Universidade do Minho
Escola de Engenharia

Pedro Leonel Couto de Oliveira

Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux

outubro de 2013

UMinho | 2013
Pedro Leonel Couto de Oliveira
Sistema de Aquisição de
Sinais em Tempo Real Baseado em Linux

Universidade do Minho
Escola de Engenharia

Pedro Leonel Couto de Oliveira

Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux

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

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Miguel Nunes Santos Cabral

outubro de 2013

DECLARAÇÃO

Pedro Leonel Couto de Oliveira
Endereço eletrónico: pelco89@gmail.pt   Telefone: 916881326
Número do Bilhete de Identidade: 13809970
Título da Tese:

Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux

Orientador: Professor Doutor Jorge Miguel Nunes Santos Cabral
Ano de conclusão: 2013
Tese submetida na Universidade do Minho para a obtenção do grau de
Mestre em Engenharia Eletrónica Industrial e de Computadores
Área de Especialização: Sistemas Embebidos.
Departamento de Eletrónica Industrial.

Universidade do Minho, ____/____/_______

Assinatura: ____

Agradecimentos

As primeiras palavras de agradecimento vão para o supervisor desta tese, Prof. Jorge Cabral, pelo apoio, disponibilidade e sugestões nos momentos mais importantes. Aos Professores Paulo Cardoso e Adriano Tavares, e ao grupo de pessoas que constitui o Laboratório de Sistemas Embebidos da Universidade do Minho, por todo apoio manifestado no decorrer desta dissertação.

O presente trabalho não teria sido realizado sem o apoio do Engenheiro Vítor Silva, de quem recebi amizade, sugestões e disponibilidade, que tornaram menos árida e mais aliciante a sua elaboração. Aos meus colegas de curso, companheiros de longas noites, pela sua amizade, pelos bons momentos que partilhamos e pela forma como contribuíram ao longo destes anos.

Gostaria também de agradecer ao corpo de funcionários das oficinas deste departamento. Um palavra de especial destaque para os funcionários Carlos Torres, Joel Almeida e Ângela Macedo, três exemplos de empenho e competência que gostaria de assinalar e agradecer, de modo muito particular.

Quero também dirigir uma palavra de apreço aos meus familiares e amigos mais próximos que, durante este trabalho, dedicaram apoio e disponibilidade. À minha Mãe e ao meu Pai por me terem educado e dado tudo que eu precisava. Tenho muito orgulho de ser vosso filho.

Agradeço, ainda e sobretudo, à Ana, pelo apoio e pela forma como soube compreender as minhas prolongadas ausências e os meus horários pouco ortodoxos.

A todos o meu muito obrigado.

Resumo

Vivemos na era digital mas o mundo real que conhecemos é analógico. Na fronteira entre os dois mundos, residem os sistemas de aquisição de sinais, responsáveis por quantificar a informação relevante do mundo que nos rodeia, o mundo analógico, no formato adequado, compatível com o mundo digital das aplicações de software e sistemas de controlo. A necessidade de eficiência nos processos, exatidão, repetibilidade, são fatores de impulsão na automatização de sistemas, a necessidade de obter a informação relevante, acerca de determinadas tarefas, da qual dependem decisões que o sistema de controlo automático deve tomar, lançam a questão de quando, como e que parte dessa informação seria necessária para caracterizar um sistema automático, tornando-o capaz de responder a um determinado conjunto de operações de forma autónoma. Sistemas de aquisição de sinais são normalmente subsistemas de um sistema maior, sendo muitas vezes modulados na forma de sistemas embebidos. A necessidade de segurança, proteção, confiabilidade e disponibilidade, dos sistemas embebidos, pode ser proporcionado pelo ambiente de um sistema operativo. O crescente sucesso do sistema operativo Linux e a contínua evolução ao longo das décadas, levou a sua difusão no mundo dos sistemas embebidos. Alguns sistemas embebidos apresentam requisitos de tempo real, como são exemplos o controlo de alguns processos na indústria moderna, sistemas usados na aviação, indústria espacial, automóvel, entre outros. Neste tipo de sistemas, a falha no cumprimento de um requisito, como é exemplo o *deadline* na execução de determinada tarefa, pode comprometer a integridade do sistema de controlo, sendo que neste caso, o sistema entra em falha por definição. Nesta dissertação, pretende-se desenvolver um sistema de aquisição de sinais, com requisitos de tempo real, baseado em Linux. A validação deste sistema, será verificada através da integração deste no domínio dos sistemas de energia, sendo considerados como sistemas de tempo real e que possuem requisitos incompatíveis com a grande maioria dos sistemas operativos da atualidade, sendo que muitas vezes, estes requisitos apenas podem ser satisfeitos através de *hardware* dedicado.

Abstract

We live in the digital age but the real world that we know is analog. On the boundary between the two worlds, lies the signal acquisition systems, responsible for quantifying the relevant information from the world around us, the analog world, in the proper format, compatible with the digital world of software applications and control systems. The need for process efficiency, accuracy, repeatability, are impelling factors in automation systems, the need to obtain relevant information about certain tasks, which depend on decisions that the automatic control system should take, introduce the question when, how and how much of that information would be needed to characterize an automatic system, making it capable of responding to a given set of operations autonomously. Signal acquisition systems are typically subsystems of a larger system, often modulated as embedded systems. The need for safety, security, reliability and availability of embedded systems can be provided by an operating system environment. The growing success of the Linux operating system and continuous evolution over the decades, led to its spread in the world of embedded systems. Some embedded systems have real-time requirements, as are examples of some of the control processes in modern industry, systems used in aviation, space industry, automotive, among others. In such systems, the failure to comply with a requirement, as is the example deadline in execution of certain task, can compromise the integrity of the control system, in which case, the system goes into failure by definition. In this dissertation, is intended to develop a signal acquisition system with real-time requirements, based on Linux. The validation of this system, will be verified by its incorporation in the field of power systems, being considered as real-time systems and the requirements are incompatible with most operating systems, and many times, these requirements can only be satisfied by dedicated hardware.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

ABS	*Anti-lock Braking System*
ADC	*Analog-to-Digital Converter*
ALU	*Arithmetic Logic Unit*
ASIC	*Application-Specific Integrated Circuit*
BSP	*Board Support Package*
CAN	*Controller Area Network*
CC	Corrente Contínua
CISC	*Complex Instruction Set Computer*
CPU	*Central processing unit*
DAC	*Digital-to-Analog Converter*
DDR	*Double Data Rate*
DMA	*Direct Memory Access*
DOS	*Disk Operating System*
DSP	*Digital Signal Processor*
DVD	*Digital Versatile Disc*
DVD-RW	*Rewritable Digital Versatile Disc*
FAP	Filtro Ativo Paralelo
FAT	*File Allocation Table*
FFT	*Fast Fourier Transform*
FPGA	*Field-Programmable Gate Array*
FSL	*Fast Simpex Link*
GPIO	*General-Purpose Input/Output*
GPOS	*General Purpose Operating System*
GUI	*Graphical User Interface*
HDL	*Hardware description language*
HDMI	*High-Definition Multimedia Interface*
HPC	*High-Performance Computing*
HTTP	*Hypertext Transfer Protocol*

ICU	*Instruction Cache Unit*
IDE	*Integrated Development Environment*
IPS	*Instructions Per Second*
IRQ	*Interrupt Request*
ISR	*Interrupt Service Routine*
JTAG	*Joint Test Action Group*
LAN	*Local Area Network*
MIPS	*Million Instructions Per Second*
MMC	*MultiMediaCard*
MMU	*Memory Management Unit*
NTFS	*New Technology File System*
PC	*Personal Computer*
PCB	*Printed Circuit Board*
PID	*Proportional-Integral-Derivative*
PLB	*Processor Local Bus*
POSIX	*Portable Operating System Interface*
PTC	*Positive Temperature Coefficient*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
ROM	*Read-Only Memory*
RT	*Real-Time*
RTD	*Resistance Temperature Detector*
RTOS	*Real-Time Operating System*
SAS	Sistemas de Aquisição de Sinais
SD	*Secure Digital*
SPI	*Serial Peripheral Interface*
SO	Sistema Operativo
SoC	*System-on-Chip*
TCP	*Transmission Control Protocol*
TCS	*Traction Control Cystem*
USART	*Universal Synchronous/Asynchronous Receiver/Transmitter*
USB	*Universal Serial Bus*
VGA	*Video Graphics Array*
WAN	*Wide Area Network*
WLAN	*Wireless Local Area Network*
XUP	*Xilinx University Program*

Capítulo 1

Introdução

Neste capítulo é contextualizado o âmbito desta dissertação, bem como definidos a motivação e objetivos. São depois apresentadas as contribuições deste trabalho, finalizando o capítulo com a organização da dissertação.

1.1 Contextualização

Aquisição de sinal é o processo onde fenómenos físicos do mundo real são convertidos num formato digital para posterior análise, armazenamento e processamento em sistemas computacionais. Os sistemas de aquisição de sinais permitem registar uma vasta gama de sinais do mundo real como pressão, fluxo e temperatura. A forma como os sinais são captados, quantidade de informação que é captada, como essa informação é disponibilizada e a porção de tempo que a informação demora a chegar no seu formato digital desde a sua ocorrência no mundo real, são principais características que permitem distinguir os sistemas de aquisição de sinais. Na maioria das aplicações, os sistemas de aquisição de sinais são desenvolvidos não só para fazer aquisição, mas também para atuar sobre esses sinais. Esta habilidade acoplada com a facilidade de realizar interface com outros sistemas computacionais utilizando protocolos como USART, I2C, I2S e SPI torna estes sistemas ainda mais apelativos.

Os sistemas de aquisição de sinais são muitas vezes implementados na forma de um subsistema de um sistema maior, e assume muitas vezes a natureza de um sistema embebido. Estes, os sistemas embebidos, são uma vertente dos sistemas informáticos, cuja parametrização se define com um propósito específico, possuindo apenas recursos de *hardware* e *software* estritamente necessários, de modo que, muitas das vezes o utilizador nem se apercebe da sua existência. Alguns sistemas embebidos apresentam requisitos de tempo real, como são exemplos o controlo de alguns processos na indústria moderna, sistemas usados na aviação, indústria espacial, automóvel, entre outros. Estes sistemas são muitas vezes considerados sistemas do tipo *hard*, onde são utilizadas técnicas e procedimentos rígidos para promover soluções bem definidas e ausentes de ambiguidades. Na Figura 1.1 está representado alguns setores industriais onde se pode encontrar sistemas deste tipo.

Figura 1.1: Setores industriais com requisitos de tempo real.

De forma a satisfazer os requisitos destes setores, estes exigem muitas vezes amostragens simultâneas, fiabilidades nas amostras e latências reduzidas nos sistemas de aquisição de sinais, com funcionalidades estendidas como por exemplo capacidade de armazenamento em massa, ligação USB e Ethernet. A necessidade de satisfazer todos os requisitos anteriores e proporcionar a estes sistemas proteção, acesso seguro aos dados, facilidade no controlo e gestão dos periféricos em *hardware*, e bom funcionamento, pode ser proporcionado pelo ambiente de um sistema operativo.

Dentro dos vários sistemas operativos, comerciais ou livres, o Linux tem vindo a destacar-se e a atrair cada vez mais utilizadores, sendo cada vez mais o sistema operativo de suporte aos sistemas embebidos.

Uma solução que beneficia de uma distribuição do SO Linux, modulada especificamente para aplicações com requisitos de tempo real, do tipo *hard*, apenas fornece o conjunto de serviços estritamente necessários ao bom funcionamento do mesmo. A junção sistema de aquisição de sinais e o ambiente de execução Linux, na versão modulada para tempo real, configura uma solução capaz de satisfazer os requisitos da grande maioria dos sistemas de controlo automáticos, no domínio dos sistemas computacionais, e pode ser usada na forma genérica e independente da plataforma, beneficiando da virtualização do *hardware* que o SO Linux lhe confere.

A utilização de sistemas automáticos em ambientes de SO como o Linux, potencia a utilização de *software* legado e a integração de bibliotecas de *software* do domínio da aplicação, facto que se tem vindo a revelar indispensável, devido à crescente complexidade e ao reduzido *time-to-market* observado na indústria deste tipo de sistemas. A validação do sistema de aquisição de sinais, no domínio dos sistemas do tipo *hard* sendo considerados como sistemas de tempo real, que possuem requisitos temporais e incompatíveis com a grande maioria dos SO da atualidade, sendo que muitas vezes, através de *hardware* modular dedicado, como é exemplo a aplicação de DSPs programáveis.

1.2 Motivação e Objetivos

Sistemas embebidos são o motor da inovação e do crescimento em muitos setores da indústria como são exemplos a indústria automóvel, aeroespacial, telecomunicações, automação e doméstica. O desenvolvimento de sistemas embebidos implica o domínio das ferramentas de forma a aumentar produtividade no sistema, garantir previsibilidade no sistema, segurança e confiabilidade. A utilização de um sistema operativo, como o Linux para complementar estes sistemas, possibilita a abstração da camada de *hardware* reduzindo drasticamente o *time-to-market*. A variedade de distribuições Linux e técnicas de performance do sistema operativo, vai permitir aprofundar o conhecimento na área dos sistemas embebidos que sempre despertou o interesse do autor.

O projeto destes sistemas focados em aspetos de tempo real, exige domínio no desenvolvimento de *software* dependente de *hardware* e de *hardware* dependente de *software*. A utilização de métodos como *Hardware-Software Co-Design*, são obrigatórios no desenvolvimento de sistemas com requisitos extremamente exigentes de forma a satisfazer o sistema global ao nível do desempenho, consumo de energia, confiabilidade e segurança o que torna esta área aliciante. Estes métodos requerem competências novas que combinam o conhecimento de *software* com o de *hardware*.

1.3 Contribuições

A crescente complexidade, do ponto de vista do desempenho exigido às plataformas de *hardware* dedicadas, das soluções projetadas e utilizadas em processos industriais, controlo de robôs, sistemas de aquisição de sinais e uma variedade de equipamento que desempenham tarefas do tipo *time-sensitive*, faz disparar o custo total das soluções e aumentar o *time-to-market* em virtude do esforço de engenharia requerido às equipas de desenvolvimento. Na grande maioria das aplicações mencionadas, os custos com a plataforma de *hardware* e do sistema operativo podem ser significativamente reduzidos utilizando uma distribuição Linux modelada especificamente para cada um dos cenários de aplicação. Espera-se que a identificação dos fatores chave que influenciam as tarefas do tipo *time-sensitive* contribua ainda para a redução do esforço de engenharia, e consequentemente no *time-to-market*, exigido no desenvolvimento deste tipo de aplicações.

1.4 Organização da Dissertação

Neste documento é descrito o desenvolvimento de um sistema de aquisição de sinais em tempo real baseado no sistema operativo Linux. Este documento está dividido em seis capítulos que serão apresentados de seguida.

No segundo capítulo é feito um levantamento teórico das tecnologias onde esta dissertação se insere. Inicialmente é feita uma pequena descrição da evolução histórica dos sistemas de aquisição. São apresentados e descritos alguns sistemas de aquisição mais modernos. Segue-se uma descrição dos fundamentos de um sistema de aquisição, classificação dos sinais, definição de sistema embebido e tecnologia FPGA. Posteriormente pretende-se inserir o sistema operativo Linux no mundo dos sistema embebidos, vantagens da sua utilização e introduzir o conceito de tempo real.

No capítulo três é descrito a plataforma que serviu de base para o desenvolvimento desta dissertação. Será feita uma breve descrição das componentes essenciais da plataforma como as ferramentas de suporte.

O *core* desta dissertação concentra-se no capítulo quatro. Pretende-se no início descrever o sistema de aquisição no seu geral, posteriormente será feita a descrição detalhada *bottom-up* do sistema, começando pelo conversor analógico-digital e terminando na aplicação de monitorização no espaço utilizador.

O quinto capítulo apresenta alguns cenários de aplicação onde o sistema de aquisição foi aplicado. No início é demonstrado um teste seguro para depurar o sistema e assegurar o seu funcionamento correto antes de avançar para a próxima fase. No final, o sistema é testado no domínio dos sistemas de energia, onde é demonstrado o funcionamento da teoria p-q em *hardware*.

O documento termina com a discussão de resultados e conclusões acerca do trabalho desenvolvido no sexto capítulo.

Capítulo 2

Estado da Arte

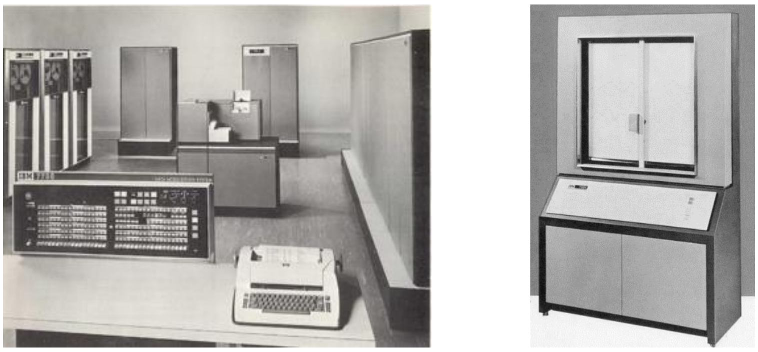
Neste capítulo é apresentada uma visão geral sobre as tecnologias e conceitos abordados nesta dissertação. Inicialmente é descrito a evolução dos sistemas de aquisição e dado alguns exemplos destes sistemas comerciais. Posteriormente será feita uma descrição dos elementos básicos que constituem um sistema de aquisição de sinais. Segue-se a classificação dos sinais analógicos em formato digital e definição de taxa de amostragem. Também faz parte neste capítulo apresentar a definição de sistemas embebidos, descrição da tecnologia FPGA e sua vantagem. Por fim, é apresentado algumas das vantagens da utilização do sistema operativo Linux nos sistemas embebidos, bem como a definição de tempo real.

2.1 Evolução dos Sistema de Aquisição

2.1.1 IBM 7700 *Data Acquisition System*

A IBM (International Business Machines Corporation) produziu em 1963 o primeiro computador especializado em aquisição de dados. O IBM 7700 representado na Figura 2.1(a), tinha capacidade de receber dados de 32 fontes distintas, processá-las e enviar para 16 equipamentos remotos como por exemplo as unidades gráficas IBM 7404 Figura 2.1(b).

O IBM 7700 tinha uma arquitetura com *words* de 18-bits, mas as instruções ocupavam duas *words*. Cada instrução demorava 2 ou 3 ciclos de máquina expecto a multiplicação e a divisão que demoravam 8 e 12 ciclos respetivamente. Naquele tempo o IBM 7700 era um computador bastante rápido, o tempo de cada ciclo



(a) (b)

Figura 2.1: Computador IBM 7700 em (a) e Unidade gráfica IBM 7404 em (b).

de máquina era de 2 micro-segundos. O espaço de endereçamento podia ir até 262.144 *words* que correspondia a uma capacidade de aproximadamente de 7280 instruções. Este computador foi rapidamente substituído pelo IBM 1800 *Data Acquisition and Control System* em 1964 (IBM Data Processing Division) por este ser mais rápido, bastante mais compacto e por ter maior número de interfaces para entrada/saída.

2.1.2 NI LabVIEW e *Plug-in Boards*

Em 1986 a NI (National Instruments Corporation) lança o LabVIEW. O LabVIEW é plataforma de desenvolvimento gráfica que permite aos utilizadores "programar" graficamente através de junção de blocos. Ao introduzir LabVIEW a NI permitiu uma revolução nos sistemas de aquisição baseados em computadores.

Entre 1986 e 1988, a NI lança as primeiras *plug-in boards* para Macintosh e IBM PC, permitindo aos utilizadores fazer medições e adquirir dados diretamente para os seus computadores. Devido à flexibilidade oferecida pelo LabVIEW e as *plug-in boards*, estes produtos começaram a surgir cada vez mais na indústria, engenheiros podiam substituir material de instrumentação de elevado custo por um sistema computacional que tinha a capacidade de adquirir, analisar e processar dados a custo muito mais baixo.

Ao longo das últimas décadas os produtos da NI tornaram-se bastante conceituados pelos engenheiros e cientistas. A NI oferece, hoje, uma vasta gama de produtos e ferramentas para medição e análise para uma grande variedade de cenários de aplicação.

Os próximos dois produtos são exemplos de sistemas de aquisição mais populares desenvolvidos pela NI.

2.1.2.1 NI USB-6009

A variedade de produtos e inovações da National Instruments levou uniformização da comunicação USB nas aplicações de medição. O NI USB-6009, Figura 2.2, é um sistema de aquisição *low-cost*, de tamanho reduzido e de fácil instalação. As vantagens da ligação *plug-and-play* USB ao computador e a compatibilidade do sistema com o LabVIEW, permite iniciar e modelar um sistema de medição e controlo de forma mais rápida.

Figura 2.2: National Instruments *Plug-in board* NI USB-6009

O NI USB-6009 apresenta as caraterísticas essenciais que um sistema de aquisição de sinais deve ter, este permite a ligação de 8 canais analógicos para medição de sinais, 2 canais analógicos para geração de formas de onda e 12 entradas/saídas digitais que podem ser usadas como *trigger* ou para uso genérico. Em suma, as principais características deste produto são:

- 8 entradas analógicas (14-bit, 48 kS/s);
- 2 saídas analógicas (12-bit, 150 S/s);
- 12 E/S digitais;
- Contador 32-bit;
- *Bus-powered* para elevada mobilidade;
- Compatível com o LabVIEW;

A portabilidade do dispositivo torna-o essencial para experiências académicas. O NI USB-6009 é acessível para estudantes e ainda podem obter uma licença gratuita para o LabVIEW.

2.1.2.2 NI USB-6343

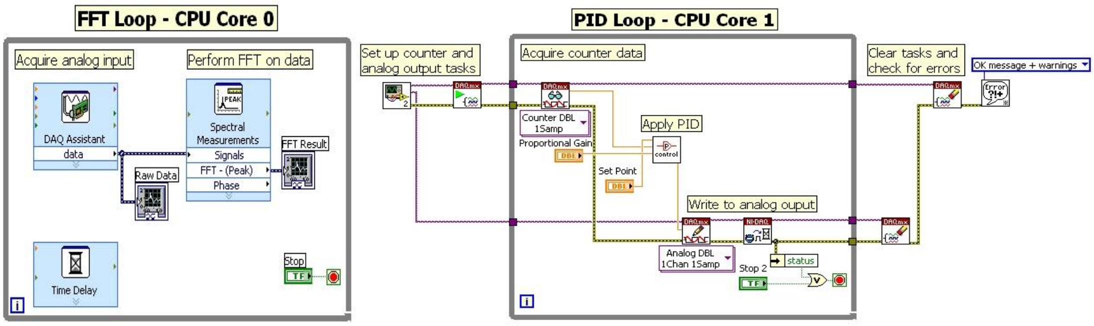
O NI USB-6343, representado na Figura 2.3, faz parte dos produtos *X Series Multifunction Data Acquisition*. Os produtos *X Series* da NI são os sistemas de aquisição de dados mais avançados alguma vez desenvolvidos pela NI. O NI USB-6343 é bastante mais caro que o anterior mas apresenta características únicas e essenciais que mais nenhum produto oferece no mercado.

Figura 2.3: National Instruments *Plug-in board* NI USB-6343

A principal caraterística deste produto está no *timing*, *triggering* e otimizações para uso nos computadores *multicore*. Isto significa que é possível fazer processamento paralelo em computadores *multicore*, ao tirar partido dos subsistemas de *hardware* paralelo destes produtos e ao ser possível desenvolver aplicações *multithread* no LabVIEW. Na Figura 2.4 está representado um exemplo de uma aplicação em LabVIEW onde o processamento da *thread* FFT é da responsabilidade do *core 0* e a *thread* PID da responsabilidade do *core 1* do CPU do computador.

A NI USB-6343 é um produto que consegue satisfazer quase qualquer cenário de aplicação, desde o simples registo de sinais até à monitorização de processos exigentes da indústria. Resumidamente, as principais características deste produto são:

- 32 entradas analógicas (16-bit, 500 kS/s);
- 4 saídas analógicas (16-bit, 900 kS/s);



(a) (b)

Figura 2.4: Aplicação *multithread* no LabVIEW em computador *multicore* (National Instruments Corporation).

- 48 E/S digitais;
- 4 contadores/*timers* 32-bit;
- Sistema avançado de *trigger*;
- Compatível com o LabVIEW;

A única desvantagem deste produto é mesmo o valor comercial e por isso não está acessível para qualquer pessoa. O custo adicional da ferramenta LabVIEW torna o sistema global bastante caro.

2.1.3 HBM - *Genesis HighSpeed*

A empresa HBM (HBM Corporation) foi fundada em 1950 e hoje é líder no mercado em tecnologia de pesagem, teste e medição na indústria aeroespacial, automóvel, farmacêutica e agricultura. Ao contrário da National Instruments que apresenta uma variedade de produtos acessíveis para a comunidade académica e outros produtos para a indústria, a HBM apenas está focada ao nível industrial.

Os produtos, Genesis, da HBM, apresentam-se no mercado como tendo as taxas de amostragem mais altas e com maior número de canais. Um exemplo deste produto é apresentado a seguir.

2.1.3.1 GEN5i

O GEN5i, representado na Figura 2.5, traz consigo as ferramentas necessárias para o transformar num autêntico laboratório de medição portátil para registo e avaliação de sinais extremamente rápidos. Amplificadores com uma precisão de 0,1%, em combinação com taxas de medição de até 100MS/s, este produto pode capturar, em detalhes, rápidas sequências de sinais até 40 canais.

Figura 2.5: Sistema de aquisição de dados Genesis 5i.

Com computador integrado, baseado em processador *Dual Core 2*, Microsoft Vista 64bit Ultimate Edition e o *software* integrado Perception. É possível medir diretamente no lugar do evento e analisar os resultados. Os dados estão disponíveis através de várias interfaces padrão como WLAN, Gigabit Ethernet ou USB, para análises mesmo em lugares remotos. Estes sistemas são de fácil operação e podem ser empregados rapidamente.

O *software* integrado Perception é um pacote de *software* estruturado em módulos para aquisição de dados de medição de alta velocidade. O Perception permite a seleção de uma ou mais tarefas para:

- Parametrização de sensores e controlo do *hardware* de medição;
- Exibição em tempo real de valores medidos;
- Armazenamento e reprodução;
- Análise e geração de relatórios;
- Exportação de dados;

Perception é um *software user friendly* que não precisa de programação para iniciar as medições com o *hardware* ligado. Com a tecnologia patenteada “StatStream”, que permite que 10GB de dados podem ser exibidos em menos de 10 segundos. Isso significa que, com medições ainda em execução, é possível fazer *zoom in/out* e observar dados armazenados previamente sem que a gravação seja interrompida.

A possibilidade de criação de planos de *backup*, para os dados críticos, é outra caraterística deste produto. Existem várias opções disponíveis como armazenamento de dados em disco rígido removível de 2.5", DVD-RW ou armazenamento externo via Gigabit Ethernet, USB ou WLAN.

O GEN5i, e tal como outros produtos da HBM, apresenta robustez, segurança, confiabilidade e estabilidade. Este produto foi desenvolvido para funcionar 24 horas por dia 365 dias por ano sem erros, pois cada *crash* no laboratório de medição pode custar muito dinheiro às empresas.

2.2 Fundamentos dos Sistemas de Aquisição

Os sistemas de aquisição de sinais (SAS), que integram o poder e flexibilidade oferecida por um sistema embebido, são normalmente constituídos por uma variedade de componentes de *hardware* de diversos fabricantes. A integração destas componentes num sistema funcional é da responsabilidade das equipas do projeto.

Em muitas aplicações, e principalmente nos sistemas de aquisição, o poder e flexibilidade de um sistema embebido, permite que os SAS possam ser configuradas de inúmeras formas e cada uma com vantagens únicas. A chave para a utilização eficaz de um sistema embebido, é a combinação precisa e cuidadosa dos requisitos específicos de uma determinada aplicação, com o *hardware* e *software* apropriado. Os elementos básicos que constituem um sistema de aquisição de sinal podem ser observados no diagrama funcional da Figura 2.6 e são os seguintes:

- Sensores e Transdutores;
- Ligações Elétricas;
- Acondicionamento de Sinal;
- *Hardware* de Aquisição de Sinal;
- Aplicação de Monitorização e Controlo;

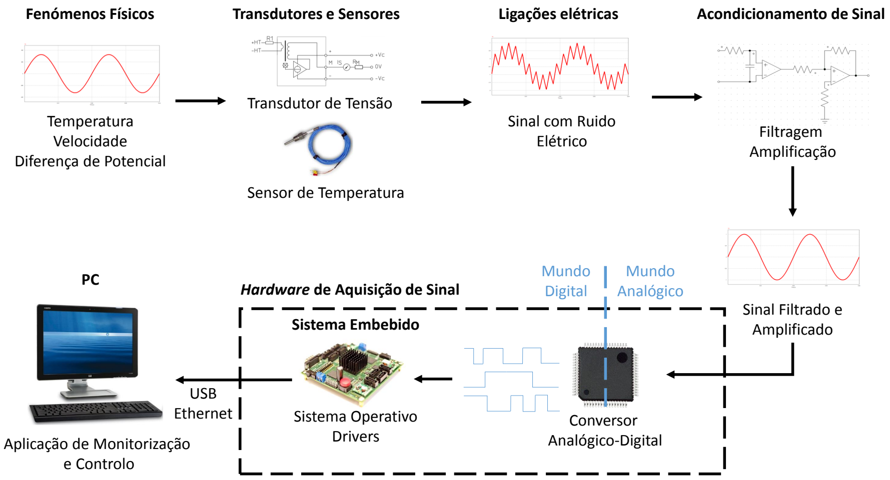


Figura 2.6: Diagrama funcional de um SAS baseado num sistema computacional.

Cada um destes elementos do sistema, tem um papel importante na medição precisa e fiável das amostras recolhidas dos fenómenos físicos monitorizados. Nas secções que se seguem são descritos cada um destes elementos.

2.2.1 Sensores e Transdutores

Sensores e transdutores são os elementos que realmente fornecem a interface entre o mundo real e o SAS pela conversão dos fenómenos físicos em sinais elétricos. Hoje, praticamente não existem fenómenos físicos pela qual não existem transdutores que possam medir e fornecer a correspondência num sinal elétrico. Por exemplo, transdutores de temperatura convertem temperatura num sinal analógico, enquanto que sensores de fluxo produzem pulsos de sinais digitais cuja frequência depende da velocidade do fluxo. Existem transdutores e sensores com capacidade de medir pressão, força, deslocamento linear e angular, velocidade e aceleração, luminosidade, diferença de potencial, correntes, resistividade ou pulsos. Em cada caso, os sinais elétricos são produzidos de acordo com uma relação definida pelo *datasheet* do transdutor ou sensor.

2.2.2 Ligações Elétricas

As ligações elétricas representam as ligações físicas desde os transdutores e sensores até ao acondicionamento de sinal. Muitas vezes estas ligações representam a maior componente do sistema e são mais suscetíveis aos efeitos do ruído externo, especialmente em ambientes industriais com equipamentos elétricos. As ligações corretas da terra e escolha da melhor blindagem dos cabos ou fios são de extrema importância para a redução dos efeitos de ruído. Os componentes passivos são muitas vezes esquecidos com componentes essenciais na redução de ruído, tornando um sistema seguro e preciso, num sistema impreciso e inseguro devido à utilização incorreta das técnicas de redução de ruído.

2.2.3 Acondicionamento de Sinal

Os sinais elétricos gerados pelos transdutores, precisam, na maioria das vezes, de ser convertidos numa gama de valores de forma a serem aceites pelo *hardware* do sistema de aquisição, como por exemplo os conversores analógicos-digital (ADC) que convertem sinais analógicos em formato digital.

As principais formas de acondicionamento de sinal são:

- Filtragem;
- Amplificação;
- Linearização;
- Isolamento;
- Excitação;

2.2.3.1 Filtragem

Em ambientes com sinal ruidoso, é muito difícil os sinais elétricos provenientes dos transdutores (na ordem dos millivolt), chegarem, aos ADCs sem que estes tenham sido comprometidos. Ruídos nas grandezas, ou maior que os sinais elétricos a medir, têm de ser filtrados. O *hardware* de acondicionamento de sinal contém normalmente filtros passa-baixo para eliminar o ruído de altas frequências que podem levar a leituras erradas nos ADCs.

2.2.3.2 Amplificação

Depois de filtrar o ruído, é necessário amplificar o sinal para aproveitar a resolução do conversor analógico-digital. A amplificação máxima do sinal é definida pelo conversor analógico-digital que se encontra no *hardware* de aquisição de sinal.

O sinal tem ser amplificado de tal forma que este esteja no limite da ordem de grandeza suportada pelo ADC.

Colocar o amplificador o mais próximo possível do transdutor reduz os efeitos dos ruídos no sinal entre o transdutor e o *hardware* de aquisição de sinal.

2.2.3.3 Linearização

Muitos transdutores, como por exemplo os termopares, apresentam uma relação não linear entre o fenómeno físico e o sinal elétrico de saída. Muitas vezes é necessário obter uma caracterização linear para posteriormente processar no *hardware* de aquisição de sinal. Os métodos de linearização podem variar dependendo do tipo de *hardware* usado no acondicionamento de sinal. Por exemplo, no caso dos termopares, alguns produtos no mercado fornecem já o *hardware* necessário para amplificar e linearizar o sinal. O método mais usado, barato e flexível, na linearização do acondicionamento de sinal, é recorrendo a técnicas de *software*.

2.2.3.4 Isolamento

O acondicionamento de sinal pode também proporcionar isolamento entre os sinais do transdutor e o *hardware* de aquisição de sinal. Esta característica permite prevenir estragos se existirem transições de alta tensão que podem ocorrer dentro do sistema que está a ser monitorizado, seja devido a uma descarga eletrostática ou falha elétrica. O isolamento, além de proteger os equipamentos, protege, acima de tudo, o ser Humano de lesões graves. Existem aplicações onde é comum o uso de altas tensões, como por exemplo nos equipamentos médicos, com recurso ao isolamento é possível obter medições seguras e precisas.

2.2.3.5 Excitação

Alguns transdutores, como por exemplo PTCs (*Positive Temperature Coefficient*) ou RTDs (*Resistance Temperature Detector*), precisam de uma fonte externa de corrente ou tensão excitação para funcionarem. É a responsabilidade do *hardware* de acondicionamento de sinal fornecer essa excitação.

2.2.4 *Hardware* de Aquisição de Sinal

O *hardware* de aquisição de sinal é o *core* do sistema global e pode executar qualquer um dos seguintes procedimentos:

- Os sinais analógicos à entrada do sistema são processados e convertidos em formato digital utilizando ADCs;
- Os dados são transferidos para um sistema embebido para serem analisados e processados;
- Os dados são enviados para o computador pessoal (PC) onde está a aplicação gráfica de monitorização e controlo;
- Após processamento os dados são convertidos em formato analógico usando DACs (conversores digital-analógico);
- Os sinais analógicos gerados são usados para controlar o sistema;
- São usados sinais digitais para efeitos de controlo;

Existe uma grande diversificação de *hardware* de aquisição de sinal que está disponível para o consumidor sob várias formas e diversos fabricantes. A ligação entre o sistema embebido e o PC pode ser feito, por exemplo, por placas de expansão que são bastante usados nas indústrias. Um meio de comunicação muito utilizado é a interface RS-232 que permite através de um PC controlar e monitorizar o sistema embebido.

Nenhum *hardware* de aquisição de sinal funciona sem *software*. É o *software* que corre no sistema embebido que transforma o sistema global num sistema de aquisição e análise de sinais. O *software* que corre no sistema embebido pode ser um *software* desenvolvido especificamente para uma dada aplicação ou pode ser desenvolvido para um sistema multi-tarefa, onde cada tarefa trata de uma parte específica do *hardware*, este último tem como base um sistema operativo genérico (GPOS), como por exemplo o Linux, ou um RTOS como é exemplo o FreeRTOS.

2.2.5 Aplicação de Monitorização e Controlo

A aplicação de monitorização, que executa num PC, corre sobre um sistema operativo *single-tasking*, como por exemplo o DOS, ou *multitasking* que permite mais que uma aplicação possa ser executada ao mesmo tempo, como por exemplo Windows e Linux. A aplicação pode ter diversas funcionalidades como por exemplo armazenar os dados recebidos do *hardware* de aquisição de sinal em disco e ter um ecrã interativo para controlar o *hardware* de aquisição de sinal. Na Figura 2.7 está representado uma aplicação de monitorização gráfica que recebe os dados do *hardware* de aquisição de sinal.

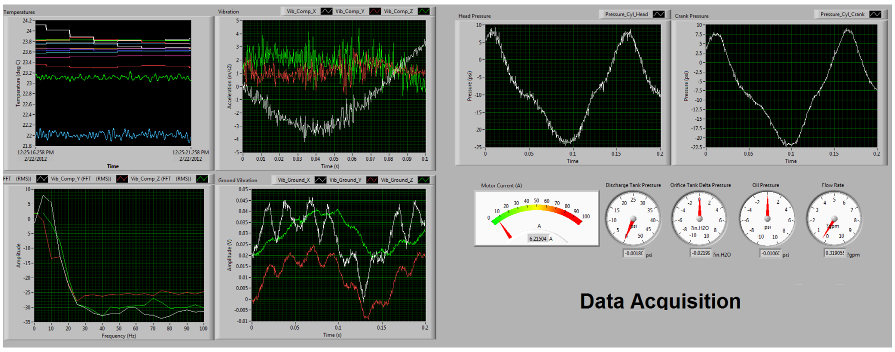


Figura 2.7: Exemplo de uma aplicação de monitorização num PC.

Este tipo de abordagem que utiliza num PC permite maior flexibilidade. Por exemplo, é possível mudar *hardware* de aquisição de sinal sem ser preciso alterar a aplicação de monitorização, para isso basta manter o protocolo de comunicação com o PC e a aplicação de monitorização continua a funcionar da mesma forma.

Figura 2.8: Exemplo de uma aplicação de monitorização num sistema embebido.

Existem sistemas de aquisição que executam a aplicação de monitorização diretamente no *hardware* de aquisição de sinal. Esta característica implica maior responsabilidade do desenvolvimento da aplicação de monitorização, uma vez que esta não deve interferir no correto funcionamento do sistema. Na Figura 2.8 está um exemplo simples deste tipo de sistemas. Esta abordagem é uma solução com menor custo no sistema final uma vez que não precisa do PC, por outro lado, exige mais esforço de engenharia devido à grande variabilidade do *hardware* e ferramentas de *software*.

2.3 Classificação dos Sinais

Os sinais contêm informações sobre uma variedade de fenómenos físicos do mundo real. Um exemplo são informações sobre o tempo que está contido em sinais que representam a temperatura do ar, pressão, velocidade do vento e outros. Para monitorizar um reator nuclear são utilizados equipamentos para medir uma variedade de parâmetros relevantes e cada equipamento produz um sinal.

A extração de informação de um conjunto de sinais, para processamento, é normalmente realizado por sistemas eletrónicos. Para tal ser possível o sinal físico tem de ser convertido num sinal elétrico, isto é, numa tensão ou corrente. Este processo é feito normalmente recorrendo a transdutores que são descritos com mais detalhe na Secção 2.2.1.

Pela descrição anterior, um sinal é uma grandeza que varia continuamente no tempo e que pode ser representado por um gráfico como o da Figura 2.9. A informação contida no sinal é representada pela variação da sua magnitude ao longo de tempo.

Este tipo de forma de onda, como o da Figura 2.9, é difícil de caracterizar por uma expressão matemática. Essa caracterização é importante para circuitos que realizam processamento de sinal.

2.3.1 Sinais Analógicos e Digitais

O sinal representado na Figura 2.9 é um sinal analógico. Um sinal analógico é aquele que pode tomar um conjunto de valores infinitos, ou seja, varia de forma contínua.

Uma forma alternativa de representar estes sinais seria por uma sequência de números, onde cada número representa a magnitude do sinal num dado instante de

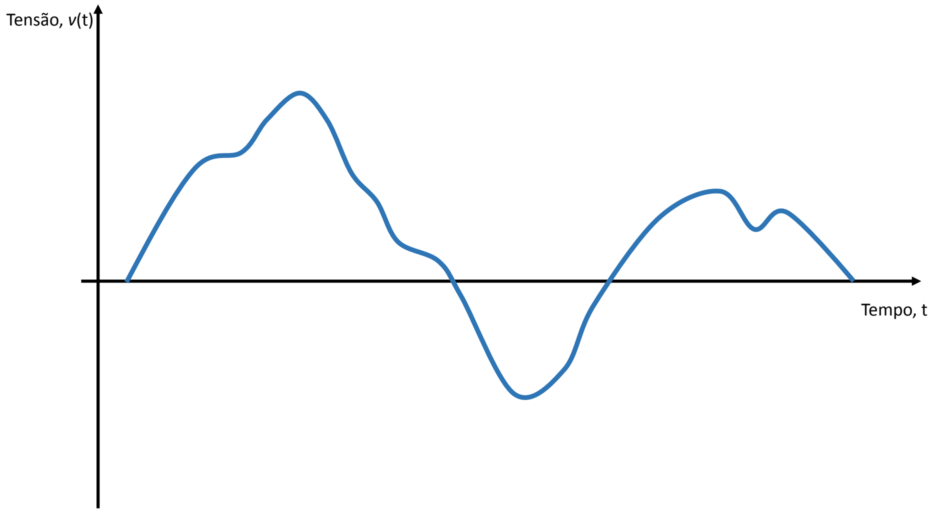


Figura 2.9: Sinal contínuo v(t).

tempo. O sinal resultante deste processo é chamado sinal digital. A conversão do sinal analógico num sinal digital está representado nas Figuras 2.10 e 2.11 respetivamente. A Figura 2.10 representa o sinal da Figura 2.9 marcada nos instantes *t0*, *t1*, *t2*, etc. Em cada um destes instantes a magnitude do sinal é medido, este processo é conhecido como amostragem. A Figura 2.11 representa o sinal da Figura 2.10 em termos das suas amostras. O sinal da Figura 2.11 só existe nos instantes amostrados, deixa de ser sinal contínuo no tempo e passa a ser um sinal discreto no tempo.

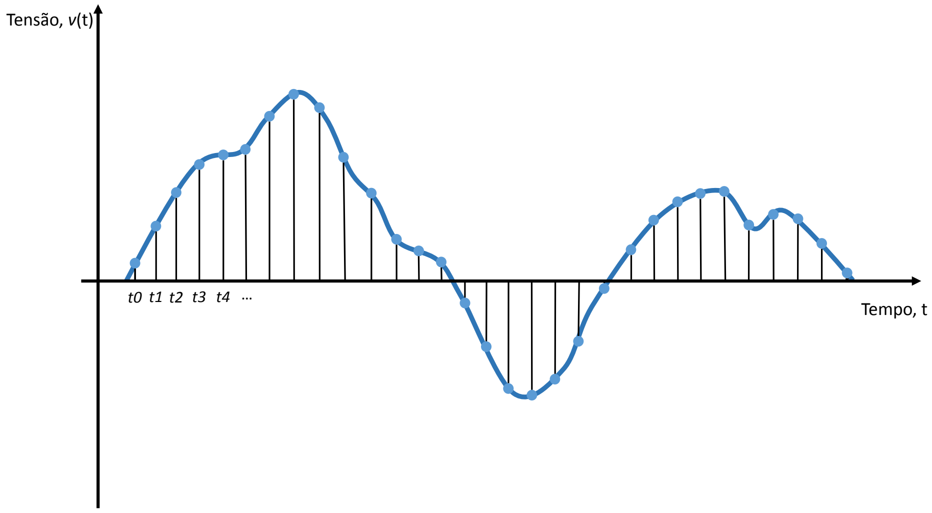


Figura 2.10: Amostragem constante do sinal analógico.

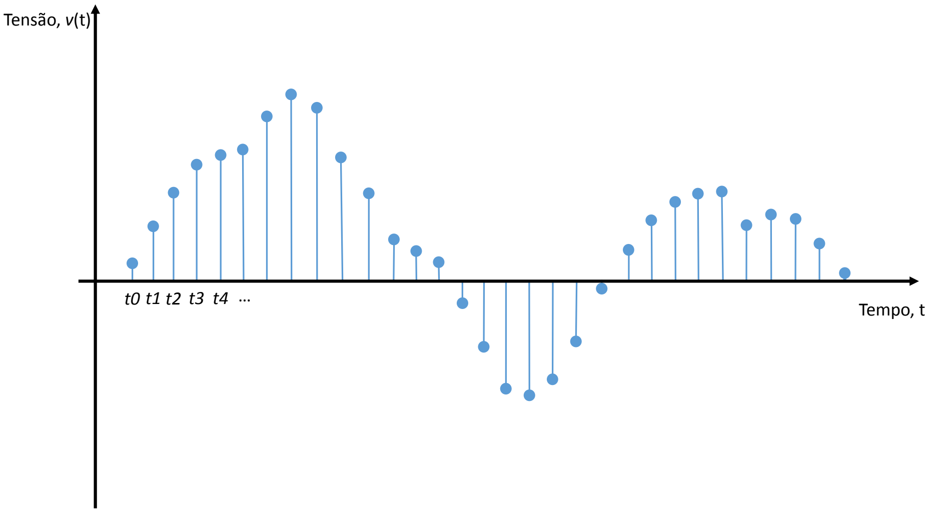


Figura 2.11: Resultado do sinal analógico no domínio discreto.

Se cada amostra do sinal for representado por um número com um número finito de dígitos, este passa a estar quantificado e passa a ser um sinal digital. O sinal digital resultante é então simplesmente uma sequência de números que representam as grandezas das amostras sucessivas do sinal. Essa sequência é normalmente representada no sistema de representação binária por ser a forma mais simples de representação e a mais utilizada nos sistemas informáticos (Sedra and Smith, 2009).

2.4 Sistema Embebido

Um sistema embebido é um sistema computacional desenvolvido para exercer um conjunto de funções restritas e são normalmente subsistemas de um sistema maior. Estes, os sistemas embebidos, são uma vertente dos sistemas informáticos, cuja parametrização se define com um propósito específico, possuindo apenas recursos de *hardware* e *software* estritamente necessários.

Os sistemas embebidos podem ser encontrados em vários lugares, sob várias formas e são desenvolvidos para interagir com o mundo físico de forma mais eficiente e segura possível. Alguns exemplos de sistemas embebidos podem se encontrados na Figura 2.12.

Os relógios digitais, leitores de MP3, impressoras e os sistemas de alarme são exemplos mais comuns onde se pode encontrar sistemas embebidos. Os carros

Figura 2.12: Exemplos onde se pode encontrar sistemas embebidos.

modernos têm um sistema embebido central que controla vários subsistemas ao mesmo tempo, como por exemplo ignição, sistemas ABS, controlo de tração (TCS) e ainda faz leitura de vários sensores do carro.

Comparando os computadores tradicionais, os PCs executam vários processos e são utilizados para satisfazer uma ampla gama de necessidades do utilizador final como por exemplo navegar na Internet, verificar emails, ouvir música, processamento de texto e muito mais. Os sistemas embebidos, ao contrário dos PCs, podem ir desde os mais simples, com apenas uma tarefa que realiza sempre a mesma função para que foram programadas, aos mais complexos, que utilizam a flexibilidade de um sistema operativo para aumentar a produtividade no desenvolvimento de *software*.

A utilização de um sistema operativo, nos sistemas embebidos, faz com que estes sejam mais caros porque requerem maiores recursos de *hardware*. Por outro lado, dependendo das funcionalidades pretendidas a utilização de um sistema operativo permite, com menos complexidade, adicionar componentes de *software* como por exemplo protocolos de rede como TCP/IP, HTTP e adicionar capacidade de armazenamento em massa como por exemplo FAT ou NTFS.

2.4.1 Fundamentos dos Sistemas Embebidos

O desenvolvimento moderno de sistemas embebidos requer uma visão unificada de *hardware* e *software*, vendo-os não como domínios completamente diferentes, mas sim como duas opções de implementação que se complementam uma à outra. Os sistemas embebidos são os sistemas computacionais que apresentam as métricas de projeto mais exigentes e menos tolerantes. As métricas de projeto definem os sistemas embebidos e podem ser por exemplo custo, tamanho, desempenho, flexibilidade, potência e segurança. Muitas vezes os sistemas embebidos têm de ser de baixo custo, ser dimensionado para caber em um único circuito integrado, devem executar rápido o suficiente para processar dados em tempo real, e devem consumir o mínimo de energia para prolongar a vida da bateria e evitar necessidade de ventilação.

Para desenvolver uma solução otimizada capaz de satisfazer os requisitas da aplicação, é necessário dominar a variabilidade de tecnologias de implementação de *hardware* e de *software*.

2.4.1.1 *Hardware*

O *hardware* refere-se às partes físicas (circuitos integrados e componentes mecânicos) que constituem o sistema embebido. Na Figura 2.13 está representado um diagrama com os elementos básicos que constituem alguns sistemas embebidos.

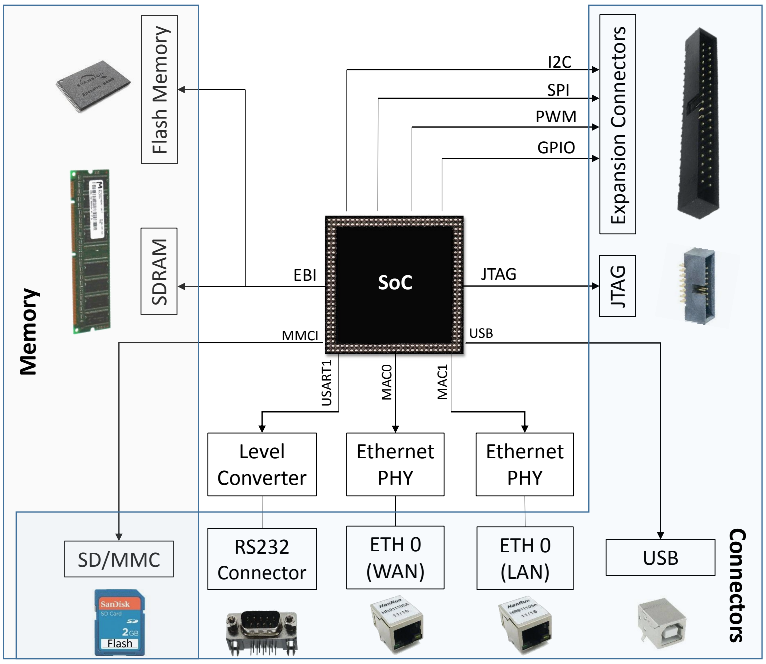


Figura 2.13: Diagrama de blocos do *hardware* de um sistema embebido genérico.

Alguns sistemas embebidos são desenvolvidos em torno de um *System-on-Chip* (SoC), com uma ou mais interfaces de comunicação para o exterior como por exemplo a porta série RS232, interface USB e Ethernet que são normalmente as mais utilizadas. Também fazem parte destes sistemas, memórias não voláteis como por exemplo memória *flash* ou ROM para armazenamento do *software*, e memórias voláteis como a SDRAM (*Synchronous Dynamic Random Access Memory*). O uso de cartões SD nos sistemas embebidos tem crescido nos últimos anos e são usados como uma memória complementar a memória *flash* do sistema. A expansão de conectores são utilizados para ligar outros periféricos externos ao sistema ou até mesmo para comunicar com outros sistemas embebidos.

SoC

Um SoC é um circuito integrado que integra todas as componentes essenciais de um sistema computacional em um único *chip*. Na Figura 2.14 está representado um diagrama de blocos simplista de um SoC.

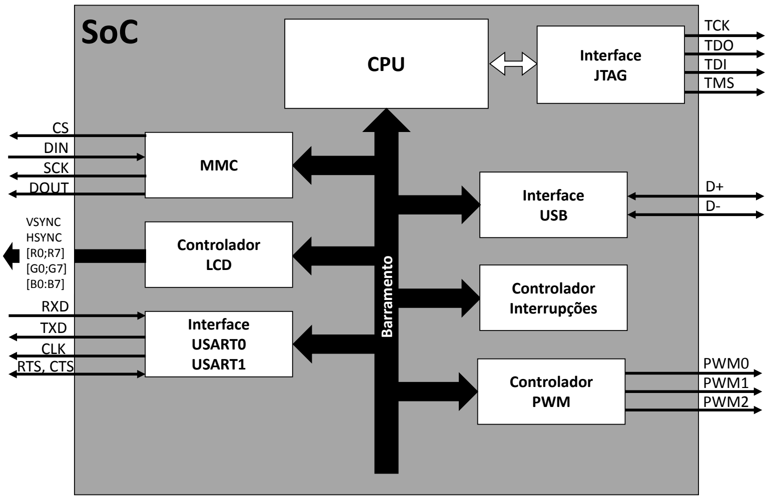


Figura 2.14: Diagrama de blocos simplista de um SoC.

Enquanto um microcontrolador inclui o CPU, RAM, ROM e E/S, um SoC tem componentes adicionais. Por exemplo, um SoC para *smartphone* pode incluir controladores para áudio e de vídeo, para além dos componentes do microcontrolador. Os SoCs com grande poder computacional têm capacidade para correr sistemas operativos como Linux, estes são constituídos:

- Numa ou mais unidades de processamento;
- Controladores de memória externa e interna (RAM, ROM, *Flash*);
- *Timers*, Contadores e *timers* de tempo real;
- Controladores USB, Ethernet, USART, SPI, I2C, etc.;

Este componentes comunicam e trocam dados entre eles através de um barramento. As próximas secções descrevem algumas das partes essenciais que constituem um SoC.

CPU

A unidade de processamento central pode ser classificada de acordo com a complexidade do conjunto de instruções, ou seja, podem ser classificados como RISC (*Reduced Instruction Set Computer*) ou CISC (*Complex Instruction Set Computer*). Um processador CISC tem a capacidade de executar um conjunto de instruções mais complexas do que um processador RISC, tornando-o extremamente versátil e o desenvolvimento de compiladores para CISC é mais simples. O processador RISC tem a capacidade de executar um conjunto reduzido de instruções, mas a uma velocidade muito superior à dos processadores CISC. Exemplos de processadores RISC são PowerPC, ARM e Atmel AVR. Do ponto de vista do programador, a vantagem de um processador CISC é a existência de muitas instruções requeridas na elaboração de um programa. Por outro lado num processador RISC essa tarefa é mais exigente e às vezes é necessário combinar várias instruções para executar uma tarefa mais complexa.

A operação fundamental nos CPUs é a execução de uma sequência de instruções armazenadas, chamado programa. Há quatro etapas que quase todos os CPUs utilizam na sua operação : *fetch*, *decode*, *execute* e *writeback*.

O primeiro passo, o *fetch*, consiste em adquirir uma instrução (qual é representada por uma sequência de números) a partir de uma memória do programa. A localização na memória do programa é determinada pelo *program counter*, que armazena um número que identifica a posição atual do programa. Depois do *fetch* , o *program counter* é incrementado para a próxima instrução da memória. Muitas vezes, a instrução é adquirida a partir da memória lenta, fazendo com que o processador espere pela próxima instrução. Este problema é muito abordado nos processadores da atualidade e resolvido através de *caches* e arquiteturas em *pipeline*.

A instrução adquirida pelo processador a partir da memória é utilizada para determinar o que o processador vai fazer. No passo de *decode*, a instrução é dividida de acordo com as partes importantes do CPU. Uma das partes da instrução indica a operação a ser executada, por exemplo operação de adição, e as restantes partes fornecem as informações necessárias para execução da instrução, como por exemplo os valores das constantes a serem somadas.

Após as etapas o *fetch* e o *decode*, a etapa *execute* é realizada. Durante este passo, as várias porções do CPU são ligadas de forma a ser possível executar a operação desejada. Se, por exemplo, foi pedida uma operação de adição, a unidade de lógica e aritmética (ALU) é ligada a um conjunto de entradas e saídas. As entradas fornecem os valores das constantes a serem somadas, e as saídas contêm o resultado da soma.

A etapa final, *writeback*, simplesmente devolve os resultados da etapa *execute* para uma memória. Muitas vezes, os resultados são armazenados num registo interno do CPU para acesso rápido se posteriores instruções precisarem.

Em processadores mais complexos, várias instruções podem ser obtidas, descodificadas e executadas simultaneamente. A próxima secção descreve o que é geralmente referido como o *pipeline* RISC clássico, que na verdade é bastante comum entre os CPUs simples.

Pipeline

Pipeline em computadores é utilizados para melhorar o desempenho do CPU. O objetivo é melhorar o *throughput* do sistema computacional, ou seja, o número de instruções por segundo (IPS). Esta técnica é atualmente utilizada em todos os sistemas computacionais. A Figura 2.15 mostra um diagrama de temporização do funcionamento do *pipeline*. A única etapa não mencionada anteriormente é a de *Memory Access*. Durante esta fase, as instruções que demoram apenas um ciclo passam simplesmente para a próxima fase, mas garante que as instruções que demoram dois ciclos possam ser escritas na memória no mesmo estágio de *pipeline* que as instruções que demoram apenas um ciclo. Deste modo apenas é necessário uma porta para escrita na memória.

Um *pipeline* é dividido em estágios e cada estágio:

- Executa o mesmo número de ciclos de *clock*;
- Tem seu próprio *hardware* dedicado;
- Pode executar em paralelo com as outras etapas;

Não é obrigatório que cada estágio execute em um ciclo de *clock*, mas para o *pipeline* funcionar bem as instruções em cada estágio devem demorar o mesmo número de ciclos.

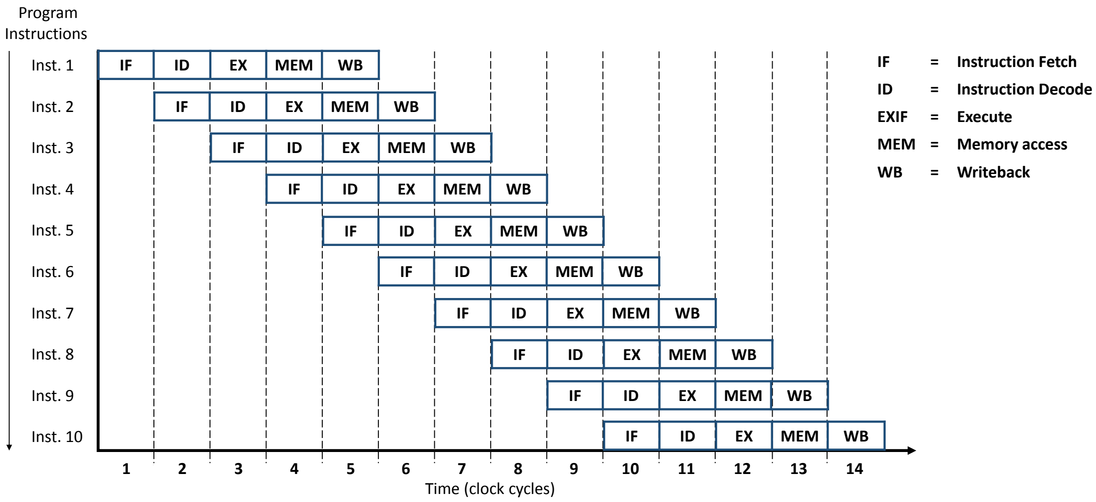


Figura 2.15: *Pipeline* básico de cinco estágios numa máquina RISC.

Interrupções

As Interrupções fazem o processador suspender a execução do programa principal e saltar para uma rotina de serviço de interrupção (ISR) de curto prazo que preenche uma necessidade especial. Em detalhe, o processador armazena o *program counter* atual, e ajusta-o para o endereço da ISR. Após a conclusão da ISR, o processador restaura o *program counter* e continua a execução do programa.

Um exemplo comum pode ser o pressionar de um botão acionado pelo utilizador. Em vez de fazer a verificação por *polling* em todo o programa, apenas é preciso escrever uma rotina de serviço de interrupção, registá-la e associar o pino que liga o botão. O processador chamará a rotina automaticamente quando o botão for pressionado. Mais detalhes estão descritos na Secção 2.6.2.4.

Barramentos

O CPU é a unidade que processa toda a informação do sistema. A memória, periféricos de entrada e saída, são ligadas ao CPU através de um barramento de endereço, um barramento de dados e de um barramento de controlo. As portas de entrada e saída são interfaces físicas pela qual os dados são enviados para periféricos externos. Na Figura 2.16 está representado um diagrama com as ligações dos barramentos ao CPU, memória e os periféricos de entrada e saída.

- **O Barramento de Endereços**: para cada operação de leitura ou de escrita, o CPU especifica o endereço da localização que pretende aceder, colocando

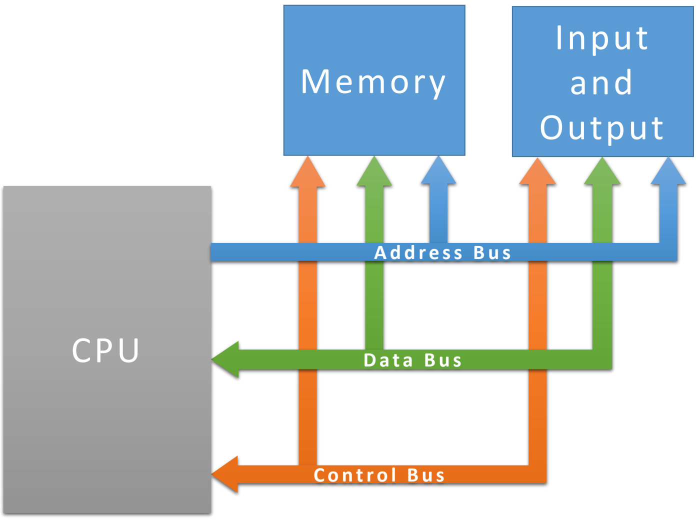


Figura 2.16: Ligação dos barramentos ao CPU, memória e periféricos

um endereço no barramento de endereços. A memória ou periféricos de entrada/saída (cada periférico tem associado um endereço único) obtém o endereço presente no barramento de endereço, usando-o para aceder à localização pretendida;

- **O Barramento de Dados**: transporta os dados que serão escritos ou lidos pelo CPU, memória ou pelos dispositivos do SoC;
- **O Barramento de Controlo**: carrega informações de controlo entre a CPU e os outros dispositivos do sistema. O barramento de controlo também transporta sinais com a informação do estado dos dispositivos Estes sinais funcionam como sinais de temporização gerados pelo CPU para sincronização da transferência de informação presente nos barramentos de endereço e de dados. Assim, durante as operações de escrita ou de leitura, o CPU ativa um sinal no barramento de controlo indicando se a operação é de escrita ou de leitura (Tavares et al., 2009);

A largura de um barramento é importante porque determina a quantidade de informação que pode ser transmitida. Por exemplo, um barramento de 16 bits pode transmitir 16 bits de dados, enquanto que um barramento de 32 bits pode transmitir 32 bits de dados. Cada barramento tem uma velocidade de *clock* medido em MHz. Um barramento rápido permite que os dados sejam transferidos rapidamente, o que torna as aplicações mais rápidas. As principais ações realizadas pelo CPU envolvem a leitura ou escrita para a memória principal. As etapas que envolvem o ciclo de leitura são:

- O CPU coloca o endereço no barramento de endereços;
- São colocados os sinais de controlo para memória no barramento de controlo;
- A memória vai buscar os dados na localização dada pelo barramento de endereços e coloca os dados no barramento de dados;
- O CPU lê os dados do barramento de dados;

As etapas que envolvem o ciclo de escrita são:

- O CPU coloca o endereço no barramento de endereços;
- O CPU coloca os dados no barramento de dados;
- São colocados os sinais de controlo para memória no barramento de controlo;
- A memória vai buscar os dados ao barramento de dados e coloca-os no endereço dado pelo barramento de endereços;

Memórias

Para que o CPU possa executar as suas instruções, este vai à memória buscar a informação necessária ao processamento. Nos sistemas computacionais as memórias são responsáveis pelo armazenamento de dados e instruções em forma de sinais digitais. Existem, essencialmente, duas categorias de memórias: ROM (*Read-Only Memory*), que permitem apenas a leitura dos dados e não perde informação na ausência de energia; e RAM (*Random-Access Memory*), que permite ao processador a leitura e a escrita de dados, e perdem informação quando não há alimentação elétrica (Alecrim, 2011).

Memória ROM

As memórias ROM (*Read-Only Memory*) recebem esse nome porque os dados são gravados apenas uma vez nelas. Depois disso, as informações não podem ser apagadas ou alteradas, apenas lidas pelo CPU. Outra característica das memórias ROM é que elas são do tipo não voláteis, isto é, os dados gravados não são perdidos na ausência de energia elétrica. Os principais tipos de memória ROM:

- **PROM** (*Programmable Read-Only Memory*): é um dos primeiros tipos de memória ROM. A gravação de dados neste tipo de memórias é realizada por meio de aparelhos que trabalham através de uma reação física com elementos elétricos. Uma vez que isso ocorre, os dados gravados na memória PROM não podem ser apagados ou alterados;

- **EPROM** (*Electrically Programmable Read-Only Memory*) : as memórias EPROM têm como principal característica a capacidade de permitir que dados sejam regravados no dispositivo. Isso é feito com o auxílio de um componente que emite luz ultravioleta. Nesse processo, os dados gravados precisam ser apagados por completo. Somente depois disso é que uma nova gravação pode ser feita;
- **EEPROM** (*Electrically Erasable Programmable Read-Only Memory*): é similar à EPROM, mas ao contrário destas, os processos para apagar e gravar dados são feitos eletricamente. Assim, esta característica permite que a informação possa ser alterada, mesmo se estiver em funcionamento num circuito eletrónico. Este tipo de memórias são populares nos microcontroladores para guardar variáveis importantes do programa;
- ***Flash***: as memórias *flash* também podem ser vistas como um tipo de EEPROM, no entanto, o processo de gravação (e regravação) é muito mais rápido. Além disso, memórias *flash* são mais duráveis e podem guardar um volume elevado de dados. As memórias *flash* consomem pouca energia, ocupam pouco espaço físico e são bastante resistentes. O grande problema das memórias *flash* é o seu preço elevado. Felizmente, ao longo dos anos esta tecnologia tem vindo a diminuir o seu custo. O *software* desenvolvido para os sistemas embebidos, a *firmware*, é armazenada, na maioria das vezes, numa memória do tipo *flash*. Existem dois tipos de memória *flash*, a NOR e NAND:

  NOR: a memória *flash* NOR (*Not OR*) permite acesso às células de memória de maneira aleatória com alta velocidade. Em outras palavras, o tipo NOR possibilita o acesso de dados em posições diferentes da memória de maneira rápida, sem necessidade de esta operação ser sequencial. O tipo NOR geralmente é aplicado em chips de BIOS, telemóveis e em placas de rede.

  NAND: por sua vez, a memória *flash* NAND (*Not AND*) também trabalha em alta velocidade, porém faz acesso sequencial às células de memória e em blocos, em vez de as aceder de de forma individual. Assim, a *flash* NAND é mais adequada para armazenamento e acesso de dados em massa, ou seja não se adequa à leitura particular de um bit ou de uma *word* e são normalmente utilizadas para acesso à informação sob um sistema de ficheiros;

Memória RAM

As memórias RAM (*Random-Access Memory*) constituem uma das partes mais importantes dos sistemas embebidos, pois são nelas que o CPU guarda os dados com os quais está a trabalhar. Este tipo de memórias a escrita/leitura de dados é extremamente rápida, quando comparadas com as memórias ROM. No entanto, as informações gravadas perdem-se na ausência de energia, isto é, quando o sistema é desligado, sendo, portanto, um tipo de memória volátil.

Existem dois tipos de tecnologia de memória RAM que são muito utilizados: estático e dinâmico, isto é, SRAM e DRAM, respetivamente. Existe também um tipo mais recente chamado MRAM.

- **DRAM** (*Dynamic Random-Access Memory*): as memórias desse tipo têm grande capacidade de armazenamento, isto é, podem comportar grandes quantidades de dados. No entanto, o acesso a estes dados costuma ser mais lento que o acesso às memórias estáticas (SRAM). Em compensação tem preços bem menores que as memórias do tipo estático, pois utiliza uma tecnologia mais simples.

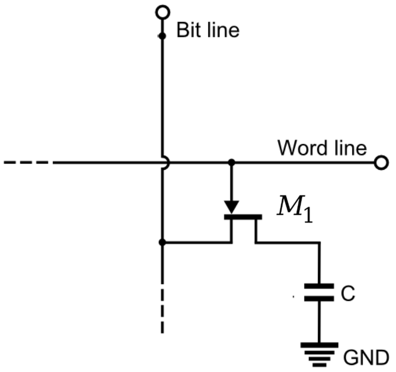


Figura 2.17: Uma célula de memória na arquitecura DRAM

As memórias DRAM são formadas por chips que contém uma quantidade elevadíssima de condensadores e MOSFETs, onde um condensador e um transístor, juntos, formam uma célula de memória (ver Figura 2.17). O condensador tem a função de armazenar uma carga elétrica por um certo tempo, enquanto que o transístor controla essa carga. Se o condensador tiver carga então tem-se um bit 1. Se não tiver, tem-se um bit 0. O problema é que a informação é mantida por um curto período de tempo e, para que não haja perdas de dados da memória, um componente do controlador de memória é responsável pela função de *refresh*, que consiste em atualizar o conteúdo da célula de tempos em tempos. Este processo é realizado milhares de vezes por segundo.

O *refresh* é uma solução, porém é acompanhada de “efeitos colaterais”: este processo aumenta o consumo de energia e, como consequência, aumenta a dissipação de calor. Além disso, a velocidade de acesso à memória DRAM acaba por ser reduzida;

- **SRAM** (*Static Random-Access Memory*): são muito mais rápidas que as memórias DRAM, porém têm menor capacidade de armazenamento de dados e possui preço mais elevado. As memórias SRAM são responsáveis pelo armazenamento dos dados e das instruções que o processador precisa para executar suas tarefas. A memória SRAM é bastante diferente da DRAM. A SRAM utiliza seis MOSFETs (ver Figura 2.18(a)), ou quatro MOSFETs e duas resistências (ver Figura 2.18(b)) para formar uma célula de memória. Na verdade, dois MOSFETs são responsáveis pelo controlo, enquanto que os restantes ficam responsáveis pelo armazenamento elétrico, isto é, pela formação do bit.

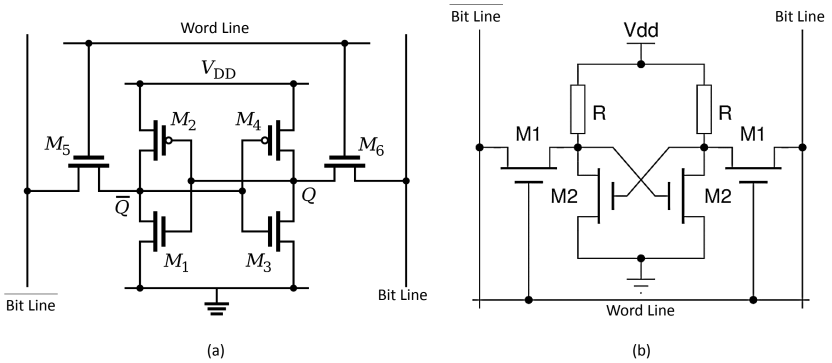


Figura 2.18: Uma célula de memória na arquitecura SRAM com seis MOSFETs em (a) e com quatro MOSFETs em (b).

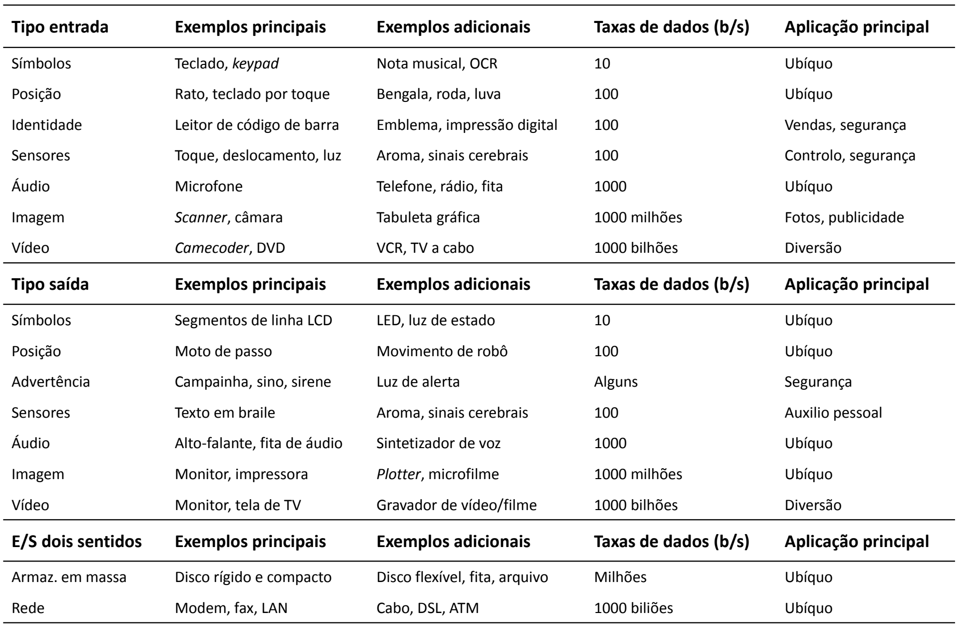
A vantagem deste esquema é que o *refresh* não é necessário, fazendo com que a memória SRAM seja mais rápida e consuma menos energia. Por outro lado, como a sua estrutura é mais complexa e requer mais componentes, o seu custo acaba por ser maior que a DRAM;

- **MRAM** (*Magnetoresistive Random-Access Memory*): as memórias MRAM têm vindo a ser estudadas ao longo dos anos, apenas nos últimos anos é que começaram a surgir as primeiras unidades. Trata-se de um tipo de memória até certo ponto semelhante à DRAM, mas que utiliza células magnéticas. Graças a isso, estas memórias consomem menos energia, são mais rápidas e armazenam dados por mais tempo, mesmo na ausência de energia elétrica. O problema das memórias MRAM é que elas armazenam pouca quantidade de dados e são muito caras, portanto, não serão adotadas em larga escala nos próximos tempos;

Entradas/Saídas

Nos sistemas embebidos, as entradas/saídas, ou *I/O*, permitem a comunicação entre o sistema e o mundo exterior, como por exemplo um outro sistema embebidos ou um utilizador. Os dispositivos de entrada são por exemplo os teclados, ratos e sensores, enquanto que dispositivos de saída são considerados monitores, impressoras e atuadores. Também existem dispositivos que são de entrada e de saída como por exemplo as placas de rede. Na Tabela 2.1 estão alguns tipos de dispositivos (Parhami, 2008).

Tabela 2.1: Dispositivos de entrada, saída e que atuam nos dois sentidos.



Tipo entrada	**Exemplos principais**	**Exemplos adicionais**	**Taxas de dados (b/s)**	**Aplicação principal**
Símbolos	Teclado, *keypad*	Nota musical, OCR	10	Ubíquo
Posição	Rato, teclado por toque	Bengala, roda, luva	100	Ubíquo
Identidade	Leitor de código de barra	Emblema, impressão digital	100	Vendas, segurança
Sensores	Toque, deslocamento, luz	Aroma, sinais cerebrais	100	Controlo, segurança
Áudio	Microfone	Telefone, rádio, fita	1000	Ubíquo
Imagem	*Scanner*, câmara	Tabuleta gráfica	1000 milhões	Fotos, publicidade
Vídeo	*Camecoder*, DVD	VCR, TV a cabo	1000 bilhões	Diversão
Tipo saída	**Exemplos principais**	**Exemplos adicionais**	**Taxas de dados (b/s)**	**Aplicação principal**
Símbolos	Segmentos de linha LCD	LED, luz de estado	10	Ubíquo
Posição	Moto de passo	Movimento de robô	100	Ubíquo
Advertência	Campainha, sino, sirene	Luz de alerta	Alguns	Segurança
Sensores	Texto em braile	Aroma, sinais cerebrais	100	Auxilio pessoal
Áudio	Alto-falante, fita de áudio	Sintetizador de voz	1000	Ubíquo
Imagem	Monitor, impressora	*Plotter*, microfilme	1000 milhões	Ubíquo
Vídeo	Monitor, tela de TV	Gravador de vídeo/filme	1000 bilhões	Diversão
E/S dois sentidos	**Exemplos principais**	**Exemplos adicionais**	**Taxas de dados (b/s)**	**Aplicação principal**
Armaz. em massa	Disco rígido e compacto	Disco flexível, fita, arquivo	Milhões	Ubíquo
Rede	Modem, fax, LAN	Cabo, DSL, ATM	1000 biliões	Ubíquo

Estes dispositivos, para comunicarem com os sistemas embebidos, utilizam interfaces de comunicação que muitas das vezes variam de dispositivo para dispositivo.

As interfaces mais populares, que se podem encontrar nos sistemas embebidos para ligação de dispositivos externos são o USB, RS-232, Ethernet e *GPIO*. Sistemas embebidos que apresentam capacidades para processamento gráfico podem ter presentes uma interface VGA ou HDMI.

2.4.1.2 *Software*

Na Figura 2.19 está representado uma arquitetura simplificada de um sistema embebido. No topo da hierarquia dos sistemas embebidos encontra-se a camada da aplicação. O *software* na camada da aplicação é quem define o tipo de dispositivo que o sistema embebido é, pois as funcionalidades da aplicação representam o objetivo pela qual o sistema foi desenvolvido.

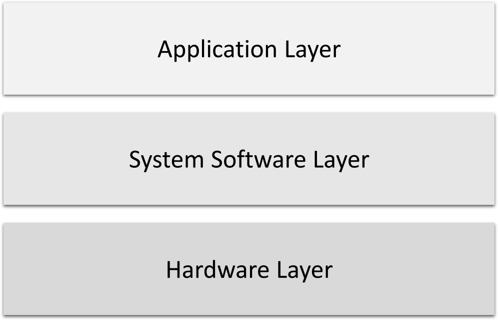


Figura 2.19: Arquitetura de um sistema embebido.

Por sua vez, a camada de *software* do sistema é a camada de suporte a camada da aplicação e só existe em sistemas embebidos mais complexos que utilizam o poder de um sistema operativo. Na Secção 2.6 será feito uma descrição detalhada sobre a utilização de sistemas operativos nos sistemas embebidos.

Arranque do Sistema Embebido

O *booloader* é o primeiro conjunto de instruções que o sistema embebido executa quando este é alimentado ou reiniciado. As rotinas de inicialização do *bootloader* são responsáveis pela inicialização do *hardware* do sistema (como por exemplo a memória e a porta série), dependem da arquitetura do CPU e do *hardware* do sistema. Além disso, em sistemas embebidos simples o *bootloader* é responsável pela gravação do ficheiro com o código máquina (*.hex*) e em sistemas embebidos mais avançados o *bootloader* também é responsável pelo carregamento da imagem do sistema operativo. Este trabalho assenta essencialmente em sistemas embebidos avançados que utilizam o poder de um sistema operativo, e neste caso o *bootloader* executa as seguintes rotinas:

- Inicialização da memória principal e de uma comunicação série;
- A imagem do sistema operativo é colocada num endereço apropriado da memória principal do sistema;
- Carregar parâmetros de inicialização;
- Descomprimir a imagem e transferir o controlo para o sistema operativo;

Neste procedimento, uma memória não volátil, geralmente uma *flash*, armazena o *bootloader*, os seus parâmetros, o *kernel* e o sistema de ficheiros. A Figura 2.20 representa a estrutura típica desta memória com todos os componentes essenciais para o arranque do sistema.

Figura 2.20: *Layout* típico da memória não volátil de um sistema embebido.

Não são indicados quaisquer endereços de memória porque os intervalos variam muito dependendo do tipo e das funcionalidades do sistema. Por exemplo, um sistema embebido com muitos periféricos vai ter um *kernel* maior porque precisa de mais *drivers* para o sistema.

É da responsabilidade do programador do sistema embebido escolher e configurar o *bootloader*. Essa escolha depende sempre da arquitetura do sistema embebido e existem vários *bootloader open source* que podem ser usados. Os mais populares nos sistemas embebidos são:

- **U-Boot**, *Universal Boot Loader* (DENX Software Engineering, 2013): é sem dúvida o *bootloader* mais usado nos sistemas embebidos, com maior flexibilidade e atividade na comunidade *open source*. O U-Boot suporta várias plataformas de desenvolvimento que se baseiam em processadores ARM, AVR32, Blackfin, x86, Motorola 68K, Xilinx Microblaze, MIPS, Alterra NIOS, NIOS2, PowerPC, Super-H, e outros;

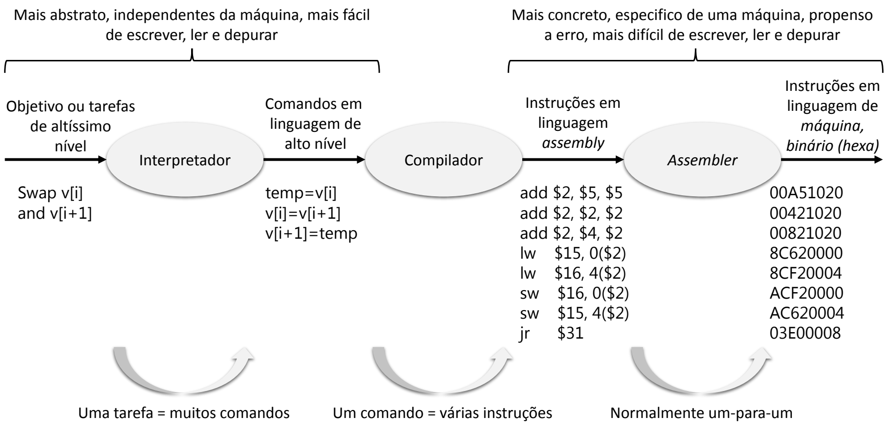
- **RedBoot** (Red Hat, 2013): é um *bootloader* baseado no eCos (*Embedded Configurable Operating System*), escrito pela Cygnus Solutions e posteriormente comprado pela Red Hat. eCos é bastante popular em sistemas embebidos que são muito pequenos para correr um sistema operativo como o Linux, mas o RedBoot foi estendido para fazer o *boot* de outros sistemas operativos incluindo o Linux;

- **Barebox** (Barebox.org, 2013): deriva do U-boot mas não oferece a mesma flexibilidade e suporte. O Barebox está bastante ativo na comunidade *open source*, mas foca-se essencialmente em fazer o *boot* do Linux e não está preocupado com outros sistemas operativos como por exemplo o Windows Embedded Compact;

A região da memória onde se encontra o *bootloader* tem normalmente mecanismos de proteção para garantir que este não é corrompido. A forma mais utilizada para programar ou atualizar o *bootloader* é através da interface JTAG (*Joint Test Action Group*) que dá instruções ao CPU para executar as ações necessárias para programar a memória não volátil.

Software **do Sistema e Aplicações**

As instruções compreendidas pelos sistemas computacionais são codificados como cadeias de 0s e 1s. Um conjunto de tais instruções constitui um programa em linguagem de máquina que especifica passo a passo o processo computacional (Figura 2.21, lado direito). Os primeiros sistemas computacionais digitais eram programados em linguagem de máquina, um incómodo processo que foi aceitável somente porque os programas naqueles dias eram muito simples. Desenvolvimentos subsequentes levaram à invenção da linguagem *assembly*, a qual permite uma representação simbólica de programas em linguagem de máquina, e linguagens procedimentais de alto nível, que se assemelham à notação matemática. Essas representações mais abstratas, em conjunto com a tradução do *software* (compiladores e *assemblers*) para conversão automática de programas para linguagem de máquina, simplificaram de modo significativo o desenvolvimento de programas e aumentaram a produtividade dos programadores. Atualmente, muita da computação para os sistemas embebidos é feita por meio de notações de alto nível, que possuem um grande poder de expressão para domínios de interesse específicos. Exemplos incluem processadores de texto, imagem e áudio, algoritmos de controlo e redes neuronais. Esses níveis de abstração em programação, e o processo de ir de um nível para o nível mais baixo, são ilustrados na Figura 2.21.



```
Swap v[i]
and v[i+1]
```

```
temp=v[i]
v[i]=v[i+1]
v[i+1]=temp
```

```
add $2, $5, $5
add $2, $2, $2
add $2, $4, $2
lw   $15, 0($2)
lw   $16, 4($2)
sw   $16, 0($2)
sw   $15, 4($2)
jr    $31
```

```
00A51020
00421020
00821020
8C620000
8CF20004
ACF20000
AC620004
03E00008
```

Figura 2.21: Modelos de abstração em programação (Parhami, 2008).

Como mostra a Figura 2.19, o *software* de um sistema embebido pode ser dividido nas classes de *software* de aplicação e *software* do sistema. O *software* de aplicação, em sistemas embebidos simples, inclui programas simples de controlo e de monitorização, Em sistemas embebidos complexos e avançados inclui programas para GUIs (*Graphical User Interfaces*), processadores de imagem e áudio, e também inclui programas controlo e de monitorização que são desenvolvidos para resolver tarefas específicas do sistema global. O *software* do sistema, em sistemas embebidos mais complexos e poderosos, é dividido em programas que traduzem ou interpretam instruções escritas em várias linguagens de programação (tais como linguagem *assembly* ou linguagem de programação C), e têm funções de gestão para programas e recursos do sistema.

Essas são as funcionalidades requeridas por uma grande maioria de utilizadores de sistemas computacionais e, portanto, incorporados num sistema operativo. Mais detalhes das funções dos sistemas operativos são encontrados na Secção 2.6.

2.5 *Field-Programmable Gate Arrays*

As *Field-Programmable Gate Arrays* (FPGAs) são chips de silício reprogramáveis. A adoção das FPGAs pela indústria é conduzida pelo fato de que estas combinam os melhores recursos dos circuitos integrados construídos para tarefas específicas (ASICs - *application-specific integrated circuits*) e dos sistemas baseados em processadores. A utilização de FPGAs nos sistemas embebidos têm crescido nos últimos anos e são cada vez mais populares em plataformas de desenvolvimento. A flexibilidade oferecida pela FPGA permite reduzir o *overhead* do *software* nos sistemas embebidos ao permitir implementar partes de *software* em *hardware* dedicado.

Na Figura 2.22 está representado a estrutura básica das FPGAs. As FPGAs são constituídos por blocos lógicos configuráveis ligados entre si por interligações programáveis. Os blocos lógicos são feitos de dois componentes básicos: *flip-flops* e *lookup tables* (LUTs). Várias famílias de FPGAs diferem na maneira com que os *flip-flops* e LUTs são agrupados. Xilinx e Altera são dois dos principais fabricantes de FPGAs.

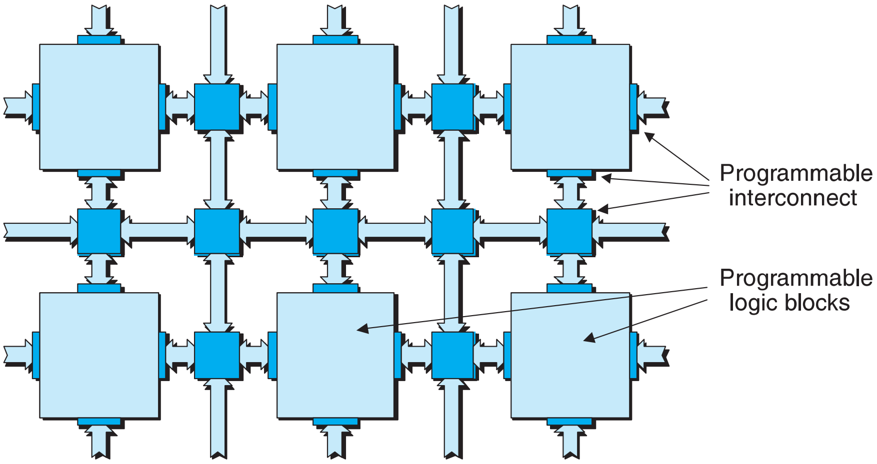


Figura 2.22: Estrutura interna de uma FPGA (Clive Max, 2009).

As LUTs e as interligações programáveis são suficientemente flexíveis para implementarem qualquer função lógica. Contudo, eles são de uma ordem de magnitude menos eficiente na velocidade e custo (área de pastilha) do que as versões *hard-wired* das mesmas funções. Assim sendo, os fabricantes de FPGAs incluem blocos especializados, tais como memórias, multiplicadores e microprocessadores.

Dependendo do fabricante, os blocos lógicos podem ter denominações diferentes. Por exemplo, a ALTERA denomina o seus blocos lógicos de LABs (*logic array blocks*) e a Xilinx chama-os CLBs (*configurable logic block*) (Clive Max, 2009). Nesta dissertação será dado ênfase ao fabricante de FPGAs Xilinx.

2.5.1 *Lookup Tables*

A maior parte da lógica num CLB é implementada usando quantidades muito pequenas de RAM na forma de LUTs. É fácil assumir que o número de portas do sistema numa FPGA se refere ao número de portas NAND ou NOR num chip em particular. Mas na realidade, toda lógica combinatória (ANDs, ORs, NANDs, XORs, etc) é implementada na forma de tabelas de verdade dentro da memória LUT.

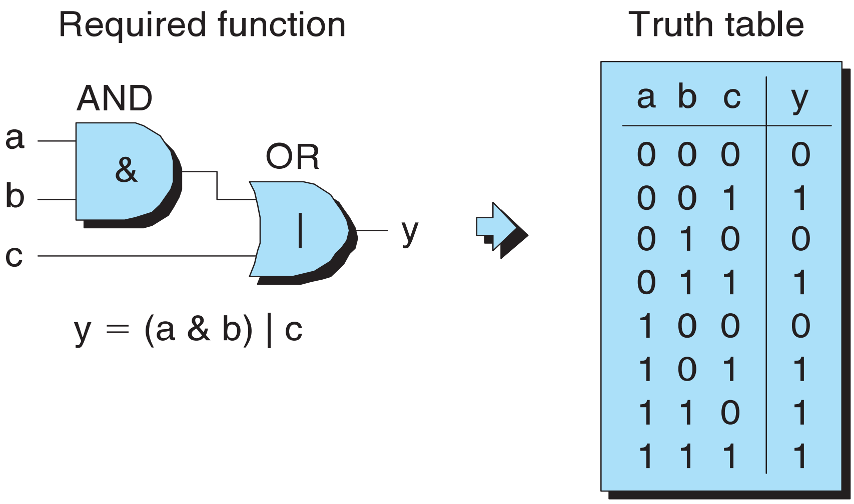


Figura 2.23: Função pretendida e tabela de verdade associada (Clive Max, 2009).

Uma tabela de verdade é uma lista predefinida de saídas para cada combinação de entradas. A Figura 2.23 mostra um exemplo de um circuito pretendido a ser implementado e a forma como é implementado na LUT da FPGA.

2.5.2 Células Lógicas

A célula lógica é o elemento básico que constitui uma grande parte da FPGA. A Figura 2.24 apresenta os principais elementos de uma célula lógica. De um modo simplista, uma célula lógica contém uma LUT de 4 entradas, um *multiplexer* e um registo.

O registo pode ser configurado para atuar como um *flip-flop* ou como uma *latch*. O sinal *clock* pode ser configurado como *rising-edge triggered* ou *falling-edge triggered*,

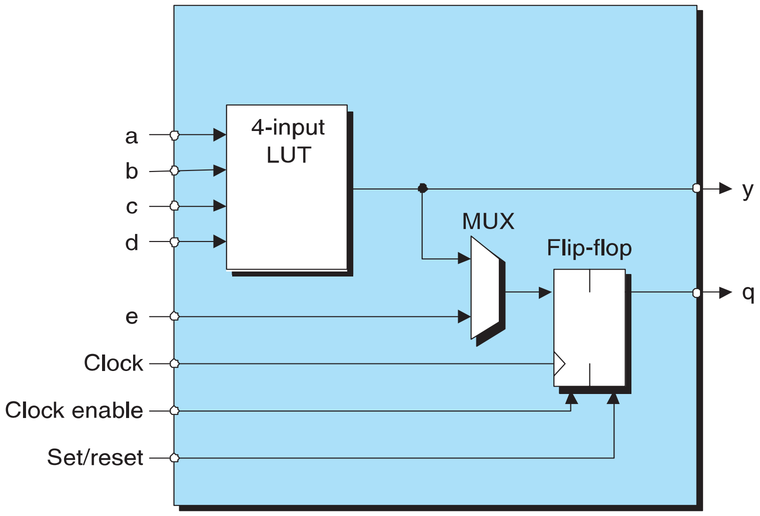


Figura 2.24: Visão simplificada da célula lógica da Xilinx (Clive Max, 2009).

assim como o *clock enable* e os sinais de *set/reset*. Além da LUT, *multiplexer*, e registo, a célula lógica também contém outros elementos como o *fast carry logic* para utilização em operações aritméticas (Clive Max, 2009).

As células lógicas consistem de lógica combinacional que criam *lookup tables*, que implementa funções como AND, OR, NAND e soma. Os *flip-flop* e as ligações com as células adjacentes também são implementadas na célula lógica. Várias células lógicas são agrupadas para criar uma única unidade, chamado um *slice*.

2.5.3 *Slice*

A próxima etapa na hierarquia é o *slice*. Um *slice* pode conter duas ou mais células lógicas (ver Figura 2.25).

As interligações internas foram omitidas na figura para manter a simplicidade, contudo cada célula lógica tem uma LUT, um *multiplexer*, e um registo com os seus próprios sinais de entradas e saídas. O *slice* tem um sinal de *clock*, *clock enable* e *set/reset* comum a cada célula lógica.

2.5.4 Blocos Lógicos Configuráveis

Numa etapa mais acima encontram-se os blocos lógicos ou CLBs. Usando exemplos de CLBs da Xilinx, algumas FPGAs têm dois *slices* em cada CLB e outras têm

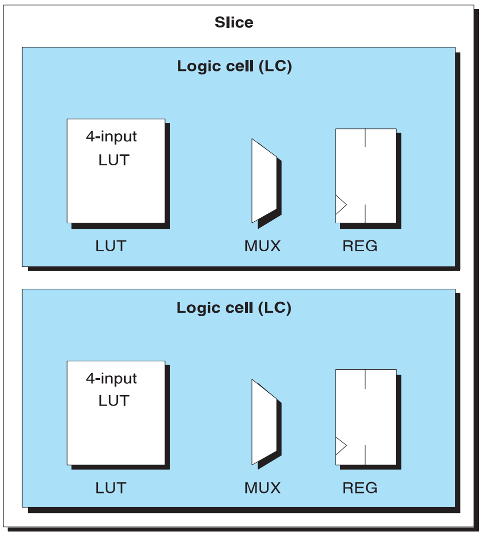


Figura 2.25: *Slice* com duas células lógicas (Clive Max, 2009).

quatro ou mais. Um exemplo de quatro *slices* pode ser visto na Figura 2.26.

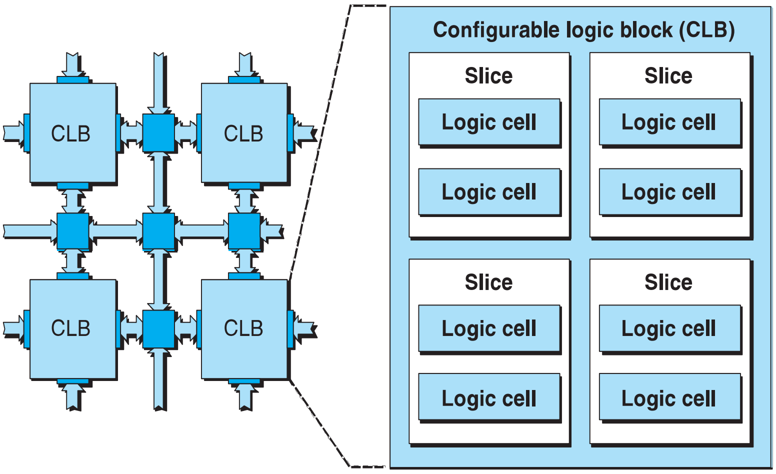


Figura 2.26: CLB com 4 *slices*(Clive Max, 2009).

Por exemplo, os CLBs da Virtex-4 têm quatro *silces* e a Virtex-5 e Virtex-6 têm dois *silces*. Mas os CLBs da Virtex-4 têm apenas oito células lógicas enquanto que a Virtex-5 e Virtex-6 têm doze células lógicas. Esta diferença deve-se ao facto de existir ligações mais rápidas do que outras, sendo as ligações entre as células lógicas as mais rápidas. À medida que se sobe na hierarquia as ligações tornam-se mais lentas e por isso as ligações entre *slices* são lentas e as ligações entre CLBs ainda mais lentas são. Por outro lado, as ligações entre CLBs são as mais flexíveis, entre os *slices* são menos flexíveis e entre as células lógicas são muito limitadas.

2.5.5 *Embedded Processor Core*

As FPGAs mais recentes e de gama alta, trazem formas diferentes de fazer uso aos processadores embebidos. Estes podem se de dois tipos *hard-cores* ou *soft-cores*.

Nos *hard-cores* o processador é incorporado na FPGA no momento de fabrico. Um exemplo de *hard-cores* é a FPGA Xilinx Virtex-II Pro com dois PowerPC 405 representado na Figura 2.27 (Fletcher, 2005).

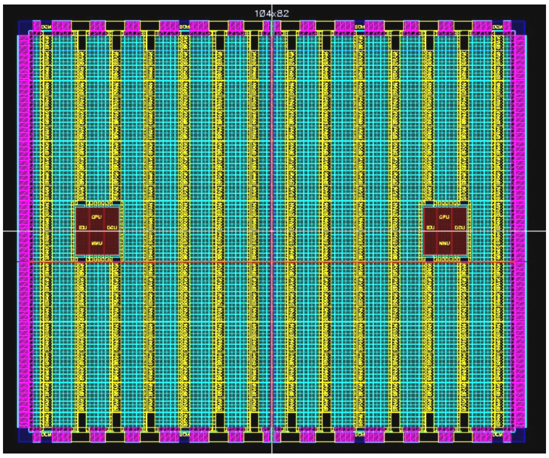


Figura 2.27: Xilinx Virtex-II Pro com os *hard-cores* PowerPC 405 (Fletcher, 2005).

Outras implementações de *hard-cores* nas FPGAs estão representadas na Figura 2.28(a), onde o processador é implementado em paralelo com o "tecido" da FPGA. A comunicação entre o processador e os periféricos é feita por um barramento dedicado.

Nos *soft-cores* (Figura 2.28(b)) o processador é usualmente especificado através de uma linguagem de descrição do *hardware*. Ao contrário dos *hard-cores*, os *soft-cores* têm de ser sintetizados e programados na FPGA.

Os *soft-cores* são mais lentos do que os *hard-cores*, contudo os *soft-cores* são flexíveis e podem ser implementados mais do que um ou só quando forem precisos, libertando recursos da FPGA (Clive Max, 2009). Em ambos, *hard-cores* e *soft-cores*, a memória, barramentos, periféricos internos e controladores têm de ser construídos através de uma linguagem de descrição do *hardware*.

2.5.6 Linguagem de Descrição do *Hardware*

A linguagem de descrição do *hardware* ou HDL (*Hardware Description Language*) é uma linguagem especializada usada para modular e descrever o *hardware*. O HDL

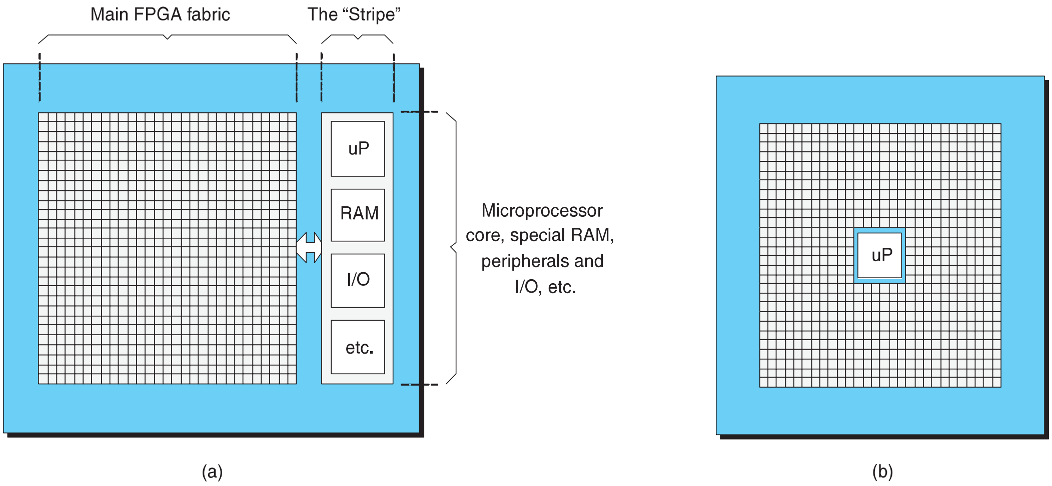


Figura 2.28: Processador embebido dentro (b) e fora (a) do "tecido" da FPGA (Clive Max, 2009).

permite, além da descrição por si própria, a possibilidade de simulação do *hardware* e a possibilidade de sintetizar *hardware* a partir da descrição. A maioria dos HDLs permite a simulação a vários níveis, facilitando o desenvolvimento. Ou seja, inicialmente o sistema pode ser concebido através de um conjunto de diagramas de blocos, numa segunda fase é feita uma descrição do sistema ao nível das funções (*flip-flops*, memórias, *multiplexers*, etc) e numa fase final é descrito o sistema da forma como vai ser efetivamente implementado: portas lógicas. A descrição através do HDL permite a simulação do sistema em todas estas fases (Dias, 2010).

Existem muitos exemplos de HDLs, mas as duas linguagens que estão especificadas através de *standards* e são as mais usadas atualmente são o VHDL (VHSIC HDL, ou seja, *Very High Speed Integrated Circuits Hardware Description Language*) e o Verilog.

A comunidade de utilizadores de HDLs não é de todo unânime qual das linguagens constitui uma melhor solução. É, no entanto, consensual que o Verilog é mais simples e que o VHDL tem mais potencialidades.

Existem genericamente quatro níveis de abstração: Sistema, *Register Transfer Level* (RTL), Lógico e Dispositivo. A abstração define quanto detalhe sobre o *design* existe numa descrição particular.

Os sistemas podem ser descritos de forma comportamental (definindo o seu comportamento), estrutural (definindo a sua estrutura) ou físico (definindo o que os compõe fisicamente). Esta situação está ilustrada na Figura 2.29, conhecida como o gráfico Gajski-Kuhn (Grout, 2008).

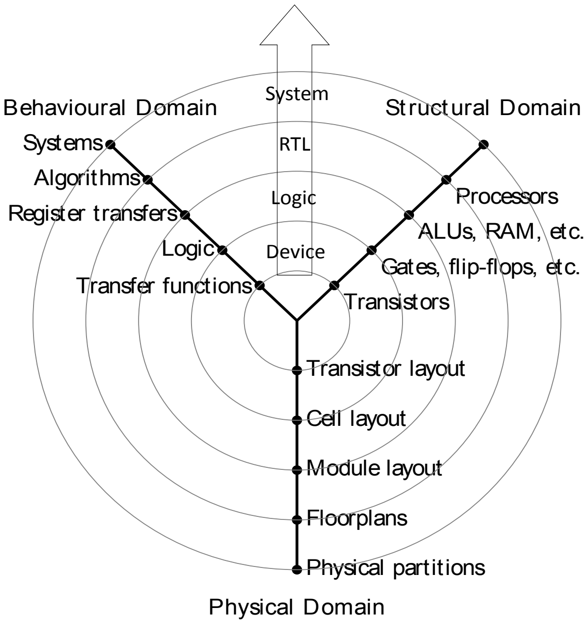


Figura 2.29: Níveis de abstração e tipos de descrição.

Nesta Figura 2.29 o sentido indicado pela seta aponta a direção de um aumento da abstração. No sentido inverso poderia ser indicado o aumento do detalhe. Desta forma percebe-se que ao abstrair será perdido a informação mais detalhada. Por exemplo, numa descrição comportamental ao nível do sistema não existirá detalhe sobre as funções dos transístores.

Num projeto de grande dimensão, a descrição ao nível do sistema é muitas vezes utilizada como um primeiro nível de validação da solução idealizada para o projeto. Após esta validação, a descrição não sintetizável ao nível de sistema é substituída por uma descrição de um dos níveis abaixo que seja sintetizável.

O nível de abstração correspondente ao dispositivo não é normalmente usado nas HDLs, uma vez que não apresenta grande vantagem em relação ao *design* dos transístores em silício. O *design* dos transístores em silício é uma representação gráfica das suas caraterísticas físicas que permite determinar todas as suas propriedades. A descrição ao nível do dispositivo obriga a especificar os parâmetros físicos correspondentes a estas caraterísticas, pelo que não representa uma grande vantagem (Dias, 2010).

2.5.7 *State Machine Charts*

Para a representação dos módulos em *hardware* é usado o *State Machine Charts* (Roth, 1997). Da mesma forma que os *flowcharts* são usados para representação do algoritmo de *software*, os *flowcharts* também podem ser úteis na representação dos circuitos digitais. Os *SM Charts* são uma alternativa aos *state graphs* mas com mais vantagens. Um *SM Charts* apresenta maior legibilidade e, se forem respeitadas certas regras, pode ser diretamente traduzido para uma linguagem HDL. Na Figura 2.30 estão representados os três componentes principais de um *SM Charts*.

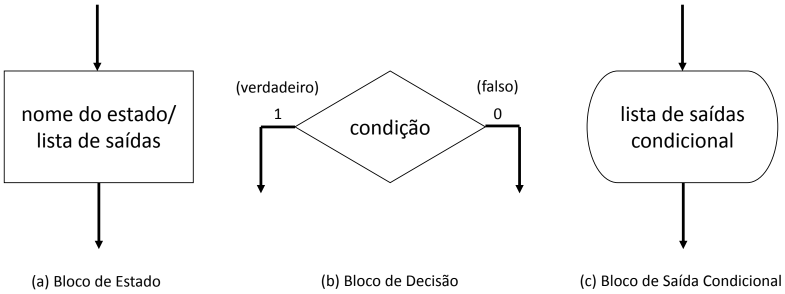


Figura 2.30: Componentes de um *SM Chart*.

Os estados do *SM Charts* são representados pelos blocos de estado. Estes blocos contêm o nome do estado, seguido por uma barra ("/") e uma lista opcional de saídas. O bloco de decisão é representado sob a forma de diamante e a sua condição de saída é uma expressão booleana. O bloco de saída condicional, contém uma lista de saídas condicional. Chama-se condicional porque este bloco só é colocado depois do bloco de decisão. A lista condicional do bloco depende do estado do sistema e das entradas.

Quando o sistema entra num bloco de estado, os sinais na lista de saídas do bloco tornam-se verdadeiros (nível lógico "1"). Entrando num bloco de decisão, a condição do bloco é avaliada e determina o caminho que tomará o sistema. No caso de entrar num bloco de saída condicional, tal como no bloco de estado, os sinais na lista condicional de saída tornam-se verdadeiros (nível lógico "1"). Se durante o fluxo de execução lógico do *SM Charts* não for encontrado algum sinal, por defeito, esse sinal é falso (nível lógico "0").

2.5.8 FPGA *Design Flow*

A Figura 2.31 mostra o processo de conceção de um sistema digital num FPGA. O desenvolvimento é usualmente especificado com uma linguagem de descrição do *hardware*. O projeto é depois simulado. São aplicados valores às entradas e os resultados das saídas são comparados para verificar que a lógica está correta. A seguir, a síntese lógica converte o HDL em funções booleanas. As ferramentas de síntese modernas produzem um esquema das funções, e o programador pode examinar estes esquemas com detalhe, bem como os avisos produzidos durante a síntese, para garantir que a lógica desejada foi produzida.

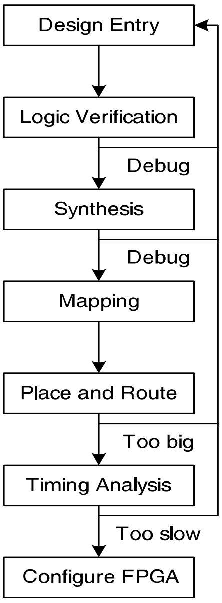


Figura 2.31: Processo de conceção de um sistema digital numa FPGA.

Quando os resultados de síntese são bons, a próxima ferramenta mapeia as funções para as CLBs. A ferramenta *place and route* determina que funções vão em que *lookup tables* e como elas estão ligadas entre si. O comprimento das ligações aumenta o atraso dos sinais, por isso os circuitos críticos devem ser colocados juntos. Se o projeto for demasiado grande para a FPGA, este deve ser redesenhado. A ferramenta de análise de *timing* compara as restrições temporais (por exemplo, se a velocidade pretendida do *clock* for 100 MHz) contra os atuais atrasos no circuito e indica se há erros. Se a lógica for lenta e não corresponder aos requisitos deverá ser redesenhada. Quando o projeto está correto, é gerado um ficheiro que especifica o conteúdo de todas as CLBs e a programação de todas as interligações da FPGA. Essa informação de configuração é armazenada numa RAM estática que é carregada para a FPGA cada vez que esta é ligada. A FPGA também pode carregar essa informação a partir de um computador, ou até ler de uma ROM quando a energia é aplicada pela primeira vez.

2.6 Linux e Sistemas Embebidos

Linux é um sistema operativo livre criado, em 1991, como *hobby* de um estudante universitário em ciências da computação. Linus Torvalds, criador do Linux, é hoje responsável pelo desenvolvimento do *kernel* para o sistema operativo Linux. Entusiastas (na maioria programadores) exploraram o crescimento da Internet no início da década de 90, para construir comunidades *online* com o objetivo de continuar o desenvolvimento do Linux. Estas comunidades foram as principais responsáveis pelas primeiras distribuições de *software* Linux, contendo todas as componentes de *software* necessárias para a instalação e uso do sistema operativo Linux sem a necessidade de técnicos avançados. Existem vários motivos responsáveis pela popularidade do Linux, como por exemplo (Yaghmour et al., 2008):

- **A Modularidade e Estrutura**: as funcionalidades estão separadas em módulos distintos. Dentro de cada módulo, as funcionalidades complexas estão subdivididas em funções mais simples;

- **Legibilidade**: o código é bastante legível e por isso é fácil, para os programadores que entendem a estrutura interna do Linux, reescrever o código;

- **Extensibilidade**: adicionar novas funcionalidades ao Linux é bastante simples. Se for necessário fazer modificações estruturais ou lógicas no código, estas são facilmente identificadas;

- **Configurabilidade**: é possível selecionar as funcionalidades que devem integrar a aplicação final. Esta característica do Linux permite selecionar apenas as componentes necessárias para o sistema final;

- **Previsibilidade**: durante a execução, o comportamento dos programas são coerentes com a sua implementação e não se tornam imprevisíveis;

- **Recuperação de Erros**: na ocorrência de uma situação problemática, o programa irá tomar medidas para recuperar o problema e, em seguida, alertar o Linux da ocorrência, registando-a e alertando o utilizador;

- **Longevidade**: os programas executam sem assistência durante longos períodos de tempo conservando a sua integridade;

A maioria dos programadores concordam que o Linux é um sistema operativo de qualidade e de confiança. Isto deve-se ao facto de o modelo de desenvolvimento ser *open source*, o que possibilita vários programadores contribuirem para o Linux através da identificação de problemas existentes, debater soluções e resolver de forma rápida e eficiente os problemas.

Nas últimas décadas o Linux evoluiu para um sistema operativo robusto e seguro, tornando-se o sistema operativo de suporte aos sistemas embebidos. Hoje, o Linux pode ser encontrado praticamente em todos os tipos de sistemas informáticos, desde o mais pequeno (por exemplo mp3) ao maior e mais complexo (como exemplos supercomputadores e servidores). A seguir será feita uma descrição das principais características do sistema operativo Linux e o que o destinge de outros sistemas operativos comerciais:

- Mais nenhum sistema operativo oferece um suporte tão alargado a variabilidade do *hardware* como o Linux.
- A distribuição livre pela Internet do Linux faz atrair, cada vez mais, comunidades *online* que ajudam ativamente no desenvolvimento do Linux.
- Possibilita uma grande variedade de protocolos de comunicação (CAN, USB, SPI, I2C, etc.) que permite, com facilidade, à interligação de outros sistemas informáticos, sensores e atuadores.
- Os grandes produtores de plataformas de desenvolvimento como a IBM, Texas Instruments, Samsung e Atmel dão suporte ao sistema operativo Linux.

Estas características e as anteriores fazem do Linux o sistema operativo mais utilizado na atualidade pelos sistemas embebidos.

2.6.1 O Linux *Kernel*

A Figura 2.32 apresenta a arquitetura genérica de um sistema operativo Linux. O *kernel* é o principal componente de um sistema operativo. Este fornece serviços de baixo nível para a camada da aplicação, permitindo a comunicação entre o *hardware* e o *software* do espaço utilizador. Alem disso, é responsável pela gestão dos processos na camada da aplicação, pela partilha e alocação dos recursos do sistema para esses processos, e proteger esses recursos alocados de corrupção por outros processos. Um dos recursos mais importante de um sistema é o CPU, que é tipicamente partilhado por vários processos de execução. Por isso, o *kernel* é responsável por decidir qual o processo que irá executar a seguir e por quanto tempo. Isto é chamado escalonamento de processos e é determinado pela política de preempção do sistema operativo que será visto com mais detalhe na Secção 2.7.2.

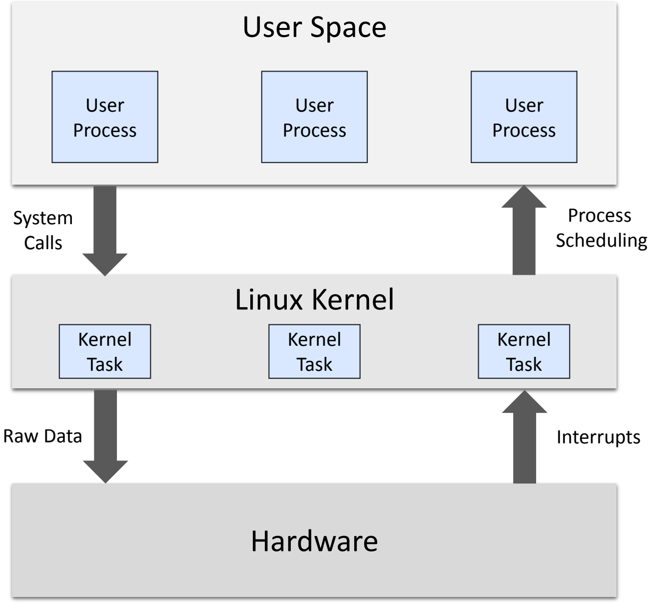


Figura 2.32: Linux *kernel* de um sistema embebido.

Além da gestão dos processos na camada de aplicação, o *kernel* fornece o *software* necessário para a gestão das interrupções de *hardware*, e fornece *drivers* para operar e controlar os dispositivos presentes no sistema embebido. Normalmente, no arranque do sistema, o *kernel* inicializa todos os dispositivos, tais como controladores de rede, temporizadores, dispositivos de entrada/saída e ISRs para lidar com os sinais gerados pelos dispositivos. Em seguida, são inicializadas as interrupções de *software* para processar as chamadas ao sistema, feitas pelos processos na camada da aplicação, para solicitar serviços do sistema operativo.

2.6.2 Funcionalidades do Sistema Operativo

As principais funções do sistema operativo são gestão e proteção da memória, escalonamento dos processos, interfaces com o *hardware*, gestão do sistema de ficheiros e comunicação com os dispositivos internos e externos. Será feita uma breve descrição de cada uma delas.

2.6.2.1 Gestão da Memória

Gestão de memória é o mecanismo fornecido pelo sistema operativo para alocar memória solicitada por um processo e libertá-la quando um processo termina. Outro requisito consiste em garantir que a memória, alocada anteriormente para um processo, que depois deixa de ser necessária, é libertada e disponibilizada para outro processo usar.

O espaço de memória disponível para qualquer processo é uma combinação de memória *cache*, memória física e memória virtual. A memória *cache* é uma parte integral do silício do CPU e armazena um conjunto pequeno de instruções e dados do processo atualmente em execução. A memória física (RAM) armazena instruções e dados para o processo atualmente em execução. A memória virtual é uma percentagem da memória de armazenamento em massa (Discos rígidos ou cartões SD) e normalmente armazena instruções e dados de processos que não cabem no espaço de memória física. As vantagens da utilização de memória virtual incluem a facilidade dos processos em aceder à memória partilhada, maior segurança no acesso à memória e os processos podem usar mais memória do que aquela fisicamente disponível, utilizando a técnica de *paging*. A memória *cache* pode ser acedida 10 a 100 vezes mais rápido do que a RAM, que por sua vez é 100 a 1000 vezes mais rápida que o acesso à memória de armazenamento em massa.

O *kernel* utiliza os recursos do CPU e o *hardware* responsável pela gestão da memória (MMU - *memory management unit*) para mover de uma forma transparente as instruções e dados entre a *cache*, memória física e memória virtual.

2.6.2.2 Proteção da Memória

Um requisito básico para um sistema operativo é garantir que cada processo do espaço utilizador execute de forma isolada no seu próprio espaço de endereçamento. Nenhum processo do espaço utilizador deve ser capaz de escrever no espaço de memória alocado para outro processo. Por isso, a MMU do CPU tem a função de proteger a memória destes processos. Existem sistemas embebidos que não têm uma MMU, e por isso, os processos do espaço utilizador têm de ser implementados com cuidado para não corromper variáveis de outro processos em execução.

Um caso especial é o próprio *kernel*, onde todas as tarefas dentro do *kernel* partilham o mesmo espaço de endereçamento. Por isso, os programadores do Linux *kernel* têm de escrever o código do *kernel* com muito cuidado para não escrever na memória de que eles não são responsáveis.

2.6.2.3 Escalonamento e Gestão das Tarefas

O *kernel* é responsável por escalonar a execução dos programas no espaço do utilizador. O escalonamento é feito de tal modo que o utilizador não se apercebe que o programa é impedido de executar e colocado em espera por vários milissegundos para permitir que outro programa possa executar. O escalonamento é feito através da atribuição, ao programa, de uma percentagem de tempo do CPU, ou os programas podem ser priorizados o que quer dizer que programas com maior prioridade têm maior percentagem de tempo do CPU.

Quando um programa de mais baixa prioridade é interrompido por um de mais alta prioridade e colocado em suspensão, diz-se que este sofreu preempção. A preempção rápida e determinística é a chave para os sistemas operativos de tempo real, que será discutida na secção posterior.

Um conceito fundamental do escalonamento é a divisão do tempo de execução em *slices* fixos. A cada processo é dado um pouco de tempo para a sua execução. Se a execução do processo não estiver concluída, este vai ser colocado em modo de suspensão até obter o seu *time slice* e depois disso continua a sua execução.

O *standard* POSIX (Computer Society, 1992), define uma *thread* como um fluxo de instruções dentro de um processo. Todas as *threads*, pertencentes ao mesmo processo, partilham o mesmo espaço de memória.

Um processo é definido pelo *standard* POSIX como sendo um único espaço de memória com um ou mais segmentos (*threads*) de execução nesse espaço. Como já referido, cada processo do sistema operativo tem o seu próprio espaço de memória. É unânime que os processos estão localizados no espaço utilizador em vez do espaço do *kernel*

Não existe uma definição consensual sobre a definição de uma tarefa. Nesta dissertação, a palavra tarefa é usada para descrever os componentes de código que estão a ser executados no espaço do *kernel*.

2.6.2.4 Interfaces com *Hardware*

Os sistemas operativos fornecem *softwares* de baixo nível, chamados *device drivers*, que controlam o *hardware* do sistema embebido. Os *drivers* podem ser chamados pelo *kernel*, ou podem operar de forma independente e interromper o *kernel* quando algo ocorre que precisa de ser tratado, por exemplo interrupções do *hardware*. Este último modo é chamado *interrupt-driven*.

A maioria dos sistema operativos trabalham com *drivers interrupt-driven*. Dados provenientes de fontes externas ao sistema embebido podem ser imprevisíveis e chegar a qualquer momento. Os dispositivos de interface como rato, teclado, porta série e interface de rede estão constantemente a sinalizar o CPU da chegada de dados. Esta sinalização é feita, de modo controlada, através dos pinos de interrupção fornecidos pelo CPU. Normalmente, quando acontece uma interrupção, o sistema operativo interrompe o CPU de executar o processo atual e salta para uma ISR que lê os dados do dispositivo. No fim da ISR, o controlo é retornado ao processo interrompido. Estas ISRs devem terminar o mais rápidos possível, ou seja, devem executar uma quantidade mínima de código, normalmente apenas fazem leituras de dados e escrevem esses dados num espaço de memória para armazenamento.

As interrupções podem ser atribuídas prioridades para que os processo que precisam de uma resposta mais rápida possam interromper processo de menor prioridade.

O tempo desde o acontecimento da interrupção, até à execução da primeira instrução dessa interrupção pelo CPU, é chamado de latência da interrupção.

Outro papel importante do *kernel* é a sincronização do espaço utilizador e o acesso partilhado aos recursos de *hardware*. O método tradicional é a utilização de *software locks* no *kernel* quando um programa do espaço utilizador utiliza um recurso físico, e desabilitar esses *software locks* quando o programa não precisar mais do recurso. Todos os outros programas do espaço utilizador estão proibidos (pelo *kernel*) de aceder ao recurso físico quando está bloqueado. Um tipo específico de *lock* utilizado pelo *kernel* é o *spinlock*, onde o programa fica num *loop* até o recurso de *hardware* estar disponível. Este método maximiza a resposta do programa.

2.6.2.5 Sistema de Ficheiros

Uma das funções mais importantes do sistema operativo é fornecer capacidade de armazenamento em massa da informação criada na memória pela execução dos processos do sistema. Essa informação é normalmente armazenada em ficheiros num sistema de ficheiros suportado pelo sistema operativo.

O *kernel* guarda a informação de todos os setores, do dispositivo de armazenamento (disco rígidos, cartões SD, etc.), usados para armazenar os blocos de informação dos ficheiros. O sistema operativo abstrai o utilizador dessa informação e este não se apercebe que os ficheiros são armazenados em blocos de informação não contínuos nos setores do dispositivo de armazenamento. A utilização de um sistema operativo permite ainda que a informação aparenta estar organizada em diretorias.

Nos sistemas embebidos é bastante comum utilizar o sistema de ficheiros *second extended filesystem* (Ext2). Ext2 foi o sistema de ficheiros padrão em diversas distribuições Linux, incluindo Debian e Red Hat Linux, até ser substituída pelo Ext3, que é compatível com Ext2. Ext2 ainda é o sistema de ficheiros de seleção para os dispositivos de armazenamento de dados baseados em *flash* (como cartões SD e USB *flash* drives).

2.6.3 *Toolchain*

Uma *toolchain* é um conjunto de ferramentas de desenvolvimento de *software* distintos que estão ligados por etapas específicas. A *toolchain* contém ferramentas como depuradores e compiladores para uma linguagem de programação específica, tais como C e C++.

Muitas vezes, a *toolchain* é utilizada para o desenvolvimento embebido através de ferramentas de *cross compiler.* As ferramentas da *toolchain* correm num sistema *host* de arquitetura específica (por exemplo PC x86) mas é gerado código binário para ser executado numa arquitetura diferente (por exemplo, ARM, PowerPC, MIPS ou AVR32). Esta é a forma típica de desenvolvimento de *software* para sistemas embebidos (Elinux.org, 2013).

2.7 Linux e Tempo Real

Os sistemas embebidos são o *core* das aplicações de tempo real, onde obter os resultados no tempo certo é tão importante quanto o cálculo da resposta correta. Assim, um sistema de tempo real é aquele que depende não só da resposta certa, mas também o momento em que o resultado é produzido. Se as restrições de tempo não forem satisfeitas, o sistema entra em falha por definição.

Tempo real não é sinónimo de rapidez. Uma aplicação executa rapidamente se for utilizado *hardware* mais rápido (por exemplo processador e memórias mais rápidos). Uma aplicação em tempo real pode conseguir, com *hardware* mais lento e mais barato, resultados das operações críticas nos tempos certos.

Existem dois tipos de sistemas de tempo real, os sistemas do tipo *soft* e *hard real-time*. Um sistema do tipo *hard real-time* é aquele em que uma atividade deve ser concluída antes de uma *deadline*. A *deadline* pode ser um intervalo de tempo ou pode ser a chegada de um evento. Sistemas do tipo *hard* entram em falha se não satisfizerem os requisitos temporais. Os sistemas *soft real-time* têm normalmente algum tipo de requisito temporal, mas se o sistema perder uma *deadline* ou a resposta chegar mais tarde a sua integridade não é comprometida.

Ao longo dos últimos anos, versões de tempo real do Linux amadureceram de tal forma que podem ser considerados candidatos viáveis para aplicações de tempo real. Aplicações bem sucedidas por grupos de pesquisa em universidades e indústrias, têm demonstrado que as modificações no *kernel* do Linux, para aplicações com requisitos temporais, são estáveis e seguras.

2.7.1 Latências e *Jitter*

A característica mais importante de um sistema operativo de tempo real é o quão responsivo é ao atender eventos internos e externos. Estes eventos podem ser interrupções externas por *hardware* e interrupções internas por *software* ou *timers* internos. Uma medição da resposta é a latência, que é o tempo entre a ocorrência de um evento e a execução da primeira instrução de código desse evento. A segunda medição para a resposta do sistema operativo de tempo real é o *jitter*, que corresponde à variação do período dos eventos periódicos. Para oferecer baixa latência e *jitter*, o sistema operativo deve garantir que qualquer tarefa do *kernel* seja capaz de sofrer preempção pela tarefa de tempo real.

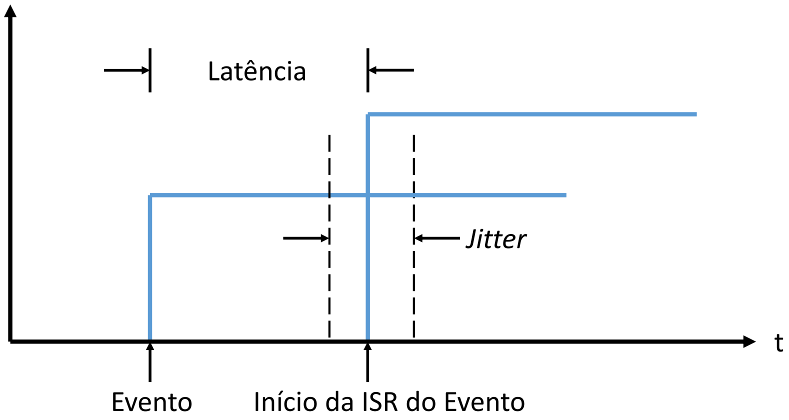


Figura 2.33: Latência e *jitter* de um evento (NIST, 2001).

Na Figura 2.33 está representado a latência e *jitter* de um evento. A latência, de uma interrupção por *hardware*, pode ser medida através de equipamentos de instrumentação externos ao sistema computacional como por exemplo osciloscópio, *timers*/contadores, etc. Na medição da latência é preciso ter em consideração os atrasos introduzidos pelo barramento do SoC (*system-on-chip*). Esses atrasos são originados, normalmente, por outros periféricos que podem momentaneamente estar a transferir dados pelo barramento e por isso bloqueiam o acesso ao CPU.

O tempo desde a interrupção gerada pelo sistema operativo, até a execução da primeira instrução da interrupção é a latência. Esta latência pode ser medida através do registo contador do *clock* do CPU. Este registo é incrementado à frequência do *clock*, por exemplo em cada 10 nanosegundos para um *clock* de 100MHz. Esta técnica de medição não afeta nem introduz qualquer atraso, uma vez que este registo é interno ao CPU e incrementado por *hardware*.

É fundamental a eliminação do *jitter* em aplicações com requisitos temporais exigentes e periódicos, por exemplo em aplicações de amostragem de sinais no setor da energia nuclear, onde os algoritmos de controlo são calculados com uma frequência fixa igual a taxa de amostragem dos sinais e qualquer instabilidade ou variação no tempo da amostra leva à imprecisão e falha do sistema.

2.7.2 Preempção

De forma a satisfazer os requisitos das aplicações mais exigentes, o *kernel* com características de tempo real tem de ser capaz de mudar o mais rápido possível de um contexto de baixa prioridade para uma tarefa de alta prioridade que precisa de tempo de CPU. Esta característica de interromper a atividade atual e colocar outra em execução chama-se preempção. O tempo da preempção, é o tempo mais longo que pode ser gasto à espera e define o pior caso possível de latência do *kernel*.

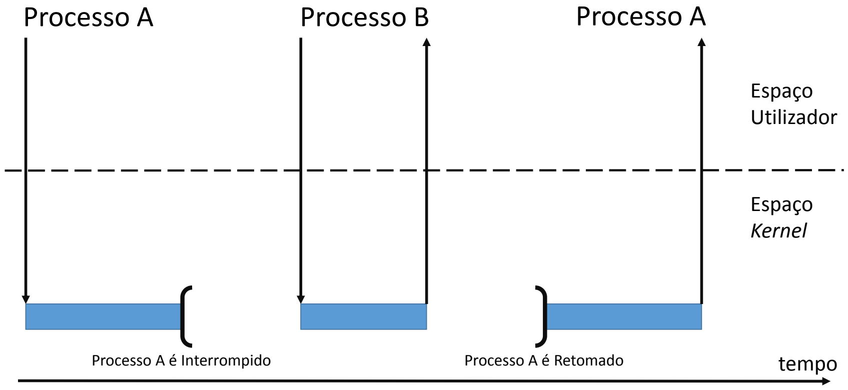


Figura 2.34: Preempção do *kernel*.

Fazer o *kernel* preemptivo significa que enquanto um processo está em execução no *kernel*, outro processo pode interromper o primeiro e ter a possibilidade de executar, embora o primeiro processo não tenha concluído o seu processamento. A Figura 2.34 mostra essa situação.

Nesta Figura 2.34, o processo A inicia a sua execução, entretanto, o processo B com maior prioridade é acordado por uma interrupção. O *kernel* interrompe o processo A e atribui o CPU ao processo B, sem que o processo A tenha terminado. Quando o processo B termina a sua rotina de serviço a interrupção o *kernel* volta a atribuir o CPU ao processo A.

2.7.2.1 Modelos de Preempção

A versão 2.6 do kernel oferece três modelos de preempção:

- **PREEMPT_NONE**, *No forced preemption (server)*: Neste modelo não é forçada a *preempção* do *kernel*. A latência global deste modelo é aceitável mas podem ocorrer alguns atrasos Este modelo é ideal para maximizar o *throughput* global sendo por exemplo o modelo mais adequado para servidores;
- **PREEMPT_VOLUNTARY**, *Voluntary kernel preemption (desktop)*: Este modelo reduz a latência do *kernel* através da inserção de pontos de preemp-

ção no *kernel.* Estes pontos, introduzidos em posições estratégicas, permitem reduzir a latência global do sistema. O modelo sacrifica o *throughput* global do sistema mas em troca melhora a resposta do sistema;

- **PREEMPT_DESKTOP**, *Preemptible kernel (low-latency desktop)*: Este modelo reduz a latência ao tornar todo o *kernel* (exceto nas regiões críticas do sistema) preemptivo. O modelo permite a sua utilização em sistemas do tipo *soft real-time*, como por exemplo aplicações de áudio e multimédia. O *throughput* global é reduzido mais do que no caso anterior, mas com reduções ainda maiores na latência do sistema;

É da responsabilidade do programador escolher o modelo de preempção mais adequado para sua aplicação. No caso de nenhum dos modelos satisfazer os requisitos da aplicação é possível intervir diretamente no *kernel* de forma obter melhores resultados. A secção seguinte descreve outros métodos utilizados para aumentar a performance e *throughput* do sistema.

2.7.3 Técnicas de Aceleração

Existem vários métodos para potenciar a utilização e a performance de um sistema operativo tempo real. Métodos que se baseiam em *hardware*, consistem em empregar processadores mais rápidos ou *hardware* dedicado, geralmente recorrendo às *field-programmable gate array's* (FPGAs), co-processadores ou outro tipo de dispositivos aceleradores. Outros métodos, recorrem a *software* e são geralmente os mais utilizados.

2.7.3.1 Técnicas de Aceleração por *Software*

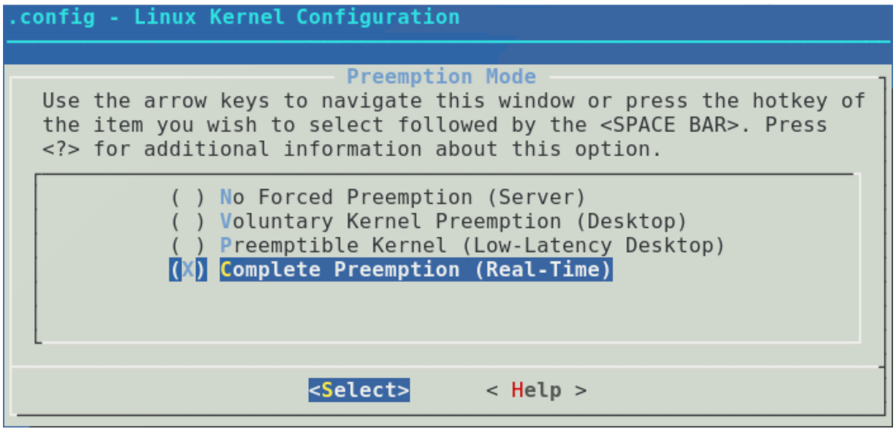
Técnicas de *software* são as mais utilizadas na comunidade *open source* uma vez que são as mais baratas e acessíveis. Exemplos destas técnicas incluem a utilização de várias linguagens de programação para o desenvolvimento de *software* para os sistemas embebidos. Por exemplo, pode ser usado linguagem C para desenvolver a maior parte do sistema e usar *assembly* para as regiões críticas, onde o tempo de execução deve ser controlado com detalhe. Porém a técnica mais utilizada nos sistemas embebidos consiste em empregar um *patch* ao código original do *kernel.*

Real-Time Kernel Patch

O objetivo do *rt patch* é melhorar a latência e tempo de resposta do Linux. O *rt patch* faz isso ao alterar centenas de locais no *kernel* do Linux e linhas de código.

O *rt patch* tem vindo a ser usado cada vez mais em toda a indústria. Isto deve-se ao facto de o seu desenvolvimento ser aberto e focado em aplicações do tipo *hard real-time*, o que tem permitido a utilização do Linux RT em áreas desde o processamento profissional de áudio ao controlo de processos exigentes na industrial (Fu and Schwebel, 2013).

O *rt patch* converte o Linux para um *kernel* completamente preemptivo. Este *patch* adiciona um modelo de preempção extra designado de PREEMPT_RT que pode ser visto na Figura 2.35.



```
.config - Linux Kernel Configuration
                     Preemption Mode
 Use the arrow keys to navigate this window or press the hotkey of
 the item you wish to select followed by the <SPACE BAR>. Press
 <?> for additional information about this option.

          ( ) No Forced Preemption (Server)
          ( ) Voluntary Kernel Preemption (Desktop)
          ( ) Preemptible Kernel (Low-Latency Desktop)
          (X) Complete Preemption (Real-Time)

               <Select>      < Help >
```

Figura 2.35: Configuração do *kernel real-time*.

Neste modelo estão incluídas as modificações descritas na Secção 2.7.2.1, e ainda um conjunto de funcionalidades discutidas a seguir. Estas modificações permitem ao *kernel* do Linux lidar com aplicações do tipo *hard real-time* onde a previsibilidade, latência e a repetibilidade são fatores críticos.

Funcionalidades do *Real-Time Kernel*

A escolha do novo modelo de preempção, como indicado na Figura 2.35, habilita várias funcionalidades extras do *kernel* que permitem ao sistema lidar com aplicações do tipo *hard real-time*. Uma das funcionalidades mais importantes é a substituição em zonas do *kernel* bloqueadas, com *spinlocks*, por *rtmutexes* e assim passam a ser preemptíveis. Os *mutexes*, ao contrário dos *spinlocks*, permitem a interrupção durante a execução. Esta funcionalidade reduz a latência do sistema, mas acrescenta *overhead* no processamento dos *mutexes* uma vez que são preemptíveis.

Outra alteração é no serviço de atendimento às interrupções, o sistema passa a permitir a escolha entre *Hard IRQ* e *Soft IRQ*:

- **CONFIG_PREEMPT_HARDIRQ**: Ao escolher esta funcionalidade, força a execução das ISRs no contexto de processo. Isto permite ao desenvolvedor controlar a prioridade das ISRs porque passam a ser entidades escalonáveis. Por isso, também se tornam entidades preemptíveis o que significa que as ISRs de maior prioridade são tratadas em primeiro lugar. Este recurso é muito poderoso, principalmente em arquiteturas de *hardware* que não impõem prioridades nas interrupções, possibilitando ao programador a escolha das prioridades mais adequadas para a sua aplicação com requisitos de tempo real;
- **CONFIG_PREEMPT_SOFTIRQ**: Esta funcionalidade também permite reduzir a latência do sistema, mas menos que a funcionalidade anterior. Aqui as ISRs executam dentro do contexto de um *kernel softirq daemon* (*ksoftirqd*). Se o *kernel* for configurado para o tempo real, a tarefa do *kernel ksoftirqd* é promovida a tarefa de tempo real e o seu escalonamento é atribuído ao SCHED_FIFO, onde a primeira a entrar em espera é a primeira a sair e colocada em execução;

A escolha destas funcionalidades reduz o desempenho do sistema, mas em troca obtém-se maior previsibilidade e redução da latência. Estas duas funcionalidades permitem ao programador escolher se pretende forçar mais ou menos a preempção do *kernel* durante as ISRs (Hallinan, 2007).

2.7.3.2 Técnicas de Aceleração por *Hardware*

Empregar processadores mais rápidos, como os DSPs, são bastante comuns para aumentar a performance do sistema. Este tipo de método é bastante mais simples do que, por exemplo, implementar *hardware* dedicado. As FPGAs, discutidas na Secção 2.5, são a chave para o *high-performance computing* (HPC). À medida que esta tecnologia se torna mais acessível, as FPGAs têm vindo a ter um papel cada vez mais importante no aumento da performance de um sistema operativo.

O uso de FPGAs em HPC é destinado para aplicações que passam a maioria do tempo a fazer operações matemáticas de forma intensiva no *kernel*. A utilização da tecnologia FPGA permite reduzir o *overhead* do *kernel* ao implementar estas operações em *hardware*. Desta forma, e aproveitando o paralelismo do *hardware*, as FPGAs podem exceder o poder dos DSPs ao realizar mais instruções por ciclo de *clock*. Adicionalmente, ao controlar as entradas/saídas de dados ao nível do *hardware* permite obter tempos de resposta mais rápidos uma vez que envolve menos camadas de abstração do sistema.

Uma forma de aumentar a performance, diminuir latências e aumentar o determinismo do sistema operativo, seria migrar os *device drivers* do *kernel* para *hardware*. Assim, o *overhead* do sistema operativo seria reduzido drasticamente uma vez que não seria necessário fazer a maioria do processamento. Por exemplo, se um *driver* estiver com uma frequência elevada a ler os valores digitais dos GPIOs do sistema e ainda fizer tratamento matemático aos dados lidos, todo este processamento será dispendioso para o *kernel*. Mas, se a leitura dos sinais e o tratamento matemático for implementado ao nível do *hardware*, o *driver* apenas teria que ler os resultados dessas operações, reduzindo drasticamente o *overhead* do *kernel*.

2.8 Outras Funcionalidades

Nesta secção serão mencionadas outras funcionalidades e tecnologias relevantes nesta dissertação.

2.8.1 *Direct Memory Access*

A tecnologia *direct memory access* (DMA) é uma ferramenta poderosa que permite a transferência de grandes quantidades de dados sem envolver e sobrecarregar o CPU.

Num sistema de tempo real o processador é usado inúmeras vezes por um processo para produzir os melhores resultados possíveis. A fim de reduzir o tempo de processamento que é desperdiçado em transferir dados dos I/Os para a memória principal, vários sistemas incluem um periférico que realiza as operações de transferência de dados sem usar o processador.

O DMA ajuda a produzir melhores resultados em termos de uso do CPU e, consequentemente, maior *throughput* do sistema. O DMA é usado nos seguintes tipos de transferências de dados:

- Transferência de Memória para Memória;
- Transferência de Memória para Periférico;
- Transferência de Periférico para Memória;
- Transferência de Periférico para Periférico;

Um *device driver* pode usar o DMA de forma síncrona ou assíncrona. No primeiro caso, as transferências de dados são acionadas pelos processos, enquanto no segundo caso, as transferências de dados são acionadas por dispositivos de *hardware*.

Um exemplo de sincronismo pode ser uma placa de som que está a reproduzir uma música. Uma aplicação no espaço utilizador escreve os dados do som (amostras) num arquivo associado a placa de som. O *driver* da placa de som acumula estas amostras num *buffer* no *kernel*. Ao mesmo tempo, o *driver* instruí a placa de som para copiar as amostras do *buffer* do *kernel* para a placa num *timing* bem definido. Quando a placa de som termina a transferência de dados, gera um interrupção, e o *driver* verifica se o *buffer* do *kernel* ainda contém amostras para produzir. No caso de ainda existir amostras o *driver* ativa outra transferência de dados.

Um exemplo de assíncrono é uma placa de rede que recebe pacotes de dados (*frames*) via LAN. O periférico armazena as *frames* numa memória de I/O partilhada, em seguida gera uma interrupção. O *driver* da placa de rede recebe a interrupção, e depois indica ao periférico para copiar a *frame* na memória partilhada para um *buffer* do *kernel*. Quando a transferência de dados é concluída, a placa de rede gera outra interrupção, e o *driver* notifica o *kernel* da receção da *frame*.

Cada transferência de dados envolve (pelo menos) um *buffer* de memória, que contém os dados a serem lidos ou escritos pelo dispositivo de *hardware*. Em geral, antes de ativar a transferência, o *driver* deve garantir que o dispositivo possa aceder diretamente aos locais da memória.

2.9 Conclusões

Uma vez que se pretende implementar um sistema de aquisição de sinais em tempo real baseado em Linux, foi intenção do autor descrever todos os pontos chave que contribuem para o sucesso deste trabalho.

Inicialmente é feita uma descrição com todas as componentes essenciais que constituem um SAS. Porém estas componentes não fazem os SAS um sistema capaz de funcionar corretamente num ambiente do tipo *hard real-time*, por isso, é necessário estudar as técnicas utilizadas para tornar os sistemas de aquisição num sistema apto para ambientes com requisitos temporais. Para isso, é necessário recorrer a técnicas de *software* e *hardware*.

Assim, foi feita uma descrição com as principais técnicas que contribuem para o sucesso deste trabalho. As técnicas de aceleração por *software*, é possível melhorar a latência e tempo de resposta do Linux através do *patch rt*. A técnica de aceleração por *hardware* consiste em implementar partes de *software* em *hardware* dedicado, recorrendo a tecnologia FPGA.

Capítulo 3

Plataforma de Desenvolvimento

Após recolha bibliográfica, pretende-se neste capítulo fazer uma breve descrição da plataforma desenvolvimento utilizada nesta dissertação. Como foi discutido anteriormente, a utilização da tecnologia FPGA permite diminuir latências, aumentar o determinismo e performance do sistema operativo. Por isso, foi utilizada uma plataforma que integra a tecnologia FPGA, com pelo menos um *hard-core*, com capacidade de customizar o SoC e suportado pelo *kernel* do Linux.

3.1 XUP Virtex-II Pro

Na Figura 3.1 está representada a plataforma de desenvolvimento XUP Virtex-II Pro. Esta plataforma fornece uma coleção abrangente de periféricos que podem ser usados para criar o *system-on-chip* suportado pelo *kernel* do Linux, e inclui:

- **Xilinx XC2VP30**: É uma FPGA Virtex-II Pro com dois *hard-cores* PowerPC 405 (ver Figura 2.27) e uma grande quantidade de lógica configurável. Mais detalhes podem ser vistos na Tabela 3.1;
- **Memória DDR SDRAM**: Esta memória é essencial para correr o *kernel* do Linux, aplicações do usuário e para a utilização do *framebuffer* de vídeo do XSGA. Esta memória pode ir até 2 GB;
- **CompactFlash**: Permite a utilização de uma unidade de memória não volátil através do controlador Xilinx SystemACE. Esta memória *flash* permite configurar a FPGA, arrancar o *kernel* do Linux e utilizar o sistema de ficheiro do Linux quando a plataforma é alimentada;

- **Porta Série RS-232**: É possível utilizar uma porta série para efeitos de comunicação com outro sistema computacional e enviar comandos;

- ***Clock* de 100 MHz**: A plataforma contém um *clock* de 100 MHz. Este *clock* é utilizado, por exemplo, para fornecer aos *hard-cores* um *clock* até 300 MHz através do Xilinx DCM (*digital clock manager*). O Xilinx DCM é responsável por dividir ou multiplicar o *clock* do sistema;

- ***On-board* XSGA**: A saída de vídeo da plataforma pode ir até 1200 x 1600 com uma taxa de atualização de 70 Hz;

- **Outros**: Esta plataforma tem seis conectores de expansão ligados a 80 Pinos da FPGA para propósitos I/O com proteção contra sobre tensões. Cada um destes pinos pode ser usado por exemplo para implementar protocolos de comunicação como I2C, SPI, RS232, CAN, etc.;

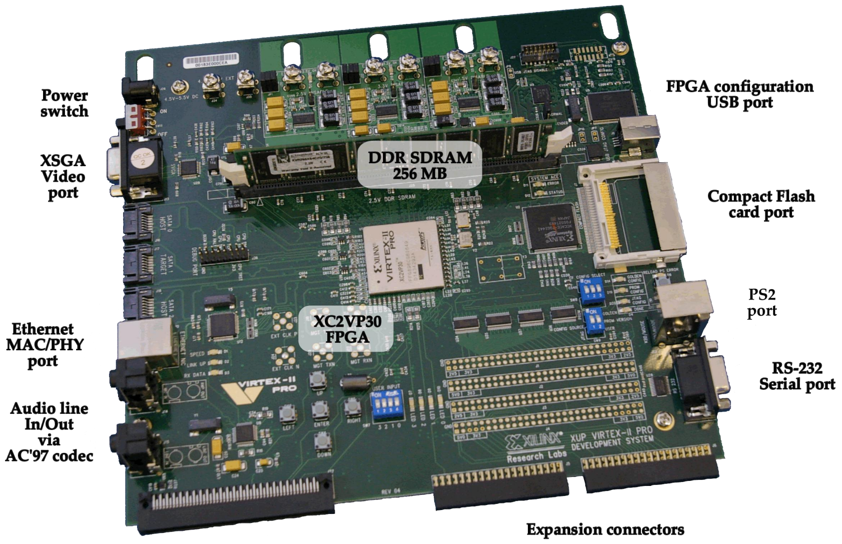
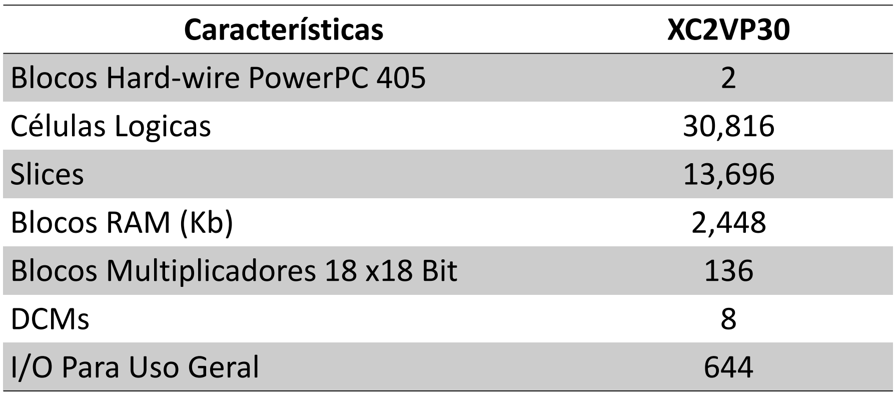


Figura 3.1: Plataformas de Desenvolvimento XUP Virtex-II Pro.

A Xilinx fornece uma ferramenta de desenvolvimento para FPGAs chamada *Integrated Software Environment* (ISE). O ISE tem ferramentas para verificar e sintetizar um projeto lógico, desenvolvido numa linguagem HDL (Verilog/VHDL), e produzir o ficheiro de configuração *.bit* (*bitstream*). Em seguida, o *bitstream* é utilizado para configurar a FPGA. Além disso, a Xilinx também fornece o *Embedded Development Kit* (EDK) para sistemas de processamento embebidos. Este *kit* inclui o *Xilinx Platform Studio* (XPS) e o *Software Development Kit* (SDK), bem como toda a documentação e IPs que são necessários para o desenvolvimento de *hardware* e *software* dos *hard-cores* PowerPc 405 .

Tabela 3.1: Características da FPGA Xilinx XC2VP30 (Xilinx Inc., 2011).



Características	**XC2VP30**
Blocos Hard-wire PowerPC 405	2
Células Logicas	30,816
Slices	13,696
Blocos RAM (Kb)	2,448
Blocos Multiplicadores 18 x18 Bit	136
DCMs	8
I/O Para Uso Geral	644

O XPS inclui um IDE (*integrated development environment*) para desenvolvimento de *hardware* para aplicações embebidas. Esta ferramenta inclui um *Base System Builder* que permite a criação de um sistema funcional em poucos minutos. O XPS também inclui outras ferramentas de *design* para configuração rápida de arquitetura de sistemas embebidos, barramentos e periféricos.

O SDK é um ambiente de desenvolvimento de *software* baseada no Eclipse IDE e inclui uma *toolchain* completa, com compiladores C/C++ e depuradores, para os processadores PowerPC e MicroBlaze da Xilinx.

3.1.1 Unidades de Processamentos

O PowerPC 405 opera com instruções num *pipeline* de cinco estágios que consistem no *fetch*, *decode*, *execute*, *writeback* e *load write-back.* Um *pipeline* de cinco estágios aumenta o *throughput* e facilita a operação do CPU acima dos 200 MHz.

O CPU suporta ordenação *big-endian* ou *little-endian* para as instruções armazenadas na memória externa. Internamente a arquitetura PowerPC é *big-endian*, a ICU (*Instruction Cache Unit*) reorganiza as instruções armazenadas na memória como *little-endian* para o formato *big-endian.* Portanto, o ICU contém sempre as instruções em formato *big-endian* para que a ordem dos bytes esteja correta para a unidade de processamento.

Além do PowerPc 405 a plataforma da suporte ao *soft-core* MicroBlaze. Ao contrário do *hard-core* PowerPc 405, o MicroBlaze pode ser configurado pelo utilizador:

- Tamanho da *cache*;

- Profundidade do *pipeline* (3 estágios ou 5 estágios);
- MMU;
- Personalização dos barramentos;
- Periféricos integrados (multiplicadores, divisores, etc.);

A versão otimizada do MicroBlaze utiliza um *pipeline* de três estágios e sacrifica a frequência de *clock* para 100 MHz. Porém a versão de desempenho expande o *pipeline* para cinco estágios, permitindo frequência de *clock* máxima de 210 MHz. O MicroBlaze apenas suporta ordenação *big-endian*.

3.2 Conclusões

A plataforma de desenvolvimento suporta vários periféricos essenciais para dar suporte ao sistema operativo Linux, como por exemplo suporte de leitura de cartões CompactFlash para armazenamento do sistema de ficheiro Linux.

Nesta dissertação foi utilizado o processador PowerPC 405 uma vez que este é um processador *hard-core* e suporta maiores frequências de *clock* do que o MicroBlaze. Além disso a licença disponível não permite utilizar a versão de desempenho do MicroBlaze, e por isso a frequência de *clock* máxima fica limitada em 100 MHz com um *pipeline* de apenas três estágios.

Capítulo 4

Sistema de Aquisição

Este capítulo descreve a implementação do sistema de aquisição de sinais, que será aplicado no domínio dos sistemas de energia, para monitorização das componentes do sistema elétrico trifásico. Aqui será apresentada a estrutura do sistema de aquisição, as componentes fundamentais do sistema e a interação entre elas. Este capítulo é focado no desenvolvimento do SoC com suporte ao Linux utilizando o *Base System Builder*, no *driver* desenvolvido em *hardware* para comunicação e controlo do conversor analógico-digital, no driver em *software* para comunicação com o *driver* em *hardware* e no desenvolvimento da aplicação de monitorização do sistema.

4.1 Estrutura do Sistema

O sistema de aquisição está dividido em quatro componentes essenciais:

- Uma componente de acondicionamento de sinal;
- Um conversor analógico-digital (ADC);
- A plataforma de desenvolvimento XUP Virtex-II Pro;
- Uma componente de monitorização;

A Figura 4.1 apresenta o diagrama com componentes mencionadas acima, alguns módulos importantes de cada um destes componentes e os principais pontos de ligação.

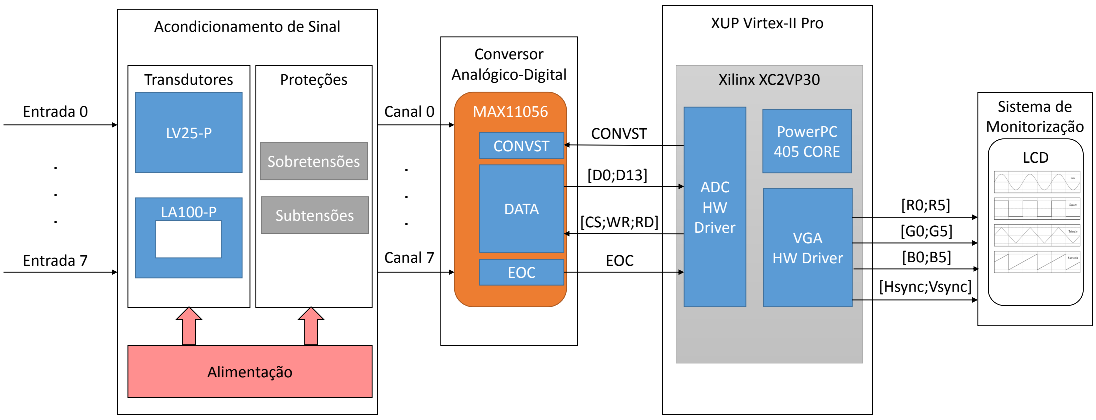


Figura 4.1: Componentes do Sistema de Aquisição.

4.2 Acondicionamento de Sinal do Sistema

Como foi visto na Secção 2.2, os sensores são os elementos que permitem fazer a interface entre os fenómenos físicos e os SAS ao converter fenómenos físicos em sinais elétricos. Porém, se o fenómeno físico for de natureza elétrica é necessário utilizar transdutores para transformar esses fenómenos num valor analógico correspondente aos valores de máxima escala admitidos pelo conversor analógico-digital. Assim, o *hardware* de acondicionamento de sinal é constituído por três transdutores de tensão e três transdutores de corrente. Estes transdutores estão representados nas Figuras 4.2 e 4.3 e ambos funcionam sob o princípio de efeito Hall (Edwin Halbbert Hall – 1879).

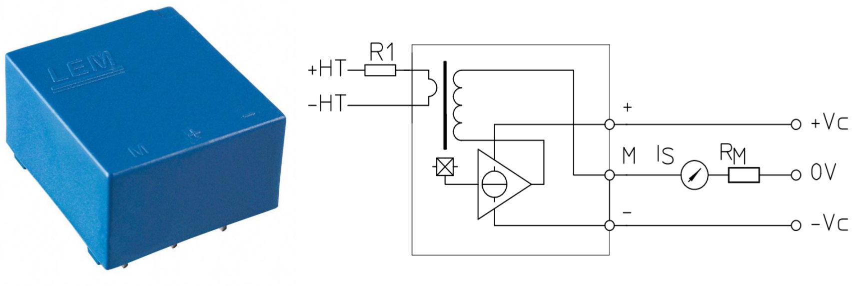


Figura 4.2: Transdutor de tensão LV25-P.

O transdutor de tensão permite medir até 500 V (valor eficaz) no lado primário, gerando uma corrente proporcional no lado secundário 2.5 vezes maior do que a corrente de entrada do primário. A corrente de entrada do primário está limitada a uma corrente máxima de 10 mA (valor eficaz), por isso é necessário inserir uma resistência em série com o primário do transdutor para limitar a corrente. Este transdutor é alimentado com uma tensão bipolar de $\pm 12V$ ou $\pm 15V$, e precisa de uma resistência no secundário para converter a sua saída em corrente, numa tensão, sendo essa tensão convertida posteriormente pelo conversor analógico-digital.

O transdutor de corrente permite medir até 105 A (valor eficaz), sendo a sua saída uma corrente proporcional 2000 vezes menor, tem a capacidade de medir correntes com uma gama de frequência compreendida entre 0 e 200 kHz, e possui um isolamento que pode ir até os 2500 V (valor eficaz). Tal como o transdutor de tensão é necessário converter a saída de corrente numa tensão. Este transdutor também é alimentado com uma tensão bipolar de de $\pm 12V$ ou $\pm 15V$.

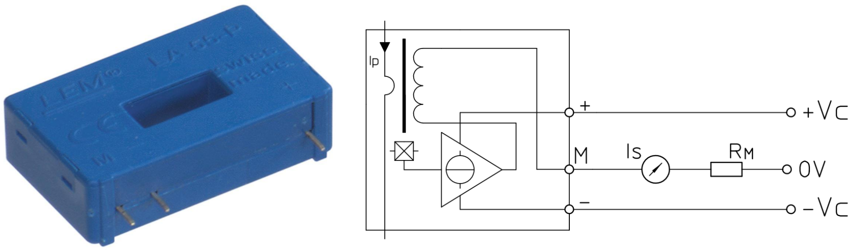


Figura 4.3: Transdutor de corrente LA100-P.

A utilização destes transdutores tem como objetivo ligar o SAS a um sistema elétrico trifásico. Um sistema trifásico possui três fases de tensão alternada com valor de pico de 325 V. Por isso, para dimensionar os restantes componentes estabeleceu-se que a tensão máxima a medir seria de 350 V (valor de pico) e a corrente de linha até 100 A (valor de pico).

Assim sendo, o transdutor de tensão LV25-P (LEM Co., 2011), como mostra na Figura 4.2, precisa de duas resistências de polarização, R_1 e R_M para manter as correntes nos enrolamentos dentro dos valores limites estabelecidos pelo fabricante. No enrolamento primário definiu-se uma tensão máxima de 325 V e como a corrente está limitada pelo fabricante até 10 mA, temos para:

$$R_1 = \frac{V_{pn}}{I_{pn}} = \frac{350V}{10mA} = 35k\Omega \tag{4.1}$$

$$P_{R_1} = R_1 * I_{pn}^2 = 3,5W \tag{4.2}$$

O valor prático para R_1 foi de 2 x 18kΩ, com um potência superior para evitar sobreaquecimento e consequente desvio térmico no valor resistivo das mesmas. Assim, com este valor de resistências é possível medir no enrolamento primário uma tensão máxima de 360 V.

No enrolamento secundário, sabendo que a corrente é 2.5 vezes maior à corrente de entrada do primário, a resistência R_M determina a tensão máxima de saída do transdutor que será lida pelo conversor analógico-digital. Deste modo, e tendo em consideração o ADC descrito na Secção 4.3, definiu-se que a tensão máxima de saída do transdutor seria de 5 V e assim obtém-se uma resistência de medida:

$$I_{sn} = 2.5 * I_{pn} = 2.5 * 10mA = 25mA \tag{4.3}$$

$$R_M = \frac{V_{max}}{I_{sn}} = \frac{5V}{25mA} = 200\Omega \tag{4.4}$$

De acordo com o fabricante os valores das resistências e correntes estão dentro dos valores nominais definidos.

O transdutor de corrente LA100-P (LEM Co., 2011), ao contrário do anterior apenas precisa de uma resistência de polarização, R_M (ver Figura 4.3), no terminal de saída do transdutor. A corrente é medida ao fazer o condutor elétrico passar através da cavidade existente no transdutor (ver Figura 4.3).

Para este transdutor definiu-se uma corrente máxima de 100 A e no lado secundário a corrente é 2000 vezes menor à corrente medida, e tendo em conta a resistência de polarização R_M máxima definida pelo fabricante varia entre 100Ω a 110Ω:

$$I_s = \frac{I_p}{2000} = \frac{100A}{2000} = 50.0mA \tag{4.5}$$

$$R_M = 100\Omega \Rightarrow V_{max} = 100\Omega * 50.0mA = 5V \tag{4.6}$$

Como já foi referido, devido ao conversor analógico-digital, foi definida uma tensão máxima à saída do transdutor de 5 V que está dentro dos limites deste transdutor.

Estes transdutores oferecem medições bastantes precisas (com um erro menor 5%) e um tempo de resposta menor que 25μs, o que os torna componentes bastante bons para medições nos sistemas elétricos. É ainda importante salientar que a utilização do SAS para medição de outras grandezas físicas apenas depende da alteração destas componentes.

4.3 Conversor Analógico-Digital

O ADC é responsável por estabelecer a ligação entre o mundo analógico e o mundo digital. Este destina-se a converter os sinais analógicos e contínuos, em sinais digitais discretos. Tendo em conta os transdutores utilizados, e sabendo que irão operar num sistema elétrico onde as tensões e correntes são alternadas, a tensão de saída destes transdutores é bipolar. Isso significa que a tensão máxima de saída varia entre $\pm 5V$. Além disso, tendo em conta que o objetivo é ligar o SAS a um sistema elétrico trifásico, são precisos pelos menos seis canais (três tensões e três correntes) para poder medir todas as grandezas. Tendo em conta estes requisitos foi selecionado um ADC da MAXIM que está representado na Figura 4.4.

O MAX11056 é um conversor analógico-digital com capacidade de converter simultaneamente oito canais de 14-bits. Algumas características deste ADC, trazem várias vantagens para utilização destes nos sistemas elétricos como:

- **Elevada impedância de entrada (1 GΩ)**: Esta impedância permite leituras bastante precisas sem ser preciso utilizar um impedância externa quando o ADC está a converter o sinal de entrada;
- **Medição bipolar**: Esta característica permite simplificar o sistema ao permitir medir diretamente sinais de saída de transformadores;
- **8 canais**: A presença de oito canais neste ADC permite medir todas as componentes dos sistemas elétricos;
- **Modo *stand-by***: Este ADC permite o modo de operação *stand-by*, permitindo baixar o consumo de energia quando não estiver em funcionamento;

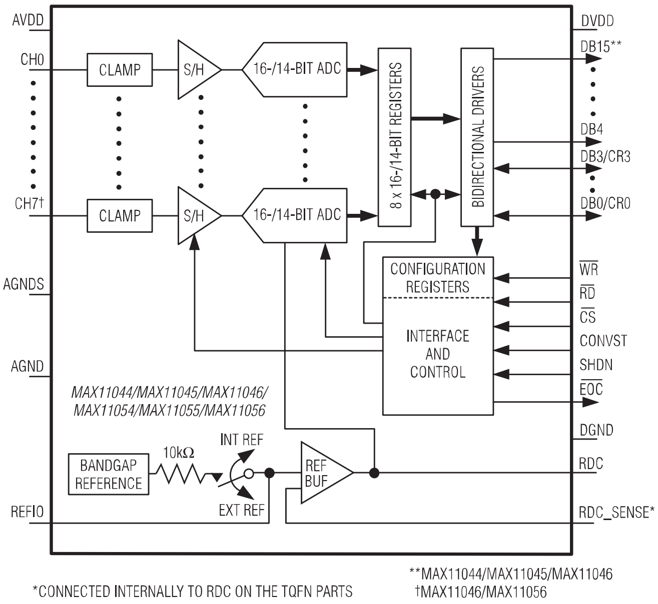


Figura 4.4: Diagrama funcional do ADC MAX11056.

- **Tempo de conversão de 3μs**: O ADC permite simultaneamente amostrar nos 8 canais numa frequência até 250.000 amostras por segundo;

No MAX11056 os 14 bits de dados são acedidos de forma paralela e o processo de leitura dos 8 canais está descrito na Figura 4.5.

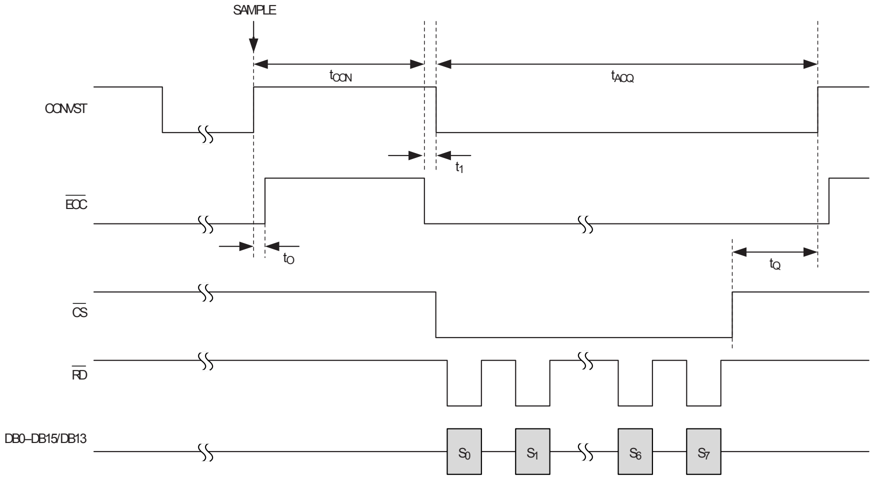


Figura 4.5: Modo de funcionamento do ADC MAX11056.

CONVST, *Conversion Start input*: Cada transição positiva deste sinal inicia uma nova conversão. Quando o sinal tem o valor “0” o ADC está em modo de aquisição.

$\overline{\textbf{EOC}}$, *Active-low End-of-Conversion output*: O sinal $\overline{EOC}$ vai a “0” quando a conversão do ADC terminar e vai a “1” quando a conversão é iniciada.

$\overline{\textbf{CS}}$, *Active-low Chip-Select input*: Colocar este sinal a “0” permite ler ou escrever nos registos de configuração do ADC.

$\overline{\textbf{RD}}$, *Active-low Read input*: Para ler 14 bits de dados do ADC é necessário colocar este sinal a “0”. Cada transição positiva do $\overline{RD}$ avança um canal e coloca os dados à saída.

$\overline{\textbf{WR}}$, *Active-low Write input*: Para escrever nos registos de configuração do ADC é preciso colocar este sinal a “0”. Na transição positiva os registos são configurados. Os registos de configuração são acedidos pelos pinos de dados, estes permitem escolher a referência do ADC (externa ou interna) e configurar o modo de funcionamento do ADC.

SHDN, *Shutdown input*: Se este pino permanecer a “1” o ADC entrará em modo *stand-by*.

Foi desenvolvido uma PCB, Figura 4.6, para fazer a ligação entre o conversor analógico-digital e a plataforma de desenvolvimento que será responsável pela leitura dos dados.

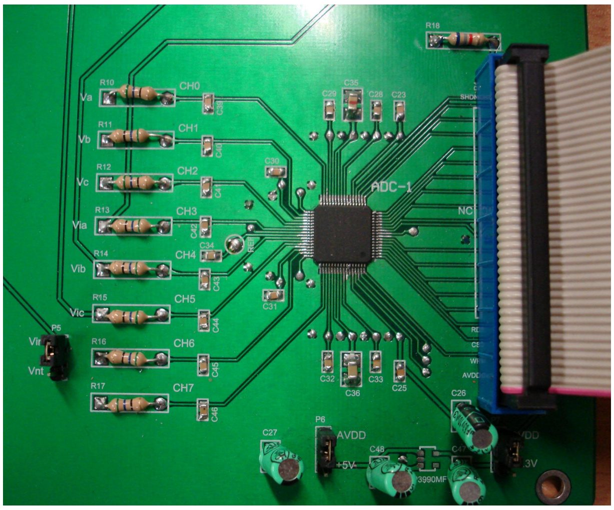


Figura 4.6: PCB com o ADC MAX11056.

4.4 Linux Embebido para *System-On-Chip*

A plataforma XUP Virtex-II Pro, descrita na Secção 3.1, permite desenvolver um SoC personalizado. Como já foi referido, esta caraterística permite desenvolver um sistema modelado especificamente para uma aplicação. Deste modo o sistema apenas usa os recursos essenciais para o funcionamento do sistema, é economizada energia e área de silício.

O Linux é o sistema operativo ideal para SoC customizáveis. Graças a modularidade do *kernel* do Linux é possível *on-the-fly* inserir ou remover *drivers* a partir de um sistema em execução. Por isso, fazer *upgrades* ao sistema é um procedimento muito menos "pesado".

Independentemente da arquitetura do CPU, o *core* precisa de pelo menos um barramento para aceder aos periféricos e memória externa. Nesta dissertação, a maioria dos periféricos são implementados na FPGA Virtex-II da Xilinx. A ferramenta EDK da Xilinx permite aos utilizadores desenvolver um SoC completo com uma ou mais unidades de processamento e periféricos. Outra vantagem desta ferramenta é que permite desenvolver módulos para aplicações específicas e incorpora-los no projeto FPGA.

Muitos sistemas de FPGAs não têm a capacidade de criar um sistema capaz de correr Linux porque não têm suporte a certos periféricos essenciais para o funcionamento do sistema operativo, como por exemplo suporte para sistema de ficheiros. A plataforma XUP Virtex-II Pro permite (através de um projeto EDK baseado em PowerPC) criar um SoC com todos os periféricos essenciais para suportar o *kernel* do Linux.

Na Figura 4.7 é possível observar que o ciclo de desenvolvimento do SoC e do Linux, é constituído por três blocos essenciais: O desenvolvimento do SoC através das ferramentas do EDK para produzir o *bitstream* e o *device-tree*; Construção do *kernel* Linux, para desenvolvimento da imagem do sistema operativo; E o Buildroot (Developers, 2013) para desenvolvimento do sistema de ficheiro para o *kernel*.

Esta secção apenas é focada no desenvolvimento do SoC com suporte ao Linux utilizando as ferramentas do EDK. O EDK Xilinx Platform Studio (XPS) fornece uma ferramenta auxiliar (o *Base System Builder*) que facilita o processo de desenvolvimento do SoC. O desenvolvimento da *Toolchain* e do sistema de ficheiros utilizando o Buildroot é abordado no Anexo A e B respetivamente. A criação da imagem do Linux tempo real é descrita no Anexo C.

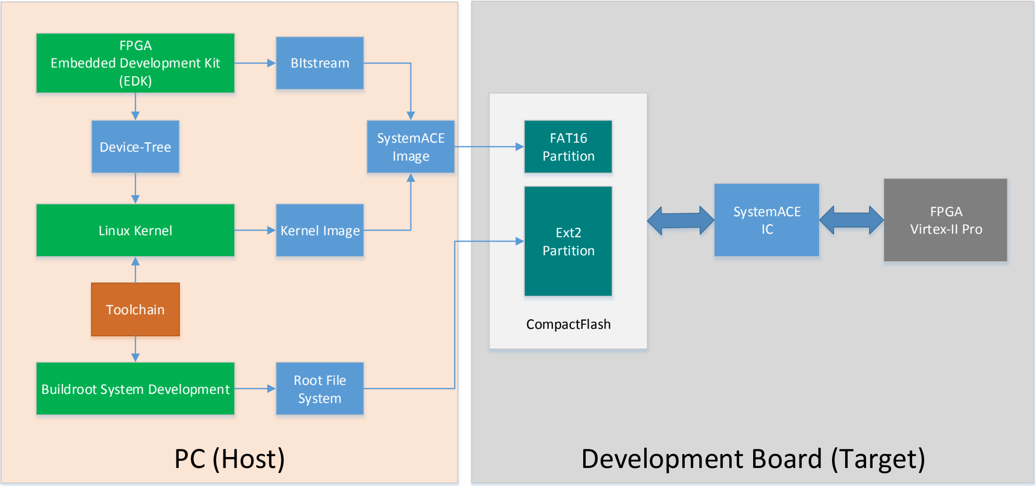
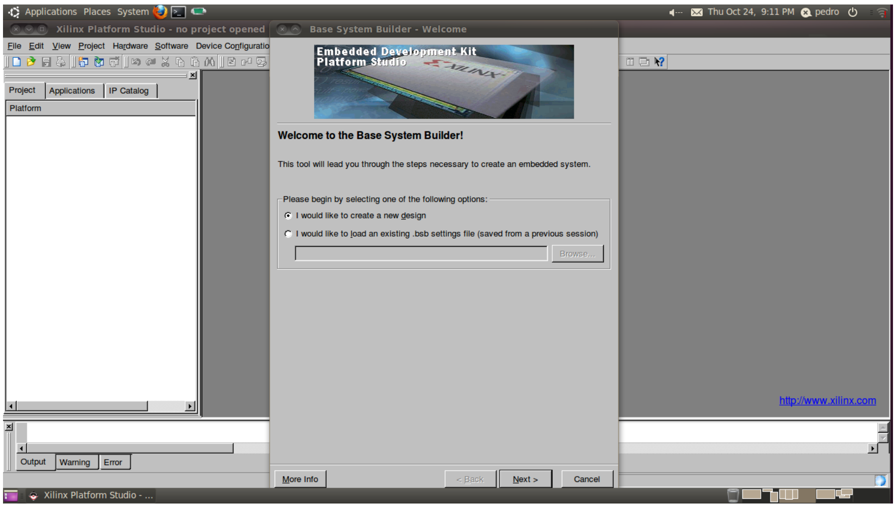


Figura 4.7: Ciclo de desenvolvimento do SoC e do Linux.

4.4.1 *Base System Builder*

O *Base System Builder* (BSB), representado na Figura 4.8, permite automatizar a maioria das tarefas de configuração do SoC. O BSB permite escolher vários periféricos (disponibilizados pela Xilinx) que são suportados pela plataforma de desenvolvimento, fazer a atribuição automática dos pinos da FPGA e criar um SoC pronto para configurar e correr na FPGA.



Welcome to the Base System Builder!

This tool will lead you through the steps necessary to create an embedded system.

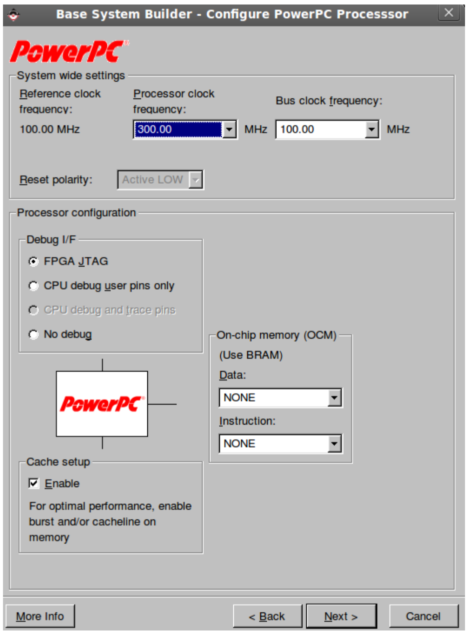
Please begin by selecting one of the following options:

● I would like to create a new design

○ I would like to load an existing .bsb settings file (saved from a previous session)

Figura 4.8: *Base System Builder*.

Durante o desenrolar do BSB é possível tomar várias decisões sobre as características e necessidades do sistema desejado. Os processadores propostos pela BSB são dependentes da plataforma FPGA em uso. Neste caso, o BSB apresenta duas opções de processadores, o *soft-core* MicroBlaze ou o *hard-core* PowerPC 405. Tendo em conta o tipo de aplicação que o sistema de aquisição terá, por razões de performance e discutidas na Secção 2.5.5 foi escolhido o processador PowerPC 405 (ver Figura 4.9).



System wide settings

Reference clock frequency: 100.00 MHz

Processor clock frequency: 300.00 MHz

Bus clock frequency: 100.00 MHz

Reset polarity: Active LOW

Processor configuration

Debug I/F

● FPGA JTAG

○ CPU debug user pins only

○ CPU debug and trace pins

○ No debug

On-chip memory (OCM)
(Use BRAM)

Data: NONE

Instruction: NONE

Cache setup

☑ Enable

For optimal performance, enable burst and/or cacheline on memory

Figura 4.9: Configuração do PowerPC 405.

Dentro do menu de configuração do PowerPC é possível selecionar algumas características essenciais para conseguir o melhor desempenho. A frequência do *clock* do CPU foi configurado para 300 MHz e o do barramento PLB para 100 MHz, estas são as frequências máximas permitidas pelo BSB. Além disso, foi incluída uma interface JTAG efeitos de para *debug* do processador e uma memória *cache* para melhorar o desempenho do processador.

Após a configuração do CPU, os próximos passos do BSB estão relacionados com os periféricos IP *cores* que irão estar acoplados como *slaves* ao barramento PLB. Os IP *cores* selecionados são fundamentais para o correto funcionamento do *kernel* Linux e são os seguintes:

- **xps_uartlite**: Este periférico é responsável pela comunicação série RS-232. Possibilita a receção das mensagens do *kernel* ao PC (*host*) e o envio de comandos do PC para o *kernel*;

- **xps_sysace**: Este periférico é responsável pela interface com a memória CompactFlash. O periférico tem a função de manter a relação física entre o processador e a memória CompactFlash onde se encontra o sistema de ficheiros do Linux e a imagem do sistema operativo;

- **xps_intc**: Este periférico é o controlador de interrupções. Os periféricos com capacidade de gerar interrupção são ligados ao controlador de interrupções que tem uma única saída que é ligado ao pino de interrupção do PowerPC 405;
- **mpmc**: Este periférico é o controlador de memória. O periférico tem a função de fazer a interface entre a memória externa SDRAM de 256 MB, o CPU e com os periféricos com necessidades DMA;

O SoC produzido pelo BSB está apto para suportar o *kernel* Linux. Porém nesta dissertação é pretendido desenvolver uma aplicação de monitorização gráfica e neste momento isso não é possível porque o SoC não tem qualquer tipo de controlador vídeo. A plataforma XUP Virtex-II Pro tem um controlador de vídeo (VGA) com capacidade de exibir 256.000 cores, e por isso foi adicionado o periférico de vídeo ***xps_tft***. Este periférico precisa de aceder a uma memória de vídeo (*framebuffer*) e por isso faz uso ao mecanismo de DMA. Portanto, para o correto funcionamento deste periférico é necessário adicionar um barramento PLB extra para fazer uma ligação *master* com o controlador de memória. Assim, o controlador de vídeo pode aceder ao *framebuffer* via *master*, e a ligação *slave* serve para indicar ao controlador de vídeo o endereço base na memória do *framebuffer*. O *driver* de vídeo do *kernel* é responsável por reservar espaço na memória para o *framebuffer* e indicar ao controlador de vídeo o endereço de memória base desta memória.

4.4.2 *Device-Tree*

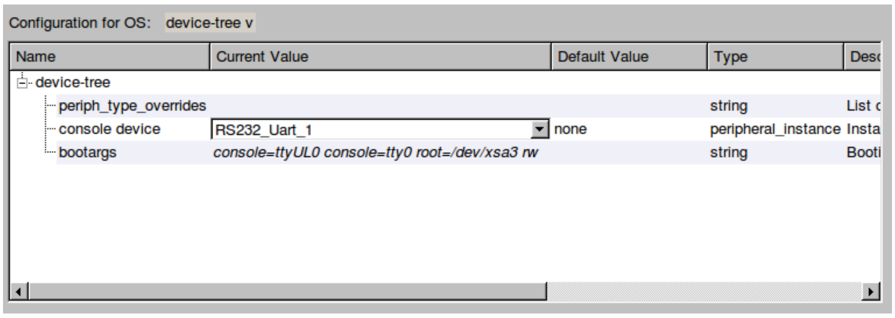
Depois de terminado a configuração do SoC é preciso gerar os ficheiros *board support package* (BSP) para serem utilizados na compilação do *kernel* Linux.

O *device-tree* é uma estrutura de dados para descrever o *hardware* de um sistema. Em vez de descrever todos os detalhes de cada periférico dentro do sistema operativo, é possível descrever todos os periféricos do SoC numa estrutura de dados que é posteriormente transmitida para o sistema operativo no momento do arranque. A estrutura de dados descreve várias características de *hardware* do SoC, e inclui:

- A arquitetura e o número de processadores;
- O tamanho e endereço base da memória principal (RAM);

- Barramentos do SoC;
- Ligações dos periféricos;
- Controlador de interrupções e as linhas IRQ dos periféricos;

A Xilinx Git Tree (SIMEK, 2013) fornece um mecanismo suportado pelo EDK para gerar uma estrutura de dados para descrever o SoC. Antes de gerar a estrutura de dados é necessário configurar alguns parâmetros essenciais para o arranque do *kernel*. Os argumentos de arranque estão indicados na Figura 4.10.



Configuration for OS: device-tree v

Name	Current Value	Default Value	Type	Desc
device-tree				
periph_type_overrides			string	List c
console device	RS232_Uart_1	none	peripheral_instance	Insta
bootargs	*console=ttyUL0 console=tty0 root=/dev/xsa3 rw*		string	Booti

Figura 4.10: Atribuição dos argumentos de arranque.

O argumento inserido no *bootargs* indica ao *kernel* os *drivers* de consola (*ttyUL0* e *tty0*) e a localização do sistema de ficheiros, que está localizado na partição três do cartão CompactFlash (*xsa3*).

Depois de definidos os argumentos é necessário gerar o *device-tree* do SoC através da ferramenta Xilinx *Generate Librarise and Drivers*. Essa estrutura de dados gerada, está representada num ficheiro *xilinx.dts*, contém todas as informações necessárias para o *kernel* durante a fase de compilação da imagem do sistema operativo de tempo real que é descrita no Anexo C.

4.5 *Hardware Device Driver*

Nesta secção é discutido a implementação de um *driver* em *hardware* para fazer a comunicação e controlo do conversor analógico-digital. O objetivo desta implementação (como foi discutido na Secção 2.7.3.2), é conseguir aumentar o determinismo e performance do sistema operativo, uma vez que a implementação em *software* requeria mais recursos e tempo de CPU.

Nesta secção pretende-se descrever a implementação em *hardware* do *adc_driver*, representada na Figura 4.11, que será responsável pela aquisição das *samples* do ADC MAX11056.

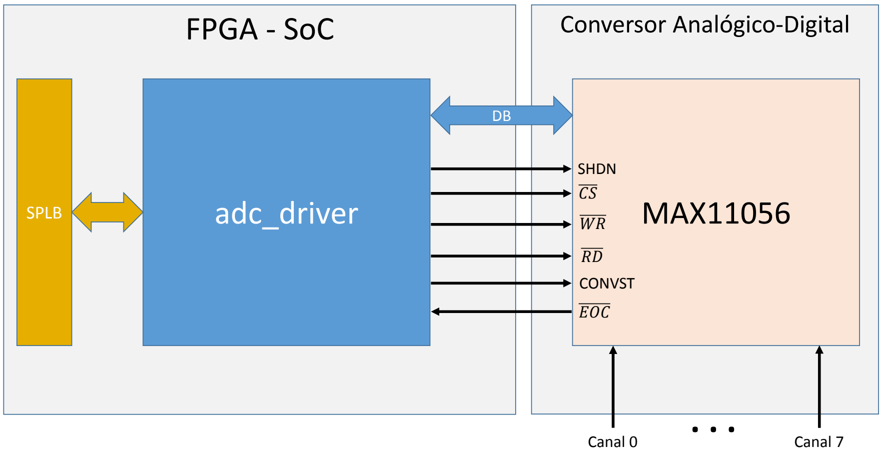


Figura 4.11: ADC *hardware driver*.

Este periférico, tal como os outros descritos na Secção 4.4.1, precisa de ser ligado como *slave* ao barramento PLB (SPLB) do SoC para poder receber comandos posteriormente do sistema operativo *kernel*. Mas, a implementação da comunicação entre o barramento SPLB e o periférico *adc_driver* é complexo e requer muito tempo. Porém as ferramentas da Xilinx permitem facilitar este processo.

Dentro do XPS existe uma ferramenta chamada *Create and Import Peripheral* (CIP) que permite criar periféricos básicos com:

- Interface com o barramento PLB ou FSL (*Fast Simpex Link*);
- Capacidade de gerar interrupção;
- Registos de configuração que podem ser acedidos por *software* (até 4096 registos);

Utilizando o CIP, foi criado um periférico básico, *adc_driver*, com capacidade de ligar ao barramento SPLB, gerar interrupção e com treze registos para leitura e escrita. Como será visto posteriormente, a capacidade de gerar interrupção é essencial para o periférico notificar ao sistema operativo após as leituras dos dados do ADC. Os registos são usados para controlar a lógica do *adc_driver* e para ler as amostras convertidas pelo ADC. Após a configuração do periférico, o CIP gera dois ficheiros: *user_logic.vhd* e o *adc_driver.vhd*. Estes são os ficheiros base para todo o desenvolvimento que se segue. O ficheiro *adc_driver.vhd* implementa toda a lógica de comunicação entre o barramento SPLB e o periférico, e o *user_logic.vhd* é o ficheiro que implementa a lógica de leitura e escrita dos registos, e a lógica para geração de interrupção.

4.5.1 Estrutura do *adc_driver*

Na Figura 4.12 está representada a estrutura interna do *hardware driver* do ADC. Este periférico é constituído por pequenos módulos com tarefas específicas e essenciais para o correto funcionamento do sistema de aquisição.

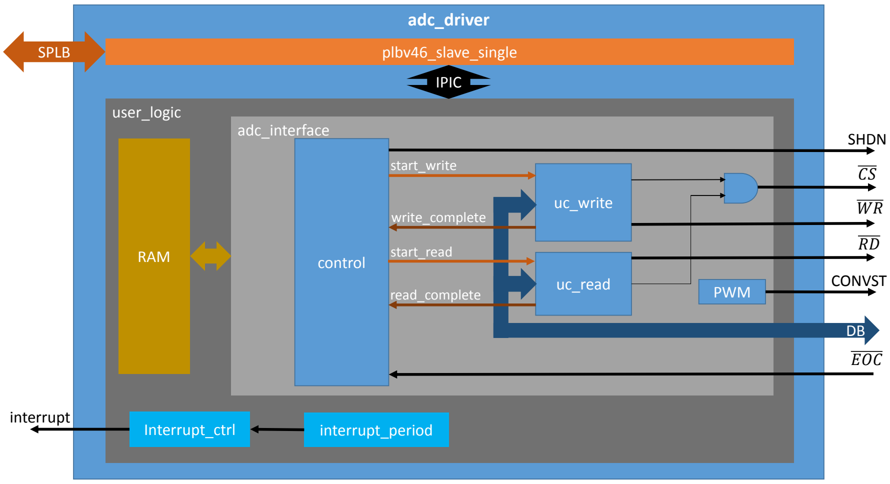
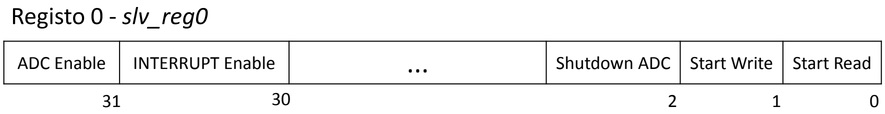


Figura 4.12: Estrutura interna do *adc_driver*.

O módulo *plbv46_slave_single*, criado pelo CIP, tem a responsabilidade de estabelecer a ponte entre o periférico *adc_driver* e o barramento SPLB, descodificação do endereço, geração de *timeout* e a ligação com o *user_logic* via IPIC (*Intellectual Property Interconnect*). A ligação com o *user_logic* tem como objetivo aceder aos registos de configuração do módulo que serão usados para controlar do periférico via *software*. Antes de passar às próximas secções, que descrevem o funcionamento dos restantes módulos, é necessário definir os comandos que ditarão o funcionamento do *adc_driver*.



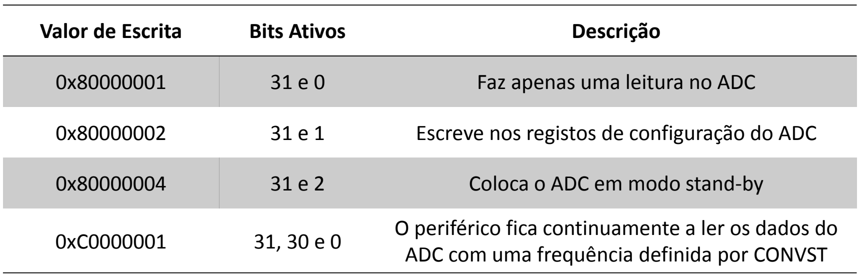
Registo 0 - *slv_reg0*

ADC Enable	INTERRUPT Enable	...	Shutdown ADC	Start Write	Start Read
31	30		2	1	0

Figura 4.13: Estrutura do registo *slv_reg0* do periférico.

Dos treze registo de configuração criados pelo CIP, foi definido que o registo "0" (*slv_reg0*, ver Figura 4.13) ditará o funcionamento do periférico *hardware* do ADC. O conjunto de comandos para *slv_reg0* estão descritos na Tabela 4.1.

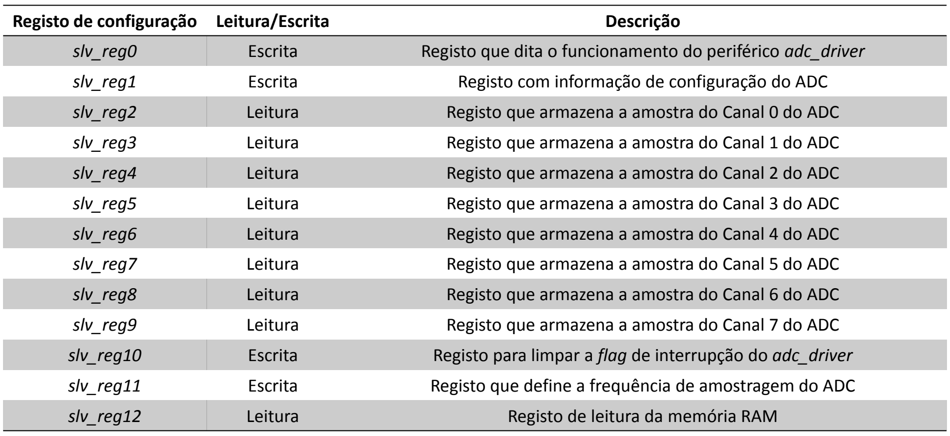
Tabela 4.1: Descrição dos comandos permitidos para *slv_reg0*.



Valor de Escrita	Bits Ativos	Descrição
0x80000001	31 e 0	Faz apenas uma leitura no ADC
0x80000002	31 e 1	Escreve nos registos de configuração do ADC
0x80000004	31 e 2	Coloca o ADC em modo stand-by
0xC0000001	31, 30 e 0	O periférico fica continuamente a ler os dados do ADC com uma frequência definida por CONVST

A descrição dos restantes registos criados pelo CIP está representada na Tabela 4.2. Todos os registos, quer sejam para leitura ou escrita, são acedidos por *software* através do *driver* do *kernel*.

Tabela 4.2: Descrição dos registos do periférico.



Registo de configuração	Leitura/Escrita	Descrição
slv_reg0	Escrita	Registo que dita o funcionamento do periférico *adc_driver*
slv_reg1	Escrita	Registo com informação de configuração do ADC
slv_reg2	Leitura	Registo que armazena a amostra do Canal 0 do ADC
slv_reg3	Leitura	Registo que armazena a amostra do Canal 1 do ADC
slv_reg4	Leitura	Registo que armazena a amostra do Canal 2 do ADC
slv_reg5	Leitura	Registo que armazena a amostra do Canal 3 do ADC
slv_reg6	Leitura	Registo que armazena a amostra do Canal 4 do ADC
slv_reg7	Leitura	Registo que armazena a amostra do Canal 5 do ADC
slv_reg8	Leitura	Registo que armazena a amostra do Canal 6 do ADC
slv_reg9	Leitura	Registo que armazena a amostra do Canal 7 do ADC
slv_reg10	Escrita	Registo para limpar a *flag* de interrupção do *adc_driver*
slv_reg11	Escrita	Registo que define a frequência de amostragem do ADC
slv_reg12	Leitura	Registo de leitura da memória RAM

Tendo em conta a descrição da Secção 4.3 sobre o funcionamento do ADC, foram implementados *SM Charts* para os módulos de controlo (*control*), de escrita (*uc_write*) e leitura (*uc_read*) de dados do ADC. Esses *SM Charts* foram posteriormente desenvolvidos numa linguagem HDL (VHDL) e efetuou-se a simulação dos mesmos utilizando a ferramenta da Xilinx ISim (ISE Simulator).

4.5.2 Módulo: *control*

O módulo *control* é o *core* do periférico. Este é responsável pela interpretação do comando guardado no registo *slv_reg0* e habilitar os sinais necessários de acordo com este. Na Figura 4.14 está representado o *SM Chart* deste módulo. O fluxo lógico de execução depende dos bits ativos do registo *slv_reg0* (Figura 4.13) e dependendo disso o módulo ativa os sinais necessários para colocar em funcionamento o pedido do utilizador.

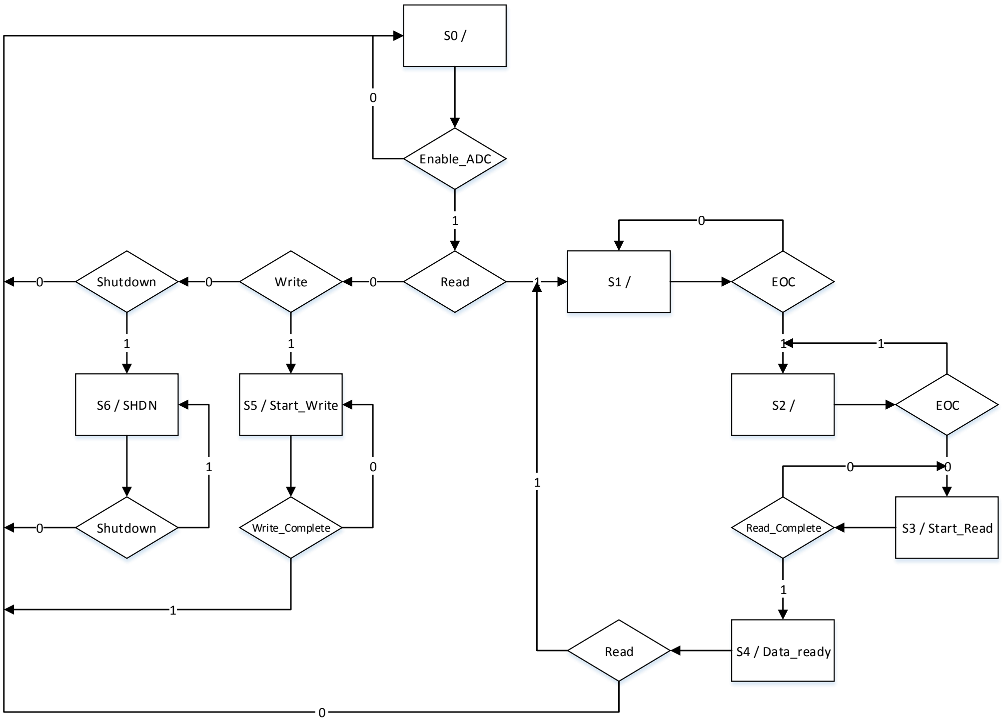


Figura 4.14: *SM Chart* para controlo do ADC.

Quando o sistema arranca (*slv_reg0* = 0x00000000) o módulo fica no estado *S0* até receber algum comando. Como será visto na Secção 4.6, os comandos são enviados pelo *driver* do *kernel* que por sua vez são definidos pelo utilizador do sistema.

Um exemplo em particular, se for pretendido configurar os registos do ADC, o utilizador terá de escrever 0x80000002 para o *slv_reg0*. Neste caso os bits *ADC Enable* e *Start Write* ficam com valor lógico "1", o módulo *control* entrará no estado *S5* do *SM Chart* e é iniciado o módulo *uc_write*. O módulo *control* sai do estado *S5* assim que receber o sinal *write_complete* (ver Figura 4.12) do módulo *uc_write*.

4.5.3 Módulo: *uc_write*

O *SM Chart* para configuração dos registos do ADC está representado na Figura 4.15.

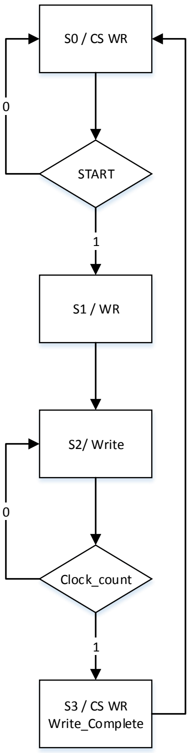


Figura 4.15: *SM Chart* para escrita nos registos do ADC.

No estado inicial *S0*, o módulo fica à espera do sinal *start* que é proveniente do módulo *control* discutido anteriormente. Quando o processo de escrita é iniciado, no estado *S1*, o sinal $\overline{CS}$ assume valor "0" e o módulo avança para o próximo estado. Em *S2* o sinal $\overline{WR}$ assume valor "0" e é feita a escrita para os registos de configuração. Após um tempo mínimo de escrita de 15 *num_clk_write* (15 ciclos de *clock*) os sinais $\overline{WR}$ e $\overline{CS}$ assumem valor "1", e é enviado o sinal *write_complete* que indica fim da escrita do módulo.

Na Figura 4.16 estão representados os resultados de simulação do módulo *ADC_Write*. O registo *write_reg* armazena a informação para configurar os registos do ADC, o seu valor é definido, por *software*, pelo utilizador através do registo "1" (*slv_reg1*). Quando é enviado o comando de escrita pelo utilizador, o módulo escreve o valor de *slv_reg1* nos registos do ADC, *adc_o* (*adc output*).

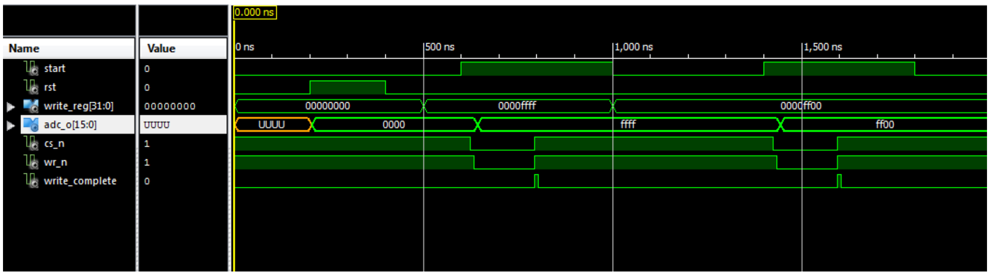


Figura 4.16: Resultado da simulação Xilinx ISim do módulo *uc_write*.

4.5.4 Módulo: *uc_read*, *PWM* e RAM

A implementação que é descrita nesta secção é idêntica ao diagrama temporal representado na Figura 4.5. O módulo *PWM* tem a função de fornecer o sinal CONVST que dita a frequência de amostragem do ADC. Essa frequência é definida pelo registo *slv_reg11*. Quando o ADC termina a conversão das amostras, este informa ao módulo *control* através do sinal $\overline{EOC}$ (ver Figura 4.12). Depois, o módulo *control* envia o sinal de *start_read* ao módulo *uc_read* e este lê as amostras do ADC. O *SM Chart* da Figura 4.17 mostra o funcionamento do módulo *uc_read*.

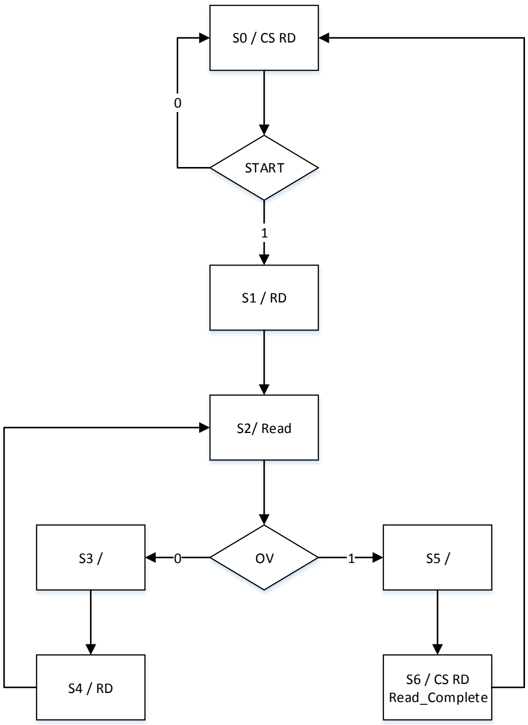


Figura 4.17: *SM Chart* para leitura de dados do ADC.

Após receber o sinal de *start*, o módulo *uc_read* lê os oito canais do ADC e atribui o valor da amostra de cada canal a um registo como indicado na Tabela 4.2. Quando o módulo *uc_read* chega à oitava leitura, o sinal *OV* (*overflow*) é sinalizado e termina a execução do módulo informando o módulo *control* através do sinal *read_complete*.

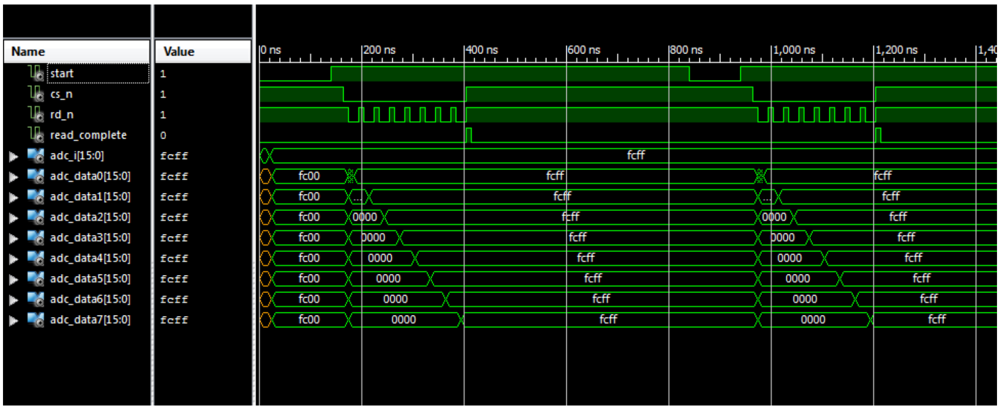


Figura 4.18: Resultado da simulação Xilinx ISim do módulo *uc_read*.

O resultado da simulação para este módulo esta representado na Figura 4.18. Na simulação é possível ver dois ciclos de leitura do *uc_read*. Comparando os resultados da simulação com o diagrama temporal da Figura 4.5, é possível observar que o resultado obtido é muito semelhante.

No fim de cada leitura dos canais do ADC, as amostras nos registos são armazenados numa RAM. Como será visto na Secção 4.6, esta RAM é posteriormente acedida pelo sistema operativo para ler os valores das amostras convertidos pelo ADC. A leitura é feita através do registo *slv_reg12*.

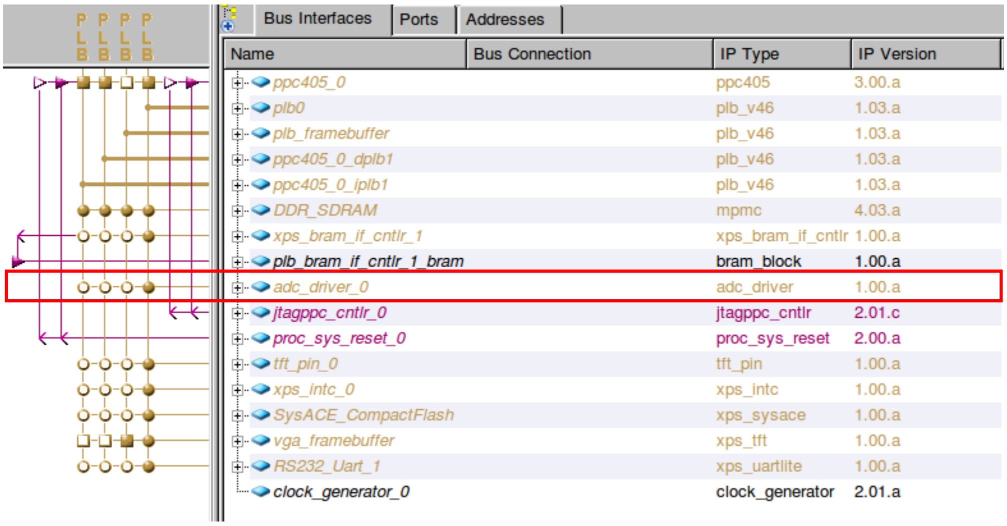
4.5.5 Módulo: *interrupt_ctrl* e *interrupt_period*

Estes módulos são responsáveis por gerar uma interrupção com frequência constante, para sinalizar o sistema operativo para a leitura de dados da RAM. Depois o sistema operativo fica responsável pela leitura dos dados e publica-los na aplicação no espaço utilizador.

A frequência da interrupção é definida pelo módulo *interrupt_period* e a implementação deste é igual ao módulo PWM. O módulo *interrupt_ctrl* é responsável pela gestão da interrupção e pela monitorizar o registo *slv_reg10* quando o sistema operativo atende a interrupção. O sinal de interrupção (*interrupt*) é ligado ao controlador de interrupções (*xps_intc*) que é responsável pela gestão das interrupções do SoC.

4.5.6 Ligação do *adc_driver* ao SoC

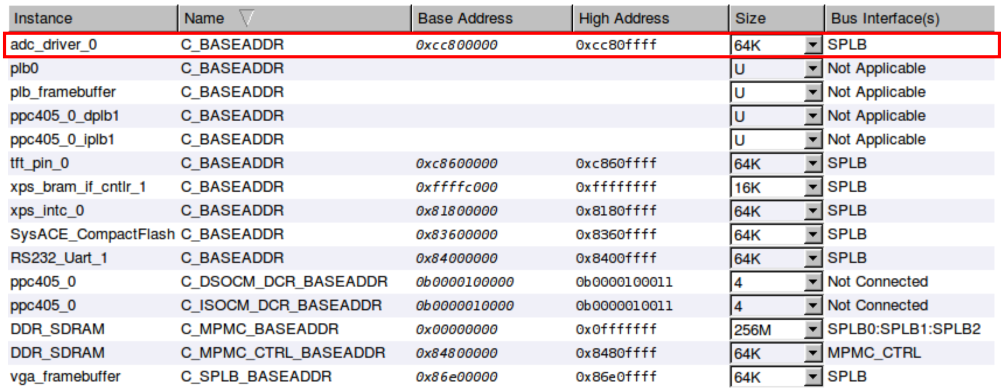
Terminado a implementação do *hardware driver* do conversor analógico-digital, é necessário incluí-lo no SoC criado na Secção 4.4. Como foi visto, a ferramenta CIP já fornece, no modelo base do periférico, uma interface para o barramento PLB e a integração do SoC torna-se assim bastante simples. Para isso, basta importar adicionar o periférico ao SoC e ligar a interface do SPLB do *adc_driver* ao barramento PLB onde estão ligados os restantes periféricos do sistema. Na Figura 4.19 está representado o SoC final utilizado para correr o sistema operativo Linux.



Name	Bus Connection	IP Type	IP Version
ppc405_0		ppc405	3.00.a
plb0		plb_v46	1.03.a
plb_framebuffer		plb_v46	1.03.a
ppc405_0_dplb1		plb_v46	1.03.a
ppc405_0_iplb1		plb_v46	1.03.a
DDR_SDRAM		mpmc	4.03.a
xps_bram_if_cntlr_1		xps_bram_if_cntlr	1.00.a
plb_bram_if_cntlr_1_bram		bram_block	1.00.a
adc_driver_0		adc_driver	1.00.a
jtagppc_cntlr_0		jtagppc_cntlr	2.01.c
proc_sys_reset_0		proc_sys_reset	2.00.a
tft_pin_0		tft_pin	1.00.a
xps_intc_0		xps_intc	1.00.a
SysACE_CompactFlash		xps_sysace	1.00.a
vga_framebuffer		xps_tft	1.00.a
RS232_Uart_1		xps_uartlite	1.00.a
clock_generator_0		clock_generator	2.01.a

Figura 4.19: Ligação do *adc_driver* ao SoC.

É possível ver que o SoC é constituído por quatro barramentos PLB. Os dois primeiros são utilizados apenas pelo PowerPC 405 para aceder à memória de dados e instruções na memória RAM. O terceiro barramento é utilizado pelo controlador de vídeo aceder ao *framebuffer* na memória RAM. Como foi visto na Secção 4.4.1, o controlador de vídeo precisa de duas interfaces PLB (*master* e *slave*). O último barramento PLB é onde os periféricos são ligados como *slaves* para receberem instruções do CPU. É neste barramento que o sistema operativo pode aceder aos registos dos periféricos para escrita ou leitura.



Instance	Name	Base Address	High Address	Size	Bus Interface(s)
adc_driver_0	C_BASEADDR	0xcc800000	0xcc80ffff	64K	SPLB
plb0	C_BASEADDR			U	Not Applicable
plb_framebuffer	C_BASEADDR			U	Not Applicable
ppc405_0_dplb1	C_BASEADDR			U	Not Applicable
ppc405_0_iplb1	C_BASEADDR			U	Not Applicable
tft_pin_0	C_BASEADDR	0xc8600000	0xc860ffff	64K	SPLB
xps_bram_if_cntlr_1	C_BASEADDR	0xffffc000	0xffffffff	16K	SPLB
xps_intc_0	C_BASEADDR	0x81800000	0x8180ffff	64K	SPLB
SysACE_CompactFlash	C_BASEADDR	0x83600000	0x8360ffff	64K	SPLB
RS232_Uart_1	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB
ppc405_0	C_DSOCM_DCR_BASEADDR	0b0000100000	0b0000100011	4	Not Connected
ppc405_0	C_ISOCM_DCR_BASEADDR	0b0000010000	0b0000010011	4	Not Connected
DDR_SDRAM	C_MPMC_BASEADDR	0x00000000	0x0fffffff	256M	SPLB0:SPLB1:SPLB2
DDR_SDRAM	C_MPMC_CTRL_BASEADDR	0x84800000	0x8480ffff	64K	MPMC_CTRL
vga_framebuffer	C_SPLB_BASEADDR	0x86e00000	0x86e0ffff	64K	SPLB

Figura 4.20: Endereço dos periféricos do SoC.

Como todos os periféricos estão ligados ao mesmo barramento, estes têm um endereço único utilizado para os distinguir quando acedidos por *software*. A Figura 4.20 mostra os endereços dos periféricos do SoC. Este endereço é atribuído automaticamente pelo XPS. Além disso é necessário voltar a gerar o *device-tree* para o *kernel* do Linux detetar o novo periférico no sistema.

4.6 *Software Device Driver*

A principal função de um *device driver* é controlar um determinado dispositivo ligado ao sistema computacional. Neste caso seria necessário um *driver* para o conversor analógico-digital que está ligado ao SoC. Porém, uma vez que a maioria das tarefas foram implementadas por *hardware*, o *driver* apenas terá de controlar o *adc_driver*, que está ligado ao SoC, que por sua vez controla o conversor analógico-digital. Assim sendo, o *driver* em causa, apenas fornece tratamento de interrupção onde serão lidos os valores da RAM do periférico *adc_driver*. Esses valores são colocados numa memória partilhada com uma aplicação de monitorização do espaço utilizador.

4.6.1 Inserção do *Driver* no *Kernel*

Quando o *driver* é inserido no *kernel* do Linux (através do comando *insmod*), este vai preparar todos os recursos necessários para tal. Isso é feito na seguinte ordem:

- O *kernel* aloca memória para armazenar o *driver* (esta memória é alocada com o *vmalloc*);

- Em seguida, o *kernel* copia o módulo para a região de memória alocada;
- Resolve as referências do *kernel* no módulo via tabela de símbolos do *kernel*;
- Por fim chama a função de inicialização do módulo;

A primeira coisa que a função de inicialização faz é registar o periférico no sistema. As principais rotinas de inicialização do *driver* estão representadas na Figura 4.21. Uma vez que a maioria das rotinas foram implementadas em *hardware*, o *driver*, do periférico *adc_driver*, torna-se muito simples. Este *driver* tem como principal função registar o serviço de atendimento a interrupção do periférico, alocar uma região de memória para partilha de dados com a aplicação no espaço utilizador e iniciar o periférico *adc_driver*.

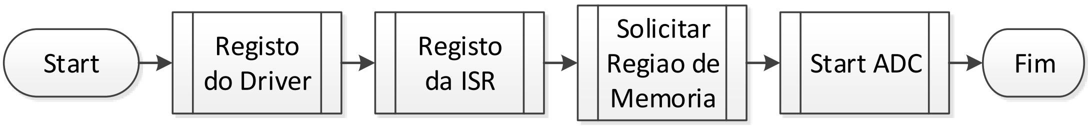


Figura 4.21: Principais rotinas de inicialização.

O serviço de atendimento à interrupção é acionado pelo periférico *adc_driver* a cada dois segundos. Aqui a ISR lê todos os valores da RAM do periférico *adc_driver* e coloca-os na memória para partilha. No fim, a ISR limpa a *flag* de interrupção do *adc_driver*, indicando o sucesso da leitura.

A frequência da interrupção a cada dois segundos é intencional. Como todo tratamento matemático e armazenamento das amostras foi implementado em *hardware*, não é necessário estar constantemente a interromper o sistema operativo para ler e tratar as amostras. Assim, apenas em cada dois segundos, os dados são atualizados na memória partilhada contribuindo para a redução do *overhead* do *kernel*.

A memória partilhada tem capacidade para armazenar dois ciclos quando a frequência de amostragem corresponde 20 kHz no *adc_driver*. Esta memória é muito importante para a aplicação de monitorização, pois esta permite a aplicação ler todas as amostras sem fazer chamadas ao sistema. Desta forma também se contribui para a redução do *overhead* e melhor resposta da aplicação de monitorização.

4.6.2 *Input/Output Control*

As chamadas de sistemas *ioctl* oferecem uma maneira de emitir comandos específicos aos dispositivos. Por isso, para controlar o periférico *adc_driver* foi implementado no *driver* comandos de controlo que são enviados pela aplicação através da função de *ioctl()*.

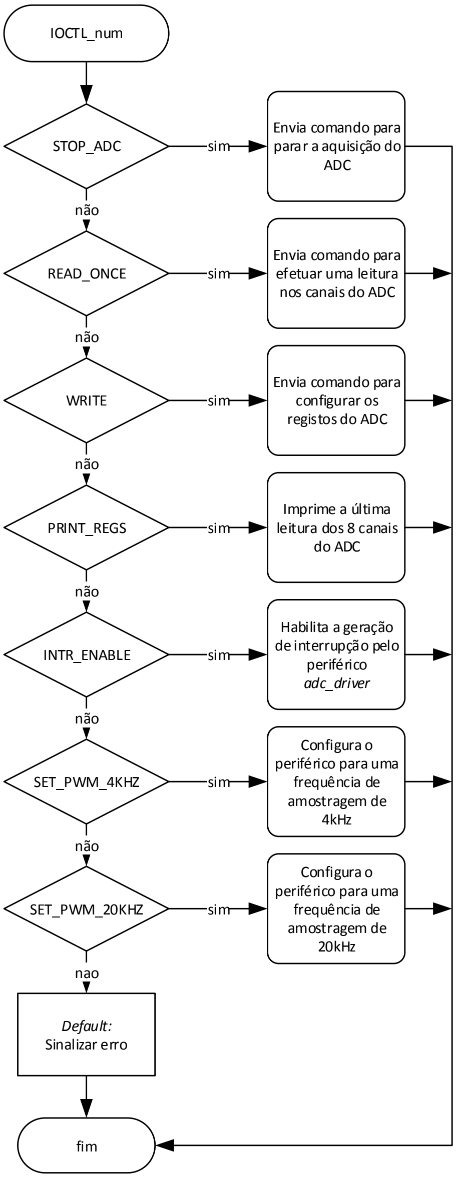


Figura 4.22: Fluxograma da função *kernel adc_ioctl*.

A função que implementa as chamadas ao sistema designa-se por *adc_ioctl*, e o seu fluxograma está apresentado na Figura 4.22. Os comandos desenvolvidos para este dispositivo são: início e paragem do sistema de aquisição; parametrização da frequência de amostragem; configuração dos registos do ADC; leitura da última amostra convertida pelo ADC; e habilitação da interrupção. Este comandos fazem leituras/escritas para os registo indicados na Tabela 4.2. Esta função permite à aplicação de monitorização o total controlo do periférico *adc_driver*.

4.7 Aplicação de Monitorização

A aplicação de monitorização foi desenvolvida em Qt (Nokia, 2013) e corre diretamente na plataforma de desenvolvimento. O Qt é uma *framework* para desenvolvimento de interfaces gráficas em C++. Para correr as aplicações do Qt neste sistema, foi necessário fazer o *cross-compile* das bibliotecas do Qt e adicioná-las no sistema de ficheiros Linux. Depois a aplicação é desenvolvida utilizando as ferramentas do Qt no *host* e com recurso a *toolchain* compila-se a aplicação para correr na plataforma de desenvolvimento (*target*).

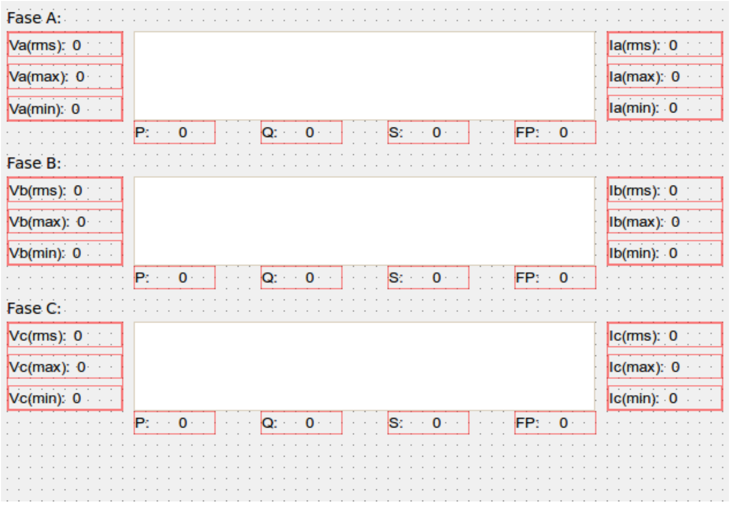


Figura 4.23: Aplicação de monitorização.

Na Figura 4.23 está representado o *layout* da aplicação de monitorização. A aplicação terá capacidade de mostrar todos os parâmetros essenciais do sistema elétrico: valores eficazes, mínimos e máximos da tensão e corrente; fatores de potência; valores de potência; e formas de onda de cada fase. Os valores são lidos através da memória partilhada pelo *kernel* e são atualizados a cada dois segundos, uma vez que corresponde à frequência com que os dados são atualizados pelo *driver* descrito na Secção 4.6.1.

4.8 Conclusões

O sistema de aquisição está dividido em quatro partes: o acondicionamento de sinal, com os elementos essenciais para fazer interface entre os sinais da rede elétrica e conversor analógico-digital; o *front-end* analógico, para medição dos seis sinais da rede elétrica (3 tensões e 3 correntes) e converter esses sinais contínuos no tempo em sinais digitais discretos no tempo; a plataforma de desenvolvimento com o SoC customizado, onde estão inseridos todos os periféricos mais o *adc_driver* desenvolvido; por fim, a aplicação de monitorização, desenvolvida em Qt que é responsável por mostrar todos os parâmetros essenciais do sistema elétrico.

Capítulo 5

Cenários de Aplicação

Este capítulo pretende demonstrar o funcionamento do sistema de aquisição. Numa primeira fase o sistema foi testado num sistema trifásico gerado por um microcontrolador. Nesta fase pretende-se depurar todo o sistema antes de passar para o sistema elétrico trifásico. Posteriormente é descrita a implementação da teoria p-q em *hardware* e demonstrado os resultados de teste.

5.1 Teste de Depuração

Antes de ligar o sistema de aquisição à rede elétrica, é necessário depurar todo o sistema de forma a determinar que este funciona bem. Para isso foi utilizado o microcontrolador ATmega328, representado na Figura 5.1, para gerar, através da modulação por PWM, três formas de onda sinusoidais idênticas ao sistema trifásico representadas na Figura 5.2.

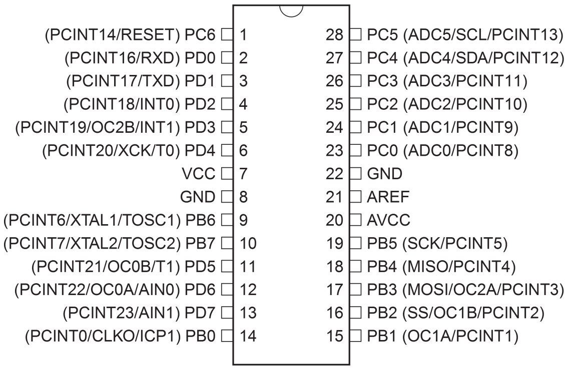


Figura 5.1: *Pinout* do ATmega328.

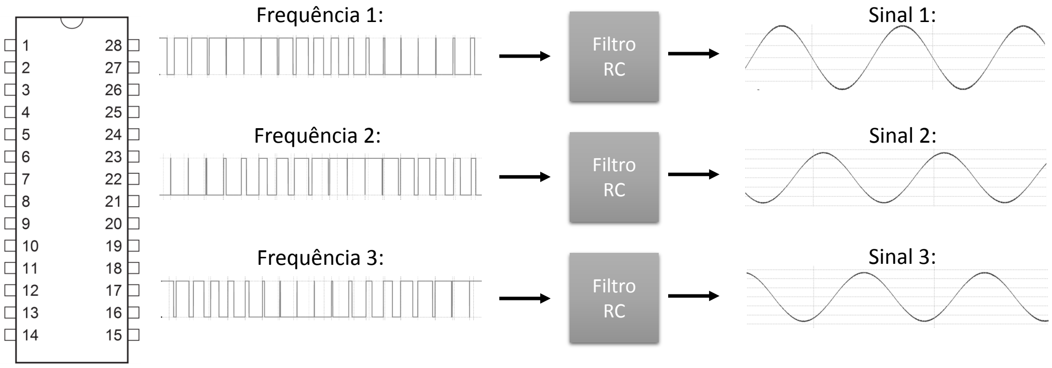


Figura 5.2: Formas de onda geradas pelo microcontrolador através da modulação por PWM.

As formas de onda geradas foram ligadas ao conversor analógico-digital para simular a saída dos transdutores de tensão e corrente. Essas ligações estão representadas na Figura 5.3.

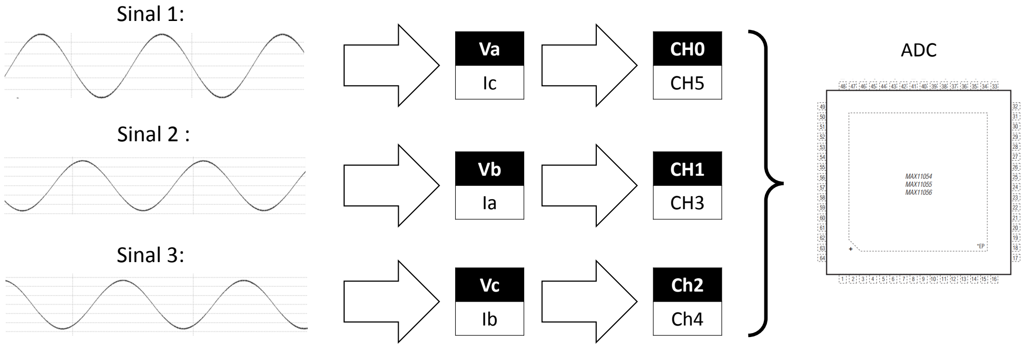


Figura 5.3: Ligações dos sinais ao ADC.

As ligações ao conversor analógico-digital foram feitas de forma a obter um sistema indutivo. Assim, o “Sinal 1” representa a tensão da fase A e a corrente da fase C, que corresponde respetivamente ao canal “0” e “5” do ADC. O “Sinal 2” representa a tensão da fase B e a corrente da fase A, que corresponde respetivamente ao canal “1” e “3” do ADC. Por fim, o “Sinal 3” representa a tensão da fase C e a corrente da fase B, que corresponde respetivamente ao canal “2” e “4” do ADC. Desta forma, é possível determinar se todas as componentes da aplicação de monitorização estão a funcionar corretamente, como por exemplo o cálculo da potência reativa (“Q”) e do fator de potência (“FP”).

Na Figura 5.4 é apresentada uma fotografia do sistema de aquisição utilizado para o teste de depuração. Como já foi referido, não são utilizados quaisquer

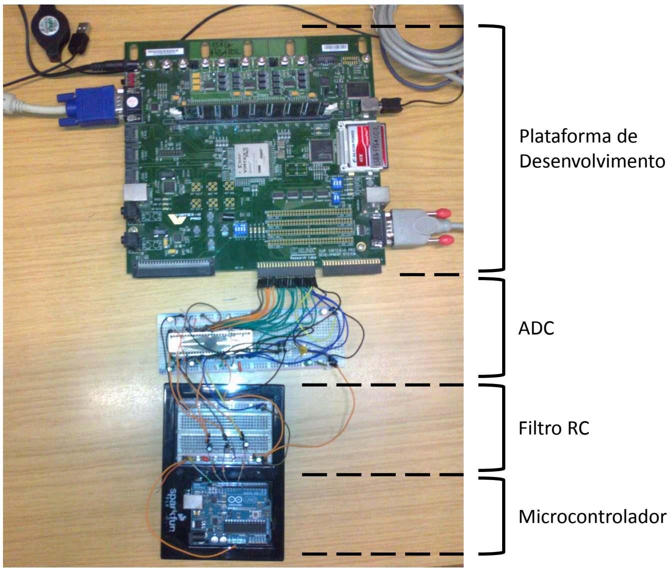
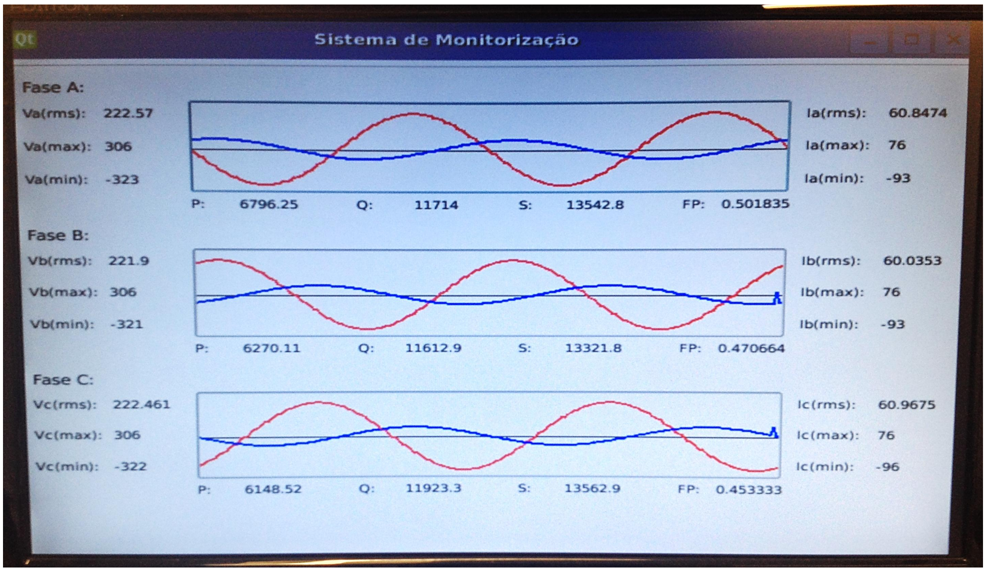


Figura 5.4: Ligação das componentes do sistema de aquisição para depuração.

transdutores uma vez que o microcontrolador simula a saída de tensão destes. A plataforma de desenvolvimento contém o SoC customizado com o *driver* em *hardware* responsável pela aquisição dos sinais do ADC. Assim, a plataforma é ligada e dá-se início ao arranque do sistema operativo Linux. Após o arranque do Linux inicia-se a aplicação de monitorização e o resultado está representado na Figura 5.5.



Fase A:
Va(rms): 222.57
Va(max): 306
Va(min): -323
Ia(rms): 60.8474
Ia(max): 76
Ia(min): -93
P: 6796.25 Q: 11714 S: 13542.8 FP: 0.501835

Fase B:
Vb(rms): 221.9
Vb(max): 306
Vb(min): -321
Ib(rms): 60.0353
Ib(max): 76
Ib(min): -93
P: 6270.11 Q: 11612.9 S: 13321.8 FP: 0.470664

Fase C:
Vc(rms): 222.461
Vc(max): 306
Vc(min): -322
Ic(rms): 60.9675
Ic(max): 76
Ic(min): -96
P: 6148.52 Q: 11923.3 S: 13562.9 FP: 0.453333

Figura 5.5: Aplicação de monitorização em teste de depuração.

Como já foi referido, os valores na aplicação são atualizados a cada 2 segundos, essa temporização é dado pela interrupção gerada pelo *hardware driver adc_driver* descrito na Secção 4.5. Após a receção da interrupção pelo *kernel* este notifica a aplicação que existem mais dados para serem consumidos. Depois de consumir os dados, a aplicação calcula algumas componentes como valor eficaz, máximos, mínimos e potências das três fases e coloca-os na aplicação.

O filtro RC, utilizado para obter as formas de onda sinusoidais, atenua um pouco as formas de onda, fazendo com que estas deixem de ser simétricas. Esse fenómeno pode ser visto nas formas de onda na aplicação de monitorização, que apresentam um ligeiro desvio e pode ser confirmado pelos valores máximos e mínimos. A solução seria colocar um filtro com uma frequência de corte maior, mas mesmo assim o desvio não seria eliminado na totalidade.

5.2 Teoria p-q em *Hardware*

5.2.1 Teoria p-q

A Teoria p-q, é conhecida no mundo da eletrónica como a Teoria da Potencia Ativa e Reativa Instantânea (Akagi et al., 1983). Antes de entrar na teoria de potência ativa e reativa instantânea propriamente dita é importante caracterizar precisamente um sistema trifásico genérico. A Figura 5.6 mostra a forma de onda de tensão de dois sistemas trifásicos. No caso geral, as correntes podem estar também como qualquer uma das opções mostradas nesta figura.

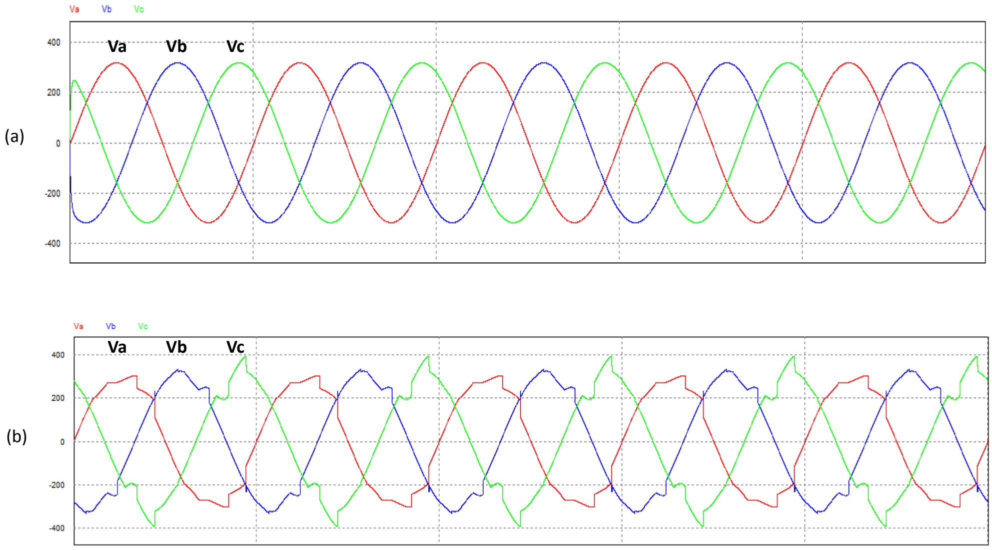


Figura 5.6: Formas de onda de tensão de um sistema trifásico: (a) sistema equilibrado; (b) sistema distorcido.

Numa situação com carga ideal e equilibrada, as formas de onda da tensão e corrente teriam o aspeto das formas de onda da Figura 5.6(a) com fator de potência unitário, ou seja, tensão e corrente em fase. Porém a maioria das vezes isso não acontece. A grande maioria das cargas apresentam consumos de corrente não linear (com harmónicos) o que provoca quedas de tensão nas linhas, e consequentemente distorção das formas de onda de tensão como os da Figura 5.6(b). A distorção das formas de onda de tensão trazem consequências como sobreaquecimento dos componentes, avarias nos equipamentos, consumos errados de energia e provocam com que cargas lineares consumam correntes com harmónicos. Um exemplo está ilustrado na Figura 5.7

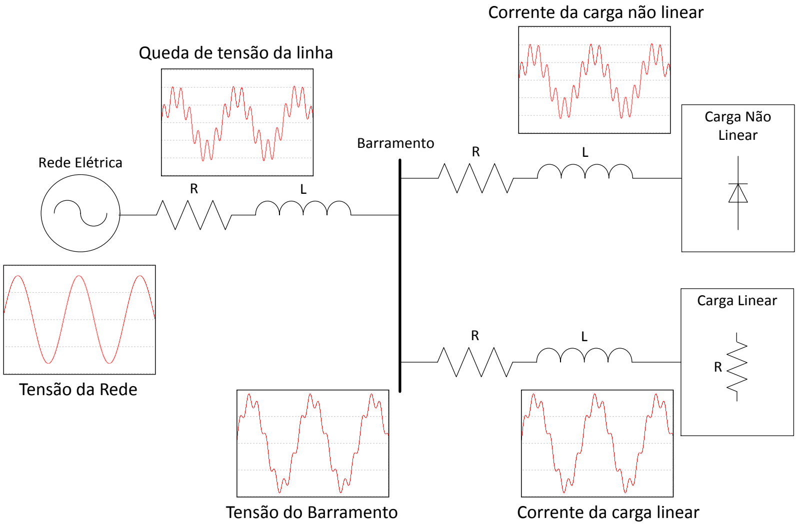


Figura 5.7: Sistema elétrico com uma carga linear e uma não linear.

Akagi et al. (1983), quando propuseram a teoria de potência instantânea, tinham como objetivo o controlo dos filtros ativos de potência. Um exemplo desses filtros é o Filtro Ativo Paralelo (FAP). Este dispositivo, o FAP, garante que a corrente na linha, a montante do seu ponto de ligação, está sempre em fase com a tensão, com a forma de onda sinusoidal, sem harmónicos. Esta condição impõe do lado da fonte um consumo de potência constante, desde que as tensões do sistema sejam sinusoidais e equilibradas. A Figura 5.8 mostra o mesmo cenário da Figura 5.7 mas com a introdução do Filtro Ativo Paralelo.

A corrente produzida pelo filtro, corrente de compensação, cancela os harmónicos existentes a montante do seu ponto de ligação. Esta compensação permite melhorar a forma de onda da tensão no barramento e consequentemente melhora a forma de onda de corrente das cargas lineares. Através da teoria p-q, é possível calcular

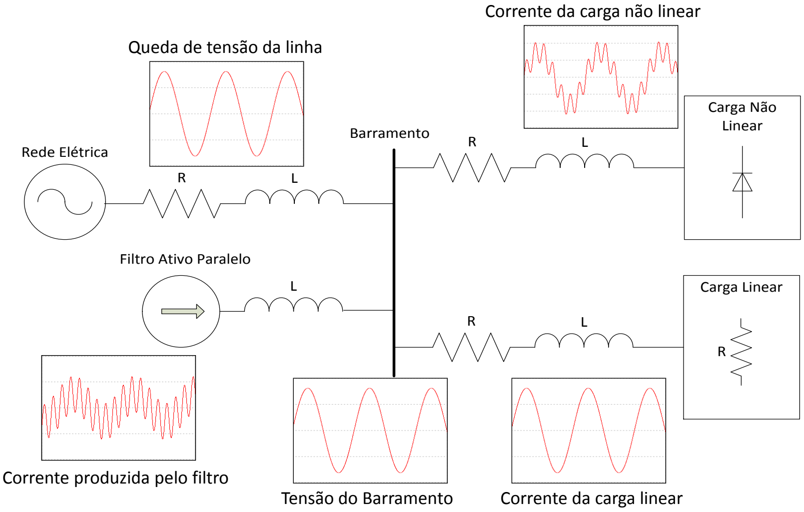


Figura 5.8: Sistema eléctrico na presença do Filtro Activo Paralelo.

as componentes indesejáveis num sistema elétrico, e baseado nestas calcular os valores das respetivas correntes de compensação.

5.2.2 Implementação da Teoria p-q

Este tipo de sistemas, discutidos anteriormente, são considerados sistemas de tempo real e possuem requisitos incompatíveis com a grande maioria dos sistemas operativos, sendo muitas vezes, implementados com recurso ao DSP (*Digital Signal Processor*).

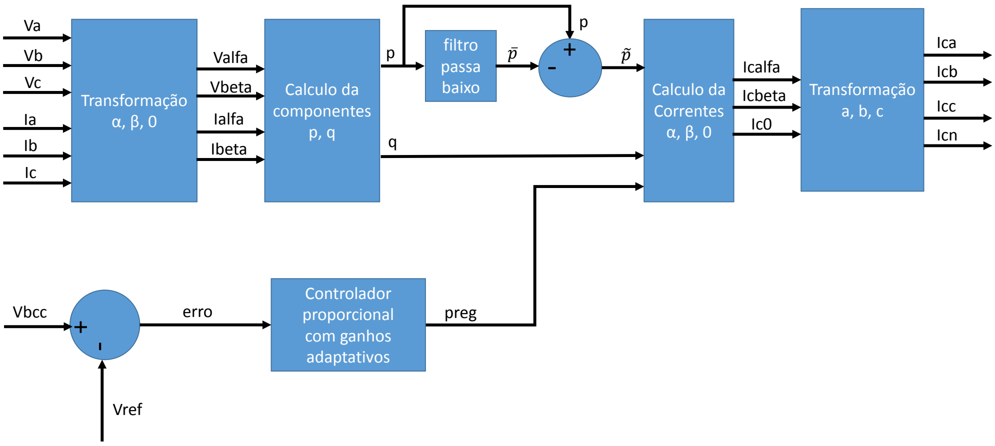


Figura 5.9: Diagrama de blocos do algoritmo de controlo implementado na FPGA.

No decurso desta dissertação a teoria p-q foi implementada em *hardware*, o que soluciona este problema. O módulo está junto ao periférico *adc_driver* do SoC. A utilização da tecnologia FPGA permite reduzir o *overhead* do sistema operativo ao implementar a teoria p-q em *hardware*. Desta forma, e aproveitando o paralelismo do *hardware*, a FPGA excede o poder do DSP ao realizar mais instruções por ciclo de *clock*. O algoritmo está representado na Figura 5.9 e pode ser consultado com mais detalhe no Anexo D.

Foram adicionadas outras funcionalidades para aumentar a performance da teoria p-q: o *pipelining*; controlo adaptativo para os ganhos do barramento de tensão contínua (barramento CC); e a modulação por PWM das correntes de compensação para geração dos sinais de *gate drive* do inversor.

A frequência com que as correntes de compensação são calculadas é dada pela frequência de amostragem do conversor analógico-digital. Após recolha da amostra pelo *adc_driver* as correntes de compensação são calculadas em 80 ciclos de *clock*, que corresponde a 800 nanosegundos, que para este sistema já é bastante bom. Também foram implementadas técnicas de *pipelining* que permitem reduzir o número de ciclos de *clock* necessários até 40 ciclos mas requerem uma taxa de amostragem superior a 2 MHz, que está limitado pelo próprio conversor analógico-digital. Não é possível reduzir mais o tempo de cálculo das componentes de compensação, porque existe uma operação de divisão que requer pelo menos 38 ciclos *clock*, e o tempo mínimo de conversão do ADC é de de 3μs. Logo, a teoria p-q implementada no *adc_driver* fica mais tempo à espera de novas amostras do que a fazer cálculos das componentes de compensação.

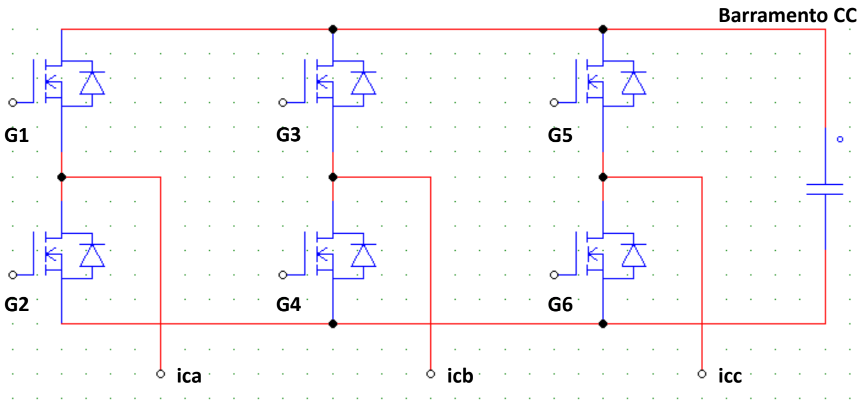


Figura 5.10: Inversor trifásico mais barramento de tensão contínua.

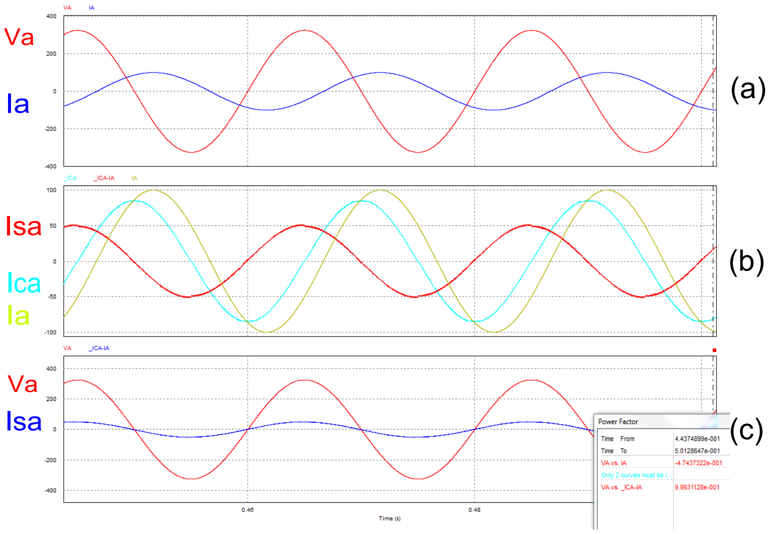
O sistema de controlo do FAP, é também responsável por manter constante a tensão no barramento CC do inversor, representado na Figura 5.10. O barramento CC é responsável pelo armazenamento de energia que Filtro usa para injetar a corrente de compensação na rede elétrica. Dependendo do tipo de carga ligado ao sistema elétrico, esta pode influenciar mais, ou menos, as trocas de energia entre a rede e o barramento CC. Para garantir níveis de tensão aceitáveis, foi implementado um controlo adaptativo que ajusta o valor dos ganhos do controlo do barramento CC. Desta forma não é necessário estar constantemente a alterar os valores dos ganhos sempre que a carga no sistema elétrico mudar.

O inversor representado na Figura 5.10 é usado para injetar a corrente de compensação na instalação elétrica. Para tal, é preciso recorrer a técnicas de modulação PWM para gerar os sinais que vão atuar no *gate drive* do inversor representado na Figura 5.10 pelos sinais *G1* ao *G6*. As técnicas de modulação PWM e os sinais para o *gate drive* são implementados na FPGA, permitindo reduzir o custo global do sistema.

5.2.3 Teste da Teoria p-q

Para verificar o correto funcionamento da teoria p-q recorreu-se ao teste descrito na Secção 5.1. Como as correntes estavam atrasadas em relação à tensão, o sistema é ideal para testar a teoria p-q no sistema de aquisição.

Antes de testar efetuou-se uma simulação utilizando a ferramenta de simulação PSim (Powersim Inc., 2013) para verificar os resultados esperados.



Time From 4.4374899e-001
Time To 5.0128647e-001
VA vs. IA -4.7437322e-001
Only 2 curves must be i...
VA vs. _ICA-IA 9.9931128e-001

Figura 5.11: Resultados de simulação do PSim.

Na Figura 5.11 está representado o resultado da simulação apenas para a fase A. A Figura 5.11(a) representa a tensão e a corrente consumida pela carga da fase A. Esta forma de onda é semelhante ao resultado da Figura 5.5 durante o teste de depuração. Na Figura 5.11(b) está representado a corrente de compensação gerada pelo filtro (Ica), a corrente consumida pela carga (Ia) e a corrente do sistema elétrico após compensação (Isa). Para realçar, a Figura 5.11(c) mostra a corrente e tensão em fase após compensação.

Após verificar, através das ferramentas de simulação, os resultados esperados o sistema de aquisição é ligado. Tal como na simulação, e para manter a simplicidade, apenas será demonstrado uma das fases do sistema elétrico. Na Figura 5.12 estão representados os resultados do sistema de aquisição. Como se pode comprovar os resultados são muito semelhantes aos resultados de simulação da Figura 5.11(b).

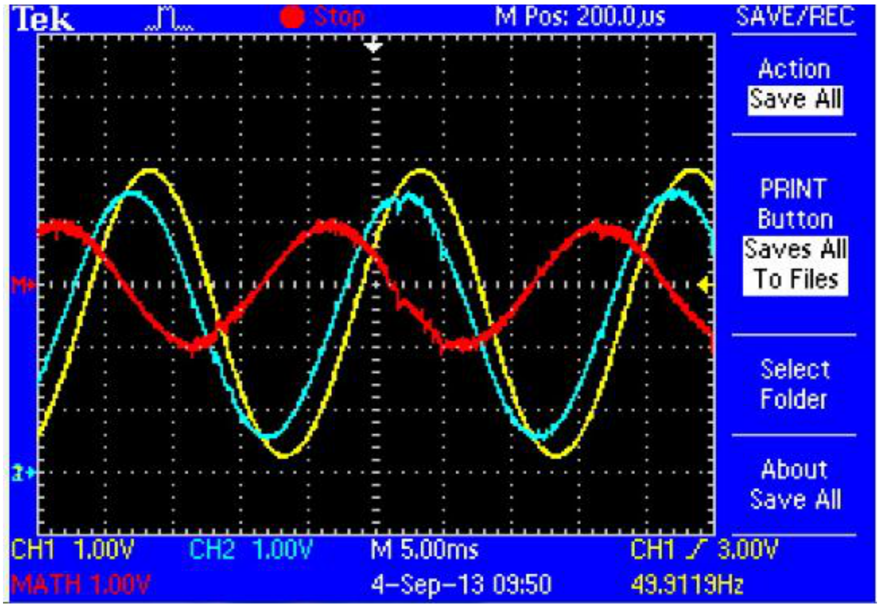


Figura 5.12: Resultado da compensação no osciloscópio.

O sinal proveniente da FPGA, destinado ao *gate drive* do inversor para sintetização da corrente de compensação da fase A (Ica), está representado na Figura 5.12 de cor azul. O sinal amarelo representa a corrente consumida pela carga (Ia). Uma vez que o sistema de aquisição não foi ligado ao sistema elétrico, foi necessário recorrer às funcionalidades aritméticas do osciloscópio para determinar a corrente após compensação (Isa). Essa corrente está representada a vermelho e corresponde à subtração de Ica com Ia.

O mesmo foi verificado para a fase B e C, e comprovou-se o sucesso da teoria p-q na FPGA. O último passo consiste em verificar o mesmo resultado na aplicação de monitorização. Assim, na Figura 5.13 estão representados os resultados da aplicação gráfica.

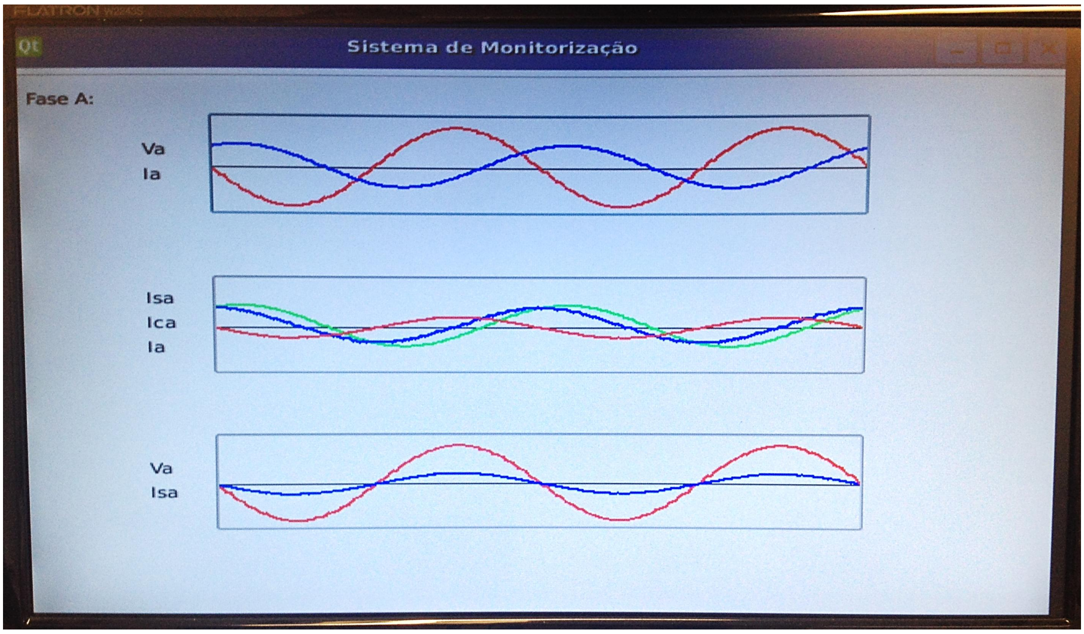


Figura 5.13: Aplicação de monitorização em teste da teoria p-q.

Verifica-se que os resultados obtidos pela aplicação de monitorização da Figura 5.13 são muito semelhantes aos resultados da simulação da Figura 5.11.

5.3 Conclusões

Inicialmente foi feito um teste de depuração para verificar que o sistema está a funcionar bem. Usou-se um microcontrolador para gerar 3 sinais para simular as saídas dos transdutores de tensão. Propositadamente, as ligações ao conversor analógico-digital foram feitas de tal forma a ter um sistema indutivo, com intuito de verificar se todas as componentes da aplicação estavam a funcionar corretamente, tal foi verificado. Além disso, e para enriquecer o trabalho, foi implementada a teoria p-q em *hardware* junto ao periférico *adc_driver*. Esta implementação exigiu um grande esforço de engenharia, uma vez que a teoria p-q foi implementada com *pipelining*, controlo adaptativo para os ganhos do barramento de tensão e a modulação por PWM das correntes de compensação para o *gate drive* do inversor. No fim testou-se o sistema, e verificou-se que os resultados são muito bons, através da comparação dos resultados de simulação e os práticos.

Capítulo 6

Discussão e Conclusões

O projeto juntou duas áreas distintas, Eletrónica de Potência e os Sistemas Embebidos, tornando-o muito desafiante. Durante o desenvolvimento deste trabalho foram adquiridos vários conhecimentos, desde a customização do SoC à compensação dos harmónicos de corrente na rede elétrica, através da implementação de raiz da Teoria p-q em linguagem de descrição de *hardware*. Além disso também foram obtidos conhecimentos de desenvolvimento e *patching* do *kernel* Linux, desenvolvimento de *device drivers*, implementação de mecanismos de partilha de memória entre o *kernel-space* e *userspace*. Por fim, também foram obtidos conhecimentos de desenvolvimento de aplicações gráficas em ambiente Qt.

A adoção da tecnologia FPGA continua a aumentar à medida que esta e as ferramentas de alto nível se tornam mais acessíveis. A possibilidade da customização do SoC abre um vasto leque de oportunidades e soluções para aplicações como a que foi implementada neste trabalho. O desenvolvimento de uma solução para um ambiente do tipo *hard*, não chega só compilar o *kernel* com os *patches*, é preciso lidar diretamente com as questões de baixo nível como a interface com *hardware* e gestão de memória. A integração destas tecnologias numa aplicação com requisitos temporais traz uma mais valia. A utilização da tecnologia FPGA associado ao poder de um sistema operativo de tempo real permite aplicar este sistema a uma maior variabilidade de aplicações.

6.1 Trabalho desenvolvido

Esta tese descreveu as considerações e decisões tomadas na implementação de um sistema de aquisição de sinais em tempo real baseado em Linux. O sistema de aquisição tem capacidade de atingir frequências de amostragem superiores a 2MHz, que está limitado pelo conversor analógico-digital. O sistema de aquisição de sinais tem capacidade de fornecer à aplicação de monitorização 400 amostras de oito canais simultâneos por ciclo de rede, conferindo a este sistema uma capacidade de compensar dinamicamente harmónicos de corrente, corrigir fator de potência da instalação para a unidade e controlar de forma estável as tensões do barramento de tensão contínua do inversor através do algoritmo do controlo adaptativo dos ganhos.

Grande parte do sucesso deste trabalho foi devido à redução do *overhead* do *kernel*, através da implementação em *hardware* dos *drivers* críticos. Desta forma todas as operações matemáticas e sinais de controlo foram deixada a cargo do *hardware*. Comprovou-se que o trabalho desenvolvido funciona e está apto para ser integrado num subsistema de um sistema maior como por exemplo num Filtro Ativo Paralelo, realizando assim aquele que era o principal objetivo desta dissertação.

6.2 Trabalho Futuro

Nos objetivos iniciais desta dissertação fazia parte a implementação do FAP. Contudo devido a exigência e complexidade do projeto tal não foi conseguido. Durante o desenvolvimento do trabalho foram desenvolvidas duas PCBs com objetivo de ligar o sistema de aquisição à rede elétrica. Essas PCBs estão representadas na Figura 6.1 e 6.2.

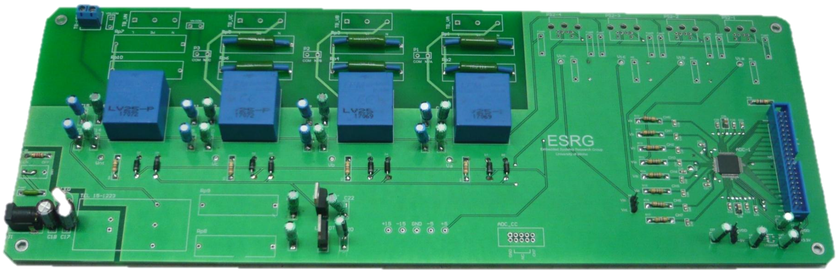


Figura 6.1: PCB para leituras dos transdutores de tensão e corrente.

Esta PCB integra os transdutores de tensão discutidos na secção 4.2, o conversor analógico-digital e as ligações para os transdutores de corrente da Figura 6.2, também estes discutidos na secção 4.2.

Figura 6.2: PCB para leitura das correntes do sistema elétrico.

Para completar este sistema seria necessário dimensionar um inversor para injetar a corrente de compensação na instalação elétrica. O dimensionamento do inversor requer bons conhecimentos de eletrónica de potência e não basta apenas dimensionar. É fundamental fazer uma caracterização térmica do inversor, proteger contra sobretensões da rede e fornecer uma *gate drive* com alimentação isolada. Com frequências de comutação cada mais elevadas é necessário respeitar, cada vez mais, estes procedimentos.

Bibliografia

E. Alecrim, “Memórias,” *InfoWester*, 2011. [Online]. Available: http://www.infowester.com/memoria.php

a.Z. Alkar and M. Karaca, “An Internet-Based Interactive Embedded Data-Acquisition System for Real-Time Applications,” *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 3, pp. 522–529, Mar. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4636728

Barebox.org, “The Barebox Bootloader,” 2013. [Online]. Available: http://www.barebox.org/index.html

G. Bruzzone, M. Caccia, A. Bertone, and G. Ravera, “Standard Linux for embedded real-time manufacturing control systems,” *2006 14th Mediterranean Conference on Control and Automation*, pp. 1–6, Jun. 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4124912

M. Clive Max, *FPGAs: World Class Designs*. Newnes, 2009.

Computer Society, “IEEE Standards Interpretations for IEEE Standard Portable Operating System Interface for Computer Environments,” Tech. Rep., 1992. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=2893

DENX Software Engineering, “Das U-Boot - the Universal Boot Loader,” 2013. [Online]. Available: http://www.denx.de/wiki/U-Boot/WebHome

T. B. Developers, “Buildroot: Making Embedded Linux Easy,” 2013. [Online]. Available: http://buildroot.uclibc.org/

M. Dias, *Sistemas Digitais - Princípios e Prática*, 2nd ed. FCA - Editora de Informática, Lda, 2010.

Elinux.org, “Toolchains,” 2013. [Online]. Available: http://elinux.org/Toolchains

B. H. Fletcher, "FPGA Embedded Processors," Embedded Systems Conference, San Francisco, Tech. Rep., 2005. [Online]. Available: http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf

Fluke Corporation, "Fluke Corporation." [Online]. Available: http://www.fluke.com/

Fluke Products, "Fluke Products." [Online]. Available: http://www.fluke.com/fluke/caen/Power-Quality-Tools/Logging-Power-Meters/Fluke-437-Series-II.htm?PID=73946

L. Fu and R. Schwebel, "RT PREEMPT HOWTO," 2013. [Online]. Available: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

I. Grout, *Digital Systems Design with FPGAs and CPLDs.* Newnes, 2008.

C. Hallinan, "The lowdown on Embedded Linux and its use in real-time applications," 2007. [Online]. Available: http://www.embedded.com/design/operating-systems/4007107/The-lowdown-on-Embedded-Linux-and-its-use-in-real-time-applications-Part-2

HBM Corporation, "HBM." [Online]. Available: http://www.hbm.com/

IBM Data Processing Division, "The IBM Data Processing Division." [Online]. Available: http://www-03.ibm.com/ibm/history/exhibits/dpd50/dpd50_chronology2.html

International Business Machines Corporation, "IBM." [Online]. Available: http://www.ibm.com/

Z. Karakehayov, *Data Acquisition Applications*, 1st ed., Z. Karakehayov, Ed. Rijeka: InTech, Aug. 2012. [Online]. Available: http://www.intechopen.com/books/data-acquisition-applications

KOAN, "KOAN embedded software engineering." [Online]. Available: http://www.kaeilos.com/

LEM Co., "LEM Co." 2011. [Online]. Available: http://www.lem.com/

Lineo, "Lineo Solutions, Inc." [Online]. Available: http://www.lineo.co.jp/modules/english/

National Instruments Corporation, "FPGA Fundamentals," 2013. [Online]. Available: http://www.ni.com/white-paper/6983/en/

——, “National Instruments Corporation.” [Online]. Available: www.ni.com/

National Instruments Products, “NI USB-6009.” [Online]. Available: http://sine.ni.com/nips/cds/view/p/lang/pt/nid/201987

NIST, “Introduction to Linux for Real-Time Control,” National Institute of Standards and Technology, Tech. Rep., 2001. [Online]. Available: http://www.aeolean.com/html/RealTimeLinux/RealTimeLinuxReport-2.0.0.pdf

——, “National Institute of Standards and Technology,” 2013. [Online]. Available: http://www.nist.gov/

T. Noergaard, *Embedded Systems Architecture.* Newnes, 2005. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1276764

Nokia, “Qt Project,” 2013. [Online]. Available: http://qt-project.org/

OpenEmbedded Team, “OpenEmbedded Project.” [Online]. Available: http://www.openembedded.org

B. Parhami, *Arquitetura de Computadores - de Microprocessadores a Supercomputadores.* McGraw-Hill, 2008.

J. Park and S. Mackay, *Practical Data Acquisition for Instrumentation and Control Systems*, 1st ed., I. TECHNOLOGY, Ed. London: Elsevier Science, 2003.

R. S. Pinto, F. J. Monaco, J. C. Faracco, and J. R. Monteiro, “Embedded Linux in Real-Time Applications: Performance Enhancements of Experimental Fully-Preemptible Capabilities over the Standard Kernel in a Critical Mobile System,” *2012 Second Brazilian Conference on Critical Embedded Systems*, pp. 76–81, May 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6227660

F. M. Proctor and W. P. Shackleford, “TIMING STUDIES OF REAL-TIME LINUX FOR CONTROL,” 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9048&rep=rep1&type=pdf

Red Hat, “RedBoot,” 2013. [Online]. Available: https://sourceware.org/redboot/

J. C. H. Roth, *Digital Systems Design Using VHDL*, 2nd ed. Cengage Learning, 1997.

G. Sally, *Pro Linux Embedded Systems*, 1st ed. Berkeley, CA: Apress, 2010. [Online]. Available: http://www.springerlink.com/index/10.1007/978-1-4302-7226-7

A. Sedra and K. Smith, *Microelectronic Circuits*, 6th ed. Oxford University Press, 2009.

V. Silva, T. Malheiro, J. a. Mendes, J. Cabral, and a. Tavares, "Real-time low-cost industrial acquisition system," *2011 9th IEEE International Conference on Industrial Informatics*, pp. 763–767, Jul. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6034989

V. Silva, "Sistema de Aquisição de Dados Tempo Real Baseado em Linux," Ph.D. dissertation, Universidade do Minho, 2010.

M. SIMEK, "Linux device tree generator for the Xilinx SDK," 2013. [Online]. Available: https://github.com/Xilinx/device-tree

A. Tavares, C. Lima, C. Silva, J. Cabral, and P. Cardoso, *Programação de Microcontroladores*, 2009.

Xilinx Inc., "Xilinx XC2VP30," Tech. Rep., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, *Building embedded Linux systems*, 2nd ed., A. Oram, Ed. O'REILLY, 2008.

J. Yang and Y. Chen, "A Linux kernel with fixed interrupt latency for embedded real-time system," *Embedded Software and ...*, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1609868

V. Yodaiken, M. Barabanov, and C. Dougan, "High Performance and Deterministic System Software." [Online]. Available: http://www.fsmlabs.com/

Apêndice A

Toolchain

As *toolchains* são componentes essenciais para um projeto de desenvolvimento de *software*. Estas incluem todas as ferramentas de compilação, *assembler* e de *link* para desenvolvimento de código.

A.1 Preparação das Ferramentas de *Cross-Compile*

Abrir Linha de Comandos e na diretoria de trabalho fazer o *download* de Crosstool-ng:

```
$ wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-1.17.0.tar.bz2
```

Descomprimir:

```
$ tar -xjvf crosstool-ng-1.17.0.tar.bz2
```

Entrar na diretoria descomprimida:

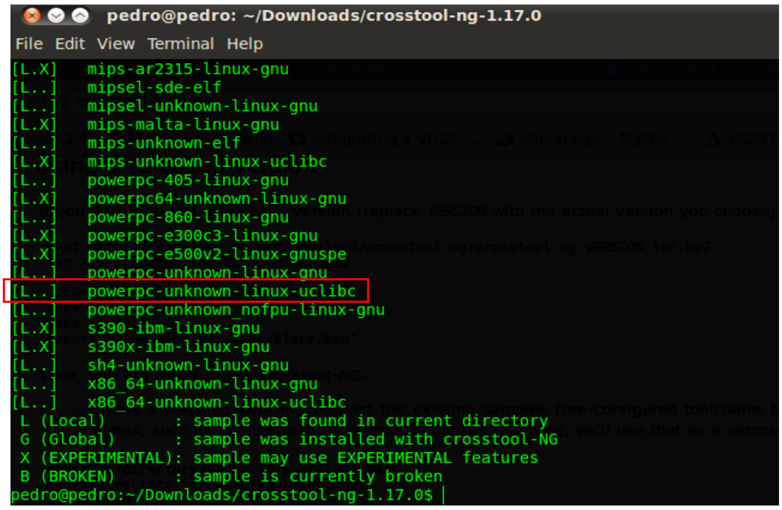
```
$ cd crosstool-ng-1.17.0/
```

E fazer configuração inicial:

```
$ ./configure –prefix=/home/
$ make
$ chmod a+rx ct-ng
$ make install
```

Listar todas as *toolchains* suportadas:

```
$ ./ct-ng list-samples
```



```
[L.X]   mips-ar2315-linux-gnu
[L..]   mipsel-sde-elf
[L..]   mipsel-unknown-linux-gnu
[L.X]   mips-malta-linux-gnu
[L..]   mips-unknown-elf
[L.X]   mips-unknown-linux-uclibc
[L..]   powerpc-405-linux-gnu
[L.X]   powerpc64-unknown-linux-gnu
[L..]   powerpc-860-linux-gnu
[L.X]   powerpc-e300c3-linux-gnu
[L.X]   powerpc-e500v2-linux-gnuspe
[L..]   powerpc-unknown-linux-gnu
[L..]   powerpc-unknown-linux-uclibc
[L..]   powerpc-unknown_nofpu-linux-gnu
[L.X]   s390-ibm-linux-gnu
[L.X]   s390x-ibm-linux-gnu
[L..]   sh4-unknown-linux-gnu
[L..]   x86_64-unknown-linux-gnu
[L..]   x86_64-unknown-linux-uclibc
 L (Local)       : sample was found in current directory
 G (Global)      : sample was installed with crosstool-NG
 X (EXPERIMENTAL): sample may use EXPERIMENTAL features
 B (BROKEN)      : sample is currently broken
pedro@pedro:~/Downloads/crosstool-ng-1.17.0$ |
```

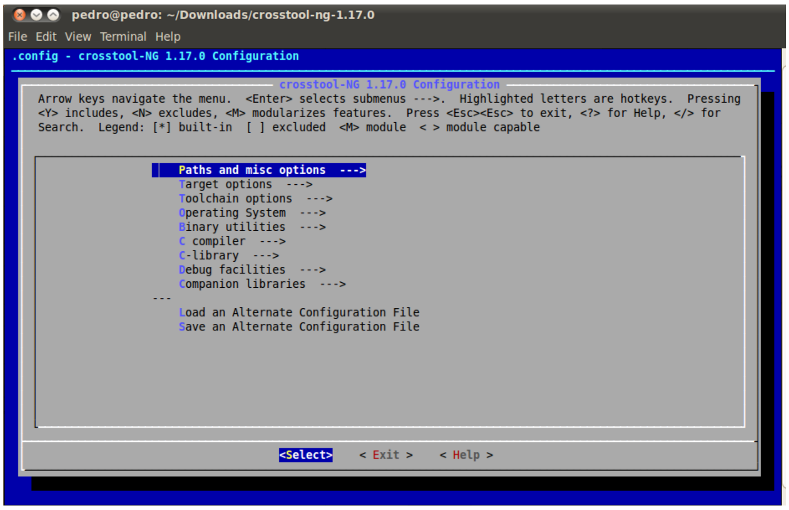
Figura A.1: Lista de *toolchains* suportados por crosstool-ng.

Configuar Crosstool-ng para powerpc-unknown-linux-uclibc:

$ ./ct-ng powerpc-unknown-linux-uclibc

Abrir janela de configuração:

$ ./ct-ng menuconfig

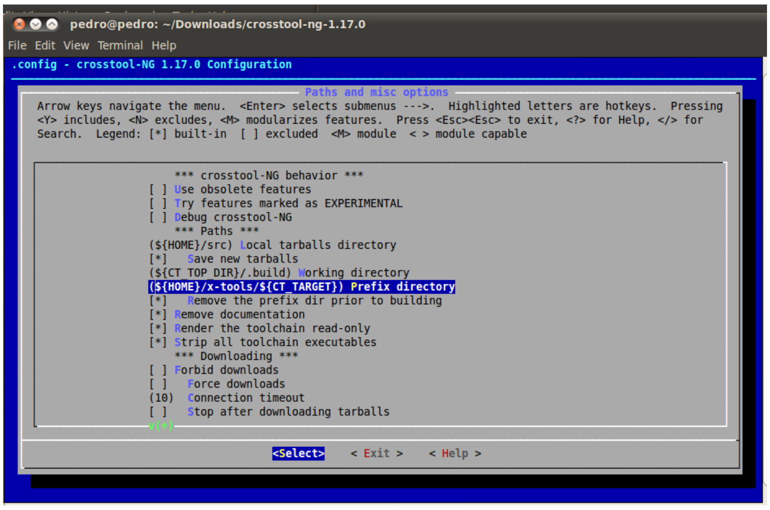


```
.config - crosstool-NG 1.17.0 Configuration
                     crosstool-NG 1.17.0 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing
  <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for
  Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                    Paths and misc options  --->
                    Target options  --->
                    Toolchain options  --->
                    Operating System  --->
                    Binary utilities  --->
                    C compiler  --->
                    C-library  --->
                    Debug facilities  --->
                    Companion libraries  --->
                ---
                    Load an Alternate Configuration File
                    Save an Alternate Configuration File

                          <Select>    < Exit >    < Help >
```

Figura A.2: Janela de configuração inicial de crosstool-ng.

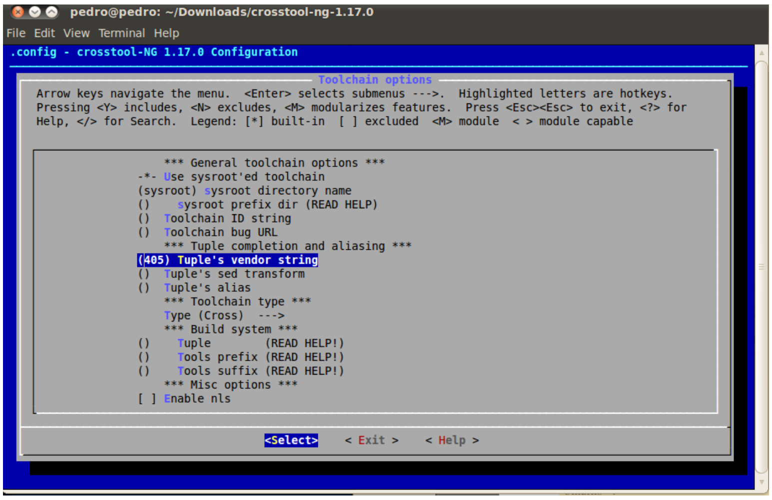
Fazer as seguintes alterações:
Se necessário, alterar o Caminho da instalação da *toolchain* para um caminho conhecido mas manter a *label* $CT_TARGET (ver Figura A.3).

Entrar em “Toolchian option”:
Mudar de “unknown” -> 405 (ver Figura A.4).



```
.config - crosstool-NG 1.17.0 Configuration
                              Paths and misc options
 Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing
 <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for
 Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                  *** crosstool-NG behavior ***
              [ ] Use obsolete features
              [ ] Try features marked as EXPERIMENTAL
              [ ] Debug crosstool-NG
                  *** Paths ***
              (${HOME}/src) Local tarballs directory
              [*]   Save new tarballs
              (${CT_TOP_DIR}/.build) Working directory
              (${HOME}/x-tools/${CT_TARGET}) Prefix directory
              [*]   Remove the prefix dir prior to building
              [*] Remove documentation
              [*] Render the toolchain read-only
              [*] Strip all toolchain executables
                  *** Downloading ***
              [ ] Forbid downloads
              [ ]   Force downloads
              (10)  Connection timeout
              [ ]   Stop after downloading tarballs
              v(+)

                        <Select>    < Exit >    < Help >
```

Figura A.3: Alteração do caminho de instalação da *toolchain*.



```
.config - crosstool-NG 1.17.0 Configuration
                              Toolchain options
 Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.
 Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for
 Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                 *** General toolchain options ***
             -*- Use sysroot'ed toolchain
             (sysroot) sysroot directory name
             ()    sysroot prefix dir (READ HELP)
             ()  Toolchain ID string
             ()  Toolchain bug URL
                 *** Tuple completion and aliasing ***
             (405) Tuple's vendor string
             ()  Tuple's sed transform
             ()  Tuple's alias
                 *** Toolchain type ***
                 Type (Cross)  --->
                 *** Build system ***
             ()    Tuple        (READ HELP!)
             ()    Tools prefix (READ HELP!)
             ()    Tools suffix (READ HELP!)
                 *** Misc options ***
             [ ] Enable nls

                        <Select>    < Exit >    < Help >
```

Figura A.4: Alteração do parâmetro "unknown" na string da *toolchain* para 405.

Compilar *toolchain*:

$ ./ct-ng build

Adicionar a *PATH* da sua máquina ao caminho das ferramentas do PowerPC:

$ export PATH=$PATH:$HOME/toolchain/x-tools/powerpc-405-linux-uclibc/bin

Criar um *Alias* para ser mais fácil futuramente o *cross-compile* do *software*:

$ alias ppcmake_uclibc="make ARCH=powerpc CROSS_COMPILE=powerpc-405-linux-uclibc-"

Apêndice B

Buildroot

O *buildroot* permite criar um sistema de ficheiros, imagem de um sistema operativo e criar ferramentas de *cross-compile.* Este anexo apenas descreve a criação de um sistema de ficheiros para o *kernel* Linux.

B.1 Criação do sistema de ficheiro Linux

Fazer *download* do *buildroot*:

```
$ wget http://buildroot.uclibc.org/downloads/buildroot-2009.02.tar.bz2
```

Descomprimir:

```
$ tar -xjvf buildroot-2009.02.tar.bz2
```

Entrar na directoria:

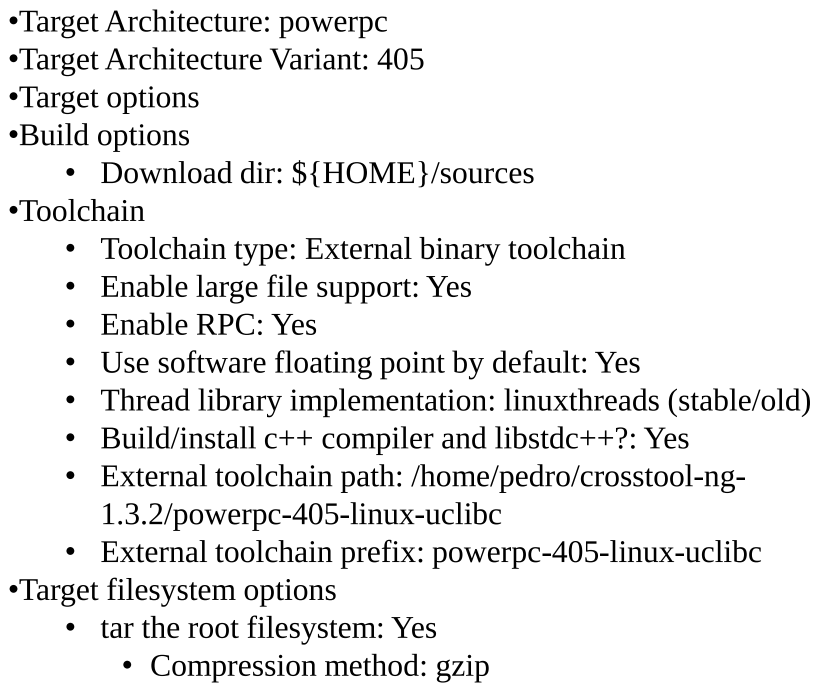
```
$ cd buildroot-2009.02
$ ppcmake_uclibc defconfig
```

Configuração do *buildroot*, ver Figura B.1:

```
$ ppcmake_uclibc menuconfig
```

Guardar e sair.
Compilar o sistema de ficheiros:

```
$ ppcmake_uclibc
```



- Target Architecture: powerpc
- Target Architecture Variant: 405
- Target options
- Build options
  - Download dir: ${HOME}/sources
- Toolchain
  - Toolchain type: External binary toolchain
  - Enable large file support: Yes
  - Enable RPC: Yes
  - Use software floating point by default: Yes
  - Thread library implementation: linuxthreads (stable/old)
  - Build/install c++ compiler and libstdc++?: Yes
  - External toolchain path: /home/pedro/crosstool-ng-1.3.2/powerpc-405-linux-uclibc
  - External toolchain prefix: powerpc-405-linux-uclibc
- Target filesystem options
  - tar the root filesystem: Yes
    - Compression method: gzip

Figura B.1: Opçoes de configuração do *buildroot*.

Quando terminar a compilação, o sistema de ficheiros estará na pasta: *buildroot-2009.02/binaries/uclibc*

Apêndice C

Linux RT

C.1 Compilação do *Kernel* RT

Download do *kernel* e *patch rt* :

```
$ wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz
$ wget http://www.kernel.org/pub/linux/kernel/projects/rt/patch-2.6.23.1-rt11.bz2
```

Fazer o *patching* do *kernel* genérico para *kernel* RT:

```
$ tar -xfj linux-2.6.23.1.tar.bz2
$ cd linux-2.6.23.1
$ bzcat ../patch-2.6.23.1-rt11.bz2 | patch -p1
```

Copiar o ficheiro *devce-tree* “xilinx.dts” para a diretoria *dts* do PowerPC no Linux:

```
$ cp $/CAMINHO/device-tree/xilinx.dts linux-2.6.23.1/arch/powerpc/boot/dts/virtex405-mine.dts
```

Entrar na diretoria do Linux:

```
$ cd linux-2.6.23.1/
```

Compilar o *kernel* do Linux com o seguinte comando:

```
$ ppcmake_uclibc -j 4 simpleImage.virtex405-mine
```

Será gerado o ficheiro com a imagem do sistema operativo *kernel* na pasta linux-2.6.23.1/arch/powerpc/boot:
simpleImage.virtex405-mine.elf

Apêndice D

Teoria p-q

Para cálculos das componentes da Teoria p-q é necessário uma transformação nas coordenadas a-b-c para um sistema de referência estacionário e de coordenadas α-β-0 ortogonais entre si. Esta transformação conhecida como transformação de Clarke (Edith Clarke, 1943).

As expressões para transformação direta do sistema de tensões e correntes em coordenada α-β-0 são as seguintes:

$$\begin{bmatrix} v_0 \\ v_\alpha \\ v_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \times T \times \begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} \tag{D.1}$$

$$\begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \times T \times \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{D.2}$$

onde:

$$T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & \frac{-1}{2} & \frac{-1}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{-\sqrt{3}}{2} \end{bmatrix} \tag{D.3}$$

A transformação inversa do sistema coordenadas a-b-c dá-se através das seguintes expressões:

$$\begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} = \sqrt{\frac{2}{3}} \times T^T \times \begin{bmatrix} v_0 \\ v_\alpha \\ v_\beta \end{bmatrix} \tag{D.4}$$

$$\begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} = \sqrt{\frac{2}{3}} \times T^T \times \begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} \tag{D.5}$$

onde:

$$T^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & 1 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{2} & \frac{\sqrt{3}}{2} \\ \frac{1}{\sqrt{2}} & \frac{-1}{2} & \frac{-\sqrt{3}}{2} \end{bmatrix} \tag{D.6}$$

As potências instantâneas real, imaginária e de sequência zero são expressas pela seguinte matriz (Silva, 2010):

$$\begin{bmatrix} p_0 \\ p \\ q \end{bmatrix} = \begin{bmatrix} v_0 & 0 & 0 \\ 0 & v_\alpha & v_\beta \\ 0 & v_\beta & -v_\alpha \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} \tag{D.7}$$