TFSim: um simulador do algoritmo de Tomasulo para apoio ao ensino de arquiteturas superescalares

Lucas Oliveira Pimenta dos Reis
Universidade Estadual de Campinas
Instituto de Computação
Campinas, Brasil
lucas.oliveirapreis@gmail.com

Liana Dessandre Duenha
Universidade Federal de Mato Grosso do Sul
Faculdade de Computação
Campo Grande, Brasil
lianaduenha@gmail.com

Abstract

Esse artigo apresenta o TFSim, um simulador funcional para a arquitetura superescalar com execução fora de ordem e especulação baseada no algoritmo de Tomasulo. O simulador utiliza arquitetura MIPS64 e uma interface gráfica completa em nível de ciclo de execução fiel às descrições presentes na literatura. O simulador foi desenvolvido com base nos padrões SystemC de simulação baseada em eventos discretos, que fornece robustez e extensibilidade para a ferramenta. Seu principal objetivo é complementar o ensino em Arquitetura de Computadores de um dos algoritmos mais importantes de execução fora de ordem presente na literatura.

1. Introdução

Durante o ensino de Arquitetura de Computadores nos cursos de graduação e pós-graduação em Ciência da Computação, Engenharia de Computação e cursos afins, os alunos devem conhecer os conceitos e a lógica necessários para conduzir um projeto de processador, entendendo como funcionam seus componentes arquiteturais e como o processador interage com demais componentes do sistema como, por exemplo, memórias e periféricos.

A abordagem adotada nas disciplinas de Arquitetura de Computadores é essencialmente teórica, baseada em exposição do conteúdo, apresentações em *slides* e discussões. Como abordagem complementar, professores adotam simuladores funcionais de conjuntos de instruções, a fim de prover a infraestrutura necessária para que os alunos façam as próprias experimentações, implementem novas funcionalidades ou simplesmente os utilizem para auxiliar no aprendizado, principalmente quando os simuladores tem uma interface interativa que permita a execução passo a passo de trechos de programas [6, 7]. Nesse contexto, existem diversos simuladores funcionais e com precisão de ciclos para processadores RISCs, máquinas virtuais ou emuladores de conjunto de instruções ou processadores mais complexos como o x86, os quais serão apresentados na seção seguinte.

No decorrer da formação do estudante nessa área, surge a necessidade de explorar paralelismo em nível de instrução e funcionalidades complexas como execução fora de ordem, despacho múltiplo e especulação. Contudo, simuladores com suporte didático a estes tópicos avançados são escassos, o que limita o professor a executar manualmente ou em apresentações baseadas em *slides* os trechos de programas que demonstrem as funcionalidades que deseja ensinar. Esta tarefa, além de limitada a pequenos exemplos, é massante é propensa a erros.

O objetivo deste trabalho é apresentar a ferramenta TFSim (do inglês, *Tomasulo Functional Simulator*), um simulador funcional para execução de instruções fora de ordem e especulação baseado no algoritmo de Tomasulo, o qual mantém a fidelidade conceitual do projeto descrito por Patterson e Hennessy[5]. O simulador utiliza os componentes de modelagem e escalonador baseado em eventos discretos do SystemC [13] para garantir simulação de processos concorrentes, sendo, para o melhor do conhecimento dos autores, o primeiro simulador superescalar do Algoritmo de Tomasulo baseado em SystemC.

Este artigo está organizado como segue: a Seção 2 apresenta os trabalhos relacionados, expondo simuladores funcionais de processadores utilizados para fins acadêmicos, didáticos ou para trabalhos de pesquisa na área; a Seção 3 apresenta o referencial teórico para embasar este projeto; a Seção 4 apresenta a metodologia

para desenvolvimento do projeto, as funcionalidades do simulador desenvolvido, bem como algumas importantes propriedades relativas ao gerenciamento da simulação pelo SystemC; a seção 5 apresenta as métricas utilizadas para validação do simulador; a Seção 6 conclui esse artigo.

2 Trabalhos Relacionados

Esta seção apresenta como simuladores funcionais têm sido utilizados durante o processo de ensino de Arquitetura de Computadores e disciplinas afins.

Zeferino et al. [17] demonstrou o uso da família de processadores BIP (do inglês *Basic Instruction-set Processor*), a qual foi concebida para auxiliar na aprendizagem de conceitos de arquitetura de computadores em distintas disciplinas da grade do curso de Ciência da Computação (Algoritmos e programação, Circuitos digitais, Compiladores, entre outras), permitindo que fosse dado um enfoque interdisciplinar aos conceitos de arquitetura e organização de computadores [7].

Um dos primeiros simuladores do conjunto de instruções MIPS desenvolvido para fins acadêmicos foi o MIPSim [10], no qual é explorado o comportamento dinâmico do processador no nível organizacional como, por exemplo, valor dos registradores, entradas e saídas dos multiplexadores e comparadores.

O DIMIPSS [9] é um software multiplataforma de simulação do caminho de dados e de sinais de controle para execução de instruções do processador MIPS Monociclo. O simulador considera como entrada um programa em linguagem de montagem para MIPS, converte-o para linguagem de máquina e representa graficamente o comportamento das instruções durante a execução.

O WebMIPS foi proposto por Branovic et al. [4] e é um simulador MIPS cuja maior vantagem é seu acesso via web sem necessidade de instalações ou configurações prévias. Esse simulador inclui visão dos estágios de pipeline, memórias, registradores e unidades de forwarding.

Ainda no contexto de simuladores MIPS, um dos mais utilizados em sala de aula é o MARS MIPS Simulator [15, 16]. Podemos destacar também ArchC, uma linguagem de descrição de arquitetura que permite a geração de instâncias de processadores, compiladores e montadores (*cross-compilers*) e diversos outros componentes de um sistema computacional. Atualmente, o ArchC disponibiliza quatro modelos de processadores que serão utilizados neste trabalho e estão descritos na Seção 4.

Entre os simuladores voltados para ensino, é interessante dar enfoque àqueles que possuem estruturas de pipeline visuais, incluindo o algoritmo de Tomasulo. O dlxview [18] foi um dos primeiros a ser construído com enfoque na parte visual do ensino, adaptado do simulador dlxsim. Baseado na arquitetura dlx (desenvolvida da arquitetura MIPS por Hennessy e Patterson), seu objetivo principal era apresentar uma representação visual das estruturas de um microprocessador, permitindo ao usuário se movimentar livremente entre os ciclos de simulação, escolher entre a execução de um pipeline comum, o uso do algoritmo de *scoreboarding* ou o próprio algoritmo de Tomasulo. Suas unidades funcionais são altamente adaptáveis, sendo possível alterar sua quantidade e latência de tempo de execução de instruções.

O simulador apresentado neste artigo procura sintetizar algumas das características apresentadas em outros simuladores já observados na literatura, como a visualização do conteúdo de todas as unidades de execução ciclo-a-ciclo, a modificação dos tempos de latência de cada instrução e do número de unidades de reserva. Seu foco principal de utilização é no aprendizado do algoritmo de Tomasulo em disciplinas de arquitetura de computadores. Entretanto, utilizando o *framework* de simulação baseada em eventos do SystemC, é possível garantir a possibilidade de expansão de seu desenvolvimento, de forma a capacitar o simulador a ser utilizado como ferramenta para outros projetos de simulação.

3 Referencial Teórico

O Algoritmo de Tomasulo [14] corresponde a uma técnica descrita por Robert Tomasulo para embasar projetos de processadores com escalonamento dinâmico desenvolvido na década de 60 para o IBM 360/91 [2]. Inicialmente, fazia parte da unidade de ponto flutuante da máquina, e tinha como objetivo aumentar o desempenho de instruções desta categoria [5].

O objetivo dessa seção é apresentar os desafios iniciais para suporte à execução fora de ordem (Seção 3.1), as estruturas necessárias e a lógica para implementar o algoritmo de Tomasulo como descrito por Hennessy e Patterson [5] (Seção 3.2) e uma visão geral sobre o *kernel* de simulação do SystemC [13] utilizado para desenvolvimento do simulador (Seção 3.3).

3.1 Desafios para execução fora de ordem

Antes de explicarmos sobre as estruturas em hardware para dar suporte à execução fora de ordem, é necessário compreender quais são os potenciais conflitos que podem ser gerados.

Uma *dependência verdadeira* de dados ocorre quando uma instrução precisa de um operando que está sendo produzido por outra instrução que encontra-se anteriormente no código original. Por exemplo, o código MIPS64[5] a seguir ilustra a dependência verdadeira entre a instrução `SUB.D` e a instrução `ADD.D`, pois o primeiro operando fonte da instrução de subtração (esperado no registrador R1) deve ser o dado produzido pela instrução de soma anterior[5].

```
ADD.D R1,R2,R3
...
SUB.D R4,R1,R5
```

O conflito resultante da execução da instrução `SUB.D` antes da instrução `ADD.D` é denominado *RAW* (*Read After Write*), demonstrando que é um erro de execução a instrução `SUB.D` ler o dado em `R1` antes da instrução `ADD.D` ter realizado a escrita do dado atualizado com o resultado da soma no mesmo registrador.

Em um processador com pipeline, conflitos RAW quebram a integridade dos dados, por consequência, seria necessário atrasar a instrução de leitura de um dado até que a instrução que produz o dado seja terminada, causando *stalls* ou atrasos na execução. Visando evitar o desperdício dos ciclos perdidos durante esse atraso, fortificou-se a ideia de execução fora-de-ordem, utilizando o tempo de processamento deste intervalo para executar instruções que não possuem dependências pendentes. Porém, seu surgimento faz com que ocorram outros tipos de conflitos de dados, que serão tratados a seguir [5].

Há outros tipos de dependência de dados entre instruções: as antidependências e as dependências de saída (ou de nome). Ambas não causam problemas em uma execução em ordem, porém podem produzir dados corrompidos, quando executadas fora de ordem. As instruções abaixo demonstram uma *antidependência* existente entre as instruções soma e substração [5]:

```
ADD.D R2,R1,R3
...
SUB.D R1,R4,R5
```

Se estas instruções forem executadas fora de ordem e a subtração terminar antes da soma, ocorrerá um conflito do tipo *WAR* (*Write After Read*), ou seja, uma escrita em R1 (pelo `SUB.D`) antes da leitura de R1 (`pelo ADD.D`), resultando em um erro de execução.

Similarmente, uma *dependência de saída ou de nome*, como ilustrada no trecho de código abaixo, pode resultar em um conflito *WAW* (*Write Aftwer Write*) e, consequentemente, um erro de execução, caso a escrita em `R1` feita pelo `SUB.D` aconteça antes da escrita em `R1` feita pelo `ADD.D`.

```
ADD.D R1,R2,R3
...
SUB.D R1,R4,R5
```

Dependências verdadeiras impõem que as instruções sejam executadas na mesma ordem em que aparecem no programa. Antidependências e dependências de saída, quando tratadas adequadamente, não fazem restrição de ordem entre as instruções dependentes. Um dos mecanismos adotados para tratar antidependências e dependências de saída é renomeação de registradores. Por exemplo, o quadro apresentado na tabela 1 mostra a aplicação de renomeação de registradores para eliminar antidependências e dependências de saída, sem que isso altere o comportamento do código original e o fluxo dos dados.

O principal objetivo da execução fora de ordem é permitir que instruções que já tenham disponíveis os seus operandos possam ser executadas, independentemente da ordem em que aparecem no programa original, respeitando as dependências de dados verdadeiras e eliminando atrasos causados por antidependências ou dependências de saída.

Tabela 1: Quadro para demonstrar como a renomeação de registradores pode eliminar antidependências e dependências de saída

Trecho de código original	Após renomeação (elimina antidependência)

`ADD.D R2,R1,R3`
...
`SUB.D R1,R4,R5`
	`ADD.D R2,R1,R3`
	...
	`SUB.D R6,R4,R5`

Trecho de código original	Após renomeação (elimina dependência de saída)

`ADD.D R1,R2,R3`
...
`SUB.D R1,R4,R5`
	`ADD.D R1,R2,R3`
	...
	`SUB.D R6,R4,R5`

O despacho de instruções em ordem garante que as dependências verdadeiras serão avaliadas e tratadas para evitar os conflitos RAW. Para evitar conflitos WAR e WAW, será adotado o mecanismo de renomeação de registradores [5].

3.2 Algoritmo de Tomasulo

A ideia-base em sua implementação é de que instruções devem ser despachadas em ordem, porém podem ser executadas fora de ordem caso não haja dependências verdadeiras entre elas.

A Figura 1 ilustra a estrutura básica de um processador com suporte a instruções de ponto-flutuante, inteiros e de acesso à memória.

Uma vez despachadas, as instruções são armazenadas em *estações de reserva* até que finalizem a sua execução e produzam resultados utilizando os valores dos operandos, caso já estejam prontos, ou fazem referência às estações de reserva que armazenam instruções que os produzirão. Assim, as referências aos operandos fontes deixam de ser a identificação dos registradores e passam a ser referências às estações de reserva que produzirão estes operandos. Esse processo é considerado como renomeação de registradores e, portanto, resolve conflitos WAW e WAR.

Assim que as instruções terminam sua execução, elas aguardam dentro do *buffer de reordenação* (ROB - *reorder buffer*) para que possam ser completadas na mesma ordem em que foram despachadas, ou seja, na mesma ordem em que aparecem no programa. Desta forma, mantém-se o fluxo do programa e são impedidas exceções fora de ordem. Uma instrução só será considerada finalizada quando passar pelo estágio de completamento (ou *commit*). Nesse estágio, a escrita dos dados produzidos pela instrução será realizada em registrador ou na memória. Similarmente, no caso de interrupções ou exceções, estas serão, de fato, tratadas no estágio de completamento.

O ROB é também imprescindível para o suporte à *especulação*, que corresponde à técnica de se executar instruções que tenham dependências de controle, ou seja, que dependam do resultado de uma instrução de desvio que ainda não foi completada. Caso seja verificado que uma instrução que foi executada de maneira especulativa não deveria ter sido executada, seus efeitos devem ser cancelados. Isso é resolvido removendo-a do ROB e não permitindo o seu completamento.

3.3 SystemC

O simulador foi descrito utilizando SystemC, o que garante robustez na modelagem, ao permitir a simulação de processos concorrentes por meio de um *kernel* de simulação baseado em eventos discretos.

O SystemC é um conjunto de classes em linguagem C++ que fornece uma interface de simulação baseada em eventos. Sua estrutura permite que o projetista simule processos concorrentes, que são sincronizados com eventos no tempo. Muito utilizado por projetistas de SoCs (*System on a Chip*), é uma ferramenta muito poderosa para simular hardware em alto nível de abstração, tornando a interação entre processos mais fácil e robusta [3]. Apesar de apresentar a possibilidade de simular processos concorrentes, a execução de programas gerados utilizando SystemC é sempre sequencial, sem permitir execução verdadeiramente paralela em sua versão original.

Os componentes que estruturam SystemC são descritos como *módulos* e o comportamento dos módulos é descrito por meio de *processos*. A execução dos processos é guida por eventos, os quais garantem a comunicação e a sincronizam entre os componentes do modelo.

O *kernel* SystemC mantém informações sobre o conjunto de processos e também a fila de eventos. Inicialmente, todos os processos são considerados prontos, a não ser que seja informado o contrário; enquanto houver processos prontos, um deles é selecionado para iniciar a execução e permanece executando até sua finalização, ou até que inicie a espera por algum evento. Se o processo corrente gerar eventos, todos os processos sensíveis a esses eventos tornam-se imediatamente prontos. A verificação de prontidão e execução constitui a fase de avaliação [11, 8].

A escolha por SystemC nesse projeto é motivada por seu *kernel* de simulação robusto e pelas diversas metodologias de comunicação e sincronização fornecidas pela linguagem e suas extensões (TLM2), o que possibilita que refinamentos do nível de abstração e inclusão de temporização possam ser feitos sem excessivo esforços.

4 Metodologia

O simulador para o algoritmo de Tomasulo foi descrito de forma a ser o mais fiel possível à proposta original. Entretanto, sua estrutura modularizada possibilita a inserção de novos componentes, novas funcionalidades aos componentes existentes, novos preditores de desvio, novas instruções, modificação do tempo de execução de cada instrução, e possivelmente pode ser usado até para *benchmarking*, desde que baseado em um conjunto reduzido de instruções.

4.1 Conjunto de Instruções

O conjunto de instruções abordado é herdado da Arquitetura **MIPS64**. Embora a ideia deste trabalho seja inicialmente de uma ferramenta educacional, é muito importante que o conjunto de instruções válidas

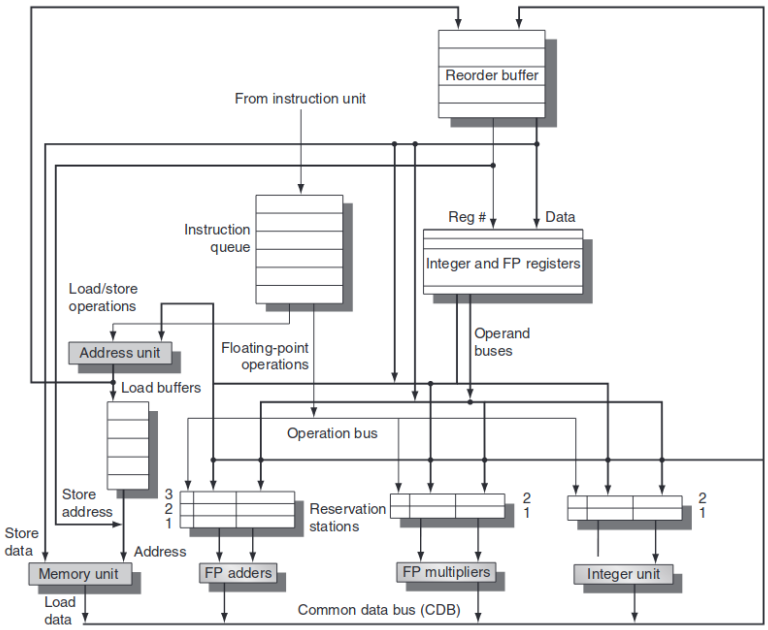

Figura 1: Estrutura simplificada do hardware do Algoritmo de Tomasulo, de [5]

seja o mais amplo possível. Portanto, o simulador é desenvolvido tendo em mente a escalabilidade do código, de modo que seja simples acrescentar novas instruções. O conjunto de instruções proposto inicialmente para o simulador é exibido a seguir:

- Instruções de memória:
  `LD` e `SD`;
- Instruções aritméticas:
  `DADD`,`DADDI`,`DSUB`,`DMUL`,`DDIV` e `DDIVU`;
- Instruções de desvio:
  `BEQ`,`BNE`,`BGTZ`,`BLTZ`,`BGEZ` e `BLEZ`;

4.2 Módulos de hardware para suporte ao Algoritmo de Tomasulo

4.2.1 Estações de Reserva (RS)

A base do algoritmo de Tomasulo são as estações de reserva, que são estruturas em hardware para armazenar as instruções e seus operandos após o despacho e durante sua execução.

Durante o despacho, é verificado se os operandos de uma instrução já estão disponíveis nos registradores fontes. Caso um operando esteja disponível, seu valor é lido do banco de registradores e armazenado em um campo específico da estação de reserva que está sendo preenchida. Caso contrário, as referências aos operandos fontes faltantes deixam de ser a identificação dos registradores e passam a ser referências às estações de reserva que produzirão estes operandos ou à linha do buffer de reordenação, caso o hardware em questão tenha suporte à especulação (descrição do mecanismo de especulação na Seção 4.2.3). Esse processo é considerado como renomeação de registradores e, portanto, resolve conflitos WAW e WAR causados pelas antidependências e dependências de saída durante a execução fora de ordem.

Nesta implementação, serão utilizados dois tipos diferentes de **RS**s:

- *Transferência de Dados* (MEM) - Usada apenas para instruções do tipo Load/Store;
- *Manipulação de instruções aritméticas* (INT)- Usada para instruções que manipulam dados inteiros e desvios, podem ser expandidas para uso de instruções de ponto flutuante;

A quantidade de cada tipo de **RS** é customizável pelo usuário, bem como o tempo de execução de cada instrução (latência). Entretanto, utilizou-se valores padrão, baseados nas latências utilizadas em [5], para cada um um destes atributos, garantindo um primeiro contato mais simples com o programa.

4.2.2 Barramento Comum (CDB)

O barramento comum de dados é uma parte essencial do algoritmo de Tomasulo, responsável por compar-

tilhar com todas as unidades funcionais o resultado da última instrução que terminou de produzir um resultado. Isso garante que outras instruções com dependências verdadeiras solucionadas por esse resultado possam ter seus operandos atualizados e possam iniciar a execução no próximo ciclo de clock (salvo nos casos em que não há unidades funcionais disponíveis, o que ocasionaria, inevitavelmente, atrasos por conflitos estruturais).

4.2.3 Buffer de reordenação (ROB)

É uma extensão em hardware de execução fora de ordem para dar suporte à especulação. Prevendo com altas taxas de acerto o resultado dos desvios, podemos despachar instruções de maneira especulativa, o que diminui consideravelmente os atrasos causados por dependências de controle. Entretanto, quando os desvios são preditos incorretamente, deve-se garantir que instruções especuladas não completem.

O mecanismo de especulação garante que nenhuma das instruções que ainda estão sendo executadas especulativamente possam escrever permanentemente na memória ou no banco de registradores. Podemos entendê-lo como um buffer que armazena todas as instruções por ordem de despacho. Conforme cada instrução termina sua execução, ela é removida do **ROB** e finalmente seu resultado pode ser escrito definitivamente (passo denominado *completamento* ou *commit*).

Caso seja confirmado que um desvio foi predito incorretamente, todas as instruções abaixo da predição são removidas do buffer sem efeitos colaterais permanentes no armazenamento dos dados. Após o término da execução de uma instrução e antes de seu completamento, qualquer valor buscado por uma **RS** pode ser encontrado no **ROB**, e não no banco de registradores. A Figura 2 apresenta a interface gráfica do simulador, que está organizada da seguinte maneira:

- (1) Fila de Instruções: Aqui se encontram as instruções que serão executadas durante a simulação, contendo informações sobre o caminho que já percorreram (despacho, execução, escrita e completamento)
- (2) Estações de Reserva: Informa as estações de reserva presentes no simulador, informando se estão ativas, quais operandos estão prontos e de quais estações serão obtidos os operandos que ainda não estão prontos
- (3) Banco de registradores (juntamente com seus status): Informa o valor atual de todos os registradores presentes, além de informar qual estação de reserva (ou posição do ROB) está programado para escrever em cada um dos registradores
- (4) Estrutura de Memória: Conjunto de valores de memória
- (5) Buffer de Reordenação: Estrutura que informa quais instruções estão no buffer de reordenação, apresentando seu destino e valor resultante (caso já tenha sido computado)

4.3 Estrutura da Implementação

O simulador é descrito utilizando as funcionalidades da biblioteca *SystemC*, de modo que a estrutura das classes e objetos criados seguem o padrão descrito em [13]. Cada classe é derivada de um módulo do *SystemC* (`sc_module`), que apresenta um ou mais métodos especiais, chamados de processos.

Dentro da terminologia dessa biblioteca, um processo é um conjunto de instruções específico, que cumpre o papel do processamento das atividades do hardware. Esses processos competem em tempo de execução assim como em um sistema operacional, e o kernel do *SystemC* administra seu escalonamento.

A comunicação entre os processos se dá exclusivamente entre canais, que são classes que apresentam a implementação de interfaces virtuais de envio e recebimento de dados. Essas mesmas interfaces são utilizadas dentro dos módulos como objetos responsáveis por realizar a comunicação com canais, e são denominadas de portas.

O desafio de sincronizar a comunicação e ordenação dos processos é parte fundamental do uso de *SystemC*, já que, sem sincronização, os processos são retirados da fila aleatoriamente.

A implementação do simulador consiste em um conjunto de métodos para cada estrutura principal do algoritmo de Tomasulo, além de alguns auxiliares necessários para organizar o escalonamento de processos. Suas funcionalidades podem ser subdivididas em dois possíveis modos de execução, com e sem especulação. Por simplicidade, será detalhado neste texto apenas o modo com especulação.

O diagrama da Figura 3 exemplifica a implementação do hardware para suporte ao algoritmo de Tomasulo com as estruturas necessárias para especulação. Nela, retângulos descrevem os módulos .Para simplificar o entendimento, a figura não possui todos os canais necessários para comunicação, com exceção do CDB. As classes definidas como módulos são:

- CLOCK: Dispara um sinal de clock a cada instante de simulação

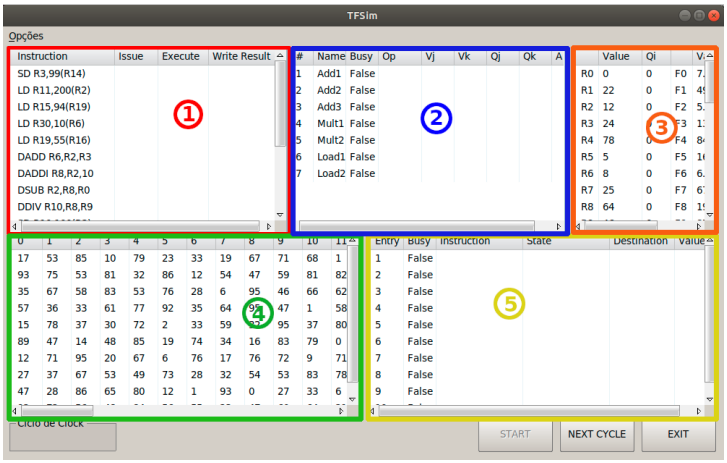

Figura 2: Estrutura da interface gráfica do simulador em execução no modo de especulação. Em ordem, está organizada: 1 - Fila de instruções, 2 - Estações de Reserva, 3 - Banco de Registradores, 4 - Estrutura de Memória, 5 - Buffer de Reordenação

- FILA DE INSTRUÇÕES: Envia uma instrução a cada sinal de clock e controla o PC
- CONTROLE DE DESPACHO: Interpreta cada instrução e define para qual módulo deve despachá-la
- CONTROLE DO BUFFER DE LOADS E STORES: Estrutura de controle do buffer de Stores/-Loads. Fragmenta as instruções, obtém as informações necessárias (registradores e seus valores, se já estão ocupados, se há estações de reserva livres,...) e despacha a instrução para uma das estações de reserva inscritas
- CONTROLE DE ESTAÇÕES DE RESERVA: Possui as mesmas tarefas do buffer de loads e stores, porém recebe apenas instruções aritméticas
- ESTAÇÃO DE RESERVA: Também realiza o trabalho das unidades funcionais. Recebe e executa instruções, lê e grava no CDB
- MEMÓRIA: Memória. Para onde são enviadas instruções de Load e Store, após o cálculo de seu endereço nas estações de reserva
- BANCO DE REGISTRADORES: Banco de registradores. Armazena os valores dos registradores inteiros e de ponto-flutuante, além do seu status (se há registrador esperando para escrever nele)
- UNIDADE DE ENDEREÇOS: Unidade de endereçamento, calcula os endereços de funções de acesso a memória antes de enviá-las para execução. Ela executa o controle de modo a impedir que haja conflito entre cálculo de endereço de loads e stores.
- BUFFER DE REORDENAÇÃO: Responsável pelo controle da especulação, armazenando todas as instruções e executando a fase de completamento em ordem.

Uma mudança importante dessa implementação em relação ao modo não-especulativo é a atribuição de instruções de store, que deixam de ser enviadas ao *buffer de loads e stores*, e tem sua execução realizada inteiramente pelo buffer de reordenação. Também é importante observar que as estações de reserva em si deixam de acessar o banco de registradores, já que quem trata de escritas nos registradores por si só é apenas o buffer de reordenação. Elas também deixam de acessar a memória para realizar escrita, apenas para realizar leituras em instruções de *load*, que são enviadas diretamente ao buffer de reordenação.

A interface do simulador é desenvolvida utilizando Nana [1], uma biblioteca *open-source* de interface gráfica multiplataforma adequada aos padrões modernos de programação em C++. Por não se tratar de um framework e sim de uma biblioteca, é muito leve, o que leva à praticidade em sua instalação e compilação e evita conflitos com SystemC.

Considerando a possibilidade de expansão do simulador, sua modularidade é um dos principais fatores de enfoque neste trabalho. Uma consequência positiva dessa decisão de projeto está na facilidade de implementar preditores de desvio. O preditor de desvios costuma ser um fator muito importante em qualquer arquitetura, já que sua eficiência é essencial para garantir que menos ciclos de execução sejam desperdiçados. Portanto, a possibilidade de inserção de novos preditores no simulador o torna uma boa ferramenta de ben-

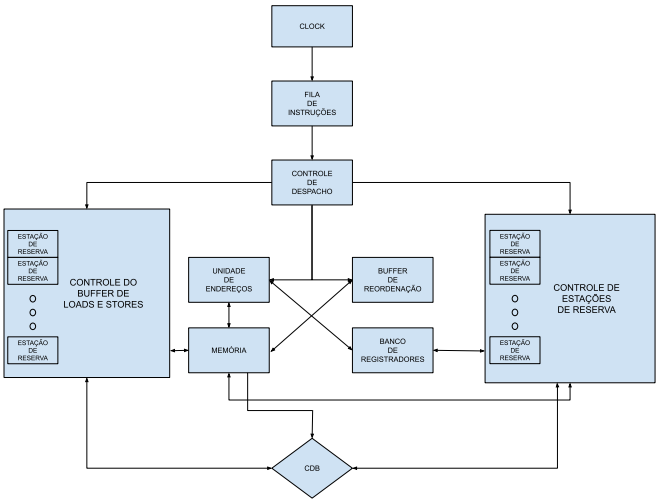

Figura 3: Diagrama de relação entre processos para a implementação com especulação por hardware

chmarks.

Outro fator considerado em relação à modularidade é a possibilidade de expansão do conjunto de instruções. Toda a execução das instruções é dada pelas estações de reserva, e portanto, é simples modificar sua estrutura de modo a aceitar novos conjuntos de instruções. Desde que se encaixem nos subtipos já citados, elas podem ser implementadas apenas alterando alguns vetores responsáveis pelo controle do tempo de latência das instruções, e adicionando seu código de execução na implementação da estação de reserva.

5 Validação

A fim de validar a execução da aplicação, foram realizados experimentos de pequenos trechos de código que refletem seu uso cotidiano por alunos. O simulador MARS [15, 16] foi utilizado como comparação, considerando sua confiabilidade e extenso uso na literatura e para apoio ao ensino.

Conjuntos de trechos de código foram executados tanto no ambiente do simulador de Tomasulo quanto no simulador MARS. Ao final de sua execução, os valores de registradores e da memória principal exibidos pelo MARS são comparados com aqueles exibidos pelo simulador de Tomasulo. A confiabilidade do simulador se dá pela confirmação da similaridade entre esses dados.

Inicialmente, pequenas adaptações foram feitas no conjunto de instruções presentes, já que o simulador MARS é baseado na arquitetura MIPS, enquanto o simulador Tomasulo utiliza o conjunto de instruções MIPS64. Além disso, por decisão de design, todos os regsitradores presentes no simulador Tomasulo são de propósito geral, enquanto o simulador MARS segue fielmente as especificações da arquitetura MIPS. Portanto, o conjunto de registradores a ser comparado deve ser limitado.

Uma funcionalidade do TFSim permite a leitura de arquivos de texto possuindo os valores finais dos registradores e da memória, retornando o resultado simplificado das discrepâncias entre os valores.

O conjunto de códigos utilizados para validação neste artigo são referentes ao cálculo dos primeiros números da sequência de *Fibonacci*, uma busca simples em um vetor não-ordenado, uma demonstração do *stall* de execução por conta de uma instrução lenta (`DDIV`), um teste de stress de execução com múltiplos acessos a memória e um último teste de stress forçando conflitos por uso excessivo de estações de reserva de adição. Os trechos de código disponíveis podem ser vistos na Tabela 2.

O trecho referente ao algoritmo de Fibonacci computa os dez primeiros valores da sequência, armazenados nos registradores `R2` e `R3`. Já o trecho referente à busca no vetor analisa em sequência um vetor de 15 elementos em busca do valor 61. Caso seja encontrado, o valor do registrador `R5` é alterado para 1. Os trechos restantes apresentam conjunto repetitivo de instruções a fim de estressar o simulador, demonstrando a maneira como o TFSim lida com conflitos.

O simulador apresenta em sua interface um menu específico para execução dos trechos citados acima, a fim de facilitar seu uso em ambiente de ensino.

Todos os trechos de código apresentaram valores idênticos para os registradores inteiros e de ponto-flutuante (levando em consideração as limitações para registradores de propósito geral), assim como em trechos de memória. Pequenos trechos de código menos significativos também foram executados e obtiveram resultados semelhantes. A Figura 4 mostra o TFSim em execução do trecho de código referente ao cálculo da sequência de Fibonacci. Nela, é possível observar como o simulador denota a execução de um trecho de código. Na fila de instruções, há demarcação indicando a última instrução a ser despachada. Também é possível observar que o TFSim indica, para cada instrução, se já foi despachada, se já iniciou sua execução e se o resultado já foi escrito. Para instruções que ainda estão em execução, as estações de reserva denotam os valores já obtidos, e apontam para outras estações que possuem os valores que ainda estão sendo calculados. O buffer de reordenação apresenta todas as instruções que ainda não passaram pela fase de completamento, indicando o destino e o valor do resultado obtido. O banco de registradores indica se há estação de reserva

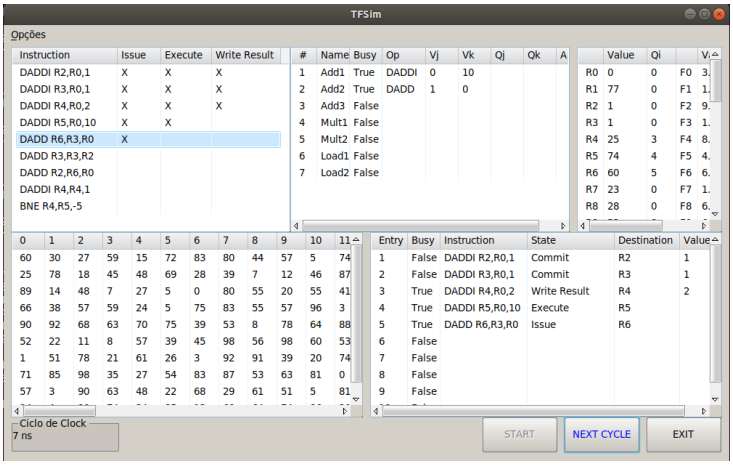

Figura 4: Execução com especulação por hardware do trecho de código de *Fibonacci*

Fibonacci	Vector Search	Stall por Divisão	Stress de Memória	Stress Aritmético
`DADDI R2,R0,1`	`DADDI R2,R0,45`	`LD R2,0(R0)`	`DADDI R2,R0,0`	`DADD R5,R0,R1`
`DADDI R3,R0,1`	`DADDI R3,R0,0`	`LD R3,4(R0)`	`DADDI R3,R0,64`	`DADD R3,R0,R2`
`DADDI R4,R0,2`	`DADDI R5,R0,0`	`DADD R5,R2,R3`	`SD R2,0(R2)`	`DADD R7,R0,R2`
`DADDI R5,R0,10`	`LD R6,0(R3)`	`DDIV R4,R2,R3`	`SD R2,4(R2)`	`DADD R3,R0,R1`
`DADD R6,R3,R0`	`DADDI R6,R6,-61`	`DSUB R6,R4,R2`	`SD R3,8(R2)`	`DADD R8,R0,R4`
`DADD R3,R3,R2`	`BEQ R6,R5,4`	`DADD R7,R4,R3`	`SD R4,16(R2)`	`DADD R2,R3,R4`
`DADD R2,R6,R0`	`DADDI R3,R3,4`	`DMUL R3,R6,R7`	`SD R5,32(R2)`	`DMUL R2,R1,R6`
`DADDI R4,R4,1`	`BNE R3,R2,-4`		`SD R6,64(R2)`	`DADD R5,R2,R1`
`BNE R4,R5,-5`	`BEQ R0,R0,2`		`LD R4,16(R2)`	
	`DADDI R5,R5,1`		`LD R5,32(R2)`	
			`LD R6,64(R2)`	
			`SD R7,128(R2)`	
			`SD R8,256(R2)`	
			`SD R9,(R2)`	
			`DADDI R2,R2,4`	
			`BNE R2,R3,-9`	

Tabela 2: Trechos de código utilizados para validação do simulador TFSim

que ainda irá escrever no registrador (no índice Q_i), além de armazenar o valor atual de cada registrador. Por fim, a memória apresenta o conjunto de valores inteiros utilizado pelas instruções de Load e Store.

5.1 Usabilidade em Sala de Aula

Devido ao fato do desenvolvimento do simulador ser extremamente recente, não houve tempo hábil para realização de testes de usabilidade em sala de aula. Entretanto, os autores acreditam que as funcionalidades presentes no TFSim o tornam uma ferramenta de bom valor para ensino. Em especial, o fato do projeto do simulador possuir foco apenas no algoritmo de Tomasulo o torna mais acessível para alunos que ainda possuem pouco contato com outros simuladores, que normalmente apresentam uma curva de aprendizado maior em relação à complexidade de uso da ferramenta.

Também é válido notar a quantidade considerável de benchmarks já presentes no TFSim, que apresentam trechos variados de código, possibilitando ao aluno observar o funcionamento do algoritmo de Tomasulo em várias possíveis situações de conflitos. Por ser um projeto fiel àquele descrito por Patterson e Hennessy [5], também possibilita que estudantes estejam mais familiarizados com o seu funcionamento, já que este livro é altamente utilizado em disciplinas de Arquiteturas de Computadores em todo o mundo.

6 Conclusão

Este artigo apresentou um simulador de execução de instruções MIPS64 fora de ordem, com base no Algoritmo de Tomasulo. Esta ferranenta pode ser utilizada para complementar o ensino de tópicos avançados do projeto de processadores como, por exemplo, a execução fora de ordem e especulação, nas disciplinas de Arquitetura de Computadores. A implementação do simulador foi feita em SystemC, o que possibilitou desenvolver um projeto mais fiel à proposta original do hardware proposto por Tomasulo, e fornecendo o

suporte para o desenvolvimento de novas funcionalidades de temporização, sincronização e refinamentos do nível de abstração do projeto inicial. O projeto foi validado usando comparação dos dados produzidos pela execução de trechos de códigos MIPS64 no simulador MARS e no simulador TFSim.

O potencial de expansão do simulador nos permite propor uma vasta gama de trabalhos futuros, entre eles:

- a criação e avaliação de novos preditores de desvio;
- uma ampliação da interface gráfica com intuito de mostrar a interação entre as estruturas do algoritmo;
- a implementação de uma estrutura de despacho múltiplo;
- a extensão do conjunto de instruções;
- o simulador poderia se beneficiar de uma expansão de sua estrutura básica, procurando seguir mais fielmente as especificações do ISA MIPS64, com a criação de chamadas de função e um sistema de pilha e de controle de contexto;

O progresso feito no desenvolvimento do simulador, assim como instruções de sua instalação e execução podem ser encontrados em sua página do Github em [12].

Referências

[1] Nana c++ library, 2015(acessed December 8,2018). `http://nanapro.org/`.

[2] D. Anderson, F. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[3] D. Black and J. Donovan. *SystemC: From the Ground Up*. Kluwer Academic Publishers, 2004.

[4] I. Branovic, R. Giorgi, and E. Martinelli. Webmips: a new web-based mips simulation environment for computer architecture education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 19. ACM, 2004.

[5] J. L. H. David A. Patterson. Arquitetura de computadores: Uma abordagem quantitativa. 2012.

[6] L. Duenha and R. Azevedo. Utilização dos simuladores do mpsocbench para o ensino e aprendizagem de arquitetura de computadores. *International Journal of Computer Architecture Education (IJCAE)*, 5(1):26–31, 2016.

[7] L. Duenha, F. Crominski, M. T. Santos, and R. Ribeiro. Avaliação de preditores de desvios por meio de simuladores como parte do processo de ensino e aprendizagem de arquitetura de computadores. *International Journal of Computer Architecture Education (IJCAE)*, 6(1):1–9, 2017.

[8] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87. July 2009.

[9] A. Felix, C. Pousa, and M. Carvalho. Dimipss: Um simulador didático e interativo do mips. In *Workshop sobre Educação em Arquitetura de Computadores*, pages 49–52, 2006.

[10] H. Grunbacher and H. Khosravipour. Windlx and mipsim pipeline simulators for teaching computer architecture. In *Engineering of Computer-Based Systems, 1996. Proceedings., IEEE Symposium and Workshop on*, pages 412–417. IEEE, 1996.

[11] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably Distributed SystemC Simulation for Embedded Applications. In *Proceedings of the International Symposium on Industrial Embedded System (SIES 2008)*, pages 271–274. 2008.

[12] L. Reis and L. Duenha. Implementação do simulador systemc de um processador superescalar baseado no algoritmo de tomasulo, 2018. https://github.com/lucasreis1/Tomasulo-Algorithm-Simulator/.

[13] SystemC. IEEE Std 1666^{TM} Standard SystemC Language Reference Manual. IEEE Computer Society, January 2012.

[14] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[15] D. K. Vollmar and D. P. Sanderson. A mips assembly language simulator designed for education. *Journal of Computing Sciences in Colleges*, 21(1):95–101, 2005.

[16] K. Vollmar and P. Sanderson. Mars: an education-oriented mips assembly language simulator. In *ACM SIGCSE Bulletin*, volume 38, pages 239–243. ACM, 2006.

[17] C. A. Zeferino, A. L. A. Raabe, P. V. Vieira, and M. C. Pereira. Um enfoque interdisciplinar no ensino de arquitetura de computadores. *C. Martins, P. Navaux, R. Azevedo, S. Kofuji. Arquitetura de Computadores: educação, ensino e aprendizado*, 2012.

[18] Y. Zhang and G. B. Adams, III. An interactive, visual simulator for the dlx pipeline. In *Proceedings of the 1997 Workshop on Computer Architecture Education*, WCAE-3 '97, New York, NY, USA, 1995. ACM.