UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

JOÃO MATEUS COLET

ENSAIOS DE ACELERAÇÃO COMPUTACIONAL DE ALGUNS ALGORITMOS CLÁSSICOS UTILIZANDO-SE FPGA

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2021

JOÃO MATEUS COLET

ENSAIOS DE ACELERAÇÃO COMPUTACIONAL DE ALGUNS ALGORITMOS CLÁSSICOS UTILIZANDO-SE FPGA

Trabalho de conclusão de curso apresentado ao curso de Engenharia De Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para a obtenção do título de Bacharel em Engenharia De Computação.

Orientador: Prof. Msc. André Macário Barros
Universidade Tecnológica Federal do Paraná

PATO BRANCO
2021

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO - TCC

ENSAIOS DE ACELERAÇÃO COMPUTACIONAL DE ALGUNS ALGORITMOS CLÁSSICOS UTILIZANDO-SE FPGA

Por

João Mateus Colet

Monografia apresentada às 19 horas 40 min. do dia 17 de agosto de 2021 como requisito parcial, para conclusão do Curso de Engenharia da Computação da Universidade Tecnológica Federal do Paraná, Campus Pato Branco. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação e conferidas, bem como achadas conforme, as alterações indicadas pela Banca Examinadora, o trabalho de conclusão de curso foi considerado APROVADO.

Banca examinadora:

Prof. Dr. Marco Antonio de Castro Barbosa	Membro
Prof. Dr. Kleiton De Morais Sousa	Membro
Prof. Msc. André Macário Barros	Orientador
Profa. Dra. Viviane Dal Molin de Souza	Professor(a) responsável TCCII

Dedico este trabalho a minha família, em especial aos meus pais os quais sempre me apoiaram na caminhada acadêmica.

AGRADECIMENTOS

Agradeço primeiramente meus pais Jacir Colet e Andréia Roberta Rossi Colet por não medirem esforços em me proporcionar as melhores condições possíveis, possibilitando minha total dedicação durante meus estudos. Ao restante da minha família e meus amigos por todo suporte oferecido. A todos os meus professores os quais contribuíram de alguma forma na evolução do meu conhecimento, em especial ao Prof. Msc. André Macário Barros que em sua orientação prestou todo o auxílio possível na elaboração deste trabalho e a UTFPR por disponibilizar os materiais necessários para implementação deste trabalho.

O ensino deve inspirar os estudantes a descobrir por si mesmos, a questionar quando não concordarem, a procurar alternativas se acham que existem outras melhores, a revisar as grandes conquistas do passado e aprender porque algo lhes interessa. (CHOMSKY, Noam).

RESUMO

COLET, João Mateus. Ensaios de aceleração computacional de alguns algoritmos clássicos utilizando-se FPGA. 2021. 68 f. Trabalho de conclusão de curso – curso de Engenharia De Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

Com o aumento da complexidade dos algoritmos computacionais, as tecnologias convencionais passaram a apresentar resultados insatisfatórios em relação ao tempo de processamento. Novas tecnologias para aceleração de código surgiram para suprir as desvantagens dos computadores que executam código sequencial, entre elas as FPGAs. Pelo método proposto por Estrin (1960), denominado arquitetura F+V, composta por um processador sequencial e uma FPGA, buscou-se combinar as vantagens de ambas tecnologias ao explorar o paralelismo nos problemas propostos. Este trabalho verificou por meio de ensaios científicos a aceleração obtida por meio desta proposta dos seguintes problemas clássicos: a detecção de números primos, o algoritmo de ordenação *bubble sort* e o algoritmo de renderização de imagens *Z buffer*. Foram obtidos resultados de aceleramento na ordem de até 200 mil, 9 e 1.3 vezes respectivamente. Mostrando assim os benefícios em utilizar as FPGAs em uma arquitetura F+V.

Palavras-chave: FPGA. VHDL. Aceleração de código. Arquitetura F+V.

ABSTRACT

COLET, João Mateus. Computational acceleration essays of some classical algorithms utilizing FPGA. 2021. 68 f. Trabalho de conclusão de curso – curso de Engenharia De Computação, Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

With the increasing complexity of computational algorithms, conventional technologies began to show unsatisfactory results regarding processing time. New technologies for code acceleration have emerged to address the disadvantages of sequential-code computers, including the FPGAs. Through the re-reading of an old concept, called F+V architecture, proposed by Estrin (1960), consisting of a sequential processor and an FPGA, it were combined the advantages of both technologies by exploring parallelism in the proposed problems. This work verified through scientific essays the acceleration achieved through this proposal of the classical problems: the detector of prime numbers, the bubble sort sorting algorithm and the Z buffer image rendering algorithm. Acceleration results were achieved in the order of up to 200 thousand, 9 and 1.3 times respectively. Showing the advantages of use FPGA in F+V architecture

Keywords: FPGA. VHDL. Code acceleration. F+V architecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BSP	*Board Support Package*
CPU	*Central Processing Unit*
EDA	*Eletronic Design Automation*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GPU	*Graphics Processing Unit*
HDTV	*High-Definition Television*
JTAG	*Joint Test Action Group*
PLD	*Programmable logic devices*
RTL	*Register Transfer Level*
SDK	*Software Development Kit*
USB	*Universal Serial Bus*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
XPS	Xilinx *Plataform Studio*

LISTA DE SÍMBOLOS

O	Notação Grande-O
B	Bytes
G	Giga
M	Mega
m	mili
u	micro
n	nano
Hz	Hertz
s	segundos

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Um dos grandes problemas na computação é o tempo de processamento. Quanto mais complexa e massiva a solução para uma determinada tarefa mais processamento precisa ser efetuado. Portanto, complexidade de uma tarefa está diretamente associada ao tempo de seu processamento. E, tal tempo, pode vir a impossibilitar a execução em computadores para determinadas tarefas. Isso ocorre por causa do modelo dos computadores usado até hoje se basear nos princípios da máquina de Turing e na arquitetura de Von Neumann. Tal arquitetura consiste em executar uma instrução por vez, resultando em um maior tempo para resolução de problemas complexos. Com raras exceções, todos os computadores de hoje obedecem a essa mesma arquitetura e são conhecidos como máquinas de Von Neumann (STALLINGS, 2010).

Computadores que obedecem a arquitetura de Von Neumann têm a característica de consumir um ou mais ciclos de *clock* para cada instrução realizada. O *clock* por definição é a quantidade de ciclos que um processador realiza em um segundo, ciclos estes mensurados em Hertz (Hz). Logo, para acelerar o processamento uma das possíveis medidas necessárias é aumentar a frequência do *clock* da máquina. Porém, esse aumento tem um limite físico, precisando assim de novas técnicas para conseguir acelerar ainda mais o processamento.

Com o avanço da tecnologia, surgiram recursos que contribuíram com o aumento do desempenho computacional sem depender fundamentalmente do aumento do *clock*, como por exemplo tecnologia MMX, adição da *cache* interna, *pipeline* de instrução e a tecnologia *multicores*. Além disso começaram a se desenvolver novos métodos para acelerar um problema. Com a ajuda das redes de computadores surgiu o paralelismo de máquinas. Em quase todos os casos, a computação em *cluster* é usada para programação paralela. No *cluster* um único programa que precisa de um trabalho de computação intenso é executado em paralelo em várias máquinas (TANENBAUM; STEEN, 2007). O *cluster* é composto por máquinas, cada uma delas operando de forma autônoma executando parte de uma tarefa encontrando-se interligadas em rede local, por este motivo é categorizado como máquinas fracamente ligadas. No entanto, o fato das máquinas estarem ligadas em rede local faz surgir um gargalo de desempenho, observado entre a velocidade de operação do computador e a velocidade de propagação da informação na rede local, dado que mesmo o computador tendo capacidade de processamento sobrando ele perde desempenho ao se deparar com uma rede congestionada (STALLINGS, 2010).

Em contrapartida, a tecnologia das *Graphics Processing Unit* (GPU) formam uma arquitetura fortemente ligada que, ao contrário das várias máquinas ligadas em rede na computação em *cluster*, o gargalo de rede é eliminado (STALLINGS, 2010). A arquitetura de uma GPU consiste em acoplar várias *Central Processing Unit* (CPU) em uma única placa onde as mesmas operam de forma paralela. Na GPU a tarefa dividida de forma que cada CPU processe uma parte do algoritmo. Para efeito de comparação, até o ano de 2009, a velocidade

de processamento das GPUs já era dez vezes maior quando comparadas às CPUs (KIRK; HWU, 2010). Para de obter uma aceleração no processamento por uma GPU é preferível que sejam selecionadas aplicações onde grande parte de sua região crítica de processamento seja paralelizável, caso contrário o desempenho não será tão superior àquele obtido em um computador com uma única CPU.

Em meados de 1970, começaram a ser introduzidos os *Programmable logic devices* (PLD), em contraste ao conceito da máquina de Turing difundida na grande maioria dos dispositivos. O PLD baseia-se em circuitos lógicos que comportam a possibilidade de serem programados. Tais circuitos lógicos podem ser combinacionais e sequenciais, no entanto, são de natureza simples com sua reprogramabilidade limitada. Um projeto utilizando PLDs consiste em programar um circuito lógico para realizar uma tarefa específica e não programar a tarefa de forma compatível com uma máquina de Turing, para que a mesma conseguisse executá-la. Porém as primeiras tecnologias envolvendo PLD eram pouco voláteis no quesito de reprogramação. Uma vez que o dispositivo fosse programado ele não poderia ser alterado, precisando assim substituir o dispositivo por um novo contendo a nova programação, caso uma alteração fosse necessária. Os PLDs começaram a ser aperfeiçoados em meados de 1980 quando foram introduzidas as *Field Programmable Gate Array* (FPGA). Entre as melhorias presentes na FPGA quando comparada aos primeiros PLDs, pode se listar a capacidade de reprogramação do circuito e a possibilidade de implementar projetos complexos com mais facilidade (PEDRONI, 2008). As FPGAs são dispositivos lógicos e sua arquitetura consiste em uma matriz de células lógicas e comutadores programáveis. Uma célula lógica pode ser projetada para executar uma função simples e um comutador pode fornecer a interconexões entre as células lógicas (CHU, 2008). É possível implementar, dessa forma, um algoritmo computacional não sequencial obtendo-se múltiplos graus de paralelismo quando comparados com os *clusters* e as GPUs. Alguns desses graus de paralelismo sequer podem ser concebidos em GPUs ou em programação convencional. Essa tecnologia promove, portanto, uma aceleração que viabiliza projetos que tem demonstrado sua superioridade em termos de velocidade de processamento. Obtendo tempos de execução menores em comparação com a mesma tarefa quando executada em computadores sequenciais e até mesmo em GPUs devido à sua forma diferente de concepção do processamento. São exemplos de aplicações das FPGAs onde o alvo principal é composto de projetos complexos que requeiram processamento computacional intenso: transceptores Gigabit, *switches* de alta complexidade, *High-Definition Television* (HDTV), *wireless networks* e outras aplicações de telecomunicações (PEDRONI, 2008).

Para se projetar sistemas com FPGA são utilizadas linguagens de descrição de *hardware*, que consiste na parte física de alguma máquina onde o processamento é efetuado, como por exemplo, a VHSIC (*Very High Speed Integrated Circuit*) *Hardware Description Language* (VHDL). A VHDL é uma linguagem cujo o código descreve o comportamento ou a estrutura do circuito eletrônico, no qual um circuito físico compatível pode ser inferido por um compilador (PEDRONI, 2010). Como um código nada mais é do que uma máquina algorítmica de estados,

é possível que tal código receba seu representante correspondente em *hardware*. Tal *hardware* possibilita, dessa maneira, o desempenho ótimo para sua tarefa, coisa que um computador sequencial por ser de uso geral não é capaz de fazer tampouco uma GPU.

Em suma, a principal função do projetista do código em VHDL cujo o objetivo é acelerar uma tarefa computacional é encontrar uma máquina de estados em *hardware* que corresponda ao algoritmo alvo de sua aceleração. Um sistema de código implementado em uma FPGA comumente pode ser tipificado e caracterizado por duas partes básicas, intituladas de parte fixa e variável (ESTRIN, 1960). Sistema este conhecido como F+V, cujo é abordado como técnica de aceleração para os códigos alvos propostos neste trabalho.

Tendo em vista o que foi apresentado, este trabalho pretende por meio de um *kit* de desenvolvimento, integrando a FPGA com um microcontrolador presente no *kit*, realizar ensaios a fim de comparar o desempenho de códigos. Desempenho esse mensurado entre um código executado no microcontrolador, que possui uma arquitetura de Von Neumann e um sistema F+V, composto pelo mesmo microcontrolador interligado com a FPGA.

1.1 OBJETIVO GERAL

Este trabalho realizou ensaios de aceleração dos códigos detector de números primos, *bubble sort* e Z *buffer* na linguagem em C, por meio do desenvolvimento do código VHDL correspondente explorando as oportunidades de equivalências de código em linguagem de programação para seu correspondente em circuitos lógicos, esperando obter uma aceleração no processamento do problema quando comparado a um modelo em C sequencial de processamento de referência.

1.2 OBJETIVOS ESPECÍFICOS

- Desenvolver e implementar um ambiente de desenvolvimento, em outras palavras, um *framework*, destinado à validação laboratorial dos ensaios.
- Identificar nos algoritmos propostos trechos de códigos em C passíveis para uma transcrição em *hardware* com a VHDL.
- Validar os ensaios propostos no âmbito laboratorial por meio da demonstração de sua eficácia com os três códigos escolhidos.

1.3 JUSTIFICATIVA

Como mencionado no objetivo geral, os problemas propostos para aceleração são algoritmos simples e amplamente utilizados, porém computacionalmente demorados por envolverem laços dentro de laços. Pretende-se com o uso de uma FPGA e o paralelismo nela presente transcrever os códigos da linguagem C para VHDL obtendo um tempo de processamento mais

rápido. Além do desenvolvimento intrínseco de um *framework* que pode ser de grande utilidade em futuros trabalhos envolvendo o tema dentro do campus.

1.4 ORGANIZAÇÃO DO TRABALHO

No Capítulo 1 o problema é introduzido, bem como sua contextualização, os objetivos do trabalho e sua justificativa. No Capítulo 2 toda base teórica julgadas necessárias para que o entendimento e execução trabalho. No Capítulo 3 estão descritos os materiais e a metodologia utilizada para atingir os objetivos propostos. O Capítulo 4 apresenta os resultados obtidos. Por fim o Capítulo 5 traz as conclusões do trabalho.

2 REVISÃO DE LITERATURA

Neste capítulo está presente a fundamentação teórica necessária para a realização desta monografia, bem como o estado da arte. Na Seção 2.1 é abordado a complexidade de algoritmos e a notação usada na estimativa de tempo para execução dos algoritmos. A Seção 2.2 explica o funcionamento das FPGAs e o tipo de sistema usado para implementação deste trabalho. Na Seção 2.3 é apresentada a VHDL, linguagem essa utilizada para programação de FPGAs. A Seção 2.4 contém a métrica para avaliação do desempenho dos problemas propostos. Por fim a Seção 2.5 apresenta alguns trabalhos relacionados a área de pesquisa na qual esta monografia situa-se.

2.1 COMPLEXIDADE DE ALGORITMOS

Uma das principais características dos algoritmos é seu tempo de execução. Para estabelecer a complexidade de algum algoritmo são usados modelos matemáticos empíricos que representem o comportamento do algoritmo. Tal modelo busca definir um tempo de execução independente de compilador, computador ou linguagem utilizada (SZWARCFITER, 2010).

Antes de serem abordados os problemas propostos é importante ser apresentada a complexidade de algoritmos. Uma boa maneira de calcular tal complexidade é analisando o tempo gasto no pior caso de entrada do algoritmo, ou seja, quando ele for realizar o maior número de iterações possíveis. De acordo com Cormen (2002), ao ser usada a notação *O* (grande-O), é possível determinar o tempo de execução de um algoritmo em seu pior caso apenas analisando sua estrutura. A vantagem desta notação é a garantia do tempo máximo necessário para sua execução, podendo terminar antes do tempo calculado dependendo da entrada recebida.

2.1.1 Notação *O*

A complexidade do pior caso considera o desempenho para todas as entradas de tamanho *n* no algoritmo, dando mais controle sobre a operação. Ao ser obtida uma complexidade $O(1)$ é dito que o número de operações fundamentais executadas é igual a uma constante e não depende do tamanho da entrada. Já se a complexidade for de $O(n)$ tem-se um comportamento de função linear, onde n representa o tamanho da entrada que influencia diretamente no tempo de execução. Usando as definições de Toscani (2012), está subseção irá explanar o conceito da notação *O*.

Na notação *O*, é necessário analisar cada parte do código, compor as análises e assim obter a complexidade. Existem alguns custos já calculados que serão usados ao longo desta monografia. O cálculo de dois problemas-exemplos irá servir como base para posteriormente ser obtida a complexidade dos problemas propostos. Antes de apresentar tais exemplos vale

esclarecer o princípio da absorção, algo fundamental na notação *O*. O princípio baseia-se na perpetuação da maior complexidade na soma obtida da análise do algoritmo. Isto é, se o resultado da análise dos trechos de código for $O(f + g)$, onde $f = n$ e $g = n^2$ resulta em $O(n^2)$, pois g é assintoticamente superior a f.

O primeiro exemplo é o Algoritmo 1 o qual determina o maior valor de um vetor. A atribuição da linha 1 tem custo desprezível, ou seja, $C1 = O(0)$. A comparação da linha 3 por ser uma estrutura condicional simples efetuando uma atribuição, é de custo $C2 = O(1)$. Por último o laço da linha 2 é uma iteração definida de tamanho n, então tem-se um custo $C3 = O(n)$. Agora as partes analisadas são compostas pela conta: $O(C1 + C2 * C3)$. Nota-se que $C2$ multiplica $C3$ pelo fato da operação condicional linhas 3 a 5 estar dentro do laço que abrange da linha 2 até 6, para então somar seu resultado a $C1$ que se encontra em um trecho separado do laço. Com o valor $O(0 + n * 1)$ é aplicado o princípio da absorção, resultando em uma complexidade $O(n)$.

Algoritmo 1: Valor máximo de um vetor.

Input: um vetor $V[]$ de tamanho N
Output: maior valor do vetor

```
1 max ← V[0]
2 for i = 1 to N do
3 |   if V[i] > max then
4 |   |   max = V[i]
5 |   end
6 end
7 return max
```

Como segundo exemplo o Algoritmo 2 classifica o vetor de forma ascendente, ou seja, o menor valor na primeira posição do vetor e assim seguindo até o maior valor na última posição. Já foi mostrado no exemplo anterior que a estrutura condicional simples efetuando atribuição de valores da linha 3 até a 7 tem custo $C1 = O(1)$. O laço da linha 2 o custo depende do tamanho da subtração de $N - i$, como a variável i também depende do tamanho de N é obtida uma complexidade $C2 = O(n)$. Agora o laço da linha 1 possui dependência apenas com o tamanho de N, dando uma complexidade $C3 = O(n)$. Compondo as análises de custos é observado que a comparação condicional das linhas 3 a 7 está contida dentro do laço mais interno, linhas 2 a 8, o qual também está contido dentro do laço mais externo, linhas 1 a 9. Esta estrutura resulta na seguinte multiplicação de complexidades: $O(C1 * C2 * C3) = O(1 * n * n)$. Novamente aplicando o princípio da absorção o resultado da complexidade do algoritmo é de $O(n^2)$.

Há certos problemas caracterizados como intratáveis. A exemplo disto os problemas NP-completos ou até mesmo os de custo polinomial para grandes instâncias de entrada demandam um alto consumo de recursos computacionais. Para tais tipos de problemas como mencionado na introdução, adotam-se algumas técnicas computacionais alternativas aos modelos atuais

Algoritmo 2: Classificação de vetor.

Input: Vetor $V[]$ de tamanho N
Output: o vetor $V[]$ classificado em ordem ascendente

```
1 for i = 0 to N − 1 do
2     for j = 0 to N − i do
3         if V[j] > V[j+1] then
4             temp = V[j]
5             V[j] = V[j + 1]
6             V[j + 1] = temp
7         end
8     end
9 end
```

que requerem o emprego de sistemas de aceleração, como por exemplo as FPGAs.

2.2 FPGA

Computadores convencionais, isto é, que sejam baseados na arquitetura de Von Neumann, tem-se uma grande flexibilidade nas aplicações devido às tarefas computacionais serem desdobradas em um conjunto fixo de instruções típicas de cada processador e assim processadas uma após a outra. Apesar de conseguir bom desempenho na maioria das atividades, esta característica de desdobramento causa um aumento no tempo de processamento computacional. Logo, o melhor desempenho não é garantido. Para atingir-se esse objetivo, entre outras alternativas, existem as FPGAs, que implementam dispositivos de lógica reconfigurável, disponibilizando flexibilidade em programar no nível de portas lógicas obtendo um alto desempenho. Para alguns algoritmos, tais dispositivos atingiram melhores desempenho quando comparados aos computadores de uso geral (SKLIAROVA; FERRARI, 2003).

Grande parte dos problemas que necessitam de otimização por terem uma alta complexidade não são resolvíveis em tempo polinomial, partindo assim para soluções que contenham paralelismo, tornando viável o uso das FPGAs na solução desses problemas. Para suportar um maior número de aplicações, as FPGAs são utilizadas em sistemas do tipo F+V, onde a FPGA é antecedida e sucedida por processadores de uso geral, sendo a parte F o processador e V a FPGA (SKLIAROVA; FERRARI, 2003).

As FPGAs são disponibilizadas em *kits* de desenvolvimento. Tais *kits* disponibilizam *softcores* que são microcontroladores programáveis em C, os quais podem ser instanciados dentro da FPGA do *kit*. Desta forma pode-se implementar um sistema embarcado similar aos desenvolvidos em *kits* convencionais de sistemas microcontrolados. Um exemplo de *kit* é a *Spartan 3E Starter kit* da *Digilent* (XILINX, 2011). Tal *kit* disponibiliza para sua FPGA um *softcore* interno a ela chamado *MicroBlaze* (XILINX, 2013b). O *MicroBlaze* nada mais é que um microcontrolador de uso geral proporcionando a flexibilidade do mesmo fortemente ligado com a FPGA. Possibilitando tanto o uso do microcontrolador quanto ao dos blocos internos

ao dispositivo, obtendo-se desta maneira desempenho superior aos *kits* de microcontroladores genéricos.

Para programação e configuração dos *kits* de desenvolvimento, os fabricantes disponibilizam ferramentas de *Eletronic Design Automation* (EDA). Oferecendo três grandes processos: síntese, implementação e programação. O processo de síntese é responsável por compilar a descrição do sistema digital feito na linguagem VHDL (a ser detalhada na Seção 2.3). O processo de implementação transforma a descrição em VHDL do circuito digital em uma linguagem denominada *Register Transfer Level* (RTL), passando após isso a fase de adequá-lo ao dispositivo alvo da programação. Processo esse funcionando de acordo com a FPGA alvo da implementação, distribuindo assim os elementos lógicos dentro da capacidade do dispositivo. É possível ocorrer nesta etapa a incapacidade de o dispositivo receber o sistema a ser implementado. Por fim o processo denominado programação é responsável pela geração do arquivo binário de configuração chamado *bitstream* responsável pela *Cross-compilation*. Esse arquivo é descarregado na FPGA implementando todo o *design* programado (PEDRONI, 2010). Uma destas ferramentas é o ISE *Design Suite* da *Xilinx*. Com um cabo USB/JTAG (*Universal Serial Bus*/*Joint Test Action Group*) o *bitstream* é descarregado para o *kit* de desenvolvimento (XILINX, 2009).

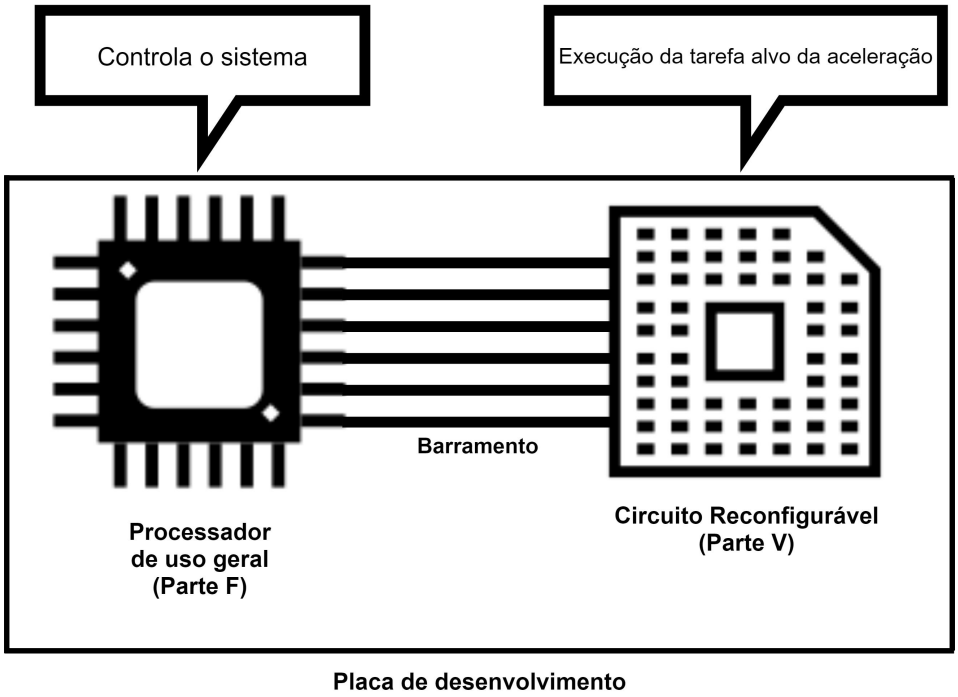
Aproveitando-se da coexistência entre *sofcore* e blocos internos de VHDL modernizou-se a implementação de uma antiga arquitetura a ser detalhada na seção a seguir.

2.2.1 Sistema F+V

Os sistemas F+V foram primeiramente idealizados por Gerald Estrin em 1959, consistem em um processador em conjunto a uma matriz de *hardware* reconfigurável, como por exemplo na Figura 1. A dificuldade em produzir um circuito reconfigurável em 1959 tornava esse sistema inviável, pois o circuito da parte V precisava ser montado ou fabricado para cada aplicação específica. Tal hardware reconfigurável nos dias atuais passou a ser uma alternativa viável com as FPGAs, como ilustrado na Figura 2. Com o uso das ferramentas de EDA a configuração do *softcore*, associado aos blocos VHDL presentes na FPGA foi amplamente facilitada, basta escrever um novo código em VHDL para que o circuito presente na FPGA seja reconfigurado. Desta maneira o processador comanda o sistema enviando e recebendo dados dos blocos em VHDL projetados, além da possibilidade de usar diversos periféricos presentes nos *kits* de desenvolvimento. Tarefas como processamento de imagens e reconhecimento de padrões usam tal sistema (ESTRIN, 2002).

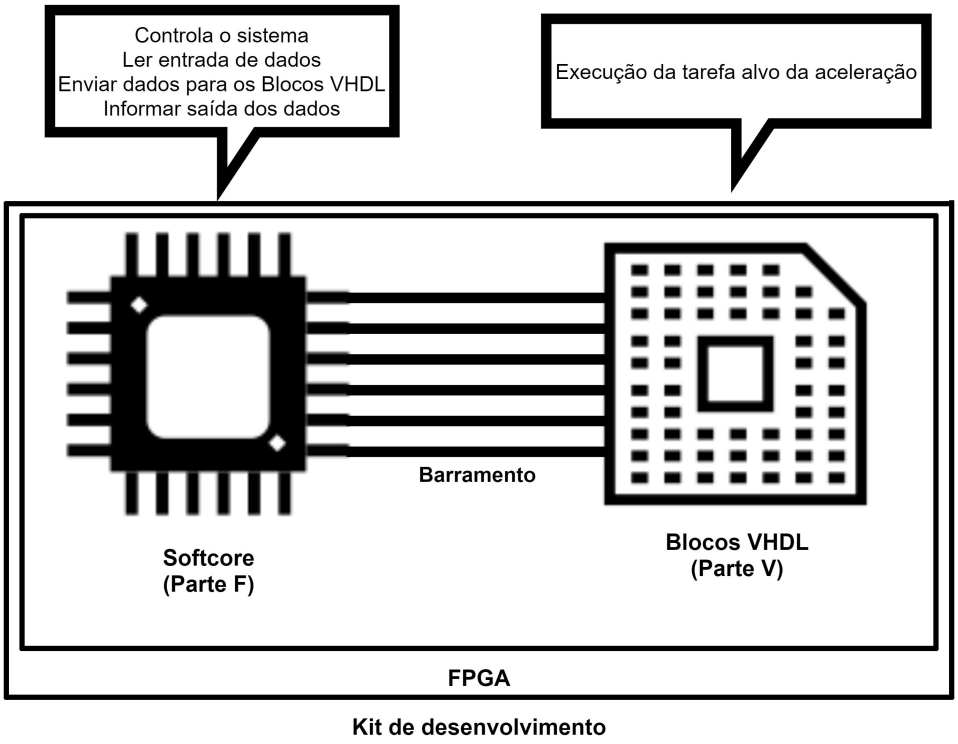
Arquitetado por Estrin (1960) a parte F, chamada parte fixa, seria uma parte destinada ao algoritmo computacional em si podendo tanto operar dentro do *chip* ou de modo sequencial convencional. A parte V seria a parte alvo da aceleração do código, englobando a região crítica do algoritmo contendo o processamento intenso, alvo da aceleração. Logo, o V de variável vem do fato de haver uma parte de processamento crítico para cada tipo de tarefa computacional a ser realizada. Enquanto que a parte F, de modo geral, corresponde a tarefas corriqueiras

Figura 1 – Sistema F+V proposto por Estrin



Fonte: Autoria Própia

Figura 2 – Sistema F+V em uma FPGA



Fonte: Autoria Própia

que um algoritmo computacional tenha que fazer antes de processar os dados, por exemplo, captura e leitura de dados, rotinas de apresentação em tela e inicialização de variáveis. Cabe

ressaltar que a parte V não necessita obrigatoriamente ser paralelizável como requerido em códigos usados em processamentos paralelos sejam eles em computadores em *cluster* ou em GPUs.

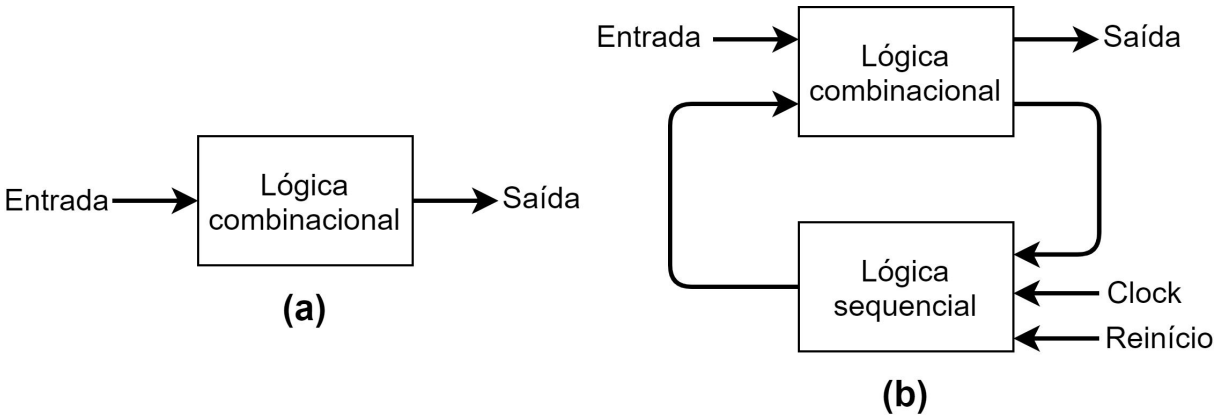
A CPU, parte F, como já dito anteriormente executa rotinas de sistema, ligada com ela a parte V representada pelos blocos VHDL computando a tarefa alvo. O *kit* de desenvolvimento *Spartan 3E starter kit* apresenta tal arquitetura, onde os blocos lógicos estão ligados por um barramento com o *softcore MicroBlaze*, esse desempenhando o papel da parte fixa. O XILINX (2013a) detalha passo a passo todo o processo necessário para implementação do sistema F+V dentro de uma placa da *Xilinx* utilizando a ferramenta de EDA *ISE Design Suite*.

2.3 VHDL

A linguagem VHDL é a linguagem usada nos blocos VHDL citados na Figura 2 que permite transcrever-se código que tenha um circuito de *hardware* correspondente, indiferente dele ser combinacional ou sequencial para dentro de uma FPGA. Por definição, um circuito lógico combinacional é aquele em que as saídas dependem apenas das entradas, assim o sistema não possui *loops* e *feedbacks*. Por outro lado, o um circuito sequencial é aquele em que a saída depende não só das entradas como também dos seus estados anteriores, portando elementos de armazenamento bem como um sinal de *clock* responsável para controlar o sistema. Tal linguagem permite a sintetização dos circuitos e sua simulação (PEDRONI, 2008).

A Figura 3 demonstra a definição acima. Na Figura 3.a o bloco de lógica combinacional contém apenas um circuito totalmente paralelo onde o sinal de entrada adentra na FPGA, percorre o circuito sintetizado e resulta no sinal de saída. Como já mencionado pode ser necessário conciliar as lógicas combinacionais e sequenciais, como está presente na Figura 3.b, com a adição do bloco de lógica sequencial computando valores advindos do bloco combinacional por meio de sinais de controle *clock* e reinício enviados pela FPGA.

Figura 3 – Modelos (a) combinacional e (b) sequencial



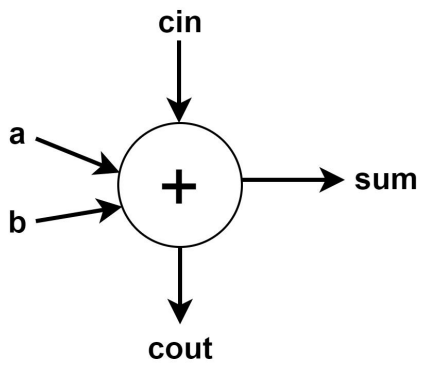
Fonte: Adaptado de Pedroni (2010)

Para transcrever um circuito em VHDL esta linguagem dispõe de sinais, variáveis,

trechos concorrentes (assemelhados aos circuitos cobinacionais) e trechos sequenciais (assemelhados aos circuitos sequenciais). Serão apresentados dois exemplos do Pedroni (2010) ensinando o funcionamento da VHDL aonde algumas palavras reservadas e exemplos de como isto é implementado serão apresentados a seguir. O código transcrito é composto por duas partes básicas, uma chamada *entity* e outra chamada *architecture*. Na *entity* são declaradas as variáveis correspondentes as entradas e saídas do circuito lógico. Na *architecture* é descrito como que entradas e saídas se relacionam para produzir o circuito funcionando como se deseja.

O primeiro exemplo baseado em uma lógica combinacional é o *Full-adder*, realiza uma operação básica da soma de dois bits com um valor de *carry-in*. A tabela da verdade da Figura 4 simplifica o entendimento da operação. O resultado é armazenado em duas varáveis, sum é o valor binário da soma das entradas a, b e cin, como a base é binária só são permitidos valores 0 e 1 fazendo necessário o uso da variável $cout$ para armazenar o valor de extrapolação da soma.

Figura 4 – Tabela da verdade do Full-adder



a	b	cin	sum	cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

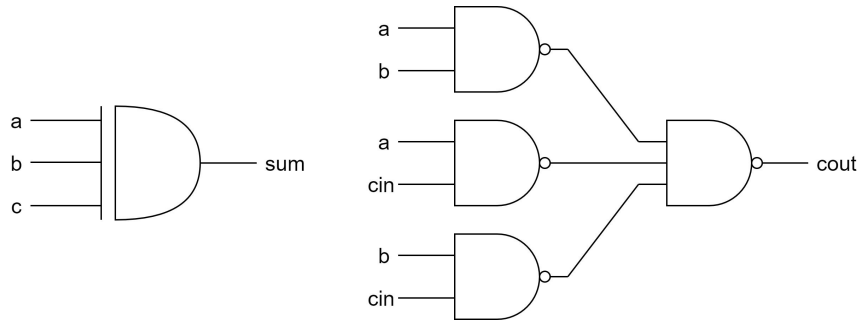
Fonte: Adaptado de Pedroni (2010)

Partindo da tabela verdade do *Full-adder* é obtido o circuito combinacional da Figura 5. Para o valor de sum é necessário o uso da porta lógica XOR ($\oplus$), $sum = a \oplus b \oplus cin$. Com esta lógica sum receberá valor 1 apenas quando nas entradas houver um número ímpar de valores 1, bem como visto na tabela da verdade. Agora para obter $cout$ é preciso um circuito um pouco mais elaborado, combinando as entradas em pares com o uso de portas lógicas NAND. Porém é possível simplificar o circuito utilizando as portas AND ($\cdot$) e OR ($+$) para seguinte forma lógica sem afetar o resultado, $cout = (a \cdot b) + (a \cdot cin) + (b \cdot cin)$. Desse modo $cout$ recebe valor 1 apenas quando ao menos duas entradas têm valor 1, condição vista na tabela da verdade que causa a extrapolação da soma.

Para encerrar o primeiro exemplo resta transcrever o circuito obtido da Figura 5 em código VHDL. Na Figura 6 o *Full-adder* é transcrito. Nesta seção ENTITY as entradas e saídas do circuito lógico, por possuírem valores binários todas são do tipo BIT (apenas 0 ou 1). Nesta seção ARCHITECTURE o circuito combinacional obtido na Figura 5 é inserido, cada uma das duas saídas recebe a operação lógica correspondente com as entradas para obter seu resultado.

Seguindo para o segundo exemplo, cujo está baseado na composição das lógicas

Figura 5 – Circuito combinacional *Full-adder*



Fonte: Adaptado de Pedroni (2010)

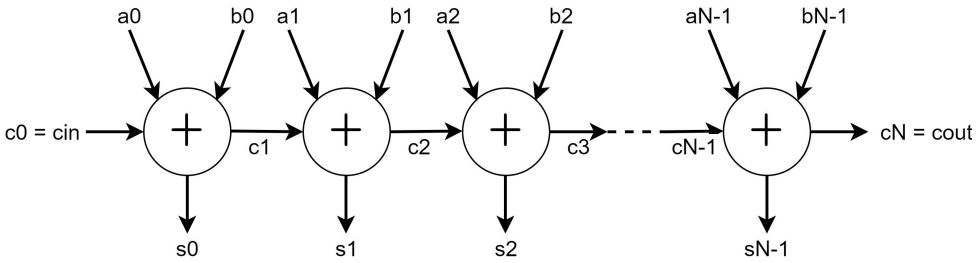
Figura 6 – Código VHDL do *Full-adder*

```
1   entity full_adder is
2       port (a, b, cin: in bit;
3             sum, cout: out bit;
4       end full_adder;
5   ------------------------------------------------------
6   architecture dataflow of full_adder is
7   begin
8       sum <= a xor b xor cin;
9       cout <= (a and b) or (a and cin) or (b and cin);
10  end dataflow;
```

Fonte: Pedroni (2010, p. 6)

combinacionais e sequenciais da Figura 3.b tem-se o *Carry-ripple adder*. A Figura 7 descreve o circuito, no qual pode ser observado que sua composição é feita pela ligação de vários *Full-adder* da Figura 5 em sequência. O resultado de *carry-out* que antecede um *Full-adder* de outro serve de entrada para o *carry-in Full-adder* sucessor, tendo como resultado final um número binário com tamanho de N bits a ser definido pelo projetista.

Figura 7 – Circuito *Carry-ripple adder*



Fonte: Adaptado de Pedroni (2010)

Como já foi visto no primeiro exemplo a elaboração do circuito *Full-adder* basta partir para a etapa programação do código em VHDL. Para obter o *carry-in* na soma do elemento

$i + 1$ é preciso antes ter o resultado de *carry-out* do elemento i, criando desse modo uma lógica sequencial na espera de resultados entre os *Full-adders*, donde cada elemento executa internamente uma lógica combinacional.

A implementação em VHDL do circuito *Carry-ripple adder* é apresentada na Figura 8. Na ENTITY a diferença em relação ao código anterior é a adição da variável genérica N, o valor dela define o tamanho do vetor de bits que se deseja somar além de adequar o tipo de dado de a, b e c para serem compatíveis com o tamanho do vetor definido em N. Definido $N = 8$ como no código é possível obter como resultado valores na faixa de 0 a 255. Em ARCHITECTURE foi declarado com PROCESS na linha 15 o início da lógica sequencial, as variáveis a, b e cin citadas no PROCESS definem que os valores das linhas 18, 20 e 21 receberão novos conteúdos a cada momento que a, b e cin receberem alterações em seus conteúdos na chamada lista de sensibilidade da linha 15. A variável interna c serve de auxílio para armazenar o valor *carry-out* em cada iteração. Por fim dentro do laço é adicionada a logica combinacional desenvolvida para um *Full-adder*, aqui é estabelecido o valor de *carry-in* do próximo elemento disparando desta forma um gatilho para o cálculo do próximo *Full-adder* no processo.

2.3.1 Conversão de código para VHDL

O D'Amore (2012) traz um capítulo dedicado a conversão de códigos para VHDL, no qual está subseção foi baseada. Converter um algoritmo para VHDL não á algo trivial, é necessário considerar vários fatores como por exemplo o armazenamento dos dados, as decisões tomadas e o controle das operações. Vale ressaltar a inexistência de uma metodologia exata para esta conversão e cada algoritmo precisa ser analisado de forma individual.

Alguns aspectos são importantes na conversão. O uso de diagrama de blocos definindo de forma isolada os trechos de código presentes no algoritmo é um deles, podendo separar de maneira mais fácil os blocos combinacionais dos sequenciais. Tendo os blocos isolados a interligação do sistema se torna mais simples, sendo capaz de reduzir os recursos utilizados na FPGA.

O exemplo do máximo divisor comum apresentado pelo D'Amore (2012) traz uma conversão de pseudocódigo para VHDL. Partindo do pseudocódigo na Figura 9 os blocos identificados serão transpostos para o código VHDL. As variáveis $inicio$ e fim nas linhas 1 e 2 desempenham um papel de controle, logo estas servirão como um sinal de gatilho enviado para FPGA iniciar o processo. Nas linhas 3 e 4 a_en e b_en são os valores de entrada da função, ou seja, são valores recebidos externamente à FPGA. Na linha 5 é necessária uma tomada de decisão para realização do laço, para isso o uso de um circuito comparador desempenha tal tarefa. Nas linhas 6 e 7, contidas em um laço é feita uma comparação condicional simples, novamente é feito uso de um circuito comparador. Dentro da condicional a operação de subtração é implementada com um circuito subtrator. Ao final o valor do máximo divisor comum é enviado para saída do circuito e o gatilho de fim de execução.

Os blocos identificados na análise do pseudocódigo são transcritos para o código

Figura 8 – Código VHDL do *Carry-ripple adder*

```
1  ----------------------------------------------------------
2  libary ieee;
3  use ieee.stf_logic_1164.all;
4  ----------------------------------------------------------
5  entity carry_ripple_adder is
6      generic (N : integer := 8); --numero de bits
7      port (a, b: in std_logic_vector(N-1 downto 0);
8            cin: in std_logic;
9            s: out std_logic_vector(N-1 downto 0);
10           cout: out std_logic;
11 end entity;
12 ----------------------------------------------------------
13 architecture structure of carry_ripple_adder is
14 begin
15     process(a, b, cin)
16         variable c: std_logic_vector(N downto 0);
17     begin
18         c(0) := cin;
19         for i in 0 to N-1 loop
20             s(i) <= a(i) xor b(i) xor c(i);
21             c(i+1) := (a(i) and b(i)) or (a(i) and c(i))
22                       or (b(i) and c(i));
23         end loop;
24         cout <= c(N);
25     end process;
26 end architecture;
27 ----------------------------------------------------------
```

Fonte: Adaptado de Pedroni (2010)

Figura 9 – Pseudocódigo do máximo divisor comum

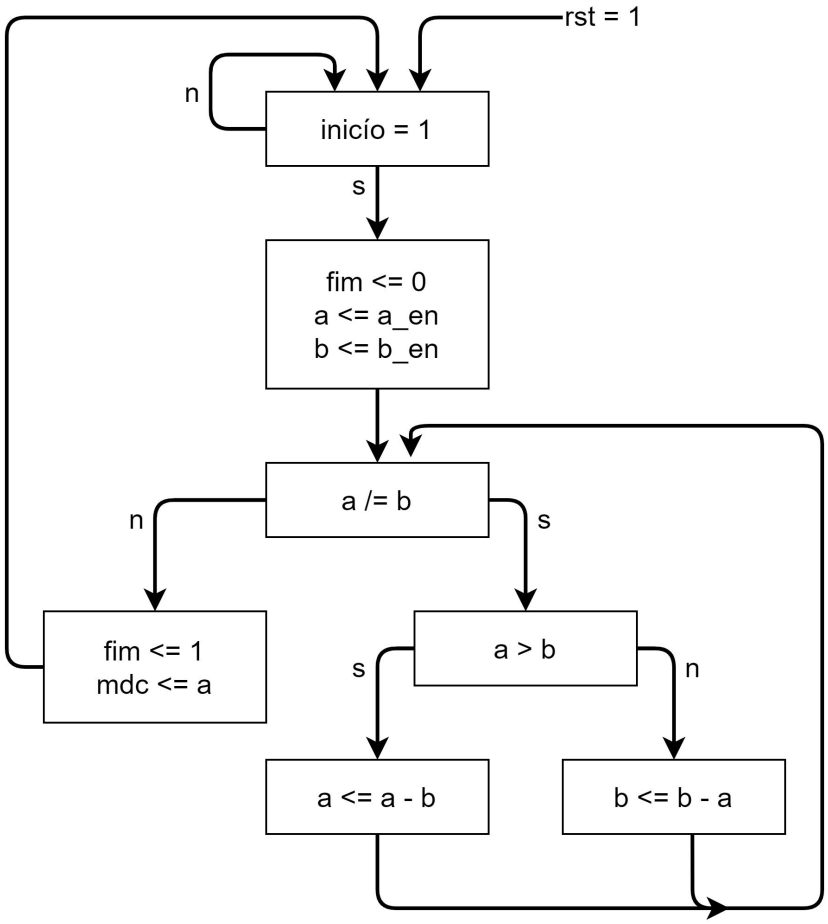
```
1  enquanto(inicio = 1){
2      fim <- 0;
3      a <- a_en;
4      b <- b_en;
5      enquanto (a /= b){
6          se (a > b) a <- a - b;
7          senao      b <- b - a;
8      }
9      mdc <- a;
10     fim <- 1;
11 }
```

Fonte: Adaptado de D'Amore (2012)

VHDL da Figura 11. Usando a Máquina de Estados Finitos (*Finite State Machine* - FSM) da Figura 10 o problema é implementado. Os blocos identificados na análise passam a formar um estado na máquina e as condições resultantes em cada bloco ocasionam a transição entre os estados.

Implementar uma FSM em VHDL torna necessário o uso de um sinal de controle *clock* e outro de *reset*. Na ENTITY do código, mais especificamente na linha 2 tais sinais são definidos, a cada sinal de *clock* enviado ao circuito, a FSM verifica a condição definida no estado na qual ela situa-se e toma a decisão entre permanecer no mesmo estado ou mudar para outro. Na estrutura de CASE e WHEN, ferramentas estas de apoio na linguagem VHDL. O WHEN executa uma ação para cada uma das entradas possíveis da variável definida no CASE, operando de maneira semelhante a um multiplexador. Da linha 20 até a 49 os blocos transcritos em circuitos estão presentes dentro da estrutura de CASE e WHEN. Nota-se que identificar corretamente os blocos de forma isolada facilita a composição dos mesmos em uma FSM. Vale ressaltar a possibilidade da obtenção de outros circuitos combinacionais equivalentes a esta FSM, podendo ou não ter um desempenho melhor a este que foi apresentado.

Figura 10 – FSM do máximo divisor comum



Fonte: Adaptado de D'Amore (2012)

Figura 11 – Código em VHDL para o máximo divisor comum

```
1   entity chap_06_GCD2 is
2   port (ck, rst, inicio : in bit;
3         a_en, b_en : in  integer range 0 to 31;
4         mdc        : out integer range 0 to 31;
5         fim        : out bit);
6   end chap_06_GCD2;
7
8   architecture xyz of chap_06_GCD2 is
9     signal a, b : integer range 0 to 31;
10    type ciclos_maq is (espera, tes_a_b, a_dif_b, a_igl_b);
11    signal estado : ciclos_maq;
12  begin
13    abc: process (ck, inicio)
14    begin
15      -- Operacoes iniciais assincronas
16      if rst = '0' then
17          estado <= espera;
18      -- Operacoes controladas pelo relogio
19      elsif (ck'event and ck = '1') then
20        case estado is
21          -- Espera condicao para executar
22          when espera =>
23            if inicio = '1' then
24              fim <= '0';
25              a <= a_en; b <= b_en;
26              estado <= tes_a_b;
27            else
28              estado <= espera;
29            end if;
30
31          -- Laco 1: Enquanto a/= b repetir
32          when tes_a_b =>
33            if a/= b then estado <= a_dif_b; -- continua
34            else          estado <= a_igl_b; -- sai
35            end if;
36          -- Operacoes no laco 1
37          when a_dif_b =>
38            if a > b then a <= a - b;
39            else          b <= b - a;
40            end if;
41            estado <= tes_a_b; -- fim do laco 1
42
43          -- Operacoes apos o fim do laco 1
44          when a_igl_b =>
45            mdc <= a;
46            fim <= '1';
47            estado <= espera;
48          end case;
49      end if;
50    end process;
51  end;
```

Fonte: Adaptado de D'Amore (2012)

2.4 AVALIAÇÃO DE DESEMPENHO

De posse de um sistema implementado na arquitetura F+V, surge a necessidade de por meio de uma métrica de avaliação de desempenho avaliar tal sistema. Nesta seção será visto uma maneira de estipular o desempenho do sistema F+V implementado quando comparado ao sistema puramente F.

A Lei de Amdahl define uma métrica para o cálculo de aceleração de códigos, utilizando de medidas de tempo entre códigos sequenciais e paralelos o *speedup* do problema obtido (AMDAHL, 1967). O cálculo de *speedup* é a divisão entre o tempo para executar um programa em um único processador pelo tempo para executar o mesmo programa em N processadores paralelos. Esta lei é descrita na Equação (1), a varável N presente na equação se refere ao número de processadores usados na execução paralela e f representa a porcentagem do código que foi paralelizável (STALLINGS, 2010).

$$Speedup = \frac{1}{(1-f) + \frac{f}{N}} \tag{1}$$

Para ter uma melhor ideia do *speedup* em sistemas *multicores*, os quais são descritos pela Equação (1) é apresentada a Figura 12. Cada curva é um código com uma respectiva porcentagem de possível paralelização, percebe-se que essa porcentagem é quem limita o valor de *speedup* que um código pode alcançar. Mesmo aumentando exponencialmente o número de processadores, como por exemplo no caso de um algoritmo onde apenas 50% de seu código foi paralelizado, não se obtém um *speedup* maior que 2. Tendendo o valor de N ao infinito na Equação (1) o valor da divisão por N na soma é zerado, podendo de tal forma descobrir o máximo de *speedup* que pode ser alcançado para uma determinada porcentagem paralelizável de código (MCCOOL; REINDERS, 2012).

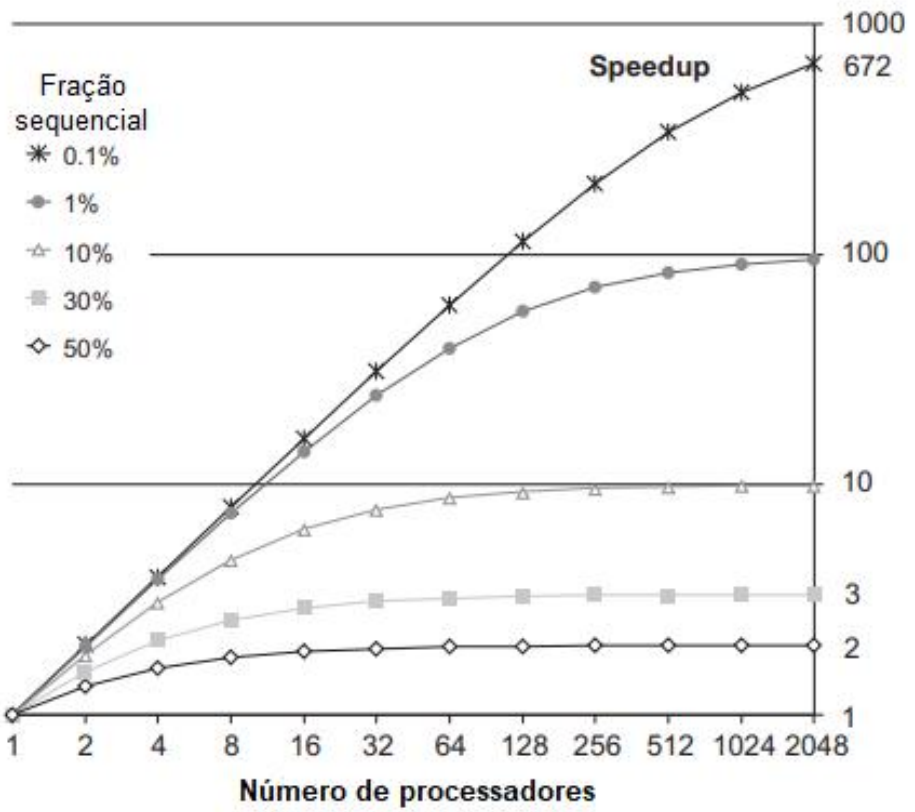
Como a FPGA não se fundamenta em acelerar um problema usando multiprocessadores, mas sim com dispositivos lógicos reprogramáveis. Sendo assim a Lei de Amdahl pode ser generalizada de acordo com a Equação (2), Tan é o tempo antes da melhoria de código e Tap o tempo após ela. Possibilitando desta forma o cálculo de desempenho em sistemas que não apresentam o uso de tecnologia *multicores* (STALLINGS, 2010).

$$Speedup = \frac{Tan}{Tap} \tag{2}$$

2.5 TRABALHOS CORRELATOS

O artigo Lipu et al. (2016) implementou o algoritmo *bubble sort* de forma paralela, utilizando a FPGA da *Spartan 6 kit*. Ainda na área dos problemas de ordenação o artigo Mueller, Teubner e Alonso (2012) desenvolve uma rede de ordenação com a FPGA *Virtex5 FX130T* da *Xilinx*. Para o algoritmo *Z buffer* foi realizado no artigo Ali (2011), uma versão

Figura 12 – Comportamento do *Speedup* em *multicores*.



Fonte: Adaptado de McCool e Reinders (2012)

combinacional do algoritmo usando a *Spartan3 starter kit* mostrando que o algoritmo é possível de ser implementado.

Trabalhos mais recentes que não necessariamente envolvam os problemas propostos, mas sim aceleração de código usando FPGA são amplamente estudados. O artigo Englund e Lindskog (2020) busca usar FPGAs baseadas em nuvem para aceleração de criptografias. O artigo Li et al. (2018) mostra que é possível obter aceleração no processamento de redes neurais quando comparado ao uso de GPUs. Juntando os dois campos de pesquisa dos trabalhos citados anteriormente neste parágrafo o artigo Zhang et al. (2020) usa redes neurais convolucionais na segurança computacional para classificar ameaças de *malware* trafegando na rede.

Tais trabalhos evidenciam que a busca por aceleração de problemas computacionais utilizando-se para tais de FPGAs é uma iniciativa considerada válida.

2.6 INTERLIGAÇÃO DE CONTEÚDOS

A complexidade de algoritmos serve para modelagem dos problemas proposto abordados. As FPGA foram abordadas por estarem contidas em *kits* de desenvolvimento, os quais apresentam um *softcore*, possibilitando a implementação de uma arquitetura F+V. Com a VHDL esta parte V nas FPGAs é programada e com a lei de Amdahl é mensurado a aceleração

obtida com o sistema F+V implementado dentro de kit de desenvolvimento contendo uma FPGA a qual foi programada em VHDL.

3 METODOLOGIA

Nesta seção estão descritos os *softwares* e *hardwares* utilizados além do método usado no desenvolvimento do trabalho.

3.1 MATERIAIS

Os seguintes materiais foram utilizados:

- *Kit* de desenvolvimento de lógica reconfigurável *Spartan 3E Starter Kit* da Digilent (XILINX, 2011), disponível na UTFPR Pato Branco.
- Para ferramenta de EDA é necessário a utilização do ISE *Design Suite* 14.7 da Xilinx (XILINX, 2009), por ser a ferramenta compatível com o *kit* utilizado e também disponibilizada pela UTFPR Pato Branco. Dos produtos contidos no ISE serão utilizados também:
  - Xilinx *plataform studio* (XPS). A ser usado para o desenvolvimento do *hadware* do *MicroBlaze*.
  - Xilinx *Software Development Kit* (SDK). Necessário para programação em C do *MicroBlaze* e no processo de *cross-compilation* com a FPGA.
- Notebook Samsung com processador i7-4510U @2.00 GHz, 8 GB de memória RAM e sistema operacional Windows 10 Home 64 bits, de propriedade do estudante para uso da plataforma de programação e comunicação com o *kit*.
- Cabo conversor USB para serial RS-232.
- Programa de acesso serial Putty de *software* livre (PUTTY, 2021).

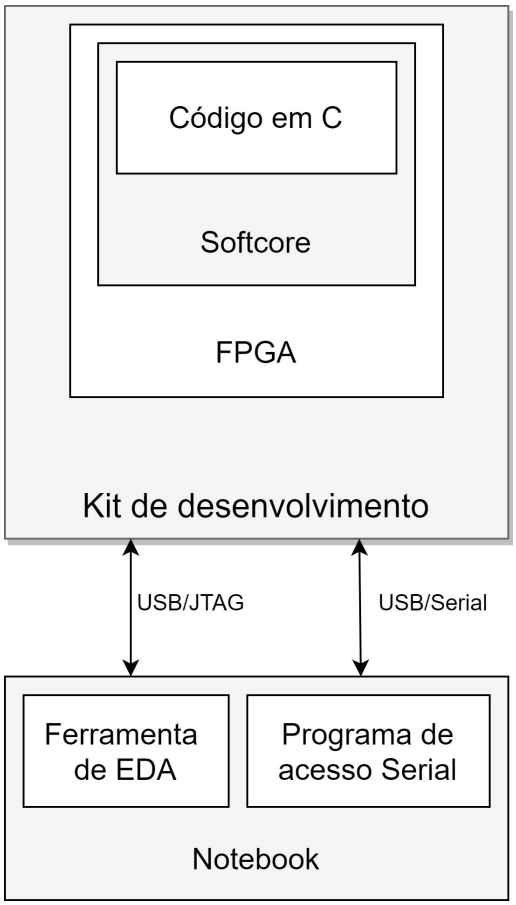
3.2 MÉTODO

A estratégia adotada para se realiziar os ensaios basea-se em um método idealizado em quatro etapas. A primeira é a execução dos códigos alvo deste trabalho em C no *MicroBlaze* da *Spartan 3E starter kit*, obtendo-se desta forma os tempos para um sistema de referência a ter seu desempenho melhorado. Na segunda etapa, é feita a identificação nos códigos de referência da primeira etapa de partes computacionalmente críticas que possam ser alvos de transcrição para um *hardware* específico. Na terceira etapa implementasse o sistema F+V, onde F corresponderá a um código baseado no sistema de referência, porém tal código conterá somente as partes corriqueiras de processamento, sendo que a parte V conterá a implementação em VHDL das partes críticas que foram identificadas na etapa anterior. Na quarta é calculado o *speedup* do código acelerado em relação ao sistema de referência obtido na primeira etapa.

3.2.1 Primeira etapa

Na primeira etapa será realizada a execução dos códigos alvos da aceleração de forma sequencial em C, no microcontrolador *MicroBlaze*. Etapa essa executada de acordo com a estrutura da Figura 13 (maiores detalhes na Seção 2.2). A programação do *MicroBlaze* é feita em duas etapas. Primeiro por meio do XPS é programado o *hardware* propriamente dito, ou seja, o barramento, sua interface e os dispositivos conectados. Em seguida sobre a programação do *hardware* no XPS é criado um elemento denominado *Board Support Package* (BSP) em cima do qual é desenvolvido o *software* do sistema. Para desenvolver o *software* programado em C, utiliza-se o SDK. Todas essas ferramentas são oferecidas dentro do pacote do ISE *Design Suite*. Como o microcontrolador se baseia no modelo de von Neumann, o tempo de execução de uma tarefa em C será o tempo de referência a ser adotado no ensaio. Tempo esse atribuído a variável Tan na Equação (2), referente ao cálculo de *speedup*.

Figura 13 – Componentes presentes no *kit* e conexões para comunicação



Fonte: Autoria Própria

O notebook presente na Figura 13 se faz necessário para que a ferramenta de EDA, *ISE Design Suite*, seja utilizada. É nesta ferramenta onde é desenvolvido o sistema tanto em termos de linguagem C quanto de VHDL e com ela seja possível a síntese, implementação e programação da placa de FPGA. Com ela os periféricos necessários do *kit* são configurados, o

softcore MicroBlaze presente na FPGA é programado com um código em linguagem C, além é claro da sintetização do código VHDL para dentro da FPGA, o *bitstream*. Desta forma a FPGA está preparada para executar a tarefa e por meio da conexão USB/Serial a *Spartan 3E Starter Kit* retorna ao computador via programa de acesso serial os seus resultados permitindo assim o monitoramento das tarefas.

3.2.2 Segunda etapa

Utilizando como base a Subseção 2.3.1, na segunda etapa será realizado o estudo e identificação de partes críticas de processamento do código de referência em C. Tais partes são os alvos de transcrição para um *hardware* correspondente, por meio da linguagem VHDL.

Esta etapa é puramente analítica e teórica, onde as partes críticas passarão por um estudo. Partes identificadas são idealizadas em um ou mais blocos de VHDL. Tais partes podem ser totalmente combinacionais dentro da FPGA ou, caso necessário, sequenciais dependendo de seus respectivos *hardwares* correspondentes.

3.2.3 Terceira etapa

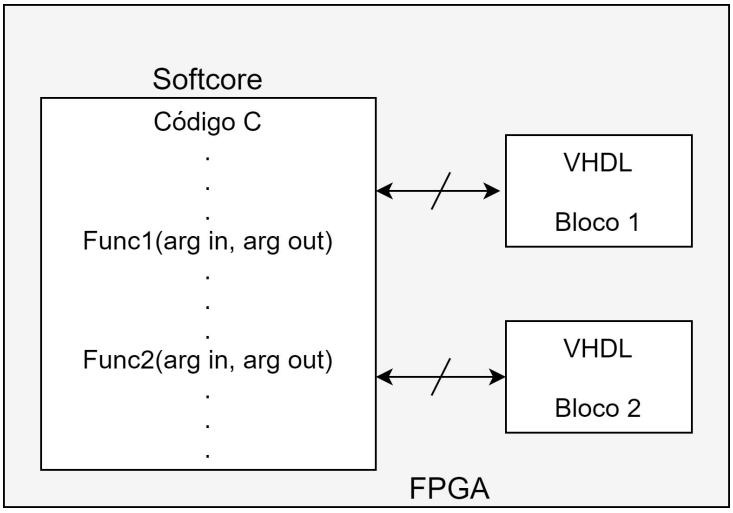
Na terceira etapa será realizado o desenvolvimento e implementação de um ou mais blocos em VHDL que permitem a execução em *hardware* dos pontos identificados na etapa dois. Tais trechos serão desenvolvidos em VHDL e integrados ao um sistema similar àquele implementado na primeira etapa, porém acrescidos dos respectivos blocos, como por exemplo, na Figura 14. Desta forma, é possível a execução do código parte no *MicroBlaze* e parte nos blocos de VHDL. A comutação entre a execução do código no *softcore* e nos blocos em VHDL ocorrerá por meio de chamada de função simples e retorno da mesma comandadas pelo código principal em C executando no *MicroBlaze*. Em outros termos, tem-se um sistema F+V descrito na Subseção 2.2.1: no *softcore* a parte fixa e nos blocos em VHDL a parte variável. O tempo de execução para cada algoritmo nesta etapa será utilizado no cálculo de *speedup* da Equação (2), mais especificamente na variável Tap.

3.2.4 Quarta etapa

Tendo em posse os resultados dos procedimentos anteriores implementados, a quarta etapa consiste em calcular o desempenho obtido na aceleração dos códigos. Utilizando a Equação (2) descrita na Seção 2.4 é comparado o tempo de referência do código em C na primeira etapa com o tempo do sistema F+V da terceira etapa. A divisão entre os tempos obtidos na primeira e terceira etapa resulta no *speedup*.

Para o cálculo de *speedup* duas análises serão feitas. Primeiro será calculado o *speedup* comparando o tempo total de execução dos algoritmos nos dois sistemas, iniciando a contagem antes das inicializações de variáveis e terminando no final do algoritmo. Na segunda análise o

Figura 14 – Fluxo do Código no *softcore* utilizando-se dos blocos VHDL



Fonte: Autoria Própria

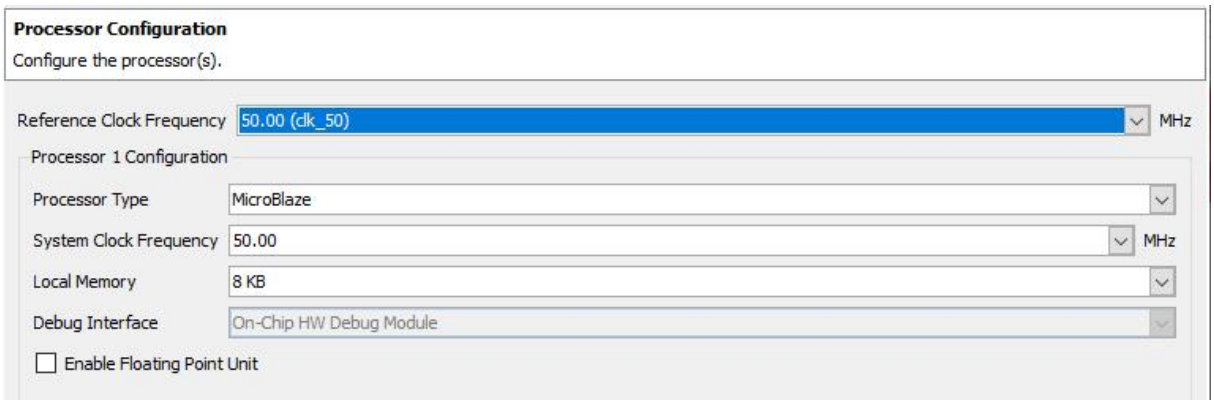
cálculo do *speedup* levará em conta apenas a parte crítica, ou seja, somente onde os trechos de códigos em C foram identificados e transcritos em código VHDL.

4 RESULTADOS

Este capítulo traz a análise de complexidade dos problemas propostos por meio de seus respectivos pseudocódigos, suas transcrições para o sistema F+V, os gráficos comparativos das medidas de tempo obtidas em suas execuções e a tabela com o cálculo do *speedup*. Os códigos desenvolvidos para cada um dos problemas estão contidos nos apêndices.

Os tempos de execução foram obtidos com o *softcore* (MicroBlaze) operando em uma frequência de 50 MHz, tal configuração está presente na Figura 15, maiores detalhes na Seção 3.1. Todos códigos foram desenvolvidos usando os periféricos adicionados ao *MicroBlaze* listados na Figura 16. A Figura 17 mostra sumário final de configuração do sistema embarcado no *MicroBlaze*, sua conexão com os dispositivos lógicos não aparece em tal sumário, a adição dos mesmos é realizada em uma etapa posterior do projeto. Para realizar a medida de tempo foi necessário o uso do dispositivo de *timer*. O próprio *MicroBlaze* disponibiliza uma função de chamada de *timer*, tal função retorna o valor de um registrador que funciona como contador, este registrador é acrescido de uma unidade a cada ciclo de *clock* do *MicroBlaze*. Fazendo uso desta função é obtido o número contido no registrador do *timer* antes do início de processamento do algoritmo e após o termino, a diferença entre o valor final e inicial, descontando mais um valor de calibração o qual é referente ao tempo gasto somente no uso da função do *timer* é o resultado de tempo para a execução do algoritmo. Como o *MicroBlaze* opera a 50 MHz, cada unidade do resultado final de tempo corresponde a 20 ns. O processo de tal calculo está contido nos códigos em C dos apêndices.

Figura 15 – Tela de configuração do *MicroBlaze*.

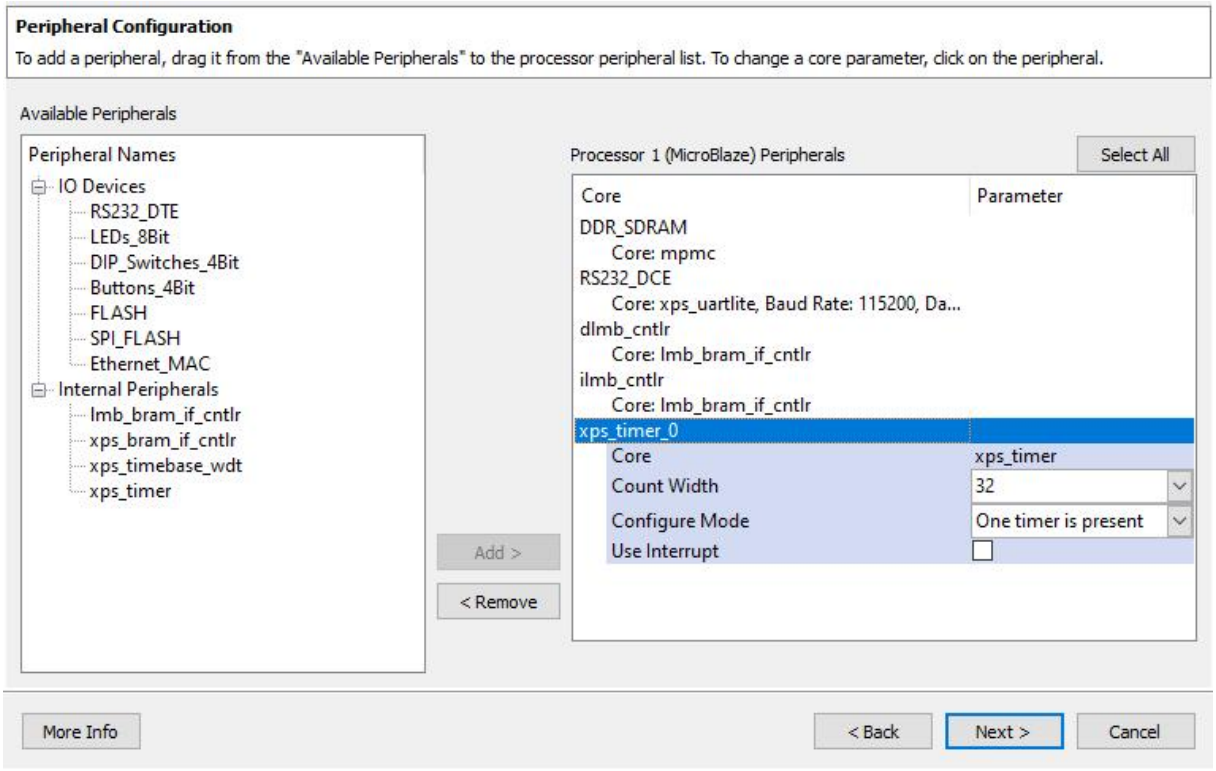


Fonte: Extraido do XPS.

4.1 DETECTOR DE NÚMEROS PRIMOS

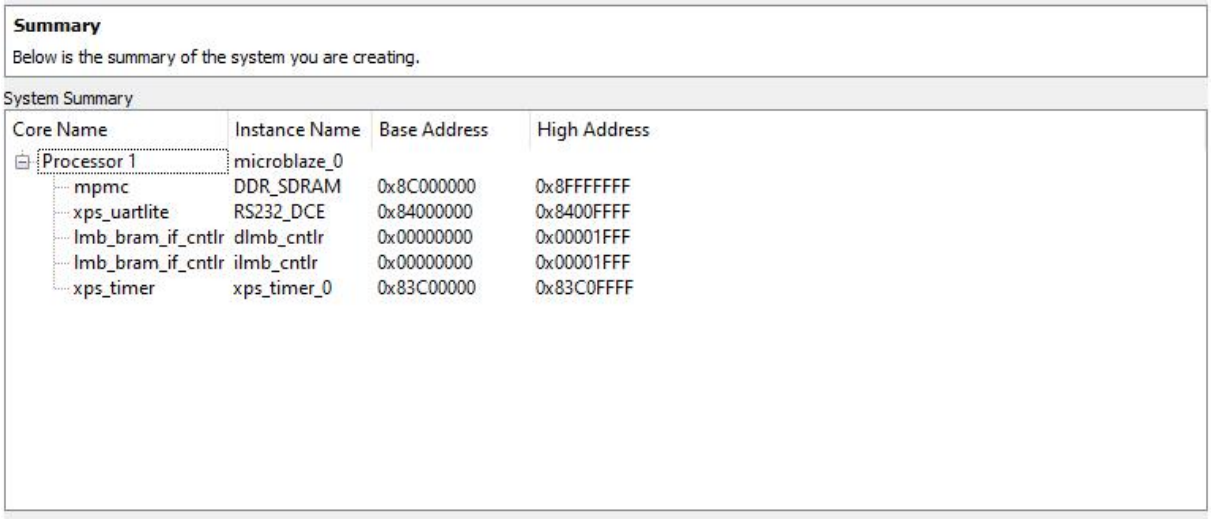
Detectar um número primo é um problema simples de ser resolvido e muito útil em algoritmos de criptografia. Todavia em uma arquitetura sequencial, seu cálculo cresce

Figura 16 – Tela de seleção de periféricos para o sistema embarcado.



Fonte: Extraido do XPS.

Figura 17 – Tela do sumário final de configuração do sistema.



Fonte: Extraido do XPS.

linearmente com o tamanho da entrada. No Algoritmo 3, tem-se seu pseudocódigo. Analisando a linha 1 a atribuição tem custo nulo de $C1 = O(0)$. O laço da linha 2 até a 6 depende somente do tamanho da entrada n, custando assim $C2 = O(n)$. A estrutura condicional com atribuição da linha 3 a 5 é de custo $C3 = O(1)$. Compondo os custos $O(C1 + C2 * C3)$, é obtida a complexidade de $O(n)$.

Algoritmo 3: Detector de números primos

Input: um número inteiro n
Output: booleano identificando se o número n é primo

```
1 divisor ← 2
2 while divisor <= n/2 do
3 |   if n % divisor = 0 then
4 |   |   return False
5 |   end
6 end
7 return True
```

Seguindo os passos propostos na metodologia, na primeira etapa o código foi elaborado na linguagem C. Sendo a detecção feita em forma de função, a qual retorna no caso de o número de entrada ser primo valor 1 ou caso contrário valor 0. Obtendo dessa forma o sistema a ser acelerado, ou seja, o sistema de referência para posteriormente com seu tempo de execução calcular o *speedup*. O código elaborado está listado no Apêndice A, tanto neste código quanto nos posteriores códigos em C é observado uma parte idêntica para inicialização do *timer* além das respectivas variáveis de cada algoritmo, tais partes juntamente com o envio dos dados finais para o Putty não entraram no cálculo do tempo de execução.

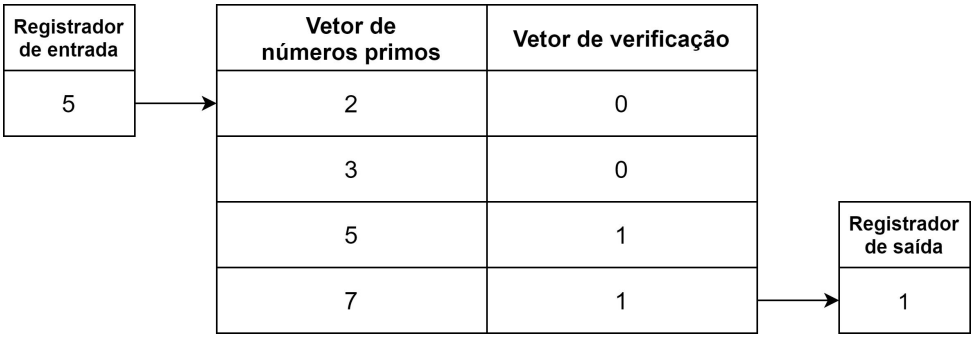
Partindo para a segunda etapa. O código obtido anteriormente serviu como base para identificação da parte crítica de seu processamento. Sendo essa parte crítica a função para identificação do número primo. As demais partes são compostas apenas por rotinas do sistema, as quais vão permanecer como parte F no sistema acelerado.

Com o trecho alvo identificado na etapa dois, na terceira etapa foi elaborado um circuito totalmente combinacional para a identificação do número primo. Como o VHDL não tem um comando especifico em sua linguagem para realizar a divisão de forma combinacional, foi elaborado uma outra maneira de identificação.

A Figura 18 ilustra um exemplo de entrada e saída para o circuito programado. O registrador de entrada recebe um valor 5 advindo do microcontrolador, tal valor percorre todo o vetor contendo os valores de números primos e em caso de igualdade, ou seja, na terceira linha verificasse que 5 é igual a 5, um vetor auxiliar de verificação de mesmo tamanho que o vetor de números primos recebe valor 1, tal valor 1 é repassado até o final do vetor. Ao final das comparações o registrador de saída recebe o valor do último elemento do vetor de verificação completando assim o algoritmo, tal lógica está transcrita em VHDL no Apêndice D. É importante mencionar que é necessário o uso de um vetor auxiliar ao invés de uma única variável, isso ocorre pelo fato do circuito físico implementado não poder receber valores de diversas fontes em uma única variável ao mesmo tempo sem o uso de uma regra lógica especifica.

Após programar o circuito em VHDL foi realizada a síntese do sistema F+V. A Figura 19 traz a taxa de ocupação dos dispositivos lógicos da FPGA para a parte V do sistema.

Figura 18 – Exemplo da lógica projetada para o sistema F+V do detector de números primos.



Fonte: Autoria Própria

Estes números foram obtidos ao sintetizar um sistema contendo um vetor de 4000 números primos. Para um vetor de 5000 posições não foi possível a sintetização, pois os elementos lógicos ocupados extrapolam a capacidade do *kit*, o qual já usa 81% no vetor de 4000 posições. Partindo do 1º número primo, é possível verificar até o número 37813, o qual é o 4000º número primo.

Figura 19 – Síntese da parte V do detector de números primos.

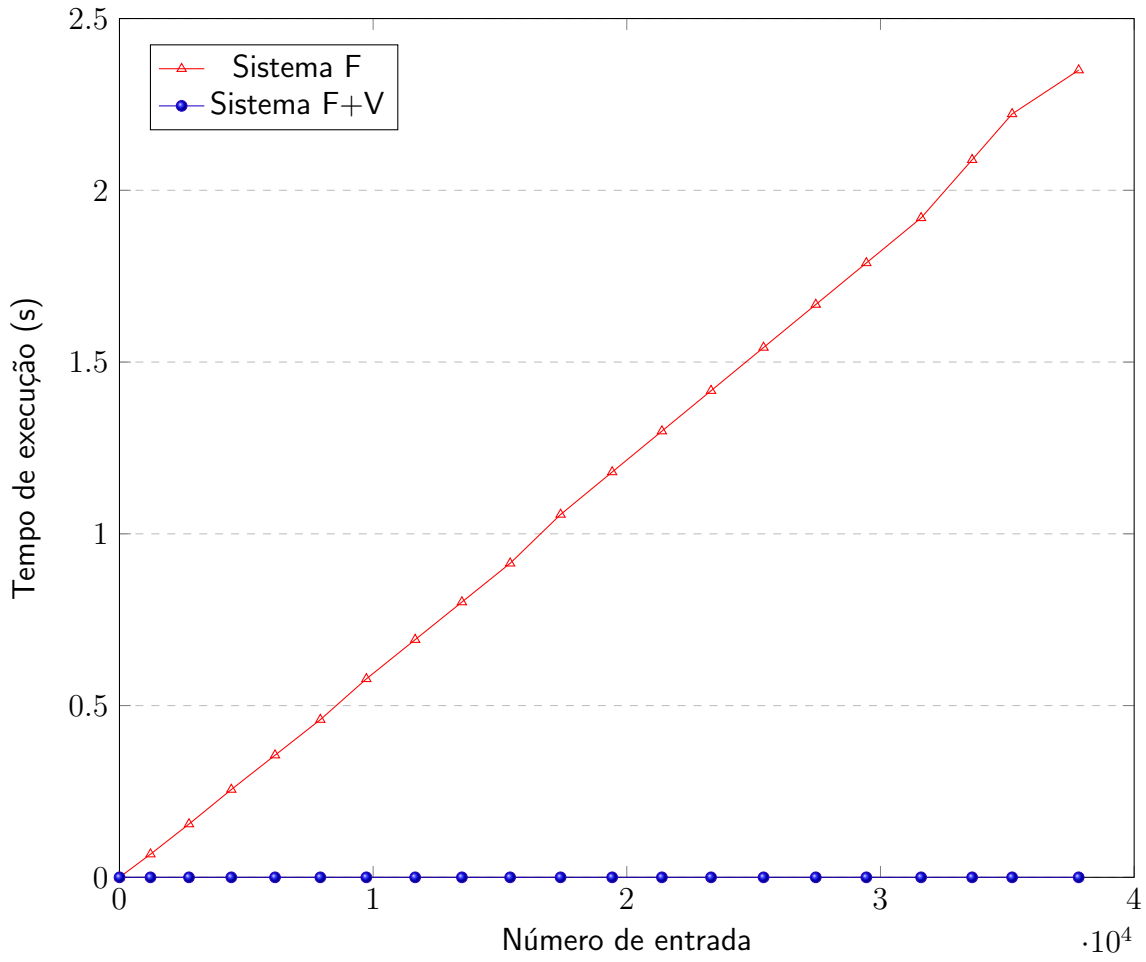
Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	2,739	9,312	29%	
Number of 4 input LUTs	6,017	9,312	64%	
Number of occupied Slices	3,790	4,656	81%	
Number of Slices containing only related logic	3,790	3,790	100%	
Number of Slices containing unrelated logic	0	3,790	0%	
Total Number of 4 input LUTs	6,141	9,312	65%	

Fonte: Extraído do XPS

Concluindo as duas primeiras partes anteriores a terceira etapa é finalizada obtendo as medidas de tempo da execução do sistema F+V, a parte fixa pode ser observada no Apêndice H. O resultado comparativo dos sistemas F e F+V está ilustrado na Figura 20. O gráfico foi elaborado com 21 pontos medidos no intervalo de 2 até 37813, todos os pontos escolhidos são números primos. Como já analisado previamente a execução puramente sequencial cresce linearmente caso a entrada seja um número primo, a linha do sistema F representa a tendência de tempo para tal caso, quando o valor de entrada não é primo o tempo de execução é menor que a linha de tendência. Analisando um número de entrada par no sistema F, o qual representa seu melhor caso, pois na primeira iteração do laço já é obtido o retorno da função, tem um tempo de execução de 14 us. Tendo desempenho inferior para a análise de qualquer número no sistema F+V, tal sistema obteve um tempo de execução de 11 us para qualquer valor de entrada.

Partindo dos resultados do tempo de execução de ambos os sistemas foi realizada a

Figura 20 – Gráfico do tempo de execução do algoritmo detector de números primos.



Fonte: Autoria Própria

quarta e última etapa, onde o valor de *speedup* é calculado. A Tabela 1 traz o cálculo para alguns casos. No melhor caso do sistema F, ou seja a análise do número 2 o sistema F+V obteve um pequeno *speedup* de 1,273. Como o tempo de execução do sistema F+V é constante enquanto o do sistema F cresce linearmente o *speedup* tem um rápido crescimento. Obtendo para o número 37813, o qual é o pior caso no sistema F para a faixa de testes o *speedup* cresceu significativamente para 213619,273.

Tabela 1 – *Speedup* para os números primos.

Número	Tempo no sistema F (s)	Tempo no sistema F+V (s)	*Speedup*
2	0,000014	0,000011	1,273
1223	0,067384	0,000011	6125,818
11657	0,691874	0,000011	62897,636
21383	1,298856	0,000011	118077,818
37813	2,349812	0,000011	213619,273

Fonte: Autoria Própria

4.2 *BUBBLE SORT*

Bubble Sort é um dos problemas clássicos na computação, se trata de um algoritmo de ordenação. Pode ser visto no seu pseudocódigo do Algoritmo 4 que sua estrutura é composta por um laço dentro de outro, ambos dependendo diretamente do tamanho da entrada. Tal estrutura resulta em uma complexidade $O(n^2)$. Sua vantagem é a fácil implementação sendo talvez o método de ordenação mais difundido (SZWARCFITER, 2010).

Basicamente o algoritmo tem a mesma estrutura do Algoritmo 2, ou seja, ele percorre o vetor de tamanho n do início ao fim por n vezes. A cada iteração compara dois números buscando o menor deles caso esteja ordenando de forma crescente ou buscando o maior número, caso a ordenação seja decrescente, colocando-o em sua posição correta no vetor.

Algoritmo 4: *Bubble Sort*

Input: um vetor $V[]$
Output: o vetor $V[]$ ordenado

```
1  N ← tamanho(V[])
2  for i = 0 to N − 1 do
3      for j = i + 1 to N do
4          if V[i] > V[j] then
5              aux = V[i]
6              V[i] = V[j]
7              V[j] = aux
8          end
9      end
10 end
```

Na primeira etapa o sistema F foi elaborado com base no pseudocódigo do *bubblesort*. O algoritmo tem a característica de ter uma complexidade quadrática no seu pior caso, que é quando o vetor está ordenado de forma inversa, necessitando assim fazer a troca de elementos em todas as comparações e uma complexidade linear no seu melhor caso, quando o vetor já está totalmente ordenado não precisando assim realizar qualquer troca. O Apêndice B detalha em código as duas medidas realizadas. Para concluir a primeira etapa as medidas de tempo para o melhor e pior caso foram obtidas.

Prosseguindo para a segunda etapa foi identificado como a parte alvo da transcrição para parte V os dois laços de ordenação. Idealizando transformar estes laços em uma matriz de comparações dentro da FPGA.

Chegando a terceira etapa a matriz de comparações idealizada na etapa dois foi transcrita para VHDL. Este circuito foi projetado para funcionar de forma totalmente combinacional sem depender de um sinal de *clock*. Para isso não seria possível realizar a troca dos elementos dentro da matriz da FPGA, pois a troca afetaria as demais comparações concorrentes. Para proporcionar uma adaptação a saída do circuito representa a posição final de cada número de entrada.

Foi obtida então uma lógica representada na Figura 21 para realizar tal tipo de ordenação. O vetor de entrada é passado para os registradores da FPGA, iniciando todos como posição inicial zero. Em cada coluna os registradores de entrada são comparados com um respectivo valor, na coluna 1 todos os números são comparados com o primeiro registrador, e caso um número seja maior que o valor comparado sua posição é acrescida em uma unidade do seu valor anterior. Como é possível ver o primeiro registrador é o de maior valor então a coluna permanece zerada. Na segunda coluna os valores de entrada são comparados com o segundo registrador, o qual tem valor 2, nesta comparação o primeiro registrador, o qual tem valor 3 é maior que o segundo, resultando assim no acréscimo de sua posição de 0 para 1. Por fim a terceira coluna compara os valores com o terceiro registrador de entrada. Os valores da terceira coluna são transferidos para os registradores de saída, os quais são lidos pelo microcontrolador. Nesta leitura o primeiro registrador tem valor 2, ou seja, a posição correta do vetor de entrada da posição zero é a posição dois. Desta forma o vetor de saída Vs[registrador 0] recebe o vetor de entrada Ve[0], Vs[registrador 1] recebe Ve[1] e Vs[registrador 2] recebe Ve[2] completando desta forma a ordenação. Caso seja feita a comparação de valores iguais aquele que tem maior índice recebe o acréscimo de posição. O Apêndice E demostra tal lógica traduzida para a linguagem VHDL, o qual realiza uma ordenação de um vetor de 4 posições, para escalonar tal código basta adequar o tamanho dos vetores.

Figura 21 – Exemplo da lógica totalmente combinacional projetada para o sistema F+V do *bubblesort*.

Vetor de entrada	Registradores de entrada	Matriz de posições			Registradores de saída	Vetor de saída
		Coluna 1	Coluna 2	Coluna 3		
Ve[0]=3	3	0	1	2	2	Vs[2]=Ve[0]
Ve[1]=2	2	0	0	1	1	Vs[1]=Ve[1]
Ve[2]=1	1	0	0	0	0	Vs[0]=Ve[2]

Fonte: Autoria Própria

Ao sintetizar o circuito foi observado a primeira limitação do *kit*, a falta de dispositivos lógicos. A taxa de ocupação lógica pode ser vista na Figura 22. Por ser um circuito denso que efetua muitas comparações de diversas fontes de entrada ele acaba ocupando muitos elementos lógicos na FPGA. A medida que a entrada cresce linearmente, a matriz de posições aumenta de forma quadrática. Na primeira síntese, realizada para um vetor de 4 posições foi alcançado uma ocupação de 65% nos elementos lógicos, em sequência para um vetor de 8 posições essa taxa subiu para 83%, partindo para o vetor de 16 posições o XPS informou que não foi possível sintetizar o circuito por falta de elementos lógicos.

Como alternativa para o problema encontrado no primeiro projeto de ordenação uma segunda tática foi abordada. Para diminuir o uso de espaço na FPGA optou-se por um sistema misto, realizando um ordenação similar ao trabalho de Mueller, Teubner e Alonso (2012).

Figura 22 – Síntese da parte V do algoritmo de ordenação totalmente combinacional.

Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	2,835	9,312	30%	
Number of 4 input LUTs	3,762	9,312	40%	
Number of occupied Slices	3,059	4,656	65%	
Number of Slices containing only related logic	3,059	3,059	100%	
Number of Slices containing unrelated logic	0	3,059	0%	
Total Number of 4 input LUTs	3,886	9,312	41%	

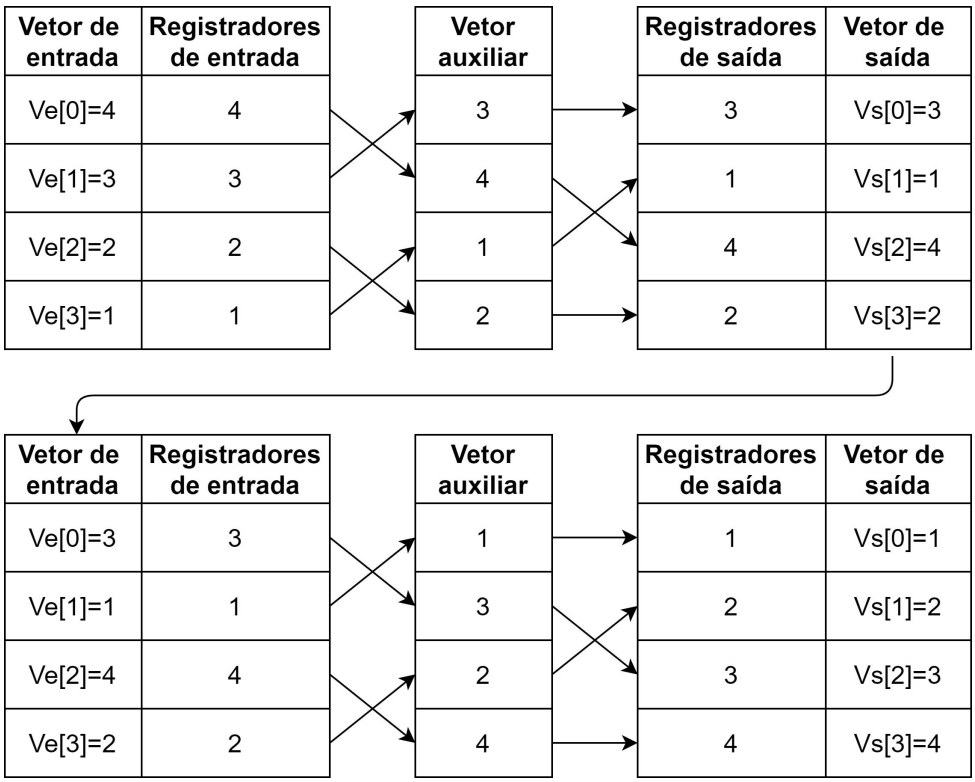
Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	2,981	9,312	32%	
Number of 4 input LUTs	5,695	9,312	61%	
Number of occupied Slices	3,910	4,656	83%	
Number of Slices containing only related logic	3,910	3,910	100%	
Number of Slices containing unrelated logic	0	3,910	0%	
Total Number of 4 input LUTs	5,819	9,312	62%	

Fonte: Extraído do XPS

Representado na Figura 23, três vetores foram implementados dentro da FPGA de mesmo tamanho do vetor de entrada. Nesta abordagem a troca de elementos é feita dentro da própria FPGA, na primeira parte o vetor de entrada é ordenado de forma par-a-par comparando o primeiro com o segundo elemento e o terceiro com o quarto elemento. A comparação é atribuída para um vetor auxiliar. A segunda etapa faz a mesma comparação pulando a primeira posição, no caso ilustrado é comparado apenas o segundo com o terceiro elemento repassando os valores para os registradores de saída. Foi observado que é preciso repetir estes passos $N/2$ vezes, sendo N o tamanho do vetor. Na figura com o vetor de 4 posições foi preciso realizar o processo duas vezes para obter a ordenação. Após cada retorno de ordenação para o microcontrolador, o mesmo repassa os valores de saída novamente para os registradores de entrada até que a ordenação seja concluída. O código elaborado para um vetor de 16 posição, presente no Apêndice F, contém a lógica projetada na etapa dois e transcrita para VHDL na terceira etapa. Assim como o primeiro circuito transcrito em *hardware* para escalonar seu tamanho basta adequar o tamanho dos vetores e variáveis, permanecendo a estrutura interna da lógica de comparação sem alterações.

Para segunda abordagem foi realizada três sínteses com tamanhos de vetor 16, 32 e 64 posições respectivamente. A Figura 24 traz as duas primeiras sínteses. É notado uma diminuição no uso dos dispositivos lógicos em relação a primeira abordagem, possibilitando a sintetização de vetores de maior tamanho. No vetor de 16 posições foram usados 70% dos elementos lógicos e no vetor de 32 posições 90%. Porém chegando ao vetor de 64 posições o *kit* não teve espaço suficiente para sua implementação.

Figura 23 – Exemplo da lógica mista projetada para o sistema F+V do *bubblesort*.



Fonte: Autoria Própria

Figura 24 – Síntese da parte V do algoritmo de ordenação misto.

Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	3,273	9,312	35%	
Number of 4 input LUTs	4,176	9,312	44%	
Number of occupied Slices	3,268	4,656	70%	
Number of Slices containing only related logic	3,268	3,268	100%	
Number of Slices containing unrelated logic	0	3,268	0%	
Total Number of 4 input LUTs	4,300	9,312	46%	

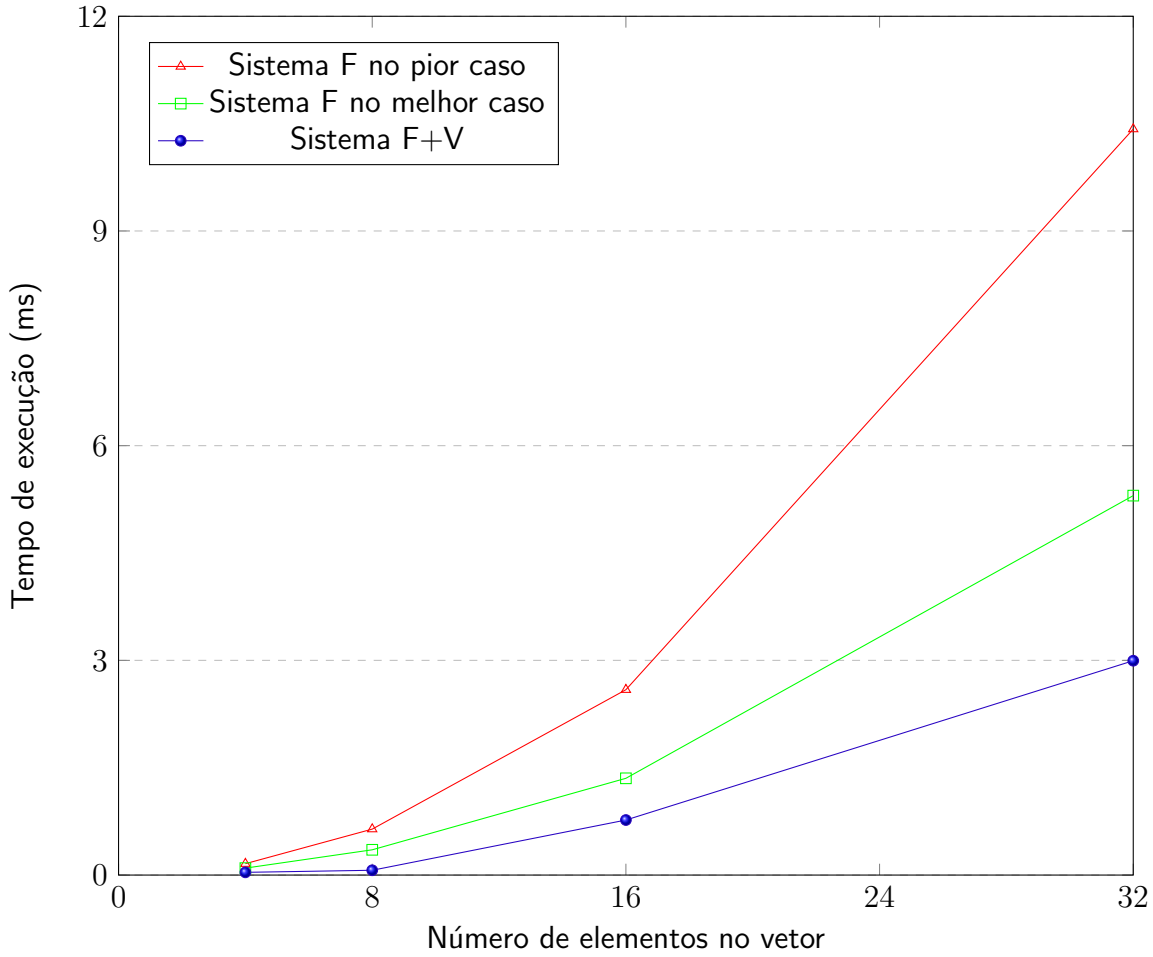
Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	3,929	9,312	42%	
Number of 4 input LUTs	6,427	9,312	69%	
Number of occupied Slices	4,212	4,656	90%	
Number of Slices containing only related logic	4,212	4,212	100%	
Number of Slices containing unrelated logic	0	4,212	0%	
Total Number of 4 input LUTs	6,647	9,312	71%	

Fonte: Extraído do XPS

Finalizando a terceira etapa ambos os Sistemas F+V foram executados. O primeiro sistema, o qual tem a comparação totalmente combinacional executando a ordenação dos

vetores com 4 e 8 posições, o segundo sistema que opera de forma mista para os vetores de 16 e 32 posições. Em ambos os sistemas o arranjo do vetor de entrada não teve influência no tempo de execução ao contrário do que acontece no sistema F. A Figura 25 ilustra o gráfico dos tempos de execuções obtidos. A linha mais acima é o sistema F para o vetor de entrada ordenado de forma inversa, a qual apresentou pior desempenho e a linha intermediária para o vetor de entrada já ordenado corretamente. Para qualquer outro arranjo de vetor o tempo de execução estará contido entre estas duas linhas. A linha mais abaixo é o tempo de execução para o sistema F+V, onde os dois primeiros pontos são para o circuito totalmente combinacional e os dois últimos pontos o circuito misto. Os códigos em C de ambos os sistemas F+V estão no Apêndice I e Apêndice J respectivamente.

Figura 25 – Gráfico do tempo de execução do algoritmo *bubblesort*



Fonte: Autoria Própria

Finalizando o algoritmo do *bubblesort* a quarta etapa realizou o cálculo de *speedup* do sistema F+V. Os cálculos estão divididos em duas tabelas, a Tabela 2 contém o *speedup* do pior caso de entrada no sistema F. Os vetores de tamanho 4 e 8 obtiveram um melhor desempenho, pois o circuito não dependia do *clock* para operar. Ao dobrar o tamanho de vetor foi observado uma melhora de 4,182 para 9,582 no *speedup*, melhorando em mais de duas

vezes o seu desempenho. Nos vetores de 16 e 32 posições o *speedup* permaneceu constante em uma faixa entre 3,3 e 3,4.

Tabela 2 – *Speedup* para o *bubblesort* no seu pior caso.

Tamanho do vetor	Tempo no sistema F (ms)	Tempo no sistema F+V (ms)	*Speedup*
4	0,159	0,038	4,182
8	0,642	0,067	9,582
16	2,590	0,769	3,368
32	10,424	2,995	3,480

Fonte: Autoria Própria

Na Tabela 3 é calculado o *speedup* no melhor caso de entrada para o sistema F. tendo como resultados uma queda de quase duas vezes em comparação ao pior caso, e apresentando o mesmo comportamento em relação aos tipos de sistemas F+V. Como a complexidade do algoritmo sequencial cai de $O(n^2)$ no pior caso, para $O(n)$ no melhor caso, a diferença está dentro das expectativas.

Tabela 3 – *Speedup* para o *bubblesort* no seu melhor caso.

Tamanho do vetor	Tempo no sistema F (ms)	Tempo no sistema F+V (ms)	*Speedup*
4	0,097	0,038	2,553
8	0.353	0,067	5,269
16	1,352	0,769	1,758
32	5,302	2,995	1,770

Fonte: Autoria Própria

4.3 *Z BUFFER*

Algoritmo desenvolvido por Catmull (1974) em sua tese de doutorado. Consiste em armazenar em uma matriz todos os elementos visíveis diante de um ponto de vista. Nesta matriz, o valor armazenado é a intensidade do objeto, valores maiores sobrepõem os menores gerando assim a superfície visível que será renderizada.

No Algoritmo 5 o pseudocódigo demonstra o funcionamento do *Z buffer*. A Matriz de Profundidade armazena o valor de intensidade dos objetos sobrepostos no plano de fundo. A Matriz de Cor armazena a cor do objeto que sobrepôs os demais. A Lista de Polígonos representa os objetos que irão renderizar contendo as coordenadas x e y de cada ponto, o valor de sua intensidade diante do ponto de vista do observador e a cor do objeto. Por conter um primeiro laço na linha 3 até a 10 percorrendo n polígonos, sendo que a cada iteração ele percorre outro laço linhas 4 a 9 contendo as n posições possíveis para cada polígono a complexidade é definida por $O(n^2)$, pois os custos de inicialização das matrizes de profundidade

e cor são menores que a do laço, desta forma é suprimida pelo princípio da absorção, assim como acontece a comparação condicional interna aos laços.

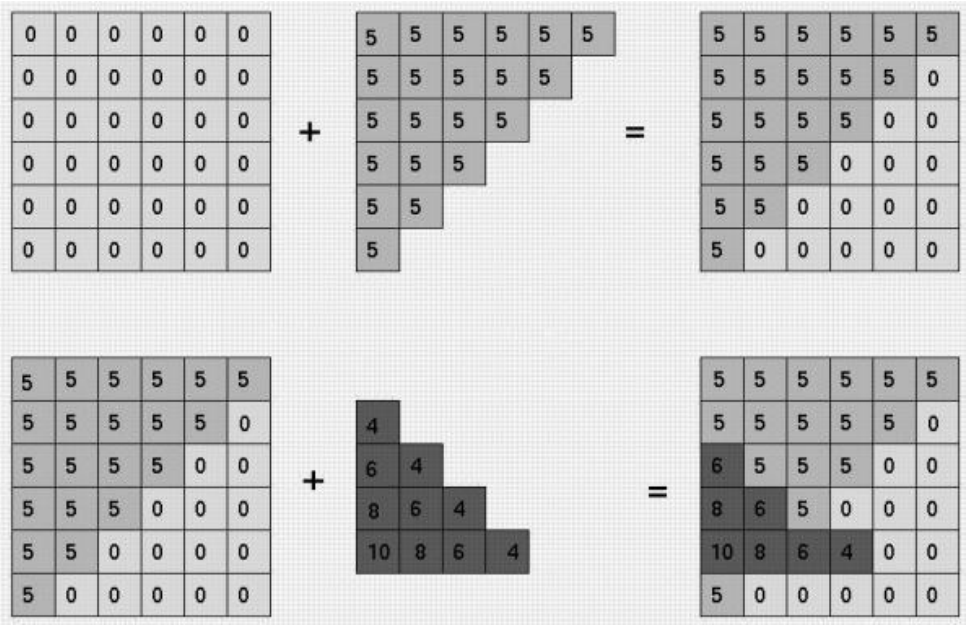
Algoritmo 5: *Z buffer*

Input: matriz Profundidade[x, y]
matriz Cor[x, y]
lista de Polígonos(P1, P2, ... Pn)
Output: matrizes de cor e profundidade com a projeção dos polígonos

```
1  Profundidade[x,y] ← 0
2  Cor[x,y] ← Cor de fundo
3  for cada P ∈ Poligonos do
4  |  for cada posicao (x,y) ∈ P do
5  |  |  if Profundidade Polígono Pn(x,y) > Profundidade[x,y] then
6  |  |  |  Profundidade[x,y] = Profundidade Poígono Pn(x,y)
7  |  |  |  Cor[x,y] = Cor Poígono Pn(x,y)
8  |  |  end
9  |  end
10 end
```

Por sua simplicidade na implementação, é um dos algoritmos de renderização de imagens mais usados (HASSELGREN; AKENINE-MöLLER, 2006). Na Figura 26 é representada a adição de dois objetos. O primeiro objeto adicionado sobrepõe o plano de fundo nas suas coordenadas, em seguida no segundo objeto os valores onde a intensidade na coordenada é menor ao dos valores contidos no *buffer* foram sobrepostos.

Figura 26 – Renderização de 2 polígonos com o *Z Buffer*



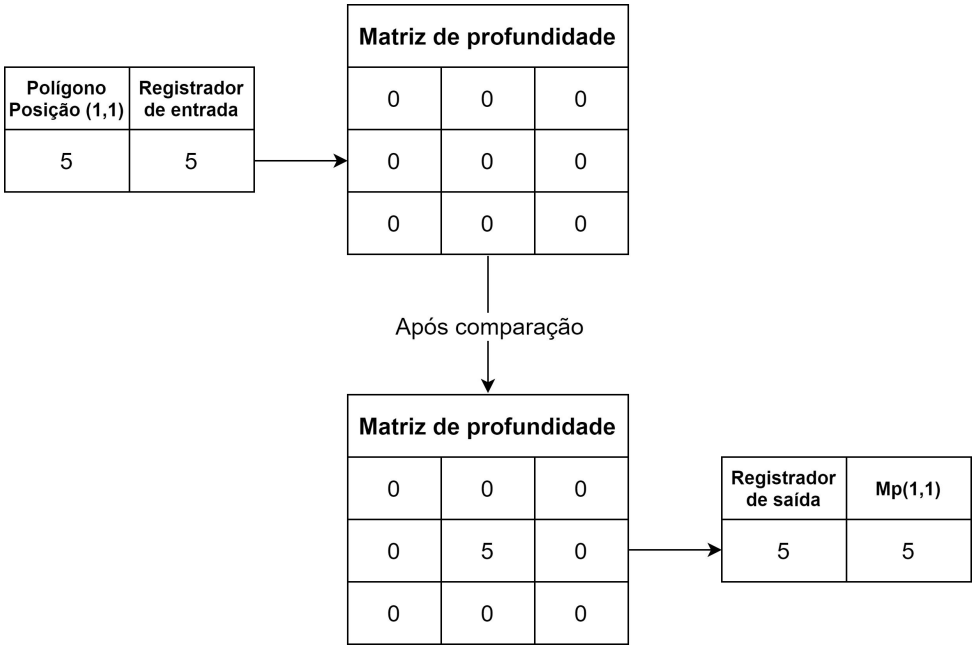
Fonte: Yauri e Nakashima (2011)

Para a primeira etapa o algoritmo foi implementado e executado no *MicroBlaze*. Afim de facilitar a sua implementação cada elemento na lista de polígonos contém uma posição (x,y), sua profundidade e cor. Cada elemento equivale a um ponto na tela de exibição a ser renderizado, ou seja, um *pixel*. A lista foi criada de duas formas, a primeira com todos os valores aleatórios e a segunda com um valor de profundidade e cor pré-definidos com um tamanho que ocupe todos os *pixels* disponíveis. A geração de polígonos juntamente com o código do sistema F pode ser visualizado no Apêndice C.

Após a execução do sistema F, seguiu-se para segunda etapa. Foi identificado como parte crítica de processamento a inserção de um novo elemento na matriz de profundidade, consistindo tal inserção na adição um *pixel* .

Partindo para terceira etapa o trecho crítico identificado anteriormente foi idealizado em uma transcrição em *hardware* como ilustrado na Figura 27. A lógica consiste em receber um valor do registrador de entrada contendo a posição do elemento na matriz, sua profundidade e sua cor, porém ao executar o sistema F+V contendo a análise de profundidade e cor dentro da parte V observou-se erros, então optou-se por realizar apenas a análise de profundidade na parte V. De acordo com o ilustrado na figura a transcrição para VHDL foi realizada, a mesma consta no Apêndice G. Deixando a análise de cor para parte F. A parte V funciona de forma combinacional onde caso o valor de entrada seja maior que o valor armazenado ele realiza a troca, passando para o registrador de saída o valor da matriz contido na posição (x,y) passada como parâmetro.

Figura 27 – Exemplo da lógica projetada para o sistema F+V do *Z buffer*.



Fonte: Autoria Própria

A Figura 28 mostra a síntese da parte V para o *Z buffer*. O tamanho de matriz sintetizado foi de 8x8, totalizando 64 *pixels*. Tendo os elementos lógicos com uma ocupação em

70%. A tentativa de sintetizar uma matriz 16x16 falhou pois o *kit* não teve espaço suficiente para o circuito.

Figura 28 – Síntese da parte V do *Z buffer*.

Device Utilization Summary (actual values)				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	3,256	9,312	34%	
Number used as Flip Flops	2,744			
Number used as Latches	512			
Number of 4 input LUTs	4,090	9,312	43%	
Number of occupied Slices	3,300	4,656	70%	
Number of Slices containing only related logic	3,300	3,300	100%	
Number of Slices containing unrelated logic	0	3,300	0%	
Total Number of 4 input LUTs	4,214	9,312	45%	

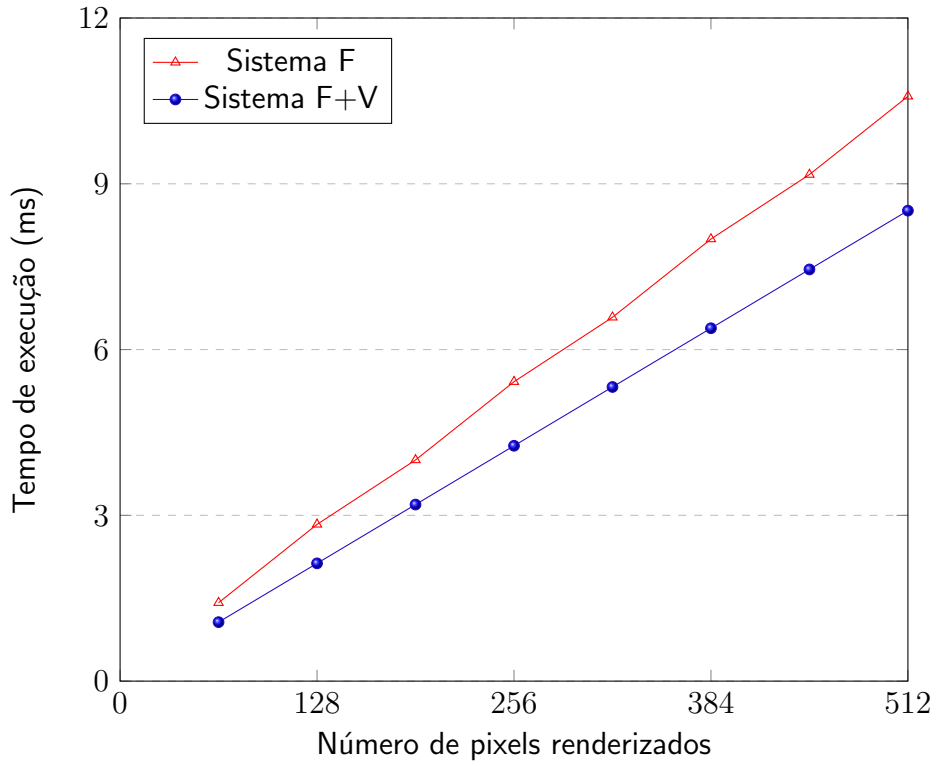
Fonte: Extraído do XPS.

Para finalizar a terceira etapa foram realizados três tipos de testes, o primeiro gerando polígonos de forma aleatória falhou pois o resultado final da matriz de profundidade estava errado. O segundo teste foi para o desempenho considerando somente o cálculo de profundidade da matriz e o terceiro considerando a profundidade e cor das matrizes. Tanto o segundo quanto terceiro testes usaram uma lista de polígonos controlada. Tendo o intuito de facilitar a visualização do resultado, cada polígono tinha tamanho de 64 *pixels* e um valor de cor e profundidade predefinido e igual para todos os *pixels*, gerados de tal forma que em cada polígono gerado a troca de valor na matriz de profundidade seria uma vez necessária e na seguinte não. A configuração em código para tais polígonos e o teste dos mesmos está contido no Apêndice K.

No caso de analisar o desempenho apenas do cálculo de profundidade obteve-se o seguinte gráfico da Figura 29, sendo o sistema F+V mais rápido que o sistema F. Para o cálculo de profundidade e cor os tempos de execuções estão representados na Figura 30, obtendo uma piora no desempenho do sistema F+V em relação ao sistema F. Em ambos os testes foram renderizados 8 polígonos, contendo 64 *pixels cada*. Para diminuir o gargalo das chamadas de função, as informações de posição e profundidade foram repassadas em apenas uma variável do tipo inteiro, a qual tem 32 bits. Os bits de número 17 ao 24 contém a posição e os bits de número 25 ao 32 a profundidade.

Finalizando o processo com a quarta etapa, o *speedup* foi calculado para ambos os testes. A Tabela 4 traz o *speedup* na análise do cálculo somente de profundiade, tendo um desempenho praticamente constante entre 1,209 e 1,330. Por sua vez a Tabela 5 contém o desempenho ao analisar o cálculo do algoritmo completo, ou seja, tanto cor quanto profundidade, neste caso o *speedup* obtido mostrou uma piora no desempenho, com um valor máximo de 0,770.

Figura 29 – Gráfico do tempo de execução do algoritmo *Z buffer* calculando apenas a profundidade.



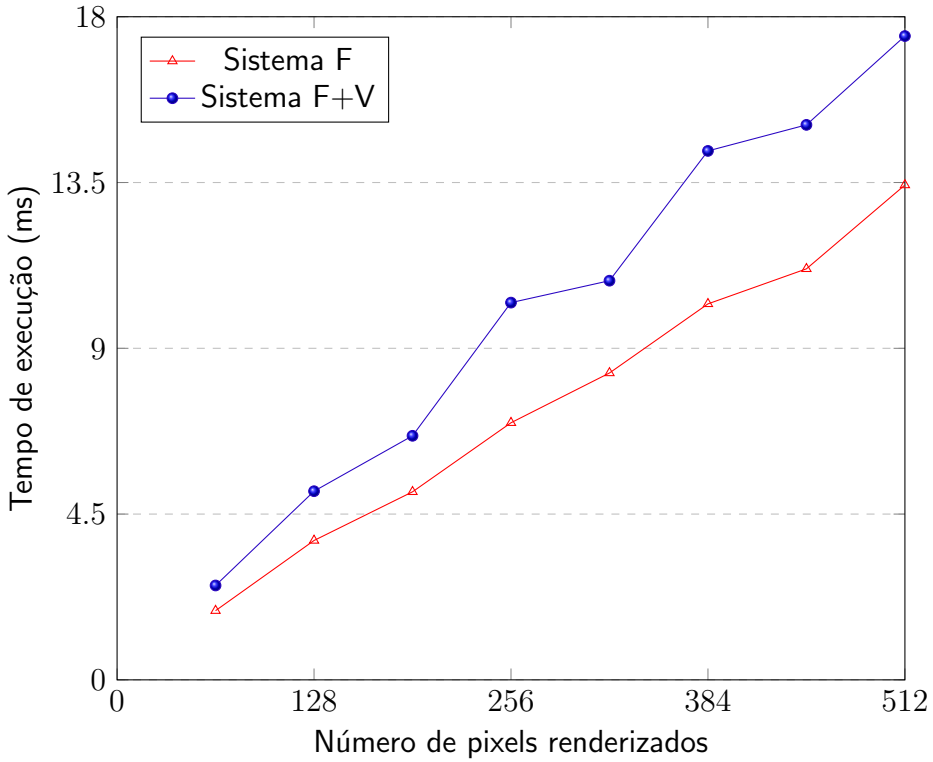
Fonte: Autoria Própria

Tabela 4 – *Speedup* para o *Z buffer* calculando apenas profundidade.

Número de pixels	Tempo no sistema F (ms)	Tempo no sistema F+V (ms)	*Speedup*
64	1,419	1,067	1,329
128	2,836	2,131	1,330
192	4,002	3,195	1,252
256	5,148	4,259	1,209
320	6,584	5,323	1,236
384	8,000	6,386	1,252
448	9,167	7,450	1,230
512	10,583	8,514	1,243

Fonte: Autoria Própria

Figura 30 – Gráfico de execução do algoritmo *Z buffer* calculando profundidade e cor.



Fonte: Autoria Própria

Tabela 5 – *Speedup* para o *Z buffer* calculando profundidade e cor.

Número de *pixels*	Tempo no sistema F (ms)	Tempo no sistema F+V (ms)	*Speedup*
64	1,878	2,562	0,733
128	3,783	5,121	0,738
192	5,104	6,625	0,770
256	6,979	10,240	0,681
320	8,330	10,835	0,768
384	10,206	14,258	0,715
448	11,157	15,065	0,740
512	13,432	17,476	0,768

Fonte: Autoria Própria

5 CONCLUSÃO

Todos os problemas tiveram um trecho crítico em processamento transcrito para um *hardware* correspondente. De acordo com a própria fabricante do *kit*, a *Spartan-3E* é direcionada para o uso em programas de lógica intensa e poucas entradas e saídas (XILINX, 2021a). Isto pode ser verificado nos três problemas propostos de acordo com as Figura 19, Figura 22, Figura 24 e Figura 28. Cada algoritmo cresce gradativamente em relação ao anterior no uso das interfaces de entradas e saídas, ao aumentar tais interfaces o *kit* dispõe de menos espaço para a implantação da lógica interna, na qual a parte crítica é transcrita.

Para o detector de números primos é possível ao analisar o gráfico da Figura 20 e a Tabela 1 concluir que seu desempenho foi amplamente superior a versão puramente sequencial, conseguindo um tempo de execução constante de 11 us para todas as entradas. Pelo fato de usar apenas um registrador de entrada e outro para saída tornou-se possível realizar um teste de maior escala em relação aos outros problemas, sendo possível alocar no *kit* dois vetores de 4000 posições e realizar a comparação do valor de entrada com todos os elementos.

No *bubblesort* foi possível notar de acordo com o gráfico da Figura 25 uma aceleração em ambos os dois sistemas F+V implementados, porém as limitações do *kit* impossibilitaram testes em maior escala. Pela Tabela 3 e Tabela 2 pode ser extraído a informação de que sistema puramente combinacional apresentou uma melhor performance a custo do uso de mais componentes lógicos, aproximadamente dobrando seu desempenho ao dobrar o tamanho da entrada, caso o circuito siga este comportamento infinitamente é possível estimar um *speedup* de aproximadamente 1000 vezes em um vetor de 1024 posições. Para o segundo sistema, o sistema misto foi preciso realizar $n/2$ ciclos para sua conclusão. Pelas Figura 22 e Figura 24 nota-se que ele ocupa uma menor quantidade de dispositivos lógicos em relação ao primeiro sistema, pois foram possíveis testes com maiores instancias, em contra partida tem um menor desempenho. Ao analisar o sistema F+V por sua complexidade é possível obter um custo de $O(n)$, já que o algoritmo precisa de $n/2$ iterações para terminar sua tarefa. Comparando o segundo circuito elaborado com o melhor caso do *bubble sort*, o qual tem custo $O(n)$ apresenta um *speedup* na casa de 1,7 vezes indo de acordo com o esperado, já que em teoria o sistema F+V realiza metade dos ciclos de *clock* que o sistema F.

Por fim o *Z buffer* foi o que apresentou mais problemas não funcionando perfeitamente, inclusive ficando mais lento que o sistema de referência, tais desempenhos estão apresentados nos gráficos da Figura 30 e Figura 29. Pelo fato de ser um algoritmo muito volátil em relação as suas entradas e saídas, fator este determinante no uso da *Spartan-3E* foi o algoritmo que teve o pior desempenho nos ensaios realizados. Analisando somente a parte transcrita em *hardware*, calculada na Tabela 4 foi obtida uma pequena aceleração, precisando inclusive comprimir as variáveis de posição, profundidade e cor em uma única variável por meio de operações de deslocamento de bits para minimizar a perca de tempo da chamada de função de escrita dos

registradores de entrada. A causa do erro ao inserir polígonos de formas aleatórias não foi identificada.

Todos os Sistemas F+V apresentavam um tempo teórico de execução menor que o tempo obtido na terceira etapa, tal tempo é estimado pelo XPS após a geração do *bitstream* ficando sempre menor que 100 ns. Esta diferença foi causada pela lentidão da função de escrita e leitura dos registradores que interligam o *MicroBlaze* com os *hardwares* transcritos. Os algoritmos do *bubblesort* e *Z bufer* por terem uma lógica simples, podem ser implementados na *Spartan-3A*, a qual é focada em aplicações onde a entrada e saída de dados é mais crucial que a densidade lógica do *hardware* (XILINX, 2021a). Para escalonar tais problemas, e possivelmente obter melhor desempenho seria necessário o uso de um *kit* mais moderno, como por exemplo a *Virtex UltraScale+ VU19P* (XILINX, 2021b). Tal *kit* dispõe de mais de 9 milhões de células lógicas, além de fornecer um alto desempenho tanto em densidade lógica quanto em velocidade de entrada e saída de dados, muito superior a *Spartan-3E*, a qual contém cerca de 10 mil células lógicas e baixo desempenho em entrada e saída dados.

Todas as implementações foram obtidas conforme pode ser observado no Capítulo 4. Das dificuldades encontradas a falta de elementos lógicos no *kit* e o baixo desempenho na comunicação entre as partes F e V foram as que mais afetaram nos resultados. O pouco espaço do *kit* impossibilitou testes de maiores instancias para os problemas do *bubble sort* e *Z buffer*. O baixo desempenho do *MicroBlaze* na comunicação com os blocos VHDL encobriram o potencial máximo dos sistemas F+V implementados, buscou-se alternativas em trocar a função fornecida pelo fabricante para escrita e leitura dos registradores por um método de acesso direto aos registradores, porém não foi obtido sucesso.

5.1 TRABALHOS FUTUROS

Com base na flexibilidade da realização dos ensaios, o método utilizado com êxito e a estrutura básica criada para todos os problemas foi possível ter a base de um *framework* para o desenvolvimento de futuras aplicações de aceleração. Podendo pôr em prática problemas de natureza mais complexa, preferencialmente para o caso da *Spartan-3E* como observado nos ensaios, problemas de alta carga de processamento e poucas interfaces de entrada e saída, a exemplo do detector de números primos.

Referências

ALI, F. H. Depth buffer dda based on fpga. **Al-Rafidain Engineering**, v. 19, n. 5, p. 1–12, 2011. Disponível em: <https://rengj.mosuljournals.com/article_26743_f1a29afb3ef5006a332644ebd016d476.pdf>. Citado na página 17.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1967. (AFIPS '67 (Spring)), p. 483–485. ISBN 9781450378956. Disponível em: <https://doi.org/10.1145/1465482.1465560>. Citado na página 17.

CATMULL, E. E. **A Subdivision Algorithm for Computer Display of Curved Surfaces.** Dezembro, 1974. 83 p. Tese (Doutorado) — The University of Utah, Salt Lake City, Utah, 1974. Citado na página 34.

CHU, P. P. **FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version**. 1. ed. Hoboken, New Jersey, USA: Wiley-Interscience, 2008. ISBN 9780470185315. Citado na página 2.

CORMEN, T. H. **Algoritmos : teoria e pratica**. 2. ed. Rio de Janeiro: Elseiver, 2002. ISBN 9788535209266. Citado na página 5.

D'AMORE, R. **VHDL: descrição e síntese de circuitos digitais**. 2. ed. Rio de Janeiro: Grupo Gen - LTC, 2012. ISBN 9788521620549. Citado 4 vezes nas páginas 13, 14, 15 e 16.

Englund, H.; Lindskog, N. Secure acceleration on cloud-based fpgas – fpga enclaves. In: **2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [s.n.], 2020. p. 119–122. Disponível em: <https://ieeexplore.ieee.org/document/9150330>. Citado na página 18.

ESTRIN, G. Organization of computer systems: The fixed plus variable structure computer. In: **Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1960. (IRE-AIEE-ACM '60 (Western)), p. 33–40. ISBN 9781450378697. Citado 3 vezes nas páginas , 3 e 8.

ESTRIN, G. Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer. **IEEE Annals of the History of Computing**, v. 24, n. 4, p. 3–9, 2002. Citado na página 8.

HASSELGREN, J.; AKENINE-MöLLER, T. Efficient depth buffer compression. **Lund University**, Setembro 2006. Citado na página 35.

KIRK, D. B.; HWU, W. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 1. ed. Burlinfton, MA, USA: Morgan Kaufmann Publishers, 2010. ISBN 0123814723. Citado na página 2.

LI, Y. et al. A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. **J. Emerg. Technol. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 2, jul. 2018. ISSN 1550-4832. Disponível em: <https://doi.org/10.1145/3154839>. Citado na página 18.

LIPU, A. et al. Exploiting parallelism for faster implementation of bubble sort algorithm using fpga. In: **2nd International Conference on Electrical, Computer & Telecommunication Engineering**. Bangladesh: [s.n.], 2016. p. 1–4. Disponível em: <https://www.researchgate.net/publication/315512656_Exploiting_parallelism_for_faster_implementation_of_Bubble_sort_algorithm_using_FPGA>. Citado na página 17.

MCCOOL, M.; REINDERS, J. **Structured Parallel Programming Patterns for Efficient Computation**. 1. ed. Waltham, MA, USA: Morgan Kaufmann, 2012. ISBN 9780124159938. Citado 2 vezes nas páginas 17 e 18.

MUELLER, R.; TEUBNER, J.; ALONSO, G. Sorting networks on fpgas. **VLDB J.**, v. 21, p. 1–23, 02 2012. Disponível em: <http://dbis.cs.tu-dortmund.de/cms/en/publications/2012/sorting-networks/sorting-networks.pdf>. Citado 2 vezes nas páginas 17 e 30.

PEDRONI, V. A. **Digital electronics and design with VHDL**. 1. ed. Burlinfton, MA, USA: Morgan Kaufmann Publishers, 2008. ISBN 9780123742704. Citado 2 vezes nas páginas 2 e 10.

PEDRONI, V. A. **Circuit Design and Simulation with VHDL**. 2. ed. Cambridge, MA, USA: The MIT Press, 2010. ISBN 9780262014335. Citado 6 vezes nas páginas 2, 8, 10, 11, 12 e 14.

PUTTY. 2021. Disponível em: <https://www.putty.org/>. Acesso em: 01 de maio de 2021. Citado na página 20.

SKLIAROVA, I.; FERRARI, A. B. Introdução à computação reconfigurável. **Revista do DETUA**, v. 2, n. 6, p. 1–16, 2003. Citado na página 7.

STALLINGS, W. **Arquitetura e organização de computadores**. 8ª. ed. São Paulo: Pearson Prentice Hall, 2010. ISBN 9788576055648. Citado 2 vezes nas páginas 1 e 17.

SZWARCFITER, J. L. **Estruturas De Dados E Seus Algoritmos**. 3. ed. Rio de Janeiro, Brasil: LTC, 2010. ISBN 9788521629948. Citado 2 vezes nas páginas 5 e 29.

TANENBAUM, A.; STEEN, M. V. **Sistemas Distribuídos: Princípios e Paradigmas**. 2. ed. São Paulo: Pearson Pratice Hall, 2007. ISBN 9788576051428. Citado na página 1.

TOSCANI, L. V. **Complexidade de Algoritmos**. 3. ed. Porto Alegre, Brasil: Bookman, 2012. ISBN 9788540701397. Citado na página 5.

XILINX. **ISE In-Depth Tutorial**. San Jose, CA, USA, 2009. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise11tut.pdf>. Citado 2 vezes nas páginas 8 e 20.

XILINX. **Spartan-3E FPGA starter kit board user guide**. San Jose, CA, USA, 2011. Disponível em: <https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf>. Citado 2 vezes nas páginas 7 e 20.

XILINX. **EDK Concepts, Tools, and Techniques**. San Jose, CA, USA, 2013. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/edk_ctt.pdf>. Citado na página 10.

XILINX. **MicroBlaze Processor Reference Guide**. San Jose, CA, USA, 2013. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf>. Citado na página 7.

XILINX. **Spartan-3 FPGA Family**. 2021. Disponível em: <https://www.xilinx.com/products/silicon-devices/fpga/spartan-3.html>. Acesso em: 31 de julho de 2021. Citado 2 vezes nas páginas 40 e 41.

XILINX. **Virtex UltraScale+ VU19P**. 2021. Disponível em: <https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html>. Acesso em: 31 de julho de 2021. Citado na página 41.

YAURI, E.; NAKASHIMA, N. **Visibilidade em Computação Gráfica**. 2011. Disponível em: <https://www.dca.fee.unicamp.br/courses/IA725/1s2011/projetos/vidalon-nakashima/final.pdf>. Acesso em: 16 de abril de 2021. Citado na página 35.

ZHANG, L. et al. Fpga acceleration of cnns-based malware traffic classification. **Electronics**, v. 9, p. 1631, 10 2020. Disponível em: <https://www.mdpi.com/2079-9292/9/10/1631>. Citado na página 18.

Apêndices

APÊNDICE A – Código em linguagem C para o sistema F do detector de números primos.

```c
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#define NUMTESTE 37813

int teste_primo_c(int x){
    int i;
    for(i=2; i<=x/2; i++){
        if(x%i==0){
            return 0;
        }
    }
    return 1;
}

int main(){
    //inicio da inicializacao do timer
    XTmrCtr xps_timer_0;
    XTmrCtr* timer_0 = &xps_timer_0;
    Xuint32 BeginTime, EndTime, Calibration, TimeRun;
    int status;
    status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
    xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
    XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    Calibration = EndTime - BeginTime;
    //fim da inicializacao do timer

    int b, i;
    BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    b = teste_primo_c(NUMTESTE);
    EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    if(b==1){
        xil_printf("%d -> %d\r\n", NUMTESTE, b);
    }
    if(b==0){
        xil_printf("%d -> %d\r\n", NUMTESTE, b);
    }

```

```
43      TimeRun = (EndTime - BeginTime - Calibration);
44      xil_printf("Tempo para exec. no uC: %d us\r\n", TimeRun/50);
45      //base de tempo 1 clock = 20ns, ou seja 50 clks = 1 us
46
47      return 0;
48  }
```

APÊNDICE B – Código em linguagem C para o sistema F do bubble sort.

```c
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#define SIZE 4

int teste_sort_c(int *vet_c){
    int aux, i, j;
    for (i=0; i<SIZE-1; i++){
        for(j=i+1; j<SIZE; j++){
            if(vet_c[i] > vet_c[j]){
                aux = vet_c[i];
                vet_c[i] = vet_c[j];
                vet_c[j] = aux;
            }
        }
    }
    return 1;
}

int main(){
    //inicio da inicializacao do timer
    XTmrCtr xps_timer_0;
    XTmrCtr* timer_0 = &xps_timer_0;
    Xuint32 BeginTime, EndTime, Calibration, TimeRun;
    int status;
    status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
    xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
    XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    Calibration = EndTime - BeginTime;
    //fim da inicializacao do timer

    //inicializacao de variaveis
    int i, vet_c[SIZE], b;

    //preenchimento do vetor em ordem decrescente
    for(i=0; i<SIZE ; i++){
        vet_c[i]=SIZE-i;
    }
    xil_printf("\r\nVetor de entrada em ordem decrescente\r\n");

```

```
44      //execucao do sistema F
45      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
46      b = teste_sort_c(vet_c);
47      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
48
49      //print do vetor de saida
50      for(i=0; i<SIZE; i++){
51          xil_printf("vet_c[%d] = %d\r\n", i, vet_c[i]);
52      }
53
54      TimeRun = (EndTime - BeginTime - Calibration);
55      xil_printf("Tempo de exec. no uC: %d us\r\n\n", TimeRun/50);
56
57      xil_printf("\r\nVetor de entrada em ordem crescente\r\n");
58
59      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
60      b = teste_sort_c(vet_c);
61      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
62
63      for(i=0; i<SIZE; i++){
64          xil_printf("vet_c[%d] = %d\r\n", i, vet_c[i]);
65      }
66
67      TimeRun = (EndTime - BeginTime - Calibration);
68      xil_printf("Tempo de exec. no uC: %d us\r\n\n", TimeRun/50);
69
70      return 0;
71  }
```

APÊNDICE C – Código em linguagem C para o sistema F do Z buffer.

```c
#include <stdio.h>
#include <stdlib.h>
#include "xparameters.h"
#include "xtmrctr.h"
#define Msize 8
#define numPolig 512

typedef struct{
    int x;
    int y;
    int prof;
    int cor;
} ListPolig;

ListPolig poligono[numPolig];
int Mdepth_c[Msize][Msize], Mcolor_c[Msize][Msize];

void GeneratePoligPadrao(int prof, int cor, int indice){
    int i, j, k=indice;
    for(i=0; i<Msize; i++){
        for(j=0; j<Msize; j++){
            poligono[k].x = i;
            poligono[k].y = j;
            poligono[k].prof = prof;
            poligono[k].cor = cor;
            k++;
        }
    }
}

void GeneratePoligAleatorio(){
    int i;
    srand(10);
    for(i = 0; i < numPolig; i++){
        poligono[i].x = rand()%Msize;
        poligono[i].y = rand()%Msize;
        poligono[i].prof = rand()%256;
        poligono[i].cor = rand()%256;
    }
}

void ZeraMatriz(){
    int i, j;
```

```
44      for(i=0; i<Msize ; i++){
45          for(j=0; j<Msize; j++){
46              Mdepth_c[i][j] = 0;
47              Mcolor_c[i][j] = 0;
48          }
49      }
50  }
51
52  int main(){
53      //inicio da inicializacao do timer
54      XTmrCtr xps_timer_0;
55      XTmrCtr* timer_0 = &xps_timer_0;
56      Xuint32 BeginTime, EndTime, Calibration, TimeRun;
57      int status;
58      status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
59      xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
60      XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
61
62      BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
63      EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
64      Calibration = EndTime - BeginTime;
65      //fim da inicializacao do timer
66
67      int i, j, k;
68
69      ZeraMatriz();
70
71      //GeneratePoligAleatorio();
72      //gera 64 poligonos por funcao
73      GeneratePoligPadrao(40, 40, 0);
74      GeneratePoligPadrao(50, 50, 64);
75      GeneratePoligPadrao(30, 30, 128);
76      GeneratePoligPadrao(60, 60, 192);
77      GeneratePoligPadrao(20, 20, 256);
78      GeneratePoligPadrao(70, 70, 320);
79      GeneratePoligPadrao(10, 10, 384);
80      GeneratePoligPadrao(80, 80, 448);
81
82      int X, Y, Prof, Cor;
83
84      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
85
86      for(k=0; k<numPolig; k++){
87          X = poligono[k].x;
88          Y = poligono[k].y;
89          Prof = poligono[k].prof;
90          Cor = poligono[k].cor;
```

```
91          if(Prof > Mdepth_c[X][Y]){
92              Mdepth_c[X][Y] = Prof;
93              Mcolor_c[X][Y] = Cor;
94          }
95      }
96
97      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
98
99      xil_printf("Matriz Profundidade C\r\n");
100     for(i=0; i<Msize; i++){
101         for(j=0; j<Msize; j++){
102             xil_printf("%d  ", Mdepth_c[i][j]);
103         }
104         xil_printf("\r\n");
105     }
106
107     xil_printf("\r\nMatriz Cor C\r\n");
108         for(i=0; i<Msize; i++){
109         for(j=0; j<Msize; j++){
110             xil_printf("%d  ", Mcolor_c[i][j]);
111         }
112         xil_printf("\r\n");
113     }
114
115     TimeRun = (EndTime - BeginTime - Calibration);
116     xil_printf("Tempo para exec. no uC: %d us\r\n\n", TimeRun/50);
117
118     return 0;
119 }
```

APÊNDICE D – Código em linguagem VHDL da parte V do detector de números primos.

```
architecture IMP of user_logic is
    --USER signal declarations added here, as needed for user logic
    type bit_array is array (0 to 4000) of std_logic;
    type int_array is array (0 to 4000) of integer range 0 to 100000;
    --0 a 4000 pois sao alocados 4 mil numeros primos no vetor
    signal result: bit_array; -- vetor de verificacao
    signal primes: int_array; -- vetor dos numeros primos
    signal enter_value: integer; -- valor do registrador de entrada

begin
  --USER logic implementation added here
  --atribuicao dos valores iniciais ao vetor primos.
  primes(0) <= 1;
  primes(1) <= 2;
  primes(2) <= 3;

  -- processo para preenchimento do vetor de primos.
  -- processo executa apenas uma vez.
  GeneratePrimeList: process
        variable cont: integer;
        variable index: integer;
  begin
        index := 3; --posicao do vetor
        --laco de verificacao do vetor
        --o valor limite depende do tamanho de vetor alocado
        --como o vetor tem 4000 primos ele verifica ate o
        --4 milesimo numero primo +1, 37814
        for i in 4 to 37814 loop
                cont := 0;
                for j in 2 to i/2 loop
                        if (i mod j = 0) then
                                cont := 1;
                        end if;
                end loop;
                if (cont = 0) then -- se for primo atribui valor no vetor
                        primes(index) <= i;
                        index := index + 1;
                end if;
        end loop;
  end process;

  --primeiro processo pode tem uma grande demora na sintetizacao
```

```
43   --o vetor pode ser povoado explicitamente definindo seus valores
44   --primes(3) <= 5; ... primes(4000) <= 37813;
45
46   -- recebe o valor de entrada e o converte de std_logic_vector para integer
47   enter_value <= conv_integer(slv_reg0);
48   result(0) <= '0';
49
50   --lista para verificacao se o valor de entrada eh primo
51   GenSearchList: for i in 1 to 1000 generate
52     result(i) <= '1' when enter_value = primes(i) else result(i-1);
53   end generate;
54
55   -- atribuicao do resutlado ao registrador de saida
56   slv_reg1 <= "00000000000000000000000000000001" when result(4000) = '1'
57               else "00000000000000000000000000000000";
58 end IMP;
```

APÊNDICE E – Código em linguagem VHDL da parte V do bubble sort totalmente combinacional.

```
architecture IMP of user_logic is
  --USER signal declarations added here, as needed for user logic
  type std_array is array (0 to 3) of std_logic_vector(0 to 31);
  type std_matrix is array (0 to 3, 0 to 4) of std_logic_vector(0 to 3);
  signal vin, vout: std_array; -- vetores de entrada e saida
  signal vpos: std_matrix; -- matriz de posicoes
  --std_array tamno 0 a 3 para receber os 4 valroes de entrada
  --std_matrix tamanho 0 a 3 para ser comaptivel com as posicoes

begin
  --USER logic implementation added here
  --registradores de entrada sao repassados ao vetor incial
  vin(0) <= slv_reg0;
  vin(1) <= slv_reg1;
  vin(2) <= slv_reg2;
  vin(3) <= slv_reg3;
  --posicoes iniciais sao zeradas
  vpos(0,0) <= "00";
  vpos(1,0) <= "00";
  vpos(2,0) <= "00";
  vpos(3,0) <= "00";

  --generate para realizar comparacoes
  gen1: for i in 0 to 3 generate
        gen2: for j in 1 to 4 generate
            vpos(i, j) <= vpos(i, j-1) + "01" when vin(i) > vin(j-1)
                          or (vin(i) = vin(j-1) and i > j-1)
                          else vpos(i, j-1);
                end generate gen2;
  end generate gen1;

  --ultima posicao de cada linha da matriz
  --atribuida ao seu registrador de saida
  slv_reg4 <= "00000000000000000000000000000000" & vpos(0, 4);
  slv_reg5 <= "00000000000000000000000000000000" & vpos(1, 4);
  slv_reg6 <= "00000000000000000000000000000000" & vpos(2, 4);
  slv_reg7 <= "00000000000000000000000000000000" & vpos(3, 4);
end IMP;
```

APÊNDICE F – Código em linguagem VHDL da parte V do bubble sort misto.

```
architecture IMP of user_logic is
  --USER signal declarations added here, as needed for user logic
  type std_array is array (0 to 15) of std_logic_vector(0 to 7);
  signal vin, vmid, vout: std_array;
  --3 vetores usados sao defidos, com tamanho de 8 bits
  --para cada elemento, ou seja 0 a 255.

begin
  --USER logic implementation added here
  --atribuicao dos registradores de entrada para o vin
  vin(0) <= slv_reg0(24 to 31);
  vin(1) <= slv_reg1(24 to 31);
  vin(2) <= slv_reg2(24 to 31);
  vin(3) <= slv_reg3(24 to 31);
  vin(4) <= slv_reg4(24 to 31);
  vin(5) <= slv_reg5(24 to 31);
  vin(6) <= slv_reg6(24 to 31);
  vin(7) <= slv_reg7(24 to 31);
  vin(8) <= slv_reg8(24 to 31);
  vin(9) <= slv_reg9(24 to 31);
  vin(10) <= slv_reg10(24 to 31);
  vin(11) <= slv_reg11(24 to 31);
  vin(12) <= slv_reg12(24 to 31);
  vin(13) <= slv_reg13(24 to 31);
  vin(14) <= slv_reg14(24 to 31);
  vin(15) <= slv_reg15(24 to 31);

  --em ambos os generates o tamanho do laco deve ser N/2 - 1
  --onde N eh o tamanho do vetor a ser ordenado
  --repassa o vin para o vmid realziando a primeira
  --parte da ordenacao
  GenVmid: for j in 0 to 7 generate
        vmid(2*j) <= vin(2*j) when vin(2*j) <= vin(2*j+1) else
                     vin(2*j+1);
        vmid(2*j+1) <= vin(2*j) when vin(2*j) > vin(2*j+1) else
                       vin(2*j+1);
  end generate;

  --repessa vmid para vout na segunda ordenacao
  GenVout: for k in 1 to 7 generate
    vout(0) <= vmid(0);
        vout(2*k-1) <= vmid(2*k-1) when vmid(2*k-1) <= vmid(2*k) else
                       vmid(2*k);
```

```
44          vout(2*k) <= vmid(2*k-1) when vmid(2*k-1) > vmid(2*k) else
45                       vmid(2*k);
46          vout(15) <= vmid(15);
47    end generate;
48
49    --atribuicao de vout para os registradores de saida
50    slv_reg16 <= "00000000000000000000000000" & vout(0);
51    slv_reg17 <= "00000000000000000000000000" & vout(1);
52    slv_reg18 <= "00000000000000000000000000" & vout(2);
53    slv_reg19 <= "00000000000000000000000000" & vout(3);
54    slv_reg20 <= "00000000000000000000000000" & vout(4);
55    slv_reg21 <= "00000000000000000000000000" & vout(5);
56    slv_reg22 <= "00000000000000000000000000" & vout(6);
57    slv_reg23 <= "00000000000000000000000000" & vout(7);
58    slv_reg24 <= "00000000000000000000000000" & vout(8);
59    slv_reg25 <= "00000000000000000000000000" & vout(9);
60    slv_reg26 <= "00000000000000000000000000" & vout(10);
61    slv_reg27 <= "00000000000000000000000000" & vout(11);
62    slv_reg28 <= "00000000000000000000000000" & vout(12);
63    slv_reg29 <= "00000000000000000000000000" & vout(13);
64    slv_reg30 <= "00000000000000000000000000" & vout(14);
65    slv_reg31 <= "00000000000000000000000000" & vout(15);
66
67  end IMP;
```

APÊNDICE G – Código em linguagem VHDL da parte V do Z buffer.

```
1 architecture IMP of user_logic is
2   --USER signal declarations added here, as needed for user logic
3   type std_vector is array (0 to 63) of std_logic_vector(0 to 7);
4   signal Mdepth: std_vector; --matriz profundidade
5   signal Prof: std_logic_vector(0 to 7); --valor de prof
6   signal Pos: integer range 0 to 63; --posicao
7   --Matriz profundidade 8x8 represntada em forma de vetor de 64 pos.
8
9 begin
10  --USER logic implementation added here
11  --posicao convertida para inteira, ela esta contida entre
12  --os bits 17 e 24 do registrador de entrada.
13  Pos  <= conv_integer(slv_reg0(16 to 23));
14  --profundidade atribuida pelos bits 24 a 31
15  --do registrador de entrada
16  Prof <= slv_reg0(24 to 31);
17
18  -- matriz profundiade recebe valor da comparacao de seu valor
19  --atual com o recebido
20  Mdepth(Pos) <= Prof when Prof > Mdepth(Pos) else Mdepth(Pos);
21
22  --resultado da comparacao enviado ao registrador de saida
23  slv_reg1 <= "00000000000000000000000" & Mdepth(Pos);
24
25 end IMP;
```

APÊNDICE H – Código em linguagem C do sistema F+V para o detector de números primos.

```c
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#include "core_primos.h"
#define NUMTESTE 37813

int teste_primo_vhdl(int x){
    CORE_PRIMOS_mWriteReg(XPAR_CORE_PRIMOS_0_BASEADDR,
                          CORE_PRIMOS_SLV_REG0_OFFSET, x);
    return CORE_PRIMOS_mReadReg(XPAR_CORE_PRIMOS_0_BASEADDR,
                                CORE_PRIMOS_SLV_REG1_OFFSET);
}

int main(){
    //inicio da inicializacao do timer
    XTmrCtr xps_timer_0;
    XTmrCtr* timer_0 = &xps_timer_0;
    Xuint32 BeginTime, EndTime, Calibration, TimeRun;
    int status;
    status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
    xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
    XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    Calibration = EndTime - BeginTime;
    //fim da inicializacao do timer

    int b, i;
    BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    b = teste_primo_vhdl(NUMTESTE);
    EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    if(b==1){
        xil_printf("%d -> %d\r\n", NUMTESTE, b);
    }
    if(b==0){
        xil_printf("%d -> %d\r\n", NUMTESTE, b);
    }

    TimeRun = (EndTime - BeginTime - Calibration);
    xil_printf("Tempo para exec. na FPGA: %d us\r\n", TimeRun/50);
```

```
43      //base de tempo 1 clock = 20ns, ou seja 50 clks = 1 us
44
45      return 0;
46  }
```

APÊNDICE I – Código em linguagem C do sistema F+V para o bubble sort totalmente combinacional.

```
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#include "core_sort.h"
#define SIZE 4

int teste_sort_vhdl(int *vin, int *vout){
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG0_OFFSET, vin[0]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG1_OFFSET, vin[1]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG2_OFFSET, vin[2]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG3_OFFSET, vin[3]);
    vout[CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG4_OFFSET)] = vin[0];
    vout[CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG5_OFFSET)] = vin[1];
    vout[CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG6_OFFSET)] = vin[2];
    vout[CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG7_OFFSET)] = vin[3];
    return 1;
}

int main(){
    //inicio da inicializacao do timer
    XTmrCtr xps_timer_0;
    XTmrCtr* timer_0 = &xps_timer_0;
    Xuint32 BeginTime, EndTime, Calibration, TimeRun;
    int status;
    status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
    xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
    XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);

    BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
    Calibration = EndTime - BeginTime;
    //fim da inicializacao do timer

    //inicializacao de variaveis
```

```
43      int i,vet_vhdl[SIZE], vet_vhdl2[SIZE], b;
44
45      //preenchimento do vetor em ordem decrescente
46      for(i=0; i<SIZE ; i++){
47          vet_vhdl[i]=SIZE-i;
48      }
49      xil_printf("\r\nVetor de entrada em ordem decrescente\r\n");
50
51      //execucao do sistema F
52      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
53      b = teste_sort_vhdl(vet_vhdl, vet_vhdl2);
54      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
55
56      //print do vetor de saida
57      for(i=0; i<SIZE; i++){
58          xil_printf("vet_vhdl[%d] = %d\r\n", i, vet_vhdl2[i]);
59      }
60
61      TimeRun = (EndTime - BeginTime - Calibration);
62      xil_printf("Tempo de exec. na FPGA: %d us\r\n\n", TimeRun/50);
63
64      xil_printf("\r\nVetor de entrada em ordem crescente\r\n");
65
66      for(i=0; i<SIZE ; i++){
67          vet_vhdl[i]=i+1;
68      }
69
70      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
71      b = teste_sort_vhdl(vet_vhdl, vet_vhdl2);
72      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
73
74      for(i=0; i<SIZE; i++){
75          xil_printf("vet_vhdl[%d] = %d\r\n", i, vet_vhdl2[i]);
76      }
77
78      TimeRun = (EndTime - BeginTime - Calibration);
79      xil_printf("Tempo de exec. na FPGA: %d us\r\n\n", TimeRun/50);
80
81      return 0;
82  }
```

APÊNDICE J – Código em linguagem C do sistema F+V para o bubble sort misto.

```
#include <stdio.h>
#include "xparameters.h"
#include "xtmrctr.h"
#include "core_sort.h"
#define SIZE 4

int teste_sort_vhdl(int *vin, int *vout){
    int i;
    for(i = 0; i<(SIZE/2)-1; i++){
        CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG0_OFFSET, vin[0]);
        CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG1_OFFSET, vin[1]);
        CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG2_OFFSET, vin[2]);
        CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                            CORE_SORT_SLV_REG3_OFFSET, vin[3]);

        vin[0] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                    CORE_SORT_SLV_REG4_OFFSET);
        vin[1] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                    CORE_SORT_SLV_REG5_OFFSET);
        vin[2] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                    CORE_SORT_SLV_REG6_OFFSET);
        vin[3] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                    CORE_SORT_SLV_REG7_OFFSET);

        }
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG0_OFFSET, vin[0]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG1_OFFSET, vin[1]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG2_OFFSET, vin[2]);
    CORE_SORT_mWriteReg(XPAR_CORE_SORT_0_BASEADDR,
                        CORE_SORT_SLV_REG3_OFFSET, vin[3]);

    vout[0] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                 CORE_SORT_SLV_REG4_OFFSET);
    vout[1] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
                                 CORE_SORT_SLV_REG5_OFFSET);
    vout[2] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
```

```
43                                      CORE_SORT_SLV_REG6_OFFSET);
44      vout[3] = CORE_SORT_mReadReg(XPAR_CORE_SORT_0_BASEADDR,
45                                      CORE_SORT_SLV_REG7_OFFSET);
46
47      return 1;
48 }
49
50 int main(){
51      //inicio da inicializacao do timer
52      XTmrCtr xps_timer_0;
53      XTmrCtr* timer_0 = &xps_timer_0;
54      Xuint32 BeginTime, EndTime, Calibration, TimeRun;
55      int status;
56      status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
57      xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
58      XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
59
60      BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
61      EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
62      Calibration = EndTime - BeginTime;
63      //fim da inicializacao do timer
64
65      //inicializacao de variaveis
66      int i,vet_vhdl[SIZE], vet_vhdl2[SIZE], b;
67
68      //preenchimento do vetor em ordem decrescente
69      for(i=0; i<SIZE ; i++){
70          vet_vhdl[i]=SIZE-i;
71      }
72      xil_printf("\r\nVetor de entrada em ordem decrescente\r\n");
73
74      //execucao do sistema F
75      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
76      b = teste_sort_vhdl(vet_vhdl, vet_vhdl2);
77      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
78
79      //print do vetor de saida
80      for(i=0; i<SIZE; i++){
81          xil_printf("vet_vhdl[%d] = %d\r\n", i, vet_vhdl2[i]);
82      }
83
84      TimeRun = (EndTime - BeginTime - Calibration);
85      xil_printf("Tempo de exec. na FPGA: %d us\r\n\n", TimeRun/50);
86
87      xil_printf("\r\nVetor de entrada em ordem crescente\r\n");
88
89      for(i=0; i<SIZE ; i++){
```

```
90          vet_vhdl[i]=i+1;
91      }
92
93      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
94      b = teste_sort_vhdl(vet_vhdl, vet_vhdl2);
95      EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
96
97      for(i=0; i<SIZE; i++){
98          xil_printf("vet_vhdl[%d] = %d\r\n", i, vet_vhdl2[i]);
99      }
100
101     TimeRun = (EndTime - BeginTime - Calibration);
102     xil_printf("Tempo de exec. na FPGA: %d us\r\n\n", TimeRun/50);
103
104     return 0;
105 }
```

APÊNDICE K – Código em linguagem C do sistema F+V para o Z buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include "xparameters.h"
#include "xtmrctr.h"
#include "core_zbuffer.h"
#define Msize 8
#define numPolig 512

typedef struct{
    int x;
    int y;
    int prof;
    int cor;
} ListPolig;

ListPolig poligono[numPolig];
int Mdepth[Msize][Msize], Mcolor[Msize][Msize], VetPol[numPolig];

void GeneratePoligPadrao(int prof, int cor, int indice){
    int i, j, k=indice;
    for(i=0; i<Msize; i++){
        for(j=0; j<Msize; j++){
            poligono[k].x = i;
            poligono[k].y = j;
            poligono[k].prof = prof;
            poligono[k].cor = cor;
            VetPol[k] = ((Msize*poligono[k].x+poligono[k].y) << 8)
                        + poligono[k].prof;
            k++;
        }
    }
}

void GeneratePoligAleatorio(){
    int i;
    srand(10);
    for(i = 0; i < numPolig; i++){
        poligono[i].x = rand()%Msize;
        poligono[i].y = rand()%Msize;
        poligono[i].prof = rand()%256;
        poligono[i].cor = rand()%256;
        VetPol[i] = ((Msize*poligono[i].x+poligono[i].y) << 8)
                    + poligono[i].prof;
```

```
44      }
45  }
46
47  void ZeraMatriz(){
48      int i, j;
49      for(i=0; i<Msize ; i++){
50          for(j=0; j<Msize; j++){
51              Mdepth[i][j] = 0;
52              Mcolor[i][j] = 0;
53          }
54      }
55  }
56
57  int main(){
58      //inicio da inicializacao do timer
59      XTmrCtr xps_timer_0;
60      XTmrCtr* timer_0 = &xps_timer_0;
61      Xuint32 BeginTime, EndTime, Calibration, TimeRun;
62      int status;
63      status = XTmrCtr_Initialize(&xps_timer_0,XPAR_XPS_TIMER_0_DEVICE_ID);
64      xil_printf("status do timer (0 iniciou certo): %d\r\n", status);
65      XTmrCtr_Start(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
66
67      BeginTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
68      EndTime = XTmrCtr_GetValue(timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
69      Calibration = EndTime - BeginTime;
70      //fim da inicializacao do timer
71
72      int i, j, k;
73
74      ZeraMatriz();
75
76      //GeneratePoligAleatorio();
77      //gera 64 poligonos por funcao
78      GeneratePoligPadrao(40, 40, 0);
79      GeneratePoligPadrao(50, 50, 64);
80      GeneratePoligPadrao(30, 30, 128);
81      GeneratePoligPadrao(60, 60, 192);
82      GeneratePoligPadrao(20, 20, 256);
83      GeneratePoligPadrao(70, 70, 320);
84      GeneratePoligPadrao(10, 10, 384);
85      GeneratePoligPadrao(80, 80, 448);
86
87      BeginTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
88
89      for(k=0; k<numPolig; k++){
90          CORE_ZBUFFER_mWriteReg(XPAR_CORE_ZBUFFER_0_BASEADDR,
```

```
91                                  CORE_ZBUFFER_SLV_REG0_OFFSET, VetPol[k]);
92          Mdepth[poligono[k].x][poligono[k].y] =
93          CORE_ZBUFFER_mReadReg(XPAR_CORE_ZBUFFER_0_BASEADDR,
94                                CORE_ZBUFFER_SLV_REG1_OFFSET);
95          //comentar este if para analise somente da profundidade
96          if(Mdepth[poligono[k].x][poligono[k].y] == poligono[k].prof){
97              Mcolor[poligono[k].x][poligono[k].y] = poligono[k].cor;
98          }
99      }
100
101     EndTime = XTmrCtr_GetValue(&xps_timer_0, XPAR_XPS_TIMER_0_DEVICE_ID);
102
103     xil_printf("Matriz Profundidade VHDL\r\n");
104     for(i=0; i<Msize; i++){
105         for(j=0; j<Msize; j++){
106             xil_printf("%d  ", Mdepth[i][j]);
107         }
108         xil_printf("\r\n");
109     }
110
111     xil_printf("\r\nMatriz Cor VHDL\r\n");
112         for(i=0; i<Msize; i++){
113         for(j=0; j<Msize; j++){
114             xil_printf("%d  ", Mcolor[i][j]);
115         }
116         xil_printf("\r\n");
117     }
118
119     TimeRun = (EndTime - BeginTime - Calibration);
120     xil_printf("Tempo para exec. na FPGA: %d us\r\n\n", TimeRun/50);
121
122     return 0;
123 }
```