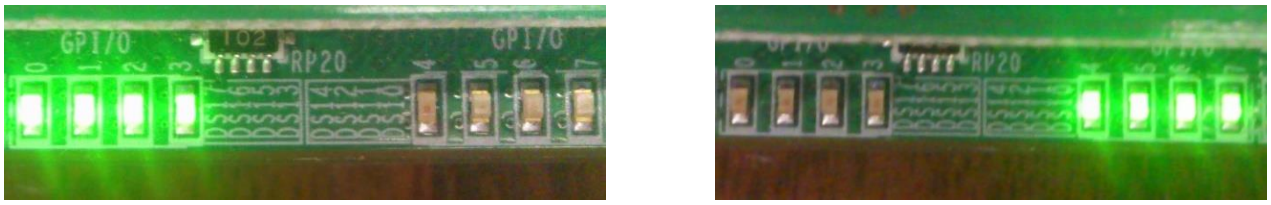


Figura 120 – Esquerda - *leds* com 0xF0; Direita - *leds* com 0x0F;

Mais uma vez fica provado que o periférico DVIC e o processador funcionam, pois o DVIC foi capaz de gerir as interrupções despoletadas pelos *timers* e informar o processador da ocorrência de interrupções e o processador foi capaz de executar o código de cada interrupção sem erros.

6.1.3.3. UART

Para verificar se este periférico funciona corretamente recorreu-se a um teste que usa o *timer* para determinar de quanto em quanto tempo é imprimida uma *string*. Sempre que o *timer* utilizado sofrer o *overflow* irá despoletar a sua interrupção, em que esta irá passar um valor diferente de zero para uma variavel. Esta variavel chama-se *timer* e é a condição do ciclo *if* presente no ciclo *while*. Sempre que for diferente de zero irá ser transmitida a *string* "UART + TIMER". Na Figura 121 é possivel ver o código que foi compilado para este teste.

```
/*****************************************/
/*************** Main ******************/
int main()
{

/*****************************************/
/******** Configurar DVIC + UART ********/

int var=1;

writeBus(var,dvicID,IRQ_UART_SEND_enable);
writeBus(var,dvicID,IRQ_UART_SEND_prio);
writeBus(var,uartID,UART_enable);

/*****************************************/

    configTimer();

    char str[]={"UART + TIMER"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        if(timer)
        {
            timer=0;
            println(addr);
        }
    }
    return 0;
}
```

```
/*****************************************/
/************ Interrupções **************/
void UART_TX_ISR_vect(void)
{
    if(txBuffer[txIter] != 0)
    {
        writeBus(txBuffer[txIter],uar-
tID,UART_send);
        txIter++;
    }
    else
    {
        txIter=0;
        txBusy=0;
    }
}
/*****************************************/
```

Figura 121 – Configuração da UART e *Timers*

Para realizar este teste foi necessário ligar a placa de desenvolvimento através da sua porta série COM1 a um terminal externo, neste caso um computador. O resultado obtido pode ser visto na Figura 122. Como esperado foi escrito no terminal de *x* em *x* tempo a *string* "UART + TIMER".

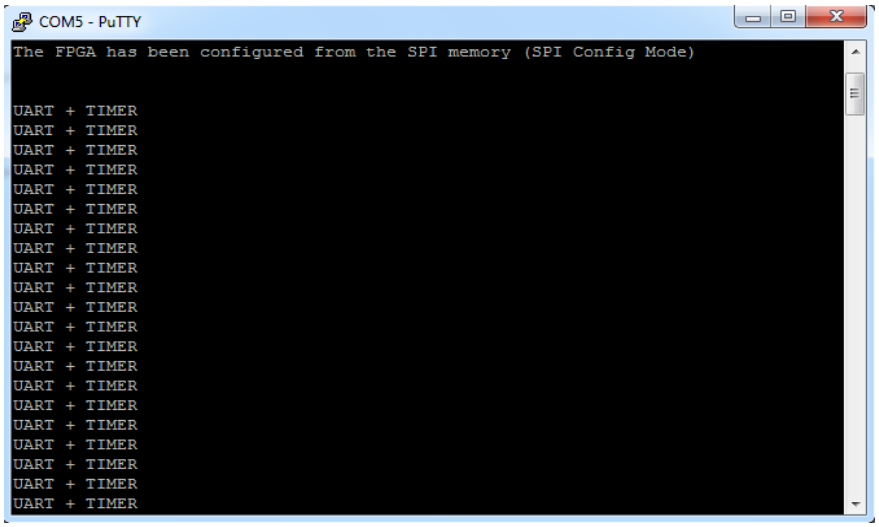


Figura 122 – Resultado do teste da UART e *Timers*

O teste anterior apenas prova que o *transmitter* funciona. Para testar o *receiver* foi compilado o código presente na Figura 123. É esperado que sempre que um caractere for recebido pela porta série o mesmo seja enviado novamente pela porta série.

```
/***************************************/
/*************** Main ******************/
int main()
{
	int var=1;

	/***************************************/
	/******** Configurar DVIC + UART ********/
	writeBus(var,dvicID,IRQ_USART_RECEIVE_enable);
	writeBus(var,dvicID,IRQ_USART_RECEIVE_prio);

	writeBus(var,usartID,USART_enable);
	/***************************************/

	while(1){}

	return 0;
}

/***************************************/
/************ Interrupções *************/
void USART_RX_ISR_vect(void)
{
	readBus(buf,usartID,USART_receive);
	writeBus(buf,usartID,USART_send);
}
/***************************************/
```

Figura 123 – Código de teste do *receiver*

Para este teste utilizou-se um terminal externo capaz de enviar *strings* pela porta série para o SoC. Na Figura 124 é possível ver o terminal antes e depois do envio da *string*.

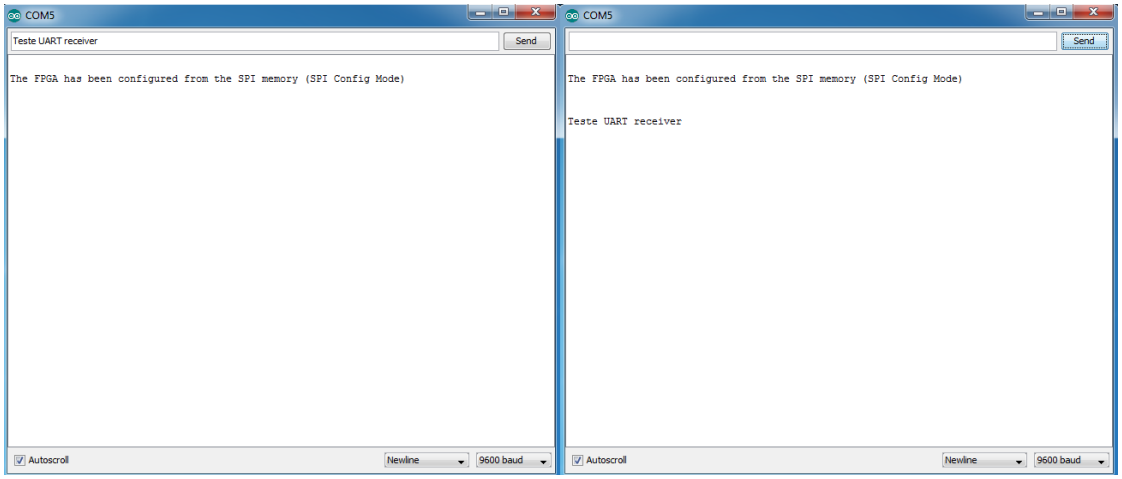


Figura 124 –Esquerda – *string* a ser enviada; Direita – *string* enviada

Na Figura 125, após validado que era capaz de receber e enviar caracteres pela porta série, verificou-se se não existiam erros no envio de determinados caracteres. Para isso enviou-se pela porta série as letras de a-z tanto em minúscula como em maiúscula e os símbolos mais comuns. Todos foram enviados corretamente pela porta série.

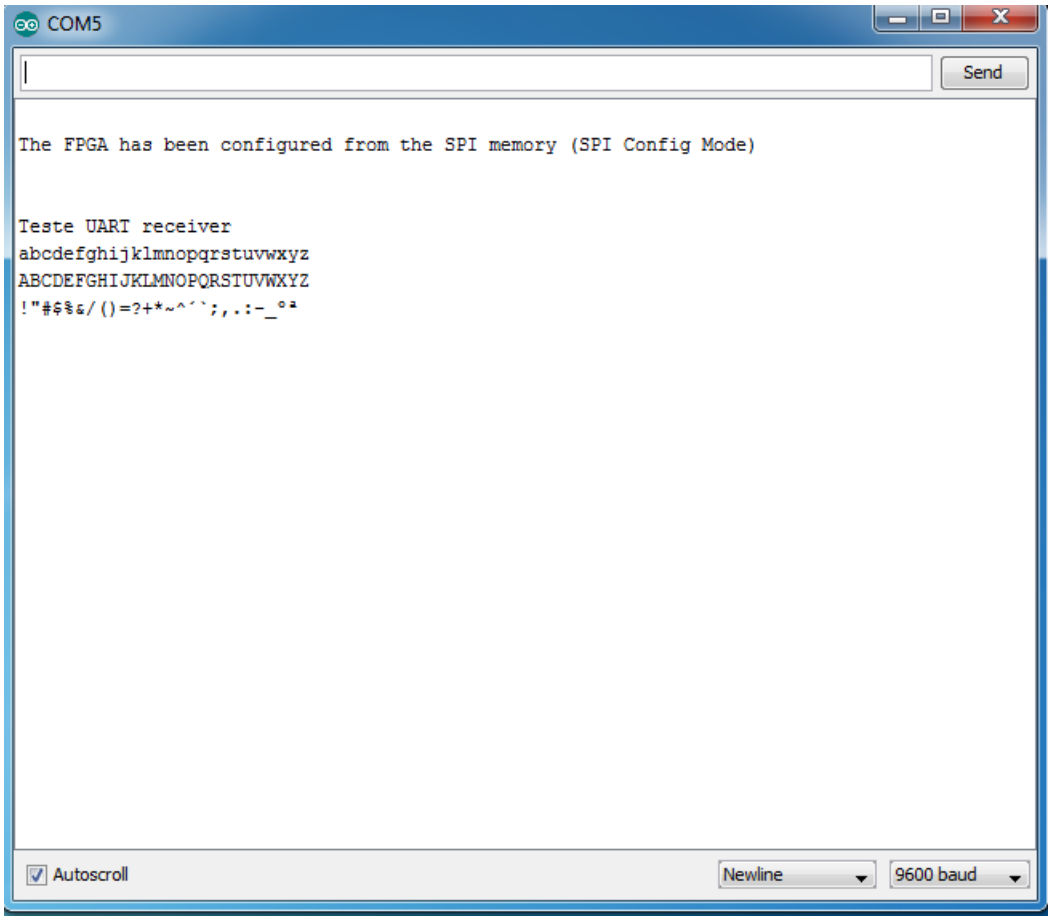


Figura 125 – Teste geral da UART

Com estes testes prova-se que o DVIC é também capaz de lidar com interrupções provenientes da UART, sejam elas do *receiver* ou do *transmitter*.

6.1.3.4. SYSTICK

Para testar este periférico, que foi criado para dar suporte a RTOS, utilizou-se um programa que faça uso de um RTOS. É possível visualizar na Figura 126 um trecho do código onde se encontra o que deve ser feito em cada tarefa e a *main* onde são feitas as configurações gerais, tais como criar tarefas e chamar o escalonador. Com este teste pretende-se provar que este

periférico é capaz de gerar uma interrupção a cada *x* tempo que serve para o escalonador, feito em *software*, seja capaz de fazer o escalonamento das tarefas.

```
/*************************************/
/*************** TASKS ***************/
void task1(void)
{
    char str[]={"TASK1"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(111);
    }
}

void task2(void)
{
    char str[]={"TASK2"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(222);
    }
}

void task3(void)
{
    char str[]={"TASK3"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(333);
    }
}
```

```
/*************************************/
/*************** MAIN ****************/
int main()
{
    uartConfig();

    genericTaskCreate(3,100);
    genericTaskCreate(2,100);
    genericTaskCreate(1,50);

    char str[]={"Start scheduler ..."};
    char *addr;
    addr=&str[0];
    println(addr);

    vStartSched();

    return 0;
}
/*************************************/
```

Figura 126 – Configuração do RTOS

Espera-se então que com este código seja enviado pela porta série primeiramente a *string* "Start scheduler…" e depois seja enviado o nome da tarefa que está a correr no momento, ou seja, se estiver a correr a tarefa 1 deve aparecer no terminal a *string* "TASK1". O resultado deste teste encontra-se apresentado na Figura 127, onde se pode ver que é impresso pela porta série a *string* inicial "Start scheduler…" e depois as diferentes tarefas vão sendo executadas, imprimindo cada uma o seu nome.

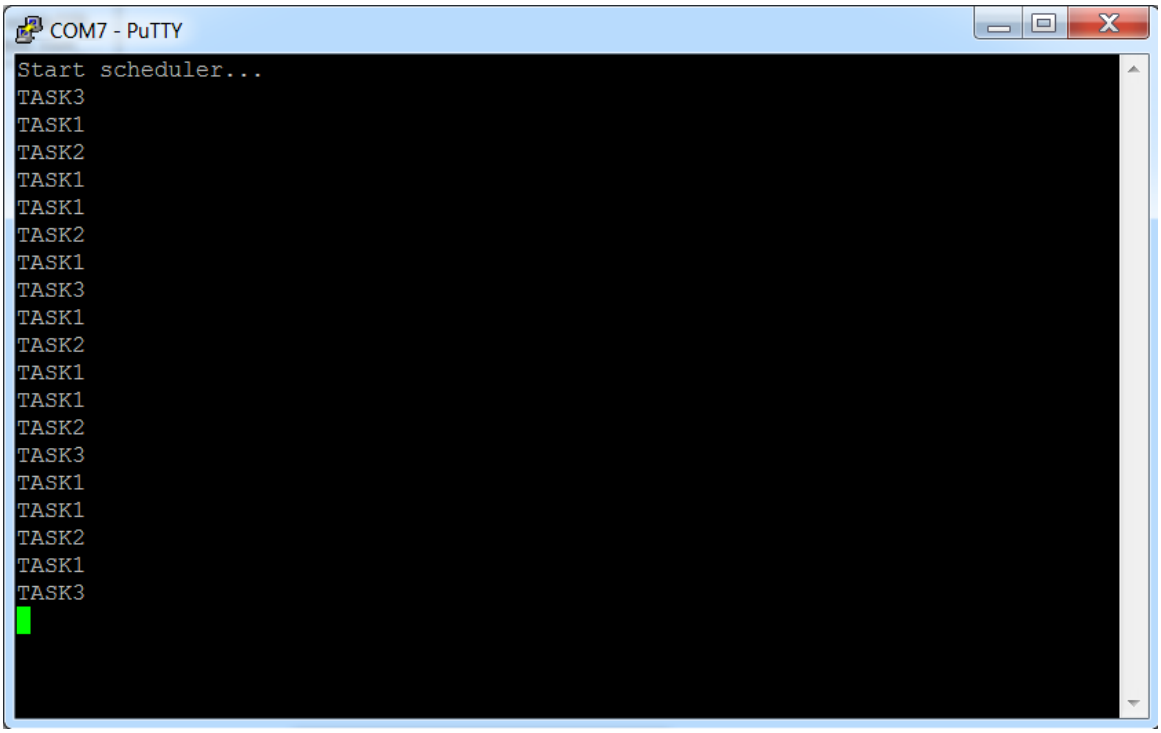


Figura 127 – Resultados do teste do RTOS

Este teste prova que o DVIC também é capaz de gerir a interrupção gerada pelo SYSTICK, pois este gera uma interrupção a cada *x* tempo para fazer o escalonamento das tarefas.

6.2. Teste de integração

Este teste tem por objetivo verificar se todo o sistema é capaz de operar corretamente estando tudo integrado num só. O código usado está apresentado na Figura 128, onde se pode ver quatro tarefas configuradas, as funções do RTOS, da interrupção do *timer* e da *main*. É configurado um *timer* para alternar o valor de saída do I/O, ou seja, dos *leds* e este valor é enviado para um terminal externo. Tudo isto acontece enquanto um RTOS está a correr quatro tarefas, onde três delas apenas imprimem o seu nome e a quarta tarefa é a responsável por alterar o dito valor de saída do I/O.

```
/************************************/
/************* Funções **************/
void task1(void)
{
    char str[]={"TASK1"};
    (…)
}

void task2(void)
{
    char str[]={"TASK2"};
    (…)}

void task3(void)
{
    char str[]={"TASK3"};
    (…)
}

void task4(void)
{
    char str[]= {"0xAA"};
    char str2[]={"0x55"};

    addr=&str[0];
    addr2=&str2[0];
    int out;

    while(1)
    {
       if(tempo > 1000)
       {
         tempo=0;
         if(state)
         {
            state=0;
            println(addr);
            out=0xAA;
            writeBus(out,parioId,write);
         }
         else
         {
            state=1;
            println(addr2);
            out=0x55;
            writeBus(out,parioId,write);
         }
       }
    }
}

void vTaskCaller(void)
{
  (…)
}

void configTimerArray(void)
{
  (…)
}
```

```
/************************************/
/************* MAIN ***************/
int main()
{
    uartConfig();
    configTimerArray();

    genericTaskCreate(4,200);
    genericTaskCreate(3,200);
    genericTaskCreate(2,200);
    genericTaskCreate(1,200);

    char str[]={"Start scheduler..."};
    char *addr;
    addr=&str[0];
    println(addr);

    vStartSched();

    return 0;
}

/************************************/
/********** Interrupções ************/
void TIMER3_ISR_vect(void)
{
    tempo++;
}
```

Figura 128 – Código do teste unitário

Sendo assim, espera-se que os *leds* alternem o seu estado entre 0x55 e 0xAA, o que é possível ver na Figura 129.

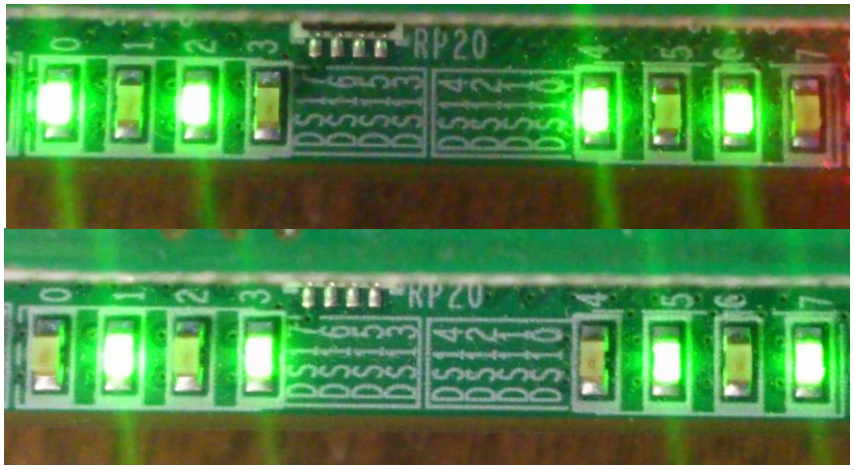


Figura 129 – Teste unitário; Cima – *leds* 0x55; Baixo *leds* 0xAA

O terminal está apresentado na Figura 130. Como se pode ver, após a *string* inicial "Start scheduler…", começam a ser imprimidas as tarefas que estão a ser executadas bem como o valor que é escrito na saída do I/O.

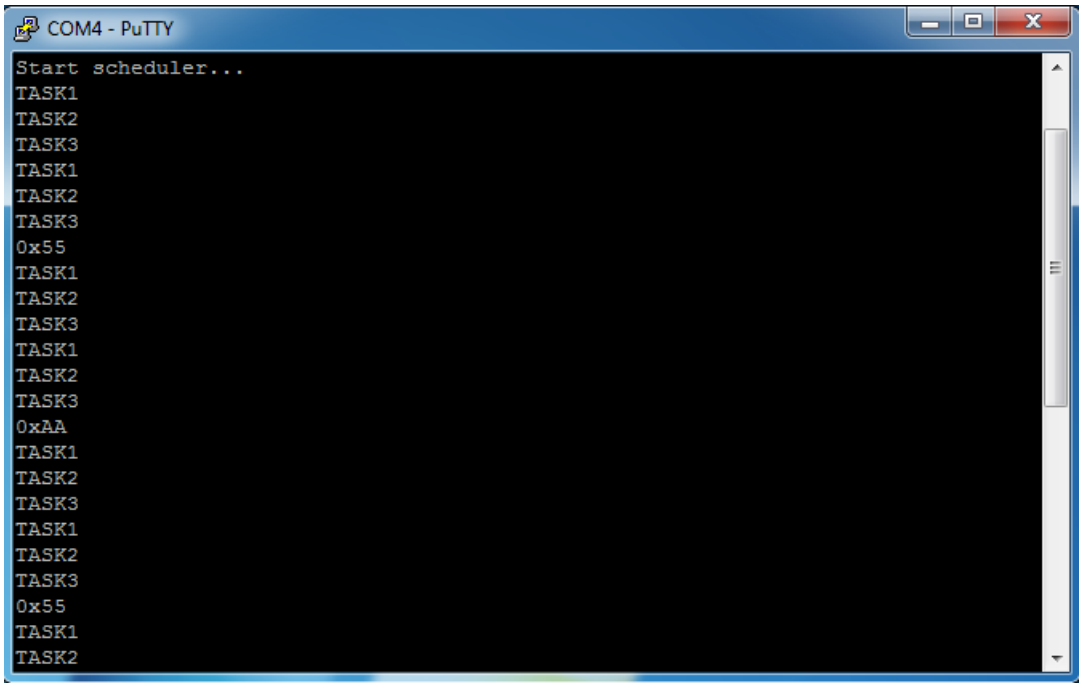


Figura 130 – Teste unitário; Terminal externo

Este teste prova que o sistema como um só funciona, pois foram utilizados todos os periféricos. O controlador I/O é utilizado para alternar o valor presente nos *leds*. Os *timers* dão o tempo para a alteração dos *leds*. O SYSTICK é utilizado para o escalonamento das tarefas do RTOS, ou seja, sempre que há um *overflow* é despoletada uma interrupção que tem de ser gerida pelo DVIC. Por fim tem-se a UART que envia informação pela porta série para um terminal externo.

O processador, que se encontra *pipelined*, bem como a *hazard unit* e o barramento também foram validados por este teste, pois o processador executou o código corretamente, a *hazard unit* foi capaz de gerir os vários *hazards* sem erros e o barramento fez a comunicação entre o processador e os periféricos.

Capítulo 7

CONCLUSÕES E TRABALHO FUTURO

7.1. Conclusões

Relembrando os objetivos, eles eram o desenvolvimento de um SoC com um processador baseado no *Microblaze*, implementação de uma *hazard unit* e inserção de periféricos de suporte a aplicações automóveis. Todos foram cumpridos, havendo inclusive a implementação de um maior número de periféricos do que os previstos inicialmente.

O processador encontra-se *pipelined* em cinco estágios para maiores velocidade e a comunicação entre cada estágio é feita através de um *buffer*. A *hazard unit* consegue responder de maneira adequada aos diversos *hazards* que surgem ao longo da execução de um código.

Implementou-se um barramento para a comunicação entre o processador e os periféricos. Este é capaz de fazer a comunicação entre eles corretamente.

Os periféricos também funcionam corretamente, como por exemplo o controlador de interrupções que é capaz de gerir as interrupções corretamente e sem erros, ou a UART que é capaz de comunicar com terminais externos.

Em suma, o SoC desenvolvido encontra-se *pipelined*, possui uma *hazard unit*, um barramento e cinco periféricos, tudo funcional.

7.2. Trabalho Futuro

Apesar de todos os objetivos terem sido alcançados existem bastantes pontos por onde melhorar este SoC. Um dos pontos principais são eventuais *bugs* que não tenham sido detetados, pois nunca se consegue descobrir a totalidade de *bugs* possuída pelo processador, ainda mais numa implementação *pipelined*. A *hazard unit* encontra-se a funcionar corretamente atualmente, mas como dito anteriormente podem existir *bugs* que não foram detetados logo terão que haver modificações na *hazard unit*. Terão de se correr *benchmarks* no SoC, pois com estes testes é possível simular aplicações usadas em sistema reais, o que irá verificar se está a funcionar corretamente, ou seja, se existem erros ou *bugs*.

O barramento implementado pode e deve ser melhorado. Existem diversos barramentos que podem ser implementados para o substituir obtendo um melhor desempenho, tais como um barramento Silicore's Whishbone, um barramento ARM AMBA ou um barramento IBM's CoreConnect.

A nível dos periféricos, todos podem sofrer melhorias. O DVIC pode ser estendido para passar a suportar um maior número de interrupções bem como funções que não foram implementadas, tais como o *tail-chaining*. Os *timers*, que são baseados nos *timers* da Tricore, não possuem todas as funcionalidades dos mesmos, por isso para trabalho futuro seria implementar as restantes funcionalidades. O controlador I/O, dependendo da placa de desenvolvimento pode também sofrer extensões. A UART pode ser estendida para suportar mais configurações.

REFÊNCIAS BIBLIOGRAFICAS

[1] S. Anthony, “SoC vs. CPU – The battle for the future of computing,” 2012. [Online]. Available: http://www.extremetech.com/computing/126235-soc-vs-cpu-the-battle-for-the-future-of-computing.

[2] M. Thompson, “FPGAs accelerate time to market for industrial designs,” 2004. [Online]. Available: http://www.eetimes.com/showArticle.jhtml?articleID=22102798.

[3] Xilinx, “FPGA vs. ASIC.” [Online]. Available: http://www.xilinx.com/fpga/asic.htm.

[4] W. Badawy, *System-on-Chip for real time applications*. United States of America: Kluwer Academic Publishers, 2003.

[5] D. Padole, P. Bajaj, M. Ieee, and G. H. R. College, “FUZZY ARBITER BASED MULTI CORE SYSTEM-ON-CHIP INTEGRATED CONTROLLER FOR AUTOMOTIVE SYSTEMS : A DESIGN APPROACH Professor , Senior Member IEEE , ECE Dept , G . H . Raisoni College of Engineering , Nagpur , India,” pp. 1937–1940.

[6] IBM Microelectronic, “IBM Advances Chip Technology With Breakthrough For Making Faster, More Efficient Semiconductors.” [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/2521.wss.

[7] G. P. Stein, E. Rushinek, G. Hayun, and a. Shashua, “A Computer Vision System on a Chip: a case study from the automotive domain,” *2005 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. - Work.*, vol. 3, pp. 130–130, 2005.

[8] T. Nguyen and A.-D. Basa, “Verification methodology of sophisticated automotive sensor interfaces integrated in modern system-on-chip airbag system,” *IECON 2013 - 39th Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 2335–2340, Nov. 2013.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Fourth. Morgan Kaufmann Publishers.

[10] C. L. G. Dumaguing, A. K. K. Khan, M. A. D. Parungao, A. B. Alvarez, and J. A. P. Reyes, “An Asynchronous Implementation of a 32-bit DLX Microprocessor,” no. Id, 2009.

[11] R.Selvakumar Rajagopal & Muhammad Mun’im Ahmad Zabidi, “FPGA Implementation of DLX Microprocessor With WISHBONE SoC Bus.”

[12] Xilinx, “MicroBlaze Soft Processor Core.” [Online]. Available: http://www.xilinx.com/tools/microblaze.htm.

[13] C. Chen, G. Novick, and K. Shimano, “RISC Architecture.” [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html.

[14] S. Craven, “OpenFIRE,” 2012. [Online]. Available: http://opencores.org/project,openfire2.

[15] A. Marschner, S. Craven, and P. Athanas, “A Sandbox for Exploring the OpenFire Processor.,” *ERSA*, 2007.

[16] ADAC, “SecretBlaze.” [Online]. Available: http://www.lirmm.fr/ADAC/?page_id=462.

[17] L. Barthe, L. V. Cargnini, P. Benoit, and L. Torres, “The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor,” *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011. .

[18] R. Balwaik, S. Nayak, and A. Jeyakumar, “Open-Source 32-Bit RISC Soft-Core Processors,” *iosrjournals.org*, vol. 2, no. 4, pp. 43–46, 2013.

[19] Lattice Semicondutor, “LatticeMico32 Open, Free 32-Bit Soft Processor.” [Online]. Available: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx.

[20] Gaisler Research, “Leon4 Processor.” [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon4.

[21] ORSoC, “OpenRISC.” [Online]. Available: http://www.rte.se/blog/blogg-modesty-corex/openrisc-1200-soft-processor.

[22] S. Tan, “AEMB 32-bit Microprocessor Core Datasheet.” [Online]. Available: http://opencores.org/websvn,filedetails?repname=aemb&path=%2Faemb%2Ftrunk%2Fdoc%2FaeMB_datasheet.pdf&rev=20.

[23] S. Tan, “aeMB.” [Online]. Available: http://opencores.org/project,aemb.

[24] “As Optimizações com o Pipelining.” [Online]. Available: http://www.dcc.fc.up.pt/~zp/aulas/9899/me/trabalhos/alunos/Processadores/pipelining/main.htm.

[25] Xilinx, “FPGA (Field Programmable Gate Array).” [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm.

[26] “FPGA Architecture for the Challenge.” [Online]. Available: http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html.

[27] M. Thompson, “Mixed-signal FPGAs provide GREEN POWER,” 2007. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1271543.

[28] Xilinx, “Virtex-7 FPGA Family.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm.

[29] Xilinx, “THE XILINX VIRTEX-7 FPGA FAMILY: UNLEASHING PERFORMANCE AND INNOVATION WITH HIGH-DENSITY, LOW-POWER 28NM TECHNOLOGY.”

[30] National Instruments, “Advantages of the Xilinx Virtex-5 FPGA,” 2014. [Online]. Available: http://www.ni.com/white-paper/7440/en/#toc1.

[31] Xilinx, “Kintex-7 FPGA Family.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/kintex-7/.

[32] Xilinx, “INTRODUCING A NEW CLASS OF FPGAS: AN IDEAL BALANCE OF INTEGRATION, PRICE, PERFORMANCE, AND POWER.” [Online]. Available: http://www.xilinx.com/publications/prod_mktg/Kintex-7-Product-Brief.pdf.

[33] Xilinx, “Artix-7 FPGA Family.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/artix-7/index.htm.

[34] Xilinx, “XILINX ARTIX-7 FPGAS: A NEW PERFORMANCE STANDARD FOR POWER-LIMITED, COST-SENSITIVE MARKETS.” [Online]. Available: http://www.xilinx.com/publications/prod_mktg/Artix-7-Product-Brief.pdf.

[35] Xilinx, “Spartan-6 FPGA Family.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/spartan-6/.

[36] Xilinx, “BALANCING COST, SPACE, POWER AND PERFORMANCE.” [Online]. Available: http://www.xilinx.com/publications/prod_mktg/Spartan6_Product_Brief.pdf.

[37] ARM, “Cortex-M Series.” [Online]. Available: http://www.arm.com/products/processors/cortex-m/.

[38] Infineon, “TriCore™ Architecture & Core.” [Online]. Available: http://www.infineon.com/cms/en/product/microcontrollers/32-bit-tricore-tm-microcontrollers/tricore-tm-architecture-and-core/channel.html?channel=ff80808112ab681d0112ab6b73d40837.

[39] “Serial Communication.” [Online]. Available: https://learn.sparkfun.com/tutorials/serial-communication.

[40] National Instruments, “Serial Communication General Concepts.” [Online]. Available: http://digital.ni.com/public.nsf/allkb/2AD81B9060162E708625678C006DFC62.

[41] J. Serrano, “Introduction to FPGA design,” *8th Work. Electron. LHC Exp.*, pp. 231–247, 2004.

[42] “Xilinx Development System Reference Guide.” [Online]. Available: http://www.xilinx.com/itp/xilinx10/books/docs/dev/dev.pdf

[43] Florida International University, “CCLI: NOVEL INSTRUCTION MATERIAL DEVELOPMENT FOR EMBEDDED SYSTEM EDUCATION IN

UNDERGRADUATE CURRICULUM." [Online]. Available: http://arcs-lab.eng.fiu.edu/projects/lab-material/.

[44] Xilinx, "Xilinx University Program XUPV5-LX110T Development System." [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm.

[45] Xilinx, "Virtex-5 LXT FPGA ML505 Evaluation Platform." [Online]. Available: http://www.xilinx.com/products/boards-and-kits/HW-V5-ML505-UNI-G.htm.

[46] "Introduction To RS232 Serial Communication," 2012. [Online]. Available: http://wcscnet.com/Tutorials/SerialComm/Page1.htm.

ANEXOS

ANEXO A

Tabela 10 – Instruções implementadas

	Mnemónica	Descrição
ADD	ADD Rd,Ra,Rb	Adiciona Ra e Rb e guarda no Rd o resultado
ADDI	ADDI Rd,Ra,Imm	Adiciona Ra e Imm e guarda no Rd o resultado
SUB	SUB Rd,Ra,Rb	Subtrai o Rb ao Ra e guarda no Rd o resultado
SUBI	SUBI Rd,Ra,Imm	Subtrai o Imm ao Ra e guarda no Rd o resultado
CMP	CMP Rd,Ra,Rb	Compara o valor no Ra com o valor no Rb e guarda no Rd o resultado
MUL	MUL Rd,Ra,Rb	Multiplica o Ra e o Rb e guarda Rd o resultado
OR	OR Rd,Ra,Rb	Operação lógica OR entre o Ra e Rb e guarda no Rd o resultado
ORI	ORI Rd,Ra,Imm	Operação lógica OR entre o Ra e Imm e guarda no Rd o resultado
AND	AND Rd,Ra,Rb	Operação lógica AND entre o Ra e Rb e guarda no Rd o resultado
ANDI	ANDI Rd,Ra,Imm	Operação lógica ANDI entre o Ra e Imm e guarda no Rd o resultado
XOR	XOR Rd,Ra,Rb	Operação lógica XOR entre o Ra e Rb e guarda no Rd o resultado
XORI	XORI Rd,Ra,Imm	Operação lógica XORI entre o Ra e Imm e guarda no Rd o resultado

NOT	NOT Rd	Operação lógica NOT sobre o Rd e valor guardado novamente no mesmo registo
BR	BR Rb	*Branch* relativo
BRI	BRI Imm	*Branch* relativo Imm
BRA	BRA Rb	*Branch* absoluto
BRAI	BRAI Imm	*Branch* absoluto Imm
LW	LW Rd,Ra,Rb	Fazer o *load* do valor na posição de memória dada pela soma de Ra e Rb e guardar em Rd
LWI	LWI Rd,Ra,Imm	Fazer o *load* do valor na posição de memória dada pela soma de Ra e Imm e guardar em Rd
LWBI	LWBI Rd,Ra,Imm	Fazer o *load* do valor na posição de memória dada pela diferença entre Ra e Imm e guardar em Rd
SW	SW Rd,Ra,Rb	Fazer o *store* de Rd na posição de memória dada pela soma de Ra e Rb
SWI	SWI Rd,Ra,Imm	Fazer o *store* de Rd na posição de memória dada pela soma de Ra e Imm
SWBI	SWBI Rd,Ra,Imm	Fazer o *store* de Rd na posição de memória dada pela diferença entre Ra e Imm
BS	BS Rd,Ra,Rb[4:0]	Fazer o *shift* ao Ra o valor em Rb e guardar em Rd
IMM	IMM Imm	Instrução auxiliar para valores de 32 bits
BEQ	BEQ Ra,Rb	*Branch* caso resultado de instrução CMP seja 0

BNE	BNE Ra,Rb	*Branch* caso resultado de instrução CMP seja diferente de 0
BGE	BGE Ra,Rb	*Branch* caso resultado de instrução CMP seja 0 ou 2
BGT	BGT Ra,Rb	*Branch* caso resultado de instrução CMP seja 2
BLE	BLE Ra,Rb	*Branch* caso resultado de instrução CMP seja 0 ou 1
BLT	BLT Ra,Rb	*Branch* caso resultado de instrução CMP seja 1
MOV	MOV Rd,Ra	Mover o valor do registo A para o registo D
MOVI	MOVI Rd,Imm	Mover um valor imediato para o registo D
CALL	CALL Imm	Chamar uma função
RET	RET	Retornar de uma função
RETI	RETI	Retornar de um interrupção
PUSH	PUSH Rd	Fazer o *push* ao registo Rd
POP	POP Rd	Fazer o *pop* ao registo Rd
PERACC	PERACC	Aceder a um periférico