



4 2014

IMPLEMENTAÇÃO DO CODIFICADOR DE FALA G.729 EM FPGA

CARLA SOFIA INÁCIO BRITO

DISSERTAÇÃO DE MESTRADO APRESENTADA

À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM

MIEEC – MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

A Dissertação intitulada

“Implementação do Codificador de Fala G.729 em FPGA”

foi aprovada em provas realizadas em 11-04-2014

o júri



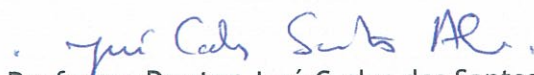
Presidente Professor Doutor António José Duarte Araújo
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto



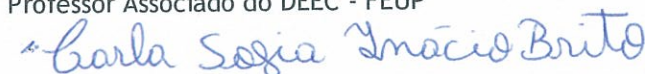
Professor Doutor Manuel Gradim de Oliveira Gericota
Professor Adjunto do Departamento de Engenharia Eletrotécnica do Instituto
Superior de Engenharia do Porto



Professor Doutor João Paulo de Castro Canas Ferreira
Professor Auxiliar do Departamento de Engenharia de Computadores da Faculdade
de Engenharia da Universidade do Porto



Professor Doutor José Carlos dos Santos Alves
Professor Associado do DEEC - FEUP



O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Carla Sofia Inácio Brito

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



FEUP

Implementação do Codificador de Fala G.729 em FPGA

Carla Sofia Inácio Brito

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Orientador: João Canas Ferreira (Prof. Dr.)

Co-orientador: José Carlos Alves (Prof. Dr.)

15 de Abril de 2014

Resumo

O objectivo do projecto consiste na implementação do codificador de fala G.729 em FPGA e, usando as características dessa plataforma reconfigurável, melhorar o desempenho do codificador.

O projecto teve como ponto de partida a aplicação em C do codificador e um estudo feito sobre o mesmo. O profiling da aplicação foi efectuado para determinar as etapas de codificação com tempos de execução mais elevadas. As duas funções com maior tempo de execução foram seleccionadas para serem implementadas em Hardware: a função *Residue* e *Lag Max*.

Foram exploradas diferentes estratégias de implementação em Hardware de cada uma das funções, explorando técnicas tais como o pipelining e paralelismo, por forma a obter melhores desempenhos. Para além disso, o Software foi modificado substituindo a função em C por um device driver que permitia a comunicação com os módulos em Hardware.

A placa de desenvolvimento ML507 da Xilinx é utilizada no projecto. Esta plataforma contém um Virtex-5 com PowerPC.

Abstract

The aim of the project is the implementation of the G.729 speech coder in FPGA, using the features of this reconfigurable platform, to improve its performance.

The starting point of this project was the application in C of the encoder and its study. The application profiling was performed to determine the encoding steps with highest execution times. The two functions with highest execution time were selected for implementation in Hardware: *Residue* and *Lag Max*.

Different strategies were explored for implementation in hardware of each function, exploring techniques such as pipelining and parallelism in order to obtain better performances. In addition, the Software was modified by replacing the C function by a device driver that allowed communication with the modules in hardware.

The Xilinx MI507 development board was used in the project. This platform contains a Virtex-5 with PowerPC.

Agradecimentos

Agradeço ao Prof. João Canas Ferreira pelo suporte e orientação ao longo do desenvolvimento da minha tese.

Agradeço aos meus colegas que, com o espírito de entreajuda e muita amizade, ajudaram a atravessar dificuldades encontradas.

Agradeço a todos aqueles que me ajudaram ao longo desses anos do meu curso, que me ajudaram a adaptar a vivência num país distante do meu e que sempre me deram forças para continuar na linha luta pelo conhecimento.

Carla Brito

*“It’s hardware that makes a machine fast.
It’s software that makes a fast machine slow.”*

Craig Bruce

Conteúdo

1	Introdução	1
1.1	Objectivos	2
1.2	Estrutura da Dissertação	2
2	Estado da Arte	5
2.1	FPGA	5
2.1.1	Aplicações de FPGA	6
2.1.2	Arquitectura básica	6
2.1.3	Virtex-5	6
2.2	G.729	8
2.2.1	Descrição geral	8
2.2.2	Codificação	9
2.2.3	Descodificação	11
2.3	Fluxo de Projeto para Aplicações Audio	12
2.3.1	Plataforma de desenvolvimento	14
2.3.2	Ferramentas	14
3	Análise da Aplicação	17
3.1	Descrição Detalhada das Funções	19
3.1.1	<i>Residue</i>	19
3.1.2	<i>Lag_max</i>	22
4	Aspectos Gerais	27
4.1	Implementação em Hardware	27
4.2	Ambiente de Verificação	29
4.3	Implementação na placa de desenvolvimento	30
4.4	Ambiente de Validação	30
5	Implementação em Hardware	33
5.1	Sub-Módulos	33
5.1.1	Multiplicador	33
5.1.2	Somador	34
5.2	<i>Residue</i>	36
5.2.1	<i>Residue10</i> , <i>Residue30</i> e <i>Residue30_10</i>	36
5.2.2	<i>Residue</i>	50
6	Implementação em Software	53
6.1	Desenvolvimento de Software de acesso a Hardware	53
6.2	<i>Residue</i>	54

7	Validação e Análise de resultados	57
7.1	Residue	58
7.1.1	Tempos de processamento	58
7.1.2	Comparação entre os três testes	60
8	Conclusão	61
8.1	Trabalhos Futuros	61
	Referências	63

Lista de Figuras

2.1	Processo de Codificação [1]	10
2.2	Processo de decodificação [2]	11
2.3	Fluxo de projecto para aplicações audio segundo [3]	13
3.1	Vector global de fala <i>speech</i>	20
3.2	Acesso ao vector <i>speech</i> nas três chamadas de <i>Residue</i> para M igual a 30	21
3.3	Acesso ao vector <i>speech</i> nas três chamadas de <i>Residue</i> para M igual a 10	21
3.4	Regiões de calculo de <i>Lag_max</i> para as três chamadas a função	25
4.1	Ambiente de Verificação	29
4.2	Sistema base	31
4.3	Ambiente de validação	31
5.1	Inferir o DSP48	34
5.2	2ª parte do módulo <i>multiplier</i>	35
5.3	O primeiro caso de teste da função <i>Residue</i> na placa de desenvolvimento	37
5.4	O segundo caso de teste da função <i>Residue</i> na placa de desenvolvimento	37
5.5	O terceiro caso de teste da função <i>Residue</i> na placa de desenvolvimento	37
5.6	Máquina de Estado dos módulos <i>Residue10</i> e <i>Residue30</i>	40
5.7	Máquina de Estado do módulo <i>Residue30_10</i>	41
5.8	Sinal de <i>start</i> e sinal de <i>done</i>	41
5.9	Registos de deslocamentos do módulo <i>Residue10</i>	43
5.10	Registos de deslocamentos dos módulos <i>Residue30</i> e <i>Residue30_10</i>	44
5.11	Estrutura de cálculo do resíduo no módulo <i>Residue10</i>	45
5.12	Estrutura de cálculo do resíduo no módulo <i>Residue30</i>	46
5.13	Estrutura de cálculo do resíduo no módulo <i>Residue30_10</i>	47
5.14	Estrutura de cálculo do resíduo no módulo <i>Residue30_10</i>	47
5.15	Latência	48
5.16	Ligação dos módulos aos respectivos DPBRAM	49
5.17	Funcionamento do módulo <i>Residue</i>	51
5.18	Ligação do módulo aos DPBRAMs	52

Lista de Tabelas

2.1	Recursos da Virtex-5 XC5VFX70T [4]	8
2.2	G.729 original e anexos [2]	9
3.1	Tempo de codificação de um segmento de fala	18
3.2	Resultados do <i>Profiling</i> para nível de optimização 1	18
3.3	Resultados do <i>Profiling</i> para nível de optimização 2	18
3.4	Resultados do <i>Profiling</i> para nível de optimização 3	19
5.1	Sinais do módulo <i>multiplier</i>	34
5.2	Sinais do módulo <i>adder</i>	35
5.3	Sinais dos módulos <i>Residue10</i> e <i>Residue30</i>	38
5.4	Descrição dos parâmetros dos três módulos	38
5.5		39
5.6		42
5.7	Mapeamento das entradas e saídas dos módulos nos registos acessíveis por Software	50
5.8	Sinais do módulo <i>Residue</i>	51
5.9	Descrição dos parâmetros do módulo	51
5.10	Mapeamento das entradas e saídas do módulo <i>Residue</i> nos registos acessíveis por Software	52
7.1	Tempo total de execução da aplicação	58
7.2	Percentagens de melhorias no Teste 1	59
7.3	Percentagens de melhorias no Teste 2	59
7.4	Percentagens de melhorias no Teste 3	59
7.5	Tempo de processamento de um segmento de fala	59
7.6	Tempo de escrita dos vectores nas BRAMs	60
7.7	Tempo de leitura dos vectores nas BRAMs	60
7.8	Resumo dos resultados obtidos	60

Abreviaturas e Símbolos

ASIC	<i>Application-Specific Integrated Circuit</i>
BRAM	<i>Block Random Access Memory</i>
CLB	<i>Configurable Logic Block</i>
DPBRAM	<i>Dual Port Block Random Access Memory</i>
DSP	<i>Digital Signal Processor</i>
DTX	<i>Discontinuous Transmission</i>
EDK	<i>Embedded Development Kit</i>
FPGA	<i>Field-Programmable Gate Array</i>
HDL	<i>Hardware Description Language</i>
ISE	<i>Integrated Software Environment</i>
ITU-T	<i>International Telecommunication Union - Telecommunication Standardization Sector</i>
LP	<i>Linear Prediction</i>
LSP	<i>Line Spectrum Pairs</i>
RAM	<i>Random Access Memory</i>
REFLECT	<i>Rendering FPGA to Multi-Core Embedded computing</i>
Voip	<i>Voice over Internet Protocol</i>
Kb	<i>Kilo bits</i>
Kbps	<i>Kilo bits per second</i>
ms	<i>mili second</i>

Capítulo 1

Introdução

Esta dissertação descreve o trabalho desenvolvido ao longo da tese sobre o tema "Implementação do codificador de fala G.729 em FPGA".

Os Sistemas Embarcados estão presentes nas mais diversas áreas de aplicação. Todos os dias estamos em contacto com vários sistemas embarcados, tendo um grande impacto sobre a nossa vida quotidiana e uma grande importância económica na nossa sociedade. As suas áreas de aplicação inclui eletrónica automóvel, eletrónica aeronautica, comboios, telecomunicações, sistemas de autenticação, aplicações médicas, aplicações militares, eletrónica de consumo, robótica, equipamentos de fabricação, energia.

A nova Era de explosão de troca de informações e do aumento extraordinário e diversificado do mercado de consumo eletrónico trouxe grandes desafios aos Sistemas Embarcados. Cada vez mais os produtos são mais complexos, verificando um exigente aumento de funcionalidades nesses produtos. Para lidar com isso, os Sistemas Embarcados devem ser cada vez mais eficientes, mais seguros e mais inovadores, trazendo benefícios tais como aumento do desempenho, redução dos recursos, energia consumida e preço e maior gestão da informação para garantir a sua segurança.

Muitas das funcionalidades dos Sistemas Embarcados são e irão ser implementadas em Software. A implementação em Software oferece grande flexibilidade de implementação mas o aumento da dimensão dos códigos fonte e da necessidade do aumento do desempenho faz com que a implementação em Hardware assume uma importância nos Sistemas Embarcados. A implementação em Hardware permite lidar com situações das quais é difícil de garantir em Software, tais como, a gestão da potência e o desempenho em tempo real.

As FPGAs são uma excelente escolha para implementação eficiente e heterogênea dos Sistemas Embarcados. A reconfigurabilidade é a principal característica que permite distinguir esta tecnologia de outras tecnologias usadas na implementação dos Sistemas Embarcados. Em vez de usar tecnologias construídas à medida, as FPGAs permitem configurar o comportamento de qualquer circuito especificado.

1.1 Objectivos

O objectivo do projecto consiste na implementação de partes do codificador de fala G.729 em FPGA e, usando as características dessa plataforma reconfigurável, melhorar o desempenho do codificador.

As principais características do codificador G.729 são alto desempenho e alta qualidade de codificação. Para tornar o codificador validável, o sinal de voz deve ser codificado em tempo real e a correcção da saída deve ser garantida. Para melhorar o desempenho, optou-se por uma implementação heterogênea (Software e Hardware). Enquanto que a implementação em Software fornece a flexibilidade necessária de implementação, a aceleração de partes do codificador em Hardware permite atingir melhores desempenhos, com a exploração de duas importantes técnicas, o Paralelismo e o *Pipelining*. Outro aspecto a ter em conta para melhorar o desempenho é a transferência de dados entre Software/Hardware, que consome uma grande parcela de tempo. Minimizando ao máximo a transferência de dados, melhores resultados temporais são obtidos.

O codificador encontra-se já implementado em Software na linguagem C. Na análise da aplicação foram identificadas um conjunto de funções com tempos de execução elevadas e candidatas a aceleração em Hardware. Dessas funções duas foram escolhidas para implementação em Hardware.

Alguns modelos de Hardware das funções críticas foram fornecidos e o trabalho centra-se no estudo e implementação desses e outros modelos em Hardware bem como na exploração do *trade-off* entre aumentar o desempenho e reduzir os recursos consumidos, e na análise do impacto da generalização e especialização em Hardware.

A plataforma de desenvolvimento ML507 da Xilinx é utilizada. Esta plataforma contém um Virtex-5 XC5VFX70T com PowerPC 440 incorporado, de onde se escolheu correr o kernel Linux. A aplicação em C é executada no PowerPC e a comunicação entre Software e Hardware é feito com recurso a um *device driver*.

1.2 Estrutura da Dissertação

Para além deste capítulo de Introdução, esta dissertação contém mais 7 capítulos:

- Capítulo 2 - Neste capítulo faz-se a revisão da literatura e levantamento do Estado da Arte.
- Capítulo 3 - Neste capítulo faz-se uma análise da aplicação em C e das funções a serem implementadas.
- Capítulo 4 - Neste capítulo são apresentados aspectos gerais de implementação, bem como os ambientes de verificação e validação.
- Capítulo 5 - Neste capítulo descreve-se as estratégias utilizadas na implementação das funções em Hardware.

- Capítulo 6 - Neste capítulo descrevem-se as modificações efectuadas em Software e a transferência de dados entre Software e Hardware.
- Capítulo 7 - Neste capítulo são apresentados e analisados os resultados obtidos.
- Capítulo 8 - Neste capítulo conclui-se o projecto desenvolvido e são apresentados possíveis trabalhos futuros.

Capítulo 2

Estado da Arte

Neste capítulo, é apresentada a arquitectura da tecnologia FPGA, com ênfase na arquitectura da família Virtex-5 que inclui a FPGA utilizada no desenvolvimento deste trabalho; é descrito a norma do Codec G.729 e por último é apresentado o fluxo de projecto para aplicações audio que foi usada na implementação do G.729 em FPGA e as ferramentas usadas no desenvolvimento do projecto.

2.1 FPGA

Field Programmable Gate Array (FPGA) [5] é um dispositivo semiconductor, desenvolvido pela Xilinx em 1985, com capacidade para ser reconfigurado pelo utilizador para uma determinada aplicação ou determinada funcionalidade, após o seu fabrico. O processo de reconfiguração é o que distingue os FPGAs de outras tecnologias, por exemplo, ASICs que são construídas sob medida para um fim específico. Em comparação com tecnologias ASIC, os FPGAs apresentam várias vantagens [6], por exemplo:

- Rápida prototipagem;
- Capacidade de reprogramação das aplicações no FPGA para verificação e validação;
- Rápida saída para o mercado.

Os FPGAs apresentam 2 mecanismo de reconfiguração [7]:

- Dinâmica - A reconfiguração Dinâmica permite que o processo de configuração seja feito durante o tempo de execução do sistema.
- Estática - A reconfiguração Estática, por outro lado, só permite a reconfiguração quando o sistema estiver inactivo.

A reconfiguração Dinâmica e Estática podem ser:

- Parcial - A reconfiguração Parcial permite que apenas uma parte da FPGA seja modificado durante o processo de reconfiguração.

- Total - Na reconfiguração Total é necessário que toda a FPGA seja modificada.

2.1.1 Aplicações de FPGA

A capacidade de reconfiguração dos FPGAs torna-os a melhor escolha e por vezes mesmo ideal para variadas aplicações. Algumas aplicações das FPGA [5] são descritas de seguida:

- Prototipagem de ASICs - A prototipagem de ASICs em FPGAs permite a aceleração do processo de validação de sistemas SOC.
- Substituição da tecnologia ASIC - Circuitos integrados implementados em FPGAs em substituição da tecnologia ASIC, que é construída por medida, permitindo maior flexibilidade.
- Processador dedicado para aceleração de aplicações - Aceleração de partes das aplicações cujo processamento em outras tecnologias seja muito intenso.

2.1.2 Arquitectura básica

A arquitectura básica de um FPGA [6] é composta, essencialmente, por três componentes:

- **Blocos Lógicos Reconfiguráveis (CLB)**
- **Blocos de entrada e saída (IOB)**
- **Matrizes de interconexão**

Os CLBs são responsáveis pela implementação de funções lógicas. A sua arquitectura inclui *Look Up Tables* (LUT) de 4 entradas que é constituída por células de armazenamento e que implementa funções lógicas combinacionais de 4 ou 6 entradas, e flip-flops.

Os IOBs são responsáveis pela ligação de sinais lógicos, provenientes dos CLB, a pinos de saída do FPGA, funcionando como um buffer de entrada e saída. Permitem uma comunicação bilateral de entrada e saída do FPGA.

As matrizes de interconexão permitem a ligação entre CLBs e IOBs através de trilhas com conexões programáveis.

Actualmente, existem várias arquitecturas de FPGAs disponíveis, variando conforme o fabricante, destacando-se os fabricados pela Xilinx e pela Altera. A Xilinx contém um conjunto de famílias de FPGAs, tendo actualmente as famílias Spartan, Spartan II, Spartan 3, Spartan 6, Virtex4, Virtex5, Virtex6, Virtex7, Kintex7, Artix7 e variadas versões dessas mesmas famílias.

2.1.3 Virtex-5

A família de FPGAs Virtex-5 [4] usa a arquitectura ASMBL (*Advanced Silicon Modular Block*) [4] de segunda geração. Contém um total de cinco sub-famílias, LX, LXT, SXT, TXT e FXT, cada uma com características diferentes permitindo lidar com diversos projectos de lógica avançada.

A arquitectura da Virtex-5 inclui lógica de fábrica de FPGA das mais avançadas e de alto desempenho, tais como, LUTs de 6 entradas, LUTs duplos de cinco entrada, Registos de Deslocamentos de 32 bits (SRL32) ou Registos de Deslocamentos duplo de 16 bits (SRL16), memória distribuída de 16 bits, para além de um conjunto de *Hard-IP*, tais como:

- Blocos de memória RAM de 32 Kbits,
- Slices DSP48E avançados de 25x18,
- Tecnologia de gestão de relógios,
- Microprocessadores de alto desempenho PowerPC 440 (apenas na sub-família FXT),

De seguida será descrito com mais detalhes dois blocos importantes para o projecto desenvolvido, *Slices* DSP48E e PowerPC.

Os slices DSP48E [4] apresentam um alto desempenho permitindo uma frequência até 550MHZ. As suas principais características são:

- implementação de multiplicadores de variáveis de 25x18 bits em complemento para dois;
- implementação opcional de adicionadores, subtratores e acumuladores;
- implementação opcional de estágios de *pipelining* para aumentar a frequência de processamento;
- implementação opcional de funcionalidades lógicas bit-a-bit.

O *Core* PowerPC 440 [4] apenas está presente na sub-família FXT. Segue uma arquitectura RISC e consegue atingir alto desempenho, até uma frequência de 550MHZ. Apresenta as seguintes características:

- Mais de 1000DMIPS por *core*
- Sete estágios de *pipelining*
- Múltiplas instruções por ciclo
- Execução fora-de-ordem

No desenvolvimento deste projecto foi utilizada a Virtex-5 da sub-família FXT, XC5VFX70T. Na tabela são apresentados os recursos disponíveis para este Virtex-5.

A escolha, para o projecto desenvolvido, de uma Virtex-5 pertencente à sub-família FXT é óbvia, visto ser a única sub-família que contém uma Microprocessador PowerPC, microprocessador esse necessário para a execução da aplicação em C do codificador G.729. No entanto, a Virtex-5 XC5VFX70T contém um número muito limitado de Blocos DSP48E (apenas 128), havendo outras Virtex-5 dessa sub-família com maior número de blocos DSP48E.

XC5VFX70T		
CLB	Array	160x38 (linha x coluna)
	slices	11200
	Máximo RAM distribuido	820 Kb
Slices DSP48E		128
Blocos BRAM	18kb	296
	32kb	148
	Máximo (kb)	5328
Blocos Processadores PowerPC		1
Ethernet MACs		4
Total de bancos I/O		19
Máximo I/O <i>user</i>		160

Tabela 2.1: Recursos da Virtex-5 XC5VFX70T [4]

2.2 G.729

G.729 [1] é um algoritmo de compressão de voz em pacotes de 10ms de duração, sob o modelo CS-ACELP (*Conjugate-Structure Algebraic-Code-Excited Linear Prediction*), aprovado pela ITU-T.

O codificador opera, na entrada, sobre sinais de fala em banda-limitada, a uma frequência de amostragem de 8kHz e codificação PCM linear de 16 bits por amostra.

O *bitrate* de saída, para o codificador original, é de 8kbit/s mas alguns anexos do algoritmo apresentam outros dois possíveis *bitrates* de saída, 6.4kbps e 11.8kbps, para pior e melhor qualidade de codificação, respectivamente. A saída do codificador é um *bitstream*.

A introdução de anexos ao algoritmo, trouxe a possibilidade de utilizar os modos de configuração DTX, ON e OFF, para ativar ou desativar o algoritmo DTX que permite reduzir a taxa de transmissão durante os períodos de silêncio.

A tabela 2.2 faz uma comparação entre o codificador original e os 11 anexos. A diferença entre o anexo A e o original deve-se à introdução de modificações no algoritmo do anexo A para diminuir a sua complexidade computacional.

G.729 apresenta um baixo *bitrate* e permite moderado atraso de transmissão, oferecendo alta qualidade de serviço, elevada robustez e melhor largura de banda pelo preço de alta complexidade de implementação. Assume grande importância em aplicações de telecomunicação, tais como, videoconferência e telefone por Internet e outros, onde o atraso e a qualidade de serviço é muito importante. Tem sido muito utilizado em aplicações VoIP, tais como, o Skype.

2.2.1 Descrição geral

O algoritmo do codificador G.729 trabalha sobre segmentos de fala de 10ms na entrada, que corresponde a 80 amostras de 16bits da fala, obtendo, assim, uma taxa de amostragem de 8000 amostras por segundos e um *bitrate* de entrada de 128kbps. Para além dos 10 ms de dados, o codificador apresenta um atraso de codificação de 5 ms, obtendo um atraso total de 15 ms.

Anexos	Características					
	Ponto-Fixa	Ponto-Flutuante	6,4kbit/s	8,0kbit/s	11,8kbit/s	DTX
Original	✓			✓		
A	✓			✓		
B	✓			✓		
C		✓		✓		
D	✓		✓	✓		
E	✓				✓	✓
F	✓		✓	✓		✓
G	✓			✓	✓	✓
H	✓		✓	✓	✓	
I	✓		✓	✓	✓	✓
C+		✓	✓	✓	✓	✓
J	✓			✓		

Tabela 2.2: G.729 original e anexos [2]

Para cada segmento de fala, o algoritmo analisa-a e extrai um conjunto de parâmetros, que são codificados e transmitidos. Posteriormente, esses parâmetros são utilizados pelo decodificador para recuperar o sinal de fala transmitido. O modelo CS-ACELP [1] baseia-se no modelo ACELP que especifica os seguintes parâmetros de codificação:

- Coeficientes do filtro de predição linear (LSP);
- Índice, ganho e sinal do dicionário adaptativo;
- Índice e ganho do dicionário fixo.

2.2.2 Codificação

A figura 2.1 mostra as etapas do algoritmo de codificação [1] do G.729.

A primeira etapa do algoritmo de codificação é o Pré-Processamento. Nessa etapa, o sinal de entrada é filtrado por um filtro passa-alto para eliminar componentes indesejáveis de baixa frequência e posteriormente, é escalado, isto é, dividido por um factor de 2, para evitar *overflow* na implementação em vírgula fixa.

De seguida, a análise LP é efectuada para cada segmento do sinal de 10 ms, para determinar os coeficientes do filtro LP. A janela de análise LP contém um total de 240 amostra de fala, sendo 120 amostras de segmentos de fala anteriores, 80 amostras do segmento actual e 40 amostras do segmento futuro. Os coeficientes LP são obtidos efectuando uma autocorrelação dessa janela, seguida da aplicação do algoritmo de Levinson-Durbin. Posteriormente, esses coeficientes são convertidos para coeficientes LSP (*line spectrum pairs*) e quantizados através da quantização vectorial preditiva em 2 estágios de 18 bits no total. Para determinar o sinal de excitação $u(n) = Ga * ua(n) + Gf * uf(n)$ utiliza-se uma abordagem Análise-por-Síntese, que consiste em minimizar o erro entre o sinal reconstruído e o sinal original, filtrando o sinal de erro com um filtro de peso perceptivo. Os parâmetros da excitação são determinados por cada subframe de 5

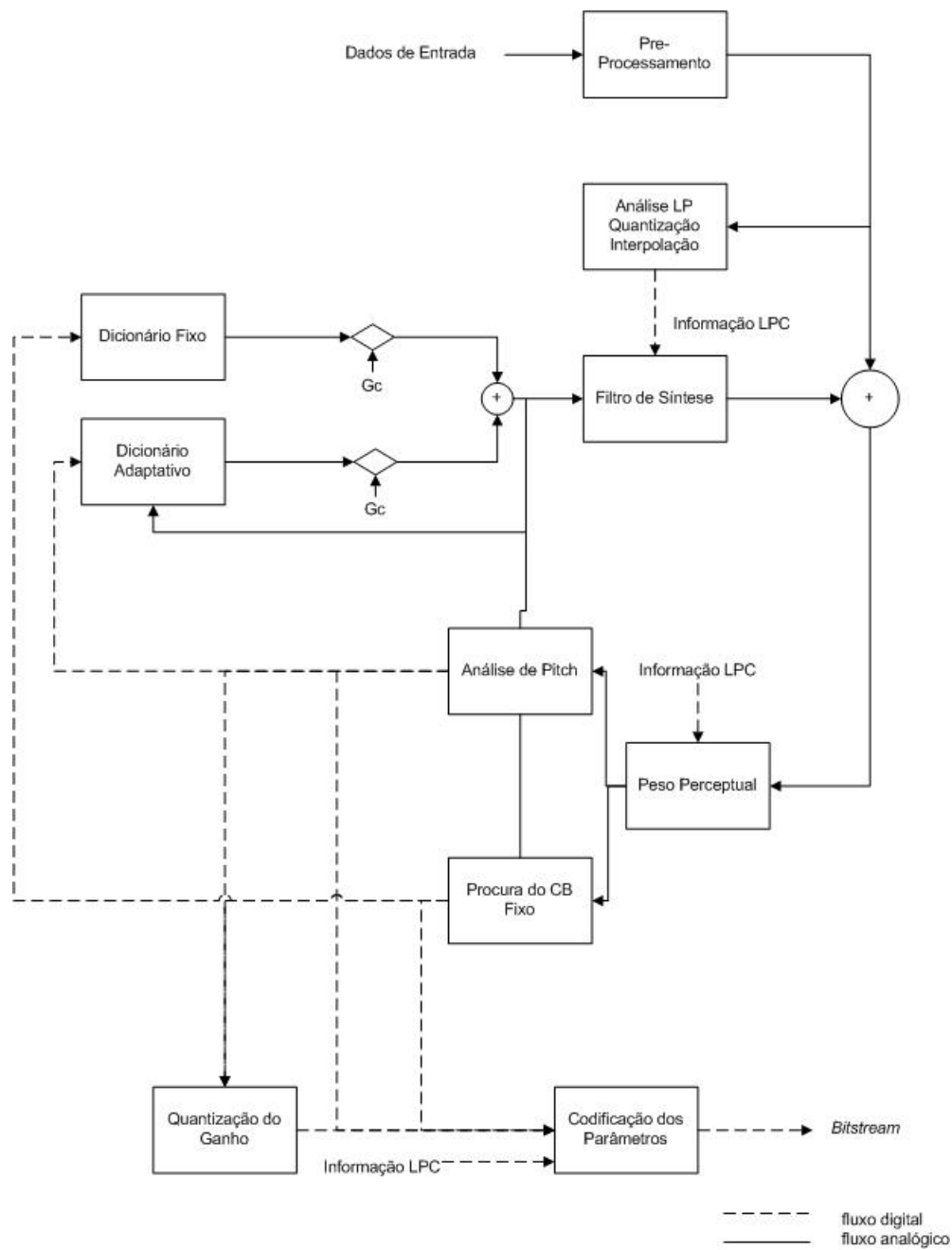


Figura 2.1: Processo de Codificação [1]

ms. A cada segmento de 10 ms, determina-se o atraso de *pitch* pela análise em ciclo aberto, que é baseado no sinal da saída do peso perceptivo. De seguida, os seguintes passos são efectuados para cada subframe de 5 ms:

- Obtém-se o sinal alvo $x(n)$, filtrando o resíduo LP $r(n)$ sobre o filtro de síntese $W(z)/A(z)$.
- Obtém-se o impulso de resposta $h(n)$ do filtro de síntese.
- A análise de ciclo fechado é feita para se determinar os parâmetros do dicionário adaptativo, através da busca em torno do atraso de *pitch* em ciclo aberto. Um atraso fraccionário de *pitch* com resolução de 1/3 é utilizado.
- O sinal alvo é atualizado, subtraindo a contribuição do dicionário adaptativo. O novo sinal alvo $x'(n)$ é usado na procura do dicionário fixo para determinar o ganho e a excitação óptima. Um dicionário algébrico de 17 bits é usado na obtenção da parcela de excitação do dicionário fixo.
- Os ganhos do dicionário adaptativo e fixo são submetidos a uma quantização vectorial com 7 bits.
- E finalmente, atualiza-se o dicionário adaptativo, usando o sinal excitado obtido $u(n)$.

2.2.3 Decodificação

A figura 2.2 mostra o processo de decodificação [2]. O decodificador recebe os parâmetros transmitidos (coeficientes LSP, índice e ganho do dicionário fixo e índice, ganho e sinal do dicionário adaptativo) e através deles recupera o sinal transmitido.

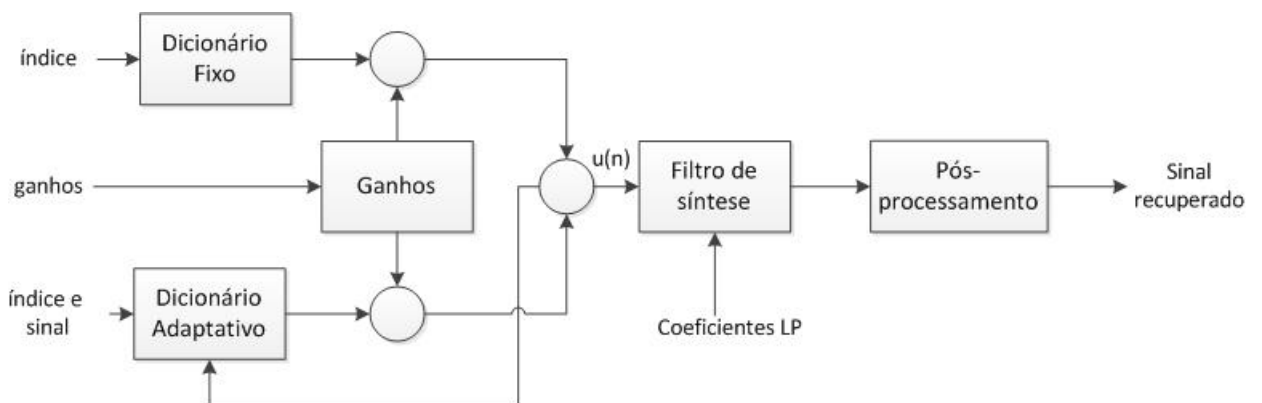


Figura 2.2: Processo de decodificação [2]

Para cada segmento de 10 ms, os coeficientes LSP são decodificados, interpolados e convertidos para coeficientes LP. Com estes coeficientes é construído o filtro de síntese. De seguida, para cada subframe de 5 ms são efectuados os seguintes passos:

- Os parâmetros do dicionário adaptativo e do dicionário fixo são decodificados;

- A excitação do filtro de síntese $u(n)$ é construída;
- Atualiza-se o dicionário adaptativo com a incorporação de $u(n)$;
- O sinal de fala é reconstruído, filtrando o sinal de excitação pelo filtro de síntese;
- E por último, o sinal reconstruído passa por um pós-processamento, que inclui um pós-filtro, seguido de um filtro passa-alto e por uma operação de escalamento.

2.3 Fluxo de Projeto para Aplicações Audio

A figura 2.3 mostra o fluxo de projeto tipicamente utilizado na implementação heterogénea de aplicações audio [3]. O fluxo de projeto foi proposto pelo projecto REFLECT e utilizado nesta implementação do codificador G.729 em FPGA.

Este fluxo de projeto tem por base os códigos fontes da aplicação implementada, geralmente, em linguagem C. Um conjunto de ficheiros contendo amostras de audio são utilizados para extrair os valores da entrada da aplicação, com o objectivo de executar a aplicação e colher resultados para validar o sistema desenvolvido.

A otimização da representação dos dados permite não só a simplificação do Software mas também a simplificação na implementação em Hardware. Consiste na redução do número de bits das variáveis ou na conversão para tipos de dados mais simples. O teste de validação de precisão é efectuado comparando os resultados obtidos com e sem optimização. Uma margem de erro aceitável e que permite manter o sistema validável é definida e, uma vez que, o erro entre as duas implementações se mantém dentro da margem, a nova aplicação deve ser considerada para as etapas seguintes.

O *profiling* da aplicação permite determinar as funções críticas, isto é, as funções com tempos de execução muito elevados. Muitas aplicações audio têm de operar em tempo real. Sendo assim, as funções críticas tornam-se candidatas a aceleração em Hardware. Após determinar quais são as funções críticas, decide-se quais as funções a serem implementadas em Hardware, fazendo, assim, a partição entre Hardware e Software. A implementação em Hardware é feita através de uma linguagem de descrição de Hardware, por exemplo, Verilog, e através de exploração de técnicas, tais como, o Paralelismo e o *Pipelining*, pode-se alcançar melhores desempenhos que em Software. A aplicação é modificada, substituindo as funções críticas por um mecanismo que permita a comunicação com o Hardware.

Na plataforma de desenvolvimento, a aplicação é executada num processador e os módulos das funções mapeadas na FPGA e, com o auxílio de um *device driver*, a comunicação entre eles é efectuada. Uma vez implementados na plataforma, a depuração do sistema é efectuada para identificar e posteriormente serem corrigidos eventuais erros na implementação, tanto em Hardware como nas modificações em Software.

Após a depuração, procede-se à validação, colheita e análise dos resultados. Nesta etapa, faz-se a validação final da correcção dos valores produzidos na saída, é medida a aceleração (ou não)

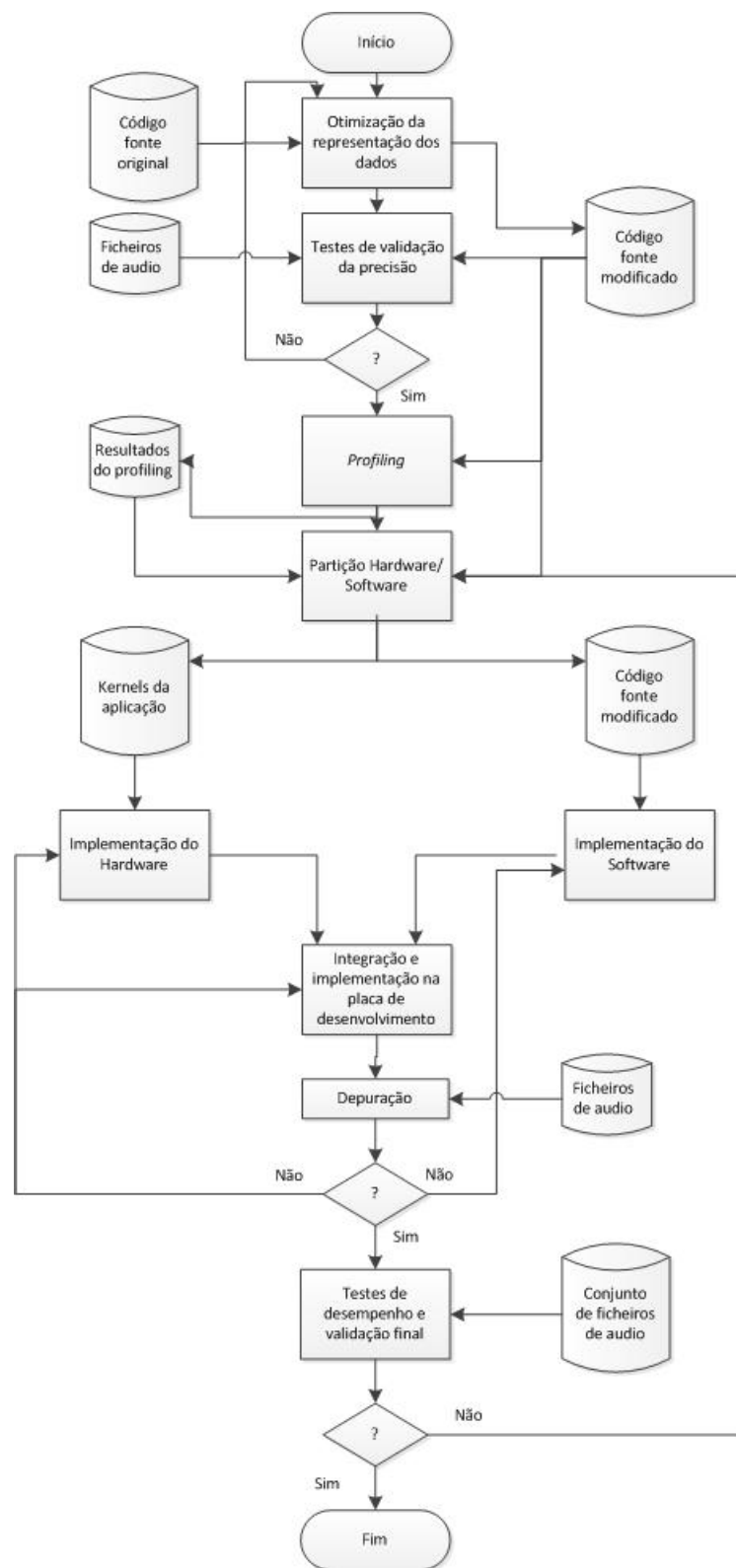


Figura 2.3: Fluxo de projecto para aplicações audio segundo [3]

obtida face a implementação só em Software e verifica-se se a aplicação é executada em tempo real. Caso não se verifique a execução da aplicação em tempo real, mais funções críticas podem ser seleccionadas para serem implementadas em Hardware.

Na implementação em FPGA do G.729 foi seguido esse fluxo de projeto com excepção das etapas de otimização da representação dos dados e teste de validação de precisão porque já foi fornecido o código fonte, em linguagem C, da aplicação, com otimização das variáveis em ponto-fixe. As funções críticas foram implementadas em Verilog.

2.3.1 Plataforma de desenvolvimento

A plataforma de desenvolvimento utilizada é o ML507 [8]. A plataforma é utilizada, como descrito anteriormente, para implementação, validação final e extração de resultados do sistema implementado. Esta plataforma contém um FPGA Virtex-5 [4] da Xilinx (XC5VFX70T-1FFG1136) que contém um processador PowerPC 440. Para além disso, a plataforma contém um conjunto de outros periféricos [8] importantes no auxílio da implementação, tais como:

- Porta série RS232,
- Porta de configuração JTAG,
- Transceptor PHY Ethernet de três velocidades,
- DDR2 de 256MB e 16 *bit* de largura.

2.3.2 Ferramentas

As seguintes ferramentas foram utilizadas no desenvolvimento do projecto:

- GNU Gprof [9] - É uma ferramenta de análise de programas escritos em linguagens de programação tal como o C. O seu principal propósito é a análise computacional de partes de código, permitindo obter quanto tempo uma parte do programa ou uma função gasta na execução. Com esta informação, pode-se saber que partes do programa são muito lentos ou gastam muito tempo e, assim, tomar uma ação para obter o tempo de execução desejado. No projecto, foi utilizado no *profiling* da aplicação do G.729, permitindo determinar quais as funções a serem implementadas em Hardware.
- ModelSim [10] - É um simulador e ambiente de depuração para as linguagens de descrição de Hardware Verilog e VHDL e para SystemC, muito utilizado no design de projectos para FPGA e ASIC. Fornece um ambiente gráfico amigável, permitindo visualizar as entradas e saídas dos módulos em formato de ondas. Neste projecto foi utilizado na simulação e análise dos módulos Verilog desenvolvidos.
- Xilinx ISE Design Suite 12.3

- Xilinx Platform Studio (XPS) [11] - É a ferramenta do Xilinx ISE que permite construir o sistema base de Hardware, criar e acrescentar ao sistema base os nossos próprios IP Cores, e gerar e descarregar o *bitstream* para a placa de desenvolvimento. Apresenta um sofisticado ambiente gráfico que auxilia e guia o desenvolvedor ao longo de todos os passos para a criação do seu sistema.
- Project Navigator [12] - É a ferramenta do Xilinx ISE que permite a gestão dos ficheiros HDL que compõe o design desenvolvido e a sintetização do design na plataforma alvo.
- Xilinx Software Development Kit (XSDK) [13] - É um ambiente gráfico de desenvolvimento de software embarcado que suporta várias funcionalidades: profiling e optimização de software, depuração de software, drives e bibliotecas de suporte de comunicação com o Hardware, interpretação do sistema criado em XPS, configuração de vários parâmetros e criação do pacote de suporte a implementação na placa de desenvolvimento.
- Embedded Linux Development Kit (ELDK) [14] - Inclui as ferramentas de desenvolvimento em ambiente GNU, tais como compiladores, para além bibliotecas e ferramentas de suporte ao funcionamento em sistemas alvos que correm Linux.

Capítulo 3

Análise da Aplicação

O trabalho desenvolvido tem por base a aplicação em linguagem C do codificador de fala G.729. É utilizado o *ITU-T G.729 Annex I Software Package Release 2*, versão 1.2, última modificação em Outubro de 2006, que implementa o algoritmo de compressão em ponto-fixa segundo o anexo I.

A aplicação recebe um ficheiro que contém dados de fala e de onde são extraídos os valores da entrada do algoritmo. Para além disso, a aplicação permite seleccionar o *bit rate* pretendido, 6.4kbit/s, 8.0kbit/s ou 11.8kbit/s, e activar ou desactivar a flag DTX. Em caso de não ser especificado o *bit rate* e/ou a flag DTX são assumidos, por omissão, o *bit rate* 8.0kbit/s e DTX desativado.

Quando especificado, a aplicação também permite a variação embarcada do *bit rate* (dentro dos três valores possíveis), isto é, permite que cada segmento de fala tenha um *bit rate* de saída diferente.

O bitstream da saída é guardado num ficheiro binário.

É de salientar que o uso de ficheiros para ler os valores de entrada e escrever os valores de saída é para fins de análise. Em caso de codificação em tempo real seria necessário uma *interface* de comunicação com o sistema que fornece os dados de entrada e com o sistema para onde a aplicação fornece os dados.

A aplicação foi executada na plataforma de desenvolvimento ML507 que corre Linux e duas importantes análises foram efectuadas:

- *Profiling* da aplicação.
- Determinação do tempo de codificação de um segmento de fala.

Dado que a codificação tem de ser efectuada em tempo real, o atraso de codificação não deve exceder os 15 ms por segmento de fala, 10 ms de dados de entrada e mais 5 ms para execução do algoritmo. A tabela 3.1 contém os tempos que a aplicação demora a processar um segmento de fala (tempo de codificação) para os diferentes valores de *bit rate* e configuração DTX. A aplicação foi compilada para nível de optimização 2, os dados de entrada foram extraídos do ficheiro *speech* e foi utilizado um temporizador para medir os tempos. A estratégia usada para medir o tempo de codificação será apresentada com mais detalhe no capítulo 7.

Pela análise dos resultados, verifica-se que estes valores encontram-se abaixo do limite máximo de 5 ms para a execução do algoritmo.

DTX	Bitrate(kbit/s)	tempo (us)
activo	6.4	1067
activo	8.0	1092
activo	11.8	3942
não activo	6.4	2666
não activo	8.0	3228
não activo	11.8	3874

Tabela 3.1: Tempo de codificação de um segmento de fala

A ferramenta Gprof foi utilizada para o *Profiling* da aplicação. Nesta análise, os valores de entrada foram extraídos dos ficheiros *speech* e *dm19.8*. A aplicação foi executada 5 vezes para cada uma das 8 combinações de ficheiro de entrada, *bit rate*, DTX e variação embarcada de *bit rate*. Para além de outras informações, a ferramenta dá-nos 4 informações importantes: o número de chamadas, o tempo de execução por chamada, o tempo total de execução, a percentagem do tempo de execução de cada função.

As tabelas 3.2, 3.3 e 3.4 mostram o resultado do *Profiling* para as quatro funções com maiores percentagens de execução, para os três níveis de compilação O1, O2 e O3.

Função	Número de Chamadas	Tempo de execução por chamada(us)	tempo total de execução(s)	Tempo total (percentagem)
Lag-Max	67380	178.84	12.05	8.87
Residue	295050	37.65	11.11	8.18
Syn-filte	366480	27.56	10.10	7.43
cor-h-vec	918240	9.26	8.50	6.25

Tabela 3.2: Resultados do *Profiling* para nível de optimização 1

Função	Número de Chamadas	Tempo de execução por chamada(us)	tempo total de execução(s)	Tempo total (percentagem)
Residue	295050	29.01	8.56	7.46
Lag-Max	67380	124.37	8.38	7.31
Syn-filte	366480	21.58	7.91	6.9
cor-h-vec	918240	7.39	6.79	5.92

Tabela 3.3: Resultados do *Profiling* para nível de optimização 2

Pelas tabelas, verifica-se que as funções *Lag_max* e *Residue* apresentam as piores percentagens de tempos de execução. No total, as duas funções somam 17.05%, 14.77% e 16.70% do tempo total de execução da aplicação, para os três níveis de optimização O1, O2 e O3 respectivamente. Por serem as funções com maior consumo de tempo, foram escolhidas para serem implementadas em Hardware. Nas duas secções seguintes, é feita a análise detalhada dessas funções.

Função	Número de Chamadas	Tempo de execução por chamada(us)	tempo total de execução(s)	Tempo total (porcentagem)
Lag_Max	67380	154.94	10.44	8.99
Residue	295050	30.3	8.95	7.71
ACELP_10i40_35bits	38260	223.21	8.54	7.35
Syn-filte	366480	21.64	7.93	6.83

Tabela 3.4: Resultados do *Profiling* para nível de otimização 3

3.1 Descrição Detalhada das Funções

3.1.1 *Residue*

A função *Residue* calcula a convolução de ordem M e lg posições, entre os vectores $x(n)$ e $a(n)$, tal como apresentado na seguinte equação

$$y(k) = \sum_{n=0}^M a(n)x(k-n) \quad (3.1)$$

com $0 \leq k < lg$

A função recebe na entrada os dois vectores $x(n)$ e $a(n)$, os dois valores limite M e lg , escreve os resultados no vector $y(n)$ e não retorna nenhum valor. No texto seguinte são mostrados os argumentos da função. A expressão Word16 indica que o tamanho é de 16 bits e que o tipo de dados é *Integer* com sinal.

Texto 3.1: Argumentos da função *Residue*

```
void Residue(Word16 m, Word16 a[], Word16 x[], Word16 y[], Word16 lg)
```

A função contém dois ciclos *for*, um externo com um total de lg repetições em que cada repetição desloca uma posição no vector $x(n)$ e no vector de saída $y(n)$, e um ciclo interno que faz $M+1$ multiplicações e acumulações. O resultado final das multiplicações e acumulações é de 32 bits, mas é truncado e arredondado para o resultado de 16 bits. De seguida é guardada na posição do vector $y(n)$ definido pelo ciclo externo. O texto mostra o segmento de código que implementa a função *Residue*.

```
for (i = 0; i < lg; i++)
{
    s = L_mult(x[i], a[0]);
    for (j = 1; j <= m; j++)
        s = L_mac(s, a[j], x[i-j]);

    s = L_shl(s, 3);
    y[i] = round(s);
}
```

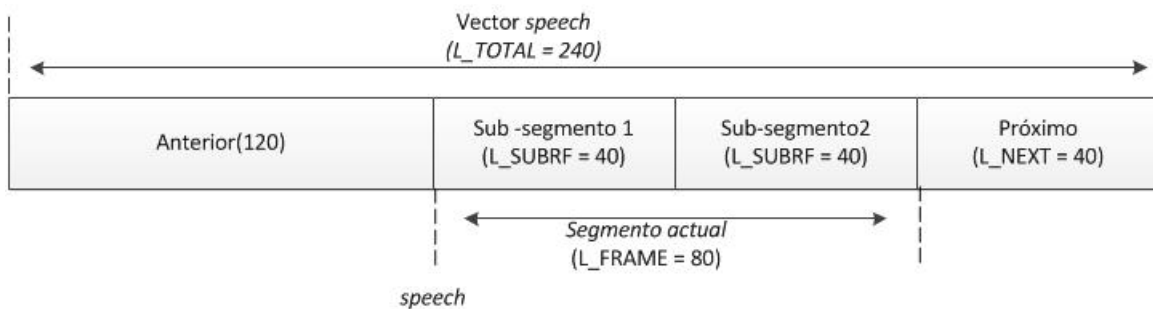


Figura 3.1: Vector global de fala *speech*

A função *Residue* é chamada nove vezes ao longo da aplicação, cinco vezes dentro da função *set_lpc_mode* e as restantes dentro da função principal de codificação *Coder_ld8c*. A função *set_lpc_mode* é chamada uma única vez dentro de *Coder_ld8c*.

A função *set_lpc_mode* contém os argumentos *signal_ptr*, *a_fwd* e *a_bwd* e outros, e quando é chamada recebe o vector de fala global *speech*, a janela *A_t_fwd_q* e a janela *A_t_bwd* respectivamente nestes três argumentos. O vector de fala *speech* tem uma dimensão total de 240 valores, sendo 80 valores actuais dividido em dois segmentos de 40 valores, 40 valores futuros e 120 valores passados. A figura 3.1 mostra esse vector.

Nas cinco chamadas à função, em três delas *M* assume o valor 30 e nas outras duas, o valor 10. Em todas as cinco chamadas o vector $x(n)$ aponta para o vector *speech*. Para *M* igual a 30, a janela $a(n)$ é um apontador para o vector *A_t_bwd* que tem um total de 62 valores. Para *M* igual a 10, a janela $a(n)$ aponta para o vector *A_t_fwd_q* que tem um total de 22 valores.

A figura 3.2 mostra o alcance ao vector *speech* para cada uma das 3 chamadas com *M* igual a 30. Na primeira chamada a função trabalha sobre os 80 valores presentes, na segunda chamada trabalha sobre o primeiro sub-segmento e na terceira chamada sobre o segundo sub-segmento. Em todos os três casos são necessários mais 30 valores anteriores. Relativamente ao vector *A_t_bwd*, a segunda chamada à função trabalha sobre os primeiros 31 valores desse vector e nas outras chamadas trabalha sobre os restantes 31 valores. O valor de *lg* é *L_FRAME* (80) na primeira chamada à função. Isto é, são calculadas 80 convoluções; na segunda e terceira chamada à função são calculadas *L_SUBFR* (40) convoluções.

Para *M* igual a 10, a primeira e segunda chamada à função trabalha sobre o primeiro e segundo sub-segmento, respectivamente, tal como é mostrado na figura 3.3. São necessários mais 10 valores anteriores. De forma semelhante, a primeira chamada à função trabalha sobre os primeiros 11 valores do vector *A_t_fwd_q* e a segunda chamada à função trabalha sobre os restantes 11 valores desse vector. Em ambas as chamadas à função são calculadas *L_SUBFR*(40) convoluções.

Dentro da função *Coder_ld8c*, a função é chamada em quatro locais, sendo eles dentro de ciclos *for*. Em dois locais, o vector $x(n)$ aponta para o vector *speech* e nos outros dois restantes aponta para o vector *error* que é calculado antes da chamada à função. O valor de *M* varia (10 ou 30) conforme um conjunto de condições verificadas antes da chamada à função e a janela $a(n)$

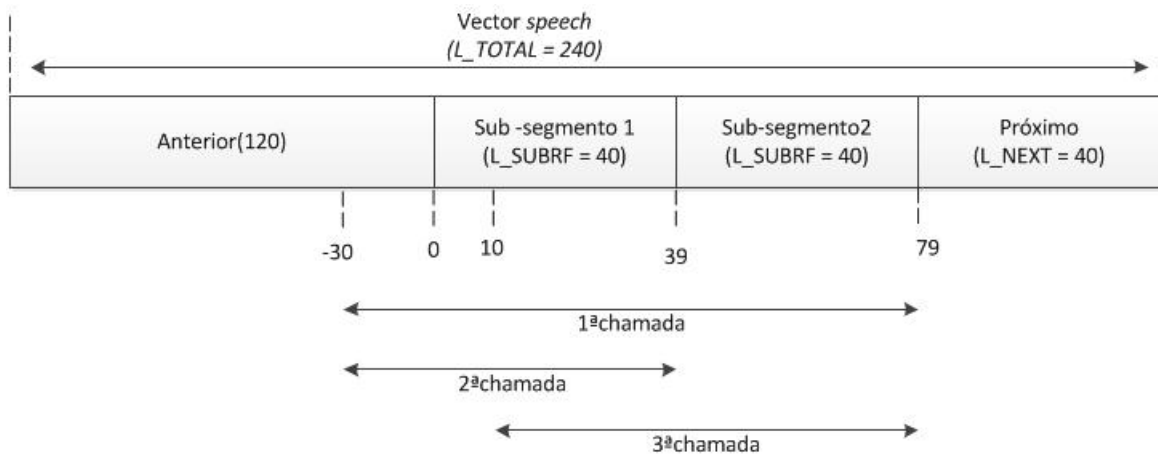


Figura 3.2: Acesso ao vector speech nas três chamadas de Residue para M igual a 30

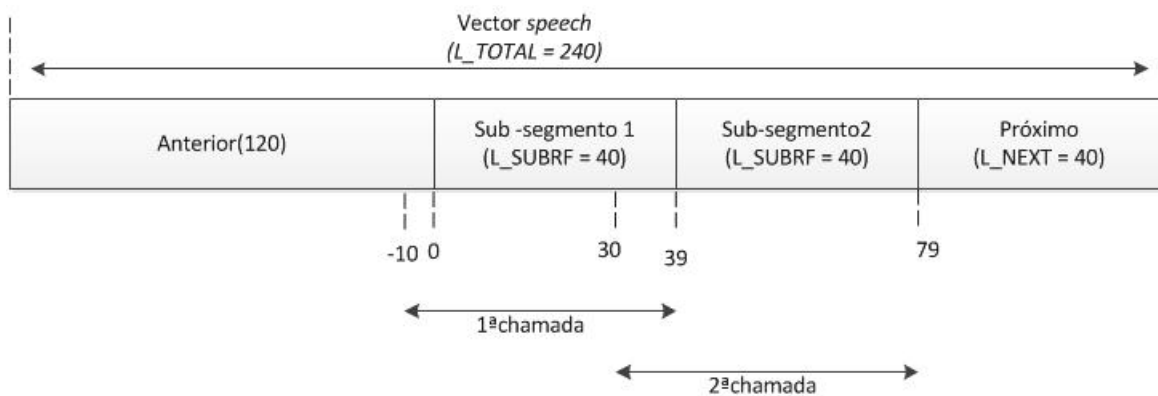


Figura 3.3: Acesso ao vector speech nas três chamadas de Residue para M igual a 10

também varia.

Na primeira vez que é chamada, a posição do vector *speech* varia com o índice do ciclo *for*, o valor de $lg = L_SUBFR = 40$ e o valor de *M* depende de *m_ap*.

```
for (i=0; i<2; i++) {
    ...
    Residue(m_ap, Ap1, &speech[i*L_SUBFR], &wsp[i*L_SUBFR], L_SUBFR);
    ...
}
```

Na segunda vez, em vez do vector *speech* é usado o vector *error*, nada depende do índice do ciclo, o valor de *M* é 10 e o valor de *lg* é 40.

```
Residue(M, Ap1, error, xn, L_SUBFR);
```

As duas chamadas seguintes são dentro do mesmo ciclo *for*, uma trabalha sobre o vector *speech* e o outro sobre *error*. Em ambas *lg* é 40.

```
for (i_subfr = 0; i_subfr < L_FRAME; i_subfr += L_SUBFR) {
    ...
    Residue(m_aq, pAq, &speech[i_subfr], &exc[i_subfr], L_SUBFR);
    ...
    Residue(m_ap, Ap1, error, xn, L_SUBFR);
}
```

3.1.2 Lag_max

A função *Lag_max* calcula a máxima auto-correlação normalizada do vector *signal* entre os limites *lag_max* e *lag_min*. Para além disso, calcula a posição onde ocorreu a máxima correlação. As duas equações seguintes mostram a operação da função.

$$R(k) = \sum_{n=lag_min}^{lag_max} signal(n)signal(n-k) \quad (3.2)$$

$$R(pmax) = \frac{R(pmax)}{\sqrt{\sum_n signal^2(n-pmax)}} \quad (3.3)$$

A função recebe na entrada o vector *signal*, os dois valores limites superiores e inferiores *lag_max* e *lag_min*, a dimensão da correlação *L_frame*, escreve os resultados no apontador *cor_max* e retorna a posição onde ocorreu a máxima correlação. No texto seguinte é mostrado os argumentos da função. A expressão *Word16* indica que o tamanho é de 16 bits e que o tipo de dados é *Integer* com sinal.

```
static Word16 Lag_max(Word16 signal[], Word16 L_frame, Word16 lag_max,
    Word16 lag_min, Word16 *cor_max)
```

A função começa por determinar a máxima correlação e a posição onde esta ocorreu. Esta parte contém dois ciclos *for*, um ciclo interno e um externo. O ciclo externo faz o varrimento começando em *lag_max* e diminuindo até atingir *lag_min*, sendo que para cada iteração a correlação é feita entre o vector *signal* cuja primeira posição é 0 e o vector *signal* cuja primeira posição é -i, que é o índice do ciclo. O ciclo interno faz *L_frame* multiplicações e acumulações obtendo, assim, um resultado da correlação. Depois esse valor é comparado com o resultado obtido anteriormente e, caso seja maior, este é considerado a máxima correlação. Ao chegar ao fim do varrimento do ciclo externo, obtém-se a máxima correlação e a posição onde esta ocorreu, que é dada pelo índice do ciclo *for*. Tal pode ser visto na listagem seguinte.

```
for (i = lag_max; i >= lag_min; i--)
{
    p = signal;
    p1 = &signal[-i];
    t0 = 0;

    for (j=0; j<L_frame; j++, p++, p1++)
        t0 = L_mac(t0, *p, *p1);

    L_temp = L_sub(t0, max);
    if (L_temp >= 0L)
    {
        max = t0;
        p_max = i;
    }
}
```

Depois de obter a máxima correlação é calculada a potência do sinal na posição onde ocorreu a máxima correlação.

```
t0 = 0;
p = &signal[-p_max];
for (i=0; i<L_frame; i++, p++)
    t0 = L_mac(t0, *p, *p);
```

Por último, com o valor da máxima correlação e com a potência do sinal, é calculada a máxima correlação normalizada. Esta é guardada no apontador *cor_max* e a função retorna a posição onde ocorreu a máxima correlação.

```
t0 = Inv_sqrt(t0);

L_Extract(max, &max_h, &max_l);
L_Extract(t0, &ener_h, &ener_l);

t0 = Mpy_32(max_h, max_l, ener_h, ener_l);
*cor_max = extract_l(t0);

return (p_max);
```

A função *Lag_max* é chamada três vezes dentro da função *Pitch_ol*. A função *Pitch_ol* tem como argumento o vector *signal*, os dois limites máximos e mínimos *pit_max* e *pit_min* e o valor *L_frame*. Os argumentos são mostrados de seguida.

```
Word16 Pitch_ol(Word16 signal[], Word16 pit_min, Word16 pit_max,
                Word16 L_frame)
```

A função *Pitch_ol* é chamada uma única vez dentro da função principal de codificação, *Coder_ld8c*. É passado à função o vector *wsp* pela qual será calculada a auto-correlação, e os parâmetros fixos *PIT_MIN* (20), *PIT_MAX*(143) e *L_FRAME*(80).

```
Pitch_ol(wsp, PIT_MIN, PIT_MAX, L_FRAME);
```

A função *Pitch_ol* começa por verificar se há risco de *overflow* no vector *signal*, e caso haja efectua um escalonamento sobre os valores desse vector. O vector *scal_sig* é construído com os valores de *signal*. Este vector tem uma dimensão de 223 valores com índice variando entre [-143:79].

De seguida, é chamada três vezes a função *Lag_max*, tal como é mostrado de seguida.

```
j = shl(pit_min, 2);
p_max1 = Lag_max(scal_sig, L_frame, pit_max, j, &max1);

i = sub(j, 1); j = shl(pit_min, 1);
p_max2 = Lag_max(scal_sig, L_frame, i, j, &max2);

i = sub(j, 1);
p_max3 = Lag_max(scal_sig, L_frame, i, pit_min, &max3);
```

A auto-correlação é efectuada sobre o vector *scal_sig*. O limite do ciclo interno de *Lag_max* é o mesmo para as três chamadas, *L_frame*(80). Os limites inferiores e superiores do ciclo externo são $4 * pit_min = 4 * 20 = 80$ e $pit_max = 143$ para a primeira chamada, $2 * pit_min = 2 * 20 = 40$ e $4 * pit_min - 1 = 79$ para a segunda chamada e $pit_min = 20$ e $2 * pit_min - 1 = 39$ para a terceira chamada. Cada um dos resultados de máxima correlação é guardado num apontador diferente, tal como acontece com a posição que são guardados em variáveis diferentes.

A figura 3.4 mostra a região de cálculo da máxima auto-correlação para as três chamadas à função. A correlação é sempre efectuada entre o vector *signal*[0 : 79] e uma parte deslocada do mesmo vector. Visto que são efectuadas 80 multiplicações e acumulação, a primeira chamada abrange [-143:0], a segunda chamada [-79:40] e a última chamada [-39:60].

Na função *Pitch_ol*, após serem calculadas as três correlações para cada uma das secções, é escolhido o menor deles e esse valor é retornado pela função.

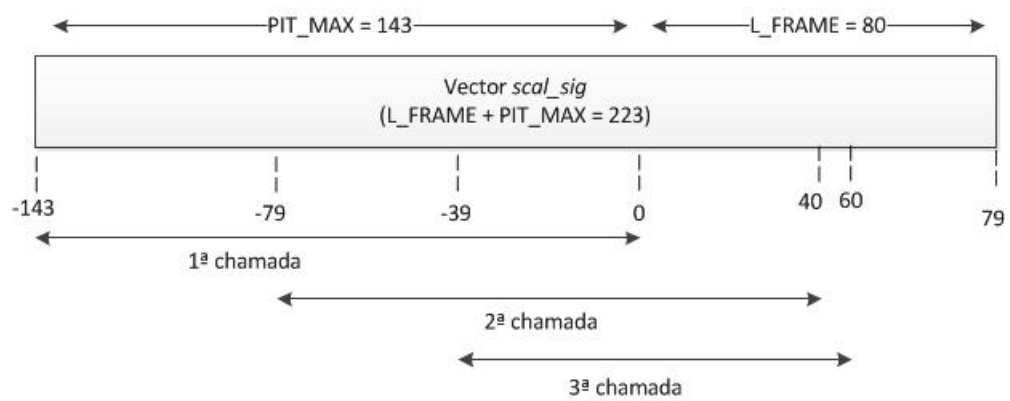


Figura 3.4: Regiões de calculo de Lag_max para as três chamadas a função

Capítulo 4

Aspectos Gerais

Neste capítulo são descritos aspectos gerais e de alto nível da implementação do projecto

4.1 Implementação em Hardware

Após o *profiling* da aplicação em C do G.729, foram identificadas quatro funções com tempos de execução elevadas e duas dessas funções foram escolhidas para serem implementadas em Hardware, *Residue* e *Lag-Max*.

A implementação dessas funções em Hardware foi baseada num conjunto de modelos de Hardware, propostos em [15].

Para a função *Residue* foram implementados quatro módulos:

- *Residue10* - Este módulo implementa a função *Residue* para *lg* variável e *M* igual a 10, com um total de 11 multiplicadores. As multiplicações são completamente paralelizadas e a técnica de *pipelining* foi utilizada, e com isso, cada resultado, após a latência introduzida pelo *pipelining*, é colocado à saída a cada ciclo de relógio.
- *Residue30* - Este módulo implementa a função *Residue* para *lg* variável e *M* igual a 30, com um total de 31 multiplicadores. À semelhança do módulo anterior, as multiplicações são completamente paralelizadas e cada resultado é colocada a saída do módulo, após a latência introduzida pelo *pipelining*, a cada ciclo de relógio.
- *Residue30_10* - Este módulo implementa a função *Residue* para *lg* variável e *M* igual a 30, com um total de 11 multiplicadores. Ao contrário dos módulos anteriores, as multiplicações não são completamente paralelizadas, e cada resultado é colocado a saída do módulo a cada 3 ciclos de relógio. Apesar deste módulo ser desvantajoso, em termos de desempenho temporal, em relação ao módulo *Residue30*, gasta menos multiplicadores, recurso este muito limitado na FPGA utilizada.

- *Residue* - É um módulo genérico que implementa a função *Residue* para *lg* variável e tanto para *M* igual a 10 como para *M* igual a 30, com um total de 11 multiplicadores. O funcionamento para *M* igual a 10 ou *M* igual a 30 é definido através de uma entrada do módulo. Apresenta um funcionamento semelhante ao *Residue10* e *Residue30_10*.

Com estes quatro módulos da função *Residue*, três casos de teste na placa de desenvolvimento são possíveis:

- 1º teste - Com os módulos *Residue10* e *Residue30*.
- 2º teste - Com os módulos *Residue10* e *Residue30_10*.
- 3º teste - Com o módulo *Residue*.

Para a função *Lag-Max*, quatro módulos foram implementados:

- *Lag_max_sp1* - Este módulo implementa a função *Lag-Max* para *L_FRAME* igual a 80 e os limites *lag_max* igual a 143 e *lag_min* igual a 80. Utiliza um total de 20 multiplicadores e cada resultado é colocado à saída a cada 8 ciclos de relógio.
- *Lag_max_sp2* - Este módulo implementa a função *Lag-Max* para *L_FRAME* igual a 80 e os limites *lag_max* igual a 79 e *lag_min* igual a 40. Utiliza um total de 20 multiplicadores e cada resultado é colocado à saída a cada 8 ciclos de relógio.
- *Lag_max_sp3* - Este módulo implementa a função *Lag-Max* para *L_FRAME* igual a 80 e os limites *lag_max* igual a 39 e *lag_min* igual a 20. Utiliza um total de 20 multiplicadores e cada resultado é colocado à saída a cada 8 ciclos de relógio.
- *Lag_max* - Este módulo implementa a função *Lag-Max* para *L_FRAME* igual a 80 e para cada uma das três regiões de cálculo, sendo que, os valores de *lag_max* e *lag_min* são entradas do módulo. Utiliza um total de 40 multiplicadores e cada resultado é colocado à saída a cada 4 ciclos de relógio.

Com estes quatro módulos da função *Lag-Max*, dois casos de teste na placa de desenvolvimento são possíveis:

- 1º teste - Com os módulos *Lag_max_sp1*, *Lag_max_sp2* e *Lag_max_sp3*.
- 2º teste - Com o módulo *Lag_max*.

Para a implementação dos módulos, foi utilizada a linguagem de descrição de Hardware Verilog na versão mais recente verilog-2001. Esta versão do Verilog trouxe muitas vantagens e simplificações ao projecto face à utilização de outras versões mais antigas do Verilog, por exemplo, Verilog-1995. Destaque-se duas principais vantagens:

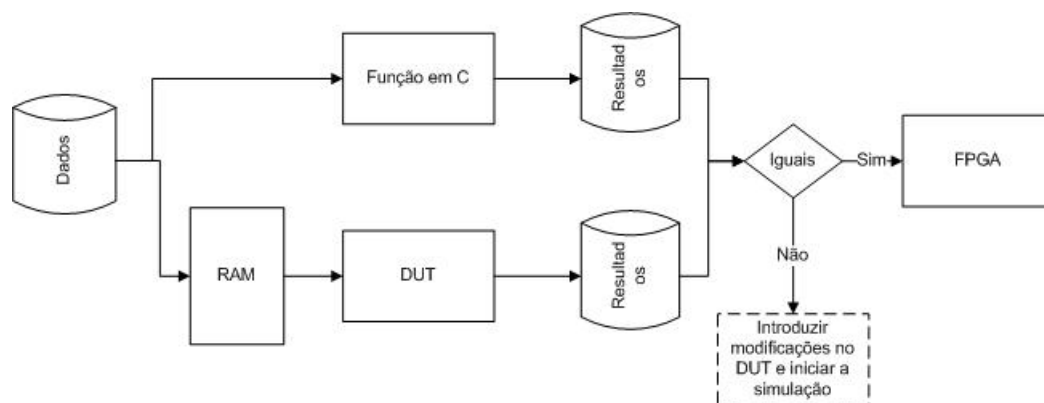


Figura 4.1: Ambiente de Verificação

- O Verilog-2001 permite definir variáveis do tipo "signed" que são interpretadas pelas ferramentas de simulação e de síntese como contendo números com sinal em complemento para dois.
- O Verilog-2001 permite fazer a comparação de números com sinais em complemento para dois através dos operadores «<» e «>».

No capítulo 5 é descrita, detalhadamente, a implementação de cada módulo em Verilog, a criação do correspondente *IP Core* e integração dos módulos no *user logic*, e cada um dos casos de teste.

4.2 Ambiente de Verificação

A figura 4.1 mostra o ambiente de verificação dos módulos desenvolvidos em Verilog. A verificação consiste, no sentido geral, na comparação entre os resultados de simulação dos módulos desenvolvidos e os resultados obtidos após execução da correspondente função em C.

Um conjunto de ficheiros contendo dados de simulação foi usado na verificação sendo que cada teste de verificação incide sobre um ficheiro. Estes ficheiros contêm, na sua maioria, dados aleatórios, excepto em alguns casos cujos dados foram escolhidos para simular casos específicos e críticos.

Na simulação do DUT (*Device Under Test*), os dados são guardados em memória RAM, de onde o DUT lerá esses dados e um conjunto de outros parâmetros são passados ao DUT conforme for o módulo a testar. Após a simulação os resultados são guardados noutro ficheiro. A ferramenta ModelSim foi usada na simulação dos módulos. Numa fase inicial foi necessário também a análise das formas de onda para uma melhor verificação do DUT.

A função em C correspondente ao DUT foi separada da aplicação do G.729 e foi criada uma nova aplicação contendo apenas a função e outras funções que esta herda. A função é executada e os resultados guardados num ficheiro. Os ficheiros contendo resultados da simulação do DUT e

da execução da função são comparados e caso houver diferenças entre elas, procede-se à análise detalhada do Verilog e à análise das formas de onda, para encontrar o problema e assim corrigi-lo.

Após verificar-se que os resultados da simulação do DUT e da execução da função são iguais para todos os diferentes ficheiros de entrada, a próxima etapa é a implementação na placa de desenvolvimento.

4.3 Implementação na placa de desenvolvimento

Para a construção do sistema base de Hardware e integração dos módulos desenvolvidos, foi utilizada a ferramenta *XPS* do *ISE Design Suite 12.3* da Xilinx. Com essa ferramenta não só se escolheram os periféricos auxiliares e se criou e integrou ao sistema base o meu próprio IPCORE, como também através dela se fez o *place and route*, geração do bitstream e descarregamento para a placa ML507 da mesma. Para síntese dos módulos do IPCORE usou-se a ferramenta *Project Navigator* também contida na *ISE Design Suite 12.3*. A figura 4.2 mostra o sistema base usado na implementação em Hardware. Todos esses periféricos estão ligados ao barramento PLB e, através dele, os periféricos comunicam-se entre si.

A aplicação e os ficheiros de entrada são transferidos do computador para a placa através da Hard Ethernet. Esses são guardados na DDR2 e posteriormente executados no PowerPC, que opera a uma frequência de 400MHZ. Durante a execução da aplicação, no momento de execução da função, que foi substituída pelo driver, os dados são transferidos para as RAMs do módulo, são processados pelo módulo e quando este terminar, a aplicação lê os dados da RAM e continua o processamento. Após o término da execução da aplicação, o ficheiro de saída é guardado na DDR2.

4.4 Ambiente de Validação

A estratégia usada para verificação da correcção dos dados e validação do sistema na placa de desenvolvimento é mostrada na figura 4.3. O mesmo ficheiro é aplicado à entrada da aplicação somente em Software e da aplicação em Software modificada para aceder a Hardware. Cada uma das aplicações é executada na placa de desenvolvimento e os resultados guardados em ficheiros. Posteriormente, esses dois ficheiros de saída foram comparados, usando o comando *diff* do linux, que compara dois ficheiros especificados e caso haja diferenças entre os dois ficheiros indica onde se verificou a diferença. Ambos os ficheiros contendo dados de fala fornecido pela REFLECT, *speech* e *dm19.8*, foram usados neste processo.

Para maior segurança, não só foram comparados os dados obtidos à saída da aplicação, isto é, os dados finais, mais também foram comparados os dados obtidos logo à saída das funções implementadas em Hardware.

No processo final de validação dos dados verificou-se a total correcção dos dados obtidos pelo processamento em Hardware.

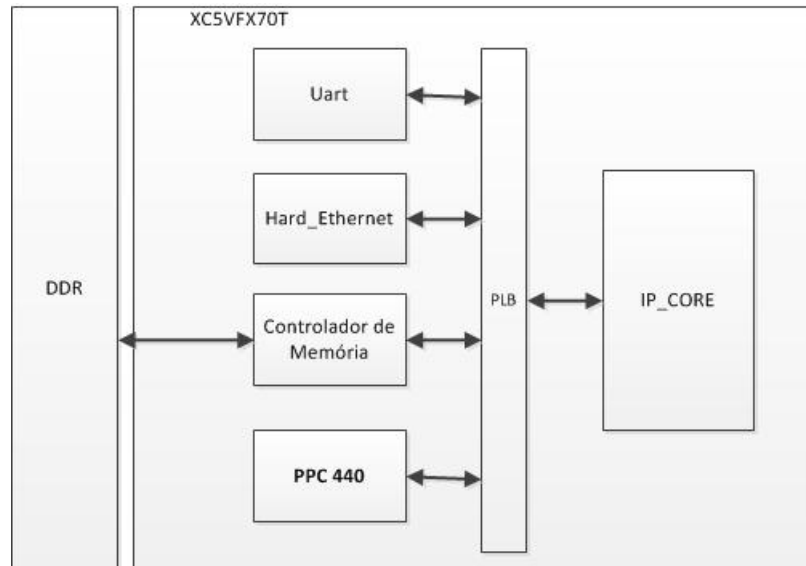


Figura 4.2: Sistema base

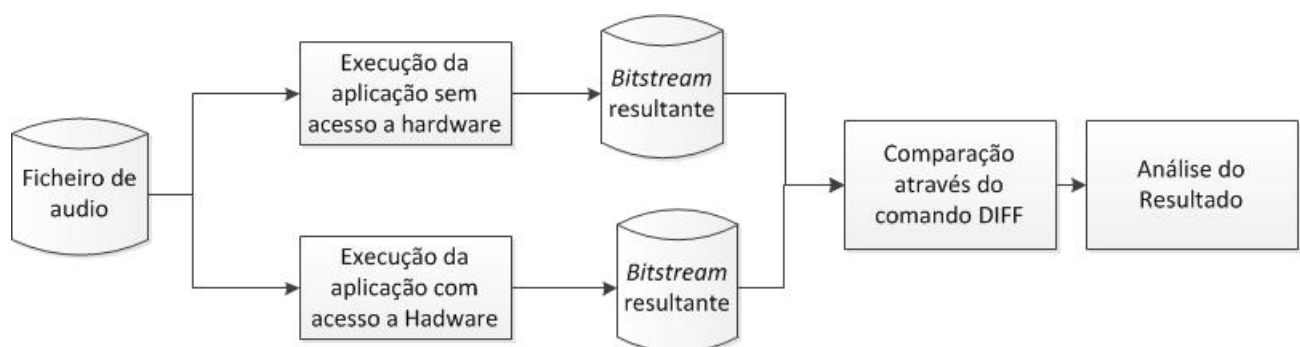


Figura 4.3: Ambiente de validação

Capítulo 5

Implementação em Hardware

Neste capítulo é descrita a implementação em Verilog dos diferentes módulos para as funções Residue e Lag-Max. Para além disso, é descrita a integração desses módulos ao *user logic*, mostrando as ligações entre os módulos aos blocos BRAM e aos registos acessíveis por Software.

5.1 Sub-Módulos

Nesta secção é descrita a implementação em Verilog de dois módulos importantes incluídos nos módulos principais, *multiplier* e *adder*. Os módulos *multiplier* e *adder* implementam, respectivamente, um multiplicador de variáveis de 16 bits e um somador de variáveis de 32 bits com sinal em complemento para 2. Estes módulos poderiam ser usados para implementar em Hardware outras funções do codificador em C para além das implementadas nesta dissertação.

5.1.1 Multiplicador

O módulo *multiplier* calcula o dobro da multiplicação de duas variáveis de 16 bits. É constituído por duas partes: na 1ª parte é efectuado a multiplicação das duas variáveis e na 2ª parte é calculado o dobro do resultado obtido na primeira parte.

Para implementar a 1ª parte do *multiplier* optou-se por inferir blocos DSP já contidos na Virtex-5. A Virtex-5 contém *slices* DSP com blocos multiplicadores de variáveis de 25x18 bits em complemento para 2. Estes blocos multiplicadores foram usados na implementação dada a sua baixa latência e elevada frequência. Para tornar o módulo mais portátil, optou-se por inferir os blocos DSP em vez de instanciar.

A tabela 5.1 mostra os sinais de entrada e saída do módulo *multiplier*:

Os sinais de entrada A e B são aplicadas a entrada do módulo que infere o DSP48. Visto que o DSP calcula a multiplicação de variáveis de 25x18, é importante fazer a extensão do sinal das duas entradas A e B, para garantir o resultado correcto.

A figura 5.1 mostra como foi inferido o bloco multiplicador DSP48. Foi necessária uma implementação rigorosa por forma a que o bloco DSP48 fosse correctamente inferido. As duas entradas A e B, de 25 e 18 bits respectivamente, são guardadas em dois registos A1 e B1. O valor

Sinal	Direcção	Dimensão(bits)	Descrição
clk	entrada	1	Sinal de relógio
rst	entrada	1	Sinal de <i>reset</i> activo a 1
en	entrada	1	Sinal de <i>enable</i> activo a 1
A	entrada	16	Sinal a multiplicar com sinal em complemento para 2
B	entrada	16	Sinal a multiplicar com sinal em complemento para 2
Out	saída	32	Resultado do dobro da multiplicação de A com B

Tabela 5.1: Sinais do módulo *multiplier*

desses 2 registos é multiplicado e guardado num registo de 48 bits, obtendo-se na saída o resultado da multiplicação de A e B com 48 bits. São necessários apenas dois ciclos de relógio para efectuar a multiplicação das duas entradas.

De seguida é calculado o dobro do resultado obtido anteriormente. Para isso basta somar o resultado a si mesmo. A figura 5.2 mostra esse processo. A entrada A mostrada na figura representa o resultado do processo anterior. Esta entrada é de 32 bits o que significa que o resultado anterior de 48 bits foi truncado para 32 bits. Isto não produz nenhum efeito negativo, visto que, as duas variáveis iniciais a multiplicar são de 16 bits, e o resultado da multiplicação de duas variáveis de 16 bits é de 32 bits. Para evitar que haja *overflow*, se o valor de A for igual a 32'h4000_0000, a saída é saturada em 32'h7000_0000, caso contrário, a saída será o dobro de A.

No total, o módulo *multiplier* produz uma latência de 3 ciclos de relógio, sendo 2 ciclos de relógio gastos na multiplicação no bloco DSP48 e mais 1 ciclo de relógio gastos para calcular o dobro do resultado anterior.

5.1.2 Somador

O módulo *adder* calcula a soma de duas variáveis de 32 bits com sinal em complemento para 2. Para prevenir *overflow*, a implementação segue as seguintes regras:

- Se a soma de duas variáveis de sinal positivo produzir um resultado negativo, houve *overflow*;
- Se a soma de duas variáveis de sinal negativo produzir um resultado positivo, houve *overflow*;

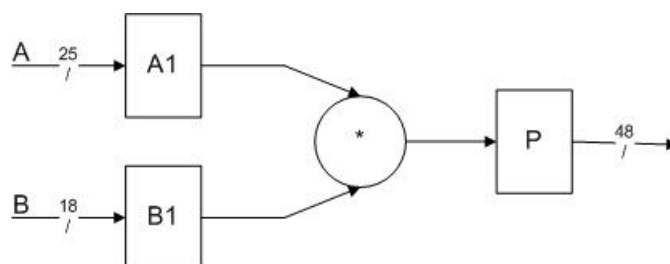
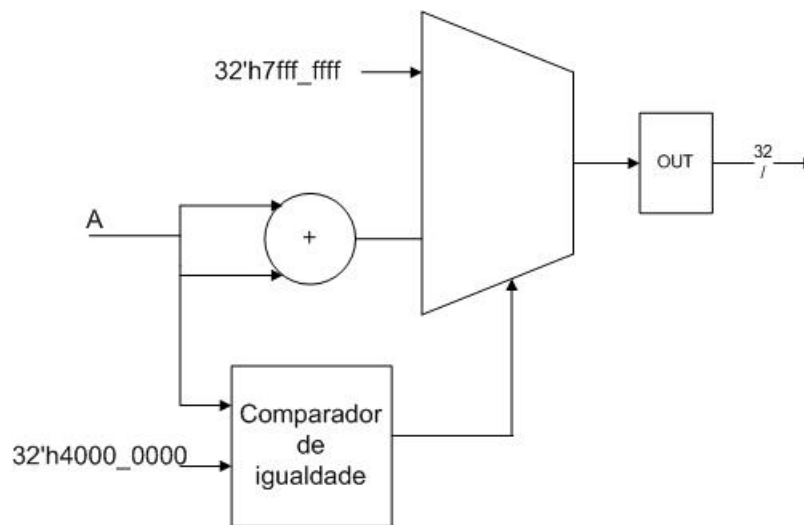


Figura 5.1: Inferir o DSP48

Figura 5.2: 2ª parte do módulo *multiplier*

- Caso contrário, não houve *overflow* no resultado.

A tabela 5.2 mostra os sinais de entrada e saída do módulo *adder*:

Sinal	Direcção	Dimensão(bits)	Descrição
clk	entrada	1	Sinal de relógio
rst	entrada	1	Sinal de <i>reset</i> activo a 1
en	entrada	1	Sinal de <i>enable</i> activo a 1
A	entrada	16	Sinal a adicionar com sinal em complemento para 2
B	entrada	16	Sinal a adicionar com sinal em complemento para 2
Out	saída	32	Resultado da soma de A com B

Tabela 5.2: Sinais do módulo *adder*

De seguida é mostrada uma parte do código que implementa a adição:

```

.
.
.
if ( enable )
    begin
        sign    <= A[31] ^ B[31];
        add     <= A + B;
        A1      <= A[31];

        if (( sign == 1'b0) && ((add[31] ^ A1) == 1'b1))
            begin
                if (A1 == 1'b1)
                    begin
                        Out <= MIN_32;
                    end
            end
    end

```

```

                                else
                                begin
                                    Out <= MAX_32;
                                end
                                end
                            end
                            else
                            begin
                                Out <= add;
                            end
                        end
                    end
                else
                .
                .
                .
            
```

O registo *sign* contém o resultado do xor lógico entre os sinais das entradas A e B, sendo 0 caso tiverem o mesmo sinal (isto é, serem ambos de sinais positivos ou ambos negativos) ou 1 caso tiverem sinais diferentes. O registo *add* contém o resultado da soma da entrada A com a entrada B. O registo A1 contém o sinal de uma das entradas (neste caso, da entrada A). Caso as duas entradas A e B tiverem o mesmo sinal, isto é, o registo *sign* for 0, e o resultado da adição de A com B tiver sinal contrário as entradas (o xor lógico entre o sinal de *add* e A1 é 1, caso tiverem sinais diferentes), a saída é saturada, caso contrário, a saída será a soma das entradas.

5.2 Residue

Para a implementação em Hardware da função Residue, três casos de teste diferentes na placa de desenvolvimento foram explorados. No primeiro caso, duas versões especializadas da função foram implementadas para $M = 10$ e $M = 30$. Esses dois módulos, *Residue10* e *Residue30*, são completamente paralelizáveis, isto é, utilizam-se 11 multiplicadores e 31 multiplicadores, respectivamente. No segundo caso, o mesmo módulo *Residue10* é usado mas, na especialização $M = 30$, é usado o módulo *Residue30_10* apenas com 11 multiplicadores. No terceiro caso, um módulo genérico *Residue* com 11 multiplicadores que permite $M = 10$ e $M = 30$ é usado. No total 4 módulos foram implementados.

As figuras 5.3, 5.4 e 5.5 mostram os três casos de teste na placa de desenvolvimento.

5.2.1 *Residue10*, *Residue30* e *Residue30_10*

O módulo *Residue10* implementa a função Residue para valor de $M=10$ e os módulos *Residue30* e *Residue30_10* implementam a função Residue para valor de $M=30$, sendo nos três casos valor de lg variável. O módulo *Residue10* usa 11 multiplicadores e o módulo *Residue30* usa 31 multiplicadores, sendo eles completamente paralelizáveis e os módulos mais rápidos mas também com maior gasto de multiplicadores. O módulo *Residue30_10*, por sua vez, usa apenas 11 multiplicadores, o que o torna menos paralelizável e mais lento que *Residue30*. Cada um dos módulos,

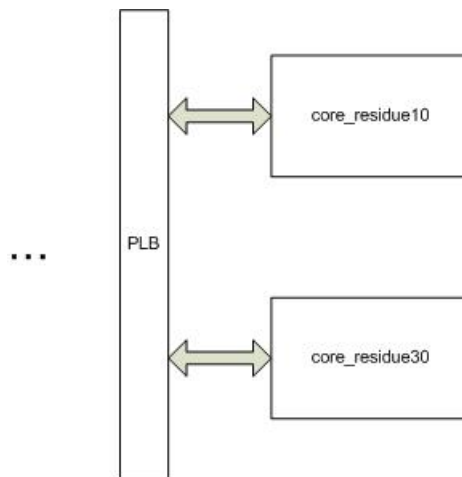


Figura 5.3: O primeiro caso de teste da função Residue na placa de desenvolvimento

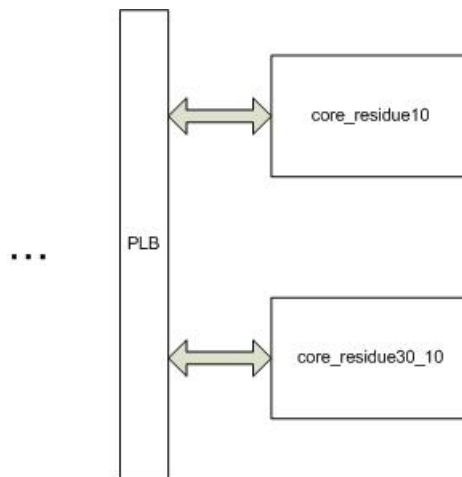


Figura 5.4: O segundo caso de teste da função Residue na placa de desenvolvimento

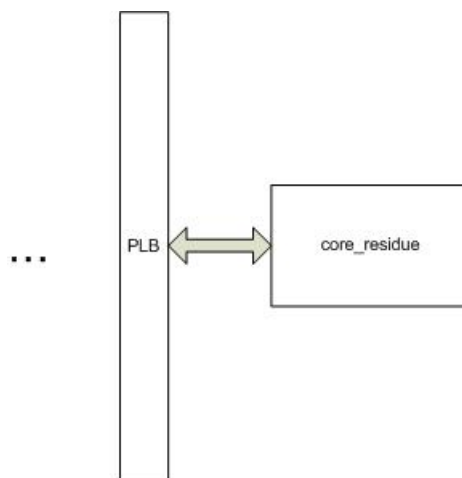


Figura 5.5: O terceiro caso de teste da função Residue na placa de desenvolvimento

lê os valores da entrada e escreve os resultados em dois blocos BRAM e também lê e escreve para variáveis acessíveis por Software.

A tabela 5.3 mostra os sinais de entrada e saída dos três módulos e a tabela 5.4 faz a descrição dos parâmetros.

Sinal	Direcção	Dimensão(bits)	Descrição
clk	entrada	1	Sinal de relógio
rst	entrada	1	Sinal de <i>reset</i> activo a 1
start	entrada	1	Sinal para o módulo começar a funcionar
done	saída	1	Sinal que o módulo transmite para indicar o fim das operações e que todos os resultados estão guardados na memória; activo a 1
addr_x	entrada	ADDR	Endereço da memória onde o módulo deve começar a ler a entrada x
addr_a	entrada	ADDR	Endereço da memória onde o módulo deve começar a ler a entrada a
addr_out	entrada	ADDR	Endereço da memória onde o módulo deve começar a escrever o resultado da saída
lg	entrada	16	O número de resultados que o módulo irá calcular
ram1_inx	entrada	16	Entrada dos valores de x
ram1_addr	saída	ADDR	Endereço para a BRAM1
ram2_addr	saída	ADDR	Endereço para a BRAM2
ram2_we	saída	1	Sinal de <i>enable</i> de escrita na BRAM2; activo a 1
ram2_ina	entrada	16	Entrada dos valores de a
ram2_out	saída	16	Saída dos resultados para a BRAM2

Tabela 5.3: Sinais dos módulos *Residue10* e *Residue30*

Parâmetro	Descrição
ADDR	Descrição: Variável que define o limite do endereço dos blocos BRAM conforme as suas dimensões Valor padrão: 10 Valor mínimo: Depende da quantidade de dados a ser guardados nos blocos BRAM Valor máximo: Depende da tecnologia usada. A virtex-5 permite implementar blocos RAM de 36K.

Tabela 5.4: Descrição dos parâmetros dos três módulos

Os módulos *Residue10* e *Residue30* foram implementados em verilog com uma estrutura semelhante, variando apenas no número de registos que compõem os registos de deslocamento, o número de multiplicações e adições efectuadas e consequentemente na latência de saída dos resultados finais. Uma máquina de estados foi usada para controlar o funcionamento dos módulos. A figura 5.6 mostra a máquina de estados e a tabela 5.6 faz a descrição de cada um dos estados e transições de estado.

O módulo *Residue30_10* apresenta uma estrutura um pouco diferente. Por utilizar apenas 11 multiplicadores, são necessários 3 ciclos de relógio para cada iteração de lg. Os resultados são

Estado/Transição de estado	Descrição
IDLE	Neste estado o módulo está parado a aguardar que o sinal de entrada <i>start</i> fique a 1
IDLE -> START	É detectado a subida do sinal de entrada <i>start</i>
START	Neste estado aguarda-se que o sinal <i>start</i> se altere
START -> FILL	É detectada a descida do sinal de entrada <i>start</i>
FILL	Neste estado são preenchidos os registos de deslocamentos <i>shiftrega</i> e <i>shiftregx</i> com os valores de entrada <i>ram2_ina</i> e <i>ram2_inx</i> , respectivamente, lidos dos blocos BRAM. O controlo do preenchimento dos registos de deslocamento é efectuado com um contador <i>counter</i> que é inicializado com a dimensão dos registos de deslocamento e vai ser decrementado.
FILL -> CALC	O contador <i>counter</i> chega a 0 indicando que os registos estão preenchidos. O contador é inicializado com o valor lg-1.
CALC	Neste estado serão efectuados lg-1 deslocamentos do registo <i>shiftregx</i> e para cada iteração é efectuado o cálculo do resíduo de <i>shiftregx</i> com <i>shiftrega</i> . O cálculo do resíduo inclui um conjunto de multiplicações, adições e por fim o truncamento do resultado, o que vai gerar uma latência até que o resultado final seja obtido. A latência é previamente calculada e é usada um contador <i>counter_lat</i> para controlar a escrita do resultado final no bloco BRAM.
CALC -> WAIT	O contador <i>counter</i> chega a 0.
WAIT	Neste estado serão escritos na BRAM os restantes resultados finais que não foram escritos no estado anterior devido a latência. O contador <i>counter_lat</i> é usado para controlar a latência.
WAIT -> IDLE	O contador <i>counter_lat</i> chega a 0 indicando que todos os resultados já estão escritos no bloco BRAM e a saída <i>done</i> é colocada a 1 para indicar o fim das operações.

Tabela 5.5

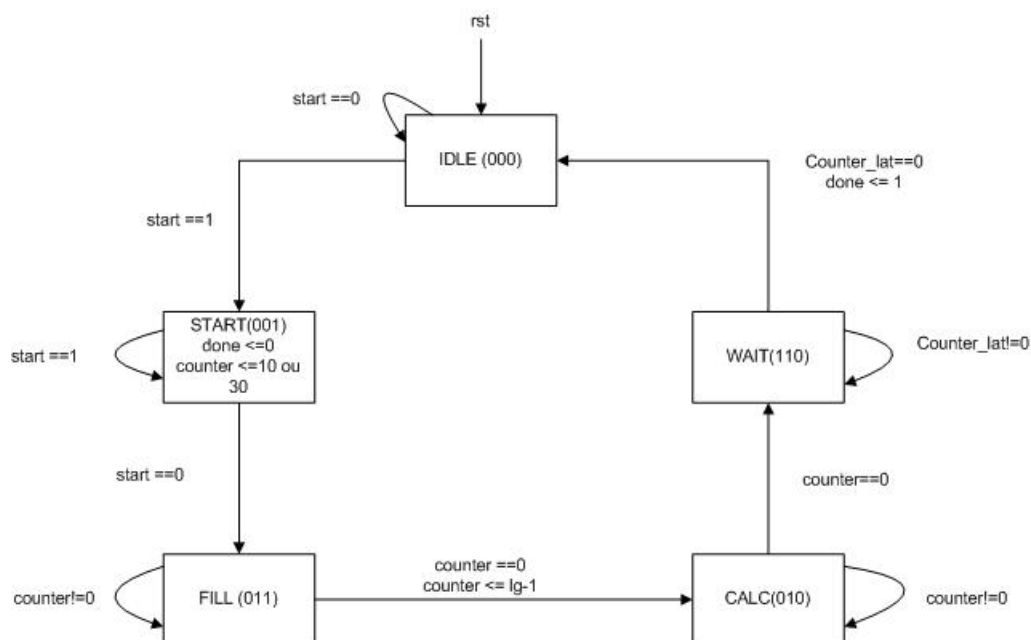


Figura 5.6: Máquina de Estado dos módulos *Residue10* e *Residue30*

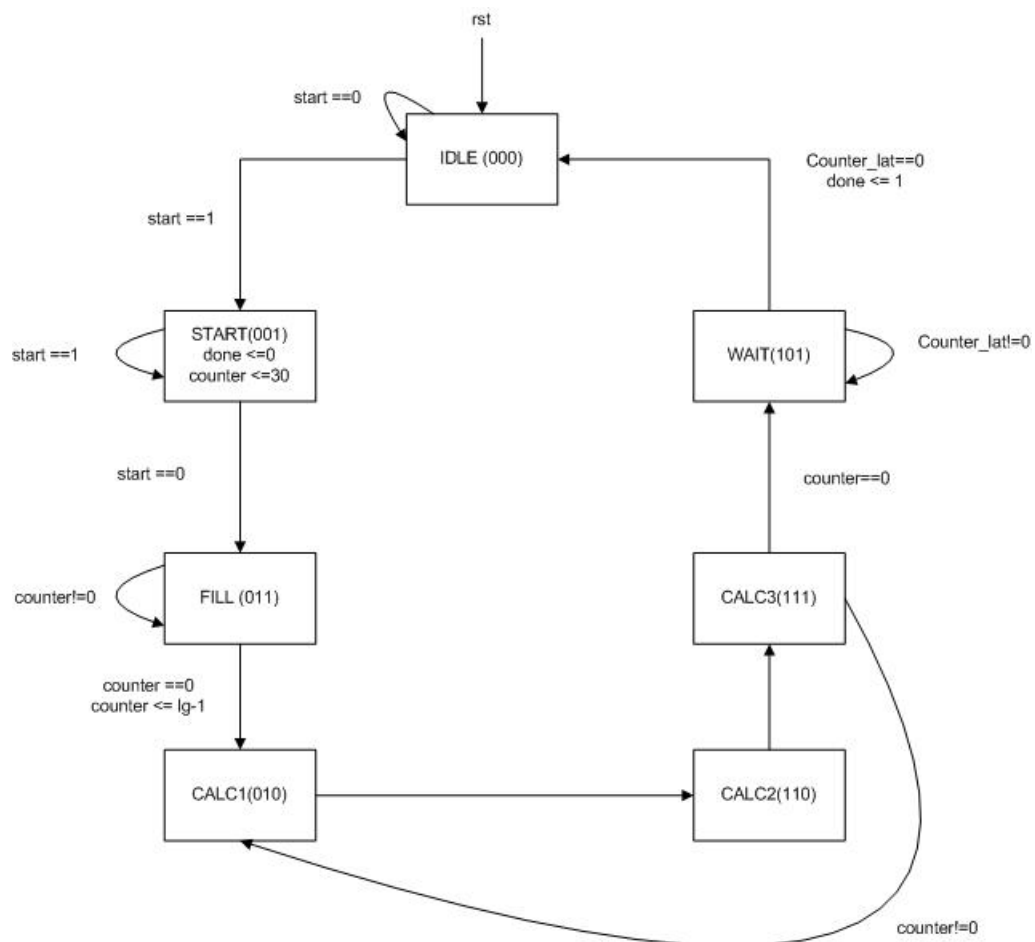
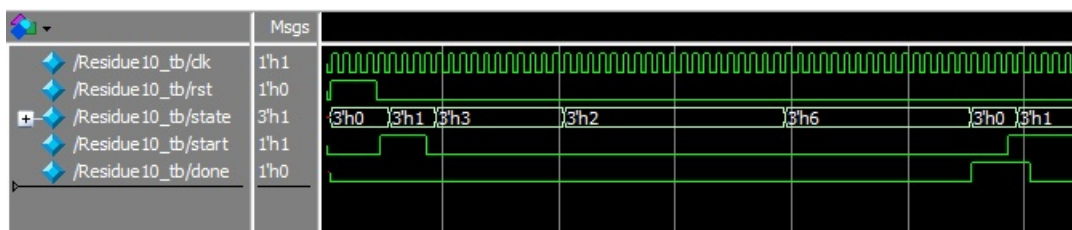
colocados à saída a cada 3 ciclos de relógio. A figura 5.7 mostra a máquina de estado e a tabela faz a descrição dos estados e transições.

5.2.1.1 Sinal de *start* e de *done*

O sinal de entrada *start* é transmitido pelo Software aos módulos para que estes iniciem o funcionamento. Os módulos iniciam funcionamento após detectar a subida seguida da descida do sinal *start*. O Software que transmite o sinal deve garantir que o pulso do sinal *start* seja no mínimo 2 ciclos de relógio a que opera os módulos em verilog para que estes o possam detectar correctamente.

O sinal de saída *done* é transmitido pelos módulos ao Software a indicar o fim das operações. Os módulos colocam o sinal a 1 após estarem todos os resultados escritos nos BRAM e só voltam a colocar o sinal a 0 por acção do *reset* ou se detectarem uma nova subida do sinal de *start*, de forma a garantir que o sinal de *done* seja visto pelo Software.

Na figura 5.8 vesse os sinais *start* e *done* e as mudanças de estado da máquina de estado. A máquina de estado está inicialmente no estado IDLE(3'h0), após detectar a subida do sinal *start*, o estado muda para START (3'h1) e após detectar a descida do sinal, passa para o estado FILL(3'h3). Na final do estado WAIT (3'h6), o sinal *done* é colocado a 1 e só voltou a ser colocado a 0 após a detecção de um novo sinal de *start*.

Figura 5.7: Máquina de Estado do módulo *Residue30_10*Figura 5.8: Sinal de *start* e sinal de *done*

Estado/Transição de estado	Descrição
IDLE	Neste estado o módulo está parado a aguardar que o sinal de entrada <i>start</i> fique a 1
IDLE -> START	É detectada a subida do sinal de entrada <i>start</i>
START	Neste estado aguarda-se que o sinal <i>start</i> se altere
START -> FILL	É detectada a descida do sinal de entrada <i>start</i>
FILL	Neste estado são preenchidos os registos de deslocamento <i>shiftrega</i> e <i>shiftregx</i> com os valores de entrada <i>ram2_ina</i> e <i>ram2_inx</i> , respectivamente, lidos dos blocos BRAM. O controlo do preenchimento dos registos de deslocamento é efectuado com um contador <i>counter</i> que é inicializado com a dimensão dos registos de deslocamento e vai ser decrementado.
FILL -> CALC1	O contador <i>counter</i> chega a 0 indicando que os registos estão preenchidos. O contador é inicializado com o valor lg-1
CALC1	Neste estado são calculadas as primeiras 11 multiplicações.
CALC2	Neste estado são calculadas as segundas 11 multiplicações.
CALC3	Neste estado são calculadas as últimas 11 multiplicações. O registo de deslocamento <i>shiftregx</i> recebe mais um valor do bloco RAM e o contador <i>counter</i> vai ser decrementado
CALC -> WAIT	O contador <i>counter</i> chega a 0
WAIT	Neste estado serão escritos na BRAM os restantes resultados finais que não foram escritos no estado anterior devido a latência. O contador <i>counter_lat</i> é usado para controlar a latência.
WAIT -> IDLE	O contador <i>counter_lat</i> chega a 0 indicando que todos os resultados já estão escritos no bloco BRAM e a saída <i>done</i> é colocada a 1 para indicar o fim das operações.

Tabela 5.6



Figura 5.9: Registos de deslocamentos do módulo *Residue10*

5.2.1.2 Os registos de deslocamento

A figura 5.9 mostra os registos de deslocamento do módulo *Residue10*. Cada um dos registos de deslocamento é constituído por 11 registos de 16 bits, sendo, no total utilizados 22 registos de 16 bits. O registo de deslocamento *shiftrega* é preenchido a cada ciclo de relógio com os valores da entrada *ram2_ina* e após estar cheio mantém-se estático. O registo de deslocamento *shiftregx* é preenchido com os valores da entrada *ram2_inx* e após estar cheio é deslocado mais $\lg-1$ vezes, tudo isto feito a cada ciclo de relógio.

A figura 5.10 mostra os registos de deslocamento dos módulos *Residue30* e *Residue30_10*. Cada um dos registos de deslocamento é constituído por 31 registos de 16 bits, sendo, no total utilizados 62 registos de 16 bits. Para o módulo *Residue30*, os registos de deslocamento funcionam de forma semelhante aos registos de deslocamento do módulo *Residue10*. O registo de deslocamento *shiftrega* é preenchido a cada ciclo de relógio com os valores da entrada *ram2_ina* e após estar cheio mantém-se estático e o registo de deslocamento *shiftregx* é preenchido com os valores da entrada *ram2_inx* e após estar cheio é deslocado mais $\lg-1$ vezes. Para o módulo *Residue30_10*, o registo de deslocamento *shiftrega* é preenchido a cada ciclo de relógio com os valores da entrada *ram2_ina* e após estar cheio deixa de receber valores da entrada *ram2_ina* e passa a rodar sobre si mesmo. O registo de deslocamento *shiftregx* é preenchido a cada ciclo de relógio com os valores da entrada *ram2_inx* e após estar cheio passa a ser deslocada a cada 3 ciclos de relógio, sendo que cada conjunto de 3 ciclos de relógio, perfaz uma iteração de $\lg$.

5.2.1.3 Cálculo do resíduo

Após os registos de deslocamento estarem cheios, dá-se início ao cálculo do resíduo entre os valores dos registos. No total são calculados $\lg$ valores de resíduo. Para cada iteração de $\lg$, são efectuados $M+1$ multiplicações entre os registos que compõem os registos de deslocamento, os resultados das multiplicações são adicionados obtendo um valor que será deslocado três bits para a esquerda e por fim truncado para um valor de 16 bits, obtendo no final o resultado do resíduo.

A figura 5.11 mostra o cálculo do resíduo no módulo *Residue10*. 11 blocos *multiplier* são utilizados em paralelo para efectuar a multiplicação entre os registos dos registos de deslocamento



Figura 5.10: Registos de deslocamentos dos módulos *Residue30* e *Residue30_10*

shiftrega e *shiftregx*. Após as multiplicações, mais 4 níveis de blocos *adder* são necessários até que os valores das multiplicações sejam adicionados. E por fim, o valor obtido, que é de 32 bits, passa para o bloco *shift_round* onde será deslocado 3 bits à esquerda e depois truncado para um valor de 16 bits. A saída *out_final* é, assim, o valor do resíduo de 16 bits para cada iteração de lg. Esta estrutura de cálculo introduz uma latência de 15 ciclos de relógios. Tal como dito anteriormente, os blocos *multiplier* apresentam uma latência de 3 ciclos de relógio, os blocos *adder* apresentam uma latência de 2 ciclos de relógios, e o bloco *shift_round* introduz mais uma latência de 4 ciclos de relógios.

$$3\text{ciclosderelogio} + 4\text{niveis} * 2\text{ciclosderelogios} + 4\text{ciclosderelogios} = 15\text{ciclosderelogio} \quad (5.1)$$

Esta estrutura de cálculo é também em *pipelining*, o que vai permitir que a cada ciclo de relógio sejam aplicados os valores à entrada dos blocos *multiplier*, e a cada ciclo de relógio, após a latência, é obtido um valor de resíduo na saída *out_final* até se obterem os lg valores de resíduo.

Para o módulo *Residue30*, a estrutura de cálculo do resíduo é semelhante (fig. 5.12). Neste caso, são utilizados 31 blocos *multiplier* em paralelo e são necessário 5 níveis de blocos *adder* para obter um resultado das adições. E posteriormente, este valor é aplicado ao bloco *shift_round*, obtendo na saída *out_final* o valor do resíduo. A latência introduzida é

$$3\text{ciclosderelogio} + 5\text{niveis} * 2\text{ciclosderelogios} + 4\text{ciclosderelogios} = 17\text{ciclosderelogio} \quad (5.2)$$

Após a latência, a cada ciclo de relógio é obtido na saída *out_final* um valor de resíduo.

Para o módulo *Residue30_10* 5.13 são utilizados apenas 11 blocos *multiplier* para efectuar 31 multiplicações, o que leva a que as multiplicações não sejam todos paralelizáveis mas sim divididos em 3 ciclos de relógios. Um bloco *accumulator* é usado para acumular os valores das adições de cada uma das 3 partes que constituem o cálculo de um valor de resíduo. Só a cada 3 ciclos de relógio obtém-se na saída *out_final* um valor de resíduo.

Tal como visto na figura 5.13, as multiplicações são efectuadas entre o registo de deslocamento

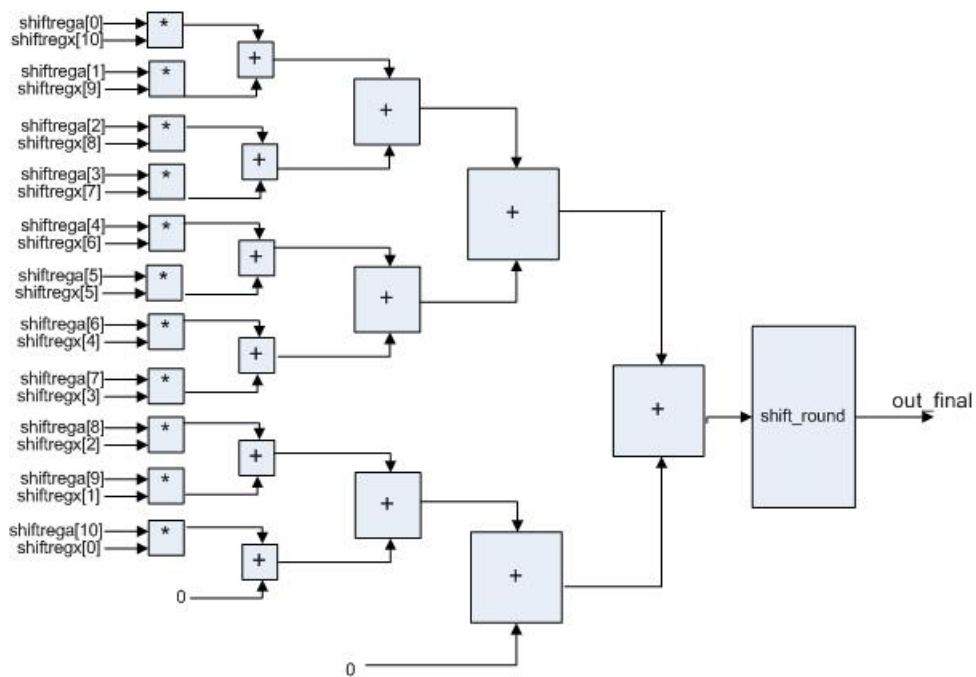


Figura 5.11: Estrutura de cálculo do resíduo no módulo *Residue10*

shiftrega nas posições 0 a 10 e um conjunto de 11 registros auxiliares chamados *regs*. A cada ciclo de relógio, o registro de deslocamento *shiftrega*, de 31 registros, é rodado 10 posições à esquerda por forma a colocar nas posições de 0 a 10 os valores a serem calculados. Para os 11 registros auxiliares *regs* são copiados partes do registro *shiftregx*, tal como mostrado na figura 5.14. No primeiro ciclo de relógio é copiado para *regs shiftregx[0:10]*, no segundo ciclo de relógio é copiado *shiftregx[11:21]* e no terceiro ciclo de relógio é copiado *shiftregx[22:30]*

Porque foi escolhido utilizar os registros auxiliares *regs* em vez de multiplexers? Foi escolhido utilizar estes registros auxiliares para tornar o módulo mais rápido. O uso de multiplexers (lógica combinacional) teria um custo mais negativo em termos de frequência de síntese do módulo. E, por outro lado, registros são recursos abundantes na FPGA utilizada.

Latência Para exemplificar com mais detalhe os efeitos da latência no cálculo do resíduo, é mostrado a figura 5.15. O cálculo dos lg valores de resíduo apresenta uma estrutura em trapézio. A parte superior do trapézio representa as lg iterações em que os registros de deslocamento são deslocados e enviados para os blocos *multiplier*, e a parte inferior do trapézio representa a saída dos resultados dos resíduos. A latência vai introduzir um desfasamento de n ciclos de relógio até à saída dos resultados do resíduo. Isto justifica a necessidade de existência do estado WAIT na máquina de estado. Após complementar as lg iterações de deslocamento dos registros de deslocamento e envio dos valores para cálculo, é necessário aguardar que os restantes valores de resíduo sejam obtidos e guardados na memória e só depois o módulo termina o seu funcionamento e coloca a saída *done* a 1.

Nos módulos *Residue10* e *Residue30*, as partes superiores e inferiores do trapézio têm uma

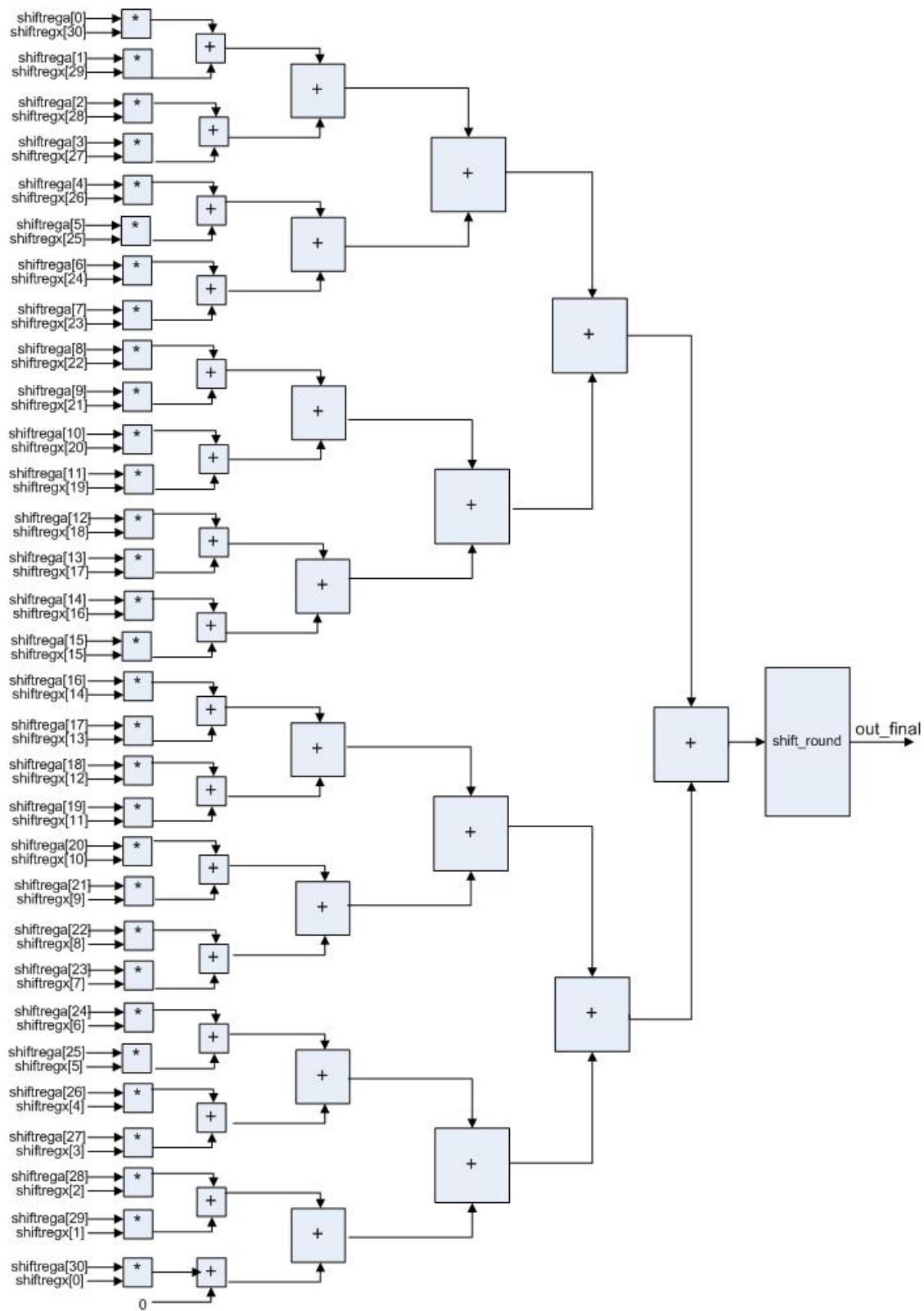


Figura 5.12: Estrutura de cálculo do resíduo no módulo *Residue30*

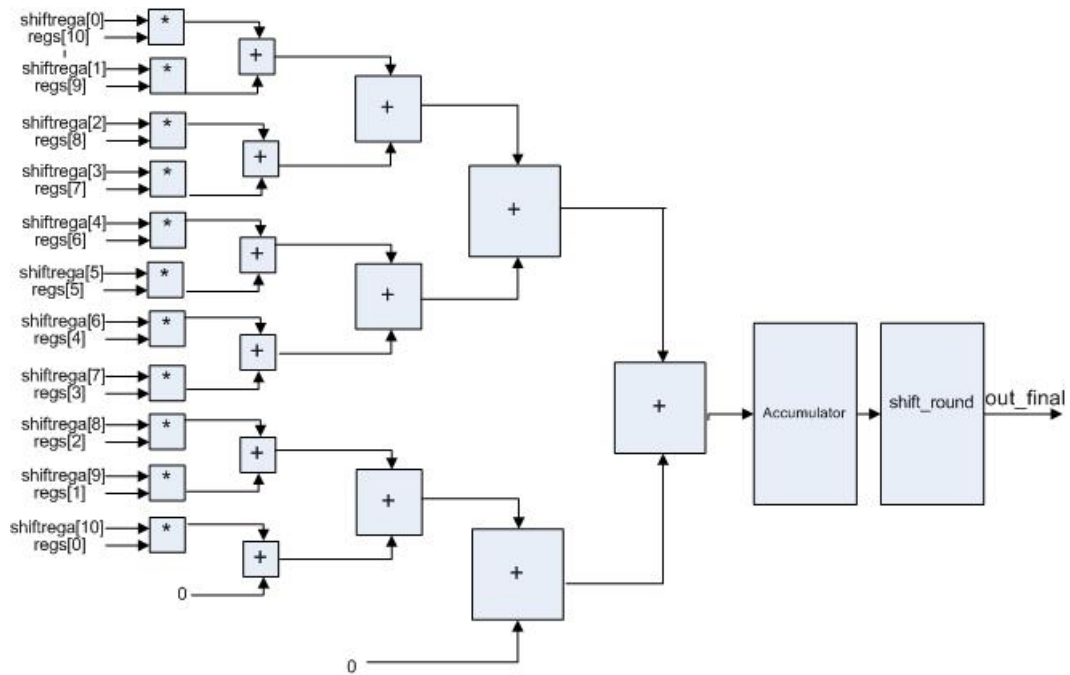


Figura 5.13: Estrutura de cálculo do resíduo no módulo *Residue30_10*

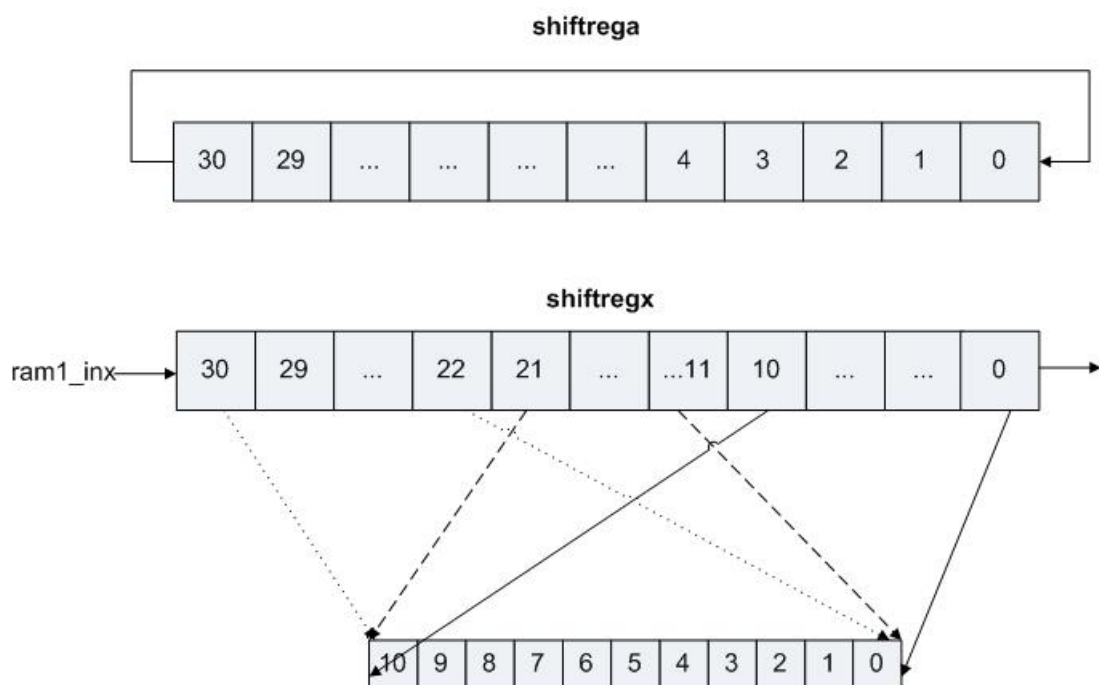


Figura 5.14: Estrutura de cálculo do resíduo no módulo *Residue30_10*

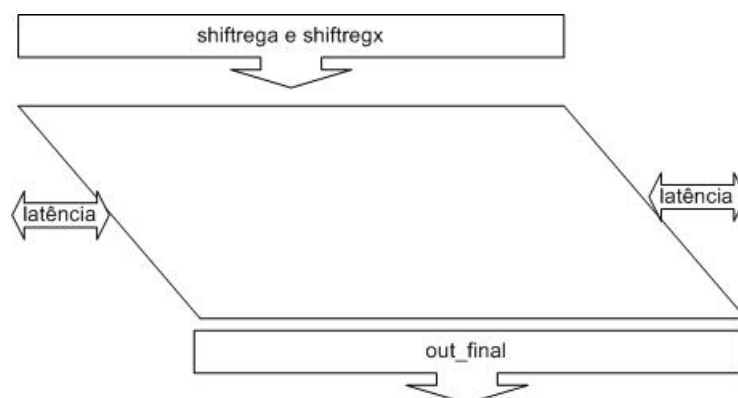


Figura 5.15: Latência

duração de lg ciclos de relógios porque são completamente paralelizáveis e os resultados de resíduo são produzidos a cada ciclo de relógio. No módulo *Residue30_10*, ao contrário dos módulos anteriores, as lg iterações não correspondem a lg ciclos de relógios mas sim a $3*lg$ ciclos de relógios. Este módulo não é completamente paralelizável sendo cada resultado de resíduo produzido a cada 3 ciclos de relógios.

5.2.1.4 Leitura e escrita das BRAM

Os módulos recebem em três portas de entrada, *addr_x*, *addr_a* e *addr_out* os endereços onde devem começar a ler os valores de *x* e *a* e onde deve começar a escrever os valores de resíduo, respectivamente. Três contadores são utilizados para gerar os endereços, *counter_addrx*, *counter_addra* e *counter_addrout*. Esses contadores são inicializados com os valores das entradas *addr_x*, *addr_a* e *addr_out* respectivamente, e vão sendo incrementadas para gerar o próximo endereço. É colocada nas saídas do endereço para as RAMs os valores desses contadores. Após a latência dos cálculos, na porta de saída para a RAM2, *ram2_out*, é colocada, o valor de resíduo calculado, *out_final*, e a saída *ram2_we* é colocada a 1.

5.2.1.5 Ligação dos módulos as BRAM e aos registos acessíveis por Software

Na integração dos módulos ao *userlogic* para construir os seus respectivos **IP_Core**, os módulos são ligados a duas *Dual Port BRAM* de onde lerá os valores das entradas *a* e *x* e de onde escreverá os valores dos resíduos calculados. Os módulos são ligados a porta B das DPBRAM e a porta A das DPBRAM é ligada a lógica de ligação a PLB.

A figura 5.16 mostra as ligações. Cada uma das DPBRAM são de 16Kbits. Na primeira DPRAM os módulos lêem os valores da entrada *x*, e na segunda DPBRAM lêem os valores de entrada *a* e escrevem os resultados da saída. O uso de duas DPRAM permitiu que as entradas *a* e *x* fossem lidas ao mesmo tempo diminuindo os ciclos de relógios necessários para encher os dois registos de deslocamentos *shiftrega* e *shiftregx* porque são preenchidas ao mesmo tempo. E também permitiu que os valores de saída fossem escritas na RAM enquanto é lida a entrada *x*.

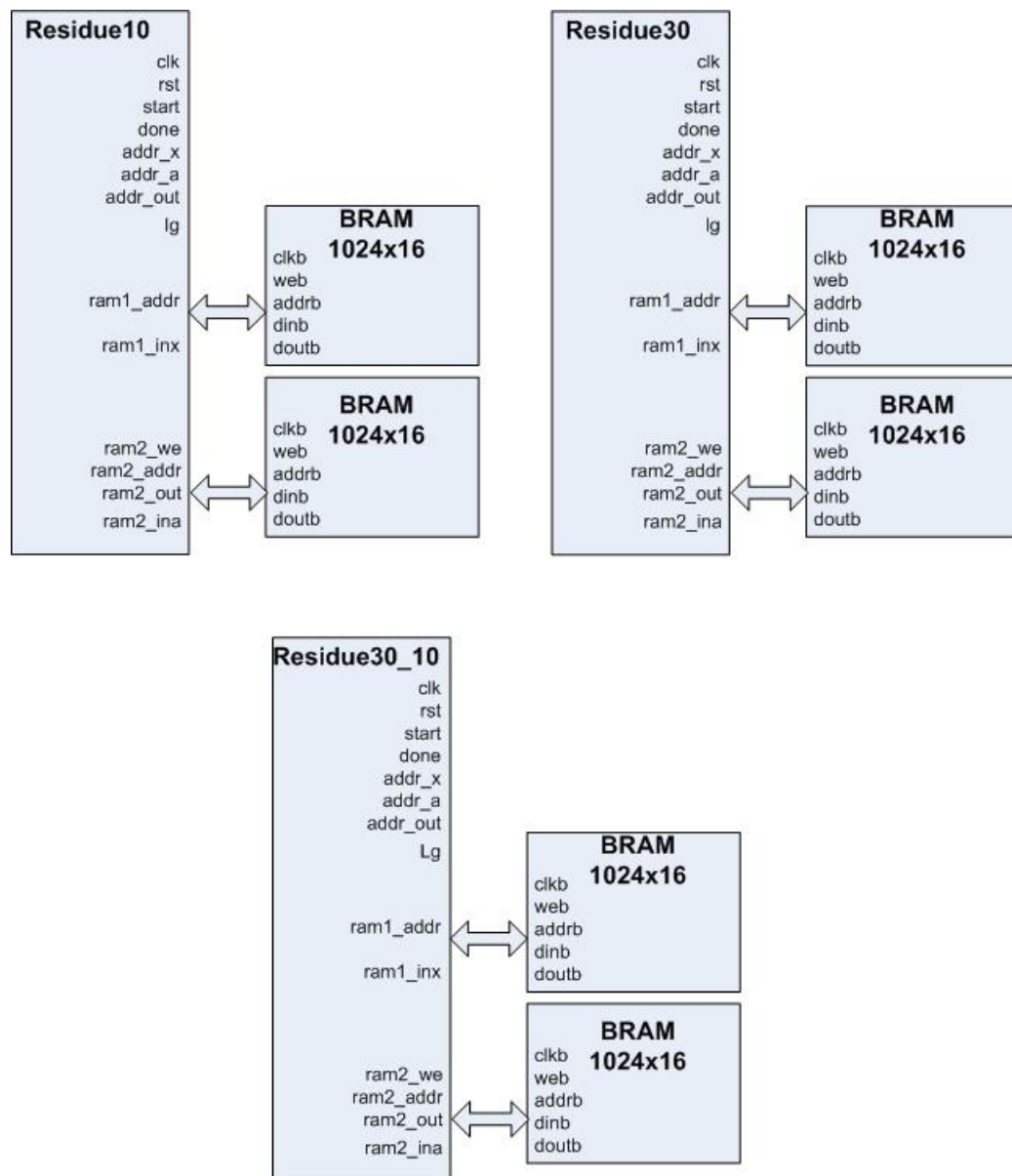


Figura 5.16: Ligação dos módulos aos respectivos DPBRAM

As restantes portas dos módulos, com excepção da porta de entrada do relógio, estão ligadas a registos acessíveis por Software. A porta de entrada do relógio *clk* está ligada ao relógio da PLB. A tabela 5.7 mostra o mapeamento das portas do módulo aos registos acessíveis por Software, registos esses de 32 bits. Na tabela estão representados apenas os 16 bits mais significativos dos registos. Os outros 16 bits menos significativos não foram utilizados.

Índice	Registos	Bits															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	slv_reg0	start															
1	slv_reg1	done															
2	slv_reg2	lg															
3	slv_reg3	addr_x															
4	slv_reg4	addr_a															
5	slv_reg5	addr_out															
6	slv_reg6	reset															

Tabela 5.7: Mapeamento das entradas e saídas dos módulos nos registos acessíveis por Software

5.2.2 Residue

O módulo *Residue* implementa a função *Residue* para ambos os valores de $M=10$ e $M=30$, sendo o valor de lg variável. A escolha de funcionamento para cada um dos valores de M é feita através de uma porta de entrada de 1 bit. O módulo *Residue* usa apenas 11 multiplicadores. Funciona como o módulo *Residue10* caso for escolhido $M=10$ e funciona como o módulo *Residue30_10* caso for escolhido $M=30$. Este módulo vai permitir uma minimização da área da FPGA utilizada. O módulo lê os valores da entrada e escreve os resultados em dois blocos BRAM e também lê e escreve para variáveis acessíveis por Software.

A tabela 5.8 mostra os sinais de entrada e saída do módulo e a tabela 5.9 faz a descrição dos parâmetros.

Para permitir o funcionamento para os dois valores de M , é utilizado registos de deslocamento para o caso de maior valor de M , isto é, são utilizados 31 registos para cada um dos registos de deslocamento. Assim sendo, os registos de deslocamento são semelhantes aos do módulo *Residue30_10*, tal como, mostrada na figura 5.10.

O módulo *Residue* apresenta um funcionamento igual ao módulo *Residue10* caso a entrada m seja 0, isto é, a cada ciclo de relógio um resultado é guardado na memória; e apresenta um funcionamento igual ao módulo *Residue30_10* caso a entrada m seja 1, guardando um resultado na memória a cada três ciclos de relógio. A figura 5.17 mostra o funcionamento do módulo para cada valor de m .

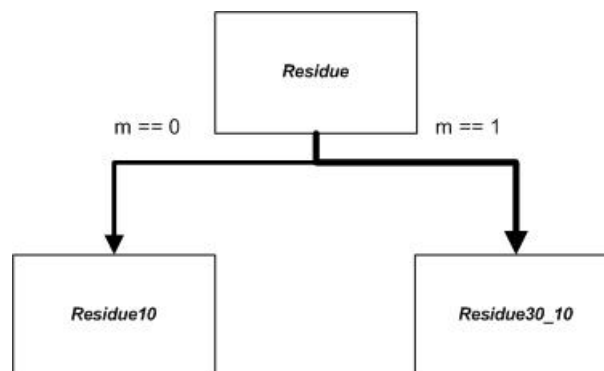
À semelhança dos três módulos anteriores, o módulo *Residue* está ligado a duas *Dual Port BRAM* de 16Kbits, de onde lerá os valores das entradas a e x e de onde escreverá os valores dos resíduos calculados. O módulo é ligado a porta B das DPBRAM e a porta A das DPBRAM é ligada a lógica de ligação a PLB. A figura 5.18 mostra as ligações. Na primeira DPRAM o módulo lê os

Sinal	Direcção	Dimensão(bits)	Descrição
clk	entrada	1	Sinal de relógio
rst	entrada	1	Sinal de <i>reset</i> activo a 1
start	entrada	1	Sinal para o módulo começar a funcionar
m	entrada	1	Sinal que permite escolher o valor de M, sendo 0 para M=10 e 1 para M=30
done	saída	1	Sinal que o módulo transmite para indicar o fim das operações e que todos os resultados estão guardados na memória; activo a 1
addr_x	entrada	ADDR	Endereço da memória onde o módulo deve começar a ler a entrada x
addr_a	entrada	ADDR	Endereço da memória onde o módulo deve começar a ler a entrada a
addr_out	entrada	ADDR	Endereço da memória onde o módulo deve começar a escrever o resultado da saída
lg	entrada	16	O número de resultados que o módulo irá calcular
ram1_inx	entrada	16	Entrada dos valores de x
ram1_addr	saída	ADDR	Endereço para a BRAM1
ram2_addr	saída	ADDR	Endereço para a BRAM2
ram2_we	saída	1	Sinal de <i>enable</i> de escrita na BRAM2; activo a 1
ram2_ina	entrada	16	Entrada dos valores de a
ram2_out	saída	16	Saída dos resultados para a BRAM2

Tabela 5.8: Sinais do módulo *Residue*

Parâmetro	Descrição
ADDR	Descrição: Variável que define o limite do endereço dos blocos BRAM conforme as suas dimensões Valor padrão: 10 Valor mínimo: Depende da quantidade de dados a ser guardados nos blocos BRAM Valor máximo: Depende da tecnologia usada. A virtex-5 permite implementar blocos RAM de 36K.

Tabela 5.9: Descrição dos parâmetros do módulo

Figura 5.17: Funcionamento do módulo *Residue*

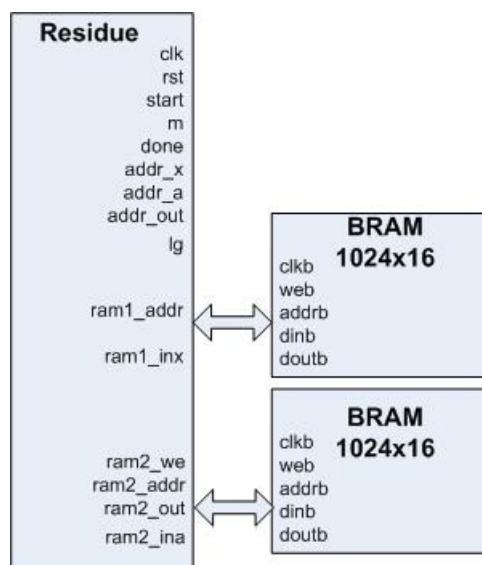


Figura 5.18: Ligação do módulo aos DPBRAMs

valores da entrada x, e na segunda DPBRAM lê os valores de entrada e escrevem os resultados da saída.

As restantes portas do módulo, com exceção da porta de entrada do relógio, estão ligadas a registos acessíveis por Software. A porta de entrada do relógio *clk* está ligada ao relógio da PLB. A tabela 5.10 mostra o mapeamento das portas do módulo aos registos acessíveis por Software, registos esses de 32 bits. Na tabela estão representados apenas os 16 bits mais significativos dos registos. Os outros 16 bits menos significativos não foram utilizados.

Índice	Registos	Bits															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	slv_reg0	start															
0	slv_reg1	m															
1	slv_reg2	done															
2	slv_reg3	lg															
3	slv_reg4	addr_x															
4	slv_reg5	addr_a															
5	slv_reg6	addr_out															
6	slv_reg7	reset															

Tabela 5.10: Mapeamento das entradas e saídas do módulo *Residue* nos registos acessíveis por Software

Capítulo 6

Implementação em Software

Neste capítulo é descrito o software desenvolvido para aceder a Hardware e as modificações efectuadas no código fonte da aplicação para substituir a chamada as funções pelo acesso aos módulos em Hardware.

6.1 Desenvolvimento de Software de acesso a Hardware

Para mapear os módulos escolheu-se usar o dev/mem em vez de desenvolver um kernel driver. O dev/mem proporciona o acesso a memória física do sistema e é usado para aceder a memória IO de Hardware periférico. Ele permite que o sinal seja transmitido a uma alta velocidade mas não é muito seguro por isso só deve ser usado para situações de teste.

O primeiro passo é abrir o dev/mem. Este é aberto da mesma forma que se abre um ficheiro, usando a função `open`. Para leitura e escrita deve ser aberto usando o modo `O_RDWR`.

```
#define RESIDUE_PATH    "/dev/mem"

Word32 Open(void)
{
    fd = open(RESIDUE_PATH, O_RDWR);
    if (fd == -1){
        perror("Error opening file.");
        return -1;
    }

    return 0;
}
```

Para fechar o dev/mem usa-se:

```
close(fd);
```

Após abrir o dev mem, cada um dos módulos e as RAMs devem ser mapeados na dev/mem. Para mapear usa-se a função *mmap*.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Esta função cria um novo mapeamento no espaço de endereço virtual do processo chamada. O endereço de início do mapeamento é dado por *addr* e caso este seja 0, a função escolhe este endereço de início. O comprimento do mapeamento é dado por *length*. O argumento *prot* permite indicar se é mapeado para escrita e/ou leitura. O conteúdo da página mapeada é inicializado usando *length* bytes, começando a partir de *offset* no objecto referenciado pelo descritor *fd*.

Por exemplo, para mapear o módulo Residue10 em dev/mem, foi usado o exemplo abaixo. Este foi mapeado para leitura e escrita. O endereço base do módulo, RESIDUE10_BASE_ADDRESS, e o endereço máximo, RESIDUE10_HIGH_ADDRESS, são gerados pelo XPS.

```
#define RESIDUE10_BASE_ADDRESS 0x83C3C000
#define RESIDUE10_HIGH_ADDRESS 0x83C3C1FF
#define RESIDUE10_MAP_SIZE (RESIDUE10_HIGH_ADDRESS -
    RESIDUE10_BASE_ADDRESS)

map_residue10 = mmap(NULL, RESIDUE10_MAP_SIZE, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, RESIDUE10_BASE_ADDRESS);
```

Para desmapear usa-se a função *munmap*.

Para transferir os dados, foi criado 4 funções:

- *write_one* - Esta função escreve um valor no Hardware, num dado *offset* especificado.
- *read_one* - Esta função lê um valor do Hardware, num dado *offset* especificado.
- *read_vector* - Esta função lê um conjunto *n* de valores do Hardware, a partir de um *offset* especificado e guarda os valores num vector passado por argumento.
- *write_vector* - Esta função escreve o conteúdo de um vector de tamanho *n* no Hardware, a partir de um dado offset.

6.2 Residue

Nesta secção é descrita as modificações efectuadas no código fonte da aplicação, substituindo a função *Residue* pelas funções do *device driver*, permitindo aceder aos módulos implementados na FPGA.

A arquitectura de Software utilizada para substituição da chamada a *Residue* é:

1. Escrita do vector x na BRAM1;
2. Escrita do vector a na BRAM2;
3. Escrita do valor de lg no registo acessível por Software respectivo;

4. Escrita do endereço onde o módulo deve começar a ler o vector *x* no registo acessível por Software respectivo;
5. Escrita do endereço onde o módulo deve começar a ler o vector *a* no registo acessível por Software respectivo;
6. Escrita do endereço onde o módulo deve começar a escrever o resultado no registo acessível por Software respectivo;
7. Envio do *reset*.
8. Envio do *start*.
9. Espera que o módulo termine, isto é, espera que o bit do *done* seja posto a 1.
10. Leitura dos resultados da BRAM2.

Esta arquitectura é usada para substituir cada uma das chamadas a função *Residue*. No entanto, o primeiro e o segundo ponto desta arquitectura não se verifica em todas as chamadas à função. Tal como descrito no capítulo 3, na maior parte das chamadas a função, é usado o vector *speech* que é global e só é modificada uma única vez para cada *frame* de dados da entrada processado. Sendo assim, basta transferir esse vector para a BRAM1 uma única vez para cada *frame*. O vector *error* por sua vez é calculada antes da chamada da função *Residue* sendo necessário transferi-lo sempre. Este vector é transferido para uma posição de memória que não sobrepõe ao vector *speech*. A janela de convolução $a[n]$ em algumas chamadas à função é usada a mesma janela mas acedendo a posições diferentes. Então esta janela só é transferida quando necessária.

Capítulo 7

Validação e Análise de resultados

Neste capítulo é apresentado os resultados obtidos após a partição Software/Hardware e é feita uma análise destes resultados comparando-os com os resultados sem esta partição. Por fim, é feita uma comparação entre cada um dos casos de teste, mostrando as vantagens e desvantagens de cada um.

No capítulo 4 é mostrada a estratégia usada para verificação da correcção dos dados e validação do sistema na placa de desenvolvimento. Após verificar-se a completa correcção dos dados na saída, procedeu-se a medição dos tempos de execução e da determinação (ou não) da aceleração na FPGA Virtex-5.

Para a medição dos tempos de execução foi utilizada a função **gettimeofday()**, presente na biblioteca **<sys/time.h>**, que permite medir tempo de execução de um procedimento ou aplicação em C para sistemas Linux. Esta função permite declarar variáveis do tipo **struct timeval**. A **struct timeval** é uma estrutura que contém dois atributos, **tv_sec** (os segundos de agora) e **tv_usec** (os microsegundos do segundo de agora).

Na aplicação em C do G.729, foi criada duas variáveis do tipo **struct timeval**, sendo uma para a determinação do tempo inicial e o outro para o tempo final, tal como mostrado de seguida.

```
struct timeval startTime , endTime;
```

Para medir o tempo de execução de uma função, a função **gettimeofday()** é chamada antes e depois dessa função, obtendo o tempo inicial e final de execução.

```
gettimeofday(&startTime , NULL);  
  
Coder_ld8c(prm , frame , dtx_enable , rate);  
  
gettimeofday(&endTime , NULL);
```

E por fim é determinado o tempo total da execução da função.

```
seconds = endTime.tv_sec - startTime.tv_sec;  
useconds = endTime.tv_usec - startTime.tv_usec;  
time = ((seconds*1000) + (useconds/1000.0)) + 0.5;
```

Para minimizar os erros de medição, cada um dos resultados temporais a seguir apresentados, foram obtidos a partir de, pelo menos, quatro amostras.

7.1 Residue

7.1.1 Tempos de processamento

Na tabela 7.1 é mostrado o tempo total da execução da aplicação na plataforma de desenvolvimento ML507 para os três casos de teste de implementação da função *Residue* em Hardware e para a implementação só em Software. Tal como verificado na tabela, em todos os três casos de teste obteve-se melhores desempenhos temporais do que na implementação só em Software. O objetivo de acelerar a aplicação foi conseguida. Os melhores resultados foram obtidos no teste 1, em que foram implementados na placa os módulos *Residue10* e *Residue30*, que são completamente paralelizáveis e também os módulos mais rápidos. As diferenças de resultados entre os testes 2 e 3 são mínimas embora o teste 2 apresenta melhores resultados. Tal situação já era esperado, visto que, o módulo *Residue* tem o mesmo funcionamento que o módulo *Residue10* e *Residue30_10*. A melhoria obtida no teste 2, deve-se ao aproveitamento de se ter duas especializações e assim conseguir minimizar a transferência de dados para as memórias.

Nível de Optimização	DTX	Bitrate (kb/s)	Tempo (ms)			
			Teste 1	Teste 2	Teste 3	Software
2	Activo	6,4	3408	3422	3425	3540
		8,0	3485	3499	3499	3618
		11,8	12007	12049	12092	12798
	Não activo	6,4	8596	8595	8596	8748
		8,0	10350	10402	10398	10541
		11,8	11825	11846	11912	12639

Tabela 7.1: Tempo total de execução da aplicação

Para ser mais claro perceber a aceleração obtida com a partição Software/Hardware, nas tabelas 7.2, 7.3 e 7.4 são mostradas as percentagens de aceleração. A máxima aceleração obtida foi de 6.44% no teste 1 com *bitrate* de saída de 11.8kb/s e DTX não ativo. Para os valores padrão de codificação de 8.0kb/s de *bitrate* de saída e DTX não activo, a máxima aceleração obtido foi 1.81% também no teste 1. Essas percentagens de aceleração são bastantes baixas, o que mostra que o efeito da aceleração de apenas uma função em Hardware não é suficiente para se obter todo o sistema bastante acelerado. Para isto, seria necessário a implementação de mais funções. Outra conclusão a tirar dessas tabelas é que nos casos de maior "esforço" de processamento da aplicação, isto é, *bitrate* de saída de 11.8kb/s e/ou DTX não ativo, a aceleração em Hardware trouxe mais valias ao sistema.

Na tabela 7.5 é mostrado o tempo de processamento de apenas um segmento de fala. À semelhança dos resultados obtidos na tabela 7.1, houve aceleração, face à Software, nos três casos de teste e as maiores acelerações foram obtidas no teste 1.

DTX	Bitrate (kb/s)	Tempo(ms)		Aceleração (%)
		Teste 1	Software	
Activo	6,4	3408	3540	3,73
	8,0	3485	3618	3,68
	11,8	12007	12798	6,19
Não activo	6,4	8596	8748	1,74
	8,0	10350	10541	1,81
	11,8	11825	12639	6,44

Tabela 7.2: Percentagens de melhorias no Teste 1

DTX	Bitrate (kb/s)	Tempo(ms)		Aceleração (%)
		Teste 2	Software	
Activo	6,4	3422	3540	3,33
	8,0	3499	3618	3,28
	11,8	12049	12798	5,85
Não activo	6,4	8595	8748	1,75
	8,0	10402	10541	1,32
	11,8	11846	12639	6,27

Tabela 7.3: Percentagens de melhorias no Teste 2

DTX	Bitrate (kb/s)	Tempo(ms)		Aceleração (%)
		Teste 3	Software	
Activo	6,4	3425	3540	3,26
	8,0	3499	3618	3,29
	11,8	12092	12798	5,52
Não activo	6,4	8596	8748	1,75
	8,0	10398	10541	1,36
	11,8	11912	12639	5,75

Tabela 7.4: Percentagens de melhorias no Teste 3

Nível de Optimização	DTX	Bitrate (kb/s)	Tempo (us)			
			Teste 1	Teste 2	Teste 3	Software
2	Activo	6,4	988	995	995	1067
		8,0	1012	1018	1018	1092
		11,8	3633	3635	3651	3942
	Não activo	6,4	2575	2576	2578	2666
		8,0	3127	3143	3140	3228
		11,8	3568	3567	3586	3874

Tabela 7.5: Tempo de processamento de um segmento de fala

Para mostrar o peso que a transferência de dados de/para as memórias BRAMs tem no desempenho temporal da aplicação, nas tabelas 7.6 e 7.7 são mostrados os tempos de leitura e escrita das BRAMs. O vector *speech*, sendo o vector de maior dimensão, apresenta maior tempo de transferência. No entanto, este vector é transferido uma única vez para os BRAMs. Os vectores *A_t_bwd* e *A_t_fwd_q* são também transferidos uma única vez para os BRAMs. Para além desses vectores, outros vectores são também transferidos, tais como o vector *erro*. Mas a transferência desses vectores, dá-se dentro de ciclos *for* o que tornou difícil a sua medição, e por outro lado, esses vectores são menos expressivos para os resultados. A leitura dos dados das BRAMs acontece sempre, o que faz com que tenha uma expressão significativa no tempo de processamento da função.

Vector	Comprimento (bits)	Tempo (us)
speech	90	9
A_t_bwd	62	8
A_t_fwd_q	22	5

Tabela 7.6: Tempo de escrita dos vectores nas BRAMs

Vector	Comprimento (bits)	Tempo (us)
res	80	9
	40	6

Tabela 7.7: Tempo de leitura dos vectores nas BRAMs

7.1.2 Comparação entre os três testes

A tabela 7.8 faz um resumo do desempenho temporal e dos recursos usados para cada um dos casos de teste. Pela tabela é possível verificar que o teste 1 é o teste mais rápido mas a que usa mais recursos importantes da FPGA, tais como, os blocos DSP48. Este caso de teste pode ser escolhido para implementação em FPGA em que o desempenho temporal e a aceleração são os principais factores a ter em conta. No caso em que a área e a limitação dos recursos utilizados são os principais factores na implementação em FPGA, o teste 3 torna-se o principal candidato para implementação em FPGA. O teste 2 procura um equilíbrio entre os testes 1 e 2.

	Teste 1	Teste 2	Teste 3
Desempenho temporal	Máxima Aceleração: 6.44%	Máxima Aceleração: 6.27%	Máxima Aceleração: 5.75%
Recursos da FPGA	Nº de blocos DSP48: 42, Nº de BRAMs de 18Kbits: 4	Nº de blocos DSP48: 22, Nº de BRAMs de 18Kbits: 4	Nº de blocos DSP48: 11, Nº de BRAMs de 18Kbits: 2

Tabela 7.8: Resumo dos resultados obtidos

Capítulo 8

Conclusão

Neste projeto foram implementadas em Hardware duas funções, *Residue* e *Lag-Max*, presentes no codificador de fala G.729, com o objetivo de acelerar a aplicação em C do codificador. A plataforma de desenvolvimento ML507 foi utilizada. A aplicação em linguagem C do codificador é executada no processador PowerPC 440 que corre Linux e a comunicação entre Software e Hardware é feita através de um *device driver*.

Foi cumprido o objetivo principal do projeto de aceleração da aplicação, tendo obtido uma percentagem máxima de 6,44% com a implementação do teste 1 da função *Residue*.

Concluí que a partição Software/Hardware permitiu melhorar o desempenho total da aplicação; a exploração do paralelismo das operações permitiu tornar os módulos mais rápidos, refletindo-se no desempenho da aplicação; e que a transferência de dados de/para as memórias BRAMs tem um peso significativo no desempenho total da aplicação.

8.1 Trabalhos Futuros

Três estudos podem ser feitos no projeto já implementado:

- Implementação em Hardware de outras funções para além das já implementadas - No capítulo 2 foram apresentadas quatro funções com maiores tempos de execução. Destas quatro funções apenas duas foram implementadas em Hardware, *Residue* e *Lag-Max*. As outras duas funções poderiam, futuramente, serem também implementadas em Hardware e analisada o impacto que teriam na aceleração da aplicação em C do codificador de fala.
- Implementação das duas funções, *Residue* e *Lag-Max*, na plataforma de desenvolvimento ML507 - No projeto desenvolvido, as duas funções foram implementadas de forma separadas. A implementação das duas funções ao mesmo tempo na plataforma de desenvolvimento poderia trazer melhor desempenho temporal da aplicação. No entanto, os recursos da FPGA são limitados, o que levaria a um estudo de quais os casos de teste em que ambas as funções poderiam ser implementadas juntas na FPGA.

- Exploração da Reconfiguração Dinâmica - A Reconfiguração Dinâmica permitiria a partilha de recursos entre os módulos e assim ultrapassar algumas limitações de implementação na FPGA. No entanto, o processo de reconfiguração dinâmica apresenta um custo temporal o que tornaria necessário explorar até que ponto a reconfiguração dinâmica permitiria acelerar ou não a aplicação.

Referências

- [1] REFLECT consortium. *Deliverable D1.5 - Technical report about application requirements for reconfigurability and hardware templates*, 06 2011.
- [2] INTERNATIONAL TELECOMMUNICATION UNION. *GENERAL ASPECTS OF DIGITAL TRANSMISSION SYSTEMS - CODING OF SPEECH AT 8 kbit/s USING CONJUGATE-STRUCTURE ALGEBRAIC-CODE-EXCITED LINEAR-PREDICTION (CS-ACELP)* - ITU-T Recommendation G.729, 03 1996.
- [3] REFLECT consortium. *Deliverable D1.4 - Technical report of applications delivered by Coreworks*, 06 2011.
- [4] Xilinx Inc. *Virtex-5 Family Overview, Product Specification*, February 6 2009. DS100 (V5.0).
- [5] Xilinx Inc. Field programmable gate array (fpga). <http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm>.
- [6] ALTERA. Fpgas. <http://www.altera.com/products/fpga.html>.
- [7] REFLECT consortium. *Deliverable D2.4 - Technical Report on Generic Architectures and Reconfigurable Schemes*, 09 2011.
- [8] Xilinx Inc. *ML505/ML506/ML507 Evaluation Platform - UG347*, May 16 2011. v3.1.2.
- [9] Jeffrey Osier. Gnu gprof. <http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html>, 1993.
- [10] Mentor Graphics. Modelsim - advanced simulation and debugging. <http://model.com/>.
- [11] Xilinx Inc. Xilinx platform studio. <http://www.xilinx.com/tools/xps.htm>.
- [12] Xilinx Inc. Project navigator overview. http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_project_navigator_overview.htm, 2008.
- [13] Xilinx Inc. Xilinx software development kit (sdk). <http://www.xilinx.com/tools/sdk.htm>.
- [14] DENX Software Engineering. Embedded linux development kit. <http://www.denx.de/wiki/DULG/ELDK>, Nov 2008.
- [15] João Canas Ferreira e Nuno Paulino. *Application Analysis: G.729, Internal Report*, 09 2011.