Universidade de Aveiro Departamento de Electrónica e Telecomunicações.
2000

Ruben Jaime Alegria Leote Mendes

Implementação de Sistemas de Telecomunicações Utilizando Dispositivos Lógicos Programáveis

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor João Nuno Pimentel da Silva Matos, Professor Auxiliar do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.

O júri

Presidente

Doutor José Fernando da Rocha Pereira
Professor Associado da Universidade de Aveiro

Doutor António Gusmão Correia
Professor Auxiliar do Instituto Superior Técnico da Universidade Técnica de Lisboa

Doutor João Nuno Pimentel da Silva Matos
Professor Auxiliar da Universidade de Aveiro

Dedicatória

Esta dissertação é dedicada aos meus Pais e ao meu Irmão.

Agradecimentos

Quero agradecer às seguintes pessoas pela colaboração que prestaram na realização desta dissertação:

Ao Prof. Dr. João Nuno Matos, meu orientador, por ter aceite supervisionar este trabalho, pelo apoio científico e pela permanente disponibilidade.

A todos os amigos que deixei na *Fachhochschule Kiel* pela maneira como me receberam e por me terem feito sentir como se estivesse em casa. Quero agradecer em particular ao Prof. Dr. Helmut Dispert, Engº. Manfred Fränz, Engº. Matthias Karger, Dirk Metzner.

Aos meus colegas da Universidade de Aveiro em especial à Engª. Lurdes Sousa e Engª. Filipa Borrego pela preciosa ajuda técnica e logística que me prestaram.

Ao Engº. Henrique Miranda do INESC-Porto pelas interessantes discussões técnicas e por toda a bibliografia que me facultou.

Resumo

A utilização de dispositivos lógicos programáveis, especialmente FPGA's e CPLD's, é cada vez mais comum. Estes dispositivos podem mesmo vir a ter um impacto semelhante ao provocado pelo microprocessador. Os sistemas de telecomunicações são dos que mais têm a ganhar com a utilização de lógica programável.

O tema principal desta dissertação consistiu em criar uma plataforma que facilite a simulação, desenvolvimento e teste de sistemas de rádio digital, tirando partido da versatilidade oferecida pelos dispositivos lógicos programáveis. Para tal foi montada uma placa com um destes dispositivos, desenvolvido software de comunicação entre a placa e um PC e implementados alguns dos algoritmos essenciais utilizando a linguagem VHDL.

Após uma introdução aos conceitos básicos relacionados com os sistemas de rádio digital é apresentada uma descrição desta plataforma e do trabalho que foi realizado no âmbito desta dissertação.

Abstract

The use of programmable logic devices, especially FPGA's and CPLD's, is becoming very common. This devices can have in the future an impact as important as the microprocessor had in the past. Telecommunication systems can have great advantages by the use of programmable logic devices.

The main subject of this thesis is the creation of a platform that allows the simulation, development, and testing of digital radio systems, taking advantage of the versatility of programmable logic devices. A board was built with a programmable device, software was written for the communication between the board and an PC, and some essencial algorithms were implemented using VHDL.

After an introduction to the basic concepts of digital radio, we give a description of this platform and the work performed for this thesis.

Índice

Lista de Siglas

AC	*Alternate Current*
ADC	*Analog-to-Digital Converter*
AM	*Amplitude Modulation*
ASK	*Amplitude Shift Keying*
ASIC	*Application Specific Integrated Circuit*
BB	*Base Band*
CDROM	*Compact Disk Read Only Memory*
CMOS	*Complementary Metal Oxide Silicon*
CORDIC	*COordinate Rotation DIgital Computer*
CPLD	*Complex Programmable Logic Device*
DAC	*Digital-to-Analog Converter*
DC	*Direct Current*
DIP	*Dual In-line Package*
DSB	*Double Side Band*
DSP	*Digital Signal Processor*
EDIF	*Electronic Design Interface Format*
EIA	*Electronic Industries Association*
FEC	*Forward Error Correction*
FFT	*Fast Fourier Transform*
FIR	*Finite Impulse Response*
FPGA	*Field Programmable Gate Array*
FM	*Frequency Modulation*
FSK	*Frequency Shift Keying*
HDL	*Hardware Description Language*
HDLC	*High-level Data Link Control*
IEEE	*Institute of Electrical and Electronic Engineering*
IF	*Intermediate Frequency*

IIR	*Infinite Impulse Response*
ISP	*In-System Programmable*
LPM	*Library of Parameterized Modules*
LSB	*Least Significant Bit*
LUT	*Look-Up Table*
MSB	*Most Significant Bit*
NCO	*Numerically Controlled Oscilator*
OOK	*On-Off Keying*
PC	*Personal Computer*
PDF	*Portable Document Format*
PLCC	*Plastic j-Lead Chip Carrier*
PLD	*Programmable Logic Device*
PLL	*Phase Locked Loop*
PM	*Phase Modulation*
PSK	*Phase Shift Keying*
RAM	*Random Access Memory*
RF	*Radio Frequency*
ROM	*Read Only Memory*
SNR	*Signal to Noise Ratio*
VCO	*Voltage Controlled Oscilator*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Actualmente verifica-se uma tendência para a substituição de electrónica analógica por circuitos digitais que desempenham uma função equivalente. A utilização de circuitos digitais apresenta diversas vantagens relativamente aos seus predecessores analógicos, nomeadamente uma maior imunidade a variações de temperatura e ao envelhecimento, uma maior homogeneidade no desempenho de unidade para unidade, uma maior integração permitindo, entre outras, uma maior miniaturização. Nos sistemas de telecomunicações a utilização de filtros digitais por exemplo, permite a construção de filtros de fase linear que são complicados de implementar através de filtros analógicos [1].

Embora grande parte deste texto seja válido para outros tipos de sistemas de telecomunicações, será dada uma ênfase especial aos sistemas de comunicação sem fios, e em particular aos que utilizam as ondas de rádio como meio de comunicação.

Nos sistemas de comunicação por rádio importa distinguir entre a utilização de modulação digital e a utilização de circuitos digitais. A primeira refere-se à utilização de moduladores e desmoduladores desenhados para comunicar utilizando sinais cuja origem é discreta em amplitude (por oposição aos sistemas de modulação analógica que utilizam sinais contínuos em amplitude). A segunda compreende a utilização de conversores analógico-digitais (ADC) e conversores digital-analógicos (DAC) para transformar o sinal do domínio analógico para o digital e vice-versa, sendo parte do processamento efectuado digitalmente. Note-se que é perfeitamente possível implementar uma modulação analógica (AM por exemplo) utilizando circuitos digitais ou implementar uma modulação digital utilizando circuitos analógicos (usando um modulador de FM para modular FSK por exemplo). Aos sistemas de telecomunicações que utilizam processamento digital nas etapas de rádio-frequência (RF), frequência intermédia (FI) ou banda base (BB), damos o nome de rádios digitais.

A partir do momento em que o sinal está na forma digital as operações que sobre ele se efectuam passam para o domínio do processamento digital de sinal. A qualidade da implementação depende apenas dos algoritmos utilizados que obviamente estão sujeitos às limitações de velocidade e capacidade impostas pelo processador digital escolhido. Este processador pode ser um microprocessador de uso geral, tal como o utilizado num comum computador pessoal (PC), mas mais frequentemente é utilizado um processador optimizado para o processamento digital de sinal, conhecido por DSP.

Outra hipótese, que tem vindo a ganhar popularidade, é a utilização de circuitos integrados de aplicação específica (ASIC) ou, em alternativa, a utilização de dispositivos lógicos programáveis (PLD) tais como FPGA's e CPLD's. A produção de um ASIC só se torna economicamente viável se este for fabricado em grandes quantidades. Para produtos fabricados em pequenas quantidades, ou durante a fase de desenvolvimento de um produto, a utilização de PLD's é a solução preferida. Uma característica presente na quase totalidade dos PLD's modernos é a possibilidade de serem programados repetidas vezes sem ser necessário retirá-los do circuito (ISP). Isto facilita imenso a fase de desenvolvimento do produto e torna possível, por exemplo, o envio de um ficheiro para os clientes contendo novas funcionalidades e/ou correcções de defeitos (*bugs*). Recentemente vulgarizou-se a utilização do nome *software radio* para designar os rádios digitais cuja funcionalidade é reconfigurável [2].

Quando comparados com os DSP's, os PLD's possuem uma maior integração, uma menor dissipação de potência e um maior desempenho [3] o que os torna na primeira escolha para sistemas portáteis e que utilizem frequências de trabalho elevadas e grandes larguras de banda.

Pelas razões apresentadas, e tendo em conta o aumento constante da capacidade e progressiva redução do preço, será de esperar que cada vez mais sistemas de rádio digital utilizem PLD's para efectuar o processamento do sinal digital.

1.1 Objectivos

Pretendeu-se com este trabalho criar uma plataforma que facilite a simulação, desenvolvimento e teste de sistemas de rádio digital, tirando partido da versatilidade oferecida pelos dispositivos lógicos programáveis. Para se obter este objectivo previu-se:

- A montagem de uma placa contendo:
  - um PLD reprogramável
  - um conversor analógico-digital
  - um conversor digital-analógico
  - um interface para comunicação com um PC
- O desenvolvimento de software no PC para comunicação com a placa
- A implementação de algoritmos de rádio digital

A placa não contém quaisquer andares analógicos tal como filtros, andares de frequência intermédia ou de rádio-frequência, pois os requisitos destes andares variam de projecto para projecto. Os próprios conversores não são de utilização universal pois cada projecto tem necessidades diferentes de frequências de conversão e de resolução (número de bits). No entanto, a presença dos conversores é indispensável para efectuar testes e medições pelo que estes devem ser incluídos.

Um emissor é composto por vários andares que transformam sucessivamente o sinal de informação em banda base até obter o sinal de saída modulado à frequência de interesse. Este sinal propaga-se pelo canal de comunicação até atingir o receptor que efectua as operações inversas. Praticamente todos os andares do emissor e receptor, desde o andar de rádio-frequência até à banda base passando pela frequência intermédia, são passíveis de serem implementados digitalmente. Neste trabalho, a implementação das etapas mais junto à antena, tais como os moduladores e desmoduladores, são as etapas prioritárias.

1.2 Outras Considerações

Este documento foi preparado utilizando o programa LaTeX 2_ε [4]. Além da versão em papel está disponível também em CDROM, em formato PDF e Postscript, através dos Serviços de Documentação da Universidade de Aveiro. No CDROM encontra-se também o código presente nos apêndices.

O trabalho aqui descrito foi parcialmente efectuado na *Fachhochschule Kiel* na Alemanha ao abrigo do programa *Erasmus*.

Capítulo 2

Fundamentos de Rádio Digital

Neste texto estamos principalmente interessados nos algoritmos de processamento de sinais digitais. No entanto, os sinais de rádio-frequência são inerentemente analógicos pelo que é importante abordar as etapas que efectuam a conversão do domínio analógico para o digital e vice-versa, bem como as topologias analógicas situadas entre a antena e os conversores.

As operações efectuadas na emissão são semelhantes às efectuadas na recepção. No entanto a complexidade do receptor é normalmente maior que a do emissor uma vez que precisa de extrair um sinal fraco do meio do ruído. Por esta razão vamos neste capítulo cingir-nos ao estudo dos receptores.

2.1 Conversão Analógico-Digital

Nesta secção vamos abordar o desempenho dos conversores analógico-digitais pois eles influenciam largamente o desempenho dos receptores digitais. O ADC irá influenciar, entre outros, a(s) frequência(s) intermédia(s) e a gama dinâmica, que está relacionada com a sensibilidade do receptor uma vez que esta é o limite inferior da gama dinâmica. O ADC é por natureza um dispositivo não linear pelo que a análise do seu desempenho é limitada e bastante complexa.

2.1.1 Ruído de quantificação

Num ADC ideal o único erro existente deve-se ao ruído de quantificação. O sinal de entrada analógico, que pode assumir qualquer valor (espera-se que apenas dentro da gama de entrada do conversor), é convertido para uma sequência numérica de precisão finita. Uma vez que o erro pode ser qualquer valor dentro do nível de quantificação, podemos assumir que a distribuição do sinal de erro é uniforme. Neste caso a relação sinal-ruído (SNR), quando a entrada é um sinal sinusoidal com amplitude abrangendo toda a gama de entrada do conversor, é dada por [5][6]:

$$\mathrm{SNR_q} = 6,02b + 1,76 \text{ dB} \tag{2.1}$$

em que b é o número de bits do conversor.

2.1.2 Incerteza temporal de amostragem

Na prática o ADC não é ideal e existem outros factores que contribuem para reduzir a SNR. Destes factores destaca-se a incerteza temporal de amostragem (*aperture jitter*), ou seja a oscilação do instante exacto em que é feita a amostragem. Se considerarmos que esta incerteza segue uma distribuição normal com média zero e desvio padrão igual ao *jitter*, t_a, então a SNR devido apenas à incerteza temporal é dada por [6]:

$$\mathrm{SNR_j} = 20 \log_{10}\left(\frac{1}{2\pi f t_a}\right) \text{ dB} \tag{2.2}$$

Como seria de esperar, a SNR decresce à medida que a frequência,f, do sinal de entrada aumenta. Se juntarmos a contribuição do ruído devido à incerteza temporal de amostragem à contribuição do ruído de quantificação obtemos a expressão [5]:

$$\mathrm{SNR_{qj}} = 10 \log_{10}\left[\frac{3\ 2^{2b}}{2 + 3(2^b 2\pi f t_a)^2}\right] \text{ dB} \tag{2.3}$$

2.1.3 Sobre-amostragem

Um método de aumentar a SNR é através da utilização de sobre-amostragem, ou seja, amostrar o sinal a uma frequência muito maior do que a essencial. Como a potência do ruído de quantificação ($P_q = \frac{q^2}{12R}$) é independente da frequência de amostragem, f_s, então ao aumentarmos esta última estamos a espalhar a potência do ruído por uma

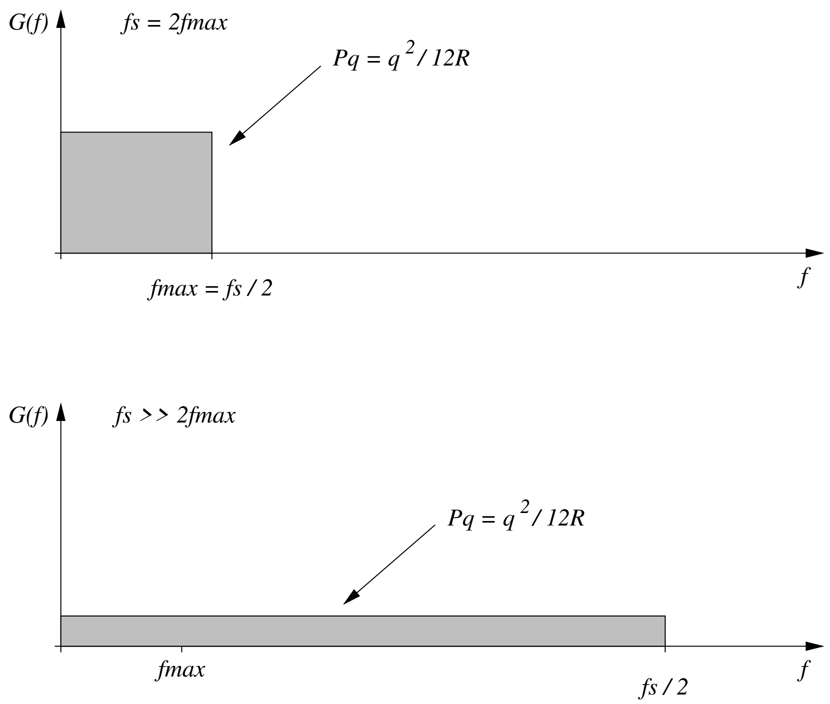


Figura 2.1: Efeito da sobre-amostragem na densidade espectral do ruído de quantificação, $G(f)$.

banda de frequências maior diminuindo a sua densidade espectral [6][7]. Como resultado, o ruído presente na banda de interesse ($f < f_{max}$) vem diminuído, melhorando efectivamente a relação sinal-ruído. Este fenómeno está ilustrado na figura 2.1.

A acção da sobre-amostragem é por vezes usada para obter um valor de SNR superior ao que inicialmente parece ser possível: um conversor A/D de 8-*bit* a operar à taxa de conversão de 20 Msps, por exemplo, pode obter valores máximos de SNR de 68 dB em vez de 48 dB para sinais de 100 KHz de banda, usando filtragem digital adequada [7].

2.1.4 Amostragem harmónica

Normalmente assume-se que a frequência de amostragem tem de ser pelo menos duas vezes maior que a frequência mais elevada presente à entrada do ADC, de modo a satisfazer o critério de Nyquist e evitar o *aliasing*. No entanto, esta condição apenas é necessária se o sinal tiver componentes desde DC. Normalmente num receptor o sinal desejado é um sinal de banda estreita em redor de uma determinada frequência. Neste caso é possível diminuir a frequência de amostragem para um valor muito abaixo do

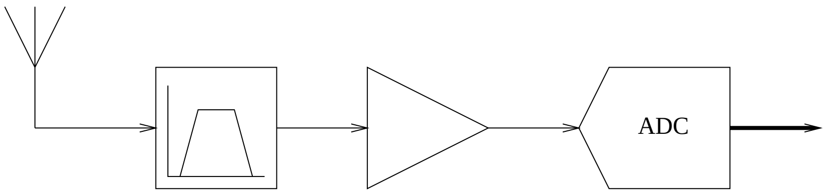


Figura 2.2: Receptor de digitalização directa

dobro da frequência máxima do sinal que se pretende receber. Para determinados valores da frequência central do sinal será possível utilizar uma frequência de amostragem igual ao dobro da largura de banda do sinal. Matematicamente a equação que garante que não ocorre *aliasing* é a seguinte [1]

$$(Mf_s + B) < 2f_c < (M+1)f_s - B \tag{2.4}$$

onde f_s é a frequência de amostragem, B é a largura de banda do sinal, f_c é a frequência em que o sinal está centrado, e M é um inteiro positivo. Como exemplo considere-se um sinal de micro-ondas com 20 MHz de largura de banda que foi convertido e filtrado de modo a ficar situado entre os 120 MHz e os 140 MHz. Este sinal poderá ser amostrado a uma frequência igual a 40 MHz em vez dos 280 MHz que seria de esperar. Isto, obviamente, se as características do conversor analógico-digital, tal como a incerteza temporal de amostragem, o permitirem. A este tipo de amostragem dá-se o nome de amostragem harmónica ou sub-amostragem.

2.2 Topologias

2.2.1 Receptor de digitalização directa

O método mais directo de digitalização é o apresentado na figura 2.2. Neste método, o sinal de rádio-frequência é digitalizado directamente, existindo além do conversor analógico-digital apenas um filtro passa banda e um amplificador. O filtro tem uma dupla função. Por um lado atenua as frequências fora da banda de interesse, diminuindo a gama dinâmica do sinal. Por outro lado limita a banda de frequências entregues ao ADC de modo a evitar o *aliasing*. Depois de filtrado, o sinal é amplificado de modo a ter uma amplitude compatível com a entrada analógica do ADC.

Este método apresenta no entanto dificuldades de realização prática e não é viável excepto para recepção de sinais de frequências baixas. Existe um limite prático para o

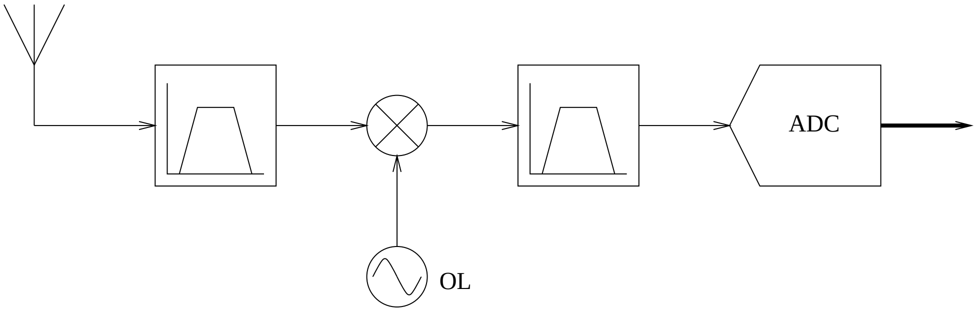


Figura 2.3: Receptor heterodino

factor de qualidade (Q) na construção de filtros analógicos. À medida que a frequência central do filtro se torna mais elevada, a banda passante irá em consequência aumentar. Isto exige mais do ADC, por um lado porque a frequência de amostragem tem de acompanhar o aumento da largura de banda para que não ocorra *aliasing*, e por outro lado porque o aumento da largura de banda implica um acréscimo da gama dinâmica que tem de ser acompanhado por um aumento do número de bits do conversor. Além do ADC, o processador digital necessita também de maior poder de processamento para acompanhar o maior ritmo de dados resultante da maior taxa de amostragem e maior número de bits, e porque terá de efectuar filtragens digitais que não foram possíveis de efectuar analogicamente.

Apesar das dificuldades, será de esperar que à medida que os ADC's e os processadores digitais evoluam, a utilização desta topologia se torne viável em bandas de frequências cada vez mais elevadas.

2.2.2 Receptor heterodino

O receptor heterodino, também chamado de receptor super-heterodino ou receptor de frequência intermédia [8], é sem dúvida a topologia mais utilizada tanto em receptores clássicos como nos mais modernos receptores digitais. A sua principal característica consiste na transladação do sinal de rádio-frequência para uma frequência mais baixa, a que se dá o nome de frequência intermédia (FI), de modo a facilitar a implementação dos filtros. A figura 2.3 representa esquematicamente este tipo de receptor. As etapas amplificadoras não estão representadas de modo a não sobrecarregar a figura.

O sinal de rádio-frequência desejado é convertido para a frequência intermédia através da multiplicação (mistura) com um sinal sinusoidal gerado no oscilador local (OL). Sendo $c(t) = \cos(\omega_c t)$ o sinal de rádio-frequência (representado apenas pela portadora),

$l(t) = \cos(\omega_l t)$ o sinal do oscilador local (OL), e dada a identidade trigonométrica

$$\cos(x)\cos(y) = \frac{1}{2}\{\cos(x+y) + \cos(x-y)\} \tag{2.5}$$

então o sinal de frequência intermédia $i(t)$ é dado por

$$i(t) = \frac{1}{2}\{\cos[(\omega_c + \omega_l)t] + \cos[(\omega_c - \omega_l)t]\} \tag{2.6}$$

A primeira componente (componente soma) encontra-se a uma frequência superior à do sinal pretendido pelo que não tem interesse e deverá ser eliminada através de filtragem. A componente diferença encontra-se a uma frequência mais baixa e é portanto a componente de interesse. O sinal $i(t)$, ignorando a constante multiplicativa, simplifica então para

$$i(t) = \cos[(\omega_c - \omega_l)t] \tag{2.7}$$

A frequência intermédia, ω_i, é então dada por

$$\omega_i = \omega_c - \omega_l \tag{2.8}$$

Existe, no entanto, outra frequência além de ω_c que é convertida para ω_i quando multiplicada por ω_l. Essa frequência, a que se dá o nome de frequência imagem, é dada por

$$\omega_m = \omega_c - 2\omega_i \tag{2.9}$$

Podemos verificar que assim é calculando qual o sinal presente após a multiplicação quando aplicamos na entrada uma sinusóide a esta frequência.

$$\begin{aligned} i'(t) &= \cos(\omega_m t)\cos(\omega_l t) & (2.10)\\ &= \frac{1}{2}\{\cos[(\omega_m + \omega_l)t] + \cos[(\omega_m - \omega_l)t]\} & (2.11)\end{aligned}$$

Ignorando mais uma vez a componente soma e a constante multiplicativa temos

$$i'(t) = \cos[(\omega_m - \omega_l)t]$$

Substituindo a equação 2.9 vem

$$i'(t) = \cos[(\omega_c - \omega_l - 2\omega_i)t]$$

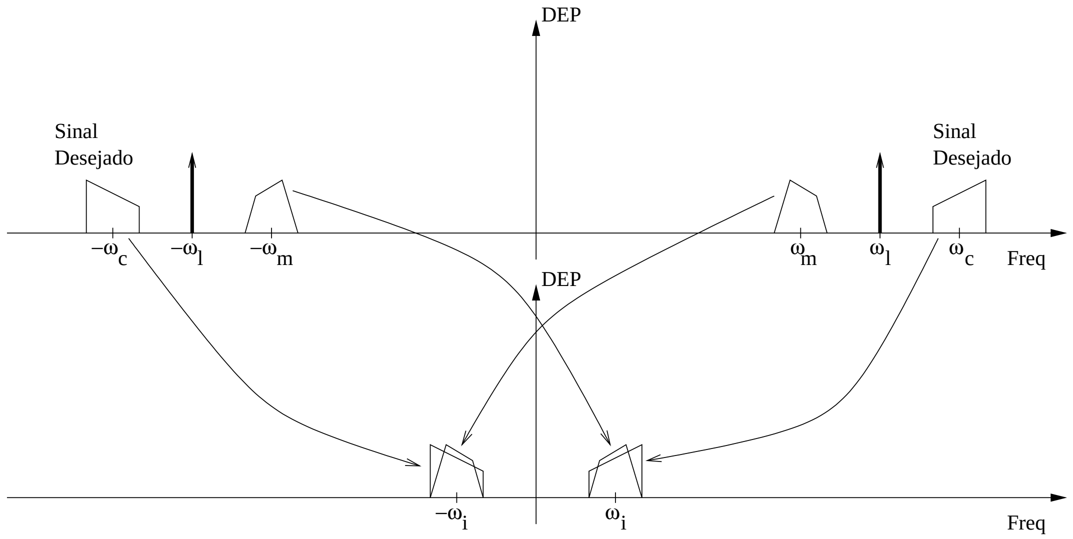


Figura 2.4: Efeito da frequência imagem num receptor heterodino

Utilizando a equação 2.8 vem

$$i'(t) = \cos[(\omega_i - 2\omega_i)t] \quad (2.12)$$
$$= \cos(-\omega_i t) \quad (2.13)$$

Uma vez que o coseno é uma função par, ou seja $\cos(x) = \cos(-x)$, fica provado que a frequência imagem é convertida também para a frequência intermédia.

A figura 2.4 ilustra no domínio da frequência o que demonstrámos no domínio do tempo. As frequências negativas (positivas) são transladadas para a direita (esquerda) devido à convolução com o impulso delta[1] de frequência positiva (negativa). O sinal imagem fica portanto sobreposto ao sinal desejado. Para que tal não aconteça é necessário eliminar o sinal imagem antes da transladação para a frequência intermédia. É essa a função do filtro passa banda que se encontra entre a antena e o misturador na figura 2.3. O segundo filtro tem a mesma função do filtro do receptor de digitalização directa que vimos anteriormente.

A escolha da frequência intermédia passa por um compromisso entre a complexidade destes dois filtros. Se a FI for baixa o segundo filtro será mais simples, mas a frequência imagem fica mais próxima da frequência desejada complicando o primeiro filtro. A utilização de uma FI alta afasta a frequência imagem facilitando o primeiro filtro mas aumenta a complexidade do segundo filtro. Um método que permite a simplificação de

[1]Este sinal é também conhecido como impulso de Dirac.

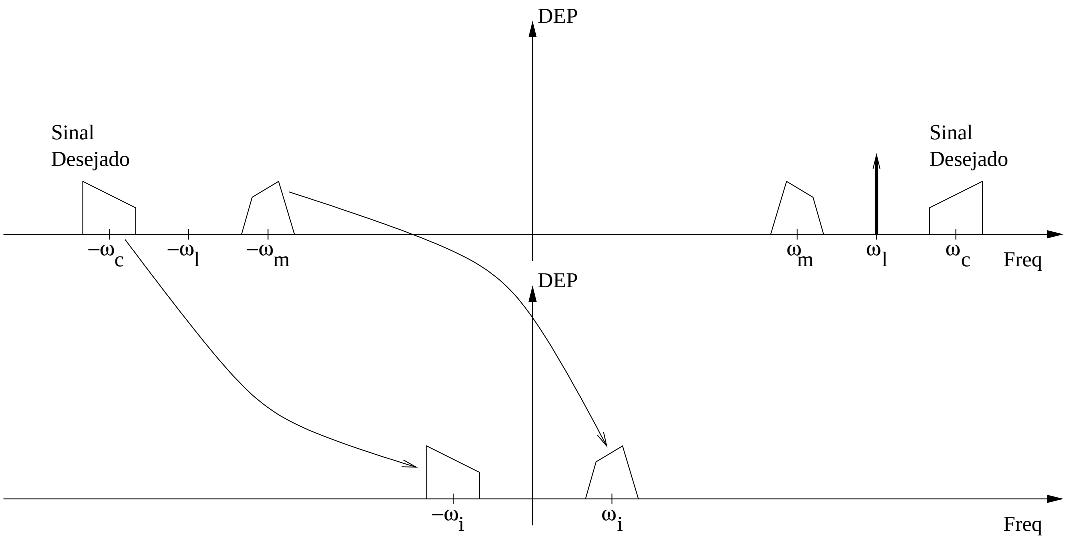


Figura 2.5: O processo de conversão do receptor de quadratura ideal

ambos os filtros é a utilização de mais do que uma FI. Por cada FI será necessário um misturador e um filtro passa banda para eliminar a nova frequência imagem.

2.2.3 Receptor de quadratura

O problema da frequência imagem do receptor heterodino existe porque o sinal da antena é multiplicado por um coseno que, no domínio da frequência, é composto por dois impulsos delta, um numa frequência positiva e outro numa frequência negativa. O espectro resultante é pois a convolução do espectro presente na antena com estes dois impulsos, ficando o sinal imagem sobreposto ao sinal pretendido.

Se o espectro do sinal do oscilador local for composto por apenas um impulso delta, seja ele a uma frequência positiva ou negativa, o problema referido já não ocorre como se pode observar na figura 2.5. Matematicamente o sinal do oscilador local é dado por

$$l(t) = e^{\pm \mathrm{j}\omega_l t} \tag{2.14}$$

$$= \cos(\omega_l t) \pm \mathrm{j}\sin(\omega_l t) \tag{2.15}$$

O sinal $l(t)$ é um sinal complexo logo o sinal de frequência intermédia será também complexo.

A implementação deste receptor está esboçada na figura 2.6 onde o misturador simples utilizado no receptor heterodino foi substituído por um misturador de quadratura

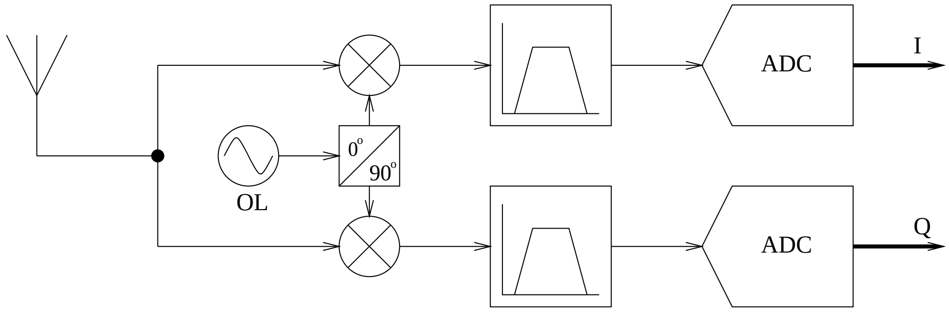


Figura 2.6: Receptor de quadratura

com duas saídas, I e Q. Esta topologia apresenta várias vantagens. Por um lado o filtro de rejeição da frequência imagem não é necessário. Além disso é ainda possível a utilização de uma frequência intermédia baixa. O sinal não desejado que se encontra na parte inferior do lado direito na figura 2.5 pode ser eliminado analogicamente ou digitalmente utilizando um filtro complexo que rejeite apenas frequências positivas.

O que foi dito anteriormente parte do princípio que os sinais do oscilador local aplicados nos misturadores têm a mesma amplitude e que estão desfasados de exactamente 90 graus. Na prática existem sempre desvios de amplitude e fase, bem como desequilíbrios entre os misturadores, o que faz com que o impulso delta de frequência negativa não tenha amplitude zero como desejado. Este efeito está ilustrado na figura 2.7. Embora atenuado, o sinal imagem encontra-se sobreposto ao sinal pretendido. Se existir, por exemplo, um erro de fase de 2 graus então o sinal imagem estará atenuado de 35 dB supondo um alinhamento perfeito das amplitudes [6]. Em alguns sistemas este valor pode não ser suficiente pois a potência do sinal imagem pode ultrapassar em 50 dB ou mais a potência do sinal desejado. Neste caso é necessário a inclusão de um filtro que atenue o sinal imagem, semelhante ao utilizado no receptor heterodino. Para que este filtro seja realizável na prática poderá ter de se utilizar uma FI elevada.

2.2.4 Receptor de dupla quadratura

Uma observação atenta do processo de conversão revela que apenas a componente de frequência negativa do sinal desejado é importante, e que as componentes de frequência positiva tanto do sinal desejado como do sinal imagem não são necessárias e no caso do sinal imagem são mesmo prejudiciais. Assim sendo, podemos eliminar as frequências positivas através de um filtro complexo como exemplificado na figura 2.8.

A implementação desta topologia está esquematizada na figura 2.9. O filtro com-

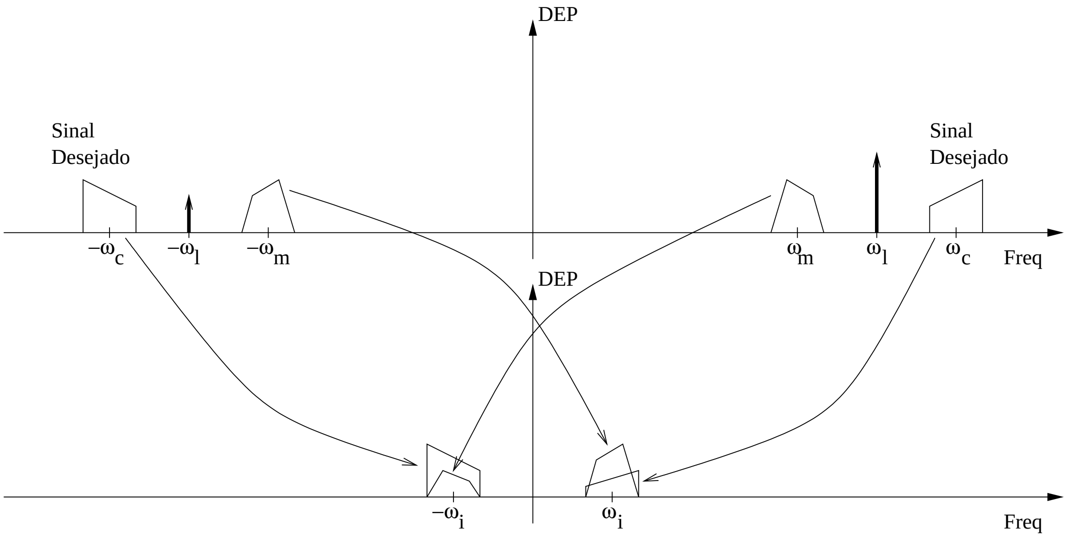


Figura 2.7: O processo de conversão do receptor de quadratura real

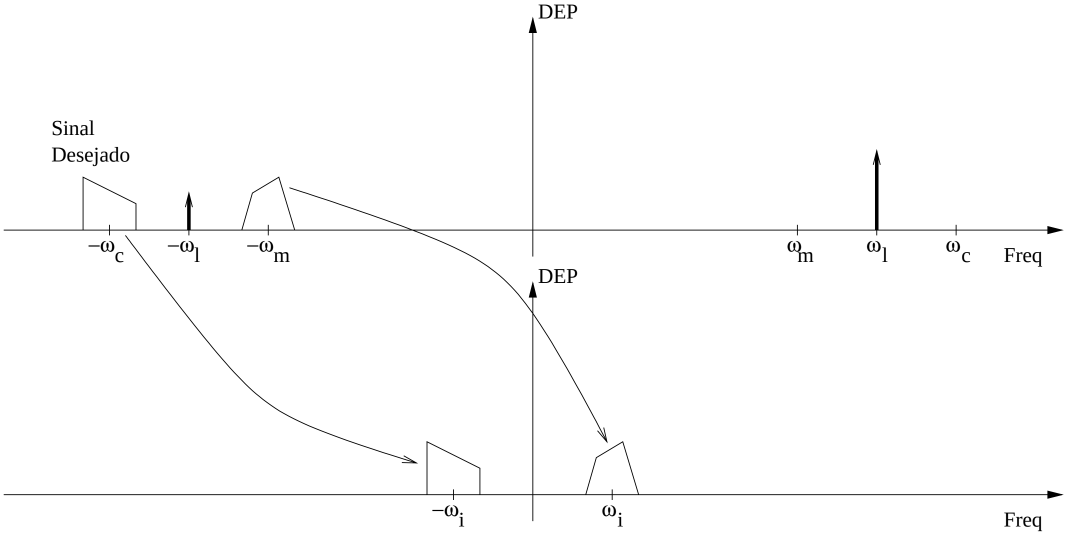


Figura 2.8: O processo de conversão do receptor de dupla quadratura

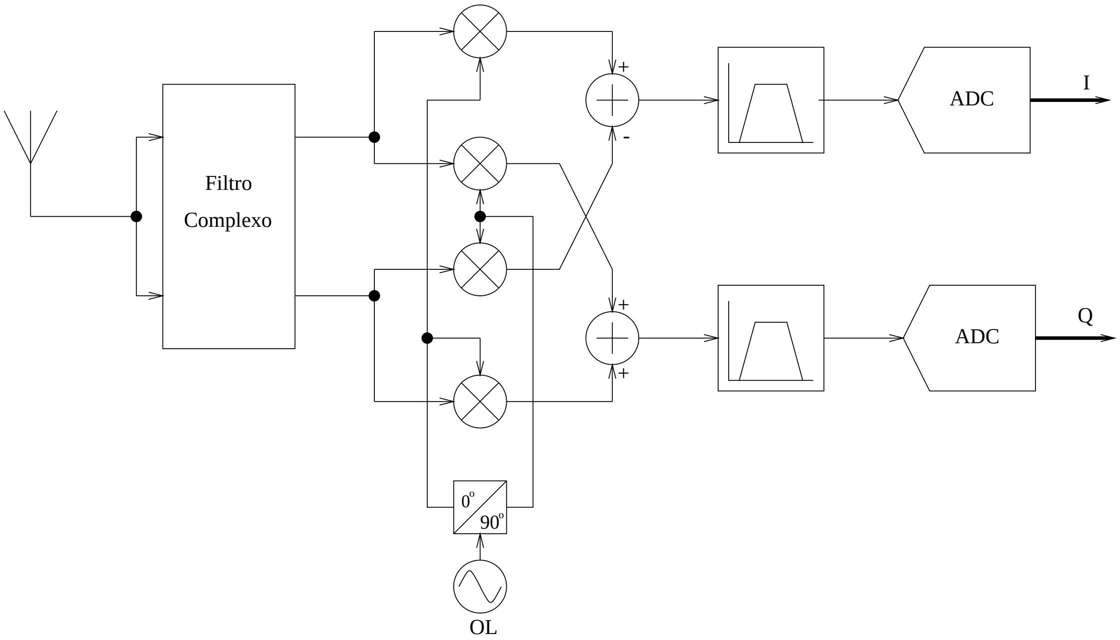


Figura 2.9: Receptor de dupla quadratura

plexo não necessita de ter um factor de qualidade (Q) muito alto como era o caso nas topologias referidas anteriormente, mesmo quando a frequência intermédia é baixa e portanto o sinal imagem está próximo do sinal pretendido. Esta filtragem pode ser implementada utilizando um filtro RC polifásico [8]. Uma vez que a saída do filtro complexo é também um sinal complexo, é necessário utilizar uma estrutura composta por quatro misturadores e dois somadores para efectuar a conversão para a FI. Esta estrutura tem o nome de misturador complexo completo ou de dupla quadratura e daí o nome deste receptor.

2.2.5 Receptor FI-zero

Um caso particular de utilização das topologias que fazem uso de uma frequência intermédia é o da conversão do sinal para banda base, ou seja, para uma FI igual a zero. Neste caso o sinal imagem é o próprio sinal desejado. Note-se que isto não elimina o problema do sinal imagem pois este é o espelho do sinal desejado. A utilização da topologia heterodina, em que a conversão é feita pela multiplicação por uma sinusóide, apenas é possível se a modulação empregue gerar sinais em que a banda lateral superior e inferior são o espelho uma da outra, como é o caso por exemplo da modulação de amplitude (AM). Nos restantes casos é necessário o emprego de um conversor de quadratura ou de dupla quadratura para rejeitar a frequência imagem. A figura 2.10

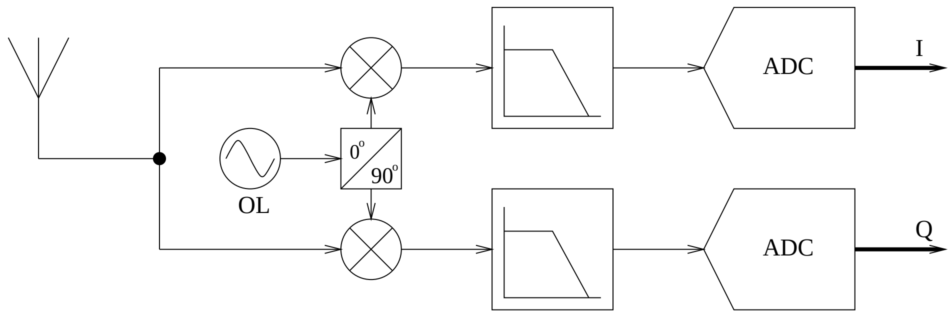


Figura 2.10: Receptor de quadratura com FI zero

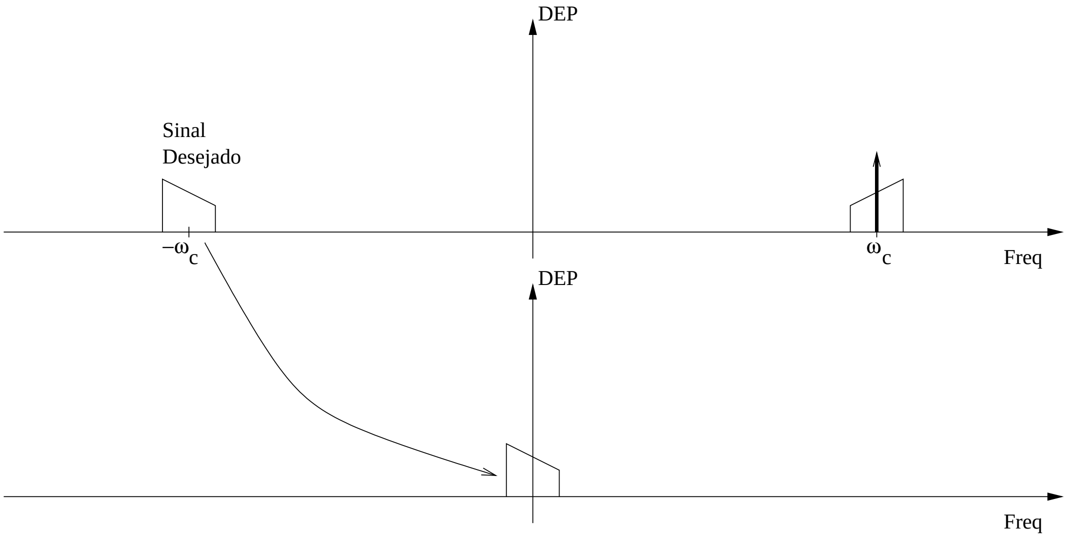


Figura 2.11: O processo de conversão do receptor de quadratura com FI zero

representa o diagrama de blocos de um receptor de quadratura em que a FI é zero, estando as respectivas conversões de frequência ilustradas na figura 2.11. Os filtros utilizados são agora do tipo passa-baixo.

A importância da supressão da frequência imagem não é tão grande nos receptores de FI-zero como nos receptores em que a FI não é zero uma vez que nestes últimos o sinal imagem pode ter uma potência muito maior que o sinal desejado. O principal problema na utilização da FI igual a zero prende-se com a existência de sinais parasitas em redor de DC que são criados durante a conversão devido essencialmente a imperfeições dos misturadores [8][9]. A utilização de circuitos CMOS de rádio-frequência, cuja utilização é cada vez mais comum pois facilita a sua integração com os circuitos digitais, aumenta também o ruído *flicker* (ruído $1/f$) que é especialmente nocivo a baixas frequências [10].

Capítulo 3

Algoritmos Essenciais

Neste capítulo vamos abordar alguns dos algoritmos necessários à implementação de um sistema de rádio digital. Normalmente um algoritmo tem várias implementações possíveis. A mais intuitiva será usualmente a implementação equivalente ao seu homólogo analógico. Um amplificador (ou atenuador), por exemplo, pode ser implementado digitalmente como um multiplicador do sinal por uma constante. Em alguns casos particulares é possível simplificar significativamente o algoritmo. No caso do amplificador, se o ganho for uma potência de 2 (ou seja um múltiplo de ± 6 dB), o algoritmo pode ser implementado através de uma deslocação (*shift*) dos bits para a direita ou para a esquerda conforme o ganho seja respectivamente maior ou menor que um. Num PLD esta deslocação é feita simplesmente pelo encaminhamento (*routing*) dos bits de saída do andar anterior para os bits de entrada do andar seguinte. Poupa-se assim um multiplicador que é uma operação que ocupa muitos recursos do dispositivo. Este tipo de simplificações são muito importantes pois podem significar a diferença entre a possibilidade ou impossibilidade da realização prática de um sistema.

3.1 Oscilador

Nos sistemas de telecomunicações os osciladores têm presença obrigatória normalmente com a função de gerar a frequência dos osciladores locais. Nos sistemas analógicos os osciladores são tipicamente implementados utilizando um amplificador com realimentação positiva. O equivalente digital consiste na utilização de um filtro de resposta impulsional infinita (IIR) calculado de modo a ter um pólo no círculo unitário do plano-z. Um filtro deste tipo irá oscilar quando excitado por um impulso unitário.

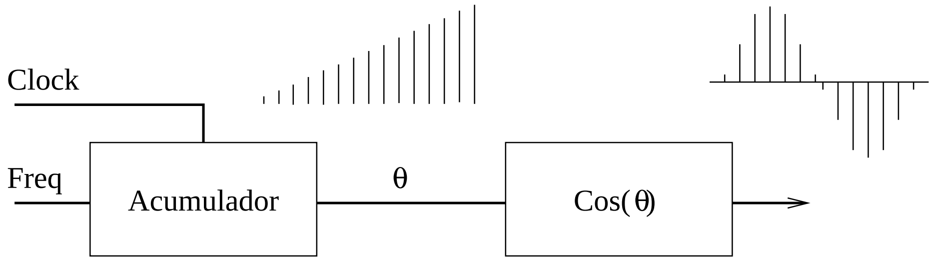


Figura 3.1: Oscilador de acumulação de fase.

Para mudar a frequência de oscilação é necessário alterar os coeficientes e reinicializar o filtro. Esta característica dificulta a utilização deste tipo de oscilador como modulador de frequência ou de fase. Esta é uma das razões porque este oscilador é pouco utilizado na prática. Não iremos dedicar mais tempo a este tipo de oscilador. Uma abordagem mais detalhada é feita no capítulo 6 da referência [1].

3.1.1 Método da acumulação de fase

O método mais popular para implementar osciladores digitais é o método da acumulação de fase. Este oscilador é também chamado de oscilador controlado numericamente (NCO) por comparação ao oscilador controlado por tensão (VCO) analógico. A figura 3.1 ilustra o seu funcionamento.

O funcionamento do NCO é bastante simples. Em cada impulso do relógio o acumulador coloca na sua saída a soma da entrada com o valor da saída anterior. Como o acumulador tem um número finito de bits chegará uma altura em que o valor máximo será ultrapassado (*overflow*) e o acumulador continuará a contar a partir de zero. O sinal na sua saída será então uma onda em dente de serra. Este sinal é depois entregue a um andar que calcula o seno ou coseno. A saída do acumulador é portanto a fase da sinusóide. Quanto maior for o valor presente na entrada do acumulador maiores serão os saltos da fase e em consequência mais elevada será a frequência da sinusóide.

Existem vários métodos para calcular o coseno. Vamos apenas mencionar o método da tabela (LUT) que consiste em guardar numa memória (ROM por exemplo) o valor do coseno em vários instantes e usar o valor presente na saída do acumulador como um índice nesta memória. Outros métodos são descritos nas referências [1] e [11].

Um diagrama de blocos dum NCO que utiliza uma LUT está representado na figura 3.2. Este NCO utiliza um acumulador com N bits. Destes N bits, os W (com $W \leq N$)

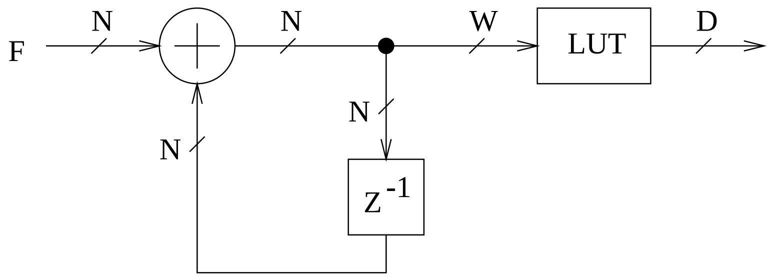


Figura 3.2: Diagrama de blocos dum NCO com LUT.

mais significativos são entregues à LUT. A resolução de saída da LUT (resolução em amplitude) é de D bits.

Como foi referido, o sinal presente na saída do acumulador representa a fase instantânea da sinusóide. Este sinal varia entre 0 e $2^N - 1$ que corresponde a uma variação da fase entre 0 a 2π ou 360°. O valor na entrada do acumulador (F) define o incremento da fase a cada impulso do relógio. Para $F = 1$ o acumulador irá passar por todos os valores possíveis desde 0 a $2^N - 1$ voltando então de novo a 0 e assim sucessivamente. O período do sinal de saída é portanto $T_o = T_{ck}2^N$ onde T_{ck} é o período do sinal do relógio. Este é o sinal de frequência mais baixa gerado pelo NCO. Se $F = 2$ o acumulador apenas tomará metade dos valores possíveis e percorrerá um ciclo completo em metade do tempo anterior. No caso geral, a frequência de saída é dada por

$$f_o = f_{ck}\frac{F}{2^N}\ \text{Hz} \tag{3.1}$$

em que $f_{ck} = 1/T_{ck}$ é a frequência do sinal de relógio. A equação anterior só é válida para valores de F compreendidos entre 0 e 2^{N-1}. Esta limitação pode ser compreendida através do Teorema da Amostragem. Este teorema afirma que um sinal tem de ser amostrado a mais do dobro da sua frequência máxima para que não exista ambiguidade no sinal amostrado. Para valores de F entre 2^{N-1} e $2^N - 1$ este critério é violado fazendo com que a frequência realmente gerada seja a mesma que utilizando $F' = 2^N - F$ ou seja $f'_o = f_{ck} - f_o$. Este efeito está exemplificado na figura 3.3 em que se supõe uma reconstrução perfeita da sinusóide. Na prática, devido ao número finito de bits de saída da LUT e a outras imperfeições, irão aparecer riscas indesejáveis no espectro. A área a claro da figura é a zona de interesse, sendo a zona a sombreado constituída por réplicas da área a claro que ocorrem devido ao sinal ser amostrado à frequência f_{ck}.

Outra maneira de verificar que F e F' geram realmente a mesma frequência é analisar a sequência de saída do acumulador. As sequências, representadas em binário,

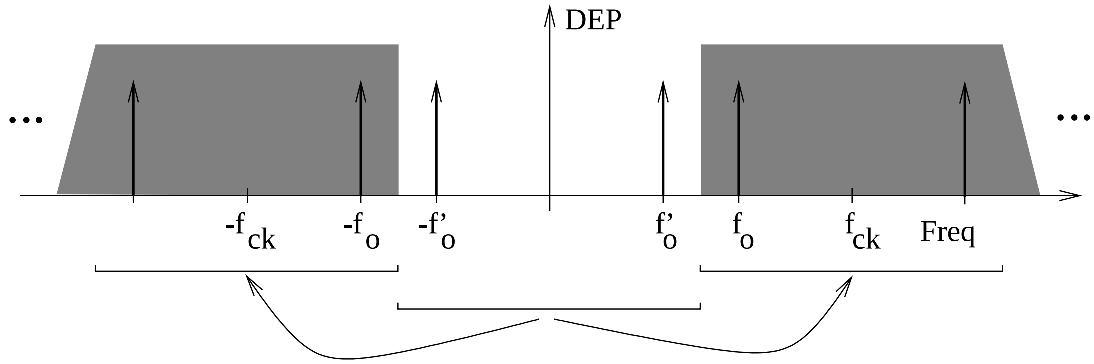


Figura 3.3: Espectro do sinal do NCO ideal.

```
F='001'  000 001 010 011 100 101 110 111 000 ...

F='111'  000 111 110 101 100 011 010 001 000 ...
```

Figura 3.4: Saída em binário do acumulador no caso $N = 3$ para $F = 1$ e $F = 7$.

para $F = 1$ e $F = 7$ estão ilustradas na figura 3.4 para o caso $N = 3$. Vê-se claramente que ambas as sequências necessitam de 8 ciclos de relógio para percorrer um ciclo completo pelo que ambas geram a frequência $f_o = f_{ck}/8$.

NCO de quadratura

Em certas aplicações é necessário a utilização de duas sinusóides em quadratura, ou seja um seno e um coseno. É o caso, por exemplo, da modulação de quadratura. É usual considerar o conjunto das duas sinusóides em quadratura como sendo um único sinal complexo representado como

$$\mathrm{e}^{\mathrm{j}\omega_o n T_{ck}} = \cos(\omega_o n T_{ck}) + \mathrm{j}\sin(\omega_o n T_{ck}) \tag{3.2}$$

O NCO pode ser facilmente adaptado para gerar ambas as componentes. Um dos métodos mais simples consiste em utilizar duas LUT's, uma para o seno e outra para o coseno. Outro método utiliza apenas uma LUT e faz duas consultas na tabela por cada amostra. A primeira consulta é feita no endereço que vem directamente do acumulador. A segunda consulta deverá estar desfasada de $\pi/2$ o que equivale a somar 2^{N-2} ao endereço do acumulador. Esta técnica necessita de um relógio que funcione ao dobro da frequência de amostragem.

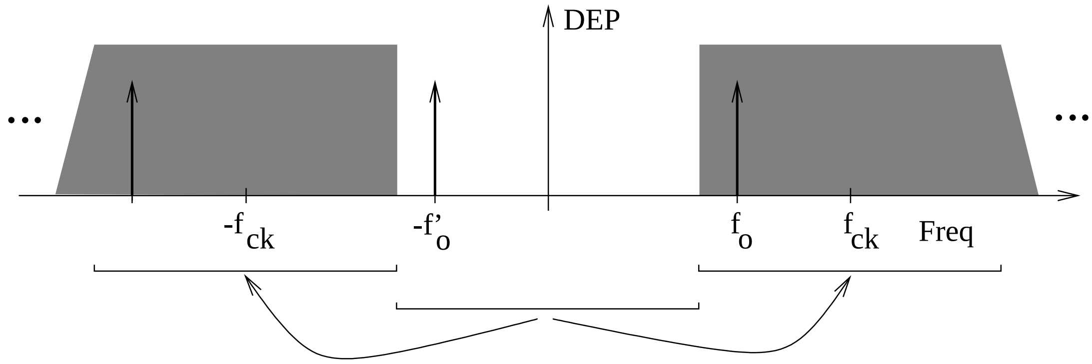


Figura 3.5: Espectro do sinal do NCO de quadratura.

Como foi referido no capítulo 2, as duas componentes do sinal de quadratura podem ser vistas como um único sinal complexo cujo espectro contém apenas uma risca numa frequência positiva ou negativa. Neste caso, valores de F entre 0 e 2^{N-1} geram um sinal cujo espectro contém uma risca numa frequência positiva dada por $f_o = f_{ck}\frac{F}{2^N}$, enquanto que os restantes valores de F geram um sinal cujo espectro contém uma risca numa frequência negativa dada por $f_o = -f_{ck}\frac{2^N-F}{2^N}$. Este resultado está ilustrado no espectro da figura 3.5.

Controlo de fase

Como vimos, a entrada do acumulador (F) define a frequência de saída do NCO. Esta entrada não necessita de ser constante, pode aliás ser alterada uma vez por cada impulso do relógio. Sempre que F é alterado, a sua alteração nota-se no impulso de relógio seguinte e sem descontinuidades na fase da sinusóide. Isto torna atractiva a utilização do NCO como modulador de frequência, e também em sistemas que mudam frequentemente a frequência da portadora, como é o caso dos sistemas de espalhamento de espectro por salto em frequência (*frequency hopping*). Com uma pequena modificação, esboçada na figura 3.6, é possível utilizar o NCO também para efectuar modulação de fase. O valor P é somado ao valor presente na saída do acumulador antes de ser entregue à LUT. Deste modo, o bit mais significativo de P representa 180°, o seguinte representa 90°, o próximo 45°, e assim sucessivamente. A resolução de fase, ou seja a diferença de fase mais pequena que o NCO consegue gerar é dada por $360/2^N$ graus. Note-se, no entanto, que se $W < N$ a resolução de fase de saída é condicionada pelo valor de W. Um modulador PSK, por exemplo, pode ser construído muito facilmente utilizando apenas um NCO. Uma vez que neste modulador as mudanças de fase são de 180°, basta utilizar o bit de dados a ser transmitido para controlar o bit mais

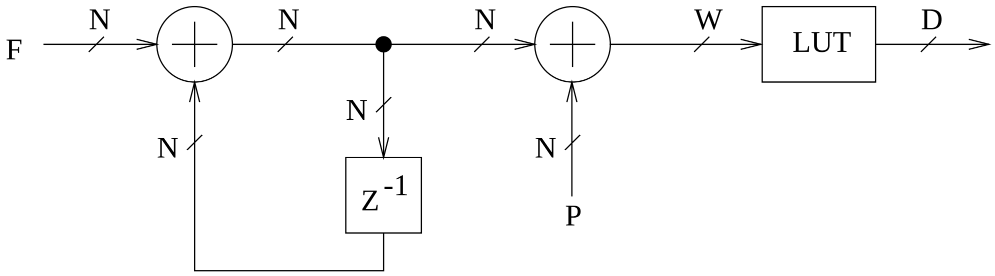


Figura 3.6: Diagrama de blocos do NCO com controlo de fase.

D\W	10	12	14	16
8	8.192	32.768	131.072	524.288
10	10.240	40.960	163.840	655.360
12	12.288	49.152	196.608	786.432

Tabela 3.1: Tamanho da LUT sem compressão, em bits.

significativo de P mantendo todos os outros bits a um nível constante.

Compressão da LUT

A função da LUT é efectuar a conversão da fase actual para o valor instantâneo da amplitude, ou seja, efectuar a transformação $\theta \rightarrow \sin(\theta)$. O método mais directo de construção da LUT consiste em guardar as 2^W amplitudes correspondentes às respectivas fases calculadas em intervalos equidistantes entre 0° e 360°. Cada uma das amplitude é guardada utilizando D bits. Assim sendo, a memória necessária para construir a LUT utiliza

$$M = 2^W D \text{ bits.} \tag{3.3}$$

A tabela 3.1 contém o tamanho necessário da memória para alguns valores de D e W.

Existem várias técnicas para reduzir o tamanho necessário da memória. Vamos mencionar uma delas de seguida. Outras técnicas encontram-se descritas na referência [11]. Se visualizarmos com atenção a figura 3.7 notamos que existem simetrias entre os quatro quadrantes da sinusóide. Note-se que os dois bits mais significativos (MSB) do endereço indicam qual o quadrante a que se refere esse endereço. Os 3º e 4º quadrantes são o espelho vertical dos 1º e 2º quadrantes. Matematicamente é o equivalente a afirmar que $\sin(\theta) = -\sin(180 + \theta)$. Sendo assim basta guardar na memória as amplitudes referentes aos primeiros dois quadrantes e utilizar o bit mais significa-

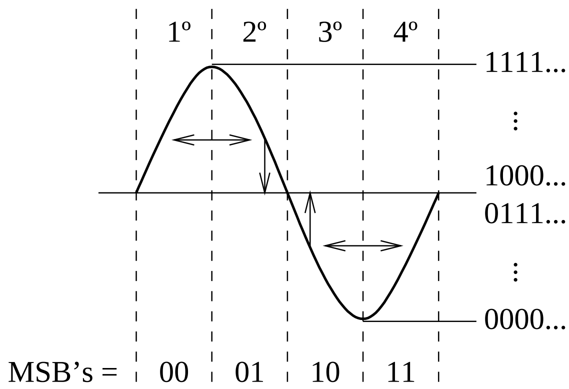


Figura 3.7: Simetria dos quadrantes da sinusóide.

tivo do endereço para decidir se se multiplica a saída por 1 ou por -1. Comprime-se assim a memória para metade do tamanho original. Se, como está indicado na figura, as amplitudes estiverem codificadas no sistema binário directo (também chamado de complemento-para-1), então a multiplicação por -1 efectua-se simplesmente pela negação (complemento) de todos os bits dessa amplitude. Se repararmos que no 1º e 2º quadrantes o bit mais significativo da amplitude não varia, então podemos guardar a amplitude com $D-1$ bits poupando mais um pouco de memória. Este bit pode também ser recuperado através do valor do bit mais significativo do endereço.

Além da simetria vertical que já identificámos, existem também simetrias horizontais entre o 1º e 2º quadrantes e entre o 3º e 4º quadrantes. Matematicamente estas simetrias derivam da relação $\sin(\theta) = \sin(90 - \theta)$. Podemos então comprimir por mais um factor dois o tamanho da tabela, se guardarmos na memória apenas o 1º quadrante e derivarmos os restantes a partir deste. Desta vez necessitamos de usar o segundo bit mais significativo do endereço para distinguir se estamos nos quadrantes pares ou nos quadrantes impares. Nos quadrantes impares tudo se processa como no caso anterior. Nos quadrantes pares é necessário proceder ao cálculo do endereço correspondente a $90-\theta$. Esse cálculo é efectuado simplesmente através da negação dos $W-2$ bits menos significativos (LSB) do endereço.

A figura 3.8 contém o diagrama de blocos que implementa os algoritmos de compressão descritos. O tamanho necessário da memória passa a ser dado por

$$M = 2^{W-2}(D-1) \text{ bits} \tag{3.4}$$

Os tamanhos da LUT após a compressão podem ser consultados na tabela 3.2 para comparação com a tabela anterior.

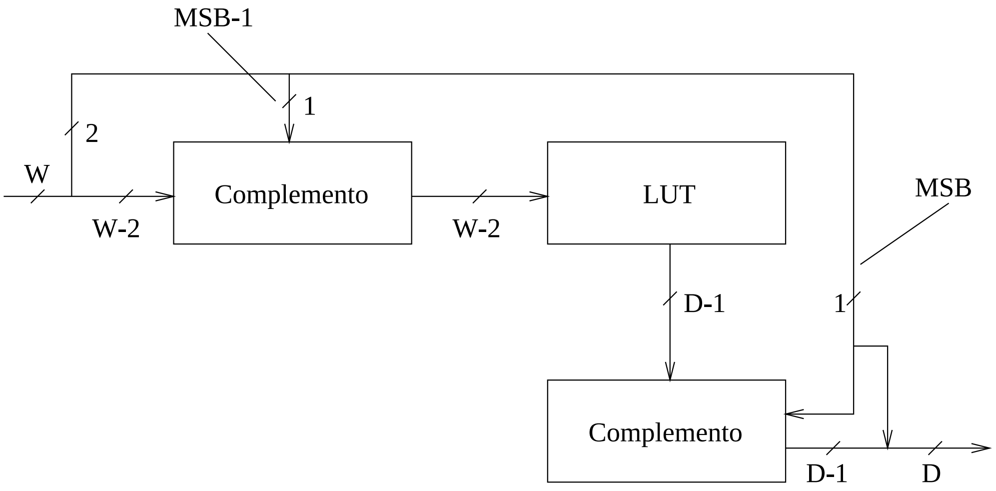


Figura 3.8: Algoritmo de compressão da LUT.

D\W	10	12	14	16
8	1.792	7.168	28.672	114.688
10	2.304	9.216	36.864	147.456
12	2.816	11.264	45.056	180.224

Tabela 3.2: Tamanho da LUT com compressão, em bits.

Em resumo, utilizando este algoritmo consegue-se diminuir a memória da LUT necessária para menos de um quarto do tamanho em troca da utilização de dois grupos de negadores, um de $D-1$ bits e o outro de $W-2$ bits.

3.2 Misturador

Os misturadores digitais têm a função, tal como os seus correspondentes analógicos, de converter o sinal de entrada que está centrado numa dada frequência de modo a que este sinal fique centrado numa outra frequência desejada. Existem essencialmente quatro tipos de conversão possíveis em consequência de existirem dois tipos de sinais, reais ou complexos, e existirem dois tipos de oscilador local, sinusóide real ou exponencial complexa. Vamos analisar estes quatro casos mais em detalhe.

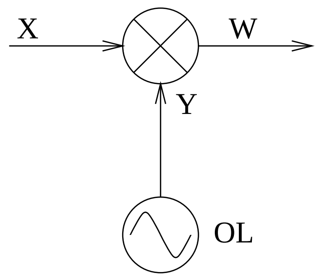


Figura 3.9: Misturador de sinal real com sinusóide.

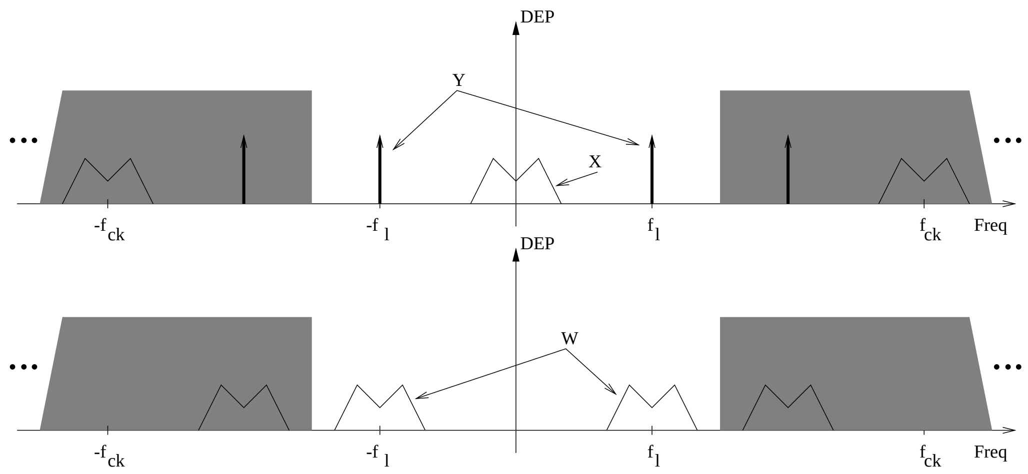


Figura 3.10: Espectro resultante da mistura de um sinal real com uma sinusóide.

Sinal real misturado com sinusóide real

Seja $x(nT_{ck})$ o sinal a ser convertido e $y(nT_{ck}) = \cos(2\pi f_l nT_{ck})$ o sinal proveniente do oscilador local. A mistura destes dois sinais resulta no sinal

$$\begin{aligned} w(nT_{ck}) &= x(nT_{ck})y(nT_{ck}) \\ &= x(nT_{ck})\cos(2\pi f_l nT_{ck}) \end{aligned}$$

A implementacão deste misturador é portanto um simples multiplicador como está esboçado na figura 3.9. O espectro resultante da mistura está exemplificado na figura 3.10.

Sinal real misturado com exponencial complexa

Neste caso, o sinal do oscilador local é dado por $y(t) = \mathrm{e}^{\mathrm{j}2\pi f_l n T_{ck}} = \cos(2\pi f_l n T_{ck}) + \mathrm{j}\sin(2\pi f_l n T_{ck})$ pelo que a mistura rege-se pela equação

$$\begin{aligned} w(nT_{ck}) &= x(nT_{ck})y(nT_{ck}) \\ &= x(nT_{ck})[\cos(2\pi f_l n T_{ck}) + \mathrm{j}\sin(2\pi f_l n T_{ck})] \\ &= x(nT_{ck})\cos(2\pi f_l n T_{ck}) + \mathrm{j}x(nT_{ck})\sin(2\pi f_l n T_{ck}) \\ &= I(nT_{ck}) + \mathrm{j}Q(nT_{ck}) \end{aligned}$$

onde

$$\begin{aligned} I(nT_{ck}) &= x(nT_{ck})\cos(2\pi f_l n T_{ck}) \qquad \text{e} \\ Q(nT_{ck}) &= x(nT_{ck})\sin(2\pi f_l n T_{ck}) \end{aligned}$$

Este misturador é constituído por dois multiplicadores como está ilustrado na figura 3.11. A figura 3.12 contém um exemplo de mistura entre um sinal real e uma exponencial complexa de frequência positiva.

Sinal complexo misturado com sinusóide real

Este caso é semelhante ao anterior pois consiste na mistura de um sinal real por outro complexo. Sendo $x(nT_{ck}) = I_1(nT_{ck}) + \mathrm{j}Q_1(nT_{ck})$ então

$$\begin{aligned} w(nT_{ck}) &= x(nT_{ck})y(nT_{ck}) \\ &= [I_1(nT_{ck}) + \mathrm{j}Q_1(nT_{ck})]\cos(2\pi f_l n T_{ck}) \\ &= I_1(nT_{ck})\cos(2\pi f_l n T_{ck}) + \mathrm{j}Q_1(nT_{ck})\cos(2\pi f_l n T_{ck}) \\ &= I_2(nT_{ck}) + \mathrm{j}Q_2(nT_{ck}) \end{aligned}$$

onde

$$\begin{aligned} I_2(nT_{ck}) &= I_1(nT_{ck})\cos(2\pi f_l n T_{ck}) \qquad \text{e} \\ Q_2(nT_{ck}) &= Q_1(nT_{ck})\cos(2\pi f_l n T_{ck}) \end{aligned}$$

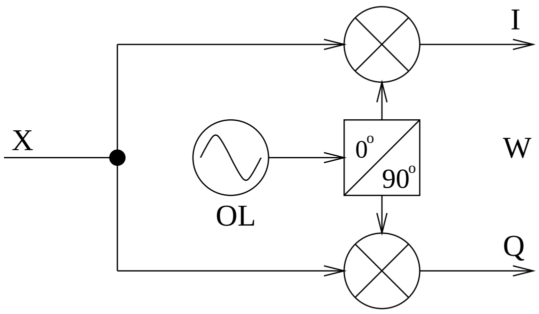


Figura 3.11: Misturador de sinal real com exponencial complexa.

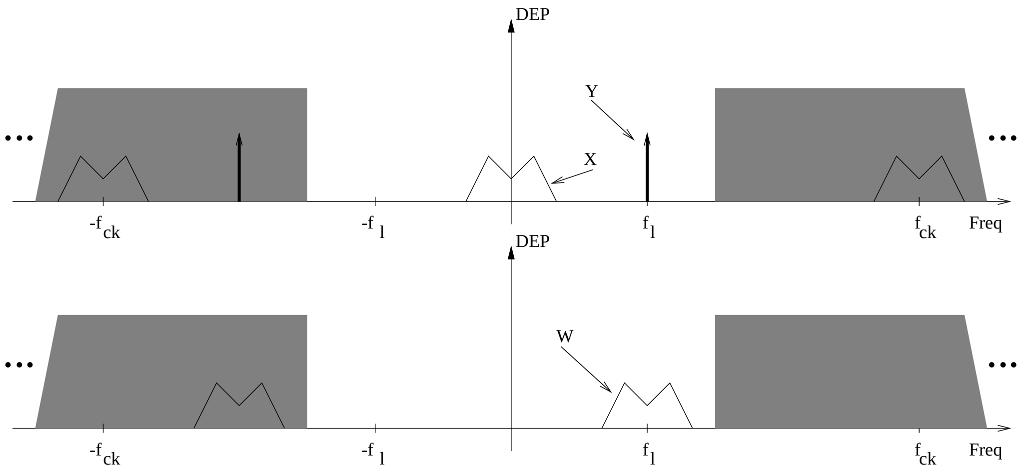


Figura 3.12: Espectro resultante da mistura de um sinal real com uma exponencial complexa.

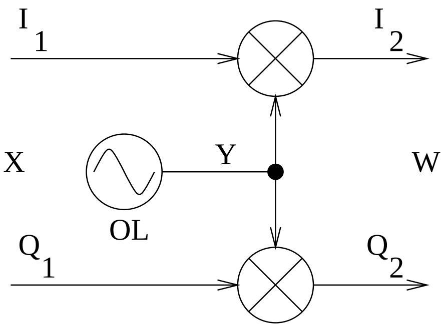


Figura 3.13: Misturador de sinal complexo com sinusóide.

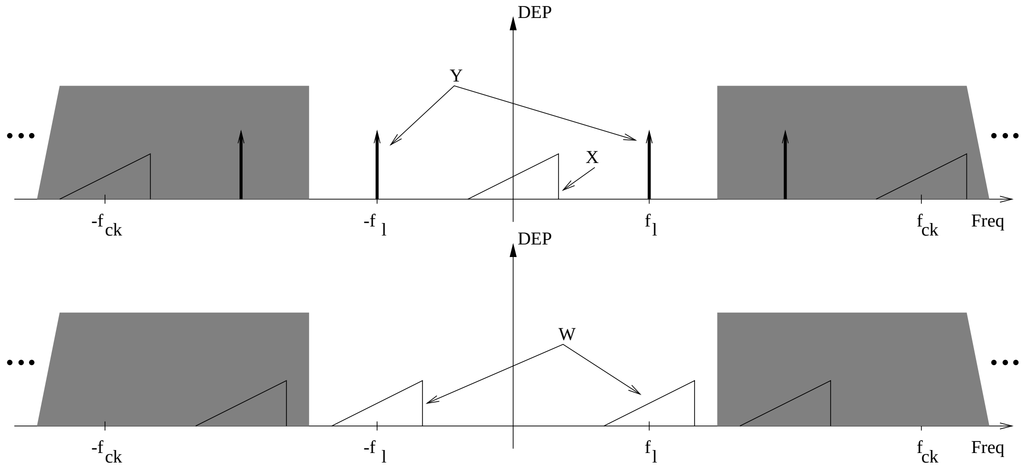


Figura 3.14: Espectro resultante da mistura de um sinal complexo com uma sinusóide.

Esta mistura, tal como a anterior, é implementada utilizando um par de multiplicadores como se pode observar na figura 3.13. O espectro resultante da mistura está exemplificado na figura 3.14.

Sinal complexo misturado com exponencial complexa

Neste último caso temos o produto de dois sinais complexos. Tal como no caso anterior $x(nT_{ck}) = I_1(nT_{ck}) + \mathrm{j}Q_1(nT_{ck})$, então

$$\begin{aligned}
w(nT_{ck}) &= x(nT_{ck})y(nT_{ck}) \\
&= [I_1(nT_{ck}) + \mathrm{j}Q_1(nT_{ck})][\cos(2\pi f_l nT_{ck}) + \mathrm{j}\sin(2\pi f_l nT_{ck})] \\
&= [I_1(nT_{ck})\cos(2\pi f_l nT_{ck}) - Q_1(nT_{ck})\sin(2\pi f_l nT_{ck})] + \\
&\quad \mathrm{j}[I_1(nT_{ck})\sin(2\pi f_l nT_{ck}) + Q_1(nT_{ck})\cos(2\pi f_l nT_{ck})] \\
&= I_2(nT_{ck}) + \mathrm{j}Q_2(nT_{ck})
\end{aligned}$$

onde

$$\begin{aligned} I_2(nT_{ck}) &= I_1(nT_{ck})\cos(2\pi f_l nT_{ck}) - Q_1(nT_{ck})\sin(2\pi f_l nT_{ck}) \qquad \text{e} \\ Q_2(nT_{ck}) &= I_1(nT_{ck})\sin(2\pi f_l nT_{ck}) + Q_1(nT_{ck})\cos(2\pi f_l nT_{ck}) \end{aligned}$$

Para implementar este misturador são necessários quatro multiplicadores e dois somadores, sendo um deles usado como subtractor tal como está ilustrado na figura 3.15. A figura 3.16 contém um exemplo de mistura entre um sinal complexo e uma exponencial complexa de frequência positiva.

3.2.1 Mistura sem multiplicadores

Existe um caso particular em que se consegue reduzir significativamente os recursos necessários à implementação de um misturador. Essa redução deve-se ao facto de não ser necessária a utilização de multiplicadores. Tal é possível se a frequência do oscilador local (f_l) for constante e igual a um quarto da frequência de amostragem (f_{ck}). Neste caso, o sinal seno e/ou o sinal coseno do oscilador local tomam apenas os valores 0, 1 e -1 como se pode observar na figura 3.17. A sequência do coseno é a seguinte

$$1, 0, -1, 0, ...$$

e a sequência do seno é dada por

$$0, 1, 0, -1, ...$$

Todos os quatro casos apresentados anteriormente podem utilizar esta técnica. Vamos ver o caso do misturador de um sinal complexo por uma exponencial complexa, uma vez que os outros casos podem ser entendidos como casos particulares deste. Como vimos, a equação que descreve este misturador é a seguinte:

$$w(nT_{ck}) = I_2(nT_{ck}) + \mathrm{j}Q_2(nT_{ck})$$

onde

$$\begin{aligned} I_2(nT_{ck}) &= I_1(nT_{ck})\cos(2\pi f_l nT_{ck}) - Q_1(nT_{ck})\sin(2\pi f_l nT_{ck}) \qquad \text{e} \\ Q_2(nT_{ck}) &= I_1(nT_{ck})\sin(2\pi f_l nT_{ck}) + Q_1(nT_{ck})\cos(2\pi f_l nT_{ck}) \end{aligned}$$

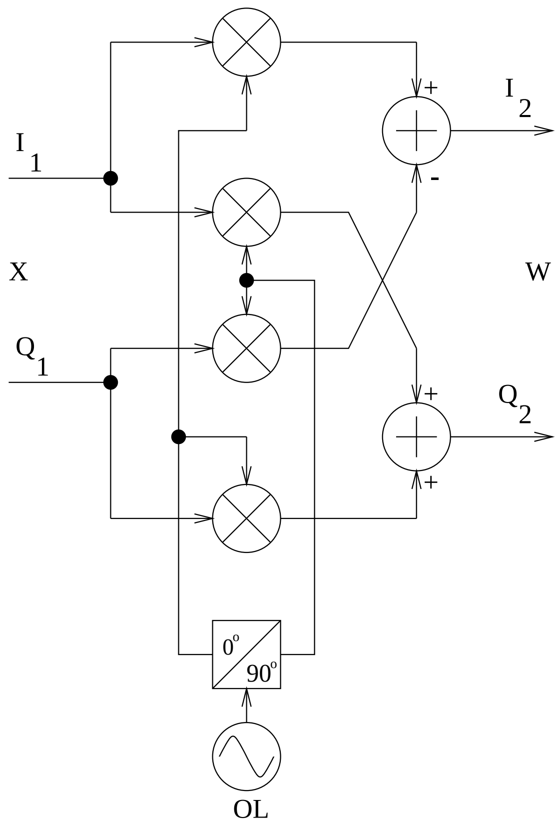


Figura 3.15: Misturador de sinal complexo com exponencial complexa.

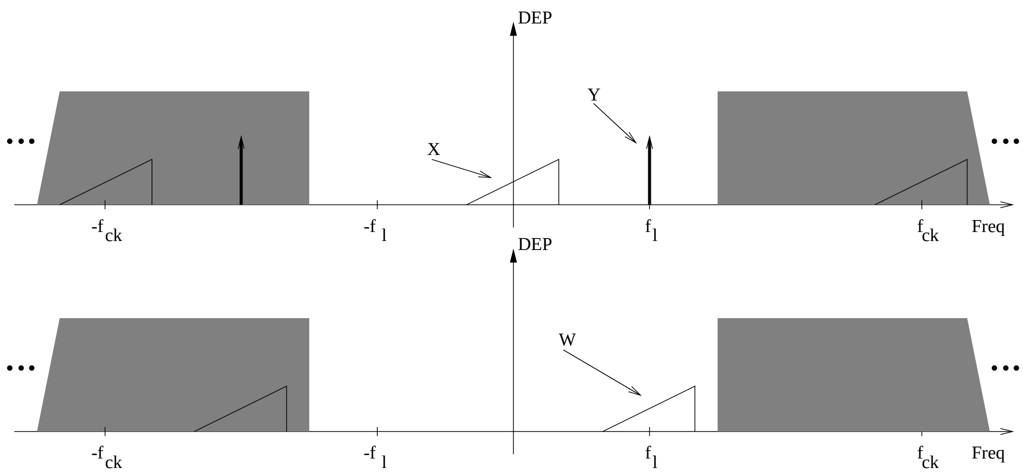


Figura 3.16: Espectro resultante da mistura de um sinal complexo com uma exponencial complexa.

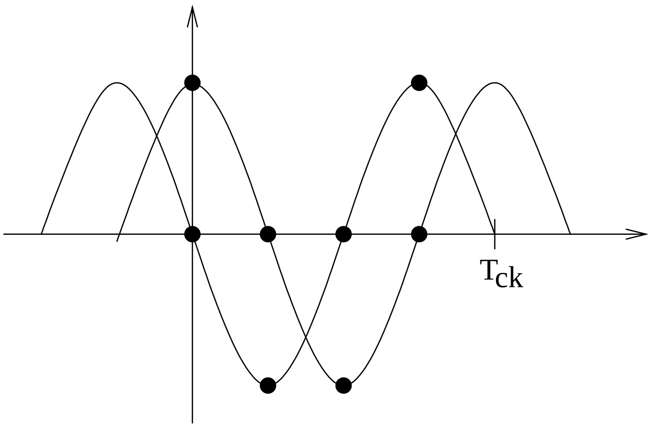


Figura 3.17: Instantes de amostragem quando $f_l = f_{ck}/4$.

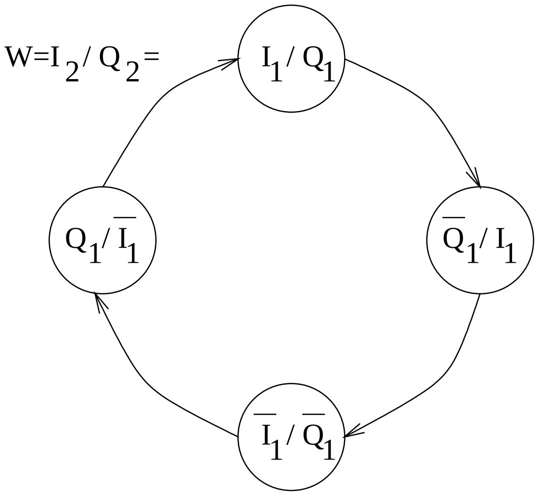


Figura 3.18: Diagrama de estados do misturador para o caso $f_l = f_{ck}/4$.

Uma vez que o seno e o coseno estão condicionados aos valores mencionados, podemos afirmar que as sequências I_2 e Q_2 são as seguintes:

$$I_2 = I_1(0), -Q_1(1), -I_1(2), Q_1(3), ...$$

$$Q_2 = Q_1(0), I_1(1), -Q_1(2), -I_1(3), ...$$

O misturador pode ser então implementado através de uma máquina de estados que percorre em sequência os quatro estados possíveis. O diagrama de estados está representado na figura 3.18. Note-se que não só não são utilizados multiplicadores como não é utilizado o oscilador local pelo que os recursos poupados são muito significativos.

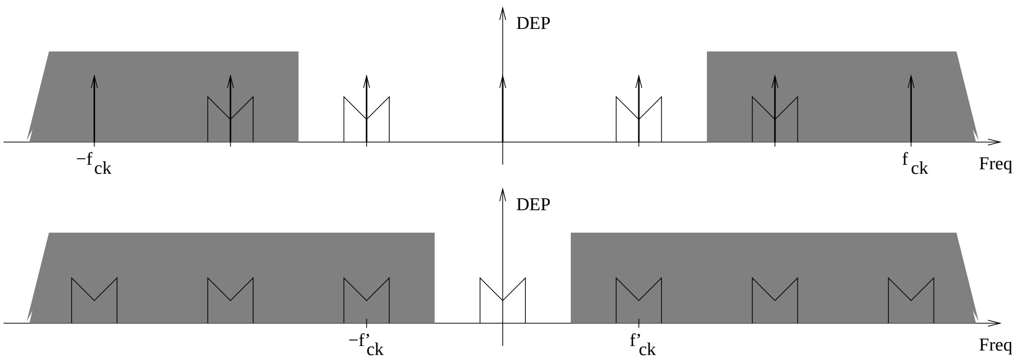


Figura 3.19: Espectro resultante da decimação com $R = 3$.

3.2.2 Mistura por decimação

A decimação é o processo pelo qual a frequência de amostragem de um sinal é modificada para outra de valor inferior. Normalmente o quociente, R, entre as duas frequências é um número inteiro pelo que a decimação consiste em aproveitar apenas uma em cada R amostras. Antes de proceder à decimação é necessário filtrar o sinal de modo a garantir que não irá ocorrer *aliasing*. O processo de decimação é equivalente à multiplicação por uma sequência de impulsos delta espaçados de RT_{ck} segundos, seguido da redução da frequência de amostragem pelo factor R. Na parte superior da figura 3.19 temos o espectro do sinal de entrada centrado em $f_{ck}/3$, bem como o espectro da sequência de impulsos delta que é ele próprio uma sequência de impulsos delta espaçados de $f_{ck}/3$. O resultado da decimação está ilustrado em baixo onde se pode observar que o sinal foi convertido para banda base. A nova frequência de amostragem está representada como f'_{ck}.

Capítulo 4

Implementação Prática

Neste capítulo será descrita a plataforma, implementada no âmbito deste trabalho, que visa facilitar o desenvolvimento, a simulação e o teste de sistemas de rádio digital. Mais especificamente pretende-se que a plataforma desenvolvida incentive a implementação, em projectos futuros, de novos algoritmos com utilidade em sistemas de rádio digital, com vista à criação de uma biblioteca de algoritmos genéricos semelhante às utilizadas nas linguagens de programação de computadores.

Esta plataforma pode ser dividida em três partes. A primeira parte engloba o circuito electrónico contendo o dispositivo lógico programável e é puramente do domínio do *hardware*. A segunda parte abarca os algoritmos digitais que implementam os diversos blocos necessários à realização de sistemas de rádio digital, nomeadamente os abordados no capítulo anterior. Tradicionalmente os algoritmos digitais eram implementados usando circuitos digitais discretos sendo portanto descritos através do seu esquema eléctrico. No entanto, por razões que serão apresentadas mais tarde, neste trabalho optou-se por usar uma linguagem de descrição de circuitos (HDL). Esta parte pode portanto ser considerada como um misto de *hardware* e *software*. A terceira parte, totalmente do domínio do *software*, engloba o programa que corre no computador pessoal (PC) e que permite a comunicação do PC com a placa que contém o dispositivo lógico programável.

Devido a limitações de tempo e a outras condicionantes, apenas foi possível implementar as secções de modulação/emissão. No entanto o sistema foi desenvolvido a pensar na presença da secção de desmodulação/recepção pelo que a sua futura inclusão não deverá apresentar grandes dificuldades.

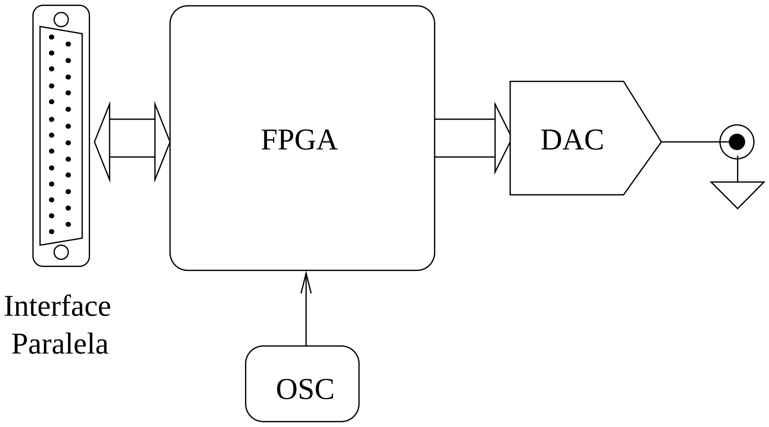


Figura 4.1: Diagrama de blocos do circuito electrónico.

As próximas três secções deste capítulo serão dedicadas à apresentação das três partes em que dividimos este trabalho, o *hardware*, os algoritmos de rádio digital, e o *software* do PC.

4.1 O Hardware

Esta secção descreve o circuito electrónico que serve de base à plataforma. O esquema de blocos deste circuito está representado na figura 4.1. Em destaque, ao centro, encontra-se a FPGA. Em baixo está o oscilador de onda quadrada que fornece o relógio que sincroniza todos os circuitos internos. Do lado direito encontra-se o conversor digital-analógico que disponibiliza o resultado da conversão num conector. Por último, do lado esquerdo, um conector tipo D de 25 pinos permite a ligação do circuito à porta paralela de um vulgar computador pessoal. Os blocos de programação da FPGA e de alimentação não estão representados para não sobrecarregar a figura.

O esquema eléctrico completo encontra-se no apêndice A. O circuito em si é muito simples uma vez que quase toda a funcionalidade é implementada pela FPGA. O esquema consiste, portanto, apenas na interface da FPGA com o DAC e com a porta paralela do PC. De salientar também a existência de uma ficha para a programação da FPGA. Note-se que esta FPGA é baseada em RAM estática pelo que é necessária a sua programação sempre que se liga o circuito.

O coração do circuito é uma FPGA FLEX 10K10 fabricada pela *Altera* [12]. Este dispositivo contém o equivalente a 10.000 portas-lógicas (*gates*) sendo parte delas utilizadas para implementar a memória RAM interna de 6.144 bits [13]. A opção pela

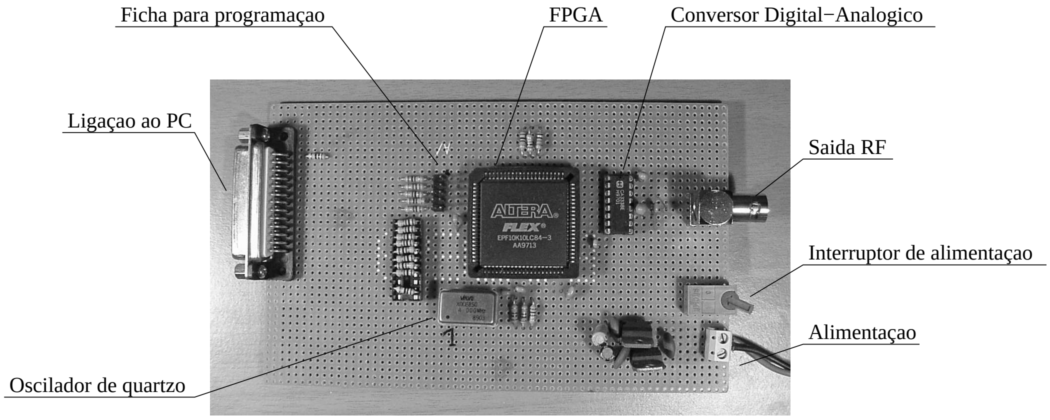


Figura 4.2: Fotografia do circuito.

utilização de um dispositivo da *Altera* foi condicionada pelo facto deste ser o único fabricante cujo *software* estava disponível tanto na Universidade de Aveiro bem como na *Fachhochschule Kiel* onde foi efectuado parte do trabalho. A utilização da versão 10K10 (a mais pequena em termos de recursos da série FLEX) deveu-se ao facto de ser a única com um encapsulamento (PLCC) que permite uma montagem fácil utilizando um suporte (*socket*) de uso comum. Apesar das suas modestas capacidades, este dispositivo adequa-se bem às necessidades deste trabalho. Quanto ao conversor digital-analógico optou-se por utilizar o CA3338 da *Intersil* anteriormente conhecida por *Harris Semiconductor* [14]. Este é um conversor de 8 bits de resolução que permite uma taxa de conversão de 50 MHz [15]. A facilidade de obtenção e o encapsulamento DIP foram as razões principais que conduziram à escolha deste componente. Optou-se por não incluir os filtros analógicos de reconstrução/interpolação uma vez que estes têm características distintas de aplicação para aplicação. Quanto à comunicação entre o PC e a placa, decidiu-se pela utilização da porta paralela uma vez que esta interface existe em todos os computadores pessoais e é de fácil implementação.

O circuito foi montado numa placa perfurada tipo *wire wrapping* mas optou-se por soldar os fios condutores aos componentes de modo a diminuir as capacidades parasitas. A figura 4.2 apresenta uma fotografia da placa depois de montada.

4.2 Os Algoritmos

Um dos objectivos deste trabalho era que os algoritmos desenvolvidos fossem o mais possível portáveis, ou seja, que fossem utilizáveis no maior número de dispositivos, de diversos fabricantes e com arquitecturas distintas, sem serem necessárias modificações. A utilização dos editores gráficos, onde é desenhado o esquema eléctrico, foi posta de parte uma vez que normalmente os ficheiros resultantes utilizam formatos proprietários e são incompatíveis entre diferentes fabricantes. A solução escolhida foi a utilização de uma linguagem de descrição de circuitos, mais especificamente a linguagem VHDL. Esta linguagem foi adoptada como uma norma de indústria pelo IEEE em 1987 e actualizada em 1993. As ferramentas de *software* dos principais fabricantes suportam a linguagem VHDL pelo que a portabilidade está em princípio assegurada. Além da portabilidade esta linguagem tem outras vantagens, tal como a possibilidade de descrição de circuitos lógicos complexos através de código sucinto e simples, e da organização do código em níveis hierárquicos. A linguagem VHDL é descrita em inúmeros livros nomeadamente nas referências [16][17].

Uma das características mais poderosas da linguagem VHDL é a possibilidade de implementação de *componentes genéricos*. A melhor maneira de explicar o que é um componente genérico é através de um exemplo. Vamos supor que se pretende implementar um somador completo (*full-adder*) de 8 bits. Este componente terá então duas entradas de 8 bits para os operandos, uma entrada de transporte (*carry*), uma saída de 8 bits com o resultado da soma, e uma saída com o transporte. Se mais tarde necessitarmos de um somador completo de 16 bits será necessário implementar outro componente que é em tudo semelhante ao primeiro excepto no número de bits. Utilizando as potencialidades da linguagem VHDL podemos escrever um único componente genérico em que o número de bits é passado como parâmetro.

Em 1993 foi criada uma biblioteca de componentes genéricos básicos, a que foi dado o nome de Biblioteca de Módulos Parametrizados (LPM), que foi aceite como uma norma pela Associação das Indústrias Electrónicas (EIA) em complemento da norma EDIF [18][19]. Esta biblioteca pretende ser independente da arquitectura e do fabricante e é composta por componentes de uso geral tal como somadores, comparadores e contadores. Nos algoritmos que desenvolvemos no âmbito deste trabalho foram utilizados alguns dos componentes genéricos da biblioteca LPM.

De seguida vamos descrever um por um os algoritmos implementados. O código VHDL de todos estes algoritmos encontra-se no apêndice B. Parte deste código foi desenvolvido durante a permanência do autor na *Fachhochschule Kiel* o que explica os comentários ao código estarem em língua inglesa. Para manter a coerência optou-se por também comentar em inglês o código desenvolvido na Universidade de Aveiro. É de referir ainda que o programa utilizado para o desenvolvimento foi o MAXPLUS+ II

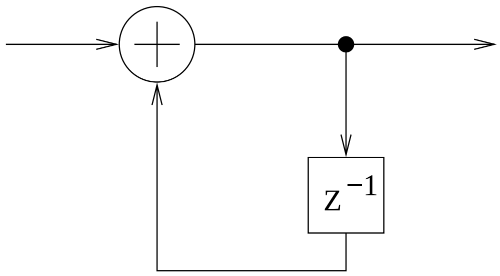


Figura 4.3: Diagrama de blocos do acumulador.

da *Altera* e que foi seguida a norma VHDL'93.

4.2.1 Acumulador

Foi necessário implementar um acumulador pois este é essencial para o funcionamento do NCO. O acumulador é, aliás, apenas utilizado no NCO. Poderia-se ter implementado o acumulador como parte integrante do NCO mas optou-se por desenvolver um módulo independente uma vez que o acumulador é um componente de uso comum e poderá ser útil para a implementação de futuros algoritmos.

O diagrama de blocos do acumulador está ilustrado na figura 4.3. O seu funcionamento é muito simples. A saída actual contém a soma da entrada actual com a saída anterior. Assim o valor presente na saída vai aumentando (acumulando) progressivamente em função do valor na entrada. O módulo em VHDL, que se encontra na secção B.1, é a implementação deste mesmo algoritmo. A *entity* deste módulo, ou seja a descrição da interface que o módulo apresenta para o exterior, é a seguinte:

```
ENTITY acumul IS
  GENERIC (acc_size: natural := 4);
  PORT(acc_clk: IN std_logic;
       acc_input: IN std_logic_vector(acc_size-1 downto 0);
       acc_output: OUT std_logic_vector(acc_size-1 downto 0));
END acumul;
```

Este módulo tira partido das potencialidades de criação de componentes genéricos do VHDL. O parâmetro `acc_size`, que toma por omissão o valor 4, pode tomar qualquer outro valor determinado pelo módulo que instancia o acumulador. Neste caso concreto este parâmetro define o número de bits dos sinais de entrada e saída e, naturalmente, do

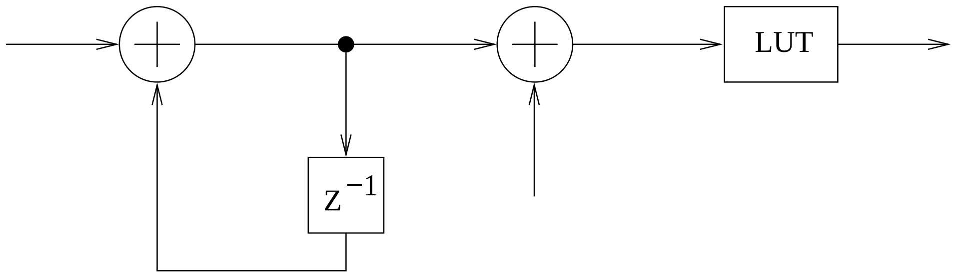


Figura 4.4: Diagrama de blocos do NCO.

somador interno. A *entity* declara ainda que este módulo contém três portos de interface com o exterior. O porto `acc_clk` é a entrada do sinal de relógio do acumulador. Em cada transição ascendente deste sinal é alterado o valor presente no porto `acc_output` que passa a conter a soma de si próprio com o valor presente no porto `acc_input`. Se a soma ultrapassar o valor máximo que se pode representar com `acc_size` bits, o resultado será truncado.

4.2.2 NCO

O método utilizado para implementar o Oscilador Controlado Numericamente (NCO) foi o método da acumulação de fase descrito em pormenor no capítulo anterior. Por conveniência repetimos na figura 4.4 o diagrama de blocos que ilustra este algoritmo. Como se pode observar, o NCO é constituído por um acumulador seguido de um somador e de uma LUT. Este algoritmo, cujo código se encontra na secção B.2, foi também implementado como um componente genérico, como se pode confirmar na *entity* seguinte:

```
ENTITY nco IS
  GENERIC (acc_size: positive := 24;
           lut_addr_size: positive := 9;
           lut_out_size: positive := 8;
           phase_size: positive := 1);
  PORT(nco_clk: IN std_logic;
       freq: IN std_logic_vector(acc_size-1 downto 0);
       phase: IN std_logic_vector(phase_size-1 downto 0));
       nco_output: OUT std_logic_vector(lut_out_size-1 downto 0);
       period: OUT std_logic;
END nco;
```

O parâmetro acc_size define o número de bits do acumulador, como foi referido na secção anterior. A resolução de frequência do NCO está dependente deste parâmetro. O número de bits da linha de endereços da LUT é dado por lut_addr_size e o número de bits de saída (que define a resolução em amplitude) é dado por lut_out_size. Não foi implementado nenhum mecanismo de compressão da LUT. A sinusóide guardada na LUT será portanto dividida em $2^{\texttt{lut\_addr\_size}}$ amostras, cada uma podendo tomar uma das $2^{\texttt{lut\_out\_size}}$ amplitudes possíveis. O último parâmetro, phase_size, especifica quantos bits serão utilizados para o controlo da fase. A entrada de modulação de fase é somada aos bits mais significativos do endereço proveniente do acumulador. Logo, se se pretender utilizar o NCO como modulador PSK deve-se definir phase_size igual a 1 uma vez que o bit mais significativo da linha de endereços corresponde a um desfasamento de 180°. Para a modulação QPSK, que necessita de desfasamentos de 90°, seria necessário definir phase_size igual a 2.

Quanto aos portos de entrada/saída temos o nco_clock que é a entrada do sinal de relógio à frequência de amostragem. A entrada freq, que internamente está ligado à entrada do acumulador, permite a escolha da frequência do sinal de saída. Esta frequência, f_o é dada por $f_o = f_{ck} \frac{\texttt{freq}}{2^{\texttt{acc\_size}}}$ onde f_{ck} é a frequência do sinal de relógio nco_clock. A frequência pode ser alterada uma vez em cada impulso de relógio. A modulação de fase mencionada anteriormente é controlada pela entrada phase ficando a saída do NCO disponível na saída nco_output. É ainda disponibilizado outro sinal de saída, period, que é um sinal periódico à frequência f_o cuja transição descendente coincide com o início de cada período do sinal de saída. Este sinal não faz normalmente parte de um NCO mas optou-se pela sua inclusão pois poderá ser útil em alguns casos.

A implementação do NCO toma ainda partido da criação condicional de circuitos utilizando a declaração if generate do VHDL. Esta propriedade do VHDL é utilizada em duas situações. Na primeira é verificado o valor de phase_size e caso este seja igual a zero, o que significa que não se pretende aplicar modulação de fase, então não é gerado o somador uma vez que este só é necessário quando existe controlo de fase. Na segunda situação verifica-se o valor de lut_out_size e caso seja igual a 1 prescinde-se de gerar a LUT, uma vez que neste caso a saída do NCO será binária. Ambas as situações resultam em poupança de recursos do dispositivo.

De referir ainda que o somador foi implementado utilizando o componente lpm_add_sub e a LUT utilizando o componente lpm_rom da biblioteca LPM. A utilização deste último componente implica que o NCO apenas possa ser utilizado em dispositivos que possuem memória interna, como é a caso do FLEX 10K10 que utilizá-

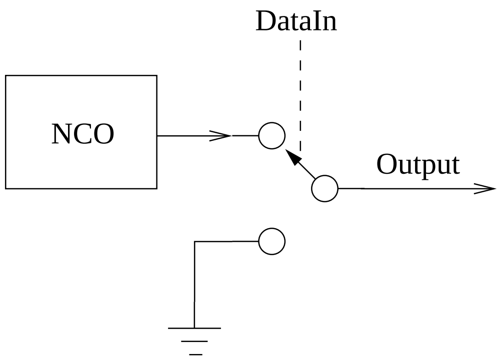


Figura 4.5: Diagrama de blocos do modulador ASK.

mos. Os conteúdos desta memória, ou seja as amostras da onda, têm de estar presentes num ficheiro chamado `wave.hex` no formato INTEL-HEX [20]. Para facilitar a criação deste ficheiro foi desenvolvido um pequeno programa que deverá ser invocado a partir dos programas *Octave* [21] ou *Matlab* [22]. O programa, chamado `intelhex.m`, tem como entrada um vector cujos elementos são as amostras da onda (usualmente uma sinusóide) que têm de estar compreendidos entre 0 e 255. Durante a execução do programa é solicitado o nome do ficheiro de saída onde será gerado o formato INTEL-HEX correspondente ao vector de entrada. O ficheiro de saída deverá então ser copiado para o directório de trabalho.

4.2.3 Modulador ASK

A modulação ASK, também conhecida por OOK (On-Off Keying) ou modulação por tudo-ou-nada, é uma modulação digital em que a portadora ou é transmitida com amplitude máxima ou não é transmitida, consoante o sinal binário a transmitir for "1" ou "0" respectivamente. Uma abordagem detalhada deste e doutros tipos de modulação pode ser encontrado nas referências [23][24][25][26]. O modulador ASK implementado não pretende ser um modulador de uso geral mas apenas um exemplo de como se poderá utilizar o NCO apresentado na secção anterior para implementar este tipo de modulador. A figura 4.5 ilustra o processo utilizado na implementação que é extremamente simples. O sinal binário a transmitir é utilizado para controlar o interruptor que comuta em função deste sinal.

A implementação em VHDL encontra-se na secção B.3 e é bastante concisa, como seria de esperar. A sua *entity* é a seguinte:

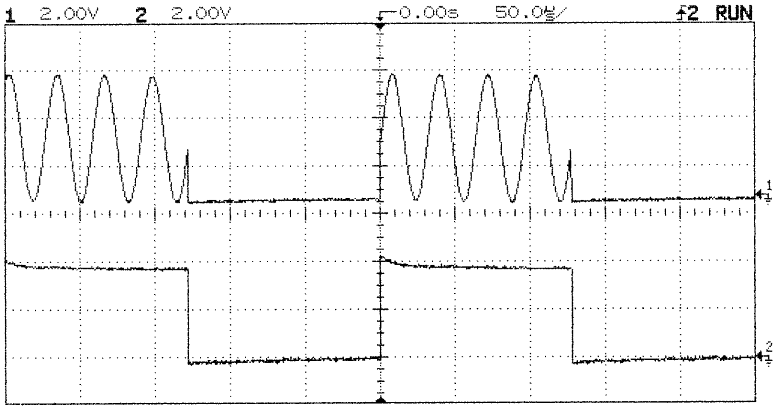


Figura 4.6: Sinal modulado em ASK.

```
ENTITY ask_mod IS
  PORT(ask_clk: IN std_logic;
       datain: IN std_logic;
       ask_output: OUT std_logic_vector(7 downto 0));
END ask_mod;
```

O porto `ask_clk` é a entrada de relógio e deverá funcionar à frequência de amostragem. O sinal binário a ser modulado deverá ser aplicado no porto `datain` saindo o sinal modulado no porto `ask_output`.

Para comprovar o bom funcionamento dos algoritmos apresentados, bem como da placa que contém o dispositivo lógico programável, foi realizado um teste simples. Utilizando o cabo de programação *Byte-blaster* foi programada a FPGA com este algoritmo. Anteriormente tinha já sido colocado um oscilador a cristal de 4 MHz para servir como referência de relógio. Com os valores presentes no código VHDL a frequência da portadora será 128 vezes menor que a frequência do relógio, ou seja de 31.250 Hz. Foi utilizado um acumulador com 24 bits, sendo o número de bits de endereços da LUT igual a 9 e a resolução de saída da LUT igual a 8 bits, ou seja a mesma resolução do DAC. O sinal binário de entrada foi gerado com o auxílio de um gerador de funções que foi ligado directamente a um pino da FPGA. Este sinal consistia numa onda quadrada a uma frequência 8 vezes menor que a frequência da portadora. O sinal medido à saída do conversor digital-analógico encontra-se esboçado na parte superior da figura 4.6 e corresponde ao sinal esperado. Na parte inferior encontra-se o sinal proveniente do gerador de funções. Posteriormente foi substituído o oscilador a cristal por um de 80 MHz tendo o circuito mantido o mesmo comportamento.

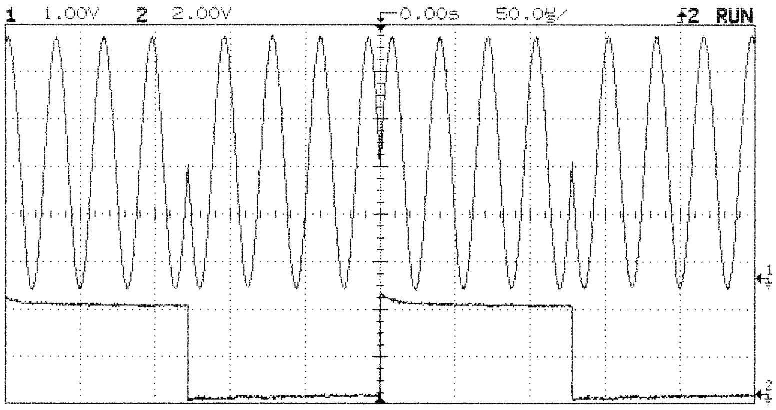


Figura 4.7: Sinal modulado em PSK.

4.2.4 Modulador PSK

A modulação PSK é uma modulação digital onde a fase da portadora é desfasada de 180° sempre que existe uma mudança no sinal binário de entrada. O modulador PSK aqui apresentado é mais um exemplo da utilização do NCO. O código VHDL encontra-se na secção B.4 e é ainda mais simples que o código do modulador ASK uma vez que o NCO já suporta ele próprio a modulação de fase. A *entity* deste modulador é em tudo semelhante à anterior pelo que a função de cada um dos portos dispensa comentários:

```
ENTITY psk_mod IS
  PORT(psk_clk: IN std_logic;
       datain: IN std_logic_vector(0 downto 0);
       psk_output: OUT std_logic_vector(7 downto 0));
END psk_mod;
```

Este modulador foi também testado da mesma forma que o anterior mantendo todos os parâmetros inalterados. Os sinais adquiridos no osciloscópio, que consistem do sinal de entrada e do respectivo sinal modulado em PSK, estão representados na figura 4.7 e não apresentam surpresas.

4.2.5 Modulador FSK

Na modulação FSK a informação é transmitida na frequência da portadora. Como esta é uma modulação digital existem duas frequências possíveis, uma quando o sinal a

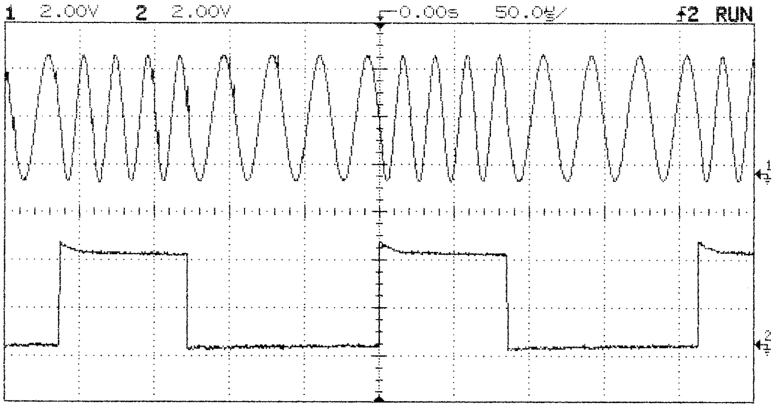


Figura 4.8: Sinal modulado em FSK.

transmitir é "0" e outra quando este sinal é "1". A implementação deste modulador, que se encontra na secção B.5, é mais uma vez feita utilizando o NCO, mas agora a entrada de controlo de frequência não é constante mas oscila entre dois valores. Note-se que devido ao método como foi implementado o NCO, não existe descontinuidade na fase da sinusóide quando se comuta de frequência. A *entity* deste modulador não apresenta novidades em relação às anteriores:

```
ENTITY fsk_mod IS
  PORT(fsk_clk: IN std_logic;
       datain: IN std_logic;
       fsk_output: OUT std_logic_vector(7 downto 0));
END fsk_mod;
```

Este modulador foi testado utilizando os mesmos parâmetros dos moduladores anteriores mas com as seguintes excepções. A frequência fixa da portadora oscila agora entre 46.875 Hz quando o símbolo a transmitir é "1" e 31.250 Hz quando este símbolo é "0". O sinal aplicado a `datain` é agora uma onda quadrada em que o ciclo de serviço (*duty cycle*) não é de 50%, pois cada símbolo dura exactamente quatro ciclos de relógio da portadora. Isto foi feito de modo a manter o sinal estável no osciloscópio apesar das alterações de frequência. Os resultados estão representados na figura 4.8.

4.2.6 Modulador de Quadratura

O modulador de quadratura, também chamado de misturador de quadratura ou modulador I/Q, foi descrito no capítulo anterior na secção sobre misturadores. A implementação em VHDL, que se encontra na secção B.6, apenas é válida para o caso particular em que a frequência da portadora é exactamente 1/4 da frequência de amostragem. Como vimos, neste caso é possível a implementação do modulador sem utilizar multiplicadores pois o modulador pode ser implementado como uma simples máquina de estados (rever a figura 3.18). O modulador de quadratura foi mais uma vez implementado como um componente genérico como se pode confirmar na sua *entity*:

```
ENTITY quadmod IS
  GENERIC (size: positive := 8);
  PORT(clock: IN std_logic;
       i, q: IN std_logic_vector(size-1 downto 0);
       output: OUT std_logic_vector(size-1 downto 0));
END quadmod;
```

O parâmetro `size` especifica o número de bits das entradas `i` e `q` e da saída `output`, tomando por omissão o valor 8. O código VHDL é extremamente simples pois apenas implementa a máquina de estados composta por quatro estados que são percorridos num ciclo infinito.

4.2.7 Modulador AM

A modulação AM é uma modulação analógica em que a amplitude da portadora é alterada (modulada) em função do sinal modulante. Sendo $x(t)$ o sinal analógico modulante, em que $-1 < x(t) < 1$ e sendo $\cos(\omega_o t)$ a portadora, então o sinal modulado em amplitude é dado por $y(t) = [1 + x(t)] \cos(\omega_o t)$. Note a constante 1 dentro dos parênteses rectos. Na ausência desta constante a modulação gerada seria DSB, ou seja a modulação de dupla banda lateral que difere da modulação AM essencialmente pelo seu espectro não incluir uma risca à frequência da portadora. Apenas nos vamos referir à modulação AM pois a adaptação para DSB é trivial.

Uma vez que a implementação em VHDL é uma implementação digital, será necessário digitalizar o sinal analógico que se pretende modular. Neste projecto o sinal analógico digitalizado foi fornecido por um computador pessoal. O código VHDL, que se encontra na secção B.7, está concebido de modo a receber dados da porta paralela do computador mas pode facilmente ser adaptado para utilizar outras interfaces. O código utiliza o modulador de quadratura apresentado na secção anterior pelo que contém a mesma limitação deste, ou seja, a frequência da portadora terá de ser 1/4 da frequência de amostragem. Como um modulador de amplitude não necessita da entrada de

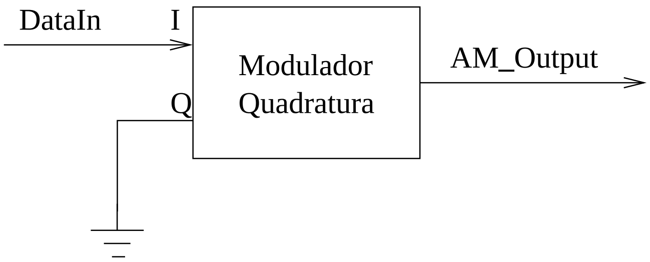


Figura 4.9: Implementação do modulador de AM.

quadratura, `q`, esta fica ligada a um referencial nulo como está ilustrado na figura 4.9. A *entity* utilizada é a seguinte:

```
ENTITY am_mod IS
  PORT(am_clk: IN std_logic;
       datain: IN std_logic_vector(7 downto 0);
       rts:IN std_logic;
       irq: OUT std_logic;
       am_output: OUT std_logic_vector(7 downto 0));
END am_mod;
```

A entrada `am_clk` é mais uma vez a entrada de relógio à frequência de amostragem (4 vezes a frequência da portadora). O sinal modulante deverá ser aplicado na entrada `datain` e deverá estar quantizado com 8 bits. A saída `am_output` contém o sinal modulado em amplitude, mas apenas quando a entrada `rts` está no nível lógico "0". Esta entrada, cujo nome provém de *Request To Send*, deve ser utilizada para ligar/desligar o modulador. Quando lhe está aplicado o nível lógico "1" o modulador está desligado. Quando o nível lógico aplicado é "0" o modulador gera uma onda quadrada à frequência de amostragem do sinal modulante (sinal audio por exemplo) no porto `irq`. Esta frequência é definida no código VHDL no processo `p1` fazendo uso de um contador `lpm_counter` da biblioteca LPM. Para uma frequência de relógio (`am_clk`) de 4 MHz será gerada uma frequência de amostragem de aproximadamente 22 KHz. Não se deve confundir esta frequência de amostragem (audio) com a frequência de amostragem da portadora (que é igual à frequência do sinal de relógio). O *software* no computador deverá responder às transições ascendentes do sinal `irq` colocando uma nova amostra do sinal modulante (quantizada com 8 bits) no porto `datain`.

Em resumo, o computador quando pretender iniciar o modulador colocará o nivel lógico "0" na entrada `rts`. A partir deste momento o computador terá de colocar uma nova amostra do sinal modulante em cada ciclo do sinal `irq`. Para desligar o modulador, o computador deverá colocar o nível lógico "1" na entrada `rts` o que resultará também na paragem de oscilação do porto `irq`. Como este modulador necessita do computador para ser testado, os testes realizados serão apresentados mais tarde após a descrição do *software* desenvolvido.

4.2.8 Modulador FM

Na modulação FM a frequência da portadora é variada linearmente em função da amplitude do sinal modulante. Quando maior for a amplitude deste sinal maior será o desvio de frequência realizado. A implementação deste modulador, que se encontra na secção B.8, é muito semelhante à do modulador AM exceptuando o facto que neste caso não é utilizado o modulador de quadratura mas sim o NCO. A sua *entity* é equivalente à do modulador AM:

```
ENTITY fm_mod IS
  PORT(fm_clk: IN std_logic;
       datain: IN std_logic_vector(7 downto 0);
       rts:IN std_logic;
       irq: OUT std_logic;
       fm_output: OUT std_logic_vector(7 downto 0));
END fm_mod;
```

A comunicação com o computador efectua-se do mesmo modo que no caso anterior. A utilização do NCO torna a implementação deste modulador muito simples pois apenas é necessário aplicar o sinal modulante, `datain`, à entrada de controlo de frequência do NCO. Para os valores presentes no código VHDL e supondo uma frequência de relógio de 25 MHz, a frequência de saída do modulador pode tomar valores entre 3.125 KHz e 3.223 KHz (98 KHz de desvio) em passos de 381,5 Hz (valores aproximados).

4.3 O *Software* do PC

Nesta secção será apresentado o *software* que efectua a comunicação com a FPGA através da interface paralela do computador. Foi decidida a inclusão de um meio de comunicação com um computador devido à flexibilidade que daí advém. O computador poderá, por exemplo, gerar em tempo real uma sequência de dados a enviar para a

FPGA ou, em alternativa, poderá ler esses dados de um ficheiro guardado no disco rígido.

O *software* foi desenvolvido como um *device driver* para o sistema operativo Linux [27]. O Linux é um sistema operativo moderno baseado nos sistemas UNIX clássicos. É um sistema multi-tarefa e multi-utilizador, muito popular no meio académico, e que pode ser utilizado nos vulgares computadores pessoais "compatíveis IBM". A característica que mais distingue este sistema operativo dos restantes é que todo o código do *kernel* está disponível gratuitamente na *Internet* para consulta, adaptação e/ou modificação [28]. Os autores do Linux incentivam mesmo terceiros a estudar o código e a sugerirem melhorias que ficam posteriormente disponíveis para quem desejar. É esta a razão da sua popularidade na comunidade científica e académica e foi também a razão principal que nos levou a optar pela utilização do Linux neste trabalho. A disponibilidade do código revelou-se mesmo muito útil pois podemos consultar o código de outros *drivers* e daí retirar vários ensinamentos. Dos algoritmos VHDL que implementámos apenas os moduladores AM e FM utilizam a comunicação com o computador. No entanto se futuramente esta plataforma for utilizada como um *modem*, poderá tirar-se vantagem da existência no Linux de protocolos como o HDLC e de todo o conjunto TCP/IP (protocolos *Internet*). Outra característica muito útil do Linux, especialmente durante a fase de desenvolvimento, é a possibilidade de inserção e remoção dos *device drivers* no *kernel* sem ser necessário reinicializar o sistema operativo.

De seguida serão apresentados os procedimentos para a utilização do *driver* e posteriormente será descrito o seu funcionamento interno.

4.3.1 Modo de utilização

Em primeiro lugar convém referir que o *driver* desenvolvido apenas funciona com as versões 2.2 do *kernel*, que à data da escrita deste documento é a última versão estável. Está também limitado ao funcionamento na plataforma x86 pois era a única que estava disponível para testes. O computador terá de ter uma porta paralela livre bem como uma linha de pedido de interrupção disponível.

Antes de poder ser utilizado, o *driver* terá de ser compilado. Para tal, o seu código (que se encontra no apêndice D) deverá estar presente num ficheiro de nome `masters.c`. Deverá então ser dado o seguinte comando:

```
$ gcc -D__KERNEL__ -DMODULE -O -I/usr/src/linux/include -c masters.c
```

O sinal `$` representa a linha de comando e não deverá ser introduzido. Após a compilação será criado um ficheiro chamado `masters.o` que é o *driver* propriamente dito que está pronto a ser introduzido no *kernel*.

O Linux segue a filosofia UNIX de acesso aos *drivers*, ou seja o acesso é feito utilizando ficheiros especiais chamados *nodes*. O próximo passo é a criação deste ficheiro que terá de ser feita com previlégios de administrador (*root*). Para tal deverá ser executado o seguinte comando:

```
$ mknod /dev/masters c 111 0
```

Este comando cria um *node* chamado `/dev/masters` cujo *major number* é 111 e *minor number* é 0. O "`c`" indica que este é um *node* associado a um *driver* tipo *char*. Não entraremos em pormenor sobre o significado destes números, basta referir que eles são usados pelo sistema operativo para identificar qual o *driver* que controla este *node*. Uma explicação detalhada encontra-se na referência [29].

Está agora na altura de inserir o *driver* no *kernel*. Na ausência de parâmetros o *driver* irá utilizar a primeira porta paralela (LPT1) e a linha de interrupção número 7. Caso se pretendam usar outros recursos que não estes, eles devem ser declarados como no exemplo seguinte.

```
$ insmod masters.o ioport=0x278 irq=5
```

Neste exemplo o *driver* é inserido no *kernel* e irá utilizar a porta LPT2 (endereço 278 hexadecimal) e a linha de interrupção 5. Este comando completa a sequência necessária para a utilização do *driver* que fica pronto a responder às solicitações das aplicações que pretendam enviar dados para a FPGA. De referir que os primeiros dois passos (a compilação e a criação do *node*) apenas são necessários efectuar uma vez, enquanto que este último passo é necessário sempre que o sistema operativo seja reinicializado ou sempre que o módulo seja retirado do *kernel* com o comando `rmmod`.

Muito resumidamente, as aplicações que pretendam utilizar o *driver* deverão abrir o ficheiro `/dev/masters` para escrita e enviar os dados utilizando a chamada ao sistema (*system call*) `write`. Informações mais detalhadas sobre o desenvolvimento de aplicações em Linux encontram-se nas referências [29][30][31]. Um método mais simples, que foi o método utilizado nos testes realizados, consiste em utilizar o comando de cópia de ficheiros para comunicar com o *driver*.

```
$ cp dados /dev/masters
```

Este comando envia o conteúdo do ficheiro de nome `dados` para o *driver* que por sua vez o envia para a FPGA. A simplicidade deste exemplo ilustra as vantagens de se ter optado por desenvolver o *software* como um *device driver*.

4.3.2 Funcionamento interno

O *device driver* é composto essencialmente por seis funções, são elas a `init_module`, a `cleanup_module`, a `open_device`, a `close_device`, a `send_data` e a `irq_handler`. Estas funções podem ser ainda agrupadas aos pares em três grupos. As funções `init_module` e `cleanup_module` são invocadas quando o módulo é respectivamente inserido ou removido do *kernel*, usando os comandos `insmod` e `rmmod` mencionados anteriormente. As funções `open_device` e `close_device` são invocadas quando a aplicação que pretende usar o *driver* executa a chamada ao sistema `open` ou `close` respectivamente. Por fim, o terceiro grupo é constituído pela função `send_data` que está associada à chamada de sistema `write`, e pela rotina de serviço às interrupções `irq_handler`. Vamos agora ver mais em detalhe cada um destes três grupos de funções.

A execução pelo utilizador do comando `insmod` insere o *driver* no *kernel* que posteriomente invoca a função `init_module`. Esta função encarrega-se de requisitar os recursos necessários ao bom funcionamento do *driver*, entre eles os portos de entrada/saída correspondentes à porta paralela e a linha de interrupção. Neste momento é também definida a função `irq_handler` como sendo a rotina de serviço à interrupção. Quando o *driver* é retirado do *kernel* utilizando o comando `rmmod` é executada a função `cleanup_module` que desfaz as operações da função `init_module`, ou seja liberta todos os recursos por esta requisitados.

Quando uma aplicação pretende utilizar o *driver* ela executa a chamada ao sistema `open` que resulta na invocação da função `open_device`. Esta função começa por verificar se já existe alguma outra aplicação que esteja a utilizar o *driver* e em caso afirmativa nega a utilização ao segundo pedido. Isto porque apenas uma aplicação pode utilizar o *driver* num determinado momento. Em seguida são inicializadas algumas estruturas internas do *driver*. Por fim é programado o controlador da porta paralela para que lance uma interrupção sempre que exista uma transição ascendente no pino $\overline{\texttt{ACK}}$ (pino 10) da porta paralela. Este pino está ligado fisicamente ao porto `irq` dos moduladores AM e FM que foram apresentados anteriormente. A função `close_device` é invocada quando a aplicação não pretende mais utilizar o *driver* e limita-se a registar que este está livre para ser utilizado por outras aplicações.

As funções `send_data` e `irq_handler` são as mais importantes pois é aqui que se processa a comunicação com a FPGA. Tudo começa quando a aplicação executa a chamada ao sistema `write` em que passa como parâmetros os dados que pretende enviar bem como o seu tamanho em bytes. A função `send_data` é então invocada e recebe estes parâmetros. Esta função utiliza um *buffer* circular em que guarda temporariamente os dados a serem enviados. Se no momento em que a aplicação executa a chamada ao sistema `write` este *buffer* estiver cheio, a aplicação é "adormecida" e o *kernel* encarrega-se de atribuir o tempo de processador a outra aplicação[1]. Por outro lado, na primeira vez que a aplicação envia dados para o *driver* o *buffer* estará vazio e a função `send_data` encarrega-se de enviar o primeiro byte de dados para a FPGA. O pino $\overline{\texttt{STROBE}}$ (pino 1) da porta paralela, que se encontra ligado fisicamente ao porto `rts` dos moduladores AM e FM, é colocado ao nível lógico "0" para sinalizar à FPGA que deve iniciar a modulação. Como vimos, a FPGA responde com a aplicação de uma onda quadrada no porto `irq`. Em cada transição ascendente deste sinal irá ser lançada uma interrupção, ou seja, será executada a função `irq_handler` que tinha sido previamente registada como sendo a rotina de serviço de interrupções. Esta função encarrega-se de enviar para a FPGA um novo byte de dados sempre que é invocada. Se a aplicação que está a utilizar o *driver* estiver adormecida devido ao *buffer* ter enchido, então ela é acordada pois agora existe pelo menos um byte livre no *buffer*. Quando o último byte de dados for enviado para a FPGA esta rotina sinaliza este acontecimento voltando a colocar o pino 1 da porta paralela, e por consequência o porto `rts`, ao nível lógico "1". O processo repete-se sempre que a aplicação envie novos dados.

4.4 Teste ao conjunto

Nesta secção serão descritos os testes efectuados à plataforma desenvolvida para nos certificarmos do seu bom funcionamento. Uma vez que os únicos algoritmos que fazem uso da comunicação com o computador são o modulador de AM e de FM, estes serão os alvos deste teste. Os testes efectuados aos restantes moduladores foram já apresentados nas respectivas secções em que que estes foram descritos.

[1]Isto é uma simplificação. Na realidade o tempo de processador é atribuído a outro processo que pode fazer parte ou não da mesma aplicação

Iniciamos o teste com o modulador de amplitude. O módulo `am.vhd` foi "compilado" e carregado para a FPGA através do cabo *Byte-blaster*. Previamente tinha sido colocado um oscilador de quartzo de 4 MHz para servir de relógio de referência. Com os valores presentes no código VHDL a portadora terá uma frequência quatro vezes inferior à do relógio pelo que o sinal será modulado numa portadora de 1.000 KHz que reside no centro da banda comercial de amplitude modulada. O circuito foi então ligado à porta paralela de um PC em que o *device driver* tinha já sido anteriormente inserido no *kernel*. Utilizando a placa de som do computador foi digitalizado um pequeno excerto de uma música utilizando 8 bits de quantificação e uma taxa de amostragem de 22 KHz. Esta é a taxa de amostragem que o código VHDL está programado para receber. Ao ficheiro resultante deu-se o nome `musica.wav`. Sintonizou-se então um vulgar receptor de onda média nos 1.000 KHz e enviou-se o excerto musical para a FPGA com o seguinte comando:

```
$ cp musica.wav /dev/masters
```

A FPGA modulou então o sinal em amplitude que se ouviu no receptor com boa qualidade de áudio. A antena do receptor tinha, no entanto, que estar próxima do conversor digital-analógico devido à baixa potência do sinal.

O segundo e último teste foi semelhante ao anterior mas desta vez utilizando o modulador de frequência. Neste teste o oscilador de quartzo foi substituído por um de 25 MHz. Com os valores presentes no código VHDL o sinal emitido estará modulado na frequência aproximada de 3 MHz e apresentará um desvio máximo de cerca de 100 KHz. Uma vez que o sinal proveniente do DAC não é filtrado, estarão presentes todas as frequências harmónicas devido ao sinal ser amostrado. Ou seja, existirão harmónicos a 25 MHz, 50 MHz, 75 MHz, 100 MHz e assim sucessivamente. Será então de esperar que se possa escutar o sinal modulado nestas frequências ± 3 MHz. Sintonizamos então um receptor da banda comercial de frequência modulada em $100 - 3 = 97$ MHz e em $100 + 3 = 103$ MHz onde pudemos escutar o excerto musical com uma boa qualidade. Utilizando um receptor de rádio-amador foi ainda possível escutar diversos outros harmónicos ao longo do espectro.

Capítulo 5

Conclusão

Esta dissertação abordou o tema da implementação de sistemas de telecomunicações tendo como base a utilização de dispositivos lógicos programáveis. A utilização de dispositivos programáveis cria um enorme leque de opções nos sistemas que os usam. Vários especialistas são da opinião que num futuro próximo os sistemas rádio serão maioritariamente definidos por *software*, os chamados *software radios*. Não só serão controlados, por *software*, os protocolos de rede em banda-base como também as interfaces rádio que permitirão modificar o tipo de modulação e o protocolo de acesso ao meio, tudo sem alterações de *hardware*. Juntamente com os DSP's, os PLD's estarão no coração destes sistemas.

O trabalho aqui descrito pretende ser uma modesta contribuição para acelerar o processo de adopção dos PLD's em sistemas de telecomunicações, em especial nos sistemas rádio. Para isso foi desenvolvida de raiz uma plataforma cujo objectivo é facilitar a simulação, desenvolvimento e teste de sistemas de rádio digital que façam uso de PLD's. Mais especificamente pretende-se que a plataforma desenvolvida incentive a implementação, em projectos futuros, de novos algoritmos com utilidade em sistemas de rádio digital, com vista à criação de uma biblioteca de algoritmos genéricos semelhante às utilizadas nas linguagens de programação de computadores.

A plataforma construída é composta por três partes: o *hardware*, os *algoritmos*, e o *software* do PC. O *hardware* é composto por um circuito electrónico baseado numa FPGA, que contém ainda um conversor digital-analógico e uma interface para ligação à porta paralela dum PC. Foram desenvolvidos alguns algoritmos utilizando a linguagem de descrição de circuitos VHDL. A escolha desta linguagem torna possível a utilização dos algoritmos implementados, sem modificação, em vários dispositivos de diversos fabricantes. Dos algoritmos desenvolvidos destaca-se o oscilador controlado numericamente (NCO) que serve de base a grande parte dos restantes. O NCO foi implementado como um componente genérico onde todos os parâmetros de interesse são configuráveis. Foram ainda implementados moduladores de ASK, PSK, FSK, AM e FM que não são componentes genéricos pois pretendem apenas servir de exemplo da facilidade e versatilidade desta plataforma. Foi também implementado um *device driver* para o sistema operativo Linux que permite a comunicação entre o computador e a FPGA. Este *driver* facilitou o teste dos moduladores de AM e FM pois tornou possível o envio de um sinal áudio digitalizado, gravado previamente no disco rígido do computador. Estes dois moduladores, bem como todos os restantes foram testados com sucesso.

5.1 Perspectivas de evolução

O próximo passo na evolução deste sistema seria dotá-lo da capacidade de recepção e desmodulação. Para isso seria necessário acrescentar um conversor analógico-digital ao circuito electrónico e complementar o *driver* com rotinas de comunicação no sentido FPGA $\rightarrow$ computador, bem como desenvolver algoritmos de desmodulação. O outro passo natural seria a criação de novos algoritmos. As possibilidades são inúmeras e passam pela implementação de uma PLL digital, filtros IIR e FIR (nomeadamente um filtro de Hilbert), e mesmo um algoritmo para calcular a FFT. A implementação de um gerador de sequências pseudo-aleatórias permitiria, em conjunto com o NCO, a criação de um modulador com espalhamento espectral de sequência directa ou de salto em frequência (*frequency hopping*) sem grande dificuldade. Embora já pertencente ao sistema de banda base, poderão implementar-se algoritmos de codificação de canal tal como blocos de detecção e correcção de erros (FEC). Um algoritmo muito interessante que também deveria ser investigado é o algoritmo CORDIC [33][34] que permite o cálculo de funções trigonométricas e outras funções transcendentais utilizando apenas deslocações (*shifts*) e somas.

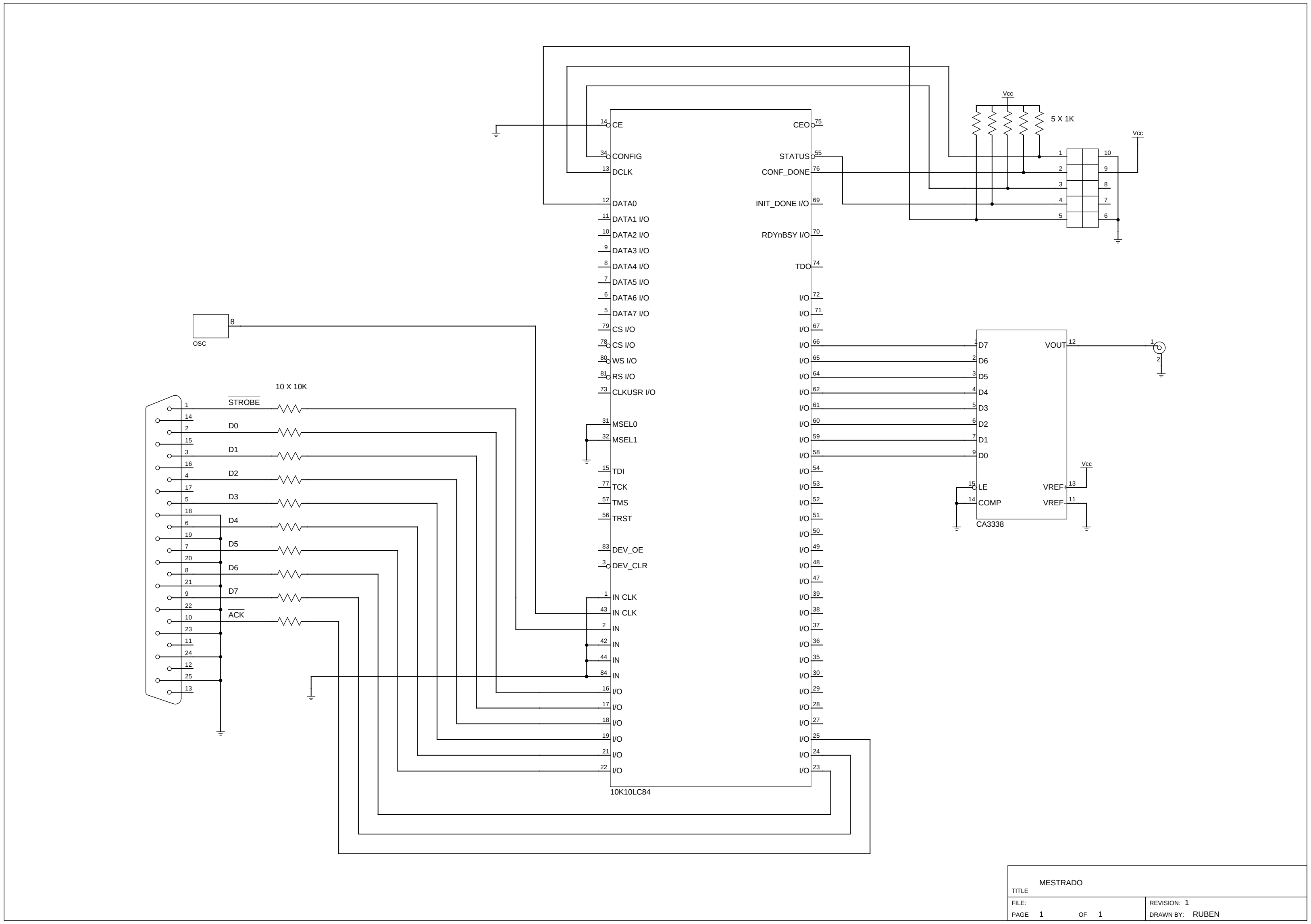
Também seria interessante dotar o sistema de uma etapa de rádio-frequência que permitiria efectuar testes em condições reais de utilização e não apenas com simulações feitas em laboratório.

Porque os temas abordados neste trabalho são de elevado interesse para o autor, é pretensão deste prosseguir com estudos e projectos nesta área e, na medida do possível, continuar o desenvolvimento da plataforma descrita.

Apêndice A

Esquema Eléctrico

Este anexo contém o esquema eléctrico da plataforma descrita no capítulo 4. Para gerar este esquema foi utilizado o programa `gschem` que é parte integrante do projecto gEDA [32].


CE
CONFIG
DCLK
DATA0
DATA1 I/O
DATA2 I/O
DATA3 I/O
DATA4 I/O
DATA5 I/O
DATA6 I/O
DATA7 I/O
CS I/O
CS I/O
WS I/O
RS I/O
CLKUSR I/O
MSEL0
MSEL1
TDI
TCK
TMS
TRST
DEV_OE
DEV_CLR
IN CLK
IN CLK
IN
IN
IN
IN
CEO
STATUS
CONF_DONE
INIT_DONE I/O
RDYnBSY I/O
TDO
10K10LC84
OSC
10 X 10K
STROBE
D0
D1
D2
D3
D4
D5
D6
D7
ACK
Vcc
5 X 1K
D7
D6
D5
D4
D3
D2
D1
D0
LE
COMP
VOUT
VREF+
VREF-
CA3338
MESTRADO
TITLE
FILE:
REVISION: 1
PAGE 1 OF 1
DRAWN BY: RUBEN

Apêndice B

Código VHDL

Este apêndice contém o código VHDL dos algoritmos descritos no capítulo 4.

B.1 Acumulador

```
--
-- acumul.vhd
--
-- This module implements an acumulator.
-- In every rising edge of the clock the output changes to the sum of
-- the actual input and the previous output.
--
-- Parameters:
--
--   acc_size - Number of bits of the input and output signals.
--
-- Ports:
--
--   acc_clk    - The clock.
--   acc_input  - Value to be added in the next rising edge of the clock.
--                This signal has acc_size bits.
--   acc_output - The output of the accumulator.
--                This signal has acc_size bits.
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY acumul IS
  GENERIC (acc_size: natural := 4);
  PORT(acc_clk: IN std_logic;
       acc_input: IN std_logic_vector(acc_size-1 downto 0);
       acc_output: OUT std_logic_vector(acc_size-1 downto 0));
END acumul;

ARCHITECTURE archacumul OF acumul IS
  signal acc: unsigned(acc_size-1 downto 0);
BEGIN

  p1: process(acc_clk)
  begin
    if (acc_clk'event and acc_clk='1') then
      acc <= acc + unsigned(acc_input);
      acc_output <= conv_std_logic_vector(acc, acc_size);
    end if;
  end process;

END archacumul;

--
-- END acumul.vhd
--
```

B.2 NCO

```
--
-- nco.vhd
--
-- This module implements an NCO (Numericaly Controlled Oscilator).
-- It uses the phase acumulation method with a non-compressed LUT.
-- To use a square wave output set lut_out_size to "1". If lut_out_size
-- is bigger than 1 then you must provide a file named "wave.hex" in
-- intel-hex format that contains the wave samples of your intended
-- signal (usually a sinewave).
--
-- Parameters:
--
--    acc_size      - Acumulator size in bits.
--    lut_addr_size - Number or address bits of the LUT.
--    lut_out_size  - LUT output resolution in bits.
--    phase_size    - Number of bits of the phase modulation signal.
--
--  Ports:
--
--    nco_clk    - The Clock.
--    freq       - Frequency control signal. It is acc_size bits wide.
--                 The output frequency is given by [(F/2^acc_size)*freq]
--                 where F is the nco_clock frequency.
--    phase      - Phase control signal. It is phase_size bits wide.
--                 The most significant bit makes 180 degrees phase
--                 changes, the next bit makes 90 degree phase changes,
--                 and so on...
--    nco_output - The output of the oscilator.
--                 It has lut_out_size bits of amplitude resolution.
--    period     - This signal is "0" during the first half period and
--                 "1" during the second half period.
--
--  Dependencies:
--
--    * acumul.vhd
--    * lpm_rom
--    * lpm_add_sub
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.lpm_rom;
USE lpm.lpm_components.lpm_add_sub;

ENTITY nco IS
  GENERIC (acc_size: positive := 24;
           lut_addr_size: positive := 9;
           lut_out_size: positive := 8;
           phase_size: positive := 1);
```

```
  PORT(nco_clk: IN std_logic;
       freq: IN std_logic_vector(acc_size-1 downto 0);
       phase: IN std_logic_vector(phase_size-1 downto 0));
       nco_output: OUT std_logic_vector(lut_out_size-1 downto 0);
       period: OUT std_logic;
END nco;

ARCHITECTURE archnco OF nco IS

  -- output of the acumulator
  signal addr1: std_logic_vector(acc_size-1 downto 0);

  -- output of the phase adder
  signal addr2: std_logic_vector(acc_size-1 downto 0);

BEGIN

  acc: work.acumul
    GENERIC MAP (acc_size)
    PORT MAP (nco_clk, freq, addr1);

  gen1: if (phase_size>0) generate
    -- This is only generated when we need to phase modulate.
    add: lpm_add_sub
      GENERIC MAP (LPM_WIDTH => phase_size,
                   LPM_DIRECTION => "ADD",
                   LPM_REPRESENTATION => "UNSIGNED")
      PORT MAP (dataa => addr1(acc_size-1 downto acc_size-phase_size),
                datab => phase,
                result => addr2(acc_size-1 downto acc_size-phase_size));

    addr2(acc_size-phase_size-1 downto 0)
      <= addr1(acc_size-phase_size-1 downto 0);
  end generate;
  notgen1: if (phase_size=0) generate
    -- This is only generated when we don´t need to phase modulate.
    addr2 <= addr1;
  end generate;

  gen2: if (lut_out_size>1) generate
    -- This is only generated when LUT output resolution is bigger
    -- than 1 bit.
    lut: lpm_rom
      GENERIC MAP (lpm_width => lut_out_size,
                   lpm_widthad => lut_addr_size,
                   lpm_file => "wave.hex",
                   lpm_address_control => "unregistered",
                   lpm_outdata => "unregistered")
      PORT MAP (address =>
                  addr2(acc_size-1 downto (acc_size-lut_addr_size)),
                q => nco_output);
  end generate;
  notgen2: if (lut_out_size=1) generate
```

```
    -- This is only generated when LUT output resolution is 1 bit,
    -- which means that the output is a square wave.
    nco_output(0) <= addr2(acc_size-1);
  end generate;

  period <= addr1(acc_size-1);

END archnco;

--
-- END nco.vhd
--
```

B.3 Modulador ASK

```
--
-- ask_mod.vhd
--
-- This module demonstrates the use of the NCO module to implement a
-- simple Amplitude Shift Keying modulator.
--
-- Ports:
--
--   ask_clk    - The clock (at sampling frequency).
--   datain     - Data to be modulated.
--   ask_output - ASK modulated output.
--
-- Dependencies:
--
--   * nco.vhd
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ask_mod IS
  PORT(ask_clk: IN std_logic;
       datain: IN std_logic;
       ask_output: OUT std_logic_vector(7 downto 0));
END ask_mod;

ARCHITECTURE archask OF ask_mod IS
  signal carrier: std_logic_vector(7 downto 0);
  signal f: std_logic_vector(23 downto 0);
  signal period: std_logic;
  signal dummy: std_logic_vector(0 downto 0);
BEGIN

  dummy <= "0";

  -- change this to use a different carrier frequency.
  f <="000000100000000000000000";

  osc: work.nco
    GENERIC MAP(24,9,8,1)
    PORT MAP(ask_clk, f, dummy, carrier, period);

  ask_output <= carrier when datain='1' else (others => '0');

END archask;

--
-- END ask_mod.vhd
--
```

B.4 Modulador PSK

```
--
-- psk_mod.vhd
--
-- This module demonstrates the use of the NCO module to implement a
-- simple Phase Shift Keying modulator.
--
-- Ports:
--
--   psk_clk    - The clock (at sampling frequency).
--   datain     - Data to be modulated.
--   psk_output - PSK modulated output.
--
-- Dependencies:
--
--   * nco.vhd
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY psk_mod IS
  PORT(psk_clk: IN std_logic;
       datain: IN std_logic_vector(0 downto 0);
       psk_output: OUT std_logic_vector(7 downto 0));
END psk_mod;

ARCHITECTURE archpsk OF psk_mod IS
  signal f: std_logic_vector(23 downto 0);
  signal period: std_logic;
BEGIN

  -- change this to use a different carrier frequency.
  f <="000000100000000000000000";

  osc: work.nco
    GENERIC MAP(24,9,8,1)
    PORT MAP(psk_clk, f, datain, psk_output, period);

END archpsk;

--
-- END psk_mod.vhd
--
```

B.5 Modulador FSK

```
--
-- fsk_mod.vhd
--
-- This module demonstrates the use of the NCO module to implement a
-- simple Frequency Shift Keying modulator.
--
-- Ports:
--
--   fsk_clk    - The clock (at sampling frequency).
--   datain     - Data to be modulated.
--   fsk_output - FSK modulated output.
--
-- Dependencies:
--
--   * nco.vhd
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fsk_mod IS
  PORT(fsk_clk: IN std_logic;
       datain: IN std_logic;
       fsk_output: OUT std_logic_vector(7 downto 0));
END fsk_mod;

ARCHITECTURE archfsk OF fsk_mod IS
  signal f: std_logic_vector(23 downto 0);
  signal period: std_logic;
  signal dummy: std_logic_vector(0 downto 0);
BEGIN

  dummy <= "0";

  osc: work.nco
    GENERIC MAP(24,9,8,1)
    PORT MAP(fsk_clk, f, dummy, fsk_output, period);

  -- change this to use different frequencies.
  f <= "000000100000000000000000" when datain = '0'
    else "000000110000000000000000";

END archfsk;

--
-- END fsk_mod.vhd
--
```

B.6 Modulador Quadratura

```
--
-- quadmod.vhd
--
-- This module implements a quadrature (I/Q) modulator for the special
-- case where the carrier frequency is 1/4 of the sampling frequency.
--
-- Parameters:
--
--   size - Number of bits of i, q and output ports.
--
-- Ports:
--
--   clock  - The clock (at sampling frequency).
--   i      - The in-phase input.
--   q      - The quadrature input.
--   output - The modulated output.
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY quadmod IS
  GENERIC (size: positive := 8);
  PORT(clock: IN std_logic;
       i, q: IN std_logic_vector(size-1 downto 0);
       output: OUT std_logic_vector(size-1 downto 0));
END quadmod;

ARCHITECTURE behav OF quadmod IS
  type state_type is (s0, s1, s2, s3);
  signal state: state_type;
BEGIN
  p1:process(clock)
  begin
    if (clock'event and clock='1') then
      case state is
        when s0 =>
          state <= s1;
          output <= i;
        when s1 =>
          state <= s2;
          output <= q;
        when s2 =>
          state <= s3;
          output <= not(i);
        when s3 =>
          state <= s0;
          output <= not(q);
      end case;
    end if;
```

```
  end process;

END behav;

--
-- END quadmod.vhd
--
```

B.7 Modulador AM

```
--
-- am_mod.vhd
--
-- This module makes use of the quadmod module to implement an
-- Amplitude Modulator for the special case where the carrier frequency
-- is 1/4 of the sampling frequency. With the hardcoded values used it
-- expects a 4 MHz clock, producing a 1 MHz AM carrier, and expects
-- an 8 bit audio sampled at 22 KHz. This module was designed to
-- interface with a PC paralell port.
--
-- Ports:
--
--   am_clk    - The clock (at sampling frequency).
--   datain    - The audio input signal.
--   rts       - Request To Send. When "1" the modulator is off. When "0"
--               the modulator is on and generates irq's to request data.
--   irq       - Interrupt Request. When rts="0" this signal is a square
--               wave at the audio sampling frequency. The sender should
--               provide the a new audio sample in every rising edge of
--               this signal.
--   am_output - The AM modulated signal.
--
-- Dependencies:
--
--   * quadmod.vhd
--   * lpm_counter
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.lpm_counter;

ENTITY am_mod IS
  PORT(am_clk: IN std_logic;
       datain: IN std_logic_vector(7 downto 0);
       rts:IN std_logic;
       irq: OUT std_logic;
       am_output: OUT std_logic_vector(7 downto 0));
END am_mod;

ARCHITECTURE archam OF am_mod IS

  signal carrier: std_logic_vector(7 downto 0);
  signal b: std_logic;
  signal bb: std_logic;
  signal c: std_logic_vector(7 downto 0);
  signal res: std_logic;
  signal datap: std_logic_vector(7 downto 0);
  signal ground: std_logic_vector(7 downto 0);
```

```
BEGIN

  datap(7) <= '1';
  datap(6 downto 0) <= datain(7 downto 1);
  ground <= "10000000";

  cnt: lpm_counter
    GENERIC MAP(LPM_WIDTH => 8, LPM_DIRECTION => "UP")
    PORT MAP(clock => am_clk, q=> c, sclr => res);

  modl: work.quadmod
    GENERIC MAP(8)
    PORT MAP(am_clk, datap, ground, carrier);

  p1:process(am_clk)
  begin
    if (rising_edge(am_clk)) then
      if (c="01011001") then   -- c=Fck/(2*Fas)-2
                               -- where Fck is am_clk frequency and
                               -- Fas is audio sampling frequency
        res <= '1';
        b <= not b;
      else
        res <= '0';
      end if;
    end if;

  end process;

  am_output <= carrier when rts='0' else (others => '0');

  p2: process(b)
  begin
    if (rising_edge(b)) then
      bb <= not rts;
    end if;
  end process;

  irq <= (not rts) and b and bb;

END archam;

--
-- END am_mod.vhd
--
```

B.8 Modulador FM

```
--
-- fm_mod.vhd
--
-- This module makes use of the NCO module to implement an FM modulator.
-- With the harcoded values used it expects a 25 MHz clock and generates
-- a FM signal with 3.174 MHz center frequency and +/- 48.6 KHz frequency
-- deviation. It also expects an 8 bit audio sampled at 22 KHz.
-- This module was designed to interface with a PC paralell port.
--
-- Ports:
--
--   fm_clk    - The clock (at sampling frequency).
--   datain    - The audio input signal.
--   rts       - Request To Send. When "1" the modulator is off. When "0"
--               the modulator is on and generates irq's to request data.
--   irq       - Interrupt Request. When rts="0" this signal is a square
--               wave at the audio sampling frequency. The sender should
--               provide the a new audio sample in every rising edge of
--               this signal.
--   fm_output - The FM modulated signal.
--
-- Dependencies:
--
--   * nco.vhd
--   * lpm_counter
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.lpm_counter;

ENTITY fm_mod IS
  PORT(fm_clk: IN std_logic;
       datain: IN std_logic_vector(7 downto 0);
       rts:IN std_logic;
       irq: OUT std_logic;
       fm_output: OUT std_logic_vector(7 downto 0));
END fm_mod;

ARCHITECTURE archfm OF fm_mod IS

  signal carrier: std_logic_vector(7 downto 0);
  signal b: std_logic;
  signal bb: std_logic;
  signal c: std_logic_vector(9 downto 0);
  signal res: std_logic;
  signal f: std_logic_vector(23 downto 0);
  signal period: std_logic;
  signal dummy: std_logic_vector(0 downto 0);
```

```
BEGIN

  dummy <= "0";

  -- change this to use different carrier and modulation index.
  f(23 downto 16) <= "00100000";
  f(15 downto 8) <= datain;
  f(7 downto 0) <= "00000000";

  cnt: lpm_counter
    GENERIC MAP(LPM_WIDTH => 10, LPM_DIRECTION => "UP")
    PORT MAP(clock => fm_clk, q=> c, sclr => res);

  osc: work.nco
    GENERIC MAP(24,9,8,1)
    PORT MAP(fm_clk, f, dummy, carrier, period);

  p1:process(fm_clk)
  begin
    if (rising_edge(fm_clk)) then
      if (c="1000110110") then   -- c=Fck/(2*Fs)-2
                                 -- where Fck is am_clk frequency and
                                 -- Fs is audio sampling frequency
        res <= '1';
        b <= not b;
      else
        res <= '0';
      end if;
    end if;

  end process;

  fm_output <= carrier when rts='0' else (others => '0');

  p2: process(b)
  begin
    if (rising_edge(b)) then
      bb <= not rts;
    end if;
  end process;

  irq <= (not rts) and b and bb;

END archfm;

--
-- END fm_mod.vhd
--
```

Apêndice C

Código Octave/Matlab

Este apêndice contém o código do programa `intelhex.m` que dado um vector composto por números inteiros compreendidos entre 0 e 255, gera um ficheiro no formato INTEL-HEX. Este código destina-se a ser executado a partir do programa *Octave* ou do programa *Matlab*.

```
%INTELHEX(onda)
%Escreve o vector 'onda' para um ficheiro em formato INTEL-HEX.
%Os elementos de 'onda' te^m de estar compreendidos entre 0 e 255.

function intelhex(onda)

NCOLUNAS=68;

if ((min(onda)<0) | (max(onda)>255))
  error('O vector contem valores inferiores a 0 ou superiores a 255');
end

NOME=input('Nome do ficheiro INTEL-HEX a gravar ?','s');

FID = fopen(NOME,'w');

tamanho=2*length(onda);
indice=1;
for i=1:ceil(tamanho/NCOLUNAS)
  checksum=0;

  if i==ceil(tamanho/NCOLUNAS)
    nbytes=rem(tamanho,NCOLUNAS)/2;
  else
    nbytes=NCOLUNAS/2;
  end

  endereco = (i-1)*NCOLUNAS/2;
  checksum = checksum + nbytes + floor(endereco/256) + rem (endereco,256);

  fprintf(FID,':');
  fprintf(FID,'%2.2X',nbytes);
  fprintf(FID,'%4.4X',endereco);
  fprintf(FID,'00');

  for j=1:nbytes
    fprintf(FID,'%2.2X',onda(indice));
    checksum = checksum + onda(indice);
    indice=indice+1;
  end

  fprintf(FID,'%2.2X\r\n', rem(256-rem(checksum,256),256));

end

fprintf(FID,':00000001FF\n');

fclose(FID);

% FIM intelhex.m
```

Apêndice D

Código C

Este apêndice contém o código do *device driver* para o kernel Linux 2.2.x que efectua a comunicação do computador com a FPGA.

```c
//
// masters.c
//
// This is a device driver module for the Linux 2.2.x kernel.
// It is used to send data to the FPGA through the parallel port.
// This can be achieved by opening the /dev/masters file and writing to it.
// This file should have a major number of 111 (not an official number).
// By default this driver uses LPT1 and irq 7. To use different settings
// you must insert the module with the ioport= and irq= parameters.
// This driver should be compiled with the following command:
//   gcc -D__KERNEL__ -DMODULE -O -I/usr/src/linux/include -c masters.c
//

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ioport.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <asm/io.h>
#include <asm/uaccess.h>

#define NAME "masters"
#define MY_MAJOR 111
#define BUFFER_SIZE 1024

#define NO 0
#define YES 1

#define DATA ioport+0
#define STATUS ioport+1
#define CONTROL ioport+2

#define LPT1 0x378
#define LPT2 0x278

int ioport = LPT1;
int irq = 7;

MODULE_PARM(ioport, "i");
MODULE_PARM(irq, "i");

typedef struct Dev_Struct {

  char buffer[BUFFER_SIZE];
  volatile int head, tail;
  volatile char rts;
  char used;

} Dev_Struct;

struct Dev_Struct dev;
struct wait_queue *wqueue;
```

```
loff_t unsupported(struct file * filp, loff_t offset, int origin)
{
  return -ESPIPE;
}

//
// This routine is executed everytime an irq is requested by the FPGA
//
void irq_handler(int irq, void *dev_id, struct pt_regs *regs) {

  if (dev.head!=dev.tail) { // there is data to send
    outb(*(dev.buffer+dev.head), DATA);
    dev.head = (dev.head+1) % BUFFER_SIZE;
    wake_up_interruptible(&wqueue);
  }
  else { // buffer is empty
    if (dev.rts == NO)
      printk(KERN_ERR NAME ": Unexpected interrupt.\n");
    else {
      // no more data to send to lets...
      outb(0x10, CONTROL); /* ...disable RTS */
      dev.rts = NO;
    }
  }
}

//
// This routine is executed when a process invokes the open system call
//
int open_device(struct inode *inode, struct file *filp) {

  if (dev.used == YES)
    return -EBUSY;

  // struct initialisation
  dev.used = YES;
  dev.head = dev.tail = 0;
  dev.rts = NO;

  outb(0x10, CONTROL); /* Enable lpt interrupt, no RTS */

  MOD_INC_USE_COUNT;

  return 0;
}

//
// This routine is executed when a process invokes the close system call
//
int close_device(struct inode *inode, struct file *filp) {

  dev.used = NO;
```

```
  MOD_DEC_USE_COUNT;

  return 0;
}

//
// This routine is executed when a process invokes the write system call
//
ssize_t send_data(struct file *filp, const char *user_buf, size_t count,
                  loff_t *ppos) {

  int free, max;

  if (dev.tail>=dev.head) {
    free=BUFFER_SIZE - (dev.tail-dev.head) - 1;
    max=BUFFER_SIZE - dev.tail;
    if (free<max)
      max=free;
  }
  else {
    max=free=dev.head-dev.tail-1;
  }

  if (filp->f_flags & O_NONBLOCK & (free==0))
    return -EAGAIN;

  while (free == 0) {

    interruptible_sleep_on(&wqueue);
    if (signal_pending(current)) /* a signal arrived */
      return -ERESTARTSYS; /* tell the fs layer to handle it */

    if (dev.tail>=dev.head) {
      free=BUFFER_SIZE - (dev.tail-dev.head) - 1;
      max=BUFFER_SIZE - dev.tail;
      if (free<max)
        max=free;
    }
    else {
      max=free=dev.head-dev.tail-1;
    }
  }

  if (count>max)
    count=max;

  if (copy_from_user(dev.buffer+dev.tail, user_buf, count))
    return -EFAULT;

  cli();
  dev.tail = (dev.tail+count) % BUFFER_SIZE;
  if (dev.rts == NO) { // modulator was off
    dev.rts = YES;     // so turn it on and ...
```

```
    outb(*(dev.buffer+dev.head), DATA); // ... start sending data
    dev.head = (dev.head+1) % BUFFER_SIZE;
    outb(0x11, CONTROL); /* RTS */
  }
  sti();

  return count;
}

struct file_operations fops = {
  unsupported,  /* llseek */
  NULL,         /* read */
  send_data,    /* write */
  NULL,         /* readdir */
  NULL,         /* poll */
  NULL,         /* ioctl */
  NULL,         /* mmap */
  open_device,  /* open */
  NULL,         /* flush */
  close_device  /* release */
};

//
// This routine is executed when the module is inserted in the kernel
//
int init_module(void) {

  int err;

  EXPORT_NO_SYMBOLS;

  err=check_region(ioport, 3);
  if (err) {
    printk(KERN_WARNING NAME ": IO address already in use.\n");
    goto fail_io_in_use;
  }
  request_region(ioport, 3, NAME);

  dev.used = NO; // No process is using us
  dev.rts = NO;  // RTS is off
  outb(0x00, CONTROL); /* Disable lpt interrupt */

  err=register_chrdev(MY_MAJOR, NAME, &fops);
  if (err<0) {
    printk(KERN_WARNING NAME ": can't get major %d\n",MY_MAJOR);
    goto fail_major;
  }

  /* this should probably be done in open_device */
  err=request_irq(irq, irq_handler, SA_INTERRUPT, NAME, NULL);
  if (err) {
    printk(KERN_WARNING NAME ": can't get IRQ %d.\n", irq);
    goto fail_irq;
```

```
  }

  return 0;

  fail_irq: unregister_chrdev(MY_MAJOR, NAME);
  fail_major: release_region(ioport, 3);
  fail_io_in_use: return err;
}

//
// This routine is executed when the module is removed from the kernel
//
void cleanup_module(void) {

  outb(0x00, CONTROL); /* Disable lpt interrupt */
  free_irq(irq, NULL);
  release_region(ioport, 3);
  unregister_chrdev(MY_MAJOR, NAME);
}
```

Bibliografia

[1] M. E. Frerking, *Digital Signal Processing in Communication Systems*, Van Nostrand Reinhold, 1993. ISBN 0-442-01616-6

[2] W. H. W. Tuttlebee, *Software Radio Technology: A European Perspective*, IEEE Communications, Fev. 1999, pags. 118–123

[3] M. Cummings, S. Haruyama, *FPGA in the Software Radio*, IEEE Communications, Fev. 1999, pags. 108–112

[4] H. Kopka, P. W. Daly, *A Guide to LaTeX*, Addison–Wesley, 1999. ISBN 0-201-39825-7

[5] R. H. Walden, *Performance Trends for Analog-to-Digital Converters*, IEEE Communications, Fev. 1999, pags. 96–101

[6] J. Tsui, *Digital Techniques for Wideband Receivers*, Artech House, 1995. ISBN 0-89006-808-9

[7] H. C. Miranda, *Realização de um Sistema de Comunicação de Espalhamento Espectral Usando Técnicas de Rádio Digital*, Tese de Mestrado apresentado à Faculdade de Engenharia da Universidade do Porto, 1998.

[8] J. Crols, M. Steyaert, *CMOS Wireless Transceiver Design*, Kluwer Academic Publishers, 1997. ISBN 0-7923-9960-9

[9] H. Tsurumi, Y. Suzuki, *Broadband RF Stage Architecture for Software-Defined Radio in Handheld Terminal Applications*, IEEE Communications, Fev. 1999, pags. 90–95

[10] A. A. Abidi, *CMOS Wireless Transceivers: The New Wave*, IEEE Communications, Ago. 1999, pags. 119–124

[11] B. Goldberg, *Digital Techniques in Frequency Synthesis*, McGraw-Hill, 1996. ISBN 0-07-024166-X

[12] Altera Corporation Homepage, `http://www.altera.com`.

[13] Altera Corporation, *FLEX 10K Embedded Programmable Logic Family Data Sheet*, 1998. Versão 3.13.

[14] Intersil Corporation Homepage, `http://www.intersil.com`.

[15] Intersil Corporation, *CA3338 Datasheet*, Dezembro 1993.

[16] A. Rushton, *VHDL for Logic Synthesis* — 2ª edição, Wiley, 1998. ISBN 0-471-98325-X

[17] K. Skahill, Cypress Semiconductor, *VHDL for Programmable Logic*, Addison–Wesley, 1996. ISBN 0-201-89573-0

[18] EDIF Homepage, `http://www.edif.org`.

[19] LPM Homepage, `http://www.edif.org/lpmweb`.

[20] Intel Corporation, *Hexadecimal Object File Format Specification*, Revision A, January 1988.

[21] Octave Homepage, `http://bevo.che.wisc.edu/octave/`.

[22] MathWorks Homepage, `http://www.mathworks.com`.

[23] T. McDermott, *Wireless Digital Communications: Design and Theory*, Tucson Amateur Packet Radio Corporation, 1996. ISBN 0-9644707-2-1

[24] A. B. Carlson, *Communication Systems* — 3ª edição, McGraw-Hill, 1986. ISBN 0-07-009960-X

[25] M. Schwartz, *Information, Transmission, Modulation and Noise* — 4º edição, McGraw-Hill, 1990. ISBN 0-07-100931-0

[26] J. G. Proakis, *Digital Communications* — 3ª edição, McGraw-Hill, 1995. ISBN 0-07-051726-6

[27] Linux Homepage, `http://www.linux.org`.

[28] Linux Kernel Homepage, `http://www.kernel.org`.

[29] A. Rubini, *Linux Device Drivers*, O'Reilly, 1998. ISBN 1-56592-292-1

[30] W. R. Stevens, *Advanced Programming in the UNIX environment*, Addison–Wesley, 1992. ISBN 0-201-56317-7

[31] M. K. Johnson, E. W. Troan, *Linux Application Development*, Addison–Wesley, 1998. ISBN 0-201-30821-5

[32] gEDA Homepage, `http://www.geda.seul.org`.

[33] J. E. Volder, *The CORDIC Trignometric Computing Technique*, IRE Transactions on Electronic Computers, Setembro 1960, págs. 330–334.

[34] R. Andraka, *A survey of CORDIC algorithms for FPGA based computers*, Proceedings of FPGA'98 ACM Conference, 1998.