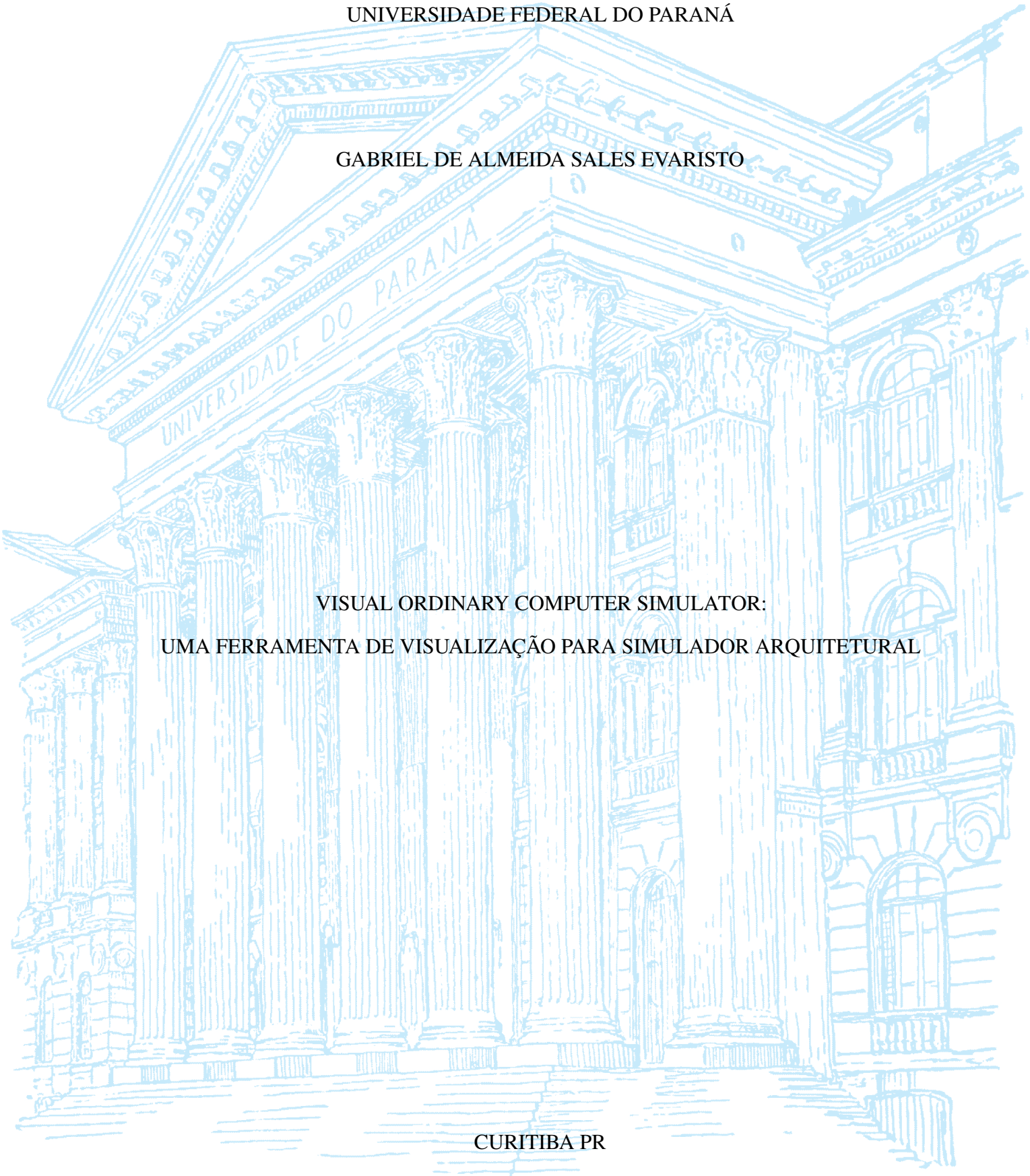
UNIVERSIDADE FEDERAL DO PARANÁ

GABRIEL DE ALMEIDA SALES EVARISTO



VISUAL ORDINARY COMPUTER SIMULATOR:

UMA FERRAMENTA DE VISUALIZAÇÃO PARA SIMULADOR ARQUITETURAL

CURITIBA PR

2024

GABRIEL DE ALMEIDA SALES EVARISTO

VISUAL ORDINARY COMPUTER SIMULATOR:

UMA FERRAMENTA DE VISUALIZAÇÃO PARA SIMULADOR ARQUITETURAL

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA PR

2024

Universidade Federal do Paraná
Setor de Ciências Exatas
Curso de Ciência da Computação

Ata de Apresentação de Trabalho de Conclusão de Curso 2

Título do Trabalho: VISUAL ORDINARY COMPUTER SIMULATOR: UMA FERRAMENTA DE VISUALIZAÇÃO PARA SIMULADOR ARQUITETURAL

Autor(es):

GRR20165266 Nome:GABRIEL DE ALMEIDA SALES EVARISTO

Apresentação: Data: 31/07/2024 Hora: 13h30 Local: Laboratório HIPES

Orientador: MARCO ANTONIO ZANATA ALVES

Membro 1: SAIRO RAONÍ DOS SANTOS

Membro 2: JORGE PIRES CORREIA

(nome) (assinatura)

AVALIAÇÃO – Produto escrito		ORIENTADOR	MEMBRO 1	MEMBRO 2	MÉDIA
Conteúdo	(00-40)				
Referência Bibliográfica	(00-10)				
Formato	(00-05)				
AVALIAÇÃO – Apresentação Oral					
Domínio do Assunto	(00-15)				
Desenvolvimento do Assunto	(00-05)				
Técnica de Apresentação	(00-03)				
Uso do Tempo	(00-02)				
AVALIAÇÃO – Desenvolvimento					
Nota do Orientador	(00-20)		**********	**********	
NOTA FINAL		**********	**********	**********	90

Os pesos indicados são sugestões.

Conforme decisão do colegiado do curso de Ciência da Computação, a entrega dos documentos comprobatório de trabalho de Conclusão de Curso 2 deve deve respeitar os seguintes procedimentos: o orientador deve abrir um processo no Sistema Eletrônico de Informações (SEI – UFPR); Selecionar o tipo: *Graduação: Trabalho Conclusão de Curso*; informar os interessados: nome do aluno e o nome do orientador; anexar esta ata escaneada e a versão final do PDF da monografia do aluno; Tramitar o processo para CCOMP (Coordenação de Ciência da Computação).

gov.br Documento assinado digitalmente
SAIRO RAONI DOS SANTOS
Data: 02/08/2024 18:53:16-0300
Verifique em https://validar.iti.gov.br

AGRADECIMENTOS

Gostaria de agradecer primeiramente meus pais, que foram os maiores responsáveis pela oportunidade que tive de fazer uma graduação, Nunca me deixaram trabalhar durante a adolescência e sempre exaltaram e destacaram o papel da edução na vida. Gostaria de agradecer a minha irmã Isabela por sempre ouvir minhas ideias de projetos relacionados à computação e por escutar as mais variadas curiosidades de Computação.

Deixo meus agradecimentos a todos as pessoas que conheci durante a graduação, meus colegas de classe, meus colegas de mobilização estudantil e colegas de outros cursos. Sem dúvidas a transformação que sofri durante a graduação teria sido muito limitada se não fosse por todas estas pessoas. Em especial deixo meus agradecimentos para meus irmãos Gabriel e Amanda, dividimos felicidades, alegrias, horas de estudo e planos pro futuro.

Agradeço minha psicóloga, Fernanda Pedroza, por me ajudar a desenvolver as ferramentas que hoje uso pra me organizar, e por me guiar nas horas em que tudo estava confuso e difícil.

Agradeço o corpo docente do curso de Ciência da Computação por tudo o que me ensinaram e pela dedicação. Destaco aqui e deixo meus agradecimentos ao Professor Zanata, pela paciência e orientação.

Por último, mas muito importante. Agradeço a minha namorada Gabi Keiber. Obrigada por todos os momentos de acolhimento durante os últimos períodos, obrigada por me dar esperança quando tudo estava tão confuso e difícil que eu cogitava parar. Obrigada pelo carinho e pela atenção genuína durante meus momentos de empolgação, explicando e ponderando sobre temas complexos de Computação, amo você completamente.

RESUMO

Computadores têm sido aplicados na resolução de problemas cada vez mais complexos, com maior volume de dados ou em ambientes de execução com restrição em termos de memória, energia ou tempo de resposta. Essa complexidade das aplicações gera a necessidade de otimização dos computadores e isso se manifesta em uma busca incessante por desempenho, que tem influência direta na pesquisa em arquitetura de computadores. É comum que projetos nessa área precisem monitorar o estado dos componentes da microarquitetura para identificar gargalos, encontrar *bugs* e propor otimizações. Os simuladores são ferramentas muitas vezes essenciais para esses projetos, por permitirem que o comportamento interno dos sistemas de hardware sejam observados mais facilmente, além de facilitarem o aprendizado de conceitos importantes da área. Esse trabalho apresenta uma ferramenta para visualização do estado de uma arquitetura simulada. O simulador alvo é o *Ordinary Computer Simulator* (OrCS) desenvolvido e utilizado no laboratório *High Performance and Efficient Systems* (HiPES) da Universidade Federal do Paraná (UFPR). O texto descreve o processo de desenvolvimento do visualizador, justifica as decisões de projeto e explica cada uma de suas funcionalidades. Como resultado, além da ferramenta de visualização desenvolvida é apresentado um estudo de caso feito com ela. Esse estudo mostra várias execuções de um mesmo programa utilizando diferentes configurações do OrCS. A comparação do comportamento dessas execuções com o que era esperado perante as modificações feitas na configuração, tornou possível a comprovação de como a ferramente é capaz de auxiliar o pesquisador na análise do sistema simulado. Por fim o trabalho apresenta ideias de melhorias que a ferramenta pode receber como trabalhos futuros.

Palavras-chave: Simulação. Visualização. Arquitetura de Computadores.

ABSTRACT

Computers have been applied to solving increasingly complex problems, with larger volumes of data or in execution environments constrained in terms of memory, energy, or response time. This application complexity generates the need for computer optimization, manifesting in a relentless pursuit of performance, which directly influences research in computer architecture. It is common for projects in this area to need to monitor the state of microarchitecture components to identify bottlenecks, find bugs, and propose optimizations. Simulators are often essential tools for these projects, as they allow the internal behavior of hardware systems to be more easily observed and facilitate the learning of important concepts in the field. This work presents a tool for visualizing the state of a simulated architecture. The target simulator is the *Ordinary Computer Simulator* (OrCS) developed and used in the *High Performance and Efficient Systems* (HiPES) laboratory at the Federal University of Paraná (UFPR). The text describes the development process of the visualizer, justifies design decisions, and explains each of its functionalities. As a result, in addition to the developed visualization tool, a case study conducted with it is presented. This study shows several executions of the same program using different OrCS configurations. Comparing the behavior of these executions with what was expected given the configuration modifications made it possible to demonstrate how the tool can assist researchers in analyzing the simulated system. Finally, ideas for improvements that the tool can receive as future work are presented.

Keywords: Simulation. Visualization. Computer Architecture.

LISTA DE FIGURAS

2.1 Um ábaco, um quipo e uma pascalina (da esquerda para direita). 14
2.2 Diagrama de blocos do Intel Skylake Server (Wikichip, 2017b). 18
2.3 Diagrama do OrCS, adaptado de (Tissei, 2021). 22
2.4 Diagrama do OrCS Simplificado, adaptado de (Tissei, 2021). 23
2.5 Processo de geração de ferramentas de visualização, adaptado de (Mazza, 2009). 25

3.1 A interface do NeanderSIM (Ullmann et al., 2014). 29
3.2 A interface do DEVSJava MIPS32 (Sarjoughian et al., 2008). 30
3.3 A interface do VisuSim (Imai et al., 2006). 31
3.4 A interface do CompSIm (Esmeraldo e Lisboa, 2017). 32
3.5 A interface do Ripes (Petersen, 2021a). 33
3.6 A interface do M2S-Visual (Ziabari et al., 2015). 34
3.7 Diagramas de tempo para uma CPU (a) e uma GPU (b) no M2S-Visual (Ziabari et al., 2015). 35
3.8 Representação de módulo de memória cache no M2S-Visual (Ziabari et al., 2015). 35
3.9 A interface do DIMIPSS (Felix et al., 2006). 36
3.10 A interface do MARS (Vollmar e Sanderson, 2006). 37
3.11 A interface do RARS (Landers, 2017). 38
3.12 A interface do QtSpim. 39

4.1 Diagrama de classes do VOrCS. 42
4.2 Janela principal do VOrCS. 46
4.3 Buffers posicionados pela opção "Tile". 48
4.4 Redimensionamento e reposicionamento dos Buffers. 48
4.5 Buffers com pacotes em seu interior. 49
4.6 Janelas de informações extras de pacotes no *Fetch* e no *ROB*. 49
4.7 Área do comando "Avançar" com valores padrões. 50
4.8 Área do comando "Avançar" com valores modificados e seletor aberto. 50
4.9 Área do comando "Avançar até" com valores padrões. 50
4.10 Resultado de execução de "Avançar até". 51
4.11 Área para seleção do modo de execução da ferramenta. 51
4.12 Área para o controle da duração do intervalo do modo contínuo. 52
4.13 Histogramas de ocupação e seus *buffers*. 52
4.14 Histogramas de fases e seus *buffers*. 53
4.15 Histogramas de ocupação para a execução sem restrições. 54

4.16 Histogramas de ocupação para a execução com "Fetch" de tamanho 1 e estágio com largura 1.. 56
4.17 Histogramas de ocupação para a execução com "Decode" tamanho 8, e com a largura do estágio associado igual a 1. 57
4.18 Histogramas de ocupação para a execução com "Unified_RS" de tamanho 13, estágio *Dispatch* com largura 1 e Unidades Funcionais reduzidas.. 59
4.19 Histogramas de ocupação para a execução com "ROB" de tamanho 28 e estágio *Commit* com largura 1. 60
4.20 Histogramas de ocupação para a execução com "ROB"de tamanho 1.. 62
4.21 Histogramas de ocupação para a execução com restrições na Memória. 63
4.22 Histogramas de performance para execuções dos estudos de caso.. 64
4.23 Performance para o comando avançar em sessão sem renderização de histograma. 65
4.24 Performance para o comando avançar em sessão sem renderização de histograma. 66

LISTA DE TABELAS

2.1 Atributos da classe opcode_package_t. 20
2.2 Alguns atributos da classe uop_package_t. 21

3.1 Classificação dos trabalhos correlatos pesquisados. 28

4.1 Classificação das bibliotecas gráficas pesquisadas. 41

LISTA DE ACRÔNIMOS

HiPES	*High Performance and Efficient Systems*
OrCS	*Ordinary Computer Simulator*
UFPR	Universidade Federal do Paraná
FPGA	*Field Programmable Gate Array*
CAD	*Computer-Aided Design*
ISA	*Instruction Set Architecture*
CPU	*Central Processing Unit*
RISC	*Reduced Instruction Set Computer*
CISC	*Complex Instruction Set Computer*
MIPS	*Microprocessor without Interlocked Pipelined Stages*
ARM	*Advanced RISC Machine*
ULA	Unidade Lógico-Aritmética
bits	*binary digits*
NOP	*No Operation*
HIVE	*High-performance Integrated Virtual Architecture*
VIMA	*Vetorial Integer and Memory Architecture*
SiNUCA	*Simulator of Non-Uniform Caches*
MOB	*"Memory Order Buffer"*
ROB	*"Reorder Buffer"*
URS	*"Unified Reservation Station"*
UFU	*"Unified Functional Units"*
IPC	*Instructions per Cycle*
VOrCS	*Visual Ordinary Computer Simulator*
DEVS	*Discrete Event System Specification*
M2S	*Multi2Sim*
DIMIPSS	Didatic Interative MIPS Simulator
MARS	MIPS Assembler and Runtime Simulator
RARS	RISC-V Assembler and Runtime Simulator
VSRTL	*Visual Simulation of Register Transfer Logic*
GPU	*Graphics Processing Unit*
AMD	*Advanced Micro Devices*
MOESI	*Modified, Owned, Exclusive, Shared and Invalid*
MDI	*Multiple Document Interface*

SUMÁRIO

1 INTRODUÇÃO 11
1.1 PROBLEMA DE PESQUISA E MOTIVAÇÃO 11
1.2 OBJETIVOS 13
2 REFERENCIAL TEÓRICO 14
2.1 COMPUTADORES COMO MÁQUINAS PROGRAMÁVEIS 14
2.2 A ARQUITETURA X86 E A MICROARQUITETURA INTEL 17
2.3 O SIMULADOR ORCS (ORDINARY COMPUTER SIMULATOR) 19
2.4 VISUALIZAÇÃO 25
2.5 O FORMATO PAJÉ 26
3 TRABALHOS CORRELATOS. 27
3.1 SUMÁRIO DOS TRABALHOS ENCONTRADOS 27
3.2 NEANDERSIM 29
3.3 DEVSJAVA MIPS32 30
3.4 VISUSIM 31
3.5 COMPSIM 32
3.6 RIPES 32
3.7 M2S-VISUAL 33
3.8 DIMIPSS 36
3.9 MARS 37
3.10 RARS 38
3.11 QTSPIM. 38
3.12 VORCS: VISUAL ORDINARY COMPUTER SIMULATOR 39
4 PROPOSTA 40
4.1 AMBIENTE DE DESENVOLVIMENTO 40
4.2 BIBLIOTECA GRÁFICA 40
4.3 ESTRUTURAÇÃO DE CÓDIGO E FUNCIONAMENTO 41
4.3.1 Arquivo de entrada 42
4.4 PROGRAMA 46
4.4.1 Buffers. 47
4.4.2 Pacotes 48
4.4.3 Funcionalidades 50
4.5 RESULTADOS 53
4.5.1 Estudo de Caso 1: Restrição no *Fetch* 55
4.5.2 Estudo de Caso 2: Restrição no *Decode* 57

4.5.3 Estudo de Caso 3: Restrição na UFU e na URS 58
4.5.4 Estudo de Caso 4: Restrição no ROB 60
4.5.5 Estudo de Caso 5: ROB de tamanho 1 61
4.5.6 Caso de Estudo 6: Restrição na Memória 62
4.5.7 Performance nos estudos de caso 64
4.5.8 Performance dos modos de execução 65
4.5.9 Conclusões sobre os experimentos 66
5 CONCLUSÕES 68
6 LIMITAÇÕES E TRABALHOS FUTUROS 69
6.1 MUDANÇAS NO TRATAMENTO DOS EVENTOS 69
6.1.1 Retroceder visualização 69
6.2 OTIMIZAÇÕES NA ESTRUTURA DE DADOS DO TRAÇO 70
6.3 MELHORIAS PARA O USO DA FERRAMENTA COM O ORCS 70
6.3.1 Visualização de registradores e Unidades funcionais 71
6.3.2 Visualização da memória 71
REFERÊNCIAS 72
APÊNDICE A – ARQUIVO DE ENTRADA PAJÉ RESUMIDO 75
APÊNDICE B – INSTRUÇÕES PARA A UTILIZAÇÃO DA FERRAMENTA. 77

1 INTRODUÇÃO

A pesquisa em arquitetura de computadores comumente envolve a análise de desempenho dos sistemas computacionais. De acordo com Ramos (2008), após a escolha de um aspecto a ser analisado, um conjunto de métricas adequado deve ser escolhido para guiar a coleta de dados. Essa coleta envolve necessariamente a execução de uma carga de trabalho, que leve o sistema a um estado diferente do natural. Ela pode ser executada em um computador, um simulador ou um protótipo em *Field Programmable Gate Array* (FPGA), por exemplo. Já os dados colhidos podem ser usados para avaliar o desempenho de novos *designs*, validar ou explorar variações de mecanismos de *hardware*. Essa análise é um passo importante para que o pesquisador possua informações que resultem em algum entendimento sobre o *hardware* que o leve a alguma ideia de melhoria, ou para que possa validar uma otimização.

Normalmente, é impraticável que um pesquisador construa algo físico a cada iteração de um projeto de pesquisa em arquitetura de computadores devido ao alto custo associado a isso (Akram e Sawalha, 2016). Dessa maneira, modelos matemáticos e ferramentas de *software* como simuladores ou programas de *Computer-Aided Design* (CAD) são aplicados nas diferentes fases de um projeto. Assim, é possível criar, otimizar e, enfim, materializar os projetos de uma maneira viável.

Simuladores arquiteturais são ferramentas muito utilizadas na pesquisa e no ensino de arquitetura e organização de computadores. White e Ingalls (2015) descrevem o processo de simulação na pesquisa como um conjunto de testes de campo, onde o sistema real é substituído por um modelo computacional, capaz de imitar comportamentos de interesse. Esse modelo é utilizado para gerar observações desses comportamentos, para uma tentativa de entendê-los ou generalizá-los.

A ferramenta desenvolvida nesse trabalho tem como alvo o OrCS, que é um simulador arquitetural baseado em traços. Akram e Sawalha (2019) definem esse tipo de simulador pelo arquivo que utiliza como entrada, denominado de traço de simulação. Esse arquivo de traço é gerado durante a execução de alguma carga de trabalho em uma máquina real, e possui uma gravação da sequência de instruções executadas. Essa sequência inclui informações como *opcodes* de instruções, endereços de memória, endereços de desvio, etc. O simulador lê os dados gravados e simula o comportamento da arquitetura, enquanto calcula métricas referentes a execução. O resultado desse cálculo pode então se tornar matéria-prima para gerar conclusões acerca dos tópicos explorados pelas pesquisas. Mesmo que o uso dessa técnica seja mais prático que a utilização de protótipos, ele não é livre de complicações, uma vez que os traços de simulação podem ser muito grandes, o que limita o número de instruções no traço e o número de traços em memória, o que pode levar a simulação a ser bem lenta.

1.1 PROBLEMA DE PESQUISA E MOTIVAÇÃO

Sistemas computacionais são complexos e envolvem muitos eventos e componentes. Quando é preciso uma granularidade muito fina para uma simulação, o processo se torna mais difícil, e em alguns casos, impraticável. São muitos sinais e componentes cujos estados precisam ser atualizados e latências consideradas. Seja pela frequência de atualização dos objetos ou simplesmente pela sua quantidade deles, acompanhar o estado de uma arquitetura mais realista e calcular as métricas sobre ela se torna um processo muito demorado. No entanto, essa ação pode ser muito útil, e talvez vital para as pesquisas em arquitetura de computadores.

Por essa razão, pesquisadores aplicam abordagens para simplificar esse processo, como: simular apenas parte do sistema alvo, simplificar detalhes arquiteturais ou resumir o traço de simulação.

Ainda assim, pode ser necessária a otimização do código do simulador, e com isso, suas estruturas de dados e seu funcionamento podem perder cada vez mais qualquer semelhança com o sistema-alvo. Isso introduz uma incompatibilidade cada vez maior entre a compreensão do funcionamento do simulador e do funcionamento da estrutura real que ele representa. Assim, embora que as otimizações sejam responsáveis pela viabilização de diversas pesquisas, através da diminuição do tempo de simulação e da conservação da validade das métricas extraídas, elas tornam mais difícil a modificação e a manutenção da validade do simulador. Isso pode facilitar que o pesquisador caia em algumas das armadilhas relacionadas com o uso de simuladores (Nowatzki et al., 2015). Essa incompatibilidade mencionada pode fazer com que potenciais *bugs* ou erros de modelagem na simulação passem desapercebidos. Junto com a dificuldade de validação, isso pode tornar os resultados adquiridos menos significantes e pode levar os pesquisadores à conclusões errôneas. Basta que o seu objeto de pesquisa não seja isolado ou estatisticamente não correlacionado com a causa dos erros de simulação.

O uso de *logs* impressos em terminal que registrem a ordem de mudanças que algum sinal sofreu, a impressão sistemática de várias das variáveis de um simulador em um teste de mesa ou o uso de depuradores podem expor mais facilmente o comportamento do código, mas podem gerar confusão quando o objetivo é a interpretação do que está acontecendo na arquitetura real simulada. A procura de soluções para esse problema tem a visualização, nesse caso, o uso de estruturas visuais em conjunto com informações de *logs* de execução, como uma uma abordagem promissora. Levie e Lentz (1982) expõem diversas maneiras como a informação visual pode contribuir para a compreensão e retenção de informação textual. Isso justifica a eficácia da aplicação dessa abordagem, já que para o uso de diagramas e outras imagens não-representativas, basta que o usuário seja instruído sobre como utilizá-las.

Existem diversos exemplos de uso de visualização como estratégia para gerenciamento e análise de sistemas complexos. A mais elementar dessas estratégias na área das ciências exatas é o uso de gráficos para visualização do comportamento de alguma função. Shneiderman e Plaisant (2019) trazem diversos exemplos do uso de ferramentas de visualização na mineração de dados, chegando a afirmar que as ferramentas aumentavam a capacidade dos analistas em lidar com a complexidade de processos reais, além de facilitar a identificação de erros e desvios dos padrões esperados.

Já Mintzer e Snodgrass (1999) discorrem sobre o efeito da superioridade da imagem, que afirma que a vantagem da imagem sobre a palavra se deve ao fato de que em contraste às palavras, a imagem tem características altamente distintivas, tanto visualmente quanto semanticamente, o que permite que elas sejam gravadas unicamente na memória e que passem por um processo de conceptualização maior que palavras, isso tudo permite que sejam lembradas mais facilmente e que possuam uma vantagem mnemônica sobre o texto.

Assim, a implementação de uma ferramenta de visualização, agrupada com a definição de uma representação visual significativa para o estado de um simulador, parece ter uma boa chance de tornar a observação e compreensão do estado do simulador uma tarefa mais fácil.

Com isso em mente, foi elaborada a seguinte pergunta de pesquisa: Como a implementação de uma ferramenta de visualização interativa pode facilitar a observação e compreensão do estado de uma arquitetura simulada no OrCS, e quais são os impactos de tal ferramenta na identificação de erros e na eficiência do processo de simulação?

1.2 OBJETIVOS

De maneira geral, esse trabalho procura ter como resultado uma ferramenta de visualização que facilite a observação do estado de uma arquitetura simulada no OrCS. O trabalho também visa comprovar essa capacidade da ferramenta conduzindo alguns estudos de caso. A partir disso é possível listar objetivos mais concretos.

Como objetivo geral, o trabalho quer:

- Criar uma ferramenta de visualização para o simulador arquitetural OrCS.

E foram definidos os seguintes objetivos específicos:

- A ferramenta deve mostrar na tela uma representação da arquitetura, que se assemelhe a estrutura do código do simulador.
- A ferramenta deve permitir a visualização do estado do simulador de maneira a compreender o comportamento da arquitetura alvo e também do simulador em si.
- A ferramenta deve ser capaz de mostra visualizações para diferentes execuções do simulador.
- A visualização da arquitetura deve ser atualizada e interativa, funcionando mais como uma animação do que como uma foto do simulador.

O restante desta monografia está dividido em 5 capítulos. O capítulo 2 apresenta alguns conceitos importantes para a compreensão da proposta desse trabalho e das escolhas feitas e também possui uma apresentação mais aprofundada do OrCS. No capítulo 3 é apresentado uma pesquisa bibliográfica de trabalhos que apresentam ferramentas de visualização de arquiteturas de computador. O capítulo 4 introduz ferramenta desenvolvida nesse trabalho, o ***Visual Ordinary Computer Simulator*** **(VOrCS)**, nesse capítulo também é apresentado um estudo de caso utilizando a ferramenta. No capítulo 5 são apresentadas as conclusões, e uma reflexão sobre o desenvolvimento da ferramenta. No último capítulo, são apresentadas ideias de melhorias que a ferramenta pode vir a receber.

2 REFERENCIAL TEÓRICO

Esse capítulo começa apresentando a diferença principal entre os computadores e outras máquinas de cálculos. São apresentados também os conceitos de arquitetura e microarquitetura, importantes para o entendimento do fluxo de execução de uma instrução. Após isso, ele apresenta a microarquitetura dos processadores Intel. Isso é importante para o entendimento do funcionamento do simulador OrCS, apresentado logo depois. O capítulo fecha com a descrição do formato escolhido para o arquivo de entrada da ferramenta de visualização e com o porquê de ser julgado adequado para essa aplicação.

2.1 COMPUTADORES COMO MÁQUINAS PROGRAMÁVEIS

Os computadores em sua essência são máquinas lógico-aritméticas, assim, eles aplicam operações sobre números e armazenam resultados. Eles são parte de uma longa evolução de máquinas utilizadas para a facilitação de trabalho intelectual, iniciada pela ideia de usar objetos físicos para representar informação (Evans, 2011b).

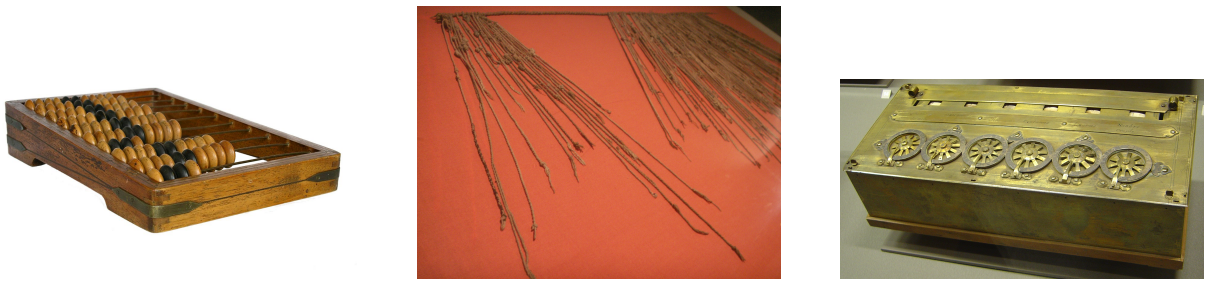


Figura 2.1: Um ábaco, um quipo e uma pascalina (da esquerda para direita).

Ao longo dos anos foram criadas uma série de máquinas com funcionalidades para fazer cálculos. A Figura 2.1 mostra três delas. Um dos exemplos é o ábaco, que é uma caixa retangular com uma série de hastes e bolinhas, chamadas de contas. Cada haste representava uma grandeza, como unidade, dezena ou centena e as bolinhas representam quantidades. Os ábacos permitiam que seus usuários executassem algumas operações matemáticas. Para isso, as contas eram arranjadas para representar um número e então manipuladas de maneira a performar a operação. A posição das contas após a manipulação representava o resultado final ou um resultado intermediário da operação (da Silva, 2021).

Os outros exemplos de máquinas mostrados são o quipo e a pascalina, os dois com essa característica de permitir que um número fosse representado por um objeto físico e que passasse por alguma operação, representada pela manipulação dos objetos em questão.

Abelson et al. (1996) definem computadores tradicionais como máquinas de registradores, o que significa que eles executam instruções que manipulam os conteúdos de um conjunto de elementos de armazenamento, chamados de registradores. Os registradores são circuitos eletrônicos, e portanto, sua operação se dá através de sinais elétricos.

É comum que sistemas computacionais usem sinais em dois níveis distintos de intensidade para a representação de números em binário, sendo assim, toda informação em um computador é representada por *binary digits* (bits), que são sequencias de zeros e uns. As funcionalidades de um computador então, são implementadas por sequências de operações como:

somar o conteúdo de dois registradores e armazenar o resultado em outro, aplicar uma operação lógica no conteúdo de algum registrador, carregar um valor em um registrador, etc.

A base para quase todos os computadores digitais é o modelo de Von Neumann. Nele, o computador tem cinco partes principais: a Unidade Lógico-Aritmética (ULA), a unidade de controle, a memória, os sistemas de entrada e saída de dados e a interconexão entre esses componentes. Juntos, a ULA e a unidade de controle formavam o "cérebro" do computador, que é o componente principal, responsável por processar as operações, além de registrar seu resultado. Em computadores modernos eles são combinados em um único *chip*, a *Central Processing Unit* (CPU), ou ainda, o processador (Tanenbaum e Austin, 2012), que é onde ficam os registradores.

A unidade de armazenamento, geralmente denominada "memória", possui diversas entradas, ou blocos, que são localidades onde se pode armazenar dados de um tamanho fixo e cada uma pode ser acessada por um índice. Se a memória possuir n blocos, ela terá os índices 0 até $n - 1$, por exemplo. Esse índice é o endereço de memória daquele bloco. Os dados armazenados são sequências de bits de tamanho fixo, no caso de um bloco possuir k bits, ele poderá armazenar quaisquer uma das 2^k combinações de k bits. As interações da CPU com a memória recebem o nome de leituras e escritas, isto é, o carregamento de um valor da memória para um registrador ou o armazenamento de um valor em um registrador em um bloco, respectivamente (Tanenbaum e Austin, 2012).

O desenvolvimento das CPUs e da memória focaram em aspectos diferentes ao longo do tempo. Enquanto os processadores ficaram cada vez mais rápidos, a capacidade das memórias foi aumentada. Isso não significa que não existam memórias rápidas, mas como o foco não foi nesse aspecto, são memórias mais caras e por isso, componentes com capacidade menor que suas contrapartidas mais lentas e baratas. Assim, a diferença entre as velocidades das CPUs e das memórias é um problema, uma vez que não é usual um sistema que possua memórias rápidas com muita disponibilidade (Carvalho, 2002). Isso significa que os processadores podem ficar ociosos enquanto esperam uma leitura ou escrita terminar, piorando a performance do computador de maneira geral.

Entre as soluções para esse problema está o conceito de hierarquia de memória, ou seja, a utilização de um sistema de armazenamento com várias camadas de memórias com diferentes capacidades e velocidades. Geralmente, memórias menores e mais rápidas próximas ao processador, guardando valores utilizados mais frequentemente e memórias maiores e mais lentas para guardar valores que não estão sendo utilizado no momento ou que devem ser armazenados permanentemente para uso futuro mais distantes.

Dois elementos comuns em uma hierarquia de memória são a memória principal, responsável por armazenar temporariamente programas e valores que o processador esteja utilizando e a memória cache, uma memória mais rápida, geralmente localizada dentro do *chip* da CPU. A cache guarda os dados dos endereços mais recentemente usadas da memória, e as leituras e escritas do processador ocorrem nela primeiro. Caso uma boa porcentagem dos endereços utilizados durante a execução das operações estejam na cache, a latência efetiva de acesso à memória é diminuída drasticamente (Tanenbaum e Austin, 2012).

Os computadores são formados por diversos componentes, mas em sua essência continuam sendo uma máquina que permite a representação de informação de maneira física e sua transformação através da aplicação de operações. No entanto, eles têm uma característica importante que os diferem das máquinas de cálculo dadas como exemplo: eles são programáveis. Em vez de executarem apenas a mesma tarefa ou sequência fixa de tarefas, eles podem ser instruídos a executar diversas tarefas diferentes (Jan Friso Groote e Watkins, 2021).

Os elementos físicos dos computadores são denominados de *hardware* e os elementos que controlam o computador de maneira a executar diferentes tarefas recebem o nome de *software*. É comum que esses elementos sejam chamados de "programas" e seus criadores de "programadores". Existe *software* que geralmente não é escrito por usuários finais de um computador e que implementa funcionalidades básicas e essenciais para a utilização de um. A interface entre o *hardware* e o *software* de baixo nível é muito importante. Ela inclui tudo que o programador necessita para fazer um programa funcionar corretamente, como as operações disponíveis no computador, o formato como essas operações são codificadas, os registradores, os dispositivos de entrada e saída, etc.

Essa interface recebe o nome de *Instruction Set Architecture* (ISA), chamada também simplesmente de arquitetura. Ela é uma abstração que permite que ideias sejam expressadas sobre as funções de um computador, sem que seja necessário se prender ao *hardware* que as implementam. Dessa maneira, existe uma diferença entre uma arquitetura e uma implementação dela, que é um hardware que obedece às especificações da arquitetura (Patterson e Hennessy, 2014). A maneira como uma arquitetura é implementada em hardware é chamada de organização, ou microarquitetura de um computador. Ela envolve os componentes físicos e a maneira como eles são interconectados e controlados de maneira a seguir as especificações da arquitetura.

Os programas são construídos com as facilidades que a ISA disponibiliza. Eles são basicamente uma lista de instruções que configuram o computador para fazer alguma tarefa. As instruções consistem em um conjunto de bits e o computador executa instrução a instrução, podendo voltar e repetir a execução de alguma parte ou pular a execução de outra com base no resultado de alguma operação lógica.

O conceito de ter um conjunto de operações que são executadas em determinada sequência para resolver um problema antecede os computadores, mas se tornou extremamente importante para eles. Esses conjuntos de operações comumente recebem o nome de programas. A especificação de um programa, para que possa ser executado por um computador, se dá por uma linguagem de programação. Elas são livres de algumas das características que tornam as linguagens naturais inadequadas para esse fim, como a complexidade, a ambiguidade e a irregularidade (Evans, 2011a). Entre as linguagens de programação mais utilizadas hoje estão: Javascript, Python, C++, Java, C, etc. Essas linguagens permitem que programadores criem soluções para problemas complexos sem se preocupar com os detalhes da máquina, resultando em programas mais legíveis e portáveis. No entanto, elas não são entendidas pelos computadores diretamente. Um programa escrito em uma delas passa por uma série de transformações até chegar no formato que a máquina entende (Gaddis, 2021) diretamente, essa representação é denominada código de máquina. Mesmo que essa representação seja entendida diretamente pela máquina, escrever programas nela se mostrou tedioso e ineficiente com o passar do tempo.

Os programadores então criaram linguagens para representar o código de máquina de uma maneira mais legível por humanos e foram criados também programas para traduzir essas linguagens para código de máquina. Esses programas são chamados de montadores ou *assemblers* e essas linguagens são denominadas de linguagens *assembly* (Patterson e Hennessy, 2014). Para essa nova representação, as sequências de *bits* eram segmentadas e cada pedaço representava alguma informação.

Geralmente, o primeiro segmento se traduzia para um mnemônico que representava a instrução, um código que representa qual operação deve ser executada cujos *bits* são utilizados para gerar sinais que configuram o computador para executar tal operação. Isso ocorre através da unidade de controle do processador, que configura ele para distribuir os outros segmentos para os componentes e assim continuar o processamento da instrução. Esse mnemônico costuma ser chamado de *operation code* ou *opcode*. Os demais segmentos geralmente são os operandos, que

são traduzidos para outra base numérica, decimal ou hexadecimal geralmente. Eles podem ser números utilizados para acessar algum componente, que pode ser um registrador, um componente de armazenamento ou até um dispositivo de entrada e saída. Os operandos também podem ser valores codificados imediatamente na instrução para serem utilizados.

As linguagens *assembly* facilitaram muito o processo de programação, mas ainda requerem que os programadores tenham conhecimento detalhado sobre o funcionamento do *hardware*, além de que, programas para tarefas complexas resultam em códigos muito longos, bastante suscetíveis a erros e pouco legíveis.

As linguagens de programação mencionadas anteriormente surgem nesse contexto, e junto com elas foram criados programas que traduzem essas linguagens para *assembly*, chamados de compiladores. Essas linguagens são independentes do hardware, ou seja, não é necessário conhecer detalhes arquiteturais para criar programas com elas. Além disso, um mesmo programa pode ser compilado para diferentes tipos de computadores, basta que o compilador saiba como traduzir para o código de máquina alvo. Por causa disso, essas linguagens são comumente chamadas de linguagens de alto nível, devido a esse distanciamento do *hardware*, enquanto *assembly* e código de máquina geralmente são chamados de linguagens de baixo nível (Gaddis, 2021). Dessa maneira, um programa escrito em uma linguagem de alto nível é traduzido para *assembly* por um compilador, o resultado em *assembly* é então transformado por um montador em código de máquina.

2.2 A ARQUITETURA X86 E A MICROARQUITETURA INTEL

Existem diversas arquiteturas de computadores, e entre as filosofias que são aplicadas em seus *designs* duas delas podem ser destacadas: as filosofias *Complex Instruction Set Computer* (CISC) e *Reduced Instruction Set Computer* (RISC). As arquiteturas CISC têm sido dominantes na computação pessoal por décadas. Elas se caracterizam por um conjunto complexo e diversificado de instruções, permitindo a realização de tarefas complexas de maneira eficiente e facilitando a programação em baixo nível. Isso se dá porque uma instrução CISC é na verdade um conjunto de operações mais simples que o computador executa. Por exemplo, uma instrução de multiplicação que na verdade é uma série de instruções de soma. O conjunto é denominado microcódigo, e as operações geralmente são chamadas de micro-operações ou μops (Mohan, 2023). No entanto, essa complexidade pode levar a uma implementação de *hardware* mais complicada e à necessidade de decodificação de instruções complexas, dificultando o trabalho dos *assemblers* e dos compiladores. Entre algumas arquiteturas CISC notáveis, está a x86, implementada nos processadores da Intel e da *Advanced Micro Devices* (AMD).

Já a filosofia RISC prioriza um conjunto reduzido de instruções simples e eficientes. As operações são geralmente realizadas apenas com dados armazenados em registradores, com instruções para carregar um registrador ou armazenar o valor de um registrador em um sistema de memória. Outra característica importante é que arquiteturas RISC geralmente possuem formatos de instruções simples, com um tamanho fixo e cujo local dos segmentos, especialmente do *opcode*, também é fixo (William Stallings e Jesshope, 2016). Entre algumas das arquiteturas RISC mais notáveis estão o *Microprocessor without Interlocked Pipelined Stages* (MIPS), o RISC-V e o *Advanced RISC Machine* (ARM), implementadas em microcontroladores, celulares e outros sistemas embarcados.

Independentemente da filosofia adotada, a busca por desempenho é presente. A técnica de *pipelining* (William Stallings e Jesshope, 2016) usa uma lógica semelhante à uma linha de montagem em uma fábrica. Com a organização da CPU dividindo os estágios de uma instrução

em etapas, mais de uma pode ser processada simultaneamente. Para esse trabalho, o *pipeline* das máquinas Intel será considerado, por ser a microarquitetura-alvo do simulador OrCS.

As microarquiteturas da Intel, são uma implementação de *hardware* da arquitetura x86. Seu pipeline consiste em um *Frontend*, um *Backend* e um subsistema de memória. A Figura 2.2 mostra um diagrama da microarquitetura *Skylake Server* da *Intel*. O *Frontend* é o conjunto de etapas responsável por buscar as instruções da memória e decodificá-las em uma série de micro-operações, entregando-as para as unidades de execução. O *Backend*, também chamado de *Execution Engine*, é responsável pela execução da instrução e pelo registro do resultado atualizando o estado arquitetural (Abel e Reineke, 2019).

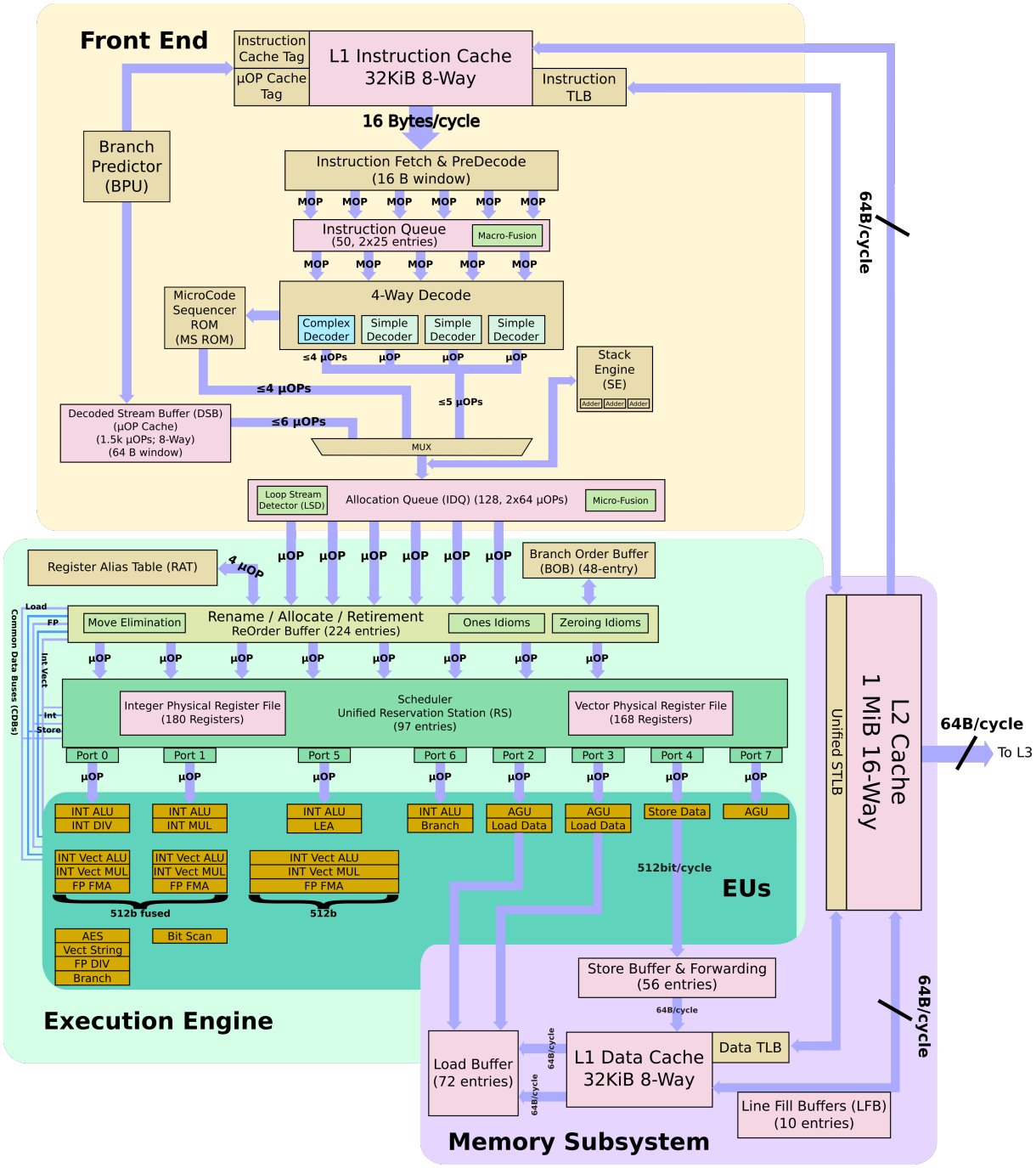


Figura 2.2: Diagrama de blocos do Intel Skylake Server (Wikichip, 2017b).

As etapas do caminho de dados executadas pelo *Frontend* são o ***Fetch*** e o ***Decode***. Na primeira etapa, as instruções, que nesse ponto ainda são instruções x86 de tamanho variável, são pré-decodificadas, isso consiste em determinar as bordas das instruções no bloco de dados buscado da cache de instruções. Na etapa de *Decode* as instruções pré-decodificadas são decodificadas para μops de tamanho fixo, que são enviadas diretamente para um componente que serve como a interface entre o *Frontend* e o *Backend* (Wikichip, 2017a).

As micro-operações geradas durante a etapa de *Decode* são passadas para uma estrutura chamada *Reorder Buffer*, do *Backend*. Ela as mantém na ordem em que foram decodificadas e é responsável por alocar recursos para a execução, que é a etapa de ***Rename***, que elimina falsas dependências de registradores entre as micro-operações. Em algumas microarquiteturas, algumas μops como *No Operation* (NOP) ou instruções que movem informações de um registrador para outro são executadas nessa etapa já, antes mesmo de serem registradas no *Reorder Buffer* (Wikichip, 2017a).

As micro-operações que sobram são enviadas para o *Scheduler*, também chamado de ***Unified Reservation Station***. Essa estrutura guarda as micro-operações enquanto esperam que seus operandos estejam prontos e suas dependências resolvidas para sua execução, ou que estejam esperando que a unidade de execução que ela precisa esteja livre. Quando a μop estiver pronta, ocorre a etapa de ***Dispatch***, onde as micro-operações mais velhas que estão prontas são despachadas para execução. Para aumentar a eficiência do processador as micro-operações não são executadas necessariamente em ordem. Assim que suas dependências estejam resolvidas elas podem ser escalonadas para serem executadas, dependendo da disponibilidade de estruturas chamadas de *Execution Ports*, que são associadas à algum componente logico-aritmético que executa a micro-operação (Wikichip, 2017a,b).

Por causa da execução fora de ordem, as micro-operações continuam no ***Reorder Buffer*** até que sejam executadas todas as micro-operações anteriores a ela. A micro-operação mais velha, depois de executada, passa pela etapa de ***Commit***, ou *Retirement*, onde a μop é removida do *Reorder Buffer* e tem os recursos previdamente alocados a ela liberados.

Como cada interação com o subsistema de memória é feita através de instruções, os acessos a memória também podem ser executados fora de ordem. Dois componentes importantes do subsistema de memória cuidam da ordenação dos acessos a memória, o ***Load Buffer*** e o ***Store Buffer*** (Wikichip, 2017a).

O simulador-alvo desse trabalho implementa microarquiteturas x86 da *Intel* e a estrutura interna do código dele reflete muito a estrutura mencionada acima.

2.3 O SIMULADOR ORCS (ORDINARY COMPUTER SIMULATOR)

O simulador OrCS, baseado no *Simulator of Non-Uniform Caches* (SiNUCA) (Alves et al., 2015), busca instruções do seu traço de simulação uma a uma e passa por etapas similares às da microarquitetura alvo, sem necessariamente executar as instruções e simulando apenas seu comportamento. Ele calcula e atualiza métricas com base no ciclo que está e os dados que possui sobre as instruções que está simulando. O OrCS mantém informações acerca das instruções em instâncias da classe **opcode_package_t** e informações das micro-operações são mantidas em objetos da classe **uop_package_t**.

A classe que representa as instruções é um pacote de informações de um *opcode*. Ela encapsula informações como seus operandos, seu tipo, os endereços de memória de seus acessos, informações de predição, etc. Além disso, oferece métodos para inicializar os objetos da classe e para depuração e análise de dados durante a execução do simulador. Seus atributos principais são descritos na Tabela 2.1.

Atributo	Descrição
opcode_assembly	*Opcode* em formato legível.
opcode_operation	Tipo de operação da instrução (ULA, memória, controle etc.).
instruction_id	Identificador para a instrução na microarquitetura.
opcode_address	Endereço da instrução na memória.
opcode_size	Tamanho da instrução em *bytes*.
read_regs	Registradores de leitura da instrução.
write_regs	Registradores de escrita da instrução.
base_reg	Registrador base associado à instrução.
index_reg	Registrador de índice associado à instrução.
reads_addr	Endereços de memória das leituras.
reads_size	Tamanhos das leituras em *bytes*.
num_reads	Número de leituras realizadas.
writes_addr	Endereços de memória das escritas.
writes_size	Tamanhos das escritas em *bytes*.
num_writes	Número de escritas realizadas.
branch_type	Tipo de *branch* da operação.
is_indirect	Indica se a operação é de natureza indireta.
is_predicated	Sinaliza se a operação é condicionada (predicada).
is_prefetch	Indica se a operação é uma pré-busca (*prefetch*).
opcode_number	Identificador único para a instrução.

Tabela 2.1: Atributos da classe opcode_package_t.

A classe **uop_package_t** contém as informações de uma micro-operação dentro da microarquitetura das máquinas *Intel*. Assim como *opcode_package_t*, *uop_package_t* possui métodos para inicialização e para depuração, além de outros para atualizar o atributo *status*, e para atualizar os atributos do objeto a partir de uma instância de *opcode_package_t*. A Tabela 2.2 apresenta uma descrição dos atributos principais presentes na classe.

Durante uma simulação, objetos dessas classes são instanciados, têm seus valores modificados e são inseridos e removidos de algumas estruturas de dados. A Figura 2.3 mostra um diagrama do simulador OrCS, nela podemos ver elementos que, no código, não necessariamente se traduzem em estruturas de dados e outros que se traduzem.

Todos essas estruturas têm um tamanho fixo desde o inicio da simulação, que é definido no arquivo de configuração do programa. O arquivo de configuração representa qual microarquitetura que deve ser simulada e é passado como parâmetro de entrada utilizando o argumento -c"ou --core"na linha de comando. Ele contém uma série de campos, que definem valores para as latências, para as larguras de alguns componentes e estágios e para o tamanho de alguns *buffers*. Outro argumento importante é o -t"ou --trace"que é o nome do traço de simulação que deve ser usado.

Após ler argumentos passados por linha de comando e determinar o número de processadores a serem utilizados na simulação, o OrCS aloca as estruturas essenciais para seu funcionamento, dentre elas o *cache_manager* e o *memory_controller*, além de configurar cada processador instanciado. Para isso, ele aloca o *trace_reader*, o *processor*, o *branch_predictor*, e outras estruturas relacionadas aos processadores.

Após essa inicialização, o simulador entra em um *loop* e, ao final, imprime diversas estatísticas, como o total de ciclos, *Instructions per Cycle* (IPC) global, tempo decorrido, entre outros. Durante esse *loop* principal, o *clock* do *memory_controller* é executado e logo depois

Atributo	Descrição
opcode_assembly	*Opcode* em formato legível.
opcode_operation	Tipo de operação da instrução.
opcode_address	Endereço da instrução na memória.
opcode_size	Tamanho da instrução em bytes.
read_regs	Registradores de leitura da instrução.
write_regs	Registradores de escrita da instrução.
uop_operation:	Tipo de operação do microcódigo.
memory_address	Endereços de memória associados à operação.
memory_size	Tamanhos das operações de memória.
num_mem_operations	Número de operações de memória.
read_address	Endereço de memória da leitura do 1o operando.
read2_address	Endereço de memória da leitura do 2o operando.
write_address	Endereço de memória da escrita do resultado.
latency	Latência em ciclos da instrução.
functional_unit	Unidade funcional responsável pela execução da μop.
opcode_number	Identificador da instrução que gerou a μop.
uop_number	Identificador único para a μop.
born_cycle	Ciclo em que a μop foi criada.
readyAt	Ciclo em que instrução está pronta.
status	Estado da operação.

Tabela 2.2: Alguns atributos da classe uop_package_t.

o *clock* do *processor* de cada um dos processadores da máquina simulada também é executado. Após isso o ciclo atual do sistema é incrementado e volta-se ao inicio do *loop*. O *clock* do controlador de memória, atualiza métricas acerca do acesso a memória principal e não faz modificações nos *buffers* de interesse.

Os estágios do caminho de dados são implementados pelo *processor*. Para simular a característica do *pipeline*, onde mais de uma instrução passa pelo fluxo ao mesmo tempo, o *clock* do *processor* executa os estágios na ordem inversa. Assim, as instruções antigas em estágios posteriores do pipeline vão sendo atualizadas e deixam seus estágios prontos para receber uma nova instrução. Antes de qualquer estágio, o *clock* do *cacheManager* é executado. Nele são tratadas as requisições para a cache, e uma vez que uma é finalizada, o *status* dos pacotes que dependiam dessa requisição são atualizados. No restante do *clock* do *processor*, uma série de verificações e operações baseadas nos *buffers* e estágios do *pipeline* do processador são executadas, na seguinte ordem: as etapas de *Commit*, *Execute* e *Dispatch* são realizadas se houver instruções pendentes no *Reorder Buffer*; o estágio *Rename* é realizado se houver instruções no *Decode Buffer*; o estágio *Decode* é acionado se houverem instruções no *Fetch Buffer* e o estágio *Fetch* acontece se o traço de simulação ainda não terminou. Cada um dos estágios possui métodos relacionados a elas, e é nesses métodos que as estruturas observadas sofrem a maioria das suas modificações.

A Figura 2.4 mostra uma versão simplificada do diagrama do OrCS. Os estágios do pipeline foram numerados e o tipo de pacote que eles manipulam, bem como os *buffers* relacionados com cada um, foram destacados. As unidades funcionais do estágio *Execute* foram substituídas pelo *buffer "Unified Functional Units"* (UFU). Os métodos que implementam os estágios do *pipeline* foram o foco desse trabalho.

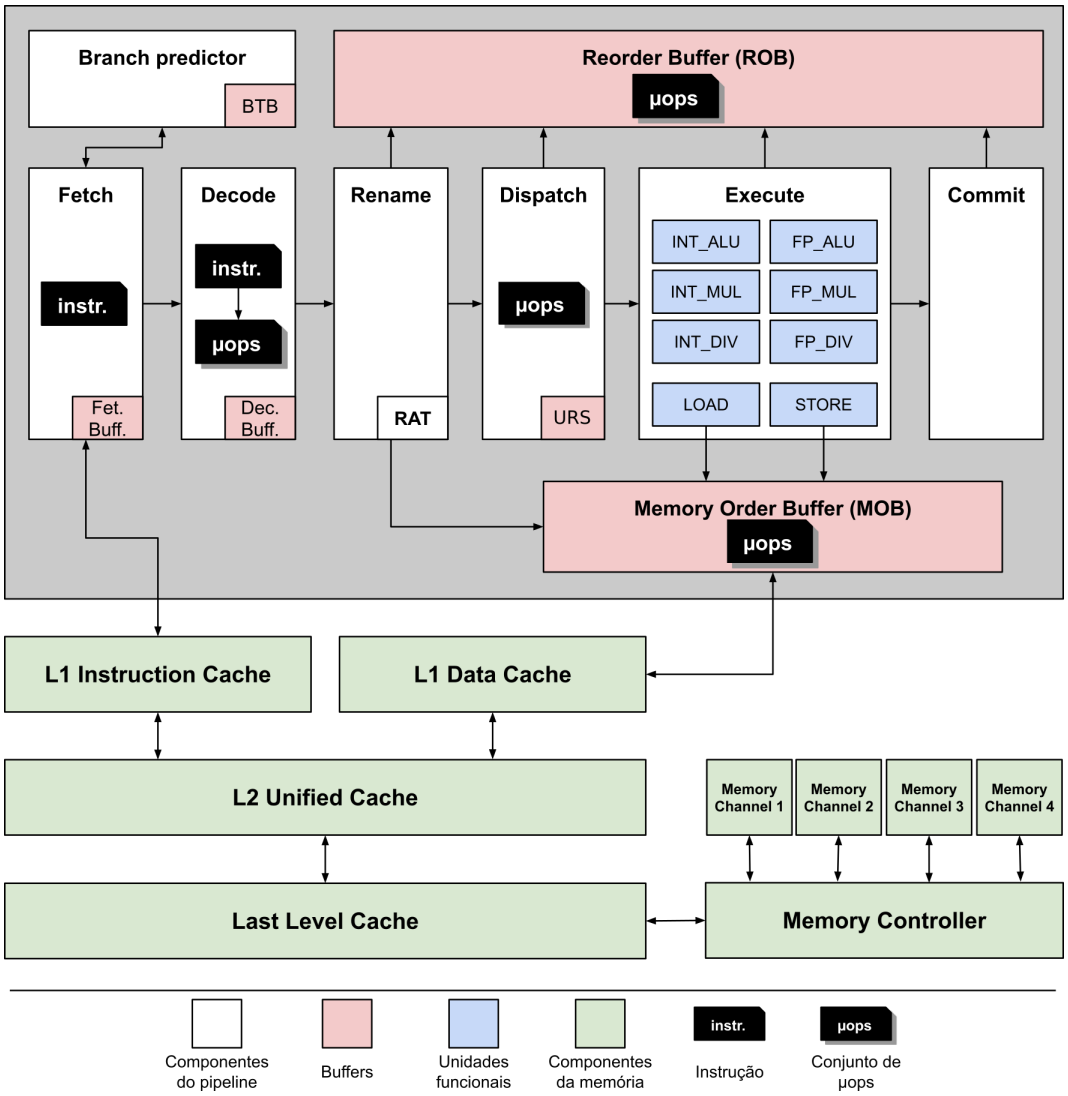


Figura 2.3: Diagrama do OrCS, adaptado de (Tissei, 2021).

1. O *Fetch*, implementado pelo método **processor::fetch()**, faz modificações em pacotes do tipo *opcode_package_t*. Ele executa operações de busca de instruções, as repetindo de acordo com a *flag* **FETCH_WIDTH**, que indica a largura do estágio de *Fetch*. Após algumas verificações serem concluídas com sucesso e nenhuma situação de espera ou de *branch* incorreto ser identificada, a busca de uma nova instrução é iniciada. O simulador lê o traço, o que resulta na criação de um objeto *opcode_package_t*. Após algumas verificações com relação a possíveis *branches* errados, o pacote é inserido no *"Fetch Buffer"* e uma requisição de instrução para a memória é criada. A primeira das requisições causa a atualização do *status* do pacote associado para **PACKAGE_STATE_DRAM_FETCH**. Quando a busca for completa, o *status* desse pacote e de todos os subsequentes, que são trazidos para a *cache* é atualizado para **PACKAGE_STATE_READY**.

2. O método **processor::decode()**, quebra um *opcode* com *status* **PACKAGE_STATE_READY** no *Fetch Buffer* em diversas instâncias do tipo *uop_package_t*. Esse processo se repete um número de vezes igual a *flag* **DECODE_WIDTH**. O pacote a ser decodificado é o que está na posição mais antiga do *Fetch Buffer*.

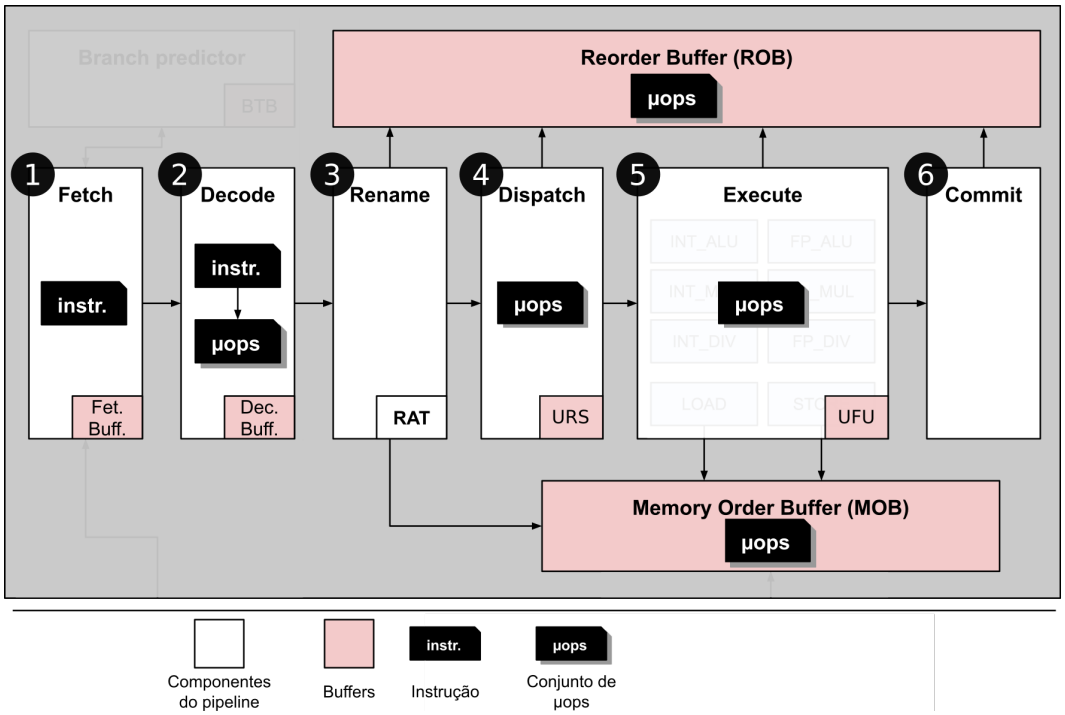


Figura 2.4: Diagrama do OrCS Simplificado, adaptado de (Tissei, 2021).

As micro-operações que serão geradas são determinadas pelo tipo de *opcode_package_t*. Se a instrução envolve leituras ou escritas na memória, as μops correspondentes são criadas e adicionadas ao *Decode Buffer*. Em casos onde há mais de duas leituras ou mais de uma escrita na memória, μops para as operações *Gather* e *Scatter* são geradas. Se a instrução não é de leitura ou escrita, e também não é de *branch*, são geradas μops para operações da ULA. Isso é feito com base no conjunto de instruções disponíveis e nos tipos de operações lógico-aritméticas existentes. Para instruções de *branch*, é gerado uma μops correspondente. Conforme essas verificações acontecem, as micro-operações são criadas com *status* **STATUS_PACKAGE_WAIT** e adicionadas ao *Decode Buffer*, se existir espaço suficiente nele. Após a geração, o *opcode package* correspondente é removido do *Fetch Buffer*.

3. No método **processor::rename()**, o simulador percorre um *loop* controlado pela constante **RENAME_WIDTH**, verificando e executando várias operações associadas a *buffers* que mantém a ordem das instruções, uma vez que a execução acontece fora de ordem. Ele faz isso utilizando o ROB, a URS, e os *buffers* MOB *Read* e MOB *Write*.

   Dentro do *loop*, cada iteração inicia verificando se existe um pacote de μop na posição mais antiga do *Decode Buffer* que esteja pronto para processamento, considerando o ciclo atual do mecanismo do processador. Em seguida, é verificado se a URS tem capacidade para mais operações. Além disso, também são verificadas as capacidades do ROB e dos MOBs. Se alguma dessas condições não for atendida, o *loop* é interrompido.

   O código prossegue e a μop é encapsulada em uma instância de uma linha do ROB, que contém informações extras acerca da execução da micro-operação e contém relações com possíveis entradas nos MOBs. Essa linha é inserida no ROB, e a μop removida do *Decode Buffer*. O *satus* do pacote é então atualizado para **PACKAGE_STATUIS_WAIT** enquanto ele passa pelas latências do estágio de *Rename* e do *Dispatch*. O pacote também é inserido na URS, que funciona como um *Scheduler* na

microarquitetura. Finalmente, se a μop for alguma operação de memória ela é inserida em algum dos dois MOBs.

4. No método **processor_t::dispatch()**, o código realiza a distribuição de μops prontas para serem executadas para as unidades funcionais do processador. Ele itera sobre os pacotes de micro-operações na URS, verifica se a quantidade total de pacotes despachados atingiu o total de **DISPATCH_WIDTH** e avalia se uma μop está pronta para ser despachada com base no ciclo global atual e na ausência de dependências de registradores. Cada pacote a ser despachado é associado a uma unidade funcional, caso existir disponibilidade de *slots* nela para a execução. As operações são então adicionadas na UFU e são removidas da URS. Ao longo do código, são feitas verificações para garantir que a instrução está no estado apropriado para o despacho e que seu tipo de operação é válido para execução. O método encerra depois que um número igual a **DISPATCH_WIDTH** de instruções é despachado.

5. O método **processor_t::execute()** percorre a UFU processando pacotes prontos, de acordo com a *flag* **EXECUTE_WIDTH**. No início do método, as micro-operações no MOB Read que estão prontas são atualizadas, e depois o *loop* de execução começa. Dependendo do tipo de operação da instrução, diferentes ações são tomadas. Para *branches*, operações aritméticas e operações de ponto-flutuante, o *status* da μop é atualizado para **PACKAGE_STATE_READY**, simbolizando que está pronta para o estágio de *Commit*, e as dependências dos registradores relacionados a ela são resolvidas. Por último, a micro-operação é removia da UFU.

   Para operações de leitura e escrita de memória, as entradas correspondentes nos MOBs são marcadas como executadas e o *status* delas atualizado para **PACKAGE_STATUS_WAIT**. Após isso, as operações são removidas da UFU. Ao final, são criadas requisições de memória com base em algumas *flags*, para as operações mais antigas de leitura e escrita nos MOBs. Essas requisições são direcionadas a *cache* e no caso dos *writes*, caso a busca na *cache* tiver sucesso, as micro-operações correspondentes já são removidas do MOB *Write*.

6. O método **processor_t::commit()** é responsável pela a etapa de *Commit* do caminho de dados, também chamado de *Retirement* nas microarquiteturas x86, onde as micro-operações que estão prontas para serem concluídas têm seus resultados registrados. O código itera sobre as instruções no ROB de acordo com a *flag* **COMMIT_WIDTH** para identificar aquelas prontas para essa etapa. A posição mais antiga do ROB é inspecionada, caso essa operação não estiver pronta, o *loop* é interrompido, garantindo a conclusão das instruções na ordem apropriada. Para cada tipo de operação como operações na ULA, operações complexas como multiplicação e divisão ou operações de ponto-flutuante, as estatísticas correspondentes são atualizadas. Se a micro-operação tiver sido enviada para a memória e for de leitura, o método remove as operações do MOB. Ao final do *loop* o pacote de μop é removido do ROB.

Existem algumas atualizações dos *buffers* que ocorrem fora desses métodos do *processor*, mas a maioria são as descritas nesse passo. Com o escopo da visualização definido, a próxima seção fala sobre o processo de visualização e suas vantagens.

2.4 VISUALIZAÇÃO

Como já mencionado anteriormente, a imagem tem uma vantagem sobre a informação textual. Mazza (2009) menciona como é possível representar uma grande quantidade de informações usando elementos gráficos como: forma, tamanho, cor, movimento ou textura. Esses elementos são imediatamente absorvidos e processados pela visão, antes mesmo que os processos cognitivos da mente aconteçam. Isso gera uma certa confiança de que a visualização pode impactar positivamente o uso do *OrCS*.

Quanto ao desenvolvimento de ferramentas de visualização, mesmo que difiram bastante entre si, elas tendem a seguir um processo de geração que pode ser enunciado. A Figura 2.5 mostra as etapas desse processo.

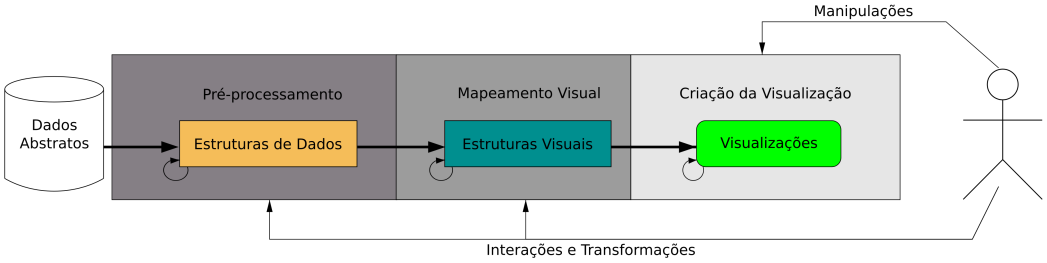


Figura 2.5: Processo de geração de ferramentas de visualização, adaptado de (Mazza, 2009).

O ponto de partida são dados abstratos, as informações que se quer visualizar, no caso desse trabalho o estado dos *buffers* e pacotes do OrCS. Esses dados iniciais raramente são encontrados em um estado próprio para a construção de uma visualização, então devem passar por algumas etapas até chegarem no usuário (dos Santos, 2013).

- **Pré-processamento**: O preprocessamento envolve a conversão de dados crus para um formato adequado para visualização. Isso inclui operações como filtragem para eliminar dados desnecessários, cálculo de novos dados a partir dos dados crus, e adição de atributos para organizar os dados logicamente. A estrutura dos dados pode variar, mas a fase de pré-processamento garante que os dados estejam prontos para serem utilizados pela ferraments de visualização.

- **Mapeamento Visual**: O mapeamento visual é o processo de definir estruturas visuais para representar dados. Isso inclui a definição do substrato espacial (dimensões físicas como: eixos x e y), elementos gráficos (pontos, linhas, superfícies, volumes), e propriedades gráficas (tamanho, orientação, cor, textura, formato). A percepção das propriedades gráficas pode variar culturalmente, especialmente no caso das cores, e é importante considerar condições como o daltonismo ao desenvolver visualizações.

- **Criação da visualização**: É o resultado final, gerando representações visuais em um computador. Um desafio comum é a grande quantidade de dados a ser representada em um espaço limitado, o que pode ser mitigado com técnicas como zoom, deslocamento e rolagem. As visões devem ser projetadas para maximizar a clareza e a utilidade da representação visual, mesmo quando a área de exibição é restrita.

Neste trabalho, a fase de pré-processamento se deu pela definição de um tipo de arquivo de entrada que sintetizasse as modificações que os pacotes e *buffers* do OrCS. O mapeamento visual se deu na definição da representação apra os *buffers* e pacotes e a criação da visualização na implementação da ferramenta. A próxima seção apresenta o tipo de arquivo utilizado na etapa de pre processamento.

2.5 O FORMATO PAJÉ

Ao iniciar o desenvolvimento da ferramenta, foi considerado não ser apropriado utilizar o traço de simulação do OrCS como arquivo de entrada para a ferramenta, uma vez que o objetivo era apenas a visualização do conteúdo dos *buffers* e não o tratamento de cada instrução. Assim, foi decidido pela adoção de um formato de arquivo específico para a descrição da visualização, o formato Pajé (Schnorr et al., 2015). Ele consiste em um arquivo de traço em texto projetado para descrever o comportamento de programas, organizado em duas partes que contêm três seções.

A primeira parte do arquivo refere-se à definição de eventos, composta por duas seções: a definição dos eventos e a declaração da hierarquia de tipos. Já a segunda parte contém os eventos registrados pelo traço e possui apenas uma seção, que lista os eventos que ocorreram. Tanto a seção de declaração da hierarquia quanto a seção dos eventos registrados possuem formatações similares, pois a declaração da hierarquia é composta por eventos.

Os eventos são objetos constituídos de campos identificados por um nome específico. Eles podem possuir vários campos, cada qual com um nome e um tipo. A descrição de um evento segue um formato específico, onde todas as linhas iniciam com o caractere de porcentagem. A definição de evento é introduzida por uma linha contendo a string **%EventDef**, seguida por um nome e um número inteiro arbitrário e único, utilizado para identificar o tipo do evento na segunda parte do traço. Cada campo é apresentado em uma linha com o nome do campo seguido pelo seu tipo. A definição do evento é finalizada com uma linha contendo a *string* **%EndEventDef**

A Listagem 2.1 mostra a definição de um evento chamado "Clock", que possui identificador 0 e um campo, "cycle", um inteiro.

Código 2.1: Definição de evento "Clock"com o campo "cycle"

```
1 %EventDef Clock 0
2 % cycle int
3 %EndEventDef
```

Os eventos são empregados na construção de visualizações, de acordo com a hierarquia de tipos. Eles são listados um por linha. O primeiro elemento na linha é o número arbitrário utilizado para identificar o evento, seguido pelos valores de cada campo do evento para aquela instância, separados por espaço. A hierarquia dos tipos é definida por instâncias de eventos padrões que definem quais tipos foram criados e como se relacionam.

A Listagem 2.2 mostra quatro eventos "Clock", identificados pelo numero 0 no início de suas linhas, o segundo número é o valor do campo "cycle"para cada um dos eventos registrados.

Código 2.2: Quatro instâncias de eventos "Clock"

```
1 0 1
2 0 2
3 0 3
4 0 4
5 ...
```

Esse formato de arquivo foi considerado adequado devido a sua economia e flexibilidade: cada evento é descrito por um número e apenas dados úteis precisam ser registrados. Com o tipo de arquivo de entrada definido, a próxima seção examina trabalhos correlatos, explorando que tipos de funcionalidades outras ferramentas de visualização implementam.

3 TRABALHOS CORRELATOS

Esse trabalho apresenta a ferramenta VOrCS. antes de sua codificação, foi realizada a busca de ferramentas já existentes com finalidades similares, com o objetivo de identificar as funcionalidades mais comuns desse tipo de programa e contrapô-las com as que são propostas pela ferramenta deste trabalho. Esse capítulo apresenta os trabalhos correlatos encontrados e suas funcionalidades em comum.

As bases *IEEEXplore* e *Google Scholar* foram utilizadas para a pesquisa. Os seguintes termos de busca foram utilizadas: "computer architecture AND visualizat*", "computer architecture AND visualizat* AND (model OR system OR tool)"e "visual computer architecture simulator".

Foram selecionados artigos que possuíam imagens da ferramenta de visualização e descrições das funcionalidades implementadas. Ferramentas que continham alguma representação gráfica da arquitetura e que fossem interativas receberam prioridade. *Overviews* e comparações de ferramentas existentes foram usadas como fontes adicionais para encontrar mais artigos que falavam sobre outras ferramentas.

3.1 SUMÁRIO DOS TRABALHOS ENCONTRADOS

Para ter um panorama geral sobre as ferramentas de visualização e simulação gráfica dos artigos selecionados, a Tabela 3.1 apresenta algumas categorias, de acordo com características encontradas nos trabalhos.

- A categoria **Arquitetura** se refere à qual é a arquitetura do processador representada pelo programa.
- A categoria **Arquivo de Entrada** indica qual o tipo de arquivo utilizado pela ferramenta. Dentre os tipos encontrados existem, arquivos com código *assembly* da arquitetura (Asm), arquivos com códigos da linguagem C (C) e arquivos com traços de simulação (Traço).
- A categoria **Implementação** exibe qual é a linguagem de programação utilizada para o desenvolvimento da ferramenta.
- As categorias **Execução Contínua** e **Execução Step** se referem à maneira como a simulação/representação acontece. No primeiro modo a execução se dá sem interrupções e essa funcionalidade está normalmente acompanhada de maneiras de controlar a frequência de execução dos ciclos ou de atualização da representação gráfica. No segundo modo a execução se dá por saltos, passo a passo, seja desencadeando o avanço de um número fixo de ciclos ou avançando até determinado ponto da simulação e parando.
- A categoria **Reset** indica se a ferramenta permite que a simulação possa ser reiniciada sem que uma nova sessão da ferramenta se inicie.
- A categoria **Customizável** indica que o programa permite a customização da representação gráfica, seja permitindo a visualização de diversos modelos da arquitetura, permitindo a visualização em diferentes níveis de abstração ou permitindo a modificação da disposição dos componentes visualizados.

- A categoria **Visualização da memória** indica que a ferramenta representa/simula a memória principal ou cache.
- A categoria **Métricas** indica que o programa exibe métricas acerca da execução simulada, como instruções por ciclo (IPC), uso de memória, latências máximas, entre outras.

Na seção seguinte, são apresentadas as ferramentas e suas funcionalidades principais.

	Arquitetura	Arquivo de Entrada	Implementação	Execução Contínua	Execução em Salto	Reiniciar	Customizável	Memória	Métricas
NeanderSIM	Neander	Asm	Java	•				•	
DEVSJava MIPS32	MIPS	Asm	Java	•	•	•	•	•	•
VisuSim	Von-Neumann	Asm	Java	•	•			•	
CompSim	Cariri	Asm	Python	•	•	•		•	•
Ripes	RISC-V	Asm / C	C++	•	•	•	•	•	•
M2S-Visual	x86	Traço	C		•		•	•	•
DIMIPSS	MIPS	Asm	Java		•			•	
MARS	MIPS	Asm	Java	•	•	•		•	
RARS	RISC-V	Asm	Java	•	•	•		•	
QtSpim	MIPS	Asm	C++	•	•	•	•		
VOrCS	Qualquer	Traço	Python	•	•	•	•	•	•

Tabela 3.1: Classificação dos trabalhos correlatos pesquisados.

Os trabalhos analisados abrangem uma variedade de arquiteturas, incluindo arquiteturas criadas exclusivamente para o ensino e outras de máquinas comerciais.

A maioria dos simuladores utiliza arquivos de entrada em linguagem *assembly* e a maioria incluiu um simulador junto à visualização. A ferramenta Ripes também suporta arquivos em C como entrada. Dos simuladores encontrados, apenas o M2S-Visual utiliza traços. A implementação dos simuladores varia entre Java, Python e C++. Java é a escolha mais comum, provavelmente devido à sua portabilidade, com ferramentas sendo distribuídas inclusive como aplicações para *web*. Python é uma boa escolha para prototipagem rápida e simplicidade de desenvolvimento. C++ é a linguagem que apresenta melhor desempenho. A escolha de linguagem é influenciada principalmente pelas necessidades de desempenho e portabilidade da ferramenta.

Sobre os modos de execução, é comum que as ferramentas tenham a capacidade de executar o simulador de forma contínua ou passo a passo. O segundo modo se mostra essencial para a depuração e entendimento detalhado do comportamento do processador. Algumas ferramentas permitem que a simulação seja pausada e os valores dos registradores do processador ou de endereços de memória, modificados. A capacidade de reiniciar a simulação também é importante para testes repetitivos e análise de diferentes cenários de execução. Essa funcionalidade está presente em diversos dos simuladores.

Poucos trabalhos permitem a customização da interface, e a maioria permite apenas a variação de qual microarquitetura está sendo visualizada. A visualização de memória muita das

vezes envolve que a interface gráfica ou janela de visualização da ferramenta seja modificada. Na maioria de ferramentas que possuem visualização da memória, ela é mostrada como uma tabela, com endereços e seus valores.

A coleta de métricas é vital para a análise de desempenho e otimização. Algumas ferramentas incluem essa funcionalidade, facilitando estudos detalhados sobre eficiência e comportamento de diferentes arquiteturas e implementações. Geralmente as métricas mostradas variam quando a memória ou o processador está sendo visualizado. Algumas ferramentas contém métricas como IPC e Cache Miss Ratio. Como o OrCS já traz essas métricas, e o objetivo da ferramenta desenvolvida não é substituir o simulador, ela não exibe essas métricas.

As seções a seguir apresentam um resumo das funcionalidades e da representação gráfica dos trabalhos selecionados.

3.2 NEANDERSIM

Ullmann et al. (2014) apresentam um simulador gráfico chamado **NeanderSIM**, projetado para auxiliar o ensino de arquiteturas de computadores. Ele aceita como entrada código *assembly* para o computador teórico NEANDER, definido por Weber (2012). A ferramenta é implementada na linguagem Java.

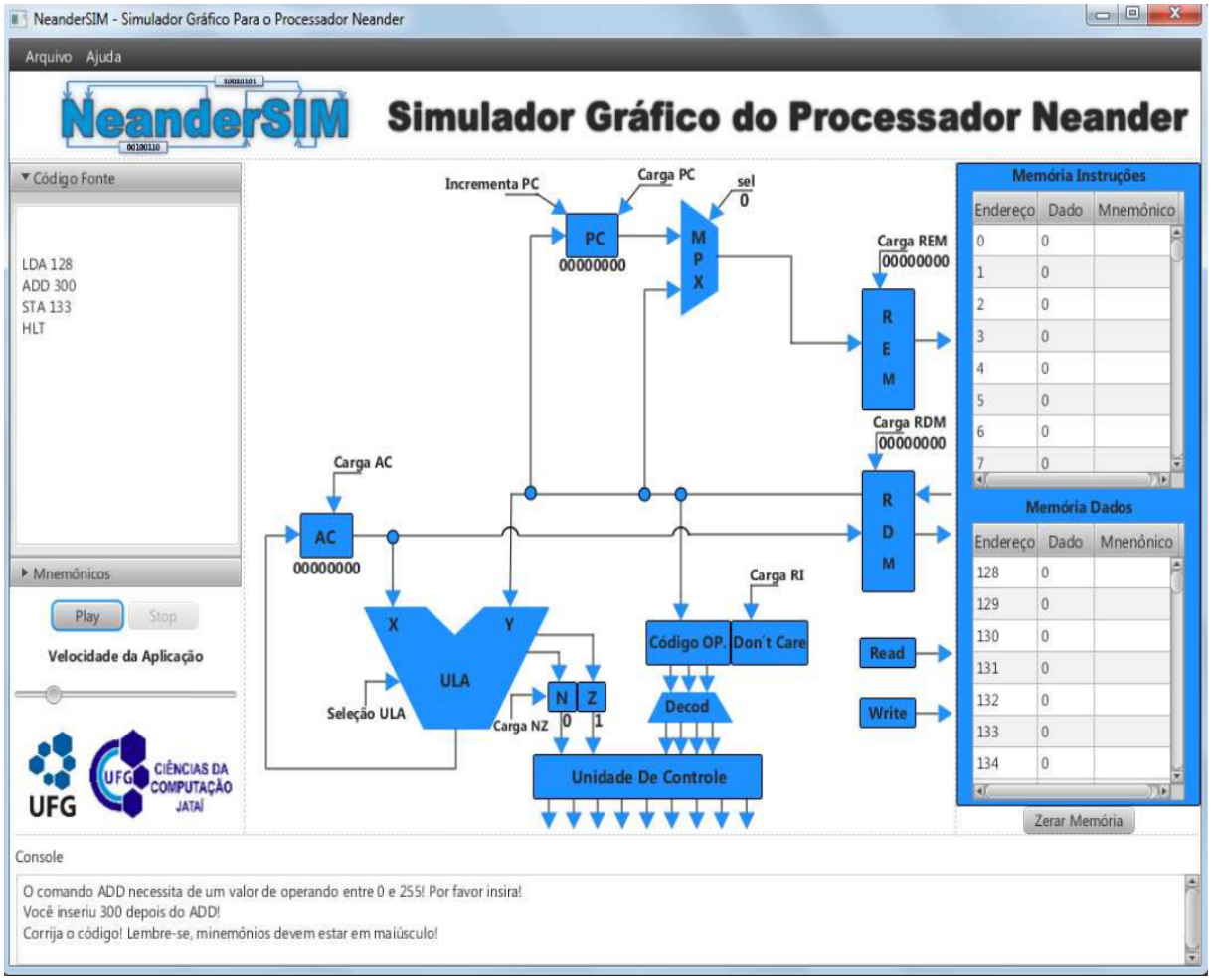


Figura 3.1: A interface do NeanderSIM (Ullmann et al., 2014).

A Figura 3.1 mostra a interface do NeanderSIM. Ela conta com um componente para inserção de dados de entrada, componentes para controle da simulação, uma representação

gráfica da arquitetura que ele simula, um componente para a visualização do conteúdo da memória de instruções e de dados e um console que imprime mensagens de erro e informações acerca da execução das instruções.

São implementados controles para iniciar, parar e modificar a velocidade de animação da simulação. Cada instrução da arquitetura simulada possui uma função Java correspondente, que quando é chamada, gera animações na representação gráfica, mostrando como componentes, barramentos de sinais e valores na memória são afetados.

3.3 DEVSJAVA MIPS32

Sarjoughian et al. (2008) apresentam o simulador **DEVSJava MIPS32**, também com aplicação educacional. O *Discrete Event System Specification* (DEVS) em DEVSJava é um formalismo para modelagem de sistemas complexos (Zeigler et al., 2018). O programa implementa vários sabores da arquitetura MIPS32, como descritos por Patterson e Hennessy (2005), e lê arquivos *assembly* à partir de um sistema que permite configurar experimentos, pelo qual é definido não só o arquivo de entrada, mas também parâmetros para a simulação.

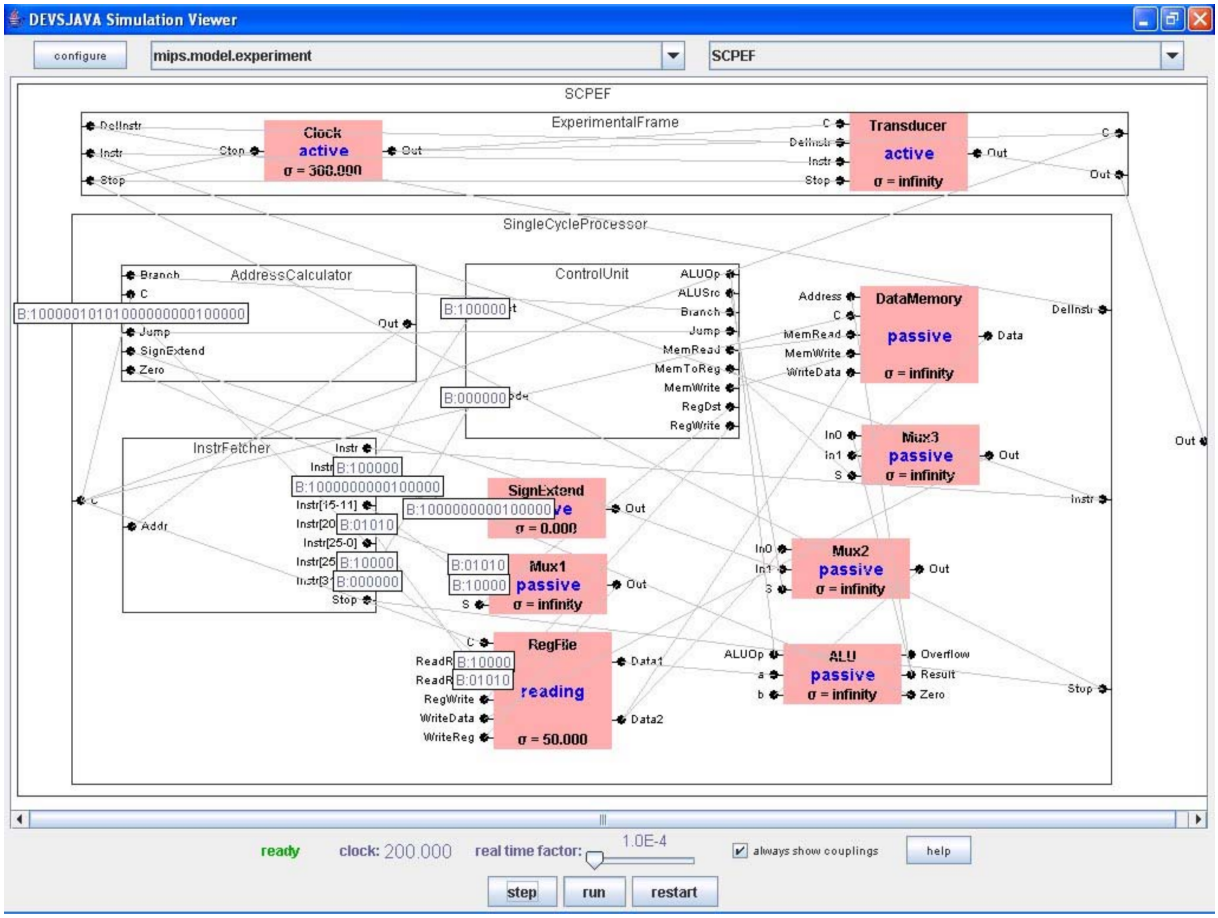


Figura 3.2: A interface do DEVSJava MIPS32 (Sarjoughian et al., 2008).

A Figura 3.2 mostra a interface do programa, que conta com uma área para visualização e componentes para controle e configuração da simulação. A ferramenta permite visualizar componentes, suas entradas, saídas e conexões com outros componentes. Visualmente, um componente é representado por um retângulo, com portas de entrada à esquerda e portas de saída à direita. As ligações entre componentes são representadas por linhas e os valores binários

representam a saída dos modelos. O simulador faz distinção entre componentes acoplados e atômicos. Um componente acoplado é representado com a cor branca e permite que os componentes internos dele e suas ligações sejam observados. Os componentes atômicos são representados com a cor vermelha.

A ferramenta possui uma barra para controle da velocidade da animação, um botão para avançar um passo da simulação (que é definido por um parâmetro de configuração), um botão para executar indefinidamente e parar a simulação e outro para reiniciá-la.

3.4 VISUSIM

Imai et al. (2006) introduziram o simulador **VisuSim**, escrito em Java puro, com aplicação real no ensino de Arquitetura de Computadores. A Figura 3.3 mostra a interface do programa.

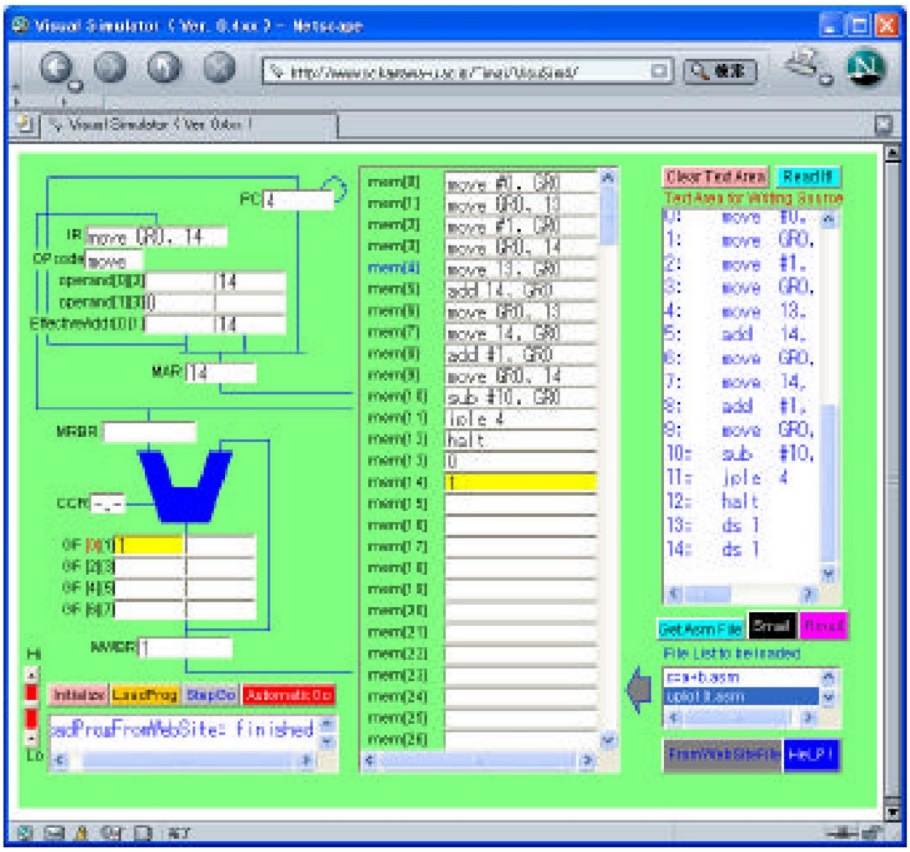


Figura 3.3: A interface do VisuSim (Imai et al., 2006).

Ele tem várias funcionalidades voltadas para o compartilhamento e a facilitação do seu uso por professores e alunos. A arquitetura implementada é um modelo de arquitetura de Von Neumann, com um conjunto de instruções próprias criado para o simulador.

A representação gráfica se dá por uma série de áreas de entradas de dados, preenchidas com valores numéricos ao lado de rótulos indicando o nome do sinal ou componente que representam. A visualização é feita assim para que a qualquer momento, o usuário possa parar a execução, modificar um valor qualquer e continuar a simulação. Essa exploração pode auxiliar na compreensão da estrutura interna da arquitetura.

Existe um conjunto de entradas de dados, dispostas como uma tabela, cujas linhas representam o conteúdo da memória. O programa *assembly* simulado é representado ali, junto

com os dados na memória. (Imai et al., 2007) posteriormente complementa o simulador com funcionalidades para comunicação entre usuários e compartilhamento de código.

3.5 COMPSIM

O simulador gráfico **CompSim**, apresentado por Esmeraldo e Lisboa (2017), implementa uma simulação de um processador conceitual denominado Cariri. O programa pode ser visto na Figura 3.4. Ele é formado por múltiplas janelas e a principal contém a representação do processador, uma área com os controles dos parâmetros da visualização, uma área de texto para edição de código *assembly*, uma tabela que mostra o estado das linhas da cache e outra que mostra o estado da memória principal, além de uma área com duas caixas de texto para representar dispositivos de entrada e de saída. A representação do processador se dá por um conjunto de áreas de texto com os conteúdos dos registradores e seus respectivos rótulos.

Os controles do simulador contém funcionalidades para definição da frequência do relógio e para definição do tempo atual e controles de fluxo da simulação, como iniciar, pausar, reiniciar e execução passo-a-passo.

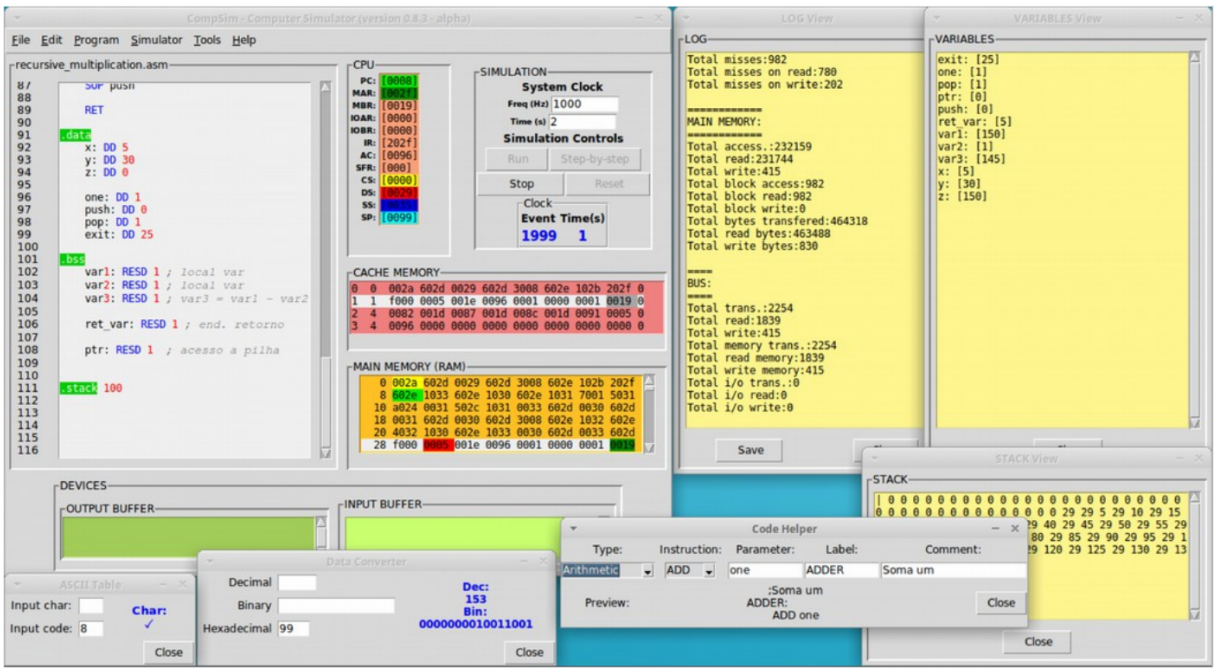


Figura 3.4: A interface do CompSIm (Esmeraldo e Lisboa, 2017).

Ainda existem outras janela do programa que funcionam como ferramentas com funcionalidade adicionais, exibição de informações de eventos e estatísticas de simulação, exibição das variáveis do programa *assembly* executado, conversão entre bases numéricas e diversos outros.

3.6 RIPES

Petersen (2021a) descreve o **Ripes**, que é definido como simulador visual de arquitetura de computadores, inicialmente construída para simular uma arquitetura RISC-V como definido por Waterman et al. (2014), e com capacidade de visualização de diversos modelos de processadores dessa arquitetura.

A ferramenta tem uma barra de ferramentas com funcionalidades para controle da visualização, onde é possível reiniciar a simulação, retroceder e avançar um ciclo de relógio,

além de avançar os ciclo de *clock* indefinidamente com uma determinada frequência, atualizando os elementos animados do programa ou sem atualizá-los. A seção principal do programa conta com as abas *Editor*, *Processor* e *Memory*. A Figura 3.5 mostra a interface do Ripes, com a aba *Processor* aberta.

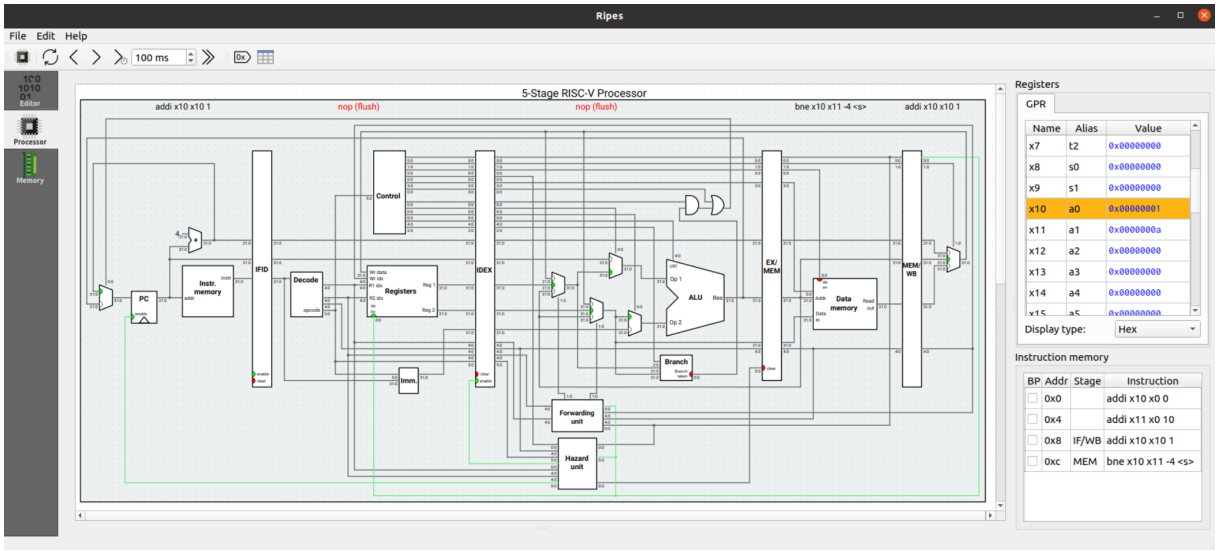


Figura 3.5: A interface do Ripes (Petersen, 2021a).

Na aba *Editor* existe um editor de código integrado e uma área que mostra o código executável correspondente. A aba *Processor* é a principal aba de visualização. Ela mostra o caminho de dados da microarquitetura, uma tabela que mostra os registradores e outra que mostra a memória de instruções. A aba *Memory* contém uma tabela com o conteúdo da memória principal e uma área com um simulador configurável de memória cache. Essa área também exibe métricas acerca da utilização e funcionamento da memória.

Os dados de entrada são providos utilizando o editor integrado. O simulador conta com um *assembler* interno e têm suporte a compilação, o que permite a inserção de código *assembly* ou código C. Devido a completude da visualização que o Ripes provê, ele não é pensado para ser um substituto de outros simuladores voltados para pesquisa, mas ele tem uma aplicação robusta no ensino.

A visualização nessa ferramenta é feita através do *framework Visual Simulation of Register Transfer Logic* (VSRTL) (Petersen, 2021b), usado para descrever e visualizar circuitos digitais com precisão de ciclo. O VSRTL é composto por uma linguagem de descrição de hardware implementada em C++ e uma biblioteca de interação e visualização de circuitos digitais.

3.7 M2S-VISUAL

O **M2S-Visual**, descrito por Ziabari et al. (2015), é uma adição ao *Multi2Sim* (M2S) definido por Ubal et al. (2012). Esse programa permite a visualização de traços de execução de programas paralelos, especificamente de programas que usam *OpenCL*. Ele tem suporte para a visualização de programas para CPU x86 e para *Graphics Processing Unit* (GPU) *Evergreen* e *Southern Islands* da AMD.

Pela grande quantidade de processadores em sistemas paralelos, os traços de uma aplicação curta podem estar na ordem de milhões de linhas e isso torna inviável ou no mínimo muito difícil a observação e compreensão deles. O M2S-Visual suporta a reprodução de traços

de execução de sistemas com CPU, GPUs ou sistemas heterogêneos. A Figura 3.6 mostra a janela principal do M2S-Visual.

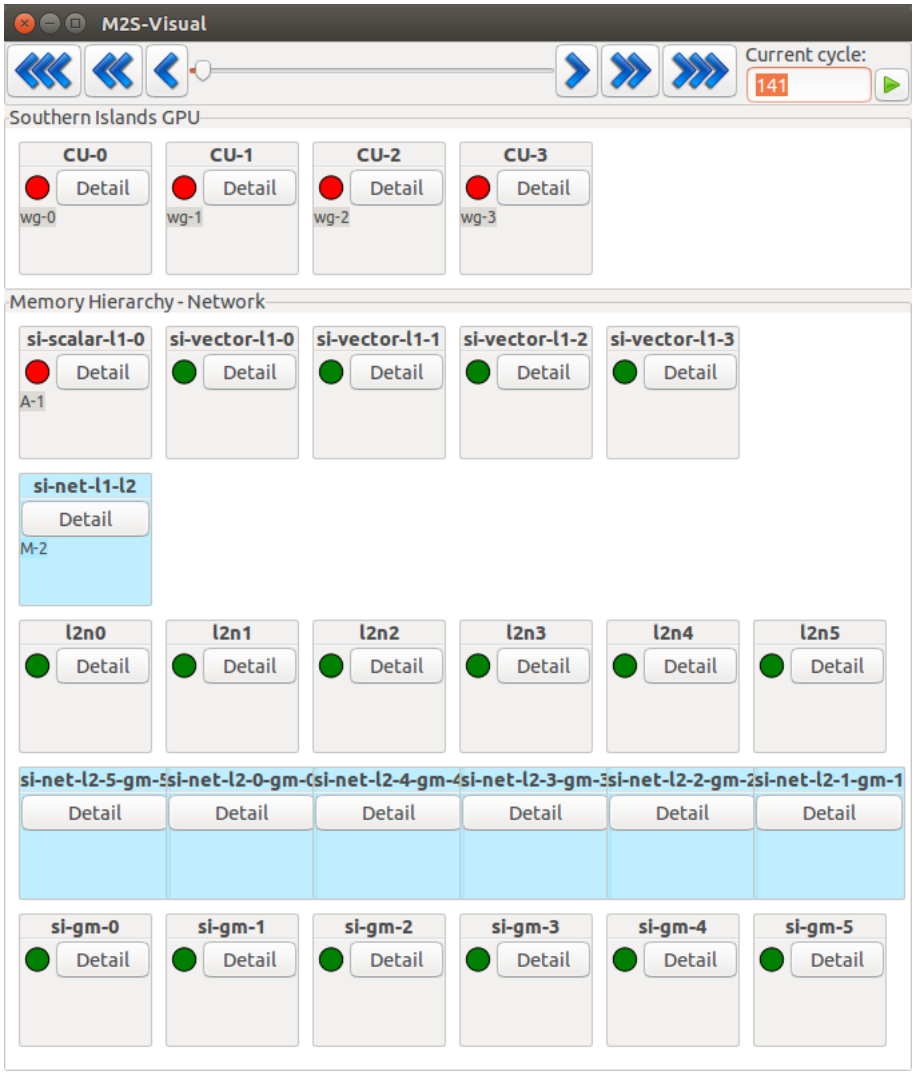
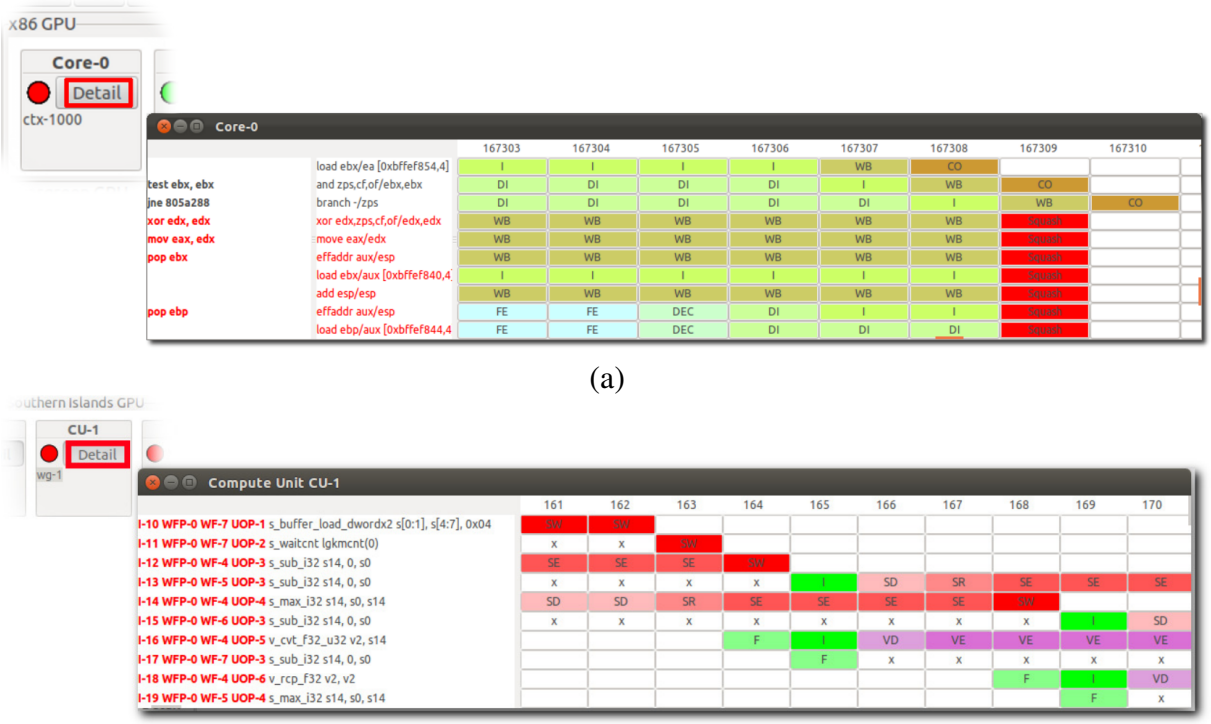


Figura 3.6: A interface do M2S-Visual (Ziabari et al., 2015).

O programa conta com uma barra que permite a navegação pelos diferentes ciclos da simulação além de um botão para iniciar e pausar a execução da simulação. A representação gráfica mostrada pela ferramenta abstrai a arquitetura, mostrando apenas dois painéis. O primeiro é uma área com componentes representando as CPUs ou as GPUs, dependendo do tipo de simulação detalhada que foi iniciada. Cada um componentes possui um botão de detalhes, que ao ser clicado abre uma nova janela com o estado detalhado do elementos. São mostrados diagramas de tempo no caso das CPUs e GPUs. Eles consistem em uma tabela cujas linhas são instruções que estão sendo executadas, e as colunas, indexadas pelos ciclos, mostram o estágio no qual as instruções se encontravam naqueles determinados ciclos. A Figura 3.7 mostra esses diagramas.

Outro painel mostra informações sobre a hierarquia de memória e das conexões da rede interna entre seus elementos. Assim como as GPUs e as CPUs, cada módulo de memória é representado por um elemento que possui um botão que pode ser clicado para mostrar detalhes. No caso dos componentes cinzas, que representam módulos de memória cache ou principal, os endereços que estão sendo servidos são mostrados em uma tabela. A Figura 3.8 mostra a janela



(a)

(b)

Figura 3.7: Diagramas de tempo para uma CPU (a) e uma GPU (b) no M2S-Visual (Ziabari et al., 2015).

para um módulo de memória cache. Cada célula é um bloco, contendo uma etiqueta e um estado. O estado pode ser um dos cinco estados no protocolo de coerência de cache MOESI (Ibrahim et al., 2021), e ele é representado pela cor da célula. As duas células menores à direita da etiqueta representam o número de compartilhadores na cache de nível superior para este bloco e o número de acessos em trânsito para o bloco, respectivamente.

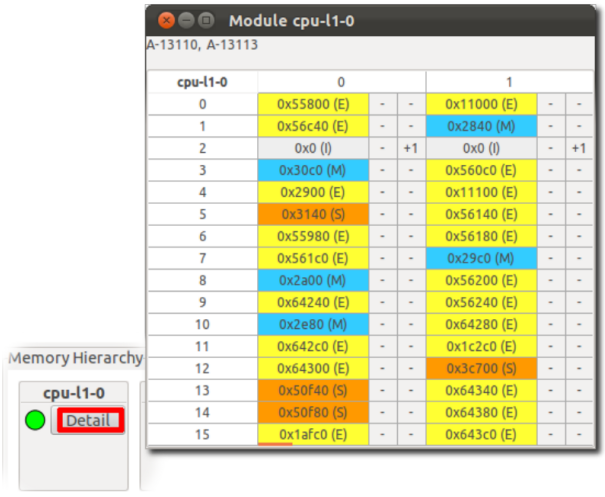


Figura 3.8: Representação de módulo de memória cache no M2S-Visual (Ziabari et al., 2015).

Os elementos azuis no painel de hierarquia de memória representam uma conexão entre um nível maior de memória e um menor. Abaixo do botão são mostrados rótulos com as mensagem que estão atravessando essa conexão em determinado ciclo e, ao clicar neles

informações sobre a mensagem como seu ciclo de criação e estado são mostradas. Ao clicar no botão de detalhes desses componentes uma janela com uma visão mais detalhada sobre a rede interna é mostrada.

O M2S-Visual possui apenas controles para execução passo-a-passo, mas que possui 3 modos de avanço. Um deles permite avançar 1 ciclo, um permite um avanço de 10 e o outro de 100 ciclos, além de possuir uma caixa de texto que permite que um ciclo seja escolhido para que a execução aconteça até chegar nele.

3.8 DIMIPSS

Felix et al. (2006) apresenta o **Didatic Interative MIPS Simulator (DIMIPSS)**, um simulador gráfico para o MIPS monociclo. Esse programa permite a simulação gráfica da execução de um subconjunto das instruções do MIPS. A Figura 3.9 mostra a interface gráfica do programa.

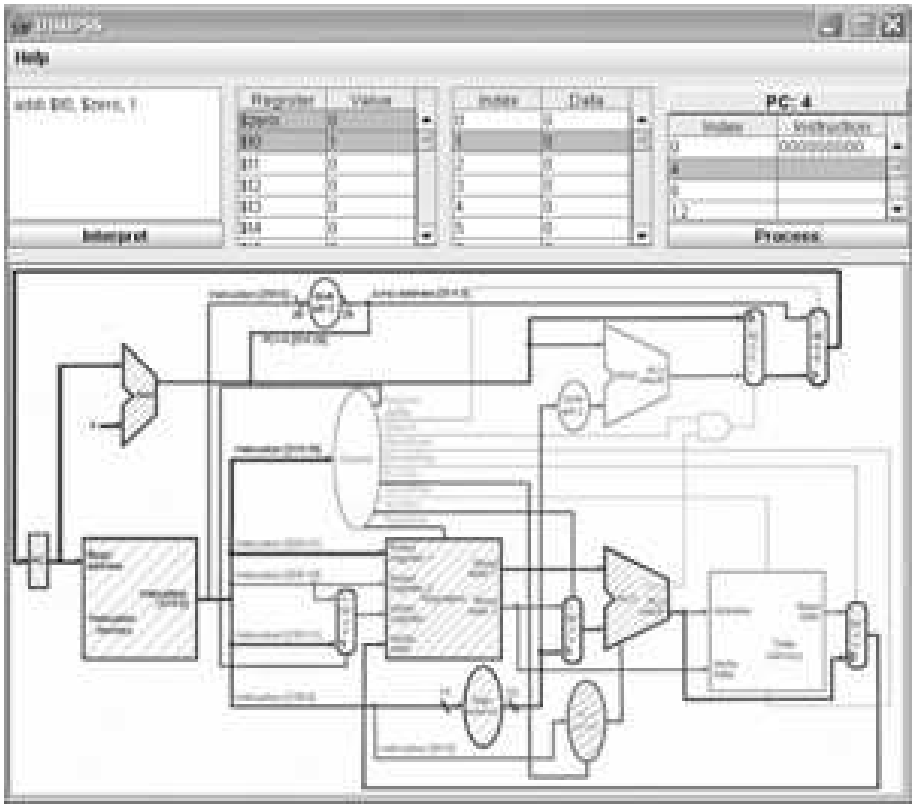


Figura 3.9: A interface do DIMIPSS (Felix et al., 2006).

Ele possui um campo de texto que permite a inserção de código *assembly* MIPS além de outro campo que mostra a cache de instruções junto com o PC, e permite o processamento de cada uma delas. Além destes, existem dois campos com tabelas exibindo a memória de dados e os registradores. A parte principal do programa é uma área com a representação gráfica do processador MIPS, que é mostrada como os desenhos comuns utilizados em livros de arquitetura de computadores. Após inserir um conjunto de instruções no campo de dados e submeter essas instruções para análise, o usuário pode ver o estado da arquitetura. Durante a execução, o campo de visualização e os outros dois campos de dados são atualizados. Os componentes e sinais utilizados por aquela instrução são destacados, bem como os registradores e regiões de memória utilizados.

3.9 MARS

Vollmar e Sanderson (2006) descrevem o **MIPS Assembler and Runtime Simulator (MARS)**, com um enfoque no aprendizado da linguagem *assembly* do MIPS, podendo ser usado para cursos de organização e arquitetura de computadores, programação em linguagem *assembly* e criação de compiladores. Ele implementa uma parte do conjunto de instruções definidas por Patterson e Hennessy (2005).

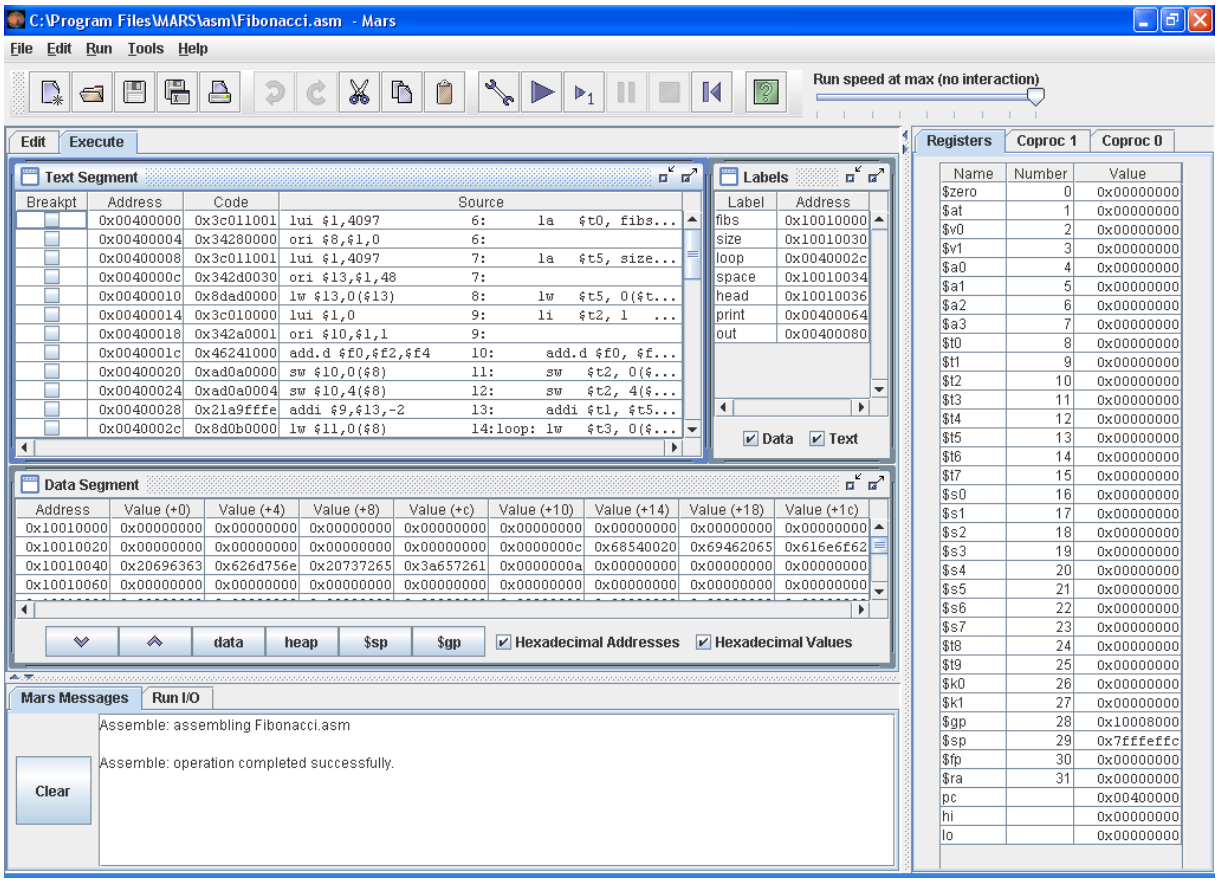


Figura 3.10: A interface do MARS (Vollmar e Sanderson, 2006).

A parte principal do simulador é uma área com duas abas, uma delas é um editor integrado, onde o código de um programa *assembly* pode ser adicionado. Após a montagem desse programa, o usuário pode acessar a outra aba, que contém informações acerca da execução do programa escrito. Essa aba possui três janelas, uma com o segmento de dados, outra com o de texto e uma com os rótulos do programa.

O simulador também conta com uma área com um console com duas abas: uma mostra mensagens do simulador, como erros de sintaxe no *assembly*, e outra serve como entrada e saída de dados da simulação.

A última área de interesse fica ao lado dessas duas, mostrada de maneira vertical, e contém os registradores da arquitetura, representados por uma tabela com três colunas contendo o nome, o número e o valor do registrador em questão.

A ferramenta tem funcionalidades como execução passo-a-passo, execução contínua e também permite que o usuário reinicie a simulação, além do usuário poder controlar a frequência de execução das instruções.

3.10 RARS

O RARS é um simulador para RISC-V construído em cima da versão 4.5 do MARS e possui as mesmas funcionalidades. A Figura 3.11 mostra a interface do programa.

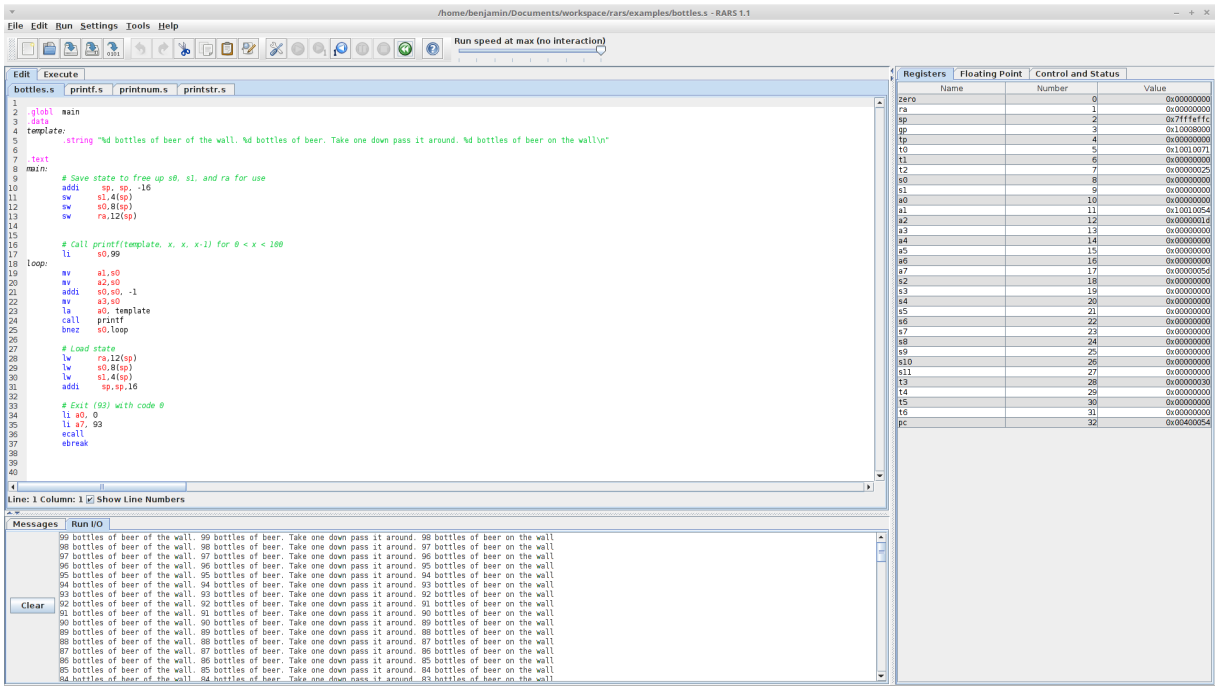


Figura 3.11: A interface do RARS (Landers, 2017).

3.11 QTSPIM

QtSpim é a versão mais nova do simulador SPIM (MIPS ao contrário), apresentado por Patterson e Hennessy (2005). É um simulador para MIPS32 que recebeu com algumas versões ao longo do tempo, como o XSPIM e o PCSPIM (Durand, 2006), a versão com janelas e a para o sistema operacional *Windows*, respectivamente. A Figura 3.12 mostra sua interface.

A janela principal conta com três áreas. A primeira possui duas abas e mostra tabelas com os registradores do processador e seus valores, uma delas para os registradores de inteiros e outra para os de ponto flutuante. A segunda área mostra o código *assembly* que está sendo simulado e possui duas abas também uma para o segmento de texto e outra para o segmento de dados do código. A última área é um espaço de texto horizontal onde é mostrado um histórico de mensagens do simulador, como informações de versão, avisos e erros. Ele também conta com uma janela extra com um console que mostra a saída do programa.

O *Qtspim* conta com dois modos de execução. O modo continuo executa as instruções sem parar, mas não oferece controle da frequência de execução do programa. O simulador também possui um modo de execução que permite o processamento linha a linha do código. É possível adicionar *breakpoints* no código para a exploração do comportamento do programa. A execução pode ser pausada e resumida, ou parada e reiniciada. Não é necessário parar a execução para reiniciar pois o simulador possui um botão de ação para isso.

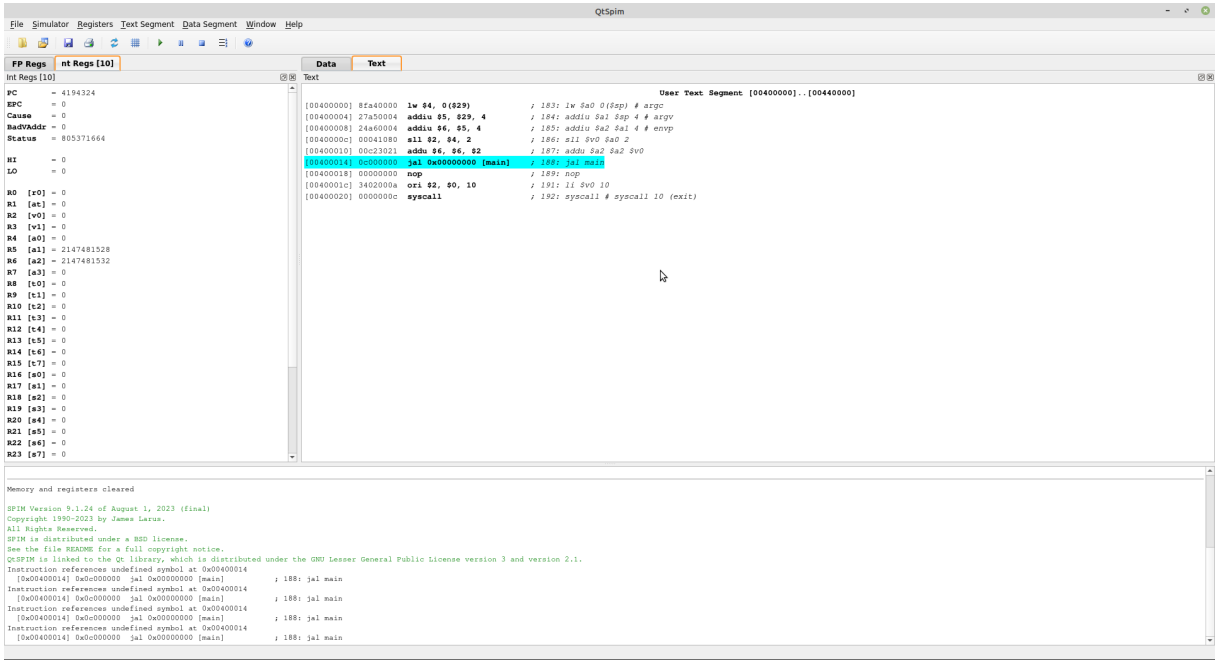


Figura 3.12: A interface do QtSpim.

3.12 VORCS: VISUAL ORDINARY COMPUTER SIMULATOR

A próxima seção apresenta a ferramenta VOrCS de forma mais aprofundada, junto com seu processo de desenvolvimento. Ela foi pensada para a visualização do estado do OrCS e suas funcionalidades foram inspiradas pelas ferramentas encontradas na literatura.

Os dados a serem exibidos na ferramenta são extraídos dos registros observados durante a depuração do simulador, baseada na impressão dos estados dos *buffers* no terminal e na observação dos atributos dos pacotes de instruções, de micro-operações e de requisições para memória. O processo é controlado por meio de uma série de *flags*, que ativam ou desativam funções de impressão, permitindo ao usuário acompanhar informações relevantes.

Alguns dos detalhes observados incluem os atributos *status* dos pacotes, a quantidade de posições ocupadas nos *buffers* dos estágios, o número do ciclo atual do *clock* do processador simulado, os valores de outros atributos das instâncias de *opcode_package_t* e de *uop_package_t*.

Essas observações demonstraram a viabilidade de monitorar o comportamento do simulador simplesmente observando o conteúdo dos *buffers* dos estágios. Uma representação gráfica desse conteúdo e a possibilidade de visualizar esse estado continuamente facilitaria significativamente o processo de depuração inclusive.

Embora a ferramenta suporte visualização de memória, optou-se por não visualizar o estado da *cache* e da memória diretamente, focando, sobretudo, na visualização do processador. No entanto, esses módulos ocasionam alterações de *status* nos pacotes e períodos de ociosidade na CPU, nos quais ela aguarda os dados serem carregados na *cache* a partir da memória principal. Esta situação cria muitos cenários em que apenas o ciclo do *clock* avança, mas nenhuma mudança é observada nos *buffers* do simulador. Esses períodos e a presença das micro-operações de leitura e escrita no *pipeline* evidenciam o uso da memória.

4 PROPOSTA

Nesta seção, as escolhas tomadas no processo de desenvolvimento do trabalho serão apresentadas. Decidiu-se por usar o próprio OrCS para gerar o arquivo de entrada para ferramenta de visualização. Esse arquivo foi denominado de **traço de visualização**. A ferramenta proposta por esse trabalho, o VOrCS, lê o traço linha à linha, instanciando e atualizando as representações gráficas dos *buffers* do simulador e posteriormente os pacotes que trafegam por eles.

4.1 AMBIENTE DE DESENVOLVIMENTO

A linguagem *Python* foi escolhida para o desenvolvimento da ferramenta, na versão 3.8. Essa escolha se deu primeiramente por familiaridade, mas outros pontos, como a rapidez na prototipação, resultante da simplicidade da sintaxe e o ciclo de desenvolvimento reduzido, causado pela não-necessidade de compilação, foram fatores importantes para essa escolha inicial.

Foi utilizado o ambiente virtual *pipenv* para a instalação das bibliotecas necessárias, isso permitiu o desenvolvimento em computadores e sistemas operacionais diferentes, pois o ambiente era facilmente configurado e atualizado. A máquina onde o trabalho foi desenvolvido possui um processador *Intel Core i5-1035G1* com 3.3 Ghz. O sistema operacional usado foi o *Linux Mint Ulyssa*.

4.2 BIBLIOTECA GRÁFICA

Foi necessário escolher uma biblioteca gráfica que seria utilizada para a implementação da interface gráfica e da representação dos elementos simulados. Algumas bibliotecas foram consideradas e submetidas a um teste que consistia na implementação de um programa simples, que tinha como objetivo averiguar como a biblioteca funcionava. O programa teste lia de um arquivo de texto comandos para criar um triângulo na tela, atualizar sua posição, retorná-lo à posição inicial e deletá-lo da tela. Cada comando era lido quando o usuário apertava a seta direita do teclado. Com isso, era possível avaliar:

- Como funcionava a implementação e renderização de elementos gráficos customizados na biblioteca, através da criação de uma classe específica para o triângulo.
- Como seria possível renderizar um elemento, modificar alguma característica dele e renderiza-lo novamente, através da mudança de posição do triângulo.
- Como seria possível parar de renderizar um elemento após sua instanciação, através da deleção do triângulo.
- Como funcionava o tratamento de eventos de entrada do usuário e de entrada de arquivo na biblioteca, através do uso do arquivo de texto com os comandos e da seta direita para avançar o programa.

A estrutura de código ficou similar para todas as bibliotecas, e elas foram comparadas com base em algumas características desejadas. A Tabela 4.1 mostra cada biblioteca e se possui ou não determinada característica desejada, entre elas:

- **Facilidade de Uso**, que refere-se à simplicidade e acessibilidade da biblioteca para os desenvolvedores, especialmente iniciantes.
- **Recursos gráficos**, que determina a capacidade que a biblioteca tem de criar interfaces de usuário visualmente atraentes e interativas.
- A **Portabilidade** diz respeito à capacidade da biblioteca de funcionar em diferentes sistemas operacionais e ambientes de desenvolvimento sem a necessidade de modificações significativas no código.
- **Comunidade e Suporte** caracteriza se a biblioteca possui uma rede de conteúdo e usuários ativos atualmente, dispostos a fornecer ajuda, recursos adicionais e solução de problemas aos desenvolvedores.
- A categoria **Desempenho** refere-se à sua eficiência em termos de consumo de recursos de hardware e velocidade de execução.
- A característica **Elementos complexos** se refere a presença ou não de elementos de interface mais complexos que botões e caixas de texto na biblioteca.

	Facilidade de uso	Recursos gráficos	Portabilidade	Comunidade e suporte	Desempenho	Elementos complexos
Pygame	•	•	•	•		
Pyglet	•	•	•	•	•	
Tkinter	•		•	•		•
PyQt		•	•	•	•	•

Tabela 4.1: Classificação das bibliotecas gráficas pesquisadas.

As bibliotecas *Pyglet* e *PyQT* foram as que mais possuíam características desejadas, mas a disponibilidade de elementos de interface complexos na *PyQT* foi definitiva para a escolha. Embora a biblioteca *PyQt* tivesse uma curva de aprendizado maior, a gama de elementos disponíveis para customização se mostrou mais desejável do que a facilidade da biblioteca *Pyglet*.

Com a escolha da biblioteca gráfica, foi possível iniciar o desenvolvimento da ferramenta. A próxima seção entra em detalhes sobre a organização do código e o funcionamento do programa.

4.3 ESTRUTURAÇÃO DE CÓDIGO E FUNCIONAMENTO

A proposta recebe como entrada um arquivo que possui informações sobre como atualizar a visualização. Ele é lido conforme comandos de execução são acionados pelo usuário e mudanças gráficas acontecem de acordo com o que é lido. Quando o final do arquivo é

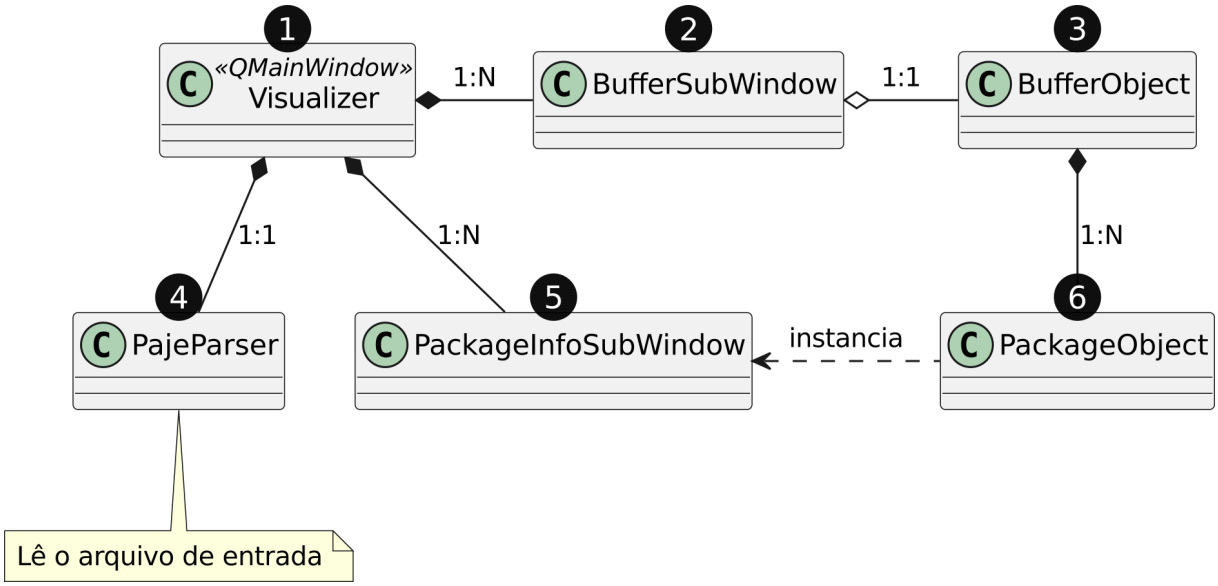


Figura 4.1: Diagrama de classes do VOrCS.

atingido, o programa mostra uma caixa de diálogo avisando que o traço foi finalizado. Sempre que o programa inicia, o último traço aberto é carregado automaticamente.

A Figura 4.1 mostra um diagrama com as seis classes existentes no programa enumeradas.

1. **Visualizer** é o componente que implementa a janela do VOrCS e é instanciado pela função principal do programa. É essa classe que trata os eventos do usuário, bem como os comandos contidos no arquivo de entrada.

2. **BufferSubWindow** é a classe que representa uma sub-janela relacionada a uma instância da classe *BufferObject* do programa.

3. A classe **BufferObject**, representa uma estrutura do OrCS que foi selecionada para visualização.

4. **PajeParser** é a classe responsável por ler o arquivo de entrada e passar os eventos em ordem para serem tratados.

5. **PackageInfoSubWindow** implementa uma sub-janela que contém informações sobre um *operation package* ou *µop package* do OrCS.

6. A classe **PackageObject** representa um pacotes de instruções ou de micro-operações do OrCS, que trafegam por diferentes estruturas ao longo da visualização.

A próxima subseção explica quais os elementos que compõem o arquivo lido e processado pela classe *PajeParser*.

4.3.1 Arquivo de entrada

O arquivo de entrada da ferramenta utiliza o formato Pajé, como definido por Schnorr et al. (2015). Assim como as informações de *debug* do OrCS, a geração do arquivo é feita durante a simulação. Antes mesmo de implementar as principais funcionalidades, foi criado um traço à mão, denominado de traço sintético. Ele foi útil para a depuração e desenvolvimento da

lógica de atualização da visualização, e também serviu para a seleção de quais eventos seriam adicionados no arquivo de entrada.

Código 4.1: Evento DefineBufferType

```
%EventDef DefineBufferType 0
% Name string
% Color string
% IsContainer bool
%EndEventDef
```

O evento **DefineBufferType**, cuja definição é mostrada no Código 4.1, é usado para definir os tipos de *buffers* existentes e suas cores. No VOrCS, os tipos podem ser "FETCH_BUFFER", "DECODE_BUFFER", "UNIFIED_RESERVATION_STATION", "UNIFIED_FUNCTIONAL_UNITS", "MEMORY_ORDER_BUFFER_READ", "MEMORY_ORDER_BUFFER_WRITE" ou "REORDER_BUFFER", que são as estruturas do OrCS selecionadas para observação. O evento possui possui três campos:

- **Name**, o nome do tipo. Usado para indexar um dicionário que guarda os tipos definidos.
- **Color** é uma *string* que contém um código de cor hexadecimal. O dicionário de tipos contém esse código de cor e é usado para colorir os *buffers* que possuam aquele determinado tipo.
- **IsContainer** é uma variável booliana que é utilizada para definir o comportamento do *buffer* em relação a inserções e remoções. Se for falso, o *buffer* se comporta de maneira circular, ou seja, inserções sempre ocorrem no final dele e quando o tamanho máximo é atingido a próxima inserção é feita no primeiro índice vazio a partir do início do *buffer*. Além disso, quando um elemento é removido, os outros elementos permanecem na mesma posição e os índices dos pacotes removidos podem novamente serem ocupados. Se o campo for verdadeiro, as inserções continuam acontecendo no final do *buffer*, mas após uma remoção, todos os elementos depois da posição do elemento removido são deslocados para ocupar o índice vazio do meio.

Código 4.2: Evento CreateBuffer

```
    %EventDef CreateBuffer 1
    % Id int
    % Name string
    % TypeName string
    % Width int
    % Size int
    %EndEventDef
```

A definição do evento **CreateBuffer** é mostrada no Código 4.2. Esse evento é usado para instanciar os *buffers* e contém os seguintes campos:

- O campo **Id** é um identificador único, utilizado como índice da estrutura de dados que representa o *buffer* e que aparece nos outros eventos do traço de visualização.
- **Name** é o nome do *buffer*, usado para exibição e também como chave para indexar um dicionário com as *Ids* dos *buffers*.
- **TypeName** é o nome do tipo do *buffer*, como definido pelo evento *DefineBufferType*.

- **Width** indica a largura do *buffer*, que no OrCS é relacionado à largura do estágio em que ele é usado.
- **Size** indica a capacidade do *buffer*, como definido no arquivo de configuração do OrCS.

Código 4.3: Evento DefinePackageType

```
%EventDef DefinePackageType 2
% Id int
% Name string
%EndEventDef
```

O evento **DefinePackageType**, mostrado no Código 4.3 é usado para definir os tipos de pacote do visualizador, que podem ser "OperationPackage", ou "UopPackage"neste trabalho. Esse evento possui dois campos:

- **Id**, o identificador único do tipo, usado para indexar um dicionário que guarda os tipos e usado por outros eventos.
- **Name**, o nome do tipo, utilizado para display.

Código 4.4: Evento DefineStatusColor

```
%EventDef DefineStatusColor 9
% Status string
% Color string
%EndEventDef
```

O evento **DefineStatusColor**, como visto na figura 4.4, é utilizado para definir cores para os estados que os pacotes podem ter ao longo da visualização e contém os campos Status e Color, ambos *strings*.

- **Status**, o nome do *status*, usado para indexar um dicionário.
- **Color**, código de cor hexadecimal para representar visualmente determinado *status*.

A parte do arquivo que contém os eventos mostrados até aqui foi criada de maneira fixa pelo programador no código do OrCS, descrita antes da simulação em si. Outros eventos, relacionados aos pacotes e ao *clock* são gerados durante a simulação. Esses eventos são usados para descrever o comportamento do OrCS por meio da descrição de operações sobre os pacotes. Eles são apresentados à seguir.

Código 4.5: Evento InsertPackage

```
%EventDef InsertPackage 5
% TypeId int
% Id string
% BufferIds string
% Content string
%EndEventDef
```

O evento **InsertPackage** definido no Código 4.5 contêm quatro campos:

- **TypeId** o identificador do tipo do pacote, como definido pelo evento *DefinePackageType*.

- **Id**, o número identificador do pacote. Junto com o TypeId, permite uma identificação única, mesmo que o pacote esteja em qualquer *buffer* e que possuam pacotes de outro tipo com o mesmo Id.
- **BufferIds** contém uma representação em *string* de um vetor com as ids dos *buffers* onde esse pacote está sendo inserido, isso permite que um pacote seja inserido em mais de um *buffer* ao mesmo tempo.
- **Content** é uma representação em *string* de um dicionário com informações extras sobre o pacote, para serem mostradas quando o usuário inspecionar um pacote.

Código 4.6: Evento RemovePackage

```
1  %EventDef RemovePackage 6
2  % TypeId int
3  % Id string
4  % BufferIds string
5  %EndEventDef
```

A definição do evento **RemovePackage** no Código 4.6 mostra os campos:

- **TypeId**, o tipo do pacote a ser removido.
- **Id**, o identificador do pacote.
- **BufferIds**, novamente contém um vetor com as ids que indicam de quais *buffers* esse pacote deve ser removido.

Código 4.7: Evento UpdatePackage

```
1  %EventDef UpdatePackage 7
2  % TypeId int
3  % Id string
4  % BufferIds string
5  % Content string
6  %EndEventDef
```

O evento **UpdatePackage**, do Código 4.7 contém os campos **TypeId**, **Id**, **BufferIds** e **Content**. Esses campos têm o mesmo significado dos campos do evento InsertPackage, contudo, em vez de causar a inserção de um pacote, ele faz com que o conteúdo do dicionário no campo Content seja utilizado para atualizar o conteúdo do pacote com Id correspondente em cada *buffer* indicado pelo campo BufferIds.

Código 4.8: Evento Clock

```
1  %EventDef Clock 8
2  % Cycle int
3  %EndEventDef
```

O último evento impresso pelo OrCS durante a simulação é o evento **Clock**, definido no Código 4.8 e ele possui apenas um campo, Cycle, que indica qual é o ciclo atual do processador.

Os três eventos de pacote e o de *clock* dependem da simulação do OrCS, mas as definições dos eventos e os eventos de criação de *buffers* são fixos, eles aparecem embutidos no código do simulador. O Apêndice A é um exemplo resumido de um arquivo pajé de entrada para a ferramenta.

4.4 PROGRAMA

A Figura 4.2, apresenta a janela principal do visualizador. A versão base do programa possui cinco áreas:

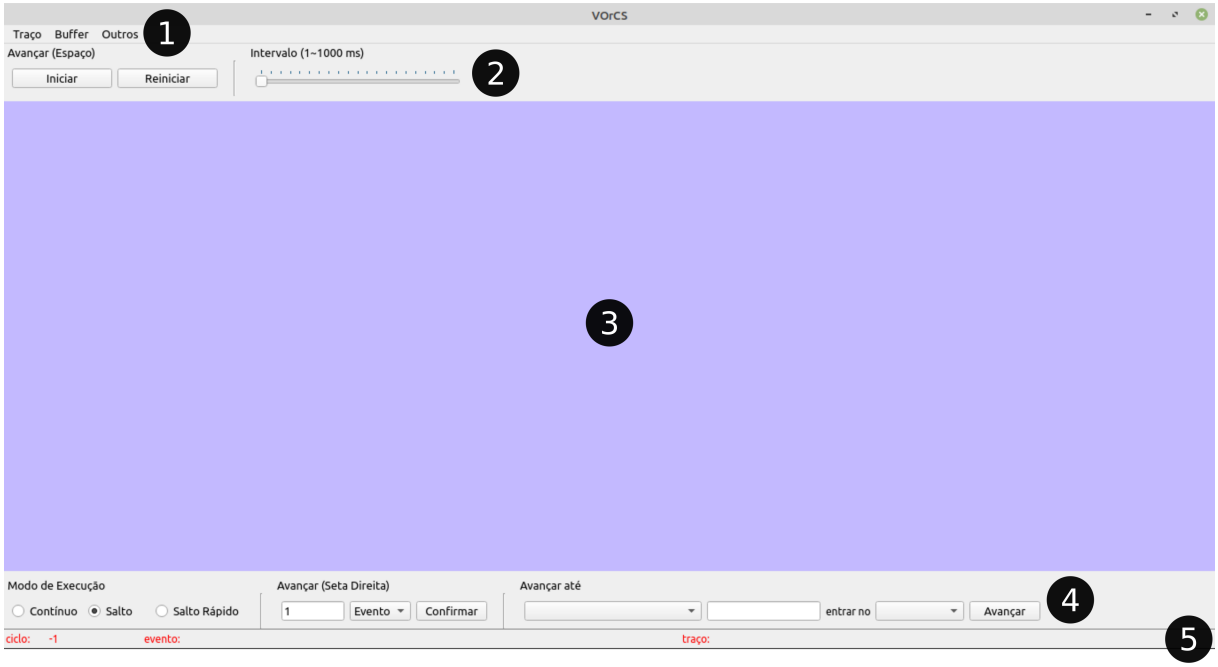


Figura 4.2: Janela principal do VOrCS.

1. Uma **barra de menus** que possui três opções: a primeira, "Traço", contém as ações "Carregar" e "Sair", para carregar um traço de simulação no programa e encerrar a visualização, respectivamente. A segunda opção, "Buffer", contém duas ações, "Cascata" e "Ladrilho", para organizar automaticamente os elementos na área de visualização. A última opção, "Outros", possui duas ações, "Tela Cheia", para inciar a visualização em tela cheia, e "Limpar Configurações", para apagar configurações salvas.

2. Uma **seção de comandos** para controlar a execução contínua, que possui duas áreas, "Avançar" e "Intervalo".

3. Uma **área de visualização**, vazia na Figura 4.2, é onde serão mostradas as representações gráficas dos *buffers* e pacotes da ferramenta, que são instanciados conforme os eventos pajé do traço de simulação são tratados.

4. A segunda **seção de comandos** para controlar a execução passo-a-passo e a busca. Ela contém três áreas, "Modo de Execução", "Avançar" e "Avançar até".

5. Uma **barra de status** que contém três informações, o ciclo, o evento atual e o endereço do arquivo do traço de simulação.

A seção a seguir entra em detalhes sobre como os *buffers* do OrCS são representados na ferramenta de visualização. Após isso a representação dos pacotes e as funcionalidades da ferramenta são apresentadas.

4.4.1 Buffers

Na fase inicial do desenvolvimento foi necessário definir uma representação gráfica para as estruturas do OrCS que seriam observadas. As principais discussões foram acerca do formato delas e de sua posição e orientação na tela. Foi considerado a implementação de uma representação parecida com um diagrama de blocos, onde cada componente da arquitetura e seus sinais e conexões seriam mostrados. Contudo, essa escolha não solucionava o problema da disparidade entre a implementação do simulador e o *hardware* simulado, além de gerar mais complexidade no gerenciamento da visualização.

As discussões também incluíram a maneira como isso seria configurado. À princípio, o arquivo de configuração do OrCS seria usado para definir a estrutura da visualização e um programa que extraísse as informações relevantes dele e codificasse isso em eventos pajé que construíssem a representação seria escrito. Os elementos da representação assumiriam posições, tamanhos e orientações fixas uma vez que esses eventos iniciais fossem processados, mas isso não solucionava o problema de como seriam representados os pacotes que trafegam por essas estruturas. Além de que o processamento dos pacotes do OrCS e sua tradução em sinais aumentaria a complexidade da ferramenta de visualização.

Também seria necessário considerar como definir essa visualização de maneira genérica para diferentes tamanhos de tela e talvez como adaptá-la conforme o usuário modificasse o formato da janela da ferramenta, inclusive. Adicionalmente, como o OrCS possui suporte a diferentes microarquiteturas, então seria necessário a criação de eventos que gerassem a representação visual para cada uma dessas microarquiteturas.

No fim, a desconexão entre os *buffers* do OrCS e a microarquitetura simulada desmotivou o uso de uma representação gráfica fixa e semelhante a um diagrama de blocos. Foi observado que a principal informação que a ferramenta mostraria seriam os pacotes, pois observando o seu fluxo durante o ciclo de vida da simulação, era possível compreender o comportamento do programa simulado. Sendo assim, o formato dos *buffers* poderia ser abstraído, uma vez que observando um pacote dentro de um *buffer* e sabendo de qual *buffer* se tratava, já era possível identificar em qual estágio o ciclo de execução aquela instrução estava e sob qual operação ela seria submetida.

Com isso, os *buffers* foram abstraídos em sub-janelas, a serem mostradas na área de visualização.Essa implementação sofreu influencia do conceito de *Multiple Document Interface* (MDI) (Kumar, 2005), um tipo de interface gráfica onde várias janelas residem dentro de uma única janela principal. Nesse tipo de interface cada sub-janela tem a sua gerência de dados, e embora esse tipo seja utilizado para a visualização de diversos documentos ao mesmo tempo, o mesmo pode ser aplicado para os *buffers*. A Figura 4.3 mostra a ferramenta com diversas sub-janelas abertas, a cor de fundo de cada sub-janela é definida pelo evento **DefineBufferType**, bem como o comportamento da inserção e remoção dos pacotes deles, como mencionado na subseção 4.3.1.

A escolha de uma MDI para a representação gráfica dos *buffers* simplificou o processo de desenvolvimento, possibilitando que as discussões avançassem para o problema da representação dos pacotes. A responsabilidade pela definição da disposição dos *buffers* foi retirada do programador e transferida para o usuário, proporcionando maior flexibilidade à interface gráfica e permitindo a personalização para a observação de um conjunto específico de *buffers*, inclusive. Além disso, quando os *buffers* são instanciados, suas posições e dimensões anteriores são restauradas, mantendo a disposição customizada pelo usuário na última vez que o traço foi aberto.

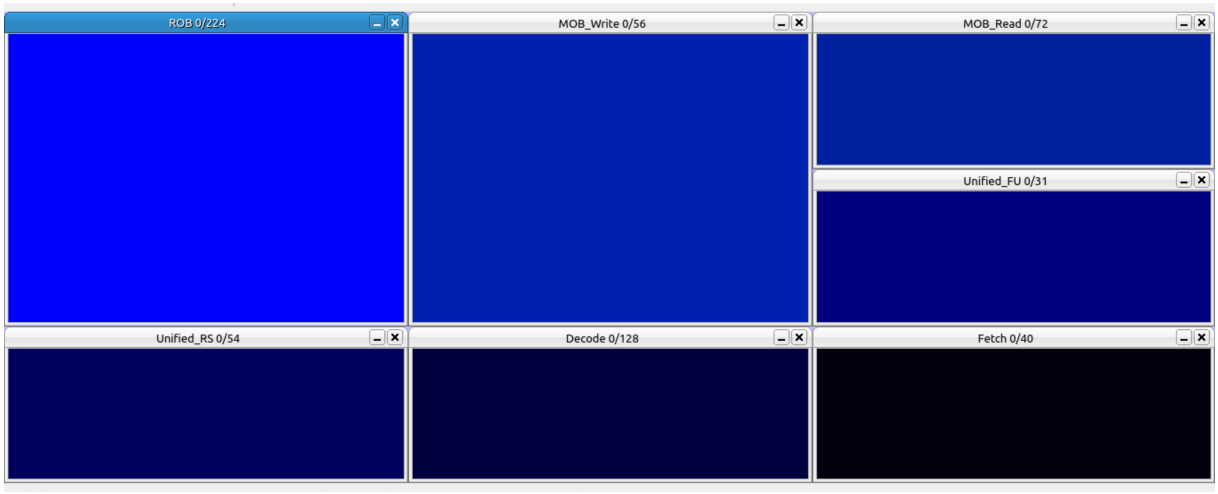


Figura 4.3: Buffers posicionados pela opção "Tile".

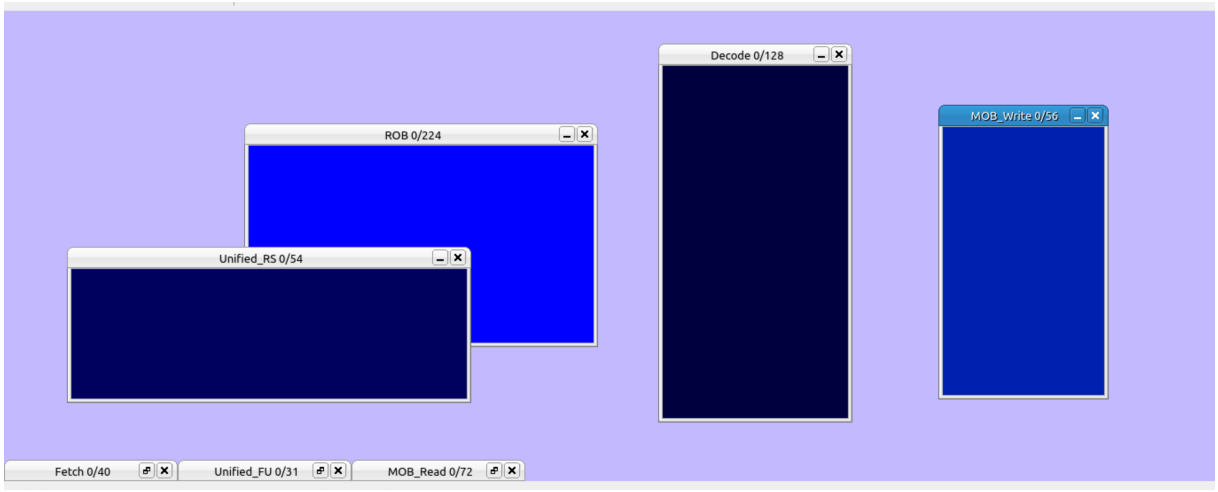


Figura 4.4: Redimensionamento e reposicionamento dos Buffers.

O uso desse modelo de interface também permite que as sub-janelas sejam reposicionadas, redimensionadas ou minimizadas, como pode ser visto na Figura 4.4. A opção "Buffer" da barra de menus contém ações que permitem a organização em massa das sub-janelas. A Figura 4.3 mostra a disposição após a opção "Ladrilho" ser clicada.

4.4.2 Pacotes

Os pacotes representam as instâncias de *opcode_package_t* e *uop_package_t* do OrCS, os quais são representados por retângulos coloridos rotulados na ferramenta de visualização. O atributo *status* dos pacotes determina a cor do retângulo, e a cor de cada valor possível de *status* é definida pelos eventos do tipo **DefineStatusColor**. O texto do rótulo varia de acordo com o tipo de pacote, que para o OrCS pode ser *OperationPackage* ou *UopPackage*. Os rótulos dos *OperationPackages* consistem no valor do atributo *opcode_number* da instância, enquanto o rótulo dos *UopPackages* contém o valor de *opcode_number* do *OperationPackage* que gerou a instância de μop, seguido de uma barra e o valor do campo *uop_number* da instância. Rótulos muito grandes aparecem resumidos, e são revelados quando o mouse é posicionado acima do pacote.

Os pacotes são renderizados dentro das sub-janelas e sua disposição é definida pela ordem em que foram inseridos, pela capacidade do *buffer* e a largura do estágio do pipeline associado à ele no OrCS. A Figura 4.5 mostra algumas sub-janelas com pacotes renderizados.

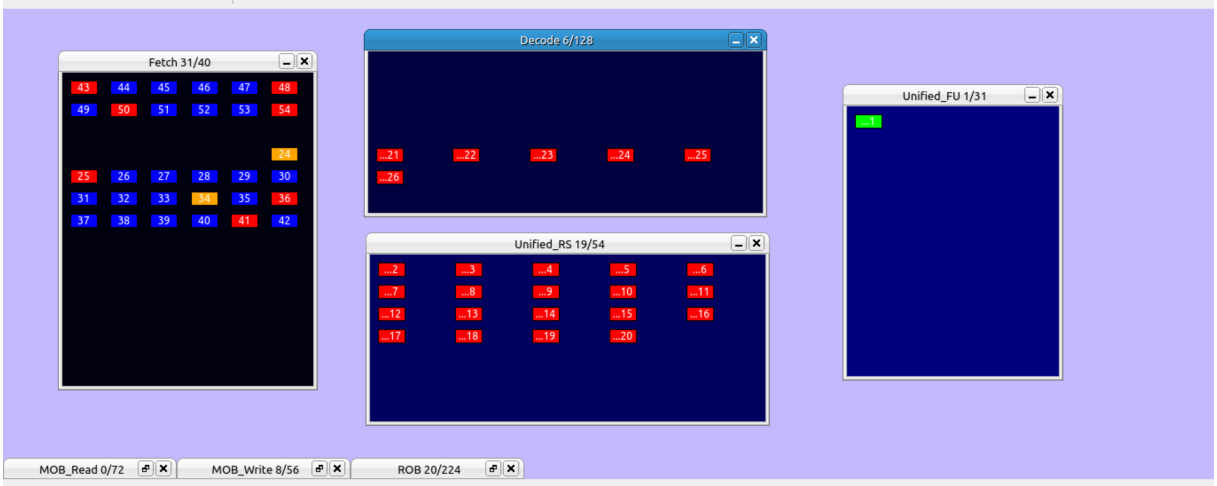


Figura 4.5: Buffers com pacotes em seu interior.

Ao clicar em um pacote é possível acessar informações extras sobre ele. Estas informações são alguns dos atributos das classes *opcode_package_t* e *uop_package_t*. As informações extras são mostradas como uma nova sub-janela e enquanto ela está aberta o pacote fica destacado por um contorno da cor amarela, indicando que está selecionado. Quando destacados, rótulos grandes são revelados mesmo se o mouse não estiver posicionado acima do pacote.

O cabeçalho da janela de informações extras possui o tipo do pacote seguido pelo seu identificador, juntamente com botões para minimizar e fechar a janela. Mais de uma janela pode ser aberta simultaneamente e seu conteúdo é atualizado conforme a visualização avança. A Figura 4.6 mostra duas sub-janelas de informações extras abertas, do *OperationPackage* 106 no *Fetch* e do *UopPackage* 26 no *ROB*.

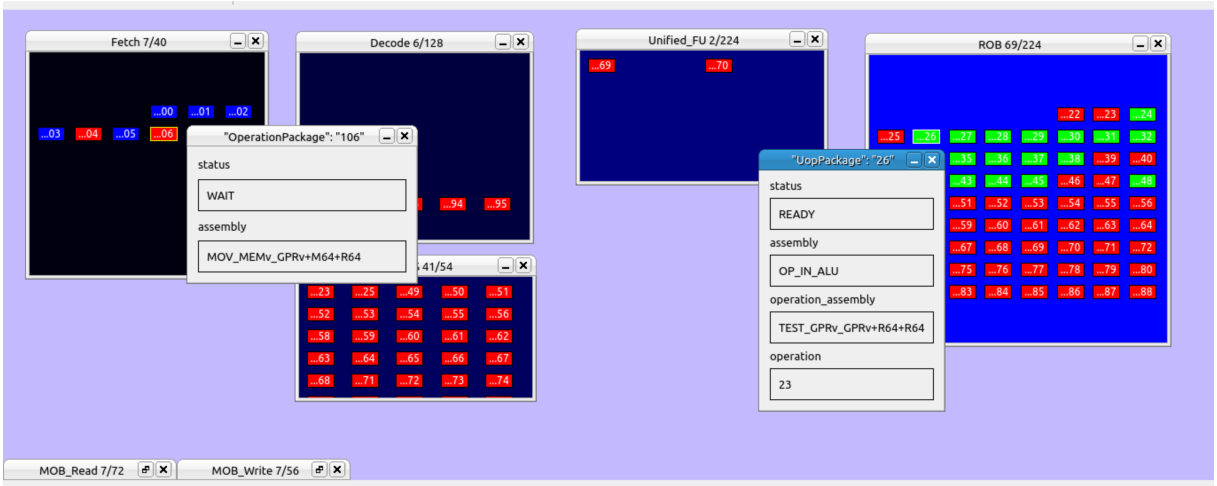


Figura 4.6: Janelas de informações extras de pacotes no *Fetch* e no *ROB*.

4.4.3 Funcionalidades

Os principais comandos implementados na ferramenta estão relacionados com o tratamento dos eventos do traço Pajé pela ferramenta. Foram implementados os comandos **Avançar**, **Avançar até**, **Iniciar/Pausar** e **Reiniciar**. Eles são utilizados através das seções de comando na parte superior e inferior da janela principal, mostradas na Figura 4.2.

O comando **Avançar** é implementado por um campo de entrada de texto, um seletor e um botão. Ele pode ser utilizado para avançar ciclos ou eventos, e é possível avançar uma única unidade ou várias unidades. A Figura 4.7 mostra a área "Avançar" da seção de comandos, com seus valores padrões.

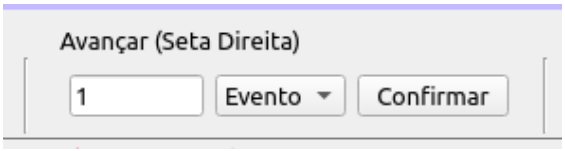


Figura 4.7: Área do comando "Avançar" com valores padrões.

A Figura 4.8 mostra a área "Avançar" com o valor da caixa de texto alterado e o seletor de tipo de avanço aberto. Em determinado momento, o usuário pode preencher o campo com um número, escolher "Ciclo" ou "Evento" no seletor e após isso clicar no botão "Confirmar" com o mouse ou apertar a seta direita do teclado e a visualização avança o número indicado de ciclos ou eventos.

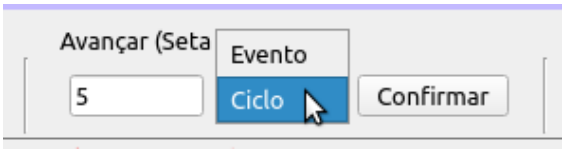


Figura 4.8: Área do comando "Avançar" com valores modificados e seletor aberto.

O comando **Avançar até** funciona como uma busca, permitindo procurar por uma instrução ou por uma micro-operação dentro de algum *buffer*. Ele utiliza dois seletores e um campo de entrada de texto para isso. A Figura 4.9 mostra a área "Avançar até". Os valores possíveis para os seletores dependem dos eventos do tipo **DefinePackageType** e **CreateBuffer**.

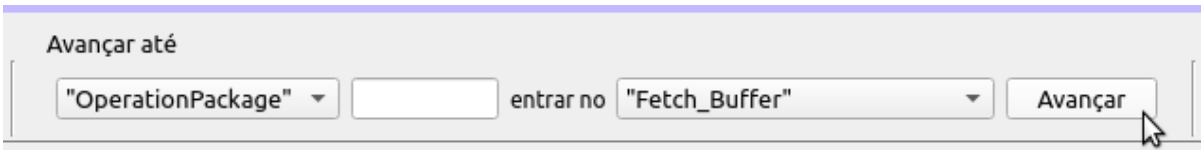


Figura 4.9: Área do comando "Avançar até" com valores padrões.

O primeiro campo é um seletor para o tipo de pacote que será procurado, o segundo é uma caixa de entrada de texto para a indicação do identificador do pacote e o terceiro campo é um seletor para o *buffer* que será considerado. Quando o botão "Avançar" dessa área é clicado, a visualização avança até que o pacote do tipo escolhido com o identificador especificado seja inserido no *buffer* selecionado. Nesse momento, o pacote em questão é destacado.

A Figura 4.10 mostra o resultado desse comando para um *OperationPackage* com identificador 10 no *buffer* "Fetch".

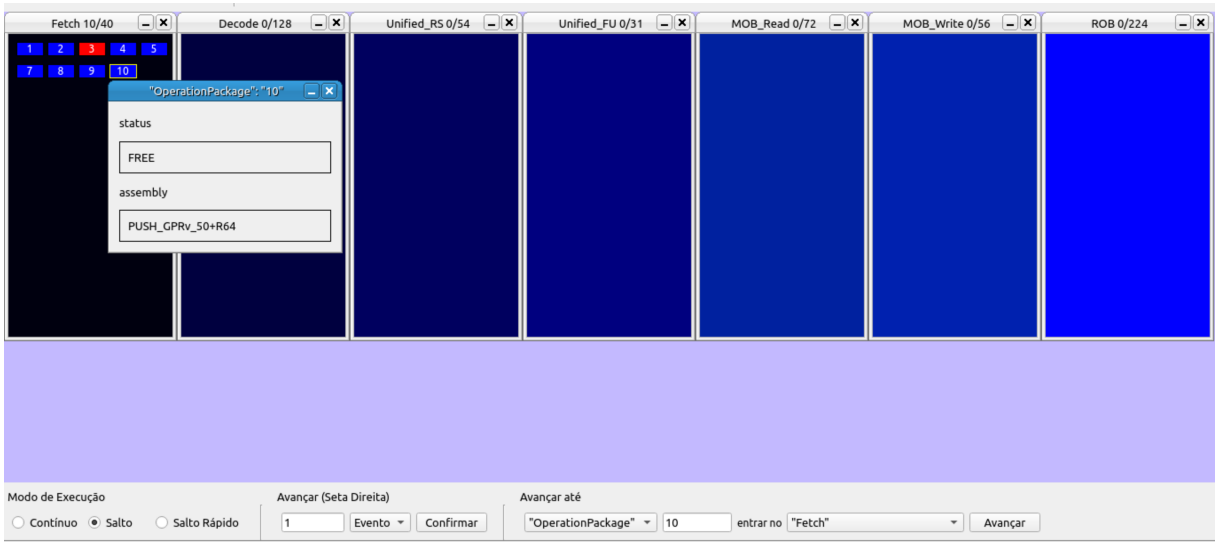


Figura 4.10: Resultado de execução de "Avançar até".

Quando um *OperationPackage* é buscado dentro de qualquer *buffer* que não seja o *Fetch*, a visualização pausa quando a primeira *UopPackage* que teve como origem esse pacote estiver dentro do *buffer* selecionado.

O comando **Iniciar/Pausar** serve para controlar um laço contínuo de tratamento de eventos. Ao clicar no botão "Iniciar" ele muda para "Pausar" e inicia o laço, avançando a visualização até que o botão seja clicado novamente ou que o traço termine. O comando **Reiniciar** para a visualização do traço, remove todos os *buffers* e pacotes e retorna a ferramenta para o inicio do arquivo de entrada. Ele pode ser utilizado a qualquer momento e, ao clicar no botão, quaisquer modificações feitas no traço desde seu carregamento na ferramenta são registradas.

Os comandos mencionados até aqui formam a base necessária para a exploração do traço. Com eles é possível buscar por instruções ou ciclos específicos ou observar o tráfego de pacotes pelo pipeline. O comando Reiniciar permite que o usuário volte a execução no início, removendo a necessidade de iniciar uma nova seção para observar eventos ou ciclos já visualizados. Existem ainda duas áreas muito importantes, que controlam a maneira como os comandos de avanço tratam os eventos e a atualização da representação gráfica dos *buffers* e pacotes. A principal delas é a área de **Modos de Execução**, visível na Figura 4.11.

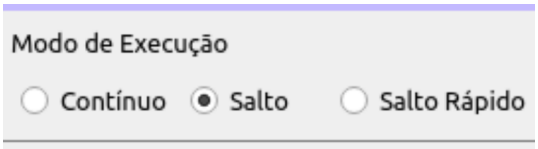


Figura 4.11: Área para seleção do modo de execução da ferramenta.

Quando o modo **Contínuo** está selecionado, cada evento é processado e atualiza os pacotes e *buffers*, a ferramenta espera um intervalo entre cada evento. A área **Intervalo**, mostrada na Figura 4.12, possui um controle deslizante utilizado para escolher a duração do intervalo em milissegundos, que pode ser de 1ms até 1000ms.

No caso do modo **Salto**, os eventos do avanço selecionado são processados e apenas depois a representação gráfica é atualizada, causando um salto no estado da visualização.

No último modo, **Salto Rápido**, a ferramenta só processa os eventos de ciclo e ignora todas as inserções, remoções e atualizações. O uso desse tipo de avanço pode fazer com que

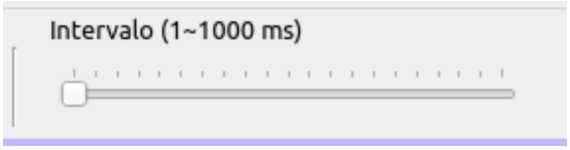


Figura 4.12: Área para o controle da duração do intervalo do modo contínuo.

remoções ou atualizações façam referência à pacotes que não foram inseridos em algum *buffer* e também pode fazer com que alguns pacotes nunca sejam removidos de algum *buffer*, se o evento de remoção dele for pulado. Para sanar essas ocorrências, a ferramenta limpa todos os *buffers* no momento do avanço e quaisquer referências à pacotes não existentes em algum *buffer* são capturadas e registradas no terminal, sem causar erros.

Importante ressaltar que para os comandos "Avançar" ou "Avançar até", o modo de execução selecionado determina como se dá a visualização. Isso não se aplica ao comando "Iniciar/Pausar", que sempre inicia um laço de execução no modo Contínuo.

Além das funcionalidades relacionadas à simulação, a ferramenta é capaz de mostrar métricas acerca da ocupação dos *buffers* durante a visualização. As métricas são mostradas na forma de histogramas contidos em sub-janelas, que também podem ser redimensionadas, reposicionadas e minimizadas. Os histogramas são atualizados sempre que um evento do tipo **Cycle** é tratado. O eixo *x* representa o número de elementos do *buffer* e o eixo *y* a porcentagem de ciclos em que o *buffer* se encontra com aquele número de elementos. Esses histogramas foram denominados de "Histogramas de Ocupação". A Figura 4.13 mostra os *buffers* "Fetch" e "Decode" e seus histogramas.

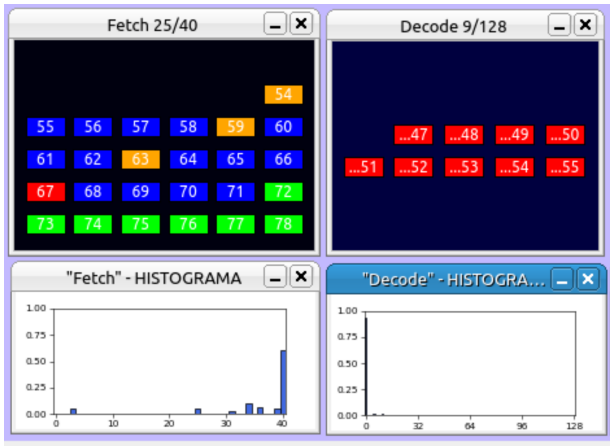


Figura 4.13: Histogramas de ocupação e seus *buffers*.

Os histogramas de ocupação mostram de uma maneira geral como foi a ocupação de um determinado *buffer* durante uma execução, mas eles sacrificam uma informação importante acerca da temporalidade da informação que mostra. Considerando um *buffer* com apenas 1 posição e duas situações: na primeira ele fica 5 ciclos vazio e depois 5 ciclos a posição ocupada, na segunda situação, ele fica os 5 primeiros ciclos com a posição ocupada e os 5 últimos vazios.

Nas duas situações o histograma de ocupação seria o mesmo, o *buffer* passa metade do tempo vazio e a outra com 1 elemento. Em algumas situações a informação de quais ciclos foram os vazios ou não é importante, por isso foi implementado um segundo tipo de histograma na ferramenta, denominado de "Histograma de Fases". A Figura 4.14 mostra esses histogramas, para mudar o tipo de histograma mostrado basta clicar no cabeça-lho da sub-janela do histograma.

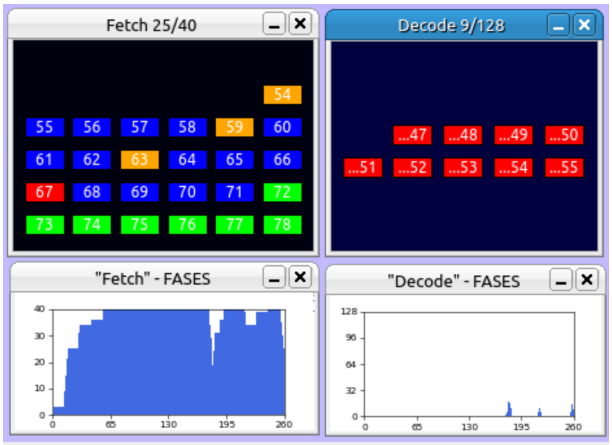


Figura 4.14: Histogramas de fases e seus *buffers*.

Após um avanço no modo Salto Rápido, os dados dos histogramas são zerados e a partir daí a informação mostrada só diz respeito à porção de ciclos após o final do salto.

4.5 RESULTADOS

Para comprovar a capacidade da ferramenta em ajudar na compreensão do comportamento do simulador e também observar como os comportamentos esperados apareceriam na visualização, foram realizados alguns estudos de caso. Essa seção apresenta dados coletados durante a execução da ferramenta de visualização desenvolvida. vários experimentos foram realizados, cada um com uma limitação específica adicionada ao processador na configuração do OrCS e métricas de performance foram coletadas. O programa mostrado no Código 4.9 foi utilizado nos experimentos.

Código 4.9: add_imediate.c

```
1 int main(){
2    int a, b = 0;
3    b = a + 1;
4 }
```

Após sua compilação e geração dos traços do OrCS, esse programa foi simulado e um arquivo pajé para visualização foi gerado. Embora o programa seja simples, existem características inerentes a como os processadores e compiladores funcionam que fazem com que diversas outras instruções precisem ser executadas além do código do programa em si, como, instruções para inicialização e configuração do ambiente de execução do programa e da memória e inicialização de bibliotecas. Isso resulta em traços de visualização muito grandes. No caso do programa do Código 4.9 são 1.792.034 eventos que ocorrem durante 162.018 ciclos do processador.

A observação dessa grande quantidade de eventos é muito demorada, mesmo que o comportamentos das instruções nessa escala seja compreensível isoladamente. É possível acompanhar as instruções e observar seu ciclo de vida no *pipeline*, mas fazer isso para milhares de instruções de maneira a validar o comportamento do visualizador é inviável.

Da mesma maneira, validar a ferramenta utilizando os histogramas de fases é complexo devido ao número de ciclos que o programa simulado pode ter. Um ponto a se destacar é que existem expectativas acerca da ocupação dos *buffers* com base no arquivo de configuração, sendo assim é possível utilizar os histogramas de ocupação para comprovar que a ferramenta muda

seu comportamento conforme alguma modificação na configuração da arquitetura simulada. Se essa mudança na distribuição das ocupações se alinhar com a expectativa, podemos confirmar que a ferramenta distribui os pacotes corretamente nos *buffers*.

Optou-se então por utilizar os histogramas da ocupação das posições dos *buffers* para análise do comportamento. Dessa maneira, seria possível ter uma noção geral de como a ferramenta distribuía os pacotes ao longo do tempo e, consequentemente, do comportamento do programa.

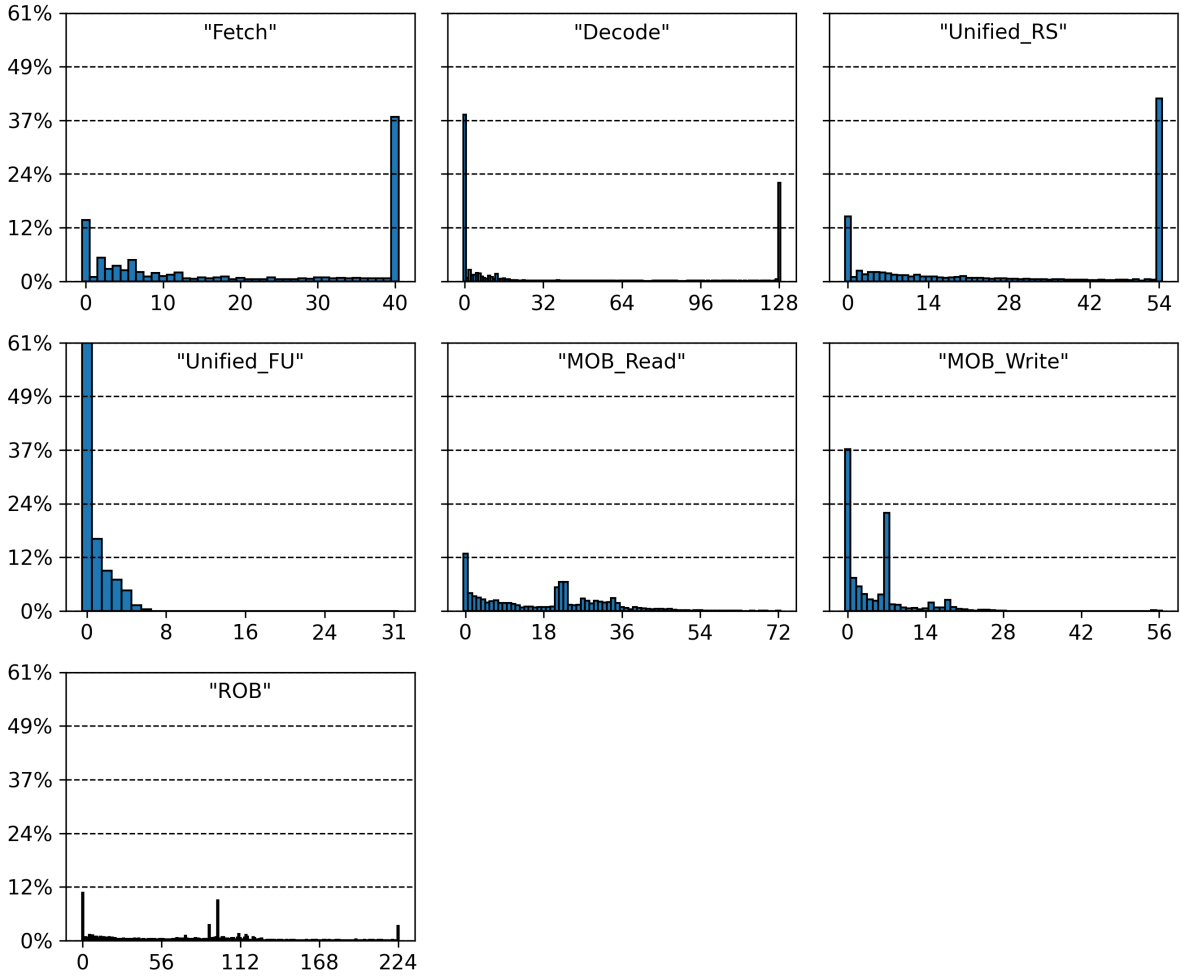


Figura 4.15: Histogramas de ocupação para a execução sem restrições.

A Figura 4.15 apresenta os histogramas de ocupação de cada *buffer* em uma simulação sem restrições, utilizando a **configuração base** do OrCS para a microarquitetura *Skylake Server* da *Intel* (Wikichip, 2017b). No eixo x dos histogramas, está o número de pacotes no *buffer* e, no eixo y, a porcentagem de tempo de execução em que o *buffer* continha essa quantidade de pacotes. Em cada conjunto de histogramas, o tamanho máximo do eixo y foi definido para a porcentagem máxima dentre as ocorrências, que no caso do experimento sem restrições foi de 61% no "Unified_FU".

- O *buffer* "Fetch" tem 40 posições e a largura do estágio associado é 6, ou seja, a cada ciclo, o processador insere 6 instruções no *buffer*. No histograma, ele fica um pouco mais de 12% do tempo vazio e mais de 37% do tempo totalmente preenchido. Os casos quando o *buffer* tem menos de 15 pacotes tendem a ocorrer mais.

- O *buffer* "Decode" tem 128 posições e seu estágio tem largura 5. Ele fica um pouco mais de 37% do tempo vazio e quase 24% do tempo cheio, e os casos onde ele tem menos de 16 pacotes tem maior ocorrência.

- O *buffer* "Unified_RS" tem tamanho 54 e largura 5, referente a largura do estágio *Rename*. O histograma mostra que ele passa um pouco mais de 12% cheio e mais de 37% vazio, além dos casos com menos de 28 elementos no *buffer* terem maior ocorrência.

- O *buffer* "Unified_FU" tem 31 posições, somando o tamanho de todas as unidades funcionais disponíveis e das unidades de *load* e *store*. Sua largura é 6, baseada no estágio "Dispatch". Esse *buffer* fica 61% do tempo vazio e no restante do tempo ele tem ocorrências de 8 ou menos elementos.

- No *buffer* "MOB_Read", nenhuma posição ocupa mais de 12% do tempo e os casos onde o *buffer* tem menos de 36 elementos ocorrem com maior frequência. A largura desse *buffer* é 5, relacionada ao estágio "Rename", e ele tem 72 posições.

- O *buffer* "MOB_Write" tem algumas situações destacadas: ele fica vazio quase 37% do tempo e com 7 elementos por quase 24% do tempo. Além disso, as ocorrências com 6 ou menos elementos se repetem mais. O "MOB_Write"tem 56 posições e largura igual a 5.

- O "ROB" tem largura 8, com base no estágio "Execute", e tamanho 224. Ele fica quase 12% do tempo vazio, e menos de 6% do tempo cheio. Além desses existem mais dois picos de ocorrências em 96 e 90 elementos. As ocorrências de menos de 112 elementos acontecem com maior frequência.

As subseções a seguir descrevem cada restrição colocada na configuração para os estudos e seus efeitos nas execuções através da apresentação dos histogramas de ocupação dos *buffers*.

4.5.1 Estudo de Caso 1: Restrição no *Fetch*

A primeira limitação foi diminuir o tamanho do *buffer* e largura do estágio "Fetch" para 1, além de aumentar sua latência de 3 ciclos para 6 ciclos. Isso gerou um traço de visualização com 2.436.195 eventos e 804.967 ciclos, um aumento de quase 40% eventos e quase 5 vezes mais ciclos em relação ao experimento sem restrições. A Figura 4.16 mostra os histogramas para essa configuração. O eixo y nesses histogramas vai até 96%.

- O histograma do *buffer* "Fetch" tem apenas 2 posições: uma para as ocorrências em que o *buffer* está vazio e outra para quando contém apenas 1 pacote. O histograma mostra que o *buffer* passa cerca de 96% do tempo cheio, o que era esperado devido à redução do seu tamanho e à largura do estágio.

- O *buffer* "Decode" passa quase 58% do tempo vazio e menos de 1% do tempo cheio, o que nem chega a ser visível na imagem. A distribuição nas demais posições se concentra em poucas ocorrências. Isso ocorre devido à redução de instruções buscadas e aos momentos em que o *buffer* fica vazio enquanto uma instrução do "Fetch" é buscada na memória principal.

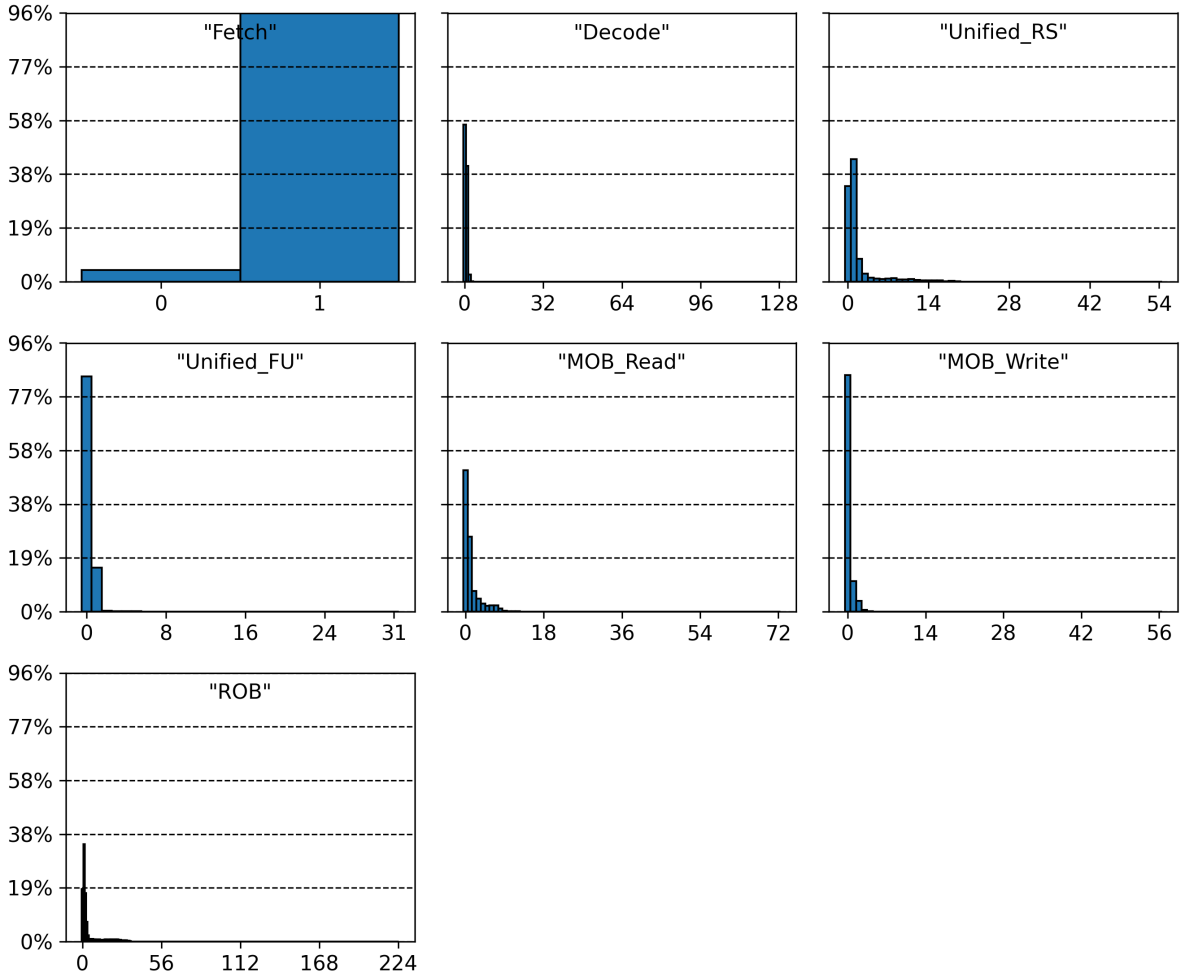


Figura 4.16: Histogramas de ocupação para a execução com "Fetch" de tamanho 1 e estágio com largura 1.

- O histograma do *buffer* "Unified_RS" mostra que ele passa cerca de 34% do tempo vazio e 44% do tempo com apenas 1 pacote. O restante do tempo é distribuído entre menos de 28 posições preenchidas. A concentração de casos com 1 elemento ocorre porque as instruções ficam no *buffer* aguardando que seus operandos fiquem prontos e devido à redução no "Fetch", menos elementos são inseridos nesse *buffer*.

- O *buffer* "Unified_FU" tem 31 posições. Ele passa cerca de 84% do tempo vazio e 16% do tempo com apenas um pacote. As demais posições raramente são ocupadas, pois, devido à restrição, menos instruções são executadas simultaneamente.

- O *buffer* "MOB_Read" passa a maior parte do tempo, cerca de 50%, vazio e cerca de 27% do tempo com apenas um pacote. Existem ocorrências de até 15 posições preenchidas também, mesmo que menos frequentes.

- O "MOB_Write" passa quase 85% do tempo vazio. Cerca de 11% do tempo ele fica com uma posição preenchida, e existem ocorrências de até 10 posições preenchidas, embora bem menos frequentes.

- O "ROB" passa quase 19% do tempo vazio e cerca de 35% do tempo com apenas um pacote. O restante do tempo é distribuído entre ocorrências de até 34 posições preenchidas.

Todos os *buffers* acumularam mais ocorrências de zero ou um elementos preenchidos e nunca ficaram completamente cheios, o que era esperado ao se limitar o "Fetch".

4.5.2 Estudo de Caso 2: Restrição no *Decode*

A restrição no "Decode" consistiu na diminuição de seu tamanho para 8, da largura do estágio associado para 1, e no aumento da latência de *Decode* de 3 para 6. O traço de visualização resultante ficou com 1.822.586 eventos e 191.998 ciclos, um aumento de apenas um pouco mais de 1% para os eventos e de 18% para os ciclos. É possível ver os histogramas na Figura 4.17, o eixo y vai até 51%.

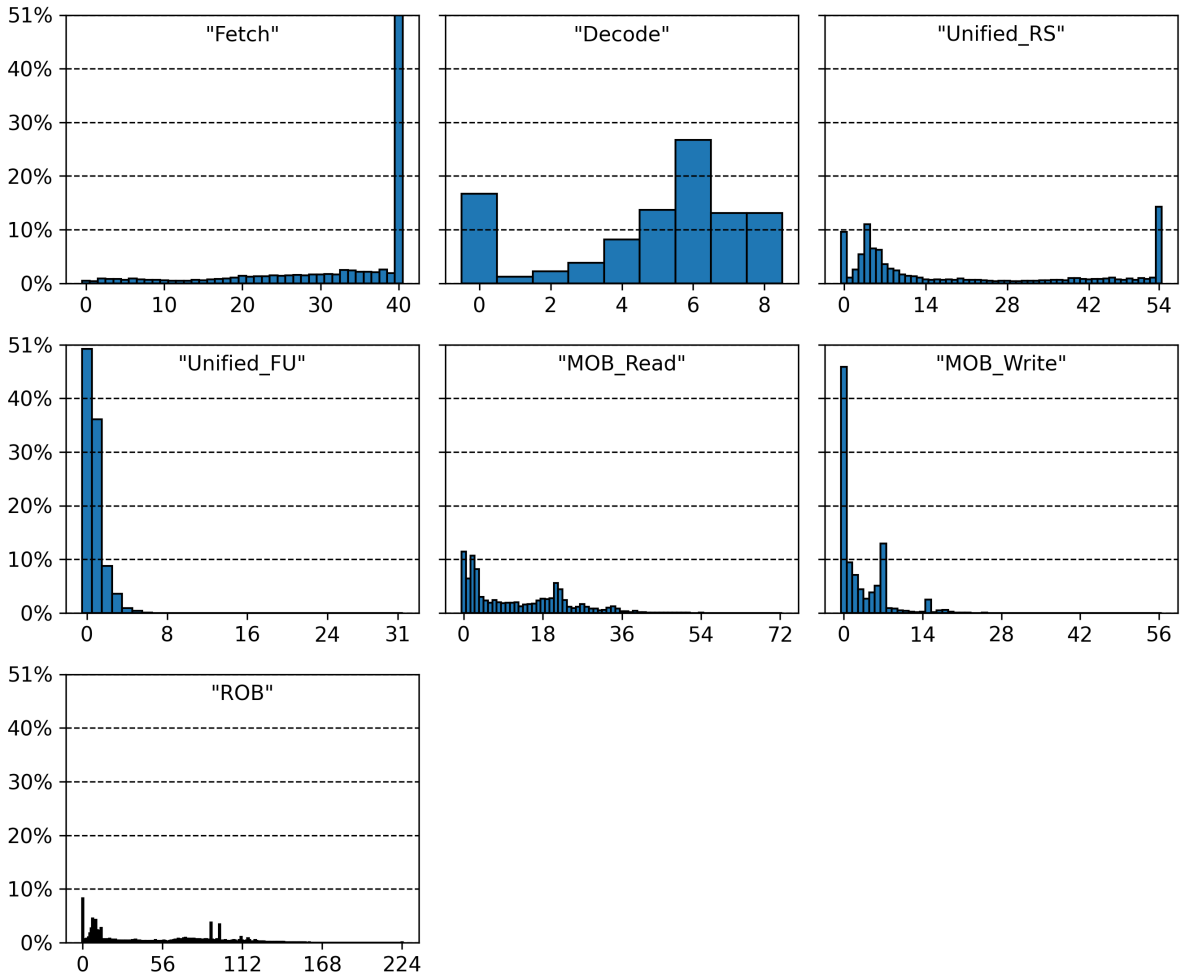


Figura 4.17: Histogramas de ocupação para a execução com "Decode" tamanho 8, e com a largura do estágio associado igual a 1.

- O *buffer* "Fetch" passa 51% do tempo com todas as suas posições ocupadas. As demais ocorrências têm uma quantidade de tempo similar, de até cerca de 2% cada.
- O *buffer* "Decode" tem 8 posições, devido a limitação. Ele passa um pouco mais que 15% do tempo vazio e um pouco mais que 10% do tempo com as 8 posições preenchidas. Das ocorrências possíveis restantes a que se destaca é quando o *buffer* contém 6 pacotes, que acontece cerca de 25% do tempo.

- O *buffer* "Unified_RS" passa cerca de 15% do tempo de com todas as suas posições preenchidas e quase 10% do tempo vazio. As ocorrências com menos de 14 elementos acontecem mais frequentemente.

- O *buffer* "Unified_FU" passa quase 51% do tempo vazio. Na quantidade de tempo restante ele fica com 1 à 3 pacotes na maioria das ocorrências.

- O *buffer* "MOB_Read" tem elementos na maior parte do tempo, ficando apenas um pouco mais de 10% do tempo vazio. As ocorrências que mais se repetem são as que o *buffer* tem menos de 36 pacotes.

- Já o "MOB_Write" fica cerca de 47% do tempo vazio e no restante do tempo as ocorrências que mais se repetem são as com 7 ou menos pacotes.

- O "ROB" fica poco tempo vazio, cerca de 8%. As ocorrências mais comuns são as com menos de 112 pacotes.

A diferença na execução dessa configuração para a base foi pequena. Isso se deu porque tão logo uma posição era liberada na "Unified_RS", algumas μops que saíam do *buffer* "Decode" iam para ela, que também estava a todo ciclo despachando instruções para execução.

Quando limitações na largura foram feitas, um comportamento interessante foi observado. Quando o tamanho selecionado para o *Decode Buffer* era menor do que o maior número de μops que uma instrução poderia gerar, a simulação do OrCS não parava, pois o OrCS nunca conseguia inserir as novas μops no *buffer*. Isso fazia com que apenas os ciclos passassem e o arquivo de visualização crescesse até um tamanho que esgotava a memória do sistema.

4.5.3 Estudo de Caso 3: Restrição na UFU e na URS

A seguir foi criado uma configuração com apenas uma unidade funcional de cada tipo, uma "Unified_RS" de tamanho 13 e com a largura do estágio "Dispatch" igual a 1. O traço de visualização ficou com 1.900.142 eventos distribuídos em 269.319 ciclos, um aumento de 6% nos eventos e de 66% nos ciclos. Os histogramas são mostrados na Figura 4.18, o eixo y vai até 92%.

- O *buffer* "Fetch" passa quase 65% do tempo com suas 40 posições preenchidas, e quase 2% do tempo vazio. O restante do tempo é distribuído entre as outras ocorrências, com uma repetição maior de casos com mais de 25 pacotes.

- O *buffer* "Decode" passa um pouco mais que 74% do tempo cheio, já que por causa da largura 1 do estágio *Dispatch*, apenas 1 μop saía da "Unified_RS" por ciclo, o que fez com que os pacotes ficassem mais no *buffer* "Decode" quando "Unified_RS" ficava cheio, além de fazer com que "Fetch" ficasse cheio quando "Decode" ficava.

- O *buffer* "Unified_RS" tem 13 posições, como foi configurado. Ele passa 92% do tempo com todas as posições preenchidas e quase 4% do tempo vazio. Nenhuma das outras ocorrências acontece muito mais que 0,5% do tempo.

- O *buffer* "Unified_FU" tem 15 posições, pois as unidades repetidas foram removidas. Ele passa quase metade do tempo sem nenhum elemento e o restante com 1 à 4 pacotes. As ocorrências de 1 e 2 pacotes acontecem com mais frequência, a primeira 26% do tempo e a segunda 23%.

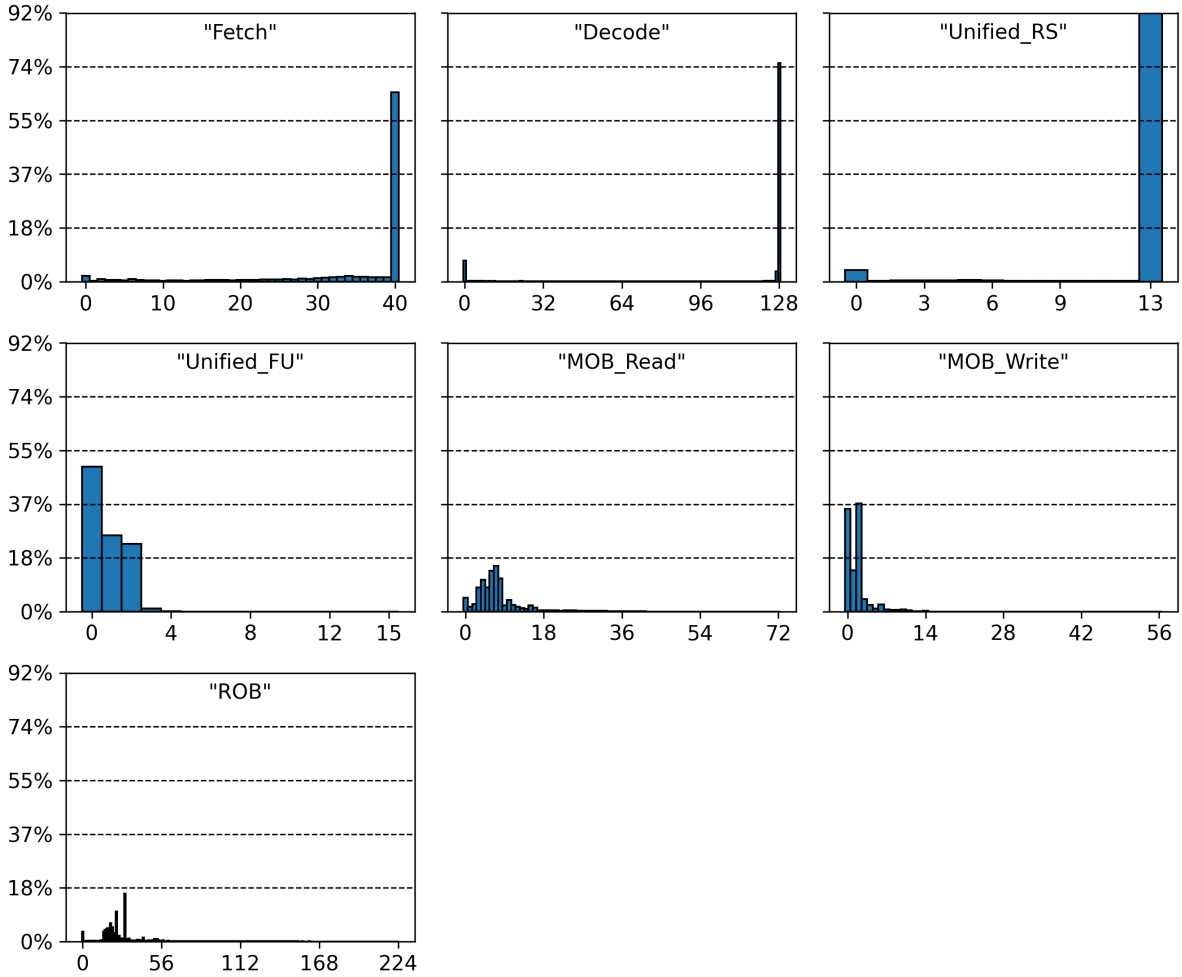


Figura 4.18: Histogramas de ocupação para a execução com "Unified_RS" de tamanho 13, estágio *Dispatch* com largura 1 e Unidades Funcionais reduzidas.

- O *buffer* "MOB_Read" fica cerca de 4% do tempo vazio e ao longo da execução o *buffer* fica com no máximo 54 das 72 posições preenchidas. As ocorrências mais frequentes são as com menos de 18 posições preenchidas, com as maiores ocorrências em torno da com 7 pacotes, que acontece um pouco mais de 15% do tempo.
- O *buffer* "MOB_Write" passa cerca de 35% do tempo sem nenhum elemento, e a maioria do restante do tempo, cerca de 37%, com 2 elementos. Para essa configuração 39 das 56 posições são ocupadas durante a execução.
- O "ROB" fica pouco tempo vazio, cerca de 3%. A ocorrência que se destaca é a de 30 pacotes no *buffer*, que acontece em cerca 16% do tempo. O restante do tempo é distribuído entre ocorrências de até 214 elementos.

O *buffer* "Unified_FU" fica menos tempo vazio do que na execução sem restrições, uma vez que como as unidades funcionais repetidas foram removidas, foram necessários mais ciclos para executar as instruções. Todos os *buffers* do *Backend*, a não ser o "Unified_RS" não ficam completamente preenchidos. A distribuição de tempo nas ocorrências tendem a se concentrar em casos com menos pacotes, devido a diminuição de pacotes sendo despachados por ciclo e da falta de unidades funcionais para execução simultânea de de instruções do mesmo tipo.

4.5.4 Estudo de Caso 4: Restrição no ROB

A restrição aplicada no "ROB" impactou todos os *buffers*. Nesse teste o tamanho foi diminuído para 28 e a latência do estágio *Commit* foi aumentada de 1 para 2. Isso gerou um traço de visualização com 1.882.204 eventos e 251.586 ciclos, um aumento de 5% nos eventos e de 55% nos ciclos. Os histogramas da Figura 4.19 mostram o resultado, o eixo y vai até 85% para eles.

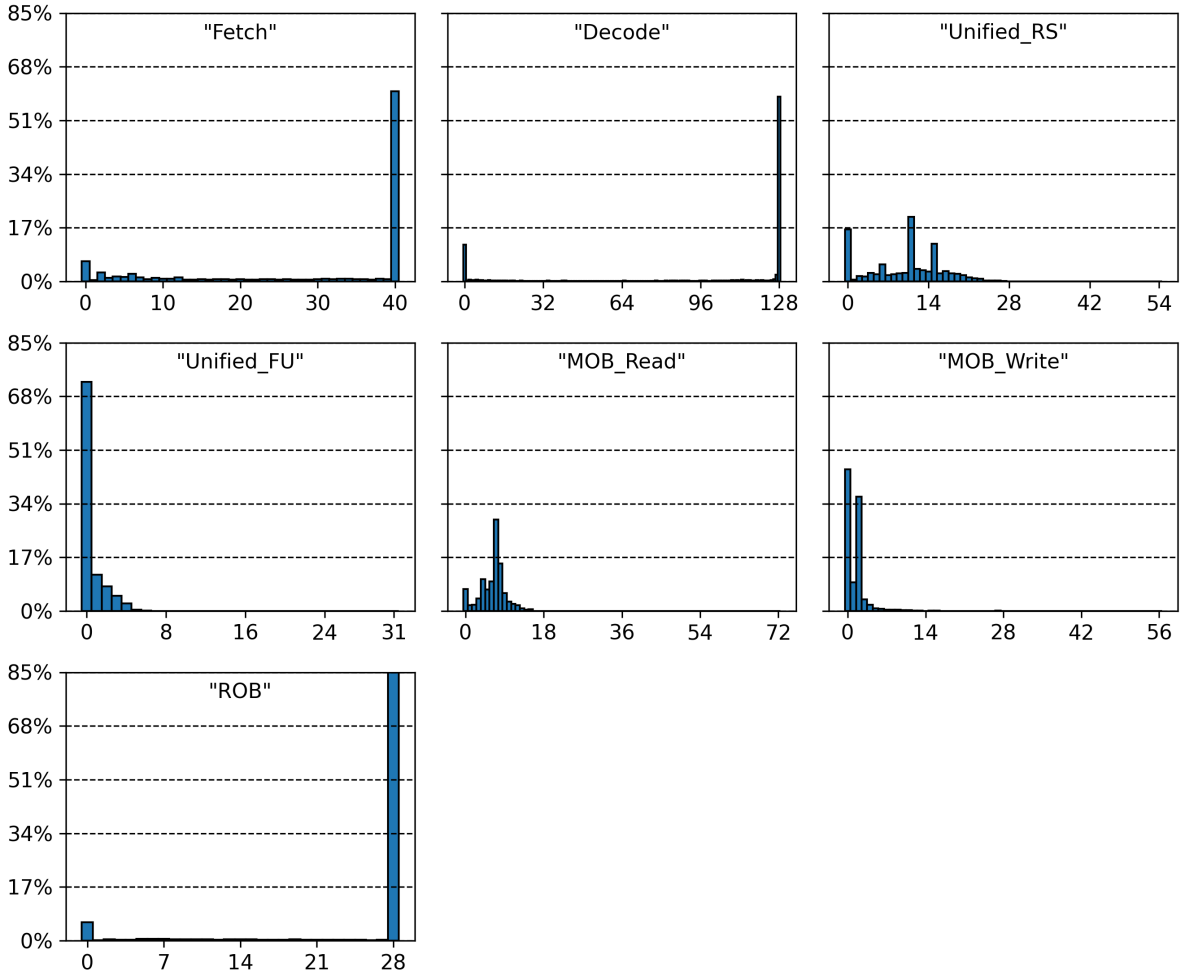


Figura 4.19: Histogramas de ocupação para a execução com "ROB" de tamanho 28 e estágio *Commit* com largura 1.

- O *buffer* "Fetch" passa cerca de 60% do tempo com pacotes em todas as suas posições e 6% do tempo vazio, as ocorrências com 12 ou menos elementos se repetem mais.
- O *buffer* "Decode" fica mais da metade do tempo, cerca de 55% com suas 128 posições preenchidas e um pouco mais de 11% do tempo vazio. Das ocorrências restantes, a única com mais de 0,6% é a com 127 pacotes, que acontece 2% do tempo.
- O *buffer* "Unified_RS" fica pouco tempo vazio, cerca de 17%. As ocorrências de 11 e 15 pacotes no *buffer* se destacam, acontecendo 20% e 11% do tempo, respectivamente. O restante do tempo é distribuído em ocorrências com até 27 pacotes.

- O *buffer* "Unified_FU" passa 68% do tempo sem nenhum elemento e o restante do tempo com ocorrências de até 8 posições preenchidas, todas com menos de 12%.
- O "MOB_Read" fica quase 7% do tempo vazio e a ocorrência mais frequente é a de 7 pacotes no *buffer*, que acontece 29% do tempo. O tempo restante é distribuído entre ocorrências de até 22 posições preenchidas.
- O "MOB_Write" fica quase 45% do tempo vazio, sua ocorrência mais frequente. A segunda ocorrência que se destaca é a de 2 pacotes no *buffer*, que acontece 36% do tempo. Existem ocorrências com até 28 posições preenchidas.
- O "ROB" tem 28 posições e passa a maior parte do tempo, 85%, com todas elas preenchidas. Ele fica um pouco mais que 5% do tempo vazio e as outras ocorrências não passam de 0,6%.

Os efeitos da limitação do "ROB" nos *buffers*, são similares aos da limitação no "Unified_RS". No *Frontend*, as instruções ficam mais ciclos no "Fetch" e as μops mais tempo no "Decode", uma vez que como o "ROB" fica cheio mais rápido, o processador precisa esperar que suas posições sejam liberadas para passar as μops para "Unified_RS" e daí despachá-las para execução. No *Backend*, o *buffer* "Unified_RS" nunca fica cheio, ele fica mais tempo vazio e tem mais ciclos com poucos elementos. Isso se deve à limitação no "ROB", que diminui o número de pacotes que entra no "Unified_RS" e causa mais ciclos onde ele fica vazio, esperando operações serem retiradas do "ROB" para que novas possam ser adicionadas.

4.5.5 Estudo de Caso 5: ROB de tamanho 1

A sensibilidade do pipeline ao tamanho do "ROB" pode ser observado novamente em outro teste executado, cujo tamanho do "ROB" foi modificado para 1 além do aumento da latência de *Commit* de 1 para 2. O traço de visualização ficou com 2.722.869 eventos executados em 1.091.626 ciclos do processador, um aumento de quase 52% no número de eventos e cerca de 6,7 vezes mais ciclos. Os efeitos podem ser vistos nos histogramas na Figura 4.20. O eixo vertical vai até 100%.

- O *buffer* "Fetch" passa 97% do tempo com pacotes em todas as suas posições.
- O *buffer* "Decode" passa 94% do tempo com todas as suas 128 posições preenchidas.
- O *buffer* "Unified_RS" passa 54% vazia e 46% do tempo com um pacote.
- O *buffer* "Unified_FU" passa a maior parte do tempo, 88%, sem nenhuma pacote e o restante do tempo com apenas 1 posição preenchida.
- O *buffer* "MOB_Read" passa a 54% do tempo com vazio e 46% do tempo com apenas 1 pacote.
- O *buffer* "MOB_Write" passa 95% do tempo vazio e o restante com 1 pacote.
- O "ROB" tem apenas 1 posição, e passa 99% do tempo com ela preenchida.

Este teste mostra que ao diminuir para 1 o tamanho do "ROB" o paralelismo do *Backend* do processador é removido. As μops se acumulam no "Decode", são adicionadas à "Unified_RS" e despachadas para execução uma a uma, em ordem, de maneira que o pipeline trava para esperar leituras e escritas. Nenhum *buffer* a partir de "Unified_RS" usa mais de 1 posição.

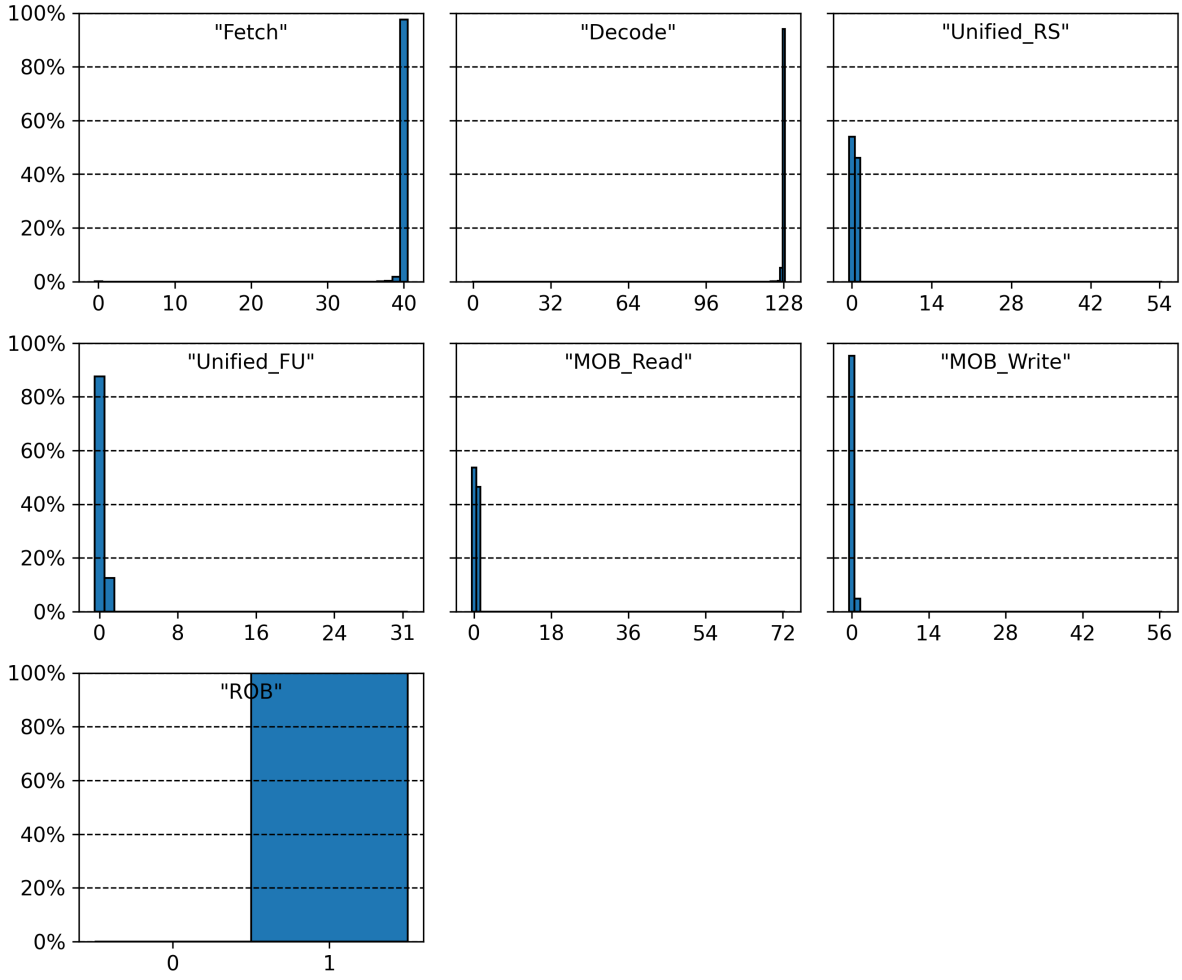


Figura 4.20: Histogramas de ocupação para a execução com "ROB"de tamanho 1.

4.5.6 Caso de Estudo 6: Restrição na Memória

Por último, era interessante aplicar restrições na memória, uma vez que mesmo não observando a memória diretamente, é possível observar através da troca de status dos pacotes que o processador está esperando uma operação de memória acabar, além de acompanhar o conteúdo dos *buffers* de leitura e escrita. A restrição consistiu em diminuir as especificações da cache e da memória principal em cerca de 90%, e quadruplicar as latências delas. Isso resultou em um traço com 1.976.896 eventos e 347.170 ciclos, um aumento de 10% no número de eventos e 2,14 vezes mais ciclos. Os histogramas de ocupação se encontram na Figura 4.21.

- O *buffer* "Fetch" fica 0,8% do tempo vazio e 42% do tempo totalmente preenchido. No tempo restante, as ocorrências com 5 à 30 pacotes acontecem mais vezes.
- O *buffer* "Decode" fica um pouco mais de 34% do tempo vazio e quase 42% do tempo cheio. As outras ocorrências acontecem menos de 0,6% do tempo cada.
- O *buffer* "Unified_RS" passa um pouco mais de 16% vazio e quase 55% do tempo com todas as posições preenchidas. Os casos com menos de 21 elementos no *buffer* ocorrem mais frequentemente.
- O *buffer* "Unified_FU" fica 79% do tempo vazio e no restante do tempo ele tem ocorrências de 8 ou menos elementos.

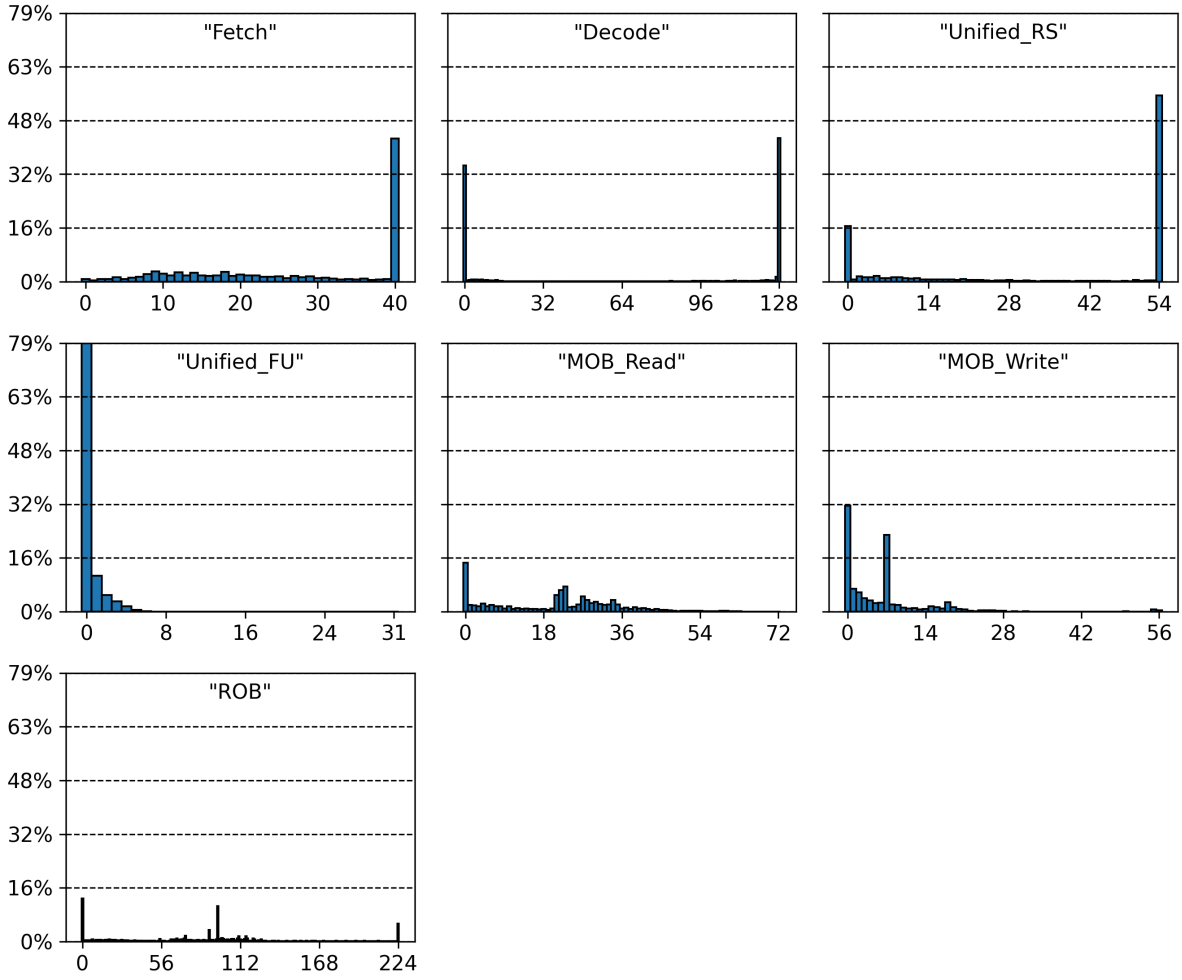


Figura 4.21: Histogramas de ocupação para a execução com restrições na Memória.

- No *buffer* "MOB_Read", nenhuma posição ocupa mais de 16% do tempo e os casos onde o *buffer* tem entre 20 e 36 elementos ocorrem com maior frequência.
- O *buffer* "MOB_Write" tem algumas situações destacadas: ele fica vazio quase 32% do tempo e com 7 elementos por quase 22% do tempo. Além disso, as ocorrências com 20 ou menos elementos se repetem mais.
- O "ROB" fica 12,7% do tempo vazio, e menos de 5% do tempo cheio. Além desses existem mais dois picos de ocorrências em 96 e 90 elementos.

A distribuição das ocorrências foi similar ao da configuração base, mas com algumas diferenças: o "Fetch" fica bem menos tempo vazio, já que as requisições de instruções para a memória principal fazem os pacotes ficarem mais tempo no *buffer*. O "Decode" passa mais tempo cheio do que vazio, devido ao *pipeline* ficar travado esperando requisições de memória terminarem por mais ciclos.

A próxima subseção analisa a performance do simulador, a partir de métricas coletadas durante a execução dos experimentos apresentados.

4.5.7 Performance nos estudos de caso

Após a observação do comportamento dos *buffers* para diferentes configurações, foi realizado uma análise da performance da ferramenta. Como a atividade principal do usuário é a exploração do traço de visualização, a quantidade média de **eventos por segundo** e **ciclos por segundo** que a ferramenta consegue tratar foram escolhidas como métricas.

As métricas foram calculadas usando a função **perf_counter** da biblioteca **time** de *Python*, que retorna uma referência de tempo com a qual é possível calcular uma duração utilizando uma segunda chamada para a função. Assim, no inicio do experimento era marcado o tempo inicial, durante a execução a quantidade dos ciclos e eventos processados eram anotadas e no final a duração do experimento era calculada pela subtraindo o tempo final do tempo inicial marcado. Como o resultado desse calculo era em segundos, para calcular as métricas desejadas, bastava dividir a quantidade de eventos e ciclos pelo valor da duração.

A Figura 4.22 mostra três histogramas, com a duração em segundos e os valores dessas duas métricas para cada uma das execuções mencionadas anteriormente.

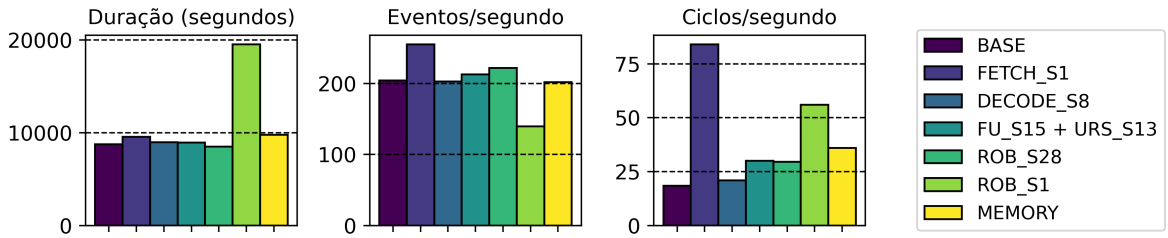


Figura 4.22: Histogramas de performance para execuções dos estudos de caso.

- O histograma **"Duração"** mostra o que já tinha sido observado durante os experimentos, a execução com "ROB" de tamanho 1 tem uma duração muito maior e as outras são parecidas. Isso se dá pela natureza do comportamento da microarquitetura: mesmo com uma restrição no "Fetch", todo ciclo uma nova instrução é inserida e começa a avançar pelo *pipeline*. Na hora da execução, uma instrução que faz um acesso à memória não trava a execução das instruções subsequentes, uma vez que ainda há espaço no "ROB". Já na execução com "ROB" de tamanho 1, a restrição faz com que quando existe alguma instrução que acessa a memória, o *pipeline* fica parado, esperando que a instrução seja concluída.

- **"Eventos/segundo"** mostra que o experimento com "Fetch" tamanho 1 é o mais rápido e o com "ROB" de tamanho 1 é o mais lento. Nesses dois experimentos, o traço de visualização tem mais eventos de *Clock* que os outros, mas no caso do "Fetch" a restrição faz com que poucos elementos estejam nos *buffers* e assim a atualização da representação gráfica é mais simples, fazendo com que os eventos novos de *Clock* sejam tratados rapidamente, aumentando o número de eventos por segundos que a ferramenta faz. Já no experimento do "ROB", os *buffers* do *Frontend* ficam cheios, tornando o tratamento desses eventos de *Clock* mais lentos por causa da renderização dos pacotes, diminuindo a quantidade de eventos por segundo.

- No histograma **"Ciclos/segundo"** podemos ver que os experimentos do "Fetch" e do "ROB" são os maiores. Isso se dá justamente pelo que foi explicado no ponto anterior: esses traços têm mais eventos de ciclo, por isso são tratados mais ciclos por segundo,

mas a diferença na ocupação do *pipeline* faz com que o experimento do "ROB" execute menos ciclos por segundo.

Esses dados foram calculados a partir da execução dos estudos de caso, em modo contínuo, para uma execução completa. Em uma sessão comum de uso da ferramenta, o usuário pode escolher uma combinação de comandos avançar em diferentes modos. Outro ponto a se considerar é que uma sessão pode ser configurada para renderizar histograma ou não. Para demonstrar a diferença da performance da ferramenta durante a exploração do traço nessas diferentes situações, uma nova abordagem para a coleta dos dados foi feita.

4.5.8 Performance dos modos de execução

A métrica escolhida para essa segunda análise foi *eventos por segundo*. Foram executados avanços de tamanhos diferentes e o valor da métrica para cada um foi registrado, sempre reiniciando o traço antes de um novo avanço. Para o modo de execução contínuo, devido a presença do intervalo entre o tratamento dos eventos, existe um limitador natural para essa métrica, 1000 eventos por segundo, no caso da seleção do menor intervalo, 1 milissegundo.

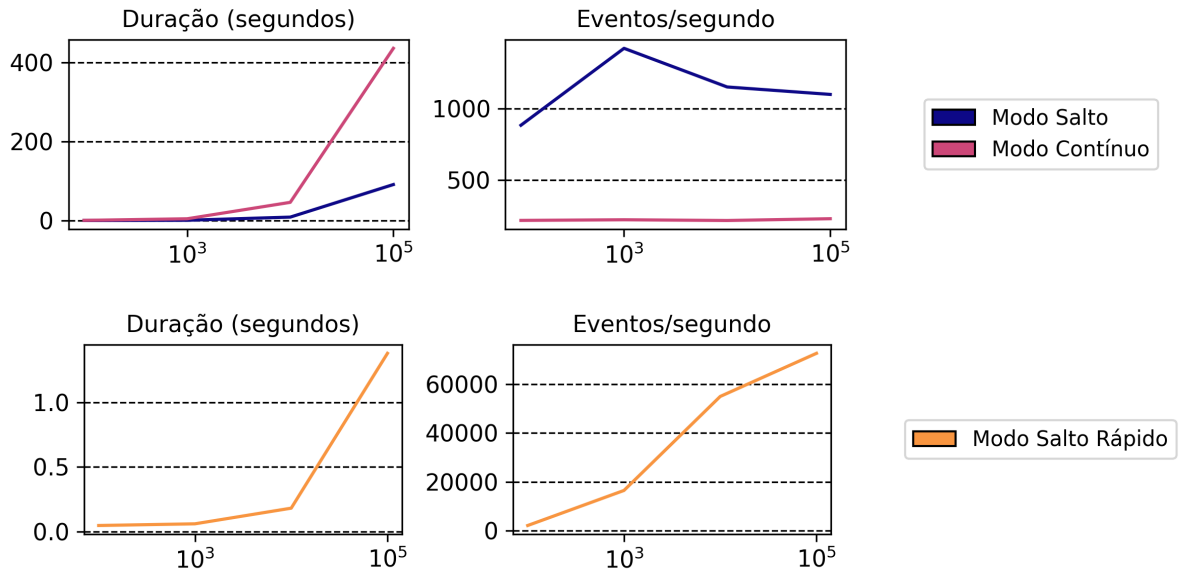


Figura 4.23: Performance para o comando avançar em sessão sem renderização de histograma.

Os valores da métrica e a duração dos avanços para os 3 modos são mostrados na Figura 4.23. O modo salto rápido foi colocado em um gráfico separado devido a diferença de magnitude dos valores dele e dos outros modos. As 3 linhas foram criadas utilizando os valores da métrica para avanços de 100, 1.000, 10.000 e 100.000 eventos.

A Figura 4.24 mostra também um gráfico com o valor da métrica para os diferentes tamanhos de avanço, mas para execuções onde o histograma estava ativado.

- O **Modo Contínuo** tem o menor número de eventos por segundo em todos os avanços, com ou sem histograma. O tempo de processamento aumenta significativamente à medida que o tamanho do avanço aumenta também.
- O **Modo Salto** tem um desempenho melhor do que o Modo Contínuo em termos de eventos por segundo. O tempo de processamento ainda aumenta com o avanço, mas

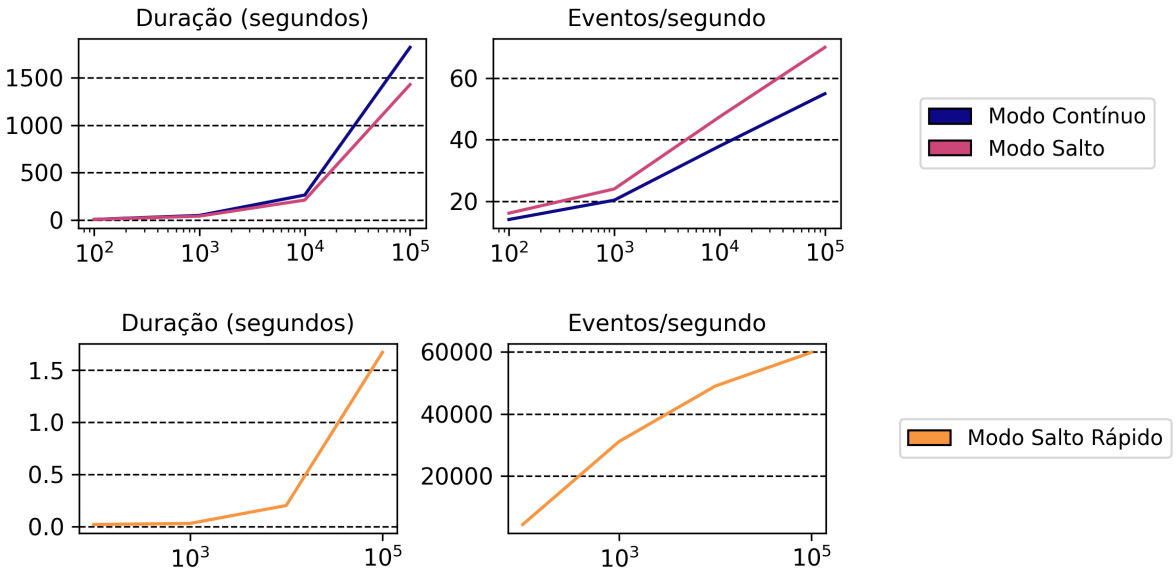


Figura 4.24: Performance para o comando avançar em sessão sem renderização de histograma.

a uma taxa menor do que no Modo Contínuo, tanto para a sessão com o histograma quanto sem.

- O **Modo Salto Rápido** é o modo de melhor desempenho. Ele tem o maior número de eventos por segundo. O tempo de processamento aumenta com o tamanho do avanço, mas a uma taxa muito menor do que nos outros dois modos.

Com a análise foi possível observar que a performance varia de acordo com o tamanho do salto e da quantidade de elementos no pipeline que precisam ser redesenhados cada vez que um evento é processado.

Considerando uma simulação em um computador executando à 3GHz, ou seja, fazendo 3 bilhões de ciclos por segundo, 1 segundo de simulação pode resultar em 3 bilhões de instruções sendo executadas. Isso pode gerar dezenas de bilhões de eventos pajé, o que mostra que a visualização de um programa real, que pode durar muito mais que alguns segundos, é inviável, uma vez que mesmo utilizando o modo de execução mais rápido, seriam horas e horas de duração. Isso expõe que o uso da ferramenta é mais adequado para a visualização de sequencias curtas de execuções, utilizando o OrCS para encontrar e selecionar o que se quer visualizar.

4.5.9 Conclusões sobre os experimentos

Os testes executados comprovaram a capacidade da ferramenta de trazer mais entendimento para o funcionamento do simulador. Inclusive, as diferenças entre a execução base e as execuções com restrições se alinharam com o esperado.

Com a ferramenta, é possível observar as instruções sendo buscadas, algumas ficando com status de pesquisa na memória principal e depois várias em sequência ficando prontas para serem decodificadas, o que evidencia a pesquisa das instruções na memória e atualização da memoria cache. Após isso, é possível identificar as instruções sendo decodificadas em diversas μops que são adicionadas ao "Decode", para então serem adicionadas ao "ROB" e a "Unified_RS"para que possam ser despachadas pra execução, sendo adicionadas na "Unified_FU" e nos *buffers* de memória, e enfim, após a execução das operações ou finalização das operações de memória, as micro-operações ficarem com status de prontas no "ROB", e serem removidas

dele em ordem. Cada restrição feita mudou o comportamento, mas mantendo esse fluxo básico, com algum atraso ou limitação.

Durante a execução, um pacote com identificador "0" era referenciado pelos eventos, diversas vezes. Esta situação era estranha, uma vez que nenhuma das instâncias de *opcode_package_t* ou *uop_package_t* poderiam ter *opcode_number* ou*uop_number* igual à "0" no OrCS. Por ser um erro que não influenciava o cálculo das métricas do *orcs*, optou-se por ignorá-lo, capturando-se a exceção causada pelo acesso ao pacote inexistente e seguindo com o desenvolvimento. No entanto, isso expôs a capacidade do visualizador em identificar um erro que passava despercebido, o que evidenciou a capacidade de depuração do visualizador.

5 CONCLUSÕES

Essa monografia teve como objetivo a criação de uma ferramenta de visualização para o estado das estruturas de um simulador, com o intuito de verificar se essa abordagem auxiliaria o usuário na compreensão do comportamento e estado do sistema real simulado[1].

O processo de desenvolvimento da ferramenta resultou na instrumentação do simulador OrCS para a criação de um arquivo de entrada para o programa de visualização desenvolvido. Esse programa implementa diversas funcionalidades para exploração do arquivo de entrada, e por consequência, da simulação em si.

A ferramenta permite a observação de um conjunto de estruturas de dados do simulador e possibilita o monitoramento de inserções, remoções e atualizações de elementos nessas estruturas. O estudo de casos realizado comprovou sua capacidade de expôr o comportamento do sistema simulado, mas também expôs sua limitação em relação à velocidade de execução. O uso da ferramenta deve se aliar com o uso do *orcs* para selecionar o intervalo que se quer visualizar, de maneira que não seja necessário visualizar a execução completa de um programa muito grande.

A ferramenta se mostrou útil também para a depuração do simulador, pois ela permite uma identificação mais rápida e precisa de erros e comportamentos anômalos. Isso, por sua vez, melhorou a eficiência do simulador, ao auxiliar a depuração e a validação dos resultados obtidos com ele.

Além disso, como a representação gráfica é maleável e o arquivo de entrada possui definições próprias, a ferramenta é desacoplada do simulador, ou seja, é possível utilizá-la para a visualização do estado de outros simuladores, com diferentes arquiteturas inclusive. Basta que o usuário crie um arquivo de entrada coerente. Para qualquer simulador que se tivesse acesso ao código fonte, é possível a instrumentação, para geração dos eventos de inserção, atualização, remoção e *clock*. Seria necessário apenas que o usuário passasse pelo processo de escolher quais componentes gostaria de observar, e talvez seria necessário a adaptação do tratamento de diferentes tipos de pacotes e a seleção de informações extras adequadas.

Inclusive, a escolha de se afastar de uma representação gráfica muito próxima de sinais e barramentos, concede uma capacidade extra à ferramenta desenvolvida nesse trabalho. Com adaptações relacionadas à obrigatoriedade de eventos de *clock*, ela pode se tornar genérica o suficiente para ser uma ferramenta de visualização para qualquer processo que possa ser abstraído para tabelas (*buffers*) que contenham elementos (pacotes) com um certo *status* e atributos, que sejam inseridos e removidos das tabelas e que sofram alguma atualização. Diversas aplicações de gestão visual poderiam ser criadas, a partir também da criação de um traço de visualização no formato adequado.

[1] As instruções para uso da ferramenta proposta podem ser encontradas no Apêndice B.

6 LIMITAÇÕES E TRABALHOS FUTUROS

Durante o desenvolvimento do trabalho, identificaram-se algumas melhorias para a ferramenta. Essas melhorias têm impacto tanto na usabilidade quanto nos potenciais usos e na generalidade da ferramenta.

6.1 MUDANÇAS NO TRATAMENTO DOS EVENTOS

Uma melhoria útil seria a criação de uma funcionalidade que se manifestasse como uma barra de execução, semelhante àquelas presentes em softwares reprodutores de mídia, proporcionando maior controle e liberdade na visualização.

Com isso, também seria possível incluir a opção de inserir marcadores ao longo da linha do tempo da visualização e movimentar-se entre eles. Para alcançar essa mobilidade, considerando não apenas as funcionalidades em si, mas também uma maior eficiência na execução delas, alguns pontos precisariam ser observados.

6.1.1 Retroceder visualização

Quando a ferramenta é empregada para localizar e compreender o comportamento de alguma execução, é natural que haja o desejo de retroceder e avançar a execução para observar o fluxo dos pacotes dentro dos *buffers*. Em diversas ocasiões, identificou-se essa necessidade de retroceder a execução do programa. Portanto, essa figura como uma das opções para novas funcionalidades.

A funcionalidade de retroceder a execução não operaria da mesma maneira que a funcionalidade de avançar. A funcionalidade de avançar possui um modo contínuo e um modo discreto para progredir, denominado "step". Acompanhar um evento de maneira contínua de trás para frente não possui lógica. Portanto, a funcionalidade de retrocesso operaria em "step", permitindo retroceder de maneira única ou em mais de uma unidade, que podem ser ciclos ou eventos. Pode-se retroceder um único evento, um único ciclo, ou mais de um ciclo ou evento, efetivamente pulando a visualização para um estado anterior.

Existem duas abordagens para a implementação dessa funcionalidade. Na primeira, para cada evento no traço de simulação, seria criado um evento inverso. Por exemplo, para um evento de *clock*, o evento inverso seria também um evento de *clock*, mas para o ciclo anterior. No caso da inserção de um pacote, o evento inverso seria a remoção desse pacote. É importante observar que apenas o identificador do pacote seria necessário para realizar a remoção. Para a remoção de um pacote, o evento inverso consistiria na inserção desse pacote com o estado que ele possuía quando a remoção ocorreu. No caso da atualização de um pacote, o evento inverso seria uma atualização do pacote para o estado anterior ao evento de atualização em questão.

Essa abordagem pode apresentar desafios no gerenciamento de memória caso o tamanho do traço seja consideravelmente grande, uma vez que seria necessário armazenar efetivamente duas vezes o tamanho do traço. Isso decorre do fato de que, para cada evento, haveria um evento inverso correspondente. Além disso, a criação desses eventos inversos pode demandar um tempo considerável, dada a natureza dos eventos de atualização e remoção, os quais dependem de eventos prévios.

Pela natureza do traço, que já pode ser muito grande, o evento prévio do qual o evento inverso depende pode estar muito antes no traço e o processo de percorrer o traço várias vezes procurando esses eventos anteriores pode ser muito custoso.

Isso implicaria em uma inicialização demorada se os eventos inversos forem calculados antes do tratamento dos eventos pela ferramenta, ou implicaria em uma demora extra no tratamento dos eventos se os eventos inversos forem calculados durante a visualização.

Uma abordagem alternativa que se mostra mais eficiente é implementar o retrocesso através de *"checkpoints"*. Eles seriam basicamente uma foto do estado da ferramenta antes de um novo evento ser aplicado. Assim, ao retroceder a visualização de um determinado traço, bastaria restaurar o estado anterior à aplicação desse evento. Os *checkpoints* podem ser criados conforme a ferramenta trata os eventos de maneira muito rápida, já que seria apenas salvar o conteúdo dos *buffers* e o estado dos pacotes antes do evento ser aplicado. Uma troca entre tempo de execução e memória, essencialmente.

A segunda abordagem diminui o tempo de inicialização e evita o gasto de tempo com lógicas para encontrar os eventos inversos. Além disso, é possível definir um limite para o retrocesso, evitando o problema de guardar uma grande quantidade de estados antigos. Por exemplo, seria possível definir que só é possível retroceder 50 eventos ou ciclos, o que iria requerer salvar apenas 50 estados anteriores à aplicação de um certo evento e uma referência a linha do traço de visualização que corresponde a esses eventos para que, uma vez retrocedido a visualização possa ser continuada normalmente.

6.2 OTIMIZAÇÕES NA ESTRUTURA DE DADOS DO TRAÇO

O tamanho do traço de visualização se mostra um problema. Um programa pode ter bilhões ou trilhões de instruções, e cada instrução gera vários eventos visuais, o arquivo de entrada com os eventos pode se tornar maior. Tratar um arquivo de entrada tão grande pode acarretar problemas, como uso excessivo de memória e até dificuldades para abrir e processar o arquivo. Percorrer as linhas desse arquivo também pode ser um trabalho muito lento.

Uma possibilidade para solucionar esse problema seria criar uma árvore binária com apontadores para as linhas do arquivo. Isso possibilitaria mais eficiência ao percorrer o arquivo para se chegar a uma linha específica, seja no final, no início ou no meio do arquivo. Essa mudança na maneira de tratar o arquivo de entrada seria essencial caso fosse implementada a funcionalidade de pular eventos, já que seria necessário percorrer rapidamente o arquivo e continuar a leitura das linhas em outras partes do arquivo.

Não seria necessário indexar todas as linhas do arquivo nessa árvore binária; algumas linhas, espaçadas de uma maneira útil poderiam ser indexadas, criando uma árvore binária com elas. Dessa forma, ao se pesquisar por uma linha específica, seria possível encontrar rapidamente a linha mais próxima na árvore binária e acessá-la, percorrendo linearmente o caminho até a linha desejada. Desse modo, um arquivo com bilhões de linhas poderia ser resumido e acessado de maneira muito mais rápida.

6.3 MELHORIAS PARA O USO DA FERRAMENTA COM O ORCS

Algumas melhorias são específicas para o uso da ferramenta com o simulador OrCS, mas que exigem muito cuidado caso não se queira perder generalidade. Primeiramente, a ferramenta pode ser adaptada para a visualização de versões mais atuais da microarquitetura *Intel*, isso poderia implicar na atualização do próprio OrCS para tal e portanto foge um pouco do escopo do trabalho.

6.3.1 Visualização de registradores e Unidades funcionais

Seria interessante visualizar os registradores da arquitetura, uma vez que várias operações sobre eles, por vezes, não refletem nenhuma mudança nos *buffers* do OrCS.

A visualização dos dados neles poderia ser útil, inclusive, já que tudo isso está codificado no traço de simulação do OrCS. Isso evitaria que algumas instruções passassem desapercebidas na visualização, uma vez que elas resultariam em modificações nos registradores.

6.3.2 Visualização da memória

Dado que o OrCS incorpora muitos componentes do SiNUCA, um simulador focado em simulação de memória e cache, uma melhoria natural seria implementar a visualização da memória na ferramenta. É comum atingir um ponto em que os únicos eventos tratados por ela são eventos de *clock*, e os *buffers* não sofrem nenhuma atualização enquanto o processador aguarda algum resultado da memória.

Poder observar essa dinâmica da memória sendo acessada e a cache sendo atualizada com os valores retirados da memória, se mostra interessante. Além disso, visualizar o funcionamento do acesso aos canais da memória principal seria valioso. O OrCS possui um subsistema de memória amplamente documentado que poderia ser observado com eventos similares aos que existem para os *buffers*, considerando apenas eventos de inserção, remoção e atualização dos pacotes de requisições para memória.

Outra possibilidade seria pensar em como observar até mesmo os dados da memória sendo retirados e populando os registradores.

REFERÊNCIAS

Abel, A. e Reineke, J. (2019). Uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. ASPLOS '19, New York, NY, USA. Association for Computing Machinery.

Abelson, H., Sussman, G. J. e with Julie Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition.

Akram, A. e Sawalha, L. (2016). Œ86 computer architecture simulators: A comparative study. Em *IEEE 34th International Conference on Computer Design (ICCD)*, páginas 638–645.

Akram, A. e Sawalha, L. (2019). A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B. e Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. Em *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, páginas 605–610.

Carvalho, C. (2002). The gap between processor and memory speeds.

da Silva, D. R. (2021). Ábaco: prática pedagógica matemática. *Produções didáticas*, página 23.

dos Santos, S. R. (2013). Visualização de memória utilizando java reflection. Dissertação de Mestrado, Pós-Graduação em Ciência da Computação - Universidade Federal Rural do Semi-Árido, Mossoró - RN.

Durand, D. P. (2006). Xspim / pcspim introduction. `https://www.cs.kent.edu/~durand/CS35101F06/Help/spimintro.html`.

Esmeraldo, G. A. e Lisboa, E. B. (2017). Uma ferramenta para exploração do ensino de organização e arquitetura de computadores. *International Journal of Computer Architecture Education*, 6:68–75.

Evans, D. (2011a). *Introduction to Computing: Explorations in Language, Logic, and Machines. Chapter 3: Programming.*, páginas 35–52. University of Virginia.

Evans, D. (2011b). *Introduction to Computing: Explorations in Language, Logic, and Machines. Chapter 6: Machines.*, páginas 105–123. University of Virginia.

Evaristo, G. (2023). Repositório do orcs architecture visualizer. `https://github.com/GabrielCambe/OrCS-visual`.

Felix, A. F., Pousa, C. V. e Carvalho, M. B. (2006). Dimipss: Um simulador didático e interativo do mips.

Gaddis, T. (2021). *Starting out with Python. Chapter 1: Introduction to Computers and Programming.*, páginas 12–19. Pearson Education.

Ibrahim, R. K., Jumma, L. F., Amory, I. A. e Al-Hilali, A. A. (2021). Design of moesi protocol for multicore processors based on fpga. *International Journal of Nonlinear Analysis and Applications*, 12:1229–1242.

Imai, Y., Kaneko, K. e Nakagawa, M. (2006). A visual computer simulator and its applications to an ict-based higher education. Em *2006 7th International Conference on Information Technology Based Higher Education and Training*, páginas nil13–nil20.

Imai, Y., Kaneko, K. e Nakagawa, M. (2007). A web-based visual simulator with communication support and its application to computer architecture education. Em *Seventh IEEE International Conference on Advanced Learning Technologies (ICALT 2007)*, páginas 147–151.

Jan Friso Groote, Rolf Morel, J. S. e Watkins, A. (2021). *Logic Gates, Circuits, Processors, Compilers and Computers. 1st. ed.* Springer Cham.

Kumar, R. (2005). *Human Computer Interaction*. Firewall Media.

Landers, B. (2017). Rars github repo. `https://github.com/TheThirdOne/rars`.

Levie, W. H. e Lentz, R. (1982). Effects of text illustrations: A review of research. *Educational Communication and Technology*, 30(4):195–232.

Mazza, R. (2009). *Introduction to Information Visualization*.

Mintzer, M. Z. e Snodgrass, J. G. (1999). The picture superiority effect: Support for the distinctiveness model. *The American Journal of Psychology*, 112(1):113–146.

Mohan, D. (2023). A new architecture of risc cum cisc processor architecture. *International Journal of Flexible Manufacturing Systems*, Volume 2:65.

Nowatzki, T., Menon, J., Ho, C.-H. e Sankaralingam, K. (2015). Architectural simulators considered harmful. *IEEE Micro*, 35(6):4–12.

Patterson, D. A. e Hennessy, J. L. (2005). *Computer Organization and Design: The Hardware/-Software Interface. 3rd. ed.* Morgan Kaufmann.

Patterson, D. A. e Hennessy, J. L. (2014). *Computer Organization and Design: The Hardware/-Software Interface. 5th. ed.* Morgan Kaufmann.

Petersen, M. B. (2021a). Ripes: A visual computer architecture simulator. Em *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, páginas 1–8. IEEE.

Petersen, M. B. (2021b). Vsrtl: Visual simulation of register transfer logic. `https://github.com/mortbopet/VSRTL`. Acessado em 12/11/2023.

Ramos, N. A. (2008). *Avaliação e Comparação de Desempenho de Computadores: Metodologia e Estudo de Caso*. Tese de doutorado, UFPE - Universidade Federal de Pernambuco. 60 pgs.

Sarjoughian, H., Chen, Y. e Burger, K. (2008). A component-based visual simulator for mips32 processors. Em *2008 38th Annual Frontiers in Education Conference*, páginas F3B–9–F3B–14.

Schnorr, L. M., Stein, B. O., Faverge, M., Trahay, F. e Kergommeaux, J. C. (2015). PajÉ trace file format, version 1.3.1. Relatório técnico.

Shneiderman, B. e Plaisant, C. (2019). Interactive visual event analytics: Opportunities and challenges. *Computer*, 52(1):27–35.

Tanenbaum, A. S. e Austin, T. (2012). *Structured Computer Organization*. Pearson, 6 edition.

Tissei, B. (2021). Representação de instruções x86 para simuladores com precisão de ciclos.

Ubal, R., Jang, B., Mistry, P., Schaa, D. e Kaeli, D. (2012). Multi2sim: A simulation framework for cpu-gpu computing. Em *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, páginas 335–344.

Ullmann, M., Inocêncio, A. C., Neto, E. M., Freitas, M. e Jr, P. A. P. (2014). Neandersim: Simulador gráfico de apoio ao ensino de arquitetura de computadores. Em *Anais do XXII Workshop sobre Educação em Computação*, páginas 361–370, Porto Alegre, RS, Brasil. SBC.

Vollmar, K. e Sanderson, P. (2006). Mars: An education-oriented mips assembly language simulator. SIGCSE '06, página 239243, New York, NY, USA. Association for Computing Machinery.

Waterman, A., Lee, Y., Patterson, D. A. e Asanovic, K. (2014). The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Relatório Técnico UCB/EECS2014-54, EECS Department, University of California, Berkeley.

Weber, R. F. (2012). *Fundamentos de arquitetura de computadores. 4. ed.* Bookman.

White, K. P. e Ingalls, R. G. (2015). Introduction to simulation. Em *2015 Winter Simulation Conference (WSC)*, páginas 1741–1755.

Wikichip (2017a). Skylake (client) - microarchitectures - intel. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.

Wikichip (2017b). Skylake (server) - microarchitectures - intel. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`.

William Stallings, P. Z. e Jesshope, C. R. (2016). *Computer Organization and Architecture: Designing for Performance*. Pearson Education Limited, 10th edition.

Zeigler, B., Muzy, A. e Kofman, E. (2018). *Theory of Modeling and Simulation: Discrete Event Iterative System Computational Foundations.* Academic Press.

Ziabari, A. K., Ubal, R., Schaa, D. e Kaeli, D. (2015). Visualization of opencl application execution on cpu-gpu systems. Em *Proceedings of the Workshop on Computer Architecture Education*, WCAE '15, New York, NY, USA. Association for Computing Machinery.

APÊNDICE A – ARQUIVO DE ENTRADA PAJÉ RESUMIDO

Código A.1: Arquivo de eventos Pajé para o programa add_imediate.c, resumido até o 5^{o} evento de *clock*

```
%EventDef DefineBufferType 0
% Name string
% Color string
% IsContainer bool
%EndEventDef
%EventDef CreateBuffer 1
% Id int
% Name string
% TypeName string
% Width int
% Size int
%EndEventDef
%EventDef DefinePackageType 2
% Id int
% Name string
%EndEventDef
%EventDef InsertPackage 5
% TypeId int
% Id string
% BufferIds string
% Content string
%EndEventDef
%EventDef RemovePackage 6
% TypeId int
% Id string
% BufferIds string
%EndEventDef
%EventDef UpdatePackage 7
% TypeId int
% Id string
% BufferIds string
% Content string
%EndEventDef
%EventDef Clock 8
% Cycle int
%EndEventDef
%EventDef DefineStatusColor 9
% Status string
% Color string
%EndEventDef
0 "FETCH_BUFFER" "#00000F" False
0 "DECODE_BUFFER" "#00003F" False
0 "UNIFIED_RESERVATION_STATION" "#00005F" True
0 "UNIFIED_FUNCTIONAL_UNITS" "#00007F" True
0 "MEMORY_ORDER_BUFFER_READ" "#00219F" False
0 "MEMORY_ORDER_BUFFER_WRITE" "#0021AF" False
0 "REORDER_BUFFER" "#0000FF" False
2 0 "OperationPackage"
2 1 "UopPackage"
9 "FREE" "#0000FF"
9 "WAIT" "#FF0000"
```

```
9 "READY" "#00FF00"
9 "UNTREATED" "#FFFF00"
9 "DRAM_FETCH" "#FFA500"
1 0 "Fetch" "FETCH_BUFFER" 6 40
1 1 "Decode" "DECODE_BUFFER" 5 128
1 2 "Unified_RS" "UNIFIED_RESERVATION_STATION" 5 54
1 3 "Unified_FU" "UNIFIED_FUNCTIONAL_UNITS" 6 31
1 4 "MOB_Read" "MEMORY_ORDER_BUFFER_READ" 5 72
1 5 "MOB_Write" "MEMORY_ORDER_BUFFER_WRITE" 5 56
1 6 "ROB" "REORDER_BUFFER" 8 224
8 0
5 0 "1" [0,] {"status":"FREE","assembly":"MOV_GPRv_GPRv_89+R64+R64"}
5 0 "2" [0,] {"status":"FREE","assembly":"CALL_NEAR_RELBRd+Rel32"}
5 0 "3" [0,] {"status":"WAIT","assembly":"NOP_GPRv_GPRv_0F1E+R32+R32"}
8 1
8 2
8 3
8 4
8 5
...
```

APÊNDICE B – INSTRUÇÕES PARA A UTILIZAÇÃO DA FERRAMENTA

O código fonte e exemplos de traços de visualização foram salvos em um repositório do *Github* (Evaristo, 2023). Para utilizar ou modificar a ferramenta, considerando que nenhuma configuração prévia exista na máquina, é necessário:

1. Instalação do pacote **git**.
2. Clonagem ou *download* do repositório.
3. Instalação do **python-3.8** e do pacote **pip**.
4. Instalação do pacote **pipenv**.
5. Ativação do ambiente virtual utilizando o comando "pipenv shell".
6. Instalação das bibliotecas de Python necessárias, utilizando o comando "pipenv install".
7. Inicialização da ferramenta, com o comando "python tool.py"para iniciar a ferramenta.