International Journal of Computer Architecture Education, 2025, 14:1
ISSN: 2316-9915 • doi: 10.5753/ijcae.2025.6319

ARTIGO DE PESQUISA

Desenvolvendo simuladores para arquitetura de computadores com auxílio de modelos generativos de linguagens

Ricardo Ferreira [Universidade Federal Viçosa | *ricardo@ufv.br*]
Racyus Delano Garcia Pacífico [Universidade Federal Ouro Preto | *racyus.pacifico@ufop.edu.br*]

Departamento de Informática, Universidade Federal de Viçosa, Viçosa, cep 36570-900, Viçosa, Brazil.

Resumo. Este trabalho apresenta o uso de modelos generativos para o desenvolvimento de simuladores interativos voltados ao ensino de arquitetura de computadores em diferentes níveis de abstração. Os simuladores contam com interfaces que incluem janelas de edição de código, visualização gráfica da execução e integração com o Google Colab, promovendo a experimentação prática e acessível. O desenvolvimento foi realizado no contexto das disciplinas de arquitetura de computadores da Universidade Federal de Viçosa, com participação ativa dos alunos. Como parte das atividades, os estudantes receberam a demanda dos requisistos desejados para cada simulador, acompanhada de um código inicial com número reduzido de funcionalidades, que deveria ser ampliado e aprimorado. Foram desenvolvidos simuladores para diversas arquiteturas, incluindo: processadores RISC-V com e sem pipeline, arquiteturas com múltiplas unidades funcionais, simuladores baseados no algoritmo de Tomasulo, interpretadores para *assembly* vetorial, arquiteturas com *array processors* SIMD, além de modelos de multiprocessadores com memória compartilhada e com troca de mensagens. Além dos processadores, um projeto de uma cache 4-*way* também foi implementado. A cache e os processadores RISC-V têm implementação em Verilog e Python. As demais arquiteturas usaram simuladores em Python. A abordagem visa tornar o aprendizado mais visual, prático e alinhado com os fundamentos teóricos da área.

Palavras-chave: Simuladores, modelos generativos de linguagem, arquitetura de computadores, computação paralela.

Recebido: 04 Agosto 2025 • **Aceito:** 28 Setembro 2025 • **Publicado:** 21 Janeiro 2026

1 Introdução

Este artigo aborda o processo de desenvolvimento e ferramentas adotadas na criação de simuladores interativos para o ensino de arquitetura de computadores, com participação dos alunos. Nosso objetivo é duplo: primeiro, fomentar a troca de experiências entre educadores da área de engenharia e computação, promovendo a criação e o compartilhamento de ferramentas de código aberto com acesso remoto; segundo, refletir sobre as inovações pedagógicas impulsionadas pela disponibilidade de novas ferramentas como modelos generativos de linguagem (LLMs) e suporte de ambientes virtuais com alto desempenho e acesso facilitado via navegadores [Canesche *et al*., 2021; Jamieson *et al*., 2025], que podem levar a práticas mais eficazes.

O uso de simulação no ensino tem se mostrado uma estratégia interessante [Hwang *et al*., 2025] para promover o aprendizado, apoiar tomada de decisões e facilitar a compreensão de sistemas complexos. No entanto, este trabalho vai além da simples utilização de simuladores: propomos um modelo no qual os estudantes são envolvidos como colaboradores no desenvolvimento das ferramentas, atuando como ”desenvolvedores dos simuladores” [Ferreira *et al*., 2015]. Esta abordagem estimula não apenas a compreensão técnica dos conteúdos, mas também habilidades práticas de programação, uso de novas ferramentas com LLMs e colaboração.

A proposta tem caráter incremental. Os alunos recebem, como ponto de partida, um código parcial funcional com funcionalidades limitadas, além de uma especificação clara da funcionalidade desejada. A partir disso, são desafiados a expandir e aprimorar os simuladores, compreendendo tanto a arquitetura do sistema quanto os conceitos computacionais envolvidos. A metodologia de desenvolvimento incluem o uso de técnicas convencionais de programação, uso de LLM e uso de programação baseada em exemplos, onde protótipos iniciais dos simuladores são fornecidos. A estrutura modular e acessível dos simuladores permite que os estudantes rapidamente comecem a gerar códigos funcionais para validação além da compreensão de diversas arquiteturas computacionais e de como elas podem ser desenvolvidas e validadas.

Além de sua aplicação nas disciplinas de arquitetura de computadores da Universidade Federal de Viçosa, acreditamos que essa abordagem pode ser estendida para diversas outras áreas da engenharia e da computação. As ferramentas são de código livre e documentadas para a criação de um espaço colaborativo voltado ao suporte técnico e pedagógico para educadores interessados no desenvolvimento e uso de ferramentas educacionais interativas.

Experiências práticas são parte essencial da formação em engenharia, e o desenvolvimento de simuladores interativos pode ser interpretado como diretrizes de três categorias clássicas de laboratórios educacionais [Jamieson *et al*., 2021]:

1. **Laboratório de desenvolvimento:** Está alinhado ao aprendizado baseado em projetos, os alunos são desafiados a projetar, expandir e avaliar simuladores computacionais, partindo de demandas funcionais concretas e códigos-base com funcionalidades limitadas.

2. **Laboratório de pesquisa:** O ambiente de simulação também funciona como espaço para exploração aberta, onde os estudantes podem investigar o comportamento de arquiteturas alternativas, propor extensões e compreender sistemas complexos por meio da experimentação.
3. **Laboratório educacional:** Os simuladores permitem que os estudantes validem na prática os conceitos teóricos discutidos em aula, como pipeline, paralelismo, acesso à memória e comunicação entre processadores, promovendo uma ponte direta entre teoria e aplicação.

Ao integrar essas três dimensões, o processo de desenvolvimento e uso dos simuladores não apenas reforça o conteúdo curricular, mas também estimula habilidades de resolução de problemas, colaboração e pensamento computacional de forma contextualizada e progressiva.

Cursos de arquitetura e organização de computadores tradicionalmente envolvem atividades práticas como tarefas e avaliações, onde os alunos desenvolvem programas em linguagem de montagem, constroem arquiteturas simples e complexas, implementando otimizações para compreender o projeto de baixo nível e a operação de processadores e das arquiteturas. Nesse contexto, os simuladores desenvolvidos permitem não apenas a observação desses processos, mas também sua experimentação ativa em um ambiente intuitivo e acessível.

2 Metodologia

A proposta deste trabalho baseia-se na integração de modelos de linguagem de grande escala (LLMs), Python/JavaScript e Google Colab para o desenvolvimento de simuladores voltados ao ensino de arquitetura de computadores. A metodologia explora as vantagens dessas tecnologias de forma complementar para facilitar tanto a criação quanto o uso pedagógico de simuladores interativos.

O uso de LLMs permite gerar rapidamente interpretadores, estruturas de dados e interfaces gráficas para diferentes arquiteturas computacionais, como processadores vetoriais, processadores em array, e conjuntos de instruções reduzidas (como subconjuntos de RISC-V ou montadores didáticos) [Jamieson *et al*., 2025; de Figueiredo *et al*., 2024]. As LLMs também auxiliam na explicação do código gerado e na adaptação de funcionalidades com base nas necessidades didáticas de cada experimento, promovendo uma abordagem iterativa e incremental no desenvolvimento dos simuladores.

A base da infraestrutura adotada é o Google Colab [Canesche *et al*., 2021; Ferreira *et al*., 2024a,b], que oferece várias vantagens para o contexto educacional: acesso gratuito e multiplataforma, execução em nuvem sem necessidade de instalação local, facilidade para compartilhar *notebooks* com exemplos e exercícios, e suporte a recursos multimídia para explicações interativas (como gráficos, *widgets* e animações).

Cada estudante recebe como ponto de partida um *notebook* contendo um simulador funcional baseado em um domínio específico, por exemplo, um subconjunto de instruções de um processador SIMD ou um emulador de cache com visualização. Esses simuladores já vêm acompanhados de descrições e exemplos gerados com o apoio de LLMs. O exercício proposto aos alunos consiste em analisar, modificar e expandir essas implementações.

3 Simuladores de processadores

Há uma grande variedade de simuladores educacionais voltados ao estudo de processadores, especialmente para as arquiteturas MIPS e, mais recentemente, RISC-V [Giorgi and Mariotti, 2019; Savaton, 2021; Mezger *et al*., 2022; Böseler *et al*., 2022; Esmeraldo *et al*., 2023]. Entre os simuladores disponíveis, destacam-se aqueles que implementam processadores de um ciclo (monociclo) e com pipeline, permitindo a visualização do fluxo de instruções ao longo das etapas do ciclo de execução.

Entretanto, mesmo entre os simuladores com interface gráfica [Giorgi and Mariotti, 2019; Savaton, 2021], incluindo muitos de código aberto, é comum que o funcionamento interno seja fixo, com opções limitadas de personalização. Para que educadores e alunos possam modificar ou estender tais ferramentas, é necessário um esforço considerável para compreender tanto a lógica de simulação quanto a estrutura da interface gráfica [Garcia *et al*., 2024]. Outro ponto de limitação frequente é que muitos desses simuladores não produzem um código intermediário ou de saída que possa ser mapeado diretamente para implementação física, como em FPGA ou ASIC. Outros simuladores produzem códigos em linguagem de hardware (Verilog ou VHDL), mas não oferecem interfaces didáticas [Schoeberl, 2025] ou envolvem projetos de implementação dos processadores [Zekany *et al*., 2021]. Nossa abordagem é ampla, pois inclui montador, simulador de alto nível, implementação em Verilog e visualização gráfica da implementação.

Nos últimos anos, surgiram alternativas mais flexíveis, como simuladores baseados no *DigitalJS* [Passe *et al*., 2020] ou ambientes interativos como o Google Colab integrando código em Verilog, que permitem simulação funcional e podem gerar artefatos utilizáveis em síntese digital [Jamieson *et al*., 2025; de Figueiredo *et al*., 2024]. Os simuladores apresentados em [de Figueiredo *et al*., 2024] vão além ao permitir a geração de saídas gráficas no formato SVG, que são editáveis e possibilitam a inclusão de extensões personalizadas.

Neste contexto, desenvolvemos extensões sobre os simuladores descritos em [de Figueiredo *et al*., 2024], incorporando um simulador de alto nível em Python, capaz de gerar automaticamente o código *assembly* (montador) correspondente para o simulador em Verilog. Além disso, implementamos uma visualização interativa do caminho de dados, descrita na Seção 3.1. O estudante pode facilmente adicionar novas instruções, incluindo a decodificação, o montador, a simulação e a implementação, interagindo com o código Python, o código Verilog e o desenho estrutural do projeto.

Em um segundo momento, abordamos arquiteturas mais avançadas, como processadores com múltiplas unidades funcionais em pipeline e com escalonamento dinâmico de instruções baseado no algoritmo de Tomasulo. Nessa etapa, a simulação foi realizada exclusivamente em Python, com ênfase na lógica da execução fora de ordem, sem geração de código Verilog, conforme discutido na Seção 3.2.

3.1 RISC-V

Esta seção apresenta um simulador interativo para o processador RISC-V monociclo, desenvolvido a partir das exten-

Figura 1. (a) janela de edição do código *assembly*; (b) execução e emulação para visualizar os registradores e memória.

sões propostas em [de Figueiredo *et al.*, 2024]. O simulador possui um editor integrado de alto nível em Python, visualização dos registradores e da memória (ilustrado na Figura 1), geração automática de código *assembly* e visualização passo a passo do caminho de dados (ilustrado na Figura 2), permitindo ao aluno explorar instrução por instrução em um ambiente gráfico e personalizável.

```
elif opcode in alu_operations:
    rd = self.parse_register(parts[1])
    rs1 = self.parse_register(parts[2])
    rs2 = self.parse_register(parts[3])
    f7, f3 = dict_rtype[opcode] # montador
    inst = f7 + format(rs2, '05b') + format(rs1,
        '05b') + f3 + format(rd, '05b') + opcodeR
    hex_inst = bin2hex(inst)
    self.code.write(hex_inst+"\n") # verilog
    inst = "32'b" + f7 + "_" + format(rs2, '05b')
        + "_" + format(rs1, '05b') + "_" + f3 +
        "_" + format(rd, '05b') + "_" + opcodeR
    self.instruction_memory.append(inst)
    return Instruction(opcode, rd, rs1, rs2)
# ....
# Trecho do emulador
    def execute_instruction(self, instruction):
        if instruction.opcode == 'add':
            self.registers[instruction.rd] = self
                .registers[instruction.rs1] +
                self.registers[instruction.rs2]
# Trecho do Editor/Exemplo
example_code = """#␣Example:
addi␣x5,x0,10
....
sw␣x6,␣8(x0)
"""
simulator_ui = RISCVSimulatorUI()
simulator_ui.display()
```

Listagem 1: Exemplos de trechos com *parser* e montador de instrução do tipo R, emulador `add` e programa exemplo.

O trecho de código 1 ilustra a simplicidade e modularidade do código gerado pela LLM. Para cada tipo de instrução, há o trecho que faz o *parser* e o montador do código binário para o próximo passo de execução no simulador Verilog. O simulador, além de executar o código, faz o papel do montador.

O simulador Python exporta um arquivo de memória com o código binário. A proposta é ter uma dupla validação do código: no nível do simulador Python e no nível de Verilog. A verificação da execução do processador implementado em Verilog oferece uma depuração com a visualização do caminho de dados.

Os estudantes recebem o código básico do simulador com um pequeno conjunto de instruções. A tarefa é completar o simulador/montador com instruções de desvio e memória, acrescentar sua visualização posterior no Verilog, fazer programas de teste e também criar novas instruções resolvendo exercícios do livro texto de RISC-V [Patterson and Hennessy, 2017].

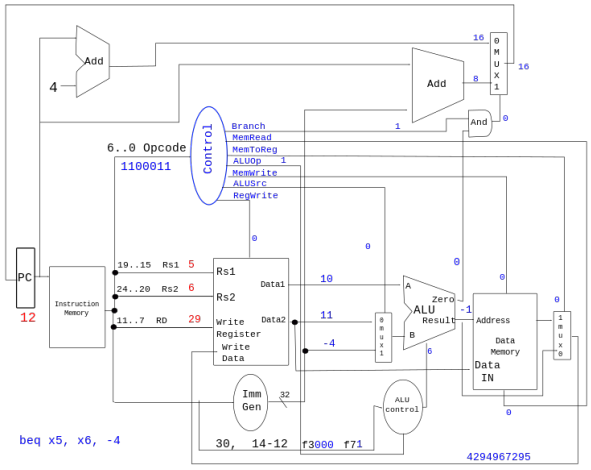

Figura 2. Visualização da execução do código Verilog com interface gráfica editável no formato SVG.

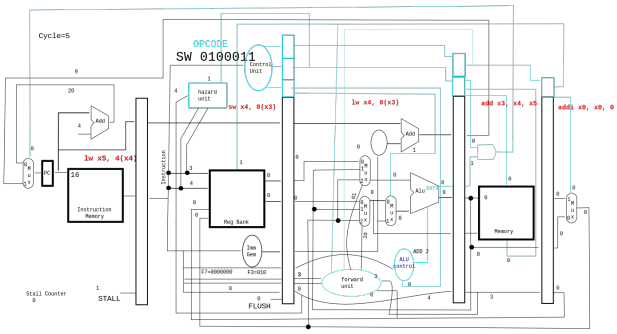

Figura 3. Caminho de dados do pipeline RISC-V com desvio no segundo estágio e encaminhamento com a visualização da execução do código Verilog passo a passo.

Uma versão com pipeline também foi adaptada para inclusão de editor de texto embutido para avaliar diferentes exemplos e a geração automática da depuração em Verilog no caminho de dados. O desenho do caminho de dados derivado da implementação [de Figueiredo *et al.*, 2024] foi modificado, juntamente com o código Verilog. A tarefa dos estudantes foi fazer a extensão do simulador para a inclusão de encaminhamento (*forward*) e do tratamento de desvio no segundo estágio, conforme ilustrado na Figura 3.

Além das novas versões de processadores RISC-V com depuração gráfica, outro conceito que pode ser trabalhado em alto nível é a transferência de informações entre os estágios da pipeline com uma estrutura de dados bem definida, como sugerido em [Railing, 2023]. Com o auxílio das LLMs e das facilidades da linguagem Python, apresentamos uma proposta simples e didática para repassar os valores entre os estágios com dicionários. Cada dicionário tem os sinais (chave) e seus valores.

A Figura 4 ilustra um exemplo de um pequeno trecho no editor do simulador Python com cinco instruções. A parte superior mostra o dicionário e uma ilustração quando a primeira instrução `add x1,x2,x3` está no estágio de decodificação e a instrução `sub x4,x1,x5` está no estágio de busca

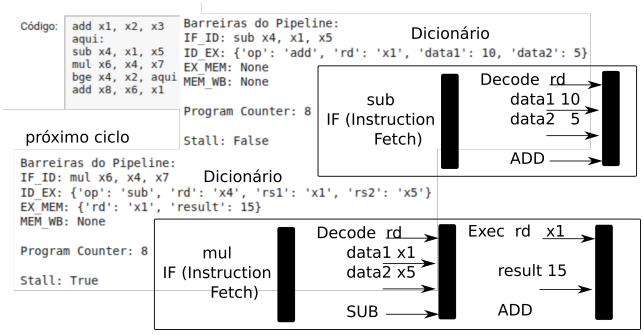

Figura 4. Exemplo de execução com emulação em Python usando dicionários para os sinais entre os estágios do pipeline.

(*fetch*). O destaque está nos sinais que irão ser propagados para o estágio de execução.

Já na parte inferior da Figura 4, ilustramos o próximo ciclo que tem a instrução `mul x6,x4,x7` sendo buscada no estágio de *fetch*, a instrução `sub` já no estágio de decodificação e a instrução `add` sendo executada e propagando seu resultado no estágio de execução.

3.2 Múltiplas unidades e tomasulo

Para explorar conceitos avançados como paralelismo interno ao processador, dependências de dados e execução fora de ordem, desenvolvemos simuladores em Python para arquiteturas com múltiplas unidades funcionais em pipeline com diferentes latências. A primeira tarefa foi implementar um pipeline estático com múltiplas unidades em paralelo. A segunda tarefa foi simular o algoritmo de Tomasulo, com escalonamento dinâmico de instruções, uso de estações de reserva, *buffers* e a técnica de renomeação de registradores, permitindo ao aluno visualizar a resolução dinâmica de dependências e a execução eficiente de instruções fora de ordem.

Como resultado da atividade da disciplina de arquitetura de computadores do primeiro semestre de 2025, foram apresentados seis simuladores funcionais do algoritmo de Tomasulo. A Figura 5 ilustra três exemplos das interfaces dos simuladores que permitem editar o código, visualizar o quadro de execução passo a passo e o estado das estações de reserva. Cada grupo apresentou um simulador funcional com diferentes recursos de visualização com o auxílio das LLMs.

4 Arquiteturas Paralelas

O ensino de arquiteturas paralelas é essencial para compreender o desempenho e a escalabilidade de sistemas modernos. Entretanto, muitas ideias de arquitetura são ensinadas sem exercícios de programação, às vezes apenas com pseudocódigo de forma teórica, sem validação da execução ou simulação. Nesta seção, apresentamos simuladores interativos personalizados para uma determinada arquitetura e uma linguagem de domínio específico com construções para expressar o paralelismo. Esta abordagem é inovadora e permite ao professor e estudante desenvolver e prototipar ideias rapidamente, além de programar vários algoritmos clássicos da literatura com construções paralelas. Os simuladores exploram diferentes formas de paralelismo em nível de dados e tarefas. A proposta inclui arquiteturas vetoriais, *array processors* do tipo SIMD (*Single Instruction, Multiple Data*) e arquiteturas multiprocessadas, oferecendo ao estudante ferramentas para experimentar com diferentes formas de organização paralela e estratégias de comunicação e sincronização.

4.1 Processador Vetorial

Os processadores vetoriais representam uma abordagem clássica de paralelismo em nível de dados, operando sobre grandes conjuntos de elementos simultaneamente. São um marco importante na história da computação paralela, onde foram desenvolvidas várias técnicas com pipeline e escalonamento de instruções. Desde os anos 90, com o Pentium MMX, a Intel introduziu a programação com vetores nos processadores comerciais x86, que evoluíram para o SSE e atualmente o conjunto AVX. Devido à demanda das aplicações científicas e de inteligência artificial, várias extensões do conjunto de instruções dos processadores RISC-V estão focando nas operações vetoriais. Entretanto, não existem simuladores disponíveis para exercitar os conceitos e novos conjuntos de instruções do ponto de vista didático. Por exemplo, na disciplina de arquitetura de computadores foi definido um *assembly* vetorial simples, gerado pela ferramenta ChatGPT, juntamente com um simulador que permite a execução de instruções vetoriais com suporte a carregamento, armazenamento, operações aritméticas e redução, além da visualização da estrutura de registradores vetoriais e do controle de *stride*. A ferramenta visa reforçar conceitos de SIMD e eficiência computacional em operações sobre vetores, conforme ilustrado na Figura 6.

As instruções vetoriais iniciais do *assembly* são:

1. **Registradores vetoriais** `V0` a `V7`: Cada registrador armazena um vetor de oito elementos e possui suporte a acesso com *stride*;
2. **Registradores escalares** `F0` a `F7`: São utilizados para armazenar valores escalares de ponto flutuante;
3. **Instruções suportadas**:
   - `VLOAD Vx, addr, stride`: Carrega oito elementos consecutivos da memória em `V_x`, com espaçamento definido por *stride*;
   - `VSTORE Vx, addr, stride`: Armazena oito elementos de `Vx` na memória;
   - `VADD Vx, Vy, Vz`: Realiza a soma vetorial;
   - `VMUL Vx, Vy, Vz`: Executa a multiplicação vetorial;
   - `VBROADCAST Vx, Fk`: Copia valor contido no registrador `Fk` para todos os elementos do registrador vetorial `Vx`.
   - `VREDUCE_SUM Fk, Vx`: Efetua redução do vetor.

Como tarefas, os estudantes implementaram extensões do conjunto de instruções, por exemplo, instruções para laços de repetição. Para validação, o trabalho inicial foi a execução de uma versão paralela do algoritmo de multiplicação de matrizes e do algoritmo de criptografia TEA [Wheeler and Needham, 1994]. Além disso, foi requisitada a execução passo a passo e a validação da execução com um código equivalente em Python. A Figura 7 ilustra o simulador vetorial com capacidade de executar uma, N ou todas as instruções do código.

A Listagem 2 ilustra um trecho do código do simulador com a instrução de *load* vetorial e a instrução de multiplicação, mostrando o *parser* e o interpretador.

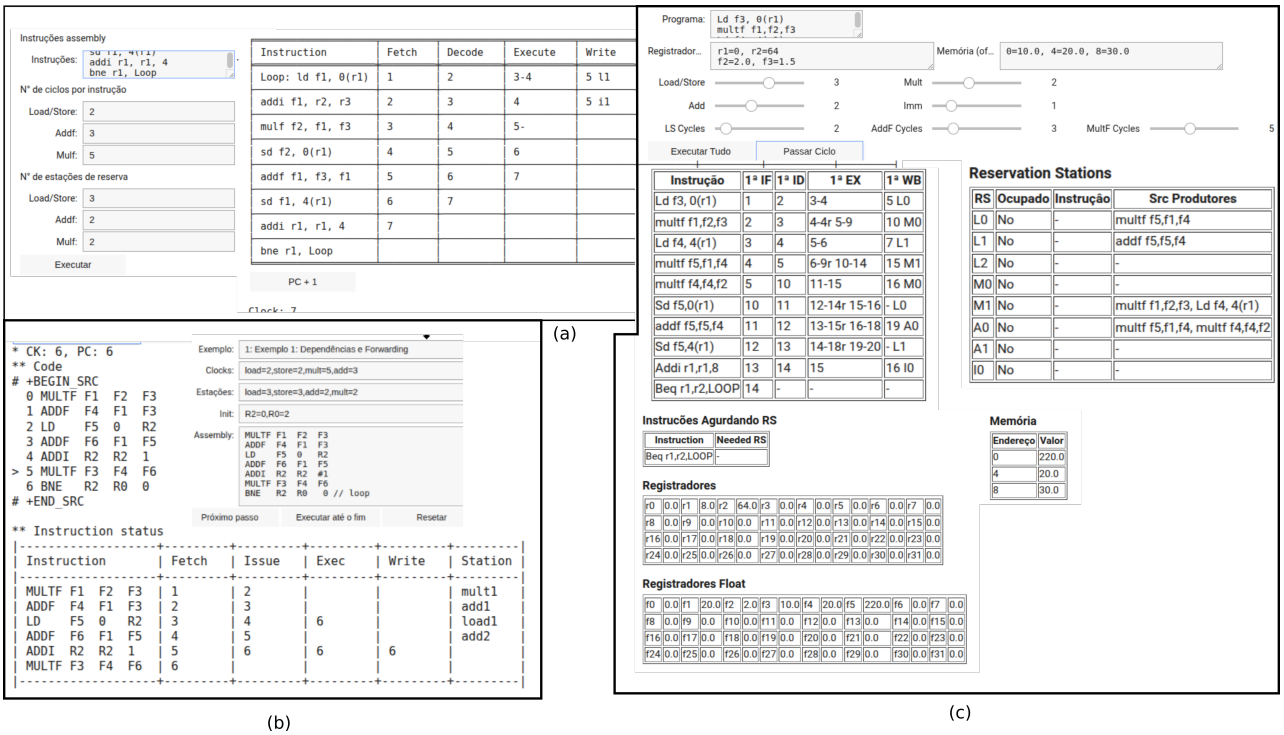

Figura 5. Três implementações de simuladores Tomasulo:(a) simulador com configuração de estações de reserva e quadro de execução passo a passo para *Loop*; (b) código com marcador, quadro de estações e janela de edição; (c) janela edição e configuração, quadro de execução, estações, registradores e memória

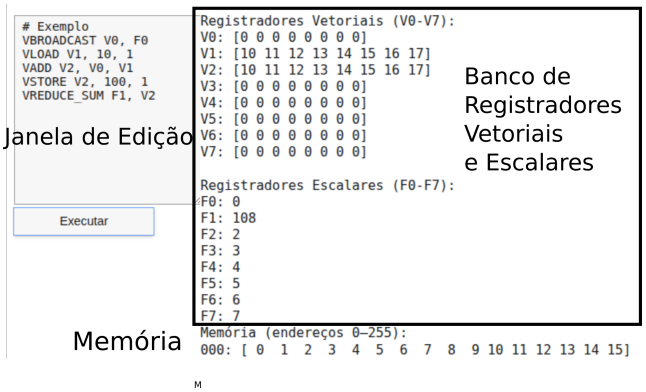

Figura 6. Visualização da execução de um *assembly* vetorial com registradores e memória.

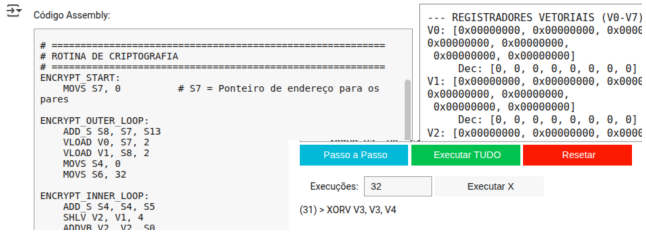

Figura 7. Visualização da execução de um *assembly* vetorial com registradores e memória, execução de N passos para o algoritmo TEA de criptografia.

O exemplo de multiplicação de matrizes é uma referência básica. A execução foi elaborada fixando a linha da matriz e com dois laços aninhados. A Figura 8 mostra o resultado do trabalho da disciplina: um simulador de *assembly* vetorial desenvolvido executando a multiplicação de matrizes.

Figura 8. Visualização da execução de um *assembly* vetorial com registradores e memória para multiplicação de matrizes com laços aninhados em $O(n^2)$.

```
    if instr == 'VLOAD':
        vx, addr, stride = tokens[1].rstrip(','),
              self.parse_value(tokens[2]), self.
            parse_value(tokens[3])
        self.V[vx] = self.memory[addr:addr + 8*
            stride:stride].copy()
    elif instr == 'VMUL':
        vx, vy, vz = tokens[1].rstrip(','),
            tokens[2].rstrip(','), tokens[3]
        self.V[vx] = self.V[vy] * self.V[vz]
```

Listagem 2: Exemplos de trechos com *parser* do *assembly* vetorial.

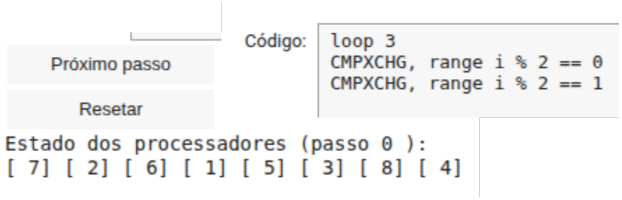

Figura 9. Array Processor Linear Programável com exemplo ordenação Par-Impar com laço e instrução Compara e Troca (CMPXHG) e o identificador i para o número do processador.

É importante ressaltar que as tarefas das atividades práticas da disciplina incluem a extensão do conjunto de instruções vetoriais e a elaboração de vários algoritmos. Como o simulador é interpretado e tem um editor, vários exemplos podem ser validados. Existe a opção do uso de bibliotecas de *parser*, como a *lark* do Python, para gerar um código com uma gramática bem definida e mais genérica para extensão [Coura *et al.*, 2025], ou uso de JavaScript para execução no Google Colab sem a necessidade de conexão.

4.2 *Array processor*

A arquitetura de *array processors* pode ser implementada com diversas topologias: em anel, grade, hipercubo, etc. Os múltiplos processadores executam instruções de forma síncrona ou condicional com máscaras. Com o auxílio das LLMs, é possível gerar rapidamente simuladores para diversas topologias com um conjunto de instruções variado. Como os *array processors* não geraram produtos comerciais, existe uma escassez de exemplos na literatura de linguagem de programação para esta arquitetura. Este recurso foi explorado para gerar tarefas onde os estudantes tinham que criar as linguagens respeitando o paradigma de programação SIMD. A simulação foi feita em Python com suporte à visualização gráfica da comunicação entre processadores.

Diferentemente da programação para GPU, onde o paralelismo está vinculado ao número da *thread*, no *array processor* a programação é indexada com o número do processador. O estudante tem várias tarefas no projeto do simulador, que envolvem a estrutura de memória de cada elemento de processamento, a topologia e as primitivas de comunicação, a linguagem de domínio específico e elaborar exemplos de código e validá-los.

A Figura 9 ilustra um simulador de exemplo para o problema de ordenação em um *array processor* linear de oito elementos de processamento, onde i é o identificador do processador. O simulador tem o código editável da linguagem de domínio específico criada para programá-lo.

4.3 Multiprocessadores

Multiprocessadores são sistemas compostos por múltiplas unidades de processamento independentes, capazes de executar tarefas em paralelo. Nesta subseção, exploramos dois modelos distintos de comunicação entre processadores: com memória compartilhada e com troca de mensagens. Ambos os simuladores permitem ao aluno observar os mecanismos de sincronização, consistência de dados e comunicação entre processos de forma visual e interativa.

4.3.1 Memória Compartilhada

O simulador com memória compartilhada modela um ambiente em que os processadores acessam um espaço comum de

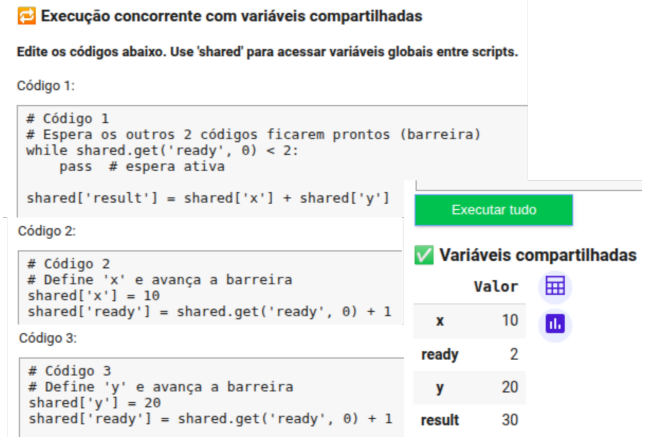

Figura 10. Código de demonstração com memória compartilhada com três processadores em um multiprocessador.

memória, exigindo controle de concorrência para garantir a integridade dos dados. Através do uso de LLM, a tarefa de gerar um simulador inicial com três processadores foi realizada com facilidade. A LLM criou primitivas simples de sincronização e variáveis compartilhadas.

A Figura 10 mostra um exemplo gerado pela ferramenta ChatGPT com três janelas com capacidade de editar o código e executar. O simulador encapsula a comunicação e criou a primitiva `shared` para compartilhar uma variável e o método `shared.get('NomeDaVariavel', 0)` para ler uma variável compartilhada e implementar o sincronismo.

O exemplo ilustrado na Figura 10 tem três processadores. O processador 1 aguarda a variável `ready` ter o valor 2 para somar as variáveis x e y. Os processadores 2 e 3 inicializam as variáveis compartilhadas x e y, respectivamente. Depois, os processadores 2 e 3 incrementam a variável compartilhada `ready` para sinalizar ao processador 1 que concluíram a tarefa. De posse do exemplo, os estudantes têm como tarefa criar simuladores com mais processadores, outras primitivas de sincronização e elaborar exemplos de código paralelo. Novamente, o editor permite mudar o código e testar novas implementações.

No primeiro semestre de 2025, a tarefa dos estudantes da disciplina de arquitetura de computadores da Universidade Federal de Viçosa foi programar o algoritmo K-NN (*K-Nearest Neighbors*) distribuído em um multiprocessador com quatro processadores.

4.3.2 Troca de Mensagens

No modelo baseado em troca de mensagens, cada processador possui sua própria memória local e se comunica explicitamente com os demais por meio de envio e recepção de dados. O simulador inicial gerado pela ferramenta ChatGPT permite exercitar os conceitos de paralelismo distribuído e sincronização explícita.

A Figura 11 mostra um exemplo gerado pela ferramenta ChatGPT com três janelas com capacidade de editar o código e executar. O simulador encapsula a comunicação e criou as primitivas `send(processadorDestino,valor)` e `receive(processadorOrigem)` para comunicar os processadores e implementar o sincronismo. A tarefa dos estudantes também foi programar o algoritmo K-NN distribuído em um multiprocessador com quatro processadores.

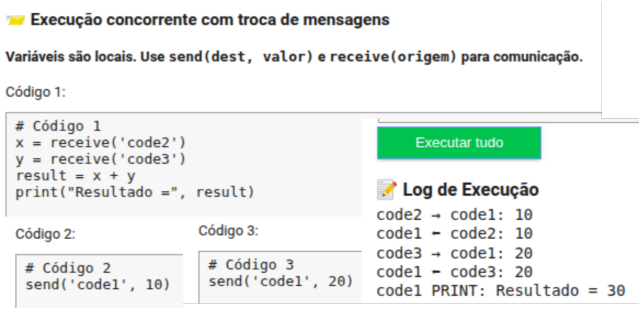

Figura 11. Código de demonstração de multiprocessador com troca de mensagens com um exemplo com três processadores.

A implementação do código base do simulador usou os mecanismos de *multi-thread* da linguagem Python. Portanto, os processadores são programados com a sintaxe de Python junto com as primitivas de comunicação, o que possibilita a validação de vários algoritmos paralelos em ambientes de multiprocessadores.

5 Memória cache 4-*Way*

A compreensão de hierarquias de memória é fundamental para o estudo de desempenho computacional. A descrição de detalhes de implementação de uma memória cache é um tópico que não está disponível nos livros-texto da área de arquitetura de computadores. Esta seção apresenta uma proposta de tarefa para os estudantes da disciplina de arquitetura de computadores para fazer a extensão de um simulador de cache 2-*way* [Ferreira *et al.*, 2023] para criar um novo simulador de uma cache 4-*way*. A ferramenta deve ter implementação validada em Verilog com visualização gráfica para depuração. As implementações da literatura mostram apenas formas de onda que são complexas para validação e códigos comportamentais sem uma visão estrutural do projeto. A construção de um simulador com visualização estrutural dos sinais permite ao aluno realizar várias sequências de acesso à memória, observando o funcionamento interno passo a passo. Foi solicitada a implementação da política de substituição LRU (*Least Recently Used*) com uma solução distribuída e escalável para cache com maior associatividade com uma cache 8-way. O objetivo é observar que, mesmo em caso de acerto (*hit*), a cache tem várias tarefas paralelas para realizar internamente, dando uma visão mais aprofundada das arquiteturas de caches.

Com relação a trabalhos da literatura, um projeto Verilog foi apresentado em [Chauan *et al.*, 2015]. Porém, o código não está disponível e o trabalho apenas apresenta a descrição em alto nível da implementação. Outros trabalhos recentes apresentaram projetos em VHDL [Kaur *et al.*, 2021; Hazlan *et al.*, 2023], mas não detalham o projeto da política de substituição do controlador. Alguns trabalhos abordam as implementações do controlador LRU ou pseudo-LRU [Omran and Amory, 2018; Puidenko and Kharchenko, 2020], mas não fornecem o código-fonte. Para uma cache 4-*way*, o código-fonte foi apresentado em [Patel, 2021], porém para uma política de substituição FIFO. Duas implementações em Verilog estão disponíveis em [Airin, 2015], mas o código tem uma máquina de estados complexa e não está documentado. Em todos os exemplos, a depuração é complexa usando apenas formas de onda.

Nosso ponto de partida é um projeto documentado que possui implementações de uma cache de mapeamento direto e 2-*way* [Ferreira *et al.*, 2023], parametrizadas e divididas em módulos em Verilog, com documentação em um Google Colab e uma visualização estrutural do projeto. A tarefa dos estudantes foi a extensão para 4-*way*. Por se tratar de uma tarefa complexa, as LLMs foram usadas apenas para auxiliar na compreensão do código e comando *generate*, uma vez que o projeto base tinha uma estrutura modular, que é relativamente difícil de ser gerada por uma LLM com poucos *prompts*.

A base do projeto é o módulo distribuído LRU. Uma visão geral do esquemático da cache está ilustrada na Figura 12. O desenho foi implementado com rótulos no formato vetorial SVG, seguindo a proposta apresentada em [de Figueiredo *et al.*, 2024] pelos estudantes da disciplina de arquitetura de computadores de 2025. Cada sinal a ser monitorado possui o rótulo com o prefixo `"@"`. A simulação é executada em Verilog, onde a cada ciclo são gravados os valores dos sinais. Estes valores são aplicados no arquivo SVG para substituir os rótulos com os valores, possibilitando a geração de uma animação e visualização passo a passo.

A Figura 13 destaca um dos quatro módulos que compõem a cache. Além dos campos tradicionais de validação, *tag*, número de sequência do LRU e dados, cada módulo tem seu próprio atualizador da política LRU. O LRU mais recente é o menor valor, 0 no caso da 4-*way*, e o menos recente tem o valor 3. No caso de *hit*, o módulo recebe o seu valor de LRU e o valor do LRU do módulo onde aconteceu o *hit*. Se seu valor de LRU for menor, ele deve ser incrementado, fica menos recente, já que outro bloco foi acessado e será o mais recente. Se ele próprio é o bloco que está sendo acessado, ou seja, o mesmo valor de LRU, seu valor deve ser zerado. Se seu valor for maior que o valor do LRU que teve o *hit*, não é necessário ser alterado. Dessa maneira, o projeto é escalável. A parte mais complexa é a inicialização, quando a cache está vazia. A unidade LRU pode também tratar esta situação utilizando o número do módulo e os bits de validação em sequência.

A Figura 14 ilustra outra interface desenvolvida pelos estudantes para a depuração do código Verilog da cache 4-*way*, oferecendo uma visão mais funcional da cache e ocultando, em um primeiro momento, os detalhes de implementação. Todos os sinais exibidos são rótulos extraídos diretamente da simulação Verilog e automaticamente mapeados no desenho SVG. A parte inferior da figura destaca o módulo 0 da cache, exibindo os rótulos antes da substituição e evidenciando que todos os campos podem ser facilmente depurados. Esta interface é específica para uma cache 4-*way* com quatro blocos de 4 bytes cada. Para uma saída mais generalizada, uma alternativa é o uso de bibliotecas como *svgwrite* [Jamieson *et al.*, 2025].

6 Trabalhos Relacionados

No nível de circuitos, podemos destacar o simulador Digitaljs [Materzok, 2019], que utiliza a ferramenta Yosys para gerar o desenho esquemático de projetos descritos em Verilog, uma extensão voltada ao ensino com programação por blocos [Castro and Azevedo, 2020] e um projeto hierárquico de processador MIPS [Passe *et al.*, 2020]. Apesar da exis-

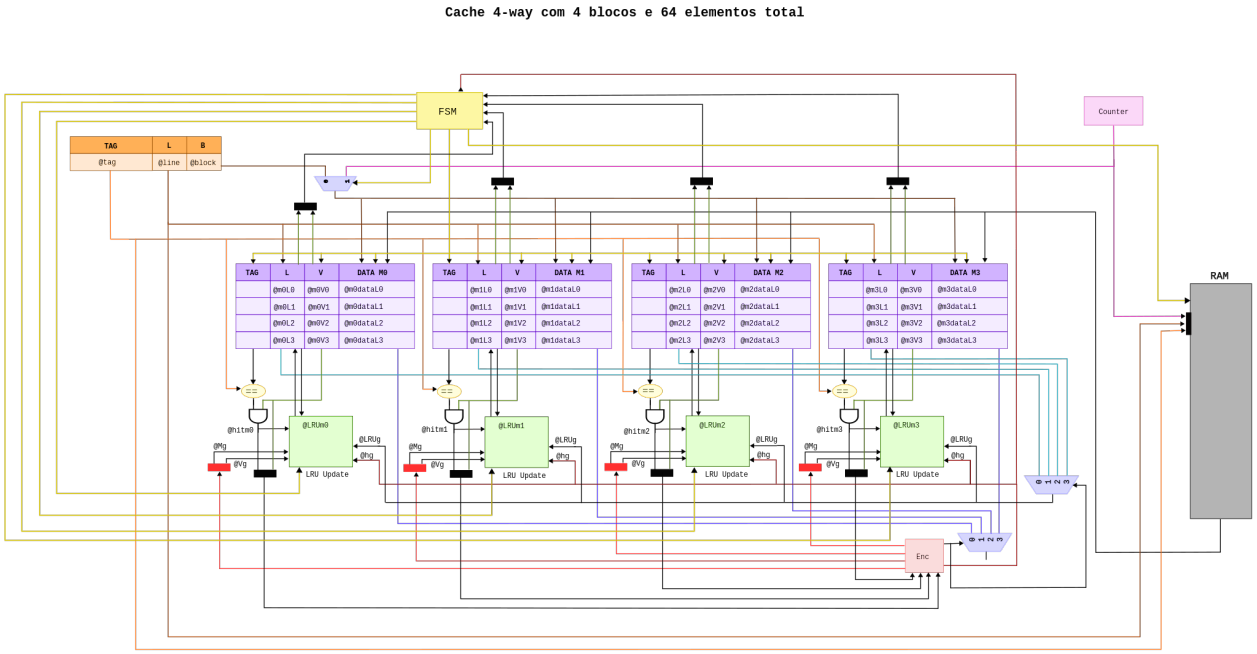

Figura 12. Caminho de dados da cache 4-*way* com módulos LRU locais em cada conjunto e formato editável para depuração do código Verilog.

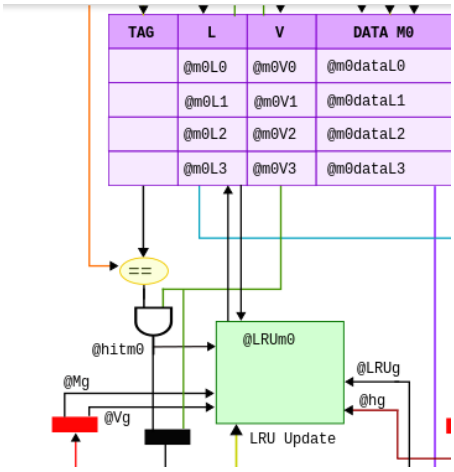

Figura 13. Módulo LRU e cache com campos: validação, *tag*, sequência LRU e bloco de dados.

tência de diversas ferramentas com interface visual [Siever *et al.*, 2025], a maioria delas não foi projetada para extensibilidade. Um levantamento de simuladores para arquitetura organiza várias ferramentas de código aberto [Materzok, 2019; Hwang *et al.*, 2025]; entretanto, essas ferramentas são voltadas para atividades de pesquisa e não foram desenvolvidas com foco no ensino de conceitos básicos em nível de graduação. Além disso, não apresentam facilidades para atualizações e extensões.

Trabalhos anteriores introduziram o uso do Google Colab para o ensino de circuitos digitais [Canesche *et al.*, 2021], o uso da biblioteca Gradio e do protocolo MQTT para acesso remoto interativo [Ferreira *et al.*, 2024a], o uso de depuração com interface gráfica editável em SVG para o processador RISC-V e a aplicação de LLMs em exercícios de codificação [de Figueiredo *et al.*, 2024], a visualização gráfica com apoio das bibliotecas *svgwrite* e Graphviz [Jamieson *et al.*,

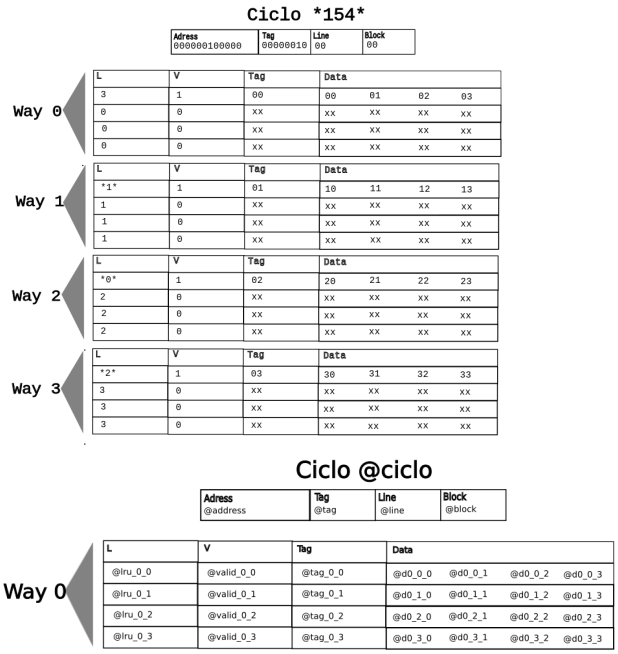

Figura 14. Código de demonstração multiprocessador com troca de mensagens e exemplo com três processadores.

2025], além de simuladores de cache e processadores em nível *assembly* [Jamieson *et al*., 2025; Ferreira *et al*., 2023].

Este trabalho é uma extensão dessas iniciativas. O primeiro aspecto é a incorporação da emulação de processadores RISC-V com um montador *assembly* integrado a um simulador Verilog do processador, incluindo depuração gráfica em SVG. A mesma metodologia foi adotada no simulador de memória cache 4-*way*, com projeto detalhado e documentado. Além disso, destacam-se a avaliação e construção de diversos simuladores funcionais para processadores com múltiplas unidades funcionais e escalonamento dinâmico utilizando o algoritmo de Tomasulo. Finalmente, explora-se o uso de LLMs para simuladores e a criação de linguagens de domínio específico (DSLs) voltadas a diferentes arquiteturas paralelas.

No melhor conhecimento dos autores, não existem ferramentas didáticas para o ensino de arquiteturas paralelas, como processadores vetoriais, *array processors* e multiprocessadores que ofereçam: (i) programação via editor de texto; (ii) código-fonte acessível e estruturado para extensões; e (iii) suporte à criação de linguagens de domínio específico que expressem o paralelismo com base nas restrições arquiteturais. Nesse sentido, este trabalho é inovador.

Um exemplo de *assembly* MIPS vetorial como extensão do simulador MARS foi apresentado por [Alves *et al*., 2015]. Em relação a *array processors*, apenas simuladores voltados a arquiteturas específicas com foco em pesquisa foram apresentados [Barr and Dudek, 2008; Herbordt *et al*., 2000], além de exemplos de aplicação de *array processors* em supercomputadores [Potter and Meilander, 2002].

7 Conclusão

Este trabalho apresenta uma sequência integrada de simuladores desenvolvidos com finalidade didática para o ensino de processadores RISC-V, com implementação em Verilog, simuladores de memória cache também descritos em Verilog, além de diversos simuladores de arquiteturas paralelas de alto nível, baseados em modelos de linguagem generativa (LLMs) para a criação automática de códigos. Esses simuladores foram utilizados como material de apoio na disciplina de arquitetura de computadores, ofertada no primeiro semestre de 2025 na Universidade Federal de Viçosa. O objetivo pedagógico consistiu em disponibilizar aos estudantes plataformas de simulação interativas, nas quais as tarefas práticas incluíram tanto a extensão funcional dos simuladores quanto a elaboração de exemplos aplicados.

Foram desenvolvidos simuladores específicos para os principais tópicos abordados na disciplina: processadores RISC-V, memória cache 4-*way*, algoritmo de Tomasulo, processadores com pipeline e múltiplas unidades funcionais de latência variável, processadores vetoriais, *array processors* e sistemas multiprocessados. Todos os simuladores incorporam um editor de texto interativo que permite a modificação do código de entrada e sua reexecução imediata. A implementação em ambiente Google Colab assegura portabilidade e facilidade de uso, eliminando a necessidade de instalações e configurações adicionais. Um dos principais diferenciais deste trabalho é o uso de LLMs na criação de linguagens de domínio específico com primitivas voltadas à programação paralela. Essas linguagens foram utilizadas para ilustrar a implementação de algoritmos paralelos e validação.

Como direções para trabalhos futuros, pretende-se generalizar as linguagens desenvolvidas por meio da formalização de gramáticas específicas, aliadas ao uso de LLMs e ao suporte de bibliotecas baseadas em avaliação preguiçosa (*lark*) em Python. Tal abordagem visa a construção de simuladores mais flexíveis, passíveis de extensão e reutilização em múltiplos contextos educacionais e de pesquisa.

Declarações complementares

Agradecimentos

Gostaríamos de agradecer a colaboração de todos os estudantes da turma do primeiro semestre de 2025 da disciplina INF450 arquitetura de computadores da Universidade Federal de Viçosa.

Financiamento

Apoio financeiro do Projeto FAPEMIG APQ-01577-22, CNPq e CAPES.

Contribuições dos autores

O autor Ricardo Ferreira elaborou os exemplos base dos simuladores e realizou a redação do texto. O autor Racyus Delano Garcia Pacífico colaborou com sugestões e revisão do texto.

Conflitos de interesse

Os autores declaram não haver conflitos de interesse.

Disponibilidade de dados e materiais

As ferramentas desenvolvidas neste trabalho são de código aberto e estão disponíveis no link `https://colab.research.google.com/drive/1L80E1YoeXnuQxKSNcm1KxJp-xDGZAiHC?usp=sharing`.

Outras informações relevantes

O texto deste artigo é de responsabilidade dos autores, onde ferramentas de IA foram usadas apenas para revisão ortográfica e gramatical, além de algumas sugestões. O tema do trabalho é sobre o uso de IA, neste aspecto os modelos de IA foram avaliados para geração dos simuladores apresentados.

Referências

Airin (2015). Verilog-caches. Available at: `https://github.com/airin711/Verilog-caches` Accessed: 2025-07-19.

Alves, F. A., Almeida, D., Bragança, L., Gomes, A. B., Ferreira, R. S., and Nacif, J. A. M. (2015). Ensinando arquiteturas vetoriais utilizando um simulador de instruções mips. *International Journal of Computer Architecture Education*, 4(1):9–12. Available at: `https://www2.sbc.org.br/ceacpad/ijcae/v4_n1_dec_2015/IJCAE_v4_n1_dez_2015_paper_3_vf.pdf`.

Barr, D. R. and Dudek, P. (2008). A cellular processor array simulation and hardware prototyping tool. In *2008 11th International Workshop on Cellular Neural Networks and Their Applications*, pages 213–218. IEEE. DOI: 10.1109/cnna.2008.4588680.

Böseler, F., Walter, J., and Perjikolaei, B. R. (2022). A comparison of virtual platform simulation solutions for timing prediction of small risc-v based socs. In *2022 Forum*

on Specification & Design Languages (FDL), pages 1–8. IEEE. DOI: 10.1109/FDL56239.2022.9925667.

Canesche, M., Bragança, L., Neto, O. P. V., Nacif, J. A., and Ferreira, R. (2021). Google colab cad4u: Hands-on cloud laboratories for digital design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE. DOI: 10.1109/iscas51556.2021.9401151.

Castro, L. and Azevedo, R. (2020). Circuitly: A visual and constructive framework for teaching digital circuits. *International Journal of Computer Architecture Education*, 9(1):10–15. DOI: 10.5753/ijcae.2020.4839.

Chauan, P., Singh, G., and Singh, G. (2015). Cache controller for 4-way set-associative cache memory. *International Journal of Computer Applications*, 129(1):8887. DOI: 10.5120/ijca2015906787.

Coura, P., Freitas, I., Costa, H., Nacif, J., and Ferreira, R. (2025). Desmistificando o ensino de inteligência artificial e aprendizado de máquina. In *Simpósio Brasileiro de Educação em Computação (EDUCOMP)*, pages 25–27. SBC.

de Figueiredo, G. A., de Souza, E. S., Rodrigues, J. H., Nacif, J. A., and Ferreira, R. (2024). Desenvolvendo ferramentas para ensino de risc-v com python, verilog, matplotlib, svg e chatgpt. *International Journal of Computer Architecture Education*, 13(1):43–52. DOI: 10.5753/ijcae.2024.5343.

Esmeraldo, G. Á. R., Feitosa, R. G. F., da Silva Barros, E. N., Proto, E. C. P. d. S., de Mello, H. M., Lisboa, E. B., Bispo Jr, E. L., and de Campos, G. A. L. (2023). Uma abordagem para ensino-aprendizado de projetos de sistemas computacionais com utilização do simulador compsim com suporte à arquitetura risc-v. *Revista Brasileira de Informática na Educação*, 31:271–288. DOI: 10.5753/rbie.2023.2951.

Ferreira, R., Canesche, M., Jamieson, P., Neto, O. P. V., and Nacif, J. A. (2024a). Examples and tutorials on using google colab and gradio to create online interactive student-learning modules. *Computer Applications in Engineering Education*, 32(4):e22729. DOI: 10.1002/cae.22729.

Ferreira, R., Canesche, M., and Penha, J. (2023). Google colab para ensino de computação. In *Simpósio Brasileiro de Educação em Computação (EDUCOMP)*, pages 46–47. SBC. DOI: 10.5753/educomp$_{e}stendido$.2023.228279.

Ferreira, R., Nacif, J., Magalhaes, S., de Almeida, T., and Pacifico, R. (2015). Be a simulator developer and go beyond in computing engineering. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE. DOI: 10.1109/fie.2015.7344416.

Ferreira, R., Sabino, C., Canesche, M., Neto, O. P. V., and Nacif, J. A. (2024b). Aiot tool integration for enriching teaching resources and monitoring student engagement. *Internet of Things*, 26:101045. DOI: 10.1016/j.iot.2023.101045.

Garcia, M., Niyaz, Q., Yang, X., Javaid, A. Y., and Paheding, S. (2024). An interactive visualization tool for computer organization and design course. In *2024 IEEE International Conference on Electro Information Technology (eIT)*, pages 457–462. IEEE. DOI: 10.1109/eit60633.2024.10609897.

Giorgi, R. and Mariotti, G. (2019). Webrisc-v: A web-based education-oriented risc-v pipeline simulation environment. In *Proceedings of the workshop on computer architecture education*, pages 1–6. DOI: 10.1145/3338698.3338894.

Hazlan, M. A. A.-Z., Gunawan, T. S., Yaacob, M., Kartiwi, M., and Arifin, F. (2023). Design and performance analysis of a fast 4-way set associative cache controller using tree pseudo least recently used algorithm. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)*, 11(4):1051–1063. DOI: 10.52549/.v11i4.5014.

Herbordt, M. C., Cravy, J., Sam, R., Kidwai, O., and Lin, C. (2000). A system for evaluating performance and cost of simd array designs. *Journal of Parallel and Distributed Computing*, 60(2):217–246. DOI: 10.1006/jpdc.1999.1602.

Hwang, I., Lee, J., Kang, H., Lee, G., and Kim, H. (2025). Survey of cpu and memory simulators in computer architecture: A comprehensive analysis including compiler integration and emerging technology applications. *Simulation Modelling Practice and Theory*, 138:103032. DOI: 10.1016/j.simpat.2024.103032.

Jamieson, P., Ferreira, R., and Nacif, J. (2025). Leveraging large language models to create interactive online resources for digital systems and computer architecture education. In *ASEE Annual Conference Exposition*. Available at: `https://peer.asee.org/board-72-leveraging-large-language-models-to-create-interactive-online-resources-for-digital-systems-and-computer-architecture-education`.

Jamieson, P., Ferreira, R., and Nacif, J. A. (2021). Personalizing online computer engineering resources and labs for digital, embedded, and computer system courses. In *2021 IEEE frontiers in education conference (FIE)*, pages 1–5. IEEE. DOI: 10.1109/fie49875.2021.9637244.

Kaur, G., Arora, R., and Panchal, S. S. (2021). Implementation and comparison of direct mapped and 4-way set associative mapped cache controller in vhdl. In *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 1018–1023. IEEE. DOI: 10.1109/spin52536.2021.9566081.

Materzok, M. (2019). Digitaljs: A visual verilog simulator for teaching. In *Proceedings of the 8th Computer Science Education Research Conference*, pages 110–115. DOI: 10.1145/3375258.3375272.

Mezger, B. W., Santos, D. A., Dilillo, L., Zeferino, C. A., and Melo, D. R. (2022). A survey of the risc-v architecture software support. *IEEE Access*, 10:51394–51411. DOI: 10.1109/access.2022.3174125.

Omran, S. S. and Amory, I. A. (2018). Implementation of lru replacement policy for reconfigurable cache memory using fpga. In *2018 International Conference on Advanced Science and Engineering (ICOASE)*, pages 13–18. IEEE. DOI: 10.1109/icoase.2018.8548892.

Passe, F., Canesche, M., Neto, O. P. V., Nacif, J. A., and Ferreira, R. (2020). Mind the gap: Bridging verilog and computer architecture. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE. DOI: 10.1109/iscas45731.2020.9180650.

Patel, R. (2021). 4-way-set-associative-cache-verilog. Available at: `https://github.com/rajshadow/4-way-`

`set-associative-cache-verilog` Accessed: 2025-07-19.

Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. The Morgan Kaufmann Series. Book.

Potter, J. L. and Meilander, W. C. (2002). Array processor supercomputers. *Proceedings of the IEEE*, 77(12):1896–1914. DOI: 10.1109/5.48831.

Puidenko, V. and Kharchenko, V. (2020). The minimizating of hardware for implementation of pseudo lru algorithm for cache memory. In *2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 65–71. IEEE. DOI: 10.1109/dessert50317.2020.9125054.

Railing, B. P. (2023). Cadss: Computer architecture design simulator for students. In *Proceedings of the Workshop on Computer Architecture Education*, pages 34–40. DOI: 10.1145/3605507.3610626.

Savaton, G. (2021). A visual simulator for teaching computer architecture using the risc-v instruction set. *Guillaume-Savaton-ESEO/emulsiV*. Available at: https://-github.com/ESEO-Tech/emulsiV.

Schoeberl, M. (2025). Wildcat: Educational risc-v microprocessors. *arXiv preprint arXiv:2502.20197*. DOI: 10.1007/978-3-032-03281-$2_1$3.

Siever, B., Hall, M., Feher, J., and Chamberlain, R. (2025). Teaching digital logic and computer architecture using open source tools. In *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, pages 38–41. DOI: 10.1145/3706594.3726971.

Wheeler, D. J. and Needham, R. M. (1994). Tea, a tiny encryption algorithm. In *International workshop on fast software encryption*, pages 363–366. Springer. DOI: 10.1007/3-540-60590-$8_2$9.

Zekany, S. A., Tan, J., and Connolly, J. A. (2021). Teaching out-of-order processor design with the risc-v isa. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, pages 1–8. IEEE. DOI: 10.1109/wcae53984.2021.9707143.