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

Danilo D. Almeida, Lucas Bragança da Silva, Ricardo Ferreira,José Augusto Miranda Nacif
Universidade Federal de Viçosa, Brazil
{danilo.damiao,lucas.braganca,ricardo,jnacif}@ufv.br

***Resumo*—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. Este trabalho 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.**

***Index Terms*—Circuitos Integrados, Computação Heterogênea, Verificação, Pré-Silício**

## I. 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 sendo cada vez mais requisitada. 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 vem 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 [10]. 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 [13], e atualmente as falhas *MeltDown* e *Spectre* [9] 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 este 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 [14], vêm mostrando que sua 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 [3] 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 [18]. 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 regravar a FPGA e aplicar novos vetores de teste.

Este artigo está organizado na seguinte forma: a seção II apresenta os principais aspectos da verificação utilizando FPGAs e as vantagens da utilização de *Overlays*. A seção III contextualiza o trabalho proposto em relação a outros que utilizam FPGA para verificação. Na seção IV iremos apresentar a estrutura da arquitetura proposta para coleta de dados em hardware, a seção V detalha a implementação em software

responsável pela comunicação com a FPGA e a ferramenta responsável pela geração parametrizada da arquitetura. Por fim, nas seções VI, VII iremos apresentar e discutir os resultados e os trabalhos futuros para melhorias da arquitetura proposta.

## II. Fundamentação Teórica

A complexidade dos circuitos digitais atuais e o espaço exponencial de estados possíveis. Isso torna 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 [12].

O uso de FPGAs reduz significativamente o custo final da verificação, pois não necessita com 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 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.

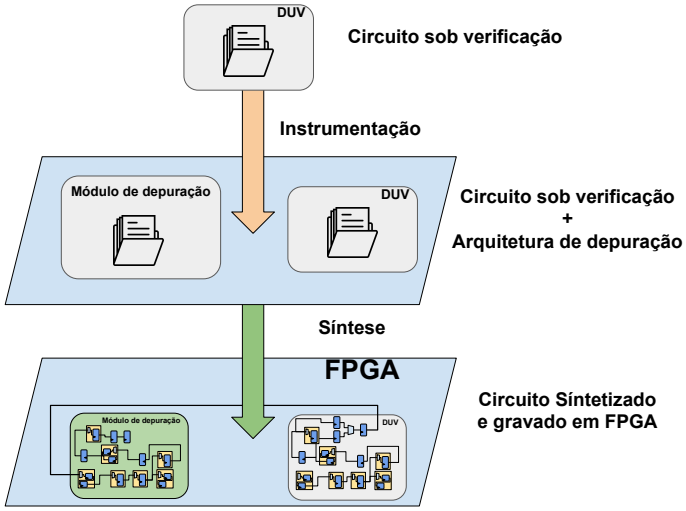


Figura 1. Fluxo de síntese de um circuito em FPGA.

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 nova iteração, será 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, já que um circuito, dependendo de sua complexidade pode ter seu tempo de síntese variado consideravelmente. Sabendo disso, a utilização de arquiteturas de *Overlays* voltados a depuração estão sendo cada vez mais utilizadas na verificação de circuitos digitais [5].

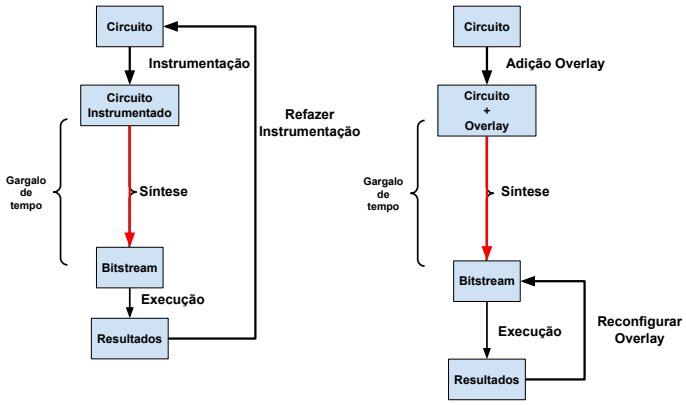


Figura 2. Diferenças de verificação em FPGA utilizando Overlays e técnicas gerais.

Estas plataformas reduzem significativamente o tempo de verificação, já que como funcionam 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 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.

## III. Trabalhos relacionados

O projeto de arquiteturas para verificação de circuitos integrados já é uma área bastante explorada pela literatura. O trabalho [8], propõe um modelo arquitetural para verificação de circuitos gerados através da ferramenta *LegUp* [1], uma ferramenta de síntese de alto nível que permite a elaboração de *dataflows* baseados em código C-ANSI. A arquitetura proposta em [1], permite que o usuário depure o circuito de forma semelhante ao que é feito no uso de depuradores a nível de software, como o depurador GDB [16]. Porém o trabalho é limita às arquiteturas geradas pela ferramenta *LegUp*, o que restringe o seu uso em outros tipos de circuitos.

O trabalho apresentado em [15] 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 a cerca de 8 anos. O trabalho apresentado em [7] propõe a utilização de *Overlays* reconfiguráveis no formato de redes de interconexões, que permitem com que uma maior variabilidade de sinais possam

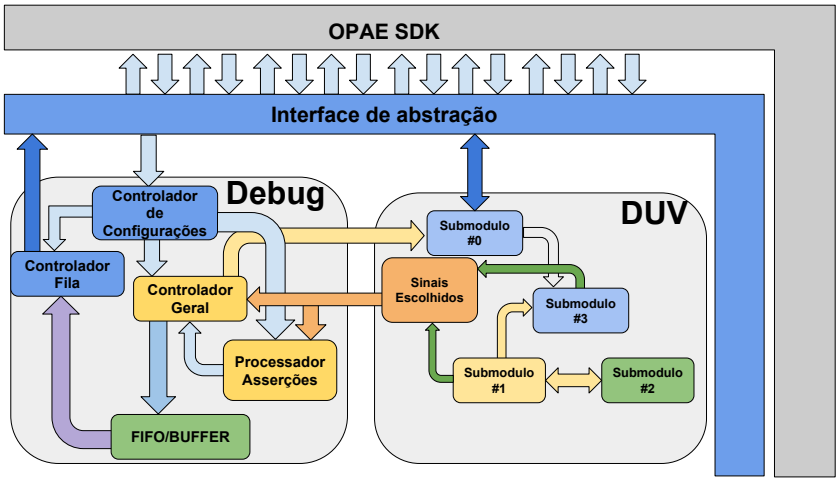


Figura 3. Arquitetura de depuração.

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 [5] 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, o trabalho 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 afim de encontrar falhas e realizar um mapeamento dos resultados de forma mais rápida que o uso tradicional dos FPGAs no domínio de verificação, que muitas vezes apresenta restrições de comunicação, restrição 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.

## IV. 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.

### A. 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 através da arquitetura proposta em [4], 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 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 [4]. A arquitetura de depuração (Debug) se comunica com o software através da 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.

### B. Processador de Asserções

Para que seja possível a identificação de erros do circuito, foi construído 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 [11] com asserções sintetizáveis. As asserções deste trabalho, possuem submódulos combinacionais e sequenciais, que permitem com que através da técnica de encadeamento de asserções, o módulo gerenciador de asserções descubra exatamente qual foi a asserção violada.

### C. 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 paralisar o seu funcionamento quando necessário. A Figura 4 mostra o conjunto de sinais compartilhados entre a arquitetura de depuração e o circuito alvo.

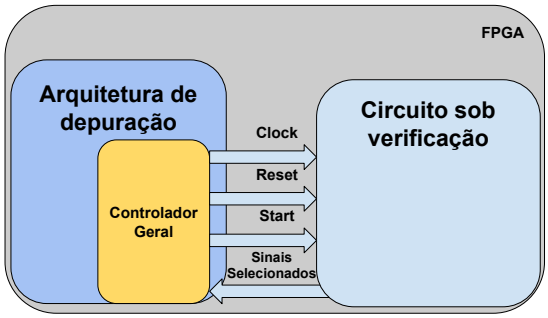


Figura 4. Comunicação entre o módulo de controle geral, e o circuito alvo.

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, irá servir 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 5 mostra o esquema de alocação de memória utilizada pela arquitetura. Dentro de um total de 64 bytes, os 4 bytes menos significativos serão utilizados para armazenar o ciclo onde aquele evento ocorreu, e o restante será utilizado para armazenar cada um dos sinais selecionados pelo projetista.

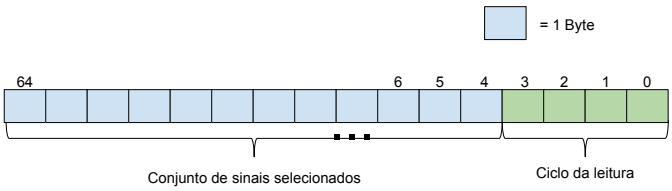


Figura 5. Alocação de dados de depuração no overlay.

### D. Controlador Fila

Como mencionado na seção IV-C 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.

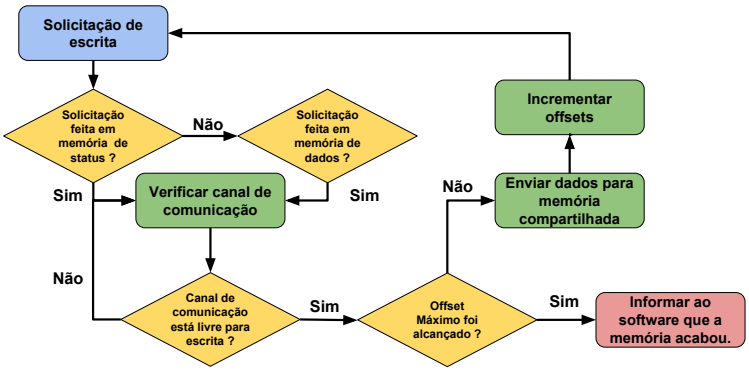


Figura 6. Fluxograma de execução para cópia de dados do buffer para memória compartilhada.

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 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, caso tenha sido, 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.

### E. 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 7 ilustra a estrutura do *buffer* circular utilizado na arquitetura.

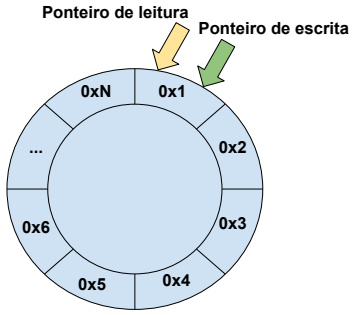


Figura 7. Estrutura de buffer circular implementado na arquitetura.

## V. Interface

Para estabelecer a comunicação entre CPU e FPGA foi implementada uma interface ou API (*Application programming interface*) que 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 [4]. 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 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 [4]. 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.

```
1  #include <AccManagement.h>
2  #define ENTRADA 5004
3  typedef struct {
4      uint8_t buffer[64];
5  }dados;
6  int main(){
7   auto *accMgr = new AccManagement();
8   bool accMask[accMgr->getNumAccelerators()];
9   accMgr->startAccelerators(accMask);
10  DebugAccelerator *debugAccelerator;
11  size_t numBytes = sizeof(dados)*ENTRADA;
12  auto *buffer = (dados*) malloc(numBytes);
13  for (auto &acc:accMgr->getAccelerators()) {
14   debugAccelerator = acc->
         getDebugAccelerator();
15  }
16  debugAccelerator->runCircuit(100,10000);
17  debugAccelerator->copyDebugQueue(numBytes,
        buffer);
18  delete accMgr;
19  return 0;
20  }
```

Figura 8. Código exemplo de execução de uma sessão de depuração.

### A. Ferramenta para geração de código

Também foi implementado 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* [17]. A Figura 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 IV. Por fim, um conjunto de arquivos descritos em *Verilog* será automaticamente criado pronto para uso no projeto em questão.

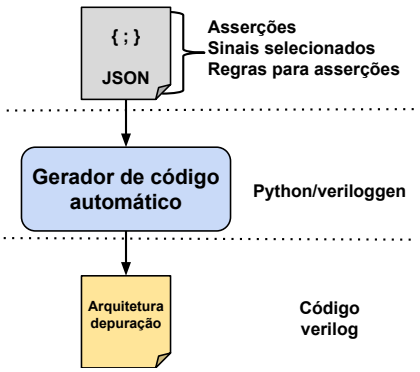


Figura 9. Fluxo para geração do código verilog do módulo de depuração.

## VI. 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 [2]. Estes módulos foram selecionado 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 [6].

### A. 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 [19] que ilustra os tipos de falhas dos grandes projetos de circuitos digitais.

*1) Caso 1:* Um erro recorrente, no desenvolvimento de circuitos digitais, é a atribuição incorreta de sinais em um determinado módulo [19]. Este tipo de erro que pode ocorrer se mantermos 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 caso 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 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.

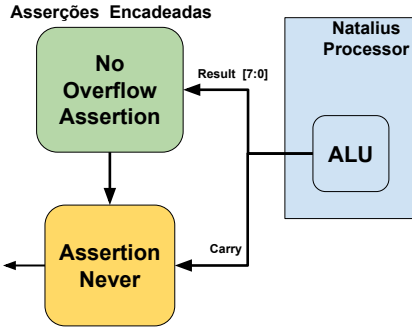


Figura 10. Conjunto de asserções utilizados.

A figura 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.

*2) 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. Este tipo de falha 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 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.

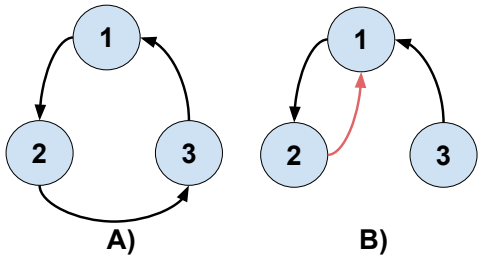


Figura 13. Máquinas de estados.

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 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 12 mostra o exato momento onde a asserção **Never** é violada, significando que foi passado um total de 482 ciclos e o sinal responsável por realizar uma solicitação de escrita na memória compartilhada não foi habilitado.

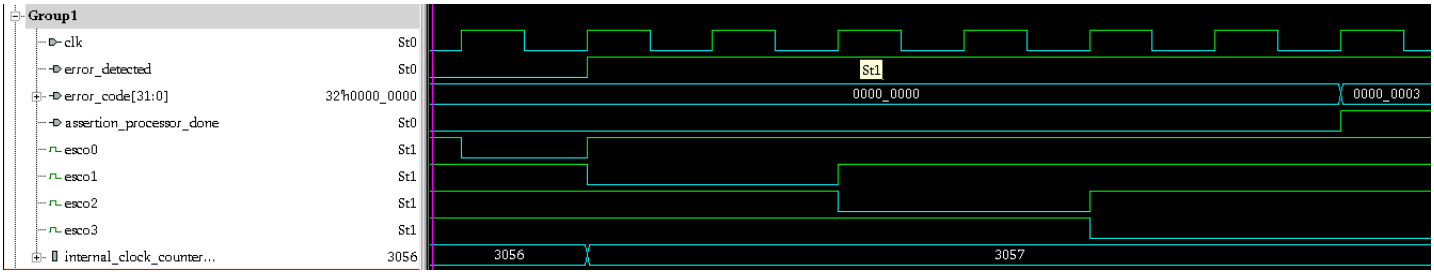


Figura 11. Violação da asserção No Overflow no ciclo de clock 3057.

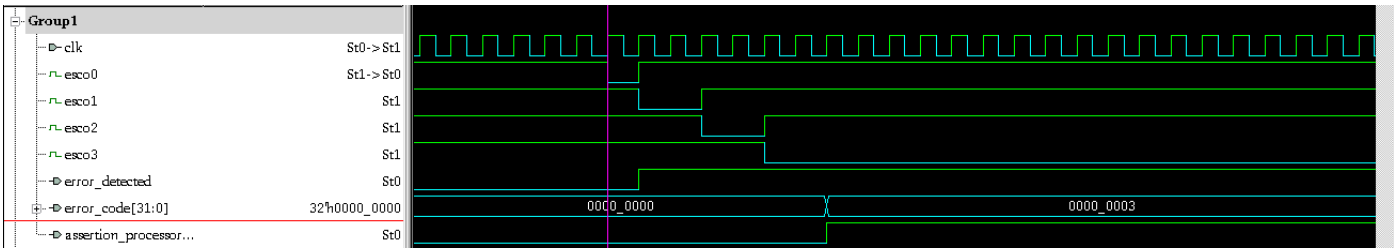


Figura 12. Violação da asserção No Overflow no ciclo de clock 485, encriptação foi realizada porém não foi copiada para memória compartilhada.

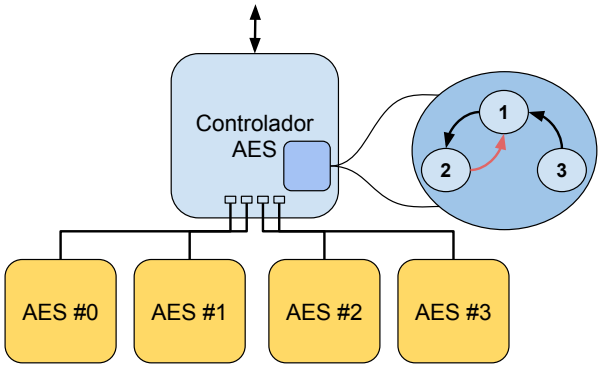


Figura 14. Acelerador de encriptação com máquina de estados de controle de escrita com transição inválida.

### B. *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 I
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 I, 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, é possível notar também 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, é possível notar que o impacto final no *clock* será gerado exclusivamente pelo circuito sob verificação não impactando no desempenho do circuito.

## VII. 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 com 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 com 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.

## AGRADECIMENTOS

Os autores deste trabalho gostariam de agradecer a UFV e a CAPES pelo suporte financeiro. Também gostariam de agradecer a Intel por fornecer o acesso a arquitetura CPU-FPGA utilizada neste trabalho, e agradecer a Synopsys pelo fornecimento das ferramentas de simulação.

## REFERÊNCIAS

[1] 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.

[2] Open Cores. Free open source ip cores and chip design, 2006.

[3] FPGA Cross-Platform. Simplify software integration for fpga accelerators with opae.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.