Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Pedro Arthur da Cunha Medeiros

Aplicação do RISC-V Formal Verification Framework para verificação formal de um núcleo de processamento

Campina Grande
2024

Pedro Arthur da Cunha Medeiros

Aplicação do RISC-V Formal Verification Framework para verificação formal de um núcleo de processamento

Trabalho de Conclusão de Curso 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

Aplicação do RISC-V Formal Verification Framework para verificação formal de um núcleo de processamento

Trabalho de Conclusão de Curso 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

A verificação de *hardware* é essencial para garantir a qualidade de circuitos eletrônicos. Nesse contexto a verificação formal é capaz de encontrar *bugs* que seriam muito difíceis de encontrar em uma simulação dinâmica, impulsionando a qualidade da verificação. Baseada em propriedades, a verificação formal busca provar que um design apresenta os comportamentos intencionados e não apresenta comportamentos indesejados. O RISC-V Formal Verification Framework é uma ferramenta de verificação formal de código aberto, que possibilita a verificação de núcleos de processamento baseados na arquitetura RISC-V. Nos propomos, então, a aplicar essa ferramenta no processador CV32E40P utilizado na plataforma PULP, também de código aberto. Com isso, pudemos encontrar *bugs* no design e compreender o porquê deles acontecerem.

Palavras-chave: Verificação de *Hardware*, Design de *Hardware*, Verificação Formal, Núcleo de Processamento, RISC-V.

ABSTRACT

Hardware verification is essential to ensure the quality of electronic circuits. In this context, formal verification is capable of finding bugs that would be too hard to find in a dynamic simulation, boosting verification quality. Based on properties, formal verification attempts to prove that a design shows the intended behaviors and does not show undesired behaviors. The RISC-V Formal Verification Framework is an open-source formal verification tool, which enables the verification of processing cores based on RISC-V architecture. We therefore proposed to apply this tool on the CV32E40P processor used in the PULP platform, which is also open-source. With that, we were able to find bugs in the design and understand why they happened.

Keywords: Hardware Verification, Hardware Design, Formal Verification, Processing Core, RISC-V.

LISTA DE ILUSTRAÇÕES

Figura 1 – Hierarquia de módulos do RISC-V Formal. 25

Figura 2 – Hierarquia dos módulos em um teste de instrução. 33

Figura 3 – Um laço de *hardware* saltando para endereço desalinhado. 42

Figura 4 – Um laço de *hardware* de apenas 1 instrução. 43

Figura 5 – Erro no alvo do laço de *hardware*. 44

Figura 6 – Instrução ilegal sendo decodificada como `p.sw`. 44

Figura 7 – Acesso ilegal a CSR de apenas leitura que não gera uma *trap*. 45

LISTA DE ABREVIATURAS E SIGLAS

COI	*Cone of Influence*
CSR	*Control and/or Status Register*
DEE	Departamento de Engenharia Elétrica
DUT	*Design Under Test*
EX	*Execute*
HDL	*Hardware Description Language*
ID	*Instruction Decode*
IF	*Instruction Fetch*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
OBI	*OpenBus Interface*
RFVF	*RISC-V Formal Verification Framework*
RTL	*Register Transfer Level*
RFVI	*RISC-V Formal Interface*
SIMD	*Single Instruction, Multiple Data*
SVA	*SystemVerilog Assertions*
tcl	*Tool Command Language*
UFCG	Universidade Federal de Campina Grande
ULA	Unidade Lógico-Aritmética
UVM	*Universal Verification Methodology*
WB	*Writeback*

SUMÁRIO

Lista de ilustrações . . . 5

1 INTRODUÇÃO . . . 8
1.1 Objetivos . . . 10
1.1.1 Objetivo Geral . . . 10
1.1.2 Objetivos Específicos . . . 10

2 VERIFICAÇÃO FORMAL . . . 12
2.1 Complexidade das Provas . . . 13
2.2 Técnicas de Redução de Complexidade . . . 14

3 SYSTEMVERILOG ASSERTIONS . . . 17

4 NÚCLEO DE PROCESSAMENTO CV32E40P . . . 21

5 IMPLEMENTAÇÃO DO RFVF . . . 23
5.1 Wrapper . . . 24
5.2 RISC-V Formal Interface . . . 26
5.3 Configuração . . . 29

6 CUSTOMIZAÇÕES DA FERRAMENTA . . . 32
6.1 Expansão da RVFI . . . 32
6.2 Testes para instruções xpulp . . . 33
6.3 Gerador de testes para o Jasper™ . . . 34

7 DIFERENÇAS ENTRE O JASPER™ E O SYMBIYOSYS . . . 39

8 BUGS ENCONTRADOS . . . 41

9 CONSIDERAÇÕES FINAIS . . . 46

REFERÊNCIAS . . . 47

1 INTRODUÇÃO

Com os avanços tecnológicos constantes da modernidade, as empresas que desenvolvem tais tecnologias são desafiadas a continuar inovando e criando *hardwares* cada vez mais complexos para que possam continuar a ser competitivas no mercado. Todos os anos modelos novos de *smartphones* e notebooks são lançados com novas características que os distinguem dos anteriores, como um aumento de memória RAM, introdução de núcleos de alta eficiência energética, núcleos de inteligência artificial, entre outros.

Para aumentar o poder computacional dos chips, ao mesmo que mantendo um nível aceitável de consumo energético, foram desenvolvidas técnicas para alocar cada vez mais transistores na mesma área de silício. No entanto, isso também implica no aumento da complexidade dos circuitos, e consequentemente, na possibilidade de haver *bugs* no projeto. Deixar um erro passar despercebido para a etapa de fabricação, ou, pior, a de distribuição, pode gerar prejuízos de milhões de dólares e até mesmo afundar a reputação de uma empresa. Para conciliar com essa realidade, as metodologias de verificação vem constantemente sendo evoluídas e novas técnicas vem surgindo.

Um fluxo de produção tradicional parte da especificação de um projeto. Em geral, o documento de especificação pode ser elaborado por uma equipe de engenheiros *senior* da empresa, aqueles com mais experiência, ou então pode ser fornecido diretamente pelo cliente. Após isso, o fluxo é dividido em pelo menos 3 equipes: a equipe de design de alto nível, também chamado de *front-end*, responsável por criar um modelo do circuito utilizando uma linguagem de descrição de *hardware* (*Hardware Description Language* - HDL), utilizando como base a especificação do projeto; a equipe de verificação, responsável por realizar os devidos testes com o design a fim de garantir o seu funcionamento correto, conforme especificado; a equipe de *backend*, responsável por realizar a síntese do circuito, transformando descrição de hardware em células com descrições detalhadas sobre seu real comportamento físico, para que, com isso, o chip possa ser fabricado. Existe um *loop* entre as etapas de design e verificação em que os *bugs* são encontrados pela equipe de verificação e corrigidos pela de design.

Tomando como base o fluxo brevemente descrito, podemos dizer que esse trabalho gira em torno da etapa de verificação do projeto. Quanto a essa atividade, ela pode se apresentar em duas formas, cada uma com áreas em que se saem melhor ou pior. Uma delas é a verificação funcional dinâmica, comumente chamada de simulação dinâmica ou simplesmente simulação, a qual consiste em elaborar testes de forma a explorar quais são as respostas do design perante a determinados estímulos. Essas respostas do design, independentemente de como tenha sido escrito o código, devem corresponder à funciona-

lidade definida na especificação do projeto. Nesse método, é necessário que o design seja levado a um estado de estresse, o que pode ser atingido ao aplicar um número elevado de estímulos aleatórios ou sequências específica de dados. Atualmente, a metodologia universal de verificação (*Universal Verification Methodology* - UVM) é utilizada de maneira abrangente para realizar a simulação dinâmica, oferecendo um método padronizado, com propriedades intelectuais reutilizáveis. A UVM consiste de uma biblioteca de classes em SystemVerilog, utilizando o conceito de programação orientada a objetos para construir um ambiente de verificação de forma segura e eficiente.

A segunda vertente seria a verificação estática, ou verificação formal. A análise formal é uma abordagem rigorosa, baseada em regras. Ao invés de analisar as respostas a estímulos como na simulação, são declarados comportamentos permitidos que o design pode assumir, os quais são provados matematicamente por uma ferramenta de análise formal (SANTOS, 2022). Quando um desses comportamentos é provado, isso significa que não há nenhuma combinação de estímulos que o modelo pode receber que irá fazer aquela assertiva ser violada. Isso é particularmente importante em sistemas que possuem diferentes níveis de acesso e necessitam de alta confiabilidade.

Microprocessadores são sistemas digitais que leem e executam instruções, as quais são palavras binárias (HARRIS; HARRIS, 2021). Ao unir o conjunto de instruções e a localização dos operandos de um microprocessador, temos a sua arquitetura. RISC-V é uma arquitetura de conjunto de instruções (*Instruction Set Architecture* - ISA) *open-source* baseada nos princípios de computador com conjunto reduzido de instruções (*Reduced Instruction Set Computer* - RISC). O RISC-V, atualmente, é amplamente difundido na indústria. Isso se dá em parte por ser uma ISA *open-source* e livre de *royalties* ou outras taxas, permitindo que diversas empresas começassem a desenvolver seus próprios processadores. Por ser um padrão aberto, o RISC-V pode ser modificado e melhorado conforme as necessidades de cada um, permitindo uma liberdade de inovação muito maior do que se tinha antes e diminuindo os custos de desenvolvimento. Além disso, existe um ambiente colaborativo em que as falhas e sucessos são compartilhados para promover o processo inovativo. A arquitetura possui um conjunto de instruções base e várias extensões que podem ser implementadas de acordo com cada projeto, permitindo que os desenvolvedores implementem apenas as extensões necessárias para suas aplicações específicas, podendo ainda criar suas próprias extensões. Isso possibilita a criação de processadores altamente otimizados, resultando em maior eficiência energética e desempenho.

Esse trabalho visa utilizar uma ferramenta de verificação formal *open-source* chamada RISC-V *Formal Verification Framework* (RFVF), desenvolvida atualmente pelo grupo YosysHQ (RISC-V. . . , 2024a). O YosysHQ ficou popular por sua ferramenta *open-source* de síntese de *hardware*, o Yosys. Associada ao Yosys, temos a ferramenta de verificação formal SymbiYosys, sobre a qual é baseada o RFVF. O *framework* possui uma

descrição formal da ISA RISC-V que é independente de implementação, ou seja, deve ser válida para qualquer *core* RISC-V. Com isso, é possível fazer a validação de *cores* de forma eficiente, corrigindo *bugs* em que o design não se comporta conforme a especificação (RISC-V. . . , 2019), seja por erro de design ou até mesmo por interpretação errônea do documento.

No entanto, as empresas fornecedoras de serviços de microeletrônica, em geral, pagam por licenças de *softwares* profissionais com muito mais capacidades e funções do que aqueles disponíveis com código aberto. Uma das maiores empresas que oferecem esse tipo de produto é a Cadence Design Systems®, a qual oferece programas especializados para cada etapa de desenvolvimento de uma propriedade intelectual (IP) de *hardware*. A ferramenta de análise formal da Cadence® é chamada Jasper™, a qual oferece diversos Apps com diferentes finalidades para fazer checagens formais nos designs. Temos, por exemplo, o *Coverage* App que oferece métricas do quanto o espaço de estados do design foi explorado, o *X-Propagation Verification* App que identifica os pontos em que um valor X (valor indeterminado) pode ser propagado, o *Control and Status Register Verification* App, que cria as assertivas automaticamente para todos os *Control and Status Registers* (CSRs) declarados, testando-os exaustivamente (SANTOS, 2022), etc. Com isso, propõe-se utilizar o *Formal Property Verification* App, capaz de provar as propriedades formais, para provar as mesmas propriedades geradas pelo RFVF, e em seguida comparar a performance e resultados de ambas as ferramentas utilizadas, o Jasper™ e o SymbiYosys.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

O desenvolvimento deste trabalho tem como objetivo o estudo e aplicação da ferramenta RISC-V Formal Verification Framework para a verificação formal de um *core* baseado na arquitetura RISC-V, além da comparação do SymbiYosys com o Jasper™.

1.1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos para o desenvolvimento desse projeto são:

- Estudar a ferramenta RISC-V Formal Verification Framework;
- Adaptar um *core* para o uso conjunto à ferramenta, o qual não tenha sido previamente verificado formalmente (de acordo com as informações disponíveis publicamente);
- Aplicar a ferramenta para a verificação desse *core*.

- Desenvolver um programa que converte os *scripts* do SymbiYosys em *scripts* do Jasper™.
- Comparar a performance e resultados das provas realizadas em ambos os *softwares*.

2 VERIFICAÇÃO FORMAL

A verificação formal é um processo em que o modelo do *hardware* em *Register Transfer Level* (RTL) é transformado em um modelo matemático e testado com todas as combinações possíveis de entradas para testar a presença ou não de certos comportamentos, definidos por propriedades. Como a verificação é feita de forma exaustiva, quaisquer *bugs* que possam ser encontrados serão encontrados, não importa o quão específicos sejam. Isso é algo que a simulação não é capaz de fazer, pois, em projetos complexos, a verificação sempre deixa "buracos" na cobertura do estado de espaços do design sob teste (*design under test* - DUV). Isso se deve ao fato de que o espaço de estados ser muito grande, tornando impossível de checar todos, além de existirem estados muito difíceis de se atingir apenas com estímulos aleatórios. O que é comumente feito é uma verificação baseada em métricas (*metric-driven verification*), que define as funcionalidades que um design deve ter, os testes necessários para testar essas funcionalidades e a quantidade de cobertura necessária para considerar que o design de fato tem essas funcionalidades. Quando a cobertura é atingida sem que os testes falhem, consideramos o design verificado.

Por outro lado, a verificação formal não é viável para verificação *end-to-end* de sistemas de larga escala (ALMEIDA, 2018). A complexidade de uma prova formal depende do espaço de estados, o qual cresce de forma exponencial com o número de registradores. Isso significa que, quando o número de registradores em um design aumenta, uma prova que depende dos valores desses registradores terá sua complexidade aumentada, e consequentemente o tempo necessário para concluí-la, com proporção exponencial. Falaremos mais sobre isso na seção 2.1. Também é necessário definir no plano de verificação a cobertura necessária para provas formais. Uma ferramenta pode, por exemplo, checar quais registradores são exercitados por alguma propriedade. Isso não garante que essa propriedade é suficiente para achar erros na implementação desses registradores, pois ela pode ter sido escrita de forma errônea. Para ajudar a garantir que as funcionalidades são exercitadas em uma análise formal, o Jasper™ possui os Apps Superlint e Coverage App.

A abordagem utilizada na verificação formal é baseada em propriedades, partindo do princípio de que os circuitos digitais são síncronos, ou seja, são sincronizados a um sinal de *clock*, e portanto apresentam comportamentos que evoluem de acordo com os pulsos desse sinal (ALMEIDA, 2018). O design é transformado em um modelo matemático através de um processo de síntese lógica, e então a ferramenta aplica automaticamente estímulos ao modelo e analisa as propriedades de acordo com as diretivas utilizadas, como veremos no capítulo 3.

2.1 COMPLEXIDADE DAS PROVAS

A complexidade é um dos principais desafios da verificação formal. O seu conceito está relacionado ao nível de esforço computacional necessário para se exercitar uma análise. Quanto maior a complexidade de uma prova, mais dados e cálculos serão necessários para a sua análise, e, consequentemente, mais memória consumirá e mais tempo levará para ser concluída. Quando a ferramenta de análise formal atinge o tempo limite de uma prova, que pode ser definido pelo usuário, o *status* da prova é alterado para indeterminado, ou seja, não é conclusivo. O controle da complexidade deve ser um dos pontos mais importantes a se considerar ao se preparar o ambiente de verificação, uma vez que pode facilmente aumentar o tempo consumido pela ferramenta para além dos prazos viáveis exigidos por um projeto.

Para entender as causas da complexidade, precisamos entender o que é um cone de influência (*Cone of Influence* - COI): são todas as entradas primárias e registradores que podem afetar o resultado de uma propriedade, ou seja, o seu *fan-in*. Quanto mais entradas, lógica combinacional e registradores afetam uma propriedade, maior o seu COI e maior a sua complexidade. Ao analisar uma propriedade, uma ferramenta de análise formal irá trabalhar apenas nos elementos de seu COI, uma vez que todo o resto não afetará o resultado. O tamanho do COI é um forte indicativo do esforço computacional que a prova exigirá ao ser analisada, mas podem haver casos de propriedades com grande *fan-in* que não são problemáticas, quando apenas parte desse conjunto afetará de fato o resultado. Desse modo, podemos definir as propriedades *end-to-end*, que são aquelas que precisam do COI inteiro, e, portanto, devemos alocar mais esforços para otimizá-las, e as propriedades locais, que utilizam apenas parte de seu COI.

Outro conceito importante é o de estado. Um estado de um design pode ser entendido como um longo vetor contendo os valores de todos os seus registradores (ou variáveis) e suas entradas primárias. Quando um registrador ou uma entrada muda, temos um novo estado. O espaço de estados do design é, então, o conjunto de todos os estados que o design pode assumir. Do estado de espaços, é possível que apenas uma parcela dele seja possível de acontecer, partindo de um estado inicial (normalmente o estado após o *reset*). Se, por exemplo, uma máquina de estados inicializa em algum estado ilegal, é possível que ela não consiga retornar a um estado legal de funcionamento, tornando aquele estado inatingível.

Precisamos do conceito de estado de espaços para definir o segundo indicativo de complexidade: o diâmetro. Ele é definido pelo número de passos (ciclos de *clock*) necessários para se chegar a todos os espaços atingíveis do espaço de estados, dado um estado inicial. O diâmetro é relacionado ao número de bits de estado, em que o seu valor máximo é 2 elevado ao número de bits. Podemos dizer que podem existir estados "profundos" no design, ou seja, que necessitam de vários ciclos e condições atendidas para

serem atingidos. *Bugs* podem estar escondidos nesses estados profundos, que são difíceis de encontrar na simulação mas que certamente serão explorados em uma análise formal, já que a análise formal é exaustiva e explora todos os estados (desde sejam exercitados por alguma assertiva). Mesmo considerando estados que não sejam profundos, a simulação gera estímulos aleatórios que vão seguir certos caminhos no espaço de estados, mas não todos os caminhos possíveis. Como no caso do COI, é possível existir um grande diâmetro sem grande impacto na performance. Por exemplo, se temos um contador que interage com o design apenas em poucos estados.

A complexidade é aumentada, portanto, pelo tamanho do *fan-in* de uma propriedade e da quantidade de bits de estados de um design. Além de existirem das características do próprio RTL, os bits de estados podem ser inferidos de acordo com a maneira em que assertivas e código auxiliar são escritos. O conceito de assertiva será apresentado no capítulo 3. Se a assertiva contém uma sequência temporal, por exemplo, isso é traduzido a uma máquina de estados, adicionando bits de estados, portanto sequências temporais devem ser evitadas para reduzir a complexidade. Um fator que comumente impacta bastante a complexidade são operações matemáticas. Transformações nos bits dos dados impedem que a análise seja simplificada, forçando a ferramenta a analisar todos os bits em conjunto, ao invés de um de cada vez, aumentando o esforço computacional.

2.2 TÉCNICAS DE REDUÇÃO DE COMPLEXIDADE

Já que a complexidade é a medida do quão ruim será a performance da análise formal, é necessário se utilizar técnicas para otimizar o processo. Uma das técnicas de redução de complexidade é o uso de abstrações. De acordo com (MELHAM, 1990), existem 4 tipos de abstrações que são importantes para a verificação: abstração estrutural, abstração comportamental, abstração de dados e abstração temporal. Falaremos brevemente sobre elas.

A abstração estrutural consiste em suprimir os detalhes internos de uma estrutura. Isso é comumente chamado de “caixa preta” (*blackbox*). Na prática, o que acontece é que as saídas do bloco abstraído são tratadas como fios livres, e assim podem assumir qualquer valor, independente da estrutura que faz seu *drive*. Blocos interessantes de se aplicar uma caixa preta são, por exemplo, contadores. A ferramenta precisa examinar todos os estados possíveis dos contadores, então contadores grandes irão implicar em um grande diâmetro. Ao abstraí-los, podemos fazer com que o valor do contador só assuma os valores relevantes, que irão acionar alguma reação no sistema. Outra estrutura seria uma memória, já que memórias tem uma quantidade muito grande de registradores. O Jasper™ realiza o *blackboxing* automático de operadores de multiplicação e divisão, os quais tem muita complexidade combinacional.

A abstração comportamental consiste em suprimir situações de operação que não devem ocorrer. Se o objeto de verificação é um barramento, por exemplo, devemos assumir que as entradas irão sempre respeitar o protocolo da interface, e não apenas apresentar um comportamento aleatório. Isso pode ser utilizado também para situações que não importam. Por exemplo, vamos supor que um módulo opera de acordo com o valor de um contador de 4 bits, mas que esse contador só conta até 10 em condições normais, então não nos importa o que aconteça quando o valor é maior que 10.

A abstração de dados consiste em mapear duas representações de dados, sendo a primeira aquela presente na especificação e a segunda na implementação. Se na especificação temos um número inteiro que poderia ser representado com poucos bits, nós fazemos essa simplificação e fazemos uma descrição de como esses dados devem ser interpretados. Um método semelhante à abstração de dados é a redução. Podemos utilizar dos parâmetros de um módulo para reduzir suas dimensões. Reduzir o tamanho de uma memória, por exemplo, pode diminuir a complexidade sem nenhum efeito adverso, já que todos os seus elementos são simétricos. A diferença é que a redução retira funcionalidades do RTL, já que efetivamente elimina *hardware*, enquanto a abstração apenas muda a forma de interpretação. Portanto, enquanto abstrações mantém as funcionalidades originais e permite a descoberta de quaisquer *bugs*, o mesmo não vale para as reduções, podendo essas, assim, mascarar erros na implementação.

Finalmente, a abstração temporal consiste em mapear duas representações de unidades de tempo. "Por exemplo, o contador pode ser especificado em termos do que acontece para cada ciclo de *clock* enquanto a implementação usa um *clock* de duas fases" (ALMEIDA, 2018). Isso exige, claro, que haja uma descrição de como os diferentes domínios de tempo se relacionam.

O uso de suposições (*assumptions*) é outra técnica que pode reduzir a complexidade. Falaremos mais delas no capítulo 3, mas podemos entendê-las como axiomas que a ferramenta irá seguir na hora de escolher as sequências de estados. Quando esses axiomas são escritos de forma a reduzir o espaço de estados, tornando certos bits constantes, por exemplo, a complexidade é reduzida. Podemos utilizar isso para definir casos de verificação. Se um processador tem, por exemplo, 3 modos de operação, podemos usar suposições para fazê-lo operar em apenas um desses modos, o que será mais rápido do que se pudesse alternar à vontade, e então corrigir os *bugs* encontrados antes de passar para o caso mais geral. No entanto, tentar reduzir as transições das entradas primárias pode adicionar ao estado de espaços, então isso deve ser feito com cuidado. Assertivas também podem ajudar a reduzir complexidade: uma assertiva pode ser dividida em duas ou mais, com COI menores, reduzindo sua complexidade e permitindo que sejam executadas paralelamente.

Mais uma técnica que pode ser explorada é o não-determinismo, que consiste no uso de variáveis sem *driver* para diminuir o número de elementos de estado. Uma ferra-

menta formal irá testar todos os valores possíveis para essas variáveis, garantindo uma prova exaustiva. É importante notar que isso só pode ser feito em análises formais, já que na simulação as variáveis sem *driver* tem valor X (indeterminado). Para obter os melhores resultados, utilizamos constantes não-determinísticas (*Non-Deterministic Constant* - NDC), que são aquelas que não possuem *driver*, então podem assumir qualquer valor, mas que esse valor é mantido constante durante todos os ciclos de *clock*, podendo assim ser utilizado em assertivas e código auxiliar a qualquer momento.

Um ponto a se tomar cuidado na hora de escrever as suposições é na criação de "caminhos sem saída" (*dead-ends*). Eles são estados que irão violar as suposições no futuro. Quando se escreve suposições em variáveis que não são as entradas primárias do DUT, a análise formal só é capaz de identificar comportamento ilegal nessas variáveis quando o comportamento acontecer. Portanto, é possível que um estado atual irá gerar um comportamento ilegal no futuro, mas a ferramenta não tem como saber já que ainda não aconteceu. Isso causa esforço inútil, em que sequências são exploradas para posteriormente serem descartadas por ter sido encontrado um comportamento ilegal. Outra causa de caminhos sem saída são suposições conflitantes, ou seja, suposições que fazem com que todas as sequências possíveis a partir do estado inicial chegarão, eventualmente, em um estado em que as suposições não poderão mais ser respeitadas. Como ferramentas de análise formal nunca violam suposições, chegamos em caminhos sem saída. O Jasper™ possui uma propriedade especial chamada `:noDeadEnd`, cujos contra-exemplos encontrados representam caminhos sem saída.

3 SYSTEMVERILOG ASSERTIONS

As regras que o engenheiro de verificação define para o design, ou seja, suas propriedades, em geral, são feitas em uma linguagem de assertivas chamada SystemVerilog Assertions (SVA). Ela é uma sub-linguagem contida no SystemVerilog que pode ser utilizada tanto para verificação formal como dinâmica para validar os comportamentos da implementação. Um exemplo de propriedade escrita em SVA é mostrado no código 3.1. Essa propriedade diz que, na borda de subida do *clock*, gnt ser verdadeiro implica em req ser verdadeiro, ou seja, se observarmos um gnt isso significa que temos um req. Isso descreve um *handshake* comum em design de *hardware* em que temos um *request* (solicitação) e um *grant* (concessão).

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

Código 3.1 – Exemplo de propriedade em SVA.

No entanto, uma propriedade por si só não faz nada. Ele deve ser utilizada por uma diretiva para que a ferramenta saiba o que fazer com ela. As diretivas mais comuns são as seguintes: *assert*, *cover* e *assume*. Cada uma delas terá um funcionamento diferente, dependendo se o código está sendo executado por uma simulação dinâmica ou análise formal.

Uma assertiva (*assertion*), gerada pela diretiva *assert*, indica uma propriedade a qual queremos que seja inerente ao design, ou seja, que o design deve respeitá-la em qualquer situação. Na simulação, em cada *slot* de tempo será verificado se as assertivas no código estão respeitadas, ou seja, se os comportamentos descritos acontecem. Caso negativo, será gerado um erro e informado ao usuário. Já na verificação formal, a ferramenta tentará provar que a assertiva é sempre verdade, através da aplicação exaustiva de estímulos ao modelo do design. Caso ache um exemplo em que isso não acontece, o que chamamos de contra-exemplo, a sequência de estados que gerou a violação será armazenada, geralmente como um arquivo de forma de ondas. Quando não há contra-exemplos, a assertiva é dada como provada. É importante ressaltar que a veracidade dos resultados dependem de que as assertivas estejam corretas, portanto elas devem ser escritas com todo o cuidado possível.

Uma cobertura (*cover*), gerada pela diretiva *cover*, indica uma sequência que você deseja observar acontecendo no design. Ela não precisa ocorrer o tempo todo, mas que-

remos que seja possível que ocorra. Na simulação, em cada *slot* de tempo será verificado se as coberturas aconteceram, e em caso afirmativo um contador irá incrementar. No final da simulação, podemos observar quantas vezes aquela propriedade ocorreu. Já na verificação formal, novamente a ferramenta fará a aplicação de estímulos ao modelo, com o objetivo de gerar a situação desejada. Se for sucedida, ela pode então guardar o exemplo como forma de onda. No Jasper™, as coberturas geram não apenas um exemplo da propriedade acontecendo, mas o exemplo mais curto possível (menos ciclos de *clock*). As coberturas são especialmente úteis para detectar *overconstraints* na análise formal, conceito que discutiremos mais a seguir.

Uma suposição (*assumption*), gerada pela diretiva *assume*, indica uma propriedade que assumiremos que é verdade. Não tentaremos provar que ela é verdade, apenas aceitaremos que o é. Isso é desejável de aplicar a entradas do sistema. Por exemplo, se temos uma entrada que faz parte de algum barramento, podemos assumir que os sinais de entrada se comportam de acordo com o seu protocolo, reduzindo a quantidade de combinações que precisamos testar, ou restringindo a análise a apenas o que nos é relevante. Ou seja, suposições podem reduzir o espaço de estados da análise. Na simulação, esse conceito não faz sentido, já que as entradas são geradas por um *testbench* e os vetores de simulação irão decorrer dessa sequência de entradas. Por isso, as suposições funcionam exatamente da mesma forma que as assertivas na verificação dinâmica. Já na verificação formal, a ferramenta irá usar as suposições para reger a análise, e nunca irá violar nenhuma delas. Sem suposições, a ferramenta testa todas as possibilidades de entradas no sistema. Existe uma outra diretiva, *restrict*, que funciona exatamente como o *assume* na verificação formal e é ignorada na simulação.

O uso de suposições, todavia, pode gerar o problema de *overconstraint*, em que comportamentos válidos ou relevantes são impedidos de acontecer (não intencionalmente), por que a ferramenta não irá desrespeitar uma suposição escrita pelo programador. Além de ser um problema difícil de ser identificado, isso pode até mesmo reduzir o estado de espaços a zero, caso em que não é possível gerar nenhuma forma de onda. Casos como esse nos dará falsos positivos, já que não é possível obter contra-exemplos se não há formas de onda, então devemos tomar muito cuidado ao escrever suposições. É possível, também, restringir estados válidos do design intencionalmente, quando se quer testar algum caso específico. Isso ajuda a reduzir o esforço de *debugging*. Restrições como essa são, claro, temporárias, e devem ser removidas conforme seus testes forem concluídos. A diretiva *restrict* é destinada a ser usada nesses casos de restrições temporárias. É importante ressaltar que, se as suposições são incorretas, os resultados de todas as assertivas serão inválidos.

Mesmo que uma assertiva seja marcada como "provada" ou "contra-exemplo encontrado", isso não nos garante um resultado conclusivo. Os resultados da análise formal

dependem da qualidade das assertivas, bem como do espaço de estados definido pelas suposições, ambos pontos que são definidos pelo código escrito pelo engenheiro de verificação, os quais são, claro, propensos a erros. Por isso, é importante sempre analisarmos os resultados para validar sua relevância.

No código 3.2, mostramos como podemos usar as diretivas mencionadas anteriormente para exercitar a propriedade descrita no código 3.1. A assertiva AST1 irá provar que a propriedade é sempre verdade. A suposição ASM1 irá assumir que a propriedade é sempre verdade, e impedirá a geração de formas de onda em que isso não acontece. A cobertura COV1 irá procurar um exemplo de forma de onda em que a propriedade acontece.

```
1 AST1: assert property (no_gnt_wo_req);
2 ASM1: assume property (no_gnt_wo_req);
3 COV1: cover  property (no_gnt_wo_req);
```

Código 3.2 – Como utilizar propriedades em SVA.

No código 3.2, as diretivas são usadas na forma simultânea (*concurrent*), forma em que é utilizada a palavra chave "*property*", mas também podem ser escritas na forma imediata. Os equivalentes na forma imediata são apresentados no código 3.3. Nesse caso, o código é escrito como linhas normais de SystemVerilog dentro de um bloco *always*, sem utilizar propriedades. As diretivas imediatas também funcionam como linhas normais de SystemVerilog: são checadas proceduralmente e contém apenas expressões booleanas. Em oposição, as diretivas simultâneas são executadas paralelamente aos demais blocos procedurais do código e podem conter sequências de sinais. No código 3.4, temos um exemplo de sequência temporal com o seguinte comportamento: ao observar req, depois de 1 a 4 ciclos devemos observar gnt. O uso de propriedades é considerado um recurso mais avançado de SVA, e ferramentas gratuitas geralmente não tem suporte, ou tem pouco. A versão gratuita do SymbiYosys, por exemplo não tem suporte total a SVA, e o RFVF é construído totalmente baseado em assertivas imediatas.

```
1 always @(posedge clk) begin
2     if (gnt) begin
3         assert (req);
4         assume (req);
5         cover  (req);
6     end
7 end
```

Código 3.3 – Exemplo de propriedade em SVA

```
property gnt_follows_req;
    @(posedge clk) disable iff (!rst_n)
    req |-> ##[1:4] gnt;
endproperty

AST_gnt_follows_req: assert property (gnt_follows_req);
```

Código 3.4 – Exemplo de propriedade com sequência temporal.

Enquanto é possível fazer testes bem relevantes com esse recurso básico de SVA, como verificar formalmente uma arquitetura de processadores, se torna muito mais difícil explorar sequências ou comportamentos mais complexos, muitas vezes exigindo uma grande quantidade de código auxiliar para compensar as limitações. Com isso, a complexidade do código auxiliar por si só já se torna comparável à do próprio design, exigindo o mesmo nível de esforço para fazer o *debugging*. Além disso, deprecia a legibilidade do código já que a intenção normalmente não é óbvia de se entender.

4 NÚCLEO DE PROCESSAMENTO CV32E40P

O núcleo de processamento escolhido para se realizar a verificação foi o CV32E40P. Ele é um *core* desenvolvido pelo grupo OpenHW, um grupo global não-lucrativo no qual os colaboradores desenvolvem IPs de *hardware* e *software* de código aberto (OPENHW. . . , 2024a). Desse modo, o CV32E40P é um processador RISC-V de código aberto, o qual pode ser encontrado em sua página do GitHub (OPENHW. . . , 2024b). Ele é um núcleo com arquitetura de 32 bits, execução em ordem e 4 estágios de *pipeline*. Esses estágios são: *Instruction Fetch* (abreviado como IF), *Instruction Decode* (abreviado como ID), *Execute* (abreviado como EX) e *Writeback* (abreviado como WB). Essas abreviações são utilizadas como sufixos no RTL para identificar os sinais mais facilmente.

A arquitetura RISC-V possui um conjunto base de instruções, denominado RV32I, contendo as instruções julgadas essenciais para o funcionamento minimamente eficiente de um processador, incluindo operações matemáticas básicas, saltos condicionais e incondicionais, instruções de acesso à memória, entre outros. Para ampliar suas funcionalidades, existem diversas extensões ao conjunto de instruções, algumas das quais o CV32E40P implementa. Listando-as, temos:

- Extensão "M": operações de multiplicação e divisão de números inteiros contidos em registradores. Essa extensão é sempre ativa em nosso *core*.
- Extensão "F": operações para números de ponto flutuante de precisão simples, acrescentando também um banco de registradores adicional com 32 registradores para números de ponto flutuante, além de um CSR chamado *fcsr* contendo o modo de operação da unidade aritmética de ponto flutuante. Essa extensão é opcionalmente habilitada através do parâmetro "FPU".
- Extensão "Zfinx": permite o uso dos registradores de números inteiros para armazenar números de ponto flutuante, removendo os registradores extra da extensão "F". É importante atentar que essa extensão ainda não foi oficializada na especificação do RISC-V, mas é indicada a intenção de definí-la no futuro (RISC-V. . . , 2019). Essa extensão é opcionalmente habilitada através do parâmetro "Zfinx".
- Extensão "C": adiciona instruções codificadas em 16 bits ao invés de 32, permitindo a redução do tamanho do código. Essa extensão é sempre ativa.
- Extensão "Zicsr": define operações de acesso aos CSRs. Essa extensão é sempre ativa.

Além disso, o *core* também implementa as extensões customizadas da plataforma PULP (*Parallel Ultra-Low-Power*), uma arquitetura para microcontroladores *multi-core* voltada a aplicações de Internet das Coisas (*Internet of Things* - IOT). As extensões customizadas são as seguintes:

- *Loads* e *Stores* com pós-incremento: instruções de acesso à memória que, adicionalmente, incrementam o endereço utilizado pelo acesso.
- Laços de *Hardware*: acrescenta CSRs adicionais para armazenar endereço de início, fim e contagem de um laço (*loop*), bem como instruções para manipulá-los.
- Extensões da Unidade Lógico-Aritmética (ULA): acrescenta diversas operações matemáticas mais avançadas que aquelas do RV32I, como por exemplo manipulações de bits, extrações de fatias (*slice*) de registradores e fusões de sequências comumente utilizadas.
- Multiplicar e acumular: operações de multiplicar e acumular o resultado, além de multiplicações de meias-palavras (números de 16 bits).
- *Single Instruction, Multiple Data* (SIMD): operações realizadas em múltiplos elementos ao mesmo tempo, dividindo as palavras de dados em pedaços de 8 ou 16 bits os quais são operados individualmente.

É importante salientar que a versão do CV32E40P em que esse trabalho se baseia não é a versão mais nova na data em que é escrito. Como a IP está em desenvolvimento ativo, trabalhar na versão mais recente implicaria a desvalidação do trabalho com a próxima atualização no código. Desse modo, foi decidido trabalhar na versão utilizada pelo PULPissimo, a arquitetura de microcontroladores mais recente da plataforma PULP (PULPISSIMO, 2022). A versão v7.0.0 do PULPissimo, lançada em 25 de Junho de 2021, contém o CV32E40P na versão PULPissimo-v3.4.0, de 18 de Novembro de 2019, a qual é utilizada nesse trabalho. Com isso, podemos esperar que *bugs* encontrados nessa versão já tenham sido corrigidos em versões mais atualizadas. Uma diferença importante que podemos citar rapidamente é que na versão antiga não é possível desabilitar as extensões customizadas, enquanto na nova é. A grande maioria dos erros encontrados e corrigidos no processador pelos seus desenvolvedores são relacionados a essas extensões, e não poder desabilitá-las no RTL torna mais complicada a verificação das funcionalidades mais básicas.

5 IMPLEMENTAÇÃO DO RFVF

A plataforma que usaremos para verificar o processador escolhido é o RISC-V Formal Verification Framework, que se baseia no fluxo de verificação formal do SymbiYosys, mencionado anteriormente. Para garantir a melhor compatibilidade possível com outras ferramentas, as propriedades do *Framework* são expressas totalmente pelo uso de assertivas, suposições e coberturas imediatas. A estrutura de diretórios do repositório da ferramenta é apresentada a seguir:

```
riscv-formal
├── bus
├── checks
├── cores
│   ├── VexRiscv
│   ├── nerv
│   ├── picorv32
│   ├── rocket
│   └── serv
├── docs
├── insns
├── monitor
└── tests
    ├── coverage
    ├── semantics
    └── spike
```

Trataremos das pastas de interesse nesse projeto. Nas pasta `checks`, temos os testes (*checkers*) para cada caso, cada qual contido em um módulo de SystemVerilog. Temos também um *testbench* (`rvfi_testbench.sv`), o qual instancia o teste e o *wrapper* (falaremos deste na seção 5.1). Nessa pasta também encontramos o arquivo de macros utilizadas pela ferramenta (`rvfi_macros.vh`), e o *script* em Python utilizado para gerá-lo (`rvfi_macros.py`). Por fim, encontramos o arquivo `genchecks.py`, o qual é responsável por ler um arquivo de configurações e gerar os testes solicitados para um *core*. Na pasta `cores`, encontramos exemplos de núcleos que foram verificados utilizando a ferramenta. Na pasta `insns`, encontramos os módulos em Verilog implementando a especificação de cada instrução. Quando o teste executado é o `rvfi_insn_check.sv`, ele irá instanciar um desses módulos, testando assim uma instrução a cada vez. Além disso, temos o *script* `generate.py`, que gera os arquivos de cada instrução, bem como uma lista de instruções para cada ISA que a ferramenta suporta.

O RISC-V formal possui dois tipos de testes, os testes de instrução e os testes de consistência. Os testes de instrução correspondem à grande maioria dos testes gerados, pois é gerado um teste para cada instrução presente na ISA. Eles checam se os sinais da interface RVFI (seção 5.2) correspondem à especificação para a instrução em questão. Existe um *checker* para instruções gerais e outro específico para instruções de ler e modificar CSRs. Já os principais testes de consistência que são gerados pelo RFVF são:

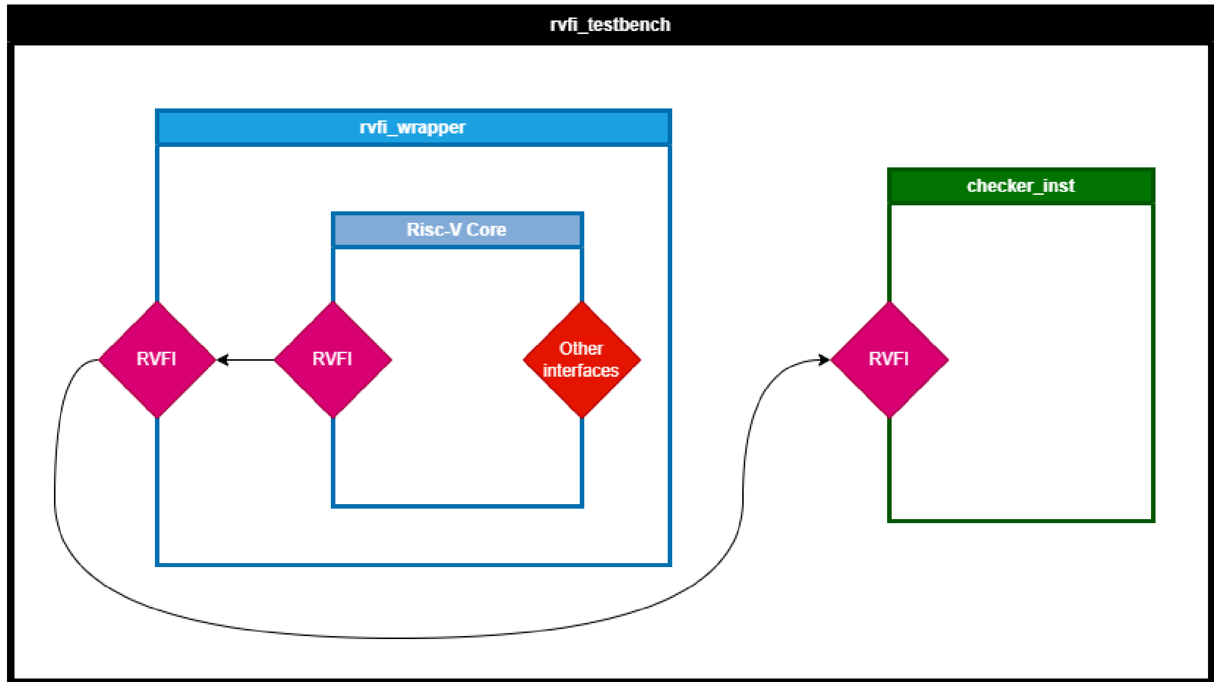
- Teste de registradores: checa se um valor escrito a um registrador corresponde ao valor lido do mesmo registrador em instruções posteriores.
- Teste de causalidade: causalidade significa que se uma instrução escreve um dado em um registrador e uma instrução posterior irá ler esse mesmo registrador, elas devem ser processadas na ordem correta. Isso pode não acontecer em processadores com execução fora de ordem, o que não é o caso do nosso DUT, logo esse teste checa se a causalidade é respeitada.
- Teste de unicidade: checa se o processador não gera duas instruções com o mesmo índice de ordem (`rvfi_order`). Como o valor desse sinal deve ser único para cada instrução válida na interface, podemos interpretar isso como não devem haver instruções repetidas.
- Teste de contador de programa: o teste `pc_fwd` checa se a instrução atual tem o próximo contador de programa correspondente à instrução anterior. O teste `pc_bwd` serve para execução fora de ordem, quando uma instrução posterior é executada antes de uma instrução anterior, e checa se a instrução posterior tem o próximo contador de programa correspondente à instrução anterior.
- Teste de *liveness*: checa se o processador nunca trava, ou seja, se é possível haver traçados de onda de comprimento infinito.
- Teste de falha (*fault*): checa que acessos falhos à memória são tratados.

5.1 WRAPPER

Após criar a pasta `cores/cv32e40p` e clonar o RTL e as IPs utilizadas para essa pasta, o primeiro passo é escrever um arquivo de *wrapper*. Esse arquivo é um módulo em SystemVerilog que deve ser escrito para cada processador que se deseja conectar a ferramenta. O *testbench* instancia um *wrapper* para poder padronizar sua interface, e assim não ser forçado a se conectar aos sinais específicos de cada *core*. O *wrapper* é, então, responsável pela instância do *core*, fornecendo a ele os sinais de *clock* e *reset*. Todos os sinais internos, que se conectarão ao *core*, devem ser fios livres ou designados a um valor constante de *tie-off*, no caso de entradas. As saídas, exceto aquelas da interface

RVFI que definiremos a seguir, podem ser deixadas desconectadas ou conectadas a fios quaisquer, pois não serão utilizadas pelo RFVF. Para melhor representar a hierarquia do ambiente, a apresentamos em forma de diagrama na figura 1.

Figura 1 – Hierarquia de módulos do RISC-V Formal.



Fonte: autoria própria.

No *wrapper* que fizemos para instanciar o CV32E40P, como é exigido pelo *testbench*, temos sinais de entrada de *clock* e *reset*, além dos sinais de saída da interface RVFI, instanciados automaticamente pela macro `` `RVFI_OUTPUTS ``. Definimos os parâmetros do *core* com os mesmos valores que ocorrem na instanciação desse módulo no PULPissimo. O único grau de liberdade que o PULPissimo permite é a habilitação ou não da unidade de ponto flutuante, e consequentemente da extensão "F", a qual deixamos desabilitada em nosso ambiente. Alguns *tie-offs* são feitos, os quais são mostrados no código 5.1.1.

É no *wrapper* que podemos adicionar as nossas suposições, caso julguemos necessário. As interfaces das memórias de instruções e dados do CV32E40P obedecem ao *OpenBus Interface* (OBI) (OBI, 2022), o qual estabelece um *handshake* simples de *request* e *grant*. Para estar em conformidade com a interface, foram criadas suposições para garantir que não haverão *grants* sem *requests*. Essas suposições são apresentadas no código 5.1.2, contruídas com formato imediato. As suposições nas linhas 5 e 6 garantem que o dispositivo subordinado (memória) não levantará um *grant* a menos que o dispositivo gerente (processador) levante um *req*. Já as linhas 27 e 29 garantem que o dispositivo subordinado não levantará o sinal de resposta válida se não houverem transações inacabadas.

```
// Tie-offs
assign clk_i               = clock;
assign rst_ni              = !reset;
assign clock_en_i          = 1'b1;
assign test_en_i           = 1'b0;
assign fregfile_disable_i = 1'b0;
assign boot_addr_i         = BOOT_ADDR;
assign core_id_i           = CORE_ID;
assign cluster_id_i        = CLUSTER_ID;
// Disable interrupts for now (they are asynchronous)
assign irq_i      = 1'b0;
assign irq_id_i   = 5'b0;
assign irq_sec_i = 1'b0;
// Disable halts requested by debug module (for now)
assign debug_req_i = 1'b0;
// Fetches are always enabled
assign fetch_enable_i = 1'b1;
// Disable external performance counters
assign ext_perf_counters_i = '0;
```

Código 5.1.1 – Tie-offs no wrapper

5.2 RISC-V FORMAL INTERFACE

Como brevemente mencionado, o RTL do processador deve possuir uma certa interface para se comunicar com o nosso *testbench*. Essa é a *RISC-V Formal Interface* (RVFI), que define uma série de sinais que conterão informações sobre as instruções executadas. Do ponto de vista do *core*, todos esses sinais são saídas, e podem ser facilmente incluídos com a macro `‘RVFI_OUTPUTS`. Além de incluir as novas conexões, é necessário, claro, fazer o seu *drive*. Essa foi, certamente, a tarefa que exigiu mais esforço ao implementar o RISC-V Formal no CV32E40P, uma vez que é necessário ter um conhecimento considerável com relação ao funcionamento do RTL. Os sinais da interface estão listados no código 5.2.1, em que NRET é o número máximo de instruções que podem ser finalizadas em um mesmo ciclo de *clock* (será diferente de 1 para processadores superescalares), ILEN é o tamanho máximo de uma instrução e XLEN é o tamanho de um registrador do banco de registradores. No nosso caso, esses valores são 1, 32 e 32, respectivamente. Podem ser acrescentados outros sinais dependendo dos *defines* utilizados, os quais são geralmente definidos no arquivo de configurações. Descreveremos brevemente os sinais da RVFI a seguir.

- `rvfi_valid`: deve ser nível lógico alto quando a execução de uma instrução é finalizada. Todos os outros sinais da interface devem ter valores válidos quando `rvfi_valid` é 1. Como o CV32E40P não possui um mecanismo claro para infor-

```
// No grant without request
wire instr_gnt_wo_req = instr_gnt_i && !instr_req_o;
wire data_gnt_wo_req  = data_gnt_i  && !data_req_o ;
always @(posedge clock) begin
    ASM_no_gnt_wo_req_instr: assume (!instr_gnt_wo_req);
    ASM_no_gnt_wo_req_data:  assume (!data_gnt_wo_req );
end
// No valid response without pending transaction
reg instr_trans_pnd;
reg data_trans_pnd;
always @(posedge clock or posedge reset) begin
    if (reset) begin
        instr_trans_pnd <= 1'b0;
        data_trans_pnd  <= 1'b0;
    end else begin
        if (instr_gnt_i && instr_req_o)
            instr_trans_pnd <= 1'b1;
        else if (instr_rvalid_i)
            instr_trans_pnd <= 1'b0;

        if (data_gnt_i && data_req_o)
            data_trans_pnd <= 1'b1;
        else if (data_rvalid_i)
            data_trans_pnd <= 1'b0;

        if (!instr_trans_pnd)
            ASM_no_rvld_wo_pnd_instr: assume (!instr_rvalid_i);
        if (!data_trans_pnd)
            ASM_no_rvld_wo_pnd_data: assume (!data_rvalid_i);
    end
end
```

Código 5.1.2 – Suposições para as interfaces de memória.

mar quando uma instrução termina, o desenvolvimento do *driver* desse sinal exigiu bastante tempo e testes, e continua não sendo ideal.

- `rvfi_insn`: corresponde ao código binário da instrução executada. O DUT propaga essa informação do estágio IF para o ID, mas não além disso. Isso também torna complexa a identificação da instrução que está no estágio de WB a cada ciclo.

- `rvfi_trap`: deve ser nível lógico alto quando a instrução não pode ser decodificada como instrução legal, quando um acesso desalinhado à memória é feito em sistemas que não o permitem e quando o alvo de um salto é um endereço desalinhado.

- `rvfi_halt`: deve ser nível lógico alto quando a instrução é a última executada antes que o DUT pare sua execução.

```
1  // Instruction Metadata
2  output [NRET        - 1 : 0] rvfi_valid
3  output [NRET *   64 - 1 : 0] rvfi_order
4  output [NRET * ILEN - 1 : 0] rvfi_insn
5  output [NRET        - 1 : 0] rvfi_trap
6  output [NRET        - 1 : 0] rvfi_halt
7  output [NRET        - 1 : 0] rvfi_intr
8  output [NRET * 2    - 1 : 0] rvfi_mode
9  output [NRET * 2    - 1 : 0] rvfi_ixl
10 // Integer Register Read/Write
11 output [NRET *    5 - 1 : 0] rvfi_rs1_addr
12 output [NRET *    5 - 1 : 0] rvfi_rs2_addr
13 output [NRET * XLEN - 1 : 0] rvfi_rs1_rdata
14 output [NRET * XLEN - 1 : 0] rvfi_rs2_rdata
15 output [NRET *    5 - 1 : 0] rvfi_rd_addr
16 output [NRET * XLEN - 1 : 0] rvfi_rd_wdata
17 // Program Counter
18 output [NRET * XLEN - 1 : 0] rvfi_pc_rdata
19 output [NRET * XLEN - 1 : 0] rvfi_pc_wdata
20 // Memory Access
21 output [NRET * XLEN   - 1 : 0] rvfi_mem_addr
22 output [NRET * XLEN/8 - 1 : 0] rvfi_mem_rmask
23 output [NRET * XLEN/8 - 1 : 0] rvfi_mem_wmask
24 output [NRET * XLEN   - 1 : 0] rvfi_mem_rdata
25 output [NRET * XLEN   - 1 : 0] rvfi_mem_wdata
```

Código 5.2.1 – Sinais básicos da RVFI.

- `rvfi_intr`: deve ser nível lógico alto quando a instrução é a primeira de um tratador de exceção (*trap handler*).
- `rvfi_mode`: corresponde ao nível de privilégio do DUT para a instrução executada. O RISC-V define 4 níveis de privilégio: máquina, hipervisor, supervisor e usuário, do nível de maior acesso ao menor.
- `rvfi_ixl`: corresponde ao valor de campo MXL (*Machine XLEN*) do CSR *misa*.
- `rvfi_r(s1|s2|d)_addr`: é o endereço dos registradores rs1, rs2 e rd, respectivamente, utilizados pela instrução.
- `rvfi_r(s1|s2)_rdata`: é o dado lido do banco de registradores a partir do endereço rs1 e rs2, respectivamente.
- `rvfi_rd_wdata`: é o dado a ser escrito no banco de registradores no endereço rd com a execução da instrução.

- `rvfi_pc_(r|w)data`: é o valor atual do contador de programa e o valor do contador de programa da próxima instrução a ser executada, respectivamente. A falta de suporte do RFVF aos laços de *hardware* faz com que a informação no `rvfi_pc_wdata` seja inconsistente. Isso será explanado no capítulo 6.
- `rvfi_mem_addr`: é o endereço de memória no qual é realizada uma operação de *load* ou *store*.
- `rvfi_mem_(r|w)mask`: é uma máscara indicando quais bytes de `rvfi_mem_rdata` ou `rvfi_mem_wdata`, respectivamente, são válidos.
- `rvfi_mem_(r|w)data`: é o dado lido ou escrito, respectivamente, do endereço `rvfi_mem_addr`.

A adição da RVFI à lista de portas do *core* e o *drive* de seus sinais foram as únicas modificações feitas no RTL. Afinal, não é objetivo da verificação fazer alterações no design, existem times diferentes para isso. Todas essas alterações estão dentro de estruturas "*ifdef*", então só terão qualquer efeito quando executadas com os *defines* do RFVF.

5.3 CONFIGURAÇÃO

Com o *wrapper* feito e a interface implementada, tudo que resta para executar a ferramenta é um arquivo de configurações. Por convenção, esse arquivo é chamado de `checks.cfg`. O arquivo de configurações criado para o CV32E40P é apresentado resumidamente no código 5.3.1. Podemos notar que ele é separado em seções. São elas:

- `options`: opções referentes ao DUT. Utilizamos apenas a opção que define a ISA. RV32I seria o conjunto base de instruções, incluindo também as extensões "M" e "C", que, no momento da escrita desse trabalho, são as únicas suportadas pelo RFVF. O "x" se refere à extensão customizada da plataforma PULP, a qual não é nativa do RFVF, nós mesmos adaptamos a ferramenta para oferecer suporte.
- `depth`: define quais testes serão criados e os ciclos de *clock* em que alguns eventos acontecerão. O número mais à direita, por exemplo, é a profundidade do teste (*depth*). Em outras palavras, define o ciclo de *clock* em que será feita a checagem das assertivas.
- `sort`: define a ordem que os testes aparecerão no Makefile, e, consequentemente, serão executados.
- `defines`: contém definições que serão incluídas no arquivo `defines.sv`.

```
[options]
isa rv32imcx

[depth]
insn            20
reg        10    25
causal     10    30
unique     1  10 30
pc_fwd     10    30
pc_bwd     10    30
csrw            20

[sort]
insn_.*
csrw
reg
causal
unique
pc_fwd
pc_bwd

[defines]
`define RISCV_FORMAL_UMODE
`define RISCV_FORMAL_CUSTOM_ISA

[csrs]
misa
mstatus
...
mscratch

[custom_csrs]
7C0 mu hwlp_start0
7C1 mu hwlp_end0
7C2 mu hwlp_counter0
7C4 mu hwlp_start1
7C5 mu hwlp_end1
7C6 mu hwlp_counter1

[verilog-files]
# List of Verilog/SystemVerilog files

[include-dirs]
# List of included directories
```

Código 5.3.1 – Arquivo checks.cfg.

- `csrs`: lista os CSRs oficiais da arquitetura RISC-V que o DUT implementa. Não são todos os CSRs oficiais que o RFVF suporta, logo a compatibilidade deve ser verificada.
- `custom_csrs`: aqui é onde podem ser adicionados os CSRs não-oficiais. Utilizamos essa seção para incluir os CSRs relacionados aos laços de *Hardware*. O *core* permite a implementação de até dois laços simultâneos (aninhados), associados aos índices 0 e 1. Para cada índice, existe um CSR de início, fim e contagem.
- `verilog-files`: contém a lista dos arquivos fonte contendo o RTL do design.
- `include-dirs`: contém a lista de pastas com arquivos a serem incluídos com `‘include`. Essa seção não é nativa do RISC-V Formal, foi adicionada por nós para a conveniência que traria ao nosso caso.

Com o arquivo de configurações pronto, trabalhando no diretório do *core*, deve-se usar o *script* `checks/genchecks.py`, que irá ler o `checks.cfg` e criar uma pasta chamada `checks` com todos os testes solicitados, além de um Makefile para executá-los. Desse modo, `checks/genchecks.py` deve ser chamado de um subdiretório de `cores`, no nosso caso `cores/CV32E40P`.

6 CUSTOMIZAÇÕES DA FERRAMENTA

Para tornar a ferramenta mais adequada à verificação do *core* escolhido, tornou-se necessária a sua expansão, de modo a acomodar as extensões não-oficiais da plataforma PULP. Na versão utilizada, as extensões são unificadas em uma única extensão denominada xpulp. Além disso, também nos propomos a adaptar a ferramenta para o uso com o Jasper™, ferramenta de análise formal da Cadence Design Systems®. A seguir, entraremos em detalhes de como foram feitas as customizações.

6.1 EXPANSÃO DA RVFI

O primeiro passo foi expandir a RVFI, acrescentando os seguintes sinais:

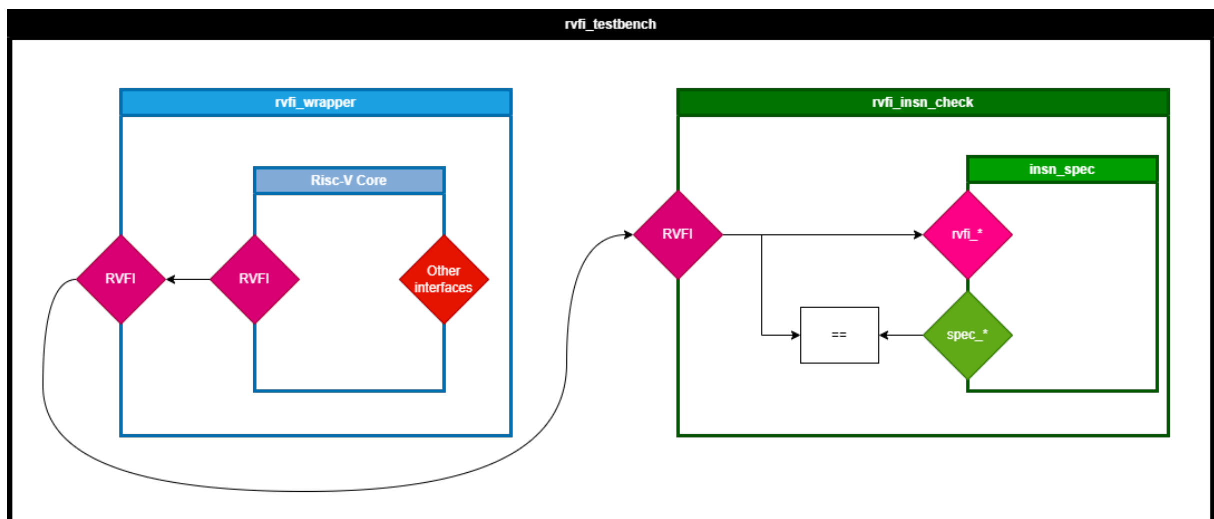
- `rvfi_rs3_addr`: é o endereço do registrador rs3 utilizado pela instrução. Certas instruções do xpulp utilizam 3 registradores fonte. Por exemplo: a *store* com pós-incremento utiliza o rs1 como endereço, rs2 como dado a ser armazenado na memória e rs3 como incremento. Algumas instruções, como a `p.insert`, utilizam o rd tanto como registrador destino como fonte. Nesses casos, o seu valor é lido pelo rs3.
- `rvfi_rs3_rdata`: é o dado lido do banco de registradores a partir do endereço rs3.
- `rvfi_post_rd_addr`: é o endereço do registrador rd utilizado como destino pelas instruções de *load* e *store* com pós-incremento. O único registrador rd definido no RISC-V torna-se insuficiente para a instrução `p.lw` e suas variantes porque ela escreve o dado vindo da memória em um registrador e o pós-incremento em outro, tornando necessário um segundo registrador de destino.
- `rvfi_post_rd_wdata`: é o dado a ser escrito no registrador do endereço pelo pós-incremento.
- `rvfi_is_hwlp`: indica se a instrução atual é o fim de um laço de *hardware*. Essa informação é utilizada para decidir se o contador de programa da próxima instrução será PC+4 ou o valor armazenado em um dos CSRs `hwlp_start`. Até o presente momento, não foi possível definir um drive consistente para esse sinal. Isso implica que as assertivas do `rvfi_pc_wdata` falha em todos os testes.
- `rvfi_hwlp_start`: quando o contador de programa não é PC+4 por causa do `rvfi_is_hwlp` ser nível lógico alto, o `rvfi_pc_wdata` deverá ser igual a `rvfi_hwlp_start`. Esse valor deve ser igual a um dos valores lidos dos CSRs `hwlp_start0` e `hwlp_start1`.

Como mencionamos, utilizamos a macro `RVFI_OUTPUTS` para listar automaticamente todas as saídas da RVFI no RTL. Outras macros também são utilizados por diversos arquivos da ferramenta. Naturalmente, precisamos atualizar essas macros para incluir os novos sinais da interface. O arquivo que define as macros no RFVF é o `checks/rvfi_macros.vh`, o qual é gerado por `checks/rvfi_macros.py`. Nós, portanto, copiamos esse *script* e renomeamos para `checks/rvfi_macros_xpulp.py`. O novo arquivo foi editado para gerar as macros corretamente com as adições do xpulp. Para que as alterações tenham efeito, é necessário que seja definido `RISCV_FORMAL_CUSTOM_ISA` no arquivo de definições, o que é feito com a edição da seção `defines` do arquivo de configurações. Podemos observar isso no código 5.3.1. Fizemos dessa forma para que nossa versão customizada não perca a compatibilidade com outras arquiteturas que não implementam o xpulp.

6.2 TESTES PARA INSTRUÇÕES XPULP

O RFVF possui um teste para cada instrução da ISA base e das extensões suportadas, que no momento são a "M" e "C". Esses testes se encontram na pasta `insns`, e cada um deles recebe como entrada parte dos sinais da RVFI e entrega como saída os valores corretos de outros sinais da RVFI. Os valores corretos devem então ser comparados com os valores vindos do DUT para realizar a verificação. Um diagrama da hierarquia dos módulos é apresentado na figura 2.

Figura 2 – Hierarquia dos módulos em um teste de instrução.



Fonte: autoria própria.

Já que expandimos a RVFI, precisamos adaptar os testes das instruções. Para fazer isso, precisamos editar o arquivo `insns/generate.py`, responsável por gerar todos os testes das instruções, bem como uma lista para cada ISA possível (rv32imc, rv32ix, etc). Copiamos o `insns/generate.py` e renomeamos para `insns/generate_xpulp.py`

para realizar as customizações. Foram adicionadas as portas listadas no código 6.2.1. Além disso, substituímos, em algumas instruções, o *assign* padrão de `rvfi_pc_wdata` (normalmente PC+4) para acomodar os laços de *hardware*. Não fizemos isso em todas porque não foi possível implemetar um *drive* adequado para os sinais relacionados.

```
1  input  [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_rs3_rdata,
2  input                                rvfi_is_hwlp,
3  input  [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_hwlp_start,
4
5  output [                       4 : 0] spec_rs3_addr,
6  output [                       4 : 0] spec_post_rd_addr,
7  output [`RISCV_FORMAL_XLEN   - 1 : 0] spec_post_rd_wdata,
```

Código 6.2.1 – Sinais adicionados aos arquivos de instruções.

Como a extensão xpulp acrescenta dezenas de novas instruções, não podemos considerar o *core* verificado a menos que chequemos todas elas. Logo, iniciamos o desenvolvimento de testes para elas, e, até o presente momento, finalizamos todas as instruções de *load* e *store* com pós-incremento e as extensões da ULA, além de parte das instruções de SIMD. Na medida que novas mudanças são feitas na interface ou novas instruções são adicionadas ao *script*, basta executá-lo novamente para gerar tudo.

Além de editar os arquivos `insn_*.v` já existentes e criar os novos para a extensão xpulp, precisamos editar `checks/rvfi_insn_check.sv`, o qual instancia os testes de instrução. Foram adicionadas assertivas para `spec_rs3_addr`, `spec_post_rd_addr` e `spec_post_rd_wdata`. Além disso, atribuímos nomes a todas as assertivas, coberturas e suposições, o que ajuda bastante na sua identificação. No código 5.1.2, podemos identificar os nomes das suposições nas linhas 6 e 7, antes dos dois pontos.

6.3 GERADOR DE TESTES PARA O JASPER™

Uma das proposições desse trabalho foi adaptar o RISC-V Formal para gerar *scripts* em *Tool Command Language* (tcl) para funcionarem junto ao Jasper™, possibilitando o uso das duas ferramentas. No RFVF, o *script* que gera os testes para o SymbiYosys é o `checks/genchecks.py`, o qual copiamos e renomeamos para `checks/genchecks_jg.py`. O `checks/genchecks.py` cria uma pasta chamada `checks` no diretório em que é chamado. Dentro desse diretório, são gerados diversos arquivos ".sby", um para cada teste definido no arquivo de configurações, além de um Makefile para executar todos os testes.

Um exemplo de *script* ".sby" para um teste de instrução `add` pode ser encontrado no código 6.3.1. Na seção `[options]`, encontramos algumas configurações utilizadas pelo SymbiYosys. Podemos destacar as opções `depth` e `skip`. `depth` é o número de ciclos de

```
[options]
mode bmc
expect pass,fail
append 0
depth 21
skip 20

[engines]
smtbmc boolector

[script]
read -sv insn_add_ch0.sv path/to/file1.sv ... path/to/fileN.sv
prep -flatten -nordff -top rvfi_testbench
chformal -early

[files]
riscv-formal/cores/core_name/../../checks/rvfi_macros.vh
riscv-formal/cores/core_name/../../checks/rvfi_channel.sv
riscv-formal/cores/core_name/../../checks/rvfi_testbench.sv
riscv-formal/cores/core_name/../../checks/rvfi_insn_check.sv
riscv-formal/cores/core_name/../../insns/insn_add.v

[file defines.sv]
`define RISCV_FORMAL
`define RISCV_FORMAL_NRET 1
`define RISCV_FORMAL_XLEN 32
`define RISCV_FORMAL_ILEN 32
`define RISCV_FORMAL_RESET_CYCLES 1
`define RISCV_FORMAL_CHECK_CYCLE 20
`define RISCV_FORMAL_CHANNEL_IDX 0
`define RISCV_FORMAL_CHECKER rvfi_insn_check
`define RISCV_FORMAL_INSN_MODEL rvfi_insn_add
`include "rvfi_macros.vh"

[file insn_add_ch0.sv]
`include "defines.sv"
`include "rvfi_channel.sv"
`include "rvfi_testbench.sv"
`include "rvfi_insn_check.sv"
`include "insn_add.v"
```

Código 6.3.1 – Script SBY para o teste insn_add.

clock que são considerados na análise, ou seja, não são procurados contra-exemplos com comprimento maior que o número especificado. `skip` é o número de ciclos de *clock* que são ignorados ao analisar as assertivas. É relevante notar que `skip` é igual ao valor de profundidade definido para o teste no arquivo de configurações, e `depth` é igual a esse

valor mais 1. Na seção `[script]` é onde são lidos os arquivos fonte e o modelo para análise é elaborado. A seção `[files]` contém arquivos que serão copiados para o diretório de trabalho e serão utilizados como fonte ou incluídos em outros arquivos pela diretiva `` `include``. Uma das funções do SymbiYosys é a capacidade de criar arquivos a partir do próprio *script*, o que podemos observar sendo feito pelas seções `[file defines.sv]` e `[file insn_add_ch0.sv]`, que criam os arquivos `defines.sv` e `insn_add_ch0.sv`, respectivamente, com as mesmas linhas que aparecem na seção verbatim.

Já no código 6.3.1, temos um exemplo de *script* ".tcl" para o mesmo teste. Nós utilizamos a variável `depth` com o mesmo objetivo que no caso anterior, a qual é usada nos comandos `set_prove_target_bound` e `set_max_trace_length`. O comando `analyze` é responsável por ler os arquivos fontes, podendo conter também diretórios para incluir através da sintaxe "`+incdir+path/to/directory`". O comando `elaborate` faz a elaboração do o modelo para análise formal. Os comandos `clock` e `reset`, como os nomes sugerem, servem para indicar os sinais de *clock* e *reset*, respectivamente. `set_trace_optimization` garante que os traçados de forma de onda gerados conterão todos os sinais do ambiente, o que é necessário porque o padrão é salvar apenas os sinais diretamente ligados à propriedade em questão. `prove` irá realizar as provas das assertivas e coberturas. Usamos argumentos para escolher o que será provado, que no caso são apenas as propriedades do teste do RISC-V Formal, e para salvar os traçados de forma de onda dos resultados. Isso salva os resultados de ambos sucessos e falhas das assertivas, diferente do que acontece no SymbiYosys em que apenas falhas são salvas.

Algo que se mostrou um empecilho ao se utilizar o SymbiYosys foi a ausência de um recurso para incluir arquivos e diretórios, como o que mencionamos no parágrafo anterior para o Jasper™. O que encontramos na documentação que permitisse fazer algo semelhante foi a inserção da lista desses arquivos na seção `[files]`, o que implica que todos esses arquivos serão copiados para o diretório de trabalho do teste. Isso é ineficiente do ponto de vista de utilização de espaço, já que cria dezenas ou até centenas de cópias dos mesmos arquivos. Apesar de ser possível, o RFVF não permite fazer isso de forma automática através do arquivo de configurações, então achamos conveniente adicionar a funcionalidade por conta própria. É possível utilizar *links* simbólicos para evitar o uso desnecessário de espaço de armazenamento, mas, até o momento, não implementamos essa customização. Criamos então a seção `[include-dirs]` no arquivo de configurações, que é processada automaticamente pelo *script* `checks/genchecks_jg.py`, no caso do Jasper™. No caso do SymbiYosys, para não alterar o trabalho original em `checks/genchecks.py`, criamos o `checks/insert_incdirs.py` que permite acrescentar a lista de arquivos na seção `[files]` do *script*.

O SymbiYosys gera um arquivo chamado *PASS*, *FAIL* ou *ERROR* ao fim da execução de uma análise formal, a depender de seu resultado. Para imitar esse comportamento,

```
clear -all

set depth 21

analyze -sv12 \
    insn_add_ch0.sv \
    path/to/file1.sv \
    ... \
    path/to/fileN.sv

elaborate -top rvfi_testbench -no_preconditions

clock clock
reset reset

set_prove_target_bound $depth
set_max_trace_length $depth
set_trace_optimization standard
prove -instance checker_inst -dump_trace -dump_trace_type vcd -dump_trace_dir traces

report -summary
```

Código 6.3.2 – Script tcl para o teste insn_add.

criamos, adicionalmente o *script* `checks/get_jg_summary.py`. O programa lê o arquivo de *log* do Jasper™ e decide se o resultado foi sucesso (se não há contra-exemplos para as assertivas e todas as coberturas foram atingidas), falha (caso as condições de sucesso não sejam atendidas) ou erro (caso não seja possível achar um sumário ou não existam assertivas).

```
`rvformal_rand_const_reg [31:0] ndc_1;
          rand const reg [31:0] ndc_2;

                     reg [31:0] ndc_3;
ASM_ndc_3_const: assume property (@(posedge clock) ndc_3 == $past(ndc_3));
```

Código 6.3.3 – Diferentes formas de NDC.

Outro ponto a se levar em consideração na hora de gerar os testes para o Jasper™ é o modo como o SymbiYosys identifica NDCs. O RFVF utiliza da macro mostrado na linha 1 do código 6.3.3 para declarar NDCs, o qual pode ser expandido para o que observamos na linha 2, quando o Yosys é utilizado. Essa ferramenta utiliza a palavra-chave `rand` para identificar que o valor da variável é aleatório e a palavra-chave `const` para identificar que o valor não deve mudar no decorrer da análise. Quando o Yosys não é utilizado, a mesma macro é expandida para o que observamos na linha 4, logo, é isso que

o Jasper™ observará no código. Precisamos garantir que essas NDCs funcionem como esperado no Jasper™. Por padrão, variáveis sem *driver* no Jasper™ já são consideradas aleatórias, então tudo que falta é indicar que o valor da variável não deve mudar. Para isso, utilizamos a suposição lida na linha 5. Nosso programa `checks/genchecks_jg.py` identifica o uso dessa macro nos testes e adiciona uma suposição no arquivo tcl para todas as NDCs identificadas.

7 DIFERENÇAS ENTRE O JASPER™ E O SYMBIYOSYS

O SymbiYosys, por ser um *software* gratuito, apresenta muito menos recursos que o Jasper™. Falaremos brevemente sobre alguns deles a seguir. Existe um recurso chamado *Jasper Expert System*, uma aplicação que contém informações sobre boas práticas de verificação formal. Ele analisa o ambiente de verificação automaticamente, e pode ser consultado para obter dicas e recomendações para obter melhores resultados e sobre como utilizar a própria ferramenta. Ele pode, inclusive, proporcionar soluções para problemas comuns e ajudar a reduzir a complexidade das assertivas, permitindo que o usuário consiga mudar um resultado indeterminado para um resultado concreto. Para ajudar a identificar os focos de complexidade, também há o *Complexity Manager*, o qual tem a capacidade de informar todos os elementos no COI de uma propriedade, podendo filtrá-los entre fios, portas lógicas, registradores, *arrays*, entre outros.

O Jasper™, apesar de permitir que tudo seja feito a partir de *scripts* tcl e linha de comando, possui uma interface gráfica muito completa, permitindo uma maior facilidade de apuração de resultados e organização do trabalho. No *Formal Property Verification* App, que é o App que usamos para as nossas provas, podemos adicionar novas propriedades diretamente da interface gráfica, sem precisar editar o código diretamente, agilizando o processo. Essas propriedades podem ser posteriormente exportadas para serem adicionadas no código. Já a interface do SymbiYosys contém apenas recursos básicos, mas que já facilitam o trabalho, como a capacidade de dar "*play*" no teste que quiser a partir de uma lista, um visualizador de arquivos, e abrir o visualizador de forma de onda com apenas um *click*.

Ainda relacionado à parte gráfica, o Jasper™ possui um visualizador interno de formas de onda, chamado Visualize, enquanto o SymbiYosys depende de ferramentas externas, como o GTKWave, para visualizar os traçados gerados. Não apenas isso, como o Visualize possui uma gama de recursos para facilitar o *debug* de contra-exemplos. Temos a possibilidade de aplicar restrições diretamente do Visualize e re-desenhar as formas de onda com as condições que desejarmos. Temos a função "*quiet trace*", que procura o traçado com o menor número de transições possível que gere um contra-exemplo, melhorando a visualização do problema. Temos a função "*Why?*", que nos mostra por que um sinal apresenta determinado valor em qualquer ciclo, destacando os sinais que contribuíram para aquele resultado. Temos as funções "*Driver*" e "*Loads*", que identifica e destaca o *driver* e as cargas, respectivamente, de algum sinal. Esses são apenas os recursos do Visualize que os autores estão acostumados a usar, existem vários outros.

O Jasper™ tem a funcionalidade de gerar coberturas relacionadas às assertivas automaticamente. Por padrão, ele sempre cria coberturas para as pré-condições de uma assertiva durante a elaboração. Se, por exemplo, temos uma assertiva dentro de um bloco `if`, a pré-condição para essa assertiva é que a expressão avaliada pelo `if` seja verdadeira, então é criada uma cobertura que mostra esse evento acontecendo. Podemos adicionar uma opção ao comando `elaborate` para criarmos coberturas de *witness* ao invés de pré-condições, a qual mostra não apenas a pré-condição acontecer, mas toda a sequência temporal da assertiva. Isso nos ajuda a visualizar o que acontece em um exemplo da nossa assertiva passando.

Com relação à performance, criamos um grupo com 12 instruções para compararmos o tempo levado por cada ferramenta para concluir sua execução. As instruções nesse grupo são: `add`, `auipc`, `blt`, `c.j`, `c.jal`, `c.jalr`, `c.jr`, `jal`, `jalr`, `lh`, `p.sw` e `sh`. O Jasper™ levou 32min37s para concluir todos os testes. Já o SymbiYosys levou 25min19s. Então, apesar do Jasper™ possuir muito mais recursos, a ferramenta mais simples acabou apresentando melhor performance em nosso teste.

8 BUGS ENCONTRADOS

Com a nossa verificação, fomos capazes de identificar alguns erros no CV32E40P. Os demonstraremos a seguir. É relevante notar que todos os erros encontrados são relacionados às extensões personalizadas da plataforma PULP. Primeiro, gostaríamos de apontar certos erros na documentação das instruções xpulp.

- `p.clb rd, rs1`

  A instrução `p.clb` significa *count leading bits*. Segundo a documentação: rd recebe o número de 1's ou 0's consecutivos em rs1 a partir do bit mais significativo, ou 0 se rs1 = 0. O que realmente acontece: retorna o valor descrito anteriormente, menos 1, se rs1 != 0. Se procurarmos na documentação da versão mais nova do *core*, vemos que o texto foi esclarecido.

- `p.extract rd, rs1, ls3, ls2`

  A instrução `p.extract` tem o objetivo de extrair uma fatia de rs1 e salvá-la em rd, de modo que $rd = extS(rs1[ls3 + ls2 : ls2])$, em que extS significa extensão de sinal e com as devidas correções caso ls3+ls2 > 31 bits. Na documentação é apresentada a seguinte expressão, a qual está errada:

  rd = extS((rs1 & ((1 « ls3) - 1) « ls2) » ls2).

  Uma expressão mais correta seria:

  rd = extS((rs1 & ((2 « ls3) - 1) « ls2) » ls2).

  Erros semelhantes ocorrem nas variantes `p.extractu`, `p.extractr` e `p.extractur`.

- `p.insert rd, rs1, ls3, ls2`

  A instrução `p.insert` tem o objetivo de inserir uma fatia de rs1 em rd, sem alterar os bits restantes, de modo que $rd[ls3+ls2 : ls2] = rs1[ls3 : 0]$, com as devidas correções caso ls3+ls2 > 31 bits. Na documentação é apresentada a seguinte expressão, a qual está errada:

  rd = rd | (rs1[ls3:0] « ls2).

  Uma expressão mais correta seria:

  rd = (rd & ~((2 « ls3) - 1) « ls2)) | (rs1[ls3:0] « ls2).

  Erro semelhante ocorre na variante `p.insertr`.

- `p.clipr rd, rs1, rs2`

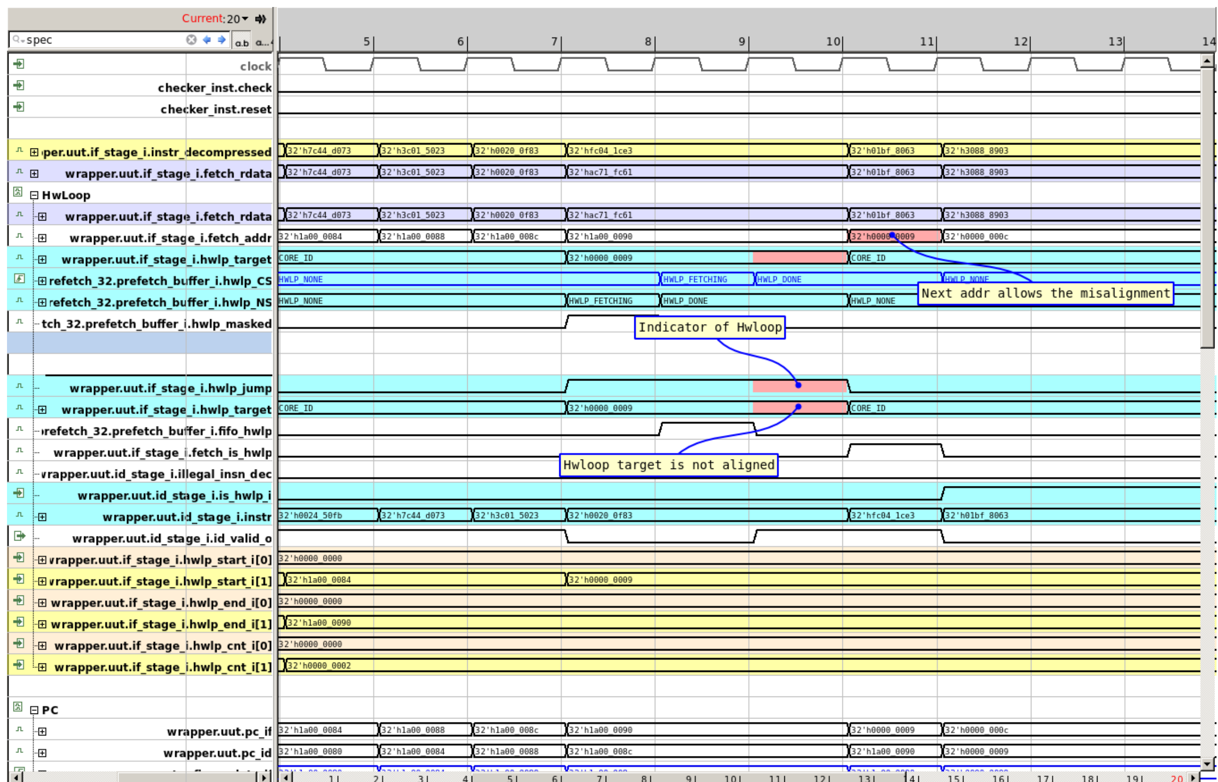
Na documentação, tem o valor do campo `funct3` igual a 010, enquanto o correto é 101. Semelhantemente, a variante `p.clipur` tem 010 na documentação, enquanto o correto é 110.

- Instruções SIMD

  As instruções de SIMD que envolvem o deslocamento de bits (*shift*) não indicam que apenas uma quantidade de bits menos significativos é utilizada. Isso é esclarecido em versões mais novas da documentação.

Quanto aos *bugs* propriamente ditos, é notável que o aspecto mais problemático se deu com os laços de *hardware*. O primeiro ponto é que eles permitem saltos para endereços desalinhados, ou seja, endereços que não são múltiplos de 32 ou 16 bits. Isso aparentemente foi corrigido em versões mais recentes, já que a documentação nova indica que os 2 bits menos significativos dos endereços de início e fim do laço são curto-circuitados para 0. No entanto, isso não acontece na nossa versão do DUT. Verificamos se alguma exceção ou *trap* é gerada nesses casos, porém isso não ocorre. Na figura 3, podemos observar que no ciclo 10 o sinal `hwlp_jump` é nível alto, indicando que está ocorrendo a busca da instrução de início de um laço. O endereço dessa função, indicado em `hwlp_target`, é um número ímpar, e, portanto, desalinhado.

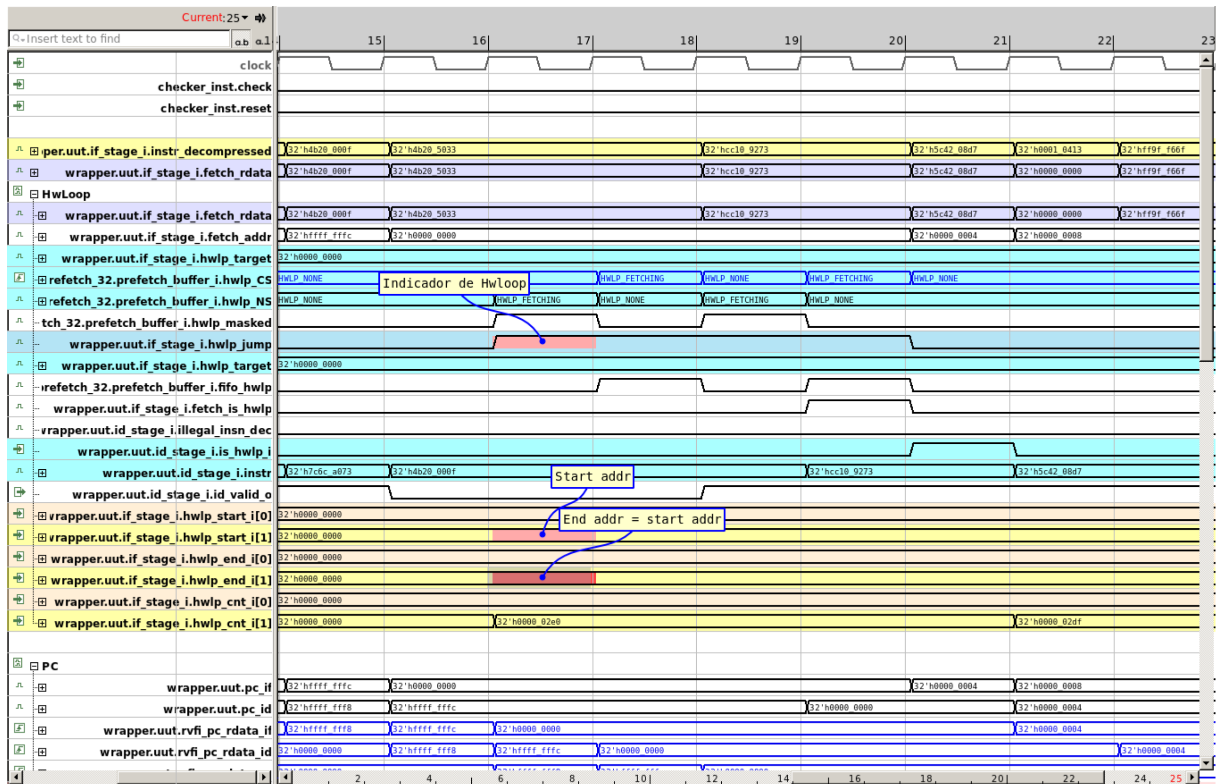
Figura 3 – Um laço de *hardware* saltando para endereço desalinhado.



Fonte: Jasper™.

Segundo a documentação, o corpo de um laço de *hardware* deve ser composto por, no mínimo, 2 instruções. É esperado que isso seja respeitado pelo *assembler* na hora de escrever o *software*, mas não existe mecanismo em *hardware* que impeça que isso aconteça, ou, se existe, é insuficiente. A análise formal, sendo exaustiva, acaba acusando esse tipo de situação inesperada. Na figura 4, podemos observar que no ciclo 17 se inicia um laço de apenas uma instrução, a qual se localiza no endereço 0. Podemos evidenciar que um laço está ocorrendo porque o sinal `hwlp_jump` é nível alto.

Figura 4 – Um laço de *hardware* de apenas 1 instrução.

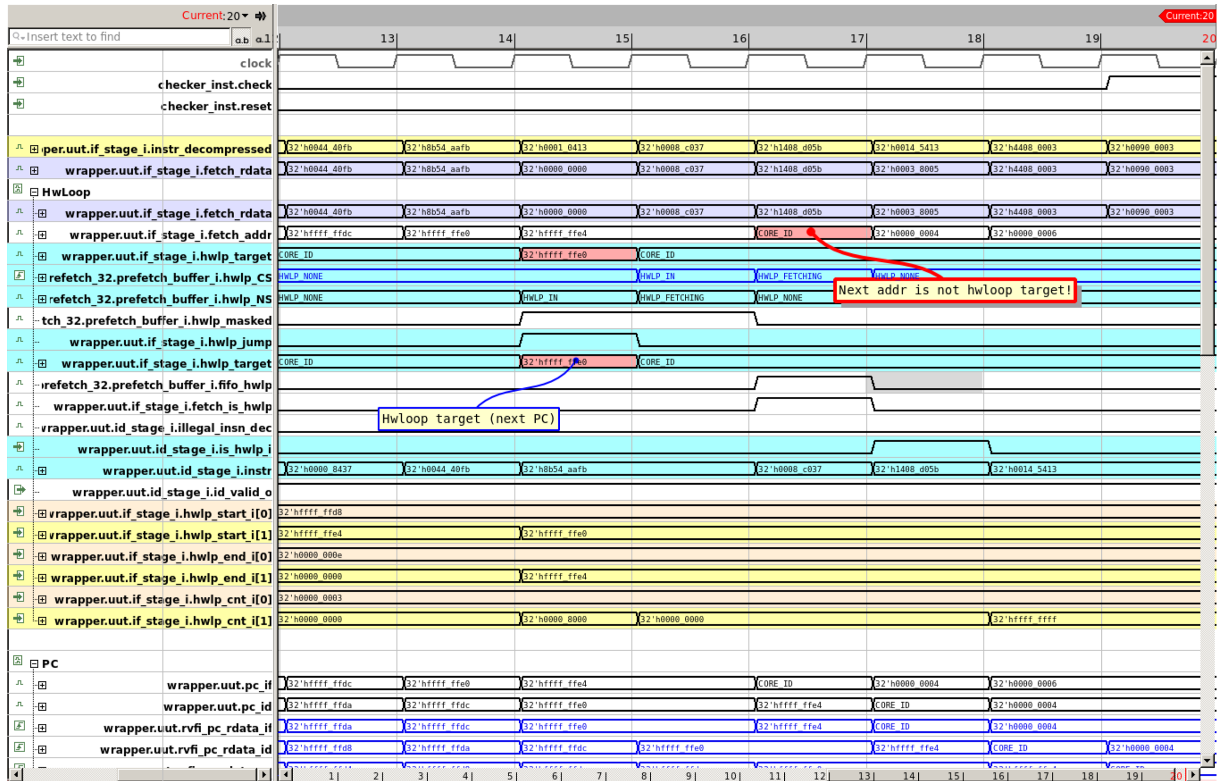


Fonte: Jasper™.

Além disso, pudemos observar casos em que o salto ocorre para um endereço incorreto, ou seja, o contador de programa para o qual é feito a busca da próxima instrução por causa do laço de *hardware* é diferente do endereço guardado no CSR `hwlp_start`. Isso é exemplificado na figura 5. Percebemos que o alvo no ciclo 16 deveria ser o endereço 0xFFFF_FFE0, mas o próximo endereço buscado é diferente. Isso ocorre porque o sinal `hwlp_masked` está nível alto nos ciclos 15 e 16, causando o atraso do salto por 1 ciclo, mas o sinal `hwlp_target` não é mantido estável no ciclo 16 para acompanhar o atraso, fazendo com que seu valor errado seja lido.

Outro *bug* que ocorre é que o decodificador não confere todos os campos da instrução para definir se ela é legal ou ilegal. Por exemplo, duas variantes do *store* com pós-incremento são definidos com o campo `funct7` = 0. Isso não é conferido, tanto que no teste que criamos para essa instrução nós também não incluímos essa checagem. Se-

Figura 5 – Erro no alvo do laço de *hardware*.



Fonte: Jasper™.

ria esperado que as assertivas falhassem já que não estamos checando todos os campos necessários da instrução, mas nenhuma acaba falhando, comprovando que o processador executa as instruções codificadas erroneamente. A figura 6 apresenta uma sequência de instruções obtida em um teste da instrução `p.sw`, no entanto, a última instrução da sequência, a qual deveria ser uma `p.sw`, não é reconhecido pelo *disassembler*, logo é uma instrução ilegal. No entanto, o *core* a decodifica e a executa como uma `p.sw` e todas as assertivas passam. Nesse caso, podemos dizer que se trata de um erro "inofensivo", já que não existem outras instruções com o mesmo `opcode` e `funct3`, então isso não causa conflito com a codificação de outras instruções.

Figura 6 – Instrução ilegal sendo decodificada como `p.sw`.

```
Disassembly of section .text:

00000000 <.text>:
   0:   9c02                    c.jalr  x24
   2:   98005057                pv.dotsp.sc.b   x0,x0,x0
   6:   1000602b                0x1000602b
```

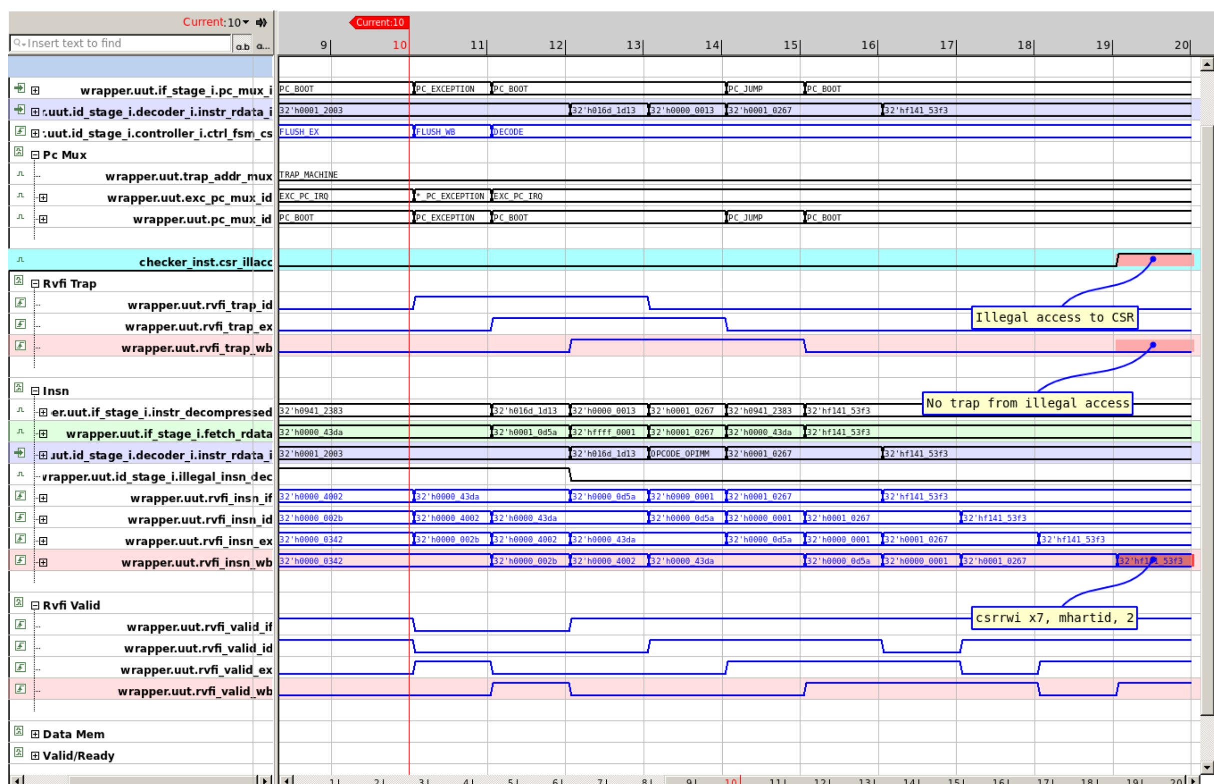
Fonte: autoria própria.

No entanto, descobrimos alguns casos em que essa decodificação "otimista" gera problemas. A instrução `p.abs`, a qual calcula o valor absoluto de um número inteiro, é codificada com o campo `rs2` igual a 0, mas isso não é conferido pelo decodificador. A

princípio, pensamos que isso não deveria ser problema, já que a instrução calcula o módulo de rs1 e não rs2. Mas, após uma análise, percebemos que a ULA utiliza o valor de rs2 para realizar alguma operação, assumindo que está lendo o registrador 0 dali. Como o *core* não garante que o rs2 lê o registrador 0, o resultado pode ficar errado. Assim como no caso dos laços de *hardware*, é esperado que a codificação correta seja respeitada pelo *assembler* na hora de escrever o *software*, mas não existe mecanismo em *hardware* que impeça que isso aconteça, o que possibilita a execução errônea de instruções ilegais sem o levantamento de qualquer exceção. O mesmo erro em que não é checado se o campo rs2 é zero acontece nas instruções `pv.abs.h` e `pv.abs.b`.

Por fim, temos um erro que ocorre no acesso do CSR mhartid, o qual tem permissão de apenas leitura. O erro é ilustrado na figura 7. O *core* não permite a escrita nesse registrador, como é de se esperar. No entanto, ele executa a instrução e realiza a leitura do CSR. A especificação do RISC-V nos indica que qualquer tentativa de escrever em um CSR de apenas leitura deve levantar uma exceção de instrução ilegal, o que não acontece em nosso *core*.

Figura 7 – Acesso ilegal a CSR de apenas leitura que não gera uma *trap*.



Fonte: Jasper™.

9 CONSIDERAÇÕES FINAIS

Nesse trabalho, foi implementado um *framework* de verificação formal no processador CV32E40P, utilizado no PULPissimo, o qual contém diversas extensões customizadas. Para testar apropriadamente essas extensões, expandimos a ferramenta, criando diversos novos testes. Diante do que foi apresentado, podemos considerar que o objetivo do trabalho foi alcançado, já pudemos encontrar diversos erros na implementação do CV32E40P a partir da aplicação do RFVF.

No entanto, a nossa implementação da interface do RISC-V Formal não está perfeita em nosso RTL. É possível identificar que instruções são perdidas no *pipeline* da interface quando, por exemplo, há um *stall* na unidade de acesso à memória, fazendo com que elas não cheguem ao *testbench*. Isso significa que nossos testes não checam todos os casos possíveis das instruções, então não podemos dizer que estão completos. Isso também causa falhas no teste de consistência de registradores, já que, como a instrução não chega ao *checker*, ele não sabe que foi escrito um novo valor no registrador. A correção desse erro seria facilitada se o *core* propagasse a instrução por todos os seus estágios de *pipeline* (essa informação só chega ao estágio ID), permitindo identificar exatamente onde está cada instrução. O maior desafio de nosso trabalho foi tentar contornar isso alterando apenas a implementação da interface, sem modificar o design do processador.

Podemos mencionar também os testes de instrução de multiplicação e divisão. Como mencionamos, operações matemáticas, e operações que modificam os dados em um *datapath* de forma geral, acrescentam muita complexidade às provas. O RFVF reconhece isso, e permite a implementação de operações alternativas para substituir a multiplicação e divisão (RISC-V. . . , 2024a). Nesse caso, a operação realizada na ULA sofre um *bypass*, permitindo que possamos checar todas os dados da instrução, exceto o resultado da operação, só que sem a complexidade adicionada pela lógica combinacional. Como nós evitamos fazer qualquer edição no design do processador, isso não foi implementado, e pode ser uma possibilidade para um trabalho futuro.

Como relatado, criamos dezenas de novos testes para as instruções não-oficiais da extensão xpulp. Algumas não foram feitas por envolverem multiplicação, como é o caso das instruções de multiplicar e acumular. Pelo motivo ilustrado no parágrafo anterior, operações como essa não são ideais de se verificar formalmente, mas podemos aproveitar das operações alternativas do RFVF para testar alguma coisa. O restante não pode ser elaborado por falta de tempo hábil. Desse modo, outra possibilidade para trabalhos futuros seria finalizar esse conjunto de instruções.

REFERÊNCIAS

ALMEIDA, M. A. de. Verificação formal para hardware. In: UFCG - CEEI - DEE - PROJETOS DE ENGENHARIA ELÉTRICA. [S.l.], 2018. Citado nas páginas 12 e 15.

HARRIS, S.; HARRIS, D. *Digital Design and Computer Architecture: RISC-V Edition*. 1. ed. [S.l.]: Morgan Kaufmann, 2021. Citado na página 9.

MELHAM, T. F. *Formalizing abstraction mechanisms for hardware verification in higher order logic*. [S.l.], 1990. Citado na página 14.

OBI. 2022. <https://github.com/openhwgroup/obi>. Accessed: 4 de maio de 2024. Citado na página 25.

OPENHW Group. 2024. <https://www.openhwgroup.org/>. Accessed: 4 de maio de 2024. Citado na página 21.

OPENHW Group CORE-V CV32E40P RISC-V IP. 2024. <https://github.com/openhwgroup/cv32e40p>. Accessed: 4 de maio de 2024. Citado na página 21.

PULPISSIMO. 2022. <https://github.com/pulp-platform/pulpissimo>. Accessed: 4 de maio de 2024. Citado na página 22.

RISC-V Formal Verification Framework. 2024. <https://github.com/YosysHQ/riscv-formal>. Accessed: 4 de maio de 2024. Citado nas páginas 9 e 46.

RISC-V Formal Verification Framework - Pulpissimo Edition. 2024. <https://github.com/ArthurMdrs/riscv-formal>. Accessed: 4 de maio de 2024. Sem citações.

RISC-V Specifications. 2019. <https://riscv.org/technical/specifications/>. Accessed: 4 de maio de 2024. Citado nas páginas 10 e 21.

SANTOS, M. da S. Verificação formal de registradores de controle e status do bloco keccak message authentication code e secure hashing algorithm 3 do projeto opentitan. In: UFCG - CEEI - DEE - PROJETOS DE ENGENHARIA ELÉTRICA. [S.l.], 2022. Citado nas páginas 9 e 10.