DANILO DAMIÃO DE ALMEIDA

EXPLORANDO O USO DE ARQUITETURAS HETEROGÊNEAS CPU-FPGA NA DEPURAÇÃO DE SOFTWARE E HARDWARE

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da computação, para obtenção do título de *Magister Scientiae*.

Orientador: José Augusto Miranda Nacif

VIÇOSA - MINAS GERAIS
2020

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
A447e
2020

Almeida, Danilo Damião de, 1994-
Explorando o uso de arquiteturas heterogêneas CPU-FPGA na depuração de software e hardware / Danilo Damião de Almeida. – Viçosa, MG, 2020.
60 f. : il. (algumas color.) ; 29 cm.

Inclui apêndice.
Orientador: José Augusto Miranda Nacif.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.54-59.

1. Arquitetura de computador. 2. Arranjos de lógica programável em campo. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.22

DANILO DAMIÃO DE ALMEIDA

EXPLORANDO O USO DE ARQUITETURAS HETEROGÊNEAS CPU-FPGA NA DEPURAÇÃO DE SOFTWARE E HARDWARE

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 28 de fevereiro de 2020.

Assentimento:

Danilo Damião de Almeida
Autor

José Augusto Miranda Nacif
Orientador

Dedico este trabalho a todos que um dia acreditaram em mim.

AGRADECIMENTOS

Como disse uma vez Newton: "Se eu vi mais longe, foi por estar sobre ombros de gigantes", sabendo disso, não tenho como não agradecer a todos que tornaram todo esse trabalho possível. Agradeço primeiramente ao professor José Augusto Miranda Nacif, pela orientação, e auxilio em todos os projetos aqui apresentados. Agradeço a todos os professores da Universidade Federal de Viçosa, pelas ótimas disciplinas, que sem sombra de dúvidas me agregaram ainda mais bagagem acadêmica. Agradeço ao programa de pós-graduação em ciência da computação, pela possibilidade de atuar como monitor em disciplinas de programação para alunos. Agradeço a CAPES pelo apoio financeiro que possibilitou a dedicação exclusiva para realização deste trabalho. Agradeço a minha namorada Aline, por sempre estar ao meu lado e me apoiando em todas as decisões, se hoje consegui chegar até aqui, devo boa parte disso a ela. Agradeço a todos os amigos que fiz nessa jornada, tanto em Viçosa quanto em Florestal, sem o apoio e a parceria de cada um de vocês essa jornada teria sido bem mais complicada. Agradeço a Patrícia, pelos puxões de orelha necessários. Agradeço a *DTI Digital Crafters* pelo apoio e por todo conhecimento que me proporcionou nesses últimos meses, em especial a todo o time SPX, que me acolheu de uma maneira muito especial, a qual eu não tenho palavras de como agradecer. Agradeço a minha família por estar ao meu lado nos momentos bons e ruins.

"O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001".

Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes.
(Marthin Luther King)

RESUMO

ALMEIDA, Danilo Damião de, M.Sc., Universidade Federal de Viçosa, fevereiro de 2020. **Explorando o uso de arquiteturas heterogêneas CPU-FPGA na depuração de software e Hardware**. Orientador: José Augusto Miranda Nacif.

A crescente demanda por sistemas computacionais mais eficientes em termos de consumo energético e poder de processamento vem fazendo com que pesquisadores e empresas de todo mundo invistam cada vez mais em novas arquiteturas e circuitos integrados mais eficientes. Um desses modelos arquiteturais, são as arquiteturas heterogêneas compostas por CPUs e *FPGAs* (*Field-Programmable Gate Array*). Os *FPGAs* possuem uma grande utilidade na criação de diversas aplicações, devido ao seu baixo consumo energético e grande versatilidade. Com base em todo potencial dos dispositivos *FPGA* e o seu uso na computação, apresentamos, neste trabalho, aplicações de uso destas arquiteturas na depuração de software e validação de circuitos digitais. Com isso, dois trabalhos foram desenvolvidos onde em cada um deles foi criada uma ferramenta capaz de mostrar o potencial destes dispositivos, em arquiteturas heterogêneas. No trabalho 2 foi desenvolvido uma ferramenta voltada a verificação de circuitos digitais, capaz de aproveitar o endereçamento de memória compartilhado entre CPU e FPGA para armazenar os dados de uma depuração. No trabalho 3 foi criado um arcabouço capaz de auxiliar desenvolvedores de aplicações paralelas a detectar possíveis condições de corrida utilizando segmentos de memória compartilhados entre *threads* via FPGA e seus canais de alta velocidade com a memória principal.

Palavras-chave: Validação pré-silício. Arquiteturas Heterogêneas. Validação de Software. CPU. FPGA. Microeletrônica

ABSTRACT

ALMEIDA, Danilo Damião de, M.Sc., Universidade Federal de Viçosa, February, 2020. **Exploring the use of heterogeneous architectures CPU-FPGA in software and hardware validation**. Advisor: José Augusto Miranda Nacif.

The growing demand for computer systems more efficient in terms of power consumption, processing, and size. researchers and companies around the world invest a big effort to reach powerful integrated circuits and new architectures. One of this architectural models is the CPU-FPGA architecture. The FPGA *(Field-Programmable Gate Array)* have a bunch of utilities in accelerators scenarios, due your possiblity of reconfigure and emulate other digital circuits, and your low power consumption. Unfortunately, your use for accelerators scenarios is pretty low due factors like: The difficulty of develop an accelerator and the difficulty to test an accelerator. Knowing all potetial of the FPGA and your growing in different computer scenarios, we present in this dissertation, applications using the *FPGA* architecture to develop accelerators, and build architectured to debug digital circuits/software. To do all described in this resume, we make three projects when each project the proposed architecture has proposed to help in a specific problem. In 2 we developed an FPGA overlay aimed to debug other digital circuits. This architecture can explore the interface between FPGA and main memory to decrease the time for collect and store the data about the circuit under analysis. And in 3 we propose a new methodology to help a tester for debug multithreaded algorithms using the *FPGA*.

Keywords: *Pre-silicon validation. Heterogeneous Architectures. Software Test. CPU. FPGA. microelectronics*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

Introdução

A computação tem o seu uso cada vez mais difundido por diversos setores da sociedade. Tal feito, fez com que uma necessidade natural surgisse com o passar do tempo: que é a melhoria em desempenho, e também em consumo energético destes equipamentos. Assim, um grande esforço foi e é feito com esse propósito tanto na parte de hardware quanto de software. Um exemplo disso no quesito hardware, são as técnicas de miniaturização de circuitos. O *VLSI* (*Very Large Scale Integration*) [3] permitiu que os circuitos fossem miniaturizados em escalas nanométricas e encapsulados em pequenas pastilhas de silício. Isso corroborou para que estruturas mais sofisticadas como: pipelines [4], preditores de desvios [5] e múltiplos núcleos de processamento, pudessem ser implementadas sem ocupar uma grande área do circuito final. Entretanto, mesmo com a diminuição dos componentes para escalas nanométricas, um limite de tamanho já está bem próximo de ser alcançado. Com isso, a *Lei de Moore* [6], que na década de 60 previa que a quantidade de transistores em um circuito integrado dobraria a cada 18 meses, vem chegando cada vez mais próxima ao seu limite [7]. Pesquisadores de todo mundo vem trabalhando em outras abordagens para suprir toda essa demanda, que tende a aumentar ainda mais com a popularização das técnicas de análise de dados, [8] e da Aprendizagem de máquina [9].

Desta forma, o surgimento de novos modelos computacionais, como a computação heterogênea, tornou essa demanda factível ao explorar conceitos como o paralelismo a nível de instrução. A computação heterogênea consiste na utilização de circuitos **dedicados** [10] dentro de um modelo computacional. Uma estrutura de hardware dedicado consiste em hardware voltado ao processamento de algum tipo de computação específica. Como exemplos de estruturas dedicadas, existem as **GPUs** (*Graphics Processing Units*) que são capazes de acelerar aplicações com alto nível de paralelismo a nível de instrução, como: processamento gráfico [11] e redes neurais artificiais. Existem também os **ASICs** (*Application Specific Integrated Circuits*), que são circuitos construídos com um propósito específico. Estes são normalmente utilizados em tarefas de processamento massivo, como na mineração de *criptomoedas* [12]. Os **FPGAs** (*Field Programmable Gate Arrays*) também são comumente utilizados na aceleração de aplicações, uma vez que elas podem ser utilizadas como ferramentas de aceleração

de software. Este modelo arquitetural propicia que classes de aplicações que possuem uma alta demanda por processamento possam ser particionadas e executadas de forma mais rápida por estruturas de hardware dedicadas. Um exemplo clássico de arquitetura utilizada por esse tipo de sistema é apresentado na figura 1.1 que consiste em uma *CPU* acoplada a um *FPGA* e uma *GPU* que podem ser utilizados como aceleradores.

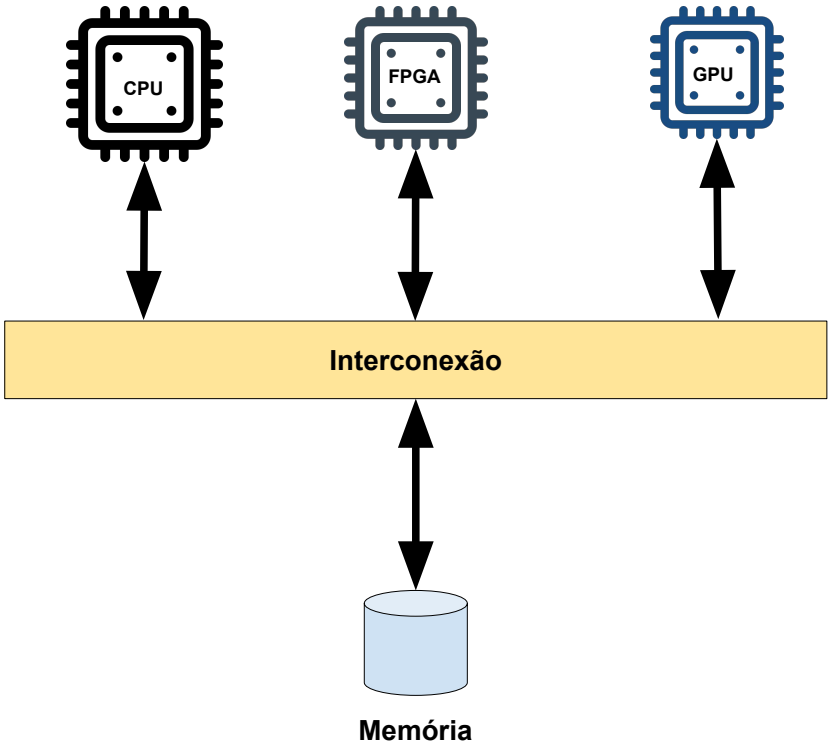


Figura 1.1: Exemplo de arquitetura heterogênea composta por *CPU*, *GPU* e *FPGA*.

Devido a evolução dos circuitos integrados nos últimos anos, a verificação destes dispositivos vem se tornando um desafio crescente. As ferramentas voltadas à simulação e análise formal, já não conseguem explorar a grande quantidade de estados que estes dispositivos alcançam durante a execução de aplicações do mundo real como, exemplo: jogos e aplicações científicas. Com isso, vários grupos de pesquisa investem em metodologias para reduzir o tempo necessário no processo de validação, que em certos casos chega a consumir até metade do ciclo de produção de um circuito integrado [13]. Além de se tratar de uma etapa extremamente cara, em termos de tempo, falhas como a do FDIV bug [14], *Meltdown* e *Spectre* [15], podem causar grandes prejuízos financeiros, caso uma falha crítica passe por todo o processo de construção, chegando ao usuário final. A figura 1.2 apresenta a queda das ações da Intel, em relação à AMD, logo após a exposição das vulnerabilidades *Meltdown* e *Spectre* [1].

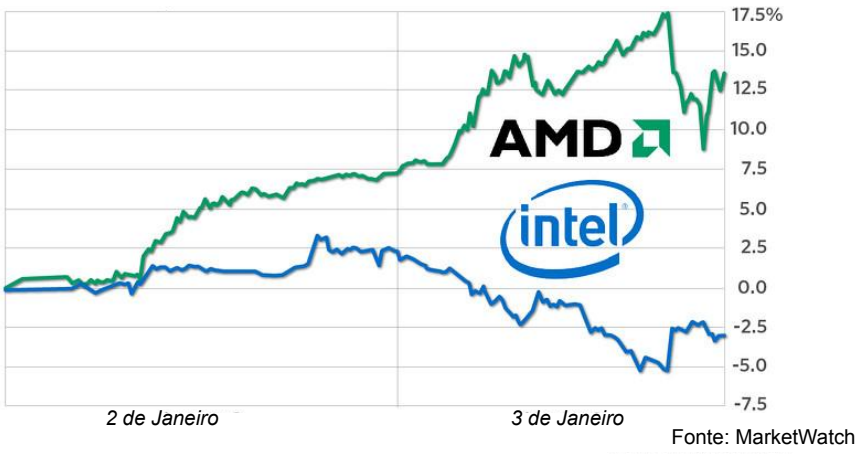


Figura 1.2: Redução no valor das ações da Intel em relação a AMD após exposição das vulnerabilidades *Meltdown* e *Spectre*, Fonte: [1]

Para aumentar o nível de confiabilidade dos circuitos projetados várias técnicas são adotadas no processo de verificação. Estas técnicas podem ser divididas em: pré e pós-silício. Na verificação pré-silício um dispositivo é validado antes de seu projeto ser construído. Sua validação é feita através do uso de aplicações como: simuladores, analisadores formais e emulação [16]. A verificação pré-silício, tem como principal vantagem, a cobertura de sinais completa daquele circuito, o que facilita ao desenvolvedor identificar erros de projeto, através da visualização do sinal, ou através do uso de asserções em ferramentas de verificação formal. Entretanto, apesar de bastante descritiva em termos de informação, a verificação pré-silício possui um baixo desempenho, uma vez que a complexidade dos circuitos tem crescido cada vez mais, e alcançar todos os estados de um dispositivo em análise é inviável em espaço e tempo de execução. A figura 1.3 ilustra a verificação de um circuito digital, através da verificação pré-silício, onde utilizamos uma ferramenta de verificação formal, para validar o funcionamento do dispositivo através do uso de asserções e conjuntos de dados de teste.

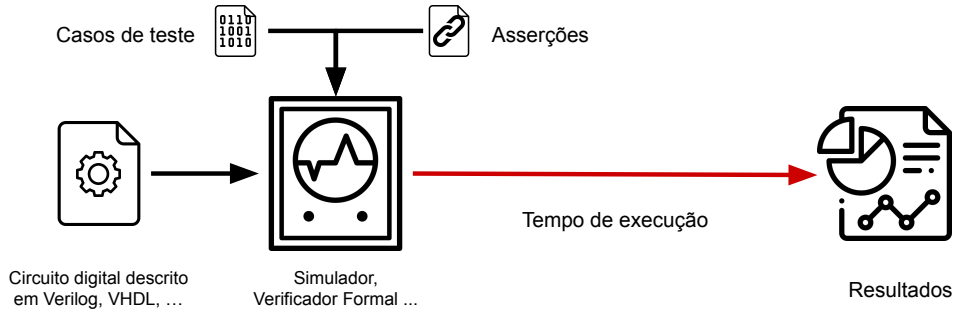


Figura 1.3: Fluxo de verificação pré-silício, baseada em ferramentas de simulação e análise formal.

Na verificação pós-silício, como ilustra a figura 1.4 um dispositivo é verificado logo após a sua fabricação. Esse tipo de verificação, é utilizado na detecção de problemas como: falhas de construção e também erros estruturais [16]. Suas principais vantagens, estão no tempo de execução, que normalmente é muito menor se comparado ao uso de simuladores, e também na possibilidade de se utilizar aplicações do mundo real em seus fluxos de teste. Porém, a verificação pós-silício, apresenta desvantagens como: Baixa observabilidade do circuito sob análise, e janelas de visualização limitadas pelo tipo de metodologia de verificação adotada, como: *Trace-buffers* [17] e asserções [18].

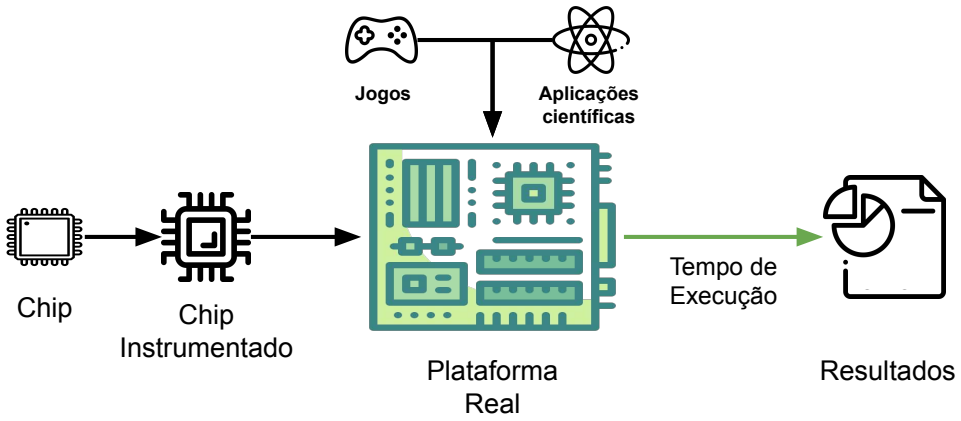


Figura 1.4: Fluxo de verificação pós-silício, baseada no uso real do dispositivo em aplicações do mundo real.

Entretanto, o problema de validação, não se limita apenas ao hardware. Com a evolução dos processadores e das aplicações paralelas, a validação destes sistemas, é um grande desafio para os desenvolvedores. Problemas como: Condição de Corrida [19], Inanição e *Deadlocks* [20] são bastante comuns na computação paralela. Isso ocorre pois, a ocorrência de falsos-positivos na execução deste tipo de aplicação é bastante comum, uma vez que o problema pode ou não ser mascarado pela ordem que as *threads* serão executadas pela CPU. O código 1.5 representa uma aplicação que

pode ou não ser influenciada pelo escalonador de processos, uma vez que devido a inexistência de uma política de acesso ao recurso compartilhado, a variável *sum*, estará sujeita a apresentar resultados corretos e incorretos em diferentes execuções.

```
1  void somatorio(int *somatorio, int vet[],int begin,int end){
2      for (int i=begin; i<end; i++){
3          *sum += vet[i];
4      }
5  }
6  int main(){
7      int sum;
8      int vet[10000];
9      thread t1(&sum,&vet,0,4999);
10     thread t2(&sum,&vet,5000,9999);
11     t1.join();
12     t2.join();
13     cout << sum;
14 }
```

Figura 1.5: Exemplo de somatório paralelo com condição de corrida.

A figura 1.6, por sua vez, ilustra uma condição de corrida entre duas *threads* compartilhando o mesmo recurso. O problema acontece devido a um acesso incorreto da *thread* 1, enquanto ainda era feita uma computação desse valor pela *thread* 2, resultando em uma saída incorreta.

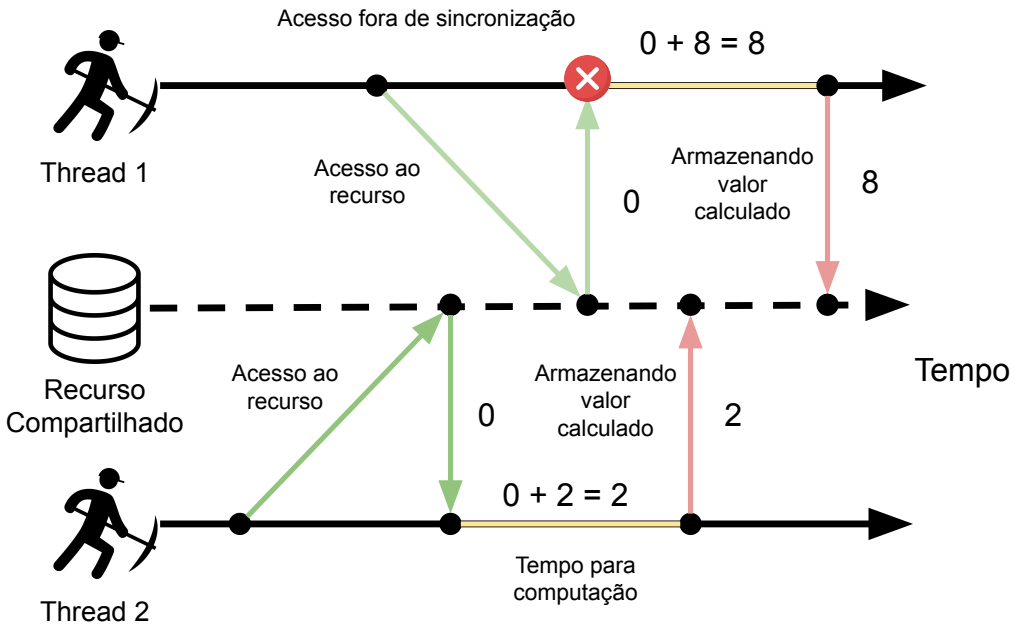


Figura 1.6: Exemplo de condiçao de corrida em aplicação compartilhando mesmo recurso entre duas *threads*.

Apesar de se tratar de um problema que pode ser solucionado através do uso de estruturas como: semáforos, *mutexes* e boas políticas de acesso a recursos compartilhado. A condição de corrida é um problema que pode passar muitas vezes despercebido, com isso um grande esforço é realizado no desenvolvimento de técnicas capazes de identificar quando esse tipo de problema ocorre. No geral, estas técnicas são divididas entre análise estática e dinâmica de código. Na análise estática, o código alvo é analisado, e dele são extraídas informações capaz de identificar possíveis condições de corrida. Já na análise dinâmica, o código é instrumentado com *footprints*, capazes de determinar se ocorreu ou não uma condição de corrida dentro daquele código.

1.1 Problema

Neste trabalho serão abordados dois problemas no contexto de validação de sistemas. O primeiro problema, está relacionado a dificuldade na validação de circuitos digitais na etapa de verificação pós-silício. Nele iremos falar sobre a dificuldade do fluxo de validação de circuitos digitais, e como a utilização de circuitos FPGA fortemente acopladas a CPU pode ser positiva, através da utilização de seus canais de alta velocidade para armazenamento do grande volume de dados gerados neste tipo de emulação.

O segundo problema por sua vez, será a identificação de possíveis falhas de sincronização em aplicações paralelas, através do uso de circuitos FPGA como módulos de co-debugging. Essa abordagem tem como objetivo, reduzir o tempo de execução de ferramentas de análise dinâmica de código, que é possui um *overhead* de tempo.

1.2 Objetivo

O objetivo principal deste trabalho, consiste em explorar o uso de uma arquitetura heterogênea, composta de um CPU e FPGA altamente acoplados, no desenvolvimento de sistemas voltados tanto a verificação de software quanto hardware. Sua realização se deu por meio do desenvolvimento de ferramentas que auxiliem projetistas de software e hardware na identificação de falhas em seus projetos. Assim, foi utilizada uma arquitetura composta por CPU e FPGA, encapsulados dentro de um mesmo chip (*die*). Tal arquitetura possui mecanismos de rápido acesso a memória RAM do sistema, o que a torna uma alternativa no desenvolvimento de aplicações de alto desempenho, explorando tanto os benefícios do paralelismo do FPGA, como também da CPU.

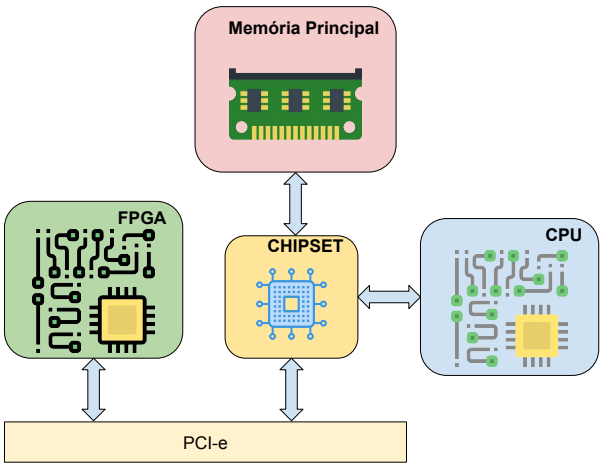
O primeiro objetivo deste trabalho consiste no desenvolvimento de uma camada de abstração capaz de instrumentar circuitos digitais com uma capacidade rápida de reconfiguração. Utilizando os rápidos canais de comunicação entre CPU e FPGA, a camada de abstração tem como funcionalidades: Controle de sinais básicos, e coleta de dados otimizada para utilização da memória RAM do sistema. Dessa forma, o sistema é capaz de controlar o fluxo de execução do dispositivo em tempo real, e ainda reduzir o uso de blocos de memória do FPGA.

O segundo objetivo deste trabalho consiste no desenvolvimento de uma arquitetura capaz de monitorar o acesso a endereços de memória auxiliando na identificação de possíveis condições de corrida em aplicações paralelas. A vantagem explorada neste trabalho está na independência entre CPU e FPGA, que permite ao desenvolvedor monitorar todo o fluxo de alterações sofrido por um recurso compartilhado, afim de se detectar possíveis inconsistências de acesso, devido a falta de políticas de acesso a recurso compartilhado.

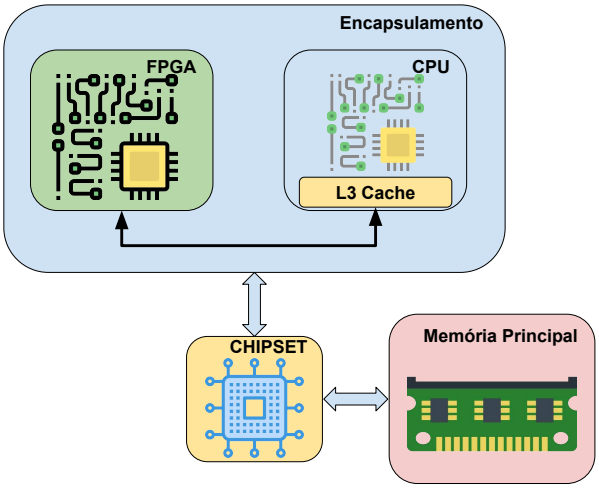
1.3 Contribuição

As contribuições deste trabalho estão no desenvolvimento de duas técnicas capazes de auxiliar desenvolvedores, tanto de aplicações, quanto de circuitos digitais no processo de verificação. Isso é possível, devido ao uso da arquitetura Xeon-FPGA, que diferente de outros modelos já existentes compartilham dentro do mesmo encapsulamento toda a estrutura necessária para uma rápida comunicação entre os dispositivos. As figuras 1.7a e 1.7b ilustram dois tipos de comunicação entre CPU e FPGA. A figura 1.7a ilustra a comunicação feita através do barramento PCI-Express, onde o FPGA pode ser conectado de forma indireta a CPU, e utilizado como plataforma de aceleração. Já a figura 1.7b exemplifica a comunicação feita de forma direta via interface *QPI* [21]. Essa interface fornece uma maior largura de banda para transmissão de dados, entretanto, por se tratar de uma plataforma já acoplada ao processador, não pode ser removida

do sistema, como o modelo da figura 1.7a.



(a) Conexão entre CPU e FPGA via barramento PCI-e.



(b) Conexão entre CPU e FPGA no mesmo encapsulamento.

Figura 1.7: Exemplos de comunicação entre CPU e FPGA.

Explorando a capacidade dos dispositivos FPGA de alto desempenho, em conjunto de processadores voltados a aplicações com alto nível de paralelismo, foi desenvolvida uma ferramenta capaz de extrair informações de circuitos digitais durante a sua emulação em circuitos FPGA. Essa ferramenta é uma alternativa a estruturas de emulação baseadas em FPGA, como as plataformas *Protium X1* e *Protium S1* [22], que apesar de altamente eficientes, possuem um custo elevado.

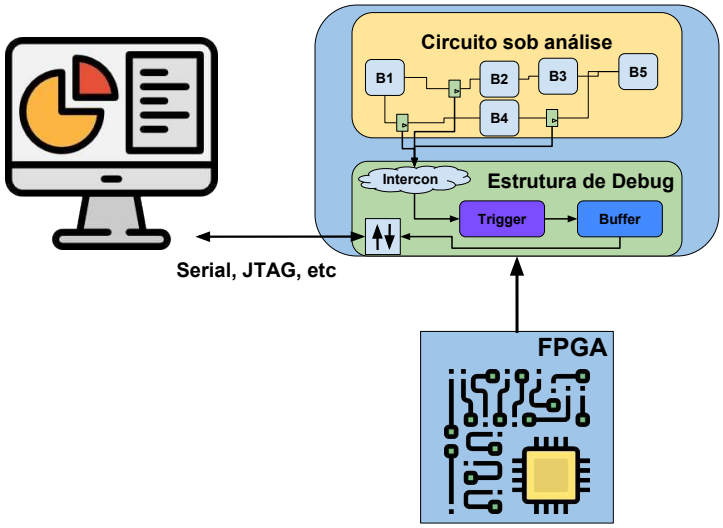


Figura 1.8: Esquema de depuração de circuitos digitais através de circuitos *FPGA*.

Entretanto, o uso de circuitos *FPGA* no processo de verificação ainda é bastante limitado. Uma dessas limitações está relacionada à restrição de espaço do *FPGA*, outra limitação é o tempo necessário para que um circuito digital passe por todas as etapas de síntese, para que enfim seja executado no FPGA. Dependendo da complexidade do circuito, esse tempo pode levar de horas até dias, o que inviabiliza a utilização de circuitos *FPGA*. Com isso, a utilização de camadas de *Overlays* voltadas a depuração de circuitos, podem reduzir drasticamente esse tempo, uma vez que a reconfiguração ocorre de forma mais rápida [23].

Um exemplo de vantagem na utilização de *Overlays* na depuração de circuitos digitais, está ilustrado na figura 1.9. É possível reconfigurar partes específicas de um circuito *FPGA* através do processo chamado: "Reconfiguração Parcial", que possibilita que apenas blocos específicos do dispositivo *FPGA* sejam reconfigurados [24]. Essa abordagem faz com que o tempo de execução no dispositivo seja reduzido, uma vez que não é preciso sintetizar toda a arquitetura, ficando limitada apenas a parte que desejamos verificar.

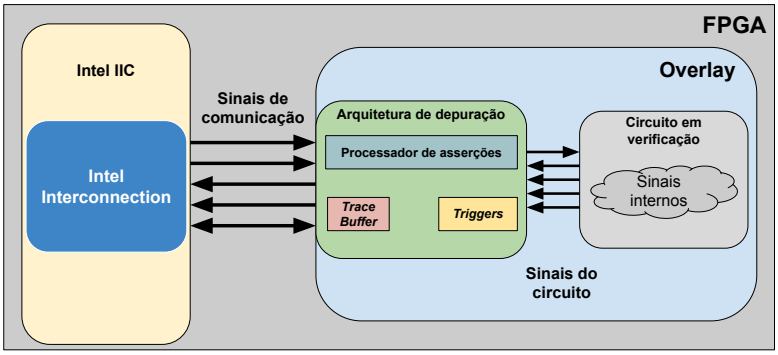


Figura 1.9: Estrutura de depuração baseada em *Overlays* para rápida configuração da metodologia de validação. Adaptado de: [2].

Com as arquiteturas de *FPGA* sendo projetadas dentro do mesmo chip, e interconectadas por um barramento de alta velocidade, como mostra as figuras 1.7a e 1.7b a possibilidade de utilização destas arquiteturas como plataformas para depuração de código é algo bastante positivo. Trabalhos como [25] propõem estruturas acopladas diretamente aos barramentos de comunicação para identificação de problemas como: Condições de corrida, Inanição e *Deadlocks* para aplicações heterogêneas como códigos executados em *GPUs*.

Estudos com essa temática vêm surgindo devido à possibilidade de acesso mais rápido a memória. Um desses trabalhos, é a nossa proposta no capítulo 3 que tem como objetivo utilizar o circuito FPGA de forma semelhante a um analisador lógico, capaz de ler, e armazenar acessos e alterações em endereços de memória compartilhados. A figura 1.10 ilustra o processo de análise da memória de dados de uma aplicação instrumentada com o *framework Hambug*. O processo é dividido em duas etapas, sendo a primeira etapa responsável por instrumentar um determinado endereço da memória alocada. Através desta instrumentação, o analisador é capaz de identificar quando uma determinada variável é lida/alterada e com base nessas alterações construir um mapa de ciclo de vida responsável por identificar possíveis problemas de sincronização quando este recurso é manipulado por duas ou mais *threads*.

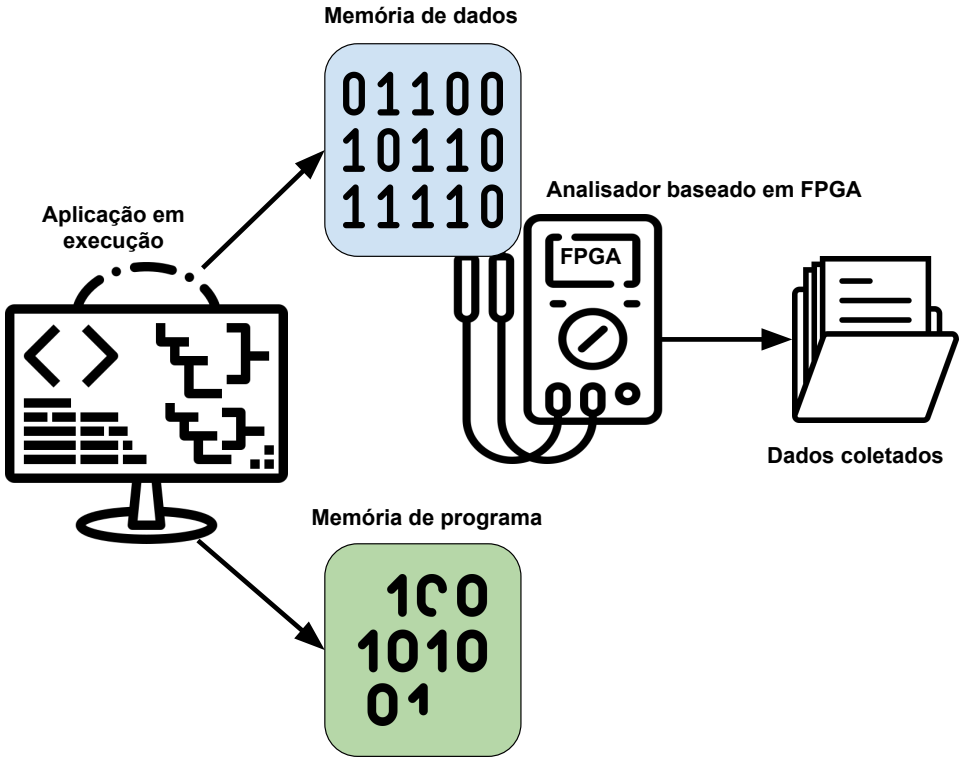


Figura 1.10: Estrutura proposta para coleta de dados de uma determinada aplicação em execução.

O maior diferencial desta abordagem, se deve ao fato de que CPU e FPGA estão completamente desacoplados a nível operacional, apenas compartilhando a interface de comunicação com a memória principal. Através do desacoplamento destas arquiteturas o dispositivo FPGA pode realizar leituras da memória compartilhada sem interferência de fatores como o sistema operacional e o não determinismo na alocação das tarefas. Além disso, por se tratar de uma estrutura de hardware separada, seu fluxo de coleta de dados não influencia na execução do algoritmo em teste.

1.4 Estrutura da dissertação

O corpo desta dissertação está estruturado em conformidade com o formato de coletânea de "artigos científicos" normalizado pelo Conselho Técnico de Pós-Graduação da Universidade Federal de Viçosa [26]. Este formato é composto pelas partes básicas: introdução geral (capítulo 1), artigos científicos (capítulos 2 e 3) e conclusões gerais (capítulo 4).

O capítulo 2 apresenta o artigo *Uma Arquitetura de Overlay para Depuração de Circuitos Digitais em Sistemas Heterogêneos CPU-FPGA*. Neste trabalho, publicado em 2018 no Simpósio Brasileiro de Computação e Engenharia de Sistemas *SBESC* desenvolvemos uma arquitetura de *Overlay*, voltada para a verificação de circuitos digitais, utilizando

o ambiente de aceleração heterogênea Intel HARP (*Heterogeneous Architecture Research Platform*. Esta arquitetura consiste em um processador *Xeon*, e um circuito *FPGA* altamente acoplados pela interface *QPI QuickPath Interconnect* da Intel [27].

Essa arquitetura, diferente dos outros modelos, é capaz de processar e copiar grandes volumes de dados de forma direta entre o *FPGA* e a memória principal do computador. Este tipo de abordagem, diminui de forma significativa o consumo de recursos do *FPGA*, uma vez que podemos utilizar a própria memória RAM do sistema no computador onde o *FPGA* está embarcado. É possível também reduzir o tempo necessário para transferência dos dados decorrentes a uma sessão de depuração, uma vez que a transferência de dados é feita de maneira direta.

Com base nisso, foi projetada uma arquitetura de *Overlay*, voltada a verificação de circuitos digitais, utilizando como base o ambiente *HARP*. Essa arquitetura é composta por uma estrutura descrita em *Verilog*, responsável por coletar, armazenar e detectar a ocorrência de problemas durante a geração de estímulos no circuito em análise. Para que isso se tornasse possível, foram utilizados diversas técnicas de depuração. Para coleta dos dados, foi utilizada a técnica de *Trace-Buffers* [28], que consiste em pequenos bancos de memória onde os sinais relacionados ao circuito em análise são armazenados temporariamente até que sejam transferidos a memória principal.

Para identificação de falhas no circuito em análise, foram utilizadas estruturas denominadas asserções. Uma asserção, segundo [29] consiste em uma afirmativa lógica que quando é violada, gera um sinal de erro informando que sua afirmativa não foi respeitada. Em conjunto a técnica de asserções sintetizáveis apresentada em [29], foi utilizada uma estrutura auxiliar denominada processador de asserções. Sua principal responsabilidade consiste em identificar o tipo de asserção violada e reportar a outras partes da camada de verificação e até mesmo incluir essa informação no conjunto de dados exportados ao responsável pela verificação.

Também foi elaborada uma estrutura para geração de estímulos que pode ser controlada diretamente pela *CPU*. Com isso, é possível controlar a arquitetura e todo o seu fluxo de execução diretamente pelo sistema e até mesmo elaborar estratégias mais eficientes para análise de um circuito como depuração em tempo real. Por fim, para validar a ferramenta, um conjunto de circuitos foram selecionados e falhas foram propositalmente injetadas em suas estruturas de controle, afim de utilizar a arquitetura em cenários práticos. Também foi feito um estudo acerca do impacto na transferência de dados entre *FPGA* e memória principal gerado pela diminuição nos blocos de *BRAM* no *Trace-Buffer* utilizado em sua infraestrutura interna. Por fim, foi analisado o impacto gerado pela utilização da arquitetura de depuração no *clock* máximo devido a possibilidade de *Slacks* decorrente ao crescimento no número de elementos utilizados e a necessidade de se "externalizar"sinais internos de um circuito.

O capitulo 3 apresenta o artigo *HAMBug: Using Heterogenous CPU-FPGA Architectures to Detect Race Conditions in Multithread Systems* que consiste na implementação de uma estrutura de hardware que possibilita a coleta dos dados de uma aplicação *multithread* diretamente da memória. O objetivo deste trabalho, consiste em mostrar como é possível utilizar essa proximidade do *FPGA* com a *CPU* também em contextos relacionados a verificação de sistemas. Pois, devido a crescente utilização de arquiteturas de processadores contendo múltiplos núcleos, problemas como condições de corrida, sincronização de *threads* e *deadlocks* se tornaram bastante comuns e difíceis de se detectar até mesmo em sistemas operacionais modernos.

Neste trabalho apresentamos uma arquitetura capaz de obter informações de uma determinada região de memória especificada pelo programador. Essa região de memória pode tanto ser a seção crítica de uma aplicação, ou a estrutura de sincronização utilizada pelo desenvolvedor. Dessa forma, é possível obter todo o ciclo de vida de uma variável, ou objeto, de forma que o programador consiga traçar e identificar possíveis condições de corrida em uma aplicação de forma totalmente desacoplada ao sistema operacional.

Para que isso fosse possível, foi implementada uma arquitetura composta por 4 camadas. Na primeira camada realizamos a solicitação dos dados de forma sequencial no endereço de memória compartilhado entre CPU e FPGA. Esse processo de leitura é feito a medida em que a interface de comunicação proprietária da *Intel* disponibiliza o canal de leitura da memória compartilhada. Logo em seguida, as camadas subsequentes são responsáveis por identificar a variação neste bloco de dados lido da memória principal e pelo processo de salvamento desses dados para análise posterior.

O conjunto de dados gerado pela arquitetura é composto por um bloco de 64 bytes. Esse bloco de 64 bytes é subdividido em: intervalo de tempo onde ocorreu uma mudança, informação anterior, informação modificada. Com esse conjunto de dados um testador pode identificar eventualidades na manipulação de informações em aplicações *multithreads*. Entretanto, algumas limitações foram identificadas durante o desenvolvimento desta arquitetura. A primeira delas foi a inconsistência nos dados recebidos da *Intel Interface Controller*. Durante boa parte das requisições, que são normalmente realizadas no mesmo endereço de memória, dados inconsistentes eram recebidos mesmo com o canal de comunicação livre para leitura.

Uma solução encontrada para o problema, foi forçar um *stall*, no processo de coleta de dados em um valor de 100 ciclos de *clock*. Outra limitação encontrada no desenvolvimento desta arquitetura, está relacionada a diferença de velocidade entre CPU e *FPGA*. Na arquitetura utilizada, que é composta por um processador *Intel Xeon*, e um *FPGA Arria 10*, temos uma diferença de *clock* significativa, uma vez que o *FPGA* possui um *clock* máximo de 400Mhz e o processador um *clock* de 2.8Ghz. Com isso, boa parte dos dados é "perdida"devido ao fato dos dados passarem por alterações

muito mais rápido do que o dispositivo *FPGA* é capaz de ler.

Para avaliação desta arquitetura, optamos por realizar uma análise a respeito do *Overhead* no tempo de execução quando arquitetura e aplicação *multithread* são executadas. Foram utilizados três algoritmos paralelizáveis, onde cada um deles foi executado com uma instância de tamanho específica. Em cada uma das instâncias, foi medido o tempo de execução, utilizando e não utilizando a arquitetura para coleta dos dados. Como resultados preliminares, foi possível observar que não houve uma mudança significativa no tempo durante ambas as execuções. Isso significa que a proposta apresenta certa viabilidade, mesmo com suas limitações arquiteturais e de velocidade.

Durante o mestrado, além dos trabalhos onde fui o primeiro autor, foram desenvolvidos alguns trabalhos em conjunto de outros alunos de mestrado. O primeiro deles, foi desenvolvido com o mestrando Lucas Bragança e teve como título *Exploring the Dynamics of Large-Scale Gene Regulatory Networks using Hardware Acceleration on a Heterogeneous CPU-FPGA Platform* [30]. Este trabalho consistiu na elaboração de um acelerador capaz de simular complexos sistemas biológicos, como proteínas, células, e bactérias através da manipulação e execução de expressões *booleanas*. Devido a alta complexidade algorítmica do problema e a computação de granularidade binária, o uso deste acelerador proporcionou um *speedup* significativo se comparado ao uso de processadores de uso geral (*CPU*) e até mesmo em *GPU's*.

O segundo trabalho, que foi desenvolvido em conjunto dos alunos Jerônimo Penha, e Lucas Bragança, de título *ADD - Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho* [31], consistiu na construção de uma ferramenta que permitisse a elaboração de *Data flows*, mapeáveis em circuitos *FPGA*. A ferramenta é uma extensão para o simulador de circuitos *HADES* [32], onde foram adicionados blocos capazes de representar operações lógicas e aritméticas possibilitando a construção de *Data Flows*. Também foi elaborado um subsistema capaz de construir cada um desses *Data flows* em uma representação sintetizável em *FPGA*.

O terceiro trabalho, que foi desenvolvido junto ao aluno Khristopher Kaio, de título: *Cryptography Algorithms in Wearable Communication: An Empirical Analysis* [33] consiste em uma análise de consumo voltada para algoritmos de criptografia quando aplicados em microcontroladores de baixo poder computacional. Uma vez, que com o aumento no uso dos dispositivos vestíveis, garantir uma conexão segura entre emissor e receptor é um grande desafio devido as restrições de processamento e energéticas.

Capítulo 2

Uma Arquitetura de Overlay para Depuração de Circuitos Digitais em Sistemas Heterogêneos CPU-FPGA

Dentre os vários desafios da computação, podemos destacar nos últimos anos a demanda por circuitos integrados com mais desempenho e também eficientes em termos energéticos. O aumento da complexidade dos circuitos integrados implica diretamente na sua complexidade da verificação. Uma técnica de projeto é prototipar os circuitos em FPGAs. Este trabalho apresenta uma nova arquitetura para depuração de circuitos digitais utilizando uma plataforma heterogênea da Intel/Altera de alto desempenho com CPU-FPGA em combinação com uma camada de software denominada OPAE (*Open Programmable Acceleration Engine*) da Intel que simplifica a integração de aceleradores em FPGA. O estudo simplifica ainda mais a interface OPAE, permitindo que o projetista configure a plataforma para detecção de falhas e coleta de dados em tempo real de execução através da memória compartilhada e seu barramento de comunicação de alta velocidade.

2.1 Introdução

Com a evolução da computação nos mais diversos âmbitos da sociedade, tais como: indústria, entretenimento e negócios, uma demanda por circuitos mais velozes e com menor consumo vem se tornando factível. Essa demanda no entanto, é atualmente bem atendida, devido as melhorias no processo de fabricação de circuitos. Contudo, com o aumento no número de funcionalidades dos circuitos integrados, a etapa de verificação está se tornando cada vez mais complexa, o que aumenta consideravelmente o tempo para o lançamento de novas tecnologias no mercado.

O processo de verificação de circuitos integrados é responsável por consumir uma grande fatia do tempo de desenvolvimento. Podendo este, chegar a cerca de 50% do tempo total [34]. Todo esse gasto apenas na etapa de verificação, se deve ao fato de que uma falha, por menor que seja, ao chegar ao usuário final pode resultar em grandes prejuízos. Como foi o caso do FDIV *bug* em 1995 [14], e atualmente as falhas *MeltDown* e *Spectre* [35] que juntas, conseguiram em apenas 10 dias após a

exposição da vulnerabilidade causar um prejuízo estimado em 17 Bilhões de dólares no valor de mercado da Intel. Portanto, o processo de verificação é crucial para garantir uma maior confiabilidade nos circuitos produzidos e reduzir suas possíveis vulnerabilidades.

Uma estratégia de verificação consiste na utilização de FPGAs para emulação dos circuitos digitais. Porém, extrair os dados do FPGA e integrar estes dados em outros softwares também acrescenta complexidade ao projeto. Outro ponto é a escassez de recursos de memória interna do FPGA para armazenar os sinais durante um evento de falha. Recentemente, as gigantes da tecnologia, como a Intel através da compra da Altera, e a Microsoft utilizando FPGAs em suas plataformas na nuvem [36], vêm mostrando que a utilização de FPGAs como aceleradores é promissora. Este trabalho propõe a união das duas estratégias verificação e aceleração com FPGA, tendo como objetivo apresentar o projeto de um *Overlay*, que é uma camada reprogramável em tempo de execução implementada no FPGA. A função é possibilitar a verificação de circuitos utilizando a nova plataforma da Intel com uma CPU Xeon/FPGA através da interface Opae [37] para comunicação do FPGA com a CPU.

A arquitetura proposta tem como objetivo permitir que o projetista implemente o hardware e a instrumentação no FPGA, selecionando os sinais do circuito para verificação e definindo também em quais condições o sistema deve automaticamente informar ao usuário sobre a ocorrência de um evento de erro naquele circuito. Além disso, o sistema inicializa um processo de coleta de informações para documentar e operacionalizar uma eventual busca pela causa do problema.

Uma das vantagens da arquitetura proposta é o modelo de memória compartilhada entre a CPU e FPGA disponibilizado na plataforma de FPGA da Intel. Portanto, é possível reduzir significativamente o uso dos blocos de RAM internos do FPGA, as BRAM, em comparação com outras ferramentas como o *SignalTAP* para verificação [38]. Por fim, esta arquitetura pode ser reconfigurada em tempo de execução, ou seja dinamicamente, pois usa uma abordagem de implementar uma camada virtual no processo de depuração em FPGA, evitando que a cada nova instrumentação seja necessário realizar a síntese do circuito que pode demorar horas para posteriormente regrava-lo e aplicar novos vetores de teste.

Este artigo está organizado na seguinte forma: a seção 2.2 apresenta os principais aspectos da verificação utilizando FPGAs e as vantagens no uso de *Overlays*. A seção 2.3 contextualiza o trabalho proposto em relação a outros que utilizam FPGA para verificação. Na seção 2.4 iremos apresentar a estrutura da arquitetura proposta para coleta de dados em hardware, a seção 2.5 detalha a implementação em software responsável pela comunicação com a FPGA e a ferramenta responsável pela geração parametrizada da arquitetura. Por fim, nas seções 2.6, 2.7 iremos apresentar e discutir os resultados e os trabalhos futuros para melhorias da arquitetura proposta.

2.2 Fundamentação Teórica

A complexidade dos circuitos digitais atuais e o espaço exponencial de estados possíveis, vem tornando o processo de verificação de circuitos digitais uma tarefa cada vez mais custosa em termos de tempo e recursos financeiros. Para evitar a fabricação dos mesmos sem uma verificação detalhada do projeto, o uso de emulação em FPGA é bastante difundido em etapas de depuração [39].

O uso de FPGAs reduz significativamente o custo final da verificação, pois não necessita que o circuito seja construído em silício. E ainda possui uma velocidade de execução próxima ou semelhante a de um circuito físico. Além disso, diferente do que acontece ao nível de silício, que não permite modificar um circuito e selecionar um novo conjunto de sinais, na verificação utilizando FPGA o circuito pode ser re-instrumentado e posteriormente, sintetizado e gravado na FPGA com uma nova configuração, em um tempo bem inferior ao necessário para fabricação de um circuito que implica em uma redução significativa nos custos. A Figura 2.1 ilustra todo o processo de execução de um circuito integrado em uma FPGA, começando pela etapa de instrumentação do circuito e sua alocação na FPGA.

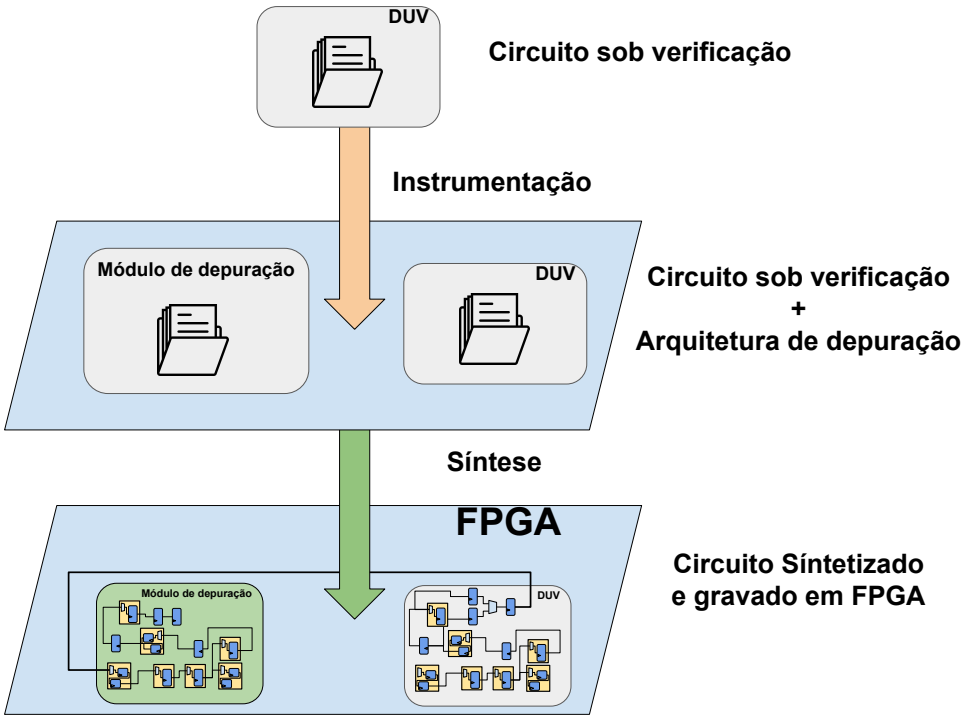


Figura 2.1: Fluxo de síntese de um circuito em FPGA.

Entretanto, a verificação de um circuito é um processo iterativo onde a cada nova iteração um subconjunto de sinais será requisitado pelo projetista. A cada novo passo, é necessário que o projetista por padrão instrumente novamente o circuito e refaça a síntese do circuito, para que dessa forma ele consiga executar seus casos de teste.

Porém, refazer a síntese a cada nova instrumentação irá gerar um grande gargalo de tempo ao processo como um todo, como mostra a figura 2.2, já que um circuito, dependendo de sua complexidade pode ter seu tempo de síntese variado consideravelmente. Sabendo disso, a utilização de arquiteturas de *Overlays* voltadas a depuração estão sendo cada vez mais utilizadas na verificação de circuitos digitais [40].

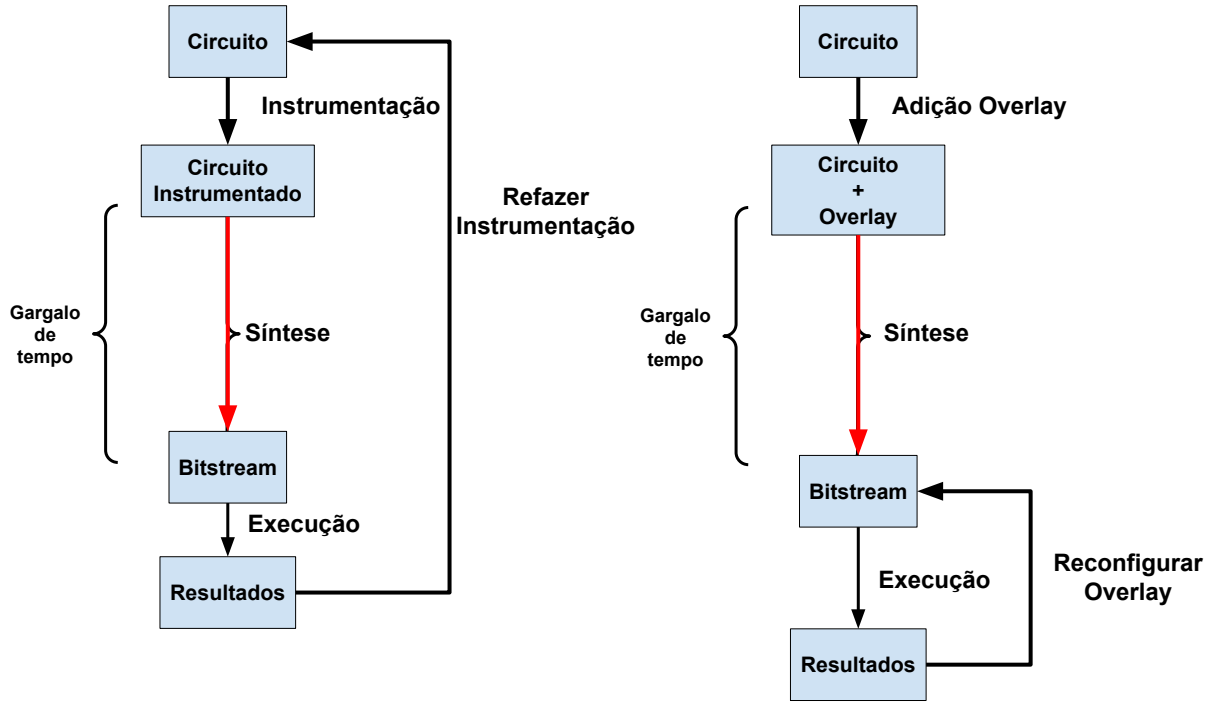


Figura 2.2: Diferenças de verificação em FPGA utilizando Overlays e técnicas gerais.

Estas plataformas reduzem significativamente o tempo de verificação, funcionando como uma camada sobreposta ao FPGA, que podem ser reconfigurada em tempo de execução, após a síntese e programação da configuração do *bitstream* do FPGA. O processo de reconfiguração do *overlay* é significativamente mais rápido se comparado a uma síntese completa do circuito ou até mesmo um processo de reconfiguração parcial. A Figura 2.2 ilustra a verificação nos dois aspectos, onde é mostrado o gargalo de tempo gerado pela síntese completa do circuito, e a redução deste gargalo com a adição do *Overlay* ao design.

2.3 Trabalhos relacionados

O projeto de arquiteturas para verificação de circuitos integrados já é uma área bastante explorada pela literatura. O trabalho [41], propõe um modelo arquitetural para verificação de circuitos gerados através da ferramenta *LegUp* [42], uma ferramenta de síntese de alto nível que permite a elaboração de *dataflows* baseados em código C-ANSI. A arquitetura proposta em [42], permite que o usuário depure o circuito de

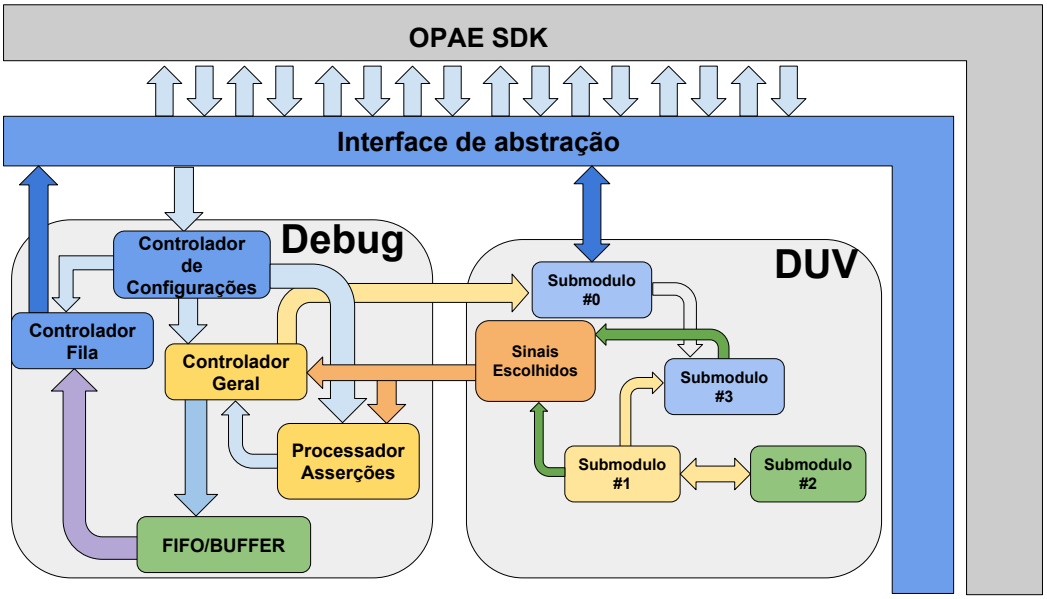


Figura 2.3: Arquitetura de depuração.

forma semelhante ao que é feito no uso de depuradores a nível de software, como o GDB [43]. Porém ,o trabalho é limitado às arquiteturas geradas pela ferramenta *LegUp*, o que restringe o seu uso em outros tipos de circuitos.

O trabalho apresentado em [44] propõe uma ferramenta para verificação de circuitos em ambientes heterogêneos CPU-FPGA, de nome JHDL (*Just-Another Hardware Description Language*), que permite a especificação e projeto de circuitos digitais utilizando a linguagem Java.

Apesar de possuir uma proposta semelhante a desse trabalho, o trabalho não especifica plataformas em FPGA para utilização da ferramenta. Além disso, a ferramenta já não é mais atualizada há cerca de 8 anos. O trabalho apresentado em [45] propõe a utilização de *Overlays* reconfiguráveis no formato de redes de interconexões, que permitem que uma maior variabilidade de sinais possam ser selecionados dentro de uma seção de depuração.

A seleção é realizada apenas reconfigurando a rede de interconexão para um novo subconjunto de sinais. Por fim, o trabalho apresentado em [40] demonstra como a utilização de *Overlays* no processo de verificação de circuitos digitais pode ser vantajosa. Mostrando primeiramente que tanto o tempo de execução quanto de reconfiguração pode ser reduzido através da reconfiguração do *Overlay*.

Além disso, apresenta um conjunto de implementações aplicadas a modelos de verificação de circuitos voltadas para *Overlays*.Diferentemente dos quatro trabalhos citados anteriormente, este artigo tem como objetivo propor uma nova arquitetura de *Overlay* voltada para o cenário de arquiteturas heterogêneas com alto acoplamento através de memória compartilhada entre CPU-FPGA.

A arquitetura proposta aliada ao alto acoplamento entre CPU-FPGA possibilita a

emulação circuitos digitais. Isso nos permite encontrar falhas, e realizar um mapeamento dos resultados de forma mais rápida que o uso tradicional dos FPGAs no domínio de verificação. Algo que muitas vezes apresenta restrições de comunicação, e restrições com relação ao grande volume de informações gerados.

Além disso, as abordagens anteriores podem perder uma parte da janela de tempo durante o passo de cópia dos dados para a memória da CPU. Ademais, devido a necessidade de uma grande quantidade de memória disponível para armazenar os estados do circuito sob verificação, a maior parte do espaço interno do FPGA com seus módulos de memória BRAM é utilizada apenas pela arquitetura de depuração, o que diminui significativamente tanto a velocidade final do circuito **Fmax**, quanto a área útil da placa.

2.4 Arquitetura Proposta

Este trabalho propõe uma nova arquitetura para coleta das informações do circuito sob verificação. A arquitetura foi descrita em *Verilog*. Além da arquitetura em hardware, através de um software que executa na CPU, o projetista é capaz de inicializar, configurar, executar e coletar informações a respeito do circuito em tempo de execução. A solução proposta possibilita também uma vazão de dados da ordem de GB/s, devido a comunicação direta entre o FPGA-CPU através da memória compartilhada. Nesta seção iremos apresentar todos os submódulos que compõem a arquitetura de depuração.

2.4.1 Controlador de configurações

Para realizar a coleta de informações do circuito de forma flexível, a arquitetura projetada precisa é configurada pelo software. Estes parâmetros de configuração são: variáveis de memória compartilhada, limite de espaço alocado, limite máximo de ciclos e controle do processador de asserções. Todas as configurações são recebidas por meio da arquitetura proposta em [46], que funciona como uma camada de abstração sobre a camada da *OPAE* da Intel, fornecendo uma interface simplificada para comunicação e transferência de dados.

A Figura 2.3 ilustra toda a arquitetura do sistema, que é composta pelas camadas de comunicação entre CPU-FPGA que são respectivamente o OPAE e a camada de abstração proposta em [46]. A arquitetura de depuração (Debug) se comunica com o software pela camada de abstração e coleta informações do circuito sob verificação (DUV). O circuito fica no módulo DUV, que recebe sinais de controle do módulo de `debug` e pode ou não receber informações diretas da camada de abstração.

Todas os dados de configuração são recebidos através de uma API que trabalha

acoplada com a arquitetura. Esta abordagem permite com que a CPU controle todo o fluxo de execução da arquitetura de depuração. Todo o procedimento para comunicação entre CPU e FPGA é feito utilizando os registradores disponíveis na plataforma OPAE e na memória compartilhada.

2.4.2 Processador de Asserções

Para que seja possível a identificação de erros do circuito, construímos um módulo denominado processador de asserções. Este módulo é responsável por analisar um subgrupo de sinais, mediante as regras de asserções. Sempre que uma asserção é violada no circuito, o processo de identificação da falha é inicializado no módulo. Ao fim do processo de identificação, a falha é enviada para a memória compartilhada. O funcionamento deste módulo é baseado no trabalho proposto em [29] com asserções sintetizáveis. As asserções deste trabalho, possuem submódulos combinacionais e sequenciais, que permitem que através da técnica de encadeamento de asserções, o módulo gerenciador de asserções descubra exatamente qual foi a asserção violada.

2.4.3 Controlador Geral

O terceiro módulo que compõe esta arquitetura é denominado controlador geral. Este módulo controla todo o fluxo de coleta de informações do circuito alvo. O módulo fornece todos os sinais de sincronização necessários para que o circuito alvo funcione.

Desta forma podemos controlar completamente o funcionamento do circuito alvo e interrompe-lo quando necessário. A Figura 2.4 mostra o conjunto de sinais compartilhados entre a arquitetura de depuração e o circuito alvo.

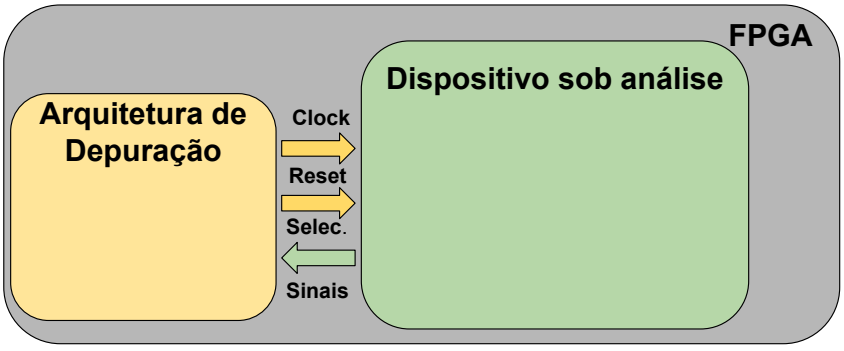


Figura 2.4: Comunicação entre o módulo de controle geral, e o circuito alvo.

O módulo de controle geral também tem como função receber os sinais selecionados pelo projetista e armazenar estes dados em um *buffer* local. Este *buffer* será periodicamente esvaziado para otimizar o uso do canal de comunicação da arquitetura CPU-FPGA na memória compartilhada.

A arquitetura proposta é capaz de armazenar um total de 480 sinais de um circuito alvo. Essa restrição se deve ao tamanho da palavra indexada pela arquitetura que possui um tamanho máximo de 512 bits. Destes 512 bits, 32 são reservados para armazenar o valor do ciclo atual. Assim, o projetista será capaz de saber a qual ciclo de execução aquele conjunto de sinais pertence.

Essa técnica também reduz significativamente o canal de comunicação e também o uso de memória, uma vez que conhecendo o ciclo de ocorrência de um evento não precisamos armazenar o mesmo conjunto de sinais. A Figura 2.5 mostra o esquema de alocação de memória utilizada pela arquitetura. Dentro de um total de 64 bytes, os 4 bytes menos significativos serão utilizadas para armazenar o ciclo onde aquele evento ocorreu, e o restante será utilizado para armazenar cada um dos sinais selecionados pelo projetista.

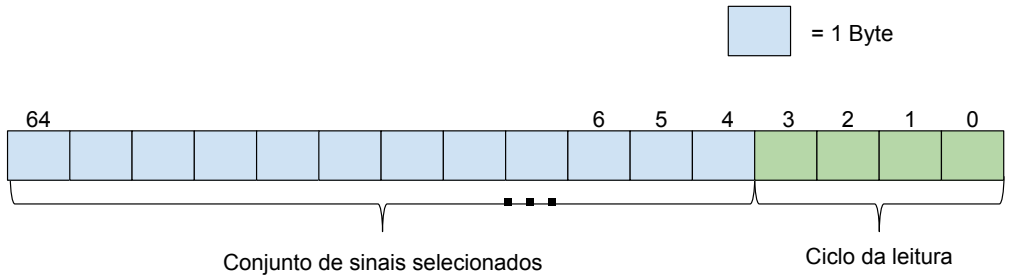


Figura 2.5: Alocação de dados de depuração no *Overlay*.

2.4.4 Controlador Fila

Como mencionado na seção 2.4.3 nem sempre o canal de comunicação estará disponível para uso. Com isso, é necessário controlar o acesso a este canal para evitar que falhas ocorram durante o processo de cópia dos dados. Desta forma, o módulo controlador de fila, tem como responsabilidade controlar o fluxo de escritas na memória compartilhada afim de evitar possíveis conflitos de endereçamento e consequentemente resultados inválidos.

O módulo controlador de fila, possui duas variáveis que serão utilizados para estabelecer uma comunicação com o software. A primeira variável, denominada de status, tem como função informar ao software o estado atual da arquitetura, que podem ser definidos como:

- Em execução: Uma seção de depuração está em execução e nenhuma informação pode ser obtida.
- Aguardando configuração: A arquitetura ainda não foi configurada.
- Asserção falhou: Durante uma seção de depuração uma asserção foi violada (código da asserção disponível no bloco de dados alocado).

- Tempo limite estourado: Uma seção de depuração configurada não foi capaz de identificar nenhuma falha.
- Erro: Alguma eventualidade, como falta de memória compartilhada disponível causou uma interrupção abrupta do sistema.
- Fim da execução: Após o processo de depuração os dados foram armazenados na memória compartilhada.

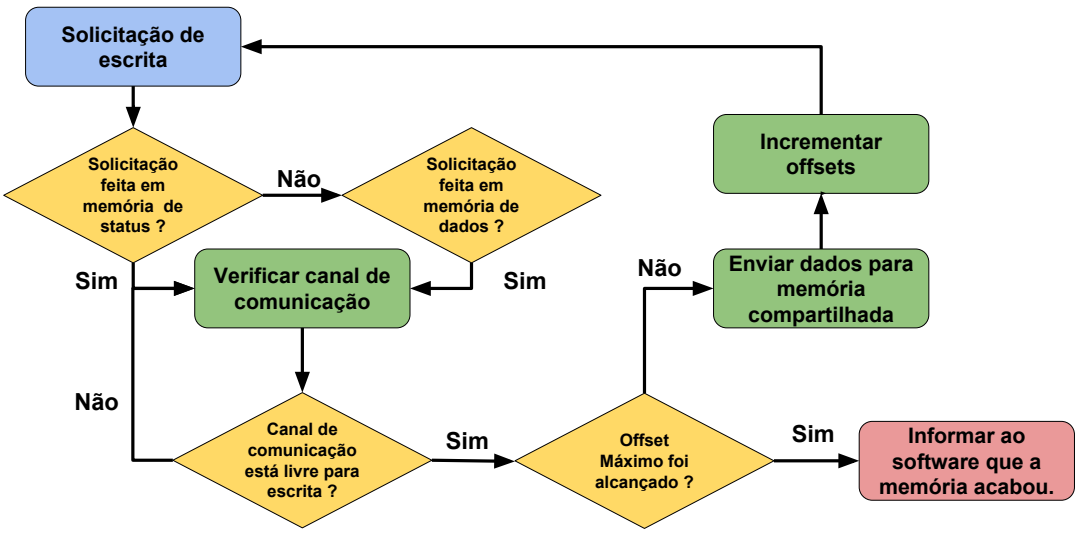


Figura 2.6: Fluxograma de execução para cópia de dados do *buffer* para memória compartilhada.

Já a segunda variável será responsável por endereçar a memória alocada para armazenamento dos dados obtidos do circuito. O uso da memória compartilhada neste caso será de grande auxilio na verificação, uma vez que devido as restrições de memória, uma FPGA é capaz de armazenar apenas pequenas porções de dados no máximo da ordem de alguns Mega Bytes de capacidade, enquanto a memória RAM possui um espaço disponível da ordem de Giga Bytes.

A Figura 2.6 ilustra o funcionamento do módulo utilizado para cópia dos dados na memória compartilhada. A cada requisição de escrita feita pelo controlador geral, uma verificação é feita a respeito do tipo de informação que se deseja armazenar, logo em seguida verificamos se o canal está livre para ser utilizado, caso esteja verificamos se o limite de memória alocado não foi alcançado, em caso de constatação positiva, o módulo deve reportar ao software que já não há espaço livre portanto o processo de depuração será finalizado. Caso contrário, os dados são escritos na memória compartilhada e o *offset* é incrementado.

2.4.5 Buffer Interno

Como nem sempre o canal de comunicação estará disponível para cópia dos dados, foi adicionada a arquitetura de depuração um *buffer* interno capaz de armazenar estas informações durante uma possível indisponibilidade. Seu funcionamento é semelhante a uma fila circular, onde a cada espaço de memória pode ser lido/escrito através de ponteiros que "giram"ao redor do espaço de memória. Seu tamanho é definido com base na necessidade do projetista, que deve levar em consideração o uso do canal de comunicação/*overhead* gerado pelo uso de memória BRAM utilizada. A Figura 2.7 ilustra a estrutura do *buffer* circular utilizado na arquitetura.

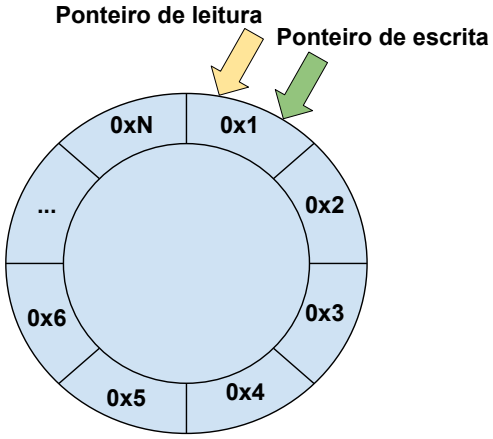


Figura 2.7: Estrutura de buffer circular implementado na arquitetura.

2.5 Interface

Para estabelecer a comunicação entre CPU e FPGA foi implementada uma interface ou API (*Application programming interface*) capaz de configurar e coletar informações resultantes da depuração. Esta API foi desenvolvida na linguagem C/C++ e é uma extensão a API proposta em [46]. Esta extensão tem como objetivo simplificar e definir padrões para o procedimento de configuração, execução e coleta de dados, deixando transparente ao projetista toda parte de comunicação. A Figura 2.8 mostra um trecho de código utilizado para executar uma seção de depuração na arquitetura que pode ser entendido da seguinte forma:

Inicialmente alocamos uma unidade de gerenciamento, assim como é feito em [46]. Logo em seguida executamos o método **runCircuit()**, que executa o circuito sob verificação em um total de 10000 ciclos de *clock*, sendo destes, 100 ciclos dedicados a reinicialização (*reset*) do circuito alvo. Desta forma, serão gerados 10000 ciclos de *clock*. Ao fim do processo, uma cópia dos dados é feita através do método **copyDebugQueue()** que terá como retorno um *buffer* de tamanho **N**, sendo N o número de dados coletados da arquitetura. Assim, o projetista pode verificar o comportamento

do circuito de forma *ad-hoc* ou utilizar técnicas de inteligência computacional, para encontrar possíveis erros de projeto.

```
#include <AccManagement.h>
#define ENTRADA 5004
typedef struct {
    uint8_t buffer[64];
}dados;
int main(){
 auto *accMgr = new AccManagement();
 bool accMask[accMgr->getNumAccelerators()];
 accMgr->startAccelerators(accMask);
 DebugAccelerator *debugAccelerator;
 size_t numBytes = sizeof(dados)*ENTRADA;
 auto *buffer = (dados*) malloc(numBytes);
 for (auto &acc:accMgr->getAccelerators()) {
  debugAccelerator = acc->getDebugAccelerator();
 }
 debugAccelerator->runCircuit(100,10000);
 debugAccelerator->copyDebugQueue(numBytes,buffer);
 delete accMgr;
 return 0;
}
```

Figura 2.8: Código exemplo de execução de uma sessão de depuração.

2.5.1 Ferramenta para geração de código

Também foi implementada em conjunto com a API, uma ferramenta para geração da arquitetura de depuração na linguagem de descrição Verilog/HDL. A ferramenta de geração de código foi construída na linguagem Python com o auxílio do framework *Veriloggen* [47]. A Figura 2.9 ilustra o processo de geração de uma arquitetura de depuração, que é estruturado da seguinte forma: Um arquivo JSON, contendo informações a respeito das asserções a serem utilizadas, o conjunto de sinais selecionados pelo projetista e as regras associadas as asserções é gerado e passado como entrada para o gerador. Em seu núcleo, o gerador de código, através do *Veriloggen* será responsável por gerar toda a arquitetura apresentada na seção 2.4. Por fim, um conjunto de arquivos descritos em *Verilog* será automaticamente criado pronto para uso no projeto em questão.

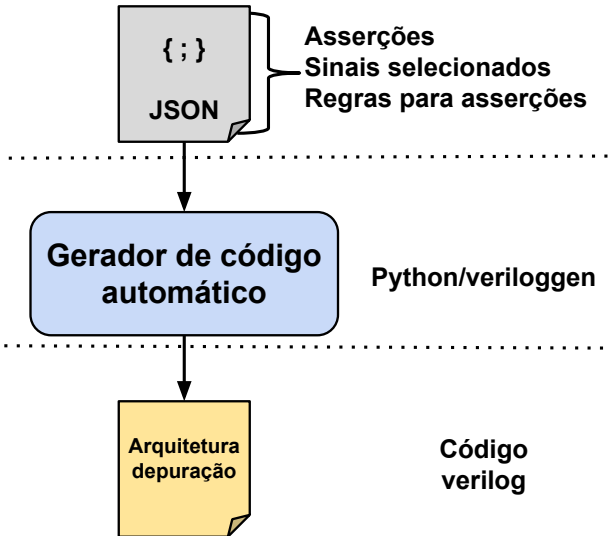


Figura 2.9: Fluxo para geração do código verilog do módulo de depuração.

2.6 Resultados

Para validação das ferramentas propostas foram utilizados dois circuitos digitais: um módulo de encriptação(128/192 AES) e um Processador (*Natalius*). Ambos os circuitos estão disponíveis em [48]. Estes módulos foram selecionados por serem representativos em cenários como aceleradores de criptografia para IOT e processadores de uso geral. Todos os experimentos foram executados na plataforma CPU-FPGA composta por um processador Xeon e uma FPGA Arria 10 fortemente acoplados à memória compartilhada, via QPI [49].

2.6.1 Cenários de utilização

De forma a validar o uso da arquitetura, também foram elaborados dois cenários de erro para ambos os módulos. O modelo de erros utilizado nesta seção foi baseado na abrodagem proposta em [50] que ilustra os tipos de falhas dos grandes projetos de circuitos digitais.

Caso 1

Um erro recorrente, no desenvolvimento de circuitos digitais, é a atribuição incorreta de sinais em um determinado módulo [50]. Este tipo de erro que pode ocorrer se mantivermos um fio desconectado ou conectado a um nível lógico fixo. Além disso, pode propagar um sinal incorreto para módulos com dependência e gerar inconsistência de outros módulos. Neste caso, foi elaborado um modelo de teste onde uma situação de *overflow* é forçada no processador *Nautilus* e sinal *Carry* responsável por informar o ocorrido fixado em 0 (zero). O conjunto de asserções elaborado para determinar

a falha pode ser visto na figura 2.10, que consiste em duas asserções encadeadas, uma conectada ao resultado da ALU, sendo responsável por identificar a ocorrência de **overflow** e uma segunda asserção responsável por verificar se um determinada expressão irá assumir valores diferentes de 0.

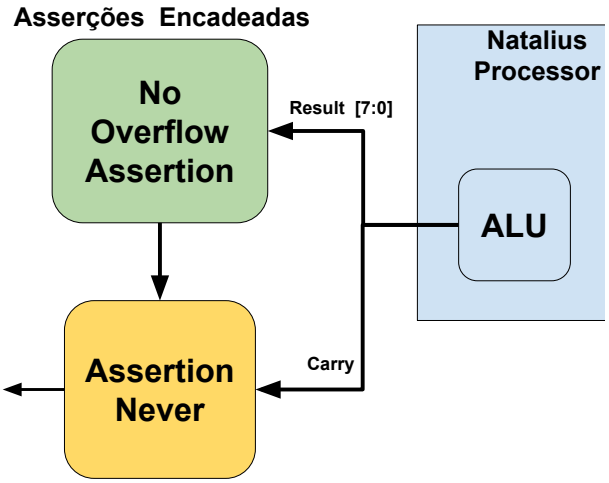


Figura 2.10: Conjunto de asserções utilizados.

A figura 2.11 mostra o momento onde a asserção **No Overflow** é violada. É possível observar o ciclo onde o evento ocorreu (será enviado para o software ao fim da execução) e a inicialização do processo de propagação do sinal **esco**, até que o processador de asserções identifique a asserção que falhou.

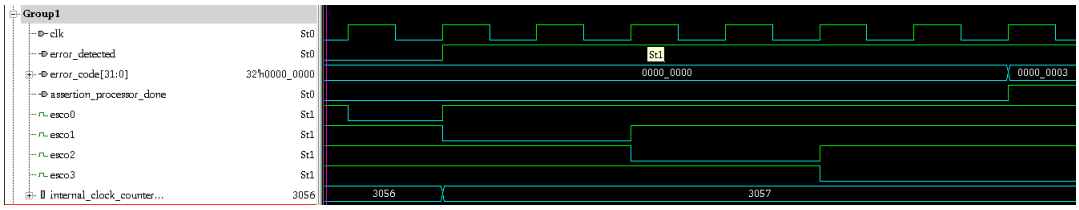


Figura 2.11: Violação da asserção No Overflow no ciclo de clock 3057.

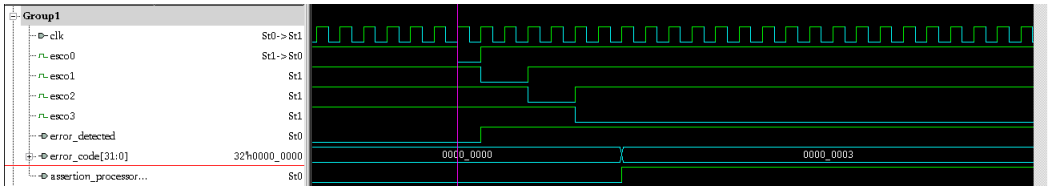


Figura 2.12: Violação da asserção No Overflow no ciclo de clock 485, encriptação foi realizada porém não foi copiada para memória compartilhada.

Caso 2

Um erro recorrente no projeto de circuitos digitais é a mudança incorreta de um estado para outro. Este tipo de falha acontece quando dentro de uma máquina de estados, o fluxo de execução é ligeiramente alterado devido a uma atribuição incorreta no registrador utilizado para controle dos estados. Isso pode ocasionalmente gerar inconsistências de informações, como dados estáticos, falhas de comunicação e consequentemente uma propagação de erro para outros módulos que compõem o circuito. A figura 2.13 ilustra duas máquinas de estados, sendo a máquina de estados *A* correta, capaz de percorrer todos os três estados do circuito e a máquina *B* com um erro de atribuição de estados entre os estados 1 e 2.

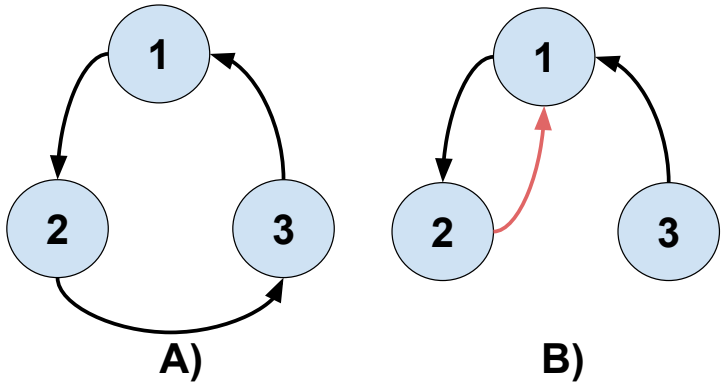


Figura 2.13: Máquinas de estados.

Neste segundo caso, foi elaborado um acelerador capaz de encriptar blocos de 512 bits por vez, utilizando o algoritmo de criptografia AES. Para isso, uma arquitetura composta por quatro módulos de criptografia AES-128 bits e uma controladora foi elaborada e é ilustrada pela figura 2.14. Porém, nesta arquitetura, foi injetada uma falha de transição entre os estados responsáveis por realizar a escrita dos dados na memória compartilhada, fazendo assim com que as informações não sejam copiadas para a memória compartilhada. O processo para identificação foi elaborado da seguinte forma: Uma encriptação de um bloco de informações leva aproximadamente 450 ciclos para ser executado e uma solicitação de escrita leva em média 1 32 ciclos, sabendo disso, o sinal responsável por solicitar uma escrita deve ser habilitado em no máximo 32 ciclos após a encriptação dos dados, caso contrário os dados não estarão sendo copiados para a memória compartilhada. A figura 2.12 mostra o exato momento onde a asserção **Never** é violada, indicando que foi passado um total de 482 ciclos e o sinal responsável por realizar uma solicitação de escrita na memória compartilhada não foi habilitado.

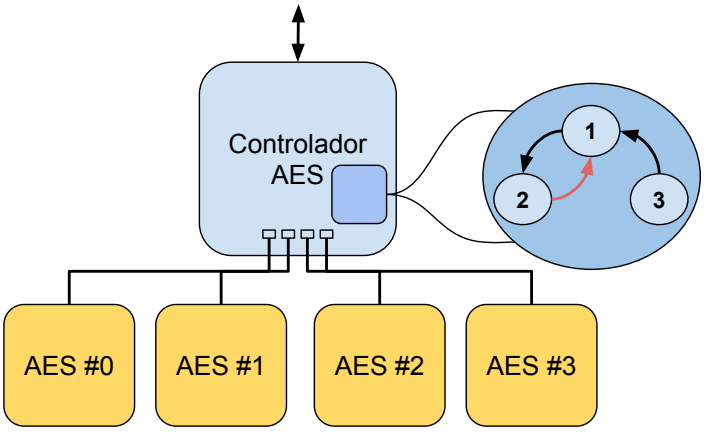


Figura 2.14: Acelerador de encriptação com máquina de estados de controle de escrita com transição inválida.

2.6.2 Custo em Elementos lógicos e Frequência de Clock

A adição da arquitetura de depuração é responsável por impactar o circuito de duas formas: primeiramente iremos aumentar o gasto em *lookup tables* (LUT's) com a adição da arquitetura reduzindo assim o espaço disponível na FPGA, em especial o *buffer* elaborado. Em segundo lugar, devido a adição de um circuito extra e consequentemente um aumento na complexidade das ligações, a frequência de *clock* base de 400 Mhz para a *Arria 10*, sofrerá uma redução em seu *clock*, diminuindo assim a velocidade máxima de execução.

Tabela 2.1: Resultado Síntese- Módulos selecionados + Arquitetura de depuração

Resultados de síntese (Arria 10)				
Circuito	**Elementos Lógicos**	**Ram Blocks**	**Clock Máximo Base 400 Mhz**	**Tamanho Buffer**
4 x AES	**84.250/427.200**	**717**	**327 Mhz**	**1024x512**
4 x AES	**84,161/427.200**	**717**	**327 Mhz**	**512x512**
4 x AES	**84,098/427.200**	**717**	**327 Mhz**	**256x512**
Natalius	**81.409/427.200**	**436**	**400 Mhz**	**1024x512**
Natalius	**81.349/427.200**	**436**	**400 Mhz**	**512x512**
Natalius	**81.294/427.200**	**436**	**400 Mhz**	**256x512**

Como é possível notar através da tabela 2.1, o consumo de elementos lógicos se mostrou relativamente alto para ambos os circuitos utilizados. Esse alto valor de elementos lógicos, ocorre pois, além da arquitetura e o circuito sob verificação, está embutido dentro deste valor todas as camadas de comunicação utilizada para viabilizar a comunicação ente CPU-FPGA. Além disso, podemos observar que a frequência

máxima de *clock* se manteve constante para ambos os circuitos, sendo esta até a velocidade máxima de execução *400 Mhz* para o processador *Natalius*. Com isso, o impacto final no *clock* será gerado exclusivamente pelo circuito sob verificação não impactando no desempenho do circuito.

2.7 Conclusão e Trabalhos futuros

Este trabalho apresenta uma nova arquitetura de verificação capaz de permitir o uso de dinâmico em tempo de execução de plataformas heterogêneas baseadas em CPU-FPGA na verificação de circuitos digitais. Esta arquitetura permite que os circuitos sejam estimulados em velocidades iguais ou próximas ao sua velocidade máxima.

Além disso, explorando a capacidade de uma transferência de dados , somos capazes de coletar um grande volume de informações em um espaço de tempo reduzido, o que aumenta significativamente a produtividade em ambientes de verificação.

Como trabalhos futuros para este projeto, iremos adicionar novas funcionalidades a arquitetura e também na API elaborada, permitindo que um conjunto maior de sinais possa ser selecionado pelo usuário através do uso de redes de interconexões. E por fim, adicionar ao sistema uma forma estruturada para visualização dos sinais coletados.

Capítulo 3

A Hybrid CPU-FPGA System to Detect Race Conditions

The evolution of computer algorithms and micro-architectures gave to developers, new mechanisms to create applications, exploring the potential of modern computers. As a result, it created new challenges in software development, specifically during application tests. This paper proposes Hambug, a method to help developers during tests of multithread applications, exploring the potential of CPU-FPGA architectures as run-time memory analyzer trough shared channels. The Hambug presents a memory analysis of parallel applications run time without CPU influence. This method shows a speedup of over six times, as compared with modern dynamic analysis tools.

3.1 Introduction

Integrated circuits are getting closer to their processing limit since the use of strategies such as pipelines and branch prediction reached are not able to contribute to new performance improvements. As a result, integrated circuits companies are investing in multi-core and heterogeneous architectures [51]. This type of architecture gave developers the capacity to create applications exploring the potential of parallelism, dividing complex into small tasks, and executing theirs in different CPUs. But with the use of parallelism in software development, new challenges arise in the development and test of applications. One of these challenges is the effort to synchronize the parallel tasks. When a parallel application doesn't adopt methods to synchronize tasks, unexpected behaviors happen, and results in application errors. A type of typical behavior in a parallel application is the race-condition. A race-condition happens as an unexpected result of an application sharing resources between tasks without a synchronization policy [52].

The standard solution to avoid race-conditions in multi-thread applications is include semaphores and mutexes in critical paths of the code [52]. The use of these structures controls the access of the critical area shared between tasks. The most common methods to detect race conditions in multi-thread applications are static [53], and

dynamic [54] analysis.

In this paper, we propose a hardware and software strategy to identify race-conditions in multi-thread applications. This strategy has the purpose of providing to the developer a runtime memory analyzer of applications under execution in CPU. With the shared memory between FPGA and CPU, the system can detect inconsistent access by tasks using Shared Variable Method [25].

In our strategy, the workflow requires, only that the user perform a quick FPGA instrumentation inside the code. Using this strategy, we reduce the overhead to execute an analysis of the system under test.

We organize this paper as follows: Section 3.2 introduces the concepts of race-condition detection strategies, using static and dynamic analysis in multithread application scenarios. Section 3.3 presents both hardware and software created to interact with DRAM memory. Also, we present an application example using the Hambug framework. In Section 3.4, we present the overhead of the Hambug method compared with the state of art race-conditions analysis tools. In Section 3.5, we discuss results and the next steps to increase Hambug efficiency.

3.2 Related Work

To detect race conditions in multi-thread applications, we adopt strategies in static and dynamic scenarios. In static source code, a graph of this code has generated to preview all possible states of this application. With this graph, static analysis tools can determine if, in this application, exists a possible race-condition or not [55]. Unfortunately, static code analysis tools are not efficient for more complex applications.

In dynamic analysis, they store all states of the code under test in a specific data structure, using this structure, the dynamic analysis can detect if the system has race conditions or not [56]. This approach has an excellent time efficiency if compared with static analysis tools. On the other hand, it can generate false positives once the data collected is subject to non-determinism on the operating system. Also, the number of states increases when you increase the number of threads [57].

Flanagan *et al.* [58], Netzer *et al.* [52], and Jannesari *et al.* [59] contextualize the race condition in a shared resource accessed by other threads or processes. Those papers present the problem of multi-threaded applications sharing the same resource. They also evaluate solutions to several problems, mainly related to how the usage of semaphores and mutexes can be useful to synchronize a shared resource.

Lai *et al.* [25] discuss solutions to synchronize a multi-threaded application in heterogeneous computing scenarios. In architectures such as GPUs and FPGAs, the detecting process of race conditions is even more complicated compared to CPUs. In addition to having no frameworks for this purpose, these platforms focus exclusively

on high-reuse parallel processing and information sharing among their processing nodes, which further increases the possibility of incorrect execution flow.

Lai *et al.* [25] proposes the development of an auxiliary circuit integrated into the PCI-E communication slot, able to manage the communication flow between GPU and CPU. The aim is to analyze communication or synchronization problems between both systems. Primarily, the project aims to detect race conditions in kernels described in OpenCL through this architecture, which had a communication slot coupled to a buffer to collect information at runtime, and an application able to read this data and extract bottlenecks and synchronization issues during data exchange on CPU and GPU. However, it is applicable not only in GPU but also in FPGA, and the proposal showed that buffer overhead would significantly increase the communication delay between CPU and GPU, which would degrade the execution of these applications.

In another perspective, they elaborate a proposal to detect race conditions in multi-threaded applications in a service application scenario. The authors use dynamic and static code analysis simultaneously through the Valgrind framework [60], reducing the number of states in purely static code analysis.

For the excessive use of memory in the dynamic code analysis process, O'Callahan *et al.* [53] present a method to reduce the number of states needed to store for a representation of the application flows. In Rhodes *et al.* [54], the authors perform the combination of static and dynamic analysis of a given code aiming to improve the search, avoiding unnecessary lines of code by markings made through a previous static analysis of the system, thus, reducing the number of dynamic checks.

Differently of previous works, the Hambug method uses the CPU-FPGA shared RAM to collect the changes of predetermined memory addresses. Here, the FPGA can read memory addresses directly from DRAM, using L3 processor cache. This approach reduces the false positives for scheduling steps and execution priorities.

3.3 Hambug Architecture

3.3.1 Hardware

The architecture proposed in figure 3.1 reads, and stores changes in memory addresses selected by the tester, for later analysis.

The architecture is divided into four main modules and has the propose of request data of a specific address previously defined in code in a memory address shared between CPU and FPGA. Each module has described as follows:

1. Requester: This module is responsible for starting the communication with the Intel back-end interface, making read requests from IIC. The response from IIC

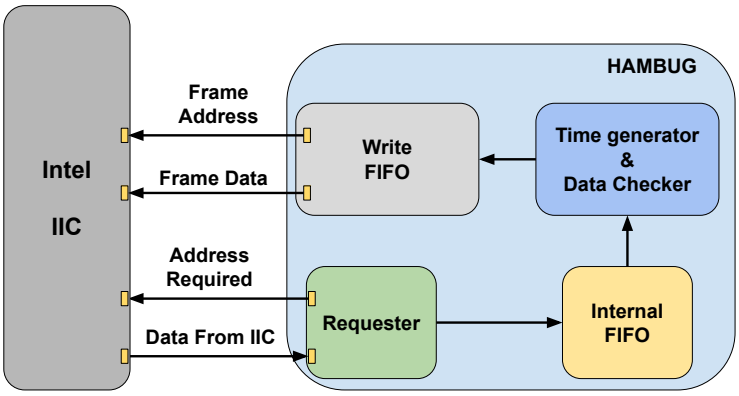


Figura 3.1: Hambug block diagram.

needs many clock cycles to receive a response, and this module is responsible for waiting and detecting when a requested value is ready for the next step.

2. Internal FIFO: It's not possible to read every change in memory. This issue happens because the interface between DRAM and FPGA uses the L3 cache level via dynamic channels, and this resource has shared between CPU and FPGA, so it's not possible to write or read when CPU is requesting or writing data on DRAM. Figure 3.2 illustrates the DRAM interconnection of CPU and FPGA to DRAM using the L3 Cache. To reduce this issue and prevent a loss of states during software execution, we use a simple FIFO to buffer all collected data when write channels are not available.

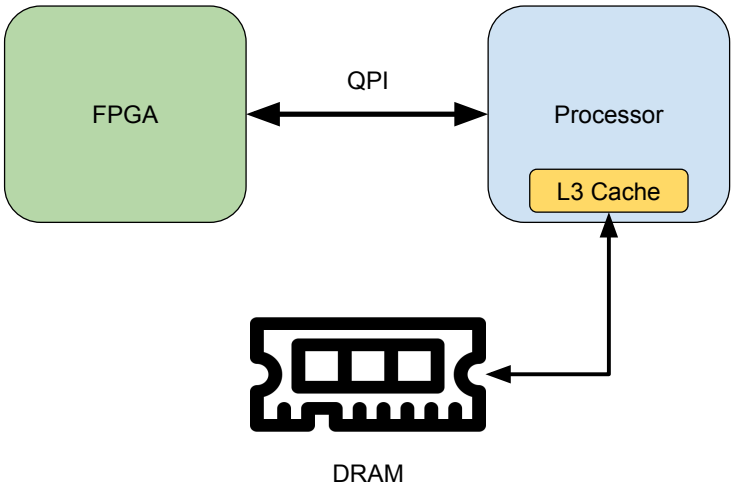


Figura 3.2: Interface between FPGA and DRAM memory.

3. Time generator & Data Checker: This module detects every change in data requested and build the data frame to analyze after code execution. The frame structure frame is composed of the data under analysis, and the exact clock time when the change happens. The data frame uses a fixed size of 64 bytes, as shown in 3.5.

4. Write FIFO: This module is responsible for saving data collected during program execution in DRAM for post-analysis. The Write FIFO controls the data saving progress, managing addresses, and signals received from IIC.

3.3.2 Software

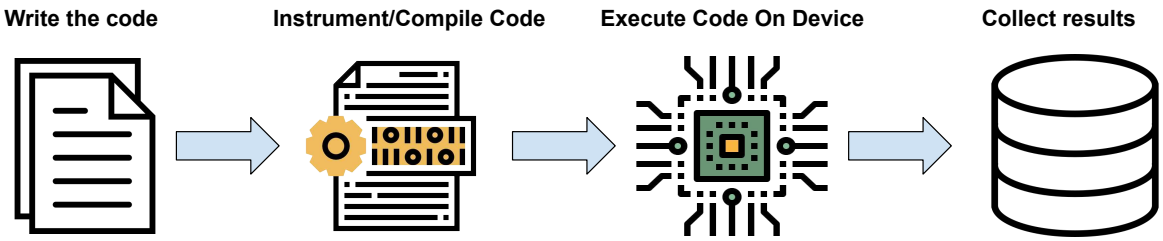


Figura 3.3: HAMBUG instrumentation fluxogram.

We divide the data analysis using heterogeneous architectures into software and hardware steps. The hardware step is responsible for detecting changes in the shared addresses during program execution. The software configures hardware design, providing to Hambug, the addresses to analyze, and the addresses to store the changes of the selected memory address. The algorithm 3.4 shows the usage of the Hambug method.

Figure 3.3 illustrates the steps to get the data generated by an arbitrary code running over *Hambug* method. First, you need to select the memory addresses to Hambug Analyze. Then, you need to run your application using Hambug framework, instantiating the FPGA modules, as shown in 3.4. After code execution, an array of the **result_t** data type. This data type stores 64 bytes, when you can use 56 bytes to monitor the changing of one or a group of variables or complex objects. We use the last 8 bytes to store the cycle time when a change was detected, providing a time factor for analyzing the behavior of that portion of memory. Figure 3.5 represents memory alignment used of a *result_t*.

```
#define window 1000;
void thread1(int *sharedResource){
        for(;;){
        //do something
    }
}
void thread2(int *sharedResource){
        for(;;){
        //do something
    }
}
int main(){
HAMbug = new FPGA();
auto sharedResource = (int*) HAMbug->allocShared(sizeof(int));
auto saveBuffer = (result_t*) HAMbug->allocShared(window*sizeof(
   result_t));
HAMbug.writeConfig(0,intptr_t(sharedResource));
HAMbug.writeConfig(1,intptr_t(saveBuffer));
HAMbug.writeConfig(2,window);
std::thread t1 (thread1,sharedResource);
std::thread t2 (thread2,sharedResource);
struct timespec pause;
pause.tv_sec = 0;
pause.tv_nsec = 800;
while (!HAMbug->done(){
 nanosleep(&pause,NULL);
}
return 0;
}
```

Figura 3.4: Example of instrumented code using Hambug.

Hambug output is a buffer containing information about each writing operation. We track the timeline, thread identification, and state before and after the change. Figure 3.5 illustrates frame format.

Figura 3.5: Hambug frame format.

3.3.3 Race Tracking Algorithm

To analyze the data-trace generated by Hambug, we use the SV-Track algorithm proposed by Lai *et al.* [25] as proof of concept. Some adaptations were necessary for the algorithm to execute in our context, which, differently from the original context, works in time with clock cycles. Figure 3.6 illustrates the operation of the algorithm, which starts determining whether the defined memory address is a shared resource or not, using the time between different thread access. The algorithm identifies an address as a shared resource or a semaphore and performs an analysis of thread memory access to determine if it is likely of these addresses to have a race condition.

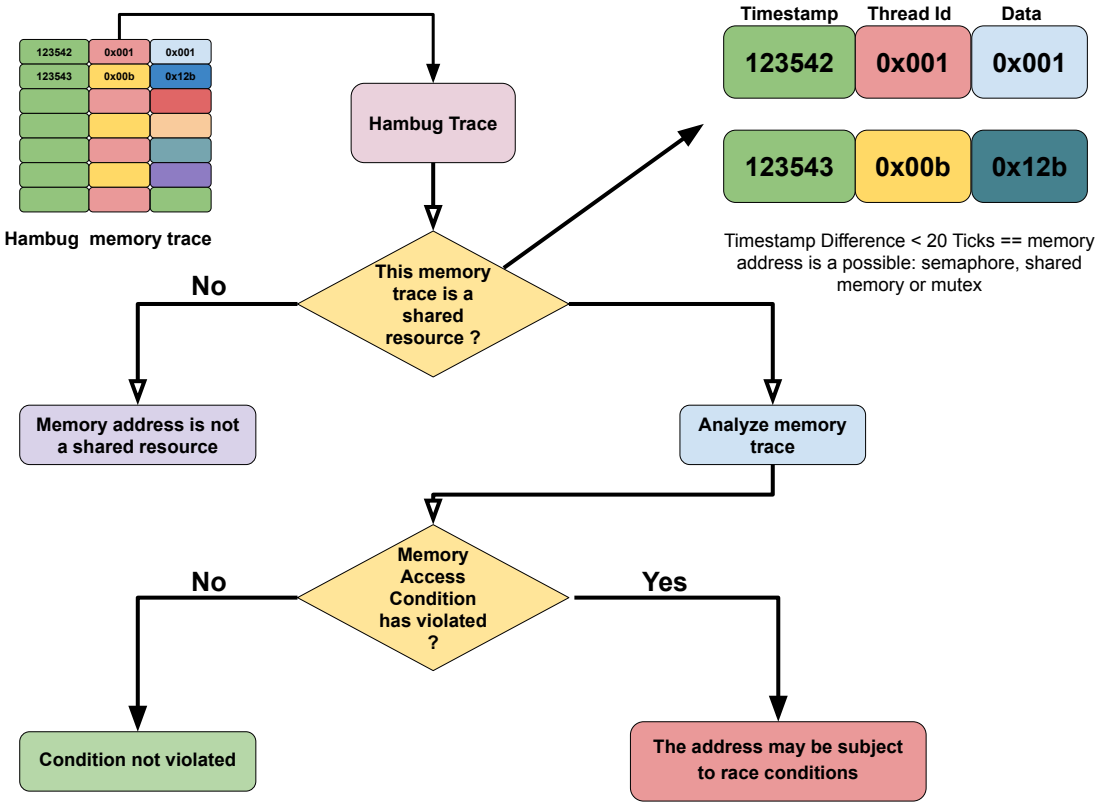


Figura 3.6: Race Tracking Algorithm.

3.3.4 Limitations

In this paper, we use a platform that provides a communication channel between CPU and FPGA using DRAM and L3 cache. These channels provide fast communication between FPGA and DRAM. However, despite the fast communication, the CPU runs up to 2.2 GHz, while the FPGA runs at 400 MHz. This difference can result in an undersample limitation. Nevertheless, this technical limitation can be solved by either using faster FPGAs or decreasing the CPU clock speed.

To detect the changes in shared memory, we use algorithm 3.7. This algorithm requests data on the memory address and executes stalls in Hambug architecture

```
Old Data: 0New Data: 0
Old Data: 0New Data: 10
Old Data: 10New Data: 20
Old Data: 20New Data: 30
Old Data: 30New Data: 20
Old Data: 20New Data: 30
Old Data: 30New Data: 40
Old Data: 40New Data: 50
Old Data: 50New Data: 40
Old Data: 40New Data: 50
```

Wrong Value

(a) Example of IIC incorrect value using a multithread vector sum algorithm, without stall algorithm.

```
Old Data: 0New Data: 0
Old Data: 0New Data: 10
Old Data: 10New Data: 20
Old Data: 20New Data: 30
Old Data: 30New Data: 40
Old Data: 40New Data: 50
Old Data: 50New Data: 60
Old Data: 60New Data: 70
Old Data: 70New Data: 80
```

Correct Value

(b) Example of IIC correct value, including a "stall"during IIC data request.

to avoid cache inconsistency. A cache inconsistency occurs when Hambug makes requests in a static memory address repeatedly. These requests eventually result in incorrect values received from IIC. Figure 3.8a is an example of cache inconsistency when executing a multi-threaded algorithm without stalls after data requests. Figure 3.8b shows an example of a valid data request using a stall during data requests.

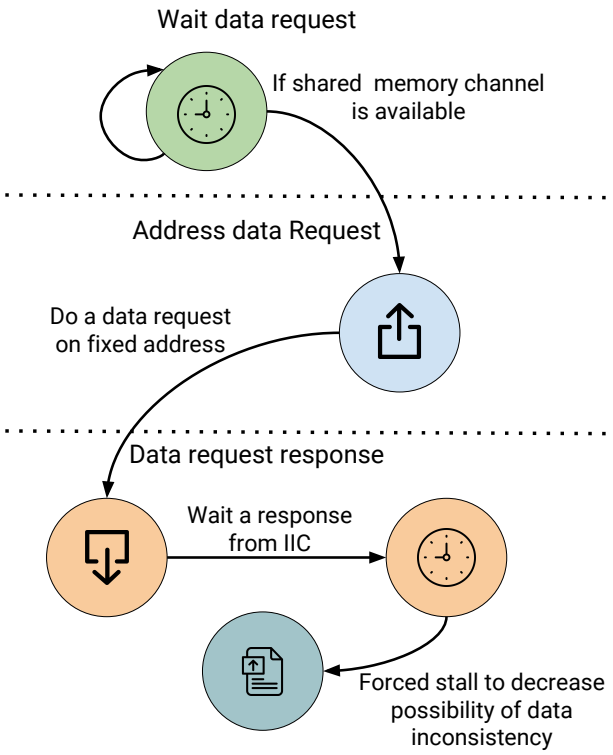


Figura 3.7: Stall Hambug Algorithm.

3.4 Experimental results

To test *Hambug* overhead, we select 11 algorithms as presented in Table 4.1. We use a Intel Xeon Processor with 14 cores, running at 2.2 GHz connected with an Arria 10 FPGA by Quick Path Interconnect (QPI) 3.2.

3.4.1 View Window Analysis

As previously mentioned in our architecture, the FPGA sampling rate is slower than the CPU clock rate, potentially reducing the view window. In this section, we test the view window of a shared address in a *Vectorial Sum Algorithm* inserting halts inside the critical code regions. The values presented in 3.9, use a view window fixed in 10,000 samples, and a **parallel sum** algorithm with 10,000 elements with four working threads. The view window in this scenario is the memory operations in the algorithm detected by Hambug.

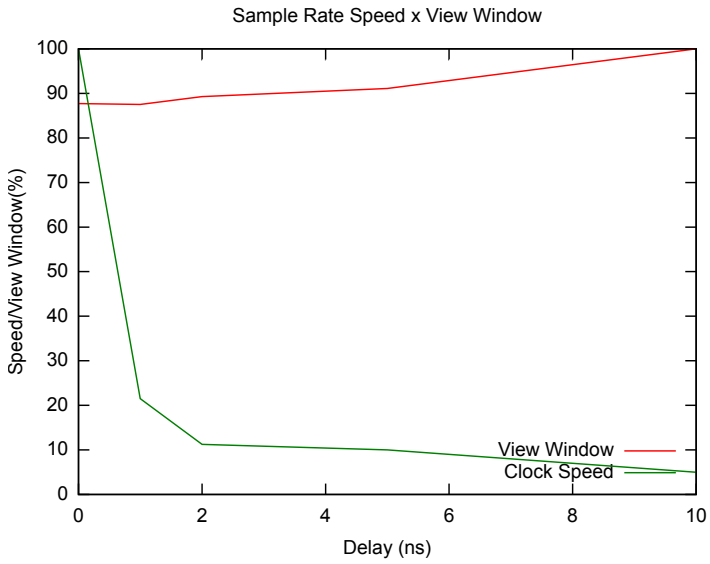


Figura 3.9: View Window vs Clock Speed.

As shown in figure 3.9, the view window increases when the clock speed decreases because, during the code execution, we insert halts inside critical sections to decrease the number of operations executed by CPU under shared memory. Thus, we allow Hambug to detect the changes in the selected memory address.

3.4.2 Evaluating Hambug Efficiency

We compare Hambug overhead with other the race detection techniques presented in [61], [56], [54]. We calculate the overhead using Equation 3.1, where x_o is the time to execute the code without instrumentation, and x_i is the time to run the application with Hambug and the SVTrack race tracking algorithm.

$$Overhead = \frac{x_i - x_o}{x_o} \tag{3.1}$$

Table 4.1 presents time and overhead to execute algorithms. Each algorithm ran in equivalent instances, and we calculate execution time using the mean of these values.

To ensure closer proximity to a real application and observe the impact of the Hambug method in time execution, we do not define a halt to increasing the view window of the method. In almost all cases, Hambug presents less overhead than other dynamic analysis tools. The only exception is the Series benchmark running under the Big Foot tool, and the difference is minimal. Hambug presents better overhead because the FPGA sampling is running in parallel, thus, not affecting the CPU running times. The small overhead observed is due to the application FPGA configuration time and SVTrack algorithm, which we execute after we perform the memory traces.

3.5 Conclusion and Future Work

In this paper, we propose Hambug, a new method to identify race conditions in multithread applications using CPU-FPGA architectures as runtime memory analyzers. The overhead imposed by Hambug is lower than other dynamic analysis tools because we use the FPGA to perform that memory address sampling in parallel of the multithread applications.

As future work, we will improve the proposed architecture, especially the memory requests from IIC, to increase the view window, and decrease the injection locks inside the coder under test. We will develop a secondary tool to use the gathered data to determine if an application has a parallel bug or not.

Capítulo 4

Conclusão

Nesta Dissertação, foram apresentados dois trabalhos referentes ao uso de arquiteturas heterogêneas no processo de verificação a nível de hardware e software. Esta dissertação, demonstrou através dos artigos apresentados, que o uso destas plataformas em projetos voltados a verificação tanto na parte de hardware, quanto de software é bastante promissor, e pode ser utilizado de forma efetiva tanto a nível de hardware, quanto a nível de software, apesar dos desafios relacionados a construção e validação dos projetos de hardware responsáveis por essas tarefas.

No capitulo 2 foi proposta uma arquitetura de depuração de circuitos digitais voltada ao ambiente HARP da Intel, composto por uma CPU e FPGA altamente acoplados. Seu principal objetivo foi propor um novo mecanismo de verificação de circuitos digitais para um ambiente de computação moderno, onde fosse possível se aproveitar de características, como a alta velocidade de comunicação entre a memória principal do sistema com o *FPGA*. Tendo como principal objetivo, reduzir o uso de recursos do circuito FPGA através do uso da própria memória RAM do sistema ao invés dos blocos de memória da FPGA. Através disso, foi possível reduzir de forma significativa o uso de recursos do FPGA, o que possibilitou o uso desse espaço na compilação de projetos maiores para verificação. Entretanto, alguns gargalos de recursos, referentes a camada de controle fornecida pela Intel, e sua infraestrutura de comunicação, foi responsável por consumir cerca de 18% de *Lookup Table* disponíveis.

Com a arquitetura proposta foram coletadas informações de circuitos previamente selecionados, e com isso foi possível observar o seu funcionamento e detectar falhas implantadas de forma arbitrária no projeto lógico. O uso da plataforma desenvolvida no capítulo 2 tem como principal vantagem a possibilidade de utilizar a memória principal do sistema (DRAM) ao invés dos blocos de memória do próprio *FPGA*, como ocorre em ferramentas como *Signal TAP* da *Altera* e o *ChipScope* da *Xilinx's*.

Foi possível observar, através de uma análise temporal, que a arquitetura de depuração não resultou na diminuição da velocidade final de *clock*. Isso possibilita com que o circuito seja emulado à uma velocidade igual, ou próxima ao seu valor real como mostra a tabela 2.1. E por fim, considerando a velocidade de comunicação entre memória principal e *FPGA*, é possível controlar a emulação feita pelo *FPGA* de forma

semelhante a adotada por ferramentas de depuração. Isso permite que o sistema apresente informações em tempo real para o projetista, dando a ele a possibilidade de visualizar possíveis problemas em seu projeto de forma semelhante aos *breakpoints*.

No capitulo 3 destacamos a forma como a computação vem evoluindo com o passar dos anos e como isso possibilitou que os sistemas computacionais se tornassem mais rápidos. Entretanto, toda essa evolução trouxe novos desafios para os desenvolvedores, sendo estes problemas relacionados a dificuldade em gerenciar o uso de recursos compartilhados, que quando mal utilizados, geram problemas como: *Race Conditions*, *Deadlocks* e *Starvation*. Além disso, se tornem problemas que além de difíceis de resolver se tornassem ainda mais difíceis de serem identificados. Isso acontece devido a fatores como a execução de tarefas por múltiplos núcleos de processamento, e a imprevisibilidade do sistema operacional ao alocar os processos durante o escalonamento.

Sabendo disso, a arquitetura proposta no capítulo 3 tem como objetivo ser uma ferramenta auxiliar na depuração de aplicações *multithread*. Isso é possível, uma vez que ao utilizarmos uma estrutura de *hardware* totalmente desacoplada ao sistema operacional e partilhando o mesmo espaço de memória leituras nos endereços compartilhados podem ser feitas de forma assíncrona ao sistema operacional e sua imprevisibilidade, permitindo assim uma visualização completa de todo o ciclo de vida que os dados em análise sofreram durante a execução.

Para coletar essas informações, foi projetada uma arquitetura composta por módulos de requisição de dados integrados diretamente ao controlador de interface da *Intel* (IIC), sendo esse responsável por coletar os dados durante a execução de um programa. Entretanto, alguns obstáculos foram encontrados durante a implementação da arquitetura, o primeiro deles foi a imprevisibilidade dos dados coletados, uma vez que a *IIC* era incapaz de fornecer dados válidos se os mesmos fossem solicitados em um endereço de memória fixo 3. Outra limitação, está ligada a diferença de *Clock* entre *CPU* e *FPGA* que impossibilita o acesso aos dados manipulados pela aplicação em sua velocidade máxima.

Superado o desafio de instabilidade da *IIC*, foi feito um estudo acerca da janela de visualização quando utilizamos a arquitetura em um algoritmo de soma vetorial. Foi possível observar através deste algoritmo que mesmo sendo impossível visualizar completamente todo o ciclo de vida da região de memória em análise, pudemos coletar aproximadamente 90% da variação dos dados sem a necessidade de ”atrasar” a seção crítica do algoritmo como mostra o gráfico 3.9.

Também foi analisado o impacto gerado pelo uso das arquiteturas propostas nesta dissertação em algoritmos de computação matemática, sob grandes massas de dados. Esse estudo demonstrou que apesar do canal compartilhado entre *CPU* e *FPGA* para acesso aos blocos de RAM ser utilizado tanto pela *FPGA*, quanto pela *CPU*, o *overhead*

não foi significativamente grande para o tempo de execução da aplicação como um todo. Entretanto, devido a diferença na velocidade de *clock* entre os dois dispositivos, só foi possível realizar uma cobertura completa da aplicação, reduzindo sua velocidade de execução, como demonstra a figura 3.9.

Referências Bibliográficas

[1] Wallace Witkowski. Intel admits vulnerability, but plays down effects; stock slides, amd gains, Jan 2018.

[2] Fatemeh Eslami and Steven JE Wilton. An adaptive virtual overlay for fast trigger insertion for fpga debug. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 32–39. IEEE, 2015.

[3] Stanley Wolf and Richard N Tauber. Silicon processing for the vlsi era, vol. 1: Process technology. *and*, 526:388, 1986.

[4] Kimiyoshi Usami. Pipeline circuit, August 17 1993. US Patent 5,237,664.

[5] Kai Hwang and Naresh Jotwani. *Advanced computer architecture, 3e*. McGraw-Hill Education, 2016.

[6] Scott E Thompson and Srivatsan Parthasarathy. Moore's law: the future of si microelectronics. *Materials today*, 9(6):20–25, 2006.

[7] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[8] Saint John Walker. Big data: A revolution that will transform how we live, work, and think, 2014.

[9] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[10] Denise Stringhin. *Introdução à Computação Heterogênea*. USP, 2012.

[11] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[12] Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 16. IEEE Press, 2013.

[13] Li Lei, Kai Cong, Zhenkun Yang, Bo Chen, and Fei Xie. Hardware/software co-monitoring. *arXiv preprint arXiv:1905.03915*, 2019.

[14] Dick Price. Pentium fdiv flaw-lessons learned. *IEEE Micro*, 15(2):86–88, 1995.

[15] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[16] Georges Gielen, Nektar Xama, Karthik Ganesan, and Subhasish Mitra. Review of methodologies for pre-and post-silicon analog verification in mixed-signal socs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1006–1009. IEEE, 2019.

[17] Agalya Rajendran and Muthaiah Rajappa. Attack on trace buffer: A study on observability versus security in post-silicon debug. In *2019 International Conference on Communication and Signal Processing (ICCSP)*, pages 0332–0334. IEEE, 2019.

[18] José Miranda Nacif, Flávio Paula, Harry Foster, Claudionor Jr, and Antônio Fernandes. The chip is ready. am i done? on-chip verification using assertion processors. pages 111–, 01 2003.

[19] Misun Yu, Joon-Sang Lee, and Doo-Hwan Bae. Adaptivelock: efficient hybrid data race detection based on real-world locking patterns. *International Journal of Parallel Programming*, 47(5-6):805–837, 2019.

[20] Faming Lu, Ranran Tao, Yuyue Du, Qingtian Zeng, and Yunxia Bao. Deadlock detection-oriented unfolding of unbounded petri nets. *Information Sciences*, 497:1–22, 2019.

[21] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

[22] Fpga-based prototyping.

[23] Séamas McGettrick, Kunjan Patel, and Chris Bleakley. High performance programmable fpga overlay for digital signal processing. In *International Symposium on Applied Reconfigurable Computing*, pages 375–384. Springer, 2011.

[24] Paul Graham, Brent Nelson, and Brad Hutchings. Instrumenting bitstreams for debugging fpga circuits. In *null*, pages 41–50. IEEE, 2001.

[25] Cheng-Kung Lai, Chih-Wei Yeh, Chia-Heng Tu, and Shih-Hao Hung. Fast profiling framework and race detection for heterogeneous system. *Journal of Systems Architecture*, 81:83–91, 2017.

[26] UFV, Conselho Técnico de Pós Graduação da Universidade Federal de Viçosa. Normas de redação de teses e dissertações. http://www.dpi.ufv.br/arquivos/ppgcc/doc/PPG-2015-normascorrigidas.pdf, 2018. Accessed: 2018-06-14.

[27] Robert Safranek. Intel® quickpath interconnect overview. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–27. IEEE, 2009.

[28] Joon-Sung Yang and Nur A Touba. Expanding trace buffer observation window for in-system silicon debug through selective capture. In *VLSI Test Symposium, 2008. VTS 2008. 26th IEEE*, pages 345–351. IEEE, 2008.

[29] José Augusto Miranda Nacif, Flávio Miana de Paula, Harry Foster, Claudionor José Nunes Coelho Jr, and Antônio Otávio Fernandes. The chip is ready. am i done? on-chip verification using assertion processors. In *VLSI-SOC*, page 111, 2003.

[30] Lucas B da Silva, Danilo Almeida, José Augusto M Nacif, Ismael Sánchez-Osorio, Carlos A Hernández-Martínez, and Ricardo Ferreira. Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pages 1–7. IEEE, 2017.

[31] Jeronimo Penha, Lucas Bragança, Danilo Almeida, Jose Nacif, and Ricardo Ferreira. Add - uma ferramenta de projeto de aceleradores com dataflow para alto desempenho. In *Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, Porto Alegre, RS, Brasil, 2017. SBC.

[32] Norman Hendrich. A java-based framework for simulation and teaching: Hades—the hamburg design system. In *Microelectronics Education*, pages 285–288. Springer, 2000.

[33] Alex Borges Michele Nogueira José Augusto M. Nacif Guevara Noubir Kristtopher Coelho, Danilo Damião. Cryptography algorithms in wearablecommunication: An empirical analysis. *IEEE COMMUNICATIONS LETTERS*, 2019.

[34] Prabhat Mishra, Ronny Morad, Avi Ziv, and Sandip Ray. Post-silicon validation in the soc era: A tutorial introduction. *IEEE Design & Test*, 34(3):68–92, 2017.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[36] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[37] FPGA Cross-Platform and Application Developers. *Simplify Software Integration for FPGA Accelerators with OPAE*, 2017.

[38] Altera Verification Tool. Signaltap ii embedded logic analyzer, 2006.

[39] Zdravko Panjkov, Andreas Wasserbauer, Timm Ostermann, and Richard Hagelauer. Hybrid fpga debug approach. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.

[40] Fatemeh Eslami, Eddie Hung, and Steven JE Wilton. Enabling effective fpga debug using overlays: Opportunities and challenges. *arXiv preprint arXiv:1606.06457*, 2016.

[41] Al-Shahna Jamal, Jeffrey Goeders, and Steven JE Wilton. Architecture exploration for hls-oriented fpga debug overlays. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 209–218. ACM, 2018.

[42] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[43] Richard M Stallman and Roland H Pesch. *Debugging with Gdb: The Gnu Source-level Debugger Fifth Edition, for Gdb Version, April 1998*. iUniverse Com, 2000.

[44] Eric Roesler and Brent Nelson. Debug methods for hybrid cpu/fpga systems. In *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*, pages 243–250. IEEE, 2002.

[45] Eddie Hung and Steven JE Wilton. Towards simulator-like observability for fpgas: a virtual overlay network for trace-buffers. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 19–28. ACM, 2013.

[46] Lucas Bragança da Silva. Simplifying hw/sw integration to deploy multiple accelerators for cpu-fpga heterogeneous platforms. *SAMOS*, 2018.

[47] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[48] Open Cores. *Open Cores*, jan 2006.

[49] Prabhat K Gupta. Xeon+ FPGA Platform for the Data Center. In *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119, 2015.

[50] David Van Campenhout, Trevor Mudge, and John P Hayes. Collection and analysis of microprocessor design errors. *IEEE Design & Test of Computers*, (4):51–60, 2000.

[51] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[52] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[53] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Acm Sigplan Notices*, volume 38, pages 167–178. ACM, 2003.

[54] Dustin Rhodes, Cormac Flanagan, and Stephen N Freund. Bigfoot: Static check placement for dynamic race detection. In *ACM SIGPLAN Notices*, volume 52, pages 141–156. ACM, 2017.

[55] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[56] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[57] Eli Pozniansky and Assaf Schuster. *Efficient on-the-fly data race detection in multithreaded C++ programs*, volume 38. ACM, 2003.

[58] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96. ACM, 2001.

[59] Ali Jannesari and Walter F Tichy. On-the-fly race detection in multi-threaded programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, page 6. ACM, 2008.

[60] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003.

[61] Cormac Flanagan, Cormac Flanagan, and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Notices*, pages 256–267. ACM, 2004.

Apêndice A

Tabela 4.1: Overhead comparison between Hambug FPGA-based verification method and oftware-based methods

Algorithm	Thread	Time			Overhead			
		Hambug	SV-Track	Total	Hambug	Atomizer	Fasttrack	Big Foot
Prod/Consum	2	6,68	0,067	6,747	1,02	N/A	N/A	N/A
Monte Carlo	2	32,7	0,041	32,741	0,62	2,2	10,11	7,38
Prime	4	0,014	0,059	0,199	1,75	N/A	N/A	N/A
Elevator	2	0,14	0,064	0,204	0,02	11,14	N/A	N/A
Sum	4	0,326	0,049	0,375	1,11	N/A	N/A	N/A
TEA	4	120,3	0,5	120,8	1,13	N/A	N/A	N/A
TSP	2	2,08	0,74	2,82	1,15	48,2	N/A	N/A
FFT	2	0,213	0,115	0,328	1,06	N/A	N/A	N/A
Series	2	6,102	0,023	6,125	0,013	1,05	0,05	0,01
Geomean	2	0,042	0,8	0,842	0,07	N/A	N/A	N/A
Matrix	2	2,204	0,113	2,317	0,001	N/A	26,86	6,68