Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Pedro Arthur da Cunha Medeiros

Relatório de Estágio Integrado no Laboratório de Excelência em Microeletrônica do Nordeste - XMEN

Campina Grande
2024

Pedro Arthur da Cunha Medeiros

Relatório de Estágio Integrado no Laboratório de Excelência em Microeletrônica do Nordeste - XMEN

Relatório de Estágio Integrado submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Eletrônica

Orientador:
Marcos Ricardo de Alcântara Morais, D. Sc.

Campina Grande
2024

Pedro Arthur da Cunha Medeiros

Relatório de Estágio Integrado no Laboratório de Excelência em Microeletrônica do Nordeste - XMEN

Relatório de Estágio Integrado submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Aprovado em / /

Marcos Ricardo de Alcântara Morais, D. Sc.
UFCG

Gutemberg Gonçalves dos Santos Júnior, D. Sc.
Professor Convidado
UFCG

Campina Grande
2024

RESUMO

Este relatório descreve o trabalho realizado durante o estágio integrado no Laboratório de Excelência em Microeletrônica do Nordeste (XMEN), focado na verificação formal e funcional de núcleos de processamento baseados na arquitetura RISC-V. Utilizando o RISC-V Formal Verification Framework (RFVF), foram conduzidos testes no núcleo CV32E40P, continuando um trabalho feito anteriormente com uma versão diferente do *core*, agora realizando testes de consistência e da extensão Xcorev. Paralelamente, o estágio incluiu o desenvolvimento do processador RISC-X, também baseado em RISC-V, em que a verificação foi realizada por meio de um *testbench* em UVM e o RISC-V Formal.

Palavras-chave: Verificação de *Hardware*, Design de *Hardware*, RISC-V, Verificação Formal, UVM, CV32E40P, RISC-X.

ABSTRACT

This report presents the work carried out during the integrated internship at the *Excelência em Microeletrônica do Nordeste* (XMEN) Lab, focusing on the formal and functional verification of processor cores based on the RISC-V architecture. Using the RISC-V Formal Verification Framework (RFVF), tests were conducted on the CV32E40P core, coninuing a previously done work with a different version of the core, now carrying out consistency and Xcorev extension tests. In parallel, the internship included the development of the RISC-X processor, also based on RISC-V, where verification was carried out through a UVM testbench and Risc-V Formal.

Keywords: Hardware Verification, Hardware Design, RISC-V, Formal Verification, UVM, CV32E40P, RISC-X.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

CSR	*Control and/or Status Register*
DEE	Departamento de Engenharia Elétrica
DUT	*Design Under Test*
EX	*Execute*
GPR	*General Purpose Register*
HDL	*Hardware Description Language*
ID	*Instruction Decode*
IF	*Instruction Fetch*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
MEM	*Memory*
OBI	*OpenBus Interface*
PC	*Program Counter*
PMP	*Physical Memory Protection*
RFVF	*RISC-V Formal Verification Framework*
RTL	*Register Transfer Level*
RFVI	*RISC-V Formal Interface*
SIMD	*Single Instruction, Multiple Data*
SoC	*System on a Chip*
SVA	*SystemVerilog Assertions*
UFCG	Universidade Federal de Campina Grande
ULA	Unidade Lógico-Aritmética
UVC	*UVM Verification Component*

UVM	*Universal Verification Methodology*
WB	*Writeback*
XMEN	Excelência em Microeletrônica do Nordeste

SUMÁRIO

1 INTRODUÇÃO

Com o progresso acelerado da tecnologia, as empresas que desenvolvem novas soluções enfrentam o desafio constante de inovar e criar *hardwares* mais avançados para se manterem competitivas no mercado. Chips mais avançados, em geral, equivalem a chips mais complexos, e isso significa que a probabilidade de falhas no projeto aumenta. Logo, uma verificação ampla e rigorosa é necessária em todo projeto para garantir sua confiabilidade.

Dentre as abordagens de verificação, a verificação funcional dinâmica, ou simulação, consiste em aplicar estímulos ao design para avaliar seu comportamento frente a diferentes condições, com o objetivo de garantir a conformidade com as especificações. A metodologia de Verificação Universal (*Universal Verification Methodology* - UVM) é amplamente utilizada nesse processo, proporcionando uma estrutura padronizada baseada em classes de SystemVerilog, que facilita a criação de ambientes de verificação reutilizáveis. Outra abordagem é a verificação formal, que segue uma análise baseada em regras, utilizando ferramentas especializadas para provar matematicamente que o design se comporta conforme o esperado. Diferente da simulação, a verificação formal não depende de estímulos específicos, mas sim de comportamentos permitidos que o design deve seguir, assegurando que nenhuma combinação de entradas possa violar esses requisitos.

Microprocessadores são dispositivos digitais que executam instruções a partir de palavras binárias, e definimos sua arquitetura pela combinação do conjunto de instruções e a localização dos operandos. A arquitetura RISC-V, baseada no conceito de conjunto reduzido de instruções (*Reduced Instruction Set Computer* - RISC), é um exemplo notável de arquitetura de conjunto de instruções (*Instruction Set Architecture* - ISA). RISC-V é uma das ISAs que mais se destaca na atualidade, principalmente por ser *open-source*. Isso significa que qualquer pessoa ou empresa pode utilizá-la, modificá-la e até mesmo comercializá-la sem a necessidade de pagar por licenças. Essa característica, em si, já é um diferencial que a torna extremamente atraente para empresas de tecnologia, instituições de pesquisa e até *startups* que desejam desenvolver soluções de hardware personalizadas de forma mais acessível. Ao contrário de arquiteturas proprietárias, como x86 ou ARM, que exigem pagamento de licenças e seguem um modelo fechado, o RISC-V possibilita uma personalização completa. Isso fez com que várias metodologias e *know-hows*, que antes só existiam em ambientes fechados na indústria, fossem difundidas na comunidade, habilitando a pesquisa e desenvolvimento de forma muito mais ampla.

Esse trabalho tem intuito de continuar o que foi feito em (MEDEIROS, 2024), o qual foca na utilização de uma ferramenta de verificação formal *open-source*, o *RISC-*

V Formal Verification Framework (RFVF), desenvolvido pela YosysHQ, que também criou a ferramenta de síntese *open-source* Yosys. O RFVF utiliza a ferramenta formal SymbiYosys para validar a conformidade de qualquer *core* RISC-V com as especificações da arquitetura, o que permite a detecção eficiente de falhas no design. Para isso, o RFVF define a interface RVFI, que captura diversas informações importantes sobre o estado processador quando este finaliza a execução de uma instrução. Em (MEDEIROS, 2024), o RFVF também foi expandido para utilizar o Jasper™, a ferramenta de verificação formal da Cadence Design Systems®, o qual oferece diversas vantagens sobre o SymbiYosys. Por exemplo, o Jasper™ possui um visualizador de forma de onda, chamado Visualize, que permite observar quaisquer sinais do modelo, informar a causa para que um sinal obtenha um certo valor em um certo ciclo, adicionar automaticamente as formas de onda dos sinais relevantes para esse valor, informar as linhas de código que contribuem para o *drive* do sinal, e ainda editar a forma de onda interativamente, permitindo explorar facilmente novos cenários. Além disso, podemos criar novas propriedades e prová-las diretamente da interface do aplicativo, podendo em seguida adicioná-las ao nosso código para futuras simulações.

Além disso, foi realizada a verificação parcial de um *core* RISC-V chamado CV32-E40P. Esse *core* foi desenvolvido pela OpenHW, uma organização global sem fins lucrativos que reúne colaboradores focados em criar IPs de *hardware* e *software* com código aberto (OPENHW..., 2024a). O CV32E40P possui código aberto, acessível através de sua página no GitHub (OPENHW..., 2024b). Ele possui arquitetura de 32 bits, implementa o conjunto de instruções RV32IMFC_Zicsr_Zifencei_Zicount, além de conjuntos de instruções customizadas da plataforma, opera em execução em ordem e conta com um *pipeline* de 4 estágios: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execute* (EX) e *Writeback* (WB).

O CV32E40P também incorpora extensões personalizadas da plataforma PULP (*Parallel Ultra-Low-Power*), uma arquitetura voltada para microcontroladores multi-core, direcionada a aplicações de Internet das Coisas (IoT). As principais extensões incluem:

- Operações de Load e Store com pós-incremento: instruções de acesso à memória que automaticamente incrementam o endereço utilizado.
- Laços de hardware: adição de CSRs (*Control and/or Status Registers*) para armazenar o endereço inicial, final e a contagem de loops, juntamente com instruções para manipulá-los.
- Extensões para a Unidade Lógico-Aritmética (ULA): introduz operações matemáticas mais avançadas que as da extensão RV32I, como manipulação de bits, extração de partes de registradores e combinações de sequências comumente utilizadas.

- Operações de multiplicar e acumular: permite multiplicações e acúmulo de resultados, além de multiplicações de meias-palavras (16 bits).
- SIMD (Single Instruction, Multiple Data): executa operações simultâneas em múltiplos elementos, dividindo palavras de dados em blocos de 8 ou 16 bits, operados de forma independente.

No entanto, a versão utilizada do CV32E40P utilizada em (MEDEIROS, 2024) era uma versão antiga, pois foi aquela utilizada em 2021 na versão 7.0.0 do PULPissimo, a arquitetura de microcontroladores mais recente da plataforma PULP (PULPISSIMO, 2022). Em nosso trabalho, nos propomos a substituí-lo por sua versão mais recente disponibilizada na página do GitHub na data de início desse trabalho. Com a versão atualizada, pretendemos realizar novamente os testes já feitos no trabalho anterior, utilizando o Jasper™, e verificar se os *bugs* encontrados anteriormente poderiam ainda ser encontrados na nova versão. Além disso, finalizaremos os itens que ficaram faltando, como por exemplo os modelos de referência para diversas instruções *custom* da plataforma PULP.

Paralelamente às atividades com o RISC-V Formal, o autor desse relatório trabalhou em um projeto no laboratório XMEN, em que participou nas atividades de design de um processador RISC-V, intitulado RISC-X, disponível em sua página do GitHub (RISC-X..., 2024). Também foi responsável por implementar metodologias de verificação como o fluxo do RISCV-DV (RISCV-DV, 2024) para verificação funcional, construção de um *testbench* em UVM e o uso do RFVF.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

O estágio integrado do autor teve como objetivo o estudo e aplicação da ferramenta RISC-V Formal Verification Framework para a verificação formal do *core* CV32E40P. Adicionalmente, foi parte das atividades do estágio participar do desenvolvimento do *core* RISC-X, em especial na área de verificação formal e funcional do design.

1.1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos para o desenvolvimento desse projeto são:

- Adaptar o *driver* da RVFI para acomodar a nova versão do CV32E40P;
- Adaptar os testes das instruções *custom* para a nova versão da extensão;
- Criar testes para as instruções que não foram contempladas no trabalho anterior;

- Realizar os testes visando a verificação do CV32E40P;
- Participar das atividades de design do *core* RISC-X e seu SoC;
- Realizar a verificação formal e funcional do *core* RISC-X.

2 VERIFICAÇÃO DE HARDWARE

2.1 VERIFICAÇÃO FORMAL

Dado que em designs mais complexos o espaço de estados é muito vasto, é inviável testar todas as combinações de estímulos possíveis, e estados difíceis de alcançar podem nunca ser exercitados com estímulos aleatórios. Isso é o que ocorre na simulação, a qual utiliza uma verificação orientada por métricas para definir quais conjuntos de estímulos são necessários para se verificar um design. Ao contrário da simulação, que cobre apenas uma parte limitada do espaço de estados, a verificação formal é exaustiva e pode detectar *bugs* extremamente específicos que poderiam passar despercebidos. A verificação formal transforma o modelo de *hardware* em *Register Transfer Level* (RTL) em uma representação matemática e, em seguida, analisa o design ao aplicar todas as combinações possíveis de entradas para verificar se determinados comportamentos, definidos por propriedades, estão presentes. No entanto, a verificação formal apresenta limitações quando aplicada a sistemas de grande porte, pois a complexidade de sua execução aumenta exponencialmente com o número de registradores no design. Isso ocorre porque as provas formais que dependem de muitos registradores exigem mais tempo e recursos computacionais para serem concluídas.

A técnica de verificação formal é baseada no conceito de propriedades, partindo do princípio de que os circuitos digitais são sincronizados por sinais de *clock*, o que faz com que seus comportamentos evoluam em sincronia com esses pulsos. A ferramenta de verificação formal gera um modelo matemático do design, realiza a síntese lógica e aplica automaticamente estímulos ao modelo, verificando as propriedades com base nas diretivas fornecidas. Além disso, um resultado de uma prova formal só é válido se a propriedade for escrita corretamente, dependendo portanto da capacidade do engenheiro de verificação de escrever boas propriedades.

A complexidade é um fator crítico na verificação formal, pois afeta diretamente o tempo e os recursos computacionais necessários para realizar as análises. Quanto maior a complexidade de uma prova, mais dados precisam ser processados, o que pode levar a um consumo excessivo de memória e a tempos prolongados de execução. Se o tempo limite definido para a análise for atingido, o resultado da prova torna-se indeterminado. Assim, é essencial controlar a complexidade ao configurar o ambiente de verificação, garantindo que o processo seja eficiente dentro dos prazos do projeto.

Um dos principais fatores que influenciam a complexidade é o Cone de Influência (COI), que representa todas as entradas e registradores que podem impactar o resultado

de uma propriedade. Quanto maior o COI de uma propriedade, maior será o esforço computacional necessário para analisá-la. Outro fator é o espaço de estados, que inclui todos os possíveis estados que o design pode assumir. O diâmetro do design, que é o número de ciclos de *clock* necessários para explorar todos os estados atingíveis, também afeta a complexidade.

As propriedades formais geralmente são escritas em SystemVerilog Assertion (SVA), uma sub-linguagem do SystemVerilog. A propriedade deve ser definida com um sinal de *clock*, o qual define o momento em que é feito a checagem. Podemos observar como isso é feito no código 2.1, que demonstra uma propriedade simultânea (*concurrent*). As diretivas simultâneas são executadas paralelamente aos demais blocos da simulação. Uma propriedade sozinha não tem efeito prático; ela precisa ser aplicada com uma diretiva para instruir a ferramenta sobre como usá-la. As principais diretivas são: *assert*, *cover* e *assume*, e seu comportamento varia conforme a verificação é realizada por simulação dinâmica ou análise formal.

```
1 property no_gnt_wo_req;
2     @(posedge clk) disable iff (!rst_n)
3     gnt |-> req;
4 endproperty
5
6 MY_ASSERT: assert property (no_gnt_wo_req);
```

Código 2.1 – Exemplo de propriedade em SVA.

A diretiva *assert* gera uma assertiva, que na simulação verifica se o design respeita a propriedade durante cada ciclo. Se violada, gera um erro. Na análise formal, tenta-se provar que a assertiva é sempre verdadeira, e qualquer falha gera um contra-exemplo, geralmente mostrado como uma forma de onda. A diretiva *cover* busca observar se uma sequência específica ocorre no design. Na simulação, ela conta quantas vezes essa sequência é observada, enquanto na análise formal, a ferramenta tenta gerar a condição para que a sequência ocorra. Já a diretiva *assume* define uma propriedade que é assumida como verdadeira, ideal para simplificar a análise formal ao restringir o espaço de estados. Na simulação, no entanto, é entendida da mesma forma que uma assertiva.

O RISC-V Formal utiliza uma forma mais básica das propriedades, denominadas propriedades imediatas. Esse tipo de propriedade é executado proceduralmente pela ferramenta e podem conter apenas expressões booleanas. Podemos observar um exemplo no código 2.2, o qual realiza a mesma verificação que o observado no código anterior. O RFVF é descreve a ISA RISC-V baseado em propriedades imediatas para garantir a maior compatibilidade possível com ferramentas.

```
always @(posedge clk) begin
    if (gnt) begin
        assert (req);
    end
end
```

Código 2.2 – Exemplo de propriedade em SVA

2.2 UNIVERSAL VERIFICATION METHODOLOGY

A Metodologia Universal de Verificação (UVM) é um padrão amplamente adotado na indústria de semicondutores para a verificação funcional de *hardware*. O principal objetivo da UVM é garantir que o design funcione conforme esperado, detectando falhas potenciais durante as fases de desenvolvimento. Ela fornece uma estrutura robusta para desenvolver *testbenches* modulares e reutilizáveis, facilitando a criação de ambientes de verificação que simulam o comportamento do design sob teste (*Design Under Test* - DUT) em diferentes condições operacionais. A UVM organiza a verificação em um conjunto hierárquico de componentes que incluem *drivers*, *monitors*, *sequencers*, entre outros. Esses blocos permitem a simulação de estímulos no DUT e a verificação das respostas geradas em comparação com os resultados esperados.

Um dos principais benefícios da UVM é a reutilização de componentes de verificação. Uma vez que os módulos são construídos de maneira genérica e padronizada, eles podem ser usados em diferentes projetos ou até mesmo em diferentes partes do mesmo projeto. Isso reduz significativamente o esforço de desenvolvimento e manutenção de ambientes de teste. Outro ponto fundamental é o uso de classes de transações e sequências de estímulos. A UVM permite que diferentes tipos de transações sejam simulados em ordem aleatória ou que sejam feitos testes dirigidos, garantindo cobertura ampla dos cenários de operação. Além disso, ferramentas de cobertura funcional podem ser integradas para monitorar se sequências relevantes do design foram devidamente exercitadas. Geralmente um componente de verificação UVM possui um modelo de referência que realiza automaticamente a checagem dos resultados, podendo, além disso, incluir *assertions* para auxiliar na verificação.

Em UVM, a hierarquia de classes é estruturada para modularizar o ambiente de verificação, facilitando sua criação, manutenção e reutilização. Cada classe tem funções específicas que, em conjunto, formam um ambiente de verificação completo. Falaremos um pouco sobre as principais classes e suas funções na hierarquia UVM.

- `uvm_driver`: O `uvm_driver` recebe transações do sequenciador e as converte em sinais de baixo nível, ou seja, nível de barramento, para o DUT. A principal função do *driver* é enviar esses sinais para as interfaces de entrada do DUT de forma

sincronizada com o *clock* e outras condições temporais. É o responsável por aplicar estímulos reais ao design.

- `uvm_sequencer`: O `uvm_sequencer` coordena a geração e o envio de transações (sequências de estímulos) para o *driver*. Ele pode gerar estímulos de maneira determinística ou aleatória, permitindo uma cobertura ampla de cenários.

- `uvm_monitor`: O `uvm_monitor` observa os sinais entre o DUT e o ambiente de verificação, capturando transações. Ele atua de forma passiva, sem interferir no DUT, e envia as informações monitoradas para outras partes do ambiente, como o *scoreboard* ou um modelo de cobertura.

- `uvm_sequence`: As sequências representam conjuntos de transações que serão enviadas ao DUT. Elas são derivadas da classe `uvm_sequence` e definem a ordem, frequência e condições em que essas transações ocorrem. Sequências podem ser configuradas para cobrir diferentes casos de uso do DUT, com possibilidade de gerar dados aleatórios ou dirigidos. Sequências complexas podem ser construídas a partir de sequências simples.

- `uvm_agent`: A classe `uvm_agent` é responsável por encapsular os componentes que geram e monitoram os sinais do DUT. Um `uvm_agent` geralmente inclui um *driver*, sequenciador e monitor. Ele pode operar em dois modos: no modo ativo, gera estímulos para o DUT (via *driver*); no modo passivo, apenas monitora as interações no barramento (via *monitor*).

- `uvm_scoreboard`: O `uvm_scoreboard` é a classe responsável por comparar os resultados gerados pelo DUT com os resultados esperados. Ele atua como um modelo de referência ou comparador, garantindo que a saída do DUT esteja correta em relação aos estímulos enviados. Além disso, ele é responsável por coletar métricas a partir das transações que recebe.

- `uvm_env`: A classe `uvm_env`, que vem de *environment* ou ambiente em português, encapsula um conjunto de componentes de verificação, como agentes, sequenciadores e *scoreboards*. Ela organiza a estrutura geral do ambiente de verificação, sendo o nível superior que conecta diferentes partes do sistema de verificação. Um ambiente UVM pode conter múltiplos agentes e componentes.

- `uvm_test`: A classe `uvm_test` define o ponto de entrada principal para a execução do *testbench*, correspondendo ao componente topo na hierarquia. Cada teste herda de `uvm_test` e pode configurar o ambiente de verificação, definir os parâmetros de execução, como quais sequências usar e como conduzir a verificação. Diferentes instâncias de `uvm_test` são usadas para cobrir diferentes objetivos de verificação.

Em um ambiente típico de UVM, as sequências são criadas no sequenciador e enviadas ao *driver*, que aplica os estímulos ao DUT. O monitor observa as transações no barramento e as envia para o *scoreboard*, onde os resultados são comparados com o modelo de referência, que pode ser um componente UVM ou uma função externa. O ambiente é configurado e gerido pelo `uvm_env`, que orquestra os componentes. Durante a execução, a verificação é controlada por um `uvm_test`, que inicializa e coordena todas as fases de simulação. Essa estrutura é genérica o suficiente para possibilitar a construção de ambientes reutilizáveis até mesmo para designs mais complexos.

Falando sobre as fases de simulação, o UVM utiliza um sistema de fases que segue um ciclo de vida bem definido para garantir que todos os componentes do ambiente de verificação sejam criados, configurados, conectados, executados e, por fim, finalizados corretamente. Essas fases organizam o fluxo de execução de um *testbench*, permitindo uma verificação ordenada e previsível. Cada fase desempenha um papel específico, desde a criação dos componentes até o término da simulação. As fases são divididas em três grupos principais: fases de construção, fases de execução e fases de limpeza. Além das fases padrões, o UVM permite que os usuários criem suas próprias fases personalizadas para atender a requisitos específicos de verificação.

As fases de construção são responsáveis pela criação e configuração dos componentes do *testbench*. O objetivo é construir a hierarquia completa de componentes UVM antes que a simulação comece. Listamos essas fases a seguir.

- `build_phase`: A fase de construção é a primeira fase na hierarquia UVM e é usada para instanciar todos os componentes do ambiente de verificação.
- `connect_phase`: Após a criação dos componentes, a fase de conexão estabelece as conexões entre os diferentes módulos, como associar portas TLM entre *sequencers* e *drivers*, por exemplo.
- `end_of_elaboration_phase`: Nesta fase, ajustes finais são feitos na estrutura de componentes e suas configurações. Ela é utilizada para realizar verificações ou ajustes antes de a simulação começar.
- `start_of_simulation_phase`: Marca o início da preparação para a execução do teste. Mensagens de status podem ser impressas ou logs configurados, mas nenhuma transação é enviada ao DUT ainda. O ambiente já está totalmente configurado e pronto para iniciar a execução dos testes.

As fases de execução são onde a simulação realmente acontece. Nessa fase, os estímulos são aplicados ao DUT e os resultados são verificados. São as únicas fases que podem consumir tempo de execução, pois essas são definidas como *tasks*, enquanto as outras são *functions*.

- `run_phase`: Esta é a fase principal da simulação. Aqui, sequências são iniciadas e estímulos são enviados ao DUT. A fase de *run* consome tempo de simulação e pode iniciar vários processos paralelos. Nessa fase, as interações entre o *testbench* e o DUT ocorrem, e as respostas são monitoradas. Paralelamente à `run_phase`, o UVM define quatro sub-fases que facilitam o controle granular das operações de simulação, e falaremos delas abaixo.

- `reset_phase`: durante essa fase, o DUT é colocado em seu estado de *reset* para assegurar que a simulação comece de uma condição consistente e previsível.

- `configure_phase`: nesta sub-fase, os parâmetros de configuração podem ser aplicados ao DUT para, por exemplo, realizar a inicialização de registradores ou a configuração de módulos internos.

- `main_phase`: é onde os estímulos principais são gerados e aplicados ao DUT, e as respostas são coletadas para verificação.

- `shutdown_phase`: a sub-fase de *shutdown* é usada para finalizar as operações do DUT e do *testbench* de maneira ordenada, garantindo que todas as tarefas de simulação sejam corretamente concluídas.

As fases de limpeza acontecem após a execução da simulação, e são responsáveis por concluir o processo de verificação, gerar os resultados finais e fazer o encerramento completo da simulação.

- `extract_phase`: A fase de extração é usada para extrair dados gerados durante a fase de execução e preparar relatórios, como a coleta de dados de cobertura ou contagens de falhas.

- `check_phase`: Aqui, o ambiente faz todas as verificações finais dos resultados, como a comparação dos dados recebidos com os resultados esperados, se isso não for feito em tempo real durante a `run_phase`.

- `report_phase`: Esta fase prepara e imprime relatórios de status final da simulação, incluindo resumos de cobertura funcional, logs de erros, alertas e falhas.

- `final_phase`: A fase de final é onde as tarefas de encerramento são executadas. É o ponto final de toda a verificação, garantindo que a simulação seja terminada corretamente e que todos os recursos sejam liberados.

Em UVM, a comunicação entre os diversos componentes do *testbench* é feita principalmente por meio de *Transaction-Level Modeling* (TLM), um conceito de modelagem que abstrai os detalhes da troca de sinais físicos entre os módulos e foca na troca de

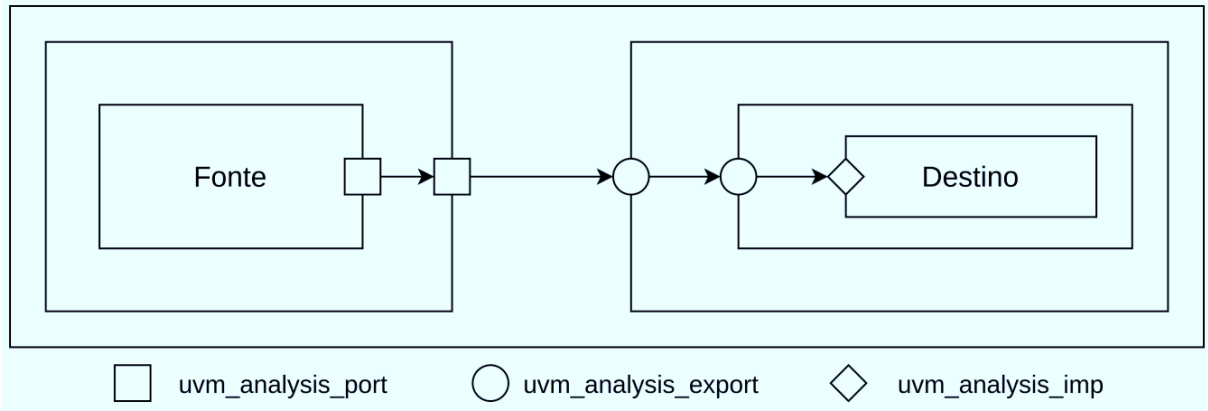
transações de alto nível. Uma transação é um objeto que encapsula uma operação ou um pacote de dados, como a leitura ou escrita de um barramento ou a transferência de dados entre módulos. A troca dessas transações ocorre através de portas TLM, que conectam os componentes. O uso de TLM facilita a criação de ambientes mais modulares, onde diferentes componentes podem ser substituídos sem a necessidade de modificar a lógica de comunicação entre eles.

A comunicação entre objetos UVM utilizando TLM envolve três elementos principais: *ports*, *exports* e *imps*. As *ports* são usadas para enviar ou requisitar transações. Quando um componente deseja iniciar uma comunicação, ele utiliza uma *port* para emitir o chamado de uma função para outro componente. As *imps* (implementações) são onde a lógica associada à conexão TLM reside. Elas implementam a função da *port* correspondente e a usa para processar a transação. Por exemplo, um *driver* recebe uma transação do *sequencer* por meio de uma conexão TLM. O *driver* possui a *port*, pois é ele quem inicia a chamada da função. O *sequencer* possui a *imp*, pois é ele quem define a função iniciada pela *port* do *driver*. Os *exports*. por sua vez, são usados para enviar transações através de diferentes componentes da hierarquia. Eles conectam uma *port* a uma *imp*.

As funções de troca de transações que o TLM define podem ser do tipo *get*, quando o iniciador busca uma transação do alvo, ou do tipo *put*, quando o iniciador envia uma transação para o alvo. *Get ports* e *put ports* devem ser obrigatoriamente conectadas a uma *get imp* ou *put imp*, respectivamente. Essas comunicações, adicionalmente, podem ser definidas como bloqueantes ou não-bloqueantes. As bloqueantes só retornam após a transação ser processada, ou seja, podem consumir tempo de simulação (*tasks*). Já as não-bloqueantes são processadas instantaneamente, independente do alvo poder processar a transação ou não, retornando um bit indicativo de sucesso ou falha (*function*).

O UVM define ainda as *analysis ports*, utilizadas para transmitir transações para múltiplos receptores, ou mesmo para nenhum receptor, diferente das *get ports* e *put ports* em que a comunicação é de 1 componente para 1 componente. Por exemplo, um monitor pode usar uma *analysis port* para enviar transações capturadas do barramento para vários receptores, como um *scoreboard* e um modelo de cobertura. Um exemplo de comunicação usando TLM e *analysis ports* é apresentado na figura 1.

Figura 1 – Conexão de componentes usando analysis ports.

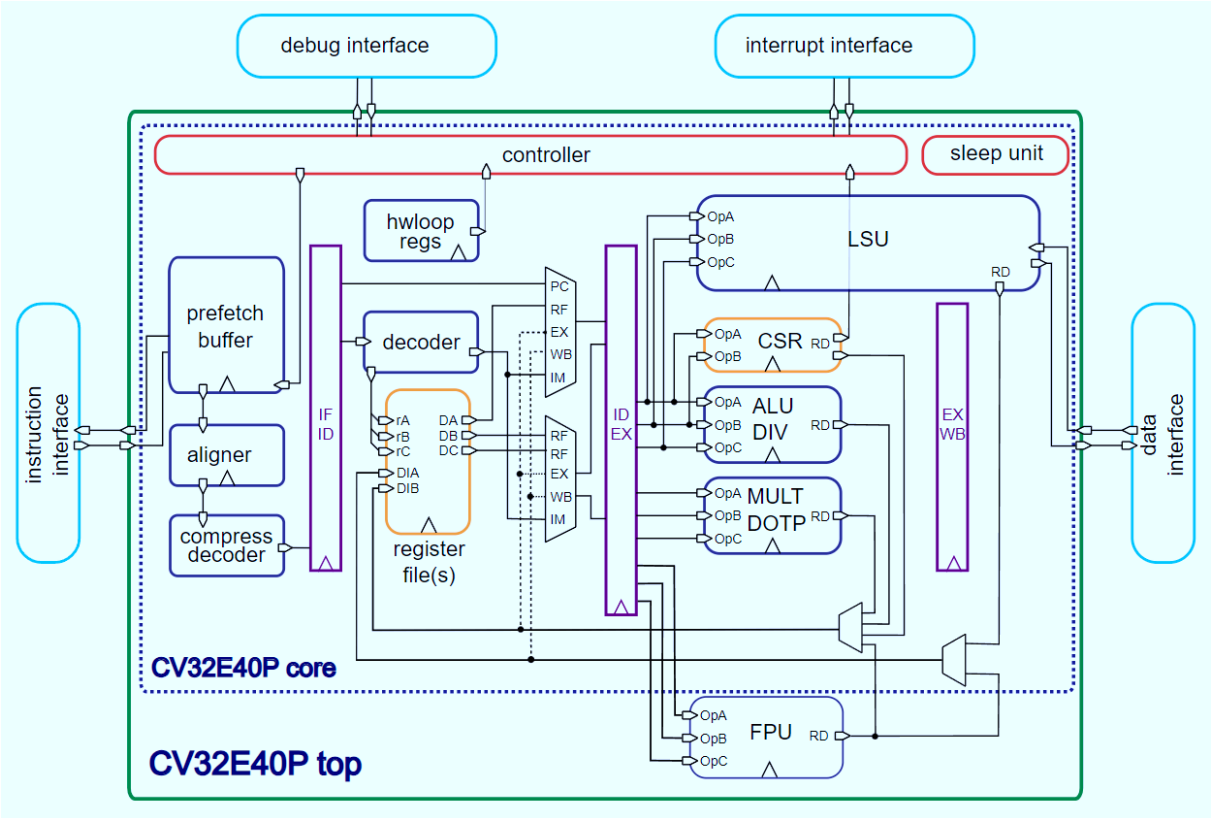


Fonte: (VALE, 2021).

3 VERIFICAÇÃO DO CV32E40P

O núcleo de processamento CV32E40P é um processador de código aberto, mantido pelo grupo OpenHW. Ele é um núcleo com arquitetura de 32 bits, execução em ordem e 4 estágios de *pipeline*: *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execute* (EX) e *Writeback* (WB). Uma ilustração de sua arquitetura é apresentada na figura 2. Podemos observar, a partir dos registradores com a caixa na cor roxa, a divisão entre os estágios do *pipeline* do *core*.

Figura 2 – Hierarquia do CV32E40P.



Fonte: (OPENHW..., 2024b).

Como mencionado, utilizamos a versão mais nova do CV32E40P na data de início desse trabalho, o que corresponde à versão 1.8.2. Versões mais novas já foram publicadas em sua página do GitHub após nossas atividades. Iremos discorrer brevemente sobre algumas diferenças que o autor julga importante (para o escopo desse trabalho) de serem comentadas acerca das versões utilizadas nesse trabalho e no trabalho anterior.

A interface que o CV32E40P tem com a memória é uma implementação simplificada do *Open Bus Interface* (OBI), uma interface criada pela plataforma PULP. O *core* tem duas cópias do OBI, uma para acessar a memória de instruções e outra a de dados.

Na versão antiga do *core*, só era possível ter uma transação em voo por vez em cada interface. Ou seja, após, por exemplo, uma requisição de leitura à memória o design esperaria até receber o sinal de resposta válida da memória para poder enviar uma nova transação. Nas versões mais recentes, é utilizado um contador de transações ativas para permitir que sejam feitos múltiplos acessos à memória de forma mais rápida. Atualmente, são permitidas duas transações simultâneas no barramento utilizado pelo *pre-fetch buffer* e duas no barramento da *load/store unit* (OPENHW..., 2024c).

Agora é possível desabilitar as instruções não oficiais. Um dos desafios encontrados no trabalho anterior foi construir um *driver* coerente para a RVFI, capaz de identificar todas as instruções que são executadas pelo processador e as informações relacionadas a ela. Um dos motivos para essa dificuldade foi a presença das instruções não oficiais da plataforma PULP, um conjunto de instruções denominado Xpulp, o que aumentava a complexidade do design. Nas novas versões, o conjunto foi renomeado para Xcorev, a codificação das instruções foi refeita e foram adicionadas instruções para operações com números complexos, além de que a documentação está bem mais clara e concisa. No RTL, foi criado um parâmetro para habilitar ou desabilitar o Xcorev, tornando opcional a sua implementação e facilitando o esforço inicial de criar um *driver* para a RVFI. Iniciamos nosso desenvolvimento com esse parâmetro desativado, e, após finalizar o *driver* para o conjunto básico de instruções, habilitamos esse parâmetro para continuar as atividades. O OpenHW também define um conjunto Xpulpcluster, que deve ser habilitado quando o processador for utilizado em *clusters*. O conjunto define uma única instrução, utilizada para sincronizar a atividade dos *cores* no *cluster*, e é habilitado opcionalmente por um parâmetro no RTL. As extensões oficiais F e Zfinx também são habilitadas por parâmetros. Em nossas atividades, utilizamos o core com o Xcorev habilitado, e Xpulpcluster, F e Zfinx desabilitados.

Uma das extensões mais problemáticas do Xcorev para o nosso trabalho foi a extensão dos laços de *hardware*, ou *hardware loops*. Sua presença tornou inviável realizar o *drive* correto do contador de programa (*Program Counter* - PC) na RVFI, fazendo com que precisássemos desabilitar essas instruções através de *assumes*, o que elevou drasticamente o tempo de finalização das provas. O funcionamento dos *hardware loops* ficou bem mais claro com a nova documentação (OPENHW..., 2024c), que lista uma série de regras que devem ser seguidas para utilizá-los. No entanto, o processador não levanta exceções para o não seguimento dessas regras, pois os desenvolvedores decidiram que não valia a pena dedicar recursos para detectar situações que nunca devem acontecer. Por esse motivo, cabe ao ambiente de verificação realizar essas restrições, o que na verificação formal é feito com os *assumes*. No entanto, não conseguimos criar os *assumes* necessários. Fizemos, então, algo que não se deve fazer na verificação: copiamos o RTL. Assim, o sinal que indica o *loop* na RVFI estaria sempre "correto" (na verdade ele é 1 sempre que o *core* realiza o salto devido a um *loop*) e não precisaríamos mais do *assume* para desabilitar as

instruções, o que estava deixando as provas demoradas.

Outra atualização que simplificou o trabalho de desenvolver a RVFI foi que o *core* não possui mais um modo de segurança (apesar de que não é excluída a possibilidade de ser adicionado como uma extensão no futuro). O modo seguro que estava presente na versão original contava com o modo de privilégio de usuário, U-mode, tendo em mente que o processador RISC-V mais básico possível tem apenas o modo de privilégio de máquina, M-mode, implementado. E havia também um módulo de *Physical Memory Protection* (PMP), o qual permite definir regiões da memória física, cada região com suas próprias permissões de acesso (escrita, leitura e execução). Como o funcionamento desses itens não era claro para o autor, isso elevou o nível de dificuldade.

O trabalho anterior deixou alguns itens propostos como atividades futuras, o que nosso trabalho se propõe a contemplar. Listaremos esses itens a seguir.

1. Implementação imperfeita da RVFI. Em certos casos era possível identificar que instruções não chegavam ao final do *pipeline* da RVFI, mesmo sendo instruções executadas normalmente pelo processador. Quando temos "buracos" no meio do programa, é de se esperar que outros testes falhem, como, por exemplo, quando testamos se o PC da instrução seguinte é o mesmo que a instrução anterior informa. Para a nova versão do CV32E40P, não identificamos nenhuma situação como essa, logo podemos considerar que o *drive* da RVFI é feito de forma correta. Com isso, foi possível avançar nos testes de consistência do processador, os quais não era possível realizar com o *drive* imperfeito da RVFI.

2. No trabalho anterior, não foram realizados testes para instruções de multiplicação e divisão. Isso se deve ao fato de que operações matemáticas complexas, como multiplicação e divisão, são atividades muito difíceis para a análise formal, o que levava os testes a demorarem bastante (mais de 48h para uma única instrução). Para permitir o teste dos outros aspectos dessas instruções, excluindo apenas o resultado da operação, o RISC-V Formal permite a implementação de operações aritméticas alternativas. Para isso, o resultado da operação deve ser substituído de acordo com a tabela mostrada na figura 3. Na primeira coluna, temos a operação original; na segunda coluna, a operação que irá substituí-la; na terceira coluna, a máscara que será aplicada, através de uma operação XOR. Implementamos as operações alternativas no *core*, e um exemplo pode ser observado na figura 4, em que é mostrada a atribuição do valor do resultado da ULA nos casos de divisões. Podemos observar que, quando não está definido o macro `RISCV_FORMAL_ALTOPS`, o resultado é igual ao resultado da unidade de divisão. Caso contrário, ou seja, quando estamos usando as operações alternativas, o resultado é igual à subtração dos operandos, seguido pela aplicação da respectiva máscara.

Figura 3 – Tabela de operações alternativas.

Integer Multiply/Divide Instructions

Operation	Add/Sub	Bitmask
MUL	Add	0x2cdf52a55876063e
MULH	Add	0x15d01651f6583fb7
MULHSU	Sub	0xea3969edecfbe137
MULHU	Add	0xd13db50d949ce5e8
DIV	Sub	0x29bbf66f7f8529ec
DIVU	Sub	0x8c629acb10e8fd70
REM	Sub	0xf5b7d8538da68fa5
REMU	Sub	0xbc4402413138d0e1

Fonte: (RISC-V. . . , 2024a).

Figura 4 – Operações alternativas para instruções de divisão.

```
983         // Division Unit Commands
984     `ifndef RISCV_FORMAL_ALTOPS
985         ALU_DIV, ALU_DIVU, ALU_REM, ALU_REMU: result_o = result_div;
986     `else
987         ALU_DIV : result_o = ($signed(operand_b_i) - $signed(operand_a_i)) ^ altop_mask_div;
988         ALU_DIVU: result_o = ($signed(operand_b_i) - $signed(operand_a_i)) ^ altop_mask_divu;
989         ALU_REM : result_o = ($signed(operand_b_i) - $signed(operand_a_i)) ^ altop_mask_rem;
990         ALU_REMU: result_o = ($signed(operand_b_i) - $signed(operand_a_i)) ^ altop_mask_remu;
991     `endif
```

Fonte: autoria própria.

3. Testes para as instruções não-oficiais. Por falta de tempo hábil, o trabalho anterior não escreveu os testes para todas as instruções do Xpulp (agora Xcorev). Como o Xcorev refez a codificação das instruções, além de alterar certas condições e/ou restrições, foi necessário refazer os testes que já haviam sido feitos, além de concluir o restante. Nesse quesito, concluímos os testes para o conjunto completo das instruções Xcorev, incluindo implementação de operações alternativas para operações com multiplicação, por exemplo nas instruções de *multiply-accumulate* ou em multiplicações de números complexos.

Algo importante que foi necessário fazer na RVFI para detectar todas as instruções executadas pelo *core* foi que tivemos que fazer um tratamento especial para os *branches*. *Branches* são saltos condicionais, ou seja, pode ocorrer um salto ou não, dependendo se uma condição for atendida ou não. No CV32E40P, essa decisão é feita no estágio EX do *pipeline*, e a instrução é finalizada nele, sem passar para o estágio WB, já que nessas instruções não é feita nenhuma escrita em registrador. Isso tem uma interação

interessante quando o estágio WB está em *stall*. Um *stall* em um estágio do *pipeline* significa que o estado arquitetural desse estágio não deve mudar. O estágio WB pode estar em *stall* quando uma instrução de *load* ou *store* está esperando por uma resposta válida da memória. O normal de se esperar quando o WB, ou qualquer outro estágio, está em *stall* é que os anteriores também sejam paralisados, uma vez que uma instrução não pode avançar para um estágio que já está em uso. E isso de fato ocorre, para todas instruções exceto os *branches*, que são as únicas instruções que são finalizadas sem passar pelo WB. Isso implica que, quando o WB está esperando por uma resposta no barramento da memória, podemos ter um número indefinido de *branches* sendo executados nos estágios anteriores. Isso não altera o estado arquitetural do nosso processador, exceto pelo valor do PC.

No ponto de vista da RVFI, inicialmente estávamos observando apenas as instruções no final do *pipeline*. Com a situação descrita anteriormente, vimos que é necessário também observar as instruções no EX. Portanto, tivemos que criar uma maneira de detectar quando um *branch* é finalizado no EX enquanto o WB está em *stall*, e então fazer um *bypass* para que essas instruções passassem para frente na RVFI, permitindo assim detectar a instrução, ou, mais especificamente, a alteração que ela faz no PC. Detectamos essa peculiaridade com o teste de consistência do PC, que, como mencionado, é o único valor do estado arquitetural que é modificado nessas situações.

Para a interface OBI das memórias de instruções e dados, temos os *assumes* ilustrados na figura 5. Podemos observar que o número de transações no barramento é limitado a 2, como especificado na documentação. Além disso, não permitimos que haja um `rvalid` sem que haja transações no barramento, já que esse sinal indica uma resposta válida.

Além disso, também fizemos *assumes* para serem utilizados apenas nos testes `reg` e `pc_fwd`. O teste `reg` realiza a verificação da consistência entre escritas e leituras nos registradores. Por exemplo, se uma instrução escreve 0x123 no registrador x4, a instrução seguinte que ler o registrador x4 deve ler o valor 0x123. Isso é útil para testar, por exemplo, se *forwardings* são realizados de forma correta. Foi necessário criar um *assume* para desabilitar as instruções de *load* e *store* com pós-incremento. Isso se deve a dois motivos. Primeiro: operações com pós-incremento realizam a escrita do incremento no estágio EX, e, no caso de uma instrução de *load*, uma segunda escrita no WB. Do ponto de vista da RVFI, os registradores são escritos apenas quando a instrução é finalizada, no WB. Imaginemos que, em decorrência de um *post-incrementing load*, o pós-incremento ocorre no registrador x4 (escrito no EX, mas a RVFI só está ciente da escrita no WB), e depois a instrução passa alguns ciclos esperando o acesso à memória ser finalizado. Uma instrução posterior que lê o registrador x4 não precisa esperar que o acesso à memória seja finalizado no WB para poder entrar no EX. Segundo: como vimos anteriormente,

Figura 5 – Assumes para interfaces de instruções e dados.

```
151     // No valid response without pending transaction
152     reg [1:0] instr_trans_pnd;
153     reg [1:0] data_trans_pnd ;
154     wire instr_cnt_up = instr_gnt_i && instr_req_o;
155     wire instr_cnt_dn = instr_rvalid_i;
156     wire data_cnt_up  = data_gnt_i  && data_req_o ;
157     wire data_cnt_dn  = data_rvalid_i ;
158
159     always @(posedge clock or posedge reset) begin
160         if (reset) begin
161             instr_trans_pnd <= 2'b0;
162             data_trans_pnd  <= 2'b0;
163         end else begin
164             unique case ({instr_cnt_up, instr_cnt_dn})
165                 2'b00: instr_trans_pnd <= instr_trans_pnd;
166                 2'b01: instr_trans_pnd <= instr_trans_pnd - 2'b1;
167                 2'b10: instr_trans_pnd <= instr_trans_pnd + 2'b1;
168                 2'b11: instr_trans_pnd <= instr_trans_pnd;
169             endcase
170
171             unique case ({data_cnt_up, data_cnt_dn})
172                 2'b00: data_trans_pnd <= data_trans_pnd;
173                 2'b01: data_trans_pnd <= data_trans_pnd - 2'b1;
174                 2'b10: data_trans_pnd <= data_trans_pnd + 2'b1;
175                 2'b11: data_trans_pnd <= data_trans_pnd;
176             endcase
177
178             ASM_max_2_pnd_trans_instr: assume (instr_trans_pnd != 2'b11);
179             ASM_max_2_pnd_trans_data : assume (data_trans_pnd  != 2'b11);
180             if (instr_trans_pnd == 2'b0)
181                 ASM_no_rvld_wo_pnd_instr: assume (!instr_rvalid_i);
182             if (data_trans_pnd  == 2'b0)
183                 ASM_no_rvld_wo_pnd_data : assume (!data_rvalid_i );
184         end
185     end
```

Fonte: autoria própria.

instruções de *branch* finalizam no estágio EX independente do estágio de WB estar em *stall* ou não. Retornando ao exemplo anterior, vamos supor que uma instrução de *branch* é aquela que utiliza o registrador x4 como operando após o *load*. Como *branches* podem terminar no EX, essa instrução passa na frente do *post-incrementing load* que está parado no WB, mas a RVFI ainda não sabe que o registrador x4 foi escrito pelo pós-incremento, já que o *load* não foi concluído. Como o valor lido em x4 ainda é desconhecido pelo *testbench*, o teste `reg` indica uma falsa falha. Para corrigir esse problema, poderíamos pensar em algum mecanismo para que os *post-incrementing loads* e *stores* possam ser divididos em 2 ciclos, como se fossem realizadas duas instruções, uma finalizada no estágio EX, escrevendo o valor do incremento, e outra no WB com as informações do acesso à memória. Para isso, também teríamos que alterar a nossa implementação da RVFI para depositar 2 instruções por ciclo. Até o momento, não foi possível implementar essas

funcionalidades.

Adicionalmente, foi necessário criar um *assume* para desabilitar os *hardware loops* no teste `reg`. Isso se deve muito provavelmente porque essas instruções exigem uma série de restrições, as quais não fomos capazes de implementar. Os *assumes* definidos para o teste `reg` são mostrados na figura 6.

Figura 6 – Assumes para o teste reg.

```
212 `ifdef CV32P_REG_CHECK
213     // Post-Increment Register-Immediate Load
214     parameter INSTR_CVLBI = {17'b?, 3'b000, 5'b?, OPCODE_CUSTOM_0};
215     parameter INSTR_CVLBUI = {17'b?, 3'b100, 5'b?, OPCODE_CUSTOM_0};
216     parameter INSTR_CVLHI = {17'b?, 3'b001, 5'b?, OPCODE_CUSTOM_0};
217     parameter INSTR_CVLHUI = {17'b?, 3'b101, 5'b?, OPCODE_CUSTOM_0};
218     parameter INSTR_CVLWI = {17'b?, 3'b010, 5'b?, OPCODE_CUSTOM_0};
219
220     // Post-Increment Register-Register Load
221     parameter INSTR_CVRRPOSTL = {7'b000?0??, 5'b?, 5'b?, 3'b011, 5'b?, OPCODE_CUSTOM_1};
222
223     wire postincr_l = `INSTR inside {INSTR_CVLBI, INSTR_CVLBUI, INSTR_CVLHI, INSTR_CVLHUI, INSTR_CVLWI, INSTR_CVRRPOSTL};
224     ASM_no_post_load: assume property (!postincr_l);
225
226     // Post-Increment Register-Immediate Store
227     parameter INSTR_CVSBI = {17'b?, 3'b000, 5'b?, OPCODE_CUSTOM_1};
228     parameter INSTR_CVSHI = {17'b?, 3'b001, 5'b?, OPCODE_CUSTOM_1};
229     parameter INSTR_CVSWI = {17'b?, 3'b010, 5'b?, OPCODE_CUSTOM_1};
230
231     // Post-Increment Register-Register Store operations encoding
232     parameter INSTR_CVRRPOSTS = {7'b00100??, 5'b?, 5'b?, 3'b011, 5'b?, OPCODE_CUSTOM_1};
233
234     wire postincr_s = `INSTR inside {INSTR_CVSBI, INSTR_CVSHI, INSTR_CVSWI, INSTR_CVRRPOSTS};
235     ASM_no_post_store: assume property (!postincr_s);
236
237     parameter INSTR_CVSTARTI = {12'b?, 5'b00000, 3'b100, 4'b0000, 1'b?, OPCODE_CUSTOM_1};
238     parameter INSTR_CVSTART = {12'b000000000000, 5'b?, 3'b100, 4'b0001, 1'b?, OPCODE_CUSTOM_1};
239     parameter INSTR_CVENDI = {12'b?, 5'b00000, 3'b100, 4'b0010, 1'b?, OPCODE_CUSTOM_1};
240     parameter INSTR_CVEND = {12'b000000000000, 5'b?, 3'b100, 4'b0011, 1'b?, OPCODE_CUSTOM_1};
241     parameter INSTR_CVCOUNTI = {12'b?, 5'b00000, 3'b100, 4'b0100, 1'b?, OPCODE_CUSTOM_1};
242     parameter INSTR_CVCOUNT = {12'b000000000000, 5'b?, 3'b100, 4'b0101, 1'b?, OPCODE_CUSTOM_1};
243     parameter INSTR_CVSETUPI = {17'b?, 3'b100, 4'b0110, 1'b?, OPCODE_CUSTOM_1};
244     parameter INSTR_CVSETUP = {12'b?, 5'b?, 3'b100, 4'b0111, 1'b?, OPCODE_CUSTOM_1};
245
246     wire hwlp = `INSTR inside {INSTR_CVSTARTI, INSTR_CVSTART, INSTR_CVENDI, INSTR_CVEND,
247                                INSTR_CVCOUNTI, INSTR_CVCOUNT, INSTR_CVSETUPI, INSTR_CVSETUP};
248     ASM_no_hwlp: assume property (!hwlp);
249 `endif
```

Fonte: autoria própria.

O teste `pc_fwd` realiza a verificação da consistência entre o PC de uma instrução e o PC da instrução seguinte. Na maior parte das instruções, sabemos que o próximo PC, chamado `pc_wdata` na RVFI, deve ser o PC + 4, dado que uma palavra de instrução tem 4 bytes, ou PC + 2 no caso de instruções de 2 bytes. Em caso de *jumps* ou *branches*, o PC pode mudar. E, no caso do CV32E40P, temos ainda os *hardware loops* que podem alterar o PC. Como mencionamos anteriormente, para sinalizar a ocorrência de *hardware loops*, nós copiamos o que o *core* fazia. Ainda assim, o teste do PC falhou, muito provavelmente porque essas instruções exigem uma série de restrições. Com isso, decidimos aplicar um *assume* para desabilitá-las, o que podemos observar nas linhas 198 a 209 da figura 7 (comentadas). O curioso foi que, mesmo desabilitando essas instruções, foi possível que o *hardware* iniciasse um *loop*. Falaremos disso a seguir, e, por esse motivo, comentamos as linhas citadas. Desse modo, foi necessário usar *assumes* mais agressivos, mostrados nas linhas 195 e 196 da figura 7, os quais impedem que o contador dos *loops* sejam diferentes

de zero. Em questão de performance, o teste `pc_fwd` levou cerca de 17 horas e 43 minutos para concluir.

Figura 7 – Assumes para o teste pc_fwd.

```
194  `ifdef CV32P_PC_FWD
195      ASM_no_hwlp_0: assume property (core_top_i.core_i.hwlp_cnt[0] == '0);
196      ASM_no_hwlp_1: assume property (core_top_i.core_i.hwlp_cnt[1] == '0);
197
198      // parameter INSTR_CVSTARTI = {12'b?, 5'b00000, 3'b100, 4'b0000, 1'b?, OPCODE_CUSTOM_1};
199      // parameter INSTR_CVSTART = {12'b000000000000, 5'b?, 3'b100, 4'b0001, 1'b?, OPCODE_CUSTOM_1};
200      // parameter INSTR_CVENDI = {12'b?, 5'b00000, 3'b100, 4'b0010, 1'b?, OPCODE_CUSTOM_1};
201      // parameter INSTR_CVEND = {12'b000000000000, 5'b?, 3'b100, 4'b0011, 1'b?, OPCODE_CUSTOM_1};
202      // parameter INSTR_CVCOUNTI = {12'b?, 5'b00000, 3'b100, 4'b0100, 1'b?, OPCODE_CUSTOM_1};
203      // parameter INSTR_CVCOUNT = {12'b000000000000, 5'b?, 3'b100, 4'b0101, 1'b?, OPCODE_CUSTOM_1};
204      // parameter INSTR_CVSETUPI = {17'b?, 3'b100, 4'b0110, 1'b?, OPCODE_CUSTOM_1};
205      // parameter INSTR_CVSETUP = {12'b?, 5'b?, 3'b100, 4'b0111, 1'b?, OPCODE_CUSTOM_1};
206
207      // wire hwlp = `INSTR inside {INSTR_CVSTARTI, INSTR_CVSTART, INSTR_CVENDI, INSTR_CVEND,
208      //                            INSTR_CVCOUNTI, INSTR_CVCOUNT, INSTR_CVSETUPI, INSTR_CVSETUP};
209      // ASM_no_hwlp: assume property (!hwlp);
210  `endif
```

Fonte: autoria própria.

Como mencionamos, é possível que um *hardware loop* seja iniciado mesmo sem que uma das instruções associadas sejam executadas. O Jasper™ mostrou um contra-exemplo que apresentava a sequência de instruções ilustrada na figura 8, que representa o *disassemble* do código binário das instruções executadas. Isso ocorreu quando as linhas 198 a 209 da figura 7 não estavam comentadas, e os *assumes* nas linhas 195 e 196 ainda não existiam. Podemos ver no *disassemble* das instruções que não foi executada nenhuma instrução de *hardware loop*. É de se esperar, já que usamos assumes para desabilitar essas instruções. Porém, como podemos observar no ciclo 9 da figura 9, a contagem de *loops* foi alterada. Explicaremos por que isso aconteceu.

Figura 8 – Instruções que levaram ao bug.

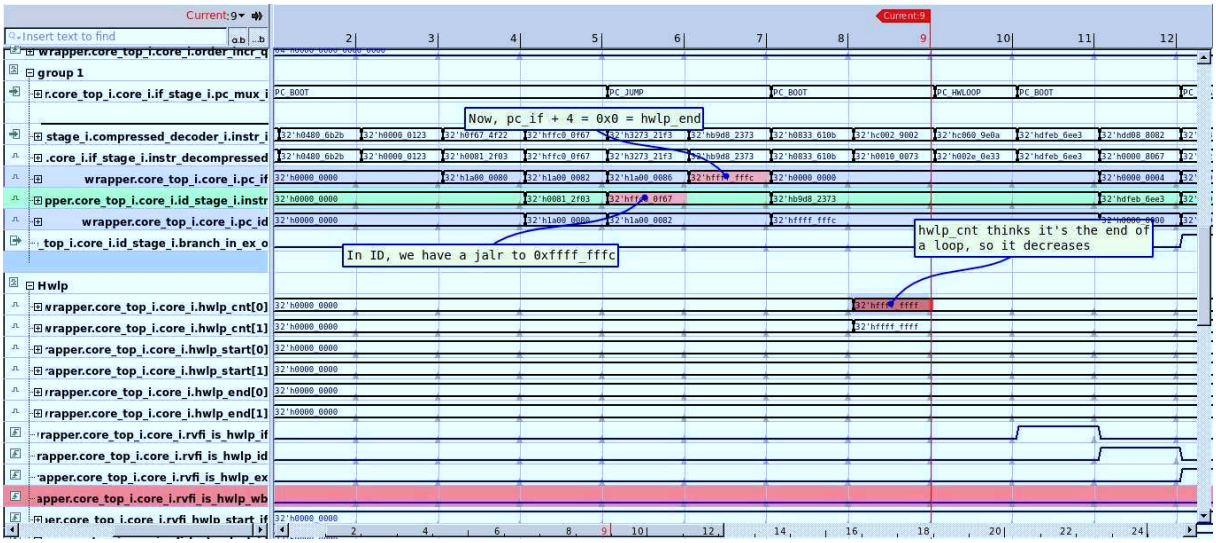
```
Disassembly of section .text:

00000000 <.text>:
   0:   4f22                c.lwsp  x30,8(x2)
   2:   ffc00f67            jalr    x30,-4(x0) # fffffffc <.text+0xfffffffc>
   6:   b9d82373            csrrs   x6,mhpmcounter29h,x16
   a:   dfeb6ee3            bltu    x22,x30,fffffe06 <.text+0xfffffe06>
   e:   106b19ab            cv.sh   x6,(x22),275
  12:   221b3b2b            cv.sh   x1,(x22),x22
  16:   2ae5                c.jal   20e <.text+0x20e>
  18:   e11f10e3            bne     x30,x17,fffffe18 <.text+0xfffffe18>
  1c:   b2e5                c.j     fffffa04 <.text+0xfffffa04>
```

Fonte: autoria própria.

A alteração no valor de `hwlp_cnt` aconteceu não por uma escrita ao registrador, mas pelo próprio mecanismo dos *loops*. Quando o controlador identifica que o core está na última instrução de um *loop*, ele realiza o decréscimo da contagem automaticamente. Quando a contagem chega em 0, sabemos que chegamos na última iteração do *loop*. Para identificar qual é a última instrução, o controlador checa o contador de programa: se o PC + 4 = `hwlp_end`, estamos no final de uma iteração e portanto devemos decrescer o contador e realizar um salto para `hwlp_start`. Então o controlador identificou, erroneamente,

Figura 9 – Ilustração do bug.



Fonte: Jasper™.

que o PC + 4 = `hwlp_end`, mesmo que `hwlp_start`, `hwlp_end` e `hwlp_cnt` fossem todos iguais a zero. Como isso ocorreu? Podemos observar que foi realizado um *jump* para o endereço 0xffff_fffc, o qual ao somarmos com 4 é igual a zero, por conta de um *overflow*. Isso atende a condição utilizada pelo controlador explicada anteriormente, pois PC + 4 = 0xffff_fffc + 0x4 = 0x0 = `hwlp_end`, e implica no decréscimo do contador de *loops*, mesmo que o contador já seja igual a zero. Em outras palavras, sempre que executamos uma instrução cujo PC é 0xffff_fffc e `hwlp_end` = 0x0, esse *bug* irá ocorrer.

No entanto, essa condição de *roll-over* em que o programa chega ao endereço máximo da memória e retorna a zero não é algo que esperamos observar em um programa normal. Isso pode ser utilizado como argumento para permitir esse tipo de comportamento no *core*. Caso desejemos evitar que isso aconteça, podemos adicionar a condição de que o contador não decresce se o seu valor já é zero.

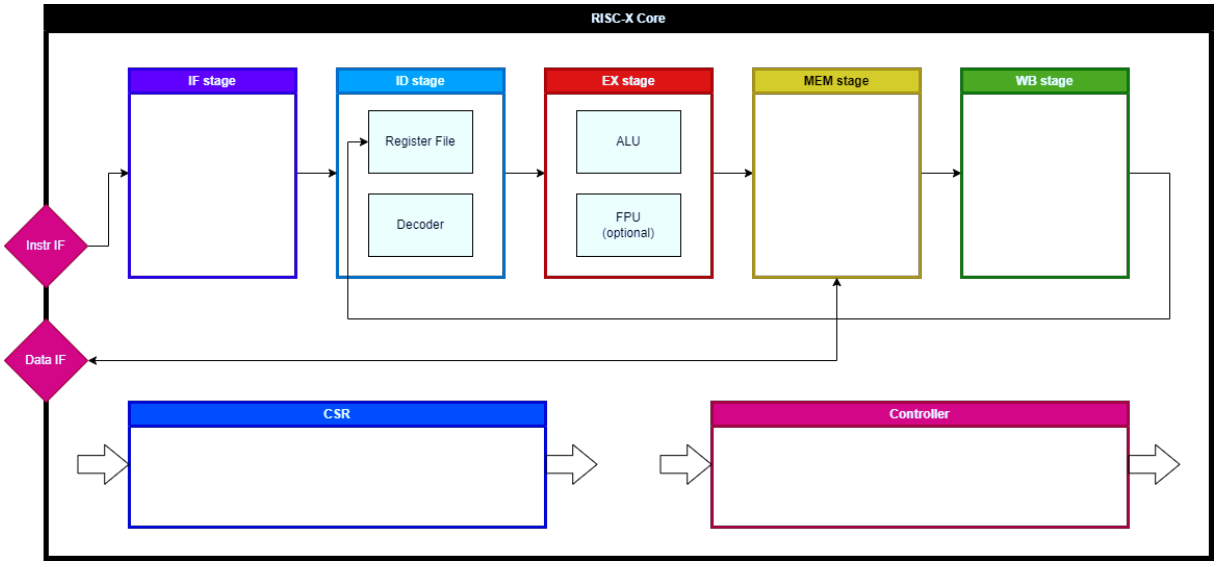
4 DESENVOLVIMENTO DO RISC-X

O RISC-X (RISC-X..., 2024) é um projeto desenvolvido por um grupo de estudantes no laboratório XMEN da UFCG. Trata-se de um núcleo de processamento RISC-V, com arquitetura de 32 bits, processamento em ordem e 5 estágios de *pipeline*. Foi seguido o texto do livro de Harris (HARRIS; HARRIS, 2021), em que é demonstrado como construir o design de um processador RISC-V em *pipeline*. Além disso, o projeto teve inspiração no CV32E40P, com o qual já tínhamos certa familiaridade. Temos alguns CSRs implementados, juntamente com as instruções de ler e modificar CSRs (extensão Zicsr). A funcionalidade completa de alguns CSRs ainda não foi implementada, como o `mstatus`, por exemplo. Opcionalmente, podem ser habilitadas as extensões C, para instruções de formato comprimido, e F, para instruções com números de ponto flutuante de precisão simples. No entanto, a interação das instruções tipo F com seus CSRs não foi totalmente implementada até o momento. Além disso, estamos desenvolvendo módulos de multiplicação e divisão para implementar a extensão M. O design possui parâmetros para habilitar ou desabilitar as extensões C, F e M.

Todos os registradores do design são sensíveis à borda de subida do *clock* e possuem *reset* assíncrono nível baixo ativo. A hierarquia do processador é mostrada na figura 10. Podemos notar que são utilizados os mesmos estágios de *pipeline* utilizados por Harris: *Instruction Fetch* (IF), em que são buscadas as instruções na memória; *Instruction Decode* (ID), em que a instrução é decodificada e os operandos são buscados do banco de registradores ou de outros estágios através do *forwarding*; *Execute* (EX), em que operações lógicas ou aritméticas são executadas; *Memory* (MEM), em que são feitos acessos à memória; e *Writeback* (WB), em que o resultado da instrução é escrito no banco de registradores. Em Harris, é feito apenas um subconjunto das instruções RISC-V, mas no RISC-X implementamos o conjunto completo, com algumas exceções. Não é implementada a instrução `ebreak`, pois essa instrução é utilizada para que o processador entre no modo de *debug*, uma funcionalidade que ainda não foi implementada por nossa equipe. Não foram implementadas as instruções `fence` e `fence.tso`, as quais são utilizadas para ordenar acessos à memória no ponto de vista de outros *harts* ou dispositivos externos. Ainda não estudamos os conceitos de *multi-hart* ou técnicas avançadas semelhantes.

Como mencionado anteriormente, além do conjunto base de instruções RISC-V, temos outros conjuntos de instruções implementados. Um deles é a extensão Zicsr, que adiciona instruções que realizam a leitura do valor de um CSR para um registrador de propósito geral (*General Purpose Register* - GPR) no banco de registradores, e adicionalmente realiza uma modificação no CSR. Essa modificação pode ser uma escrita, pode *setar* bits ou pode zerar bits. A informação de qual CSR é acessado é obtida através

Figura 10 – Hierarquia do RISC-X.



Fonte: autoria própria.

de um endereço de 12 bits. A tentativa de acessar um CSR não implementado resulta em uma exceção de instrução ilegal. Além disso, os CSRs possuem restrições de acesso, por exemplo, a tentativa de escrever um CSR de apenas leitura também resulta em uma instrução ilegal. O mesmo vale para a tentativa de acesso a um CSR sem o nível de privilégio apropriado, por exemplo ao tentar acessar um CSR de nível de privilégio máquina enquanto o *core* está no nível de privilégio de usuário. Os CSRs implementados no RISC-X são:

- `misa`: informa a largura dos registradores (XLEN) e as extensões implementadas. Ignoramos escritas nesse registrador.
- `mvendorid`, `marchid`, `mimpid` e `mhartid`: registradores de apenas leitura que informam certos números de identificação do *core*. O CSR `mhartid` pode ser inicializado durante o *reset* com uma certa entrada do *core*.
- `mstatus`: acompanha e controla o estado de operação do *core*. Não implementamos a funcionalidade desse CSR, atualmente é apenas um registrador que pode ser lido e escrito.
- `mtvec`: guarda o valor da configuração do vetor de *traps*, consistindo de um endereço base e um modo de operação. No RISC-X, o modo de operação é ligado em zero, o que implica que todas as *traps* realizam um salto para o endereço base especificado nesse registrador. Os 6 bits menos significativos do endereço também são ligados em zero para garantir um certo alinhamento. O CSR `mtvec` pode ser inicializado durante o *reset* com uma certa entrada do *core*.

- `mepc`: guarda o valor do endereço (PC) de uma instrução que causou uma exceção ou foi interrompida em decorrência de uma interrupção. No caso do RISC-X, a segunda opção não ocorre porque ainda não implementamos interrupções.
- `mcause`: quando uma *trap* ocorre, guarda um código que indica o evento que causou a *trap*.
- `mscratch`: pode ser utilizado como um armazenamento temporário pelo modo máquina, geralmente durante trocas de contexto ou rotinas de exceções.
- `mie`: habilita ou desabilita interrupções. Não implementamos a funcionalidade desse CSR, atualmente é apenas um registrador que pode ser lido e escrito.
- `fflags`: representa sinalizadores de exceções decorrentes de operações de ponto flutuante, como divisão por zero, *overflow*, etc. No RISC-X, `fflags` é sempre zero, a não ser que a extensão F esteja habilitada, e tentar acessar o CSR sem a extensão causa uma exceção.
- `frm`: representa o modo de arredondamento para instruções de ponto flutuante, usado quando os bits de arredondamento da instrução são 111. No RISC-X, `frm` é sempre zero, a não ser que a extensão F esteja habilitada, e tentar acessar o CSR sem a extensão causa uma exceção.

Atualmente, as exceções que podem acontecer no *core* são: endereço desalinhado de instrução (apenas quando a extensão C é desabilitada); instruções ilegais. As exceções geram *traps*, em que o *core* faz um salto para o endereço em `mtvec` e escreve o PC da instrução em `mepc`. Uma subsequente execução da instrução `mret` causa um salto para o PC salvo em `mepc`. Pensamos em gerar também exceções para acessos a endereços desalinhados de *loads* e *stores*. Uma interface de ambiente de execução RISC-V define o estado inicial do programa, o número e tipos de *harts*, e, entre outras coisas, a acessibilidade e atributos da memória (RISC-V…, 2019). Um exemplo seria a interface binária de aplicação (ABI) do Linux. Desse modo, podemos imaginar um ambiente de execução em que acessos a endereços desalinhados são permitidos. Para garantir maior compatibilidade, pretendemos tratar esses casos no futuro. Isso pode ser feito de forma invisível no *hardware*, identificando quando o acesso é desalinhado e transformando-o em 2 acessos alinhados, como é feito no CV32E40P. Caso isso se demonstre muito complicado para nossa equipe desenvolver, podemos implementar as exceções, que é o caso mais básico.

5 VERIFICAÇÃO DO RISC-X

Para a verificação do RISC-X, utilizamos tanto a verificação formal quanto a verificação dinâmica. Para o lado formal, utilizamos um ambiente de verificação baseado no RISC-V Formal, ferramenta que já havíamos utilizado no CV32E40P. Para a verificação dinâmica, construímos componentes de verificação em UVM, e utilizamos um gerador de instruções para gerar programas em *assembly* e alimentá-lo ao *testbench*.

Seguimos o método descrito em (MEDEIROS, 2024) para implementar o RFVF. Primeiro, fizemos o *wrapper* que encapsula o *core* e realiza os *tie-offs* de suas entradas. Fizemos um módulo contendo o *driver* para a interface RVFI, também contido no *wrapper* e conectado ao *core*. Além dos sinais básicos da RVFI, incluímos também os sinais utilizados para instruções de ponto flutuante, uma vez que nosso processador implementa a extensão F. No entanto, esses sinais não são utilizamos na verificação formal, uma vez que isso ainda não foi implementado no RFVF. Também fizemos um arquivo de configuração para fazer as definições necessárias para executar nossos testes. O RISC-X foi adicionado ao nosso *fork* do RISC-V Formal no GitHub (RISC-V..., 2024b).

Executamos todos os testes de instruções do conjunto RV32IC, bem como os principais testes de consistência, corrigindo falhas conforme foram encontradas. Conforme fomos desenvolvendo as extensões de instruções RISC-V e adicionando *hardware* ao nosso RTL, utilizamos o RISC-V Formal para verificar se as novas adições não geravam falhas, tanto no que já havia sido estabelecido e verificado como nas novas funcionalidades. A extensão M ainda está em desenvolvimento, mas pretendemos implementar as operações alternativas do RFVF quando esse trabalho for finalizado. Essas operações são necessárias para reduzir a complexidade das provas quando operações matemáticas complexas (multiplicação e divisão) estão presentes.

Para a verificação dinâmica, nós utilizamos o gerador de instruções da Google, o RISCV-DV (RISCV-DV, 2024) para gerar os programas em *assembly*. Os programas gerados pelo RISCV-DV incluem inicialização de registradores, rotinas de exceção, regiões de memória, entre outros, e podemos customizar a geração dos programas de várias formas. O repositório já vem com alguns casos de teste base, como testes aritméticos, saltos incondicionais, acessos à memória, etc, os quais aproveitamos para a verificação do RISC-X. A ferramenta possui um programa em Python que automatiza o processo: compila o gerador de instruções, cujo código é escrito em SystemVerilog; executa o gerador para gerar o *assembly*; compila o *assembly*, usando um *script* de *link* para definir os endereços das seções da memória, gerando um arquivo executável ELF; gera uma imagem binária do executável (arquivo .bin), a qual lemos no *testbench* para executar nossa simulação; chama

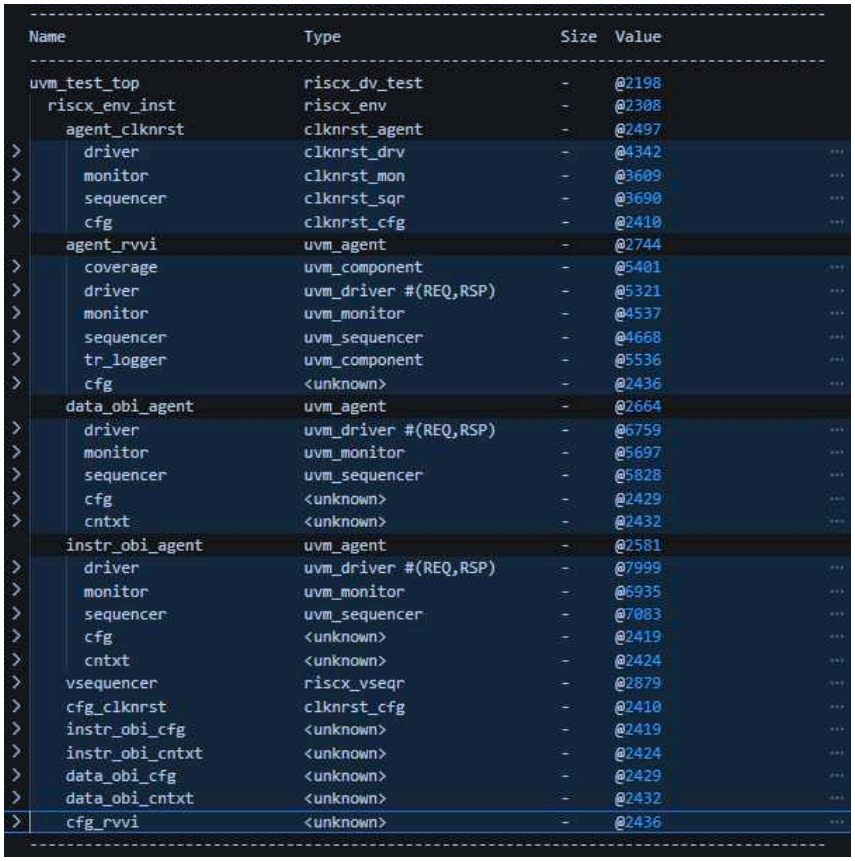
o comando para executar um ou mais simuladores (o RISC-X é considerado um simulador para essa etapa) com o ELF gerado, resultando em arquivos de log; caso utilizemos a opção de 2 simuladores, vamos para a parte de comparação, em que os arquivos de log são convertidos em um formato comum (.csv) e comparados, gerando um relatório com as divergências, caso existam. Para verificar o RISC-X, usamos esse fluxo automatizado, usando nosso *core* e o simulador Spike como os 2 simuladores, em que o Spike funciona como modelo de referência. É considerado um erro quando uma escrita em um GPR difere entre um simulador e outro. O programa não considera um erro quando uma escrita em um CSR é diferente, e também não checa acessos à memória (ainda é checado o registrador escrito por um *load*, mas não o endereço acessado).

Para o nosso ambiente de simulação, utilizamos a Metodologia Universal de Verificação, UVM. Em geral, cada componente de verificação de UVM (*UVM Verification Component* - UVC) é responsável por controlar uma interface. Em nosso *core*, temos 3 tipos de interfaces, o que significa que precisamos de 3 tipos de UVCs: interface de *clock* e *reset*, interface de memória, e a RISC-V *Verification Interface* (RVVI). A RVVI é uma espécie de sucessora da RVFI, com o objetivo de ser usada de forma eficiente em ambientes de verificação dinâmica. A UVC da RVVI não envia sinal nenhum para o RTL, apenas o monitora. Seu objetivo é gerar um *log* das instruções executadas pelo processador, o qual é utilizado para a posterior comparação com o modelo de referência. A UVC de *clock* e *reset* controla os sinais de *clock* e *reset*. Para a interface com a memória, utilizamos a OBI, o mesmo barramento utilizado no CV32E40P. Ela possui um *handshake* na entrada, feito pelos sinais `req` e `gnt`, e um *handshake* na saída, feito pelos sinais `rvalid` e `rready`. Isso é importante para que possamos conectar nosso processador a memórias que possam levar mais de um ciclo de *clock* para concluir uma transação. Estamos chamando de entrada o que o OBI define como o canal de endereço, em que o *manager*, nesse caso o *core*, envia uma requisição de transação para o barramento. Já a saída seria o canal de resposta, em que é recebida uma resposta do *subordinate*. Em nosso *testbench*, criamos dois objetos da UVC do OBI, uma vez que temos uma interface para a memória de instruções e outra para a memória de dados.

Criamos um *environment* para o RISC-X, o qual cria e conecta todas as UVCs citadas. Além disso, criamos todos os objetos de configurações e selecionamos os valores adequados para a verificação do RISC-X. Também temos que atribuir a mesma memória para as UVCs de dados e de instrução, caso contrário serão consideradas duas memórias diferentes. Por fim, temos ainda um *sequencer* virtual, responsável por controlar os *sequencers* de cada UVC. O início e fim da simulação é controlado pelas sequências virtuais: uma objeção é levantada no início da sequência, informando ao ambiente UVM que a simulação não deve ser finalizada, e a objeção é baixada ao final da sequência, informando que a simulação pode ser finalizada.

A hierarquia do *testbench* em UVM é apresentada na figura 11. Essa árvore de classes é impressa no log de nossas simulações. Podemos observar que o `uvm_test_top` instancia nosso *environment*. Esse, por sua vez, instancia os 4 agentes, um para cada interface utilizada: *clock* e *reset*, RVVI, OBI de dados e OBI de instruções. Além disso, temos o *sequencer* virtual e objetos de configuração e contexto.

Figura 11 – Hierarquia do testbench em UVM.



Name	Type	Size	Value
uvm_test_top	riscx_dv_test	-	@2198
riscx_env_inst	riscx_env	-	@2308
agent_clknrst	clknrst_agent	-	@2497
driver	clknrst_drv	-	@4342
monitor	clknrst_mon	-	@3609
sequencer	clknrst_sqr	-	@3690
cfg	clknrst_cfg	-	@2410
agent_rvvi	uvm_agent	-	@2744
coverage	uvm_component	-	@5401
driver	uvm_driver #(REQ,RSP)	-	@5321
monitor	uvm_monitor	-	@4537
sequencer	uvm_sequencer	-	@4668
tr_logger	uvm_component	-	@5536
cfg	<unknown>	-	@2436
data_obi_agent	uvm_agent	-	@2664
driver	uvm_driver #(REQ,RSP)	-	@6759
monitor	uvm_monitor	-	@5697
sequencer	uvm_sequencer	-	@5828
cfg	<unknown>	-	@2429
cntxt	<unknown>	-	@2432
instr_obi_agent	uvm_agent	-	@2581
driver	uvm_driver #(REQ,RSP)	-	@7999
monitor	uvm_monitor	-	@6935
sequencer	uvm_sequencer	-	@7083
cfg	<unknown>	-	@2419
cntxt	<unknown>	-	@2424
vsequencer	riscx_vseqr	-	@2879
cfg_clknrst	clknrst_cfg	-	@2410
instr_obi_cfg	<unknown>	-	@2419
instr_obi_cntxt	<unknown>	-	@2424
data_obi_cfg	<unknown>	-	@2429
data_obi_cntxt	<unknown>	-	@2432
cfg_rvvi	<unknown>	-	@2436

Fonte: Xcelium™.

Na sequência utilizada para a verificação do RISC-X junto ao RISCV-DV, executamos primeiro uma sequência de *clock* e *reset* em que o *reset* é ativado por um tempo, depois desativado, e em seguida a geração do *clock* é iniciada, continuando assim até o final da simulação. Em seguida, iniciamos uma sequência de carregamento da memória, em que o arquivo `.bin` gerado pelo RISCV-DV é lido e a memória do OBI é carregada com seus conteúdos. Em seguida, iniciamos paralelamente 3 processos. Dois deles são as sequências das UVCs de instruções e dados, em que elas monitoram o canal de endereço de suas respectivas interfaces e, quando uma requisição é detectada, geram os sinais do canal de resposta. Esses processos são executados em um laço infinito. O terceiro processo é o que tem duração finita e será responsável por encerrar nossa sequência. Ele simplesmente espera um sinal chamado `should_drop_objection` ser igual a 1, e encerra o processo. Esse sinal inicia na simulação com o valor 0, e é alterado via uma `uvm_analysis_imp` em nosso *sequencer* virtual. Essa `uvm_analysis_imp` é conectada a uma `uvm_analysis_port`

da UVC da RVVI, a qual é ativada quando uma certa instrução, definida pelo objeto de configuração, é detectada. Configuramos a RVVI para detectar a instrução `ecall`, que é utilizada pelo RISCV-DV para indicar que um programa chegou ao fim. Com isso, nossa simulação é encerrada quando uma instrução `ecall` é executada por nosso *core*.

6 CONSIDERAÇÕES FINAIS

Foi possível realizar avanços consideráveis na verificação do CV32E40P, com a realização dos testes de consistência e das instruções que não haviam sido contempladas no trabalho anterior. Foi possível constatar que os erros destacados na versão antiga do *core* não estão mais presentes atualmente. No entanto, enquanto precisarmos utilizar *assumptions* para impedir certas instruções de serem executadas, a verificação não estará perfeita. É relevante destacar que, após o início de nossas atividades, já foram realizados diversos *commits* no repositório do processador, incluindo uma nova versão (v1.8.3). Ao utilizar a ferramenta do GitHub de comparação alterações, pudemos observar que foram adicionados mais arquivos destinados a verificação formal, incluindo um arquivo cheio de propriedades relacionadas aos *hardware loops*. Isso pode ser aproveitado em testes futuros.

Com relação ao RISC-X, foi possível desenvolver um processador bem mais funcional do que aquele básico apresentado no texto de Harris. Ainda existem vários pontos que queremos tratar no futuro, como o suporte a interrupções e modo de *debug*. Ainda precisamos concluir a implementação da funcionalidade dos CSRs já existentes, e adicionar outros, como o `mtval`, contadores de monitoramento de performance, entre outros. A verificação foi eficaz para identificar diversos *bugs*, utilizando tanto o RISC-V Formal quanto o ambiente UVM. No entanto, há um problema sério com a nossa verificação: não estamos utilizando cobertura. Nossa ideia inicial era utilizar o programa de cobertura do próprio RISCV-DV, que é um *script* em Python que compila o ambiente de cobertura em SystemVerilog, e esse ambiente lê as instruções executadas a partir do log em formato ".csv". No entanto a coleta de cobertura não está funcionando adequadamente. Por exemplo, quando abrimos o arquivo de cobertura para visualização, consta que não foram executadas instruções de adição, mas quando abrimos o log, vemos que na verdade foram executadas centenas de adições. É uma das nossas prioridades para o futuro próximo corrigir essa coleta de cobertura, ou então construir nossa própria e incorporar ao nosso ambiente UVM.

Além disso, estamos trabalhando no desenvolvimento de um SoC, para englobar o RISC-X e outras IPs que nosso time venha a desenvolver. Ainda não definimos quais IPs serão desenvolvidas, porém sugestões envolvem módulos de criptografia, protocolos, segurança ou aceleradores. Podemos ter aceleradores, por exemplo, para atividades neuromórficas ou sensores biométricos. Para um estágio inicial do SoC, estamos pensando em incluir apenas o *core* RISC-V, um barramento, provavelmente o AXI-Lite, um controlador, um módulo de memória e uma comunicação serial, provavelmente a UART. Esperamos bastante progresso nesse desenvolvimento nos próximos meses.

REFERÊNCIAS

HARRIS, S.; HARRIS, D. *Digital Design and Computer Architecture: RISC-V Edition*. 1. ed. [S.l.]: Morgan Kaufmann, 2021. Citado na página 30.

MEDEIROS, P. A. da C. Aplicação do risc-v formal verification framework para verificação formal de um núcleo de processamento. In: UFCG - CEEI - DEE - PROJETOS DE ENGENHARIA ELÉTRICA. [S.l.], 2024. Citado nas páginas 9, 10, 11 e 33.

OPENHW Group. 2024. <https://www.openhwgroup.org/>. Accessed: 15 de outubro de 2024. Citado na página 10.

OPENHW Group CORE-V CV32E40P RISC-V IP. 2024. <https://github.com/openhwgroup/cv32e40p>. Accessed: 15 de outubro de 2024. Citado nas páginas 10 e 21.

OPENHW Group CV32E40P User Manual. 2024. <https://cv32e40p.readthedocs.io/en/latest/index.html>. Accessed: 15 de outubro de 2024. Citado na página 22.

PULPISSIMO. 2022. <https://github.com/pulp-platform/pulpissimo>. Accessed: 15 de outubro de 2024. Citado na página 11.

RISC-V Formal Verification Framework. 2024. <https://github.com/YosysHQ/riscv-formal>. Accessed: 15 de outubro de 2024. Citado na página 24.

RISC-V Formal Verification Framework - Pulpissimo Edition. 2024. <https://github.com/ArthurMdrs/riscv-formal>. Accessed: 15 de outubro de 2024. Citado na página 33.

RISC-V Specifications. 2019. <https://riscv.org/technical/specifications/>. Accessed: 15 de outubro de 2024. Citado na página 32.

RISC-X Core. 2024. <https://github.com/ArthurMdrs/RISC-X>. Accessed: 15 de outubro de 2024. Citado nas páginas 11 e 30.

RISCV-DV. 2024. <https://github.com/chipsalliance/riscv-dv>. Accessed: 15 de outubro de 2024. Citado nas páginas 11 e 33.

VALE, K. D. Ral - register abstraction layer. In: UFCG - CEEI - DEE - PROJETOS DE ENGENHARIA ELÉTRICA. [S.l.], 2021. Citado na página 20.