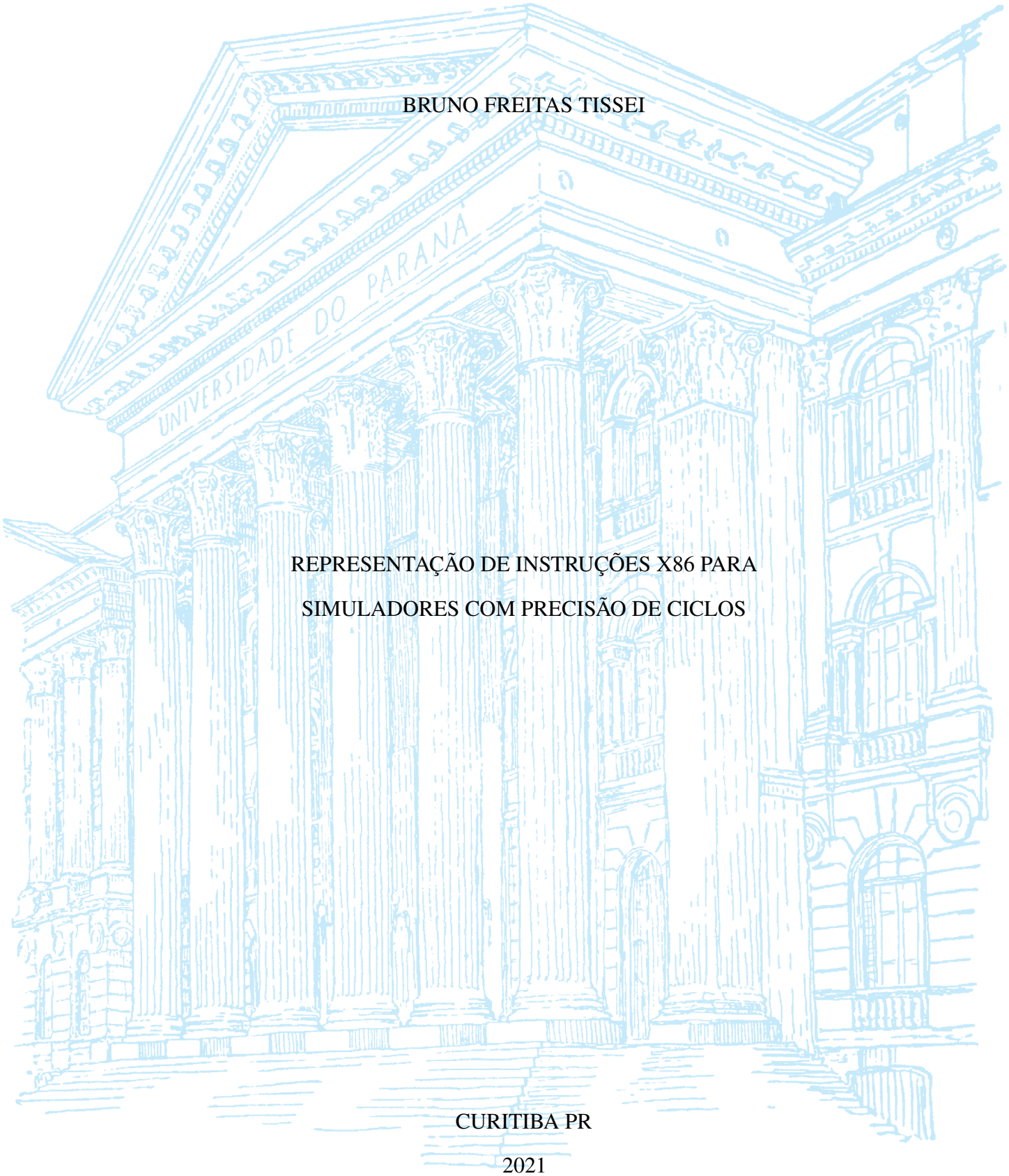
UNIVERSIDADE FEDERAL DO PARANÁ

BRUNO FREITAS TISSEI



REPRESENTAÇÃO DE INSTRUÇÕES X86 PARA SIMULADORES COM PRECISÃO DE CICLOS

CURITIBA PR

2021

BRUNO FREITAS TISSEI

REPRESENTAÇÃO DE INSTRUÇÕES X86 PARA SIMULADORES COM PRECISÃO DE CICLOS

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: *Ciência da Computação*.

Orientador: Prof. Dr. Marco Antonio Zanata Alves.

CURITIBA PR

2021

AGRADECIMENTOS

Agradeço à Universidade Federal do Paraná por todas as oportunidades e pela infraestrutura disponibilizada para a realização do meu curso de graduação. Também agradeço aos membros e colegas do Departamento de Informática, discentes e docentes, em especial ao professor Dr. Marco Antonio Zanata Alves pela orientação ao longo do desenvolvimento desse trabalho.

RESUMO

Pesquisas na área de arquitetura de computadores apresentam grandes dificuldades ao exigirem testes, análises, e validações de ideias. Isso ocorre devido à alta complexidade dos sistemas envolvidos. O uso de simuladores é uma alternativa flexível e barata, em comparação com os protótipos físicos, e precisa em comparação com as modelagens analíticas. Entretanto, é necessário um alto nível de desenvolvimento, modelagem, e validação para se obter um simulador suficientemente preciso. A forma como as instruções são representadas em um simulador de arquitetura é crucial para a precisão da simulação e a capacidade de gerar estatísticas detalhadas. Diversos fatores devem ser levados em consideração para uma representação fiel à arquitetura real, como latências das instruções, quantidades de micro operações, e distribuição de unidades de execução. Na literatura, encontram-se diversas formas de implementar uma *Instruction Set Architecture* do tipo CISC em um simulador. Porém, devido à falta de informações por motivos de proteção de propriedade intelectual, muitas vezes, são necessárias suposições sobre o funcionamento dos componentes envolvidos em uma arquitetura real. Neste trabalho, é proposto um método de se obter as informações necessárias para replicar o comportamento esperado utilizando dados gerados a partir de métricas obtidas em máquinas reais. Essa proposta, então, é implementada no OrCS, um simulador de arquitetura x86, e comparada com um método mais simples de representação interna de instruções. As análises dos resultados de diversos *microbenchmarks* e do conjunto de *benchmarks* SPEC CPU-2017 revelam que houve uma melhoria significativa de precisão obtida pelas modificações propostas. Também conclui-se que a implementação original do OrCS não era capaz de reproduzir o comportamento esperado de diversas instruções, principalmente das instruções vetoriais, resultando em possíveis interpretações incompletas de diferentes aspectos da simulação.

Palavras-chave: Simulação Arquitetural; Análise de Instruções; Simulador Baseado em Traços

ABSTRACT

Research in computer architecture presents great difficulties as it requires testing, analysis, and validation of ideas. These difficulties are due to the high complexity involved. The usage of simulators is a flexible and inexpensive alternative to physical prototypes, and it is an accurate alternative to analytical modeling. However, a high level of development, modeling, and validation are requirements to achieve a sufficiently accurate simulator. Several factors must be considered for a representation faithful to the real architecture, such as instruction latencies, number of micro-operations, and execution unit distribution. Representing instructions in a computer architecture simulator is crucial to the accuracy of the simulation and the ability to generate detailed statistics. There are several ways to implement a CISC Instruction Set Architecture in a simulator. However, given intellectual property protection, assumptions about this behavior in real architecture are often necessary due to a lack of information. In this work, we proposed obtaining the necessary information to replicate the expected behavior using data generated from metrics extracted from real machines. This proposal is then implemented on OrCS, an x86 architecture simulator, and then compared with a more straightforward method of executing instructions internally. The results from multiple microbenchmarks and the SPEC CPU2017 benchmark suite reveal significant improvement in accuracy yielded by the proposed modifications. It is also concluded that the original OrCS implementation could not reproduce the expected behavior of several instructions, mainly vector instructions, possibly resulting in incomplete interpretations of different aspects of the simulation.

Keywords: Architectural Simulation; Instructions Analysis; Trace-driven Simulator

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ISA	*Instruction set architecture*
µop	microinstrução
CISC	*Complex Instrucion Set Computer*
RISC	*Reduced Instrucion Set Computer*
OrCS	*Ordinary Computer Simulation*
SiNUCA	*Simulator of Non-Uniform Cache Architectures*
ROB	*Reorder Buffer*
RAT	*Register Alias Table*
MOB	*Memory Order Buffer*
BTB	*Branch Target Buffer*
iclass	*Instruction class*
iform	*Instruction form*
IPC	Instruções por ciclo
XED	*x86 Encoder Decoder*
ALU	*Arithmetic Logic Unit*
AGU	*Address Generation Unit*
MPKI	*Misses per kilo instructions*
LLC	*Last Level Cache*

SUMÁRIO

1 INTRODUÇÃO

Pesquisas, testes, e análises na área de arquitetura de computadores dependem de simulações capazes de modelar o microprocessador e seus componentes de forma precisa, versátil e rápida. O desenvolvimento desses simuladores traz muitos desafios, como a alta complexidade de microarquiteturas modernas, decisões sobre quais componentes devem ser incluídos na simulação, e a validação para garantir o funcionamento esperado. Além disso, esse processo de desenvolvimento é dificultado pelo fato de que muitas informações são ocultadas por motivos de proteção de propriedade intelectual das indústrias, sendo necessário, muitas vezes, o uso de estimativas fornecidas por técnicas de validação, como testes, benchmarks, e microbenchmarks capazes de evidenciar o comportamento isolado de cada subsistema modelado (Alves, 2014).

Simuladores classificados como "ciclo a ciclo" têm a proposta de simular o comportamento de uma microarquitetura a cada ciclo de relógio. Para que isso seja possível, é necessário determinar a latência de cada um dos componentes funcionais a serem simulados. O nível de granularidade implementado em um simulador determina quais componentes devem ser incluídos. Idealmente, todos os subsistemas seriam simulados detalhadamente, porém quanto mais detalhes são incluídos, maior será a complexidade, e, consequentemente, o tempo de execução da simulação. Portanto é necessário atribuir latências a procedimentos internos. No hardware, isso pode ser interpretado como a latência de um determinado comportamento de um conjunto de componentes.

Outra característica comumente atribuída aos simuladores é a orientação a traços, que, ao contrário de simuladores de sistemas completos, não executa, de fato, a computação implementada pelo programa simulado. Dessa forma, é necessário que a execução ocorra primeiramente em uma máquina real, onde traços serão extraídos com as informações necessárias para a reprodução simplificada. Isso proporciona uma maior flexibilidade na forma como as instruções serão tratadas no simulador, permitindo abstrações ao representá-las internamente.

A escolha do conjunto de instruções, ou *Instruction Set Architecture* (ISA), é uma das decisões que devem ser levadas em conta ao desenvolver um simulador, e isso depende de qual modelo de processador será simulado, ou quais programas serão aceitos pelo simulador.

Vários mecanismos contribuem com a latência de um programa, como o comportamento e configuração da hierarquia de memória, previsão de desvios, diferentes estágios do *pipeline*, comunicações internas, entre outros. O comportamento de cada instrução individual é um dos fatores que deve ser contabilizado para que uma simulação seja *cycle-accurate*. ISAs com especificações privadas, como o x86, tornam difícil o processo de estimar o passo a passo de cada instrução ao longo do *pipeline*, além da latência de cada estágio.

Ao mesmo tempo que o desenvolvimento de simuladores requer decisões de simplificações e estimativas de comportamento do conjunto de instruções, há trabalhos sendo desenvolvidos com o objetivo de prover mais informações sobre as características detalhadas de cada instrução (Abel e Reineke, 2019). Com isso é possível fornecer valores mais precisos às configurações de simuladores que necessitam desse processo.

1.1 PROBLEMA E MOTIVAÇÃO

O simulador OrCS (Köhler, 2019) foi desenvolvido sob o paradigma orientado a traço, trabalha com precisão de ciclos, suporta a ISA x86, e tem a proposta de simplificar o simulador SiNUCA (Alves, 2014). Atualmente o OrCS é utilizado por pesquisadores para auxiliar no

desenvolvimento de diversas pesquisas e experimentos na área de arquitetura de computadores. O OrCS foi desenvolvido para aceitar os mesmos traços propostos pelo SiNUCA. Em ambos os simuladores, o conjunto de instruções x86 é traduzido para uma ISA mais simples com o propósito de agrupar as diferentes funções de cada instrução pelo valor estimado de latência e também pelo tipo de unidade funcional utilizada para sua execução. Esse agrupamento foi construído com base em observações feitas sobre os dados obtidos por A. Fog (Fog, 2017), e foram utilizadas as principais características das instruções para classificá-las. Já os valores de latência utilizados no simulador foram definidos com base em observações empíricas obtidas através de microbenchmarks (Alves, 2014).

Apesar do processo de validação dos simuladores citados apresentar ótimos resultados, a simplificação excessiva implica na possibilidade de haver aplicações cuja execução abundante de certas instruções, que não são bem representadas pelo processo descrito, apresente resultados que não condizem com a realidade. Isso implicaria em falsas conclusões sobre aspectos da arquitetura simulada.

Há diversas formas de representar instruções em um simulador de arquitetura. Algumas implementações propõem comportamentos definidos manualmente para cada instrução, outros apresentam uma análise mais elaborada de como cada instrução deveria se comportar no simulador. A motivação para esse trabalho surge com a possibilidade de utilizar trabalhos publicados recentemente, como o de Abel e Reineke (Abel e Reineke, 2019), que fornecem dados detalhados sobre o conjunto de instruções x86, para propor uma nova forma de representar o comportamento das instruções compatíveis com o simulador OrCS e, possivelmente, obter resultados mais precisos.

1.2 OBJETIVOS

Devido à subjetividade do nível de granularidade a ser adotado ao desenvolver um simulador que exige um equilíbrio entre simplicidade e acurácia, esse trabalho tem como objetivo apresentar uma análise de uma possível solução para o problema descrito. Também serão apresentadas as consequências das decisões envolvidas, que serão medidas através de execuções de cargas de trabalho de teste.

Trabalhos recentes, como o de Abel e Reineke (Abel e Reineke, 2019), fornecem diversos dados sobre o comportamento de cada instrução disponível para diversas arquiteturas pertencentes à ISA x86. A análise que será apresentada utiliza esses dados para extrair informações relevantes para os experimentos propostos. Com isso, espera-se atualizar a implementação do OrCS para que as instruções utilizadas por ele sejam compatíveis com os dados de uma microarquitetura real, melhorando, assim, a acurácia dos experimentos que o simulador proporciona.

1.3 CONTEÚDO DO TRABALHO

No Capítulo 2 deste trabalho serão apresentados os fundamentos e definições necessárias para o entendimento dos trabalhos relacionados, da proposta, e dos resultados. Alguns trabalhos que tratam de simuladores, análise de instruções x86, e temas relacionados à proposta são descritos brevemente no Capítulo 3. Após isso, o Capítulo 4 apresentará as propostas junto de análises e explicações acerca das diferenças entre o OrCS atual e as novas modificações, trazendo uma possível solução para o problema descrito. Alguns experimentos e discussões são apresentados no Capítulo 5 e, por fim, serão apresentadas as conclusões e ideias de trabalhos futuros no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

A área de arquitetura de computadores engloba diversas definições e termos, muitos dos quais fogem do escopo desse trabalho. Nesse capítulo serão apresentados alguns conceitos com o propósito de contextualização. Para isso será adotado um nível de abstração necessário para a dissertação, portanto diversos detalhes serão abstraídos.

2.1 MICROARQUITETURA

A microarquitetura de um processador é o conjunto de componentes arquiteturais transparentes ao programador. A estrutura simplificada do pipeline dos microprocessadores da Intel é representada na Figura 2.1. As microarquiteturas modernas possuem diversos componentes não ilustrados na figura, porém, as informações incluídas são suficientes para ilustrar o fluxo básico das instruções x86.

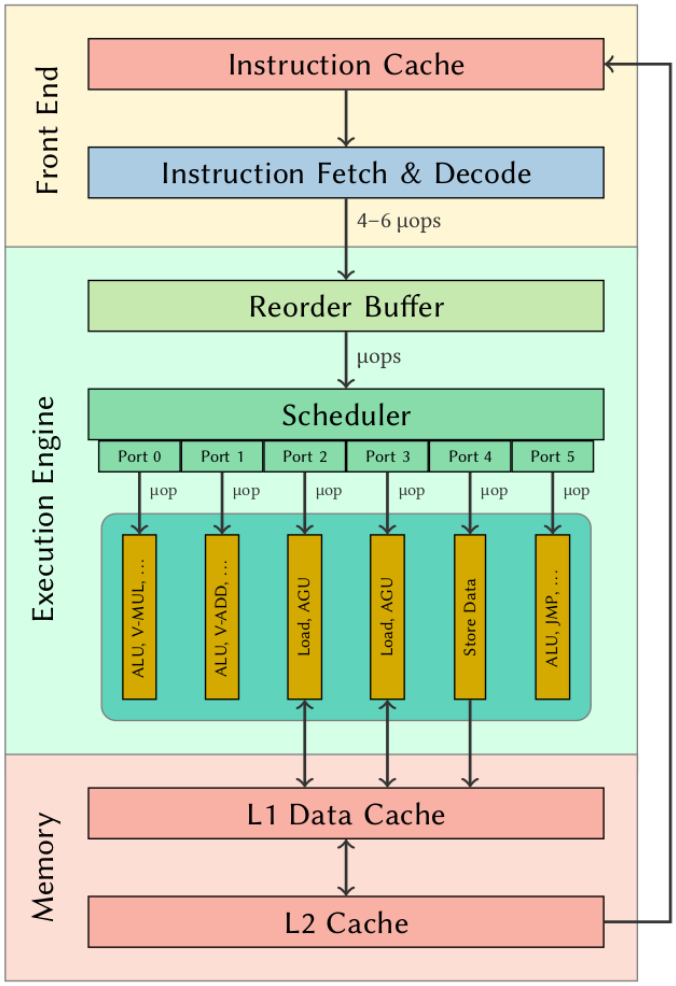


Figura 2.1: Pipeline simplificado Intel (Abel e Reineke, 2019)

A estrutura denominada *Front End* é responsável por buscar (*fetch*) as instruções da memória, decodificá-las (*decode*), transformando-as em microinstruções (µops), e entregá-las ao *Back End*, que na figura é composto por *Execution Engine* e *Memory*, onde serão executadas pelas

unidades funcionais. Ou seja, o *Frond End* precisa ter a capacidade de despachar microinstruções consistentemente para manter o *Back End* ativo (Wikichip, 2016c).

Na *Execution Engine*, as µops são inicialmente inseridas no *Reorder Buffer* (ROB), onde é feito a renomeação de registradores para evitar dependências falsas entre µops, alocação de recursos para *loads*, *stores*, e algumas instruções simples são executadas. Depois disso, as µops são encaminhadas para o *Scheduler*, onde são enfileiradas e então, assim que possível, despachadas através de portas de execução (*execution ports*) para as unidades funcionais (Abel e Reineke, 2019).

As microarquiteturas Intel possuem até 10 portas de execução, e cada porta é conectada a um conjunto de unidades funcionais (*functional units*), tais como a ALU (*arithmetic logic unit*), a unidade de multiplicação vetorial, a AGU (*address-generation unit*), entre outras. As µops não são necessariamente despachadas na ordem do programa, o que permite uma execução fora de ordem (*out-of-order*), ou seja, instruções mais novas e sem dependências podem ser executadas antes de instruções mais antigas que estejam aguardando que dependências sejam resolvidas.

2.2 INSTRUCTION SET ARCHITECTURE

Instruction Set Architecture (ISA) é uma interface entre o processador e o software (sistema operacional e demais programas). Ou seja, é uma forma de abstrair o funcionamento interno de um processador, provendo informações o suficiente para que um programador consiga se comunicar com o computador através de uma linguagem de baixo nível (Degenbaev, 2012).

Visto que a ISA está relacionada à implementação da microarquitetura de um processador, é de se esperar que, devido às divergências nas prioridades e processos de desenvolvimento, haja diferentes conjuntos de instruções para cada arquitetura. Alguns dos exemplos mais conhecidos são: x86, ARM, e MIPS.

Para uma determinada arquitetura, cada indústria pode, por motivos de propriedade intelectual, adotar uma microarquitetura diferente na implementação da mesma ISA. Tais diferenças se refletem no desempenho e consumo de energia apresentado por cada processador, embora todos suportem as mesmas instruções. Por exemplo, no caso da ISA x86, tanto a Intel quando a AMD desenvolvem arquiteturas para o mesmo conjunto de instruções.

2.2.1 RISC vs. CISC

Os conjuntos de instruções mais utilizados podem ser classificados em duas categorias principais: *Reduced Instruction Set Computer* (RISC) e *Complex Instruction Set Computer* (CISC). O motivo dessa divisão é derivado de decisões tomadas com o objetivo de respeitar restrições impostas pelo hardware ao longo da história do desenvolvimento das arquiteturas, tais como memória limitada, consumo de energia, e complexidade desejada de um microprocessador.

A principal diferença entre RISC e CISC se deve à simplicidade das instruções. Enquanto o conjunto de instruções RISC é, idealmente, pequeno, possui tamanho fixo, e é composto por instruções simples que requerem aproximadamente a mesma quantidade de ciclos para serem executadas, o conjunto CISC é composto por instruções mais complexas e especializadas, com tamanho variável, e apresentam uma grande variedade na latência.

Um exemplo clássico dessa diferença é a operação de multiplicação entre dois números armazenados em memória, ilustrado na Tabela 2.1.

Em uma arquitetura CISC, uma única instrução é capaz de carregar os dois números, M_1 e M_2, da memória, executar a multiplicação, e armazenar o resultado na memória em M_1. Já a mesma operação, em uma arquitetura RISC, requer duas instruções para carregar os dois

Tabela 2.1: Exemplo RISC vs. CISC

Multiplicação de inteiros em memória			
RISC		**CISC**	
`LOAD` A, M_1	$(A \leftarrow M_1)$	`MULT` M_1, M_2	$(M_1 \leftarrow M_1 * M_2)$
`LOAD` B, M_2	$(B \leftarrow M_2)$		
`MULT` A, B	$(A \leftarrow A * B)$		
`STORE` M_1, A	$(M_1 \leftarrow A)$		

operandos da memória em dois registradores diferentes (no exemplo, A e B), uma instrução para executar a multiplicação, e uma outra instrução para armazenar o resultado do registrador na memória (Chen et al., 2000).

A Figura 2.2 ilustra o conceito de "Envelhecimento de ISA", apresentado por B. Lopes et al (Lopes et al., 2015), demonstrando a complexidade crescente, medida através da quantidade de instruções, de diferentes ISAs. Embora essa divisão sistemática entre conjunto de instruções ainda seja bastante usada, é notável as indústrias apresentarem arquiteturas modernas cada vez menos puritanas. Ou seja, algumas arquiteturas tradicionalmente conhecidas por serem RISC, agora suportam centenas de instruções, algumas bastante complexas, na busca por melhores desempenhos computacionais.

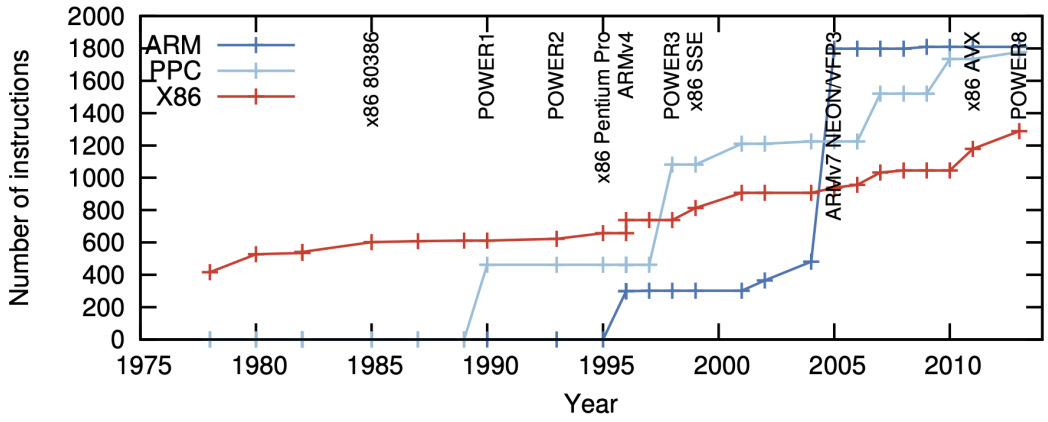


Figura 2.2: Envelhecimento de ISA (Lopes et al., 2015)

2.2.2 ISA x86

O conjunto de instruções x86 é considerado CISC e é utilizado na maioria dos processadores comercializados no mercado de servidores e *desktops*. De acordo com um levantamento realizado no terceiro trimestre de 2019, cerca de 93% dos servidores possuem arquitetura x86 (Morgan, 2019).

Mesmo esses processadores sendo tão utilizados, não há uma descrição detalhada sobre como eles executam as instruções. Há apenas documentações oficiais que descrevem informalmente e superficialmente o comportamento funcional (visível ao programador) e a variedade de tais instruções. Portanto, ao desenvolver softwares de baixo nível, muitas vezes é necessário que programadores recolham métricas obtidas a partir de execuções de testes e experimentos para estimar quais decisões devem ser tomadas para obter o código com máxima otimização (Degenbaev, 2012).

Cada instrução x86 possui tamanho entre 1 e 15 bytes (Barton et al., 2011). Como descrito anteriormente, as instruções individuais podem executar várias ações, tais como leitura de memória, operação, e escrita na memória. Por isso, essas instruções são, internamente, decodificadas como microinstruções (µops), que, por sua vez, possuem um comportamento semelhante às instruções RISC. Ou seja, comportamentos mais especializados, como operações de memória, multiplicação, entre outros.

2.2.3 Microinstruções (µops)

A informação que relaciona cada instrução ao conjunto de microinstruções correspondente não é divulgada oficialmente pelos fabricantes para a ISA x86. A documentação oficial (Corporation, 2019) afirma que tais µops são despachadas através portas, onde cada porta é atribuída a um subconjunto de unidades funcionais responsáveis por executar as operações correspondentes, conforme explicado anteriormente.

Também não há informações sobre as funções específicas de cada microinstrução, nem mesmo nomenclatura ou uma listagem de quantas µops existem. Porém é possível fazer estimativas sobre esses dados a partir da lista de unidades funcionais presentes em uma dada microarquitetura, que é disponibilizada no manual da Intel (Corporation, 2019). Um exemplo encontrado no manual oficial apresenta as portas e as operações de suas unidades funcionais relacionadas na microarquitetura *Skylake* (Tabela 2.2).

Tabela 2.2: Skylake: unidades funcionais por porta

Scheduler Ports Designation	
Port 0	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and String ops FP Add, Multiply, FMA Integer/FP Division and Square Root AES Encryption Branch2
Port 1	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and Bit Scanning FP Add, Multiply, FMA
Port 5	Integer/Vector Arithmetic, Logic Vector Permute x87 FP Add, Composite Int, CLMUL
Port 6	Integer Arithmetic, Logic, Shift Branch
Port 2	Load, AGU
Port 3	Load, AGU
Port 4	Store, AGU
Port 7	AGU

2.2.4 Instruções Assembly x86

Um conjunto de instruções de uma linguagem *assembly* busca representar todas as possíveis instruções de máquina de uma determinada arquitetura, geralmente um para um (Archer, 2016). Essa representação tem como objetivo tornar o código de máquina mais legível através do uso de mnemônicos, facilitando a análise e o desenvolvimento de programas complexos e extensos.

As instruções *assembly* da ISA x86 possuem o seguinte formato:

$$\texttt{mnemônico op}_1\texttt{, op}_2\texttt{, ..., op}_n$$

O mnemônico identifica a operação e é representado por uma palavra descritiva, por exemplo, `ADD` e `XOR`. O primeiro operando ($\texttt{op}_1$) geralmente representa o destino da operação, enquanto os outros ($\texttt{op}_2 \ldots \texttt{op}_n$) são os de origem. Esses podem ser registradores, endereços de memória, ou imediatos. Além dos operandos explícitos, uma instrução também pode utilizar operandos implícitos (Abel e Reineke, 2019).

Um exemplo de instrução é o seguinte:

```
ADD RAX, [RBX]
```

Essa instrução calcula a soma entre o valor armazenado no registrador de propósito geral `RAX` e o valor armazenado no endereço de memória indexado pelo valor do registrador `RBX`, e armazena o resultado em `RAX`. Nesse caso, `RAX` e `[RBX]` são operandos explícitos. Essa instrução também modifica registradores implícitos, como as *status flags* (e.g., *carry flag*) (Abel e Reineke, 2019).

Há uma grande variedade de registradores disponíveis na ISA x86. Isso resulta em uma grande quantidade de possíveis combinações de operandos para uma mesma instrução, porém, devido a certas decisões, restrições, e propósitos específicos de alguns registradores, nem todas as combinações são aceitas, assim aumentando a complexidade do conjunto de instruções.

A Tabela 2.3 mostra a variedade de combinações de endereçamento que existem para a instrução `ADD`, citada de exemplo anteriormente. A tabela lista apenas as instruções de adição de inteiros do conjunto de instruções base, ou seja, apenas as instruções com o mnemônico `ADD`, excluindo instruções vetoriais como `VPADDB` ou `ADDPD`. Também há outros modificadores que não são listados na tabela, como `ADD_LOCK`, etc..

2.2.5 Extensões x86

As instruções citadas anteriormente, como as variações de `ADD` e `XOR`, são exemplos que compõe o subconjunto de instruções básicas. Esse conjunto teve início com a ISA 8086/8088 e foi estendido conforme a capacidade dos processadores foi aumentando e novas técnicas foram desenvolvidas. Em versões modernas de microarquiteturas Intel, essas instruções ainda operam apenas sobre inteiros e compõe a ISA x86-64 ou AMD64, e são capazes de utilizar registradores de até 64-bits, conforme exemplificado na Tabela 2.3. Esse subconjunto será referenciado nesse trabalho como o conjunto de instruções "base", mas, oficialmente, é muitas vezes chamado de "Instruções de Propósito Geral" (Corporation, 2019).

Além do conjunto de instruções base, há diversos outros conjuntos que são suportados pelas microarquiteturas Intel atuais. Esses conjuntos são definidos como "extensões x86", portanto fazem parte da ISA x86, porém são adendos e possuem nomes próprios e versões diferentes. As extensões foram desenvolvidas com o objetivo de suportar registradores de largura maior que 64-bits, apresentar novas funcionalidades, e melhorias de desempenho.

A designação de nomes para tais extensões, facilita a identificação de quais instruções são suportadas por cada processador. Dessa maneira, programadores ou compiladores podem escrever diversas versões de uma mesma rotina, garantindo que haverá compatibilidade independente das extensões disponíveis no processador alvo.

A seguir são listadas as principais extensões de forma simplificada e generalizada, pois existem diversas extensões que, em muitos casos, contêm poucas instruções, ou possuem propósitos muito específicos, e, portanto, não são relevantes para esse trabalho.

Tabela 2.3: Variações da instrução `ADD` (Cloutier, 2019)

Instrução	**Descrição**
`ADD` AL, *imm8*	Add imm8 to AL.
`ADD` AX, *imm16*	Add imm16 to AX.
`ADD` EAX, *imm32*	Add imm32 to EAX.
`ADD` RAX, *imm32*	Add imm32 sign-extended to 64-bits to RAX.
`ADD` r/m8, *imm8*	Add imm8 to r/m8.
`ADD` r/m16, *imm16*	Add imm16 to r/m16.
`ADD` r/m32, *imm32*	Add imm32 to r/m32.
`ADD` r/m64, *imm32*	Add imm32 sign-extended to 64-bits to r/m64.
`ADD` r/m16, *imm8*	Add sign-extended imm8 to r/m16.
`ADD` r/m32, *imm8*	Add sign-extended imm8 to r/m32.
`ADD` r/m64, *imm8*	Add sign-extended imm8 to r/m64.
`ADD` r/m8, r8	Add r8 to r/m8.
`ADD` r/m16, r16	Add r16 to r/m16.
`ADD` r/m32, r32	Add r32 to r/m32.
`ADD` r/m64, r64	Add r64 to r/m64.
`ADD` r8, r/m8	Add r/m8 to r8.
`ADD` r16, r/m16	Add r/m16 to r16.
`ADD` r32, r/m32	Add r/m32 to r32.
`ADD` r64, r/m64	Add r/m64 to r64.

rN = Registrador de N-bits
mN = Endereço de memória de N-bits
r/mN = Registrador ou endereço de memória de N-bits
imm$N$ = Imediato de N-bits

- **SSE**: As instruções SSE (*Streaming SIMD Extensions*) foram adicionadas, inicialmente, aos processadores *Pentium III*, mas várias outras versões foram introduzidas posteriormente, como SSE2, SSE3, SSSE3, e SSE4. Esse conjunto representa uma extensão do modelo de execução SIMD (*Single Instruction Multiple Data*). As instruções que o compõe utilizam, em geral, os registradores XMM, que possuem 128-bits e são capazes de realizar operações vetoriais e de ponto flutuante.

- **AVX**: Inicialmente suportadas pelos processadores *Sandy Bridge*, as instruções AVX (*Advanced Vector Extensions*) introduzem a capacidade de operar sobre registradores de até 256-bits, denominados YMM, de forma vetorial. Também possui outra versão conhecida como AVX2.

- **AVX-512**: As instruções AVX-512 estendem ainda mais a capacidade do modelo SIMD, permitindo registradores de até 512-bits, chamados de ZMM. Esse conjunto foi apresentado nos processadores Xeon Phi x200 e atualmente é mais comum no mercado de servidores. O AVX-512 possui diversas instruções, tornando o número de combinações possíveis de operandos extenso, portanto é extremamente complexo.

2.3 SIMULADOR ORCS

O *Ordinary Computer Simulator* (OrCS) é um simulador inicialmente desenvolvido por R. Köhler (Köhler, 2019) e pode ser classificado como *trace-driven*, ou seja, é um simulador que recebe de entrada um arquivo contendo "traços" de um programa executado em uma máquina real. Os traços, que contêm informações sobre as instruções que foram executadas, são simulados ciclo a ciclo, ou seja, cada componente é modelado para executar operações utilizando aproximadamente o mesmo número de ciclos da microarquitetura real.

O OrCS foi desenvolvido para funcionar com traços de aplicações da ISA x86, portanto alguns dos mecanismos descritos anteriormente são modelados, tais como as unidades funcionais. A implementação possui como base o *Simulator of Non-Uniform Cache Architectures* (SiNUCA) desenvolvido por M. Alves (Alves, 2014), porém diversos componentes não foram incluídos ou foram simplificados com a proposta de reduzir a complexidade, tais como *network-on-chips* e *multi-banked caches*. Inicialmente, o OrCS foi desenvolvido para reproduzir o mecanismo de *Enhanced Memory Controller*, mas atualmente é utilizado para outros propósitos de pequisa, como mecanismos de processamento vetorial em memória e caches heterogêneas, na área de arquitetura de computadores. A Figura 2.3 contém um diagrama simplificado dos principais componentes simulados no OrCS.

Ao executar o OrCS, é necessário fornecer um arquivo de configuração estruturado no formato aceito pela biblioteca *libconfig* (Lindner, 2018). Esse arquivo exige diversas constantes, como tamanho de *buffers*, latências de componentes, taxas de transferências, entre outros. Esses valores permitem que a implementação seja adaptada de forma flexível para aproximar valores de microarquiteturas reais. Dessa forma, espera-se que as métricas obtidas a partir da simulação da execução de um programa se assemelhem às métricas obtidas a partir da execução na máquina real.

2.3.1 Componentes Principais

O pipeline do OrCS é composto por seis estágios. Cada estágio possui valores associados que definem quantos ciclos são necessários para serem concluídos (Alves, 2014). Os principais estágios e componentes relacionados são descritos a seguir:

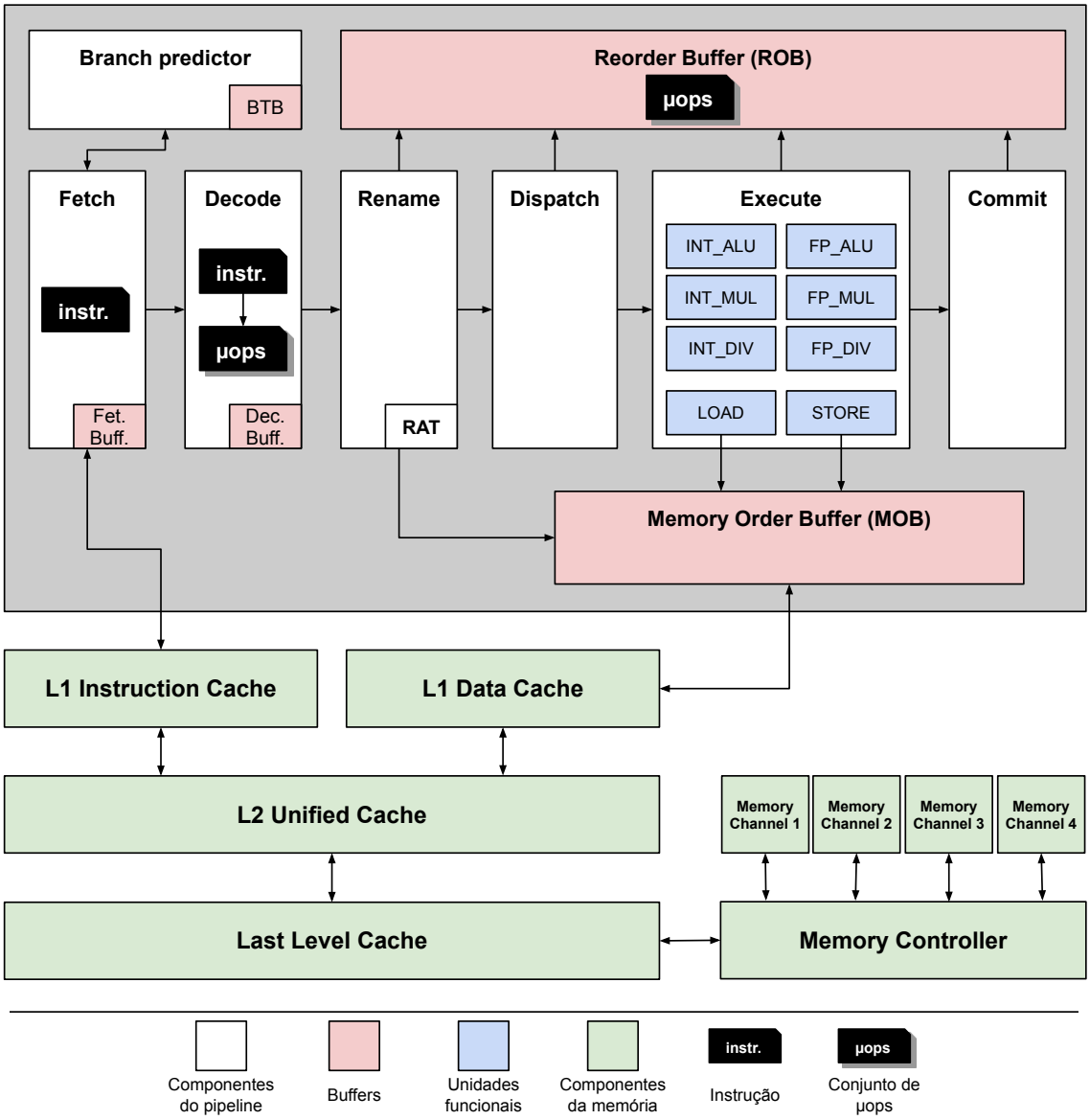


Figura 2.3: Diagrama OrCS.

- **Fetch**: Esse estágio é responsável por buscar instruções da memória. Instruções de desvio são fornecidas ao *branch predictor* as quais podem paralizar temporariamente o fetch caso haja falhas de predição. Todas as instruções são passadas para o próximo estágio (*Decode*) através do *Fetch Buffer*.

- **Decode**: Nesse estágio, as instruções são decodificadas em µops. As µops são inseridas no *Decode Buffer* que possui um tamanho fixo, portanto a vazão desse estágio é limitada pelo espaço disponível no *buffer*.

- **Rename**: As µops são removidas do *Decode Buffer* e inseridas no *Reorder Buffer* (ROB). A *Register Alias Table* (RAT) é responsável por resolver dependências entre µops que operam sobre registradores. E se a µop realiza uma operação em memória, como `LOAD` ou `STORE`, será inserida no *Memory Order Buffer* (MOB).

- **Dispatch**: Nesse estágio, as µops são despachadas para as unidades funcionais apropriadas, respeitando a quantidade e a capacidade de cada unidade funcional. Esse estágio corresponde ao *Scheduler* ilustrado na Figura 2.1.
- **Execute**: As µops são removidas das unidades funcionais, sendo elas: operações sobre inteiros (`INT_ALU`, `INT_MUL` e `INT_DIV`), operações sobre valores de ponto flutuante (`FP_ALU`, `FP_MUL` e `FP_DIV`), e operações de memória (`LOAD` e `STORE`). As µops, que dependem da instrução recém removida da unidade funcional, são marcadas como "resolvidas" e as operações de memórias são marcadas no MOB como "prontas". Importante ressaltar que o simulador apenas modela o comportamento em termos de latência das instruções e portanto não simula a execução real das instruções.
- **Commit**: Esse estágio é responsável por remover as µops em ordem do ROB.
- **Branch Predictor**: Esse componente suporta a implementação de qualquer implementação de mecanismo de *branch prediction* e possui um *Branch Target Buffer* (BTB). Ao receber a instrução atual e a próxima do traço, o *Branch Predictor* retorna se o desvio foi previsto corretamente ou não pelo componente modelado. Essa comparação é possível devido ao fato das instruções serem obtidas através de um traço, ou seja, as instruções já foram executadas previamente em uma máquina real que gerou os traços.
- **Reorder Buffer (ROB)**: Esse *buffer* armazena as µops após o estágio *Decode* e armazena o estado atual de cada µop conforme os estágios seguintes operam sobre elas. Este componente é essencial em arquiteturas com execução fora-de-ordem para garantir a coerência da execução.
- **Memory Order Buffer (MOB)**: Quando uma operação de memória é executada, a µop relacionada é marcada como "pronta" e é armazenada nesse *buffer* até os componentes responsáveis por simular a memória atenderem a requisição. No caso de instruções de escrita na memória, tais instruções só serão enviadas para a memória quando forem a mais antiga pipeline, facilmente identificável no ROB.

2.3.2 Formato dos Traços

Os traços compatíveis com o OrCS possuem o mesmo formato dos traços do SiNUCA, os quais são criados, geralmente, utilizando a ferramenta Pin e PinPoints (Patil et al., 2004) e são divididos em três arquivos diferentes (Alves, 2014):

- **Static**: Contém uma listagem dos blocos básicos, onde cada bloco contém suas instruções codificadas em um *assembly* específico.
- **Dynamic**: Contém a ordem que os blocos básicos foram executados ao longo da execução do programa.
- **Memory**: Contém informações sobre as operações de memória que foram geradas ao longo da execução do programa.

Cada arquivo possui um formato específico, porém, para o escopo desse trabalho, apenas o formato do *assembly* das instruções presentes no arquivo *Static* é relevante. Esse *assembly* contém as informações necessárias para o funcionamento do OrCS, portanto diversas informações presentes nas instruções x86 são excluídas. O formato desse *assembly* é exemplificado para a instrução `ADD` na Tabela 2.4.

Tabela 2.4: *Assembly* OrCS e x86 da instrução `ADD`

Assembly x86:	`ADD EAX, [RBP]`
Assembly OrCS:	`ADD 1 0x95720 3 2 14 65 1 34 14 0 1 0 1 0 0 0`

Os valores presentes em uma instrução do OrCS são descritos na Tabela 2.5 na mesma ordem que eles aparecem na instrução do traço. O mnemônico utilizado para identificar as instruções se trata de um valor denominado oficialmente como "*iclass*" ou *instruction class*, especificado na biblioteca XED (*x86 Encoder Decoder*) (Intel, 2019). O *iclass* é incluído na instrução apenas para facilitar a identificação visual das instruções, pois, na implementação do OrCS, o valor utilizado para, de fato, classificar as instruções (e.g. associá-las às unidades funcionais) é o *opcode number*, que, apesar de ser um valor numérico, é associado internamente a um dos seguintes valores: `NOP`, `INT_ALU`, `INT_MUL`, `INT_DIV`, `FP_ALU`, `FP_MUL`, `FP_DIV`, `BRANCH`, `LOAD`, e `STORE`. E esses são todos os tipos que uma µop pode ser classificada dentro do OrCS.

Tabela 2.5: Descrição do *assembly* OrCS

#	**Nome**	**Descrição**
1	*Iclass*	Mnemônico x86 simplificado
2	*Opcode number*	Classificação da instrução
3	*Instruction address*	Endereço da instrução
4	*Instruction size*	Tamanho da instrução
5	*#read reg.*	Número de registradores lidos
6	*Read registers*	Listagem dos IDs dos registradores lidos
7	*#write reg.*	Número de registradores escritos
8	*Write registers*	Listagem dos IDs dos registradores escritos
9	*Base register*	Utilizado para acesso à memória
10	*Index register*	Utilizado para acesso à memória
11	*is_read*	Indicação de leitura à memória
12	*is_read2*	Indicação de segunda leitura à memória
13	*is_write*	Indicação de escrita à memória
14	*Branch type*	Tipo de desvio condicional
15	*is_indirect*	Indicação de desvio indireto
16	*is_predicated*	Indicação de predicação
17	*is_prefetch*	Indicação de *prefetch*

A escolha desses tipos de instruções (*opcode number*) foi feita com o objetivo de simplificar a ISA simulada, assim agrupando instruções complexas x86 com base no funcionamento básico que elas possuem. Essa classificação, está associada às unidades funcionais disponíveis e ao valor da latência de cada operação. Portanto uma µop pode assumir um entre 10 valores diferentes de latência.

3 REVISÃO BIBLIOGRÁFICA

3.1 SIMULADORES X86

Devido à popularidade da ISA x86, diversos simuladores x86 foram desenvolvidos sob diferentes paradigmas e propósitos. Na literatura, há trabalhos, como o de Akram e Sawalha (Akram e Sawalha, 2016), nos quais foram realizadas comparações entre alguns simuladores, e avaliações de desempenho e de acurácia. Aqui serão apresentados alguns casos interessantes de simuladores amplamente utilizados e trabalhos relacionados.

O simulador **PTLsim**, desenvolvido por Yourst (Yourst, 2007), assim como o OrCS, é *cycle-accurate*. Também é configurável, com o objetivo de simular arquiteturas diferentes e a simulação ocorre em nível de µops. Segundo os autores, a funcionalidade e codificação das µops foram definidas com base nos processadores Intel Pentium 4, Core 2, e AMD K8. Por se tratar de um projeto antigo e descontinuado, apenas as extensões x87 (que não é mais implementada nos processadores modernos), SSE, e SSE2 foram incluídas. O guia do usuário e referência do PTLsim (Yourst, 2007) lista todas as µops que foram definidas, e no código fonte é possível verificar como as instruções são decodificadas em µops, porém a complexidade dessa parte do código torna extremamente difícil a compreensão dos detalhes do processo de *decode*.

Desenvolvido por Patel et al. (Patel et al., 2011), o **MARSSx86** também possui a proposta de ser *cycle-accurate*, e é apresentado como uma extensão da implementação do PTLsim, porém com diversas características adicionais e otimizações. O modelo de *pipeline* é idêntico ao do PTLsim (Akram e Sawalha, 2019), portanto as µops possuem o mesmo comportamento. A grande diferença está na integração com o QEMU que permite a alternância entre emulação em casos onde é necessário avançar a execução do programa, e simulação para regiões de interesse, onde há a necessidade de se obter dados mais precisos.

Outro trabalho interessante, publicado por Asri et al. (Asri et al., 2016), apresenta problemas com a implementação do MARSSx86. Foi observado que havia uma superestimação no número de µops por instrução, resultando em divergências entre as estatísticas obtidas da simulação versus a execução na máquina real. Mais especificamente, esse problema se tornava evidente ao utilizar muitas instruções do tipo SSE, pois, enquanto o MARSSx86 decodificava cada uma dessas instruções específicas em múltiplas µops, a decodificação real gerava apenas uma µop. No artigo publicado, é mencionada a possibilidade de solucionar o problema por meio de modificações no *decode* do MARSSx86, porém foi observada a falta de informações acerca da relação entre instruções e µops na ISA x86. A solução foi aumentar a taxa de execução de µops no estágio *execute* apenas para essas instruções específicas, assim múltiplas µops são tratadas como uma só, melhorando os resultados das comparações.

O simulador **gem5** (Binkert et al., 2011) é utilizado amplamente tanto na indústria quanto na academia. É um simulador que suporta diversas ISAs, como ARM, x86, RISC-V, SPARC, e Alpha, e é *event-driven*, ou seja, a simulação é feita por meio de determinados eventos, em vez de simular todos os ciclos. Apesar disso, esse simulador é capaz de acompanhar os eventos ciclo a ciclo, tornado sua precisão comparável a um simulador *cycle-accurate* (Akram e Sawalha, 2019). Sua implementação também contém a decodificação de instruções em µops. Com o propósito de generalizar a implementação para suportar diversas ISAs, foi desenvolvida uma linguagem de domínio específico, denominada “ISA DSL”, para representar os conjuntos de instruções. A parte do simulador responsável por implementar a ISA x86 contém uma listagem de todas as µops disponíveis, onde cada uma possui um nome descritivo, como “add” e “mul”, e

também há relações de cada instrução x86 a um conjunto de µops. Tudo isso é feito através da linguagem ISA DSL e a análise sintática dessa representação determina o comportamento das instruções no estágio *decode*. Evidentemente, a relação entre instruções e µops foi feita a partir da análise de características das instruções individuais em vez de uma documentação oficial.

As comparações entre simuladores ou entre simuladores e máquinas reais (i.e. validação), são geralmente apresentadas em termos de IPC (*instructions per cycle*) e métricas relacionadas à memória, como *cache misses*. Em Akram e Sawalha (Akram e Sawalha, 2019), diversas comparações e análises desse tipo são apresentadas.

3.2 PROCESSO DE VALIDAÇÃO DO SINUCA

Conforme citado anteriormente, o OrCS foi implementado utilizando o SiNUCA como base, portanto há diversas semelhanças entre os dois simuladores. Por esse motivo, o processo de validação do OrCS foi originalmente desenvolvido para o SiNUCA e ambos apresentam resultados semelhantes entre si.

A validação do SiNUCA é descrita em detalhes por Alves et at. (Alves et al., 2015). Os resultados desses processos se dividem em dois grupos: resultados do *benchmark* SPEC-CPU2006 e resultados de *microbenchmarks*. A validação utilizando o SPEC-CPU2006 apresenta os valores de IPC (*instructions per cycle*) para cada aplicação e os compara aos valores das máquinas reais (*Core 2 Duo* e *Sandy Bridge*). No *Core 2 Duo* o erro médio de IPC foi 19%, enquanto que no *Sandy Bridge* o erro médio foi 12%. Já os *microbenchmarks* utilizados e os resultados correspondentes são descritos a seguir:

- **Control benchmarks:** Contém diversos *microbenchmarks* desenvolvidos com o propósito de estressar o fluxo de execução por meio de desvios em excesso, assim avaliando o *branch predictor*. O erro médio de IPC foi 9% no *Core 2 Duo* e 1% no *Sandy Bridge*.

- **Dependency benchmarks:** Avalia o *forwarding* de dados utilizando cadeias de dependências entre instruções de tamanho que variam de 1 a 6. O erro médio de IPC foi 8% no *Core 2 Duo* e 1% no *Sandy Bridge*.

- **Execution benchmarks:** Estressa as unidades funcionais (`INT_ALU`, `INT_MUL`, `INT_DIV`, `FP_ALU`, `FP_MUL` e `FP_DIV`) utilizando *loops* de 32 instruções independentes que operam apenas sobre registradores, assim removendo qualquer influência que memória possa causar. O erro médio de IPC foi 1% no *Core 2 Duo* e 2% no *Sandy Bridge*.

- **Memory benchmarks:** É dividido em três partes: leitura dependente, onde listas ligadas de diversos tamanhos são percorridas; leitura independente, onde *arrays* de diversos tamanhos são lidas da memória; e escrita independente, onde dados são escritos em *arrays* de diversos tamanhos. O erro médio de IPC foi 26%, 5% e 22% para, respectivamente, leitura dependente, leitura independente e escrita independente no *Core 2 Duo* e 12%, 6% e 26% no *Sandy Bridge*.

3.3 CARACTERÍSTICAS DAS INSTRUÇÕES X86

Conforme citado anteriormente, as características, como µops relacionadas a cada instrução x86, não são divulgadas pela Intel. Portanto, cada simulador descrito nas seções anteriores definiu essas informações por meio de testes para determinar as latências e associação

manual entre as μops e as instruções. Alguns trabalhos foram desenvolvidos a fim de determinar, através das métricas disponíveis, dados sobre as instruções x86.

No trabalho feito por Fog (Fog, 2017), é apresentada uma tabela que contém diversas instruções de várias arquiteturas diferentes, e cada instrução é relacionada à sua latência, taxa de transferência, e uso de portas. Segundo o autor, as latências foram obtidas através de testes que consistiam em uma "longa cadeia de instruções idênticas, onde o resultado de cada instrução é usado como entrada da próxima instrução". Já o uso de portas foi medido através de contadores de desempenho. Porém, há diversas informações incompletas, como instruções que não possuem dados de latência ou uso de portas.

Já no trabalho mais recente publicado por Abel e Reineke (Abel e Reineke, 2019), é argumentado que além da incompletude dos dados de Fog (Fog, 2017), há também erros e imprecisões por conta de suposições incorretas feitas a partir dos dados extraídos dos contadores de desempenho que informam sobre o uso das portas. Portanto, Abel e Reineke (Abel e Reineke, 2019) propõe novas formas de se obter os mesmos dados citados anteriormente e fornece os resultados através de uma tabela[1]. Essa tabela contém as informações para as microarquiteturas mais recentes da Intel e também possui dados para todas as instruções existentes, incluindo todas as extensões modernas.

Para obter os dados aperfeiçoados, Abel e Reineke (Abel e Reineke, 2019) utilizaram *microbenchmarks* gerados automaticamente que evidenciam métricas retiradas de contadores do processador para cada instrução. A partir dessas métricas, foram definidos algoritmos e fórmulas baseados em conhecimentos de arquitetura para isolar cada uma das informações necessárias.

[1] http://www.uops.info

4 PROPOSTAS

4.1 OBJETIVOS

Como mencionado nos capítulos anteriores, a implementação do simulador OrCS define um conjunto pequeno de microinstruções e regras para a decomposição de instruções em µops. O objetivo desse trabalho é propor um novo método de representação de µops com novas formas de relacionar µops a instruções x86, provendo assim uma simulação mais precisa das unidades funcionais. Nesse processo, a simplicidade do OrCS será priorizada e a fidelidade ao comportamento das instruções x86, garantida pelos dados utilizados (Abel e Reineke, 2019), será avaliada no Capítulo 5.

4.2 MODIFICAÇÕES NO ORCS

4.2.1 Implementação Original

O OrCS, originalmente, define a relação entre os *opcodes* e as instruções através de uma simples atribuição feita no processo de geração de traços. Os *opcodes* do tipo `BRANCH`, `LOAD`, e `STORE` são atribuídas às instruções que possuem apenas operações de desvio, escrita, e leitura, respectivamente. Essas três operações são detectadas através de chamadas de funções disponíveis pela ferramenta "Pin" (Patil et al., 2004) e são especificadas nos traços, conforme explicado na Seção 2.3.2. Os outros *opcodes* (`INT_ALU`, `INT_MUL`, `INT_DIV`, `FP_ALU`, `FP_MUL`, e `FP_DIV`) fazem operações lógico aritméticas, mas elas podem também apresentar internamente acessos à memória.

De forma geral, caso um *opcode* faça apenas umas das três operações apresentadas inicialmente (`BRANCH`, `LOAD`, e `STORE`) ela será classificada com o respectivo *opcode*. Caso ela faça outras operações lógico-aritméticas, sua classificação irá refletir tal operação. Além disso, *flags* internas serão definidas indicando caso tal *opcode* deva gerar acessos a memória ou saltos.

Os *opcodes* são atribuídos através de uma relação, definida manualmente, entre o mnemônico da instrução (*iclass*) e o *opcode* correspondente. Essa atribuição também é especificada nos traços utilizando apenas um número para identificar o *opcode*. Durante a simulação do traço, cada *opcode* será convertida em uma µop, no estágio de decodificação e, possivelmente, outras µops serão geradas a partir das *flags* de operação de memória e saltos. As latências de cada µop são atribuídas às unidades funcionais através de valores definidos em arquivos de configuração.

4.2.2 Modificações na Identificação das Instruções

Devido à presente proposta de utilizar múltiplas µops por instrução em vez de um único *opcode*, o formato do arquivo de traço estático, descrito na Seção 2.3.2, teve que ser modificado. Agora, em vez do campo *iclass* conter o mnemônico simplificado, as instruções passam a ser identificadas através de um mnemônico mais complexo. O campo *opcode number* não possui mais utilidade, pois, agora, as relações entre µops e instruções são especificadas através de arquivos de configuração, cujo formato é definido na próxima seção.

Para aumentar o nível de granularidade utilizado para identificar as instruções, uma nova codificação é proposta nesse trabalho. Devido à complexidade do conjunto de instruções x86, não há um padrão bem definido que identifique cada variação possível de cada instrução.

Em nossa proposta iremos utilizar a biblioteca XED (Intel, 2019) que é capaz de codificar e decodificar instruções x86. Com essa biblioteca, é possível obter detalhes de todas as instruções necessárias para esse trabalho, como operandos, mnemônicos, *iclass* (*instruction class*), e *iform* (*instruction form*). Cada valor "*iclass*", utilizado nos traços do OrCS original, engloba diversas variações de instruções, porém não é capaz de diferenciar o comportamento de cada uma. Já o valor "*iform*", definido pela biblioteca XED, fornece mais detalhes, como os tipos dos operandos de cada instrução (registrador, memória, e imediato), e algumas variações quanto ao uso de registradores implícitos.

Na modificação introduzida ao OrCS, as instruções passaram a ser identificadas pelo *iform* concatenado às informações sobre a largura, em bits, de cada operando. Seguindo o exemplo utilizado na Seção 2.2.4, a instrução `ADD RAX, [RBX]` é representada como "`ADD_GPRv_MEMv+R64+M64`", onde "`ADD_GPRv_MEMv`" é o *iform* da instrução e "`+R64+M64`" são as informações que indicam a largura dos operandos separadas por "+", que, no caso, consistem em um registrador de 64 bits e um valor de 64 bits da memória. Essa representação é utilizada nos traços estáticos, e no arquivo de configuração interna para identificar as instruções e relacioná-las a seus respectivos µops.

Esse método para diferenciar as instruções ainda não é capaz de representar todas as variações possíveis de cada instrução. O trabalho desenvolvido por Abel e Reineke (Abel e Reineke, 2019) apresenta uma forma mais detalhada para determinar uma variante utilizando os atributos *iform*, *eosz*, *rex*, *agen*, *rep*, *zeroing*, *mask*, *bcast*, e *sae* (Abel, 2019). Alguns desses valores fornecem informações sobre o uso de registradores implícitos e funções que fogem do escopo desse trabalho. Essas informações foram ignoradas por aumentar desnecessariamente o número de instruções distintas, bem como a complexidade da proposta.

4.2.3 Modificações Internas

Com o objetivo de flexibilizar as definições de comportamento das instruções no OrCS, foi implementado uma nova forma de relacionar instruções a µops, especificar atributos das µops em si, e definir unidades funcionais. Devido ao uso extenso da biblioteca *libconfig* (Lindner, 2018) para definir variáveis relacionadas aos componentes da microarquitetura no OrCS, as modificações mencionadas também serão expressas através de arquivos de configuração no mesmo formato. Para isso, três arquivos devem ser incluídos ao executar uma simulação:

functional_units.cfg: Esse arquivo define todas as unidades funcionais e seus atributos. O campo `NAME` serve como identificação ao relacionar µops às unidades funcionais. O campo `SIZE` define a "largura" de cada unidade funcional, ou seja, a quantidade de µops que podem ser executadas simultaneamente. E `WAIT_NEXT` define o tempo de espera, em unidades de ciclo, após cada µop ser executada. O formato do arquivo é exemplificado a seguir:

```
FUNCTIONAL_UNITS = (
    { NAME = "FU_0"; SIZE = 4; WAIT_NEXT = 1; },
    { NAME = "FU_1"; SIZE = 2; WAIT_NEXT = 1; },
    { NAME = "FU_N"; SIZE = 2; WAIT_NEXT = 10; }
);
```

uops.cfg: Aqui são definidas as µops e seus atributos. Cada µop possui um nome, especificado pelo atributo `NAME`, que é utilizado para identificação ao relacionar instruções às µops. O campo `LATENCY` define a quantidade de ciclos necessários para executar uma µop, e o campo `FU` define a unidade funcional em que a µop será executada. Os valores utilizados para as unidades funcionais devem ser especificados no arquivo **funcional_units.cfg**. O formato do arquivo é exemplificado a seguir:

```
UOPS = (
    { NAME = "uop0"; LATENCY = 1; FU = "FU_N"; },
```

```
    { NAME = "uop1"; LATENCY = 9; FU = "FU_0"; },
    { NAME = "uopN"; LATENCY = 1; FU = "FU_0"; }
);
```

instructions.cfg: Esse arquivo define a relação entre as instruções e as µops. O nome de cada instrução, especificado pelo atributo `NAME`, é o mesmo valor utilizado para identificar instruções no processo de geração de traços. O atributo `UOPS` contém uma lista de µops para as quais a instrução será decodificada. Cada µop utilizada deve ser especificada no arquivo **uops.cfg**. O formato do arquivo é exemplificado a seguir:

```
INSTRUCTIONS = (
    { NAME = "INSTRUCTION_0"; UOPS = ["uop0", "uopN"] },
    { NAME = "INSTRUCTION_1"; UOPS = ["uop0"] },
    { NAME = "INSTRUCTION_N"; UOPS = ["uop1"] }
);
```

Como a proposta envolve a extração de informações sobre cada instrução individualmente para obter a decodificação em µops, não há mais a necessidade de atribuir um *opcode* às instruções, pois esse valor servia apenas como um intermediário para simplificar a decodificação que foi definida manualmente. As *flags* de operação em memória ou saltos continuam sendo utilizadas para gerar as µops correspondentes quando necessário.

Agora, com as definições das configurações necessárias e formato que será aceito pelo OrCS, basta preencher os arquivos com os dados capazes de representar as características da arquitetura real com precisão. Os processos propostos para se obter as informações desejadas para as configurações de unidades funcionais e decodificação de instruções em µops são descritos nas seções 4.3 e 4.4, respectivamente.

4.3 DEFINIÇÕES DAS UNIDADES FUNCIONAIS

Informações sobre as unidades funcionais estão disponíveis no manual oficial da Intel (Corporation, 2019). A Tabela 4.1 apresenta essas informações para a microarquitetura *Skylake*. Com isso, é possível criar o arquivo **functional_units.cfg** utilizando os mesmos nomes e tamanhos definidos pelo manual oficial, restando apenas o `WAIT_NEXT`, que foram reaproveitados da implementação original do OrCS.

Tabela 4.1: Unidades Funcionais *Skylake* (Corporation, 2019)

Unidade Funcional	Qtde.	Exemplos de Instruções
ALU	4	add, and, cmp, or, test, xor, movzx, mov...
DIV	1	divp, divs, vdiv, sqrt, vsqrt, rcp, vrcp, rsqrt, idiv...
Shift	2	sal, shl, rol, adc, sarx, adcx, adox...
Shuffle	1	(v)shufp, vperm, (v)pack, (v)unpck, (v)pshuf...
Slow Int	1	mul, imul, bsr, rcl, shld, mulx...
Bit Manipulation	2	andn, bextr, blsi, blsmsk, bzhi...
FP Mov	1	(v)movsd/ss, (v)movd gpr...
SIMD Misc	1	(v)pclmulqdq, (v)psadw...
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movap...
Vec Shift	2	(v)psllv, (v)psrlv...
Vec Add	2	(v)addp, (v)cmpp, (v)max, (v)min, (v)padds...
Vec Mul	2	(v)mul, (v)pmul, (v)pmadd...

4.4 DEFINIÇÕES DAS µOPS E DECODIFICAÇÕES DAS INSTRUÇÕES

Devido à diferença entre as diversas extensões da ISA x86, dois processos diferentes foram criados para a decodificação das instruções, um para as instruções base (instruções de propósito geral), e outro para as instruções vetoriais (i.e. SSE, AVX, e AVX-512). Esses dois grupos de instruções foram tratados separadamente pois foram observados alguns padrões relacionados à decodificação de instruções em µops que diferem entre essas duas classes. Mais detalhes sobre essa escolha são fornecidos na Seção 4.4.2.

Para fins de uniformidade, os detalhes apresentados são baseados na microarquitetura *Skylake*, porém outras microarquiteturas semelhantes poderiam ser utilizadas e resultados próximos seriam obtidos.

4.4.1 Instruções Base

Formalmente, o conjunto de instruções base considerado para esse trabalho, inclui as extensões: "Instruções de Propósito Geral", `BMI1` (*Bit Manipulation 1*), `BMI2` (*Bit Manipulation 2*), `ABM` (*Advanced Bit Manipulation*), e `x86-64`. Esses são os conjuntos de instruções presentes na família de processadores *Skylake* que operam sobre operandos de até 64 bits.

O trabalho desenvolvido por Abel e Reineke (Abel e Reineke, 2019), mencionado na Seção 3.3, fornece informações para cada instrução sobre a quantidade de µops por porta. Essas informações não apresentam dados sobre as unidades funcionais, latências individuais, ou nomenclatura descritivas das µops, porém, é possível estimá-los através de análises das combinações das portas utilizadas pelas µops e latências das instruções.

A Tabela 2.2 revela que a microarquitetura *Skylake* possui 8 portas enumeradas de 0 a 7. Além disso, nota-se que cada porta é associada a um conjunto de unidades funcionais e que pode haver diversas instâncias de uma unidade funcional. Isso indica que uma µop, será despachada por uma porta entre o conjunto de portas que estão associadas às unidades funcionais capazes de executá-la. Por exemplo, uma µop que é executada pela unidade "ALU" será despachada pela porta 0, 1, 5, ou 6, pois cada uma dessas portas possui uma instância da "ALU"; já a µop executada em "DIV" será despachada exclusivamente pela porta 0 (a qual conecta a unidade funcional de divisão).

A notação apresentada pelo trabalho de Abel e Reineke (Abel e Reineke, 2019), expõe o conjunto de portas que cada µop pode ser despachada para cada instrução. Por exemplo, a instrução `ADD RAX, RBX` é decodificada em uma µop que executa na unidade funcional "ALU", portanto a notação utilizada por essa instrução é: **p0156**, pois esses são os índices das portas associadas à essa instrução. Já uma instrução que é decodificada em duas µops de "ALU" e uma µop de "DIV" seria representada por: **2*p0156+p0**, ou seja, duas µops do tipo $p0156$ mais uma do tipo $p0$.

A partir desses dados, é possível atribuir um vetor V de inteiros à cada instrução. Esse vetor armazena 12 valores enumerados pelos seguintes subconjuntos de portas: $p0156$, $p06$, $p23$, $p237$, $p4$, $p1$, $p15$, $p015$, $p5$, $p0$, $p05$, e $p01$, pois esses são todos os valores que aparecem nos dados fornecidos por Abel e Reineke (Abel e Reineke, 2019) para o conjunto de instruções base na microarquitetura *Skylake*. O valor V_{pXYZ} indica a quantidade de µops denotadas por $pXYZ$ que estão presentes na decodificação da instrução. A Tabela 4.2 demonstra como o vetor V é formado a partir de alguns exemplos utilizados anteriormente.

A proposta para a definição da relação entre as instruções base e as µops (e.g. decodificação) consiste em utilizar as combinações de subconjuntos de portas como se fossem µops e utilizar os dados sobre o uso de portas de cada instrução para estabelecer a decodificação. Por mais que isso não descreva, de fato, as µops em si, é uma forma de criar um conjunto simples,

Tabela 4.2: Representação da notação do uso de portas em vetor.

	Vetor V											
Notação	**p0156**	**p06**	**p23**	**p237**	**p4**	**p1**	**p15**	**p015**	**p5**	**p0**	**p05**	**p01**
p0156+p23	**1**	0	**1**	0	0	0	0	0	0	0	0	0
p0156	**1**	0	0	0	0	0	0	0	0	0	0	0
2*p0156+p0	**2**	0	0	0	0	0	0	0	0	**1**	0	0

pequeno, e capaz de representar o conjunto complexo de instruções por meio de combinações diferentes, assim como a ideia original de decodificar instruções em μops na microarquitetura real. Dessa forma, é criada uma abstração do conjunto de μops suficientemente simples para o simulador, mas semelhante a implementações reais da ISA x86.

A Figura 4.1 ilustra o fluxo de execução do OrCS e da proposta desse trabalho. Nota-se que a proposta é responsável por fornecer os arquivos que serão utilizados pelo OrCS ao receber os traços. Tanto o gerador de traços quanto o gerador de instruções fazem uso da biblioteca XED (Intel, 2019) visto que ambos precisam estar em consenso ao identificar as instruções. Enquanto o gerador de traços descreve o executável ao traduzir as instruções x86 no formato aceito pelo OrCS, o gerador de instruções descreve o processo de decodificação de cada instrução aceita pelo OrCS. Portanto o formato descrito na Seção 4.2.2 é adotado em ambos os componentes.

Para gerar os arquivos que descrevem a relação entre as instruções, as μops, e as unidades funcionais, são necessárias algumas especificações que fornecem algumas informações associadas à arquitetura, como a configuração de unidades funcionais e a relação entre os conjuntos de portas, as nomenclaturas das μops, e as unidades funcionais. A Figura 4.2 ilustra um exemplo de associação utilizada pela proposta. O método utilizado para obter essas informações será descrito a seguir, e o motivo do formato diferente para as μops das instruções vetoriais será descrito na Seção 4.4.2.

A Figura 4.2 mostra que para as instruções base, cada conjunto de portas será associado através de uma relação um-para-um a uma μop que possui um nome único e uma latência. E cada μop será associada à unidade funcional que a executará. Com essas informações, é possível associar um conjunto de μops a cada instrução x86 utilizando os vetores *V* fornecidos por Abel e Reineke (Abel e Reineke, 2019).

A ideia descrita resulta em 12 μops diferentes para as instruções base, porém o arquivo **uops.cfg** ainda exige as latências de cada μop e a relação entre μop e unidade funcional. Para isso, uma análise mais aprofundada é necessária, contudo, a falta de informações oficiais exige, em alguns casos, especulações e suposições. Para auxiliar na atribuição de unidades funcionais aos subconjuntos de portas (i.e. μops), foram utilizadas as Tabelas 4.1 e 4.3 retiradas do manual de otimização da Intel (Corporation, 2019). Aqui são apresentados os processos utilizados para obter essas informações para cada subconjunto de portas:

- **p0156**: Observando a Tabela 4.3, é possível notar que a operação "ALU" (*Arithmetic Logic Unit*) é a única operação em comum entre as portas 0, 1, 5, e 6, e essa operação aparece exclusivamente nessas 4 portas. Portanto, é possível concluir que $p0156$ corresponde à unidade funcional "ALU". Quanto à latência, os dados revelam que todas as instruções do conjunto base, que é decodificada em uma única μop executada no subconjunto de portas $p0156$, possuem latência de 1 ciclo. Portanto conclui-se que a μop relacionada a esse subconjunto possui latência 1 de ciclo. O nome da μop escolhido foi `INT_ALU`.

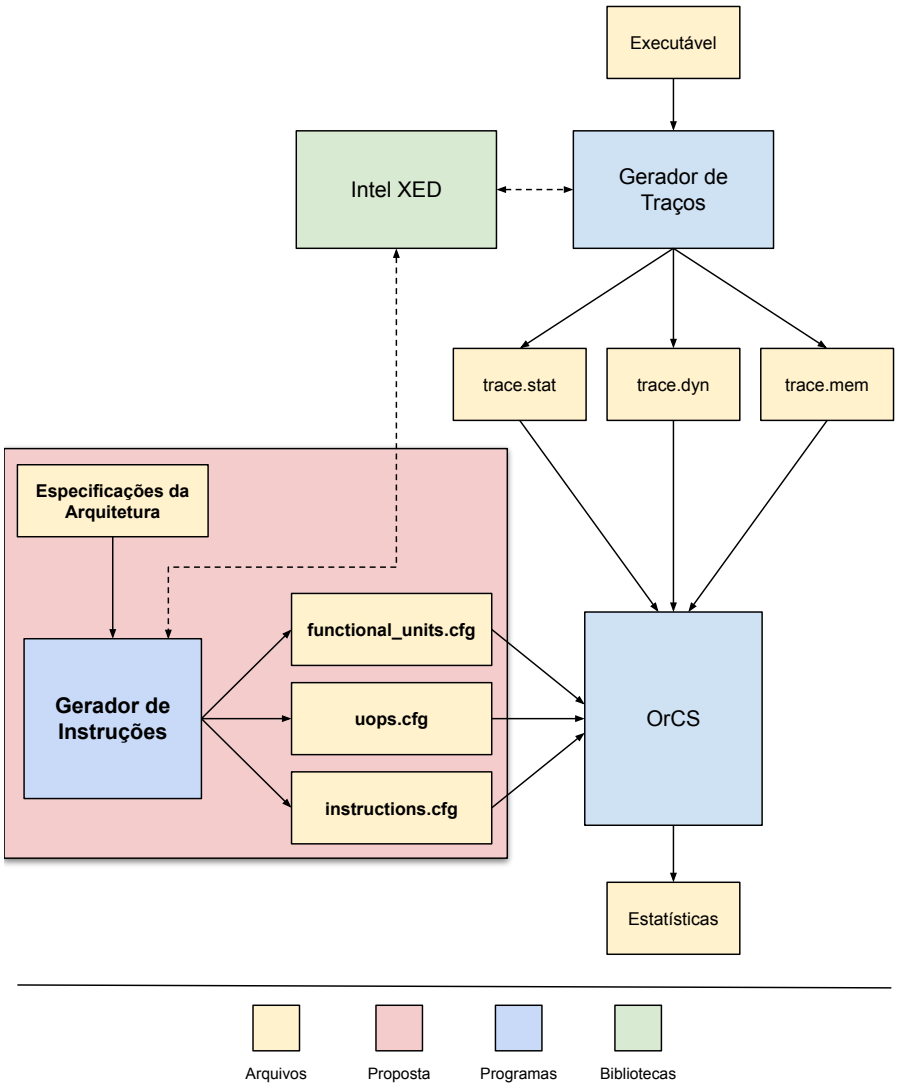


Figura 4.1: Diagrama do fluxo de execução do OrCS e da proposta

Tabela 4.3: Portas e Operações da microarquitetura Intel *Skylake* (Corporation, 2019)

Port 0	**Port 1**	**Port 2**	**Port 3**	**Port 4**	**Port 5**	**Port 6**	**Port 7**
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	Load, AGU	Load, AGU	Store	ALU, Fast LEA, Vec ALU	ALU, Shift	AGU
Vec Shift, Vec Add	Vec Shift, Vec Add				Vec Shuffle	Branch	
Vec Mul, FMA	Vec Mul, FMA						
DIV	Slow Int						
Branch	Slow LEA						

- **p1** e **p15**: Uma análise semelhante à descrita para o subconjunto $p0156$, revela que os valores $p1$ e $p15$ correspondem às unidades funcionais "Slow Int" e "Bit Manipulation", respectivamente, e possuem latências de 3 e 1 ciclos. Os nomes das µops escolhidos foram `MULT` para $p1$ e `FAST_LEA` para $p15$.

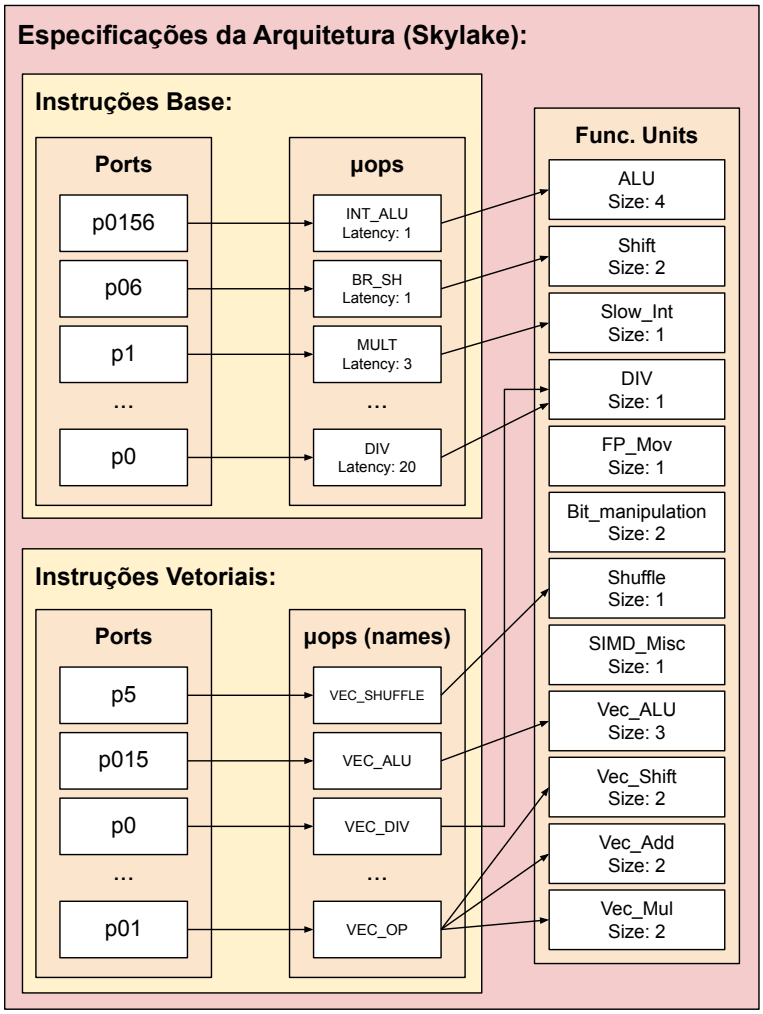


Figura 4.2: Especificações das relações utilizadas pela proposta

- **p06**: Analisando a Tabela 4.3 descobre-se que a operação "Branch" está associada às portas 0 e 6. Além disso, a partir dos dados, conclui-se que a operação "Shift" também está relacionada a esse subconjunto, pois instruções como `SHR`, `ADC`, e `ROL`, de acordo com a Tabela 4.1, executam na unidade de Shift e possuem $p06$ como única µop. Portanto a µop representada por esse subconjunto está associada à unidade funcional "Shift" e possui latência de 1 ciclo, pela mesma análise descrita anteriormente. O nome da µop escolhido foi `BR_SH`.

- **p23**, **p237**, e **p4**: De acordo com a Tabela 4.3, esses subconjuntos possuem a operação de *Load*, AGU (*Address Generation Unit*), e *Store*, respectivamente. Essas operações não fazem parte das unidades funcionais descritas na Tabela 4.1, pois são executadas por unidades funcionais específicas da memória. A implementação do OrCS, conforme explicado anteriormente, trata essas operações separadamente e possui unidades funcionais configuradas internamente que as executam. O motivo dessa separação se deve ao fato de que essas operações envolvem outros componentes integrados na implementação do OrCS que dependem de informações do uso da memória para cada instrução. Portanto, para evitar redundâncias e modificações desnecessárias, esses subconjuntos serão ignorados nos novos arquivos de configuração e continuarão sendo identificados pela geração de traços original.

- **p015**: Esse subconjunto, de acordo com a Tabela 4.3, está relacionado à operação "Vec ALU", e, consequentemente, à unidade funcional de mesmo nome, ou seja, ALU vetorial. Apesar dessa análise conter apenas o conjunto de instruções base, esse subconjunto aparece, inesperadamente, em algumas poucas instruções, como `BTC`, `BTS`, `DIV`, entre outras. Em todos esses exemplos de instruções, o subconjunto $p015$ aparece em apenas algumas combinações específicas de operandos, geralmente envolvendo operações de escrita em memória, revelando a complexidade e inconsistência da ISA x86. Estima-se que essa µop possui latência de 1 ciclo, por análise semelhante à descrita anteriormente. O nome da µop escolhido foi `VEC_ALU`.

- **p5** e **p0**: Esses subconjuntos correspondem às operações "Shuffle" e "DIV", e serão relacionados às unidades funcionais de mesmo nome. Assim como $p015$, esse subconjunto é utilizado apenas por algumas instruções mais complexas. As latências das µops correspondentes não puderam ser estimadas, pois nenhuma instrução contém esses subconjuntos isolados. Portanto, os valores foram definidos manualmente como 1 e 20 ciclos, respectivamente, para aproximar as latências das instruções que utilizam esses subconjuntos após subtrair as latências dos outros valores já definidos. Os nomes das µops escolhidos foram `SHUFFLE` para $p5$ e `DIV` para $p0$.

- **p05** e **p01**: Não foi possível aplicar a mesma análise a esses subconjuntos, pois a Tabela 4.3 não revela nenhuma operação para $p05$ e possui ambiguidade para $p01$. Assim como $p015$, $p5$, e $p0$, esses subconjuntos aparecem em apenas algumas instruções específicas. As unidades funcionais relacionadas serão definidas como "ALU" para $p05$, pois é uma operação presente nessas portas; e "Vec Shift" para $p01$, pois é uma das opções disponíveis. E as latências são de 8 e 1 ciclos, de acordo com o mesmo processo utilizado em $p5$ e $p0$. Os nomes das µops escolhidos foram `UNK` para $p05$ e `VEC_SHI` para $p01$.

Com essas informações sobre as portas e unidades funcionais é possível obter o arquivo **uops.cfg**. Os resultados discutidos anteriormente são apresentados na Tabela 4.4. A partir dessas informações, é possível criar a decodificação de cada instrução utilizando os dados descritos anteriormente.

Foi observado, através de análises sobre os dados, que, ao utilizar o vetor V como vetor de booleanos, ou seja, valores maiores que 1 são definidos como 1, resultados semelhantes são obtidos. Isso ocorre pois a grande maioria das instruções não possui mais de uma µop do mesmo tipo. E instruções como `DIV`, que apresentam diversas µops do mesmo tipo, tiveram as latência compensadas pelos valores definidos no processo descrito acima. Por exemplo, a µop $p0$ possui latência 20, que é um valor que compensa a complexidade da distribuição do resto das µops. Essa decisão simplifica ainda mais o processo de decodificação das instruções no OrCS e facilita a definição dos valores de latências das µops.

Uma das limitações dessa proposta é o fato que as múltiplas µops definidas para as instruções base não possuem uma ordem definida de execução. A única ordem de execução estabelecida na implementação do OrCS é que a µop de *Load* precede as µops definidas de operação, que precedem a µop de *Store*. Porém, estima-se que na implementação da microarquitetura real, há um algoritmo, envolvendo as µops de operação, responsável por computar a operação definida pelas instruções.

Tabela 4.4: Definições de µops

Portas	**Nome µop** (NAME)	**Latencia** (LATENCY)	**Unidade Funcional** (FU)
p0156	INT_ALU	1	ALU
p06	BR_SH	1	Shift
p1	MULT	3	Slow Int
p15	FAST_LEA	1	Bit Manipulation
p015	VEC_ALU	1	Vec ALU
p5	SHUFFLE	1	Shuffle
p0	DIV	20	DIV
p05	UNK	8	ALU
p01	VEC_SHI	1	Vec Shift
p6	SHI	1	ALU

4.4.2 Instruções Vetoriais

Nesse trabalho, o conjunto de instruções vetoriais foi definido como as extensões AVX2, FMA, SSE, SSE2, SSE3, SSE4, e SSSE3. As instruções AVX-512 serão tratadas separadamente na Seção 4.4.3. Ao contrário das instruções base, as instruções vetoriais, em geral, são mais simples em termos de quantidade de µops por instrução. A grande maioria das instruções vetoriais possuem apenas uma µop além das µops de *Load*, *AGU*, e *Store*. Essa observação, por mais que não se aplique a todas as instruções, permite uma abordagem semelhante à utilizada na implementação do OrCS original, ou seja, definir uma única µop por instrução, separada das operações de memória. Definiremos essa µop única de uma instrução como "µop unitária" de uma instrução. Assim como no método utilizado para as instruções base, as µops serão criadas a partir dos subconjuntos de portas.

Apesar do trabalho de Abel e Reineke (Abel e Reineke, 2019) também incluir essas extensões e fornecer os mesmos dados sobre os usos de portas para todas as instruções que as compõe, as análises preliminares utilizadas para as instruções base não apresentam bons resultados para esse conjunto. Dentre os problemas observados, constata-se a alta variância nas latências entre as instruções que possuem determinados subconjuntos isolados. Por exemplo, as instruções que usam uma única µop que executa na porta $p0$ possuem latências que variam de 1 a 19 ciclos. Portanto, tratar "$p0$" como uma µop traria inconsistências ao definir um valor de latência.

Para simplificar o uso de portas das instruções vetoriais, define-se o conceito de "subconjunto representativo". Um subconjunto representativo de uma instrução é o subconjunto de maior valor dessa instrução, excluindo $p23$, $p237$, e $p4$. Ou seja, é o subconjunto que possui mais µops com exceção das operações de acesso à memória. Por exemplo, a instrução vetorial `VADDPD XMM, XMM, M128`, executa a operação de adição vetorial `XMM` ← `XMM` + `M128`, onde `XMM` é um registrador de 128-bits e `M128` é um valor da memória de 128-bits, e possui o vetor de uso de portas ilustrado na Tabela 4.5. Nesse exemplo o subconjunto representativo é $p01$, pois, excluindo o subconjunto $p23$, o subconjunto $p01$ possui o maior valor de latência. Essa definição é semelhante ao conceito de µop unitária definido anteriormente, porém, engloba também os casos raros onde uma instrução possui mais de uma µop de operação.

Tabela 4.5: Uso de portas da instrução vetorial `VADDPD (XMM, XMM, M128)`

p0156	**p06**	**p23**	**p237**	**p4**	**p1**	**p15**	**p015**	**p5**	**p0**	**p05**	**p01**
0	0	1	0	0	0	0	0	0	0	0	1

Nota-se no exemplo acima que o subconjunto representativo da instrução indica, de acordo com a Tabela 4.3, o uso da unidade funcional “Vec Shift”, “Vec Add”, ou “Vec Mul”. Não é possível diferenciar entre essas três opções, pois as três são exclusivas das portas 0 e 1. Mas, tendo o subconjunto representativo de cada instrução, é possível definir um conjunto de unidades funcionais possíveis por subconjunto, a Tabela 4.6 ilustra essa relação. O método utilizado para montar essa relação foi o mesmo utilizado para as instruções base. Alguns subconjuntos, como p0156, não estão presentes nessa relação, pois não aparecem como subconjunto representativo entre as instruções vetoriais analisadas.

Tabela 4.6: Relação subconjuntos representativos e unidades funcionais

Subconjunto representativo	**Unidades funcionais**
p06	Shift
p1	Slow Int
p015	Vec ALU
p5	Shuffle
p0	DIV
p05	Vec Add
p01	Vec Shift, Vec Add, Vec Mul

O método escolhido para selecionar as unidades funcionais nos casos onde há mais de uma opção por subconjunto representativo, mais especificamente no caso “p01”, é criar uma µop para cada unidade funcional e utilizar os exemplos de instruções por unidade funcional, fornecidos pela versão completa da Tabela 4.1, para associar as instruções a cada µop através de semelhanças no mnemônico da instrução. Por exemplo, na tabela, um dos exemplos da unidade funcional “Vec Mul” é a instrução `PMADD`, portanto a instrução `PMADDUBSW`, que possui o subconjunto representativo “p01”, será relacionada a uma µop que é associada à unidade funcional “Vec Mul”.

Ao listar os valores distintos das latência das instruções por subconjunto representativo, a Tabela 4.7 é obtida. Com esses dados, observa-se a impossibilidade de atribuir uma latência a cada subconjunto ao definir uma µop da mesma forma que foi feito para as instruções base. Uma alternativa seria criar uma µop para cada latência de cada subconjunto, isso resultaria em 46 µops mais as unidades replicadas para lidar com a ambiguidade de unidades funcionais descrita no parágrafo anterior. A quantidade de latências distintas pode ser ainda maior para microarquiteturas diferentes ou extensões x86 mais complexas, como o AVX-512. O alto número de µops opõe a premissa de manter a simplicidade do OrCS, portanto um algoritmo é proposto para agrupar alguns valores de latências em G valores distintos, onde G é um parâmetro do algoritmo.

A tarefa de agrupar as latências é um problema de otimização, pois, ao modificar os valores das latências, informações serão perdidas, portanto é necessário definir um critério para que as modificações impliquem na menor perda possível. Para isso, cada latência de cada subconjunto representativo é atribuída à contagem de vezes que as instruções com essa latência e esse subconjunto representativo foram executadas no conjunto de *benchmarks* SPEC CPU2017 (Bucek et al., 2018). Essa opção foi utilizada para tentar representar o uso das instruções em aplicações reais. Os dados das instruções vetoriais são apresentados na Tabela 4.8. Com isso, o algoritmo deve tentar manter as latências das instruções que foram executadas mais vezes no total, pois modificações nesses valores resultariam em erros maiores nos resultados das simulações.

A Figura 4.3 ilustra o fluxo do algoritmo de agrupamento de latências executado sobre as instruções vetoriais com G definido arbitrariamente como 20. Ou seja, as 46 combinações de

Tabela 4.7: Ocorrências de valores de latências por subconjunto representativo

Subconjunto rep.	**Latências**	**Qtde.**
p01	[1, 4, 5, 8, 9, 10, 13]	7
p015	[1, 2, 3]	3
p0	[1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 18, 19]	16
p1	[1, 3]	2
p5	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 17]	13
p06	[7, 35, 36]	3
p05	[1, 2]	2
		Total: 46

Tabela 4.8: Latência (L) e Contagem* (C) de instruções vetoriais por subconjunto representativo

p01		**p015**		**p0**		**p1**		**p5**		**p06**		**p05**	
L	**C**	**L**	**C**	**L**	**C**	**L**	**C**	**L**	**C**	**L**	**C**	**L**	**C**
1	93403	1	6609927	1	614933	1	0	1	61904	7	115592	1	0
4	57943214	2	0	2	0	3	0	2	0	8	0	2	0
5	0	3	0	3	6162267			3	47147	35	0		
8	3434			4	7753			4	162374				
9	0			5	59			5	370687				
10	0			6	0			6	21				
13	0			7	275830			7	4347814				
				9	0			8	2				
				10	0			9	0				
				11	290946			10	0				
				12	646017			12	0				
				13	32749			16	0				
				14	107162			17	0				
				15	729859								
				18	1330								
				19	185673								

* Contagem dividida por 100000 e truncada para aumentar legibilidade

latências e subconjuntos representativos serão comprimidos em apenas 20 classes. Observa-se que a entrada do algoritmo consiste em duas matrizes, uma contém os valores de latências na mesma ordem que foi apresentada na Tabela 4.7 e a outra contém os valores de contagem de instruções (i.e., número de ocorrências) por latência de cada subconjunto representativo na mesma ordem da primeira tabela, ou seja, os mesmos dados da Tabela 4.8. Para cada subconjunto representativo (i.e. linha das matrizes na Figura 4.3 e coluna na Tabela 4.8), o algoritmo agrupa as latências priorizando os valores de maior contagem, e então, define um novo valor de latência para cada grupo.

O Algoritmo 1 resolve o problema de agrupamento minimizando uma função de custo. Essa função de custo é aplicada sobre um intervalo de latências de uma linha da matriz para calcular a diferença entre o novo valor de latência que será definido caso esse intervalo seja agrupado e os valores de latências originais. A função recebe como parâmetro o índice i da linha da matriz e o intervalo sobre os índices das colunas definido por $[L, R]$. E retorna o valor absoluto da diferença entre os produtos escalares entre as duas matrizes no intervalo especificado, exceto que, em um dos termos, a nova latência new_lat é utilizada. Com isso, a função de custo é definida pela seguinte fórmula:

$$COST(i, L, R) = |\sum_{j=L}^{R}(new\_lat \times count_{ij}) - \sum_{j=L}^{R}(lat_{ij} \times count_{ij})|$$

Onde lat_i e $count_i$ são as latências e contagens, respectivamente, da i-ésima linha, e new_lat é definido por:

$$new\_lat = \frac{1}{\sum_{j=L}^{R} count_{ij}} \times \sum_{j=L}^{R} lat_{ij} \times count_{ij}$$

Ou seja, a média ponderada pelas contagens das latências no intervalo especificado.

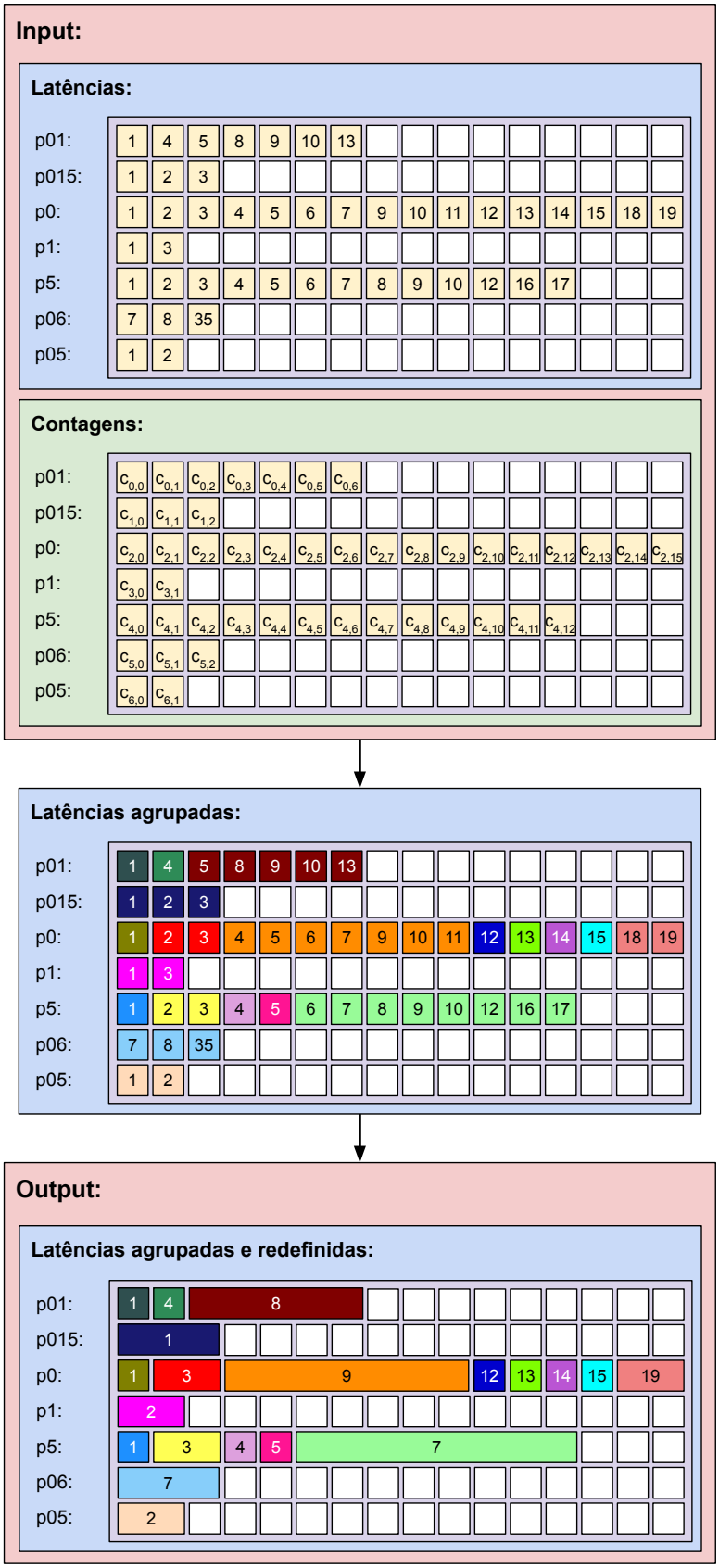


Figura 4.3: Entrada e saída do algoritmo

Algoritmo 1 Agrupamento de latências, minimizando a função de custo

Entrada:

i: índice do i-ésimo subconjunto representativo (inicialmente 0)
j: índice do j-ésimo valor de latência/contagem (inicialmente 0)
G: quantidade de grupos de latências (μops) desejada

Saída:

Valor mínimo da soma de custos do agrupamento

```
1:  function SOLVE(i, j, G)
2:      if i = N then
3:          if G ≠ 0 then
4:              return ∞
5:          end if
6:          return 0
7:      end if
8:
9:      M ← |lat_i|
10:     min ← ∞
11:     R ← 0
12:     for k ← j to (M − 1) do
13:         next_i ← i + (k+1)/M
14:         next_j ← (k + 1) mod M
15:         x ← SOLVE(next_i, next_j, G − 1) + COST(i, j, k)
16:         if x < min then
17:             min ← x
18:             R ← k
19:         end if
20:     end for
21:
22:     // O intervalo lat_i[j..R] possui o menor custo ao ser agrupado
23:     return min
24: end function
```

O algoritmo percorre todos os valores de latência de todos os N subconjuntos e calcula o custo de todas as possibilidades de agrupamento recursivamente, marcando a opção de menor custo. É possível otimizar o tempo de execução do algoritmo utilizando a técnica de programação dinâmica, tornando sua complexidade $O(N \times M^2 \times G)$, onde M é o tamanho do maior subconjunto representativo. Além disso, também é necessário reconstruir a solução para que os resultados sejam obtidos, isso é feito através de um procedimento padrão de análise sobre os resultados da tabela da programação dinâmica. As latências dos intervalos selecionados são definidas da mesma forma utilizada em "*new_lat*", ou seja, a média ponderada das latências. Essas modificações não foram incluídas no Algoritmo 1 para manter a simplicidade do código.

Após a execução do Algoritmo 1 com $G = 20$, a saída *"Output"* ilustrado na Figura 4.3 é obtido. Ou seja, o resultado consiste na mesma matriz de latências da entrada, porém com alguns intervalos agrupados com novas latências em cada linha. É possível notar na Tabela 4.8 que as latências permanecem as mesmas para os valores que possuem contagens alta e são alteradas para contagens baixas. Por exemplo, os últimos valores do subconjunto representativo $p5$ foram todos agrupados e definidos como 7, pois as contagens desses valores são 0. Nota-se que o valor de G define implicitamente o limiar entre "contagens altas" e "contagens baixas".

Tendo essa matriz como resultado, o processo de geração de µops é bem simples. Para cada grupo, uma µop será gerada com a latência definida e a unidade funcional referente ao subconjunto representativo do grupo. Então, as instruções vetoriais que possuem a latência L e subconjunto representativo S serão decodificadas, no OrCS, para a µop gerada a partir do grupo que contém a latência L e que está associado ao subconjunto representativo S.

Todo o processo descrito até agora para a geração de instruções para o OrCS está implementado em um programa disponibilizado publicamente[1]. Esse programa foi desenvolvido em *Python* exclusivamente para esse trabalho e para ser utilizado e adaptado futuramente. Ele é responsável pela análise dos dados das instruções, definições de informações sobre a arquitetura, execução do algoritmo, geração das instruções para o OrCS, e geração dos arquivos definidos anteriormente.

O resultado para as instruções vetoriais é apresentado na Tabela 4.9. Os nomes foram criados para serem descritivos de acordo com a operação realizada. Nota-se que, conforme explicado anteriormente, as µops geradas pelo subconjunto representativo "$p01$" foram replicadas para tratar as múltiplas unidades funcionais, resultando em mais de 20 µops.

Tabela 4.9: Definições de µops para instruções vetoriais (*Skylake*), cores utilizadas na Figura 4.3

Portas (Sub. Rep.)	**Nome µop** (NAME)	**Latência** (LATENCY)	**Unidade Func.** (FU)
p01	VEC_OP_0_0	**1**	Vec_Add
p01	VEC_OP_0_1	**1**	Vec_Shift
p01	VEC_OP_0_2	**1**	Vec_Mul
p01	VEC_OP_1_0	**4**	Vec_Add
p01	VEC_OP_1_1	**4**	Vec_Shift
p01	VEC_OP_1_2	**4**	Vec_Mul
p01	VEC_OP_2_0	**8**	Vec_Add
p01	VEC_OP_2_1	**8**	Vec_Shift
p01	VEC_OP_2_2	**8**	Vec_Mul
p015	VEC_ALU_0	**1**	Vec_ALU
p0	VEC_DIV_0	**1**	DIV
p0	VEC_DIV_1	**3**	DIV
p0	VEC_DIV_2	**9**	DIV
p0	VEC_DIV_3	**12**	DIV
p0	VEC_DIV_4	**13**	DIV
p0	VEC_DIV_5	**14**	DIV
p0	VEC_DIV_6	**15**	DIV
p0	VEC_DIV_7	**19**	DIV
p1	VEC_MULT_0	**2**	Slow_Int
p5	VEC_SHUFFLE_0	**1**	Shuffle
p5	VEC_SHUFFLE_1	**3**	Shuffle
p5	VEC_SHUFFLE_2	**4**	Shuffle
p5	VEC_SHUFFLE_3	**5**	Shuffle
p5	VEC_SHUFFLE_4	**7**	Shuffle
p06	VEC_BR_SH_0	**7**	Shift
p05	UNK_0	**2**	Vec_ALU

[1]https://github.com/BrunoTissei/OrCS-instruction-generator

4.4.3 Instruções Vetoriais AVX-512

As instruções da extensão AVX-512 foram tratadas separadamente, pois, de acordo com o manual de otimização da Intel (Corporation, 2019), há um tratamento especial para essas instruções. Além da adição da unidade funcional `Vec_FMA`, as μops que operam sobre registradores de 512 bits são executadas pela fusão das unidades funcionais relacionadas às portas $p0$ e $p1$. Ou seja, as unidades funcionais que executam as instruções vetoriais de operandos de 256 bits se unem para executar as instruções AVX-512. Esse processo não é compatível com o método de decodificação proposto nesse trabalho e não há nada implementado no OrCS que permitiria a fusão de unidades funcionais. Portanto a decisão tomada para simular esse comportamento foi utilizar o algoritmo proposto para as instruções vetoriais, porém decodificar as instruções AVX-512 em duas μops com a mesma latência em vez de uma, ambas μops com as mesmas dependências de entrada (dependências com instruções anteriores no traço) e gerando a mesma dependência de saída (instruções posteriores no traço).

Além desta particularidade descrita, há algumas complicações relacionadas a versões de processadores diferentes. Por exemplo, o manual de otimização da Intel afirma que, além da fusão das unidades funcionais vetoriais para executar instruções de 512 bits, há uma unidade dedicada para instruções AVX-512. Porém, de acordo com algumas fontes (Wikichip, 2016d), essa unidade está presente apenas em modelos "topo de linha", como *Intel Xeon Gold* e *Intel Xeon Platinum*. Os resultados apresentados no Capítulo 5 não incluem arquiteturas com essa unidade dedicada, mas para suportá-la, seriam necessárias algumas modificações, pois, nesse caso, a técnica de "fusão de unidades funcionais" só é utilizada para as μops que não executam na unidade adicional.

O algoritmo proposto para as instruções vetoriais deve ser executado também para as instruções AVX-512, gerando três conjuntos de μops que são tratadas da mesma forma (instruções base, vetoriais, e AVX-512). Os resultados da execução para as instruções AVX-512 são apresentados na Tabela 4.10.

Tabela 4.10: Definições de µops para instruções AVX-512 (*Skylake Server*)

Portas (Sub. Rep.)	**Nome µop** (`NAME`)	**Latencia** (`LATENCY`)	**Unidade Func.** (`FU`)
p01	VEC_OP2_0_0	1	Vec_Add
p01	VEC_OP2_0_1	1	Vec_Shift
p01	VEC_OP2_0_2	1	Vec_Mul
p01	VEC_OP2_0_3	1	Vec_FMA
p01	VEC_OP2_1_0	4	Vec_Add
p01	VEC_OP2_1_1	4	Vec_Shift
p01	VEC_OP2_1_2	4	Vec_Mul
p01	VEC_OP2_1_3	4	Vec_FMA
p01	VEC_OP2_2_0	8	Vec_Add
p01	VEC_OP2_2_1	8	Vec_Shift
p01	VEC_OP2_2_2	8	Vec_Mul
p01	VEC_OP2_2_3	8	Vec_FMA
p015	VEC_OP1_0_0	1	Vec_Add
p015	VEC_OP1_0_1	1	Vec_Shift
p015	VEC_OP1_0_2	1	Vec_Mul
p015	VEC_OP1_0_3	1	Vec_FMA
p015	VEC_OP1_0_4	1	Vec_ALU
p0	VEC_DIV_0	1	DIV
p0	VEC_DIV_1	5	DIV
p0	VEC_DIV_2	11	DIV
p0	VEC_DIV_3	12	DIV
p0	VEC_DIV_4	13	DIV
p0	VEC_DIV_5	14	DIV
p0	VEC_DIV_6	15	DIV
p0	VEC_DIV_7	19	DIV
p1	MULT_0	4	Slow_Int
p5	VEC_SHUFFLE_0	1	Shuffle
p5	VEC_SHUFFLE_1	3	Shuffle
p5	VEC_SHUFFLE_2	5	Shuffle
p5	VEC_SHUFFLE_3	7	Shuffle
p05	UNK_0	1	Vec_ALU
p05	UNK_1	4	Vec_ALU
p05	UNK_2	5	Vec_ALU

5 RESULTADOS E DISCUSSÕES

Neste capítulo os resultados serão apresentados e discutidos a partir da proposta mencionada no Capítulo 4. Os experimentos foram escolhidos para demonstrar as diferenças entre a máquina real, o simulador OrCS original, e a versão do OrCS com as modificações descritas no capítulo anterior. Aqui também serão apresentadas algumas análises e possíveis justificativas para os resultados obtidos.

O simulador visa simular os componentes com acurácia na quantidade de ciclos e as modificações propostas impactam diretamente nas latências dos componentes. Portanto, a métrica utilizada nos testes é o IPC (*Instructions per Cycle*), ou seja, a razão entre a quantidade de instruções e a quantidade de ciclos necessários para concluir a execução do teste. O objetivo, então, é aproximar o valor de IPC da simulação do valor de IPC da execução na máquina real.

Para esses experimentos, foram utilizadas duas máquinas diferentes com microarquiteturas semelhantes. Mais detalhes sobre as configurações das duas máquinas, utilizadas também no simulador, se encontram na Tabela 5.1:

- **Intel Core i5-7400 @ 3.00GHz:** *Desktop* de microarquitetura *Kaby Lake*, cujo pipeline é idêntico ao do *Skylake* (Wikichip, 2016b). Essa máquina foi utilizada exclusivamente para os *microbenchmarks* apresentados na Seção 5.1, com exceção das instruções AVX-512.
- **Intel Xeon Silver 4214 @ 2.20GHz:** *Servidor* de microarquitetura *Cascade Lake*, cujo pipeline é, em grande parte, idêntico ao do *Skylake*. Possui apenas algumas pequenas modificações, mais especificamente, duas unidades funcionais que executam algumas instruções AVX-512 novas (Wikichip, 2016a). Essa máquina, ao contrário da outra, suporta instruções AVX-512. Foi utilizada para os *microbenchmarks* apresentados na Seção 5.1 e para os *benchmarks* SPEC CPU2017 apresentados na Seção 5.2.

Entre as modificações propostas, um fator impactante nos resultados é a configuração das unidade funcionais. A comparação das unidades funcionais entre as versões do OrCS, já apresentada e explicada no capítulo anterior, é sumarizada na Tabela 5.2.

5.1 MICROBENCHMARKS

Os *microbenchmarks* consistem em testes para determinar o comportamento de componentes específicos. Foram utilizados os mesmos *microbenchmarks* desenvolvidos para o processo de validação do simulador SiNUCA (Alves, 2014), também utilizados pelo OrCS (Köhler, 2019), porém algumas variações foram adicionadas para acentuar as diferenças geradas pelas modificações no OrCS. Esses testes foram implementados na linguagem C, porém foram utilizadas instruções *assembly inline* para garantir a execução das instruções desejadas. Os programas foram compilados utilizando o *GCC 9.3.0* e código completo está disponível publicamente[1].

Cada *microbenchmark* que será apresentado foi executado 1000 vezes na máquina real para obter os valores de IPC, sendo que cada teste possui um *loop* de 1 milhão de iterações que executa as instruções desejadas. As métricas foram obtidas utilizando a ferramenta `perf`.

[1]https://github.com/BrunoTissei/microbenchmarks

Tabela 5.1: Configurações das máquinas reais usadas como parâmetros no simulador

Parameter	**Intel Core i5-7400 @ 3.00GHz**	**Intel Xeon Silver 4214 @ 2.2GHz**
Fetch Width	6	6
Decode Width	5	5
Rename Width	5	5
Dispatch Width	6	6
Execute Width	8	8
Commit Width	4	4
Fetch Buffer	40	40
Decode Buffer	128	128
RAT size	260	260
ROB size	224	224
MOB size (read)	72	72
MOB size (write)	56	56
L1 instr. cache	32KB, 8-way	32KB, 8-way
L1 data cache	32KB, 8-way	32KB, 8-way
L2 data cache	256KB, 4-way	1MB, 16-way
L3 data cache	6MB, 12-way	16MB, 11-way
Cache line size	64B	64B

Tabela 5.2: Configurações de Unidade Funcionais OrCS

OrCS Original		**OrCS Modif. (desktop)**		**OrCS Modif. (server)**	
Unid. Func.	**Largura**	**Unid. Func.**	**Largura**	**Unid. Func.**	**Largura**
INT_ALU	4	ALU	4	ALU	4
INT_MUL	1	Bit Manip.	2	Bit Manip.	2
INT_DIV	1	DIV	1	DIV	1
FP_ALU	2	FP Mov	1	FP Mov	1
FP_MUL	2	SIMD Misc	1	SIMD Misc	1
FP_DIV	1	Shift	2	Shift	2
LOAD_UNIT	2	Shuffle	1	Shuffle	1
STORE_UNIT	1	Slow Int	1	Slow Int	1
		Load Unit	2	Load Unit	2
		Store Unit	1	Store Unit	1
		Vec ALU	3	Vec ALU	3
		Vec Add	2	Vec Add	2
		Vec Mul	2	Vec Mul	2
		Vec Shift	2	Vec Shift	2
				Vec FMA	2

5.1.1 Controle

O conjunto de *microbenchmarks* de controle contém testes relacionados a desvios condicionais. O objetivo desses testes é validar o mecanismo de previsão de desvios. Para isso, esse conjunto contém diversos programas que executam desvios condicionais utilizando comportamentos diferentes (Alves, 2014):

- **Control Complex:** Executa *switch* e *if-then-else* de uma forma complexa e difícil de prever em *loop*;
- **Control Conditional:** Executa um *if-then-else* dentro de um *loop*, onde o desvio é tomado alternadamente;
- **Control Random:** Executa um *if-then-else* em *loop*, onde, a cada iteração, o desvio é tomado aleatoriamente;
- **Control Small BBL:** Executa um *loop* simples, sendo a condição do *loop* o único desvio condicional.
- **Control Switch:** Executa um comando *switch* com 10 *cases* dentro de um *loop*, onde cada *case* é executado $n/10$ vezes;

A falta de informações acerca de valores associados à previsão de desvios impõe limitações na implementação do simulador. De acordo com a Intel, houve uma melhoria na latência da penalidade do erro de predição na microarquitetura *Skylake* com relação às versões anteriores, porém não há informações sobre os valores específicos ou estratégias e algoritmos utilizados.

Apesar desse trabalho não propor nenhuma modificação no mecanismo de previsão de desvios, a inclusão dos resultados desse conjunto de *microbenchmarks* demonstra o impacto no IPC causado pelos componentes envolvidos nesse mecanismo. Portanto espera-se que comparações entre simulações e execuções em máquinas reais de aplicações mais complexas, como *benchmarks* completos, apresentem diferenças significantes em diversas métricas devido ao uso extenso de instruções de desvio ou outros componentes difíceis de replicar.

Para demonstrar o impacto do mecanismo de previsão de desvios, uma comparação do IPC entre o OrCS e a execução real dos *microbenchmarks* de controle para diferentes valores latência da penalidade do erro de predição é apresentada na Figura 5.1.

Na Figura 5.2, a latência da penalidade foi definida para 8 ciclos, e é apresentada a comparação entre a máquina real, a implementação do OrCS original e o modificado, demonstrando que não houve diferenças significativas no IPC, pois a grande maioria das instruções utilizadas nesses *microbenchmarks* não sofreram modificações na latência.

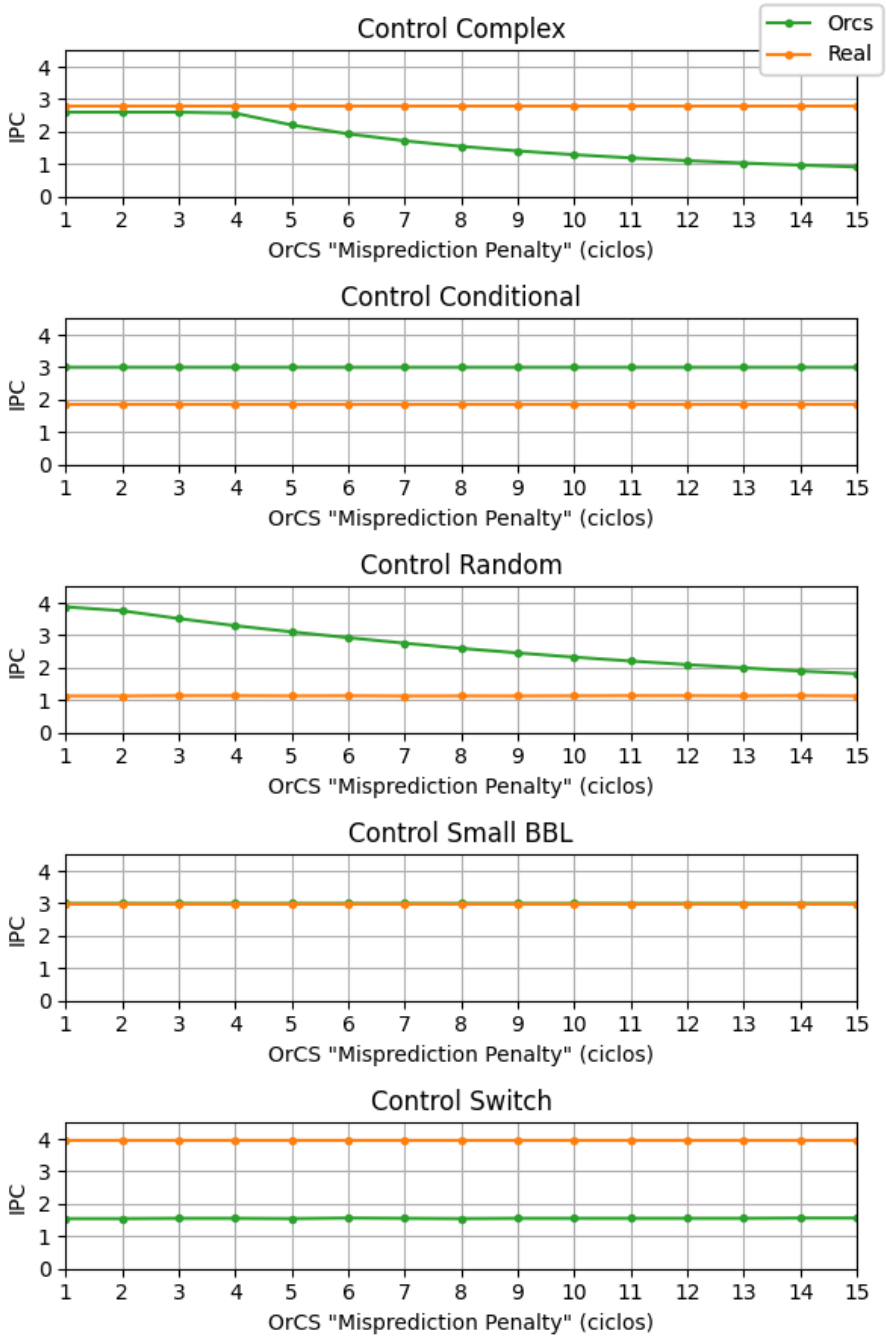


Figura 5.1: *Microbenchmarks* de Controle: Variação na penalidade do erro de predição de desvios (Intel Core i5-7400)

5.1.2 Dependência

Os *microbenchmarks* de dependência avaliam o adiantamento (*forwarding*) de dependências entre instruções. Nesse conjunto de testes, sequências de tamanhos que variam de 1 a 6 instruções de adição de inteiros são executadas em *loop*, onde cada instrução depende do resultado da anterior. Como esperado, o resultado é o mesmo entre o OrCS antigo e o OrCS modificado, pois nenhuma alteração foi proposta para o mecanismo de adiantamento de dependências, e as instruções utilizadas não tiveram as latências alteradas, pois a operação de adição de inteiros já era representada corretamente. Os resultados são apresentados na Figura 5.3.

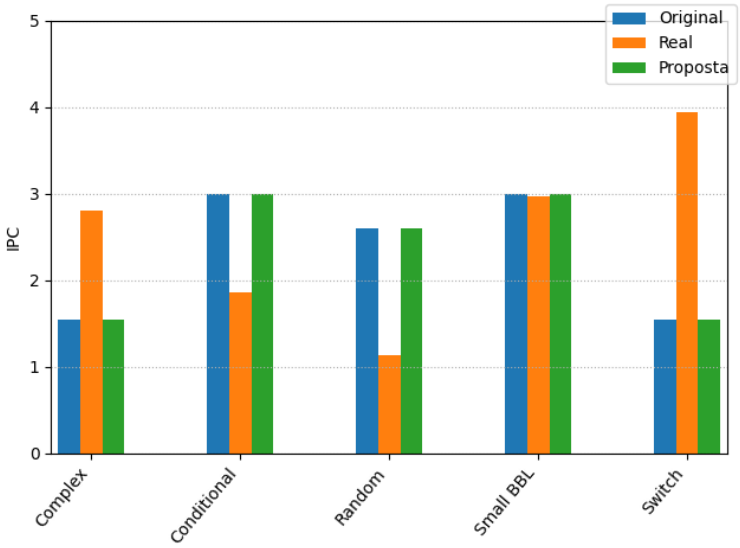


Figura 5.2: *Microbenchmarks* de Controle (Intel Core i5-7400)

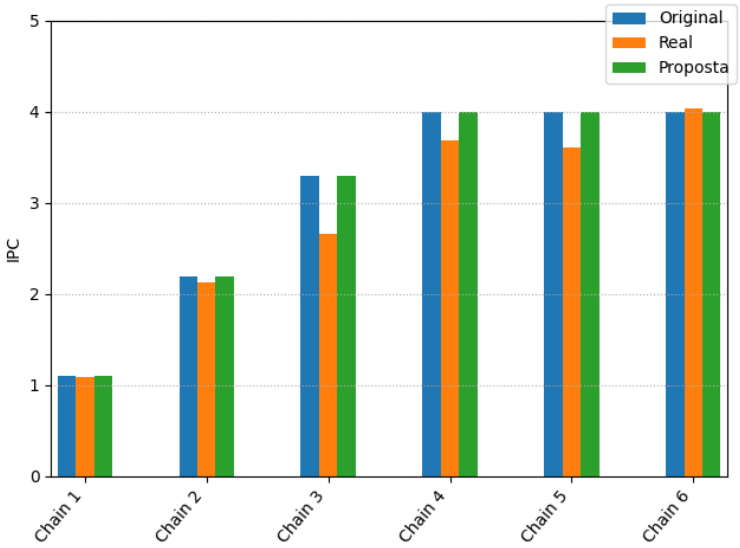


Figura 5.3: *Microbenchmarks* de Dependência (Intel Core i5-7400)

5.1.3 Execução INT (Instruções base)

Esse conjunto de *microbenchmarks* avalia métricas relacionadas às unidades funcionais. Cada *microbenchmark* executa 32 operações independentes em *loop*, ou seja, instruções consecutivas utilizam registradores distintos, removendo qualquer tipo de latência associada a *data forwarding*, com algumas exceções onde há registradores implícitos. Por não envolver fatores externos de forma significativa, como operações de memória e dependências de controle/dados, há uma alta correlação entre o IPC, a largura das unidades funcionais utilizadas, e a latência da instrução em questão. Portanto esse conjunto de testes é capaz de validar de forma mais assertiva as mudanças propostas no simulador.

Para demonstrar a taxa de execução ideal que esses testes propõem, na Figura 5.4 são apresentadas múltiplas execuções do *microbenchmark* que executa a instrução ADD repetidamente, porém variando a largura da unidade funcional ALU, que, na máquina real, possui valor 4. Nota-se uma correlação positiva perfeita para os valores de largura entre 1 e 4. Isso ocorre pois a

instrução ADD possui latência de 1 ciclo, portanto uma instrução será executada a cada ciclo em cada unidade funcional. Isso não ocorre a partir da largura igual a 5, pois nesse ponto, de acordo com a Tabela 5.1, a limitação torna-se a largura 4 do estágio de *Commit*.

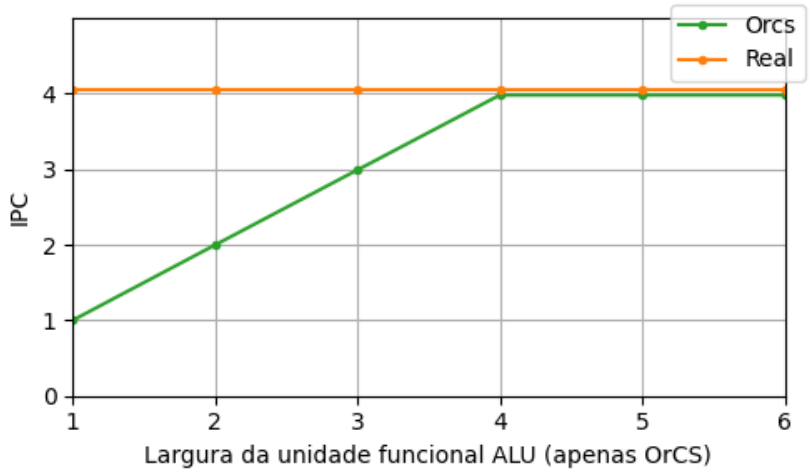


Figura 5.4: *Microbenchmark* de Execução: Instrução ADD variando a largura da unidade funcional ALU (Intel Core i5-7400)

Para testar as modificações propostas no OrCS, foram escolhidas algumas instruções para representar o conjunto de instruções base. As operações selecionadas são listadas na Tabela 5.3 com suas respectivas unidades funcionais e as latências das µops utilizadas nas duas versões do OrCS. Essa seleção foi construída para demonstrar algumas combinações de unidade funcionais diferentes. O restante das instruções base possui combinações semelhantes e, portanto, resultados semelhantes.

Tabela 5.3: Comparação das unidades funcionais utilizadas pelas instruções executadas no *microbenchmark* de execução (instruções Base)

	OrCS Original		**OrCS Modificado**	
Instrução	**Unid. Func.**	**Latência (µops)**	**Unid. Func.**	**Latência (µops)**
ADD	INT_ALU	1	ALU	1
ANDN	INT_ALU	1	Bit Manipulation	1
BEXTR	INT_ALU	1	Bit Manipulation, Shift	1, 1
DIV	INT_DIV	32	DIV, Vec ALU, ALU, Shift, Slow Int, Shuffle	20, 1, 1, 1, 3, 1
MUL	INT_MUL	3	Slow Int	3
RCL	INT_ALU	1	ALU, Slow Int, Shift	1, 3, 1
SHL	INT_ALU	1	Shift	1

A comparação do IPC entre a implementação original do OrCS, a máquina real, e a proposta é apresentada na Figura 5.5 para o *Intel Core i5-7400* e na Figura 5.6 para o *Intel Xeon Silver 4214*. Nota-se que um erro comum no OrCS original é a classificação das instruções em `INT_ALU`, resultando em um IPC próximo de 4, conforme explicado anteriormente. Com isso, a versão modificada do OrCS apresenta resultados mais precisos para as operações `ANDN`, `BEXTR`, e `SHL`, pois agora são executadas nas devidas unidades funcionais e com as latências corretas.

Apesar desse conjunto de *microbenchmarks* propor execuções de instruções independentes, algumas operações alteram registradores implícitos, como as instruções `DIV` e `RCL`,

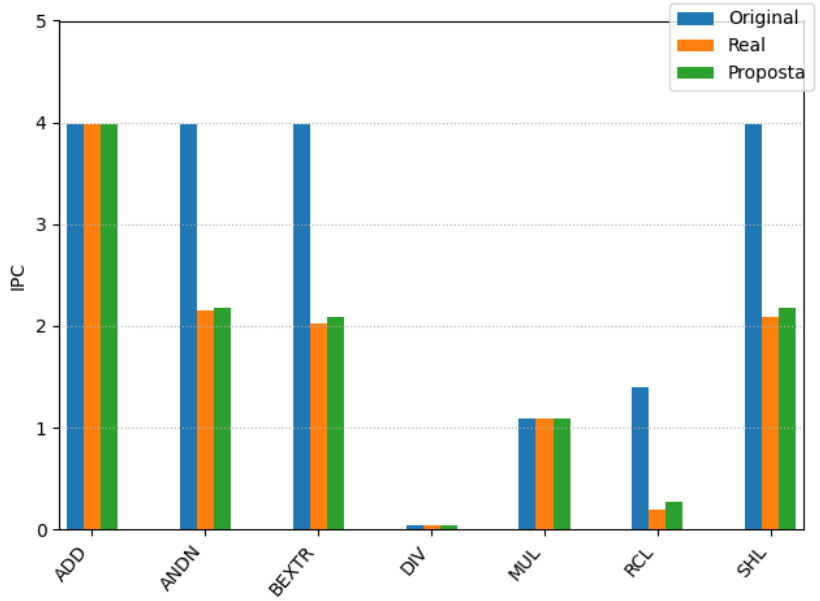


Figura 5.5: *Microbenchmark* de Execução: Instruções base de inteiros (Intel Core i5-7400)

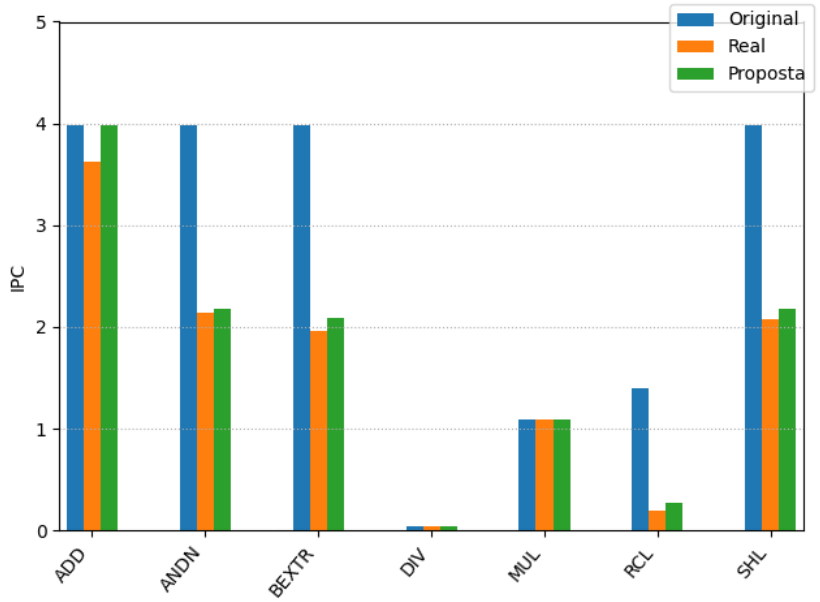


Figura 5.6: *Microbenchmark* de Execução: Instruções base de inteiros (Intel Xeon Silver 4214)

impossibilitando a independência. Por isso, o IPC das execuções de `RCL` é baixo no OrCS original, mesmo sendo classificado como `INT_ALU`. Mesmo assim, a proposta é capaz de aproximar ainda mais do resultado esperado, pois inclui µops semelhantes à máquina real.

Os valores de Erro Percentual Absoluto Médio (MAPE, ou *Mean Absolute Percentage Error*) em relação à execução na máquina real são apresentados na Tabela 5.4. Portanto houve um ganho significativo com as modificações propostas, entretanto é importante lembrar que esses resultados refletem as instruções selecionadas para o conjunto de *microbenchmarks*. Em aplicações reais, a tendência é que as instruções mais utilizadas sejam as instruções que já eram classificadas corretamente no OrCS original, como `ADD`, `DIV`, e `MUL`, além das operações de memória e desvio.

Tabela 5.4: MAPE (*Mean Absolute Percentage Error*) do IPC dos *microbenchmarks* de execução INT (instruções base)

	OrCS Original	**OrCS Modificado**
Intel Core i5	131,45%	9,67%
Intel Xeon Silver 4214	134,08%	12,42%

5.1.4 Execução FP (Instruções vetoriais previstas pelo OrCS)

Algumas instruções vetoriais, mais especificamente instruções que pertencem às extensões SSE, foram incluídas no algoritmo de classificação do OrCS original. Isso foi feito para adicionar suporte a operações sobre valores de ponto flutuante, porém essa classificação apresenta problemas semelhantes aos demonstrados para as instruções sobre inteiros.

Da mesma forma, foram escolhidas algumas operações para demonstrar os ganhos obtidos pela proposta. Os resultados foram obtidos utilizando o algoritmo de agrupamento de latências apresentado na Seção 4.4.2, com $G = 20$. As classificações das instruções são comparadas na Tabela 5.5 e os resultados são apresentados na Figura 5.7 para o *Intel Core i5-7400* e na Figura 5.8 para o *Intel Xeon Silver 4214*.

Nota-se que as instruções `PSLLD` e `SHUFPD` demonstram o mesmo problema apontado anteriormente, isto é, são classificadas como `FP_ALU`, que nesse caso, além das latências reais serem diferentes das latências escolhidas, são executadas em unidades funcionais com larguras diferentes, conforme especificado na Tabela 5.2.

Tabela 5.5: Comparação das unidades funcionais utilizadas pelas instruções sobre valores de ponto flutuante executadas no *microbenchmark* de execução (instruções vetoriais)

	OrCS Original		**OrCS Modificado**	
Instrução	**Unid. Func.**	**Latência (µops)**	**Unid. Func.**	**Latência (µops)**
ADDSD	FP_ALU	4	Vec Add	4
DIVSD	FP_DIV	13	DIV	15
MAXSD	FP_ALU	4	Vec Add	4
MULSD	FP_MUL	4	Vec Mul	4
PSLLD	FP_ALU	4	Shuffle	1
PXOR	FP_ALU	4	Vec ALU	1
SHUFPD	FP_ALU	4	Shuffle	1
SQRTSD	FP_DIV	13	DIV	19

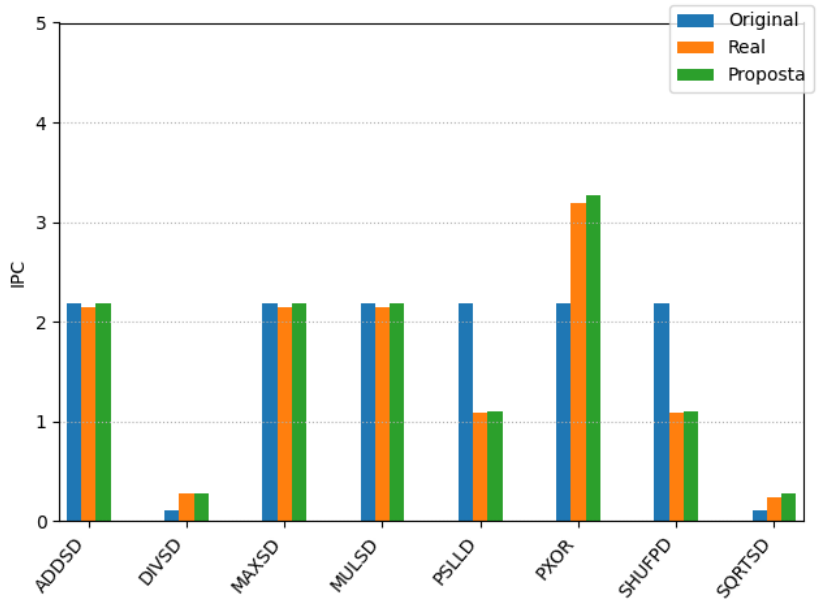


Figura 5.7: *Microbenchmark* de Execução: Instruções de Ponto Flutuante (Intel Core i5-7400)

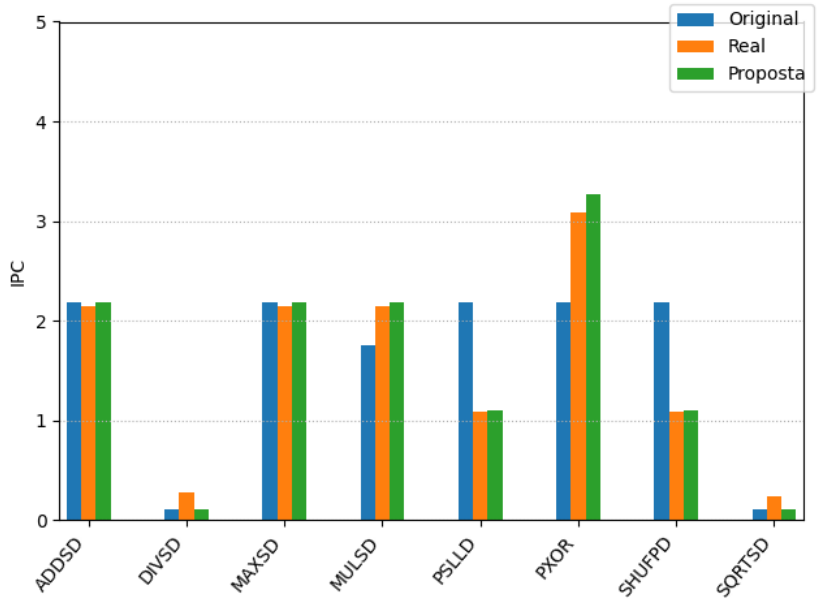


Figura 5.8: *Microbenchmark* de Execução: Instruções de Ponto Flutuante (Intel Xeon Silver 4214)

Para o conjunto de instruções de ponto flutuante, isto é, instruções vetoriais que foram incluídas no OrCS original, os erros relativos à execuções nas máquinas reais dos valores de IPC são apresentados na Tabela 5.6. Percebe-se que a diferença não é tão expressiva quanto nos testes sobre o conjunto de instruções base. Concluí-se que, devido à simplicidade da decomposição das instruções vetoriais em uma única µop na máquina real, o OrCS original é capaz de aproximar dos valores das latências reais e larguras das unidades funcionais com uma acurácia suficiente.

5.1.5 Execução Vetorial

Conforme explicado, todas as instruções utilizadas até agora em todos os *microbenchmarks* foram incluídas no processo de *Decode* do OrCS original. Porém uma grande quantidade

Tabela 5.6: MAPE (*Mean Absolute Percentage Error*) do IPC dos *microbenchmarks* de execução FP (instruções vetoriais previstas pelo OrCS)

	OrCS Original	**OrCS Modificado**
Intel Core i5	44,01%	2,96%
Intel Xeon Silver 4214	46,24%	16,02%

de operações não aparece no algoritmo de classificação antigo, especialmente instruções vetoriais de extensões SSE, AVX, entre outras. O comportamento do algoritmo implementado pelo OrCS original é classificar as instruções desconhecidas como `INT_ALU`. Portanto, como é de se esperar, diversas instruções terão a latência mais baixa que o previsto, e suas µops serão executadas em unidades funcionais de largura 4, resultando em uma vazão muito maior que a execução na máquina real.

A seguir, é apresentado um conjunto de *microbenchmarks* de execução contendo apenas instruções vetoriais não previstas pelo OrCS original, porém inclusas na proposta deste trabalho. Da mesma forma que o conjunto de testes anterior, a configuração de µops utilizada foi gerada a partir do algoritmo de agrupamento de latências com $G = 20$.

A Tabela 5.7 descreve as instruções selecionadas para esse conjunto de testes. Essa tabela contém a unidade funcional que cada instrução utiliza, a latência da µop de cada instrução, a latência original fornecida por Abel e Reineke (Abel e Reineke, 2019), e a quantidade de vezes (contagem) que a operação foi executada no *benchmark* SPEC2017.

Tabela 5.7: Listagem das instruções selecionadas para o *microbenchmark* de execução de instruções vetoriais ignoradas pelo OrCS original

	OrCS Modificado			
Instrução	**Unid. Func.**	**Latência OrCS**	**Latência Real**	**Contagem SPEC**
DPPS	Vec Add	8	13	0
VADDPS	Vec Add	4	4	255584701320
VCVTSI2SD	Shuffle	7	7	1228983438
VDIVSD	DIV	15	15	27193871384
VMULSD	Vec Mul	4	4	461305959635
VPSLLVQ	Vec Shift	1	1	0
VSHUFPS	Shuffle	1	1	0
VSQRTPD	DIV	19	19	0
VXORPD	Vec ALU	1	1	19281508899

A instrução `DPPS` foi escolhida por apresentar uma diferença significativa de latência em relação ao valor real. Isso é um resultado do algoritmo de otimização que não priorizou essa instrução por não estar presente em nenhum *benchmark* do conjunto SPEC CPU2017. Evidentemente, o fato dessa instrução não aparecer no SPEC não significa que deve ser ignorada, pois outros programas podem fazer um uso intenso dessa operação, resultando em valores inesperados de IPC, conforme ilustrado pelo resultado deste *microbenchmark*. As formas de contornar esse problema seria aumentar o valor do parâmetro G no algoritmo, ou utilizar outros programas que incluem essa instrução ao gerar os valores de contagem.

Os resultados da execução dos *microbenchmarks* são apresentados na Figura 5.9 para o *Intel Core i5-7400* e na Figura 5.10 para o *Intel Xeon Silver 4214*. Nota-se um ganho relevante da proposta em relação ao OrCS original, conforme esperado. O teste com o pior resultado foi, de fato, a execução da instrução `DPPS`, demonstrando a limitação do algoritmo proposto. A

proximidade do valor de IPC do OrCS modificado confirma a validade dos valores fornecidos por Abel e Reineke (Abel e Reineke, 2019) e a acurácia do simulador com relação à instruções de execução independentes.

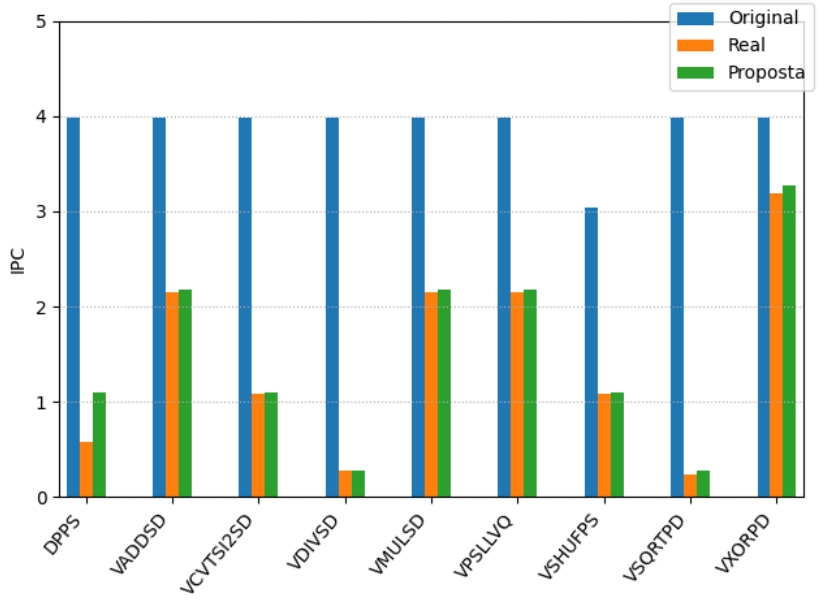


Figura 5.9: *Microbenchmark* de Execução: Instruções Vetoriais (Intel Core i5-7400)

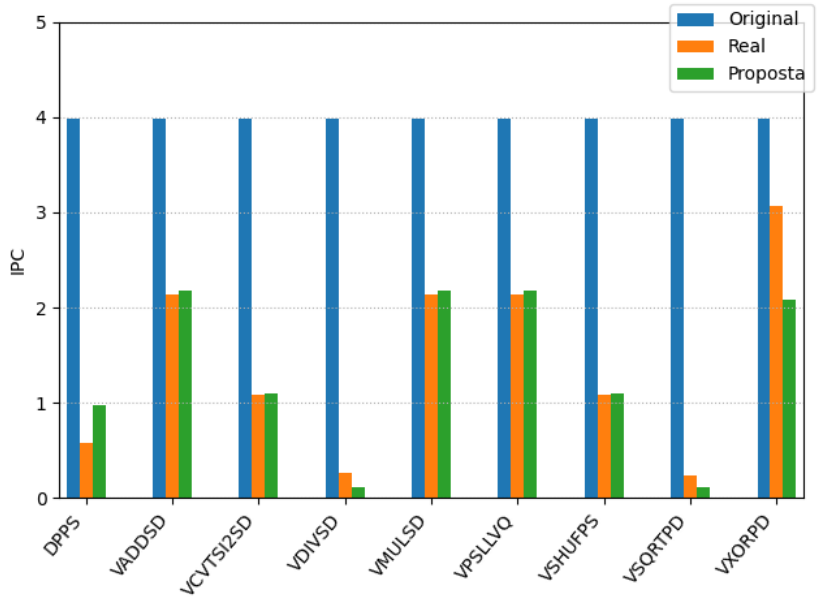


Figura 5.10: *Microbenchmark* de Execução: Instruções Vetoriais (Intel Xeon Silver 4214)

Os erros relativos à execuções nas máquinas reais dos valores de IPC são apresentados na Tabela 5.8. Conclui-se que a inclusão das instruções vetoriais no simulador é essencial para uma modelagem mais precisa, principalmente em casos onde instruções vetoriais compõem a maioria das operações de um programa. Um caso extremo seria um uso intenso da instrução `VDIVSD`, que seria classificada como `INT_ALU` no OrCS original, resultando em um IPC mais alto que o desejado.

Tabela 5.8: MAPE (*Mean Absolute Percentage Error*) do IPC dos *microbenchmarks* de execução vetorial

	OrCS Original	**OrCS Modificado**
Intel Core i5	465,23%	12,46%
Intel Xeon Silver 4214	482,39%	24,68%

5.1.6 Execução Vetorial AVX-512

Assim como no conjunto de *microbenchmarks* de instruções vetoriais, não há nenhum suporte à extensão AVX-512 no OrCS original. Portanto, da mesma forma, todas as instruções desse conjunto serão interpretadas como `INT_ALU` na simulação. Com isso, é de se esperar que a proposta gere resultados melhores para as execuções das instruções AVX-512. Porém, conforme explicado no Capítulo 4, a inconsistência e complexidade dessas instruções torna difícil uma modelagem capaz de prever o comportamento de todas as instruções. Esse conjunto de *microbenchmarks* foi executado apenas no processador Intel Xeon Silver 4214, pois o Intel Core i5 não implementa operações vetoriais de 512 bits.

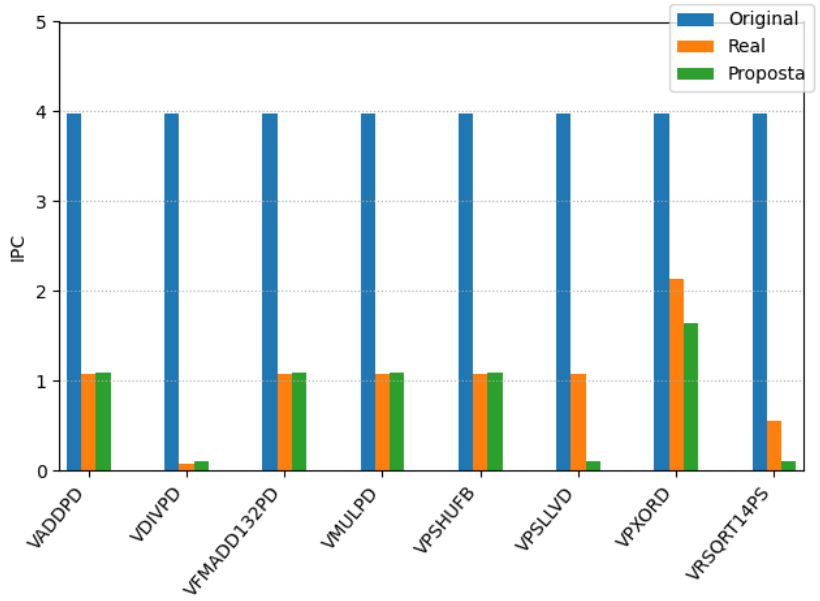


Figura 5.11: *Microbenchmark* de Execução: Instruções AVX-512 (Intel Xeon Silver 4214)

Observando os resultados ilustrados pela Figura 5.11, nota-se que as execuções de instruções como `VADDPD`, `VMULPD`, e `VFMADD132PD` resultaram em um IPC de aproximadamente 1. Considerando que essas instruções possuem as mesmas latências que as vetoriais que foram executadas nos *microbenchmarks* anteriores com o IPC próximo a 2 e em unidades funcionais com largura 2, conclui-se que o Xeon Silver 4214, de fato, não possui a unidade dedicada às µops AVX-512 e, portanto, as executa apenas na fusão das unidades vetoriais de 256 bits.

Pelo gráfico, nota-se uma limitação do algoritmo com relação à execução da instrução `VPSLLVD`. Essa instrução possui latência de 1 ciclo e executa na unidade funcional `DIV`, que possui largura igual a 1. Portanto, a execução real resulta em um IPC de, aproximadamente, 1, porém a proposta obtém um IPC de 0,109. Isso se deve à latência de espera (`WAIT_NEXT`) das unidades funcionais após a execução de cada µop. Esse valor foi definido como 10 ciclos para a unidade funcional `DIV` para simular a baixa vazão normalmente associada às instruções que são executadas por ela. Porém esse *microbenchmark* revela que esse nem sempre é o caso,

logo, a suposição, herdada do OrCS original, de que esse valor está intrinsecamente associado às unidades funcionais não está correto, pois cada unidade pode ser implementada com um pipeline interno que pode ter um comportamento que varia para cada μop. Uma alternativa seria uma análise mais elaborada para atribuir essas latências entre às μops.

Tabela 5.9: MAPE (*Mean Absolute Percentage Error*) do IPC dos *microbenchmarks* de execução vetorial AVX-512

	OrCS Original	**OrCS Modificado**
Intel Core i5	-	-
Intel Xeon Silver 4214	954,55%	31,71%

5.2 BENCHMARKS SPEC CPU2017

Ao utilizar programas mais complexos, como *benchmarks*, para testar a proposta descrita para o OrCS, espera-se que os resultados não sejam tão acentuados quanto os valores apresentados para os *microbenchmarks*. Isso se deve ao fato de que, além da complexidade envolvida por outros componentes que são utilizados intensivamente, a grande maioria das instruções mais frequentes, isto é, as instruções base mais comuns, já era modelada corretamente na implementação original do OrCS.

Para verificar o impacto das alterações no OrCS, foi utilizado o conjunto de *benchmarks* SPEC CPU2017 (Bucek et al., 2018) formulado pela *Standard Performance Evaluation Corporation* (SPEC, 1995). Esses *benchmarks* são utilizados com frequência por pesquisadores que buscam execuções replicáveis e semelhantes a cargas de trabalho de aplicações do mundo real. Portanto, entre todos os *benchmarks* desse conjunto, diferentes aspectos da arquitetura serão estressados.

Há dois conjuntos de *benchmarks* fornecidos pelo SPEC CPU2017, o *Speed*, que obtém o tempo de execução sequencial (*single thread*) através de métricas como o IPC; e o *Rate*, que mede a taxa de transferência (Singh e Awasthi, 2019). Por ser mais relevante a esse trabalho, será apresentada a comparação apenas sobre o conjunto *Speed*. Cada um desses conjuntos é dividido em *benchmarks* que utilizam instruções que operam sobre valores inteiros e os que utilizam instruções que operam sobre valores de ponto flutuante.

Para as execuções na máquina real, as aplicações completas foram consideradas. Já para as execuções nas duas versões do simulador, foram selecionadas seções representativas compostas por dois bilhões de instruções selecionadas através da ferramenta PinPoints (Patil et al., 2004). Essa decisão foi necessária para manter um tempo de execução aceitável, visto que o OrCS possui uma alta complexidade.

A Figura 5.12 apresenta, para cada aplicação do SPEC CPU2017, a comparação entre o IPC resultante da execução na máquina real, no OrCS original, e no OrCS modificado com a proposta desse trabalho. Os nomes das aplicações terminadas em `CFP` utilizam instruções que operam sobre valores de ponto flutuante e as terminadas em `CINT`, valores inteiros.

O erro (MAPE) dos valores de IPC para a proposta é de 127,98%, enquanto que para a implementação original do OrCS, o erro é de 121,87%. Nota-se que, em grande parte dos testes, não houveram mudanças significativas geradas pela proposta descrita no Capítulo 4 em relação à implementação original do OrCS. Conforme mencionado anteriormente, isso já era esperado. Porém, algumas aplicações, como `imagick` e `nab`, apresentaram resultados consideravelmente piores. Através das estatísticas geradas pelo OrCS, é possível extrair informações para investigar os resultados obtidos.

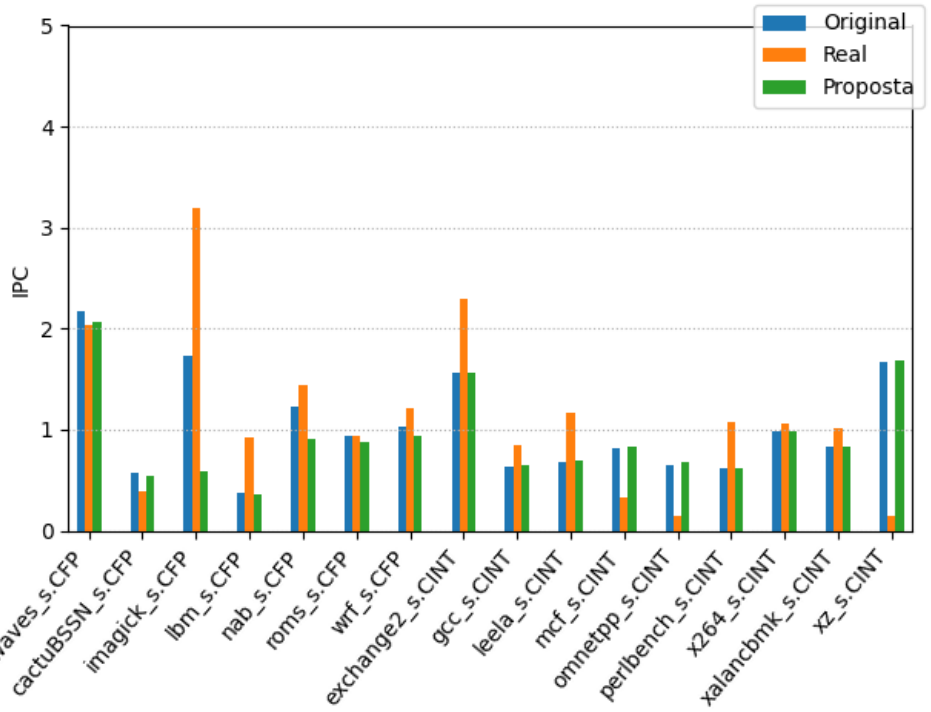


Figura 5.12: Comparação entre resultados do SPEC CPU2017

Para auxiliar na análise dos resultados, a Figura 5.13 ilustra, para cada aplicação, a média das latências de todas as µops executadas no OrCS original e no OrCS modificado. Apenas quatro *benchmarks* se destacam: `cactuBSSN`, `imagick`, `lbm`, e `nab`. O restante das aplicações, conforme esperado, apresentam resultados muito semelhantes entre si e se aproximam de valores de latência igual a um ciclo, pois a grande maioria das µops executadas consistem em operações do tipo `ALU` e operações de memória.

A Figura 5.14 apresenta a comparação das porcentagens de µops do tipo `ALU` (`INT_ALU` no OrCS original e `ALU` na proposta) em relação a todas as µops executadas pela aplicação. O destaque para esse gráfico é o fato de que, a diferença entre *benchmarks* do tipo `CINT` e `CFP` só é notável para os valores da proposta. Isso ocorre pois, como demonstrado nos resultados dos *microbenchmarks*, o OrCS original considerava diversas instruções vetoriais sobre operandos de ponto flutuante como `INT_ALU` e a modificação proposta é capaz de corrigir isso.

É importante lembrar que o gráfico da Figura 5.14 não é uma comparação entre quantidade de operações de ponto flutuante e inteiros, pois há diversas outras instruções que operam sobre valores inteiros mas executam em outras unidades funcionais como `Bit_Manipulation`, `Slow_Int`, entre outras. A diferença entre `CFP` e `CINT` seria ainda mais acentuada caso a comparação mencionada fosse apresentada.

Para auxiliar na análise dos resultados, também será incluído a seguir, na Figura 5.15, um gráfico ilustrando a comparação entre a máquina real e o OrCS em termos de uso de memória, mais especificamente MPKI (*Misses per kilo instructions*). Ou seja, trata-se de uma métrica que informa a quantidade, proporcional ao número de instruções, de acessos à memória principal após um *miss* na LLC (*Last Level Cache*)

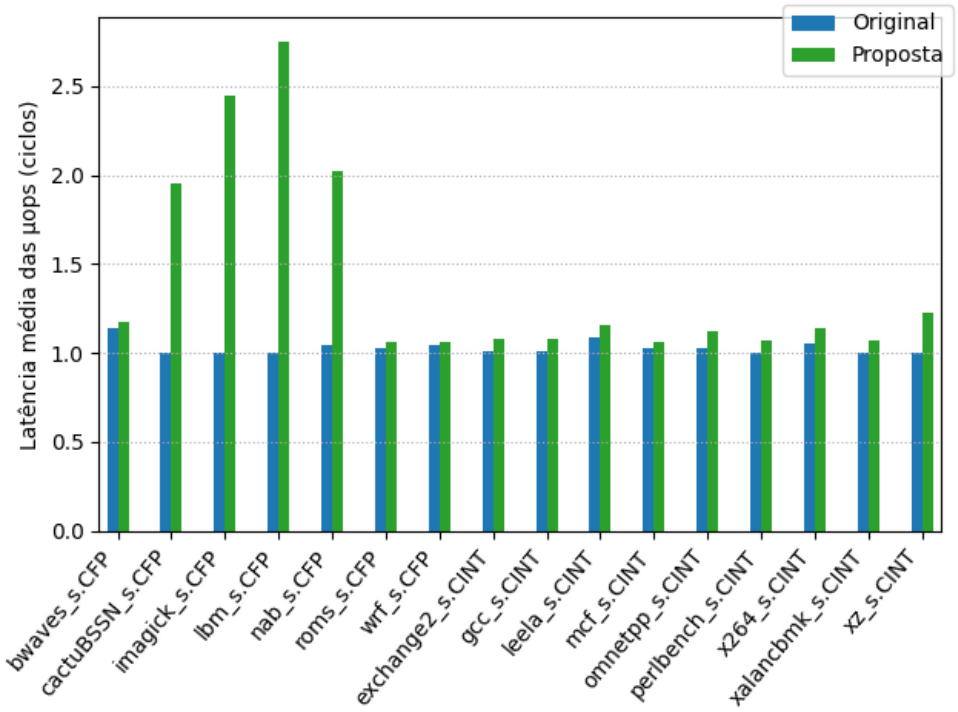


Figura 5.13: Comparação das latências médias das µops por aplicação SPEC CPU2017

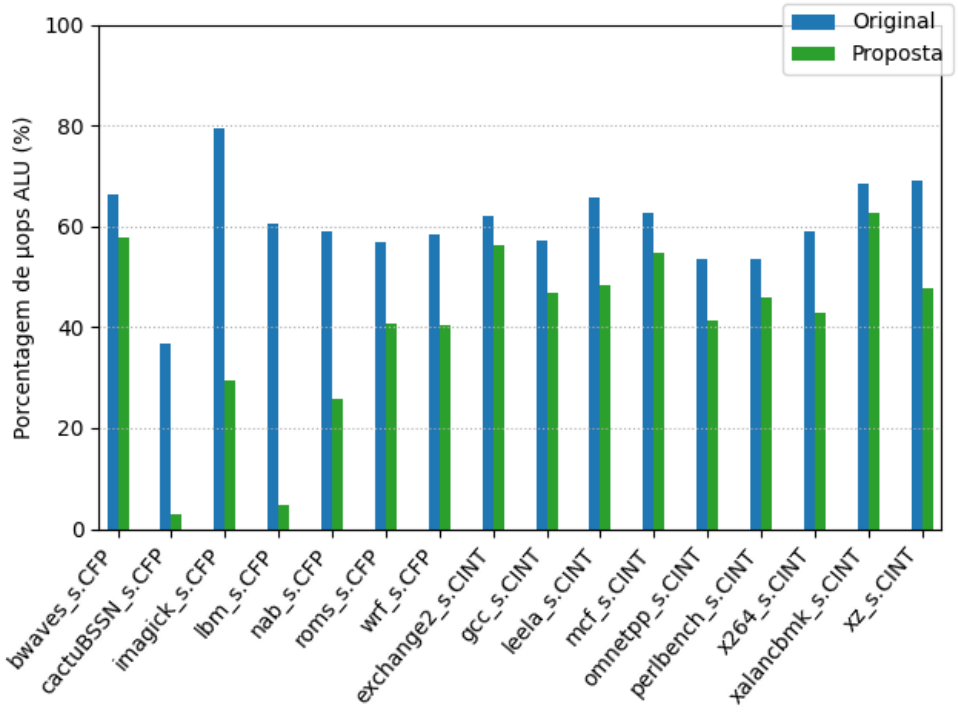


Figura 5.14: Comparação das proporções de µops `ALU` por aplicação SPEC CPU2017

5.2.1 Análise dos resultados do `imagick_s.CFP`

A maior diferença de IPC foi obtida na aplicação `imagick`, onde, enquanto a execução na máquina real resulta em um IPC de 3,20, o OrCS original gera um IPC de 1,73 e a proposta, 0,59. Ou seja, houve um distanciamento significativo em relação ao valor esperado.

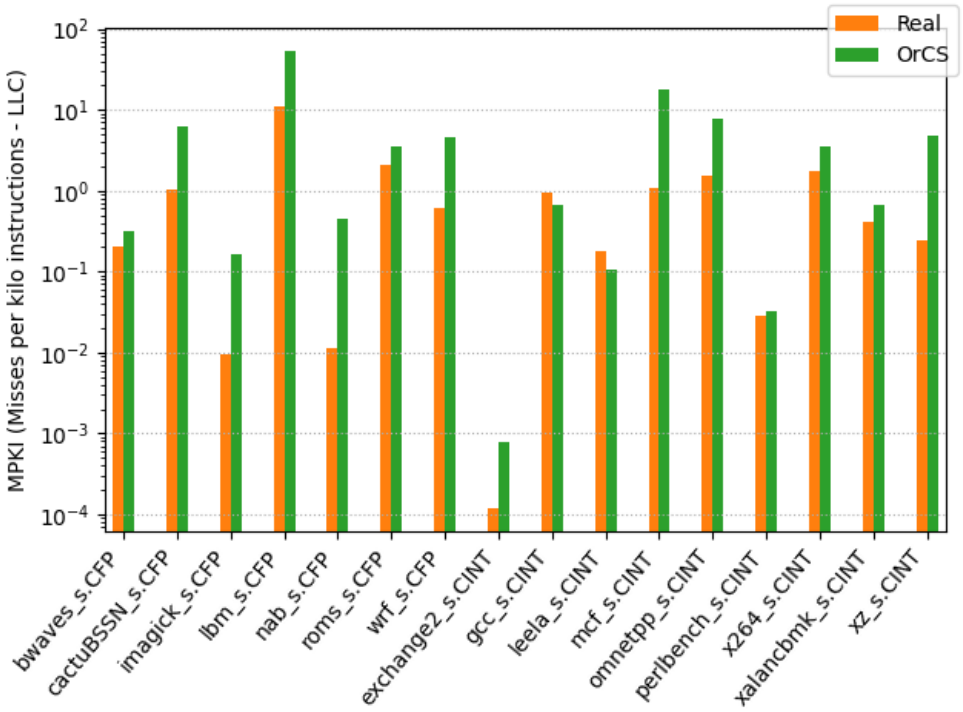


Figura 5.15: Comparação de MPKI por aplicação SPEC CPU2017

Ao analisar os dados da Tabela 5.10, observa-se que no OrCS original, 79% das µops executadas foram classificadas como `INT_ALU` com a latência média de 1 ciclo, e o restante, em grande parte, são operações de memória. Porém, a classificação da proposta indica que apenas 29,56% das µops são do tipo `ALU`, enquanto que 12,7% são `Vec_Add` com latência média de 4 ciclos, 12% `Vec_Mul` (latência média de 4), 11,98% `Shift` (latência média de 1), 11,65% `Shuffle` (latência média de 6,99), e 3,49% `DIV` (latência média de 1,2). Ou seja, as instruções haviam sido decodificadas pelo OrCS original em µops muito mais eficientes, em termos de latência e largura da unidade funcional, do que elas de fato são.

Sabemos que o único fator alterado no OrCS foi a classificação das instruções que se tornou mais precisa, e que, no `imagick`, o OrCS original assumia que as instruções tinham uma latência mais baixa que o ideal. De acordo com Singh et al. (Singh e Awasthi, 2019) e com o gráfico da Figura 5.15, o `imagick` é uma aplicação que faz um bom uso da cache, ou seja, trata-se de uma aplicação *CPU bound*. Portanto, o IPC depende mais das unidades funcionais do que de acessos lentos à memória, ou seja, qualquer alteração nas latências e decodificação das instruções gera um impacto considerável no IPC. Uma possível explicação para a diferença entre os simuladores é que o resultado do OrCS original se aproximava mais do resultado desejado devido a uma decodificação otimista demais e não pela precisão da simulação. Ou seja, outros componentes precisariam ser melhorados para que o valor de IPC aumente, em vez de simplesmente abaixar artificialmente as latências das instruções.

Uma outra hipótese para esse resultado se encontra em possíveis mecanismos presentes na arquitetura real, ignorados por esse trabalho, que não são evidenciados pelos dados utilizados para gerar a decodificação das instruções. Um exemplo observado durante o desenvolvimento dos *microbenchmarks* é o fato da instrução `XOR`, na máquina real, ser decodificada em zero µops quando o mesmo registrador é utilizado em todos os operandos. Isso é muito utilizado para executar uma instrução que não altera o estado dos registradores, como a instrução `NOP`. Ao utilizar os dados da instrução `XOR` para montar uma decodificação em µops, o OrCS modificado

Tabela 5.10: Estatísticas de uso de unidades funcionais da aplicação imagick_s.CFP

Original			Proposta		
Unid. Func.	**% de µops**	**Lat. Média**	**Unid. Func.**	**% de µops**	**Lat. Média**
INT_ALU	79,52%	1,00	**ALU**	29,56%	1,00
INT_MUL	0,00%	3,00	**Bit_Manip.**	0,00%	1,00
INT_DIV	0,00%	32,00	**DIV**	3,49%	1,20
FP_ALU	0,14%	3,00	**FP_Mov**	0,00%	0,00
FP_MUL	0,00%	0,00	**SIMD_Misc**	0,00%	0,00
FP_DIV	0,00%	10,00	**Shift**	11,98%	1,00
LOAD	19,24%	1,00	**Shuffle**	11,65%	6,99
STORE	1,10%	1,00	**Slow_Int**	0,32%	3,00
			Vec_ALU	0,21%	1,00
			Vec_Add	12,70%	4,00
			Vec_FMA	0,00%	0,00
			Vec_Mul	12,00%	4,00
			Vec_Shift	0,00%	1,00
			LOAD	17,10%	1,00
			STORE	0,97%	1,00

trataria esse caso como uma instrução qualquer, ignorando essa característica da arquitetura e atribuindo uma latência maior que a desejada para essa operação, resultando em uma simulação com o IPC menor que a execução real.

O alto valor de IPC obtido da execução real do `imagick`, mesmo com a alta quantidade de instruções lentas, é um forte indício de que há algum mecanismo envolvido pela arquitetura *Skylake* que não foi previsto pelo OrCS. Uma hipótese possível envolve um paralelismo de instruções mais sofisticado, aproveitado pelo `imagick`, capaz de executar determinadas sequências de instruções vetoriais com alta vazão.

Outra possibilidade consiste em uma possível má representação da aplicação por parte do PinPoints, que talvez não foi capaz de isolar uma fatia do executável capaz de fornecer um IPC representativo do *benchmark* completo. De acordo com Singh et al. (Singh e Awasthi, 2019), a aplicação `imagick_s.CFP` é um dos testes do SPEC CPU2017 com o maior número de instruções no total, com aproximadamente 5 trilhões de instruções. Ou seja, a seção de 2 bilhões de instruções selecionada trata-se de uma porção pequena, representando apenas 0,04% da aplicação total.

5.2.2 Análise dos resultados do `nab_s.CFP`

No caso da aplicação `nab`, a execução na máquina real resulta em um IPC de 1,44, o OrCS original gera um IPC de 1,23 e a proposta, 0,90. Ou seja, semelhante à comparação do resultados do `imagick`, a proposta se distanciou do valor de IPC esperado, porém, nesse caso, a diferença não foi tão acentuada.

Ao observar a Tabela 5.11, que foi construída da mesma forma que a tabela apresentada anteriormente, nota-se um padrão semelhante aos dados do `imagick`. Isto é, muitas µops do tipo `INT_ALU` passaram a ser classificadas como `Vec_Add`, `Vec_Mul`, `DIV`, `Shift`, e `Slow_Int`. Podemos estimar que o impacto dessa nova classificação não foi tão grande quanto no caso do `imagick`, pois agora a proporção de µops de memória é muito maior. Além disso,

de acordo com a Figura 5.13, a diferença da latência média das µops entre as duas versões dos simuladores não é tão grande quanto no `imagick`.

Assim como o `imagick`, o `nab` é uma aplicação *CPU-bound*, portanto estamos lidando com o mesmo problema nesse caso. Isso contribui com a hipótese levantada anteriormente de que a versão anterior do OrCS havia se aproximado do valor de IPC real por estar superestimando a performance das instruções.

Tabela 5.11: Estatísticas de uso de unidades funcionais da aplicação nab_s.CFP

Original			**Proposta**		
Unid. Func.	**% de µops**	**Lat. Média**	**Unid. Func.**	**% de µops**	**Lat. Média**
INT_ALU	58,88%	1,00	**ALU**	25,78%	1,00
INT_MUL	0,82%	3,00	**Bit_Manip.**	0,00%	0,00
INT_DIV	0,00%	0,00	**DIV**	7,04%	5,84
FP_ALU	1,14%	3,00	**FP_Mov**	0,00%	0,00
FP_MUL	0,00%	5,00	**SIMD_Misc**	0,00%	0,00
FP_DIV	0,00%	10,00	**Shift**	2,79%	1,00
LOAD	32,05%	1,00	**Shuffle**	0,23%	7,00
STORE	7,11%	1,00	**Slow_Int**	2,82%	3,00
			Vec_ALU	0,80%	1,00
			Vec_Add	8,21%	4,00
			Vec_FMA	0,00%	0,00
			Vec_Mul	12,17%	4,00
			Vec_Shift	0,00%	0,00
			LOAD	32,87%	1,00
			STORE	7,29%	1,00

5.2.3 Análise dos resultados do `lbm_s.CFP`

A aplicação `lbm` é um dos casos em que não houve uma grande diferença entre os resultados do OrCS original e a proposta. Na máquina real, o IPC obtido é de 0,92, enquanto que no OrCS original esse valor é de 0,37 e na versão modificada é 0,36.

A princípio, a semelhança do IPC entre as duas versões do simulador é contraditória com o gráfico apresentado na Figura 5.13, onde foi observado que o `lbm` possui, entre todas as aplicações, a maior diferença da latência média das µops entre os dois simuladores. As diferenças podem ser observadas com mais detalhes na Tabela 5.12, onde nota-se que 60,6% das instruções estavam sendo classificadas como `INT_ALU`, quando, na verdade, apenas 4,6% executam nessa unidade funcional, 35,4% executam em `Vec_Add` com latência de 4 ciclos, e 19,3% em `Vec_Mul` com 4 ciclos.

Seguindo a hipótese levantada, é observado na Figura 5.15 que o `lbm` possui a maior proporção de acessos à memória principal entre todas as aplicações. Além disso, Singh et al. (Singh e Awasthi, 2019) demonstra através de análises mais aprofundadas, que essa aplicação faz um uso muito intenso da memória principal por não ser capaz de acomodar todos os dados em cache. Isso ocorre devido à natureza complexa do *benchmark* que simula um fluido incompressível em 3D. Portanto trata-se de uma aplicação *memory-bound* e, por isso, não houve uma mudança significativa no valor de IPC, mesmo aumentando as latências das instruções.

A quarta aplicação que mais apresentou diferenças nas latências médias das µops foi o `cactuBSSN`, que também se enquadra no caso de uma aplicação *memory-bound*. Onde, da

Tabela 5.12: Estatísticas de uso de unidades funcionais da aplicação lbm_s,CFP

Original			Proposta		
Unid. Func.	**% de µops**	**Lat. Média**	**Unid. Func.**	**% de µops**	**Lat. Média**
INT_ALU	60,60%	1,00	**ALU**	4,62%	1,00
INT_MUL	0,00%	3,00	**Bit_Manip.**	0,00%	1,00
INT_DIV	0,00%	32,00	**DIV**	0,74%	15,00
FP_ALU	0,00%	0,00	**FP_Mov**	0,00%	0,00
FP_MUL	0,00%	0,00	**SIMD_Misc**	0,00%	0,00
FP_DIV	0,00%	0,00	**Shift**	0,00%	1,00
LOAD	24,19%	1,00	**Shuffle**	0,00%	2,50
STORE	15,21%	1,00	**Slow_Int**	0,00%	3,00
			Vec_ALU	0,00%	1,00
			Vec_Add	35,47%	4,00
			Vec_FMA	0,00%	0,00
			Vec_Mul	19,37%	4,00
			Vec_Shift	0,00%	0,00
			LOAD	24,43%	1,00
			STORE	15,36%	1,00

mesma forma que o `lbm`, também não houveram mudanças significativas no IPC entre as duas versões do simulador.

6 CONCLUSÕES

Este trabalho teve como objetivo melhorar a precisão da simulação do OrCS através da proposta de utilizar uma decodificação das instruções a partir de dados desse processo extraídos da máquina real. E, por fim, foram apresentados experimentos e análises para validar as mudanças implementadas.

Com a implementação definida, foi possível estabelecer a relação entre instruções, μops, unidades funcionais, e latências de forma genérica e organizada. Isso proporciona a fácil manipulação dessas informações, possibilitando a modelagem de outras arquiteturas, incluindo as do tipo RISC. Isso tudo foi feito a partir de alterações mínimas na implementação do OrCS, o que indica que o restante dos mecanismos representados não sofreram nenhum impacto e que não houve perda significativa de desempenho no tempo de execução do OrCS.

A partir dos resultados obtidos das execuções dos *microbenchmarks*, conclui-se que as latências das μops e as larguras das unidades funcionais definidas são compatíveis com as implementações desses componentes na máquina real, e que o OrCS é capaz de simular isso com precisão. Também ficou claro que a versão original do OrCS apresentou diversos problemas ao lidar com instruções que não foram incluídas na decodificação estabelecida manualmente, conforme exemplificado na Seção 5.1.5 com os microbenchmarks de instruções vetoriais onde foi evidenciado o impacto de classificar instruções desconhecidas de uma forma muito simples. Essa limitação ressalta ainda mais a vantagem de automatizar esse processo utilizando os dados extraídos por Abel e Reineke (Abel e Reineke, 2019), pois agora não será necessário repetir o trabalho manual quando novas instruções forem incluídas.

Os *benchmarks* SPEC CPU2017 demonstraram a complexidade envolvida ao tentar simular uma arquitetura executando aplicações completas. Os resultados, apesar de não terem melhorado em relação ao OrCS original, revelaram que, ao simular o comportamento das instruções com mais precisão, outros problemas podem ser revelados. Foi observado através das estatísticas do simulador, que um uso intenso da memória é capaz de esconder a performance das unidades de execução. Isso limita a capacidade dos desenvolvedores de arquiteturas e simuladores de correlacionar as melhorias de precisão com métricas de desempenho.

Na arquitetura real, quando diversos componentes são envolvidos, cada subsistema irá aumentar ou diminuir o valor de uma métrica resultante. Portanto, muitas vezes é necessário isolar as partes em questão através de testes especializados, como *microbenchmarks*, para analisar e garantir a precisão da simulação. Para aumentar a confiança das métricas geradas por um simulador, é fundamental a garantia de que a maior quantidade possível de componentes esteja modelando o comportamento esperado. Isso foi alcançado nesse trabalho, pois, conforme os experimentos apontaram, a versão original do OrCS apresentou diversas falhas que o aproximava das métricas desejadas porém de forma errônea.

6.1 TRABALHOS FUTUROS

Considerando a complexidade da arquitetura que o OrCS tenta replicar, é natural que haja diversas oportunidades de melhorias ao simulador. Qualquer componente implementado atualmente é passível de análises semelhantes ao que foi apresentado nesse trabalho. Esse tipo de análise contribuiria com a capacidade de isolar problemas e falhas de representação, permitindo uma conclusão mais assertiva acerca dos impactos causados pelas modificações propostas nas instruções.

Durante o processo de análise dos resultados do SPEC, foram levantadas algumas hipóteses para tentar explicar os resultados de alguns *benchmarks*. A confirmação dessas hipóteses requer análises mais aprofundadas que poderiam ser feitas a partir de *microbenchmarks* e métricas que não foram utilizadas nesse trabalho.

Alguns processos de validação do simulador e da proposta apresentada foram excluídos desse trabalho devido à alta complexidade envolvida. Por exemplo, seria possível utilizar contadores específicos do processador para obter métricas relacionadas ao uso das portas durante a execução de *benchmarks* e *microbenchmarks*. Com isso, pode-se realizar uma correlação das métricas do OrCS com os dados da execução na máquina real de uma forma mais minuciosa.

A expressividade do formato definido para representar as instruções no OrCS não foi testada para outras arquiteturas além das versões mais recentes dos processadores Intel. A simulação de uma arquitetura RISC qualquer daria a oportunidade de observar as métricas produzidas a partir de uma "decodificação" mais simples e bem documentada. Isso, supostamente, poderia facilitar o processo de validação por tratar-se de uma microarquitetura mais simples e transparente.

REFERÊNCIAS

Abel, A. (2019). XED to XML converter. `https://github.com/andreas-abel/XED-to-XML`. Acessado em 22/11/2020.

Abel, A. e Reineke, J. (2019). uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. Em *ASPLOS*, ASPLOS '19, páginas 673–686, New York, NY, USA. ACM.

Akram, A. e Sawalha, L. (2016). ×86 computer architecture simulators: A comparative study. Em *2016 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. (2014). *Increasing Energy Efficiency of Processor Caches via Line Usage Predictors*. Tese de doutorado, Programa de Pós-Graduação em Computação - Universidade Federal do Rio Grande do Sul.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B. e Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. Em *Proceedings of the 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 Conf on Embedded Software and Systems*, HPCC-CSS-ICESS '15, página 605–610, USA. IEEE Computer Society.

Archer, B. (2016). *Assembly Language For Students*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

Asri, M., Pedram, A., John, L. K. e Gerstlauer, A. (2016). Simulator calibration for accelerator-rich architecture studies. Em *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, páginas 88–95.

Barton, C. et al. (2011). The multi2sim simulation framework a cpu-gpu model for heterogeneous computing. `http://www.multi2sim.org/downloads/m2s-guide-4.2.pdf`.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. e Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

Bucek, J., Lange, K.-D. e v. Kistowski, J. (2018). SPEC CPU2017: Next-generation compute benchmark. Em *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, página 41–42, New York, NY, USA. Association for Computing Machinery.

Chen, C., Novick, G. e Shimano, K. (2000). RISC vs. CISC. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/`. Acessado em 20/08/2020.

Cloutier, F. (2019). x86 and amd64 instruction reference. `https://www.felixcloutier.com/x86/`. Acessado em 08/10/2020.

Corporation, I. (2019). Intel 64 and IA-32 architectures optimization reference manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

Degenbaev, U. (2012). *Formal specification of the x86 instruction set architecture*. Tese de doutorado, Universität des Saarlandes.

Fog, A. (2017). Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf.

Intel (2019). Intel XED. https://intelxed.github.io/. Acessado em 22/11/2020.

Köhler, R. (2019). Aceleração de cache misses prováveis através de requisições paralelas. Dissertação de Mestrado, Pós-Graduação em Informática - Universidade Federal do Paraná.

Lindner, M. (2018). libconfig: C/C++ library for processing configuration files. http://hyperrealm.github.io/libconfig/. Acessado em 08/10/2020.

Lopes, B., Auler, R., Ramos, L., Borin, E. e Azevedo, R. (2015). Shrink: reducing the isa complexity via instruction recycling. Em *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, páginas 311–322.

Morgan, T. P. (2019). Datacenters are hungry for servers again. https://www.nextplatform.com/2019/12/09/datacenters-are-hungry-for-servers-again/.

Patel, A., Afram, F., Chen, S. e Ghose, K. (2011). MARSSx86: A Full System Simulator for x86 CPUs. Em *Design Automation Conference 2011 (DAC'11)*.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A. e Karunanidhi, A. (2004). Pinpointing representative portions of large intel ® itanium ® programs with dynamic instrumentation. Em *37th International Symposium on Microarchitecture (MICRO-37'04)*, páginas 81–92.

Singh, S. e Awasthi, M. (2019). Memory centric characterization and analysis of SPEC CPU2017 suite. *CoRR*, abs/1910.00651.

SPEC (1995). Standard performance evaluation corporation. https://www.spec.org/. Acessado em 09/06/2021.

Wikichip (2016a). Cascade lake - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. Acessado em 21/03/2021.

Wikichip (2016b). Kaby lake - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Acessado em 21/03/2021.

Wikichip (2016c). Skylake (client) - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). Acessado em 01/10/2020.

Wikichip (2016d). Skylake (server) - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server). Acessado em 01/10/2020.

Yourst, M. T. (2007). PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. Em *2007 IEEE International Symposium on Performance Analysis of Systems Software*, páginas 23–34.

Yourst, M. T. (2007). PTLsim User's Guide and Reference. `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.5818&rep=rep1&type=pdf`. Acessado em 26/10/2020.