UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA –
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

JOSIAS DIEGO MARTINS

Experimentos em Síntese de Alto Nível Orientada à Minimização de Área e Potência

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Microeletrônica.

Orientador. Dr. Sergio Bampi

Porto Alegre
Agosto de 2019

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Martins, Josias Diego

Experimentos em Síntese de Alto Nível Orientada à Minimização de Área e Potência / Josias Diego Martins– Porto Alegre: Programa de Pós-Graduação em Micoeletrônica da UFRGS, 2019.

97 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Micoeletrônica. Porto Alegre, BR – RS, 2018, Orientador: Sergio Bampi

1. HLS. 2. Area Saving. 3. Low Power. 4. Standard Cells 5. FPGA. I. Bampi, Sergio. II Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretor do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do PGMicro: Prof. Tiago Roberto Balen
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

RESUMO

O desenvolvimento de CIs complexos representa alto custo de engenharia devido à quantidade de horas consumidas em projeto. O projeto e a fabricação de "*Application Specific Integrated Circuits*" (ASIC) foi impulsionada pelo advento das Linguagens de Descrição de Hardware (HDL) e de ferramentas de síntese lógica. Entretanto, devido à crescente demanda por CIs, mesmo os ganhos em produtividade obtidos ao utilizar HDL já não são suficientes, abrindo espaço para adoção de *High Level Sinthesys* (HLS), uma vez que sua adoção representa grandes ganhos em tempo de desenvolvimento (COMPANHIA4HLS, 2018).

Este trabalho trata de técnicas de otimização de área e de redução de consumo elétrico, visando a implementação em FPGA e CMOS *Standard Cells*. São revisadas técnicas como *clock gating*, *power gating* e *frequency scaling* juntamente com as principais técnicas destinadas à redução de área, sendo abordado o compartilhamento de recursos de *hardware* e uso de módulos/núcleos especializados de alta performance.

A pesquisa desta Dissertação é direcionada à aplicação destes conceitos em exploração de ferramentas HLS acadêmicas e comerciais, realizando comparação entre os resultados obtidos através de HLS e codificação manual diretamente em HDL. Para base de comparação, são selecionados 3 grupos de circuitos: uma ULA 16 bits, filtros de Resposta ao Impulso Finita (FIR) de 40 e 120 estágios ("*taps*") e um processador *Very Long Instruction Word* (VLIW). Através da inserção de diretivas (*pragmas*), é possível guiar a ferramenta HLS em determinada direção, seja para aumento de desempenho ou para redução de área. São adotadas neste trabalho técnicas de "*Design Space Exploration*" (DSE), realizando testes iterativos de modo a buscar o melhor conjunto de diretivas possível para guiar a HLS.

Neste contexto, este trabalho apresenta os resultados de exploração do uso de ferramentas HLS através de uma perspectiva orientada ao baixo consumo e à redução de área para FPGAs e *Standard Cell* mediante adoção de DSE. Comparações entre os resultados são realizadas, utilizando uma normalização por custo de implementação lógica para diferentes alvos tecnologias (FPGAs ou std-cell). Verificou-se que as ferramentas destinadas a FPGAs apresentam ganhos em consumo elétrico por operação e área apenas em circuitos de menor complexidade, sendo que ferramentas de HLS destinadas às *Standard Cells* representam boas oportunidades, rivalizando com resultados obtidos em codificação manual.

Palavras-chaves: Engenharia Eletrônica. Microeletrônica. Síntese de Alto Nível (HLS). FPGA. Técnicas *Low-Power*.

ABSTRACT

The development of complex integrated circuits carries a high non-recurring engineering cost, due to the number of man-hours spent in the design phase. The design and fabrication of digital Application Specific Integrated Circuits (ASIC) was facilitated by the introduction of Hardware Description Languages (HDL) and logic synthesis tools. However, a growing demand for more complex ICs makes the productivity gains enabled by HDL and logic synthesis not sufficient, which opens up for the adoption of High Level Synthesys (HLS).

This work deals with design optimization techniques targeting both power and area, with implementation in FPGAs or CMOS *Standard-Cells*. Techniques like clock gating, power gating, and frequency scaling, together with the main techniques for area reduction, are initially reviewed. The issues of hardware resource sharing and the use of specialized hardware blocks/modules of high performance are also dealt with.

The research in this M.Sc. Thesis targets the application of these concepts in the exploration of both academic and comercial HLS tools available in the Market, comparing the results obtained through HLS with results obtained from manual (man-made) coding directly in HDL. Three groups of circuits are selected here for comparisons: one 16-bit ALU, digital finite impulse response (FIR) filters with 40 and 120 stages (taps), and a VLIW (very long instruction word) processor. This work shows that the insertion of HLS compiling directives (*pragmas)* it is possible to guide the HLS tool for increase the IC performance or reduce his hardware area. Design space exploration (DSE) techniques are adopted in this work, performing iterative tests aiming at the best possible solution set (i.e. the set of directives).

In this context, the results of DSE are presented in this work, using 3 different HLS tools to synthesize specific test circuits. The goal is set in the DSE is to reduce power and area in FPGAs and CMOS Standard Cells.. Comparisons are presented between the results from different tools, using the normalization of cost in terms of hardware area and power dissipation for each implementation target (FPGAs or std-cells). It was verified that the tools destined to FPGAs present gains in power consumption by operation and area only in circuits of less complexity, and the tools destined to std-cell represent good opportunities, rivaling with results obtained in manual codification.

Keywords: Electronic Engineering. Microelectronics. High Level Synthesis (HLS). FPGA. *Low-Power* Techniques.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ALM	Adaptative Logic Module
ASIC	Application Specific Integrated Circuits
AVS	Adaptative Voltage Scaling
BRAM	Block RAM
CE	Clock Enable
CIs	Circuitos Integrados
CLP	Controlador Lógico-Programável
DFS	Dynamic Frequency Scaling
DPG	Dynamic Power Gating
DSE	Design Space Exploration
DSRAM	Dual-port Static Random Access Memory
FF	Flip Flop
FIFO	First In First Out
FIR	Resposta ao Impulso Finita
HDL	Linguagens de Descrição de Hardware
HLL	Linguagens de alto nível
HLS	High Level Sinthesys
HM	Hand Made
LUT	LookUp Table
MCM	Multiple Constant Multiplications
RAM	Random Access Memory
RTL	Register Transfer Level
SoC	System on a Chip

SSRAM	Single-port Static Random Access Memory
ULA	Unidade Lógica Aritmética
VHDL	VHSIC Hardware Description Language
VID	Voltage Identification Bit
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VM	Máquina Virtual

LISTA DE SÍMBOLOS

P_{avg}	Potência média
$P_{switching}$	Potência de dinâmica
$P_{short-circuit}$	Potência de Curto Circuito
P_{static}	Potência Estática
I_G	Corrente de fuga por tunelamento no oxido do porta
I_{SUB}	Corrente de *Subtreshold*
I_{REV}	Corrente polarização reversa
I_{GIDL}	corrente de fuga através do dreno induzida pelo porta
I_H	corrente de porta por injeção
ON	Nível lógico alto
Vdd	Tensão de dreno ou alimentação do circuito
Si	Silício
SiO_2	Oxido de silício
Vth	Tensão de limiar
C_L	Capacitância de carga
P_{din}	Potência dinâmica
α	Probabilidade de chaveamento
f	Frequência
$Time_{process}$	Tempo de processamento, também chamado de atraso total
T_{clock}	Período de clock
$Ciclos$	Ciclos de clock necessários para conclusão de um processamento

SUMÁRIO

1 INTRODUÇÃO

A rápida evolução dos processos de fabricação de Circuitos Integrados (CIs) visto na indústria na última década impulsionou a adoção destes dispositivos. Um crescente movimento em torno do mercado *mobile* tem agitado a indústria, de modo que o tempo de desenvolvimento para projetos envolvendo CIs se torna um problema recorrente ao balancear os custos de projeto. Em meados de 1980, uma alternativa para descrever CIs foi desenvolvida com base na descrição comportamental destes circuitos, dando origem a linguagens como *Verilog e VHSIC Hardware Description Language* (VHDL) (COUSSY, 2009). Esta metodologia permite que CIs sejam desenvolvidos em grande velocidade, reduzindo custos e encurtando o caminho até as prateleiras. No entanto, na atualidade o tempo necessário para descrever grandes circuitos digitais em HDL já é considerado muito longo. Como alternativa ao HDL, surge o *High Level Synthesis* (HLS) , técnica que consiste em converter uma Linguagem de Alto Nível (HLL) em HDL para posterior conversão em Register Transfer Level (RTL), que pode ser sintetizado diretamente para um FPGA ou ASIC.

Junto a demanda por agilidade, cresce no mercado a demanda por circuitos integrados de baixo consumo elétrico. Toda a sorte de dispositivos como smartphones, câmeras e até mesmo marca-passos necessitam de um *System On a Chip* (SoC) ou ASIC de baixíssimo consumo elétrico. Desta forma as empresas de ferramentas de auxílio ao projeto eletrônico estão em uma corrida para entregar a melhor ferramenta HLS, que proporcione agilidade no desenvolvimento, entregando um RTL capaz de gerar um CI com baixo consumo e pequena área ocupada. O resultado final de potência não depende apenas da síntese de alto nível, uma vez que as ferramentas de síntese lógica também fazem otimizações guiadas pelas heurísticas adotadas nesta segunda etapa genérica de síntese. E as ferramentas de síntese física de CIs customizados, em uma terceira etapa, também utilizam técnicas para minimização de área e de potência. Assim, a otimização de potência segue sendo um tema de pesquisa, e as ferramentas de HLS necessitam ser avaliadas em conjunto com as melhores práticas e técnicas das ferramentas de síntese lógica e de síntese física.

Sendo assim, objetiva-se analisar o estado atual das ferramentas HLS, contrastando-as com soluções obtidas através de codificação manual (feita diretamente pelo projetista em HDL), mapeando e corroborando os ganhos possíveis através do seu uso. Busca-se também desenvolver uma metodologia para uso destas ferramentas, seja através de DSE ou através da

seleção sistemática de diretivas através de políticas de boas práticas de projeto, analisando seu impacto na qualidade dos RTL.

1.1 Motivação

O projeto de CIs sempre foi algo acessível apenas a empresas de grande porte (como Intel, Toshiba, Texas Instruments, AMD e Samsung), com grande volume de produção. Porém, após o ano de 1980, houve uma mudança no mercado, devido ao surgimento do modelo de negócio de empresas especializadas em fabricação de *wafers* de silício para terceiros (as empresas "*foundries*"), segmento comercial que tem atualmente a empresa TSMC como líder de mercado. Além disto, para baratear os custos de prototipação, novas tecnologias também permitem o compartilhamento do *wafer* de silício por diferentes CIs, reduzindo os custos de produção para menores escalas (baixo número de CIs) e facilitando à customização de circuitos digitais por empresas de médio porte. Este fato traz à tona um novo mercado, permitindo a produção de ASICs para as mais diversas aplicações com novas possibilidades para uma grande quantidade de empresas de médio porte. No entanto, o desenvolvimento (na etapa denominada de "design") destes circuitos até o ponto da especificação completa do leiaute físico das máscaras que possibilitarão a posterior fabricação do CI, ainda permanece custoso, tornando inviável sua adoção por um universo expressivo de empresas de eletrônica. Por outro lado, com a ascensão do HLS como alternativa de codificação para especificar o hardware digital, é possível ao *designer* descrever o comportamento do circuito diretamente em C/C++ e converter (após diversas e complexas etapas de síntese: síntese de alto nível, síntese lógica e finalmente a síntese física) em equivalente de hardware para FPGAs ou *Standard Cell*. A adoção desta abordagem proporciona maior vantagem competitiva no mercado em relação a este mesmo código executando sobre um processador de uso geral (seja em desempenho, custo ou consumo de potência e de energia elétrica). Analisando o atual cenário dos FPGAs, fabricantes iniciam uma corrida pela melhor ferramenta HLS e melhores resultados. Os dois principais *players* do mercado ("Companhia 1"[1] e "Companhia 2"[2]) apresentam FPGAs com capacidades de *LowPower* (ainda rudimentares em relação aos ASICs implementados com *Standard Cells*), e

[1] Nome da Companhia 1 é omitido por acordo de confidencialidade no uso de ferramentas.
[2] Nome da Companhia 2 é omitido por acordo de confidencialidade no uso de ferramentas.

podem representar uma boa relação custo x benefício para produção de dispositivos customizados com baixo volume de produção.

Deste modo, busca-se analisar os resultados de ferramentas de Síntese Lógica de Alto Nível adotadas para FPGAs e *StandardCell* de forma a obter um panorama atualizado destas ferramentas e a qualidade do resultado obtido, bem como os ganhos em desempenho e custos de desenvolvimento. Salienta-se que, o processo de análise realizada estará norteado pelo custo final por chip e pelo desempenho e/ou número de unidades lógicas (quando FPGAs), uma vez que não existem formas de comparação direta entre tecnologias de FPGAs de diferentes fabricantes e *Standard Cell*, apresentando assim um panorama geral e atualizado do esforço de desenvolvimento de circuitos *Very Large Scale Integration* (VLSI) através de HLS que pode servir de auxílio na tomada de decisão em relação a escolha da tecnologia a ser adotada e forma de codificação.

1.2 Contribuições

A contribuição principal desta dissertação de mestrado está relacionada à análise da atual situação de algumas ferramentas HLS de mercado. Uma metodologia para exploração de espaço de design, que realiza testes iterativos combinando diferentes pragmas para mapeamento das soluções possíveis em um universo de possibilidades para análises posteriores e finalmente, uma análise qualitativa levando em consideração o desempenho (i.e. a "*performance*" no domínio tempo) da solução final e consumo elétrico comparado ao custo de produção.

1.3 Organização da Dissertação

Uma revisão sobre os conceitos básicos de síntese de alto nível aplicada ao projeto de sistemas digitais é apresentada no Capítulo 2 desta Dissertação, bem como a apresentação das ferramentas HLS exploradas, que iniciou com o presente Capítulo de introdução. O Capítulo 3 versa sobre as principais fontes de consumo elétrico em Circuitos Integrados CMOS, abordando técnicas para redução de potência dissipada e para redução de área. O Capítulo 4 discorre sobre o uso de *Design Space Exploration* para teste iterativo de conjunto de diretivas de projeto, apresentando a metodologia adotada nesta dissertação. Os resultados obtidos a partir dos testes são apresentados no Capítulo 5, junto a breves análises sobre cada caso posto à prova. Finalmente, no Capítulo 6, são apresentadas as conclusões.

2 HIGH LEVEL SYNTHESIS

Com o crescimento de complexidade dos circuitos integrados e algoritmos implementados, novas abordagens para projetos são necessárias. Ferramentas para design de circuitos integrados passam a realizar diversos níveis de abstração, automatizando e acelerando os processos de síntese e verificação, de modo a proporcionar maior agilidade durante a etapa de projeto.

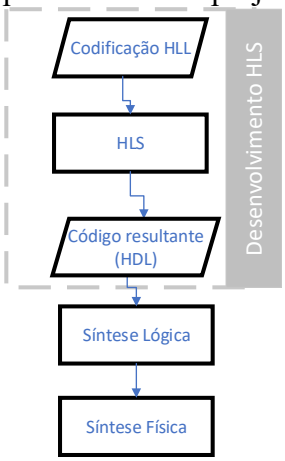
Com o advento da computação, na década de 50 é criado o conceito da linguagem *assembly*, e finalmente, são desenvolvidas linguagens de alto nível (HLL), que proporcionam maior velocidade para o desenvolvimento de softwares por não demandar conhecimento específico acerca do processador ao programador. Estas novas linguagens apresentam maior flexibilidade aliadas a uma linguagem textual de sintaxe mais simples e humana. O acesso a estruturas de hardware passa a ser gerenciada pelo compilador, tornando o algoritmo desenvolvido compatível com qualquer *hardware* cujo compilador e bibliotecas sejam compatíveis (COUSSY, P. 2009).

Da mesma forma no domínio do *hardware* novas abordagens surgem em contraponto a metodologia tradicional de desenvolvimento de CIs – onde originalmente cada transístor é desenhado manualmente – através das Linguagens de Descrição de *Hardware* (HDL), criadas em 1986 (COUSSY, P. 2009).

Como uma nova abordagem para descrição de circuitos digitais, surge o *High-Level Synthesis* (HLS), metodologia que propõe o uso de linguagens de alto nível para descrição comportamental de circuitos, e é na década de 90 que surgem as primeiras ferramentas comerciais voltadas a este fim (COUSSY, P. 2009).

Deste modo, HLS passa a ser adotado antes do processo de Síntese Lógica, acelerando a etapa de codificação – isto é, de desenvolvimento dos algoritmos que serão traduzidos em hardware – através da adoção de descrição comportamental diretamente em HLL. A Figura 2.1 apresenta um fluxograma contendo as etapas principais de um projeto VLSI adotando o fluxo HLS.

Figura 2.1 – Fluxo simplificado de um projeto VLSI adotando HLS



Fonte – O Autor

Conforme a Figura 2.1, o HLS é inserido no processo de desenvolvimento do circuito onde tradicionalmente ocorre a codificação em HDL, de modo a acelerar o processo de descrição do circuito através do foco em seu comportamento.

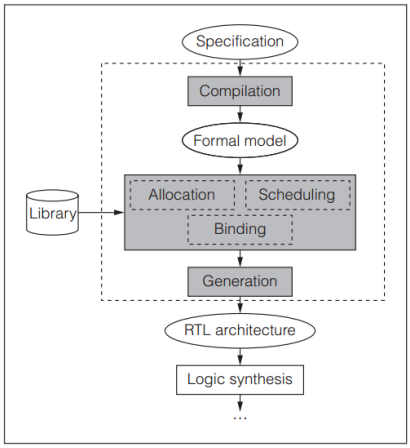
Deste modo, a ferramenta HLS tem como objetivo realizar a análise do código sequencial do algoritmo escrito em linguagem de alto nível como C/C++ convertendo no equivalente em RTL. Este processo é extremamente complexo, e necessita de diversas aproximações, uma vez que linguagens de alto nível como C/C++ não são temporizadas (no-timed), executando suas funções a cada ciclo do processador (ZHANG, Z. 2015).

A solução adotada consiste em avaliar cada operação lógica realizada pelo algoritmo durante o fluxo HLS, transformando estas operações em passos (*steps*) que correspondem a ciclos de clock. A ferramenta HLS, desta forma, realiza o mapeamento dos recursos necessários para implementar o comportamento do algoritmo descrito no equivalente em *hardware* (ZHANG, Z. 2015).

O fluxo HLS é separado em três tarefas principais: descrição do sistema *(System Description*), escalonamento (*Scheduling*) e alocação de recursos e vinculação (*resource allocation and bind*). Onde a etapa de descrição de sistemas está diretamente ligada a especificação de funcionamento (descrição comportamental), escalonamento realiza um mapeamento das operações verificando qual o requisito de tempo (*timing*) para execução e por fim, alocação de recursos e vinculação define as estruturas em hardware necessárias para replicar o comportamento descrito (EVANS, J. 2010).

Durante o fluxo HLS, estas etapas aparecem em pontos diferentes. A Figura 2.2 apresenta um fluxograma que descreve o fluxo HLS e suas principais etapas.

Figura 2.2 – Fluxo HLS



Fonte – COUSSY, 2009

Conforme a Figura 2.2, é possível verificar que para Coussy, a etapa de Descrição do Sistema (Especificação) não é considerada parte do fluxo HLS, sendo que este tem início a partir da compilação e é finalizado quando o arquivo RTL é gerado. Desta forma, o fluxo HLS passa pelas etapas listadas a seguir:

- Compilação (*Compilation*): Onde é realizada a compilação da descrição comportamental do sistema, obtendo como resultado um modelo formal (*formal model*);
- Alocação (*allocation*): Realiza a alocação de recursos de Hardware (unidades funcionais, registradores, barramentos e etc);
- Escalonamento (*Schedulling*): A partir da análise do comportamento do algoritmo e a separação das operações em *steps*, é possível realizar uma análise de *timing*, convertendo em ciclos de clock;
- Vinculação (*Binding*): Realiza a vinculação das operações em unidades funcionais, variáveis em elementos de memória e transferências em barramentos;
- Geração (*Generation*): Etapa onde é criado o arquivo RTL, contendo uma descrição de hardware para o algoritmo comportamental.

Observa-se que a descrição de etapas proposta por Coussy assemelha-se a descrição proposta por Evans, onde a única diferença reside na exclusão da etapa de descrição do sistema.

Para Evans, a qualidade do resultado final do RTL gerado pela ferramenta também está ligada a qualidade da descrição funcional/comportamental do sistema, que tem início com a especificação. Este fato ocorre devido a estrutura de descrição comportamental de um sistema apresentar problemas significativos de concorrência e hierarquia, necessitando de uma ferramenta eficiente para gerenciar os recursos (EVANS, 2009). Sendo assim, a etapa de Descrição do Sistema deve ser considerada uma das etapas do fluxo HLS, tendo em vista a criação de uma descrição mais amigável as ferramentas HLS, pensada para paralelismo, através da adoção de máquinas de estado finito (FSM) e outras técnicas de segmentação.

As etapas de alocação, escalonamento e vinculação são interdependentes e podem ser executadas ao mesmo tempo, porém esta abordagem torna o modelamento extremamente complexo. Usualmente, estas etapas são executadas em sequência, de modo a atingir um determinado objetivo, sendo delimitado através das restrições de projeto (*pragmas*) descritos na ferramenta. A alocação pode ser realizada primeiro, enquanto a etapa de escalonamento é responsável por minimizar a latência e maximizar o *throughput*. A alocação pode ser determinada durante o escalonamento enquanto este minimiza a área atendendo a especificação de *timing* (COUSSY, P. 2009).

Desta forma, as oportunidades de aperfeiçoamento e melhoria nos resultados obtidos através de HLS encontram-se dentro destas três etapas, não obstante a isto, também em uma boa segmentação no código desenvolvido.

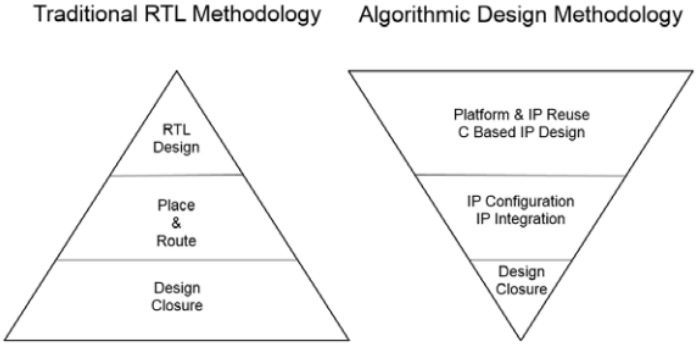
2.1 FERRAMENTAS HLS

Existem diversas ferramentas HLS disponíveis atualmente, sejam elas comerciais ou acadêmicas. Cada desenvolvedor adota uma técnica diferenciada para realizar o processo de conversão de código comportamental não temporizado para o equivalente temporizado, através do fluxo HLS. Algumas ferramentas utilizam uma vasta biblioteca de IPCore (*Intelectual Property Core*) – módulos e estruturas prontas que realizam determinada função – disponível para mapear o código comportamental em equivalente HDL, convertendo estes posteriormente em *primitives* de uma tecnologia específica (normalmente do próprio desenvolvedor da ferramenta HLS), enquanto outras ferramentas possuem como opção realizar parte do mapeamento através da conversão de instruções de alguma arquitetura de processador (como "Ferramenta 2", através de um processador MIPS ou ARM). Nas subseções a seguir, são apresentadas as peculiaridades do fluxo de desenvolvimento de cada ferramenta – conforme é possível, mediante liberação de informações dos próprios desenvolvedores – permitindo uma análise mais aprofundada e aplicação dos conceitos do fluxo HLS padrão, *Low Power* e *Area Saving*, para obtenção de melhores resultados no processo de síntese lógica.

2.1.1 "Ferramenta 1"

O "Ferramenta 1" surge como uma alternativa para acelerar o desenvolvimento de IPs. Baseado no fluxo HLS e integrado ao "Ferramenta 1" Design Suite, a ferramenta traz flexibilidade e agilidade ao fluxo de design lógico-digital.

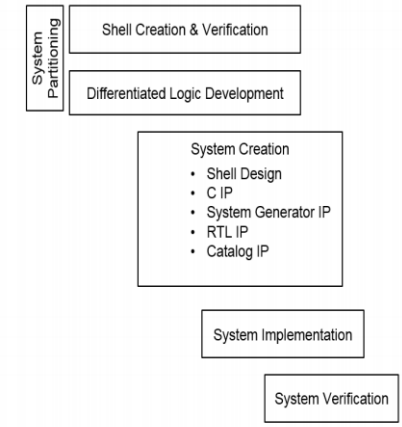
Figura 2.3 – Comparativo entre metodologias de Design



Fonte – COMPANHIA1g, 2017

A Figura 4.1, apresenta a metodologia de desenvolvimento tradicional e a metodologia de desenvolvimento baseada em algoritmo, proposta pela "Companhia 1"[3] através da suíte "Ferramenta 1" e sua ferramenta HLS (denominada "Ferramenta 1"). A metodologia tradicional (*Traditional RTL Methodology*) tem início através da experiência de *designers* profissionais que realizam uma estimativa de como o design deve ser implementado na tecnologia disponível. Desta forma, estes iniciam o modelamento do projeto em nível RTL, passando por etapas iterativas de síntese, *place and route* e testes, de modo a confirmar as estimativas iniciais e garantir que o *design* realiza as funções especificadas. Ao final do fluxo, o *designer* (ou equipe) deve reconfirmar que o *design* implementado representa as especificações corretamente. A metodologia algorítmica (*Algorithmic Design Methodology*) proposta através do "Ferramenta 1" e "Ferramenta 1" representa os mesmos passos básicos, porém, este se foca no desenvolvimento de uma casca (*shell*), onde o *designer* aplica mais tempo analisando a estrutura e os atributos do dispositivo. Desta forma, o projetista tem como intuito realizar o mapeamento dos I/Os necessários e descrever a lógica por trás de seu funcionamento utilizando uma linguagem baseada em C ou C++. A suíte , assim, realiza o processo de síntese mapeando o comportamento do algoritmo descrito através do fluxo de síntese HLS, convertendo no equivalente presente nas bibliotecas da ferramenta (COMPANHIA1g, 2017). A Figura 4.2 apresenta o fluxo de design proposto pela "Companhia 1".

Figura 2.4 – Fluxo de Design de alta produtividade



Fonte – COMPANHIA1g, 2017

[3] Nome da Companhia 1 é omitido por acordo de confidencialidade

Conforme o fluxo proposto pela "Companhia 1"[4] na Figura 4.2, o desenvolvimento tem início com a criação da casca (*shell Design*) e sua verificação. A casca consiste na visualização macro do dispositivo, contendo todos os I/Os necessários e descrição de itens internos. A ferramenta consegue realizar um mapeamento prévio e auxiliar na seleção de IPs e na conversão de *primitives* providas pela própria "Companhia 1" de modo a agilizar o processo de desenvolvimento. A criação do sistema tem início com a descrição comportamental do dispositivo através da linguagem C (*C IP*), seguindo o fluxo HLS padrão, realizando o mapeamento das instruções e rotinas descritas para o equivalente em *hardware* disponível nas bibliotecas da ferramenta (COMPANHIA1g, 2017).

Todas as etapas do fluxo podem ser realizadas de forma manual ou automática através de scripts. A ferramenta proporciona flexibilidade ao *designer* para customizar cada etapa do processo e inserir diretivas *(pragmas)* que podem representar certos ganhos em áreas distintas do circuito final (*área*, performance*,* consumo elétrico e etc). Deste modo, é de grande importância a exploração dos manuais técnicos da ferramenta sob a ótica do fluxo HLS, de modo a extrair o máximo que a ferramenta pode oferecer.

2.1.2 Ferramenta 2

O "Ferramenta 2" é uma poderosa ferramenta HLS desenvolvida e mantida pela Universidade de Toronto, desde 2010, cujo objetivo principal visa trazer a pesquisadores a experiência de realizar todo o processo de Síntese Lógica de Alto Nível a partir da menor abstração de *hardware* possível, permitindo um minucioso estudo acerca do processo HLS. Como um objetivo secundário, o "Ferramenta 2" visa tornar o desenvolvimento (descrição) de circuitos digitais para FPGA mais simples e rápida (COMPANHIA3, 2015).

O "Ferramenta 2" aceita ANSI C *Vanilla* como entrada, sem a necessidade de qualquer parâmetro especial, produzindo uma descrição de hardware em Verilog como saída que pode ser sintetizado diretamente em FPGAs da "Companhia 2"[5]. De modo a realizar este procedimento, o "Ferramenta 2" possui duas metodologias distintas, que são listadas a seguir (COMPANHIA3, 2015):

---

[4] Nome da Companhia 1 é omitido por acordo de confidencialidade
[5] Nome da Companhia 2 é omitido por acordo de confidencialidade

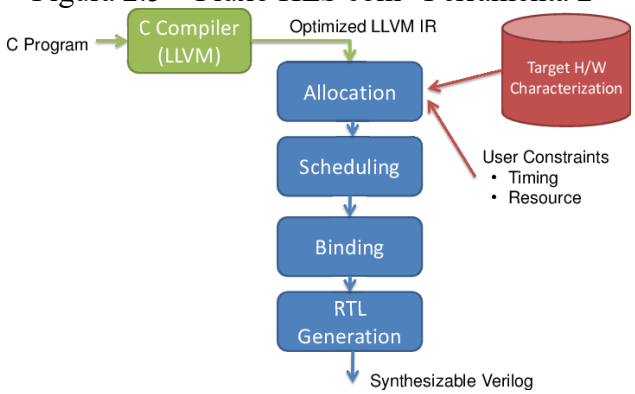
1. Hardware Dedicado: Onde o comportamento descrito em C é convertido diretamente para estruturas equivalentes em hardware, sem uso de um processador;
2. Híbrido: Onde uma parte do código C é executado em um processador TigerMIPS ou ARM, e o restante é convertido em estruturas de *hardware* equivalentes que são mapeados em lógica configurável (do tipo FPGA).

A ferramenta é executada sobre plataforma Linux, e faz uso de scripts em TCL para guiar todo o fluxo HLS. São suportados FPGAs da família CycloneII e StratixIV, sendo definidos por parâmetros através de script TCL. O "Ferramenta 2" utiliza um período de clock alvo *default* de 15ns para FPGAs CycloneII e 5ns para FPGAs StratixIV (COMPANHIA3, 2015).

2.1.2.1 Fluxo de Projeto com "Ferramenta 2"

A partir das duas técnicas distintas de síntese lógica, o "Ferramenta 2" possui quatro fluxos de projetos predefinidos. Cada fluxo de projeto apresenta uma sequência diferenciada de modo a obter diferentes resultados, baseando-se na adoção de estruturas de hardware e compartilhamento de núcleo TigerMIPS para execução de trechos de código C. A apresenta o fluxograma que demonstra o Fluxo HLS base adotado no "Ferramenta 2".

Figura 2.5 – Fluxo HLS com "Ferramenta 2"



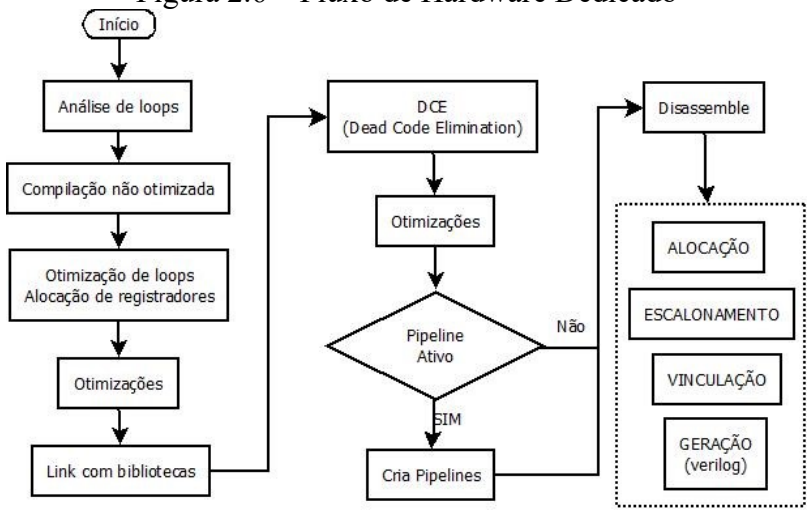
Fonte – COMPANHIA3, 2015

Conforme a , o fluxo base adotado segue o fluxo HLS padrão. A partir do fluxo base, são derivados os quatro fluxos de projeto permitidos no "Ferramenta 2": Fluxo de *Hardware* Dedicado, Fluxo Paralelo, Fluxo Híbrido e Fluxo de Software Puro.

2.1.2.1.1 Fluxo de *Hardware* Dedicado

O fluxo de *Hardware* Dedicado realiza a síntese de todo o código C inserido diretamente em *Hardware*. Neste fluxo, o processo de síntese de alto de nível é similar ao presente em outras ferramentas HLS (COMPANHIA3, 2015). A apresenta o fluxograma contendo as etapas do fluxo de Hardware Dedicado expandidas, compilado a partir da documentação do "Ferramenta 2".

Figura 2.6 – Fluxo de Hardware Dedicado



Fonte – O Autor

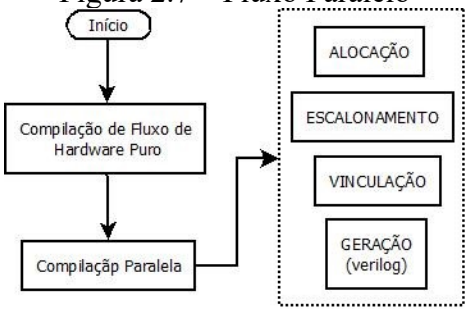
A apresenta o fluxo HLS expandido através do Fluxo de *Hardware* Dedicado presente no "Ferramenta 2". Os procedimentos que ocorrem entre o início do processo de compilação e o *disassemble* compreendem a primeira etapa do processo HLS. O "Ferramenta 2" inicia realizando uma análise do código C de modo extrair os loops. Posteriormente, uma compilação não otimizada é realizada de modo a mapear o comportamento do código e realizar a otimização dos loops e alocação de registradores (convertendo elementos de memória como variáveis em registradores). Uma nova etapa de otimização é realizada e as estruturas são comparadas e vinculadas a estruturas equivalentes presentes na biblioteca do "Ferramenta 2". Na sequência, é realizado um processo de "eliminação de código morto", isto é, o "Ferramenta 2" excita todos os caminhos lógicos e verifica quais caminhos nunca são percorridos, eliminando-os de modo a reduzir o código e consequentemente a área ocupada pelo circuito final, que será obtido após as sínteses lógica e física . Este processo é seguido de nova otimização, e posteriormente, a ferramenta realiza a conversão de determinadas

estruturas em *pipelines*, caso esta opção tenha sido configurada. Por fim, é realizado um processo de *disassamble*, onde é criado o arquivo final, que representa o resultado de todo o processo. O restante do fluxo HLS segue o padrão, onde a ferramenta realiza o processo de alocação, escalonamento e vinculação com base na escolha do FPGA e as estruturas disponíveis neste, para gerar um arquivo Verilog sintetizável (COMPANHIA3, 2015).

2.1.2.1.2 Fluxo Paralelo

O “Ferramenta 2” possui como opção de gerar código RTL para execução paralela, isto é, criando aceleradores. Este processo é realizado utilizando Pthreads e OpenMP, onde cada *thread* é compilada como um acelerador e o “Ferramenta 2” realiza o instanciamento destes diretamente, onde o número de *threads* presentes no código C será equivalente ao número de aceleradores criados. Esta abordagem apresenta um grande atrativo para computação de alta performance e otimização de procedimentos específicos, como comunicação e processos ligados a DSP (COMPANHIA3, 2015). A apresenta o Fluxo Paralelo, compilado a partir da documentação do “Ferramenta 2”.

Figura 2.7 – Fluxo Paralelo



Fonte – O Autor

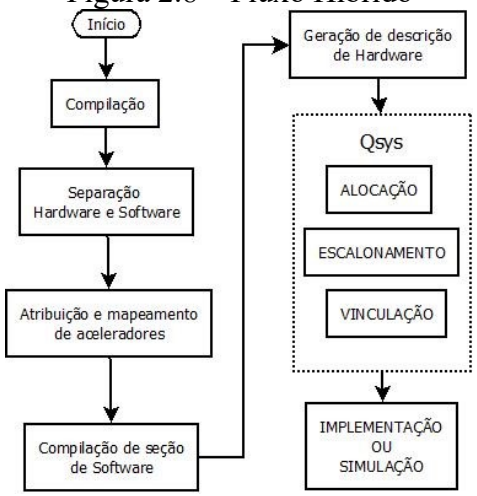
Conforme apresentado na , o Fluxo Paralelo consiste basicamente em uma etapa complementar ao Fluxo de Hardware Dedicado, sendo executada antes dos processos de alocação, escalonamento e vinculação. Desta forma, através desta etapa complementar, a ferramenta é capaz de “desmembrar” o circuito e identificar estruturas que podem ser geradas como aceleradores antes de realizar os procedimentos de mapeamento tecnológico.

2.1.2.1.3 Fluxo Híbrido

O Fluxo de Design Hibrido é um dos diferenciais do “Ferramenta 2”, e basicamente consiste em um processo que realiza o mapeamento do código C em hardware (gerando aceleradores) e separando restante do código, de maior complexidade de conversão, para ser executado em um processador TigerMIPS ou ARM Cortex-A9. A comunicação entre os aceleradores gerados e o processador é realizado através de um barramento Avalon, gerado

automaticamente pelo QSys System Integration Tool da “Companhia 2” [6]. A apresenta o fluxograma de descrição do fluxo Hibrido, compilado a partir da documentação do “Ferramenta 2”.

Figura 2.8 – Fluxo Hibrido



Fonte – O Autor

A apresenta as etapas que compõe o Fluxo Hibrido, onde o processo tem início com a compilação e a separação dos trechos de código C que deverão ser convertidos em aceleradores em *hardware* ou compilados para um dos processadores disponíveis (TigerMIPS ou ARM). O processo de alocação, escalonamento e vinculação é realizado diretamente pela ferramenta Qsys da “Companhia 2”, responsável por converter o código RTL nas estruturas de hardware disponíveis no FPGA alvo e realizar a interligação dos aceleradores através do barramento Avalon ao processador (COMPANHIA3, 2015).

É importante salientar que o Fluxo Hibrido somente é compatível com FPGAs da série Cyclone V e StratixV. A separação de código para alocação dos aceleradores e software alvo para processador também é diretamente influenciado pelo FPGA alvo, uma vez que apenas a família Cyclone V possui núcleo ARM disponível em Hardware, possibilitando a configuração deste como alvo no Fluxo Híbrido. Para a família Stratix V, está disponível o mapeamento apenas para o processador TigerMIPS, que passa a ser sintetizado diretamente no FPGA.

---

[6] Nome da Companhia 2 é omitido por acordo de confidencialidade

2.1.2.1.4 Fluxo de *Software* Puro

O Fluxo de Software Puro consiste basicamente na compilação do código C tendo como alvo o processador TigerMIPS ou ARM. Este Fluxo tem como objetivo permitir a simulação e teste do código desenvolvido de forma a compatibilizá-lo a um dos processadores alvo, tornando o processo de desenvolvimento de descrição comportamental Hibrida mais simples e fácil de ser testada (COMPANHIA3, 2015).

2.1.3 "Ferramenta 3 Suite"

O "Ferramenta 3 Suite" surge a partir do "Ferramenta 3", uma ferramenta HLS desenvolvida pela "Companhia 4"[7] (e em um certo período de tempo pela entidade comercial "Ferramenta 3") e adicionada à sua suíte de ferramentas em 2004, com foco em desenvolvimento de circuitos digitais para múltiplas plataformas – FPGAs e ASICs – tendo recebido atualizações recentes que permitem um fluxo guiado diretamente a *Low Power*. O "Ferramenta 3 Suite", atualmente corresponde a uma suíte constituída de três ferramentas voltadas a síntese de alto nível ("Ferramenta 3"), verificação formal ("Ferramenta 3 Verify") e otimização de potência ("Ferramenta 3 LPOPT"), possibilitando processo de Design Space Exploration (DSE) do tipo *Closed-Loop* (COMPANHIA4DS, 2017) diretamente orientado a *Low Power*.

O principal diferencial desta suíte ("Ferramenta 3 Suite") reside na integração entre as três principais ferramentas, o "Ferramenta 3", "Ferramenta 3 LPOPT" e "Ferramenta 3 Verify", tornando possível a partir da própria ferramenta, realizar processo de exploração arquitetural (DSE) em malha fechada (closed-loop) guiado a *Low Power*, aplicando diversas otimizações voltadas a potência (COMPANHIA4LP, 2018). As otimizações, no entanto, podem apresentar resultados diferentes a depender da tecnologia alvo (ASIC ou FPGA), devido principalmente, as limitações inerentes à arquitetura escolhida, como suporte a múltiplos domínios de *clock*, *clock gating*, *frequency scaling*, múltiplos domínios de tensão e *power gating*, que somente são suportados em sua plenitude com *Standard Cell*. Porém muitas otimizações apresentam efeito indireto na potência em todas as arquiteturas devido a

[7] Nome da Companhia 4 é omitido por acordo de confidencialidade

simplificações e otimizações que causam a redução de área, impactando diretamente na quantidade de elementos lógicos ativos.

2.1.3.1 “Ferramenta 3”

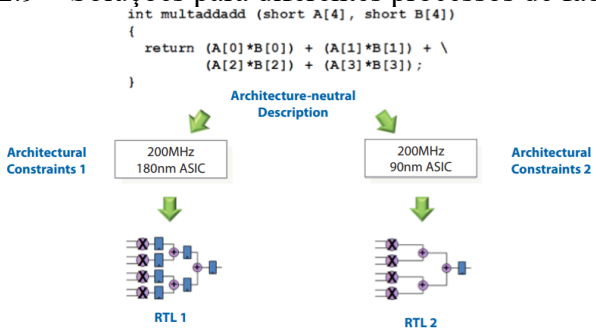
O “Ferramenta 3” atua de forma similar as demais ferramentas HLS, realizando a análise das rotinas comportamentais de modo a separar as seções de código que possam ser e quebradas em pequenos passos (*setps*) para temporização do circuito, seguindo todos os demais passos do fluxo HLS. Consequentemente, a ferramenta apresenta em seu fluxo padrão a possibilidade de seleção entre fluxo orientado a área (*area*) ou performance (*latency*), realizando a alteração do fluxo e otimizações conforme a necessidade para atingir o *target* (alvo, em português) estipulado (COMPANHIA4DS, 2017).

Como entrada, o “Ferramenta 3 Suite” suporta System C e C/C++, proporcionando grande flexibilidade para descrição comportamental de circuitos. De acordo com COMPANHIA4DS, o uso da ferramenta e das supracitadas linguagens proporcionam um ganho de até 50% no tempo de desenvolvimento de projetos, média obtida após análise de mais de 1000 projetos finalizados, tornando evidente os ganhos em tempo de desenvolvimento obtidos através do fluxo HLS.

2.1.3.2 “Ferramenta 3 LPOPT”

O “Ferramenta 3 LPOPT”, vem como uma resposta a crescente necessidade do mercado em relação a projetos com baixo consumo elétrico, apresentando oportunidade de otimização em potência para projetos HLS. A ferramenta atua em conjunto com ao “Ferramenta 3”, aplicando rotinas de otimizações ao circuito digital sintetizado (já em RTL). Entre as diversas otimizações aplicadas pela ferramenta destacam-se os múltiplos domínios de clock, *frequency scaling* e *bit-width optimization* – este último, reduzindo a largura de barramentos – além de demais otimizações estruturais que podem variar conforme a tecnologia alvo da síntese (COMPANHIA4LP, 2018). A apresenta a diferença entre soluções com mesmo alvo de *clock* porém para diferentes processos de fabricação.

Figura 2.9 – Soluções para diferentes processos de fabricação



Fonte – COMPANHIA4LP, 2018

Conforme a , a ferramenta “Ferramenta 3 LPOPT” realiza a otimização do RTL final com base no alvo de clock, onde a solução “RTL1 ” secciona o processo matemático em múltiplos estágios para um processo de 180 nanômetros, enquanto solução “RTL 2” realiza todo o cálculo em apenas um ciclo de clock visando um processo de 90 nanômetros.

2.1.3.3 Otimizações em malha fechada

O “Ferramenta 3 Suíte” apresenta ferramentas e fluxos específicos para otimização do RTL gerado, a partir de ferramenta de DSE incluída na própria suíte “Ferramenta 3”. São implementadas sete diferentes otimizações, que podem ser utilizadas em conjunto ou separadamente, para atingir determinado objetivo (COMPANHIA4UR, 2018). A Tabela 1 apresenta a lista de otimizações disponíveis na ferramenta “Ferramenta 3 Suite”

Tabela 2-1 – Otimizações disponíveis

Operator Clustering	Analisa o código agrupando rotinas iguais em blocos
“Ferramenta 3” Optimized Reusable Entities (CCOREs)	Permite a síntese de blocos, possibilitando compartilhamento de recursos (reutilização)
Pipeline Flushing and Bubble Compression	Cria *pipeline* em laços
Sharing Analysis and Guidance	Realiza análise das diretivas e comandos aplicando-as durante o processo de síntese
Low Power Optimization	Aplica otimizações de potência (“Ferramenta 3 LPOPT”)
User Defined Delay scaling of ROMs	Define um delay máximo para uso de ROMs
Scheduler Optimization	Realiza otimização em malha fechada (Design Space Exploration) com base em fluxos predefinidos

Fonte – COMPANHIA4UR, 2018

Conforme a , é possível guiar a ferramenta a realizar 7 tipos de otimizações que são inseridas como diretivas da ferramenta, sendo que estas podem ser adotadas de forma concomitante de modo a melhorar os resultados obtidos. Destaca-se que estas otimizações dizem respeito a alterações incluídas no fluxo HLS de modo a obter um resultado específico, e a inserção automática de diretivas (pragmas) para guiar o compilador. No entanto, é importante salientar que a diretiva "*Sharing Analysis and Guidance*" permite que a ferramenta aplique pragmas inseridas manualmente pelo usuário, de modo a refinar os resultados obtidos. Destaca-se ainda, a otimização (ou diretiva) "*Scheduler Optimization*", que habilita um método iterativo realizando um *Design Space Exploration* (DSE) e selecionando automaticamente pragmas de modo a atingir um determinado objetivo (Esforço Lógico, Design, Área ou Latência). Utilizando-se a diretiva "*Schedule Optimization*" concomitante à "*Low Power Optimization*", é possível guiar a ferramenta a criar um circuito digital otimizado para Área e Potência, ou Performance e Potência.

2.1.4 Comparação entre ferramentas

Cada ferramenta HLS descrita apresenta algumas peculiaridades que podem vir a proporcionar ganhos em qualidade do código resultante ou em produtividade. Por exemplo, o "Ferramenta 1" possui IDE integrada, permitindo o desenvolvimento e compilação diretamente a partir de sua interface, enquanto o "Ferramenta 2" em sua versão 4 não possui este suporte. A Tabela 2-2 apresenta um comparativo entre os recursos disponíveis em cada ferramenta.

Tabela 2-2 – Comparativo entre ferramentas HLS testadas

Ferramenta	IDE	FPGA "Companhia 1"[8]	FPGA "Companhia 2"[9]	Standard Cell	Módulos de alto desempeno	Modularização de funções	Múltiplos domínios de relógio	Fluxo HLS de p/ baixo consumo
"Ferramenta 1"	SIM	SIM	NÃO	NÃO	SIM	NÃO	NÃO	NÃO
"Ferramenta 2 v1"	NÃO	NÃO	SIM	NÃO	SIM	NÃO	NÃO	NÃO
"Ferramenta 2 v2"	SIM	SIM	SIM	NÃO	SIM	NÃO	NÃO	NÃO
"Ferramenta 3 Suite"	SIM	SIM	SIM	SIM	SIM	SIM	NÃO	SIM

Fonte – O Autor

Conforme a Tabela 2-2, observa-se que apenas o "Ferramenta 3" apresenta suporte a uma maior gama de tecnologias bem como, suporte a um fluxo HLS guiado a minimização de potência dissipada (Low Power). Insta salientar que, o suporte a "modularização de funções" somente é listado como disponível quando a ferramenta permite que o projetista selecione quais funções devem ser convertidas diretamente em módulos. As ferramentas "Ferramenta 1" e "Ferramenta 2" realizam a conversão de funções em módulos de forma automática, analisando o comportamento do código e selecionando quais serão convertidas em módulos e quais serão inseridas *inline*.

[8] Nome da Companhia 1 é omitido por acordo de confidencialidade
[9] Nome da Companhia 2 é omitido por acordo de confidencialidade

3 ÁREA E POTÊNCIA

Em se tratando de circuitos integrados, a redução de área e potência podem andar juntas. Comumente técnicas destinadas a redução de área apresentam como efeito colateral a redução de potência devido ao melhor uso de recursos de hardware, e a redução da quantidade de componentes ativos no circuito. Entretanto, técnicas voltadas a redução de área devem ser analisadas cuidadosamente, de modo a serem introduzidas e adotadas concomitantemente a técnicas específicas para *Low Power*, analisando o efeito no desempenho global do sistema.

Nas próximas subseções, serão abordadas as principais técnicas para redução de área e consumo elétrico em CIs, bem como as fontes de consumo elétrico encontradas nestes dispositivos.

3.1 Técnicas de redução de área

Atualmente a técnica destinada a redução de área mais utilizada está relacionada com o compartilhamento de recursos e substituição de determinados elementos por equivalentes mais otimizados (BERGAMASCHI, 1997).

Um circuito digital que apresenta diversos processos de soma, por exemplo, poderia adotar uma Unidade Logico-Aritmética (ULA) centralizada, e multiplexar o seu uso entre todas as etapas do circuito, em detrimento de sintetizar um circuito específico para soma a cada módulo que o necessite, reduzindo a área ocupada.

A adoção de módulos especializados como memórias do tipo *Random Access Memory* (RAM) , também são uteis em detrimento do uso de *Block* RAM (BRAM), por ocuparem menor área e apresentarem desempenho superior para a função requerida. Desta forma, é fato que a adoção deste tipo de técnica pode auxiliar inclusive na redução de potência consumida pelo CI como um todo, ao reduzir a quantidade de transístores ou elementos lógicos adotando módulos mais eficientes e compartilhando seus recursos entre outros circuitos que os demandam.

3.2 Fontes de consumo elétrico

A compreensão das fontes de consumo de potência é imprescindível para a correta adoção de técnicas para controle da potência dissipada . Algumas destas fontes estão ligadas diretamente à tecnologia, reduzindo desta forma os ganhos através de técnicas destinadas a *Low Power* em projetos VLSI envolvendo *Standard Cell* e FPGAs, por estes já apresentarem estruturas, modelos e células pré-definidos.

Tipicamente, todo circuito digital apresenta determinadas características de consumo elétrico, que são modeladas em três fontes de consumo. Somadas, estas três fontes constituem o consumo médio de potência de um sistema VLSI. A equação 3.1 apresenta os termos que compõe o consumo médio de potência de um circuito digital.

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{static} \tag{3.1}$$

Conforme a equação 3.1 o consumo elétrico médio para um circuito digital é dado pela soma entre a Potência de chaveamento ou dinâmica ($P_{switching}$), potência de curto circuíto ($P_{short-circuit}$), e a potência estática (P_{static}). A potência dinâmica (chaveamento) representa o componente de consumo relacionado a atividade da célula lógica, isto é, consumo devido a carga e descarga das capacitâncias dos circuitos, a potência de curto circuito está ligada ao caminho direto formado entre VDD e GND quando ambos os transístores N e P-MOS estão momentaneamente ativos durante o chaveamento da célula, este efeito é minimizado através do correto *scaling* dos transístores, porém nunca é eliminado. A potência estática está diretamente ligada a tecnologia do processo de fabricação envolvido, isto é, provém das correntes de fuga e correntes *subtreshold* (WESTE, 1994).

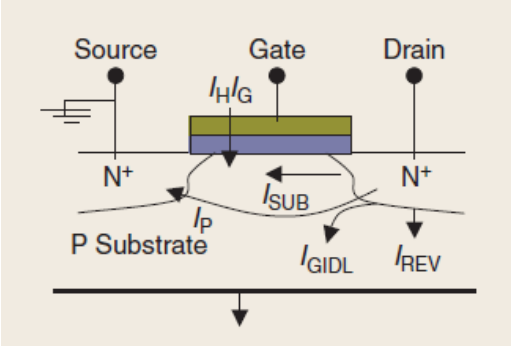
3.2.1 Potência Estática

A potência estática em tecnologias *submicron* representam a menor fatia do consumo elétrico de um circuito VLSI, e está ligada diretamente à tecnologia e aos dispositivos MOS.

Porém, devido ao *downscaling* agressivo havido nas últimas décadas, esta fonte de consumo passa a representar uma porcentagem crescente (EKEKWE, 2010), representando uma fatia considerável das fontes de consumo (SARKAR et al, 2016)

O consumo estático é basicamente composto por corrente de fuga presente no circuito, que pode ser modelado por cinco fontes principais: (1) Corrente de fuga por tunelamento no oxido do porta (*gate*) (I_G), (2) corrente *subthreshold* (I_{SUB}), (3) corrente de polarização reversa (I_{REV}) em diodos, (4) corrente de fuga através do dreno induzida pelo porta (I_{GIDL}) e (5) corrente de porta por injeção (I_H) (EKEKWE, 2010).

Figura 3.1 – Fontes de consumo estático



Fonte – EKEKWE, 2010

A fonte (1), I_G, agrava-se conforme o processo de fabricação atinge escalas cada vez menores. O óxido de porta torna-se extremamente delgado, sendo formado por uma camada cada vez mais fina, proporcionando o tunelamento de cargas, pois devido a imperfeições no óxido as cargas conseguem saltar a barreira, gerando uma pequena corrente de fuga. A fonte (2), I_{SUB}, é a corrente de dreno-fonte do transistor quando este opera em modo de inversão fraca, ou seja, quando os transistores chaveiam para ON e a fonte de tensão do porta está abaixo do limiar. A fonte (3), I_{REV}, trata-se da corrente de polarização reversa nos diodos parasitas entre dreno/fonte e substrato, sendo que a magnitude desta corrente depende da área da difusão do dreno/fonte, da densidade de corrente de saturação do diodo, que pode ser determinada com base nas concentrações de dopantes. A fonte (4), I_{GIDL}, ocorre pois à proximidade entre dreno e fonte é cada vez maior devido ao *downscaling*, resultando em tunelamento de cargas. A camada de óxido de porta cada vez mais fina e tensões elevadas de Vdd contribuem para o aumento deste efeito, sendo o controle da concentração de dopantes no dreno o melhor caminho para redução da I_{GIDL}. A fonte (5), I_H, ocorre devido ao grande campo elétrico próximo a interface entre o silício e o oxido de silício ($Si \leftrightarrow SiO_2$), que faz

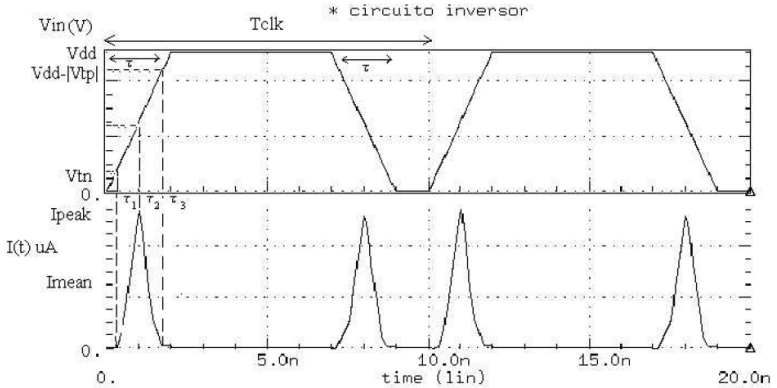
com que elétrons ou lacunas ganhem energia suficiente para saltar a barreira de potencial entre o isolante e a superfície do semicondutor. Uma vez que os elétrons possuem massa efetiva menor, é mais comum que estes ganhem energia suficiente para saltar, causando um efeito chamado “injeção”, que produz uma corrente de fuga para o substrato. Um caminho para reduzir esta corrente de fuga é a redução da tensão de Vdd (EKEKWE, 2010).

Desta forma, a fonte de consumo estático está diretamente ligada a tecnologia, sendo controlada pelo ajuste de dopante, qualidade do oxido, intensidade de campo elétrico e principalmente, pela proximidade entre porta, fonte e dreno. Conforme a tecnologia CMOS continua em seu processo de *downscaling*, o consumo de energia estática tende a aumentar devido aos efeitos de tunelamento, efeito de canal curto e capacitâncias de acoplamento, aumentando sua contribuição para a potência média consumida pelo *chip.*

3.2.2 Potência de curto-circuito

O consumo de potência por curto-circuito acontece quando os transístores P e NMOS estão conduzindo simultaneamente. Este fato ocorre devido a uma transição de sinal na entrada da célula lógica com tempo de descida e subida não zeros (COSTA, 2002). Contribui ainda a diferença de velocidade de comutação existente entre os transístores P e NMOS devido a concentração de dopante diferir, sendo necessário ajuste de escala entre os transístores de *Pull-UP* e *Pull-DOWN*, mas que não resolvem completamente. O Gráfico 3.1 apresenta variação da corrente fornecida a um inversor CMOS durante a transição de estado lógico.

Gráfico 3.1 – Corrente de curto-circuito em um inversor sem carga na saída



Fonte – COSTA, 2002

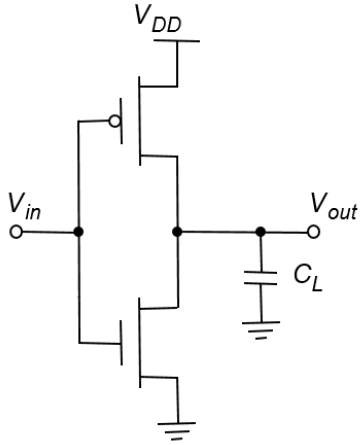
Conforme o Gráfico 3.1, durante a primeira transição *low-to-high*, observa-se que ambos os transístores permanecem ativos (conduzindo) enquanto o sinal na entrada não atinge o limiar (*treshold*) (NMOS) e a diferença da tensão de alimentação e a tensão de limiar (Vdd-Vth) para o PMOS.

Desta forma, é possível afirmar que a potência consumida por curto circuito está fortemente ligada à velocidade de transição do sinal e a própria atividade do circuito. O correto *scaling* entre os transístores P e NMOS, bem como o correto *scaling* das células lógicas em relação ao Fan-IN e Fan-OUT são de grande importância para a redução do tempo de transição dos sinais, e consequentemente da potência consumida por curto-circuito.

3.2.3 Potência Dinâmica

A componente de consumo dinâmica representa, atualmente, uma das maiores fatias do consumo geral do circuito. Esta é relacionada a atividade das células logicas e a carga e descarga das capacitâncias que aparecem nas junções. A carga desta capacitância (C_L) é responsável pelo consumo dinâmico, sendo que a energia drenada da fonte de alimentação para uma transição *low-to-high* é dada pela equação $C_L * {V_{dd}}^2$ (CHANDRAKASAN, 1995). A Figura 3.2 apresenta um circuito inversor com a capacitância C_L em sua saída.

Figura 3.2 – Representação da capacitância CL em uma célula inversora



Fonte – RABAEY, 2002

Conforme observa-se na Figura 3.2, a capacitância C_Lé carregada quando o transistor PMOS conduz corrente da fonte para a saída. Quanto maior C_L, maior a corrente necessária para carregá-la (e consequentemente, maior deverá ser o transístor PMOS).

É importante salientar que o consumo de potência dinâmica não representa um fator de 1 para 1 em relação a frequência de operação do circuito síncrono, sendo regida pela probabilidade α de determinada célula lógica ter sua saída, à qual está associada a caga C_L, transicionando ou não. A equação 3.2 apresenta o consumo de potência dinâmica normalizado através da probabilidade α de chaveamento para uma determinada porta digital (COSTA, 2002).

$$P_{din} = \frac{1}{2}\alpha f C_L * {V_{dd}}^2 \tag{3.2}$$

3.3 Técnicas de "Low Power"

As técnicas de Low Power adotadas visam reduzir ao máximo o consumo elétrico modelado nas fontes de consumo apresentadas no Capítulo 3.2. Estas técnicas podem ser separadas em dois grandes grupos: (1) Clock Gating e Frequency Scaling, que trata diretamente o consumo de potência dinâmica e curto circuito e (2) Power Gating e Voltage Scalling, que trabalham a redução do consumo de potência estática.

3.3.1 *Clock Gating* e *Frequency Scaling*

Uma das principais fontes de consumo elétrico em sistemas VLSI reside na própria atividade do circuito. Quanto maior a atividade, maior o consumo elétrico devido aos ciclos de carga e descarga das capacitâncias existentes nas junções e barramentos de interligação e comunicação. As parcelas de potência consumida por atividade de chaveamento e curto-circuito representam a maior parte do consumo de potência global, representando cerca de 95% deste, sendo 85% por atividade de chaveamento e 10% por corrente de curto-circuito (COSTA, 2002).

As fontes de consumo envolvidas na atividade do circuito são duas a saber: $P_{switching}$ e $P_{short-circuit}$. Em circuitos VLSI síncronos, a atividade de chaveamento está diretamente ligada a mudança dos níveis lógicos devido ao regime de clock adotado. Junto à própria atividade de chaveamento aparece uma fonte de consumo de potência por curto

circuito. Esta fonte, por sua vez, forma um caminho entre Vdd e GND durante a transição de estados dos transístores P e NMOS, que apresentam velocidades diferentes (CHANDRAKASAN, 1995).

Duas técnicas atacam diretamente este problema de modo a reduzir o consumo elétrico: O *Clock Gating* e o *Frequency Scalling*. A primeira técnica, mais clássica, visa desativar toda árvore de *clock* de um determinado circuito quando este não é necessário, coibindo seu chaveamento, reduzindo assim o consumo pelas duas fontes já citadas ($P_{short-circuit}$ e $P_{switching}$). Neste âmbito também deve-se destacar que a própria árvore de clock apresenta grande consumo elétrico devido a capacitância de sua malha e *bufferes* envolvidos. A segunda técnica, tem como objetivo reduzir a frequência de todo ou de parte do circuito, mantendo este ativo, porém reduzindo seu consumo proporcionalmente a redução de atividade. Um circuito específico é responsável por selecionar a fonte de clock ideal para uma determinada atividade, ou mesmo desativar a arvore de clock do circuito após alguns ciclos de inatividade e a ativá-la novamente quando necessário.

3.3.2 *Power Gating* e *Voltage Scaling*

As fontes de consumo mais difíceis de serem reduzidas são aquelas ligadas as limitações da própria tecnologia. Uma vez que um dos componentes dominantes do consumo elétrico em circuitos é proporcional ao quadrado da fonte de tensão que alimenta este circuito, é correto afirmar que, a redução da tensão principal ou mesmo o desligamento do circuito é um bom caminho para redução do consumo (CHANDRAKASAN, 1995).

As técnicas de *Power Gating* e *Voltage Scaling* foram criadas de modo a controlar as fontes de consumos estático e dinâmico, ambas ligadas diretamente ao processo de fabricação e diretamente influenciadas pelo nível de tensão aplicada à Vdd. Em processos estado-da-arte, estes componentes de consumo são cada vez menores, porém ainda representam uma porcentagem significativa do consumo nos circuitos digitais.

A técnica de *Power Gating* consiste em desativar todo o circuito ou parte deste quando o mesmo não é necessário, eliminando completamente as componentes de consumo estático e dinâmico, uma vez que não ocorre carga do capacitor C_L e consumo pelas fontes estáticas descritas em 3.2.1. Como técnica complementar, surge o *Voltage Scaling*, técnica esta que foca em reduzir a tensão de alimentação do circuito, alterando a fonte que alimenta o *Power Rail* de todo um módulo. Conforme a atividade do circuito, o dispositivo tem sua

tensão de operação reduzida. Esta técnica tem como vantagem o reduzido tempo de recarga do circuito, uma vez que o esforço para carregar todas as capacitâncias e voltar a operação normal tende a ser maior caso o mesmo seja completamente desligado. Entretanto, a redução da tensão de Vdd traz como efeito colateral o aumento de tempo de transição entre os níveis lógicos, fazendo com que seja necessária a redução do frequência de operação neste circuito para que não ocorra violação de *timing* nos caminhos críticos. Sendo assim, a técnica de *Voltage Scaling* deve ser adotada em conjunto com a técnica de *frequency scaling* de forma a garantir a correta operação do circuito digital em questão.

3.3.3 Considerações acerca de baixo consumo elétrico

Algumas considerações são importantes de serem realizadas acerca das técnicas de *Low Power* apresentadas. O uso de apenas uma das técnicas pode reduzir significativamente o consumo de determinado circuito em determinada circunstância, porém, apenas a adoção de modo eficiente de todas as quatro técnicas apresentadas, trará bons resultados em circuitos de propósito geral.

É de grande importância também a correta aplicação das técnicas e avaliação de seu impacto ao desempenho geral do dispositivo. Técnicas como *Power Gating* tendem a aumentar em demasiado a latência do circuito, uma vez que após desligado, as capacitâncias existentes nas portas lógicas levam determinado tempo para serem carregadas e ter os níveis lógicos restabelecidos, acarretando aumento de consumo por este breve período de tempo (bem como demora para retorno a operação normal). Devido a isto, circuitos estado-da-arte combinam as técnicas de *frequency scaling* e *voltage-scaling*, relacionando diversos passos (*steps*) de tensão e clock antes do total desligamento do circuito e árvore de clock. A adoção desta metodologia, apresentando diversos passos, representa o estado da arte em relação a *low power*, trazendo maior desempenho global ao circuito aliado a redução de consumo. Desta forma, os níveis de gerenciamento de energia são alterados conforme a estatística de uso do circuito envolvido, variando de tensão máxima com frequência máxima, até total desligamento da alimentação (*Power Gating*) e desligamento da árvore de clock (*Clock Gating*).

3.3.4 Técnica de baixo consumo elétrico em FPGAs

A adoção de FPGAs na indústria normalmente ocorre quando é requerido o desenvolvimento de um dispositivo extremamente específico, cuja fabricação não atinge massa crítica suficiente para o desenvolvimento de um ASIC, quando um processador não atinge os requisitos do projeto ou simplesmente quando este necessita de constantes atualizações e reconfigurações lógicas, aliada a desempenho. Desta forma, a preocupação com o consumo elétrico também chegou aos FPGAs, pois estes deixaram de ser meramente de uso para prototipagem, sendo adotados em uma grande gama de equipamentos, desde um Controlador Lógico-Programável (CLP) até equipamentos sofisticados de coleta de dados e satélites, sistemas estes que estão em ambientes isolados e demandam uma arquitetura mais otimizada em relação ao consumo elétrico.

No entanto, a aplicação de técnicas de *Low Power* e redução de área em FPGAs representa um grande desafio para os fabricantes destes dispositivos, uma vez que grande parte do consumo elétrico e área ocupada nestes se dá através da malha de interligação entre elementos lógicos configuráveis. Não obstante, fabricantes tem investido em técnicas que permitem a seleção de mais de uma fonte de alimentação, proporcionando um *Voltage Scaling* (mesmo que rudimentar), e virtualmente, todos os FPGAs modernos permitem a adoção de técnicas de *clock gating* e *frequency scaling*.

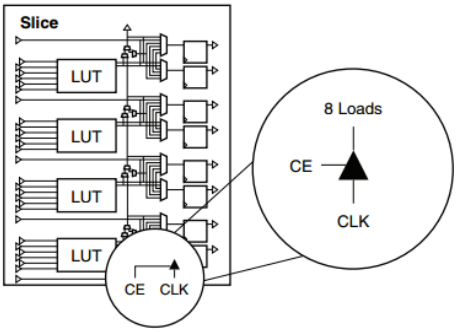
3.3.4.1 Clock gating

A técnica de *clock gating* pode ser aplicada em basicamente qualquer FPGA moderno, sendo adotado localmente (em um único módulo) ou globalmente, desligando todo um segmento da árvore de clock. A redução do consumo pode chegar a até 30% ao adotar a técnica, sendo relatados ganhos substanciais a depender da metodologia aplicada (COMPANHIA1d, 2016)(HUDA, 2009).

O controle do *clock gating* pode ser efetuado de forma transparente, adotando o pino de *enable* presente nos módulos descritos, que é roteado através de malha de roteamento de propósito geral, sendo esta uma abordagem bastante comum para aplicação de *clock gating* em uma região específica, isto é, em um módulo. Porém, dispositivos modernos permitem a configuração de *clock gating* diretamente na árvore de clock, possibilitando que todo um ramo seja desativado conforme a atividade do circuito (HUDA, 2009)

Basicamente, as células lógicas apresentam um pino de *Clock Enable* (CE), responsável por ativar ou desativar o *clock* na entrada de um conjunto de Flip-Flops que compõe um *slice*. A Figura 3.3 apresenta a estrutura de um *slice* e o pino CE.

Figura 3.3 – Pino CE em um Slice



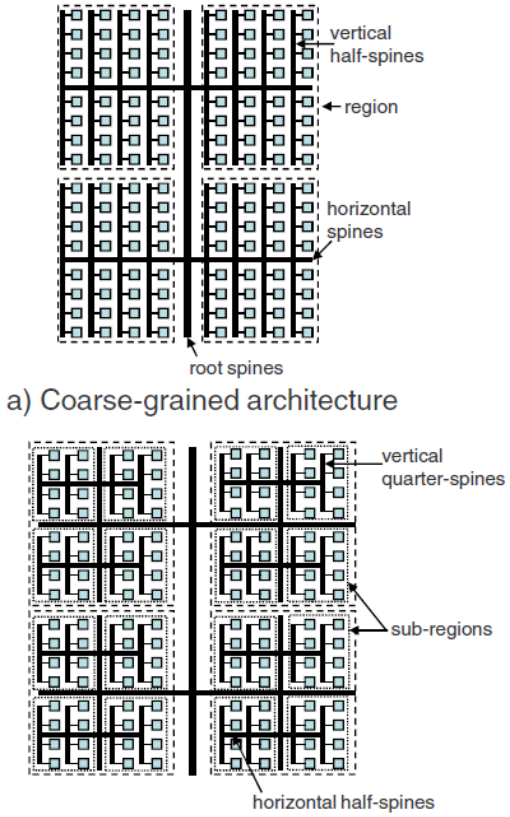
Fonte – COMPANHIA1d, 2016

Conforme a Figura 3.3, cada *slice* apresenta um pino CE, responsável por habilitar ou desabilitar o sinal de *clock* na árvore de *clock* interna, onde são ligados 8 registradores.

O roteamento da árvore de *clock* ainda segue uma orientação, através de uma topologia muito semelhante a topologia H. Desta forma, a árvore de *clock* pode ser segmentada em regiões, que por sua vez são divididas em colunas, sub-regiões e sub-colunas. A

Figura 3.4 apresenta a arquitetura da árvore de *clock* adotada em FPGAs.

Figura 3.4 – Arquitetura de árvore de Clock



Fonte – HUDA, 2009

Conforme a

Figura 3.4 (a), a árvore de *clock* apresenta uma coluna central (root spine), de onde derivam malhas de *clock* horizontais que alimentam uma região com o sinal. Esta região, ainda pode ser subdivida em sub-regiões, conforme a

Figura 3.4 (b), que são ligadas a malha de *clock* horizontal através de uma segunda malha vertical (sub-coluna) (HUDA, 2009).

Nos FPGAs ”Companhia 1”[10] Serie 7 a separação de regiões e sub-regiões é realizada através de 2 *buffers* de *clock* principais (BUFFMR/BUFFMRCE), que separam a árvore de *clock* em duas regiões, e mais 12 *buffers* que separam em sub-regiões (BUFFH/BUFFHCE). Ambos os *buffers* possuem pino *Clock Enable* (CE), responsável por

[10] Nome da Companhia 1 é omitido por acordo de confidencialidade

habilitar ou desabilitar o *buffer*, tornando-o um chave (ou *gate*), permitindo que o circuito seja desativado de modo a reduzir o consumo elétrico (COMPANHIA1e, 2017).

Desta forma, os FPGAs que adotam esta tecnologia, onde podem ser citados os FPGAs Virtex 5/6 e Stratix III, apresentam um *gate* entre cada sub-coluna, região e sub-região, permitindo um melhor aproveitamento de técnicas de *clock gating*.

A adoção desta topologia apresenta grande ganho em relação ao *clock gating* local apresentado na Figura 3.3, uma vez que não apenas o registrador é desabilitado, mas como toda a malha de clock em uma determinada região, reduzindo o consumo elétrico proveniente da própria malha e suas capacitâncias de carga (HUDA, 2009).

3.3.4.2 Frequency Scaling

Atualmente, todo FPGA moderno pode adotar algum tipo de técnica de *Dynamic Frequency Scaling* (DFS) – ajuste dinâmico de frequência – de modo a reduzir a dissipação térmica e consumo elétrico. Esta técnica aliada a técnica de *Voltage Scaling*, apresenta grande eficiência para redução do consumo elétrico (BELDACHI, 2014)(NUNEZ-YANEZ, 2016).

Uma estrutura específica chamada DFS *Unit* (Unidade DFS) é responsável por avaliar o *clock* máximo cujo circuito pode atingir, realizando uma análise através de teste de *timing* de modo a avaliar se há alguma violação. O ajuste da frequência é efetuado de modo dinâmico, podendo adotar diversas metodologias, desde o ajuste de configuração de um oscilador programável externo (BELDACHI, 2014) até mesmo a seleção de saídas em um PLL embarcado no próprio FPGA (como adotado hoje pela indústria em geral). Ocorrendo violação de *timing*, o *clock* é reduzido ou a tensão do circuito é aumentada – caso esteja sendo adotada *Voltage Scaling* – de modo a garantir performance.

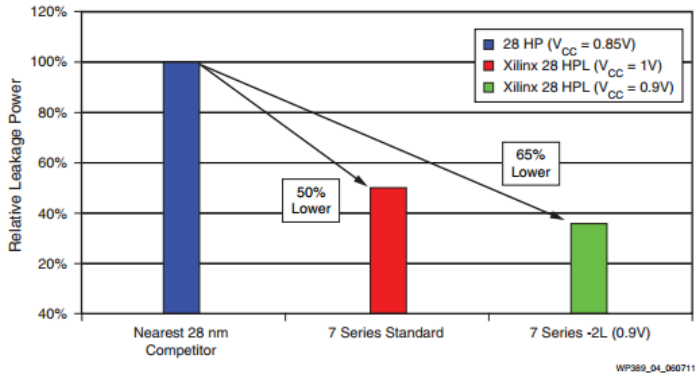
Sendo assim, esta política visa reduzir o *clock* de operação de todo circuito (ou parte dele), obtendo redução no consumo de potência dinâmica devido a carga/descarga das capacitâncias das células lógicas, bem como, da árvore de *clock*. É importante salientar que a unidade DFS deve ser dotada de inteligência para não só avaliar o desempenho dos caminhos lógicos do circuito, mas também à avaliar a demanda por processamento, aumentando o *clock* quando sinalizado através do escalonador.

3.3.4.3 Voltage Scaling

O uso de técnicas de *Voltage Scaling* exige que o FPGA suporte esta função por padrão. Nem todos os FPGAs permitem a alteração da tensão, e mesmo aqueles que o permitem, a mudança da tensão de VDD não ultrapassa os 10%, porém, os ganhos no consumo podem ser interessantes.

Neste quesito, podem ser citados os FPGAs da série Virtex 7 (Companhia1), onde alguns modelos suportam *Voltage Scaling*, apresentando bons ganhos no consumo de potência estática. O Gráfico 3.2 apresenta um comparativo entre FPGAs de 28 nanômetros utilizando diferentes tensões.

Gráfico 3.2 – Ganhos no consumo de potência estática com o uso de *Voltage Scaling*



Fonte – COMPANHIA1a, 2015

O principal ponto de comparação no Gráfico 3.2, diz respeito a diferença de consumo entre os FPGAs *"Companhia 1"*[11] *28 HPL*, operando com tensão de 1V e 0.9V. A diferença de consumo estático entre estes dois dispositivos chega a 15% ao efetuar a troca da fonte de alimentação de 1V para 0.9V.

Desta forma, a adoção desta técnica depende de suporte por parte do FPGA, sendo necessária a identificação da metodologia adotada pelo fabricante para alteração da tensão de alimentação. Nos FPGAs "Companhia 1" Série 7, é adotado o *Voltage Identification Bit* (VID), como uma forma para implementação de *Adaptative Voltage Scaling* (AVS). Basicamente, o FPGA possui um bit que informa a atual situação do dispositivo, se este pode receber uma tensão de operação menor ou se demanda maior tensão. Quando VID retorna o

[11] Nome da Companhia 1 é omitido por acordo de confidencialidade

valor “1”, o FPGA pode operar com tensão de 0,9V, desta forma, um circuito dedicado a AVS, deve sinalizar a fonte de tensão externa para reduzir de 1V para 0,9V. Caso VID apresente valor “0”, o circuito AVS deve sinalizar à fonte que a tensão deve retornar a 1V, devido a demanda por performance (COMPANHIA1b, 2012).

3.3.4.4 Power Gating

A técnica de *Power Gating* figura entre umas das mais comuns e utilizadas para redução de consumo estático em ASICs, porém, em FPGAs ainda não se tornou muito comum. O fato ocorre devido à dificuldade de realizar o roteamento das malhas de alimentação de forma dinâmica através de um *transmission gate* que permita desabilitar ou não um determinado CLB, *slice* ou região.

Os dispositivos da série 7 da fabricante ”Companhia 1”[12] apresentam uma forma rudimentar de *Power Gating* , sendo este “não dinâmico”, ou seja, a configuração das células que são ativadas é realizada durante a síntese lógica e configuração do *chip*, não podendo ser alterada de forma dinâmica durante a operação dos circuitos de modo a levar o CI para um modo *sleep* . Por muitas gerações a ”Companhia 1” tem adotado este tipo de metodologia, permitindo que PLLs, DCMs, Transeivers e I/Os sejam desativados quando estes não são instanciados no projeto, reduzindo o consumo estático. Na série 7 de seus dispositivos, a ”Companhia 1” passa a adotar a mesma metodologia para as BRAMs, que constituem ao menos 30% do consumo estático do FPGA. Porém, o *Power Gating* ainda não é dinâmico (COMPANHIA1e, 2015).

O motivo por detrás de o *Dynamic Power Gating* (DPG) ainda não ser uma realidade nos FPGAs diz respeito unicamente a complexidade que a adição desta funcionalidade traz a estrutura do FPGA. A adição de *Power Gating* granular dentro de FPGAs modernos pode representar em um aumento de 100% em sua área, o que se traduz em aumento de suas vias de interligação, causando aumento de capacitâncias parasitas e consequentemente, aumento de consumo ao invés de redução com a adoção da técnica (BSOUL, 2010).

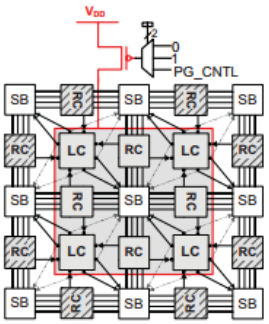
Outras metodologias foram propostas ao longo dos últimos anos, onde cita-se uma técnica de *Power Garting* para uma arquitetura similar ao Spartan-3. Esta arquitetura apresenta um *gate* que realiza a conexão entre uma *tile* e o *power rail* adjacente. Esta *tile* é

---

[12] Nome da Companhia 1 é omitido por acordo de confidencialidade

formada por um CLB, sendo associada a uma matriz de interligação, que a conecta às células vizinhas. Esta arquitetura proposta suporta um modo *sleep*, através do envio de um sinal específico, permitindo configurar regiões específicas para terem o fornecimento elétrico desativado de forma dinâmica. Este sinal (de *sleep*) é gerenciado por uma unidade de controle, que pode estar presente dentro do próprio FPGA ou ser externa (BSOUL, 2010). A Figura 3.5, apresenta o exemplo de fornecimento de tensão para uma região, através de um *power gate.*

Figura 3.5 – Exemplo de uma região com Dynamic Power Gating



Fonte – BSOUL, 2010

Conforme a Figura 3.5, uma determinada região possui um *power-ring* que é alimentado através de um transistor ligado a Vdd. Este transistor, por sua vez, recebe um sinal de modo a permitir ou não o fornecimento de energia à região, proporcionando a função de DPG ao FPGA (BSOUL, 2010).

Até a presente data, nenhuma das técnicas descritas passou a ser adotada por fabricantes de FPGA, estes se restringindo apenas a adoção parcial de técnicas de Power Gating, que proporcionam apenas a desativação das células não instanciadas de modo a coibir o consumo elétrico estático desnecessário. Sendo assim, a observância do Power Gate durante etapas de HLS está apenas ligada a redução de área, isto é, a otimização de recursos de modo a reduzir a quantidade de células lógicas instanciadas no FPGA.

4 EXPLORAÇÃO ARQUITETURAL DO ESPAÇO DE PROJETO

O uso de ferramentas de Síntese Lógica e HLS representam um grande ganho de desempenho ao desenvolver circuitos VLSI, porém, o *setup* e inserção de *pragmas* tendem a direcionar as ferramentas de modo a apresentar diferentes resultados. Cada diretriz adicionada ou a alteração da ordem do processo de tradução das funções C++ para o equivalente em HDL e RTL proporciona diferentes resultados, que pode vir a trazer melhorias em uma área, como redução de consumo elétrico, em detrimento de outra, como desempenho e *clock* máximo. Desta forma, os resultados tendem a depender de um bom *setup* da ferramenta HLS e a correta escolha dos *pragmas* e sequência do processo de conversão (*binding*, *schedulling* e *allocation*).

Sendo assim, o *design* final obtido estaria intimamente ligado a experiência do projetista, sendo um grande desafio mesmo para os mais experientes. Um projeto com execução de 15 *tasks* em uma plataforma com 4 processadores e 4 domínios de tensão apresentará mais de 100.000 alternativas viáveis (OLIVEIRA, 2013, p 45, apud SEMICONDUCTORS, 2011). Uma forma para localização do melhor *setup* para um determinado projeto reside na adoção da técnica de *Design Space Exploration* (DSE) – exploração arquitetural do espaço de projeto –, que reside em realizar testes iterativos com base em diretivas estabelecidas. Através de uma ferramenta e scripts automatizados, é possível testar cada uma das possibilidades e selecionar os melhores resultados obtendo o *setup* ideal para o projeto testado (JACKSON, 2010). Sendo assim, o uso de ferramentas e scripts automatizados garantem a troca de "horas máquina" por "horas homem", ou seja, configura-se um computador (com um script ou ferramenta para DSE) que irá realizar esta exploração automaticamente e libera-se o projetista para realização de outras tarefas.

Consequentemente, segundo Oliveira (apud 2013, apud KEUTZER et al., 2000, p 45), a técnica de DSE apresenta três etapas a saber:

1. *Design Space Exploration* (DSE): Corresponde ao processo sistemático de geração e avaliação de alternativas de *design* para localizar a melhor solução;
2. Mapeamento: Corresponde a um conjunto de soluções mapeadas, refinadas e abstraídas de um conjunto de decisões. Este conjunto de soluções depende do grau de liberdade dado a ferramenta;
3. Design Space: Corresponde ao conjunto de todos os *designs* em potencial que representam uma solução final ao problema;

Conforme as etapas apresentadas, DSE tem como objetivo realizar testes iterativos com base nas diretivas definidas pelo projetista, de modo a organizar o fluxo de síntese. A aplicação do conceito em HLS reside na inserção de *pragmas* e alteração da sequência do fluxo HLS (quando permitido pela ferramenta) de modo a obter diferentes resultados (etapa 1). Cada iteração realizada é mapeada relacionando *pragmas* e sequência de síntese com o resultado relatado pela ferramenta HLS, gerando um mapa de possíveis soluções (etapa 2). Ao final do processo iterativo, tendo sido mapeadas todas (ou quantas forem necessárias) soluções, os melhores resultados são selecionados e analisados, conforme critérios definidos (etapa 3). Qualquer linguagem de programação pode ser utilizada para criação de scripts e ferramentas para realizar DSE, onde as mais comuns são o SystemC, C++ e ferramenta Simulink.

4.1 Testes com DSE

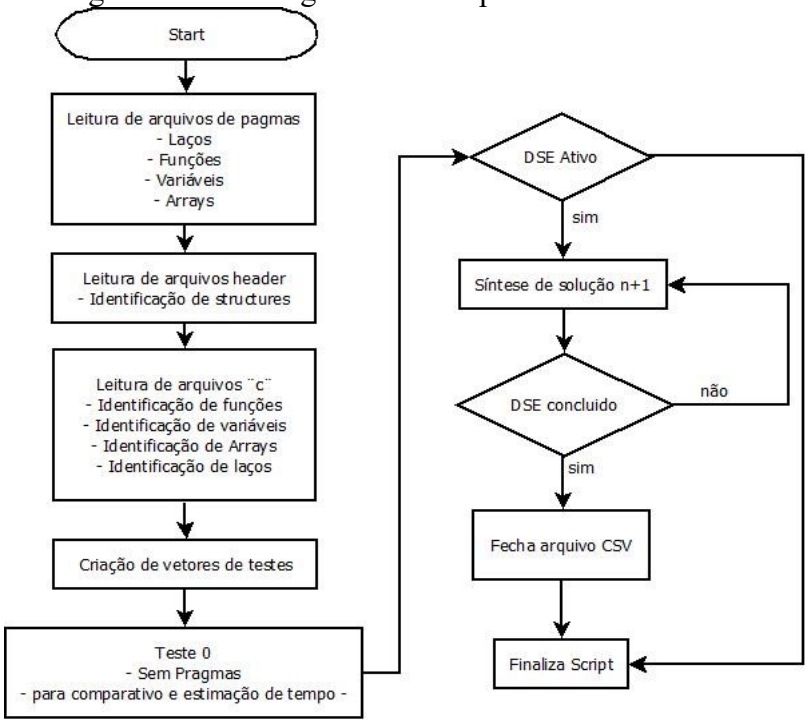
Para este estudo, foi adotado o Python em sua versão 2.6 como linguagem de programação/script para realizar o processo de DSE (para ferramenta “FERRAMENTA 1” rodando sob Windows 10 e “Ferramenta 2 v1” sob Ubuntu Linux). Basicamente, o processo DSE adotado visa realizar o teste de soluções possíveis com base em uma lista de *pragmas* predefinidas pelo projetista, que sejam suportadas pela ferramenta HLS. Seguindo esta abordagem, duas metodologias distintas podem ser seguidas: Combinar todas as *pragmas* entre si ou combinar as *pragmas* em grupos.

Ao realizar a combinação de todas as *pragmas* entre si, o espaço de projeto apresenta um crescimento proporcional ao número de *pragmas* e elementos identificados no código comportamental. Isto é, se temos 2 *pragmas* para serem testados e um único laço *FOR*, teremos duas configurações possíveis, porém, se o código apresentar 2 laços *FOR*, o número de combinações já sobe para 4.

Deste modo, de forma a reduzir o espaço de projeto, as *pragmas* são combinadas em quatro grupos distintos a saber: Laços, Funções, Variáveis e *Arrays* (vetores). Adotando esta abordagem, garantimos que independentemente da quantidade de elementos presente no código, a quantidade de soluções somente crescerá mediante a adição de novas *pragmas* a serem testadas. Ou seja, em um código contendo 1, 2 ou “n” laços, a quantidade de combinações possíveis será a quantidade de *pragmas* selecionadas para o grupo “laços”. Como resultado, a exploração de 16 *pragmas* separadas nos 4 grupos, representa um universo

de 1440 soluções possíveis. Uma vez que utiliza-se a abordagem por grupos, o conjunto de soluções somente incrementará mediante a adição de mais uma *pragma*, independente do código a ser analisado. A Figura 4.1 apresenta o fluxograma do script desenvolvido.

Figura 4.1 – Fluxograma do Script DSE desenvolvido



Fonte – O Autor

Conforme apresentado na Figura 4.1, o script desenvolvido inicia realizando a leitura de uma lista de *pragmas* suportados – ou que devem ser testados – separados em categorias. Na sequência, é realizada a leitura do(s) arquivo(s) *header* em busca de estruturas e predefinições que devem ser anexadas a lista de *arrays* ou *variáveis*. Concluída esta leitura, a lista interna de predefinições é atualizada e a busca por funções, variáveis, *arrays* e laços é realizada em todos os arquivos fonte presentes no diretório. Estes arquivos são processados, tendo os laços identificados através de um *label* (suportado apenas no "FERRAMENTA 1") para futura indexação em arquivo TCL. Os arquivos processados são armazenados em um diretório separado, de modo a não ser realizada qualquer alteração nos arquivos originais. Concluído este processamento, o script realiza a criação dos vetores de teste, ou seja, são gerados todos os scripts TCL contendo as diretivas (*pragmas*) de projeto. Na sequência, é realizada a "síntese 0", para verificação de desempenho do computador de modo a mensurar o tempo para finalização do DSE. Este tempo é obtido com base na quantidade de testes que serão executados e o tempo médio da síntese lógica realizada no "teste 0". Caso o script esteja

configurado para realizar o DSE, o processo de teste iterativo tem início, onde os resultados de cada síntese realizada mediante um conjunto de *pragmas* são selecionados e armazenados em arquivo CSV. Ao concluir o teste, o arquivo CSV é fechado e o script encerrado.

O propósito do uso deste script é de localizar o melhor *set* de *pragmas* para a síntese de um determinado circuito, relacionando a área ocupada no arquivo CSV para posterior consulta e análise. Espera-se que circuitos com menor área consumam menos energia (menos instâncias ativas, menor consumo). Também é esperado que o uso de estruturas aceleradoras ou otimizadas, tais como DSPs e memórias RAM integradas ao FPGA tragam incremento de desempenho aos circuitos, e que estas soluções apresentem um melhor rendimento.

5 EXPERIMENTOS COM HLS

Diferentes ferramentas HLS devem apresentar diferentes resultados para um mesmo código convertido. As diferenças intrínsecas entre cada ferramenta e a metodologia adotada para aplicação do fluxo HLS apresenta um resultado diferenciado para cada ferramenta, representando ganhos e perdas em áreas distintas. Deste modo, no presente capítulo são apresentados resultados de testes realizados com as ferramentas abordadas neste documento, bem como, um estudo dos ganhos obtidos em cada ferramenta ao utilizar determinados *pragmas*.

Para estes experimentos, são adotadas três classes de circuitos, com descrições em C++, conforme listadas abaixo:

- ULA 16 bits de ponto fixo.
- Filtro FIR 40 e 120 taps.
- Processador VLIW (rVex)

São coletados como resultados o ciclo mínimo de clock, quantidade de Registradores e unidades Lógicas – LookUp Table (LUT) ou Adaptative Logic Module (ALM) quando FPGAs – , células instanciadas (Standard Cell), quantidade máxima de ciclos de execução, número de unidades DSP e quantidade de memória alocada (RAM/BRAM). De modo a realizar um comparativo mais apurado acerca do desempenho das soluções, bem como, é realizado o cálculo do tempo máximo de execução, conforme equação 5.1.

$$Time_{process} = T_{clock} * Taxa \tag{5.1}$$

Onde T_{clock} é o período de clock, $Taxa$ são os ciclos de execução necessários para entrega de resultado e $Time_{process}$ é o tempo máximo de execução calculado. Assim, a partir da equação 5.1 é possível obter a energia consumida por operação de um determinado dispositivo, conforme demonstrado na equação 5.2.

$$Energia_{op} = \frac{Pot}{Freq} * Taxa = Pot * Time_{process} \quad (5.2)$$

Onde $Energia_{op}$ é a energia (potência dissipada) por operação realizada, Freq trata-se da frequência de operação do circuito.

Ainda é importante destacar que, a depender do sistema operacional e a ferramenta a ser adotada, é necessário efetuar a mudança dos tipos de variáveis utilizadas na descrição. Em sistemas Linux, sob compilador 64 bits, variáveis "*int"* podem ser representadas em ponto fixo de 64 ou de 32 bits, a depender da configuração passada ao compilador (e se a compilação tem como alvo 64 ou 32 bits), sendo necessário a mudança do tipo de variável no código e uma atenção especial aos detalhes da arquitetura onde a ferramenta HLS e o compilador estão instalados. Algumas ferramentas possuem compilador próprio, e desta forma não estão suscetíveis a este tipo de variação (como o "Ferramenta 1" e "Ferramenta 3"), sendo necessário uma leitura atenta de seus manuais para verificar o tamanho padrão que será aplicado para cada tipo de variável.

5.1 ULA 16 bits

Uma Unidade Lógico-Aritmética (ULA) é o coração de qualquer processador, sendo responsável por realizar lógica e cálculos. Para os testes, uma ULA de 16 bits e ponto fixo foi descrita em C/C++, de modo a prover as principais operações matemáticas (adição e subtração) e realizar as principais operações lógicas (and, or e xor). A

Figura 5.1, apresenta o código em C++ que descreve a ULA proposta.

Figura 5.1 – Descrição de uma ULA em C++

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void ULA16b(short OP, int A, int B, int *RESULT){


  switch (OP){
            case 1:
                        *RESULT = A+B;
                        break;

            case 2:
                        *RESULT = A-B;
                        break;

            case 3:
                        *RESULT = A&B;
                        break;

            case 4:
                        *RESULT = A|B;
                        break;

            case 5:
                        *RESULT = A^B;
                        break;

            default:
                       *RESULT = 0;
                       break;
            }
};
```

Fonte – O Autor

É importante destacar que a descrição apresentada na

Figura 5.1 leva em consideração uma compilação para barramento base de 8bits, onde variáveis do tipo inteiro apresentam capacidade de 16bits.

5.2 Filtro FIR

O Filtro de Resposta ao Impulso Finita (FIR) pode ser facilmente implementado através de um laço finito, e um acumulador mediante a aplicação do coeficiente referente ao *tap* em questão. O filtro FIR adotado é baseado em uma descrição padronizada disponível para testes na internet (KHAN, 2014). A

Figura 5.2, consta o modelo de filtro FIR adotado.

Figura 5.2 – Descrição de um filtro FIR em C++

```
#include "fir.h"

out_data_t fir_filter (inp_data_t x,   coef_t c[N])
{

  static inp_data_t shift_reg[N];

  acc_t acc = 0;
  acc_t mult;
  out_data_t y;

  Shift_Accum_Loop: for (int i=N-1;i>=0;i--)
  {

    if (i==0)
    {
      shift_reg[0]=x;
    }
    else
    {
                                        shift_reg[i]=shift_reg[i-1];
    }
    mult = shift_reg[i]*c[i];
    acc  = acc + mult;

  }

  y = (out_data_t) acc;

  return y;

}
```

O modelo de filtro apresentado na

Figura 5.2 é utilizado como base para criação dos Filtros FIR adotados nos testes realizados, sendo adicionados os coeficientes dos filtros diretamente ao código (vetor “C”).

É importante destacar que a versão de código adotada para este teste é desenvolvido para aplicação direta no ”Ferramenta 1”, fazendo uso da biblioteca “ap_fixed” para mapeamento do retorno da função diretamente em elementos de memória de ponto fixo, criando assim uma estrutura otimizada. Sendo assim, o código apresentado deve sofrer leve alteração para ser compilado em outras ferramentas HLS (como “Ferramenta 3” e “Ferramenta 2”). Abaixo, consta o arquivo *header* do Filtro FIR, onde pode ser configurado o uso de “AP_FIXED” ou variáveis “int/short” suportadas. A Figura 5.3 apresenta o *header* adotado para a descrição do filtro FIR proposto.

Para comparação, são adotados dois filtros descritos a mão, de 40 e 120 *taps*, desenvolvidos na forma transposta (apêndices A e B), cujas multiplicações de coeficientes são baseados em *Multiple Constant Multiplications* (MCM) e a etapa de soma dos resultados faz uso de somador do tipo *Ripple Carry Adder*. Os filtros utilizados apresentam uma série de otimizações lógicas que garantem ganhos em performance e área ocupada (SOARES, 2016).

Figura 5.3 – Header de um filtro FIR em C++

```
#ifndef _H_FIR_H_
#define _H_FIR_H_

#define N  40

#define SAMPLES 40

#define DB_FIXED_POINT


#ifdef DB_FIXED_POINT
#include "ap_fixed.h"

        typedef ap_fixed<18,2>          coef_t;
        typedef ap_fixed<48,12>         out_data_t;
        typedef ap_fixed<18,2>          inp_data_t;
        typedef ap_fixed<48,12>         acc_t;

#else //FULLY ANSI C*/
         typedef        short int coef_t;
         typedef        int out_data_t;
        typedef         short int inp_data_t;
          typedef       long acc_t;
#endif

out_data_t fir_filter ( inp_data_t x );

#endif // _H_FIR_H_
```

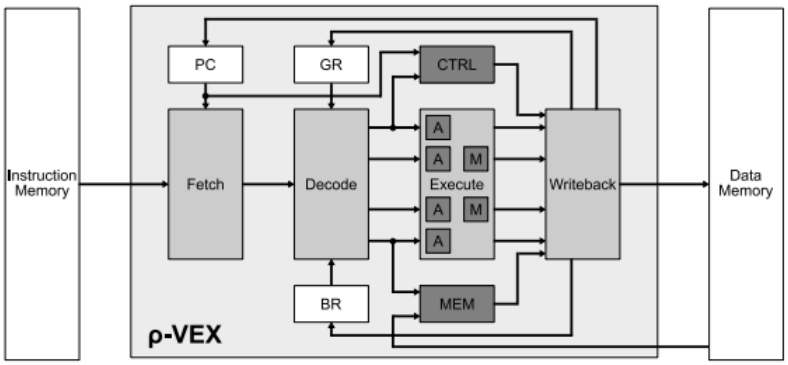
Conforme observa-se na Figura 5.3, basta comentar a linha “#define DB_FIXED_POINT” para deixar de utilizar “AP_FIXED”, compatibilizando o código com outros compiladores.

5.3 Processador VLIW

Neste experimento é adotado um processador do tipo *Very Long Instruction Word* (VLIW) acadêmico, chamado $\rho - VEX$, já testado anteriormente em mesma exploração por Jeferson Santiago. A versão HDL é obtida através da página do projeto (https://github.com/tvanas/r-vex) e a versão C/C++ é desenvolvida por Santiago (SANTIAGO, J. S.).

Este processador possui apenas um core, com capacidade de tratamento de palavras longas (até 128 bits – neste caso), representando uma excelente oportunidade para testes no que diz respeito a otimização de barramentos. Possui todas as estruturas básicas de um processador padrão, sendo composto por uma ULA, unidade de divisão e multiplicação, decodificador de operações, memória interna dentre outras estruturas. A Figura 5.4 apresenta o diagrama em blocos do processador VLIW implementado em HDL e adotado como base para comparação dos resultados obtidos com HLS.

Figura 5.4 – Diagrama em blocos do processador VLIW



Fonte – AS, 2008

5.4 Resultados com ”Ferramenta 1”

Para os testes com o ”Ferramenta 1”, é utilizado o *script* desenvolvido em Python para fins de DSE. Para tanto, foram elencados alguns conjuntos de *pragmas* para esta exploração, conforme apresentado na Tabela 5-1.

Tabela 5-1 – Diretivas adotadas

Loops		
Loop Flatten	set_directive_loop_flatten	Realiza simplificação de loops concatenados
Loop Merge	set_directive_loop_merge	Realiza a união de múltiplos loops e apenas um loop com condicionais
Loop Tripcount	set_directive_loop_tripcount	Utilizado apenas para reports - sem impacto no resultado final da síntese
Funções		
Pipeline enable flush	set_directive_pipeline -enable_flush	Quando circuito desativado, todos os elementos de memória internos são resetados/limpados
Pipeline Off	set_directive_pipeline -off	Desabilita a criação de pipeline para funções
Pipeline	set_directive_pipeline	Habilita criação de pipeline para funções que possuam loops
Latência	set_directive_latency -min 1 -max 10	Configura a concorrência mínima e máxima para FIFOs
Funções – Variáveis e Termos		
Barramento FIFO	set_directive_resource -core FIFO	Realiza mapeamento de IOs através de uma FIFO
Mapeamento de recurso	set_directive_resource	Realiza mapeamento de recursos de hardware (a própria ferramenta realiza a escolha)
Mapeamento de arrays	set_directive_array_map	Realiza mapeamento de arrays para qualquer estrutura (a própria ferramenta realiza a escolha)
Variáveis e Arrays		
Reset de variável	set_directive_reset	Realiza o reset dos elementos de memória
Mapeamento de memória	set_directive_resource -core RAM_1P	Mapeamento para memoria Single Port
Mapeamento de memória	set_directive_resource -core RAM_1P_BRAM	Mapeamento para BlockRam Single Port
Mapeamento de memória	set_directive_resource -core RAM_2P	Mapeamento para memória Dual Port
Mapeamento de memória	set_directive_resource -core RAM_2P_BRAM	Mapeamento para BlockRam Dual Port
Mapeamento de arrays	set_directive_array_map	Realiza mapeamento de arrays para qualquer estrutura (a própria ferramenta realiza a escolha)

Fonte – O Autor

Conforme a Tabela 5-1, foram escolhidas as principais diretivas relacionadas com simplificação de código, onde podemos destacar “*Loop Flatten*”, e todas as demais diretivas relacionadas com mapeamento de recursos e criação de barramentos First In First Out (FIFO) . Destaca-se que ao utilizar FIFOs e configurar um valor de concorrência máxima, a ferramenta passa a realizar *resource sharing,* compartilhando estruturas em comum,

impactando negativamente em seu desempenho, porém, positivamente em consumo de área (e possivelmente potência). Destaca-se também a possibilidade de realizar o mapeamento para aceleradores e *primitives* específicos da tecnologia dos FPGAs ”Companhia 1”[13] (directive_resource), deste modo, a ferramenta faz uso de estruturas DSP, *cores*, *Block* RAMs e memórias Single/Dual Port disponíveis no próprio chip.

Como resultado desta exploração, espera-se obter maior desempenho ao efetuar mapeamento com *pipeline* ativo, concorrência de barramento e recursos e mapeamento de recursos para estruturas e *primitives* ”Companhia 1” (tais como DSP e memórias RAM), bem como, espera-se um melhor aproveitamento de área e menor consumo elétrico com pipeline desativado (impactando drasticamente em performance).

Nos subcapítulos que seguem, é apresentado o melhor resultado obtido em cada teste iterativo, bem como, uma breve análise acerca destes.

5.4.1 ULA 16 bits

Os testes aplicados a ULA de 16bits basicamente residem em mapeamento de elementos de memória e aplicação de *Pipeline*, uma vez que se trata de uma estrutura extremamente simples. A Tabela 5-2 apresenta três resultados, sendo estes o resultado da síntese com as configurações padrões do software, o melhor resultado e o pior resultado (em área) obtido através do DSE para efeito de comparação.

Tabela 5-2 – Resultado DSE para ULA16 bits de ponto fixo – “Ferramenta 1”

Soluções Finais							
SOLUÇÃO	BRAM_18K	DSP48E	FF	LUT	Relógio (ns)	Latência	Taxa
default	0	4	0	241	2,70	1	1
Melhor HLS	0	4	0	241	2,70	1	1

Fonte – O Autor

Conforme a Tabela 5-2, observa-se que em descrições mais simples, não se obtém ganhos ao utilizar *pragmas* para instruir a ferramenta a realizar otimizações ou instanciar

[13] Nome da Companhia 1 é omitido por acordo de confidencialidade

determinados elementos de hardware. Como resultado do DSE, observou-se que mais de 50% das soluções testadas retornam o mesmo período de clock e ocupação de área, não representando, deste modo, qualquer ganho tangível. Também é observado que mesmo as soluções de menor rendimento (em área) apresentam pequena perda, porém, sem apresentar quaisquer outros ganhos. Isto ocorre principalmente devido a simplicidade do circuito, onde os ganhos por otimização são pequenos e a adição de determinados elementos (como elementos de memória) causam aumento no circuito que não garantem ganhos de performance. A Figura 5.5, apresenta as *pragmas* da solução número 10.

Figura 5.5 – Solução 10 para ULA16 bits - “Ferramenta 1”

```
set_directive_pipeline -enable_flush "ULA16b"
set_directive_resource -core RAM_1P_LUTRAM "ULA16b" A
set_directive_resource -core RAM_1P_LUTRAM "ULA16b" B
set_directive_resource -core RAM_1P_LUTRAM "ULA16b" *RESULT
```

Fonte – O Autor

A Figura 5.6, apresenta as *pragmas* da solução de número 60, que proporciona maior consumo de área em relação as demais soluções obtidas.

Figura 5.6 – Solução 60 para ULA16 bits - “Ferramenta 1”

```
set_directive_latency -min 1 -max 10 "ULA16b"
set_directive_resource -core RAM_2P_BRAM "ULA16b" A
set_directive_resource -core RAM_2P_BRAM "ULA16b" B
set_directive_resource -core RAM_2P_BRAM "ULA16b" *RESULT
```

Fonte – O Autor

Ao comparar os *pragmas* das soluções de número 10 e 60 é possível observar a adoção de elementos de memória e configuração para *pipeline* e concorrência máxima de barramento. A solução de número 60 apresenta ocupação de área ligeiramente superior, estando diretamente relacionada com a quantidade de elementos lógicos necessários para o controle de memórias Dual Port (2P_BRAM) e o instanciamento destas no FPGA. Configurações ou *pragmas* que forcem a criação de *pipeline* ou configuração de concorrência de barramento não representam qualquer ganho ou perda neste tipo de circuito devido a sua simplicidade.

5.4.2 Filtro FIR

Duas descrições de Filtros FIR foram alvo de DSE através do “Ferramenta 1”, de modo a realizar um comparativo acerca da melhor solução entre ambos os filtros. Espera-se que para este tipo de descrição, a melhor solução faça uso de DSPs e elementos de memória RAM tais como BRAM/LUTRAM, bem como de processo de *Loop Unroling* ou *flattening*, processos estes que realizam uma simplificação de loops fazendo com que parte do processo ocorra em um mesmo ciclo, de modo a reduzir as estruturas de controle necessárias tornando o circuito sintetizado menor e mais veloz (KNIJNENBURG, 1998).

Uma vez que todos os filtros aqui implementados fazem uso de uma mesma descrição base, a quantidade de soluções possíveis encontrada pelo *script* é a mesma, sendo 1440 soluções a serem testadas. O tempo para conclusão do processo de DSE em um computador Core i7 7700hq foi de 327 minutos para o filtro de 40 taps e 338 minutos para o filtro de 120 taps.

No entanto, as soluções encontradas podem apresentar leve variação devido a possibilidade de otimizações possíveis mediante ao aumento do *loop* que realiza a sequência de cálculos iterativos responsáveis pela filtragem. Quanto maior a quantidade de *taps*, maior o *loop*, proporcionando maiores ganhos em otimização, ou seja, a área ocupada pelo circuito não deve apresentar um crescimento geométrico. A Tabela 5-3 apresenta os resultados do DSE obtidos para o filtro de 40 *taps*.

Tabela 5-3 – Resultado DSE para FIR de 40 taps - ”Ferramenta 1”

SOLUÇÃO	BRAM_18K	DSP48E	FF	LUT	Relógio (ns)	Latência	Taxa
default	0	1	226	550	7,18	26	26
367	1	1	158	538	7,18	26	26
723	0	0	1045	2455	8,53	20	1

Fonte – O Autor

Conforme Tabela 5-3, a melhor solução encontrada apresenta ganho marginal no consumo de elementos lógicos (LUTs), situando-se na faixa de 2%, porém, maiores ganhos são obtidos em relação a ocupação de Flip Flop (FF) , com ganhos na faixa dos 30%. Este fato ocorre devido as *pragmas* da solução 327 orientarem a ferramenta no mapeamento de

elementos de memória, deste modo, observa-se o consumo de uma unidade de BRAM. A solução de número 723 representa o melhor resultado obtido com adoção de *pipeline*, representando maior equilíbrio entre desempenho final e ocupação de área. Observa-se que a adoção de *pipeline* representa um incremento de 446% em ocupação de LUTs e 462% em FFs. Este fato ocorre devido a esta descrição necessitar de circuitos específicos para o controle do fluxo lógico e estrutura de *pipeline*, adicionando maior complexidade. Observa-se também que a solução 367 não representa ganhos no ciclo de clock em relação a solução *default*, enquanto a solução 723 (*pipelined*) apresenta perdas na faixa dos 15%, porém, a solução 723 entrega uma nova amostra processada a cada ciclo de clock, com um atraso de 170,6 ns, enquanto a solução 367 entrega uma nova amostra a cada 186,68 ns (após 26 ciclos).

Conforme já descrito, o aumento da quantidade de *taps* em um filtro FIR tende a apresentar um crescimento na área ocupada não proporcional ao aumento destes *taps*. A Tabela 5-4 apresenta os resultados do DSE para o filtro de 120 *taps*.

Tabela 5-4 – Resultado DSE para FIR de 120 *taps* - ”Ferramenta 1”

SOLUTION	BRAM_18K	DSP48E	FF	LUT	Relógio (ns)	Latência	Taxa
0	0	1	183	1071	7.18	66	66
367	1	1	201	1035	7.18	66	66
723	0	0	1212	3021	8.53	60	1

Fonte – O Autor

Conforme a Tabela 5-4, observa-se um ganho de apenas 3,3% na alocação de LUTs e uma perda de 9% na alocação de FFs, bem como, o consumo de uma unidade de BRAM, mantendo o mesmo desempenho (clock e número de ciclos para conclusão do processamento). Este fato ocorre principalmente devido a necessidade de elementos de memória/registradores para indexação dos vetores – que passam a ser convertidos em elementos de BRAM – e a longa cadeia iterativa característica do filtro FIR. A solução de número 723 por sua vez apresenta a implementação de *pipeline*, onde é possível observar perdas de 382% na ocupação de LUTs , 662% na ocupação de FFs e 15% no ciclo de clock. Salienta-se ainda que a solução 723 entrega uma nova amostra processada a cada ciclo de clock, onde o atraso é de 511,8 ns, enquanto a solução 367 necessita de 66 ciclos de clock para entrega de uma amostra processada, isto é, 473,8 ns.

Em comparação as soluções apresentadas na Tabela 5-3 e Tabela 5-4, observa-se que a melhor solução encontrada é a mesma para ambos os filtros. Algo esperado uma vez que a descrição comportamental base para ambas as implementações é a mesma, tendo como diferença apenas a quantidade de iterações para filtragem. A Figura 5.7, apresenta as *pragmas* adotadas na solução de número 367.

Figura 5.7 – Solução 367 para filtros FIR - “Ferramenta 1”

```
set_directive_pipeline -off "fir_filter"
set_directive_resource -core FIFO  "fir_filter" x
set_directive_loop_flatten "fir_filter/FORL_0"
set_directive_reset "fir_filter" acc
set_directive_reset "fir_filter" mult
set_directive_reset "fir_filter" y
set_directive_resource -core RAM_1P_BRAM "fir_filter" shift_reg
set_directive_array_map "fir_filter" c
```

Fonte – O Autor

Observa-se a solução encontrada não faz uso de Pipeline (pipeline -off) e realiza o mapeamento da variável “x” como uma FIFO. Fato este interessante, uma vez que esta variável é passada através da própria função, sendo comumente traduzida como uma *porta*/*pino* de entrada no módulo. Também é realizado um *flatten* no LOOP (FORL_0), processo este responsável por simplificar o loop, e o reset inicial das variáveis “acc”, “mult” e “y”, que passam a ser mapeadas como elementos de memória. Também se destaca o mapeamento da variável “shift_reg”, responsável por acumular os valores calculados iterativamente no filtro em um elemento de Block RAM (BRAM) e o mapeamento do *vetor (array)* “c” como elemento de memória.

A Figura 5.8, apresenta as *pragmas* da solução de número 723, sendo esta a melhor solução com estrutura em *pipeline* encontrada durante o DSE.

Figura 5.8 – Solução 723 para filtros FIR - “Ferramenta 1”

```
set_directive_pipeline "fir_filter"
set_directive_resource -core FIFO  "fir_filter" x
set_directive_loop_flatten "fir_filter/FORL_0"
set_directive_reset "fir_filter" acc
set_directive_reset "fir_filter" mult
set_directive_reset "fir_filter" y
set_directive_array_reshape "fir_filter" shift_reg
set_directive_array_map "fir_filter" c
```

Fonte – O Autor

Observando a solução de número 723 (com pipeline) é possível verificar apenas duas alterações em relação a solução 367: Diretiva para forçar a adoção de *pipeline* e substituição do mapeamento do principal elemento de memória em BRAM para criação de *array* que acaba por ser armazenado diretamente em FFs.

A Tabela 5-5 e a Tabela 5-6 apresentam um comparativo entre o código HDL gerado através da ferramenta ”Ferramenta 1” e código feito a mão – Hand Made (HM) – para os filtros FIR de 40 e 120 *taps*, respectivamente, sintetizados através do ”Ferramenta 1 Backend”.

Tabela 5-5 – Comparativo de resultados para filtro FIR de 40 taps - HLS vs HDL HandMade - ”Ferramenta 1 Backend”

	BRAM	DSPs	FF		LUT		*TIMING*		Potência Total (mW)	Energia por Operação (nW)
			N°	HLS/HM	N°	HLS/HM	Relógio (ns)	Lat./Taxa		
HandMade	0		1274	1	2122	1	23,83	3/1	196	4,67
Melhor HLS	1	1	155	0,12	118	0,05	6,61	26/26	136	23,37
Melhor HLS c/ pipeline	0	0	848	0,66	903	0,42	6,82	20/1	134	0,91

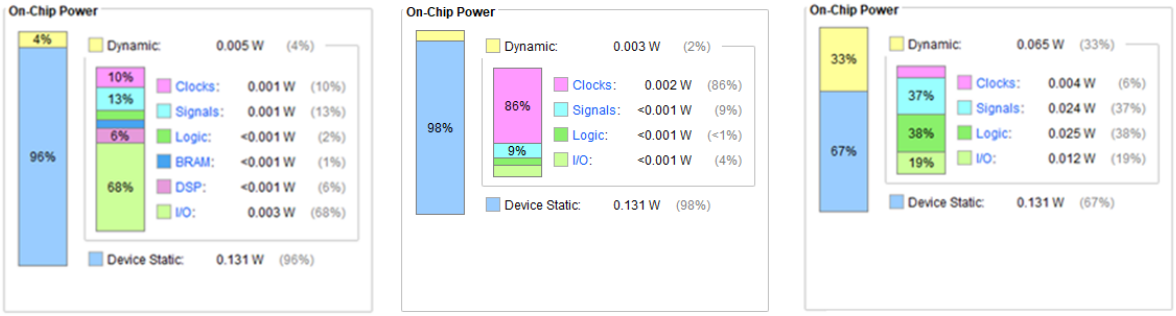
Fonte – O Autor

Conforme a Tabela 5-5, um ganho expressivo em área ocupada e consumo de recursos é obtido com o filtro sintetizado através do fluxo HLS sem *pipeline*, onde se destaca uma redução de 88% em FF e 95% em LUTs. A solução obtida com adoção de *pipeline* apresenta ganhos de 34% em FF, 58% em LUTs, e 71% em clock. No entanto, a solução produzida manualmente apresenta uma estrutura paralelizada que garante um atraso de grupo de apenas 3 ciclos de clock (um ciclo no registro inicial das amostras, um para processar, e um no registro de saída). Isto é, enquanto a solução “HLS” leva 171,9 ns para entregar o resultado de seu processamento, as soluções “*Hand Made*” e “HLS *Pipelined*” entregam uma nova amostra a cada 23,28 ns e 6,82 ns.

Em termos de consumo elétrico, uma analise qualitativa em relação ao consumo elétrico por operação (por amostra, neste caso) é necessária. Assim, aplicando-se as equações 5.1 e 5.2, observa-se que a apesar de obtermos um consumo ligeiramente inferior para a solução “Melhor HLS” em relação a solução HM, esta representa um consumo de 23,37 nW para cada amostra processada contra 4,67 nW da solução HM, porém, ao analisar os resultados obtidos com a “Melhor HLS c/ pipeline”, observamos uma redução significativa na

potência dissipada estimada, representando cerca de 1/5 do observado para a solução HM. A Figura 5.9 apresenta um comparativo entre os resultados informados pela ferramenta de síntese lógica.

Figura 5.9 – Comparativo de potência consumida entre implementações HLS e Hand Made para filtro FIR de 40 taps - ”Ferramenta 1 Backend”



HLS FIR 40 Taps | HLS Pipelined FIR 40 Taps | HandMade FIR 40 Taps

Fonte – O Autor

A análise da Figura 5.9 e os resultados obtidos demonstram que grande parte da redução de potência obtida com as versões HLS do filtro FIR provém de redução da potência dinâmica. Este fato é esperado, uma vez que o filtro modelado manualmente apresenta maior área ocupada, o que se traduz em maior potência dissipada nas interligações (signals), malha de clock (Clocks), lógica (Logic) e finalmente, nos próprios I/Os. Ainda, é importante destacar a potência estática que representa a maior parte da potência consumida em todos os testes e representa o mesmo valor (131mW). Este fato ocorre devido ao FPGA possuir uma estrutura fixa que não possui formas de ser desativada. Deste modo, quanto menor a área ocupada, maior a parcela de consumo estático justamente devido a esta estrutura disponível possuir baixa densidade de uso e permanecer sendo alimentada.

A Tabela 5-6 apresenta os resultados de síntese lógica e implementação obtidos através do ”Ferramenta 1 Backend” para os filtros de 120 *taps* manualmente desenvolvido e obtidos por HLS.

Tabela 5-6 – Comparativo de resultados para filtro FIR de 120 taps - HLS vs HDL Hand Made - ”Ferramenta 1 Backend”

Soluções	BRAM	DSPs	FF		LUT		*TIMING*		POWER (mW)	Energia por Operação (nW)
			N°	HLS/HM	N°	HLS/HM	Relógio (ns)	Lat./Taxa		
HandMade	0	0	3845	1	6094	1	23,07	3/1	269	6,2
Melhor HLS	1	1	198	0,05	194	0,03	6,59	66/66	153	66,54
Melhor HLS c/ pipeline	0	0	1070	0,27	1012	0,166	8,01	59/1	134	1,07

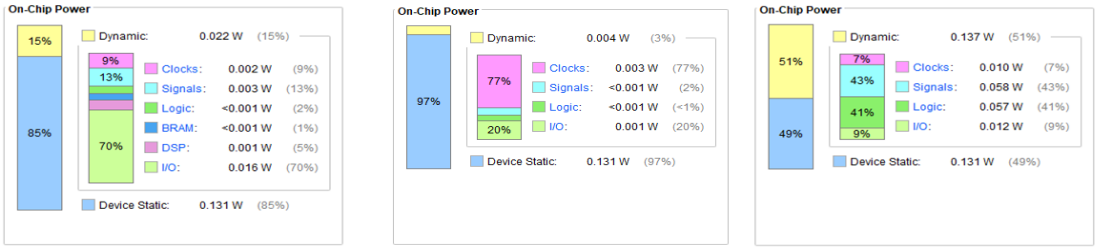
Fonte – O Autor

Conforme a Tabela 5-6, em comparação com a descrição manual (Hand Made), a versão “Melhor HLS” apresenta ganhos de 95% na ocupação de FFs e 97% na ocupação de LUTs, enquanto a solução “Melhor HLS c/ *Pipeline*” apresenta ganhos de 73% em FFs e 83% em LUTs. Destaca-se ainda que a solução “Melhor HLS” consome 66 ciclos de clock para conclusão do processamento, enquanto as soluções “Hand Made” e “Melhor HLS c/ Pipeline” consomem 3 e 59 ciclos respectivamente. Desta forma, a solução “Melhor HLS” leva 434,9 ns para entregar seu resultado, enquanto a solução “Hand Made” entrega uma amostra a cada 23,07 ns, e a solução “Melhor HLS c/ *Pipeline*” a cada 8,01 ns,.

Sendo assim, enquanto a solução de menor área ocupada obtida através de HLS apresenta ganhos expressivos em redução de área, também apresenta perdas significativas em desempenho, uma vez que somente poderá iniciar o processamento de uma nova amostra quando toda a cadeia de cálculos for concluída. Ao realizar a análise da dissipação de potência através do calculo da energia consumida por operação, observamos que a solução “Melhor HLS” representa perdas significativas em relação a solução HM, representando um consumo cerca de 10x maior. Entretanto, ao adotar uma solução com *pipeline* (Melhor HLS c/ pipeline) observamos ganhos expressivos, representando 1/6 do consumo por operação da solução HM.

Destaca-se ainda o consumo elétrico ligeiramente inferior para as soluções HLS, as quais são apresentadas em detalhes na Figura 5.10.

Figura 5.10 – Comparativo de potência consumida entre implementação HLS e HandMade para filtro FIR de 120 taps - ”Ferramenta 1 Backend”



Fonte – O Autor

Conforme observa-se na Figura 5.10 a diferença de consumo entre as implementações reside na potência dinâmica dissipada. Tal qual é observado no filtro de 40 *taps*, esta diferença está ligada principalmente a maior área ocupada pelos elementos lógicos, *nets* e árvore de clock, onde também se destaca o consumo elétrico estático.

5.4.3 Processador VLIW

O processador VLIW (rVex) testado foi alvo de DSE, de modo a obter a menor área possível para sua implementação. Para esta descrição, foram testadas 1440 soluções consumindo 396 minutos de processamento em um computador dotado de um Core i7 7700hq. A Tabela 5-8 apresenta um resumo dos resultados obtidos durante a exploração do espaço de design.

Tabela 5-7 – Resultado DSE para processador rVex – “Ferramenta 1”

Solução	BRAM_18K	DSP48E	FF	LUT	Relógio (ns)	Latência	Taxa
default	3	5	4326	8309	7,79	10	10
363	2	3	1078	3729	7,73	10	10
157	6	3	3487	4390	8,00	52	1

Fonte – O Autor

A Tabela 5-8 lista três resultados selecionados ao final do DSE. Observa-se que a solução de número 363 apresenta redução de 75% e 55% na ocupação de FFs e LUTs e uma redução marginal no ciclo de clock. A solução de número 157 representa a melhor solução obtida com adoção de *pipeline*, proporcionando redução na ordem de 20% e 47% na ocupação

de FFs e LUTs, e um pequeno ganho (2%) em relação ao ciclo de *clock*. Destaca-se ainda que as soluções *default* e 363 consomem 10 ciclos de *clock* para conclusão do processamento, representando uma latência de 77,9 ns, enquanto a solução de número 157 (*pipelined*) apresenta uma latência inicial de 416 ns e após este tempo, entrega uma nova amostra processada a cada 8 ns, mantendo um atraso de 416 ns.

Destaca-se que a melhor solução (Apêndice C) realiza simplificação de *loops* em detrimento de *unroling* e desativa a criação de *pipelines*, bem como, efetua mapeamento de *arrays*. Esta abordagem traz como resultado um código HDL mais simples e linear, porém, com performance limitada. Ainda, verifica-se que a solução 157 (Apêndice C), melhor solução com aplicação de *pipeline,* apresenta como diferenças o mapeamento de memórias *Dual Port* para alguns elementos de memória situados na função principal, bem como, força a aplicação de *pipeline* em todas as funções que compõe o processador VLIW. Também é possível verificar que qualquer *pragma* orientando algum tipo de otimização aos demais elementos de memória foi removido, bem como não existe a adoção de qualquer *pragma* que resulte em otimização a nível de loops (uma vez que é adotado apenas o *tripcount*).

A Tabela 5-8 apresenta um comparativo entre o código HDL desenvolvido por projetista e o código resultante do DSE.

Tabela 5-8 – Comparativo de resultados para processador VLIW- HLS vs HDL HandMade – "Ferramenta 1 Backend"

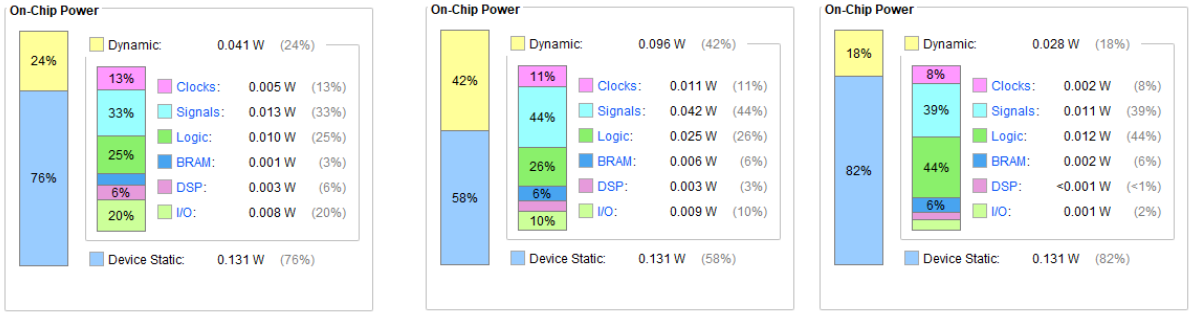
	BRAM	DSP	FF		LUT		TIMING		POWER (mW)	Energia por Operação (nW)
			N°	HLS/HM	N°	HLS/HM	Relógio (ns)	Lat./Taxa		
HandMade	1	10	962	1	2136	1	1,31	7/1	159	0,21
Melhor HLS	1	3	747	0,77	2063	0,96	8,637	10/10	172	14,85
Melhor HLS c/ pipeline	3	3	1898	1,97	2090	0,97	9,913	52/1	227	2,25

Fonte – O Autor

Conforme Tabela 5-9, observa-se pequenos ganhos em ocupação de área nos resultados obtidos através de HLS, porém as perdas em performance ultrapassam as 77 vezes. O código "*Hand Made*" é concebido com *pipeline*, desta forma, após os primeiros ciclos de clock, a cada novo ciclo, uma nova informação processada é entregue na saída do processador, enquanto a solução "Melhor HLS" não faz uso de *pipeline*, necessitando concluir todo o processamento do comando para dar início a um novo ciclo. A solução "Melhor HLS

c/ Pipeline" apresenta o melhor resultado obtido em DSE com *pipeline* habilitado. Ao comparar as soluções HLS com a solução "Hand Made", é possível verificar redução de 70% em utilização de DSPs para as soluções "Melhor HLS" e "Melhor HLS c/ *Pipeline*" bem como redução de 4% e 4% na ocupação de LUTs . Porém, apenas a solução "Melhor HLS" registrou redução na ordem de 23% em ocupação de FFs, enquanto a solução "Melhor HLS c/ *Pipeline*" registra aumento superior a 97% neste mesmo elemento em relação a solução "*Hand Made*". Também se destaca que o regime de *clock* em ambas as soluções HLS apresentam período ligeiramente superior a solução "*Hand Made*", com valores percentuais de 559% para a solução "Melhor HLS" e 656% para "Melhor HLS c/ *Pipeline*". Também se observam perdas em relação a dissipação de potência frente as soluções HLS, que pode ser constatada a partir da análise do consumo de energia por operação, onde destaca-se que mesmo a solução com pipeline obtém 2,25 nW contra apenas 0,21 nW da solução HM. Na Figura 5.11, é possível conferir o consumo de potência estimado pela ferramenta de síntese lógica.

Figura 5.11 – Comparativo de potência consumida entre implementação HLS e HandMade para o processador VLIW- "Ferramenta 1 Backend"



Fonte – O Autor

Conforme a Figura 5.11, é possível verificar que o consumo de potência estática permanece estável entre as versões sintetizadas (devido a fatores já mencionados inerentes a estrutura do FPGA), enquanto, a diferença mais relevante encontra-se na potência dinâmica. É possível verificar que a maior diferença de consumo se encontra nas linhas de *clock* e *nets de interligação*, fato este devido aos circuitos serem maiores o que causa aumento da malha de clock e destas interligações.

5.5 Resultados com “Ferramenta 2”

A realização de testes com “Ferramenta 2” segue uma metodologia semelhante à adotada para os testes com “Ferramenta 1”, através de script para DSE. Porém, algumas adaptações são necessárias, uma vez que a aplicação de parâmetros é passada através da inicialização do software (via Makefile) e através de script de parâmetros (TCL). Também se observa que o “Ferramenta 2” não realiza diferenciação entre vetores, matrizes e variáveis ao converter em elementos de memória (RAM ou BRAM) bem como entre laços e loops ao realizar *unroll* ou *pipeline*. Deste modo, algumas categorias de pragmas passam a ser unidas em uma mesma categoria, reduzindo a quantidade de iterações necessárias. A Tabela 5-9 apresenta a relação de grupos criados para a organização das *pragmas*.

Tabela 5-9 – Grupos de pragmas - DSE com “Ferramenta 2 v1”

Grupo	Descrição
Makefile	Conjunto de opções passadas a aplicação durante sua inicialização
Math	Conjunto de otimizações voltadas a funções matemáticas
Loop	Conjunto de otimizações para loops e laços
General	Conjunto de otimizações gerais do compilador
Mem	Conjunto de otimizações para mapeamento de elementos de memória
Functions	Conjunto de otimizações para funções

Fonte – O Autor

Conforme apresentado na Tabela 5-9 são criados 6 grupos distintos, cada um com sua relação de *pragmas* e configurações. Salienta-se que algumas *pragmas* passadas ao software podem ser utilizadas em mais de uma circunstância, como por exemplo, *pragmas* voltadas a pipeline podem ser adotadas em otimização de funções, loops e matemática, no entanto, a combinação destas opções deve ser cuidadosamente testada, uma vez que pode-se obter ganhos de desempenho ao realizar *pipeline* em um cálculo matemático em específico – normalmente cálculos complexos que geram longos circuitos – mas não na função inteira que realiza este cálculo, por exemplo.

Tabela 5-10 – Pragmas adotadas – “Ferramenta 2 v1”

Pragma	Descrição
Set_parameter CASE_FSM 1	Aplica controle de FSM utilizando *case statement* ou if/else
Set_parameter MB_MINIMIZE_HW 1	Habilita a redução de hardware através da análise da largura de bit de barramentos e estruturas
Set_parameter MB_MINIMIZE_HW 0	Desabilita a redução de hardware através da análise da largura de bit de barramentos e estruturas
Set_combine_basicblock 1	Efetua analise lógica das estruturas combinando-as em blocos para redução de área e otimização
Loop_pipeline	Realiza pipeline em loops
NO_OPT=1	Desabilita otimizações em nível de compilador
NO_OPT=0	Habilita otimizações em nível de compilador
NO_INLINE=1	Desativa compilação de funções como código em linha
NO_INLINE=0	Ativa compilação de funções como código em linha
CFLAG += -mllvm -unroll-threshold=0	Configura limite “infinito” para *unroll* de loops
Set_parameter GROUP_RAMS 1	Realiza mapeamento de elementos de memoria e os agrupa em blocos de memória RAM ou BRAM
Set_parameter GROUP_RAMS_SIMPLE_OFFSET 1	Calcula um *offset* para o endereçamento de memória RAM de modo a simplificar a localização de endereços
Set_parameter LOCAL_RAMS 1	Verifica se um vetor/variável é um elemento de memória local, instanciado apenas dentro da função, ou global, de modo a permitir a criação de uma BRAM dentro do próprio módulo de hardware.
Set_parameter DUAL_PORT_BINDING 1	Habilita uso/criação de memórias DualPort para melhor desempenho
Set_parameter INFERRED_RAMS 0	Desabilita o instanciamento de módulos “Companhia 2”[14] altsyncram

Fonte – O Autor

O conjunto de pragmas apresentados na Tabela 5-10 é separado nos 6 grupos apresentados na Tabela 5-10, representando um conjunto de 301 configurações. Nos subcapítulos a seguir, são apresentados os melhores resultados obtidos em testes com o “Ferramenta 2”.

É importante destacar que todos os testes envolvendo o “Ferramenta 2 v1” foram realizados em ambiente virtualizado, através de Máquina Virtual (VM) fornecida pelos desenvolvedores. Deste modo, a instalação do software segue os padrões de instalações adotados pelo desenvolvedor, sendo assim, comparativos de velocidade de síntese e DSE não são realizados devido à perda de performance mediante o uso deste tipo de solução.

Os resultados apresentados são coletados a partir dos logs da ferramenta de síntese lógica (Quartus 15) levando em consideração a síntese para o FPGA Ciclone 5 5CSEMA5F31C6. Após realizado todo o processo de síntese e este demonstrado viabilidade. São coletadas a quantidade de memória em bits (que é mapeada em BRAMs), Adaptative

[14] Nome da Companhia 2 é omitido por acordo de confidencialidade

Logic Module (ALM) ocupadas, número de registradores, período de clock máximo e número de ciclos de operação.

O número de ciclo de operação é obtido através da ferramenta "*scheduleviewer*", presente na suíte do "Ferramenta 2 v1", que realiza a análise do arquivo HDL produzido mapeando as instruções por ciclo de clock, possibilitando uma análise mais apurada acerca do desempenho da solução obtida.

5.5.1 ULA de 16 bits

A ULA de 16 bits e ponto fixo utilizada para os benchmarks não apresentou ganhos significativos ao adotar algum *pragma* frente a configuração *default* da ferramenta de síntese lógica de alto nível. A

Tabela 5-11 apresenta o resultado do DSE realizado com "Ferramenta 2 v1" para ULA de 16 bits proposta.

Tabela 5-11 – Resultado DSE para ULA16 bits - "Ferramenta 2 v1"

SOLUÇÃO	MEMÓRIA (bits)	DSP	ALMs		REG.		Clock (ns)	Lat./Taxa
			N°	HLS/HM	N°	HLS/HM		
default	0	0	196	1	220	1	2,96	25/1
Melhor HLS	0	0	196	1	220	1	2,96	25/1

Fonte – O Autor

Conforme Tabela 5-11, observa-se que não houveram ganhos em performance, obtendo ganho marginal na alocação de registradores ao adotar *pragmas* ou configuração específica no compilador. Sendo assim, observa-se que a solução de número 7 apresenta desempenho semelhante frente a solução *default*. A Figura 5.12, apresenta as *pragmas* adotadas na solução de número 7.

Figura 5.12 –Solução 7 para ULA16bits – "Ferramenta 2 v1"

```
set_parameter HB_MINIMIZE_HW 1
set_parameter DUAL_PORT_BINDING
```

Fonte – O Autor

Conforme observa-se a solução 7 apenas indica a ferramenta HLS para realização de simplificação de *Hardware* e uso de memória Dual Port. Destaca-se também que mais 6 soluções apresentam desempenho semelhante ao final do DSE, que não serão apresentadas neste documento. Isto ocorre devido a simplicidade desta descrição (ULA de 16 bits e ponto fixo), onde qualquer otimização ou técnica para otimização utilizada acaba por adicionar complexidade ao circuito que irá se traduzir em maior área ocupada e pouco ou nenhum ganho de performance. Desta forma, é seguro afirmar que os ganhos em área e performance serão visíveis apenas em circuitos maiores, como por exemplo, a descrição de uma ULA de 128 bits.

Sendo assim, é efetuada uma pequena alteração na descrição da ULA de modo a torná-la uma Unidade Lógico Aritmética de 128bits. A alteração consiste em modificar a declaração de suas variáveis para “long long”, isto é, variáveis de 16 bytes (128bits) forçando a ferramenta de síntese a criar barramentos e unidades de 128 bits. Ao realizar nova exploração do espaço de design, é obtido o resultado apresentado Tabela 5-12.

Tabela 5-12 – Resultado DSE para ULA128 bits - “Ferramenta 2 v1”

SOLUÇÃO	MEMÓRIA (bits)	DSP	ALMs		REG.		Clock (ns)	Lat./Taxa
			N°	HLS/HM	N°	HLS/HM		
default	0	0	533	1	575	1	4,633	24/1
Melhor HLS	0	0	529	0,99	565	0,98	4,422	24/1

Fonte – O Autor

Conforme Tabela 5-12, com aumento de complexidade da ULA passam a ser observados ganhos – mesmo que marginais – em relação a solução padrão mediante a aplicação de *pragmas*. Observa-se uma redução de cerca de 4,5% no ciclo de clock bem como redução na faixa de 1,7% em ALMs e 0,7% em registradores ocupados. Deste modo, é seguro afirmar que a adoção de *pragmas* pode auxiliar na redução da área, contrabalanceando área e desempenho apenas para circuitos com maior complexidade. A Figura 5.13, apresenta as *pragmas* adotadas na solução de número 33.

Figura 5.13 – Solução 33 para ULA128bits – “Ferramenta 2 v1”

```
set_parameter HB_MINIMIZE_HW 1
set_parameter LOCAL_RAMS 1
```

Fonte – O Autor

Observa-se que a melhor solução obtida (número 33) indica a ferramenta HLS a realizar simplificação de *Hardware* bem como a realizar o instanciamento de elementos de memória RAM diretamente dentro dos módulos. A adoção de elementos de memória diretamente em módulos reduz a latência (tempo de resposta) ao recuperar informações, tornando o processamento mais eficiente e colaborando para a redução do *timing slack* no caminho crítico do circuito.

5.5.2 Filtro FIR

Duas descrições de Filtros FIR foram alvo de DSE no "Ferramenta 2 v1", de modo a realizar um comparativo acerca da melhor solução entre ambos os filtros. O primeiro filtro alvo de testes apresenta 40 *taps*. A Tabela 5-13 apresenta o resultado do DSE para o filtro FIR de 40 taps, realizado com "Ferramenta 2 v1".

Tabela 5-13 – Resultado DSE para FIR de 40 taps – "Ferramenta 2 v1"

SOLUTION	MEMORIA (bits)	DSP	ALMs	Registers	Clock (ns)	Lat./Taxa
Default	6616	4	969	966	5,537	95/95
88	6616	2	596	632	4,735	68/1

Fonte – O Autor

Conforme a Tabela 5-13, a melhor solução obtida através do DSE é a solução de número 88. Observa-se uma sensível redução da ocupação de ALMs e Registradores – em relação a solução padrão da aplicação – bem como ganhos em clock e redução de DSPs, devido a otimização das funções. A Figura 5.14, apresenta as *pragmas* adotadas na solução de número 88.

Figura 5.14 – Solução 88 para filtro FIR de 40 taps – "Ferramenta 2 v1"

```
loop_pipeline "fir_filter/FORL_0"
set_combine_basicblock 1
set_parameter LOCAL_RAMS 1
```

Fonte – O Autor

Deste modo, verifica-se que a melhor solução obtida orienta a ferramenta HLS a realizar a criação de *pipeline* no *loop* de multiplicações iterativas, que garante não apenas a possibilidade de maior *clock* devido a segmentação do cálculo envolvido, mas também reduz o tempo de espera entre amostras sequenciais, uma vez que após 68 ciclos de clock, uma amostra sempre será entregue. Também se observa o uso de memória RAM local, que se traduz na criação de BRAMs (mais otimizadas) e a combinação de blocos básicos, simplificando a estrutura dos circuitos.

A Tabela 5-14 apresenta um comparativo entre as soluções HLS e *Hand Made*.

Tabela 5-14 – Comparativo de resultados para filtro FIR de 40 taps - HLS vs HDL HandMade – "Ferramenta 2 v1"

SOLUÇÃO	MEMÓRIA (bits)	DSP	ALMs		REG.		Clock (ns)	Lat./Taxa
			N°	HLS/HM	N°	HLS/HM		
HandMade	0	0	1785	1	1391	1	17,746	3/1
Default (Companhia3 4)	6616	4	969	0,54	966	0,69	5,537	95/1
Melhor HLS c/ pipeline (Companhia3 4)	6616	2	596	0,33	632	0,45	4,735	68/1
Melhor HLS c/ pipeline (Companhia3 7)	6584	4	713	0,39	785	0,56	7,02	3/1

Fonte – O Autor

Conforme a Tabela 5-14, são observados ganhos significativos em área ocupada para a solução "Melhor HLS c/ pipeline ("Ferramenta 2 v1")" em relação a solução HM, onde são obtidas reduções de 66,6% na ocupação de ALMs e 54,5% na ocupação de registradores. No entanto, a solução HLS representa grandes perdas em performance devido a longa profundidade de *pipeline*, resultando um atraso superior a 321,98 ns obtidos na solução HLS contra 53,23 ns obtidos na solução HM. Destaca-se ainda a solução "Melhor HLS c/ pipeline ("Ferramenta 2 v2")", obtida através de testes realizados com a última versão da ferramenta, a qual obtivemos acesso por curto espaço de tempo. Esta solução é obtida através da última versão comercial da ferramenta, que demonstra maior maturidade em relação a última versão acadêmica disponível. Observam-se os ganhos na ordem de 61% e 44% para ocupação de ALMs e registradores em relação a solução HM e um ciclo de relógio de 7ns contra 17,7ns,

mantendo uma profundidade de *pipeline* de apenas 3 ciclos, representando uma solução equivalente a HM.

O segundo filtro FIR alvo de DSE através do "Ferramenta 2" apresenta 120 *taps*, onde espera-se obter maiores possibilidades de otimização em relação ao *loop* de multiplicações iterativas. A Tabela 5-15 apresenta o resultado do DSE realizado com "Ferramenta 2" para o filtro FIR de 120 taps.

Tabela 5-15 – Resultado DSE para FIR de 120 taps – "Ferramenta 2 v1"

SOLUTION	MEMORIA (bits)	DSP	ALMs	Registers	Clock (ns)	Lat/Taxa
default	19420	4	2496	2575	5,819	272/272
90	19320	2	2398	2453	6,212	271/1

Fonte – O Autor

Conforme a Tabela 5-15, o melhor resultado em área é obtido na solução de número 90, com uma pequena redução em ALMs e Registradores frente a solução padrão da aplicação. Destaca-se a perda de performance no que diz respeito ao *clock*, porém, com uma redução de 1 ciclo na cadeia lógica. Abaixo, constam os *pragmas* adotados na solução de número 90.

Figura 5.15 – Solução 90 para filtro FIR de 120 taps – "Ferramenta 2 v1"

```
loop_pipeline "fir_filter/FORL_0"
set_combine_basicblock 1
set_parameter INFERRED_RAMS 0
```

Fonte – O Autor

Através da análise das *pragmas* adotadas na solução de número 90, é verificado que esta solução faz uso de combinação dos blocos básicos de modo a simplificar os blocos lógicos do circuito a ser sintetizado. Destaca-se também o uso da *pragma* "inferred rams" de modo a desabilitar o mapeamento para elementos de memória *altsyncram* da "Companhia 2"[15] e a adoção de *pipeline* sob o loop de multiplicação iterativa do filtro. Deste modo, apesar de a solução 90 apresentar *clock* ligeiramente mais lento em relação a solução padrão, seu

[15] Nome da Companhia 2 é omitido por acordo de confidencialidade

desempenho é superior. A solução padrão não faz uso de *pipeline*, sendo assim, obtemos uma amostra calculada na saída do filtro após 272 ciclos de *clock*. Na solução de número 90, com a adoção de *pipeline*, após os 271 ciclos de clock, a cada novo ciclo temos uma nova amostra processada na saída.

A Tabela 5-16 apresenta um comparativo entre as soluções HLS e *Hand Made*.

Tabela 5-16 – Comparativo de resultados para filtro FIR de 120 taps - HLS vs HDL HandMade – "Ferramenta 2 v1"

SOLUÇÃO	MEMÓRIA (bits)	DSP	ALMs		REG.		Clock (ns)	Lat./Taxa
			N°	HLS/HM	N°	HLS/HM		
HandMade	0	0	5026	1	3852	1	15,03	3/1
Melhor HLS c/ pipeline ("Ferramenta 2 v1")	19320	2	2398	0,47	2453	0,63	6,212	271/1
Melhor HLS c/ pipeline ("Ferramenta 2 v2")	0	40	1626	0,32	1869	0,48	9,38	3/1

Fonte – O Autor

Conforme a Tabela 5-16, são observados ganhos significativos em área ocupada para a solução " Melhor HLS c/ pipeline ("Ferramenta 2 v1")" em relação a solução HM, onde são obtidas reduções de 53% na ocupação de ALMs e 37% na ocupação de registradores. No entanto, a solução "Melhor HLS c/ pipeline ("Ferramenta 2 v1")" representa grandes perdas em performance devido a longa profundidade de *pipeline*, resultando um atraso superior a 1683,45 ns contra 45,09 ns da solução HM. Destaca-se ainda a solução "Melhor HLS c/ pipeline ("Ferramenta 2 v2")", obtida através de testes realizados com a última versão da ferramenta, a qual obtivemos acesso por curto espaço de tempo. Esta solução é obtida através da última versão comercial da ferramenta, que demonstra maior maturidade em relação a última versão acadêmica disponível. Observam-se os ganhos na ordem de 68% e 52% para ocupação de ALMs e registradores em relação a solução HM e um ciclo de relógio de 9,38 ns contra 15,03 ns, mantendo uma profundidade de *pipeline* de apenas 3 ciclos, representando uma solução equivalente a HM.

5.5.3 Processador VLIW

O processador VLIW testado através de DSE com o "Ferramenta 2" apresentou 301 possíveis combinações conforme *pragmas* e configurações estabelecidos na Tabela 11. Após concluído o DSE, observa-se que o melhor rendimento é obtido através da solução de número 58. O resultado do DSE é apresentado na Tabela 5-17, relacionando elementos de área e *clock* máximo estimado através da ferramenta de Síntese Lógica (Quartus).

Tabela 5-17 – Resultado DSE para processador VLIW – "Ferramenta 2 v1"

Solução	Memória (bits)	DSP	ALMs	Registradores	Clock (ns)	Lat./Taxa
Default	5024	14	2442	1976	6,244	148/148
58	5052	7	1337	1129	5,526	187/1

Fonte – O Autor

Conforme a Tabela 5-17, o melhor resultado foi obtido através do conjunto solução de número 58. É observado ganho em todos os aspectos em relação a solução padrão da ferramenta (sem inserção de *pragmas* ou configurações), onde se observa uma pequena redução na quantidade de memória (que influencia no tamanho e quantidade das BRAMs), redução de 50% de elementos DSP, 42% em registradores e 45% de ALMs, obtendo um período de clock de 5,52ns (181Mhz) contra 6,24 ns (160Mhz) obtidos na solução padrão. Observa-se ainda que a quantidade de ciclos necessários para a conclusão das operações houve um incremento de 26% em relação a solução padrão, ou seja, apesar de solução número 58 apresentar maior clock, a quantidade de ciclos necessárias para conclusão das operações é maior.

A Figura 5.16, apresenta os parâmetros passados ao compilador durante a inicialização e adotados na solução de número 58.

Figura 5.16 – Parâmetros de Inicialização 58 para processador VLIW – "Ferramenta 2 v1"

```
NO_OPT = 1
```

Fonte – O Autor

Conforme apresentado, para esta solução é indicado ao compilador que não efetue otimizações em nível de código (durante compilação do código C/C++), fazendo com que as oportunidades de otimização estejam apenas em nível de conversão HDL. Abaixo, é relacionado o conjunto de *pragmas* adotados para o processo HLS.

Figura 5.17 – Solução 58 para processador VLIW – "Ferramenta 2 v1"

```
loop_pipeline "FORL_0"
loop_pipeline "FORL_1"
loop_pipeline "FORL_2"
loop_pipeline "FORL_3"
loop_pipeline "FORL_4"
loop_pipeline "FORL_5"
loop_pipeline "FORL_6"
loop_pipeline "WHILE_7"
set_parameter CASE_FSM 1
set_parameter LOCAL_RAMS 1
```

Fonte – O Autor

Conforme a Figura 5.17, o conjunto solução número 58 faz uso de *pipeline*, otimização de máquinas de estado (case_fsm) e força o mapeamento dos elementos de memória em memórias locais (tais como BRAM instanciadas dentro dos módulos). O uso de *pipeline* tende a incrementar drasticamente a "performance" geral do sistema, uma vez que os módulos internos não necessitam aguardar o término de toda a cadeia de processamento para iniciar um novo processo. Tal qual uma esteira, após 187 ciclos de clock (preenchido todo o caminho de dados no *pipeline*), uma nova amostra é alimentada em sua entrada a cada ciclo de clock e uma amostra é entregue na saída. Deste modo, após os 187 ciclos iniciais, é necessário apenas 1 ciclo de clock para entregar o resultado do processamento. O uso de memória RAM local (dentro dos módulos) apresenta uma latência menor, o que possibilita atingir frequência mais alta de relógio geral para o sistema.

Desta forma, observa-se que o conjunto solução número 58 apresenta um desempenho sensivelmente superior a solução padrão (que não faz uso de *pipeline*), uma vez que a solução padrão somente poderá receber uma nova amostra em sua entrada após passados 148 ciclos.

Sendo assim, temos que a solução padrão consome 923,22 ns para realizar o processamento de cada amostra, enquanto a solução número 58 consome 1033,36 ns para realizar o processamento da primeira amostra, e 5,52 ns para realizar o processamento das amostras subsequentes.

É importante ressaltar que o uso de estruturas de *pipeline* muito longas somente apresentam melhor rendimento em processamentos lineares, como filtros FIR ou processadores DSP e semelhantes, que contam com alto paralelismo de dados e que não realizam, ou realizam poucas tomadas de decisões. Nestes casos (processadores multifunção) a adoção de *pipeline* deve ser realizada com certo cuidado, de modo a reduzir a quantidade de ciclos necessários, sempre balanceando *clock*, área e número de estágios de *pipeline*, inclusive analisando quais módulos se beneficiam da adoção de *pipeline* individualmente. A Tabela 5-18 apresenta um comparativo entre as soluções HLS e *Hand Made*.

Tabela 5-18 – Comparativo de resultados para processador VLIW - HLS vs HDL HandMade – "Ferramenta 2 v1"

SOLUÇÃO	MEMÓRIA (bits)	DSP	ALMs		REG.		Clock (ns)	Lat./Taxa
			N°	HLS/HM	N°	HLS/HM		
HandMade	0	22	6809	1	1631	1	13,69	7/1
Default ("Ferramenta 2 v1")	5024	14	2442	0,35	1976	1,21	6,244	148/148
Melhor HLS c/ pipeline ("Ferramenta 2 v1")	5052	7	1337	0,19	1129	0,69	5,526	187

Fonte – O Autor

Conforme a Tabela Tabela 5-18, são observados ganhos significativos em área ocupada para a solução HLS em relação a solução HM, onde são obtidas reduções de 80,3% na ocupação de ALMs e 30,7% na ocupação de registradores. No entanto, a solução HLS representa grandes perdas em performance devido a longa profundidade de *pipeline*, resultando um atraso de grupo superior a 1033,36 ns obtidos na solução HLS contra 95,83 ns da solução HM.

5.6 Comparação de resultados em FPGAs

Não existe uma forma para comparação direta entre resultados obtidos com ferramentas de HLS e FPGAs de diferentes fabricantes. Isto ocorre devido a grande diferença entre as estruturas que compõe este FPGAs. Deste modo, como abordagem para um comparativo que seja útil para uso industrial destas ferramentas, é adotado o preço de mercado dos FPGAs alvo como meio de viabilizar o comparativo, uma vez que existem limitações em relação aos FPGFAs possíveis de teste no "Ferramenta 2".

Tabela 5-19 – FPGAs Comparados - Preços e Modelos

Fabricante	Família	Modelo	Unidades Lógicas	Preço (USD)
“Companhia 2”[16]	Ciclone V	5CSEMA5F31C6	32075 ALMs	211,25
“Companhia 1”[17]	Artix 7	XC7A200-1	129000 LUTs	210,33

Fonte – O Autor - Compilado de mouser.com e digikey.com

Ainda, de modo a tornar o comparativo mais claro, todos os testes foram realizados utilizando-se a melhor solução encontrada através de DSE em ambas as ferramentas, com *pipeline* habilitado (exceto para a ULA 16 bits). As tabelas apresentadas relacionam a quantidade de LUTs, ALMs e DSPs (quando necessário) para uma comparação direta, bem como, o cálculo percentual de área ocupada em relação a LUTs/ALMs, extremamente necessário para obtenção de um panorama geral em relação a ocupação do FPGA e as possibilidades de embarcar lógica no mesmo.

A Tabela 5-20 apresenta um comparativo entre os resultados obtidos com o “Ferramenta 2” e “Ferramenta 1” ao realizar HLS para a ULA de 16 bits (ponto fixo).

Tabela 5-20 – Comparativo de resultados entre “Ferramenta 2” e “Ferramenta 1” - ULA16 bits

Ferramenta	FPGA	Área (%)	ALMs/ LUTs	Registers/ FFs	DSPs	Clock (MHz)	Lat./Taxa	Pipeline (sim/não)
”Ferramenta 2 v1”	Ciclone V	0,611	196	220	0	337	25/1	não
”Ferramenta 1”	Artix 7	-	0	0	4	370	1/1	não

Fonte – O Autor

Conforme Tabela 5-20, observa-se um consumo de área diferenciado entre os FPGAs, com uma frequência de clock ligeiramente diferente (9%). O fato ocorre devido a melhor solução obtida com “Ferramenta 1” fazer uso de DSPs para realização dos cálculos e lógica, deixando as LUTs e FFs liberados. Deste modo, é obtido grande diferença na quantidade de ciclos necessários para concluir o processamento, onde o resultado obtido com “Ferramenta 2” necessita de 25 ciclos contra apenas um ciclo do resultado obtido com

[16] Nome da Companhia 2 é omitido por acordo de confidencialidade
[17] Nome da Companhia 1 é omitido por acordo de confidencialidade

“Ferramenta 1”. Sendo assim, a solução para o Ciclone V apresenta grande desvantagem em relação a solução obtida para Artix 7, concluindo o processamento em 116,87 ns contra 2,7 ns.

A Tabela 5-21 apresenta um comparativo entre os resultados obtidos com o “Ferramenta 2” e “Ferramenta 1” ao realizar HLS para o FIR de 40 taps.

Tabela 5-21 – Comparativo de resultados entre “Ferramenta 2” e ”Ferramenta 1” - FIR 40 Taps

Ferramenta	FPGA	Area (%)	ALMs/ LUTs	Registers/ FFs	Clock (MHz)	Lat./Taxa	Pipeline (sim/não)
”Ferramenta 2 v1”	Ciclone 5	1,858	969	966	211	68/1	sim
”Ferramenta 1”	Artix 7	0,700	903	848	146	20/1	sim

Fonte – O Autor

Conforme observa-se na Tabela 5-21, o resultado de síntese obtido através do “Ferramenta 1” para o FPGA ”Companhia 1”[18] representa uma ocupação de área menor em relação ao ”Ferramenta 2 v1” (0,7% para o Artix contra 1,858% para Ciclone V). Porém, é possível observar simetria entre a quantidade de ALMs e LUTs, Registers e FFs consumidos em ambas as soluções. Ainda, uma grande diferença encontrada reside na quantidade de ciclos necessários para conclusão do processamento da primeira amostra, sendo 20 ciclos para a solução obtida com “Ferramenta 1” contra 68 ciclos para a solução obtida com “Ferramenta 2”. Desta forma, como resultado, ao utilizar uma solução obtida através do “Ferramenta 2” obtemos um atraso superior a 135%, isto é, após alimentar o sistema com uma amostra esta levará 319,6 ns para ser entregue na saída (atraso total de processamento) contra 136,4 ns para a solução obtida através do “Ferramenta 1”. Destaca-se, é claro, que não existe a necessidade de aguardar este tempo para alimentar o sistema com uma nova amostra devido a adoção de *Pipeline*, no entanto, o sinal amostrado na entrada (amostrado em velocidade do ciclo de clock do sistema) estará atrasado na saída com onde este atraso será proporcional ao número de ciclos necessários para conclusão do processamento.

A Tabela 5-22 apresenta um comparativo entre os resultados obtidos com o “Ferramenta 2” e “Ferramenta 1” ao realizar HLS para o filtro FIR de 120 taps.

[18] Nome da Companhia 1 é omitido por acordo de confidencialidade

Tabela 5-22 – Comparativo de resultados entre “Ferramenta 2” e ”Ferramenta 1” - FIR 120 Taps

Ferramenta	FPGA	Area (%)	ALMs/ LUTs	Registers/ FFs	Clock (MHz)	Lat./Taxa	Pipeline (sim/não)
”Ferramenta 2 v1”	Ciclone 5	7,476	1398	2453	160	271	Sim
”Ferramenta 1”	Artix 7	0,829	1012	1070	124	59	Sim

Fonte – O Autor

Conforme a Tabela 5-22, observa-se um consumo de área relativa obtido para o “Ferramenta 2” cerca de 801% superior a área obtida para solução Artix. Em valores absolutos, são consumidos 38% elementos lógicos e 129% de memória (Registers/FFs) a mais na solução “Ferramenta 2”. Entretanto, o período de clock obtido nesta solução é cerca de 29% menor (maior clock). No entanto, somente com uma análise do atraso total obtido pelas soluções é possível verificar qual a solução é mais eficaz. Observa-se que a solução obtida com “Ferramenta 2” necessita de 271 ciclos para concluir o processamento de uma amostra, representando um atraso de 1,69 ms, enquanto a solução obtida através do “Ferramenta 1” necessita de apenas 59 ciclos para a conclusão de seu processamento, representando um atraso de 472,59 ns. Desta forma, é correto afirmar que a solução obtida através do “Ferramenta 1” apresenta um desempenho muito superior mesmo com um ciclo de clock ligeiramente mais lento.

A Tabela 5-23 apresenta um comparativo entre os resultados obtidos com o “Ferramenta 2” e “Ferramenta 1” ao realizar HLS para o processador VLIW.

Tabela 5-23 – Comparativo de resultados entre “Ferramenta 2” e ”Ferramenta 1” - Processador VLIW

Ferramenta	FPGA	Area (%)	ALMs/ LUTs	Registers/ FFs	Clock (MHz)	Taxa./Lat	Pipeline (sim/não)
”Ferramenta 2 v1”	Ciclone 5	4,168	2398	2453	180	187/1	sim
”Ferramenta 1”	Artix 7	3,403	4390	3487	100	8/1	sim

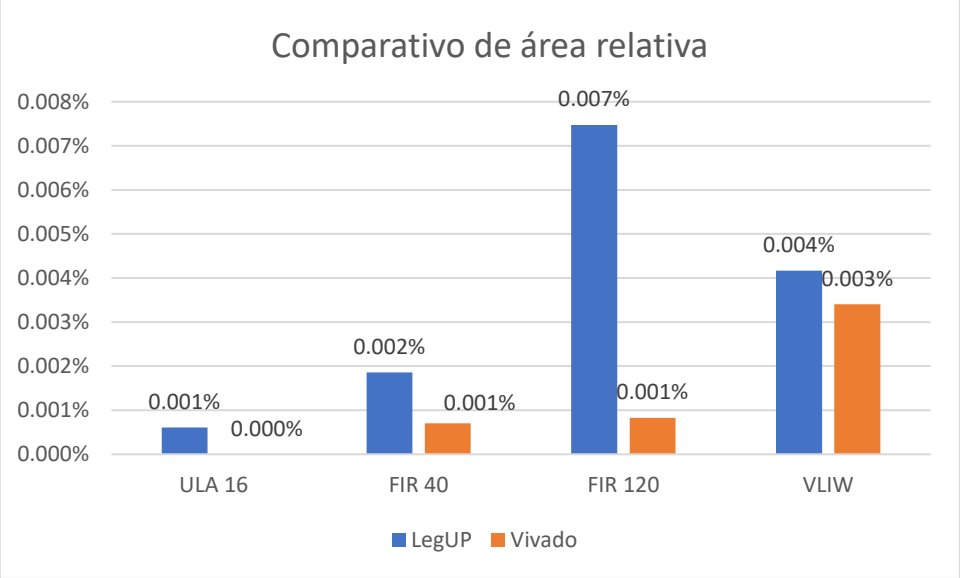
Fonte – O Autor

Conforme a Tabela 5-23, observa-se um consumo de área relativa ligeiramente superior para o resultado obtido com “Ferramenta 2” (22%), entretanto, o consumo de área absoluta representa redução de 45% dos elementos lógicos e 29,65% de elementos de

memória (Registers/FFs) para a solução “Ferramenta 2”. Também é observado ganho no período de clock que atinge a marca de 80%. Realizando comparação quanto a quantidade de ciclos necessárias para a conclusão do processamento, observa-se que o resultado obtido com “Ferramenta 2” representa incremento de cerca de 249% em relação ao resultado obtido através do “Ferramenta 1”, refletindo diretamente no atraso de grupo. Realizando a análise de atraso, temos que a solução obtida com “Ferramenta 2” consome cerca de 1036 ns contra cerca de 416 ns da solução “Ferramenta 1”, sendo deste modo, o atraso obtido pela solução Artix 7, 59,8% menor que o atraso obtido com a solução Ciclone V.

O Gráfico 5.1 apresenta um comparativo entre a área ocupada pelas soluções obtidas através do “Ferramenta 2” e “Ferramenta 1” para os FPGAs selecionados.

Gráfico 5.1 – Comparativo de área relativa

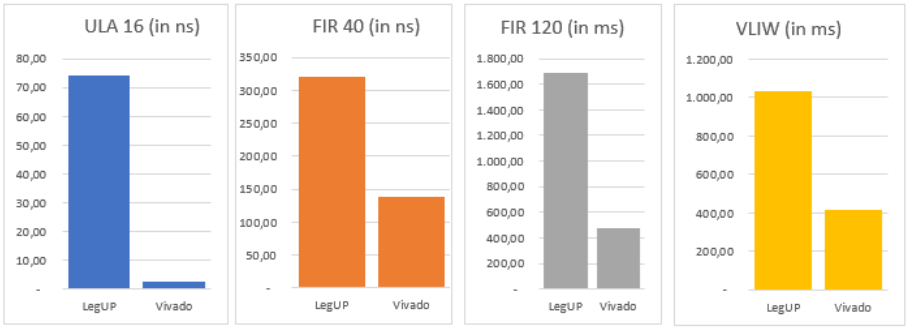


Fonte – O Autor

Conforme o Gráfico 5.1 , em todas as descrições sintetizadas através das ferramentas HLS, a solução obtida com o “Ferramenta 1” apresenta menor consumo de área, deste modo, sendo o objetivo redução de área, a adoção do “Ferramenta 1” e o FPGA selecionado apresenta uma melhor escolha de mercado, uma vez que a tendência de redução de área se sustentou em todos os experimentos realizados, proporcionando maior possibilidade para embarcar lógica em um mesmo FPGA.

O Gráfico 5.2 representa um combinado com o tempo máximo de processamento (ou atraso) para os 4 experimentos realizados com “Ferramenta 2” e “Ferramenta 1”.

Gráfico 5.2 – Gráfico comparativo de latência



Fonte – O Autor

Conforme o Gráfico 5.2 em todos os experimentos as soluções obtidas com "Ferramenta 1" para Artix 7 apresentam desempenho superior. Neste ponto, é importante destacar que apenas uma análise do atraso total não satisfaz diretamente um critério de desempenho. Esta análise deve levar em consideração não apenas o atraso a que as amostras serão submetidas ao final do processamento, mas também a *clock* do projeto principal onde estas estruturas serão inseridas e qual o requisito de latência mínima. Pode-se tomar como exemplo os resultados obtidos para o Filtro FIR de 120 Taps, onde o ciclo de clock atingido pela solução "Ferramenta 2" representa uma frequência de amostragem de cerca de 180MHz, enquanto a solução "Ferramenta 1", de 124MHz, deste modo, em situações onde o atraso total das amostras não é crítico o ganho em frequência de amostragem pode representar uma boa oportunidade.

Ainda, não obstante aos resultados, é importante destacar que a versão testada do "Ferramenta 2" se trata da última versão Acadêmica disponível, que não recebe atualizações regulares, estando estacionada na versão 4 enquanto sua contrapartida comercial encontra-se na versão 6.2 dando suporte a outros modelos de FPGA "Companhia 2"[19] e tendo suporte a FPGAs "Companhia 1"[20] incluído recentemente.

Também deve-se levar em consideração que a versão testada (e suportada pelo "Ferramenta 2") do Cyclone V trata-se de um SoC, com processador ARM embarcado, o que torna seu custo de mercado superior. Um comparativo mais direto deveria ser realizado através de uma versão FPGA não SoC da "Companhia 2" (que possuiria uma maior quantidade de ALMs) na mesma faixa de preço do Artix 7 comparado, porém, este não estava disponível para testes na versão 4 do "Ferramenta 2" acadêmico, deste modo, é ideal uma

[19] Nome da Companhia 2 é omitido por acordo de confidencialidade
[20] Nome da Companhia 1 é omitido por acordo de confidencialidade

analisa quantitativa em relação aos ALMs e Registradores consumidos ao final do processo de *back-end* em comparação a quantidade de LUTs e FlipFlops na solução Artix.

5.7 Resultados com "Ferramenta 3"

A realização dos testes com a ferramenta "Ferramenta 3" segue uma metodologia diferenciada correspondente à natureza da ferramenta. Devido a este apresentar seu próprio método para realização de DSE, não se faz necessária a aplicação de scripts externos, apenas a devida seleção de conjunto de *pragmas* que devem ser testadas durante o processo. No entanto, na época em que se realiza estes testes, a instituição (PGMICRO/UFRGS) não possui acesso a licenças do "Ferramenta 3 LPOPT", estando limitados a versão 8.3 de 2016 da suíte "Ferramenta 3 Suite" (Sendo a versão 10 lançada em 2017 a última disponível quando este estudo é realizado). Sendo assim, não é possível a realização de testes em malha fechada para otimização de potência.

Deste modo, são realizados testes com três descrições comportamentais em C/C++: Filtro FIR de 40 *taps*, Filtro FIR de 120 *taps* e processador VLIW. Cada descrição é submetida a DSE através da própria ferramenta, adotando-se 7 conjuntos de configurações principais, conforme Tabela 5-24.

Tabela 5-24 – Conjuntos de configurações - "Ferramenta 3 Suíte"

Nome	Alvo	Memória	Operator Clustering	Resource Sharing	C-COREs	Pipeline	Resource Share
Default	Área	Default	Default	Default	Default	Default	Default
Área 1	Área	DSRAM	Sim	60%	Default	Não	Sim
Área 2	Área	Register	Sim	60%	Default	Não	Sim
Área 3	Área	DSRAM	Sim	60%	Default	Sim	Sim
Área 4	Área	Register	Sim	60%	Default	Sim	Sim
Latência 1	Performance	DSRAM	Sim	40%	Default	Sim	Sim
Latência 2	Performance	Register	Sim	40%	Default	Sim	Sim
Latência 3	Performance	Register	Sim	40%	Default	Sim	Sim

Fonte – O Autor

As configurações apresentadas na Tabela 5-24 dizem respeito ao fluxo selecionado (guiado a redução de área ou desempenho), o tipo de memória adotada para armazenamento de informações (conversão de variáveis em elementos de memória) – sendo suportadas

memórias do tipo Single-port Static Random Access Memory (SSRAM), Dual-port Static Random Access Memory (DSRAM), e registratores (registers), a criação de "*clusters*" de operações que podem ser compartilhados (como *clusters* de somadores, multiplicadores e etc), a adoção de *pipeline* (criar ou não pipeline e sua profundidade) e finalmente, compartilhamento de recursos. Destaca-se ainda a opção "*C-Cores*", que para todos os casos é mantido em sua configuração padrão, exceto em exploração específica realizada para o processador VLIW que é apresentada em capítulo próprio. Esta opção de configuração guia a ferramenta a transformar funções chamadas a partir da função principal em núcleos (do inglês, *cores*) – que são instanciados como módulos – permitindo uma divisão mais clara das estruturas que compõe o circuito e mantendo uma hierarquia semelhante a apresentada na descrição comportamental original. A partir da configuração "*C-Cores*", é possível ainda definir se esta estrutura deve ser convertida em um circuito sequencial ou combinacional. Destaca-se ainda a configuração "Latência 3", que apresenta as mesmas definições de "Latência 2", porém com forçando a ferramenta a obter uma solução mais próxima as soluções HandMade (profundidade de *pipeline* e período de relógio).

Após concluída a conversão do código comportamental em HDL através do processo HLS, é realizado a síntese lógica através do Cadence Genus adotando um pacote de bibliotecas Standard Cell fornecido pela STMicroeletronics de 65 nanômetros voltada a *Low Power*.

Todos os resultados apurados e apresentados são pós síntese lógica, ou seja, antes do processo de *Back-end*, adotando estimativas de potência com base em padrões de atividade de chaveamento dos transístores obtidos por simulação do *netlist* gerado pelo Genus através da ferramenta IRUN, possibilitando uma melhor estimativa de potência para comparativos.

5.7.1 Filtro FIR

São realizados dois testes com filtros FIR, adotando uma descrição de filtro com 40 *taps* e uma segunda descrição de filtro com 120 *taps*. Ambas descrições adotam os 7 padrões de configurações apresentadas (Tabela 5-24), acrescidos de um oitavo padrão (Latência 3) de configurações que força um limite máximo de estágios de *pipeline* para o fluxo "performance" (*latency*) de modo a equiparar o *netlist* obtido a partir da descrição HDL realizada manualmente com o *netlist* obtido a partir de HLS. A Tabela 5-25 apresenta os

resultados obtidos para o filtro de 40 *taps* através de síntese lógica com Genus e biblioteca de células-padrão de 65nm.

Tabela 5-25 – Resultados para FIR 40 Taps – "Ferramenta 3" e Genus

Nomes	Alvo	Células	Área		Potência Dinâmica (mW)	Potência		Clock (ns)	Lat./Taxa	Memória	Energia por operação (nW)
			µm²	HLS/HM		mw	HLS/HM				
HandMade		7968	66440	1	21,72	21,77	1	5,4	3/1	REG	0,117
Default	Default	1257	9110	0,13	3,29	3,29	0,15	4	8/8	DSRAM	0,104
Área 1	Área	1333	9562	0,14	4,44	4,45	0,20	4	9/9	DSRAM	0,160
Área 2	Área	7027	49831	0,75	14,48	14,50	0,66	4	5/5	REG	0,290
Área 3	Área	1558	11339	0,17	4,87	4,87	0,22	4	10/1	DSRAM	0,024
Área 4	Área	4948	35294	0,53	9,39	9,41	0,43	4	6/1	REG	0,037
Latência 1	Latência	1682	11792	0,17	6,29	6,29	0,29	3	9/1	DSRAM	0,019
Latência 2	Latência	2488	19596	0,29	8,73	8,74	0,40	3	5/1	REG	0,026
Latência 3	Latência	2655	21022	0,31	4,94	4,96	0,22	5,4	3/1	DSRAM	0,026

Fonte – O Autor

Conforme a Tabela 5-25, alguns itens e resultados devem ser destacados, a iniciar pelo melhor resultado em área, obtido a partir das configurações padrões da ferramenta (*default*). Também se torna visível uma anomalia em relação a área obtida nas configurações "Área 2" e "Área 4", que fazem uso de registradores para conversão dos elementos de memória, contrastando-se ao resultado obtido em "Latência 2", *setup* este configurado de modo a realizar o mapeamento dos elementos de memória em registradores em conjunto com a otimização de desempenho em detrimento de área. Entretanto, observou-se redução da área ao utilizar o fluxo "área" (alvo *area*) em conjunto com mapeamento para memória RAM, porém, perdas em desempenho (*clock*) são percebidas, como esperado, mantendo-se a quantidade de ciclos para conclusão da operação. Destaca-se ainda o resultado obtido em "Latência 3", onde clock e profundidade de *pipeline* são equiparados a solução HDL descrita a mão, obtendo um *netlist* com cerca de 1/3 da área e ¼ da potência consumida. Ainda em se tratando da potência, ao observar a energia gasta por operação, temos ganhos significativos, onde se destaca a configuração "latência 1", que apresenta um consumo de 0,019 nW por operação e "latência 3", que representa 0,026nW por operação contra 0,117nW obtido com a solução HM.

A Tabela 5-26 apresenta os resultados obtidos para o filtro de 120 *taps* através de síntese lógica com Genus e biblioteca de 65nm.

Tabela 5-26 – Resultados para FIR 120 *Taps* - "Ferramenta 3" e Genus

Nome	Alvo	Células	Área		Potência Dinâmica (mW)	Potência		Clock (ns)	Lat./Taxa	Memória	Energia por Operação (nW)
			μm^2	HLS/HM		mw	HLS/HM				
HandMade		22523	178993	1	52,580	52,704	1	5,4	3/1	Register	0,284
Default	Default	1688	11622	0.06	4,953	4,959	0.09	4	9/9	DSRAM	0,171
Área 1	Área	1660	11776	0.07	4,409	4,415	0.08	4	9/9	DSRAM	0,153
Área 2	Área	19096	134441	0.75	39,926	39,993	0.76	4	5/1	Register	0,159
Área 3	Área	1958	14128	0.08	5,350	5,357	0.10	4	10/1	DSRAM	0,021
Área 4	Área	19221	136274	0.76	38,598	38,667	0.73	4	5/1	Register	0,154
Latência 1	Latência	1994	14747	0.08	7,288	7,296	0.14	3	10/1	DSRAM	0,022
Latência 2	Latência	5511	44935	0.25	18,638	18,664	0.35	3	5/1	Register	0,056
Latência 3	Latência	5791	45156	0.25	11,246	11,273	0.21	5,4	3/1	Register	0,061

Fonte – O Autor

Conforme Tabela 5-26, observam-se ganhos em área com a configuração "Área 1", onde é adotado um *Resource Sharing* de até 60% e o fluxo HLS próprio para otimização de área. Ainda se destaca o consumo elétrico reduzido para esta solução, em detrimento de uma maior latência total. Também torna-se visível uma anomalia em relação as configurações "Área 2" e "Área 4" que apresentam consumo de área superior a configuração "latência 2", comportamento este já observado com os testes realizados para o filtro de 40 taps (Tabela 5-25). Destaca-se ainda os resultados obtidos através da equiparação por desempenho listados na configuração "latência 3" em relação a solução "*HandMade*" (descrição HDL Manual), apresentando cerca de ¼ da área obtida através da solução manual e 1/5 da potência. Observa-se ainda a tendência de reduzido consumo por operação, tal qual observado para o filtro de 40 *taps*, onde destaca-se os 0,022 nW obtidos com a solução "Latência 1" e 0,061 nW obtidos com a solução "Latência 3", representando grande ganho em relação a solução HM e seus 0,284 nW por operação.

Importa ainda destacar que todos os ganhos de desempenho obtidos a partir da solução "Latência 3" para ambos os filtros representam grande aumento em consumo elétrico. Isto se dá não devido a maior quantidade de células lógicas adotadas, mas sim a necessidade de células com maior esforço lógico para garantir um maior alvo de clock. Ainda em relação a "anomalia" detectada ao adotar o fluxo área, importa salientar que a ferramenta foi instruída a realizar mapeamento dos elementos de memória em registradores, o que pode levar o algoritmo a buscar granularidade destes registradores em detrimento à composição de um pequeno bloco ou banco de registradores, causando efeito contrário em relação ao consumo de área. Este efeito é esperado uma vez que a ferramenta realiza o "*allocation*" antes do "*scheduling*" durante um fluxo para minimização de área. Desta forma, os registradores que

compõem os elementos de memória tendem a ficar espalhados e longe das unidades de execução lógica, gerando um efeito cascata que se traduzirá na necessidade de barramentos mais longos e *buffers*, enquanto ao realizar o *scheduling* antes do *allocation* em um fluxo dedicado ao desempenho, busca-se reduzir a latência de acesso a estes elementos, alocando-os diretamente dentro dos módulos. Este fato pode ser comprovado ao utilizar um bloco de memória *Dual-Port* disponibilizado pelo "Ferramenta 3" , que apesar de ser obtido através de descrição HDL (fornecido pela empresa "Companhia 4"[21]), tende a não ser quebrado em unidades menores de memória devido a ser instanciado em uma área específica no *floorplaning* da ferramenta de síntese lógica, sendo instanciado em uma posição otimizada entre os blocos lógicos que necessitam de acesso ao mesmo.

5.7.2 Processador VLIW

A descrição do processador VLIW apresenta maior complexidade, traduzindo-se em um *netlist* mais extenso garantindo maiores possibilidades para análise e otimizações. Para esta descrição, são realizados os sete (7) testes padrões apresentados na Tabela 5-24 acrescido de mais dois testes a saber: "Latência 3" e "Latência 4".

O teste "Latência 4" apresenta em sua configuração a profundidade de *pipeline* fixada em 7 estágios e uma frequência alvo de 174 MHz, equiparando o *netlist* resultante a solução manual (*HandMade*). A solução "Latência 3", apresenta como diferencial a configuração de "*C-Cores*" realizada manualmente, de modo a permitir uma escolha de como as funções que compõe a descrição deverão ser traduzidas (*inline* ou como um módulo/núcleo). A Tabela 5-27 apresenta como foi realizada esta configuração.

Tabela 5-27 – Processador VLIW - Configuração "Latência 3" - C-Cores

Função	C-CORE	Tipo de Circuito
advance_pc	Core	Sequencial
fetch	Core	Combinacional
decode	Core	Combinacional
ctrl	Core	Sequencial
alu	Core	Sequencial
mult	Core	Sequencial

Fonte – O Autor

[21] Nome da Companhia 4 é omitido por acordo de confidencialidade

Conforme a Tabela 5-27, optou-se em transformar todas as funções possíveis do código comportamental que descreve o processador VLIW em núcleos independentes (*Core*) em detrimento da tradução deste código em um circuito inserido diretamente no módulo principal (*Inline*). Também se destaca a possibilidade de guiar a ferramenta na tradução destes circuitos em circuitos sequenciais ou combinacionais, permitindo deste modo um melhor aproveitamento e desempenho. Para esta configuração, as funções "*fetch*" e "*decode*", responsáveis pela carga de dados e decodificação de instrução respectivamente, são configuradas como "sequenciais", uma vez que resultam em circuitos menores. Sendo assim, para garantir maior desempenho em uma estrutura de *pipeline*, devem ser executas em um único ciclo de *clock*.

A Tabela 5-28 apresenta os resultados obtidos a partir da síntese lógica realizada através do Genus para as soluções HDLs obtidas a partir do "Ferramenta 3" .

Tabela 5-28 – Resultados para processador VLIW – "Ferramenta 3" e Genus

Nome	Alvo	Cél.	Área		Potência Dinâmica (mW)	Potência		Clock (ns)	Lat./ Taxa	Mem.	Energia por Operação (nW)
			μm^2	HLS/HM		mw	HLS/ HM				
HandMade	-	22720	107629	1	4,562	4,63	1	5,7	7/1	Register	0,026
Default	Def.	9733	71520	0,66	30,789	30,84	6.65	2,5	12/12	DSRAM	0,925
Área 1	Área	8639	63918	0.59	16,786	16,82	3.63	4	9/9	DSRAM	0,605
Área 2	Área	21602	162040	1.51	39,054	39,13	8.44	4	5/5	Register	0,782
Área 3	Área	8926	63058	0.59	16,628	16,66	3.59	4	8/1	DSRAM	0,333
Área 4	Área	20713	152459	1.42	32,052	32,13	6.93	4	7/1	Register	0,128
Latência 1	Lat.	8413	58133	0.54	20,928	20,96	4.52	3	8/1	DSRAM	0,062
Latência 2	Lat.	25002	182593	1.70	62,792	62,89	13.56	3	9/1	Register	0,188
Latência 3	Lat.	9486	67000	0.62	20,582	20,61	4.44	3	9/1	DSRAM/REG	0,061
Latência 4	Lat.	11336	82250	0.76	12,931	12,97	2.80	5,7	7/1	Register	0,074

Fonte – O Autor

A partir dos resultados apresentados na Tabela 5-28, é possível verificar que existem ganhos em área ao adotar o fluxo HLS "área" (Área 1) em relação a configuração padrão (*default*) da ferramenta, porém, o consumo de área ainda permanece superior a solução "Latência 1". Também observa-se que o menor consumo elétrico é obtido a partir da solução "Latência 3", que apresenta uma arquitetura de memória hibrida, adotando módulos de memória dual-port para os maiores blocos (conversão de longos *arrays*) e registradores para pequenos blocos (variáveis de poucos *bytes*), bem como, o uso de lógica combinacional para as funções *fetch* e *decode*. A melhor performance é obtida através da configuração "Latência

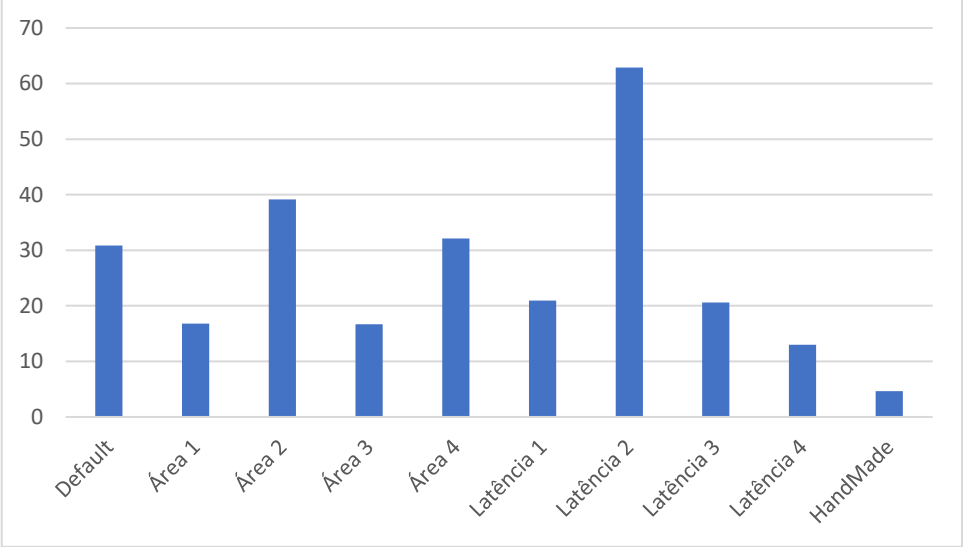
1", que faz uso de pipeline e memória dual-port, proporcionando uma baixa latência de inicialização (24 ns) e posteriormente uma latência de 3 ns para a entrega de resultados em fila (devido ao maior clock). No entanto, em uma comparação direta à solução manual (*HandMade*) realizada através da solução "Latência 4", é possível verificar ganhos em área (ocupação de cerca de ½ da área) e perdas em potência (consumo cerca de 2,7 vezes superior). Entretanto, observando-se a energia dissipada por operação, temos que "Latência 3" representa uma melhor solução em consumo entre as soluções HLS com um resultado de 0,061 nW por operação, porém, este resultado ainda representa um consumo superior aos 0,026 nW obtidos com a solução HM. Deste modo, a solução "Latência 3" será utilizada como ponto de partida para exploração de baixo consumo que é apresentada no capítulo 5.9.

Deste modo, observa-se que para descrições mais complexas, os ganhos residem principalmente na correta escolha em relação a estrutura de memória, profundidade de *pipeline* aliado ao tipo de circuito ao qual uma função deve ser traduzida (combinacional ou sequencial). Entretanto, otimização em relação a potência permanece sendo apenas um efeito colateral em relação ao quão otimizado (em termos lógicos) é o *netlist* produzido, acompanhando diretamente a área consumida. Oportunidades para otimização de potência residem principalmente na adoção de *Power* e *Clock Gating*, e possivelmente na adoção de *Frequency Scaling* com base no "*load*" sobre o processador (isto é, com base na quantidade de requisições e instruções demandadas), onde este último afetará diretamente a performance do dispositivo.

5.8 Análise de potência Dinâmica

A potência dinâmica atualmente corresponde a maior parcela de potência consumida por um CI. Devido a modernização dos processos de litografia temos redução da tensão de operação, causando efeito direto sobre as potências estáticas, sendo sumariamente reduzidas. Desta forma, a redução e a análise da potência dinâmica consumida pelo layout obtido passam a ser o foco do estudo acerca do rendimento e eficiência energética de um circuito, seja através de HLS, seja por codificação manual. O Gráfico 5.3 apresenta a potência dinâmica consumida por todos os circuitos sintetizados através do Cadence Genus.

Gráfico 5.3 – Potência Dinâmica para Processador VLIW em processo CMOS 65nm



Fonte – O Autor

A potência dinâmica consumida para o processador VLIW apresentada através do Gráfico 5.3 é estimada através de técnica PLE adotando vetores de simulação (com arquivoVCD). Observa-se que o circuito obtido através de descrição HDL manual representa maiores ganhos em relação aos demais circuitos obtidos através de HLS (4,63mW), onde a opção "Latência 4" apresenta o melhor resultados (12,93 mW) entre os circuitos obtidos através de HLS. No entanto, apesar de observar-se uma grande variância entre o circuito de menor rendimento enérgico ("Latência 2" – 62,79mW) e o circuito de maior rendimento ("Latência 4"), este último (melhor rendimento) ainda representa um consumo cerca de 179% maior do que o circuito descrito manualmente.

Com base nos resultados obtidos a partir da exploração, é possível verificar que os ganhos em consumo elétrico residem basicamente na adoção de estruturas e circuitos otimizados, tais como blocos de memória e blocos aritméticos avançados, bem como, a adoção de *clock* e *power gating* pela ferramenta de síntese lógica, conforme revisado nos capítulos anteriores desta dissertação.

5.9 *Low Power* em HLS

Conforme os estudos apresentados nesta dissertação, a otimização de potência em circuitos obtidos através de HLS pode ser realizada por meio de duas formas: Otimização de

código e adoção de macro blocos de alto desempenho. No entanto, melhoras pontuais ainda podem ser alcançadas ao realizar-se o processo de síntese lógica e *back-end*, por meio da adoção de *power* e *clock-gating*, múltiplos domínios de tensão e frequência variável (*frequency scalling*). Deste modo, são realizadas explorações e testes forçando a ferramenta HLS ("Ferramenta 3") a adotar técnicas que visam otimizar o código e adotar marco blocos otimizados como memórias dual-port.

Para tanto, quatro novas sínteses HLS são realizadas, adotando uma estrutura de memória hibrida baseada na solução "Latência 3", que faz uso de blocos de memória *dual-port* para acomodar as maiores matrizes (elemento de memória antes da conversão HLS). Deste modo, são reduzidos a quantidade de transístores necessários para composição de bancos de registradores para armazenamento de grandes cadeias de dados, uma vez que um bloco de memória é otimizado para tal, apresentando consumo elétrico reduzido devido a menor quantidade de elementos. Ainda, é adotado uma configuração que força a ferramenta na adoção de *resouce sharing* global de 50% e uma profundidade de *pipeline* mínima de 6 níveis.

É importante destacar que tentativas anteriores de realizar otimização a partir da solução "Latência 4" não obtiveram ganhos, devido a dificuldades da ferramenta em realizar uma alocação mais otimizada. Isso ocorre devido ao *netlist* resultante tornar-se mais extenso mediante a adoção de registradores como elementos de memória para propósito geral, tornando o processo de síntese mais complexo.

Cada solução obtida adota um conjunto de configurações diferenciado no que tange a aplicação de políticas globais (a partir do *top*) e locais (diretamente em cada módulo). Assim, é possível forçar a ferramenta a adotar um esforço diferenciado entre módulos ou simplesmente a utilizar *clusters* para cálculo e lógica localizados dentro de um módulo específico ou apenas no *top*. A Tabela 5-29 apresenta a relação de configurações adotadas em cada solução.

Tabela 5-29 – Configuração adotada para otimização de potência - "Ferramenta 3"

Sol.	CCORE	Pipelines	Sinal Enable	Sinal Enable local	STALL Flags	STALL Flags Local	Esforço	Esforço local	Cluster	Local Cluster
LP1	SIM	9	padrão	não	Sim	não	médio	médio	não	não
LP2	SIM	7	sim	sim	Sim	não	médio	médio	não	não
LP3	SIM	6	sim	sim	Sim	não	médio	alto	sim	sim
LP4	SIM	6	sim	sim	Sim	sim	médio	alto	sim	parcial

Fonte – O Autor

Conforme a Tabela 5-29, a configuração de esforço lógico global é mantida em "médio", sendo que esta configuração é escolhida devido ao esforço "alto" conduzir a um incremento substancial no tempo de síntese. No entanto, esforço lógico local (isto é, nos "c*cores*") é alterado para "alto" nas soluções "LP3" e "LP4". Ainda se observa que a ferramenta é instruída a utilizar o sinal *enable* global e local em todas as soluções a partir de "LP2", possibilitando assim uma identificação de comportamento simplificada para a criação de *clock* e *power gating* a partir da ferramenta de síntese lógica (Genus). As soluções "LP2" e "LP3", fazem uso de *clusters* locais, sendo que a solução "LP4" ainda realiza o uso parcial desta configuração, conforme apresentado na Tabela 5-30.

Tabela 5-30 – Configuração de *clusterização* para solução LP4

Função	Local Cluster
advance_pc	não
Fetch	não
Decode	não
Ctrl	não
Alu	sim
Mult	sim

Fonte – O Autor

O uso de *clusters* para operações matemáticas pode representar ganhos em performance e potência, uma vez que estes são otimizados para tal tarefa, ocupando menor área e apresentando uma estrutura lógica otimizada. Porém, seu uso indiscriminado pode representar perda de performance elétrica quando o módulo não demanda uso extensivo de operações matemáticas, fato este que ocorre nas funções "*advance_pc*", "*fetch*", "*decode*" e "*ctrl*"; Porém, as funções "*alu*" e "*mult*" são destinadas justamente à compor a Unidade Logico-Aritmética e Unidade de Cálculos (multiplicação e divisão) do processador VLIW. A

Tabela 5-31 apresenta os resultados obtidos através da síntese lógica com Cadence Genus para um processo de 65nm.

Tabela 5-31 – Resultados de otimização de potência

Nome	Alvo	Cels.	Área	Potência Dinâmica (mW)	Potência (mW)	Clock (ns)	Ciclos	Memória	Energia por Operação (nW)
LP1	Latência	8116	57318	10,51	10.53	5,7	9/1	DSRAM/REG	0,0599
LP2	Latência	7822	58812	10,58	10,61	5,7	7/1	DSRAM/REG	0,0603
LP3	Latência	7837	58834	10,48	10,51	5,7	6/1	DSRAM/REG	0,0597
LP4	Latência	8151	60838	9,78	9,81	5,7	6/1	DSRAM/REG	0,0557

Fonte – O Autor

Conforme

Tabela 5-31 observa-se queda no consumo elétrico entre a solução “Latência 3” e “LP1” explicada unicamente pela redução no *clock*, uma vez que “LP1” trata-se da mesma solução base sintetizada para outro alvo de *clock*. Porém, ao passar a adotar as novas políticas, observa-se melhoria de desempenho global (redução da profundidade de *pipeline*) e redução da quantidade de células necessárias, porém, com incremento na área (células de maior esforço lógico). Sendo assim, ao combinar as técnicas de alocação de recursos e otimização, é obtido um ganho de 7% em consumo elétrico sob a solução base (LP1/Latência 3), que também se reflete ao analisar a energia gasta por operação.

Deste modo, é realizada uma exploração adicional baseada na solução “LP4”, porém alterando-se o tipo de memória de dual-port para single-port. Esta, por sua vez, apresenta uma estrutura mais simples que garante menor área e consumo elétrico, porém, ainda operando dentro da faixa de performance exigida pelo circuito testado. A Tabela 5-32 apresenta o resultado da síntese lógica para o circuito obtido.

Tabela 5-32 – Resultados de otimização de potência com SSRAM

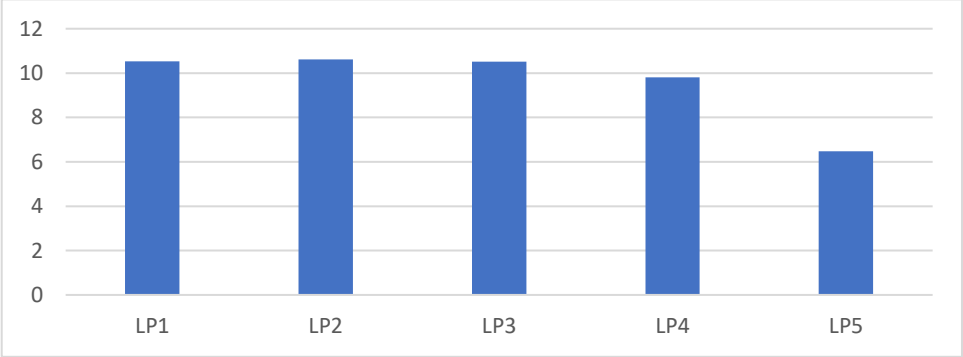
Nome	Alvo	Cel.	Área	Potência Dinâmica (mW)	Potência (mW)	Relógio (ns)	Lat./ Taxa	Memória	Energia por Operação (nW)

LP5	Latência	8284	61336	6,44	6,47	5,7	7/1	SSRAM/REG	0,0369

Fonte – O Autor

Conforme a Tabela 5-32, é possível observar um ganho considerável em potência ao adotar elemento de memória *single-port* em detrimento de memória *dual-port*, representando um ganho de 34% em relação aos resultados obtidos na solução LP4 e 38,8% em relação a solução LP1, que representa um consumo de energia por operação de 0,0368 nW. O Gráfico 5.4 apresenta um comparativo entre a potência consumida pelas soluções “LP1” até “LP5”.

Gráfico 5.4 – Comparativo de potência consumida entre soluções em exploração para LowPower



Fonte – O Autor

Observa-se através do Gráfico 5.4 a tendência de queda do consumo elétrico conforme a ferramenta é guiada para uma alocação de recursos mais eficiente, adotando blocos otimizados. Através de um *binding* orientado ao uso de estruturas otimizadas, observa-se a queda de consumo elétrico e incremento na performance global do sistema, sendo assim, é seguro afirmar que a alocação de estruturas e circuitos otimizados representa ganhos tangíveis em potência. A Tabela 5-33 apresenta o resultado comparativo entre a melhor solução HLS e a solução descrita manualmente (*HandMade*).

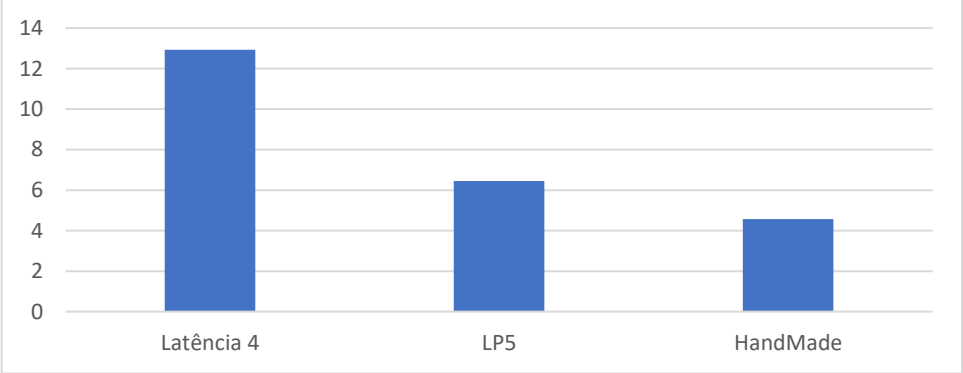
Tabela 5-33 – Comparativo de potência entre melhores soluções

Nome	Alvo	Cel.	Área	Potência Dinâmica (mW)	Potência (mW)	Relógio (MHz)	Lat./ Taxa	Memória	Energia por Operação (nW)
LP5	Latência	8284	61336	6,44	6,475	5,7	7/1	SRAM/REG	0,0369
HandMade	-	22720	107629	4,562	4,639	5,7	7/1	Register	0,0264

Fonte – O Autor

A comparação entre os resultados obtidos é listada na Tabela 5-33, onde se observa que a solução HLS representa uma ocupação de área inferior, representando cerca de 36% da quantidade de células totais e 56% da área total frente a solução “manual”, no entanto, seu consumo elétrico permanece superior, representando um consumo 40% maior para uma mesma especificação. O Gráfico 5.5 apresenta um comparativo entre a potência das melhores soluções obtidas com a solução descrita manualmente (*HandMade*).

Gráfico 5.5 – Comparativo de potência entre melhores soluções



Fonte – O Autor

Conforme o Gráfico 5.5, a solução “Latência 4” apresenta um consumo elétrico de 12,9mw, representando o consumo antes da exploração para redução de potência, a solução “LP5” apresenta um consumo de 6,44mw, obtido após a exploração para redução de potência e a solução “HandMade” apresenta um consumo de 4,56mw, obtida através de um circuito descrito manualmente em HDL por um *designer*.

Destaca-se ainda, que a mesma descrição HDL obtida através de HLS para o processador VLIW de melhor performance em potência (LP5) pode ser sintetizada para um alvo de *clock* maior, diferente da solução “HandMade”, cujo clock máximo obtido é de 174MHz para este conjunto de células de 65nm. A viabilidade de incremento no *clock* desta descrição proporciona a possibilidade da adição de uma unidade PLL ou um divisor de *clock* na entrada de *clock* principal do dispositivo de modo a permitir a seleção da frequência principal do processador VLIW em tempo de execução, proporcionando o uso de *frequency scalling* para aumento de performance por curtos períodos de tempo, bem como a redução do *clock* em períodos de inatividade, reduzindo o consumo global do dispositivo.

6 CONCLUSÕES

O uso de ferramentas de síntese de alto nível (HLS) representam um grande ganho em produtividade, pela redução considerável do tempo requerido para o desenvolvimento de sistemas digitais complexos. Deve-se ressaltar que esta redução impacta na economia de tempo necessário apenas para a codificação RTL do sistema em projeto. As grandes oportunidades de pesquisa em síntese de alto nível estão associadas à necessidade de permitir que, desde a codificação em HLL e durante a síntese HLS, possa o projetista controlar adequadamente as otimizações voltadas para a minimização de área e de consumo de potência. Como revisamos nesta Dissertação, há no mercado ferramentas HLS dedicadas para dois grandes tipos de plataformas de implementação. O primeiro grupo de ferramentas são aquelas voltadas especificamente para que, na fase de síntese lógica e mapeamento tecnológico subsequente, se use exclusivamente os dispositivos FPGAs, como é o caso do "Ferramenta 1" e do "Ferramenta 2". O segundo grupo de ferramentas é de uso estritamente em HLS, independente de tecnologia de implementação – podendo serem utilizadas estas ferramentas de HLS tanto para síntese em FPGAs quando para síntese física em *Standard Cells*.

Este trabalho se propôs a realizar uma revisão acerca das principais técnicas destinadas a *Low Power* e *Area Reduction* adotadas na atualidade, bem como, explorar sua aplicação através de ferramentas HLS comerciais e a qualidade dos resultados. Para tanto, foram realizados testes em três ferramentas comerciais, duas voltadas a FPGAs de diferentes fabricantes e uma voltada a *Standard Cells* através de técnica DSE. Como alvo de testes, são selecionadas quatro descrições comportamentais (ULA 16bits, FIR de 40 *taps*, FIR de 120 *taps* e processador VLIW) que possuem descrição HDL equivalente obtida manualmente (*HandMade*). Os resultados extraídos das ferramentas HLS passa por processo de síntese lógica de modo a realizar comparação de desempenho e área com as soluções HandMade.

No âmbito de FPGAS, as ferramentas apresentam sérias limitações no que tange a otimização do algoritmo, onde podemos destacar a dificuldade em reduzir a profundidade de *pipeline* para determinados circuitos que, em uma descrição manual, podem ser consideradas elementares, onde podemos destacar os filtros FIR cuja forma transposta representa ganho de desempenho e não foi obtida através de HLS. Este fato causa perda de performance global para sistema, aumentando a latência do circuito. Entretanto, as ferramentas permitem obter circuitos com reduzido consumo de área aliado a um curto tempo de desenvolvimento. Ainda

se destaca a falta de suporte à políticas orientadas a baixo consumo elétrico diretamente na fase de HLS, de modo a aplicar boas práticas de modularização, múltiplas linhas de clock, inserção de *flags* para tratamento de *clock* e *power gating* (para os FPGAs que suportam) e inserção de métodos para *frequency scalling* além de proporcionar a escolha e adoção de *cores* de alta eficiência para compartilhamento de recursos. Grande motivo para falta de alguns destes recursos, está diretamente ligado a falta suporte em nível de hardware para algumas supracitadas técnicas, justamente devido ao FPGA possuir uma estrutura fixa e de alta complexidade devido às interligações necessárias. Deste modo, é seguro afirmar que a aplicação de técnicas de baixo consumo elétrico através de HLS e o desenvolvimento de um hardware FPGA que proporcione ampla aplicação das principais técnicas, representa vasta área de pesquisa com inúmeras oportunidades, que abrangem desde o *hardware* do FPGA até as ferramentas EDA que venham a suportar estas técnicas nativamente, tomando decisões mais assertivas.

No âmbito de ferramentas para uso geral, destaca-se a aplicação do "Ferramenta 3" em conjunto com *Standard Cells* (testado com suíte Cadence), onde é possível verificar uma ferramenta competente no que diz respeito à otimização de área, performance lógica e performance elétrica, rivalizando com as soluções obtidas manualmente e representando menor área em todos os testes realizados. Este fato ocorre principalmente pela liberdade proporcionada pelo uso de *Standard Cells*, que possibilita a aplicação de diversas técnicas de otimização lógica que indiretamente causam efeitos sobre o consumo elétrico, disponíveis através da própria ferramenta HLS em sua interface. Contudo, pode-se observar certa carência de uma forma direta para criação de múltiplos domínios de clock, bem como a geração de PLLs ou divisores de clock embarcados ao sistema, de modo a propiciar a aplicação de *frequency scalling* nativo ao dispositivo. Entretanto, destaca-se que durante estes experimentos, não foi possível realizar testes com a última versão desta ferramenta ou com suíte completa ("Ferramenta 3" com "Ferramenta 3 LPOPT") onde, possivelmente, resultados melhores seriam obtidos.

Deste modo, conclui-se que as ferramentas HLS atuais representam significativa vantagem em termos de tempo de desenvolvimento, sendo necessária, entretanto, a realização de análise em termos de alvo de performance exigido, bem como a tecnologia alvo do circuito (FPGA ou *Standard Cells*). Observa-se, ainda, que as ferramentas comerciais necessitam de amadurecimento em relação a práticas destinadas a *Low Power*, principalmente no que diz respeito a tecnologia de FPGAs, representando grandes possibilidades de pesquisa e inovação. Entretanto, ferramentas voltadas a geração de HDL de propósito geral, isto é, aquele que pode

ser aplicado diretamente em ferramentas de síntese lógica e *back-end* para *Standard Cells* (como o fluxo comercial da Cadence Genus/Inovus, por exemplo), apresentam maiores benefícios, proporcionando maior controle sobre o código gerado, no qual é possível selecionar inclusive quais funções devem ser geradas como módulo independente e, qual é o tipo de lógica a ser gerada (sequencial ou combinacional), bem como, a seleção de técnicas para tratamento de laços de repetição e otimização lógica. Soma-se ainda a estas características a possibilidade de inserção de elementos otimizados (*clusters* e memórias) que proporcionam melhor *binding,* garantindo performance lógica e elétrica superior e resultando em um RTL de maior qualidade rivalizando com soluções codificadas manualmente.

Sendo assim, é possível afirmar que o uso de HLS, pode vir a tornar-se uma crescente tendência no que tange o uso empresarial, passando a representar uma fatia cada vez maior para o desenvolvimento de circuitos aceleradores. Espera-se que, conforme ocorre o amadurecimento das técnicas HLS e novas tecnologias e algoritmos que permitam a aplicação mais direta de técnicas de *Low Power* sejam incorporadas às ferramentas de HLS, o uso mais amplo destas deve passar a ser cogitado pela indústria em geral, dado o impacto (redução) em tempo de projeto proporcionado. Ainda importa dizer que, na opinião do autor deste trabalho, o uso do HLS através da inserção manual de *pragmas* deverá ser uma tendência na indústria, em detrimento da adoção de ferramentas DSE, uma vez que permite maior controle por parte do projetista em relação ao resultado final do RTL. Esta abordagem permite que o projetista selecione como diferentes funções devem ser convertidas, orientando a ferramenta em direção a uma determinada arquitetura ou topologia. Sendo assim, o projetista necessita de experiência enquanto *layout* e topologia dos mais diferentes tipos de circuitos digitais, de modo a otimizar os resultados. Insta salientar que a adoção do DSE, apesar de representar ganho de tempo (para o projetista) resulta em arquiteturas aleatórias, que nem sempre proporcionarão o melhor equilíbrio entre desempenho, consumo elétrico e área ocupada, dificultando processos posteriores de otimização deste *netlist* resultante.

Como trabalhos futuros, indica-se (1) o aprofundamento no estudo de técnicas de *Low Power* diretamente orientadas para *Standard Cells* e (2) a proposição de metodologias para análise do "*load*" em circuitos de modo a criar formas automatizadas para aplicação de *frequency scalling* em circuitos de maior complexidades, tais como decodificadores e processadores de uso geral, (3) a melhoria dos algoritmos DSE desenvolvidos e testados com as ferramentas voltadas a FPGA, adicionando uma camada de *auto learning* guiada por uma estrutura de árvore, de modo a eliminar a sequência de testes com quaisquer *pragma* que apresentaram baixo rendimento, reduzindo o tempo necessário para convergência dos

resultados . Ainda se sugere (4) um estudo mais abrangente acerca de *Low Power* em FPGAs propondo técnicas aplicadas diretamente em nível de síntese HLS e (5) melhorias estruturais para proporcionar um melhor rendimento elétrico global. E, por fim, (6) a aplicação de *machine-learning* e redes neurais diretamente sob HLS, buscando uma conversão de código comportamental em RTL mais natural e de maior qualidade.

REFERÊNCIAS BIBLIOGRÁFICAS

AS, VAN T.; **$\rho - VEX$ A Reconfigurable and Extensible VLIW Processor**; 2008; Disponível em < https://github.com/tvanas/r-vex/blob/master/OperationsAndSemantics.md >

BELDACHI, A. F.; NUNEZ-YANEZ, J. L**.; Run-time power and performance scaling in 28 nm FPGAs**; 2014; Disponível em < http://www.bristol.ac.uk/media-library/sites/engineering/research/migrated/documents/v7scale.pdf >

BERGAMASCHI, R. A.; RAJE, S. ; **Generalized Resource Sharing**. 1997. IEEE.

BSOUL, A. A. M.; WILTON, S. J. E.; **An FPGA Architecture Supporting Dynamically Controlled Power Gating;** Department of Electrical and Computer Engineering University of British Columbia 2332 Main Mall, Vancouver, BC V6T 1Z4, Canada, IEEE , 978-1-4244-8983-1/10/$26.00, 2010

CHANDRAKASAN, A. P.; BRODERSEN, R. W.; **Low Power Digital CMOS Design**; KLUWE ACADEMIC PUBLISHERS, BOSTON, 1995.

COSTA, E. A. C.; **Operadores Aritméticos de Baixo Consumo para Arquiteturas de Circuitos DSP**; 2002, Programa de pós-graduação em Microeletrônica. UFRGS, Porto Alegre, Brasil.

COUSSY, P.; GAJSKI, D. D.; MEREDITH, M.; TAKACH, A. **An Introduction to High-Level Synthesis**. 2009. IEEE CASS. Disponível em: < http://janders.eecg.toronto.edu/1387/readings/hls.pdf>

EKEKWE, N.; **Power dissipation and interconnect noise challenges in nanometer CMOS technologies**; 2010, IEEE Potentials

EVANS, J.; GEORGAKAKIS, S.; **Overview of high level synthesis tools.** 2010. Disponível em <http://iopscience.iop.org/article/10.1088/1748-0221/6/02/C02005/pdf>

HUNDA, S.; MALLICK, M.; ANDERSON, J. H.; **Clock Gating Architecture for FPGA Power Reduction;** 2009; Disponível em < http://janders.eecg.toronto.edu/pdfs/clk_09.pdf>

COMPANHIA3; **"Ferramenta2" Documentation Release 4.0**; Universidade de Toronto; 2015; Disponível em < http://bit.ly/2Z8XOc0 > ,

NUNEZ-YANEZ, J. L**.;** HOSSEINABADY, M.;BELDACHI, A.; **Energy Optimization in Commercial FPGAs with Voltage, Frequency and Logic Scaling**; IEEE TRANSACTIONS ON COMPUTERS, VOL. 65, NO. 5, MAY 2016

SANTIAGO, J. S.; **Architectural Exploration of Digital Systems Design for FPGAs Using C/C++/SystemC Specification Languages**; 2015. Programa de pós-graduação em Microeletrônica. UFRGS, Porto Alegre, Brasil.

SARKAR, A. et al; **Low power VLSI design: Fundamentals**; 2016; Disponível em < https://www.researchgate.net/publication/306560136_Low_power_VLSI_design_Fundamentals >

SOARES, L. B.; **Design of area and energy-efficient digital CMOS FIR filters with approximate adder circuits**; 2016; ALOG; Disponível em < https://dl.acm.org/citation.cfm?id=3000739 >

WESTE, N.; ESHRAGHIAN, K. Principles of CMOS VLSI Design; Addison-Wesley Publishing Company, second ed. 1994.

COMPANHIA1a; **Lowering Power at 28 nm with "Companhia 1" 7 Series Devices**; 2015; Disponível em < http://bit.ly/2ZaVAgd >

COMPANHIA1b; **Lowering Power using the Voltage Identification Bit**; 2012; Disponível em < http://bit.ly/2Za3Ygg >

COMPANHIA1c; **Reducing Switching Power with Intelligent Clock Gating**; 2013; Disponível em < http://bit.ly/2ZbT1qm >

COMPANHIA1d; **Spartan-6 FPGA Power Management**; 2016; Disponível em < http://bit.ly/2KFh6Sz >

COMPANHIA1e; **7 Series FPGAs Clocking Resources**; 2017; Disponível em < http://bit.ly/31MpMME >

COMPANHIA1e; **"Ferramenta 1" Design Suite User Guide High-Level Synthesis**;2014, Disponível em < http://bit.ly/33Iu9d4 >

COMPANHIA1f; **Lowering Power at 28 nm with Xilinx 7 Series Devices**; 2015, Disponível em < http://bit.ly/2ZaVAgd >

COMPANHIA1g; **UltraFast High-Level Productivity Design Methodology Guide; 2017**, Disponível em < http://bit.ly/2ZcSKHG >

COMPANHIA1h; **High-Level Synthesis with "Ferramenta 1"; 2012**, Disponível em < http://bit.ly/2TF7ycZ >

ZHANG, Z.; CHEN, D.; SETEVE, D.; CAMPBELL, K.; **High-Level Synthesis for Low Power Designs**. Disponível em < https://www.jstage.jst.go.jp/article/ipsjtsldm/8/0/8_12/_article >

JACKSON, E. K; KANG, E; SCHULTE, W; **An Approach for Effective Design Space**

Exploration; Monterey Workshops; 2010.

OLIVEIRA, M. F. S.; **Model Driven Engineering Methodology for Design Space Exploration of Embadded Systems**; Universidade Federal do Rio Grande do Sul, Porto Alegre, Outubro 2013.

KNIJNENBURG, P. M.; **Flattening VLIW code generation for imperfectly nested loops**; Department of Computer Science, Leiden University, Tech. Rep., 1998.

KHAN, K.; **FIR FILTER Example for "Ferramenta 1"**; 2014; Disponível em < http://bit.ly/2Z84k2R >

COMPANHIA4DS, **“Ferramenta 3” Synthesis,**; 2017; “Companhia 4”; Disponível em < http://bit.ly/2YUuMSd >

COMPANHIA4HLS, **“Ferramenta 3” High-Level Synthesis**; 2018; “Companhia 4”, Disponível em < http://bit.ly/2N9bgu2 >

COMPANHIA4LP, **“Ferramenta 3” Lp For A Power Optimized Esl Hardware Realization Flow**; 2018; “Companhia 4”, Disponível em < http://bit.ly/2ZbIMT1 >

COMPANHIA4UR**, “Ferramenta 3” Synthesis User and Reference Manual**; 2018; “Companhia 4”

APÊNDICE A – FILTRO FIR DE 40 TAPS - VHDL

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity fir01_16b is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           x : in  STD_LOGIC_VECTOR (15 downto 0);
           s : out  STD_LOGIC_VECTOR (31 downto 0));
end fir01_16b;

architecture Behavioral of fir01_16b is

signal xi : STD_LOGIC_VECTOR(15 downto 0);
signal pt127 : STD_LOGIC_VECTOR(23 downto 0);
signal nxi : STD_LOGIC_VECTOR(23 downto 0);
signal xil7 : STD_LOGIC_VECTOR(23 downto 0);
signal pt95 : STD_LOGIC_VECTOR(23 downto 0);
signal nxil5 : STD_LOGIC_VECTOR(23 downto 0);
signal pt79 : STD_LOGIC_VECTOR(23 downto 0);
signal nxil4 : STD_LOGIC_VECTOR(23 downto 0);
signal pt93 : STD_LOGIC_VECTOR(23 downto 0);
signal nxil1 : STD_LOGIC_VECTOR(23 downto 0);
signal pt81 : STD_LOGIC_VECTOR(23 downto 0);
signal xil1 : STD_LOGIC_VECTOR(23 downto 0);
signal pt315 : STD_LOGIC_VECTOR(25 downto 0);
signal nxib26 : STD_LOGIC_VECTOR(25 downto 0);
signal pt79l2 : STD_LOGIC_VECTOR(25 downto 0);
signal pt433 : STD_LOGIC_VECTOR(25 downto 0);
signal xil9 : STD_LOGIC_VECTOR(25 downto 0);
signal npt79 : STD_LOGIC_VECTOR(25 downto 0);
signal pt57 : STD_LOGIC_VECTOR(25 downto 0);
signal pt93l2 : STD_LOGIC_VECTOR(25 downto 0);
signal npt315 : STD_LOGIC_VECTOR(25 downto 0);
signal pt173 : STD_LOGIC_VECTOR(24 downto 0);
signal pt127l1 : STD_LOGIC_VECTOR(24 downto 0);
signal npt81 : STD_LOGIC_VECTOR(24 downto 0);
signal pt215 : STD_LOGIC_VECTOR(25 downto 0);
signal pt81l3 : STD_LOGIC_VECTOR(25 downto 0);
signal npt433 : STD_LOGIC_VECTOR(25 downto 0);
signal pt45 : STD_LOGIC_VECTOR(24 downto 0);
signal nxil7 : STD_LOGIC_VECTOR(24 downto 0);
signal pt573 : STD_LOGIC_VECTOR(26 downto 0);
signal pt315l1 : STD_LOGIC_VECTOR(26 downto 0);
signal npt57 : STD_LOGIC_VECTOR(26 downto 0);
signal pt3775 : STD_LOGIC_VECTOR(28 downto 0);
signal pt127b29 : STD_LOGIC_VECTOR(28 downto 0);
signal pt57l6 : STD_LOGIC_VECTOR(28 downto 0);
signal pt391 : STD_LOGIC_VECTOR(25 downto 0);
signal pt173l1 : STD_LOGIC_VECTOR(25 downto 0);
signal pt45b26 : STD_LOGIC_VECTOR(25 downto 0);
signal pt627 : STD_LOGIC_VECTOR(26 downto 0);
signal npt93 : STD_LOGIC_VECTOR(26 downto 0);
signal pt45l4 : STD_LOGIC_VECTOR(26 downto 0);
signal pt677 : STD_LOGIC_VECTOR(26 downto 0);
signal pt79l3 : STD_LOGIC_VECTOR(26 downto 0);
signal pt45b27 : STD_LOGIC_VECTOR(26 downto 0);
```

```
signal pt753 : STD_LOGIC_VECTOR(26 downto 0);
signal pt45l2 : STD_LOGIC_VECTOR(26 downto 0);
signal pt6367 : STD_LOGIC_VECTOR(29 downto 0);
signal pt81l5 : STD_LOGIC_VECTOR(29 downto 0);
signal pt3775b30 : STD_LOGIC_VECTOR(29 downto 0);
signal pt2409 : STD_LOGIC_VECTOR(28 downto 0);
signal pt433l2 : STD_LOGIC_VECTOR(28 downto 0);
signal pt677b29 : STD_LOGIC_VECTOR(28 downto 0);
signal npt173 : STD_LOGIC_VECTOR(24 downto 0);
signal coef0 : STD_LOGIC_VECTOR(31 downto 0);
signal sum0_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef1 : STD_LOGIC_VECTOR(31 downto 0);
signal sum1 : STD_LOGIC_VECTOR(31 downto 0);
signal sum1_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt45l3 : STD_LOGIC_VECTOR(25 downto 0);
signal coef2 : STD_LOGIC_VECTOR(31 downto 0);
signal sum2 : STD_LOGIC_VECTOR(31 downto 0);
signal sum2_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt215l1 : STD_LOGIC_VECTOR(25 downto 0);
signal coef3 : STD_LOGIC_VECTOR(31 downto 0);
signal sum3 : STD_LOGIC_VECTOR(31 downto 0);
signal sum3_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef4 : STD_LOGIC_VECTOR(31 downto 0);
signal sum4 : STD_LOGIC_VECTOR(31 downto 0);
signal sum4_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef5 : STD_LOGIC_VECTOR(31 downto 0);
signal sum5 : STD_LOGIC_VECTOR(31 downto 0);
signal sum5_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef6 : STD_LOGIC_VECTOR(31 downto 0);
signal sum6 : STD_LOGIC_VECTOR(31 downto 0);
signal sum6_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt81l2 : STD_LOGIC_VECTOR(25 downto 0);
signal coef7 : STD_LOGIC_VECTOR(31 downto 0);
signal sum7 : STD_LOGIC_VECTOR(31 downto 0);
signal sum7_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt95l3 : STD_LOGIC_VECTOR(26 downto 0);
signal coef8 : STD_LOGIC_VECTOR(31 downto 0);
signal sum8 : STD_LOGIC_VECTOR(31 downto 0);
signal sum8_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt573l1 : STD_LOGIC_VECTOR(27 downto 0);
signal coef9 : STD_LOGIC_VECTOR(31 downto 0);
signal sum9 : STD_LOGIC_VECTOR(31 downto 0);
signal sum9_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt677l1 : STD_LOGIC_VECTOR(27 downto 0);
signal coef10 : STD_LOGIC_VECTOR(31 downto 0);
signal sum10 : STD_LOGIC_VECTOR(31 downto 0);
signal sum10_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt627l1 : STD_LOGIC_VECTOR(27 downto 0);
signal coef11 : STD_LOGIC_VECTOR(31 downto 0);
signal sum11 : STD_LOGIC_VECTOR(31 downto 0);
signal sum11_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt753 : STD_LOGIC_VECTOR(26 downto 0);
signal coef12 : STD_LOGIC_VECTOR(31 downto 0);
signal sum12 : STD_LOGIC_VECTOR(31 downto 0);
signal sum12_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt93l1 : STD_LOGIC_VECTOR(24 downto 0);
signal coef13 : STD_LOGIC_VECTOR(31 downto 0);
signal sum13 : STD_LOGIC_VECTOR(31 downto 0);
signal sum13_reg : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal pt95l4 : STD_LOGIC_VECTOR(27 downto 0);
signal coef14 : STD_LOGIC_VECTOR(31 downto 0);
signal sum14 : STD_LOGIC_VECTOR(31 downto 0);
signal sum14_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt391l3 : STD_LOGIC_VECTOR(28 downto 0);
signal coef15 : STD_LOGIC_VECTOR(31 downto 0);
signal sum15 : STD_LOGIC_VECTOR(31 downto 0);
signal sum15_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt2409l1 : STD_LOGIC_VECTOR(29 downto 0);
signal coef16 : STD_LOGIC_VECTOR(31 downto 0);
signal sum16 : STD_LOGIC_VECTOR(31 downto 0);
signal sum16_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef17 : STD_LOGIC_VECTOR(31 downto 0);
signal sum17 : STD_LOGIC_VECTOR(31 downto 0);
signal sum17_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt3775l1 : STD_LOGIC_VECTOR(29 downto 0);
signal coef18 : STD_LOGIC_VECTOR(31 downto 0);
signal sum18 : STD_LOGIC_VECTOR(31 downto 0);
signal sum18_reg : STD_LOGIC_VECTOR(31 downto 0);
signal xil13 : STD_LOGIC_VECTOR(29 downto 0);
signal coef19 : STD_LOGIC_VECTOR(31 downto 0);
signal sum19 : STD_LOGIC_VECTOR(31 downto 0);
signal sum19_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef20 : STD_LOGIC_VECTOR(31 downto 0);
signal sum20 : STD_LOGIC_VECTOR(31 downto 0);
signal sum20_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef21 : STD_LOGIC_VECTOR(31 downto 0);
signal sum21 : STD_LOGIC_VECTOR(31 downto 0);
signal sum21_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef22 : STD_LOGIC_VECTOR(31 downto 0);
signal sum22 : STD_LOGIC_VECTOR(31 downto 0);
signal sum22_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef23 : STD_LOGIC_VECTOR(31 downto 0);
signal sum23 : STD_LOGIC_VECTOR(31 downto 0);
signal sum23_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef24 : STD_LOGIC_VECTOR(31 downto 0);
signal sum24 : STD_LOGIC_VECTOR(31 downto 0);
signal sum24_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef25 : STD_LOGIC_VECTOR(31 downto 0);
signal sum25 : STD_LOGIC_VECTOR(31 downto 0);
signal sum25_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef26 : STD_LOGIC_VECTOR(31 downto 0);
signal sum26 : STD_LOGIC_VECTOR(31 downto 0);
signal sum26_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef27 : STD_LOGIC_VECTOR(31 downto 0);
signal sum27 : STD_LOGIC_VECTOR(31 downto 0);
signal sum27_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef28 : STD_LOGIC_VECTOR(31 downto 0);
signal sum28 : STD_LOGIC_VECTOR(31 downto 0);
signal sum28_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef29 : STD_LOGIC_VECTOR(31 downto 0);
signal sum29 : STD_LOGIC_VECTOR(31 downto 0);
signal sum29_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef30 : STD_LOGIC_VECTOR(31 downto 0);
signal sum30 : STD_LOGIC_VECTOR(31 downto 0);
signal sum30_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef31 : STD_LOGIC_VECTOR(31 downto 0);
signal sum31 : STD_LOGIC_VECTOR(31 downto 0);
signal sum31_reg : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal coef32 : STD_LOGIC_VECTOR(31 downto 0);
signal sum32 : STD_LOGIC_VECTOR(31 downto 0);
signal sum32_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef33 : STD_LOGIC_VECTOR(31 downto 0);
signal sum33 : STD_LOGIC_VECTOR(31 downto 0);
signal sum33_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef34 : STD_LOGIC_VECTOR(31 downto 0);
signal sum34 : STD_LOGIC_VECTOR(31 downto 0);
signal sum34_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef35 : STD_LOGIC_VECTOR(31 downto 0);
signal sum35 : STD_LOGIC_VECTOR(31 downto 0);
signal sum35_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef36 : STD_LOGIC_VECTOR(31 downto 0);
signal sum36 : STD_LOGIC_VECTOR(31 downto 0);
signal sum36_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef37 : STD_LOGIC_VECTOR(31 downto 0);
signal sum37 : STD_LOGIC_VECTOR(31 downto 0);
signal sum37_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef38 : STD_LOGIC_VECTOR(31 downto 0);
signal sum38 : STD_LOGIC_VECTOR(31 downto 0);
signal sum38_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef39 : STD_LOGIC_VECTOR(31 downto 0);
signal sum39 : STD_LOGIC_VECTOR(31 downto 0);
signal sum39_reg : STD_LOGIC_VECTOR(31 downto 0);

COMPONENT reg_16b is
     PORT( d : in STD_LOGIC_VECTOR(15 downto 0);
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(15 downto 0));
end COMPONENT;

COMPONENT reg_32b is
     PORT( d : in STD_LOGIC_VECTOR(31 downto 0);
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(31 downto 0));
end COMPONENT;

COMPONENT RCA24b is
     PORT( a : in STD_LOGIC_VECTOR(23 downto 0);
           b : in STD_LOGIC_VECTOR(23 downto 0);
           s : out STD_LOGIC_VECTOR(23 downto 0));
END COMPONENT;

COMPONENT RCA25b is
     PORT( a : in STD_LOGIC_VECTOR(24 downto 0);
           b : in STD_LOGIC_VECTOR(24 downto 0);
           s : out STD_LOGIC_VECTOR(24 downto 0));
END COMPONENT;

COMPONENT RCA26b is
     PORT( a : in STD_LOGIC_VECTOR(25 downto 0);
           b : in STD_LOGIC_VECTOR(25 downto 0);
           s : out STD_LOGIC_VECTOR(25 downto 0));
END COMPONENT;

COMPONENT RCA27b is
     PORT( a : in STD_LOGIC_VECTOR(26 downto 0);
```

```
           b : in STD_LOGIC_VECTOR(26 downto 0);
           s : out STD_LOGIC_VECTOR(26 downto 0));
END COMPONENT;

COMPONENT RCA29b is
     PORT( a : in STD_LOGIC_VECTOR(28 downto 0);
           b : in STD_LOGIC_VECTOR(28 downto 0);
           s : out STD_LOGIC_VECTOR(28 downto 0));
END COMPONENT;

COMPONENT RCA30b is
     PORT( a : in STD_LOGIC_VECTOR(29 downto 0);
           b : in STD_LOGIC_VECTOR(29 downto 0);
           s : out STD_LOGIC_VECTOR(29 downto 0));
END COMPONENT;

COMPONENT RCA32b is
     PORT( a : in STD_LOGIC_VECTOR(31 downto 0);
           b : in STD_LOGIC_VECTOR(31 downto 0);
           s : out STD_LOGIC_VECTOR(31 downto 0));
END COMPONENT;

begin

reg1: reg_16b PORT MAP(x,clk,reset,xi);

nxi<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&
xi(15)&xi))+1);
xil7<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(8 downto
0)&"0000000";
nxil5<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(10 downto 0)&"00000"))+1);
nxil4<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(11 downto 0)&"0000"))+1);
nxil1<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(14 downto 0)&"0"))+1);
xil1<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(14 downto 0)&"0";
nxib26<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(1
5)&xi(15)&xi(15)&xi(15)&xi))+1);
pt79l2<=pt79(23)&pt79(23)&pt79(21 downto 0)&"00";
xil9<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(6
downto 0)&"000000000";
npt79<=std_logic_vector(signed(not(pt79(23)&pt79(23)&pt79))+1);
pt93l2<=pt93(23)&pt93(23)&pt93(21 downto 0)&"00";
npt315<=std_logic_vector(signed(not(pt315))+1);
pt127l1<=pt127(23)&pt127(22 downto 0)&"0";
npt81<=std_logic_vector(signed(not(pt81(23)&pt81))+1);
pt81l3<=pt81(23)&pt81(23)&pt81(20 downto 0)&"000";
npt433<=std_logic_vector(signed(not(pt433))+1);
nxil7<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(8 downto 0)&"0000000"))+1);
pt315l1<=pt315(25)&pt315(24 downto 0)&"0";
npt57<=std_logic_vector(signed(not(pt57(25)&pt57))+1);
pt127b29<=pt127(23)&pt127(23)&pt127(23)&pt127(23)&pt127(23)&pt127;
pt57l6<=pt57(25)&pt57(25)&pt57(25)&pt57(19 downto 0)&"000000";
pt173l1<=pt173(24)&pt173(23 downto 0)&"0";
pt45b26<=pt45(24)&pt45;
npt93<=std_logic_vector(signed(not(pt93(23)&pt93(23)&pt93(23)&pt93))+1);
pt45l4<=pt45(24)&pt45(24)&pt45(20 downto 0)&"0000";
```

```
pt79l3<=pt79(23)&pt79(23)&pt79(23)&pt79(20 downto 0)&"000";
pt45b27<=pt45(24)&pt45(24)&pt45;
pt45l2<=pt45(24)&pt45(24)&pt45(22 downto 0)&"00";
pt81l5<=pt81(23)&pt81(23)&pt81(23)&pt81(23)&pt81(23)&pt81(23)&pt81(18 downto
0)&"00000";
pt3775b30<=pt3775(28)&pt3775;
pt433l2<=pt433(25)&pt433(25)&pt433(25)&pt433(23 downto 0)&"00";
pt677b29<=pt677(26)&pt677(26)&pt677;
npt173<=std_logic_vector(signed(not(pt173))+1);
pt45l3<=pt45(24)&pt45(21 downto 0)&"000";
pt215l1<=pt215(24 downto 0)&"0";
npt81l2<=std_logic_vector(signed(not(pt81(23)&pt81(23)&pt81(21 downto
0)&"00"))+1);
npt95l3<=std_logic_vector(signed(not(pt95(23)&pt95(23)&pt95(23)&pt95(20 downto
0)&"000"))+1);
npt573l1<=std_logic_vector(signed(not(pt573(26)&pt573(25 downto 0)&"0"))+1);
npt677l1<=std_logic_vector(signed(not(pt677(26)&pt677(25 downto 0)&"0"))+1);
npt627l1<=std_logic_vector(signed(not(pt627(26)&pt627(25 downto 0)&"0"))+1);
npt753<=std_logic_vector(signed(not(pt753))+1);
pt93l1<=pt93(23)&pt93(22 downto 0)&"0";
pt95l4<=pt95(23)&pt95(23)&pt95(23)&pt95(23)&pt95(19 downto 0)&"0000";
pt391l3<=pt391(25)&pt391(25)&pt391(25)&pt391(22 downto 0)&"000";
pt2409l1<=pt2409(28)&pt2409(27 downto 0)&"0";
pt3775l1<=pt3775(28)&pt3775(27 downto 0)&"0";
xil13<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(15)&xi(2 downto 0)&"0000000000000";

-- Partial terms computation

-- 127>>0 = -1<<0 +1<<7

adder0: RCA24b PORT MAP(nxi,xil7,pt127);

-- 95>>0 = -1<<5 +127<<0

adder1: RCA24b PORT MAP(pt127,nxil5,pt95);

-- 79>>0 = -1<<4 +95<<0

adder2: RCA24b PORT MAP(pt95,nxil4,pt79);

-- 93>>0 = -1<<1 +95<<0

adder3: RCA24b PORT MAP(pt95,nxil1,pt93);

-- 81>>0 = +1<<1 +79<<0

adder4: RCA24b PORT MAP(pt79,xil1,pt81);

-- 315>>0 = -1<<0 +79<<2

adder5: RCA26b PORT MAP(nxib26,pt79l2,pt315);

-- 433>>0 = +1<<9 -79<<0

adder6: RCA26b PORT MAP(npt79,xil9,pt433);

-- 57>>0 = +93<<2 -315<<0
```

```
adder7: RCA26b PORT MAP(npt315,pt93l2,pt57);

-- 173>>0 = +127<<1 -81<<0

adder8: RCA25b PORT MAP(npt81,pt127l1,pt173);

-- 215>>0 = +81<<3 -433<<0

adder9: RCA26b PORT MAP(npt433,pt81l3,pt215);

-- 45>>0 = -1<<7 +173<<0

adder10: RCA25b PORT MAP(pt173,nxil7,pt45);

-- 573>>0 = +315<<1 -57<<0

adder11: RCA27b PORT MAP(npt57,pt315l1,pt573);

-- 3775>>0 = +127<<0 +57<<6

adder12: RCA29b PORT MAP(pt127b29,pt57l6,pt3775);

-- 391>>0 = +173<<1 +45<<0

adder13: RCA26b PORT MAP(pt45b26,pt173l1,pt391);

-- 627>>0 = -93<<0 +45<<4

adder14: RCA27b PORT MAP(npt93,pt45l4,pt627);

-- 677>>0 = +79<<3 +45<<0

adder15: RCA27b PORT MAP(pt45b27,pt79l3,pt677);

-- 753>>0 = +45<<2 +573<<0

adder16: RCA27b PORT MAP(pt573,pt45l2,pt753);

-- 6367>>0 = +81<<5 +3775<<0

adder17: RCA30b PORT MAP(pt3775b30,pt81l5,pt6367);

-- 2409>>0 = +433<<2 +677<<0

adder18: RCA29b PORT MAP(pt677b29,pt433l2,pt2409);

-- Delay line computation

-- -173 = -173<<0

coef0 <=
npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt17
3;
reg2 : reg_32b PORT MAP(coef0,clk,reset,sum0_reg);

-- 315 = +315<<0

coef1 <= pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315;
adder19: RCA32b PORT MAP(sum0_reg, coef1, sum1);
```

```
reg3 : reg_32b PORT MAP(sum1,clk,reset,sum1_reg);

-- 360 = +45<<3

coef2 <= pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3;
adder20: RCA32b PORT MAP(sum1_reg, coef2, sum2);
reg4 : reg_32b PORT MAP(sum2,clk,reset,sum2_reg);

-- 430 = +215<<1

coef3 <=
pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1;
adder21: RCA32b PORT MAP(sum2_reg, coef3, sum3);
reg5 : reg_32b PORT MAP(sum3,clk,reset,sum3_reg);

-- 433 = +433<<0

coef4 <= pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433;
adder22: RCA32b PORT MAP(sum3_reg, coef4, sum4);
reg6 : reg_32b PORT MAP(sum4,clk,reset,sum4_reg);

-- 316 = +79<<2

coef5 <= pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2;
adder23: RCA32b PORT MAP(sum4_reg, coef5, sum5);
reg7 : reg_32b PORT MAP(sum5,clk,reset,sum5_reg);

-- 57 = +57<<0

coef6 <= pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57;
adder24: RCA32b PORT MAP(sum5_reg, coef6, sum6);
reg8 : reg_32b PORT MAP(sum6,clk,reset,sum6_reg);

-- -324 = -81<<2

coef7 <=
npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2;
adder25: RCA32b PORT MAP(sum6_reg, coef7, sum7);
reg9 : reg_32b PORT MAP(sum7,clk,reset,sum7_reg);

-- -760 = -95<<3

coef8 <= npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3;
adder26: RCA32b PORT MAP(sum7_reg, coef8, sum8);
reg10 : reg_32b PORT MAP(sum8,clk,reset,sum8_reg);

-- -1146 = -573<<1

coef9 <= npt573l1(27)&npt573l1(27)&npt573l1(27)&npt573l1(27)&npt573l1;
adder27: RCA32b PORT MAP(sum8_reg, coef9, sum9);
reg11 : reg_32b PORT MAP(sum9,clk,reset,sum9_reg);

-- -1354 = -677<<1

coef10 <= npt677l1(27)&npt677l1(27)&npt677l1(27)&npt677l1(27)&npt677l1;
adder28: RCA32b PORT MAP(sum9_reg, coef10, sum10);
reg12 : reg_32b PORT MAP(sum10,clk,reset,sum10_reg);

-- -1254 = -627<<1
```

```
coef11 <= npt627l1(27)&npt627l1(27)&npt627l1(27)&npt627l1(27)&npt627l1;
adder29: RCA32b PORT MAP(sum10_reg, coef11, sum11);
reg13 : reg_32b PORT MAP(sum11,clk,reset,sum11_reg);

-- -753 = -753<<0

coef12 <= npt753(26)&npt753(26)&npt753(26)&npt753(26)&npt753(26)&npt753;
adder30: RCA32b PORT MAP(sum11_reg, coef12, sum12);
reg14 : reg_32b PORT MAP(sum12,clk,reset,sum12_reg);

-- 186 = +93<<1

coef13 <=
pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l
1;
adder31: RCA32b PORT MAP(sum12_reg, coef13, sum13);
reg15 : reg_32b PORT MAP(sum13,clk,reset,sum13_reg);

-- 1520 = +95<<4

coef14 <= pt95l4(27)&pt95l4(27)&pt95l4(27)&pt95l4(27)&pt95l4;
adder32: RCA32b PORT MAP(sum13_reg, coef14, sum14);
reg16 : reg_32b PORT MAP(sum14,clk,reset,sum14_reg);

-- 3128 = +391<<3

coef15 <= pt391l3(28)&pt391l3(28)&pt391l3(28)&pt391l3;
adder33: RCA32b PORT MAP(sum14_reg, coef15, sum15);
reg17 : reg_32b PORT MAP(sum15,clk,reset,sum15_reg);

-- 4818 = +2409<<1

coef16 <= pt2409l1(29)&pt2409l1(29)&pt2409l1;
adder34: RCA32b PORT MAP(sum15_reg, coef16, sum16);
reg18 : reg_32b PORT MAP(sum16,clk,reset,sum16_reg);

-- 6367 = +6367<<0

coef17 <= pt6367(29)&pt6367(29)&pt6367;
adder35: RCA32b PORT MAP(sum16_reg, coef17, sum17);
reg19 : reg_32b PORT MAP(sum17,clk,reset,sum17_reg);

-- 7550 = +3775<<1

coef18 <= pt3775l1(29)&pt3775l1(29)&pt3775l1;
adder36: RCA32b PORT MAP(sum17_reg, coef18, sum18);
reg20 : reg_32b PORT MAP(sum18,clk,reset,sum18_reg);

-- 8192 = +1<<13

coef19 <= xil13(29)&xil13(29)&xil13;
adder37: RCA32b PORT MAP(sum18_reg, coef19, sum19);
reg21 : reg_32b PORT MAP(sum19,clk,reset,sum19_reg);

-- 8192 = +1<<13

coef20 <= xil13(29)&xil13(29)&xil13;
adder38: RCA32b PORT MAP(sum19_reg, coef20, sum20);
```

```
reg22 : reg_32b PORT MAP(sum20,clk,reset,sum20_reg);

-- 7550 = +3775<<1

coef21 <= pt3775l1(29)&pt3775l1(29)&pt3775l1;
adder39: RCA32b PORT MAP(sum20_reg, coef21, sum21);
reg23 : reg_32b PORT MAP(sum21,clk,reset,sum21_reg);

-- 6367 = +6367<<0

coef22 <= pt6367(29)&pt6367(29)&pt6367;
adder40: RCA32b PORT MAP(sum21_reg, coef22, sum22);
reg24 : reg_32b PORT MAP(sum22,clk,reset,sum22_reg);

-- 4818 = +2409<<1

coef23 <= pt2409l1(29)&pt2409l1(29)&pt2409l1;
adder41: RCA32b PORT MAP(sum22_reg, coef23, sum23);
reg25 : reg_32b PORT MAP(sum23,clk,reset,sum23_reg);

-- 3128 = +391<<3

coef24 <= pt391l3(28)&pt391l3(28)&pt391l3(28)&pt391l3;
adder42: RCA32b PORT MAP(sum23_reg, coef24, sum24);
reg26 : reg_32b PORT MAP(sum24,clk,reset,sum24_reg);

-- 1520 = +95<<4

coef25 <= pt95l4(27)&pt95l4(27)&pt95l4(27)&pt95l4(27)&pt95l4;
adder43: RCA32b PORT MAP(sum24_reg, coef25, sum25);
reg27 : reg_32b PORT MAP(sum25,clk,reset,sum25_reg);

-- 186 = +93<<1

coef26 <=
pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l1(24)&pt93l
1;
adder44: RCA32b PORT MAP(sum25_reg, coef26, sum26);
reg28 : reg_32b PORT MAP(sum26,clk,reset,sum26_reg);

-- -753 = -753<<0

coef27 <= npt753(26)&npt753(26)&npt753(26)&npt753(26)&npt753(26)&npt753;
adder45: RCA32b PORT MAP(sum26_reg, coef27, sum27);
reg29 : reg_32b PORT MAP(sum27,clk,reset,sum27_reg);

-- -1254 = -627<<1

coef28 <= npt627l1(27)&npt627l1(27)&npt627l1(27)&npt627l1(27)&npt627l1;
adder46: RCA32b PORT MAP(sum27_reg, coef28, sum28);
reg30 : reg_32b PORT MAP(sum28,clk,reset,sum28_reg);

-- -1354 = -677<<1

coef29 <= npt677l1(27)&npt677l1(27)&npt677l1(27)&npt677l1(27)&npt677l1;
adder47: RCA32b PORT MAP(sum28_reg, coef29, sum29);
reg31 : reg_32b PORT MAP(sum29,clk,reset,sum29_reg);

-- -1146 = -573<<1
```

```
coef30 <= npt573l1(27)&npt573l1(27)&npt573l1(27)&npt573l1(27)&npt573l1;
adder48: RCA32b PORT MAP(sum29_reg, coef30, sum30);
reg32 : reg_32b PORT MAP(sum30,clk,reset,sum30_reg);

-- -760 = -95<<3

coef31 <= npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3(26)&npt95l3;
adder49: RCA32b PORT MAP(sum30_reg, coef31, sum31);
reg33 : reg_32b PORT MAP(sum31,clk,reset,sum31_reg);

-- -324 = -81<<2

coef32 <=
npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2(25)&npt81l2;
adder50: RCA32b PORT MAP(sum31_reg, coef32, sum32);
reg34 : reg_32b PORT MAP(sum32,clk,reset,sum32_reg);

-- 57 = +57<<0

coef33 <= pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57(25)&pt57;
adder51: RCA32b PORT MAP(sum32_reg, coef33, sum33);
reg35 : reg_32b PORT MAP(sum33,clk,reset,sum33_reg);

-- 316 = +79<<2

coef34 <=
pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2(25)&pt79l2;
adder52: RCA32b PORT MAP(sum33_reg, coef34, sum34);
reg36 : reg_32b PORT MAP(sum34,clk,reset,sum34_reg);

-- 433 = +433<<0

coef35 <= pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433(25)&pt433;
adder53: RCA32b PORT MAP(sum34_reg, coef35, sum35);
reg37 : reg_32b PORT MAP(sum35,clk,reset,sum35_reg);

-- 430 = +215<<1

coef36 <=
pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1(25)&pt215l1;
adder54: RCA32b PORT MAP(sum35_reg, coef36, sum36);
reg38 : reg_32b PORT MAP(sum36,clk,reset,sum36_reg);

-- 360 = +45<<3

coef37 <=
pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3(25)&pt45l3;
adder55: RCA32b PORT MAP(sum36_reg, coef37, sum37);
reg39 : reg_32b PORT MAP(sum37,clk,reset,sum37_reg);

-- 315 = +315<<0

coef38 <= pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315(25)&pt315;
adder56: RCA32b PORT MAP(sum37_reg, coef38, sum38);
reg40 : reg_32b PORT MAP(sum38,clk,reset,sum38_reg);

-- -173 = -173<<0
```

```
coef39 <=
npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt173(24)&npt17
3;
adder57: RCA32b PORT MAP(sum38_reg, coef39, sum39);
reg41 : reg_32b PORT MAP(sum39,clk,reset,sum39_reg);

s <= sum39_reg;

end Behavioral;
```

APÊNDICE B – FILTRO FIR DE 120 TAPS - VHDL

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity fir10_16b is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           x : in  STD_LOGIC_VECTOR (15 downto 0);
           s : out  STD_LOGIC_VECTOR (31 downto 0));
end fir10_16b;

architecture Behavioral of fir10_16b is

signal xi : STD_LOGIC_VECTOR(15 downto 0);
signal pt129 : STD_LOGIC_VECTOR(24 downto 0);
signal xib25 : STD_LOGIC_VECTOR(24 downto 0);
signal xil7 : STD_LOGIC_VECTOR(24 downto 0);
signal pt513 : STD_LOGIC_VECTOR(26 downto 0);
signal xib27 : STD_LOGIC_VECTOR(26 downto 0);
signal xil9 : STD_LOGIC_VECTOR(26 downto 0);
signal pt1025 : STD_LOGIC_VECTOR(27 downto 0);
signal xib28 : STD_LOGIC_VECTOR(27 downto 0);
signal xil10 : STD_LOGIC_VECTOR(27 downto 0);
signal pt113 : STD_LOGIC_VECTOR(24 downto 0);
signal nxil4 : STD_LOGIC_VECTOR(24 downto 0);
signal pt133 : STD_LOGIC_VECTOR(24 downto 0);
signal xil2 : STD_LOGIC_VECTOR(24 downto 0);
signal pt903 : STD_LOGIC_VECTOR(26 downto 0);
signal npt129 : STD_LOGIC_VECTOR(26 downto 0);
signal pt129l3 : STD_LOGIC_VECTOR(26 downto 0);
signal pt545 : STD_LOGIC_VECTOR(26 downto 0);
signal xil5 : STD_LOGIC_VECTOR(26 downto 0);
signal pt3591 : STD_LOGIC_VECTOR(28 downto 0);
signal npt513 : STD_LOGIC_VECTOR(28 downto 0);
signal pt513l3 : STD_LOGIC_VECTOR(28 downto 0);
signal pt165 : STD_LOGIC_VECTOR(24 downto 0);
signal xil5b25 : STD_LOGIC_VECTOR(24 downto 0);
signal pt135 : STD_LOGIC_VECTOR(24 downto 0);
signal xil1 : STD_LOGIC_VECTOR(24 downto 0);
signal pt81 : STD_LOGIC_VECTOR(24 downto 0);
signal nxil5 : STD_LOGIC_VECTOR(24 downto 0);
signal pt887 : STD_LOGIC_VECTOR(26 downto 0);
signal nxil4b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt227 : STD_LOGIC_VECTOR(24 downto 0);
signal pt113l1 : STD_LOGIC_VECTOR(24 downto 0);
signal pt629 : STD_LOGIC_VECTOR(26 downto 0);
signal pt129l2 : STD_LOGIC_VECTOR(26 downto 0);
signal pt113b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt177 : STD_LOGIC_VECTOR(24 downto 0);
signal xil6 : STD_LOGIC_VECTOR(24 downto 0);
signal pt111 : STD_LOGIC_VECTOR(24 downto 0);
signal nxil1 : STD_LOGIC_VECTOR(24 downto 0);
signal pt19 : STD_LOGIC_VECTOR(26 downto 0);
signal npt513b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt133l2 : STD_LOGIC_VECTOR(26 downto 0);
signal pt323 : STD_LOGIC_VECTOR(25 downto 0);
signal npt129b26 : STD_LOGIC_VECTOR(25 downto 0);
```

```
signal pt113l2 : STD_LOGIC_VECTOR(25 downto 0);
signal pt391 : STD_LOGIC_VECTOR(26 downto 0);
signal nxil9 : STD_LOGIC_VECTOR(26 downto 0);
signal pt1139 : STD_LOGIC_VECTOR(27 downto 0);
signal pt513l1 : STD_LOGIC_VECTOR(27 downto 0);
signal pt113b28 : STD_LOGIC_VECTOR(27 downto 0);
signal pt265 : STD_LOGIC_VECTOR(25 downto 0);
signal nxi : STD_LOGIC_VECTOR(25 downto 0);
signal pt133l1 : STD_LOGIC_VECTOR(25 downto 0);
signal pt114 : STD_LOGIC_VECTOR(24 downto 0);
signal pt247 : STD_LOGIC_VECTOR(26 downto 0);
signal npt133l1 : STD_LOGIC_VECTOR(26 downto 0);
signal pt871 : STD_LOGIC_VECTOR(26 downto 0);
signal nxil5b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt1639 : STD_LOGIC_VECTOR(27 downto 0);
signal npt135 : STD_LOGIC_VECTOR(27 downto 0);
signal pt887l1 : STD_LOGIC_VECTOR(27 downto 0);
signal pt251 : STD_LOGIC_VECTOR(26 downto 0);
signal xil2b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt11 : STD_LOGIC_VECTOR(26 downto 0);
signal nxil3 : STD_LOGIC_VECTOR(26 downto 0);
signal pt347 : STD_LOGIC_VECTOR(25 downto 0);
signal xil9b26 : STD_LOGIC_VECTOR(25 downto 0);
signal npt165 : STD_LOGIC_VECTOR(25 downto 0);
signal pt611 : STD_LOGIC_VECTOR(26 downto 0);
signal pt81b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt265l1 : STD_LOGIC_VECTOR(26 downto 0);
signal pt171 : STD_LOGIC_VECTOR(26 downto 0);
signal pt133b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt19l1 : STD_LOGIC_VECTOR(26 downto 0);
signal pt59 : STD_LOGIC_VECTOR(24 downto 0);
signal pt114r1 : STD_LOGIC_VECTOR(24 downto 0);
signal pt285 : STD_LOGIC_VECTOR(25 downto 0);
signal npt227 : STD_LOGIC_VECTOR(25 downto 0);
signal pt623 : STD_LOGIC_VECTOR(26 downto 0);
signal pt111b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt830 : STD_LOGIC_VECTOR(26 downto 0);
signal npt114r1 : STD_LOGIC_VECTOR(26 downto 0);
signal pt870 : STD_LOGIC_VECTOR(26 downto 0);
signal nxib27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt185 : STD_LOGIC_VECTOR(24 downto 0);
signal xil3 : STD_LOGIC_VECTOR(24 downto 0);
signal pt311 : STD_LOGIC_VECTOR(26 downto 0);
signal xil6b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt233 : STD_LOGIC_VECTOR(25 downto 0);
signal nxil5b26 : STD_LOGIC_VECTOR(25 downto 0);
signal pt1205 : STD_LOGIC_VECTOR(27 downto 0);
signal pt545b28 : STD_LOGIC_VECTOR(27 downto 0);
signal pt165l2 : STD_LOGIC_VECTOR(27 downto 0);
signal pt1753 : STD_LOGIC_VECTOR(27 downto 0);
signal pt165l4 : STD_LOGIC_VECTOR(27 downto 0);
signal npt887 : STD_LOGIC_VECTOR(27 downto 0);
signal pt1687 : STD_LOGIC_VECTOR(27 downto 0);
signal npt129b28 : STD_LOGIC_VECTOR(27 downto 0);
signal pt227l3 : STD_LOGIC_VECTOR(27 downto 0);
signal pt41 : STD_LOGIC_VECTOR(24 downto 0);
signal pt1511 : STD_LOGIC_VECTOR(27 downto 0);
signal pt903b28 : STD_LOGIC_VECTOR(27 downto 0);
signal pt19l5 : STD_LOGIC_VECTOR(27 downto 0);
```

```
signal pt651 : STD_LOGIC_VECTOR(26 downto 0);
signal pt135b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt2425 : STD_LOGIC_VECTOR(28 downto 0);
signal pt135l4 : STD_LOGIC_VECTOR(28 downto 0);
signal pt265b29 : STD_LOGIC_VECTOR(28 downto 0);
signal pt749 : STD_LOGIC_VECTOR(27 downto 0);
signal npt1025 : STD_LOGIC_VECTOR(27 downto 0);
signal pt348 : STD_LOGIC_VECTOR(26 downto 0);
signal npt165b27 : STD_LOGIC_VECTOR(26 downto 0);
signal pt37 : STD_LOGIC_VECTOR(26 downto 0);
signal pt3753 : STD_LOGIC_VECTOR(28 downto 0);
signal pt81l1 : STD_LOGIC_VECTOR(28 downto 0);
signal pt1645 : STD_LOGIC_VECTOR(27 downto 0);
signal pt933 : STD_LOGIC_VECTOR(26 downto 0);
signal pt19l4 : STD_LOGIC_VECTOR(26 downto 0);
signal pt5313 : STD_LOGIC_VECTOR(29 downto 0);
signal pt129b30 : STD_LOGIC_VECTOR(29 downto 0);
signal pt81l6 : STD_LOGIC_VECTOR(29 downto 0);
signal pt10063 : STD_LOGIC_VECTOR(30 downto 0);
signal nxib31 : STD_LOGIC_VECTOR(30 downto 0);
signal pt629l4 : STD_LOGIC_VECTOR(30 downto 0);
signal pt2331 : STD_LOGIC_VECTOR(28 downto 0);
signal npt113 : STD_LOGIC_VECTOR(28 downto 0);
signal pt611l2 : STD_LOGIC_VECTOR(28 downto 0);
signal pt3563 : STD_LOGIC_VECTOR(28 downto 0);
signal pt513l2 : STD_LOGIC_VECTOR(28 downto 0);
signal pt1511b29 : STD_LOGIC_VECTOR(28 downto 0);
signal pt4357 : STD_LOGIC_VECTOR(29 downto 0);
signal pt135l5 : STD_LOGIC_VECTOR(29 downto 0);
signal pt37b30 : STD_LOGIC_VECTOR(29 downto 0);
signal pt14083 : STD_LOGIC_VECTOR(30 downto 0);
signal pt59b31 : STD_LOGIC_VECTOR(30 downto 0);
signal pt1753l3 : STD_LOGIC_VECTOR(30 downto 0);
signal pt1639l1 : STD_LOGIC_VECTOR(28 downto 0);
signal coef0 : STD_LOGIC_VECTOR(31 downto 0);
signal sum0_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef1 : STD_LOGIC_VECTOR(31 downto 0);
signal sum1 : STD_LOGIC_VECTOR(31 downto 0);
signal sum1_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt11 : STD_LOGIC_VECTOR(26 downto 0);
signal coef2 : STD_LOGIC_VECTOR(31 downto 0);
signal sum2 : STD_LOGIC_VECTOR(31 downto 0);
signal sum2_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt347 : STD_LOGIC_VECTOR(25 downto 0);
signal coef3 : STD_LOGIC_VECTOR(31 downto 0);
signal sum3 : STD_LOGIC_VECTOR(31 downto 0);
signal sum3_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt611 : STD_LOGIC_VECTOR(26 downto 0);
signal coef4 : STD_LOGIC_VECTOR(31 downto 0);
signal sum4 : STD_LOGIC_VECTOR(31 downto 0);
signal sum4_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt171l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef5 : STD_LOGIC_VECTOR(31 downto 0);
signal sum5 : STD_LOGIC_VECTOR(31 downto 0);
signal sum5_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt129l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef6 : STD_LOGIC_VECTOR(31 downto 0);
signal sum6 : STD_LOGIC_VECTOR(31 downto 0);
signal sum6_reg : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal coef7 : STD_LOGIC_VECTOR(31 downto 0);
signal sum7 : STD_LOGIC_VECTOR(31 downto 0);
signal sum7_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt59l2 : STD_LOGIC_VECTOR(24 downto 0);
signal coef8 : STD_LOGIC_VECTOR(31 downto 0);
signal sum8 : STD_LOGIC_VECTOR(31 downto 0);
signal sum8_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef9 : STD_LOGIC_VECTOR(31 downto 0);
signal sum9 : STD_LOGIC_VECTOR(31 downto 0);
signal sum9_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt135l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef10 : STD_LOGIC_VECTOR(31 downto 0);
signal sum10 : STD_LOGIC_VECTOR(31 downto 0);
signal sum10_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef11 : STD_LOGIC_VECTOR(31 downto 0);
signal sum11 : STD_LOGIC_VECTOR(31 downto 0);
signal sum11_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt81l1 : STD_LOGIC_VECTOR(24 downto 0);
signal coef12 : STD_LOGIC_VECTOR(31 downto 0);
signal sum12 : STD_LOGIC_VECTOR(31 downto 0);
signal sum12_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt623 : STD_LOGIC_VECTOR(26 downto 0);
signal coef13 : STD_LOGIC_VECTOR(31 downto 0);
signal sum13 : STD_LOGIC_VECTOR(31 downto 0);
signal sum13_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef14 : STD_LOGIC_VECTOR(31 downto 0);
signal sum14 : STD_LOGIC_VECTOR(31 downto 0);
signal sum14_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt830 : STD_LOGIC_VECTOR(26 downto 0);
signal coef15 : STD_LOGIC_VECTOR(31 downto 0);
signal sum15 : STD_LOGIC_VECTOR(31 downto 0);
signal sum15_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt227l1 : STD_LOGIC_VECTOR(25 downto 0);
signal coef16 : STD_LOGIC_VECTOR(31 downto 0);
signal sum16 : STD_LOGIC_VECTOR(31 downto 0);
signal sum16_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef17 : STD_LOGIC_VECTOR(31 downto 0);
signal sum17 : STD_LOGIC_VECTOR(31 downto 0);
signal sum17_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef18 : STD_LOGIC_VECTOR(31 downto 0);
signal sum18 : STD_LOGIC_VECTOR(31 downto 0);
signal sum18_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef19 : STD_LOGIC_VECTOR(31 downto 0);
signal sum19 : STD_LOGIC_VECTOR(31 downto 0);
signal sum19_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt177l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef20 : STD_LOGIC_VECTOR(31 downto 0);
signal sum20 : STD_LOGIC_VECTOR(31 downto 0);
signal sum20_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef21 : STD_LOGIC_VECTOR(31 downto 0);
signal sum21 : STD_LOGIC_VECTOR(31 downto 0);
signal sum21_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt251l1 : STD_LOGIC_VECTOR(26 downto 0);
signal coef22 : STD_LOGIC_VECTOR(31 downto 0);
signal sum22 : STD_LOGIC_VECTOR(31 downto 0);
signal sum22_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt133l3 : STD_LOGIC_VECTOR(27 downto 0);
signal coef23 : STD_LOGIC_VECTOR(31 downto 0);
signal sum23 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal sum23_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt311l2 : STD_LOGIC_VECTOR(27 downto 0);
signal coef24 : STD_LOGIC_VECTOR(31 downto 0);
signal sum24 : STD_LOGIC_VECTOR(31 downto 0);
signal sum24_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt233l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef25 : STD_LOGIC_VECTOR(31 downto 0);
signal sum25 : STD_LOGIC_VECTOR(31 downto 0);
signal sum25_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt111l1 : STD_LOGIC_VECTOR(24 downto 0);
signal coef26 : STD_LOGIC_VECTOR(31 downto 0);
signal sum26 : STD_LOGIC_VECTOR(31 downto 0);
signal sum26_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef27 : STD_LOGIC_VECTOR(31 downto 0);
signal sum27 : STD_LOGIC_VECTOR(31 downto 0);
signal sum27_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef28 : STD_LOGIC_VECTOR(31 downto 0);
signal sum28 : STD_LOGIC_VECTOR(31 downto 0);
signal sum28_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt323l2 : STD_LOGIC_VECTOR(27 downto 0);
signal coef29 : STD_LOGIC_VECTOR(31 downto 0);
signal sum29 : STD_LOGIC_VECTOR(31 downto 0);
signal sum29_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt391l1 : STD_LOGIC_VECTOR(26 downto 0);
signal coef30 : STD_LOGIC_VECTOR(31 downto 0);
signal sum30 : STD_LOGIC_VECTOR(31 downto 0);
signal sum30_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt19l3 : STD_LOGIC_VECTOR(26 downto 0);
signal coef31 : STD_LOGIC_VECTOR(31 downto 0);
signal sum31 : STD_LOGIC_VECTOR(31 downto 0);
signal sum31_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt1139 : STD_LOGIC_VECTOR(27 downto 0);
signal coef32 : STD_LOGIC_VECTOR(31 downto 0);
signal sum32 : STD_LOGIC_VECTOR(31 downto 0);
signal sum32_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt1753 : STD_LOGIC_VECTOR(27 downto 0);
signal coef33 : STD_LOGIC_VECTOR(31 downto 0);
signal sum33 : STD_LOGIC_VECTOR(31 downto 0);
signal sum33_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt1687 : STD_LOGIC_VECTOR(27 downto 0);
signal coef34 : STD_LOGIC_VECTOR(31 downto 0);
signal sum34 : STD_LOGIC_VECTOR(31 downto 0);
signal sum34_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt903 : STD_LOGIC_VECTOR(26 downto 0);
signal coef35 : STD_LOGIC_VECTOR(31 downto 0);
signal sum35 : STD_LOGIC_VECTOR(31 downto 0);
signal sum35_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt41l3 : STD_LOGIC_VECTOR(25 downto 0);
signal coef36 : STD_LOGIC_VECTOR(31 downto 0);
signal sum36 : STD_LOGIC_VECTOR(31 downto 0);
signal sum36_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef37 : STD_LOGIC_VECTOR(31 downto 0);
signal sum37 : STD_LOGIC_VECTOR(31 downto 0);
signal sum37_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt265l3 : STD_LOGIC_VECTOR(28 downto 0);
signal coef38 : STD_LOGIC_VECTOR(31 downto 0);
signal sum38 : STD_LOGIC_VECTOR(31 downto 0);
signal sum38_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt114l4 : STD_LOGIC_VECTOR(27 downto 0);
```

```
signal coef39 : STD_LOGIC_VECTOR(31 downto 0);
signal sum39 : STD_LOGIC_VECTOR(31 downto 0);
signal sum39_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef40 : STD_LOGIC_VECTOR(31 downto 0);
signal sum40 : STD_LOGIC_VECTOR(31 downto 0);
signal sum40_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt247l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef41 : STD_LOGIC_VECTOR(31 downto 0);
signal sum41 : STD_LOGIC_VECTOR(31 downto 0);
signal sum41_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt2425 : STD_LOGIC_VECTOR(28 downto 0);
signal coef42 : STD_LOGIC_VECTOR(31 downto 0);
signal sum42 : STD_LOGIC_VECTOR(31 downto 0);
signal sum42_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt749l2 : STD_LOGIC_VECTOR(28 downto 0);
signal coef43 : STD_LOGIC_VECTOR(31 downto 0);
signal sum43 : STD_LOGIC_VECTOR(31 downto 0);
signal sum43_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt2331 : STD_LOGIC_VECTOR(28 downto 0);
signal coef44 : STD_LOGIC_VECTOR(31 downto 0);
signal sum44 : STD_LOGIC_VECTOR(31 downto 0);
signal sum44_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt545 : STD_LOGIC_VECTOR(26 downto 0);
signal coef45 : STD_LOGIC_VECTOR(31 downto 0);
signal sum45 : STD_LOGIC_VECTOR(31 downto 0);
signal sum45_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt871l1 : STD_LOGIC_VECTOR(27 downto 0);
signal coef46 : STD_LOGIC_VECTOR(31 downto 0);
signal sum46 : STD_LOGIC_VECTOR(31 downto 0);
signal sum46_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef47 : STD_LOGIC_VECTOR(31 downto 0);
signal sum47 : STD_LOGIC_VECTOR(31 downto 0);
signal sum47_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt1025l2 : STD_LOGIC_VECTOR(29 downto 0);
signal coef48 : STD_LOGIC_VECTOR(31 downto 0);
signal sum48 : STD_LOGIC_VECTOR(31 downto 0);
signal sum48_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt348l3 : STD_LOGIC_VECTOR(28 downto 0);
signal coef49 : STD_LOGIC_VECTOR(31 downto 0);
signal sum49 : STD_LOGIC_VECTOR(31 downto 0);
signal sum49_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt37l2 : STD_LOGIC_VECTOR(26 downto 0);
signal coef50 : STD_LOGIC_VECTOR(31 downto 0);
signal sum50 : STD_LOGIC_VECTOR(31 downto 0);
signal sum50_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt3753 : STD_LOGIC_VECTOR(28 downto 0);
signal coef51 : STD_LOGIC_VECTOR(31 downto 0);
signal sum51 : STD_LOGIC_VECTOR(31 downto 0);
signal sum51_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt1645l2 : STD_LOGIC_VECTOR(29 downto 0);
signal coef52 : STD_LOGIC_VECTOR(31 downto 0);
signal sum52 : STD_LOGIC_VECTOR(31 downto 0);
signal sum52_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt3563l1 : STD_LOGIC_VECTOR(29 downto 0);
signal coef53 : STD_LOGIC_VECTOR(31 downto 0);
signal sum53 : STD_LOGIC_VECTOR(31 downto 0);
signal sum53_reg : STD_LOGIC_VECTOR(31 downto 0);
signal npt4357 : STD_LOGIC_VECTOR(29 downto 0);
signal coef54 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal sum54 : STD_LOGIC_VECTOR(31 downto 0);
signal sum54_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt933l1 : STD_LOGIC_VECTOR(27 downto 0);
signal coef55 : STD_LOGIC_VECTOR(31 downto 0);
signal sum55 : STD_LOGIC_VECTOR(31 downto 0);
signal sum55_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt5313l1 : STD_LOGIC_VECTOR(30 downto 0);
signal coef56 : STD_LOGIC_VECTOR(31 downto 0);
signal sum56 : STD_LOGIC_VECTOR(31 downto 0);
signal sum56_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt10063l1 : STD_LOGIC_VECTOR(31 downto 0);
signal coef57 : STD_LOGIC_VECTOR(31 downto 0);
signal sum57 : STD_LOGIC_VECTOR(31 downto 0);
signal sum57_reg : STD_LOGIC_VECTOR(31 downto 0);
signal pt14083l1 : STD_LOGIC_VECTOR(31 downto 0);
signal coef58 : STD_LOGIC_VECTOR(31 downto 0);
signal sum58 : STD_LOGIC_VECTOR(31 downto 0);
signal sum58_reg : STD_LOGIC_VECTOR(31 downto 0);
signal xil15 : STD_LOGIC_VECTOR(31 downto 0);
signal coef59 : STD_LOGIC_VECTOR(31 downto 0);
signal sum59 : STD_LOGIC_VECTOR(31 downto 0);
signal sum59_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef60 : STD_LOGIC_VECTOR(31 downto 0);
signal sum60 : STD_LOGIC_VECTOR(31 downto 0);
signal sum60_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef61 : STD_LOGIC_VECTOR(31 downto 0);
signal sum61 : STD_LOGIC_VECTOR(31 downto 0);
signal sum61_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef62 : STD_LOGIC_VECTOR(31 downto 0);
signal sum62 : STD_LOGIC_VECTOR(31 downto 0);
signal sum62_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef63 : STD_LOGIC_VECTOR(31 downto 0);
signal sum63 : STD_LOGIC_VECTOR(31 downto 0);
signal sum63_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef64 : STD_LOGIC_VECTOR(31 downto 0);
signal sum64 : STD_LOGIC_VECTOR(31 downto 0);
signal sum64_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef65 : STD_LOGIC_VECTOR(31 downto 0);
signal sum65 : STD_LOGIC_VECTOR(31 downto 0);
signal sum65_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef66 : STD_LOGIC_VECTOR(31 downto 0);
signal sum66 : STD_LOGIC_VECTOR(31 downto 0);
signal sum66_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef67 : STD_LOGIC_VECTOR(31 downto 0);
signal sum67 : STD_LOGIC_VECTOR(31 downto 0);
signal sum67_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef68 : STD_LOGIC_VECTOR(31 downto 0);
signal sum68 : STD_LOGIC_VECTOR(31 downto 0);
signal sum68_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef69 : STD_LOGIC_VECTOR(31 downto 0);
signal sum69 : STD_LOGIC_VECTOR(31 downto 0);
signal sum69_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef70 : STD_LOGIC_VECTOR(31 downto 0);
signal sum70 : STD_LOGIC_VECTOR(31 downto 0);
signal sum70_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef71 : STD_LOGIC_VECTOR(31 downto 0);
signal sum71 : STD_LOGIC_VECTOR(31 downto 0);
signal sum71_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef72 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal sum72 : STD_LOGIC_VECTOR(31 downto 0);
signal sum72_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef73 : STD_LOGIC_VECTOR(31 downto 0);
signal sum73 : STD_LOGIC_VECTOR(31 downto 0);
signal sum73_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef74 : STD_LOGIC_VECTOR(31 downto 0);
signal sum74 : STD_LOGIC_VECTOR(31 downto 0);
signal sum74_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef75 : STD_LOGIC_VECTOR(31 downto 0);
signal sum75 : STD_LOGIC_VECTOR(31 downto 0);
signal sum75_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef76 : STD_LOGIC_VECTOR(31 downto 0);
signal sum76 : STD_LOGIC_VECTOR(31 downto 0);
signal sum76_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef77 : STD_LOGIC_VECTOR(31 downto 0);
signal sum77 : STD_LOGIC_VECTOR(31 downto 0);
signal sum77_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef78 : STD_LOGIC_VECTOR(31 downto 0);
signal sum78 : STD_LOGIC_VECTOR(31 downto 0);
signal sum78_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef79 : STD_LOGIC_VECTOR(31 downto 0);
signal sum79 : STD_LOGIC_VECTOR(31 downto 0);
signal sum79_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef80 : STD_LOGIC_VECTOR(31 downto 0);
signal sum80 : STD_LOGIC_VECTOR(31 downto 0);
signal sum80_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef81 : STD_LOGIC_VECTOR(31 downto 0);
signal sum81 : STD_LOGIC_VECTOR(31 downto 0);
signal sum81_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef82 : STD_LOGIC_VECTOR(31 downto 0);
signal sum82 : STD_LOGIC_VECTOR(31 downto 0);
signal sum82_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef83 : STD_LOGIC_VECTOR(31 downto 0);
signal sum83 : STD_LOGIC_VECTOR(31 downto 0);
signal sum83_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef84 : STD_LOGIC_VECTOR(31 downto 0);
signal sum84 : STD_LOGIC_VECTOR(31 downto 0);
signal sum84_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef85 : STD_LOGIC_VECTOR(31 downto 0);
signal sum85 : STD_LOGIC_VECTOR(31 downto 0);
signal sum85_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef86 : STD_LOGIC_VECTOR(31 downto 0);
signal sum86 : STD_LOGIC_VECTOR(31 downto 0);
signal sum86_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef87 : STD_LOGIC_VECTOR(31 downto 0);
signal sum87 : STD_LOGIC_VECTOR(31 downto 0);
signal sum87_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef88 : STD_LOGIC_VECTOR(31 downto 0);
signal sum88 : STD_LOGIC_VECTOR(31 downto 0);
signal sum88_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef89 : STD_LOGIC_VECTOR(31 downto 0);
signal sum89 : STD_LOGIC_VECTOR(31 downto 0);
signal sum89_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef90 : STD_LOGIC_VECTOR(31 downto 0);
signal sum90 : STD_LOGIC_VECTOR(31 downto 0);
signal sum90_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef91 : STD_LOGIC_VECTOR(31 downto 0);
signal sum91 : STD_LOGIC_VECTOR(31 downto 0);
signal sum91_reg : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal coef92 : STD_LOGIC_VECTOR(31 downto 0);
signal sum92 : STD_LOGIC_VECTOR(31 downto 0);
signal sum92_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef93 : STD_LOGIC_VECTOR(31 downto 0);
signal sum93 : STD_LOGIC_VECTOR(31 downto 0);
signal sum93_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef94 : STD_LOGIC_VECTOR(31 downto 0);
signal sum94 : STD_LOGIC_VECTOR(31 downto 0);
signal sum94_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef95 : STD_LOGIC_VECTOR(31 downto 0);
signal sum95 : STD_LOGIC_VECTOR(31 downto 0);
signal sum95_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef96 : STD_LOGIC_VECTOR(31 downto 0);
signal sum96 : STD_LOGIC_VECTOR(31 downto 0);
signal sum96_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef97 : STD_LOGIC_VECTOR(31 downto 0);
signal sum97 : STD_LOGIC_VECTOR(31 downto 0);
signal sum97_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef98 : STD_LOGIC_VECTOR(31 downto 0);
signal sum98 : STD_LOGIC_VECTOR(31 downto 0);
signal sum98_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef99 : STD_LOGIC_VECTOR(31 downto 0);
signal sum99 : STD_LOGIC_VECTOR(31 downto 0);
signal sum99_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef100 : STD_LOGIC_VECTOR(31 downto 0);
signal sum100 : STD_LOGIC_VECTOR(31 downto 0);
signal sum100_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef101 : STD_LOGIC_VECTOR(31 downto 0);
signal sum101 : STD_LOGIC_VECTOR(31 downto 0);
signal sum101_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef102 : STD_LOGIC_VECTOR(31 downto 0);
signal sum102 : STD_LOGIC_VECTOR(31 downto 0);
signal sum102_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef103 : STD_LOGIC_VECTOR(31 downto 0);
signal sum103 : STD_LOGIC_VECTOR(31 downto 0);
signal sum103_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef104 : STD_LOGIC_VECTOR(31 downto 0);
signal sum104 : STD_LOGIC_VECTOR(31 downto 0);
signal sum104_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef105 : STD_LOGIC_VECTOR(31 downto 0);
signal sum105 : STD_LOGIC_VECTOR(31 downto 0);
signal sum105_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef106 : STD_LOGIC_VECTOR(31 downto 0);
signal sum106 : STD_LOGIC_VECTOR(31 downto 0);
signal sum106_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef107 : STD_LOGIC_VECTOR(31 downto 0);
signal sum107 : STD_LOGIC_VECTOR(31 downto 0);
signal sum107_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef108 : STD_LOGIC_VECTOR(31 downto 0);
signal sum108 : STD_LOGIC_VECTOR(31 downto 0);
signal sum108_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef109 : STD_LOGIC_VECTOR(31 downto 0);
signal sum109 : STD_LOGIC_VECTOR(31 downto 0);
signal sum109_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef110 : STD_LOGIC_VECTOR(31 downto 0);
signal sum110 : STD_LOGIC_VECTOR(31 downto 0);
signal sum110_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef111 : STD_LOGIC_VECTOR(31 downto 0);
signal sum111 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal sum111_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef112 : STD_LOGIC_VECTOR(31 downto 0);
signal sum112 : STD_LOGIC_VECTOR(31 downto 0);
signal sum112_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef113 : STD_LOGIC_VECTOR(31 downto 0);
signal sum113 : STD_LOGIC_VECTOR(31 downto 0);
signal sum113_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef114 : STD_LOGIC_VECTOR(31 downto 0);
signal sum114 : STD_LOGIC_VECTOR(31 downto 0);
signal sum114_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef115 : STD_LOGIC_VECTOR(31 downto 0);
signal sum115 : STD_LOGIC_VECTOR(31 downto 0);
signal sum115_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef116 : STD_LOGIC_VECTOR(31 downto 0);
signal sum116 : STD_LOGIC_VECTOR(31 downto 0);
signal sum116_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef117 : STD_LOGIC_VECTOR(31 downto 0);
signal sum117 : STD_LOGIC_VECTOR(31 downto 0);
signal sum117_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef118 : STD_LOGIC_VECTOR(31 downto 0);
signal sum118 : STD_LOGIC_VECTOR(31 downto 0);
signal sum118_reg : STD_LOGIC_VECTOR(31 downto 0);
signal coef119 : STD_LOGIC_VECTOR(31 downto 0);
signal sum119 : STD_LOGIC_VECTOR(31 downto 0);
signal sum119_reg : STD_LOGIC_VECTOR(31 downto 0);

COMPONENT reg_16b is
     PORT( d : in STD_LOGIC_VECTOR(15 downto 0);
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(15 downto 0));
end COMPONENT;

COMPONENT reg_32b is
     PORT( d : in STD_LOGIC_VECTOR(31 downto 0);
           clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR(31 downto 0));
end COMPONENT;

COMPONENT RCA25b is
     PORT( a : in STD_LOGIC_VECTOR(24 downto 0);
           b : in STD_LOGIC_VECTOR(24 downto 0);
           s : out STD_LOGIC_VECTOR(24 downto 0));
END COMPONENT;

COMPONENT RCA26b is
     PORT( a : in STD_LOGIC_VECTOR(25 downto 0);
           b : in STD_LOGIC_VECTOR(25 downto 0);
           s : out STD_LOGIC_VECTOR(25 downto 0));
END COMPONENT;

COMPONENT RCA27b is
     PORT( a : in STD_LOGIC_VECTOR(26 downto 0);
           b : in STD_LOGIC_VECTOR(26 downto 0);
           s : out STD_LOGIC_VECTOR(26 downto 0));
END COMPONENT;

COMPONENT RCA28b is
```

```
     PORT( a : in STD_LOGIC_VECTOR(27 downto 0);
           b : in STD_LOGIC_VECTOR(27 downto 0);
           s : out STD_LOGIC_VECTOR(27 downto 0));
END COMPONENT;

COMPONENT RCA29b is
     PORT( a : in STD_LOGIC_VECTOR(28 downto 0);
           b : in STD_LOGIC_VECTOR(28 downto 0);
           s : out STD_LOGIC_VECTOR(28 downto 0));
END COMPONENT;

COMPONENT RCA30b is
     PORT( a : in STD_LOGIC_VECTOR(29 downto 0);
           b : in STD_LOGIC_VECTOR(29 downto 0);
           s : out STD_LOGIC_VECTOR(29 downto 0));
END COMPONENT;

COMPONENT RCA31b is
     PORT( a : in STD_LOGIC_VECTOR(30 downto 0);
           b : in STD_LOGIC_VECTOR(30 downto 0);
           s : out STD_LOGIC_VECTOR(30 downto 0));
END COMPONENT;

COMPONENT RCA32b is
     PORT( a : in STD_LOGIC_VECTOR(31 downto 0);
           b : in STD_LOGIC_VECTOR(31 downto 0);
           s : out STD_LOGIC_VECTOR(31 downto 0));
END COMPONENT;

begin

reg1: reg_16b PORT MAP(x,clk,reset,xi);

xib25<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi;
xil7<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(8 downto
0)&"0000000";
xib27<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi;
xil9<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)
&xi(6 downto 0)&"000000000";
xib28<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi;
xil10<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(5 downto 0)&"0000000000";
nxil4<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(11 downto 0)&"0000"))+1);
xil2<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(13 downto
0)&"00";
npt129<=std_logic_vector(signed(not(pt129(24)&pt129(24)&pt129))+1);
pt129l3<=pt129(24)&pt129(24)&pt129(21 downto 0)&"000";
xil5<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)
&xi(10 downto 0)&"00000";
npt513<=std_logic_vector(signed(not(pt513(26)&pt513(26)&pt513))+1);
pt513l3<=pt513(26)&pt513(26)&pt513(23 downto 0)&"000";
xil5b25<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(10
downto 0)&"00000";
xil1<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(14 downto
0)&"0";
```

```
nxil5<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(10 downto 0)&"00000"))+1);
nxil4b27<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi
(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(11 downto 0)&"0000"))+1);
pt113l1<=pt113(23 downto 0)&"0";
pt129l2<=pt129(24)&pt129(24)&pt129(22 downto 0)&"00";
pt113b27<=pt113(24)&pt113(24)&pt113;
xil6<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(9 downto
0)&"000000";
nxil1<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(14 downto 0)&"0"))+1);
npt513b27<=std_logic_vector(signed(not(pt513))+1);
pt133l2<=pt133(24)&pt133(24)&pt133(22 downto 0)&"00";
npt129b26<=std_logic_vector(signed(not(pt129(24)&pt129))+1);
pt113l2<=pt113(24)&pt113(22 downto 0)&"00";
nxil9<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(15)&xi(15)&xi(6 downto 0)&"000000000"))+1);
pt513l1<=pt513(26)&pt513(25 downto 0)&"0";
pt113b28<=pt113(24)&pt113(24)&pt113(24)&pt113;
nxi<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&
xi(15)&xi(15)&xi(15)&xi))+1);
pt133l1<=pt133(24)&pt133(23 downto 0)&"0";
npt133l1<=std_logic_vector(signed(not(pt133(24)&pt133(24)&pt133(23 downto
0)&"0"))+1);
nxil5b27<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi
(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(10 downto 0)&"00000"))+1);
npt135<=std_logic_vector(signed(not(pt135(24)&pt135(24)&pt135(24)&pt135))+1);
pt887l1<=pt887(26)&pt887(25 downto 0)&"0";
xil2b27<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(
15)&xi(13 downto 0)&"00";
nxil3<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(15)&xi(15)&xi(12 downto 0)&"000"))+1);
xil9b26<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(
6 downto 0)&"000000000";
npt165<=std_logic_vector(signed(not(pt165(24)&pt165))+1);
pt81b27<=pt81(24)&pt81(24)&pt81;
pt265l1<=pt265(25)&pt265(24 downto 0)&"0";
pt133b27<=pt133(24)&pt133(24)&pt133;
pt19l1<=pt19(25 downto 0)&"0";
pt114r1<=pt114(24)&pt114(24 downto 1);
npt227<=std_logic_vector(signed(not(pt227(24)&pt227))+1);
pt111b27<=pt111(24)&pt111(24)&pt111;
npt114r1<=std_logic_vector(signed(not(pt114(24)&pt114(24)&pt114(24)&pt114(24
downto 1)))+1);
nxib27<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(1
5)&xi(15)&xi(15)&xi(15)&xi(15)&xi))+1);
xil3<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(12 downto
0)&"000";
xil6b27<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(
15)&xi(9 downto 0)&"000000";
nxil5b26<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi
(15)&xi(15)&xi(15)&xi(15)&xi(10 downto 0)&"00000"))+1);
pt545b28<=pt545(26)&pt545;
pt165l2<=pt165(24)&pt165(24)&pt165(24)&pt165(22 downto 0)&"00";
pt165l4<=pt165(24)&pt165(24)&pt165(24)&pt165(20 downto 0)&"0000";
npt887<=std_logic_vector(signed(not(pt887(26)&pt887))+1);
npt129b28<=std_logic_vector(signed(not(pt129(24)&pt129(24)&pt129(24)&pt129))+1);
pt227l3<=pt227(24)&pt227(24)&pt227(24)&pt227(21 downto 0)&"000";
pt903b28<=pt903(26)&pt903;
```

```
pt19l5<=pt19(26)&pt19(21 downto 0)&"00000";
pt135b27<=pt135(24)&pt135(24)&pt135;
pt135l4<=pt135(24)&pt135(24)&pt135(24)&pt135(24)&pt135(20 downto 0)&"0000";
pt265b29<=pt265(25)&pt265(25)&pt265(25)&pt265;
npt1025<=std_logic_vector(signed(not(pt1025))+1);
npt165b27<=std_logic_vector(signed(not(pt165(24)&pt165(24)&pt165))+1);
pt81l1<=pt81(24)&pt81(24)&pt81(24)&pt81(24)&pt81(23 downto 0)&"0";
pt19l4<=pt19(22 downto 0)&"0000";
pt129b30<=pt129(24)&pt129(24)&pt129(24)&pt129(24)&pt129(24)&pt129;
pt81l6<=pt81(24)&pt81(24)&pt81(24)&pt81(24)&pt81(24)&pt81(18 downto 0)&"000000";
nxib31<=std_logic_vector(signed(not(xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(1
5)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi))+1);
pt629l4<=pt629(26)&pt629(26)&pt629(26)&pt629(26)&pt629(22 downto 0)&"0000";
npt113<=std_logic_vector(signed(not(pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113)
)+1);
pt611l2<=pt611(26)&pt611(26)&pt611(24 downto 0)&"00";
pt513l2<=pt513(26)&pt513(26)&pt513(24 downto 0)&"00";
pt1511b29<=pt1511(27)&pt1511;
pt135l5<=pt135(24)&pt135(24)&pt135(24)&pt135(24)&pt135(24)&pt135(19 downto
0)&"00000";
pt37b30<=pt37(26)&pt37(26)&pt37(26)&pt37;
pt59b31<=pt59(24)&pt59(24)&pt59(24)&pt59(24)&pt59(24)&pt59(24)&pt59;
pt1753l3<=pt1753(27)&pt1753(27)&pt1753(27)&pt1753(24 downto 0)&"000";
pt1639l1<=pt1639(27)&pt1639(26 downto 0)&"0";
npt11<=std_logic_vector(signed(not(pt11))+1);
npt347<=std_logic_vector(signed(not(pt347))+1);
npt611<=std_logic_vector(signed(not(pt611))+1);
npt171l2<=std_logic_vector(signed(not(pt171(24 downto 0)&"00"))+1);
npt129l2<=std_logic_vector(signed(not(pt129(24)&pt129(24)&pt129(22 downto
0)&"00"))+1);
pt59l2<=pt59(22 downto 0)&"00";
pt135l2<=pt135(24)&pt135(24)&pt135(22 downto 0)&"00";
npt81l1<=std_logic_vector(signed(not(pt81(23 downto 0)&"0"))+1);
npt623<=std_logic_vector(signed(not(pt623))+1);
npt830<=std_logic_vector(signed(not(pt830))+1);
npt227l1<=std_logic_vector(signed(not(pt227(24)&pt227(23 downto 0)&"0"))+1);
pt177l2<=pt177(24)&pt177(24)&pt177(22 downto 0)&"00";
npt251l1<=std_logic_vector(signed(not(pt251(25 downto 0)&"0"))+1);
npt133l3<=std_logic_vector(signed(not(pt133(24)&pt133(24)&pt133(24)&pt133(21
downto 0)&"000"))+1);
npt311l2<=std_logic_vector(signed(not(pt311(26)&pt311(24 downto 0)&"00"))+1);
npt233l2<=std_logic_vector(signed(not(pt233(25)&pt233(23 downto 0)&"00"))+1);
npt111l1<=std_logic_vector(signed(not(pt111(23 downto 0)&"0"))+1);
pt323l2<=pt323(25)&pt323(25)&pt323(23 downto 0)&"00";
pt391l1<=pt391(25 downto 0)&"0";
npt19l3<=std_logic_vector(signed(not(pt19(23 downto 0)&"000"))+1);
npt1139<=std_logic_vector(signed(not(pt1139))+1);
npt1753<=std_logic_vector(signed(not(pt1753))+1);
npt1687<=std_logic_vector(signed(not(pt1687))+1);
npt903<=std_logic_vector(signed(not(pt903))+1);
pt41l3<=pt41(24)&pt41(21 downto 0)&"000";
pt265l3<=pt265(25)&pt265(25)&pt265(25)&pt265(22 downto 0)&"000";
pt114l4<=pt114(24)&pt114(24)&pt114(24)&pt114(20 downto 0)&"0000";
npt247l2<=std_logic_vector(signed(not(pt247(24 downto 0)&"00"))+1);
npt2425<=std_logic_vector(signed(not(pt2425))+1);
npt749l2<=std_logic_vector(signed(not(pt749(27)&pt749(25 downto 0)&"00"))+1);
npt2331<=std_logic_vector(signed(not(pt2331))+1);
npt545<=std_logic_vector(signed(not(pt545))+1);
pt871l1<=pt871(26)&pt871(25 downto 0)&"0";
```

```
pt1025l2<=pt1025(27)&pt1025(27)&pt1025(25 downto 0)&"00";
pt348l3<=pt348(26)&pt348(26)&pt348(23 downto 0)&"000";
npt37l2<=std_logic_vector(signed(not(pt37(24 downto 0)&"00"))+1);
npt3753<=std_logic_vector(signed(not(pt3753))+1);
npt1645l2<=std_logic_vector(signed(not(pt1645(27)&pt1645(27)&pt1645(25 downto
0)&"00"))+1);
npt3563l1<=std_logic_vector(signed(not(pt3563(28)&pt3563(27 downto 0)&"0"))+1);
npt4357<=std_logic_vector(signed(not(pt4357))+1);
pt933l1<=pt933(26)&pt933(25 downto 0)&"0";
pt5313l1<=pt5313(29)&pt5313(28 downto 0)&"0";
pt10063l1<=pt10063(30)&pt10063(29 downto 0)&"0";
pt14083l1<=pt14083(30)&pt14083(29 downto 0)&"0";
xil15<=xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15
)&xi(15)&xi(15)&xi(15)&xi(15)&xi(15)&xi(0 downto 0)&"000000000000000";

-- Partial terms computation
-- 129>>0 = +1<<0 +1<<7

adder0: RCA25b PORT MAP(xib25,xil7,pt129);

-- 513>>0 = +1<<0 +1<<9

adder1: RCA27b PORT MAP(xib27,xil9,pt513);

-- 1025>>0 = +1<<0 +1<<10

adder2: RCA28b PORT MAP(xib28,xil10,pt1025);

-- 113>>0 = -1<<4 +129<<0

adder3: RCA25b PORT MAP(pt129,nxil4,pt113);

-- 133>>0 = +1<<2 +129<<0

adder4: RCA25b PORT MAP(pt129,xil2,pt133);

-- 903>>0 = -129<<0 +129<<3

adder5: RCA27b PORT MAP(npt129,pt129l3,pt903);

-- 545>>0 = +1<<5 +513<<0

adder6: RCA27b PORT MAP(pt513,xil5,pt545);

-- 3591>>0 = -513<<0 +513<<3

adder7: RCA29b PORT MAP(npt513,pt513l3,pt3591);

-- 165>>0 = +1<<5 +133<<0

adder8: RCA25b PORT MAP(pt133,xil5b25,pt165);

-- 135>>0 = +1<<1 +133<<0

adder9: RCA25b PORT MAP(pt133,xil1,pt135);

-- 81>>0 = -1<<5 +113<<0

adder10: RCA25b PORT MAP(pt113,nxil5,pt81);
```

```
-- 887>>0 = -1<<4 +903<<0

adder11: RCA27b PORT MAP(pt903,nxil4b27,pt887);

-- 227>>0 = +1<<0 +113<<1

adder12: RCA25b PORT MAP(xib25,pt113l1,pt227);

-- 629>>0 = +129<<2 +113<<0

adder13: RCA27b PORT MAP(pt113b27,pt129l2,pt629);

-- 177>>0 = +1<<6 +113<<0

adder14: RCA25b PORT MAP(pt113,xil6,pt177);

-- 111>>0 = -1<<1 +113<<0

adder15: RCA25b PORT MAP(pt113,nxil1,pt111);

-- 19>>0 = -513<<0 +133<<2

adder16: RCA27b PORT MAP(npt513b27,pt133l2,pt19);

-- 323>>0 = -129<<0 +113<<2

adder17: RCA26b PORT MAP(npt129b26,pt113l2,pt323);

-- 391>>0 = -1<<9 +903<<0

adder18: RCA27b PORT MAP(pt903,nxil9,pt391);

-- 1139>>0 = +513<<1 +113<<0

adder19: RCA28b PORT MAP(pt113b28,pt513l1,pt1139);

-- 265>>0 = -1<<0 +133<<1

adder20: RCA26b PORT MAP(nxi,pt133l1,pt265);

-- 114>>0 = +1<<0 +113<<0

adder21: RCA25b PORT MAP(pt113,xib25,pt114);

-- 247>>0 = +513<<0 -133<<1

adder22: RCA27b PORT MAP(pt513,npt133l1,pt247);

-- 871>>0 = -1<<5 +903<<0

adder23: RCA27b PORT MAP(pt903,nxil5b27,pt871);

-- 1639>>0 = -135<<0 +887<<1

adder24: RCA28b PORT MAP(npt135,pt887l1,pt1639);

-- 251>>0 = +1<<2 +247<<0
```

```
adder25: RCA27b PORT MAP(pt247,xil2b27,pt251);

-- 11>>0 = -1<<3 +19<<0

adder26: RCA27b PORT MAP(pt19,nxil3,pt11);

-- 347>>0 = +1<<9 -165<<0

adder27: RCA26b PORT MAP(npt165,xil9b26,pt347);

-- 611>>0 = +81<<0 +265<<1

adder28: RCA27b PORT MAP(pt81b27,pt265l1,pt611);

-- 171>>0 = +133<<0 +19<<1

adder29: RCA27b PORT MAP(pt133b27,pt19l1,pt171);

-- 59>>0 = +1<<1 +114>>1

adder30: RCA25b PORT MAP(pt114r1,xil1,pt59);

-- 285>>0 = +1<<9 -227<<0

adder31: RCA26b PORT MAP(npt227,xil9b26,pt285);

-- 623>>0 = +1<<9 +111<<0

adder32: RCA27b PORT MAP(pt111b27,xil9,pt623);

-- 830>>0 = +887<<0 -114>>1

adder33: RCA27b PORT MAP(npt114r1,pt887,pt830);

-- 870>>0 = -1<<0 +871<<0

adder34: RCA27b PORT MAP(pt871,nxib27,pt870);

-- 185>>0 = +1<<3 +177<<0

adder35: RCA25b PORT MAP(pt177,xil3,pt185);

-- 311>>0 = +1<<6 +247<<0

adder36: RCA27b PORT MAP(pt247,xil6b27,pt311);

-- 233>>0 = -1<<5 +265<<0

adder37: RCA26b PORT MAP(pt265,nxil5b26,pt233);

-- 1205>>0 = +545<<0 +165<<2

adder38: RCA28b PORT MAP(pt545b28,pt165l2,pt1205);

-- 1753>>0 = +165<<4 -887<<0

adder39: RCA28b PORT MAP(npt887,pt165l4,pt1753);

-- 1687>>0 = -129<<0 +227<<3
```

```
adder40: RCA28b PORT MAP(npt129b28,pt227l3,pt1687);
-- 41>>0 = -1<<4 +114>>1
adder41: RCA25b PORT MAP(pt114r1,nxil4,pt41);
-- 1511>>0 = +903<<0 +19<<5
adder42: RCA28b PORT MAP(pt903b28,pt19l5,pt1511);
-- 651>>0 = +129<<2 +135<<0
adder43: RCA27b PORT MAP(pt135b27,pt129l2,pt651);
-- 2425>>0 = +135<<4 +265<<0
adder44: RCA29b PORT MAP(pt265b29,pt135l4,pt2425);
-- 749>>0 = -1025<<0 +887<<1
adder45: RCA28b PORT MAP(npt1025,pt887l1,pt749);
-- 348>>0 = +513<<0 -165<<0
adder46: RCA27b PORT MAP(npt165b27,pt513,pt348);
-- 37>>0 = -1<<0 +19<<1
adder47: RCA27b PORT MAP(nxib27,pt19l1,pt37);
-- 3753>>0 = +3591<<0 +81<<1
adder48: RCA29b PORT MAP(pt3591,pt81l1,pt3753);
-- 1645>>0 = -129<<0 +887<<1
adder49: RCA28b PORT MAP(npt129b28,pt887l1,pt1645);
-- 933>>0 = +629<<0 +19<<4
adder50: RCA27b PORT MAP(pt629,pt19l4,pt933);
-- 5313>>0 = +129<<0 +81<<6
adder51: RCA30b PORT MAP(pt129b30,pt81l6,pt5313);
-- 10063>>0 = -1<<0 +629<<4
adder52: RCA31b PORT MAP(nxib31,pt629l4,pt10063);
-- 2331>>0 = -113<<0 +611<<2
adder53: RCA29b PORT MAP(npt113,pt611l2,pt2331);
-- 3563>>0 = +513<<2 +1511<<0
adder54: RCA29b PORT MAP(pt1511b29,pt513l2,pt3563);
```

```
-- 4357>>0 = +135<<5 +37<<0

adder55: RCA30b PORT MAP(pt37b30,pt135l5,pt4357);

-- 14083>>0 = +59<<0 +1753<<3

adder56: RCA31b PORT MAP(pt59b31,pt1753l3,pt14083);

-- Delay line computation

-- 3278 = +1639<<1

coef0 <= pt1639l1(28)&pt1639l1(28)&pt1639l1(28)&pt1639l1;
reg2 : reg_32b PORT MAP(coef0,clk,reset,sum0_reg);

-- 251 = +251<<0

coef1 <= pt251(26)&pt251(26)&pt251(26)&pt251(26)&pt251(26)&pt251;
adder57: RCA32b PORT MAP(sum0_reg, coef1, sum1);
reg3 : reg_32b PORT MAP(sum1,clk,reset,sum1_reg);

-- -11 = -11<<0

coef2 <= npt11(26)&npt11(26)&npt11(26)&npt11(26)&npt11(26)&npt11;
adder58: RCA32b PORT MAP(sum1_reg, coef2, sum2);
reg4 : reg_32b PORT MAP(sum2,clk,reset,sum2_reg);

-- -347 = -347<<0

coef3 <= npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347;
adder59: RCA32b PORT MAP(sum2_reg, coef3, sum3);
reg5 : reg_32b PORT MAP(sum3,clk,reset,sum3_reg);

-- -611 = -611<<0

coef4 <= npt611(26)&npt611(26)&npt611(26)&npt611(26)&npt611(26)&npt611;
adder60: RCA32b PORT MAP(sum3_reg, coef4, sum4);
reg6 : reg_32b PORT MAP(sum4,clk,reset,sum4_reg);

-- -684 = -171<<2

coef5 <=
npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2;
adder61: RCA32b PORT MAP(sum4_reg, coef5, sum5);
reg7 : reg_32b PORT MAP(sum5,clk,reset,sum5_reg);

-- -516 = -129<<2

coef6 <=
npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2;
adder62: RCA32b PORT MAP(sum5_reg, coef6, sum6);
reg8 : reg_32b PORT MAP(sum6,clk,reset,sum6_reg);

-- -165 = -165<<0

coef7 <= npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165;
adder63: RCA32b PORT MAP(sum6_reg, coef7, sum7);
reg9 : reg_32b PORT MAP(sum7,clk,reset,sum7_reg);
```

```
-- 236 = +59<<2

coef8 <=
pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l
2;
adder64: RCA32b PORT MAP(sum7_reg, coef8, sum8);
reg10 : reg_32b PORT MAP(sum8,clk,reset,sum8_reg);

-- 513 = +513<<0

coef9 <= pt513(26)&pt513(26)&pt513(26)&pt513(26)&pt513(26)&pt513;
adder65: RCA32b PORT MAP(sum8_reg, coef9, sum9);
reg11 : reg_32b PORT MAP(sum9,clk,reset,sum9_reg);

-- 540 = +135<<2

coef10 <= pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2;
adder66: RCA32b PORT MAP(sum9_reg, coef10, sum10);
reg12 : reg_32b PORT MAP(sum10,clk,reset,sum10_reg);

-- 285 = +285<<0

coef11 <= pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285;
adder67: RCA32b PORT MAP(sum10_reg, coef11, sum11);
reg13 : reg_32b PORT MAP(sum11,clk,reset,sum11_reg);

-- -162 = -81<<1

coef12 <=
npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24
)&npt81l1;
adder68: RCA32b PORT MAP(sum11_reg, coef12, sum12);
reg14 : reg_32b PORT MAP(sum12,clk,reset,sum12_reg);

-- -623 = -623<<0

coef13 <= npt623(26)&npt623(26)&npt623(26)&npt623(26)&npt623(26)&npt623;
adder69: RCA32b PORT MAP(sum12_reg, coef13, sum13);
reg15 : reg_32b PORT MAP(sum13,clk,reset,sum13_reg);

-- -887 = -887<<0

coef14 <= npt887(27)&npt887(27)&npt887(27)&npt887(27)&npt887;
adder70: RCA32b PORT MAP(sum13_reg, coef14, sum14);
reg16 : reg_32b PORT MAP(sum14,clk,reset,sum14_reg);

-- -830 = -830<<0

coef15 <= npt830(26)&npt830(26)&npt830(26)&npt830(26)&npt830(26)&npt830;
adder71: RCA32b PORT MAP(sum14_reg, coef15, sum15);
reg17 : reg_32b PORT MAP(sum15,clk,reset,sum15_reg);

-- -454 = -227<<1

coef16 <=
npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt2
27l1;
adder72: RCA32b PORT MAP(sum15_reg, coef16, sum16);
reg18 : reg_32b PORT MAP(sum16,clk,reset,sum16_reg);
```

```
-- 113 = +113<<0

coef17 <=
pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113;
adder73: RCA32b PORT MAP(sum16_reg, coef17, sum17);
reg19 : reg_32b PORT MAP(sum17,clk,reset,sum17_reg);

-- 629 = +629<<0

coef18 <= pt629(26)&pt629(26)&pt629(26)&pt629(26)&pt629(26)&pt629;
adder74: RCA32b PORT MAP(sum17_reg, coef18, sum18);
reg20 : reg_32b PORT MAP(sum18,clk,reset,sum18_reg);

-- 870 = +870<<0

coef19 <= pt870(26)&pt870(26)&pt870(26)&pt870(26)&pt870(26)&pt870;
adder75: RCA32b PORT MAP(sum18_reg, coef19, sum19);
reg21 : reg_32b PORT MAP(sum19,clk,reset,sum19_reg);

-- 708 = +177<<2

coef20 <= pt177l2(26)&pt177l2(26)&pt177l2(26)&pt177l2(26)&pt177l2(26)&pt177l2;
adder76: RCA32b PORT MAP(sum19_reg, coef20, sum20);
reg22 : reg_32b PORT MAP(sum20,clk,reset,sum20_reg);

-- 185 = +185<<0

coef21 <=
pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185;
adder77: RCA32b PORT MAP(sum20_reg, coef21, sum21);
reg23 : reg_32b PORT MAP(sum21,clk,reset,sum21_reg);

-- -502 = -251<<1

coef22 <=
npt251l1(26)&npt251l1(26)&npt251l1(26)&npt251l1(26)&npt251l1(26)&npt251l1;
adder78: RCA32b PORT MAP(sum21_reg, coef22, sum22);
reg24 : reg_32b PORT MAP(sum22,clk,reset,sum22_reg);

-- -1064 = -133<<3

coef23 <= npt133l3(27)&npt133l3(27)&npt133l3(27)&npt133l3(27)&npt133l3;
adder79: RCA32b PORT MAP(sum22_reg, coef23, sum23);
reg25 : reg_32b PORT MAP(sum23,clk,reset,sum23_reg);

-- -1244 = -311<<2

coef24 <= npt311l2(27)&npt311l2(27)&npt311l2(27)&npt311l2(27)&npt311l2;
adder80: RCA32b PORT MAP(sum23_reg, coef24, sum24);
reg26 : reg_32b PORT MAP(sum24,clk,reset,sum24_reg);

-- -932 = -233<<2

coef25 <=
npt233l2(26)&npt233l2(26)&npt233l2(26)&npt233l2(26)&npt233l2(26)&npt233l2;
adder81: RCA32b PORT MAP(sum24_reg, coef25, sum25);
reg27 : reg_32b PORT MAP(sum25,clk,reset,sum25_reg);
```

```
-- -222 = -111<<1

coef26 <=
npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt1
11l1(24)&npt111l1;
adder82: RCA32b PORT MAP(sum25_reg, coef26, sum26);
reg28 : reg_32b PORT MAP(sum26,clk,reset,sum26_reg);

-- 608 = +19<<5

coef27 <= pt19l5(27)&pt19l5(27)&pt19l5(27)&pt19l5(27)&pt19l5;
adder83: RCA32b PORT MAP(sum26_reg, coef27, sum27);
reg29 : reg_32b PORT MAP(sum27,clk,reset,sum27_reg);

-- 1205 = +1205<<0

coef28 <= pt1205(27)&pt1205(27)&pt1205(27)&pt1205(27)&pt1205;
adder84: RCA32b PORT MAP(sum27_reg, coef28, sum28);
reg30 : reg_32b PORT MAP(sum28,clk,reset,sum28_reg);

-- 1292 = +323<<2

coef29 <= pt323l2(27)&pt323l2(27)&pt323l2(27)&pt323l2(27)&pt323l2;
adder85: RCA32b PORT MAP(sum28_reg, coef29, sum29);
reg31 : reg_32b PORT MAP(sum29,clk,reset,sum29_reg);

-- 782 = +391<<1

coef30 <= pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1;
adder86: RCA32b PORT MAP(sum29_reg, coef30, sum30);
reg32 : reg_32b PORT MAP(sum30,clk,reset,sum30_reg);

-- -152 = -19<<3

coef31 <= npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3;
adder87: RCA32b PORT MAP(sum30_reg, coef31, sum31);
reg33 : reg_32b PORT MAP(sum31,clk,reset,sum31_reg);

-- -1139 = -1139<<0

coef32 <= npt1139(27)&npt1139(27)&npt1139(27)&npt1139(27)&npt1139;
adder88: RCA32b PORT MAP(sum31_reg, coef32, sum32);
reg34 : reg_32b PORT MAP(sum32,clk,reset,sum32_reg);

-- -1753 = -1753<<0

coef33 <= npt1753(27)&npt1753(27)&npt1753(27)&npt1753(27)&npt1753;
adder89: RCA32b PORT MAP(sum32_reg, coef33, sum33);
reg35 : reg_32b PORT MAP(sum33,clk,reset,sum33_reg);

-- -1687 = -1687<<0

coef34 <= npt1687(27)&npt1687(27)&npt1687(27)&npt1687(27)&npt1687;
adder90: RCA32b PORT MAP(sum33_reg, coef34, sum34);
reg36 : reg_32b PORT MAP(sum34,clk,reset,sum34_reg);

-- -903 = -903<<0

coef35 <= npt903(26)&npt903(26)&npt903(26)&npt903(26)&npt903(26)&npt903;
```

```
adder91: RCA32b PORT MAP(sum34_reg, coef35, sum35);
reg37 : reg_32b PORT MAP(sum35,clk,reset,sum35_reg);

-- 328 = +41<<3

coef36 <=
pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3;
adder92: RCA32b PORT MAP(sum35_reg, coef36, sum36);
reg38 : reg_32b PORT MAP(sum36,clk,reset,sum36_reg);

-- 1511 = +1511<<0

coef37 <= pt1511(27)&pt1511(27)&pt1511(27)&pt1511(27)&pt1511;
adder93: RCA32b PORT MAP(sum36_reg, coef37, sum37);
reg39 : reg_32b PORT MAP(sum37,clk,reset,sum37_reg);

-- 2120 = +265<<3

coef38 <= pt265l3(28)&pt265l3(28)&pt265l3(28)&pt265l3;
adder94: RCA32b PORT MAP(sum37_reg, coef38, sum38);
reg40 : reg_32b PORT MAP(sum38,clk,reset,sum38_reg);

-- 1824 = +114<<4

coef39 <= pt114l4(27)&pt114l4(27)&pt114l4(27)&pt114l4(27)&pt114l4;
adder95: RCA32b PORT MAP(sum38_reg, coef39, sum39);
reg41 : reg_32b PORT MAP(sum39,clk,reset,sum39_reg);

-- 651 = +651<<0

coef40 <= pt651(26)&pt651(26)&pt651(26)&pt651(26)&pt651(26)&pt651;
adder96: RCA32b PORT MAP(sum39_reg, coef40, sum40);
reg42 : reg_32b PORT MAP(sum40,clk,reset,sum40_reg);

-- -988 = -247<<2

coef41 <=
npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2;
adder97: RCA32b PORT MAP(sum40_reg, coef41, sum41);
reg43 : reg_32b PORT MAP(sum41,clk,reset,sum41_reg);

-- -2425 = -2425<<0

coef42 <= npt2425(28)&npt2425(28)&npt2425(28)&npt2425;
adder98: RCA32b PORT MAP(sum41_reg, coef42, sum42);
reg44 : reg_32b PORT MAP(sum42,clk,reset,sum42_reg);

-- -2996 = -749<<2

coef43 <= npt749l2(28)&npt749l2(28)&npt749l2(28)&npt749l2;
adder99: RCA32b PORT MAP(sum42_reg, coef43, sum43);
reg45 : reg_32b PORT MAP(sum43,clk,reset,sum43_reg);

-- -2331 = -2331<<0

coef44 <= npt2331(28)&npt2331(28)&npt2331(28)&npt2331;
adder100: RCA32b PORT MAP(sum43_reg, coef44, sum44);
reg46 : reg_32b PORT MAP(sum44,clk,reset,sum44_reg);
```

```
-- -545 = -545<<0

coef45 <= npt545(26)&npt545(26)&npt545(26)&npt545(26)&npt545(26)&npt545;
adder101: RCA32b PORT MAP(sum44_reg, coef45, sum45);
reg47 : reg_32b PORT MAP(sum45,clk,reset,sum45_reg);

-- 1742 = +871<<1

coef46 <= pt871l1(27)&pt871l1(27)&pt871l1(27)&pt871l1(27)&pt871l1;
adder102: RCA32b PORT MAP(sum45_reg, coef46, sum46);
reg48 : reg_32b PORT MAP(sum46,clk,reset,sum46_reg);

-- 3591 = +3591<<0

coef47 <= pt3591(28)&pt3591(28)&pt3591(28)&pt3591;
adder103: RCA32b PORT MAP(sum46_reg, coef47, sum47);
reg49 : reg_32b PORT MAP(sum47,clk,reset,sum47_reg);

-- 4100 = +1025<<2

coef48 <= pt1025l2(29)&pt1025l2(29)&pt1025l2;
adder104: RCA32b PORT MAP(sum47_reg, coef48, sum48);
reg50 : reg_32b PORT MAP(sum48,clk,reset,sum48_reg);

-- 2784 = +348<<3

coef49 <= pt348l3(28)&pt348l3(28)&pt348l3(28)&pt348l3;
adder105: RCA32b PORT MAP(sum48_reg, coef49, sum49);
reg51 : reg_32b PORT MAP(sum49,clk,reset,sum49_reg);

-- -148 = -37<<2

coef50 <= npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2;
adder106: RCA32b PORT MAP(sum49_reg, coef50, sum50);
reg52 : reg_32b PORT MAP(sum50,clk,reset,sum50_reg);

-- -3753 = -3753<<0

coef51 <= npt3753(28)&npt3753(28)&npt3753(28)&npt3753;
adder107: RCA32b PORT MAP(sum50_reg, coef51, sum51);
reg53 : reg_32b PORT MAP(sum51,clk,reset,sum51_reg);

-- -6580 = -1645<<2

coef52 <= npt1645l2(29)&npt1645l2(29)&npt1645l2;
adder108: RCA32b PORT MAP(sum51_reg, coef52, sum52);
reg54 : reg_32b PORT MAP(sum52,clk,reset,sum52_reg);

-- -7126 = -3563<<1

coef53 <= npt3563l1(29)&npt3563l1(29)&npt3563l1;
adder109: RCA32b PORT MAP(sum52_reg, coef53, sum53);
reg55 : reg_32b PORT MAP(sum53,clk,reset,sum53_reg);

-- -4357 = -4357<<0

coef54 <= npt4357(29)&npt4357(29)&npt4357;
adder110: RCA32b PORT MAP(sum53_reg, coef54, sum54);
reg56 : reg_32b PORT MAP(sum54,clk,reset,sum54_reg);
```

```
-- 1866 = +933<<1

coef55 <= pt933l1(27)&pt933l1(27)&pt933l1(27)&pt933l1(27)&pt933l1;
adder111: RCA32b PORT MAP(sum54_reg, coef55, sum55);
reg57 : reg_32b PORT MAP(sum55,clk,reset,sum55_reg);

-- 10626 = +5313<<1

coef56 <= pt5313l1(30)&pt5313l1;
adder112: RCA32b PORT MAP(sum55_reg, coef56, sum56);
reg58 : reg_32b PORT MAP(sum56,clk,reset,sum56_reg);

-- 20126 = +10063<<1

coef57 <= pt10063l1;
adder113: RCA32b PORT MAP(sum56_reg, coef57, sum57);
reg59 : reg_32b PORT MAP(sum57,clk,reset,sum57_reg);

-- 28166 = +14083<<1

coef58 <= pt14083l1;
adder114: RCA32b PORT MAP(sum57_reg, coef58, sum58);
reg60 : reg_32b PORT MAP(sum58,clk,reset,sum58_reg);

-- 32768 = +1<<15

coef59 <= xil15;
adder115: RCA32b PORT MAP(sum58_reg, coef59, sum59);
reg61 : reg_32b PORT MAP(sum59,clk,reset,sum59_reg);

-- 32768 = +1<<15

coef60 <= xil15;
adder116: RCA32b PORT MAP(sum59_reg, coef60, sum60);
reg62 : reg_32b PORT MAP(sum60,clk,reset,sum60_reg);

-- 28166 = +14083<<1

coef61 <= pt14083l1;
adder117: RCA32b PORT MAP(sum60_reg, coef61, sum61);
reg63 : reg_32b PORT MAP(sum61,clk,reset,sum61_reg);

-- 20126 = +10063<<1

coef62 <= pt10063l1;
adder118: RCA32b PORT MAP(sum61_reg, coef62, sum62);
reg64 : reg_32b PORT MAP(sum62,clk,reset,sum62_reg);

-- 10626 = +5313<<1

coef63 <= pt5313l1(30)&pt5313l1;
adder119: RCA32b PORT MAP(sum62_reg, coef63, sum63);
reg65 : reg_32b PORT MAP(sum63,clk,reset,sum63_reg);

-- 1866 = +933<<1

coef64 <= pt933l1(27)&pt933l1(27)&pt933l1(27)&pt933l1(27)&pt933l1;
adder120: RCA32b PORT MAP(sum63_reg, coef64, sum64);
```

```
reg66 : reg_32b PORT MAP(sum64,clk,reset,sum64_reg);

-- -4357 = -4357<<0

coef65 <= npt4357(29)&npt4357(29)&npt4357;
adder121: RCA32b PORT MAP(sum64_reg, coef65, sum65);
reg67 : reg_32b PORT MAP(sum65,clk,reset,sum65_reg);

-- -7126 = -3563<<1

coef66 <= npt3563l1(29)&npt3563l1(29)&npt3563l1;
adder122: RCA32b PORT MAP(sum65_reg, coef66, sum66);
reg68 : reg_32b PORT MAP(sum66,clk,reset,sum66_reg);

-- -6580 = -1645<<2

coef67 <= npt1645l2(29)&npt1645l2(29)&npt1645l2;
adder123: RCA32b PORT MAP(sum66_reg, coef67, sum67);
reg69 : reg_32b PORT MAP(sum67,clk,reset,sum67_reg);

-- -3753 = -3753<<0

coef68 <= npt3753(28)&npt3753(28)&npt3753(28)&npt3753;
adder124: RCA32b PORT MAP(sum67_reg, coef68, sum68);
reg70 : reg_32b PORT MAP(sum68,clk,reset,sum68_reg);

-- -148 = -37<<2

coef69 <= npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2(26)&npt37l2;
adder125: RCA32b PORT MAP(sum68_reg, coef69, sum69);
reg71 : reg_32b PORT MAP(sum69,clk,reset,sum69_reg);

-- 2784 = +348<<3

coef70 <= pt348l3(28)&pt348l3(28)&pt348l3(28)&pt348l3;
adder126: RCA32b PORT MAP(sum69_reg, coef70, sum70);
reg72 : reg_32b PORT MAP(sum70,clk,reset,sum70_reg);

-- 4100 = +1025<<2

coef71 <= pt1025l2(29)&pt1025l2(29)&pt1025l2;
adder127: RCA32b PORT MAP(sum70_reg, coef71, sum71);
reg73 : reg_32b PORT MAP(sum71,clk,reset,sum71_reg);

-- 3591 = +3591<<0

coef72 <= pt3591(28)&pt3591(28)&pt3591(28)&pt3591;
adder128: RCA32b PORT MAP(sum71_reg, coef72, sum72);
reg74 : reg_32b PORT MAP(sum72,clk,reset,sum72_reg);

-- 1742 = +871<<1

coef73 <= pt871l1(27)&pt871l1(27)&pt871l1(27)&pt871l1(27)&pt871l1;
adder129: RCA32b PORT MAP(sum72_reg, coef73, sum73);
reg75 : reg_32b PORT MAP(sum73,clk,reset,sum73_reg);

-- -545 = -545<<0

coef74 <= npt545(26)&npt545(26)&npt545(26)&npt545(26)&npt545(26)&npt545;
```

```
adder130: RCA32b PORT MAP(sum73_reg, coef74, sum74);
reg76 : reg_32b PORT MAP(sum74,clk,reset,sum74_reg);

-- -2331 = -2331<<0

coef75 <= npt2331(28)&npt2331(28)&npt2331(28)&npt2331;
adder131: RCA32b PORT MAP(sum74_reg, coef75, sum75);
reg77 : reg_32b PORT MAP(sum75,clk,reset,sum75_reg);

-- -2996 = -749<<2

coef76 <= npt749l2(28)&npt749l2(28)&npt749l2(28)&npt749l2;
adder132: RCA32b PORT MAP(sum75_reg, coef76, sum76);
reg78 : reg_32b PORT MAP(sum76,clk,reset,sum76_reg);

-- -2425 = -2425<<0

coef77 <= npt2425(28)&npt2425(28)&npt2425(28)&npt2425;
adder133: RCA32b PORT MAP(sum76_reg, coef77, sum77);
reg79 : reg_32b PORT MAP(sum77,clk,reset,sum77_reg);

-- -988 = -247<<2

coef78 <=
npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2(26)&npt247l2;
adder134: RCA32b PORT MAP(sum77_reg, coef78, sum78);
reg80 : reg_32b PORT MAP(sum78,clk,reset,sum78_reg);

-- 651 = +651<<0

coef79 <= pt651(26)&pt651(26)&pt651(26)&pt651(26)&pt651(26)&pt651;
adder135: RCA32b PORT MAP(sum78_reg, coef79, sum79);
reg81 : reg_32b PORT MAP(sum79,clk,reset,sum79_reg);

-- 1824 = +114<<4

coef80 <= pt114l4(27)&pt114l4(27)&pt114l4(27)&pt114l4(27)&pt114l4;
adder136: RCA32b PORT MAP(sum79_reg, coef80, sum80);
reg82 : reg_32b PORT MAP(sum80,clk,reset,sum80_reg);

-- 2120 = +265<<3

coef81 <= pt265l3(28)&pt265l3(28)&pt265l3(28)&pt265l3;
adder137: RCA32b PORT MAP(sum80_reg, coef81, sum81);
reg83 : reg_32b PORT MAP(sum81,clk,reset,sum81_reg);

-- 1511 = +1511<<0

coef82 <= pt1511(27)&pt1511(27)&pt1511(27)&pt1511(27)&pt1511;
adder138: RCA32b PORT MAP(sum81_reg, coef82, sum82);
reg84 : reg_32b PORT MAP(sum82,clk,reset,sum82_reg);

-- 328 = +41<<3

coef83 <=
pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3(25)&pt41l3;
adder139: RCA32b PORT MAP(sum82_reg, coef83, sum83);
reg85 : reg_32b PORT MAP(sum83,clk,reset,sum83_reg);
```

```
-- -903 = -903<<0

coef84 <= npt903(26)&npt903(26)&npt903(26)&npt903(26)&npt903(26)&npt903;
adder140: RCA32b PORT MAP(sum83_reg, coef84, sum84);
reg86 : reg_32b PORT MAP(sum84,clk,reset,sum84_reg);

-- -1687 = -1687<<0

coef85 <= npt1687(27)&npt1687(27)&npt1687(27)&npt1687(27)&npt1687;
adder141: RCA32b PORT MAP(sum84_reg, coef85, sum85);
reg87 : reg_32b PORT MAP(sum85,clk,reset,sum85_reg);

-- -1753 = -1753<<0

coef86 <= npt1753(27)&npt1753(27)&npt1753(27)&npt1753(27)&npt1753;
adder142: RCA32b PORT MAP(sum85_reg, coef86, sum86);
reg88 : reg_32b PORT MAP(sum86,clk,reset,sum86_reg);

-- -1139 = -1139<<0

coef87 <= npt1139(27)&npt1139(27)&npt1139(27)&npt1139(27)&npt1139;
adder143: RCA32b PORT MAP(sum86_reg, coef87, sum87);
reg89 : reg_32b PORT MAP(sum87,clk,reset,sum87_reg);

-- -152 = -19<<3

coef88 <= npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3(26)&npt19l3;
adder144: RCA32b PORT MAP(sum87_reg, coef88, sum88);
reg90 : reg_32b PORT MAP(sum88,clk,reset,sum88_reg);

-- 782 = +391<<1

coef89 <= pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1(26)&pt391l1;
adder145: RCA32b PORT MAP(sum88_reg, coef89, sum89);
reg91 : reg_32b PORT MAP(sum89,clk,reset,sum89_reg);

-- 1292 = +323<<2

coef90 <= pt323l2(27)&pt323l2(27)&pt323l2(27)&pt323l2(27)&pt323l2;
adder146: RCA32b PORT MAP(sum89_reg, coef90, sum90);
reg92 : reg_32b PORT MAP(sum90,clk,reset,sum90_reg);

-- 1205 = +1205<<0

coef91 <= pt1205(27)&pt1205(27)&pt1205(27)&pt1205(27)&pt1205;
adder147: RCA32b PORT MAP(sum90_reg, coef91, sum91);
reg93 : reg_32b PORT MAP(sum91,clk,reset,sum91_reg);

-- 608 = +19<<5

coef92 <= pt19l5(27)&pt19l5(27)&pt19l5(27)&pt19l5(27)&pt19l5;
adder148: RCA32b PORT MAP(sum91_reg, coef92, sum92);
reg94 : reg_32b PORT MAP(sum92,clk,reset,sum92_reg);

-- -222 = -111<<1

coef93 <=
npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt111l1(24)&npt1
11l1(24)&npt111l1;
```

```
adder149: RCA32b PORT MAP(sum92_reg, coef93, sum93);
reg95 : reg_32b PORT MAP(sum93,clk,reset,sum93_reg);

-- -932 = -233<<2

coef94 <=
npt2331l2(26)&npt2331l2(26)&npt2331l2(26)&npt2331l2(26)&npt2331l2(26)&npt2331l2;
adder150: RCA32b PORT MAP(sum93_reg, coef94, sum94);
reg96 : reg_32b PORT MAP(sum94,clk,reset,sum94_reg);

-- -1244 = -311<<2

coef95 <= npt3111l2(27)&npt3111l2(27)&npt3111l2(27)&npt3111l2(27)&npt3111l2;
adder151: RCA32b PORT MAP(sum94_reg, coef95, sum95);
reg97 : reg_32b PORT MAP(sum95,clk,reset,sum95_reg);

-- -1064 = -133<<3

coef96 <= npt1331l3(27)&npt1331l3(27)&npt1331l3(27)&npt1331l3(27)&npt1331l3;
adder152: RCA32b PORT MAP(sum95_reg, coef96, sum96);
reg98 : reg_32b PORT MAP(sum96,clk,reset,sum96_reg);

-- -502 = -251<<1

coef97 <=
npt2511l1(26)&npt2511l1(26)&npt2511l1(26)&npt2511l1(26)&npt2511l1(26)&npt2511l1;
adder153: RCA32b PORT MAP(sum96_reg, coef97, sum97);
reg99 : reg_32b PORT MAP(sum97,clk,reset,sum97_reg);

-- 185 = +185<<0

coef98 <=
pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185(24)&pt185;
adder154: RCA32b PORT MAP(sum97_reg, coef98, sum98);
reg100 : reg_32b PORT MAP(sum98,clk,reset,sum98_reg);

-- 708 = +177<<2

coef99 <= pt1771l2(26)&pt1771l2(26)&pt1771l2(26)&pt1771l2(26)&pt1771l2(26)&pt1771l2;
adder155: RCA32b PORT MAP(sum98_reg, coef99, sum99);
reg101 : reg_32b PORT MAP(sum99,clk,reset,sum99_reg);

-- 870 = +870<<0

coef100 <= pt870(26)&pt870(26)&pt870(26)&pt870(26)&pt870(26)&pt870;
adder156: RCA32b PORT MAP(sum99_reg, coef100, sum100);
reg102 : reg_32b PORT MAP(sum100,clk,reset,sum100_reg);

-- 629 = +629<<0

coef101 <= pt629(26)&pt629(26)&pt629(26)&pt629(26)&pt629(26)&pt629;
adder157: RCA32b PORT MAP(sum100_reg, coef101, sum101);
reg103 : reg_32b PORT MAP(sum101,clk,reset,sum101_reg);

-- 113 = +113<<0

coef102 <=
pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113(24)&pt113;
adder158: RCA32b PORT MAP(sum101_reg, coef102, sum102);
```

```
reg104 : reg_32b PORT MAP(sum102,clk,reset,sum102_reg);

-- -454 = -227<<1

coef103 <=
npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt227l1(25)&npt2
27l1;
adder159: RCA32b PORT MAP(sum102_reg, coef103, sum103);
reg105 : reg_32b PORT MAP(sum103,clk,reset,sum103_reg);

-- -830 = -830<<0

coef104 <= npt830(26)&npt830(26)&npt830(26)&npt830(26)&npt830(26)&npt830;
adder160: RCA32b PORT MAP(sum103_reg, coef104, sum104);
reg106 : reg_32b PORT MAP(sum104,clk,reset,sum104_reg);

-- -887 = -887<<0

coef105 <= npt887(27)&npt887(27)&npt887(27)&npt887(27)&npt887;
adder161: RCA32b PORT MAP(sum104_reg, coef105, sum105);
reg107 : reg_32b PORT MAP(sum105,clk,reset,sum105_reg);

-- -623 = -623<<0

coef106 <= npt623(26)&npt623(26)&npt623(26)&npt623(26)&npt623(26)&npt623;
adder162: RCA32b PORT MAP(sum105_reg, coef106, sum106);
reg108 : reg_32b PORT MAP(sum106,clk,reset,sum106_reg);

-- -162 = -81<<1

coef107 <=
npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24)&npt81l1(24
)&npt81l1;
adder163: RCA32b PORT MAP(sum106_reg, coef107, sum107);
reg109 : reg_32b PORT MAP(sum107,clk,reset,sum107_reg);

-- 285 = +285<<0

coef108 <= pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285(25)&pt285;
adder164: RCA32b PORT MAP(sum107_reg, coef108, sum108);
reg110 : reg_32b PORT MAP(sum108,clk,reset,sum108_reg);

-- 540 = +135<<2

coef109 <= pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2(26)&pt135l2;
adder165: RCA32b PORT MAP(sum108_reg, coef109, sum109);
reg111 : reg_32b PORT MAP(sum109,clk,reset,sum109_reg);

-- 513 = +513<<0

coef110 <= pt513(26)&pt513(26)&pt513(26)&pt513(26)&pt513(26)&pt513;
adder166: RCA32b PORT MAP(sum109_reg, coef110, sum110);
reg112 : reg_32b PORT MAP(sum110,clk,reset,sum110_reg);

-- 236 = +59<<2
coef111 <=
pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l2(24)&pt59l
2;
adder167: RCA32b PORT MAP(sum110_reg, coef111, sum111);
```

```
reg113 : reg_32b PORT MAP(sum111,clk,reset,sum111_reg);

-- -165 = -165<<0

coef112 <=
npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165(25)&npt165;
adder168: RCA32b PORT MAP(sum111_reg, coef112, sum112);
reg114 : reg_32b PORT MAP(sum112,clk,reset,sum112_reg);

-- -516 = -129<<2

coef113 <=
npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2(26)&npt129l2;
adder169: RCA32b PORT MAP(sum112_reg, coef113, sum113);
reg115 : reg_32b PORT MAP(sum113,clk,reset,sum113_reg);

-- -684 = -171<<2

coef114 <=
npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2(26)&npt171l2;
adder170: RCA32b PORT MAP(sum113_reg, coef114, sum114);
reg116 : reg_32b PORT MAP(sum114,clk,reset,sum114_reg);

-- -611 = -611<<0

coef115 <= npt611(26)&npt611(26)&npt611(26)&npt611(26)&npt611(26)&npt611;
adder171: RCA32b PORT MAP(sum114_reg, coef115, sum115);
reg117 : reg_32b PORT MAP(sum115,clk,reset,sum115_reg);

-- -347 = -347<<0

coef116 <=
npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347(25)&npt347;
adder172: RCA32b PORT MAP(sum115_reg, coef116, sum116);
reg118 : reg_32b PORT MAP(sum116,clk,reset,sum116_reg);

-- -11 = -11<<0

coef117 <= npt11(26)&npt11(26)&npt11(26)&npt11(26)&npt11(26)&npt11;
adder173: RCA32b PORT MAP(sum116_reg, coef117, sum117);
reg119 : reg_32b PORT MAP(sum117,clk,reset,sum117_reg);

-- 251 = +251<<0

coef118 <= pt251(26)&pt251(26)&pt251(26)&pt251(26)&pt251(26)&pt251;
adder174: RCA32b PORT MAP(sum117_reg, coef118, sum118);
reg120 : reg_32b PORT MAP(sum118,clk,reset,sum118_reg);

-- 3278 = +1639<<1

coef119 <= pt1639l1(28)&pt1639l1(28)&pt1639l1(28)&pt1639l1;
adder175: RCA32b PORT MAP(sum118_reg, coef119, sum119);
reg121 : reg_32b PORT MAP(sum119,clk,reset,sum119_reg);

s <= sum119_reg;

end Behavioral;
```

APÊNDICE C – DIRETIVAS PROCESSADOR VLIW – “FERRAMENTA 1”

```
set_directive_pipeline -off "advance_pc"
set_directive_pipeline -off "fetch"
set_directive_pipeline -off "decode"
set_directive_pipeline -off "execute"
set_directive_pipeline -off "alu"
set_directive_pipeline -off "mult"
set_directive_pipeline -off "ctrl"
set_directive_pipeline -off "mem"
set_directive_pipeline -off "writeback"
set_directive_pipeline -off "r_vex_core"
set_directive_resource  "advance_pc" *next_pc
set_directive_resource  "decode" *syllable
set_directive_resource  "execute" *reg_gr
set_directive_resource  "alu" op_code
set_directive_resource  "r_vex_core" *done
set_directive_resource  "r_vex_core" *cycles
set_directive_loop_flatten "fetch/FORL_0"
set_directive_loop_flatten "decode/FORL_1"
set_directive_loop_flatten "execute/FORL_2"
set_directive_loop_flatten "execute/FORL_3"
set_directive_loop_flatten "execute/FORL_4"
set_directive_loop_flatten "writeback/FORL_5"
set_directive_loop_flatten "r_vex_core/FORL_6"
set_directive_loop_flatten "r_vex_core/WHILE_0"
set_directive_resource -core RAM_1P "r_vex_core" next_pc
set_directive_resource -core RAM_1P "r_vex_core" pc_goto
set_directive_resource -core RAM_1P "r_vex_core" c
set_directive_resource -core RAM_1P "r_vex_core" mem_rdata
set_directive_array_map  "execute" A_op
set_directive_array_map  "execute" B_op
set_directive_array_map  "execute" mem_wdata
set_directive_array_map  "execute" br
set_directive_array_map  "r_vex_core" reg_gr
set_directive_array_map  "r_vex_core" reg_br
set_directive_array_map  "r_vex_core" syllable
set_directive_array_map  "r_vex_core" op_code
set_directive_array_map  "r_vex_core" imm_op
set_directive_array_map  "r_vex_core" dest_gr
set_directive_array_map  "r_vex_core" src1_gr
set_directive_array_map  "r_vex_core" src2_gr
set_directive_array_map  "r_vex_core" dest_br
set_directive_array_map  "r_vex_core" short_imm_data
set_directive_array_map  "r_vex_core" branch_imm_data
set_directive_array_map  "r_vex_core" long_imm_data
set_directive_array_map  "r_vex_core" dmem
set_directive_array_map  "r_vex_core" alu_result
set_directive_array_map  "r_vex_core" mult_result
set_directive_array_map "fetch" imem
```