Universidade
Federal do
Pampa

Autor: Guilherme Marques da Mota

ESPECULANDO A ARQUITETURA DO PROCESSADOR MIPS PARA FILTROS FIR

Trabalho de Conclusão de Curso

BAGÉ
2011

GUILHERME MARQUES DA MOTA

ESPECULANDO A ARQUITETURA DO PROCESSADOR MIPS PARA FILTROS FIR

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Leonardo Bidese de Pinho

Co-orientador: Prof. MSc. Reginaldo da Nóbrega Tavares

Bagé
2011

GUILHERME MARQUES DA MOTA

ESPECULANDO A ARQUITETURA DO PROCESSADOR MIPS PARA FILTROS FIR

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 17 de Dezembro de 2011.

Banca examinadora:

Prof. Dr. Leonardo Bidese de Pinho
Orientador
Engenharia de Computação – UNIPAMPA

Prof. MSc. Bruno Silveira Neves
Engenharia de Computação – UNIPAMPA

Prof. Dr. Fabrício de Oliveira Ourique
Engenharia de Computação – UNIPAMPA

AGRADECIMENTOS

Aos meus pais João Sergio e Amélia Maria por terem me apoiado durante toda a minha graduação.

À minha noiva Katielli pelo apoio e dedicação nos momentos mais difíceis.

Ao meu orientador Prof. Dr. Leonardo Bidese de Pinho por ter encarado essa batalha de me orientar durante este trabalho.

Ao meu co-orientador Prof. MSc. Reginaldo da Nóbrega Tavares por ter proposto o tema do trabalho.

À Universidade Federal do Pampa.

"No meio da dificuldade vive a oportunidade."

Albert Einstein

RESUMO

Neste trabalho é feita uma especulação da arquitetura do processador MIPS usando uma descrição VHDL. Esta especulação tem como alvo um algoritmo de um filtro DSP, mais especificamente, um filtro FIR definido para atenuação das frequências da voz humana em uma música. Para realizar a especulação, é adotada, como referência, uma implementação em VHDL de uma organização MIPS pipeline minimalista do ponto de vista do subconjunto de instruções suportado e sem coprocessadores, obtida em um site de repositórios de projetos de hardware. Partindo do projeto minimalista de referência e do estudo da aplicação, é incluído suporte em hardware para outras instruções complementares. Além da especulação no subconjunto de instruções, o trabalho foca a avaliação de alternativas que supram a ausência do coprocessador de ponto flutuante, tendo em vista que a aplicação necessita a utilização de números do conjunto dos números reais e o processador MIPS adota usualmente ponto flutuante para operações desse conjunto. Em particular, num primeiro momento, é verificada a aplicabilidade, na perspectiva de consumo de ALUTs versus desempenho, da emulação do ponto flutuante através da conversão para aritmética de ponto fixo. Num segundo momento são criados softwares para carregamento dos dados para memória do processador já no formato de ponto fixo e acrescentada uma nova instrução para otimização do processador. Além disso, é usada uma metodologia para validação do processador juntamente ao algoritmo do filtro e um conjunto de experimentos para validação da síntese no FPGA.

Palavras-chave: Descrição de Hardware. Especulação da Arquitetura do Conjunto de Instruções. Processador MIPS. Filtro FIR. VHDL. Síntese em FPGA. Supressão de Voz.

ABSTRACT

This work provides a speculation of MIPS processor architecture using a VHDL description. The speculation has as target a DSP filter algorithm, more specifically, a FIR filter applied to frequency attenuation of the human voice in a song. To perform such speculation, an implementation in VHDL of a minimalist MIPS pipeline organization is adopted, corresponding to a small subset of instructions supported without coprocessors, obtained from a hardware design repository site. From the minimalist design reference and the study of the application, complementary hardware support is included for other key instructions required for the application. Besides the instruction set speculation itself, this work focus also on evaluation of alternatives that fill the lack of a floating point coprocessor, in the perspective that the target application needs to operate on the set of real numbers and that the MIPS processors usually adopts floating point operations to execute such kind of work. In particular, at first, is checked the applicability of floating point emulation conversion to fixed-point arithmetic, based on a comparison of ALUTs consumption versus performance. In a second phase, software is developed for uploading data to processor's memory already in the fixed-point format. Moreover, ISA is extended with a new instruction to optimize processor performance regarding the target voice suppression application. In addition, a methodology is proposed and evaluated to validate the efficiency of the filter algorithm applied to voice suppression in a given music and a set of basic experiments to demonstrate the correctness of the FPGA synthesis of the speculated MIPS architecture.

Keywords: Hardware description. ISA speculation. MIPS processor. FIR filter. VHDL. FPGA synthesis. Voice supression.

LISTA DE SIGLAS

ALUT – *Adaptive Look-Up Table* (Unidade Lógica do FPGA Stratix II)

ASIC - A*pplication Specific Integrated Circuit*

CAD - *Computer Aid Design*

CPI - *Cycles Per Instruction*

DSP - *Digital Signal Processor*

FIR - *Finite Impulse Response*

FPGA - *Field Programmable Gate Array*

FPU - *Floating Point Unit*

GCC - GNU *Compiler Collection*

HDL - *Hardware Description Language*

IEEE - *Institute of Electrical and Electronics Engineers*

IIR - *Infite Impulse Response*

LUT – *Look-Up Table* (Unidade Lógica do FPGA Virtex-4)

MIPS - *Microprocessor without Interlocked Pipeline Stages*

PC - *Personal Computer*

PCB - *Printed Circuit Board*

RISC - *Reduced Instruction Set Computer*

TCC - Trabalho de Conclusão de Curso

ULA - Unidade Lógica Aritmética

VHDL - *Field Programmable Gate Array*

VHSIC - *Very High Speed Integrated Circuits*

LED – *Light-Emitting Diode*

SUMÁRIO

1. INTRODUÇÃO

As operações de sinais envolvidas na construção de diferentes sistemas como, por exemplo, sistemas de comunicação, controle, sensoriamento remoto e processamento de sinais biológicos podem ser feitos de duas formas: processamento analógico de sinais ou processamento digital de sinais. O processamento digital de sinais é feito por computador ou microprocessadores DSP. Para implementar estas operações, é necessário projetar um sistema que será programado em um hardware para realizar a função de interesse. Os microprocessadores INTEL, comumente encontrados nos computadores pessoais (PC), são sistemas com característica de uso geral, podendo executar um grande número de funções, atendendo assim um maior número de usuários que fazem uso de computadores pessoais para suas atividades. No entanto o uso de PCs para processamento de sinais não se apresenta como uma boa solução, pois o desenvolvimento de uma aplicação específica busca maximizar o desempenho do sistema. Contrários aos microprocessadores de uso geral podem ser destacados os microprocessadores DSP que são específicos para processamento digital de sinais. Uma boa alternativa disponível hoje para implementar sistemas de DSP de alto desempenho, é a utilização de FPGA (*Field Programmable Gate Array* - Arranjo de Portas Programável em Campo) [1].

Tanto os filtros IIR (*Infinite Impulse Response* - Resposta de Impulso Infinita) quanto os filtros FIR (*Finite Impulse Response* - Resposta de Impulso Finita) podem ser implementados em processadores convencionais, entretanto o requisito de velocidade de processamento de algumas aplicações impede que os filtros FIR sejam implementados em processadores DSP devido à alta demanda de processamento necessária [2].

No mercado de sistemas embarcados, os projetistas trabalham dentro de rigorosas restrições como tamanho e custo de energia. Neste contexto, o uso de microprocessador de "propósito geral" é ineficiente uma vez que a maioria das aplicações nunca usa um grande subconjunto dos recursos que ele oferece. Para estas aplicações, uma solução é microprocessadores personalizado com reduzido conjunto de instruções, mas com instruções personalizadas adicionadas especificamente para a aplicação pretendida [3].

Quando uma parte de um programa é identificada como trecho crítico, responsável por parte significativa do tempo de execução, uma alternativa é transformar esta parte numa instrução especial.

Na época dos processadores com microcódigo isso era relativamente fácil, mas os processadores RISC (*Reduced Instruction Set Computer*) foram projetados excluindo todas as instruções complexas. A ferramenta C2H (*C to Hardware*) da Altera facilita este tipo de transformação para o seu processador NiosII, mas tem diversas restrições. Em [3] os blocos lógicos que implementam as novas instruções se encaixam melhor no resto do projeto de um processador MIPS tradicional [4].

Os sistemas de computação embarcados nos diferentes dispositivos utilizados no dia a dia estão auxiliando cada vez mais as atividades das pessoas, que passam a depender mais fortemente do desempenho desses sistemas. À medida que mais pessoas são beneficiadas pelas máquinas, mais prejuízo é causado pelos problemas ocorridos no funcionamento destas [5][6].

A tolerância a falhas pode ser alcançada utilizando a técnica de redundância de *hardware* na qual são replicados componentes, unidades de memória, fontes de alimentação, dentre outros, com a finalidade de detecção de erros ou reparo do sistema transferindo as tarefas de um componente falho para outro redundante [7].

A necessidade de aplicações embarcadas de alto desempenho, como processamento de vídeo e aplicações industriais de tempo crítico, tem forçado os desenvolvedores de sistemas a buscarem arquiteturas adequadas a este propósito. Um fator relevante na escolha de uma plataforma é a necessidade ou não de aritmética de ponto-flutuante. Arquiteturas que utilizam ponto-fixo são normalmente mais rápidas, baratas e tem menor consumo de energia [8], entretanto, por não possuírem *hardware* específico para aritmética de ponto-flutuante, estas são obrigadas a empregar técnicas de *software* para contornar tal limitação [9].

Aplicações que possuem pouca necessidade de funções que utilizem ponto-flutuante podem optar por uma arquitetura de ponto-fixo. Esta possibilidade só é valida quando se aumenta significativamente a capacidade de processamento das arquiteturas de ponto-fixo, permitindo que estas possam satisfazer as necessidades de aritmética de ponto-flutuante sem exigir um processador do mesmo tipo [8][9].

1.1. Motivação

Diante do cenário exposto no item anterior, surge à motivação de fazer uma especulação na arquitetura do processador MIPS, que é um processador de propósito geral.

Esta especulação tem como objetivo descrever uma versão do processador MIPS dedicado para uma aplicação especifica. A aplicação que será utilizada será um filtro FIR, que tem como uma de suas características utilizar o conjunto de números reais. Conforme a conceituação do processador MIPS na Seção 2.1 a maneira que ele trabalha com o conjunto dos números reais é através de ponto flutuante que é executado em uma unidade dedicada para esse tipo de dados, que é uma unidade de ponto flutuante que é utilizada como um coprocessador do processador MIPS.

Como o objetivo desse trabalho é a descrição de um processador dedicado para uma devida aplicação, pode ser feito um estudo dos requisitos dessa aplicação e assim verificar a necessidade da utilização coprocessador de ponto flutuante. Se for verificado que não haja necessidade de uso de coprocessador pode ser mudada a forma com que o processador faz a aritmética com o conjunto dos números reais.

1.2. Objetivo

Tem como objetivo central do trabalho a especulação da arquitetura de processador de propósito geral voltado para um algoritmo DSP, mais especificamente um filtro FIR. O processador a ser utilizado nesse trabalho será o MIPS. O objetivo da especulação é buscar melhorias na arquitetura de propósito geral, para que essa arquitetura possa atender uma aplicação especifica de alta demanda. Essas melhorias podem ser alcançadas através da criação de novas instruções customizadas e remoção de algumas instruções desnecessárias para à aplicação. Essas modificações e adequações no processador serão feitas de tal forma a não o descaracterizar e ainda assim atingir os objetivos do trabalho.

2. CONCEITUAÇÃO

Este capítulo aborda os principais conceitos básicos necessários para compreensão e desenvolvimento do trabalho. A Seção 2.1 contém uma descrição geral do processador MIPS que é o processador alvo do trabalho. A Seção 2.2 apresenta o padrão IEEE-754 de ponto flutuante, que é o padrão utilizado pelo processador MIPS para representação de números reais. A Seção 2.3 aborda a aplicação escolhida para especulação da arquitetura do processador. A Seção 2.4 apresenta o fluxo de projeto VHDL que foi o utilizado no trabalho para descrição do processador MIPS. A Seção 2.5 descreve o dispositivo FPGA de forma geral. Por fim na Seção 2.6 aborda uma breve descrição da placa DE2 da Altera, a qual foi usada nos experimentos.

2.1. O Processador MIPS

O processador MIPS (*Microprocessor without Interlocked Pipeline Stages*) foi criado nos anos 80 John Hennessy, na Universidade de Stanford. O objetivo da criação do processador era o de explorar o padrão RISC (*Reduced Instruction Set Computing*). Com o sucesso do conceito introduzido por Hennessy em 1984 foi criada a *MIPS Technologies,Inc,*, que colocou o processador MIPS no mercado comercial. MIPS segue a arquitetura RISC com restrito número de formatos de instrução, todas com o mesmo tamanho. Tem 32 registradores de propósito geral que comportam palavras de 32 bits cada [11] [16][17][18].

No inicio da década de 90 o MIPS começou a ser licenciado para terceiros. A simplicidade destes processadores tornou o licenciamento um sucesso. Por ser mais eficiente e ter um custo mais acessível, substituiu processadores de arquitetura CISC nos dispositivos que o utilizavam. Algumas empresas adotaram os processadores MIPS em seus desktops, mas não obtiveram sucesso devido à competição com outros processadores que tinham mais suporte, assim os processadores MIPS caíram em desuso em desktops. Na linha dos dispositivos embarcados a situação foi diferente, o MIPS tornou-se amplamente empregado, sendo utilizado em diversos dispositivos tais como impressoras, televisores, consoles de jogos, PDAs, set-top boxes, entre outros. Com esse sucesso, processadores MIPS são

oferecidas na forma de núcleos sintetizáveis para projeto de sistemas embarcadas. O MIPS permanece nos dias atuais como um dos processadores embarcados mais utilizados na indústria, devido ao conhecimento generalizado por parte de projetistas da arquitetura MIPS e amplo ferramental de desenvolvimento [10].

2.1.1. Características do processador MIPS

Nessa Seção será apresenta as principais características dos processadores MIPS.

2.1.2. Pipeline

A arquitetura MIPS tem suporte a *pipeline*, este é dividido em cinco estágios, cada um desses estágios com faixa de tempo fixa, que geralmente é um ciclo de *clock* do processador A Figura 1 apresenta uma ilustração padrão dos estágios da *pipeline* do MIPS:

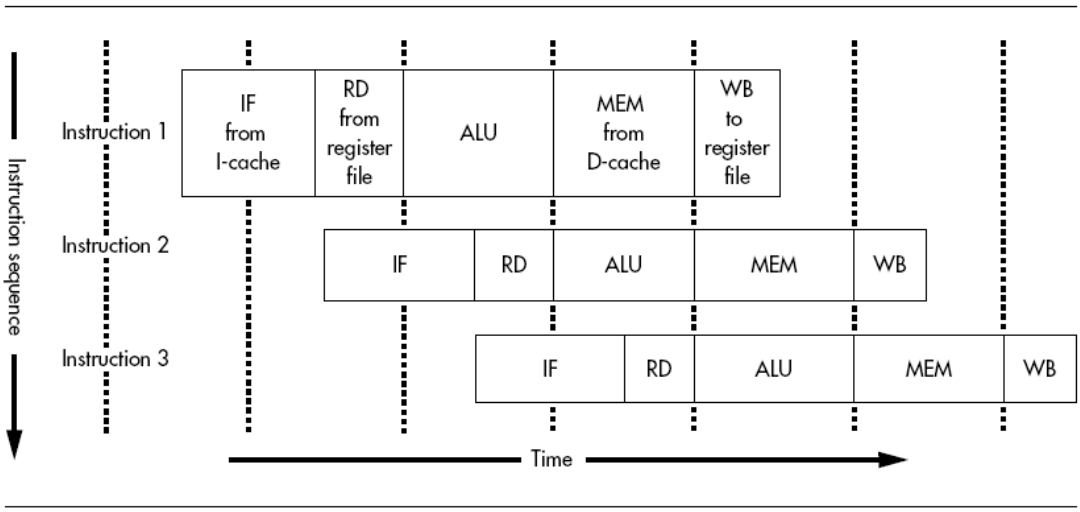

FIGURA 1 - Os Cinco estágios do *pipeline* do MIPS [11].

Cada etapa pode ser resumida conforme o que segue:

- **IF**: Obtém a próxima instrução a partir da memória de instrução;

• **RD**: Obtém o conteúdo dos registradores;

• **ALU**: Realiza uma operação aritmética ou lógica;

• **MEM**: É a fase em que a instrução pode ler ou escrever na memória de dados;

• **WB**: Armazena o valor obtido a partir de uma operação em um registrador;

2.1.3. Instruções

Existem várias características especificas no formato de instruções da arquitetura MIPS. Uma delas, é que todas as instruções têm 32 bits, ou seja, tamanho fixo de palavra. As instruções MIPS podem ser classificadas em quatro grupos, conforme descrito abaixo [11][12][13][17]:

• **R-Type**: Grupo que contém instruções que do tipo aritmética e lógica, com todos os operandos em registradores. Todas as instruções R-Type usam *opcode* 000000, e a operação é comandada pelo campo *function*, e têm o seguinte formato:

opcode (6)	*rs (5)*	*rt (5)*	*rd (5)*	*shamt (5)*	*function (6)*

Onde:
Opcode: 000000;
rs: Registrador com o primeiro operando fonte;
rt: Registrador como segundo operando fonte;
rd: registrador destino;
shamt: campo para o deslocamento;
function: código de função. É associado ao *opcode*.

• **I-Type**: Grupo que contêm as instruções com um operando imediato. Com exceção dos *opcodes* 000000, 0100xx e 00001x, todos os outros são usados por instruções I-Type. As instruções deste grupo têm o seguinte formato:

opcode (6)	*rs* (5)	*rt* (5)	*Imediado* (16)

Onde:

Opcode: Código de operação;

rs: registrador destino;

rt: Registrador como primeiro operando fonte;

imediato: Valor imediato como segundo operando fonte;

• **J-Type**: Grupo composto por instruções de salto. Estas instruções requerem um valor imediato para especificar um de seus operandos. Usam o *opcode* 00001x. Instruções deste grupo tem o seguinte formato:

opcode (6)	*destino* (26)

Onde:

Opcode: Código de operação;

Destino: Valor imediato com valor destino para deslocamento;

• **Coprocessador de Instruções**: Todos os processadores MIPS tem dois coprocessadores padrão, CP0 e CP1. CP0 é para processamento de vários tipos de exceções. O coprocessador denominado CP1 é reservado para o processamento em hardware de números em ponto flutuante. Este coprocessador deve ser compatível com o padrão IEEE-754 e deve possuir banco de registradores próprio de 32 posições com tamanho de 32 bits, designados com $f0 a $f31. Todas as instruções de uso do coprocessador usam o *opcode* 0100xx.

2.1.4. Endereçamento e acesso à memória

A memória no processador MIPS é divida em três partes, conforme a Figura 2. Tal convenção não se trata de uma imposição do hardware, e sim uma convenção a fim de facilitar que programas e ferramentas diferentes possam trabalhar em um formato padrão [11][13][17].

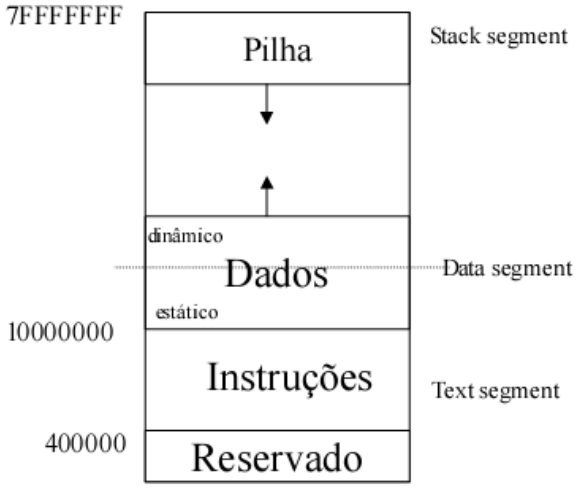

FIGURA 2 - Estrutura da memória do MIPS [13].

A primeira parte, que inicia em 400000_{16} (4194304 em decimal) é o segmento de instruções, trecho de memória onde são alocadas as instruções do programa.

A seção intermediária, iniciada em 10000000_{16} (268435456 em decimal) é o segmento de dados, que por sua vez é seccionado em dois: A primeira seção contém todos os objetos cujo tamanho é conhecido em tempo de compilação e que devem existir durante a execução do programa (dados estáticos). Imediatamente acima desta seção ficam os dados dinâmicos, isto é, aqueles que são alocados durante a execução do programa. Esta Seção é gerenciada pelo Sistema Operacional e pode ser expandida [11][13].

A terceira parte é o segmento da pilha, que reside no topo do espaço de endereçamento a partir de $7FFFFFFF_{16}$ (2147483647 em decimal), e cresce na direção de endereços abaixo [11][18].

Algumas outras características podem ser destacadas, entre elas [11]:

• **A memória é byte-endereçada**: Quando os dados são de um registro de processador MIPS, todas as operações trabalham no registro inteiro, mesmo não o ocupando todo;

• **Carregar/Armazenar devem ser alinhados**: Operações de memória só podem carregar ou armazenar dados de endereços alinhados para se adequar ao tipo de dados a ser transferido;

• **Saltar instruções**: O tamanho das instruções de 32 bits pode ser um problema. O *opcode* menor em uma instrução MIPS é de 6 bits, deixando 26 bits para definir o destino de um salto. Uma vez que todas as instruções são quatro *bytes* alinhados na memória, os dois bits de endereço menos significativos necessariamente não podem ser armazenados, permitindo um intervalo de endereços de 22^8 = 256 MB, o que pode ser um problema, dependendo do tamanho do programa.

2.1.5. Registradores

A fim de padronizar, há uma convenção para os nomes dos registradores do MIPS. A Figura 3 apresenta estes nomes relacionados ao número e à função do registrador [11][16].

Número do Registrador	*Nome*	*Usado para*
0	Zero	Sempre retorna 0
1	At	Reservado para o assembly
2-3	v0, v1	Receber o valor retornado por sub-rotinas
4-7	a0 – a3	(*argumentos*) Parâmetros para sub-rotinas
8-15	t0 – t7	(*temporários*) Para ser usado pelas sub-rotinas
16-23	s0 – s7	Registro de variáveis das sub-rotinas
24, 25	t8, t9	(*temporários*) Para ser usado pelas sub-rotinas
26, 27	k0, k1	Usado por interrupções do SO
28	Gp	(*global pointer*) Apontador de área global
29	Sp	(*stack pointer*)
30	Fp	(*frame pointer*)
31	Ra	Registrador de endereço de retorno

FIGURA 3 - Registradores do processador MIPS.

Conforme é possível observar na Figura 3, existem 32 registradores, cada um deles contendo 32 bits (para o R2000/R3000, especificamente).

Há ainda os registradores especiais, por exemplo: HI e LO. Na arquitetura MIPS a instrução de multiplicação, por exemplo, tem seu resultado distribuído entre os registradores HI e LO, onde os bits mais significativos do resultado são armazenados no registrador HI e os bits menos significativos são alocados no registrador LO [11][12].

2.1.6. Unidade de ponto flutuante do processador MIPS

A arquitetura MIPS segue o padrão desenvolvido pelo *Institute of Electrical and Electronics Engineers* (IEEE) 754, o *IEEE Standard forBinary Floating-Point Arithmetic* [19]. Este padrão define os tipos de dados em ponto flutuante, as operações aritméticas, de comparação e de conversão básicas, e um modelo computacional.

A unidade de ponto flutuante (**FPU**) do processador MIPS oferece suporte aos tipos de dados em ponto flutuante e ao tipos de dados em ponto fixo. Os tipos de dados em ponto flutuante implementados são os (*single precision)* precisão única, e (*double precision)* precisão dobrada, como definidos pelo IEEE 754. O tipo de dado em ponto fixo e o de inteiros sinalizados, fornecido pela arquitetura da CPU [14]. A estrutura do processador MIPS com seus coprocessadores pode ser observada na Figura 4.

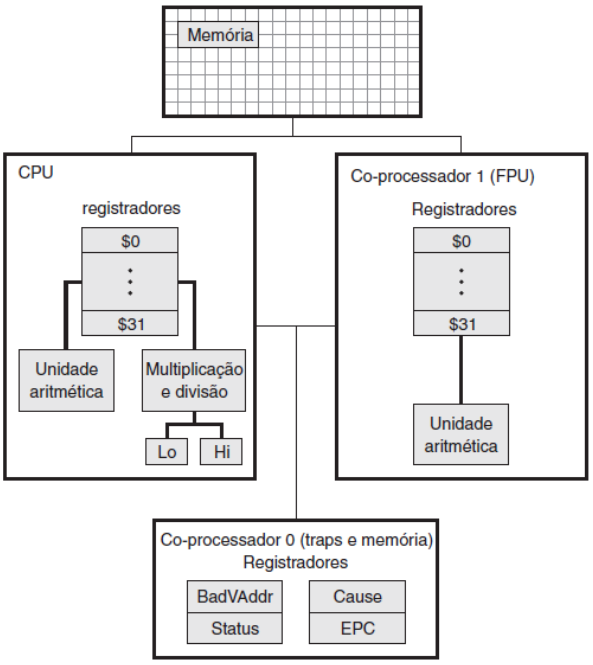

FIGURA 4 - Estrutura do processador MIPS [13].

2.2. Padrão IEEE-754 para formato ponto flutuante

O Instituto de Engenharia Elétrica e Eletrônica (IEEE), elaborou um padrão para ponto-flutuante chamado "*IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*" [19]. Este padrão define a precisão, formato, valores e operações sobre valores ponto-flutuante.

Seu formato permite a representação de valores muito pequenos, assim como valores muito grandes utilizando um conceito similar à notação científica, através de sinal/mantissa/expoente, conforme descrito por :

$$-1^{sign} * d.ddd...d * b^{exp}$$

Onde 'b' representa a base, 'd' representam os dígitos da mantissa no intervalo [0, base - 1], '*exp*' o expoente e '*sign*' o sinal em notação sinal-magnitude.

Ao contrário do formato ponto-fixo, onde o ponto binário é fixo, limitando a precisão e magnitude de seus valores, o formato ponto-flutuante permite cobrir um grande domínio de valores reais para uma quantidade de dígitos relativamente pequena para sua representação.

São duas as representações definidas pelo padrão ponto-flutuante IEEE: a de precisão simples, que utiliza uma palavra de 32 bits, e a de precisão dupla, que emprega uma palavra de 64 bits.

Para a precisão simples, a palavra de 32 bits é composta por um campo de sinal de 1 bit, representado no formato sinal-magnitude, um campo de 8 bits para expoente, representado na base 2, deslocado somando-se o valor 127 (1023 para precisão dupla) ao valor original do expoente, e por um campo de mantissa composto por 23 bits [20].

A mantissa é normalizada para que exista um dígito diferente de zero à esquerda do ponto binário, ajustando-se o expoente conforme o necessário. Como todos os números são normalizados, não há necessidade de se armazenar o bit do ponto binário, pois este sempre será 1. Este bit é chamado de bit oculto e, por não ser armazenado, fornece uma precisão adicional de um bit ao formato.

O padrão também descreve o comportamento de operações não definidas como divisão por zero e valores especiais como infinito e NaN ("*Not a Number*") .

2.3. Filtros digitais

Os filtros digitais são classificados corriqueiramente em 2 tipos: filtro de Resposta Finita ao Impulso (*FIR Finite Impulse Response*, somente entradas prévias ou não-recursivo) e filtro de Resposta Infinita ao Impulso (*IIR Infinite Impulse Response*, entradas e saídas prévias ou recursivo). Essa classificação refere-se mais especificamente ao tipo de resposta ao impulso unitário, mas existem outras classificações para filtros digitais. De acordo com a parte do espectro de frequência que deixam passar ou atenuam podem ser do tipo passa alto, passa baixo, passa banda, rejeita banda, entre outros. De acordo com sua ordem podem ser de primeira ordem, de segunda ordem, etc., e podem ainda ser classificados de acordo com a estrutura que os implementa. Nesse trabalho está sendo considerada apenas a utilização de filtros FIR [39].

2.3.1. Filtros FIR

Filtros digitais de resposta finita ao impulso (FIR) são amplamente utilizados na área de processamento digital de sinais, com destaque para os aspectos de estabilidade e fácil implementação devido à simplicidade das operações envolvidas no cálculo. Filtros FIR são estruturas responsáveis pelas características do processo de filtragem, e respondem pela duração da resposta impulsional deste filtro. Um filtro FIR pode ser implementado usando tanto técnicas recursivas quanto não recursivas, mas comumente técnicas não recursivas são usadas. Os coeficientes de um filtro, que são um conjunto de constantes, também chamados de *tap weight*, são usados para multiplicar amostras atrasadas do sinal de entrada com a estrutura digital do filtro. O projeto de um filtro digital é a tarefa de determinar os coeficientes do filtro que renderão a resposta em frequência desejada. Para um filtro FIR, por definição, os coeficientes do filtro são a resposta ao impulso do filtro. Na Figura 5 é apresentada a ilustração da estrutura de um filtro, onde o número de elementos de atraso determina a duração finita da sua resposta ao impulso (também chamada de ordem de um filtro FIR), representados por T [28][29].

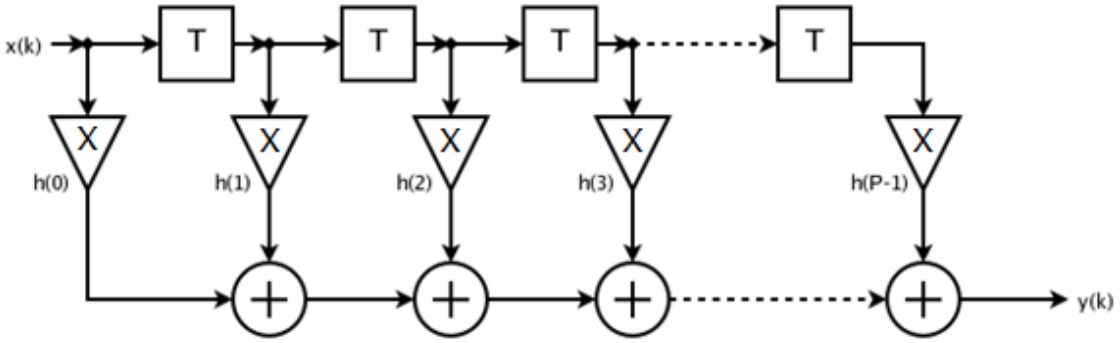

FIGURA 5 - Filtro FIR.

Os filtros FIR são utilizados no processo de filtragem no domínio de uma determinada frequência. Eles podem diferir na forma de atuação sobre o sinal. São filtros estáveis, e de fácil implementação. Este filtro, chamado filtro de resposta ao impulso de duração finita, é utilizado para filtragem adaptativa em tempo real. Sua ordem é baseada no número de elementos de atraso, e sua saída *y* é determinada pela equação abaixo [28][29].

$$y(n) = \sum_{i=0}^{P} h_i x(n-i)$$

Um filtro FIR realiza o processo de convolução no domínio do tempo somando os produtos das amostras de entrada deslocadas com a sequência de coeficientes do filtro. A sequência de saída de um filtro FIR é igual à convolução da sequência de entrada pela resposta ao impulso do filtro (coeficientes), conforme a equação acima. Percebe-se que o processo de convolução realizado por um filtro FIR é um somatório de produtos [28][29].

2.3.2. Técnica de janelamento de sinais

Em aplicações que envolvem a amostragem de sinais obtém-se somente uma gravação finita do mesmo. Isso resulta em uma forma de onda com características espectrais diferentes do sinal original, resultando em uma perda de informação. Para aumentar as características

espectrais de um sinal amostrado utiliza-se a técnica de janelamento sobre o mesmo. Ao analisar um sinal amostrado através de Fourier ou outro método de análise espectral, essa técnica minimiza as margens de transição em formas de onda truncadas, reduzindo assim a perda espectral. Existem várias razões para a utilização do janelamento de sinais segundo [40]. Algumas delas são:

- Definição da duração do período de observação do sinal.
- Redução da perda espectral.
- Separação de um sinal de pequena amplitude de um sinal de grande amplitude com frequências muito próximas uma das outras.

Aplicando uma janela a um sinal que se encontra no domínio do tempo se equivale a multiplicar o sinal pela função que representa a janela. Devido à multiplicação no domínio do tempo ser equivalente à convolução no domínio da frequência, o espectro de um sinal janelado é a convolução do espectro do sinal original com o espectro da janela. Dessa maneira, o janelamento modifica a forma do sinal tanto no domínio do tempo quanto no da frequência [40].

Existem vários tipos de janelas disponíveis para análises e algumas dessas são listadas a baixo:

- Retangular (Nenhuma)
- Hanning
- Hamming
- Triangular

2.3.3. Filtro FIR utilizado na primeira etapa do trabalho

O código do filtro FIR em Linguagem de C usado neste trabalho, conforme descrito nesta seção, foi obtido no trabalho [30]. Para compilar este código para o processador MIPS foi utilizado o compilador GCC, o qual gerou o código em Linguagem de Montagem (Assembly) exposto na sequência.

Código filtro FIR na Linguagem C:

```
// FILTRO FIR
//
// y += h[k] * x[(oldest + k) % N];
int main(void)
{
      float x[5], h[5];
      float sample, y, mul_float, output;
      int oldest=0,a,b,k,div,mul,mod;
      //inicializando vetor x
      x[0]=0;x[1]=0;x[2]=0;x[3]=0;x[4]=0;
      //vetor de coeficientes
      h[0]=-0.005627475;
      h[1]=-0.044131055;
      h[2]=0.9229929;
      h[3]=-0.044131055;
      h[4]=-0.005627475;
      b=5;
      while(1)
      {
            a=464235435; //IDENTIFICADOR DO INICIO DE PROGRAMA -
      x"1BABABAB"
            sample=452.587; // read sample
            y=0;
            for (k=0;k<5;k++)
            {
                  //funcao mod
                  a=oldest+k;
                  div=a/b;
                  mul=div*b;
                  mod=a-mul;
                  mul_float = h[k] * x[mod];
                  y = y + mul_float;
            }
            //funcao mod
            a=oldest+1;
            div=a/b;
            mul=div*b;
            mod=a-mul;
            //oldest = (oldest + 1) % N;
            oldest = mod;
            output=y; // write output
            a=481078444; //IDENTIFICADOR DO FIM DE PROGRAMA -
      x"1CACACAC"
      }
}
```

Código filtro FIR Assembly MIPS:

```
 # -G value = 8, Cpu = 3000, ISA = 1
```

```
 # GNU C version egcs-2.90.23 980102 (egcs-1.0.1 release) (sde) [AL
1.1, MM 40] Algorithmics SDE-MIPS v4.0.5 compiled by GNU C version
egcs-2.91.57 19980901 (egcs-1.1 release).
 # options passed:
 # options enabled:  -fpeephole -fkeep-static-consts -fpcc-struct-
return
 # -fcommon -fverbose-asm -fgnu-linker -fargument-alias -msplit-
addresses
 # -mgas -mrnames -membedded-data -meb -mmad -marg32 -mdebugh -
mdebugi
 # -mmadd -mno-gpconst -mcpu=3000

gcc2_compiled.:
      .text
      .align     2
      .globl     main
      .ent  main
main:
      .frame     $fp,104,$ra           # vars= 96, regs= 1/0, args= 0,
extra= 0
      .mask 0x40000000,-8
      .fmask     0x00000000,0
      subu  $sp,$sp,104
      sw    $fp,96($sp)
      move  $fp,$sp
      sw    $zero,64($fp)               #carrega X
      sw    $zero,4($fp)
      sw    $zero,8($fp)
      sw    $zero,12($fp)
      sw    $zero,16($fp)
      li.s  $f0,-4.41310554742813110352e-2        #carrega coeficiente
      s.s   $f0,28($fp)
      li.s  $f0,9.22992885112762451172e-1
      s.s   $f0,32($fp)
      li.s  $f0,-4.41310554742813110352e-2
      s.s   $f0,36($fp)
      li.s  $f0,-5.62747521325945854187e-3
      s.s   $f0,40($fp)
      li    $v0,5           # 0x00000005
      sw    $v0,72($fp)
.L2:
      b     .L4
      b     .L3
.L4:
      li    $v0,464191488               # 0x1bab0000
      ori   $v0,$v0,0xabab              #completa o valor com ORI

      sw    $v0,68($fp)
      li.s  $f0,4.5258700561523437500e2
      s.s   $f0,48($fp)
      sw    $zero,52($fp)
      sw    $zero,76($fp)
.L5:
      lw    $v0,76($fp)                 #verifica número interações
      slt   $v1,$v0,5
```

```
      bne   $v1,$zero,.L8
      b     .L6
.L8:
      lw    $v0,64($fp)                 #y=y+mul_float
      lw    $v1,76($fp)
      addu  $v0,$v0,$v1
      sw    $v0,68($fp)
      lw    $v0,68($fp)
      lw    $v1,72($fp)
      div   $v0,$v0,$v1
      sw    $v0,80($fp)
      lw    $v0,80($fp)
      lw    $v1,72($fp)
      mult  $v0,$v1
      mflo  $a1
      sw    $a1,84($fp)
      lw    $v0,68($fp)
      lw    $v1,84($fp)
      subu  $v0,$v0,$v1
      sw    $v0,88($fp)
      lw    $v0,76($fp)
      move  $v1,$v0
      sll   $v0,$v1,2
      addu  $v1,$fp,24
      addu  $v0,$v1,$v0
      lw    $v1,88($fp)
      move  $a0,$v1
      sll   $v1,$a0,2
      addu  $a0,$fp,$v1
      l.s   $f0,0($v0)
      l.s   $f2,0($a0)
      mul.s $f0,$f0,$f2
      s.s   $f0,56($fp)
      l.s   $f0,52($fp)
      l.s   $f2,56($fp)
      add.s $f0,$f0,$f2
      s.s   $f0,52($fp)
.L7:
      lw    $v0,76($fp)
      addu  $v1,$v0,1
      sw    $v1,76($fp)
      b     .L5
.L6:
      lw    $v0,64($fp)                 #calcula mod
      addu  $v1,$v0,1
      sw    $v1,68($fp)
      lw    $v0,68($fp)
      lw    $v1,72($fp)
      div   $v0,$v0,$v1
      sw    $v0,80($fp)
      lw    $v0,80($fp)
      lw    $v1,72($fp)
      mult  $v0,$v1
      mflo  $a1
      sw    $a1,84($fp)
```

```
      lw    $v0,68($fp)
      lw    $v1,84($fp)
      subu  $v0,$v0,$v1
      sw    $v0,88($fp)
      lw    $v0,88($fp)
      sw    $v0,64($fp)
      l.s   $f0,52($fp)
      s.s   $f0,60($fp)
      li    $v0,481034240              # 0x1cac0000
      ori   $v0,$v0,0xacac
      sw    $v0,68($fp)
      b     .L2
.L3:
.L1:
      move  $sp,$fp                    #fim do programa
      lw    $fp,96($sp)
      addu  $sp,$sp,104
      j     $ra
      .end  main
      .size main,.-main
```

2.3.4. Filtro FIR utilizado na segunda etapa do trabalho

Na segunda etapa do trabalho foi definido um filtro próprio, que tem como característica principal ser um filtro rejeita banda. O filtro foi definido para supressão das frequências da voz humana em um determinado sinal, visando à implementação de uma aplicação de supressão de voz em músicas. Para definição do mesmo, foi utilizado à ferramenta Matlab que já tem a função FIR1 implementada em uma de suas bibliotecas. Essa função tem como característica gerar os coeficientes de um filtro FIR conforme os parâmetros de entrada, que no caso desse trabalho foram o número de coeficientes, e as duas frequências normalizadas. O número de coeficiente utilizado foi 50 e a faixa de frequência de atenuação do filtro foi de 250 a 4000 Hz. O uso da ferramenta pode ser visto na Figura 6.

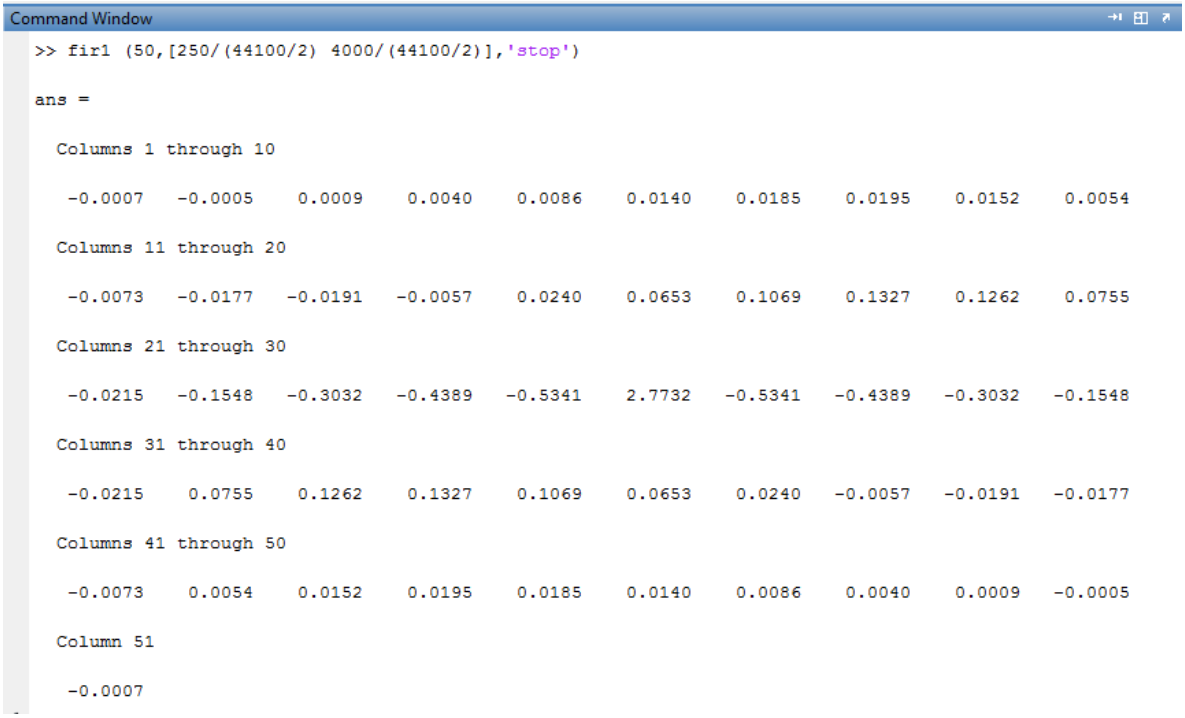

FIGURA 6 – Uso da Função FIR1 da Ferramenta Matlab.

Para realização da convolução entre os coeficientes e o sinal a ser filtrado foi desenvolvido um código na linguagem de alto nível Python que realiza essa operação. Logo em seguida esse código foi convertido manualmente para Linguagem de Montagem (*Assembly*). Esses dois códigos podem ser vistos nas Figuras 7 e 8.

```
1  i0=0
2  i1=0
3  i2 = (f.size-1)
4  i3 = 0
5
6  while i0 < d.size:
7      while i1 < f.size:
8
9          c[i0]= (c[i0] + (d[i3] * f[i2]))
10
11         i2 =i2 -1
12         i1=i1+1
13         i3=i3+1
14
15
16     i2 = (f.size -1)
17     i1=0
18     i0 = i0 + 1
19     i3 =i0
```

FIGURA 7 – Código em Python.

```
1      addi $17,$0,0
2      addi $21,$0,204
3      addi $15,$15,1588
4  L1: addi $26,$0,0
5      addi $15,$15,4
6      add $20,$0,$15
7      addi $19,$0,0
8      addi $16,$0,200
9  L2: beq $19,$21,L1
10     lw $23,0($16)
11     lw $22,0($20)
12     mul $26,$22,$23
13     sw $26,0($15)
14     addi $19,$19,4
15     addi $16,$16,-4
16     beq $17,$0,L2
```

FIGURA 8 – Código em Assembly.

Esse código foi carregado na memória de instruções do processador, os coeficientes e os dados da música foram carregados na memória de dados do processador. Os coeficientes foram carregados nas primeiras posições da memória e os dados da música foram carregados a partir da posição 400 da memória. Para conversão tanto dos coeficientes como dos dados da música foram utilizados *software* desenvolvidos na linguagem de programação Java, os quais convertem para o formato compatível com o processador e com a ferramenta de simulação da Altera. Esses softwares serão discutidos na Seção 3 deste documento.

2.4. Linguagem VHDL

VHDL é um acrônimo para VHSIC *Hardware Description Language* (Linguagem para Descrição de Hardware VHSIC). VHSIC é outro acrônimo que significa *Very High Speed Integrated Circuits* (Circuitos Integrados de Velocidade Muito Alta). Esta linguagem é utilizada para projetos de circuitos lógicos, substituindo ou complementando os diagramas esquemáticos [21].

A VHDL envolve diversos aspectos de projeto, tais como descrição, documentação, verificação e síntese de projetos digitais complexos. Ao abranger estes três objetivos pode poupar grande esforço de desenvolvimento, o que caracteriza o aspecto chave de VHDL [21].

Esta linguagem pode fazer uso de duas diferentes abordagens para a descrição de hardware, que são o método estrutural (*structural* e o comportamental (*behavioral*). Muitas vezes uma composição destas duas abordagens é empregada em um mesmo projeto. A linguagem tem sofrido revisões por parte da comunidade VHDL. A primeira versão foi a de 1987, pois se tornou padrão da IEEE (padrão 1076-1987), algumas vezes denominada como VHDL'87, mas também simplesmente como VHDL. Uma versão mais recente chamada de VHDL'93 que foi adotada em 1994 [21]. Pequenas alterações foram feitas em 2000 e 2002. Em setembro de 2008 foi aprovado pelo REVCOM a mais recente versão, IEEE 1076-2008.

2.5. Dispositivo FPGA

Um dispositivo de lógica programável FPGA (*Field Programmable Gate Array*) é um circuito integrado que contém uma matriz de blocos lógicos configuráveis (*CLBs), flip-flops* e interconexões programáveis entre os blocos lógicos. Diferentemente de um *Circuito Integrado de Aplicação Específica* (ASIC - *Application Specific Integrated Circuit*), que pode executar uma função única e direcionada, um FPGA pode ser reprogramado para executar uma função diferente após alguns microssegundos [22][23].

Os FPGAs foram introduzidos como uma alternativa aos circuitos integrados customizados (*CustomICs*), para a implementação de sistemas em um único chip e para fornecer flexibilidade de reprogramação para o usuário, resultando na melhoria de densidade quando comparado a componentes discretos (aproximadamente 10 vezes). Outra vantagem do FPGA sobre o *CustomICs* é que, com o auxilio de ferramentas *CAD (Computer Aid Design)*, os circuitos passaram a ser implementados em um intervalo de tempo menor (sem processo físico de *layout*, sem produção de mascara, e sem fabricação de circuito integrado específico)[22][23].

Internamente, os FPGAs consistem de interconexões de chaves programáveis eletricamente, conforme a Figura 9, que os diferenciam dos circuitos integrados customizados que são fabricados com interconexões fixas de metal entre os blocos lógicos. O FPGA permite

um modo de configurar as interconexões entre os blocos lógicos e a função de cada bloco lógico [24]. O FPGA pode ser configurado de forma a prover desde funcionalidade de um transistor até a complexidade de um microprocessador [22].

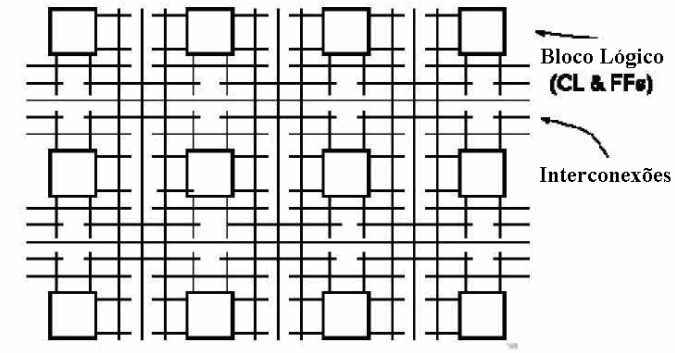

FIGURA 9 - Versão simplificada da arquitetura interna de um FPGA [25].

Uma das vantagens mais importantes no projeto baseado em FPGA é que o usuário pode projetá-lo usando ferramentas *CAD*, fornecidas pelas companhias de automação de projeto. Geralmente o fluxo de projeto de um FPGA inclui os seguintes passos [25]:

• **Projeto de Sistemas**

Nesta etapa o projetista tem que decidir que parte da sua funcionalidade tem de ser implementada em FPGA e como integrar aquela funcionalidade ao resto do sistema.

• **Descrição de Projeto**

Projetistas descrevem as funcionalidades do projeto com editores esquemáticos ou usando uma das várias linguagens de descrição de hardware (HDL) como Verilog [38] e VHDL [21]

• **Síntese**

Concluídos os dois primeiros passos, são usadas ferramentas para implementar o projeto em um determinado FPGA. A síntese inclui otimizações genéricas e otimizações de energia, seguidas por otimizações de locação de elementos lógicos e roteamento. A implementação inclui, dependendo das ferramentas utilizadas, a possibilidade de particionamento do projeto em mais de um dispositivo. A saída da fase de implementação do projeto é o arquivo de *net-list*.

• **Verificação de Projeto**

O arquivo de *net-list* é a entrada para o simulador que verifica a funcionalidade do projeto e informa erros no comportamento do projeto. Ferramentas de *timing* são usadas para determinar a frequência máxima de funcionamento do projeto e por fim o projeto é carregando para o dispositivo de FPGA final onde são feitos testes de trabalho no ambiente real.

2.6. Altera DE2

O kit Altera DE2 é composto por uma placa destinada tanto ao desenvolvimento de produtos finais quanto para uso em ensino que foi projetada por professores para os professores. Seu maior enfoque é o aprendizado de lógica digital, organização de computadores, e FPGAs. Essa placa contém o Altera Ciclone II 2C35 FPGA. Em universidades, é amplamente usada em exercícios de cursos de lógica digital e organização de computadores, viabilizando tarefas simples que ilustram conceitos fundamentais para projetos avançados. A Figura 10 mostra a placa DE2 e seus recursos [33].

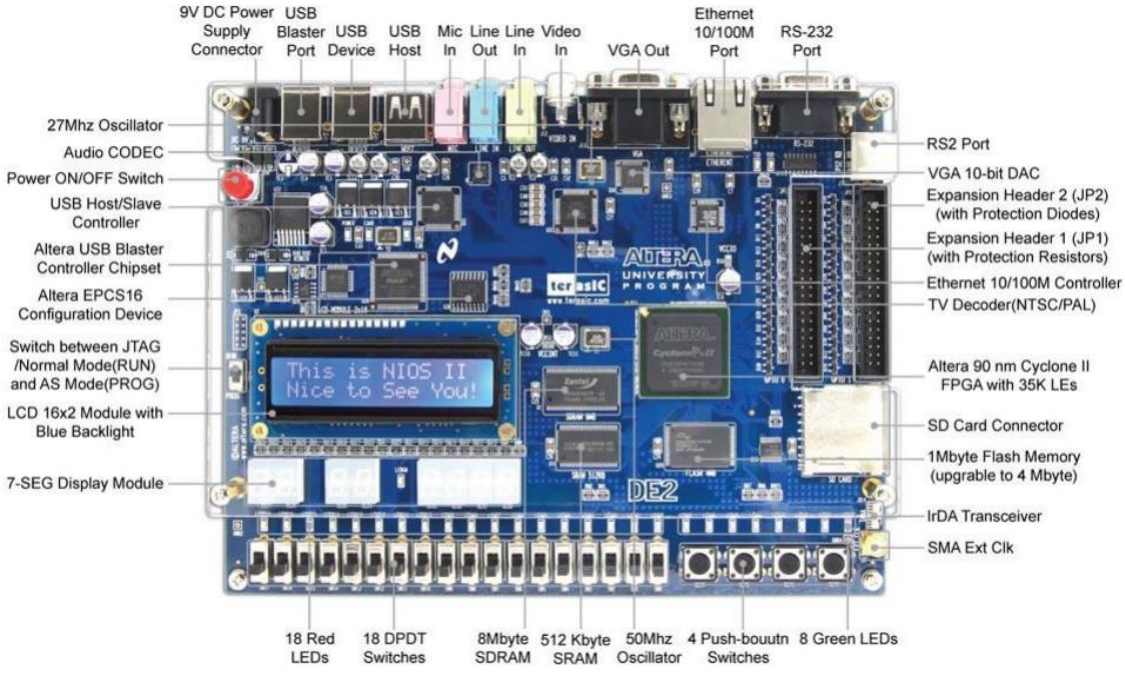

FIGURA 10 – Placa DE2 Altera.

3. ESPECULAÇÃO DA ARQUITETURA

Para atender o objetivo do trabalho de especular a arquitetura do processador MIPS, era necessária uma descrição VHDL de uma organização de processador compatível com essa arquitetura. A primeira alternativa considerada foi a de uma implementação própria. Mas esse caminho foi logo desconsiderado, pois havia a expectativa de que poderia não haver tempo hábil para uma implementação própria validada que permitisse obter resultados preliminares na primeira etapa do trabalho.

Outro caminho discutido no decorrer do trabalho foi o de obter uma implementação do processador MIPS já desenvolvida e validada em outros trabalhos. Com base em pesquisa realizada em repositórios abertos de projeto de *hardware*, foi escolhido o trabalho [26], disponível em [37]. Essa descrição do processador MIPS corresponde à versão R2000 do processador, com cinco estágios de pipeline e com um conjunto reduzido de instruções. Esse conjunto reduzido de instruções contém apenas as instruções do Tipo R (ADD, SUB, AND, OR e SLT) e do Tipo I (LW, SW, LUI e BEQ). Outra característica relevante sobre esta implementação é a ausência de coprocessador de ponto flutuante. A ferramenta utilizada para simulação desse trabalho [26] foi a GHDL [32], caracterizada como um simulador de VHDL baseado no compilador GCC disponível em Linux. Como a escolha do processador para ser utilizado nesse trabalho já havia sido feita em função de um prévio conhecimento da arquitetura em função de estudo da mesma durante algumas disciplinas do Curso de Engenharia de Computação da Unipampa [34], utilizou-se o mesmo critério para a escolha da ferramenta de síntese. Logo, foi necessário migrar o código para a ferramenta Quartus II da Altera [33]. Convém destacar que durante essa etapa surgiram problemas de compatibilidade importantes, os quais serão abordados na Seção 3.1.

Como o processador MIPS utiliza formato de ponto flutuante para trabalhar com o conjunto dos números reais, a ausência de um coprocessador para esse formato fez com que tivesse que ser feita uma pesquisa sobre alternativas para contornar essa deficiência.

A primeira solução e mais óbvia seria a implementação de um coprocessador de ponto flutuante, mas essa não se mostrava muito atraente devido aos requisitos necessários por parte da aplicação. Com isso a solução mais atraente em um primeiro momento foi a emulação do ponto flutuante que é discutida em [9][27].

Em um segundo momento, foi adotada a abordagem de utilização dos dados no formato de ponto fixo. Para isso, os dados são carregados já nesse formato para a memória do processador. Com a utilização dessa abordagem não há necessidade de hardware específico para as operações empregadas nesse tipo de operação, isto é, pode ser utilizado o próprio hardware já existente na arquitetura do processador devido que um valor em inteiro utiliza a representação de ponto fixo, onde a posição do ponto "decimal" está fixa e todos os dígitos são usados para representar o número em si.

3.1. Estudo e adaptação de código VHDL

Nessa seção é exposto como foi adaptado do código VHDL do processador MIPS descrito em [26] para utilização nesse trabalho.

3.1.1. Adaptação de código para ferramenta Altera Quartus II

Primeiramente, para a adaptação do código VHDL do processador MIPS para a ferramenta da Altera Quatus II foi feita uma compilação de todos os arquivos e nessa primeira compilação não foi apresentado nenhum erro. Entretanto, não foi possível gerar dados como utilização do FPGA devido à ferramenta interpretar a memória de instruções como uma descontinuidade, pois essa era descrita como *uma* lógica *SELECT-CASE* (saída dependente das entradas) que se comportava de forma a simular uma memória ROM. Fazendo a compilação individual de cada módulo a ferramenta conseguia gerar esses dados. Com essa compilação individual de cada módulo alguns problemas foram encontrados. O primeiro problema encontrado foi no módulo de acesso à memória, que contém a memória de dados do processador. Essa memória descrita no trabalho [26] foi feita como uma matriz, com 32 x 1024 posições. Para compilação desse módulo a ferramenta gastava em média 45 minutos. Dessa forma optou-se por substituir essa memória por um componente disponível na biblioteca da própria da Altera, contornado o problema do tempo elevado de compilação. A memória utilizada está disponível na biblioteca de módulos da Altera chamada de

MEGAFUNCTIONS, onde há vários módulos disponibilizados pela Altera que podem ser utilizados com suas ferramentas.

Mais especificamente, o módulo que foi utilizado para memória de dados do processador foi o “LPM_RAM_DQ” Figura 11, o qual possui como entradas: o dado para ser escrito, endereço onde deve acontecer à escrita, *clock,* e sinal para ativar a escrita. A memória descrita com essa função foi do mesmo tamanho que a descrita em [26].

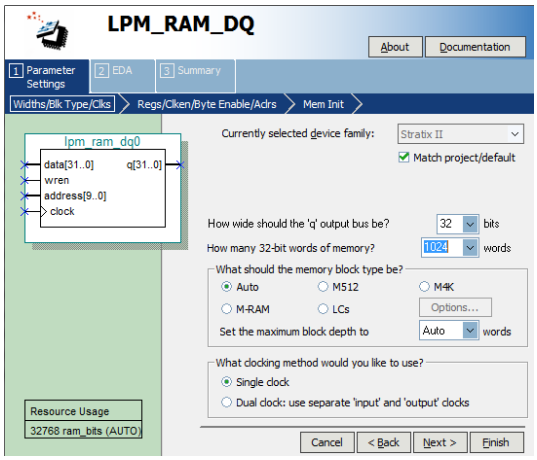

FIGURA 11 – Configuração do Módulo LPM_RAM_DQ.

A entrada e saída de dados são de 32 bits, e são utilizados 9 bits para endereçamento. Como o endereçamento no MIPS é de 32 bits, esses tem que serem truncados para poderem ser utilizados. No MIPS [13] o tamanho da palavra é 32 bits e a memória é endereçada em *byte*. Assim cada palavra do MIPS ocupa 4 posições de memória. No casso da memória implementada, cada posição da memória armazena uma palavra de 32 bits, logo a memória dever ser indexada de 1 em 1 e não de 4 em 4 como em [13]. A solução encontrada para contornar esse problema sem maiores alterações na arquitetura do processador MIPS implementado foi a fazer um deslocamento de dois bits à esquerda no endereço no momento de indexar a memória. Esse deslocamento é equivalente a uma divisão por 4.

Outra modificação feita no código MIPS [26], foi na memória de instruções. Nesse código a memória de instruções foi implementada no formato de um *SELECT CASE*, onde a entrada era o PC, e tinha a instrução correspondente ao PC como saída. A modificação teve que ser feita porque que a ferramenta Quartus II interpretava esse *SELECT CASE* como uma descontinuidade. Logo para implementação da memória de instruções foi utilizada uma segunda instancia do mesmo módulo descrito acima para memória de dados, com uma

diferença, a utilização de um arquivo de inicialização MIF [35]. Este arquivo permite que a memória seja inicializada com o programa a ser executado no processador.

Com a utilização dessas memórias da biblioteca da Altera surgiu outro problema: falta de sincronismo entre os módulos do pipeline. Isso ocorreu porque essa memória da altera por padrão tem registradores nas suas portas de entrada e saída, enquanto na memória do MIPS [13] não existe estes registradores. Na porta de saída, esse problema foi facilmente contornado, uma vez que as memórias adotadas podem ser utilizadas sem os registradores nas suas saídas, mas para as entradas não foi identificada essa opção. A alternativa encontrada foi utilizar esses registradores de entrada da memória como os próprios campos respectivos nos registradores de *pipeline*, implicando na diminuição proporcional na quantidade de bits dos registradores de *pipeline* do projeto original.

3.1.2. Adição de novas instruções

O código VHDL do MIPS [26] não tem implementado todas as instruções básicas do MIPS[13]. Assim as instruções que não estavam presentes precisaram ser implementas conforme indicado pelo estudo da aplicação específica. Com as instruções já implementadas no código, a maioria dos componentes necessários para as instruções básicas faltantes do processador já estão presentes. Dessa forma algumas instruções foram implementadas somente adicionando sinais de controle na unidade de controle.

3.1.2.1. Instruções ADDI e ORI

As instruções ADDI e ORI são do formato I “imediato” citado na Seção 2.1.3. A instrução ADDI soma e a instrução ORI faz um *or* com o valor de um registrador com um operando imediato. As instruções do tipo I são diferenciadas pelo seu *opcode*. Como essas duas instruções são desse formato tivemos que adicionar sinais específicos para essas instruções na unidade de controle, onde esse sinal deve ativar os sinais RegWrite(escreve no registrador), ALUSrc (operando imediato), sendo o primeiro desses sinais para ativar a escrita

no banco de registradores e segundo para controlar o MUX da entrada da ULA que determina se o segundo operando será o valor de um registrador ou um valor imediato. Para a instrução ORI além de ativar esses sinais teve-se que criar um sinal específico para indicar para a unidade de controle da ULA que a operação a ser executada é um ORI.

3.1.2.2. Instruções SLL e SRL

As instruções SLL e SRL são instruções do formato R "registrador" citada na Seção 2.13 e tem como função deslocar o conteúdo de registradores .A SLL faz deslocamento à esquerda e a SRL faz deslocamento à direita. O deslocamento é informado no campo *shamt* da instrução. Estas instruções sendo do tipo R o campo que informa qual operação deve ser realizada é o campo *funct* da instrução.

Para implementação destas instruções foi necessário criar um módulo para deslocamento dentro do terceiro estágio de pipeline e sinais respectivos dentro do controle da ULA. Esse módulo tem como entrada o valor do registrador RT e os *bits* de 6 a 10 do *offset* que corresponde ao campo *shamt* da instrução. A saída desse módulo é valor RT deslocado à esquerda ou à direita que entra em um *mux* que foi criado para ativar essa entrada ou a saída da ULA.

3.1.2.3. Instruções SLLV e SRLV

As instruções SLLV e SRLV tem a mesma função das instruções SLL e SRL, a diferença entre elas é que o deslocamento não é mais comandado pelo campo *shamt* e sim pelo valor do registrador RS. Com isso para implementação destas duas instruções foi somente necessário à criação de *mux* na entrada do módulo de deslocamento e respectivos sinais de controle dentro do controle da ULA. Esse *mux* criado tem como entradas o valor do registrador RS e o campo *shamt* da instrução.

3.1.2.4. Instrução MULT e DIV

As instruções MULT e DIV utilizam dois registradores especiais para armazenar seus resultados, esses registradores são os LO e HI. Para implementação dessas duas operações foi necessário adicionar esses dois registradores. Esses registradores foram adicionados no terceiro estagio do pipeline a etapa de execução.

Na instrução MULT foi utilizado um módulo multiplicador da biblioteca da altera, esse módulo multiplicador utiliza componentes DSP do FPGA e não as ALUTs do dispositivo. Na instrução DIV foram também utilizados os módulos da biblioteca da altera, mas nesse caso não havia um módulo divisor que utilizasse componentes DSP, todos os divisores utilizam as ALUTS do FPGA. Outro fator importante sobre a divisão é que essa operação reduziu a velocidade de *clock* do processador, passando de praticamente 127 MHz para 10 MHz.

3.1.2.5. Instrução MFLO e MFHI

As instruções MFLO e MFHI são instruções complementares a MULT e DIV, essas duas instruções servem para deslocamento de dados dos registradores LO e HI para o banco de registradores do processador.

Para implementação dessas duas instruções foi necessário criar alguns sinais de controle na unidade de controle e adicionar uma entrada no *mux* entre a saída da ULA e o módulo de deslocamento criado para as instruções SLL e SRL.

3.1.2.6. Instrução NOR

A instrução NOR é do formato R “registrador” citado na Seção 2.1.3. A instrução NOR é a negação da instrução. Essa instrução sendo do formato “R” tem como caraterística

ser identificada pelo campo *funct*. Este campo só é acessado na unidade de controle da ULA, logo tivemos que criar sinais específicos para essa instrução dentro dessa unidade.

3.1.2.7. Instrução MUL

A instrução MUL foi criada para personalizar a multiplicação do processador MIPS. Essa instrução tem como objetivo realizar a rotina que foi gerada para realizar multiplicação de dois valores em ponto fixo. Essa instrução tem o formato de uma instrução do tipo R "registrador" citado na Seção 2.1.3. Entretanto, o diferencial dessa instrução quando comparada à instrução de multiplicação do MIPS é o tamanho do resultado gerado pela mesma, que é de trinta e dois bits, ao invés de sessenta e quatro bits. Esses trinta e dois bits que essa operação tem como saída são os bits intermediários dos sessenta e quatro bits que a multiplicação padrão do MIPS tem como saída. Por esse fato só foram criados sinais de controle na unidade de controle do MIPS para identificar essa instrução e adicionada mais uma saída no *mux* que já havia sido criada na saída da ULA para outras instruções.

3.2. Emulando operações de ponto flutuante

Na compilação do código do filtro FIR utilizando o compilador GCC próprio para o processador MIPS, utilizamos a opção (`-msoft-float`). Passando esse parâmetro durante a compilação, o compilador não utiliza o coprocessador de ponto flutuante, substituindo hardware por software ao gerar uma macro para as respectivas operações.

Assim como em [9] e [27], a emulação do ponto flutuante foi feita via software, utilizando o próprio *assembly* do MIPS. A estratégia adotada foi a de transformar todos os valores do formato de ponto flutuante para formato de ponto fixo e realizar as operações nesse formato. Abaixo seguem os códigos em *assembly*.

Código para conversão do formato Ponto Flutuante para Ponto Fixo:

```
          Sll $t1,$to,9
          Srl $t2,$t1,16
          Lui $t3, 1          #mantissa
          Sll $t1,$t0,1
          Srl $t2,$t1,24
          Addi $t1,$t2,-127
          Slt $t2,$t1,$ZERO
          Beq $t2,$ZERO,NEG   #expoente negativo
          Sllv $t2,$t3,$t1    #mantissa + expoente
          J EXP
NEG:      Add $t2,$t1,$ZERO
          Nor $t4,$t1,$t2
          Addi $t1,$t4,1
          Srlv $t2,$t3,$t4    #mantissa + expoente
EXP:      Srl $t3,$t0,31
          Slt $t4,$t3,$zero
          Beq $t4,$zero,NEG2
          J FIM
NEG2:     Srl $t3,$t2,16
          Beq $t3,$ZERO,EXP2  #Parte Inteira Zero
          Add $t3,$t2,$ZERO   #inverte todo
          Nor $t4,$t3,$t2
          Addi $t2,$t4,1
          J FIM
EXP2:     Srl $t3,$t2,16      #inverte so a parte fracionaria
          Sll $t1,$t3,16
          Sll $t3,$t2,16
          Srl $t4,$t3,16
          Add $t3,$t4,$ZERO
          Nor $t2,$t3,$t4
          Add $t4,$t2,1
```

```
          Ori $t2,$t1,$t4
FIM:      Jr $ra
```

3.3. Software na linguagem de programação JAVA

Para utilização dos dados da música e os coeficientes do filtro no formato de ponto fixo e complemento de dois foi necessário desenvolver um software para conversão de valores inteiros para esse formato. No total foram desenvolvidos três softwares: um que converte um valor inteiro para binário no formato de ponto fixo e complemento de dois; outro que faz a operação inversa, convertendo de binário no formato de ponto fixo e complemento de dois para inteiro; e por último um software que converte um valor binário de trinta e dois bits para uma entrada válida do arquivo de inicialização das memórias da Altera que são utilizadas pelo processador descrito nesse trabalho. Os códigos fonte desses três softwares estão nos apêndices deste documento.

3.4. Multiplicação utilizando ponto fixo

Para a filtragem do sinal foi utilizada a convolução que já foi explicada na seção de conceituação. Essa operação nada mais é do que uma soma de produtos. Portanto, as multiplicações se constituem nas principais operações envolvidas nesse processo. Seguindo o princípio de projeto que propõe que os casos comuns devem ser agilizados [13], foi feita uma adequação no caminho de dados do processador para agilizar esta instrução, sem desconsiderar o fato do MIPS usar registradores de trinta e dois bits: foram incorporados dois registradores especiais de trinta e dois bits no terceiro estágio de pipeline para armazenar o resultado de 64 bits. Com isso, inicialmente, para utilizarmos os resultados gerados pelas multiplicações teve que ser criada uma rotina para adaptar o tamanho do resultado gerado pela multiplicação, para que o mesmo seja compatível com o tamanho dos registradores do processador. Essa rotina pode ser vista na Figura 12.

```
mult $22,$23
mfhi $24
sll $24,$24,16
mflo $25
srl $25,$25,16
or $25,$25,$24
```

FIGURA 12 – Código Assembly Multiplicação em Ponto Fixo no MIPS.

Analisando a FIGURA 12, que mostra a rotina para concatenação dos resultados gerados pela multiplicação, nota-se que a técnica empregada é a de descartar os dezesseis bits mais significativos do registrador HI que contém a parte inteira da multiplicação, bem como deslocar os bits restantes para as posições mais altas do registrador. Os dezesseis bits menos significativos do registrador LO que contém a parte fracionária da multiplicação também são descartados, tal que os bits restantes nesse registrador são deslocados para as posições mais baixas do mesmo. Com essas duas etapas realizadas, os valores dos registradores HI e LO podem ser somados e o resultado dessa operação é o valor da multiplicação concatenado para trinta e dois bits.

Como o objetivo do trabalho é o da customização do processador para uma aplicação específica a partir do código da mesma, constatou-se que se essa rotina para multiplicação fosse convertida em uma instrução de máquina seria possível obter um ganho no desempenho do mesmo, pois o número de instruções necessárias para execução do filtro diminuiria. Cabe destacar que nessa rotina há uma grande dependência entre as instruções e isso faz com que alguns ciclos sejam executados com o pipeline vazio, aumentando o número de ciclos necessários para execução da aplicação.

3.5. Validando a integração entre o processador e o filtro

O processo para validação do processador MIPS juntamente ao algoritmo do filtro foi feito em duas etapas. A primeira etapa foi a de executar alguns exemplos menos complexos no processador. Na segunda etapa foi executado o filtro no processador e comparado os

resultados com a execução do filtro em uma linguagem de programação de alto nível (Python). Os resultados foram comparados através de gráficos que são mostrados na Seção 4.

4. RESULTADOS

Nessa seção são abordados os resultados obtidos com a implementação reduzida do processador em termos de consumo de *hardware* do FPGA, número de instruções necessárias para execução do algoritmo do filtro FIR, resultados obtidos com a execução do filtro, síntese no FPGA e uma discussão sobre os resultados obtidos.

4.1. Consumo de células lógicas do FPGA

Para analisar os dados do consumo do FPGA após essa primeira etapa de modificações no código do processador temos que ter como base os resultados da implementação de [26] que representa um subconjunto minimalista do processador MIPS com um conjunto reduzido de instruções. Os dados dessa implementação gerados pela ferramenta da altera estão descritos na Figura 13.

Flow Status	Successful - Thu Jun 30
Quartus II Version	9.1 Build 222 10/21/2009
Revision Name	SEGMENTED_MIPS
Top-level Entity Name	SEGMENTED_MIPS
Family	Stratix II
Device	EP2S15F484C3
Timing Models	Final
Met timing requirements	Yes
Logic utilization	15 %
Combinational ALUTs	1,040 / 12,480 (8 %)
Dedicated logic registers	1,352 / 12,480 (11 %)
Total registers	1352
Total pins	169 / 343 (49 %)
Total virtual pins	0
Total block memory bits	65,536 / 419,328 (16 %)
DSP block 9-bit elements	0 / 96 (0 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)

Clock Setup: 'CLK'

	Slack	Actual fmax (period)
1	N/A	130.23 MHz (period = 7.679 ns)
2	N/A	130.99 MHz (period = 7.634 ns)
3	N/A	132.10 MHz (period = 7.570 ns)
4	N/A	132.98 MHz (period = 7.520 ns)
5	N/A	133.08 MHz (period = 7.514 ns)
6	N/A	134.39 MHz (period = 7.441 ns)
7	N/A	135.21 MHz (period = 7.396 ns)
8	N/A	135.85 MHz (period = 7.361 ns)
9	N/A	136.04 MHz (period = 7.351 ns)
10	N/A	136.61 MHz (period = 7.320 ns)
11	N/A	137.23 MHz (period = 7.287 ns)
12	N/A	137.65 MHz (period = 7.265 ns)
13	N/A	137.74 MHz (period = 7.260 ns)
14	N/A	137.89 MHz (period = 7.252 ns)
15	N/A	137.91 MHz (period = 7.251 ns)
16	N/A	138.01 MHz (period = 7.246 ns)

FIGURA 13 - Resultados gerados pela ferramenta da implementação [26].

Os dados expressos na Figura 13 são da implementação de [26] com as modificações nas duas memórias descrita na Seção 3.1.1.

Na Figura 14 temos os dados para o projeto do processador após essa primeira etapa de modificações para execução do algoritmo do filtro FIR.

Flow Status	Successful - Thu Jun 30
Quartus II Version	9.1 Build 222 10/21/2009
Revision Name	SEGMENTED_MIPS
Top-level Entity Name	SEGMENTED_MIPS
Family	Stratix II
Device	EP2S15F484C3
Timing Models	Final
Met timing requirements	No
Logic utilization	29 %
Combinational ALUTs	2,755 / 12,480 (22 %)
Dedicated logic registers	1,376 / 12,480 (11 %)
Total registers	1376
Total pins	170 / 343 (50 %)
Total virtual pins	0
Total block memory bits	65,536 / 419,328 (16 %)
DSP block 9-bit elements	8 / 96 (8 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)

Clock Setup: 'CLK'

	Slack	Actual fmax (period)
1	N/A	10.63 MHz (period = 94.030 ns)
2	N/A	10.64 MHz (period = 94.011 ns)
3	N/A	10.65 MHz (period = 93.918 ns)
4	N/A	10.65 MHz (period = 93.855 ns)
5	N/A	10.66 MHz (period = 93.815 ns)
6	N/A	10.68 MHz (period = 93.653 ns)
7	N/A	10.68 MHz (period = 93.648 ns)
8	N/A	10.68 MHz (period = 93.641 ns)
9	N/A	10.69 MHz (period = 93.545 ns)
10	N/A	10.69 MHz (period = 93.525 ns)
11	N/A	10.69 MHz (period = 93.506 ns)
12	N/A	10.70 MHz (period = 93.492 ns)
13	N/A	10.70 MHz (period = 93.490 ns)
14	N/A	10.70 MHz (period = 93.483 ns)
15	N/A	10.70 MHz (period = 93.476 ns)
16	N/A	10.70 MHz (period = 93.471 ns)

FIGURA 14 - Resultados gerados pela ferramenta após primeira etapa de modificações.

Analisando as Figuras 13 e 14 percebe-se que o consumo de ALUTs entre as duas implementações teve um acréscimo de 1715. O número de registradores se manteve praticamente o mesmo, com algumas variações decorrentes das modificações para sincronia de componentes e inclusão de dois registradores, LO e HI que são utilizados para multiplicação e divisão.

Outro ponto que pode ser notado é a utilização de módulos DSP da biblioteca da Altera, que são utilizados para a implementação do multiplicador, os quais estão ausentes na primeira versão desses blocos. O dado mais significativo que pode ser extraído com as duas implementações é a velocidade de *clock* que teve um decréscimo significativo, sendo a principal responsável por esse decréscimo à inclusão do bloco divisor que fez a velocidade de *clock* diminuir cerca de treze vezes, passando de aproximadamente 127 MHz para aproximadamente 10MHz.

Na terceira etapa de modificações, foi retirado o divisor, aumentado o tamanho da memória e adicionada à instrução personalizada para multiplicação. A exclusão do divisor ocorreu devido à definição do algoritmo próprio para filtragem do sinal, que pôde ser

modificado para substituir divisões por multiplicações. Os dados obtidos com essa implementação podem ser visto na Figura 15.

Flow Status	Successful - Tue Dec 20 22:37:31 2011
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	SEGMENTED_MIPS
Top-level Entity Name	SEGMENTED_MIPS
Family	Stratix II
Met timing requirements	Yes
Logic utilization	19 %
Combinational ALUTs	1,437 / 12,480 (12 %)
Dedicated logic registers	1,420 / 12,480 (11 %)
Total registers	1420
Total pins	30 / 343 (9 %)
Total virtual pins	0
Total block memory bits	288,768 / 419,328 (69 %)
DSP block 9-bit elements	8 / 96 (8 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)
Device	EP2S15F484C3
Timing Models	Final

Clock Setup: 'CLK'

	Slack	Actual fmax (period)
1	N/A	94.14 MHz (period = 10.623 ns)
2	N/A	94.28 MHz (period = 10.607 ns)
3	N/A	94.45 MHz (period = 10.588 ns)
4	N/A	94.54 MHz (period = 10.577 ns)
5	N/A	94.65 MHz (period = 10.565 ns)
6	N/A	96.04 MHz (period = 10.412 ns)
7	N/A	96.31 MHz (period = 10.383 ns)
8	N/A	96.31 MHz (period = 10.383 ns)
9	N/A	96.44 MHz (period = 10.369 ns)
10	N/A	96.46 MHz (period = 10.367 ns)
11	N/A	96.55 MHz (period = 10.357 ns)
12	N/A	96.60 MHz (period = 10.352 ns)
13	N/A	96.63 MHz (period = 10.349 ns)
14	N/A	96.63 MHz (period = 10.349 ns)
15	N/A	96.67 MHz (period = 10.344 ns)
16	N/A	96.69 MHz (period = 10.342 ns)

FIGURA 15 - Resultados gerados pela ferramenta após terceira etapa de modificações.

Comparando o resultado obtido com a segunda e terceira etapa de implementação pode-se observar o decréscimo de 1318 no número de ALUTs e um aumento de aproximadamente 9 vezes na velocidade de *clock*. Dados das etapas de especulação com e sem otimizações disponibilizadas pela ferramenta – velocidade ou área do circuito – podem ser vistos na Tabela 1.

TABELA 1 – Dados das etapas de especulação

Descrição do circuito	Células Lógicas (ALUTs)	Número de registradores	Velocidade de Clock (MHz)
Primeira etapa especulação - **original**	1040	1352	130,23
Primeira etapa especulação - **velocidade**	1040	1352	130,23
Primeira etapa especulação - **área**	1037	1349	135,12
Segunda etapa especulação - **original**	2755	1376	10,63
Segunda etapa especulação - **velocidade**	2748	1392	10,71
Segunda etapa especulação - **área**	2771	1393	10,24
Terceira etapa especulação - **original**	1437	1420	94,14
Terceira etapa especulação - **velocidade**	1456	1420	90,03
Terceira etapa especulação - **área**	1442	1424	90,79

Analisando a Tabela 1 pode-se notar alguns resultados inesperados, que foram gerados utilizando os aperfeiçoamentos da ferramenta para otimização circuito, onde resultados obtidos com essas otimizações são piores que o obtido sem a mesma. Isto demonstra de forma preliminar a influência da complexidade e qualidade da descrição do circuito no potencial da ferramenta em otimizar o circuito, mesmo a ferramenta utilizando avançadas técnicas de otimizações [41].

4.2. Instruções necessárias para execução do filtro FIR

Para executar o algoritmo do filtro FIR no processador MIPS descrito no trabalho sem o coprocessador para ponto flutuante, teve-se que fazer a emulação do mesmo. Para isso na Seção 3.2 foi abordada essa etapa, incluindo o código *assembly* para conversão do formato de ponto flutuante para ponto fixo. Com isso, cada dado em ponto flutuante que for carregado para um registrador terá que passar por o procedimento de conversão. Esse procedimento acrescenta 25 instruções a serem executadas para cada valor carregado em ponto flutuante, onde 25 é a quantidade de instruções da rotina de emulação.

Com a utilização dos valores já no formato de ponto fixo é necessário apenas executar as instruções para truncagem do resultado da multiplicação para o tamanho de um registrador do processador. Esse processo demanda 5 instruções, tal que tem-se um ganho de 45 instruções quando comparado com a emulação do ponto flutuante. Com a criação da instrução personalizada para multiplicação não há mais necessidade de executar essas 5 instruções extras para conversão de dados, somente as instruções do filtro.

4.3. Resultados obtidos com a execução do filtro

Os resultados gerados com a execução do filtro foram obtidos através da simulação do processador. Para validar o funcionamento do processador e por consequência da execução do filtro foram executados alguns códigos com menor complexidade. A Figura 16 mostra uma dessas execuções juntamente com o código executado na mesma.

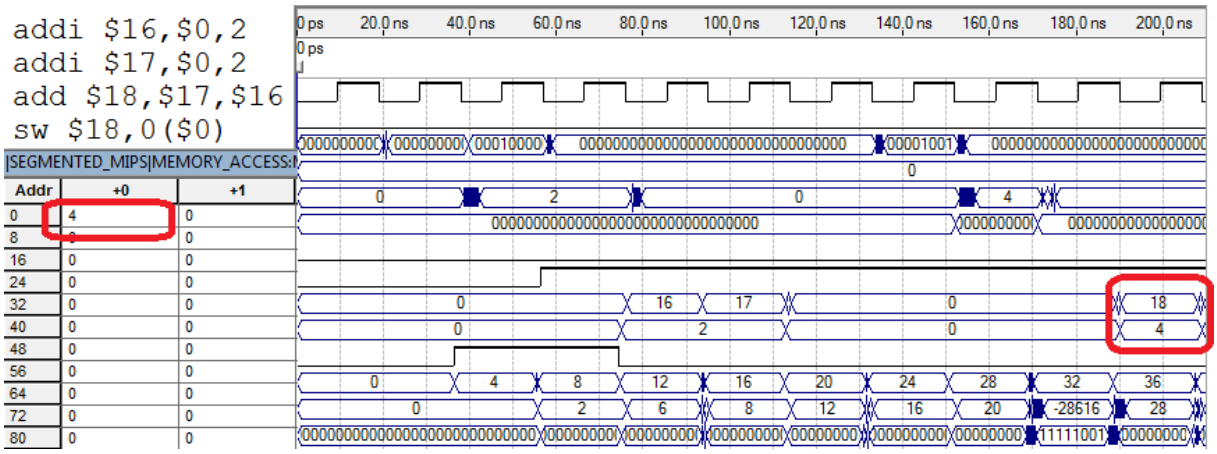

FIGURA 16 – Simulação utilizando a Ferramenta Quartus II.

Com essa etapa realizada partiu-se para a execução do filtro no processador, onde os dados da música foram carregados para memoria do processador juntamente aos coeficientes do filtro. Os resultados obtidos com essa execução foram validados por meio de gráficos comparativos com a execução do filtro utilizando uma linguagem de programação de alto nível, Python. Além desses gráficos, foi gerado outro com os dados da música anterior à filtragem, necessário para demonstrar a efetividade da atuação do filtro. Esse gráfico pode ser visto na Figura 17.

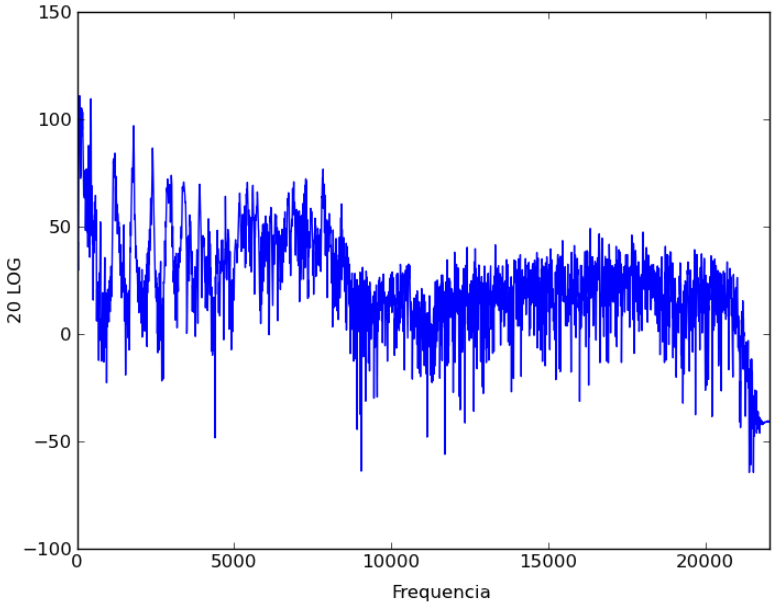

FIGURA 17 – Música antes da filtragem.

De posse do gráfico da Figura 17 que nos mostra os dados antes filtragem podemos analisar os gráficos da atuação do filtro, tanto utilizando a linguagem Python quando utilizando o processador. As Figuras 18 e 19 mostram a filtragem de sinal nas duas situações.

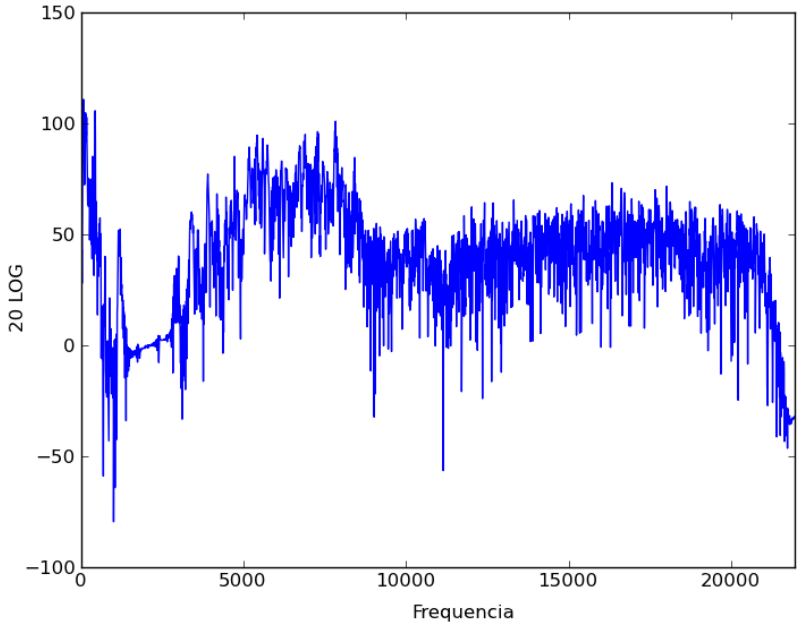

FIGURA 18 – Resultado da filtragem utilizando Python.

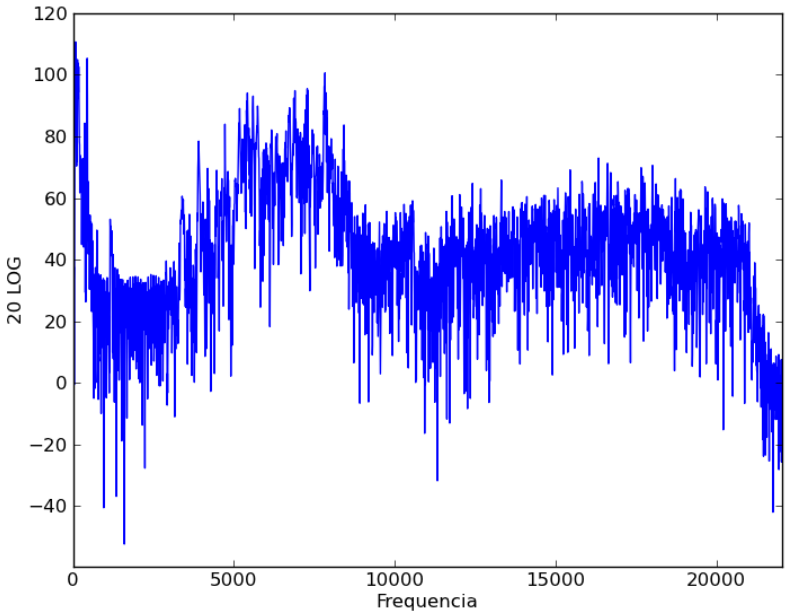

FIGURA 19 – Resultado da filtragem utilizando processador descrito no trabalho.

Analisando as Figuras 18 e 19 que contêm os gráficos de amplitude *x* frequência fica claro a atuação do filtro na filtragem do sinal. A atenuação do sinal começa em

aproximadamente 250 Hz e termina em aproximadamente 4000 Hz como havia sido especificado na descrição do filtro. Comparando os resultados obtidos da filtragem com a linguagem Python e com o processador, nota-se uma igualdade entre os gráficos, mas com um aumento no nível ruído do sinal que foi gerado pela filtragem utilizando o processador. Na Figura 20 pode ser visto os três gráficos plotados em um só deixando assim mais evidente a filtragem e o funcionamento correto do processador.

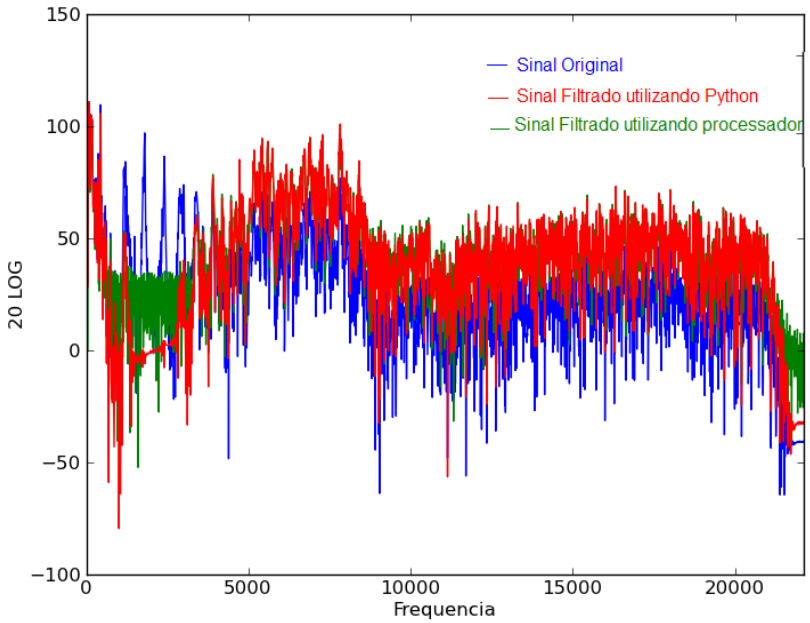

FIGURA 20 – Três gráficos plotados juntos.

4.4. Síntese no FPGA

A síntese no FPGA foi feita utilizando a placa DE2 da Altera, que contém o FPGA da família Ciclone II. Como anteriormente o código VHDL do processador tinha sido compilado para outro modelo de FPGA, foi necessário executar uma nova etapa de compilação e levantamento de resultados. O resultado dessa compilação para esse modelo especifico de FPGA pode ser visto na Figura 21.

Flow Status	Successful - Wed Dec 21 21:24:21 2011
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	SEGMENTED_MIPS
Top-level Entity Name	SEGMENTED_MIPS
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	2,557 / 33,216 (8 %)
Total combinational functions	2,450 / 33,216 (7 %)
Dedicated logic registers	1,419 / 33,216 (4 %)
Total registers	1419
Total pins	30 / 475 (6 %)
Total virtual pins	0
Total memory bits	288,768 / 483,840 (60 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)

Clock Setup: 'CLK'

	Slack	Actual fmax (period)
1	N/A	53.99 MHz (period = 18.522 ns)
2	N/A	54.00 MHz (period = 18.517 ns)
3	N/A	54.12 MHz (period = 18.477 ns)
4	N/A	54.14 MHz (period = 18.472 ns)
5	N/A	54.38 MHz (period = 18.390 ns)
6	N/A	54.51 MHz (period = 18.345 ns)
7	N/A	54.57 MHz (period = 18.324 ns)
8	N/A	54.62 MHz (period = 18.307 ns)
9	N/A	54.64 MHz (period = 18.302 ns)
10	N/A	54.71 MHz (period = 18.279 ns)
11	N/A	55.02 MHz (period = 18.175 ns)
12	N/A	55.22 MHz (period = 18.109 ns)
13	N/A	55.46 MHz (period = 18.032 ns)
14	N/A	55.55 MHz (period = 18.003 ns)
15	N/A	55.60 MHz (period = 17.987 ns)
16	N/A	55.69 MHz (period = 17.958 ns)

FIGURA 21 - Resultados gerados pela ferramenta após compilação para FPGA Ciclone II.

Analisando a Figura 21, nota-se que a velocidade de *clock* do circuito ficou próxima da velocidade máxima disponibilizada pela placa DE2, que tem um gerador interno de *clock* de 50 MHz. Sendo a velocidade do circuito sendo compatível com a velocidade da placa, não há necessidade de instanciar módulos para redução de velocidade de *clock*, simplesmente pode-se assinalar o pino gerador de *clock* da placa na entrada do circuito.

Para validação da síntese do processador com o filtro no FPGA foi estudada a alternativa de utilização do kit de áudio disponível no mesmo. A possível integração dos blocos desse kit com o processador do trabalho pode ser vista na Figura 22.

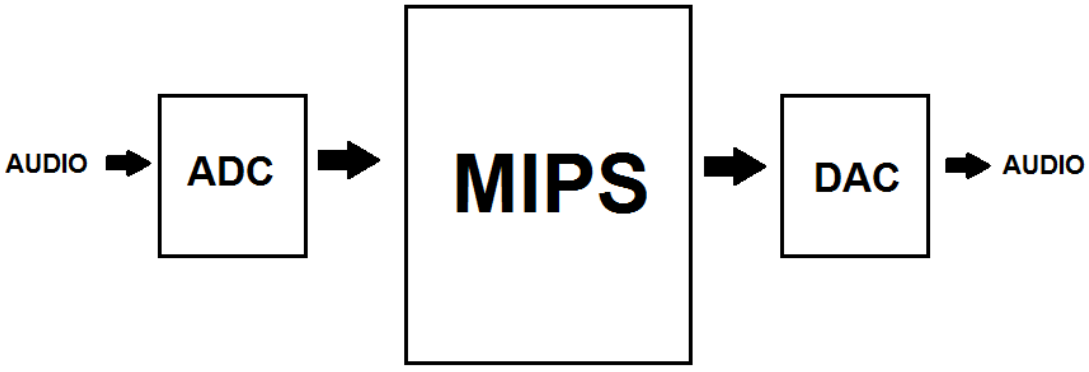

FIGURA 22 – Diagrama de blocos utilizando kit de áudio e processador.

O estudo feito para viabilização desse experimento foi através dos exemplos disponibilizados pela fabricante do FPGA. Um desses exemplos é a configuração desse

dispositivo para ter o comportamento de maquina de karaokê, tendo como entrada um sinal de áudio no formato analógico e um sinal emitido por um microfone. Na teoria seria possível utilizar esse exemplo com algumas modificações para a aplicação desse trabalho, mas essa hipótese se mostrou inviável, devido ao fato desse exemplo e todos os outros similares disponibilizados pelo fabricante do FPGA estarem em uma linguagem de descrição de hardware diferente da empregada no trabalho. Uma alternativa que poderia ser utilizada para contornar essa situação seria a de buscar uma conversão entre essas duas linguagens, mas essa abordagem não se mostrou atraente, pois não há uma conversão literal entre elas. Buscou-se então uma ferramenta que fizesse esse processo automatizado, mas não se teve êxito nessa tarefa. Como segunda alternativa cogitou-se a possibilidade de desenvolver os componentes necessários na linguagem VHDL para utilização do kit de áudio, mas logo essa abordagem foi deixada de lado, devido à alta complexidade envolvida nesse processo e ao tempo hábil para execução do trabalho.

Com a inviabilização dessa primeira abordagem foi buscada outra alternativa para validação da síntese do processador com o filtro no FPGA, que se caracterizou pela obtenção de resultados através do acendimentos de alguns LEDs e verificação de alguns resultados através dos displays da placa. Para esse processo foram feitas sínteses com algoritmos mais simples rodando no processador. A Figura 23 mostra o resultado de uma dessas sínteses no FPGA.

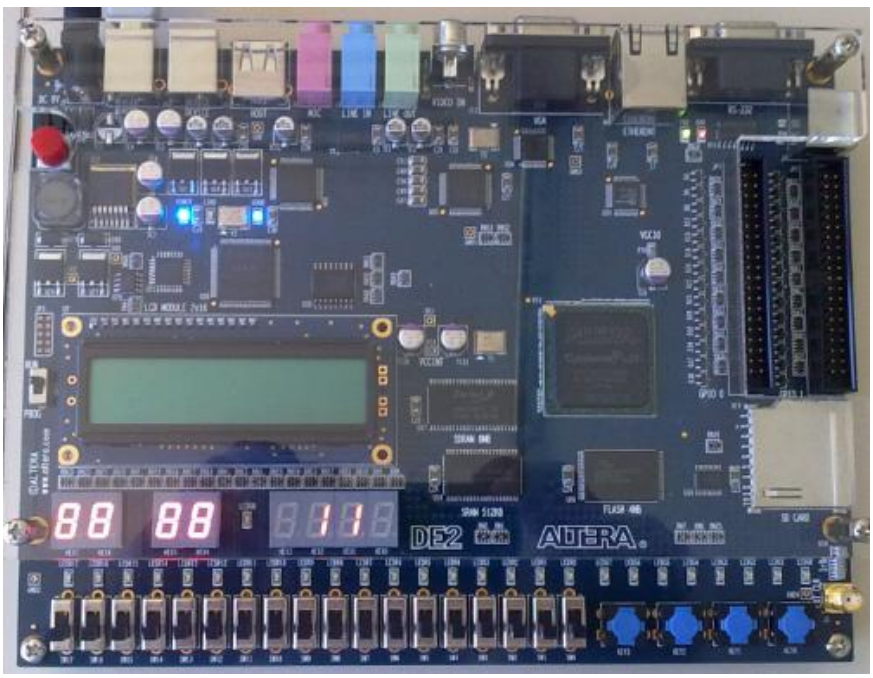

FIGURA 23 – Síntese para validação.

Para melhor entendimento do resultado dessa síntese foi aproximada a imagem na parte relevante: os quatro displays de sete segmentos que podem ser visto na Figura 24 circulado em vermelho.

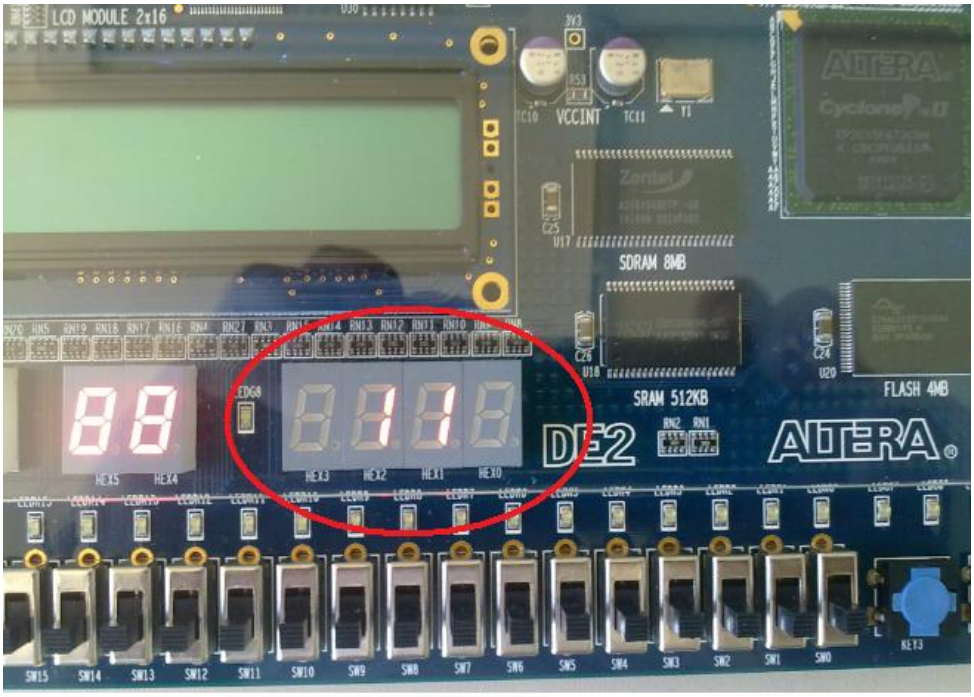

FIGURA 24 – Síntese para validação Zoom.

Esse exemplo nos mostra o funcionamento do processador para o código de multiplicação de dois valores em ponto fixo, onde está sendo multiplicado o valor 0,75 pelo valor 2 é apresentado nos quatro displays as quatro posições intermediárias do registrador que está recebendo o valor dessa multiplicação, a qual é realizada utilizando a rotina que foi explicada na seção 3.4.

Para validação da síntese do processador juntamente ao filtro, foram adicionadas algumas instruções no algoritmo do mesmo. Estas instruções adicionadas têm como objetivo contabilizar o número de execuções do filtro e setar o valor 1 em um dos registradores do processador após um determinado número de execuções. Esse registrador que está recebendo o valor 1 quando o programa termina a execução foi assinalado a um dos display de sete segmentos e a um LED verde da placa. A Figura 25 mostra o comportamento da placa após síntese do processador rodando o filtro.

FIGURA 25 – Resultado da síntese do processador.

4.5. Discussão dos resultados

Com base nos resultados explicitados na Seção 4.1 podem-se fazer uma comparação com a arquitetura MIPS completa com coprocessador de ponto flutuante. Para essa comparação vamos utilizar os dados gerados pelo trabalho [10] Figura 26 que apresentam os dados da arquitetura plasma que é uma arquitetura compatível com a arquitetura MIPS integrado com um coprocessador de ponto flutuante. Os dados expostos pelo trabalho [10] foram obtidos através do dispositivo FPGA Xilinx Virtex-4 [36].

Organização	XST			
	XC4VFX100-10		XC5VLX50T-1	
	Freq. (MHz)	Área (LUTs)	Freq. (MHz)	Área (LUTs)
Plasma	65,97	3369 (3%)	78,05	2222 (7%)
Plasma-HFP100	15,31	11489 (13%)	24,34	8320 (28%)
Plasma-HFPmin	5,85	7169 (8%)	8,64	5453 (18%)
Plasma-HFPmax	60,46	7516 (8%)	77,33	5504 (19%)
Plasma-HFP-GALS	65,49	7869 (9%)	79,09	5647 (19%)
Plasma-HFP-GALS-LP	66,08	7675 (9%)	79.60	5534 (19%)

FIGURA 26 - Resultados do trabalho [10].

Segundo uma discussão da Altera [31], de forma geral, cada ALUT, unidade lógica do FPGA Stratix II, é equivalente a 1,3 LUT, unidade lógica do FPGA Virtex-4. Essa informação sugere que o projeto do processador Plasma com coprocessador de unidade de ponto flutuante gastaria 11489 LUT, o que é correspondente a aproximadamente 8838 ALUTs. Comparando esse valor com os resultados finais abordados na Seção 4.1, nota-se que a especulação do subconjunto do processador MIPS sem unidade de ponto flutuante discutida nesse trabalhado consome aproximadamente um sexto de componentes de hardware quando comparado a uma implementação completa do processador MIPS com unidade de ponto flutuante.

Embora a análise de desempenho demande uma comparação incluindo, além da velocidade do *clock*, o número de instruções e o CPI [13], convém ressaltar que a velocidade de *clock* da implementação discutida nesse trabalho é de aproximadamente seis vezes superior que a de uma das implementações completa do processador MIPS com coprocessador de ponto flutuante discutida em [10]. Além disso, cabe destacar que o CPI da arquitetura implementada e o da versão MIPS com unidade de ponto flutuante discutida em [10] não pode ser considerado o mesmo, devido que as operações de ponto flutuante têm latência diferente.

Além disso, convém destacar que, com a inclusão da nova instrução para multiplicação, foi obtida uma redução de aproximadamente 26% no número de ciclos executados pelo processador, quando levado em conta a execução do programa com a rotina (macro) para execução da multiplicação.

5. CONSIDERAÇÕES FINAIS

Conforme os objetivos gerais e específicos estabelecidos, neste trabalho foi realizada a implementação de uma arquitetura de propósito específico partindo de uma arquitetura de propósito geral visando a melhoria na execução de uma aplicação particular. A arquitetura alvo do trabalho foi a MIPS e como aplicação foi escolhido um filtro DSP, o filtro FIR, que tem como um de seus requisitos a utilização de valores do conjunto dos números reais. Ao invés de realizar uma implementação completa, partiu-se de uma implementação reduzida do processador MIPS, correspondente a um subconjunto minimalista da arquitetura, onde foram incluídas instruções complementares necessárias para a execução da aplicação.

Outro ponto importante a salientar sobre essa implementação é a ausência do coprocessador de ponto flutuante, o que, em uma primeira proposta de solução, implicou na necessidade de emular as instruções que utilizavam esse formato. Em um segundo momento, partiu-se para uma segunda solução, caracterizada pelo carregamento de dados diretamente em ponto fixo, assim eliminando a emulação do ponto flutuante. A partir do número de instruções necessárias para a emulação, mais as instruções para execução do filtro, foi possível calcular a quantidade total de instruções que, associada ao valor referente à velocidade de *clock*, foi possível quantificar o desempenho da solução. Com as modificações no formato de dados e personalização da instrução de multiplicação conseguiu-se diminuir o número de instruções e elevar a velocidade de *clock*, melhorando significativamente o desempenho.

Na segunda etapa da especulação da arquitetura foi feito um estudo mais detalhado da aplicação para otimização da mesma, resultando na substituição do ponto flutuante por ponto fixo nas operações do processador. Para isso foram criados softwares em JAVA que carregam os dados para memória do processador já no formato de ponto fixo. Outro ponto abordado foi o da customização de instruções através da análise do código da aplicação, onde foi transformada a pseudoinstrução MUL em instrução física do processador, desta forma diminuindo o número de instruções de máquina necessárias para execução do código do filtro FIR. Realizando estas etapas, cumpriu-se todas as etapas que foram planejadas no cronograma de atividades exposto na Figura 27.

Atividade	Semestre 1				Semestre 2			
	1	2	3	4	1	2	3	4
Pesquisa de Trabalhos Semelhantes	●	●						
Escrita TCC1	●	●	●	●				
Escrita TCC2					●	●	●	●
Estudo do MIPS	●	●						
Estudo Preliminar do Algoritmo Filtro FIR		●						
Estudo e Adaptação de código VHDL		●	●	●				
Experimentos com a Arquitetura Nativa					●			
Estudo Detalhado do Algoritmo Filtro FIR					●			
Implementação de Código C e Assebly para Filtro FIR					●			
Experimentos com a Arquitetura Nativa						●		
Especular a Arquitetura						●	●	
Sintetização para Dispositivo FPGA							●	
Compilar Resultados							●	●
Apresentação 1				●				
Apresentação 2								●

FIGURA 27 – Cronograma de atividade TCC.

6. BIBLIOGRÁFIA

[1] BARBOSA, E. R. Filtros Digitais Reconfiguráveis. Monografia (Graduação em Engenharia da Computação) - Núcleo Ciências Exatas e Tecnológicas, UnicenP: URITIBA, 2006.

[2] ROSA, V. S. da. Uma Ferramenta para a Geração Otimizada de Filtros FIR paralelos com Coeficientes Constantes. [S.l.: s.n.], 2005.

[3] FORIN, R. N. P. L. L. A. eMIPS, A Dynamically Extensible Processor. Relatório Técnico MSR-TR-2006-143: [s.n.], 2006.

[4] JÚNIOR, J. M. de A. Projeto de um sistema de desvio de obstáculos para robôs móveis baseado em computação reconfigurável. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, USP: SÃO CARLOS, 2009.

[5] WEBER, T. S. Arquiteturas Tolerantes a Falhas. Programa de Pós-Graduação em Computação, PPGC – Instituto de Informática, UFRGS: PORTO ALEGRE.

[6] MENDONÇA, F. da S. Tolerância a Falhas em Sistemas Distribuídos: A Abordagem do CORBA/FT. Técnico em Processamento de Dados - Centro de Ciências Formais e Tecnológicas: ARACAJU, 2002.

[7] PRADHAN, D. K. Fault-tolerant computer system design. Prentice Hall; 1st edition: (February 14, 1996), 1996.

[8]KAT|, D., GENTILE, R; LUKASIAK, T. Floating-Point Emulation on Fixed-Point DSPs is Becoming Practical, (2003), at http://www.reed-electronics.com/ecnmag/index.asp?layout=article&articleid=CA267186

[9] GUSSO, K ; SAOTOME, O. Implementação de Funções Matemáticas de Ponto-Flutuante de Alto Desempenho em uma Plataforma DSP Ponto-Fixo. XXV Congresso da Sociedade Brasileira de Computação – UNISINOS, São Leopoldo, 2005.

[10] T. RODOLFO, Uma Exploração do Espaço de Projeto de Processadores com Hardware de Ponto Flutuante em FPGAS. Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul. Porto Alegre, 2010.

[11] E. SCHEMBERGER, K. ARAUJO, S. ANDRADE; EKSMIPS: Um Simulador para o Processador MIPS. Departamento de Informática – Universidade Estadual de Maringá (UEM). Maringá – PR – Brasil.

[12] MIPS Technologies. MIPS32® Architecture For Programmers. Vol. II: The MIPS32® Instruction Set, 2008.

[13] PATTERSON, D.; HENNESSY, J. Organização e Projeto de Computadores – A Interface Hardware/Software, 3ª ed., Campus, 2005.

[14] FERNANDES, G, R; FONSECA, D, S; Arquitetura FPU: Implementação no MIPS32; Centro Tecnológico Ciências da Computação; Universidade Federal de Santa Catarina.

[15] STALLINGS, W. Computer Organization and Architecture: designing for performance. 4. ed., Prentice-Hall, 1996.

[16] SWEETMAN, D. See MIPS Run, 2ª ed., Morgan Kaufmann, 2006.

[17] TANEMBAUM, A. Organização Estruturada de Computadores. 5ª ed., Prentice-Hall, 2007.

[18] VOLLMAR, K.; SANDERSON, P. A MIPS Assembly Language Simulator Designed for Education. In: Journal of Circuits, Systems and Computers (JCSC), 2005. pp. 95-101.

[19] IEEE, “IEEE Standard for Binary Floating Point Arithmetic: ANSI/IEEE Std 754-1985”, Institute of Electrical and Electronics Engineers, 1985.

[20] DOWD, K.; SEVERANCE, C., “High Performance Computing: RISC Architectures, Optimization & Benchmarks”, O’Reilly, 1998.

[21] PILLA, V, J, VHDL: Uma Introdução; Notas de aula para a Disciplina de Sistemas Digitais Integrados do Curso de Engenharia da Computação. Centro Universitário Positivo, Curitiba 2003.

[22] NUNES, A, C; Algoritmo de Criptografia AES em Hardware, Utilizando Dispositivo de Lógica Programável (FPGA) e Linguagem de Descrição de Hardware (VHDL); Programa de Pós-Graduação em Engenharia Elétrica; ITAJUBÁ/MG – 2008.

[23] WAIN R.; BUSH I.; GUEST M.; DEEGAN M.; KOZIN I.; KITCHEN C.; An overview of FPGAs and FPGA programming. 2006.

[24] MORELOS-ZARAGOZA R. H., The Art of Error Correcting Coding. John Wiley & Sons, Ltd. 2006.

[25] BROWN S.; ROSE J.; Architecture of FPGAs and CPLDs: A Tutorial. University of Toronto. 1996.

[26] EMMANUEL LUJAN". "mips segmentado". "2008".

[27] VILLA, L.,C.,R; ZUFFO, M., K. ; Técnicas de Emulação de Operações de Ponto Flutuante em Sistemas Operacionais Modernos; XXVII Congresso SBC; Rio de Janeiro, 2007.

[28] NEVES, A., S.; Estudo de algoritmos heurísticos para filtros digitais adaptativos.; Programa de Pós-Graduação em Informática; Universidade Católica de Pelotas; Pelotas. 2007.

[29] LYONS, Richard G. Understanding Digital Signal Processing. 2ª Ed. Prentice Hall PTR. Upper Saddle River, New Jersey, USA. Março de 2004.

[30] ARES, T., R.; Uma Exploração do Espaço de Projeto de Processadores com Hardware de Ponto Flutuante em FPGAS.; Programa de Pós-Graduação em Ciência da Computação; UFRGS ; Porto Alegre, 2010.

[31] Altera; Stratix II vs. Virtex-4 Density Comparison; White Paper; At ” http://www.altera.com/literature/wp/wpstxiixlnx.pdf” ; Agosto, 2005

[32] GHDL Where VHDL meets gcc: Site. 2011. Disponível em: <http://ghdl.free.fr/>. Acesso em: 08 Apr. 2011.

[33] ALTERA QUARTUS II: Site. 2011. Disponível em: <http://www.altera.com/products/software/quartusii/subscription-edition/qts-se-index.html>. Acesso em: 08 Apr. 2011.

[34] UNIPAMPA ENGENHARIA DE COMPUTAÇÃO: Site. 2011. Disponível em: < http://cursos.unipampa.edu.br/cursos/engenhariadecomputacao/> Acesso em: 08 Apr. 2011.

[35] MEMORY INITIALIZATION FILE (.MIF): Site. 2011. Disponível em: < http://quartushelp.altera.com/9.1/mergedProjects/reference/glossary/def_mif.htm> Acesso em: 08 Apr. 2011.

[36] XILINX VIRTEX-4: Site. 2011. Disponível em: < http://www.xilinx.com/support/documentation/virtex-4.htm > Acesso em: 08 Apr. 2011.

[37] OPENCORES - PIPELINE MIPS IN VHDL : Site. 2011. Disponível em: Disponível em: <http://opencores.org/project,vhdl-pipeline-mips> Acesso em: 08 Apr. 2011.

[38] VERILOG TUTORIAL: Site. 2011. Disponível em: < http://www.asic-world.com/verilog/veritut.html> Acesso em: 08 Apr. 2011.

[39] ALMEIDA, F. , G. , N. ; Análise de Filtros Digitais Implementados em Aritmética de Ponto Fixo usando Cadeias de Markov. ; Programa de Pós-Graduação em Engenharia Elétrica; Escola Politécnica da USP ; São Paulo, 2011.

[40] ANDRADE, A. O. ; SOARES, A. B; Técnicas de Janelamento de Sinais; Universidade Federal de Uberlândia - Faculdade de Engenharia Elétrica.

[41] ALTERA; Section III. Area Optimization & Timming Closure; Disponivel em: < http://www.altera.ru/Disks/Altera%20Documentation%20Library/literature/hb/qts/qts_qii5v2_03.pdf>

APENDICE A – Software JAVA

SOFTWARE GBIN – Converte de inteiro para binário na notação complemento de dois e ponto fixo

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.StringTokenizer;


public class GBin {

     private String errors;
     private int currentLine;

     // Constructor
     public GBin() {
          this.errors = "";
          this.currentLine = 1;
     }

     // Functions related with the errors

     private void addError(String newError) {
          this.errors += newError;
     }

     private String getErrors() {
          return this.errors;
     }

     // Function related with the current compiling line

     public int getCurrentLine() {
          return currentLine;
     }

     public void setCurrentLine(int currentLine) {
          this.currentLine = currentLine;
```

```
}

public void incLine() {
     this.currentLine++;
}


private String invert(String r) {
     r = r.replace('0', 'X');
     r = r.replace('1', '0');
     r = r.replace('X', '1');
     return r;
}

private String negativeOffset(String r) {
     double aux1 = Double.parseDouble(r);
     int aux = (int)aux1;
     if(aux1-aux == 0){
          aux = aux - 1;

     }
     r = Integer.toBinaryString(aux);
     int n = 16 - r.length();
     for (int i = 0; i < n; i++)
          r = "0" + r;
     r = this.invert(r);
     return r;
}

private String positiveOffset(String r) {
     double aux1 = Double.parseDouble(r);
     int aux = (int)aux1;

     r = Integer.toBinaryString(aux);
     int n = 16 - r.length();
     for (int i = 0; i < n; i++)
          r = "0" + r;
     return r;
}

private String toBin(int x) {
     String b = Integer.toBinaryString(x);
     int n = 32 - b.length();
     for (int i = 0; i < n; i++)
          b = "0" + b;
     return b;
}

private String getOffset(String r) {
     boolean isNeg = false;
```

```
        if (Double.parseDouble(r) < 0) {
             r = r.replaceFirst("\\-", "");
             isNeg = true;
        }

        boolean charsCorrects = true;
        r = r.replaceAll("[^0-9] \\.", "");
        boolean isANumber = (r != "");

        if (charsCorrects && isANumber) {
             if (isNeg)
                  return this.negativeOffset(r);
             else
                  return this.positiveOffset(r);
        } else {
             return " Invalid_offset  ";
        }

}

private String getBin(String r){



        boolean isNeg=false;
        double d0=Double.parseDouble(r);

        if (d0 < 0){
             isNeg = true;
        }

        int i0 = (int)d0;


        String s="";
        String s1=String.valueOf(d0);
        String s2;

        s2= this.getOffset(s1);
        double d =d0-i0;
        if(d<0){
             d=d*-1;
        }
        int i=0;
        int t=0;
        int f;
        while((d!=0) && (t!=16) ){
             d=d*2;
             i= (int)d;
             if(i==1){
```

```
                    s=s+"1";
                    d=d-1;
               }
               else{
                    s=s+"0";

               }
               t++;
          }

          if ((isNeg) && (s.length()!=0 )){
               int t2 =s.length();
               s = this.invert(s);
               f = Integer.parseInt(s, 2) +1;
               String s3;
               s3 = Integer.toBinaryString(f);
               if(s3.length() < 16){

                    while(s3.length() != s.length()){
                         s3 = "0"+s3;

                    }
                    s=s3;
               }
          }
          f=0;
          for(f=t;f < 16;f++){

               s=s+"0";
          }
          return (s2+s);
     }



     public void translate(String filePath) {
          File assemblerFile = new File(filePath);

          String binCode = "";
          String romCode = "case READ_ADDR is\n";
          try {
               BufferedReader input = new BufferedReader(new
FileReader(
                              assemblerFile));
               try {
                    String line = null;
                    String binLine;
                    int i = 0;

                    this.setCurrentLine(1);
```

```
                    while ((line = input.readLine()) != null) {


                         binLine =
this.getBin(line);//this.generateCode(line);
                         binCode += binLine + "\n";


                         i = this.getCurrentLine();
                         this.incLine();
                    }

               } finally {
                    input.close();
               }
          } catch (IOException ex) {
               ex.printStackTrace();
          }

          File binFile = new File("code.bin");
          try {
               if (!binFile.exists()) {
                    binFile.createNewFile();
               }
               if (!binFile.isFile()) {
                    throw new IllegalArgumentException("Is not
a file: " + binFile);
               }
               if (!binFile.canWrite()) {
                    throw new IllegalArgumentException(
                              "The file cannot be written: " +
binFile);
               }

               Writer binOutput = new BufferedWriter(new
FileWriter(binFile));

               try {
                    binOutput.write(binCode);
                    System.out.print(this.getErrors());
               } finally {
                    binOutput.close();
               }

          } catch (IOException ex) {
               ex.printStackTrace();
          }

     }
```

```
	// Main function

	public static void main(String[] args) {
		GBin smc = new GBin();
		String file = args[0];
		smc.translate(file);
	}


}
```

APENDICE B – Software JAVA

SOFTWARE GInt – Converte um de binário na notação complemento de dois e ponto fixo para inteiro.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;


public class GInt {

     private String errors;
     private int currentLine;

     public GInt() {
          this.errors = "";
          this.currentLine = 1;
     }

     private void addError(String newError) {
          this.errors += newError;
     }

     private String getErrors() {
          return this.errors;
     }

     public int getCurrentLine() {
          return currentLine;
     }

     public void setCurrentLine(int currentLine) {
          this.currentLine = currentLine;
     }

     public void incLine() {
          this.currentLine++;
     }

     private String invert(String r) {
```

```
        r = r.replace('0', 'X');
        r = r.replace('1', '0');
        r = r.replace('X', '1');
        return r;
    }

    private String negativeOffset(String r) {
        double aux1 = Double.parseDouble(r);
        int aux = (int)aux1;
        if(aux1-aux == 0){
            aux = aux - 1;

        }

        r = Integer.toBinaryString(aux);
        int n = 16 - r.length();
        for (int i = 0; i < n; i++)
            r = "0" + r;
        r = this.invert(r);
        return r;
    }

    private String positiveOffset(String r) {
        double aux1 = Double.parseDouble(r);
        int aux = (int)aux1;

        r = Integer.toBinaryString(aux);
        int n = 16 - r.length();
        for (int i = 0; i < n; i++)
            r = "0" + r;
        return r;
    }


    private String toBin(int x) {
        String b = Integer.toBinaryString(x);
        int n = 32 - b.length();
        for (int i = 0; i < n; i++)
            b = "0" + b;
        return b;
    }

    private String getOffset(String r) {

        boolean isNeg = false;
        if (Double.parseDouble(r) < 0) {
            r = r.replaceFirst("\\-", "");
            isNeg = true;
        }
```

```
     boolean charsCorrects = true;

     r = r.replaceAll("[^0-9] \\.", "");

     boolean isANumber = (r != "");

     if (charsCorrects && isANumber) {
          if (isNeg)
               return this.negativeOffset(r);
          else
               return this.positiveOffset(r);
     } else {
          return " Invalid_offset  ";
     }

}



private String SomaBin1(String r) {
     char val[] = r.toCharArray();
     int cont=r.length() ;
     String r1="";
     while(cont != -1){
          cont=cont-1;

          if(val[cont]=='1'){
               val[cont] = '0';

          }
          else{
               val[cont]='1';
               cont=-1;
          }


     }

     for(cont=0;cont<r.length();cont++){
          if(val[cont]=='0'){
               r1=r1+"0";
          }else{
               r1=r1+"1";
          }
     }
     return r1;
}

private String getBin(String r){
     boolean isNeg=false;
```

```
double d0=Double.parseDouble(r);

if (d0 < 0){

     isNeg = true;
}

int i0 = (int)d0;


String s="";
String s1=String.valueOf(d0);
String s2;

s2= this.getOffset(s1);
double d =d0-i0;
if(d<0){
     d=d*-1;
}
int i=0;
int t=0;
int f;
while((d!=0) && (t!=16) ){
     d=d*2;
     i= (int)d;
     if(i==1){
          s=s+"1";
          d=d-1;
     }
     else{
          s=s+"0";

     }
     t++;
}



if ((isNeg) && (s.length()!=0 )){
     int t2 =s.length();
     s = this.invert(s);
     f = Integer.parseInt(s, 2) +1;
     String s3;
     s3 = Integer.toBinaryString(f);
     if(s3.length() < 16){

          while(s3.length() != s.length()){
               s3 = "0"+s3;
          }
          s=s3;
```

```
                }
            }
            f=0;
            for(f=t;f < 16;f++){

                s=s+"0";
            }



            return (s2+s);

        }

        private String getInt(String r){

            boolean Neg = false;

            int cont;
            char c[]=r.toCharArray();
            char
c1[]={'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0',
'0','0'};
            char
c2[]={'0','0','0','0','0','0','0','0','0','0','0','0','0','0','0',
'0','0'};

            if(c[0] == '1'){
                Neg = true;
            }

            double valor = 0.0;
            double valor1 = 0.0;
            double valor2 = 0.0;


            for (cont = 0; cont < 16; cont++){
                c1[cont]=c[cont];
            }

            for (cont = 16; cont <32; cont++){
                c2[cont - 16] = c[cont];
            }


            int m=0;
            int Ineg=0;
            int pos1=-1;
            String Sneg="";
            if (Neg){
```

```
                for (cont = 0; cont < 16; cont++){
                     if (c2[cont] == '1'){
                          pos1 = cont;
                     }

                }

                if (pos1 != -1){

                     for (cont = 0; cont <= pos1; cont++){
                          Sneg=Sneg+c2[cont];

                     }
                     Sneg = this.invert(Sneg);
                     Sneg = this.SomaBin1(Sneg);


                     for (cont=Sneg.length();cont<16;cont++){

                          Sneg=Sneg+"0";
                     }

                }

                char c3[]=Sneg.toCharArray();
                m=2;
                for (cont = 0; cont < 16; cont++){
                     if (c3[cont] == '1'){
                          valor2 = valor2 + 1.0/m;
                     }
                     m = m*2;

                }

                if(valor2 == 0.0){
                     valor1 =
(int)(Integer.parseInt(this.invert(new String(c1)), 2) +1) ;

                }
                else{

                     valor1 =
(int)(Integer.parseInt(this.invert(new String(c1)), 2) ) ;
                }

                valor = (valor1+ valor2)*-1.0;

           }
```

```
          else{
               valor1 = (int)Integer.parseInt(new String(c1),
2);
               m=2;
               for (cont = 0; cont < 16; cont++){
                    if (c2[cont] == '1'){
                         valor2 = valor2 + 1.0/m;
                    }
                    m = m*2;

               }
               valor = valor1+ valor2;

          }

          r = String.valueOf(valor);

          return r;
     }

     public void translate(String filePath) {
          File assemblerFile = new File(filePath);

          String binCode = "";
          String romCode = "case READ_ADDR is\n";
          try {
               BufferedReader input = new BufferedReader(new
FileReader(
                              assemblerFile));
               try {
                    String line = null;
                    String binLine;
                    int i = 0;
                    this.setCurrentLine(1);
                    while ((line = input.readLine()) != null) {
                         String[] result = line.split("\\s");
                         for (int x=0; x<result.length; x++){

                              binLine =
this.getInt(result[x]);//this.generateCode(line);
                              binCode += binLine + "\n";


                         }


                         i = this.getCurrentLine();
                         this.incLine();
                    }
```

```
                } finally {
                    input.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }

            File binFile = new File("codeint.int");
            try {
                if (!binFile.exists()) {
                    binFile.createNewFile();
                }
                if (!binFile.isFile()) {
                    throw new IllegalArgumentException("Is not
a file: " + binFile);
                }
                if (!binFile.canWrite()) {
                    throw new IllegalArgumentException(
                            "The file cannot be written: " +
binFile);
                }

                Writer binOutput = new BufferedWriter(new
FileWriter(binFile));

                try {
                    binOutput.write(binCode);
                    System.out.print(this.getErrors());
                } finally {
                    binOutput.close();
                }

            } catch (IOException ex) {
                ex.printStackTrace();
            }

        }

        public static void main(String[] args) {
            GInt smc = new GInt();
            String file = args[0];
            smc.translate(file);
        }

}
```

APENDICE C – Software JAVA

SOFTWARE GMIF – Converte de binário para formato compatível usado para inicialização das memorias da altera.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;


public class GMif {

     private String errors;
     private int currentLine;

     public GMif() {
          this.errors = "";
          this.currentLine = 1;
     }

     private void addError(String newError) {
          this.errors += newError;
     }

     private String getErrors() {
          return this.errors;
     }

     public int getCurrentLine() {
          return currentLine;
     }

     public void setCurrentLine(int currentLine) {
          this.currentLine = currentLine;
     }

     public void incLine() {
          this.currentLine++;
     }

     private String invert(String r) {
```

```
        r = r.replace('0', 'X');
        r = r.replace('1', '0');
        r = r.replace('X', '1');
        return r;
    }

    private String negativeOffset(String r) {
        double aux1 = Double.parseDouble(r);
        int aux = (int)aux1;
        if(aux1-aux == 0){
            aux = aux - 1;

        }

        r = Integer.toBinaryString(aux);
        int n = 16 - r.length();
        for (int i = 0; i < n; i++)
            r = "0" + r;
        r = this.invert(r);
        return r;
    }

    private String positiveOffset(String r) {
        double aux1 = Double.parseDouble(r);
        int aux = (int)aux1;

        r = Integer.toBinaryString(aux);
        int n = 16 - r.length();
        for (int i = 0; i < n; i++)
            r = "0" + r;
        return r;
    }




    private String toBin(int x) {
        String b = Integer.toBinaryString(x);
        int n = 32 - b.length();
        for (int i = 0; i < n; i++)
            b = "0" + b;
        return b;
    }

    private String getOffset(String r) {

        boolean isNeg = false;
        if (Double.parseDouble(r) < 0) {
            r = r.replaceFirst("\\-", "");
            isNeg = true;
```

```
          }

          boolean charsCorrects = true;//r.matches("[0-9\\
\\t.]*");

          r = r.replaceAll("[^0-9] \\.", "");

          boolean isANumber = (r != "");

          if (charsCorrects && isANumber) {
               if (isNeg)
                    return this.negativeOffset(r);
               else
                    return this.positiveOffset(r);
          } else {
               return " Invalid_offset  ";
          }

     }

     private String getBin(String r){



          boolean isNeg=false;
          double d0=Double.parseDouble(r);


          if (d0 < 0){

               isNeg = true;
          }

          int i0 = (int)d0;


          String s="";
          String s1=String.valueOf(d0);
          String s2;

          s2= this.getOffset(s1);
          double d =d0-i0;
          if(d<0){
               d=d*-1;
          }
          int i=0;
          int t=0;
          int f;
          while((d!=0) && (t!=16) ){
               d=d*2;
```

```
                i= (int)d;
                if(i==1){
                     s=s+"1";
                     d=d-1;
                }
                else{
                     s=s+"0";

                }
                t++;
           }

           if ((isNeg) && (s.length()!=0 )){
                int t2 =s.length();
                s = this.invert(s);
                f = Integer.parseInt(s, 2) +1;
                String s3;
                s3 = Integer.toBinaryString(f);

                if(s3.length() < 16){

                     while(s3.length() != s.length()){
                          s3 = "0"+s3;
                     }
                     s=s3;
                }
           }
           f=0;
           for(f=t;f < 16;f++){

                s=s+"0";
           }



           return (s2+s);

      }


      public void translate(String filePath) {
           File assemblerFile = new File(filePath);
           String binCode = "";
           String romCode = "case READ_ADDR is\n";
           try {
                BufferedReader input = new BufferedReader(new
FileReader(
                                  assemblerFile));
                try {
                     String line = null;
```

```
                    String binLine;
                    int i = 0;
                    int i2 = 400;
                    this.setCurrentLine(1);
                    while ((line = input.readLine()) != null) {
                         binLine = i2+" : "+line+";";
                         binCode += binLine + "\n";

                         i2++;
                         System.out.println(i);
                         i = this.getCurrentLine();
                         this.incLine();
                    }

               } finally {
                    input.close();
               }
          } catch (IOException ex) {
               ex.printStackTrace();
          }

          File binFile = new File("code.mif");
          try {
               if (!binFile.exists()) {
                    binFile.createNewFile();
               }
               if (!binFile.isFile()) {
                    throw new IllegalArgumentException("Is not
a file: " + binFile);
               }
               if (!binFile.canWrite()) {
                    throw new IllegalArgumentException(
                              "The file cannot be written: " +
binFile);
               }
               Writer binOutput = new BufferedWriter(new
FileWriter(binFile));
               try {
                    binOutput.write(binCode);
                    System.out.print(this.getErrors());
               } finally {
                    binOutput.close();
               }

          } catch (IOException ex) {
               ex.printStackTrace();
          }

     }
```

```
    public static void main(String[] args) {
        GMif smc = new GMif();
        String file = args[0];
        smc.translate(file);
    }

}
```