Simulador do Algoritmo de Tomasulo com Conjunto de Instruções RISC-V

Thiago de Campos R. Nolasco, Danielle D. Vieira, João Augusto S. Silva, Henrique C. Freitas

Computer Architecture and Parallel Processing Team (CArT)
Departamento de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – MG – Brasil

{tcrnolasco, ddvieira, joao.silva.452811}@sga.pucminas.br, cota@pucminas.br

Abstract. *One of the most impactful creations in the field of processors was the concept of the superscalar pipeline, whose goal was to minimize bottlenecks and meet the demands of its users, ensuring an increase in instructions per cycle. However, as a trade-off, it introduced greater architectural complexity and the need for better control over instruction dependencies. To address this issue, Robert Tomasulo devised an algorithm that is still in use today. In the mentioned article, the authors present a simulator in the C language of the Tomasulo's algorithm using the RISC-V instruction set. This simulator demonstrates the step-by-step execution of the structures envisioned by Tomasulo, the analysis of dependencies between instructions and support for branch instructions.*

Resumo. *Uma das criações mais impactantes na área de processadores foi o conceito de pipeline superescalar, cujo objetivo era minimizar gargalos e atender demandas dos seus usuários, garantindo aumento de instruções por ciclo. Porém, como trade-off, trouxe maior complexidade à arquitetura e necessidade de maior controle sobre dependências de instruções. Para solucionar essa problemática, Robert Tomasulo idealizou um algoritmo que é utilizado até os dias de hoje. No artigo em questão, é apresentado um simulador em linguagem C do algoritmo de Tomasulo com o conjunto de instruções RISC-V, que demonstra o passo a passo das estruturas idealizadas por Tomasulo, a análise de dependências entre instruções e suporte à instruções de desvio.*

1. Introdução

As tecnologias conhecidas e utilizadas nos dias de hoje (e.g., computadores, dispositivos móveis, sensores) são consideradas voláteis por estarem sob constante melhoria, e os motivos para isso variam do fato de existir a crescente necessidade de minimizar gargalos, suprir necessidades e abrir novos horizontes para os usuários. Afim de garantir essa constante evolução, arquitetos e engenheiros da área trazem, anualmente, novas tecnologias e conceitos que são aderidos pelo mercado. Um dos componentes que mais tiveram melhorias no decorrer de sua existência são os processadores, *hardware* indispensável para quase toda tecnologia.

Os três primeiros autores são estudantes de graduação em Ciência da Computação. Os autores agradecem à FAPEMIG, ao CNPq e a PUC Minas pelo suporte parcial na execução desta pesquisa.

Com o intuito de aumentar o desempenho no processamento de instruções, processadores são desenvolvidos com suporte a execução fora de ordem de instruções fazendo uso da técnica de superescalaridade. Conforme Ungerer et al. [2003], fazer o uso de um *pipeline* superescalar dentro de um *chip* aumenta consideravelmente a complexidade da arquitetura, tendo que aumentar o número de unidades funcionais e ter um controle maior das antidependências e as dependências de saída existentes entre as instruções. Pensando nessa situação, Tomasulo [1967] apresentou e implementou uma solução, utilizada até nos dias de hoje, para os problemas persistentes da superescalaridade, tratando de forma otimizada os problemas de dependências.

O problema motivador deste artigo está na relativa dificuldade de aprender conceitos fundamentais que norteiam um projeto que envolva *pipelines* superescalares baseados no algoritmo de Tomasulo. Portanto, o objetivo deste artigo é apresentar um simulador escrito em C, no qual os usuários possam entender o passo a passo do algoritmo de forma visual, além de conseguir exercitar as análises de dependências entre as instruções. A principal contribuição deste artigo está na idealização e implementação de um simulador que seja intuitivo e permita ajustes do usuário, visando ajudar no aprendizado de importantes conceitos na formação de recursos humanos em Ciência da Computação.

O restante deste trabalho está organizado da seguinte maneira. Na Seção 2, é feita uma revisão sobre outros trabalhos que implementaram simuladores do algoritmo de Tomasulo. Já na Seção 3, é descrito detalhadamente como o algoritmo foi idealizado por Tomasulo. Na Seção 4, é apresentada a implementação do simulador criado pelos autores e como utilizá-lo. Na Seção 5 é apresentado um exemplo de execução do simulador. Por fim, na Seção 6 é feita a conclusão do trabalho e a proposição de trabalhos futuros.

2. Trabalhos Relacionados

Azimi [2013] aborda a construção de um simulador para o algoritmo de Tomasulo baseado em instruções MIPS na linguagem de programação *C++*. As instruções MIPS foram projetadas para serem simples, com tamanho fixo e bem definidas. Para isso, o conjunto de instruções MIPS possui três formatos básicos, sendo eles Formato R (*Register*), que engloba operações feitas utilizando dois registradores, Formato I (*Immediate*), envolvendo instruções que necessitam de um valor imediato e um registrador e por fim, Formato J (*Jump*), englobando instruções de desvio, sendo utilizado para controlar o fluxo do programa.

Liu et al. [2016] propõem um simulador de baixo custo computacional para contribuir no aprendizado de arquitetura de computadores. A fim de incentivar os alunos a colocarem as teorias aprendidas em prática, o simulador *PipelineSim* dá suporte a simulação de um *pipeline MIPS* de cinco estágios, ao *Scoreboarding* e também ao algoritmo de Tomasulo, assim como proposto em Azimi [2013], dando aos estudantes a possibilidade de visualizar a atuação do escalonamento de instruções nas dependências verdadeiras e falsas.

Por fim, Kehagias and Douskas-Bertlviser [2017] descrevem um simulador do algoritmo de Tomasulo, desenvolvido para sistemas *Android* por meio da linguagem de programação *Java*, a fim de apresentar o funcionamento do escalonamento dinâmico de instruções. O simulador descrito pelos autores pode ser operado de modo passo a passo e com animações gráficas para facilitar o entendimento dos estudantes. O simulador pro-

posto em Kehagias and Douskas-Bertlviser [2017] se mostrou eficiente e com uma melhor percepção no impacto do escalonamento dinâmico de instruções.

O presente trabalho se diferencia dos demais por propor um simulador desenvolvido em C, voltado para o meio acadêmico sem interface interativa, mas com telas exibindo o passo a passo do que está acontecendo na execução das instruções. O ambiente de simulação proposto a partir deste trabalho faz uso do conjunto de instruções *RISC-V* [Patterson and Hennessy, 2020; Patterson and Waterman, 2019], que por sua vez, foi adotado neste trabalho por sua simplicidade, modularidade e flexibilidade, além de ser uma proposta mais recente de conjunto de instruções. Ademais, o simulador apresenta possibilidade de manipular a quantidade e tipagem das unidades funcionais através de pequenas alterações no código fonte.

3. Algoritmo de Tomasulo e Instruções RISC-V

Nesta seção são detalhadas características do algoritmo de Tomasulo e o conjunto de instruções RISC-V, suportado no simulador desenvolvido.

3.1. Algoritmo de Tomasulo

Conforme apresentado em Tomasulo [1967], o algoritmo de Tomasulo, proposto pelo cientista da computação Robert Marco Tomasulo no ano de 1967, tem a intenção de melhorar o desempenho na execução paralela de instruções no processador e minimizar o gargalo causado por dependências de dados através da divisão das unidades funcionais e renomeação de registradores.

As unidades funcionais são componentes de *hardware* especializados projetados para realizar operações específicas, como adição, multiplicação, divisão ou acesso à memória. No algoritmo de Tomasulo, as estações de reserva são *buffers* associados a cada unidade funcional que armazenam instruções pendentes que estão aguardando execução nessa unidade.

O algoritmo de Tomasulo não elimina as dependências verdadeiras, também conhecidas como dependências de dados ou dependências de leitura-após-escrita (RAW - *Read After Write*). Porém, as dependências falsas podem ser eliminadas, pois, conforme em Hennessy and Patterson [2011], as antidependências, escrita-após-leitura (WAR - *Write After Read*), e as dependências de saída, escrita-após-escrita (WAW - *Write After Write*), são dependências de nome, ao contrário das dependências verdadeiras, já que não existe valor sendo transmitido entre as instruções. Como uma dependência de nome não é uma dependência verdadeira, as instruções envolvidas em uma dependência de nome podem ser executadas simultaneamente ou ser reordenadas se o nome (número de registrador ou local de memória) usado nas instruções for alterado de modo que as instruções não entrem em conflito. Portanto, a remoção das dependências falsas é feito por meio da renomeação de registradores para ocorrer mais execuções em paralelo.

Para a escrita em registrador ocorrer em ordem e sem conflitos é utilizado o *buffer* de reordenamento, que, apesar das instruções serem executadas fora de ordem para melhorar o desempenho no processador, este *hardware* auxilia e realiza a escrita em ordem no banco de registradores. A escrita no banco de registradores ocorre quando é realizado o *commit* de uma instrução e o *commit* só pode ser feito se as instruções anteriores a ela no *buffer* de reordenamento já estiverem com seu *commit*.

Instruções de memória	
SW R1, I(R2)	Operação de escrita em memória
LW RD, I(R1)	Operação de leitura de memória
Instruções aritméticas	
ADD RD, R1, R2	Operação de adição
SUB RD, R1, R2	Operação de subtração
MUL RD, R1, R2	Operação de multiplicação
DIV RD, R1, R2	Operação de divisão
REM RD, R1, R2	Operação de resto
Instruções de desvio	
BNE R1, R2, L, #V	Operação de desvio se valores são diferentes
BEQ R1, R2, L, #V	Operação de desvio se valores são iguais
Instruções de comparação	
SLT RD, R1, R2	Operação de comparação de menor valor com R
SLTI RD, R1, I	Operação de comparação de menor valor com I
Instruções de lógica	
OR RD, R1, R2	Operação de lógica *Bitwise* OR
AND RD, R1, R2	Operação de lógica *Bitwise* AND
XOR RD, R1, R2	Operação de lógica *Bitwise* XOR
SLL RD, R1, I	Operação de deslocamento lógico para esquerda
SRL RD, R1, I	Operação de deslocamento lógico para direita
SRA RD, R1, I	Operação de deslocamento aritmético para direita

Tabela 1. Instruções RISC-V suportadas no simulador deste trabalho.

Em linhas gerais, após a decodificação das instruções, estas são enviadas para o *buffer* de reordenamento na ordem FIFO (*First In, First Out*) e às estações de reserva de cada unidade funcional específica de sua execução. Nas estações de reserva as instruções que não possuem dependências verdadeiras são despachadas para execução na unidade funcional, se a mesma estiver livre. A dependência verdadeira é identificada nas estações de reserva se houver alguma instrução antes da que está em análise no *buffer* de reordenamento que ainda não foi executada e escreve em um registrador que a instrução em análise irá ler. Como há adiantamento de dados, a dependência verdadeira pode ser removida assim que a instrução na qual há dependência é executada. Após a execução, a instrução aguarda seu *commit* conforme a ordem de chegada no *buffer* de reordenamento.

O barramento de dados comum, também conhecido como CDB (*Common Data Bus*), é um componente importante no algoritmo de Tomasulo e é uma via de comunicação compartilhada utilizada para transmitir resultados de instruções concluídas para o *buffer* de reordenamento e estações de reserva (adiantamento de dados). O uso do CDB no algoritmo ajuda a minimizar o impacto das dependências de dados, permitindo que instruções subsequentes continuem sua execução assim que os dados necessários estão prontos. A execução em paralelo de instruções ocorre quando instruções que não tem dependências são executadas ao mesmo tempo nas unidades funcionais.

3.2. Instruções RISC-V

RISC-V [Patterson and Hennessy, 2020; Patterson and Waterman, 2019] é um conjunto de instruções universal (*Instruction Set Architecture* - ISA) que é *Open-Source* e atende a todos os tamanhos de processadores. Também funciona com uma grande variedade de softwares e linguagens de programação, é eficiente para vários tipos de microarquiteturas e é estável. Pelas características já citadas e por ser simples, conforme demonstrado

em Patterson and Waterman [2019], o simulador implementado suporta o conjunto de instruções RISC-V.

Na Tabela 1 estão todas as 17 instruções[1] RISC-V suportadas no simulador proposto. Os registradores *R1* e *R2* são registradores que terão seus valores lidos para realizar a operação de cada instrução. O registrador *RD* é o que terá seu valor escrito após a execução da instrução, quando houver o *commit* pelo *buffer* de reordenamento no banco de registradores.

O imediato *I* é um valor inteiro em que, nas instruções de memória, será somado com o valor do registrador base para ter o endereço em memória e, na instrução de comparação SLTI, *I* é comparado com o valor do registrador *R1*. Já nas instruções de lógica o imediato *I* informa quantos bits serão deslocados nas instruções de deslocamento lógico e aritmético.

O *label L* nas instruções de desvio é um rótulo que é associado a um local específico no código fonte, e é usado como uma referência para instruções ou endereços de memória para o qual o programa deve saltar (ou seja, mudar o fluxo de execução) se os valores nos registradores forem diferentes ou iguais. E o valor *#V* é utilizado como gabarito para saber se realmente o desvio de instruções deve ou não ser realizado, visto que em nosso simulador não é analisado se os valores dos registradores são iguais ou diferentes, pois não há a implementação do banco de registradores com seus valores armazenados. O *#V* pode ter apenas dois valores, 0 ou 1, se 0, significa que não deve ter o salto, se 1, deve-se saltar para o label especificado.

4. Proposta

Na idealização do simulador, foi tido como requisito uma implementação na linguagem C que fosse, ao mesmo tempo, fácil de ser usada, com uma interface intuitiva, via terminal, e ainda assim, customizável pelo usuário via código fonte. Para ser mais prático, a entrada das instruções que serão executadas é feita por leitura de arquivos, esses que deverão ser especificados pelo usuário na hora de compilação do código.

Como o intuito da simulação é demonstrar os passos do algoritmo de Tomasulo, a simulação não possui um banco de registradores com os valores existentes nos mesmos para poder fazer as operações sobre eles, ou seja, a simulação não está simulando um processador real, e sim o algoritmo de Tomasulo.

Nos valores padrões, como visto na Figura 1, o simulador possui 12 unidades funcionais para a execução fora de ordem das instruções: i) 2 unidades funcionais voltadas para o acesso à memória (*LOAD*); ii) 3 unidades funcionais de soma e subtração (*ADD*); iii) 2 unidades funcionais para operações de multiplicação e divisão (*MULT*); iv) 1 unidade funcional para desvio (*BRANCH*); v) 1 unidade funcional voltada para realizar a comparação entre registradores (*COMPARISON*); vi) 3 unidades funcionais para realização de operações lógicas e aritméticas (*LOGICAL*). A quantidade de unidades funcionais e quais instruções elas representam podem ser alteradas no código pelo usuário.

Para instruções de desvio, o padrão da simulação é presumir que sempre será feito o desvio e, no momento da execução, caso seja identificado que o desvio foi tomado

[1]Há uma generalização nos nomes dos registradores para simplificação do projeto.

erroneamente, a simulação voltará para o ponto em que saltou - ponto pós instrução de desvio - e descartará as instruções decodificadas equivocadamente.

Na versão atual do simulador, foi considerada que todas as unidades funcionais gastam apenas 1 ciclo para terminar a execução das instruções.

5. Resultados

Reorder Buffer

Entry	Busy	Instruction	State	Destination	Value
0	No	SRA R2, R5, 2	COMMITED	R2	R5 >> 2
1	Yes	LW R5, 6(R2)	ISSUE	R5	NOT CALCULATED YET
2	Yes	ADD R22, R6, R3	WRITE_RESULT	R22	R6 + R3
3	Yes	MUL R1, R2, R5	ISSUE	R1	NOT CALCULATED YET
4	Yes	BEQ R1, R2, foo #0	ISSUE	-	NOT CALCULATED YET
5	Yes	BNE R1, R5, bar #1	WAITING	-	NOT CALCULATED YET

Reservation Station

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	Address
Load1	Yes	LW	-	R2	-	-	1	6 + Regs[R2]
Load2	No	-	-	-	-	-	-	-
ADD1	No	-	-	-	-	-	-	-
ADD2	No	-	-	-	-	-	-	-
ADD3	No	-	-	-	-	-	-	-
MULT1	Yes	MUL	R2	-	-	1	3	-
MULT2	No	-	-	-	-	-	-	-
BRANCH	Yes	BEQ	-	R2	3	-	4	-
COMPARISON	No	-	-	-	-	-	-	-
LOGICAL1	No	-	-	-	-	-	-	-
LOGICAL2	No	-	-	-	-	-	-	-
LOGICAL3	No	-	-	-	-	-	-	-

Register Status

Field:	R0	R1	R2	R3	R4	R5	R6	R7
Reorder#:	-	-	0	-	-	-	-	-
Busy:	No	No	Yes	No	No	No	No	No
Field:	R8	R9	R10	R11	R12	R13	R14	R15
Reorder#:	-	-	-	-	-	-	-	-
Busy:	No	No	No	No	No	No	No	No
Field:	R16	R17	R18	R19	R20	R21	R22	R23
Reorder#:	-	-	-	-	-	-	-	-
Busy:	No	No	No	No	No	No	No	No
Field:	R24	R25	R26	R27	R28	R29	R30	R31
Reorder#:	-	-	-	-	-	-	-	-
Busy:	No	No	No	No	No	No	No	No

Figura 1. Apresentação da interface do simulador.

Nesta seção será analisada a simulação da execução de um conjunto de instruções RISC-V conforme apresentado na Subseção 3.2. Na Tabela 2 é possível observar um conjunto de instruções de exemplo que são colocadas para executar no simulador desenvolvido. Na Figura 1 é apresentada a saída das tabelas em um determinado instante da execução das instruções. É possível observar o estado *WAITING*, que é quando a instrução está no *buffer* de reordenamento porém não foi enviada para estação de reserva, pois a mesma está ocupada com outra instrução. No exemplo, a instrução *BNE R1, R5, bar #1* está neste estado, pois a estação de reserva da unidade funcional *BRANCH* está ocupada com a instrução *BEQ R1, R2, foo #0*.

É possível visualizar também o estado *ISSUE*, que é quando a instrução está na estação de reserva porém não é despachada para execução pois há dependência verdadeira em alguma instrução anterior conforme a ordem no *buffer* de reordenamento. Conforme

SRA R2, R5, 2	foo:	bar:
LW R5, 6(R2)	BNE R1, R5, bar #1	REM R16, R2, R3
ADD R22, R6, R3	DIV R10, R4, R7	SLT R30, R16, R15
MUL R1, R2, R5	OR R9, R20, R11	SLTI R4, R2, 5
BEQ R1, R2, foo #0	AND R1, R2, R7	
XOR R10, R5, R9		
SUB R2, R2, R3		

Tabela 2. Fluxo de instruções de exemplo que são executadas na Figura 1.

o exemplo da Figura 1, a instrução *MUL R1, R2, R5* está neste estado porque aguarda o valor que será escrito no registrador R5 pela instrução *LW R5, 6(R2)*, é possível verificar também que na estação de reserva na coluna *Qk*, a unidade funcional MULT1 aguarda o valor que virá da entrada 1 do *buffer* de reordenamento. Outro exemplo de instrução que está aguardando o valor de um registrador, ou seja tem dependência RAW, é a instrução *BNE R1, R5, bar #1*, e é possível verificar que a coluna *Qj* está aguardando o resultado da execução da instrução da entrada 3 do *buffer* de reordenamento.

No estado *WRITE_RESULT* a instrução já foi executada e seu resultado é enviado para o *buffer* de reordenamento. Na Figura 1 tanto a instrução *SRA R2, R5, 2*, quanto a *ADD R22, R6, R3* já saíram da tabela da estação de reserva, pois já foram executadas e já tiveram seu resultado enviado para a coluna *Value* da tabela do *buffer* de reordenamento. O simulador não realiza de fato o cálculo das operações das instruções conforme já mencionado, portanto é apenas escrito na coluna *Value* no estado *WRITE_RESULT* do *buffer* de reordenamento o que deveria ser calculado de forma escrita, facilitando a visualização e entendimento do algoritmo de Tomasulo.

E no último estado *COMMITED*, todas as instruções anteriores já tiveram seu *commit*, e seu resultado será escrito em registrador caso tenha algum registrador destino. Como a instrução *SRA R2, R5, 2* não possui nenhuma instrução que chegou antes dela no *buffer* de reordenamento, ela pode ter seu *commit*, e como ela escreve no registrador R2, há a referência da entrada 0 do *buffer* de reordenamento na tabela *Register Status* no registrador R2.

A renomeação do registrador, caracterizada na tabela *Register Status* como a referência à entrada do *buffer* de reordenamento, é realizado no momento do *commit*, pois se for realizada no momento do *WRITE_RESULT* poderá gerar erro. Se o fluxo de instruções for *ADD R2, R1, R5*, *SUB R8, R2, R7* e *MUL R2, R10, R1*, as instruções ADD e MUL serão executadas, pois não possuem dependência de dados e, após a execução, seu estado mudará para *WRITE_RESULT* e a referência colocada no registrador R2 será a da entrada do *buffer* de reordenamento da instrução MUL. Porém, quando a instrução SUB for executada, ela deve pegar a referência da instrução ADD e não da referência MUL, caracterizando um possível erro caso a renomeação de registradores ocorra no estado *WRITE_RESULT*.

Como a previsão de desvio implementada é de que sempre ocorrerá o desvio, conforme o fluxo de instruções na Tabela 2, após a instrução *BEQ R1, R2, foo #0* a próxima instrução no *buffer* de reordenamento é a *BNE R1, R5, bar #1*, ocorrendo o desvio para o *label foo*. Quando a instrução *BEQ R1, R2, foo #0* for executada, as instruções após a mesma no *buffer* de reordenamento serão descartadas, e a instrução *XOR R10, R5, R9* e as demais depois dela serão as próximas no fluxo para entrar no *pipeline* do simulador, caracterizando o descarte de instruções no final do *pipeline*. Caso o desvio deva ocorrer, como é o caso da instrução *BNE R1, R5, bar #1*, não haverá descarte de instruções e o fluxo segue conforme está no *buffer* de reordenamento.

Ao seguir adiante na execução (Tabela 2), a próxima exibição das tabelas demonstrará que a instrução *SRA R2, R5, 2* irá sair do *buffer* de reordenamento e o registrador R2 na tabela *Register Status* terá a referência a entrada 0 do *buffer* de reordenamento removida. Como uma instrução sairá e há mais para serem executadas, entrará a instrução

OR R6, R7, R7 na entrada de número 0 com estado *WAITING*, conforme a previsão de que sempre haverá desvio de instruções para o *label* informado. A instrução *LW R5, 6(R2)* estará em execução, pois não terá mais dependência de dados. As instruções *MUL R1, R2, R5* e B*EQ R1, R2, foo #0* ainda não serão despachadas para execução, pois tem dependência de dados em instruções anteriores, conforme já citado. A instrução *ADD R22, R6, R3* ainda permanecerá no estado *WRITE_RESULT*, pois só pode ter seu *commit* quando a instrução *LW R5, 6(R2)* tiver o seu *commit*. E a instrução *BNE R1, R5, bar #1* ainda permanecerá no estado de *WAITING*, pois a estação de reserva ainda estará ocupada com a instrução BEQ. Fluxograma [2] completo da execução pode ser acessado em no respositório do grupo de pesquisa.

6. Conclusão

Este artigo apresenta um simulador em linguagem de programação C para o algoritmo idealizado e implementado por Robert Tomasulo. São 17 instruções extraídas do conjunto de instruções RISC-V. O simulador realiza a identificação de dependências falsas e verdadeiras, e pode ser útil ao ajudar outros estudantes a compreender melhor como um *pipeline* superescalar funciona com o algoritmo de Tomasulo.

Para trabalhos futuros, busca-se a implementação da contagem de ciclos necessários para execução de um fluxo de instruções e variação na quantidade de ciclos necessários em cada unidade funcional. Além disso, é desejável o desenvolvimento de interface gráfica para melhorar a visualização dos estágios do *pipeline* e melhorar a parametrização do simulador. Outra possível implementação futura seria a construção de um banco de registradores para manipulação de valores. Por fim, uma análise do impacto do simulador no aprendizado de estudantes.

Referências

Reza Azimi. *Tomasulo based MIPS simulator*. Master's thesis, California State University, Northridge, 2013.

John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011.

Dimitris Kehagias and V Douskas-Bertlviser. Android-based simulator to support tomasulo algorithm teaching and learning. *International Journal of Computer Applications*, 170(2):24–29, Jul 2017. ISSN 0975-8887. doi: 10.5120/ijca2017914703.

Wen-jie Liu, Li Shen, and Zhi-ying Wang. A lightweight instruction-set simulator for teaching of dynamic instruction scheduling. In *2016 11th International Conference on Computer Science & Education (ICCSE)*, pages 871–876. IEEE, 2016. doi: 10.1109/ICCSE.2016.7581696.

David Patterson and Andrew Waterman. *Guia prático RISC-V Atlas de uma Arquitetura Aberta Primeira edição*. Strawberry Canyon LLC, 2019.

David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2020.

R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967. doi: 10.1147/rd.111.0025.

Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003. doi: 10.1145/641865.641867.

[2] https://github.com/cart-pucminas/tomasulo/blob/main/Cmulator/Fluxograma/README.md