Implementação de Testes em Integração Continua na Verificação de Processadores RISC-V

Palavras-Chave: Integração Contínua, RISC-V, FPGA

Autores/as:

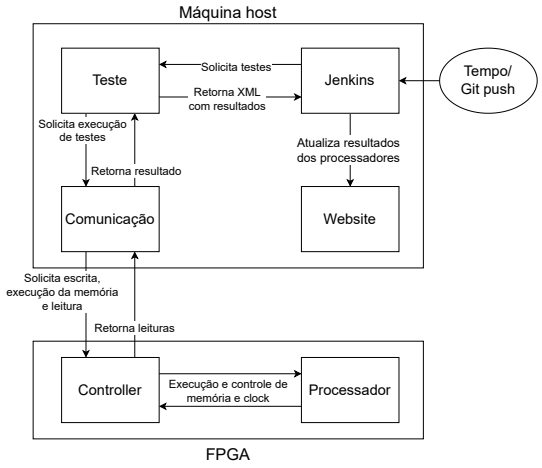
Victor Prudente Lago – LSC, IC

Prof(ª). Dr(ª). Rodolfo Azevedo (orientador(a)) – LSC, IC

INTRODUÇÃO:

Atualmente, há um número significativo de processadores presentes em nosso cotidiano, desempenhando funções que vão desde tarefas muito simples até operações complexas em diversos sistemas embarcados. Para garantir seu funcionamento correto, é essencial que esses processadores passem por um processo rigoroso de verificação, uma vez que, após a fabricação do silício, não é mais possível realizar correções físicas na placa.

O mercado, no entanto, é dominado por arquiteturas proprietárias, como x86, ARM e Power, controladas por grandes fabricantes e associadas a altos custos de licenciamento. Nesse contexto, o RISC-V surgiu como uma alternativa aberta e livre de *royalties* a essas arquiteturas, promovendo maior liberdade de desenvolvimento. Contudo, o rápido crescimento de seu uso também traz o desafio de lidar com múltiplas implementações que, embora compatíveis em teoria, podem não ter sido adequadamente testadas.



Durante este projeto, foi desenvolvida uma infraestrutura de verificação capaz de testar diferentes implementações de processadores RISC-V. Os principais objetivos foram oferecer um ambiente de integração contínua, garantir simplicidade de integração, proporcionar escalabilidade e permitir a comparação entre distintas implementações. Minhas contribuições incluíram a configuração do ambiente de testes e integração contínua por meio da ferramenta Jenkins, o desenvolvimento do módulo de comunicação entre *hardware* e *software* e a criação dos *scripts* para execução automatizada de testes na plataforma Jenkins.

METODOLOGIA:

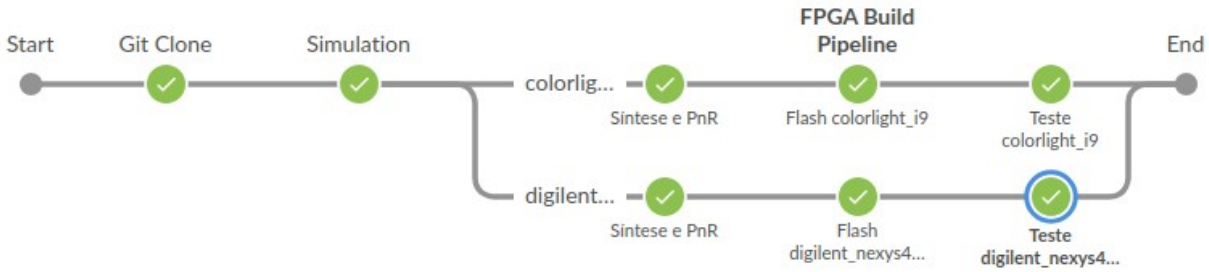
1 Integração Continua(IC)

O Jenkins foi escolhido como plataforma de integração contínua devido à ampla quantidade de suporte, ferramentas disponíveis e por ser *open-source*.

A instalação e as configurações iniciais foram realizadas na máquina *host*. Usuários foram criados para os membros da equipe, *plugins* foram instalados e, por fim, o primeiro *job* foi criado. As primeiras semanas foram destinadas à familiarização com a plataforma Jenkins.

Ao criar o primeiro *job*, utilizamos as configurações oferecidas pelo próprio Jenkins para apagar *builds* anteriores ao criar uma e definimos *triggers* para que a *build* fosse executada em caso de *push* no repositório GitHub do processador. Também foi criado um *script* do Jenkins executado sempre que um *trigger* de tempo ou *push* é ativado.

O pipeline básico para testar um processador está mostrado na figura abaixo:



2 Comunicação

Além disso foi desenvolvido um *software* de comunicação para enviar instruções ao controlador.

A implementação inicial consistiu em um *script* em Python chamado **ExtController**, que utilizava a biblioteca serial e possuía os seguintes métodos:

- **__init__()**: Recebia como parâmetros a porta (*port*), a taxa de transmissão (*baudrate*) e o tempo limite (*timeout*) para estabelecer a comunicação serial, abrindo a conexão com o dispositivo.
- **send_data()**: Escrevia na interface serial, enviando os dados para o controlador.
- **read_data()**: Lia 4 bytes da interface serial e retornava a resposta do controlador.
- **run()**: Iniciava um laço de execução contínua, recebendo comandos diretamente da linha de comando e enviando as instruções apropriadas para o controlador, de acordo com uma tabela de operações (não especificada no texto).

Dentro do método **run()**, havia um *loop* que testava diversas condições para determinar qual ação deveria ser realizada, um exemplo do como isso estava sendo realizado:

```
elif option == 'L':  # Ler a posição N de memória
    opcode = 0b01001100
    N = int(input('Digite o endereço de memória: '))
    N = N << 8
    mensagem = opcode | N
    mensagem_bytes = mensagem.to_bytes(4, 'big')
    self.send_data(mensagem_bytes)
    self.read_data()  # Lê o valor da posição de memória
```

Embora essa implementação fosse funcional, diversos problemas estavam presentes: ausência de verificação dos dados de entrada fornecidos pelo usuário, redundância no uso de variáveis temporárias, baixa modularidade, grande uso de *strings mágicas* e literais, além de uma integração difícil com outros módulos.

Posteriormente, o módulo foi reestruturado algumas vezes para resolver esses problemas e permitir o uso da solução como uma API. Também foi implementado um recurso para carregar um programa a partir de um arquivo especificado diretamente na placa.

Mesmo com as melhorias, ainda havia espaço para otimizações adicionais. Essas melhorias, entretanto, foram realizadas por outros membros da equipe, enquanto eu passei a trabalhar no sistema de testes.

2 Testes

No ProcessorCI(nome do projeto), os testes são organizados em diretórios contendo três categorias: básicos, avançados e inválidos (estes últimos devem falhar para serem considerados corretos). Cada categoria possui subdiretórios para arquivos de memória (programas em hexadecimal), referência (resultados esperados) e código-fonte em RISC-V.

Após o desenvolvimento do módulo de comunicação, foram criados *scripts* para execução automatizada dos testes, divididos em duas classes em formato de API e um *script* simples usado pelo Jenkins.

- A classe **Test** executa um teste individual, carregando o programa no processador, executando-o e comparando o resultado obtido com o esperado, registrando o tempo e o sucesso da execução.
- A classe **Directory** gerencia a execução de todos os testes de um diretório, organizando-os por categoria e gerando relatórios XML com os resultados.

Por fim, o *script* **run.py** recebe, via linha de comando, o caminho de um arquivo JSON com a lista de diretórios de teste e a porta serial utilizada. Ele cria objetos da classe **Directory** para cada diretório, executa os testes e registra automaticamente os resultados, permitindo integração direta com sistemas de CI/CD, como o Jenkins.

RESULTADOS E DISCUSSÃO:

Nos primeiros meses, foram realizadas a revisão bibliográfica, a configuração inicial do ambiente de integração contínua com Jenkins e a implementação dos primeiros *jobs*. Nesse ponto da pesquisa, já haviam sido integrados 78 *jobs* para diferentes processadores, dos quais alguns executaram testes com sucesso, embora nem todos tenham conseguido realizar a síntese, como esperado.

A etapa mais crítica foi a criação da infraestrutura para execução de testes em processadores RISC-V, concluída com o desenvolvimento de módulos não intrusivos que permitem executar instruções diretamente a partir da máquina *host*. Nesse ponto, embora a interface estivesse funcionando, ainda não havia implementação da coexecução de testes em simulação e FPGA, nem integração de todas as FPGAs disponíveis. Também foi identificado um gargalo devido ao aumento no número de processadores suportados, que inviabilizou a execução diária de todos os *jobs*.

Quanto ao desenvolvimento de testes, inicialmente utilizou-se um sistema simplificado para validar a infraestrutura, que depois foi aprimorado. Nesse momento, havia 42 testes básicos para o grupo RV32I, sem testes avançados ou inválidos implementados, mas com a plataforma já pronta para recebê-los futuramente.

CONCLUSÕES:

O projeto ProcessorCI representou um avanço significativo na verificação de processadores RISC-V, oferecendo uma plataforma flexível e robusta para a execução de testes automatizados em diferentes implementações de hardware. A criação de um ambiente de integração contínua utilizando o Jenkins permitiu a execução sistemática e periódica de testes, garantindo a confiabilidade e a eficiência do processo de verificação. Nesse ponto da pesquisa, a infraestrutura desenvolvida já era capaz de suportar a integração de múltiplos processadores, embora alguns desafios, como a coexecução de testes em simulação e FPGA, ainda necessitem ser implementados no futuro.

A metodologia de testes implementada, embora inicialmente focada em testes básicos, estabeleceu uma base sólida para a expansão do conjunto de testes, incluindo categorias avançadas e inválidas. Nesse ponto da pesquisa, a interface de comunicação desenvolvida já facilitava a interação

entre *software* e *hardware*, permitindo a execução de comandos e a análise de resultados de forma eficiente. Além disso, a estrutura modular do projeto possibilitou a fácil integração de novos processadores e testes, garantindo escalabilidade e adaptabilidade.

Os resultados obtidos demonstram que, nesse ponto da pesquisa, o ProcessorCI havia atingido seus principais objetivos iniciais, fornecendo uma ferramenta com potencial para a comunidade acadêmica e industrial. No entanto, permanecem oportunidades para melhorias, como a integração de mais FPGAs, o desenvolvimento de uma metodologia sistemática para criação de testes e a exploração de técnicas avançadas de verificação, como o paralelismo em nível de instrução.

Em síntese, o ProcessorCI contribui para a verificação de processadores RISC-V e se estabelece como uma ferramenta para a continuidade de pesquisas nessa área, utilizando-se de boas práticas e padronização de métodos de verificação. O projeto abre caminho para novas investigações e aprimoramentos, consolidando-se no campo da verificação de hardware.

BIBLIOGRAFIA

N. Bruns, V. Herdt, and R. Drechsler, "**Processor verification using symbolic execution: A RISC-V case-study**," in *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Antwerp, Belgium, 2023, pp. 1–6.

M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, and A. Tatarnikov, "**Test program generator MicroTESK for RISC-V,**" in *Proc. 19th Int. Workshop on Microprocessor and SoC Test and Verification (MTV)*, Austin, TX, USA, 2018, pp. 6–11.

E. Cui, T. Li, and Q. Wei, "**RISC-V instruction set architecture extensions: A survey,**" *IEEE Access*, vol. 11, pp. 24696–24711, 2023.

V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "**Efficient cross-level testing for processor verification: A RISC-V case-study,**" in *Proc. Forum for Specification and Design Languages (FDL)*, Kiel, Germany, 2020, pp. 1–7.

A. Joannou, P. Rugg, J. Woodruff, F. A. Fuchs, M. Van der Maas, M. Naylor, M. Roe, R. N. M. Watson, P. G. Neumann, and S. W. Moore, "**Randomized testing of RISC-V CPUs using direct instruction injection,**" unpublished manuscript, 2024.

M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "**From RTL to SVA: LLM-assisted generation of formal verification testbenches,**" unpublished manuscript, 2023.

T. G. Truong, T. G. Do, T. D. Do, *et al.*, "**RISC-V random test generator,**" in *Proc. 15th Int. Conf. on Advanced Computing and Applications (ACOMP)*, Ho Chi Minh City, Vietnam, 2021, pp. 150–155.