Processador MIPS Implementado em Simulador Visual para uso na Educação

Christofer Rodrigues
UTFPR Campo Mourão
Campo Mourão-PR, Brasil
0009-0002-8732-3445

Rogério Aparecido Gonçalves
UTFPR Campo Mourão
Campo Mourão-PR, Brasil
0000-0001-7020-6723

João Fabrício Filho
UTFPR Campo Mourão
Campo Mourão-PR, Brasil
0000-0001-6036-4031

***Resumo*—Arquitetura e Organização de Computadores (AOC) é um componente curricular comum em cursos de Computação. Pelas características físicas dos objetos de estudo dessa matéria, não é possível trazermos para sala de aula uma forma de visualizarmos o funcionamento desses objetos como se estivéssemos enxergando as engrenagens de um relógio. A partir da observação de alguns estudos, verificamos que o uso de simuladores contribui para a melhoria dessa situação. Entretanto, normalmente esses softwares simulam o funcionamento da arquitetura mas não mostram como ela funciona ou carecem de formas para interagir com o circuito simulado. Neste trabalho, apresentamos o Projeto Aperture, e nele entregamos implementações incrementais de um processador MIPS, que vão desde o monociclo até um *pipeline* com encaminhamento e *buffer* de desvios, no simulador de circuitos digitais Logisim. Nesse projeto, pode-se executar códigos no processador, visualizar seu funcionamento e modificar o circuito. Com esse trabalho pretendemos colaborar com o ensino de AOC fornecendo um material que foca na visualização e interação do usuário com o circuito simulado.**

***Index Terms*—MIPS, Arquitetura de Computadores, Processador, Pipeline, logisim**

I. Introdução

A disciplina de Arquitetura e Organização de Computadores (AOC) é um componente curricular importante para os cursos de Ciência da Computação, Engenharia da Computação e demais cursos afins. Contudo, alguns conceitos dessa disciplina podem ficar abstratos para alunos de graduação, pois é difícil de visualizar o real funcionamento de objetos de estudo como diferentes implementações de processadores. Entretanto, técnicas de visualização trazem colaborações para o entendimento de conceitos abstratos das ciências [1]. Áreas do conhecimento como Matemática, Teoria dos Grafos e Topologia [2] exploram técnicas de visualização. No ensino, essas técnicas levaram à criação de ferramentas como o Geogebra [3] e o PhET [4], plataforma criada por Carl Wieman, ganhador do nobel de física em 2002, que disponibiliza simulações interativas para auxiliar no ensino de várias áreas da ciência. De forma semelhante, verifica-se que essa situação se replica no ensino de AOC, com a exploração de simuladores visuais colaborando para a motivação e a curiosidade de alunos nessa disciplina [5]. [6] Em AOC, é difícil trazer para uma sala de aula equipamentos que possibilitem a montagem e facilitem a visualização dos mecanismos internos dos objetos de estudo dessa disciplina, como uma *cache*, uma memória RAM, ou até mesmo um processador inteiro. Algumas instituições usam dispositivos como FPGAs, como a UFERSA [6], mas ainda assim, essa situação é diferente de construir um processador como se fosse um quebra-cabeça, além da necessidade de usar parte da carga horária da disciplina para o ensino de VHDL.

Percebendo essa adversidade para o ensino de AOC, alguns pesquisadores desenvolveram ferramentas tentando colaborar para a situação. Um exemplo é o MARS [7], um software usado para simular um processador MIPS, amplamente utilizado e que possui simulador de *cache*. Essa ferramenta pode ser usada em sala de aula e apresenta bons resultados, considerando a melhora no entendimento dos conceitos e no engajamento dos alunos com a disciplina [6]. Em outros casos, ferramentas podem ser desenvolvidas para explorar conteúdos específicos, como o TFSim, que é um simulador visual para auxiliar na compreensão do *Algoritmo de Tomasulo* [8]. Entretanto, essas ferramentas apresentam pouca flexibilidade para o usuário modificar ou visualizar os sinais dos componentes digitais, eles apenas simulam o comportamento do processador com entradas e saídas de dados.

Neste trabalho, apresentamos o projeto Aperture, que visa a implementação de um processador MIPS 32 bits em um simulador de circuitos digitais chamado Logisim [9]. A intenção do projeto é entregar diferentes implementações do processador de forma incremental, seguindo as especificações do livro de David A. Patterson e John L. Hennessy [10], com cada versão aplicando conceitos que trazem melhorias de desempenho, permitindo que em aula seja possível observar o impacto que cada um desses conceitos causa no funcionamento do circuito.

Com base em resultados anteriores [11], disponibilizamos cinco implementações do processador: uma implementação monociclo, desenvolvida com base na descrição fornecida no livro [10] e outras quatro implementações com o conceito de *pipelining*. Além dos circuitos, disponibilizamos um manual explicativo sobre cada implementação do processador, e como usar de forma mais proveitosa as ferramentas disponíveis no Logisim. Todas as implementações atuais e futuras, além dos componentes individuais, estarão disponíveis na *landing page* do projeto[1].

O diferencial deste trabalho é suportado pelas funcionalidades do Logisim, com capacidade de visualização e interação com o circuito sendo observado. Dessa forma, ao executar

[1]https://christoferlv.github.io/ProjetoAperture/

um código MIPS no Aperture, o usuário pode visualizar os sinais de controle sendo ativados e desativados e até mesmo os dados transmitidos pelas conexões dos componentes, isso com uma granularidade de até meio ciclo de *clock*. Além disso, o Logisim permite que o usuário modifique o circuito com grande liberdade, podendo experimentar com a remoção ou adição de novos componentes ao circuito e observar o resultado dessas interações.

Essa flexibilidade permite que diferentes experimentos ou atividades sejam realizadas, como por exemplo a proposta da implementação de algum componente específico para a arquitetura, similar ao mostrado em [12], que explora a elaboração de um preditor de desvio para o MIPS. Ou simplesmente a comparação do funcionamento de diferentes implementações de um mesmo componente, similar ao que foi feito por [13], com o diferencial da visualização da construção do componente no nível de portas lógicas. Um outro experimento interessante é explorar o conceito de reordenação de código e como isso impacta na quantidade de ciclos para a execução de um programa. Esse tipo de atividade pode levar a um melhor entendimento do funcionamento de um processador, como explorado por [14], e incentiva o programador a ter mais cuidado com a escrita de seus códigos, pois estará mais consciente de como ele está sendo executado pelo processador.

II. Revisão da Literatura

Considerando a possibilidade do leitor não estar familiarizado com todos os termos usados no artigo, apresentamos nessa seção conceitos básicos necessários ao entendimento do trabalho.

1) CISC x RISC: Os termos *Complex Instruction Set Computer* (CISC) e *Reduced Instruction Set Computer* (RISC) se referem às características do conjunto de instruções de um processador. Neste trabalho, lidamos com uma arquitetura RISC, mas dada a existência desses dois conceitos na industria, se faz pertinente uma comparação entre eles.

Arquiteturas do tipo CISC possuem um grande número de instruções e baixa ortogonalidade, essas instruções conseguem realizar tarefas "complexas" mas podendo demorar vários ciclos de *clock* para concluírem a execução. Por exemplo, a instrução **ADD** em x86 pode realizar operações diretamente com registradores, ou entre registradores e um operador que ainda está em memória, isso adiciona complexidade na implementação pois deve considerar todas as possibilidades de execução. Além disso, as instruções CISC não possuem um tamanho uniforme, podendo variar entre 1 a 15 *bytes*, tornando a decodificação mais complexa.

Já as arquiteturas do tipo RISC têm um número reduzido de instruções, essas normalmente são altamente otimizadas para uma tarefa, com o objetivo de permitir a execução em um ciclo de *clock*. Por exemplo, a instrução **ADD** do MIPS, uma arquitetura do tipo RISC, pode realizar a soma de apenas dois operadores que estejam em registradores [15]. Além disso, as instruções MIPS possuem tamanhos uniformes, sendo todas de 32 bits, essa uniformidade facilita o processo de decodificação da instrução. No caso do RISC-V, sua base também segue a ideia de ter instruções com tamanho fixo de 32 bits, mas permite extensões com instruções de tamanho variável [16].

O uso de instruções complexas ou reduzidas depende do tipo de aplicação da arquitetura. Uma tendência é de computadores *desktop*, com pouca restrição de consumo energético, usarem um conjunto de instruções CISC. Já dispositivos móveis, que possuem uma forte restrição de consumo energético, **tipicamente** usam conjuntos de instruções RISC. Apesar dessa tendência, o consumo energético de um processador não é intimamente relacionado a sua ISA escolhida, mas depende de outras variáveis relacionadas a sua implementação [17].

Apesar dessa realidade, com os avanços recentes na área da computação, o interesse do mercado para arquiteturas RISC é crescente, como por exemplo o surgimento dos processadores *Apple Silicon* [18], computadores como o *Raspberry Pi* [19] e o surgimento de supercomputadores como o *Fugaku* que usa processadores com arquiteturas RISC [20].

2) Logisim: O Logisim é um software para simulação de circuitos digitais feito para fins educacionais. Desenvolvido originalmente por Carl Burch [9]. Sua principal característica é ser um simulador visual que permite a interação e a construção de circuitos apenas "arrastando e soltando" os componentes, abstraindo muitas complexidades relacionadas a construção de um circuito real, como interferências e roteamento, e também remove a necessidade de aprender a construir circuitos usando linguagens de descrição de *hardware*. Essas características tornam o Logisim um *software* de grande valia para usos educacionais e de pesquisa, por isso, é usado em várias universidade em cursos de arquitetura de computadores ou similares, como nas universidades de Cornell e Holy Cross [21].

Para facilitar o desenvolvimento de circuitos, o Logisim entrega componentes combinacionais e sequenciais prontos para serem utilizados, como portas lógicas, plexers e flip-flops. O *software* também já abstrai o processo de construção de alguns componentes mais complexos, como registradores, componentes para realização de operações aritméticas, como somadores e multiplicadores, e até mesmo entrega memórias RAM e ROM já prontas para serem utilizadas. O Logisim também dispõe de alguns componentes como *displays* de sete segmentos, mas esses não são explorados neste trabalho.

Apesar da boa adoção do *software* por universidades, este trabalho não usa a versão original do Logisim, mas sim uma outra implementação mais atualizada chamada Logisim-Evolution[2]. Essa escolha aconteceu pois Carl Burch suspendeu suas contribuições para o *software* original, e também porque o Logisim-Evolution traz melhorias na aparência, usabilidade, e traz novos recursos.

Além de disponibilizar os componentes para a construção dos circuitos digitais, o Logisim entrega vários recursos que podem ajudar no desenvolvimento e depuração de circuitos. Um deles é uma linha do tempo que mostra as transições de sinais das conexões dos componentes que formam o circuito. As conexões que o usuário deseja inspecionar podem ser

[2]https://github.com/logisim-evolution/logisim-evolution/releases/tag/v3.8.0

selecionadas individualmente, e a linha do tempo ainda pode ser exportada como imagem ou PDF.

Outro recurso é o *assembly viewer*, esse recurso permite relacionar as linhas de um arquivo, podendo ser um código *assembly*, ao valor armazenado em algum registrador da arquitetura. Assim, selecionando um registrador e o arquivo contendo os valores de referência, a ferramenta irá destacar a linha marcada com o valor do registrador selecionado. Isso pode ser usado para verificar se os valores sendo armazenados nos registradores estão sendo o esperado. Um exemplo de uso é marcar todas as linhas de um código que será executado no circuito e usar o Contador de Programa (PC) como registrador de referência, isso permitiria ver qual instrução está sendo executada em determinado momento.

Outra ferramenta útil entregue pelo Logisim é a de análise combinacional. Esse recurso permite ao usuário solicitar que o programa construa um circuito combinacional dada uma expressão lógica, uma tabela verdade, ou então editando um Mapa de Karnaugh. Essa ferramenta é de grande utilidade pois permite acelerar a construção de circuitos combinacionais quando se sabe as entradas e saídas esperadas. Além disso, ela ainda exporta a tabela verdade, o mapa e a expressão mínima em forma de TEX facilitando o uso em documentos e materiais didáticos. Aliás, esse recurso não faz apenas o caminho de tabela verdade para circuito, mas também é capaz de analisar um circuito construído e devolver sua tabela verdade, mapa e expressão mínima.

Uma ferramenta bem útil para o processo de depuração de um circuito é o chamado vetor de teste. Esse recurso disponível no Logisim permite especificar os valores de entrada de cada pino de um circuito, e a saída esperada nos pinos de saída. Com o carregamento do vetor de testes na ferramenta, o Logisim irá simular as entradas dadas e comparar com a saída esperada, e então, mostrará em uma tabela as entradas de cada pino e as saídas reais. Caso alguma saída esteja diferente da esperada, a ferramenta irá mostra-la em destaque.

Outros recursos do Logisim que podem ser comentados são: A janela de estados, que mostra todos os registradores do circuito e quais os seus valores no momento, isso é de grande valia para processos de depuração. E as estatísticas do circuito, que mostra a quantidade total de cada componente usado na construção do circuito, como quantas portas lógicas ou registradores foram usados.

3) Instruções MIPS: Uma característica importante das instruções MIPS é que elas possuem um tamanho regular. No caso do MIPS elas vão ter sempre 32 bits. Essa regularidade é o “princípio de projeto” número um de acordo com o livro de David A. Patterson e John L. Hennessy, descrita como “Simplicidade favorece a regularidade” [10]. Outra característica das instruções MIPS é que seus campos sempre tem os mesmos tamanhos, isso é mostrado na tabela 1, e os campos são explicados na sequência:

- **Opcode**: O código da operação a ser executada. Esse campo tem 6 bits.
- **Rs, Rt e Rd**: Nesses campos são colocados os endereços dos registradores que serão acessados, como o MIPS possui 32 registradores de propósito geral, esses campos precisam ter 5 bits de tamanho.
- **Funct**: Em instruções lógicas e aritméticas, esse campo irá codificar qual operação a unidade lógica e aritmética irá realizar, sendo soma ou subtração, por exemplo. Esse campo possui 6 bits.
- **Shamt**: Esse campo é usado nas instruções de *shift*, como *shift* à direita ou *shift* à esquerda. Ele possui 5 bits.
- **Imediato**: Esse campo tem várias utilidades no conjunto de instruções do MIPS. A primeira delas é nas instruções chamadas “Imediatas”, essas são usadas para somar um valor já especificado a um outro valor que está em um registrador. Esse tipo é utilizado quando queremos usar um valor constante sem a necessidade de ocupar um registrador com ele. Ou então, usamos essas instruções para inicializar um registrador com uma quantia dependendo da ocasião. Um outro uso para esse campo é nas instruções de desvio. Nesse caso, o valor ali colocado será usado para calcular o destino do desvio em relação ao número atual do contador do programa. A outra utilidade desse campo é para as instruções de acesso à memória. Nesse caso, o valor do campo imediato é usado para calcular a posição que será acessada na memória, calculado usando uma quantia que esteja em algum registrador como posição de referência.
- **Endereço**: Este campo é usado para a instrução de desvio incondicional, ele possui 26 bits e codifica o endereço de destino do desvio. Como os endereços possuem 32 bits, esse campo precisa ser completado, primeiro ele é deslocado dois bits para a esquerda, e depois concatenado com os 4 bits mais significativos do PC.

TABELA I
SEPARAÇÕES DE UMA INSTRUÇÃO MIPS

Formato	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Tipo R	opcode	rs	rt	rd	shamt	funct
Tipo I	opcode	rs	rt	Imediato		
Tipo J	opcode	Endereço de Destino				

Essa regularidade do conjunto de instruções MIPS acaba se refletindo em uma simplicidade do seu circuito. A separação dos valores dos campos das instruções podem ser facilmente realizadas apenas utilizando fios, sem nenhuma lógica adicional, já a maneira e se esses valores serão utilizados é ditado pela unidade de controle a partir dos valores em *opcode* e *funct*.

4) Componentes gerais de uma arquitetura MIPS: De acordo com as especificações do livro de David A. Patterson e John L. Hennessy, a arquitetura MIPS possui alguns componentes principais:

- **Contador de Programa (PC)**: Esse componente tem o funcionamento comparável ao de um registrador de 32 bits. O valor lido dele é usado para acessar a memória de instruções e fornecer para a arquitetura qual instrução deve ser executada naquele ciclo de *clock*.

- **Memória de Instruções**: Esse componente é usado para armazenar as instruções que serão executadas pela arquitetura. Ele é um componente apenas de leitura, isso significa que as informações armazenadas nesse componente não são alteradas durante a execução do programa.
- **Banco de Registradores**: Esse componente da arquitetura MIPS possui 32 registradores de propósito geral endereçáveis usando um valor de 5 bits. Ele permite que sejam lidos dois registradores ao mesmo tempo, e escrito em apenas um por vez. Uma característica importante de seu funcionamento é que as leituras são feitas na subida do sinal de *clock* e as escritas na descida, essa característica permite que a escrita e a leitura do banco de registradores aconteçam em um mesmo ciclo de *clock*, possibilitando que instruções lógicas e aritméticas sejam executadas em um único ciclo.
- **Unidade Lógica e Aritmética**: Esse componente da arquitetura MIPS serve para realizar as operações matemáticas da arquitetura, como soma, subtração, cálculo dos endereços de desvio ou de acesso à memória e comparação de igualdade.
- **Memória de Dados**: Esse componente armazena os dados necessários para a execução do programa, ou seja, valores que são recuperados de acordo com a necessidade da arquitetura, esses podem ser modificados utilizando instruções de acesso à memória.
- **Unidade de Controle**: Esse componente é responsável por gerar os sinais de controle da arquitetura a partir dos campos "op" e "funct". Os sinais são usados para indicar aos outros componentes o que eles devem fazer, e para direcionar corretamente o fluxo de dados pela arquitetura.

A. *Representação abstrata de uma arquitetura MIPS*

Sabendo como funcionam os principais componentes que compõem o MIPS, é possível mostrar de forma conceitual como essas partes se conectam. A Figura 1 mostra os principais caminhos que os dados tomam no circuito para poderem realizar a execução de uma instrução.

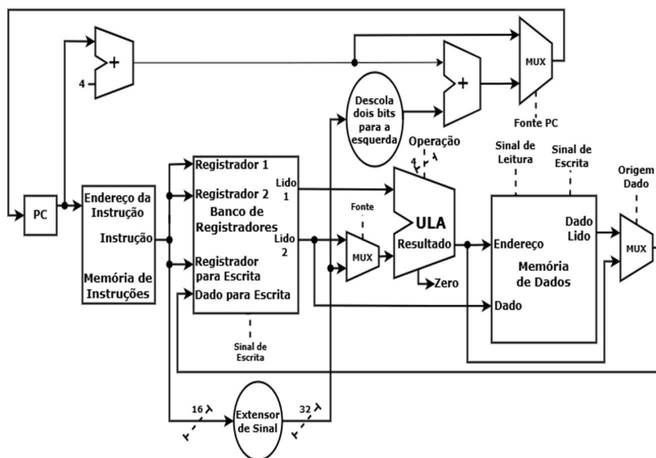

Figura 1. Representação simplificada da arquitetura MIPS. Adaptado de [10].

O movimento no circuito se inicia pelo valor contido em PC, esse é usado para buscar na memória de instruções qual delas será executada. Com a instrução buscada na memória, separamos ela em diferentes partes, usando fios, seguindo a regra mostrada na tabela I. Com essas informações é possível buscar os operadores no banco de registradores e enviá-los para a unidade lógica e aritmética. Com o valor calculado nela, é possível buscar algum dado na memória de dados, caso seja uma instrução de carregamento de dados, ou enviá-lo diretamente para ser salvo no banco de registradores, se for uma operação lógica ou aritmética. Quando outro pulso de *clock* acontece, o valor armazenado no PC incrementa em 4, indicando a próxima instrução. Se uma instrução de desvio for executada, o novo valor em PC será calculado a partir dos campos fornecidos na instrução.

O caminho que de fato é tomado por uma instrução é controlado pelos Multiplexadores (Mux). Esses componentes combinacionais servem para selecionar a fonte de um dado dentre diferentes origens ou então definir seu destino. O sinal desses "muxes" é representado na figura como uma linha pontilhada nomeada. Esses sinais são produzidos na unidade de controle, que está abstraída na figura a fim de simplificá-lá.

B. *Trabalhos Relacionados*

A apresentação de conteúdos de AOC com o auxílio de ferramentas não é uma situação nova, essa abordagem já foi explorada por alguns outros trabalhos. Um desses, que tenta colaborar com o processo de ensino de arquitetura de computadores é o EduMIPS64 [22]. Esse software simula a execução do conjunto de instruções MIPS como um simulador funcional. Em seus recursos ele entrega a simulação de um *pipeline* de cinco estágios com a capacidade de resolver conflitos através do uso de bolhas, além da opção de ativar ou não o encaminhamento (*forwarding*) para resolução de conflitos. O software também entrega várias janelas para observar diferentes aspectos da execução do código, como uma que mostra a posição de uma instrução no *pipeline*, o atual valor de cada registrador, o conteúdo na memória, uma janela com o código que está sendo executado, com a capacidade de colorir cada linha de acordo com o estágio que a instrução se encontra, e uma janela para as estatísticas da execução do código, mostrando valores como o número total de ciclos, instruções executadas, e a quantidade de cada tipo de *stall* ocorrido, que são paradas realizadas pelo processador para resolução de conflitos.

Esse trabalho apresenta um estudo que verifica o impacto do uso do *software* no aprendizado dos alunos na área de arquitetura de computadores. Ele consta que cerca de 90% dos estudantes perceberam algum impacto positivo do uso do *software* no entendimento de conceitos como funcionamento de registradores e *pipelining*. Outra medida apresentada pelos autores é de que, com o uso do EduMIPS64, houve um aumento de 22% no número de estudantes que obtinham sucesso nos exames finais da matéria. Além disso, 70% dos alunos usaram o programa em mais de 40% do seu tempo de estudo.

Apesar desses resultados apresentados, podemos aprender com algumas lacunas deixadas por EduMIPS64 para melhorar o projeto Aperture. A impossibilidade de alterar o circuito é um entrave do projeto, pois não permite incrementos pelo usuário, como implementações para melhoria de desempenho, ou a remoção de módulos já presentes, para verificação de funcionalidades. Além disso, o projeto Aperture permite a inspeção de cada componente do circuito de forma detalhada, podendo ver a lógica por trás do componente, e até mesmo os dados e sinais que são passados nas conexões durante o funcionamento do circuito. Além disso, o artigo mostra que os conceitos com menos impactados pelo uso do EduMIPS64 foram *forwarding* e *hazards*, conceitos esses que podem ter o entendimento significativamente melhorado se fosse possível ver quais sinais são ativados nos momentos em que *hazards* são detectados, funcionalidade disponível no projeto Aperture.

RIPES [23] também aborda a visualização do funcionamento de uma arquitetura de computador. Esse software é um simulador funcional e possui ainda mais recursos que o EduMIPS, como mostrar o circuito com um pouco mais de detalhes e informações, e é feito para a ISA RISC-V. Ele disponibiliza 6 versões da arquitetura que podem ser selecionadas no simulador: Uma implementação monociclo, e quatro com *pipeline* de cinco estágios, uma sem nenhuma resolução de conflitos, uma com resolução de conflitos e sem *forwarding*, uma apenas com *forwarding*, e outra com ambos os recursos, e também uma implementação de 6 estágios superescalar com escalonamento estático.

Além de oferecer a possibilidade de executar código em cada uma dessas implementações, o software RIPES fornece vários outros recursos de usabilidade. Um deles é o destaque das linhas de sinais que estão ativas e a possibilidade de inspecionar seus valores. Também possui janelas para visualização do conteúdo dos registradores, o conteúdo das memórias de dados e de instruções, e uma aba para as estatísticas de execução, com número de ciclos, instruções executadas, ciclos por instrução e instruções por ciclo.

Além de todos esses recursos, o RIPES também apresenta uma janela para o simulação da *cache* da arquitetura, mostrando o conteúdo das *caches* de dados e de instruções, e permitindo a modificação de seus parâmetros de funcionamento, como número de linhas, número de palavras por linha, política de substituição, e as políticas de escrita em caso de acerto ou erro. Além disso, o software apresenta várias métricas do funcionamento da *cache* durante a execução do código. Apesar dos simuladores MARS e RIPES apresentarem simulação de *cache*, esse é um recurso que o projeto Aperture não explora nesse trabalho.

A principal diferença do RIPES para o Aperture é que no nosso projeto o usuário tem total liberdade para modificar o circuito da arquitetura como desejar, abrindo mais possibilidades para uso e extensão das arquiteturas entregues. Além disso, o RIPES não permite a inspeção dos componentes da arquitetura, como bancos de registradores e unidade lógica e aritmética, apenas mostra como eles se conectam no processador final, essa complexidade é abstraída pelo software diminuindo a possibilidade de entendimento do usuário. Outra característica é que o RIPES permite o avanço de apenas um ciclo de *clock* inteiro, enquanto a execução feita no Logisim permite avançar a execução em meio ciclo de *clock*, diferença que pode permitir ver transições importantes de sinais, principalmente os de gravação de registradores e resolução de conflitos.

III. Proposta

Observando os resultados positivos descritos em [5], [6] e [22], que constatam um melhor rendimento e um aumento na curiosidade e motivação dos alunos pelos estudos em AOC, há impacto positivo nos alunos de graduação com o uso de simuladores nas aulas. MARS e EduMIPS são ferramentas que podem fornecem suporte à interatividade, contudo, esses simuladores não mostram o circuito ou mostram uma visualização limitada dele sem a possibilidade de interação. Não sendo possível a modificação ou inspeção dos sinais durante a execução.

Dessa forma, este trabalho apresenta uma opção com ênfase na visualização e modularidade do circuito simulado. Assim, o usuário vê os efeitos das técnicas como *forwarding* ou previsão de desvio, e também observa quais sinais essas implementações consomem e produzem para fazerem seus papéis. A flexibilidade do projeto abre possibilidades para professores como propor que os alunos implementem componentes da arquitetura e verificarem o funcionamento, tendo como base um processador já pronto.

Para esse trabalho, foram feitas 5 implementações incrementais do processador MIPS seguindo as especificações de Patterson e Hennessy [10]:

1) Arquitetura de ciclo único;
2) Pipelining com cinco estágios;
3) Pipelining com cinco estágio, resolução de conflitos com bolha e predição de desvios estática;
4) 3 com o acréscimo de *forwarding*;
5) 4 com o acréscimo do Jump Target Buffer (JTB);

As próximas subseções detalham essas implementações.

A. Arquitetura de Ciclo Único

A arquitetura de ciclo único é de fácil entendimento por ser uma implementação simples, com menos conexões e componentes. Apesar de não ser amplamente utilizada em processadores reais, essa é apropriada para o entendimento de conceitos, comportamento e características básicas de uma arquitetura MIPS. Embora simples, essa versão da arquitetura é capaz de executar qualquer código na linguagem de máquina do MIPS. O circuito final dessa arquitetura é a concretização do diagrama mostrado na Figura 1 com a adição dos sinais de controle, podendo ser vista na Figura 2.

A característica principal dessa arquitetura é que todas as instruções são executadas em um ciclo de *clock*. Todo o processo de busca de instruções na memória, recuperação dos operadores no banco de registradores, execução da operação na ULA e a gravação do resultado, seja em memória ou no banco de registradores, acontece de uma vez. Isso traz benefícios

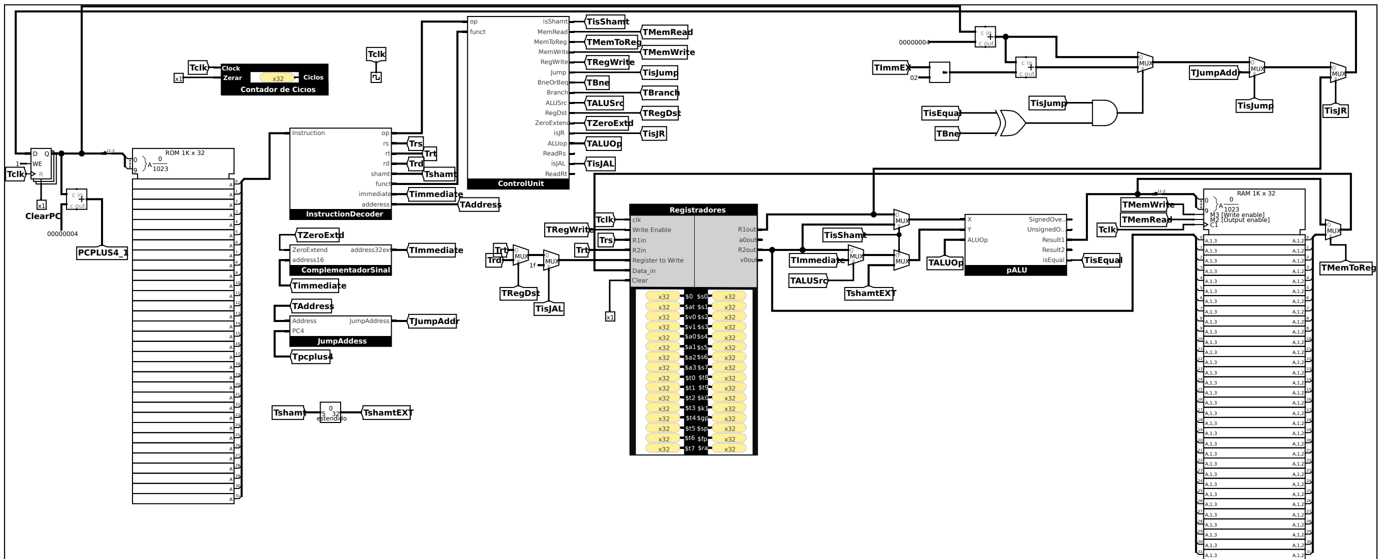

Figura 2. Arquitetura de ciclo único.

como não ser necessário se preocupar com dependências de dados que ocorram na execução de um código, visto que uma instrução sempre é concluída e tem seu resultado corretamente armazenado antes que a próxima comece a ser executada. O entendimento desses conceitos é potencializado com a visualização e inspeção dos circuitos suportadas pelo Logisim.

Apesar da simplicidade, essa variante não possui um desempenho satisfatório comparado com outras implementações. Isso acontece pois o período de *clock* é longo, pois é ajustado para conter a instrução mais lenta. E como todas as instruções são executadas em um ciclo, mesmo instruções mais simples precisam durar o tempo da instrução mais lenta.

Assim, outras implementações são preferidas quando o MIPS é usado em aplicações reais, como aquelas que implementam o conceito de *pipelininig*.

B. Pipelining

A técnica de *pipelining* melhora o desempenho da arquitetura por meio da maior vazão de instruções no processador. Seguindo a especificação de Patterson e Hennessy [10], a implementação do MIPS com *pipeline* feita neste trabalho tem 5 estágios. Com o uso dessa técnica, nosso processador tem o potencial de executar até cinco instruções simultaneamente. Os cinco estágios de *pipeline* que dividem as unidades funcionais do processador são:

- **Busca de Instruções**: Nele estarão presentes o PC e a Memória de Instruções. Há também circuitos relacionados ao controle dos desvios da arquitetura, gerenciando qual a próxima instrução apontada pelo PC. Nesse estágio também estará presente a JTB na implementação que a utiliza.
- **Decodificação**: Os principais componentes deste estágio são o Decodificador de Instruções, a Unidade de Controle, e o Banco de Registradores. Apesar do livro não apresentar uma Unidade de Decodificação, entendemos que modularizar o conjunto de ligações que separa cada parte da instrução colabora com o entendimento da arquitetura. Nesse estágio, a Unidade de Controle produz todos os sinais e são gerados todos os dados necessários para sinalizar aos estágios seguintes as funções a serem ativadas e o caminho tomado pelos dados. Por exemplo, qual operação a ULA irá executar, e se um resultado será gravado em memória ou em um registrador.
- **Execução**: O principal componente deste estágio é a ULA, ela utiliza os sinais produzidos anteriormente para realizar a operação solicitada pela instrução usando os operadores buscados no banco de registradores. Aqui, também encontra-se a lógica de desvio condicional, já que a ULA é necessária para a realização da comparação de igualdade usada para tomar a decisão do desvio. Até a execução, nenhum estágio causa alteração de estado na arquitetura, assim nenhum dado é alterado, apenas consumido. Tipicamente as gravações acontecem nos estágios posteriores, salvo nos casos de desvio que causam alterações no PC.
- **Memória**: A única unidade funcional presente neste estágio é a memória RAM. Aqui também é encontrada alguma lógica necessária no caso da presença de *forwarding* na arquitetura.
- **Escrita**: Este é o estágio final, que contém componentes de lógica combinacional que gerenciam a gravação no banco de registradores principal do resultado calculado pela ULA ou de algum valor recuperado na RAM.

Apesar do aumento de complexidade do circuito, o uso de *pipelining* traz ganhos de desempenho com o aumento da vazão de instruções, podendo ter até cinco instruções executando ao mesmo tempo, comparado com a implementação de ciclo único, que só pode ter uma instrução executando por vez a Figura 3 mostra como duas instruções são executadas no *pipeline* e na arquitetura de ciclo único, mostrando

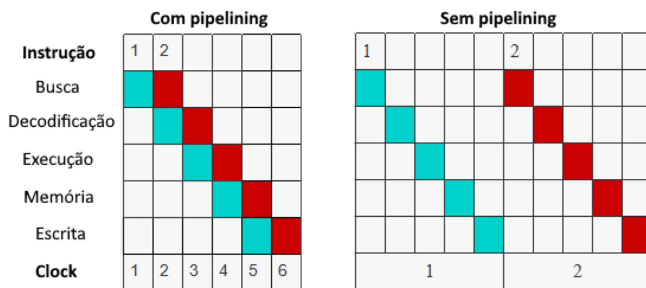

Figura 3. Comparação da execução no pipeline e no ciclo único.

a sobreposição que ocorre no *pipeline*. Além da vazão, o *pipelining* também diminui o período de *clock* em até cinco vezes, pois não é necessário restringir o período pela execução de uma instrução completa, apenas pelo tempo do estágio mais lento. É importante lembrar que esse ganho potencial se concretizaria caso todas as etapas durassem a mesma quantidade de tempo.

Seguindo o exemplo descrito no livro [10], e supondo alguns valores de tempo necessário para cada etapa realizar seu trabalho, sendo eles: 200ps para Busca de Instruções, 100ps para Decodificação, 200ps para Execução, 200ps para Memória, e 100ps para Escrita, a nossa instrução mais longa na arquitetura de ciclo único seria de 800ps, já na arquitetura com *pipeline*, considerando que o período de *clock* precisa respeitar o estágio mais lento, a execução completa de uma instrução levaria 5x200ps que são 1000ps.

Olhando apenas para esses números, parece que o ganho de desempenho não aconteceu. Mas o *pipeline* ganha na vazão total de instruções. Nessa situação, temos que cada acréscimo de instrução soma apenas 200ps ao tempo total no *pipeline*, e na arquitetura de ciclo único acrescenta completos 800ps.

Uma conta simples que checa a razão de melhoria de desempenho é: Considerado 1.000.000 de instruções, teríamos aproximadamente um tempo de execução de 200.000.000ps para o *pipeline* e 800.000.000ps para o ciclo único, fazendo uma divisão entre esses dois tempos, mostramos que o ganho de desempenho é aproximadamente 4 vezes. O ganho potencial de cinco vezes não aconteceu pois os estágios levam tempos diferentes para executarem suas tarefas, ainda assim, uma melhora de 4 vezes é um ganho significativo.

Por conta da separação de funções, o controle de uma arquitetura que usa *pipelining* se torna mais complexo, boa parte dos sinais produzidos pela unidade de controle são usados para gerenciar toda essa lógica adicional. Além disso, como cada estágio é independente, precisamos armazenar os dados gerados por cada unidade funcional em bancos de registradores intermediários que separam os estágios. Cada estágio busca os dados que serão usados por ele em um banco de registradores e grava os seus resultados no seguinte.

Um problema que se origina pelo uso de pipeline são os efeitos colaterais chamados *hazards*. Esses são causados por dependências de dados ou efeitos colaterais do funcionamento de algumas instruções. A resolução de *hazards* demanda controle adicional, o que aumenta a complexidade do circuito mas mantemos o ganho de desempenho.

C. Hazards

Hazards são conflitos entre a próxima instrução que será buscada com outras presentes no pipeline. Assim essa teria um resultado incorreto produzido se o conflito não for resolvido. Nesta seção, abordamos dois tipos de *hazards* que ocorrem no pipeline de cinco estágios implementado.

1) Hazards de Dados: *Hazards* de dados surgem por conta de dependência dos dados gerados e lidos por diferentes instruções [10]. Isso é bem intuitivo de entender, pois os programas são cheios de dependências, sempre computamos um dado para que possamos usá-lo em algum momento depois para computar algum outro dado, gerando uma grande cadeia de dependências.

A causa dos *hazards* de dados é a escrita do resultado somente no último estágio do pipeline. Se a próxima instrução precisar desse resultado, deverá esperar a instrução que o gera escrevê-lo no estágio de escrita. A arquitetura de ciclo único não possui esse conflito, pois toda instrução grava seu resultado antes do início da próxima instrução. Entretanto, no pipeline, as instruções podem demorar 4 ou 5 ciclos para gravarem seus resultados, e nesse tempo, os operadores de outras instruções poderiam ser buscados.

Os tipos de *hazards* de dados possíveis são:

- **Read after write (RAW)**: Ocorre quando uma instrução precisa ler dados gerados por outra instrução que ainda não gravou seus resultados. Esse é o único *hazard* de dados que pode acontecer no *pipeline* de cinco estágios do MIPS.
- **Write after read (WAR)**: Ocorre quando uma instrução posterior faz uma gravação sobrescrevendo um dado que uma instrução anterior, mas que ainda está no *pileline*, precisa ler. Esse problema não ocorre no *pipeline* do MIPS pois a gravação na memória de dados ou nos registradores ocorre nos estágios de acesso à memória e de escrita, que são os dois últimos.
- **Write after write (WAW)**: Ocorre quando uma instrução posterior faz uma gravação antes de sua antecessora. A consequência disso é que a instrução sobrescreve o resultado gravado pela instrução posterior. Isso não ocorre no *pipeline* do MIPS pelo mesmo motivo do *hazard* WAR.

Como explicado, RAW é o único tipo de *hazard* de dados que precisamos lidar. Se esse problema não for tratado corretamente, nosso processador poderá executar instruções com dados inexatos e gerar resultados incorretos.

2) Hazards de Controle: O comportamento típico da busca de instruções é acessar na memória de instruções o endereço armazenado no PC, e somar 4 ao valor armazenado no PC, para buscar a próxima instrução no ciclo de *clock* seguinte. O endereço das instruções é incrementado em 4 a cada ciclo de *clock*, pois cada instrução possui 4 *bytes*.

Instruções de desvio alteram o comportamento típico do estágio de busca de instruções. Instruções de desvio condi-

cional como `beq` e `bne` podem alterar o valor de PC para uma posição indicada pela instrução, e que depende do resultado de uma comparação entre registradores.

Como o resultado da comparação é gerado no estágio de execução, o valor de PC para o próximo ciclo de *clock* estará disponível somente depois de transcorridos 3 estágios. Dessa forma, existe um conflito, chamado de *hazard* de controle, no qual o *pipeline* precisa de 2 ciclos de *clock* adicionais para saber qual o novo endereço de instrução que será buscado e decodificado.

D. *Solucionando os* Hazards

Abordamos nesta seção quatro técnicas de solucionar *hazards*, que também podem contribuir com a melhoria de desempenho para a arquitetura: (1) bolha, (2) *forwarding*, (3) predição de desvios e (4) *Jump Target Buffer*. A aplicação e eficiência de cada técnica depende do tipo de *hazard* e dos efeitos colaterais.

1) Bolha: A forma mais intuitiva de resolver um *hazard* é atrasando a busca da instrução seguinte em ciclos suficientes para que a instrução que causa os conflitos possa terminar sua execução. Isso é feito por uma instrução que não causa alteração de estado no processador e que apenas ocupa um ciclo de *clock*, essa é chamada de bolha. Assim, quando detectarmos uma dependência de dados, ou passarmos por uma instrução de desvio, basta adicionar bolhas suficientes no *pipeline* até que os dados sejam corretamente gravados, ou a decisão do desvio seja calculada. No pior caso, se adicionarmos 4 bolhas após uma instrução, haveria uma única instrução não-bolha no processador, o que anularia qualquer conflito. Assim, bolhas são capazes de resolver todos os *hazards* de *pipeline*.

O problema dessa abordagem é que estamos desperdiçando ciclos de *clock* que poderíamos estar executando operações úteis. Dependendo da característica do algoritmo que está sendo executado, o uso dessa abordagem pode até mesmo negar os benefícios de usarmos a técnica de *pipelining*.

2) Forwarding: Uma alternativa para resolver os *hazards* de dados e manter o desempenho é com o encaminhamento de dados ainda não gravados, mas que já foram gerados. Assim, em vez de aguardar o resultado ser gravado, o circuito encaminha o dado gerado para a instrução que possui dependências com aquele resultado. O *forwarding* tem grande impacto, em instruções lógicas e aritméticas, que gravam seu resultado somente no estágio de escrita, mas o valor já está pronto no estágio de execução.

Quando usamos apenas bolha, é necessário um atraso de dois ciclos de *clock* entre duas instruções que tenham dependência de dados. Ao utilizarmos o *forwarding*, não precisamos mais de nenhuma espera em caso de dependência de dados do resultado de uma instrução lógica ou aritmética, e apenas um ciclo de espera para a instrução de carregar dados da memória, pois a lógica do *forwarding* é capaz de encaminhar os dados para a instrução esteja precisando.

É importante lembrar que *forwarding* apenas encaminha o resultado de uma instrução no *pipeline* de um estágio origem para ser usado em outro. A instrução que produz o dado ainda precisa completar o caminho pelo *pipeline* para ter seu resultado corretamente salvo no banco de registradores principal ou na RAM.

3) Resolvendo Hazards de Controle: A bolha é a forma mais simples de lidar com *hazards* de controle. Nesse caso, podemos colocar três bolhas no *pipeline*, pois precisamos esperar o resultado ser produzido no terceiro estágio, que é a comparação feita na ULA, porque só após esse processamento vamos saber se o desvio condicional será tomado ou não. Novamente, esse método é pouco eficiente, pois estamos perdendo ciclos de *clock* sem realizar processamento útil.

Com alguma lógica de comparação adicional, o estágio dois, de decodificação, pode gerar o resultado do desvio. Assim, no segundo estágio já temos todas as informações necessárias para verificar se o desvio é tomado ou não, sendo necessário adicionar somente uma bolha.

Uma estratégia ainda melhor é usamos uma técnica chamada predição de desvios. Nesse caso, o desvio é previsto como tomado ou não, e buscamos próxima instrução de acordo com o previsto. Quando a comparação do desvio estivar pronta no terceiro estágio, verificamos se ele está de acordo com a previsão. Em caso positivo, seguimos com a execução e não foram necessárias nenhuma bolha. Em caso negativo, limpamos os estágios 1 e 2 e continuamos o processamento a partir do alvo correto do desvio. O ganho dessa técnica é que **apenas** no caso de errarmos a previsão, a penalidade no desempenho será igual a de usarmos apenas a bolha. Essa é a técnica atualmente utilizada na nossa arquitetura.

4) Acelerando o Desvio Incondicional: Desvios incondicionais transferem o controle para um novo endereço especificado na instrução. No estágio de decodificação o endereço é identificado e somente depois desse estágio é possível transferir o controle, o que atrasa em um ciclo a busca da próxima instrução. Contudo, se a instrução de desvio já foi executada antes, podemos saber o endereço de destino previamente.

A técnica de Jump Target Buffer (JTB) armazena os endereços da instrução de desvio e da transferência de controle em uma pequena memória. A partir dessa técnica, conseguimos acelerar a execução dessa instrução, podendo executá-la com "zero ciclos". Essa façanha é realizada a partir do conhecimento que o desvio incondicional sempre levará para o mesmo destino, dessa forma, conseguimos construir no estágio 1 um componente que representa o funcionamento de uma tabela indexado pelo valor do PC que tem como valor o destino do desvio.

O componente funciona armazenando o endereço de destino da instrução de desvio incondicional em um registrador indexado pelo valor do PC, após a primeira vez que ela é executada. Assim, não é mais necessário calcular qual é o endereço de destino dessa instrução em uma próxima execução. Sempre uma nova instrução é buscada, o JTB verifica se existe uma entrada salva, e, em caso positivo, é passada o endereço de destino do desvio para a memória de instruções em vez de recalcular o endereço do desvio. Dessa forma, "executamos" o desvio em zero ciclos. O termo *Jump Target Buffer* foi usado

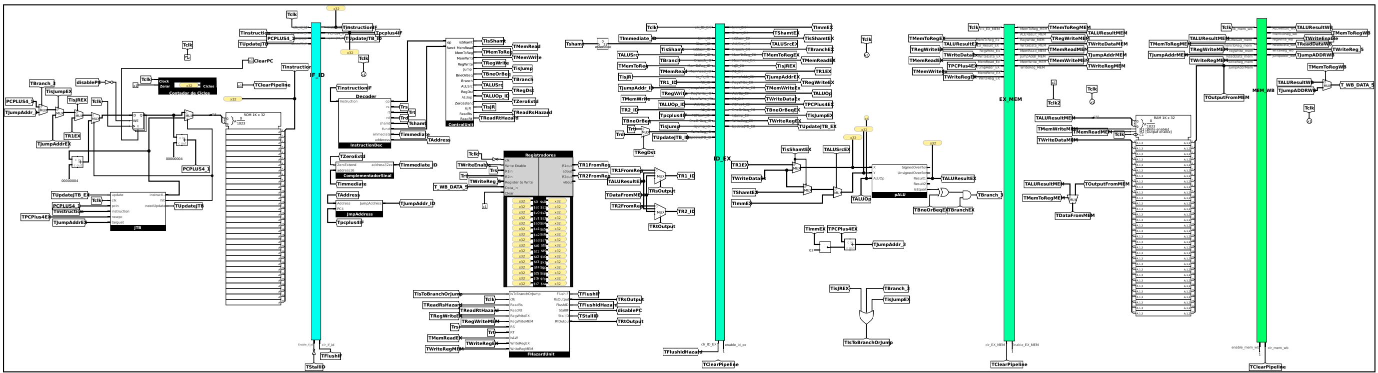

Figura 4. Implementação da Arquitetura MIPS com pipeline de cinco estágios, resolução automática de conflitos, predição de desvios estática e JTB.

neste texto pois, até a publicação deste trabalho, essa unidade indexa apenas instruções de desvios incondicionais.

IV. Resultados

A partir da implementação incremental dos conceitos apresentados, é possível chegar em um circuito final com os seguintes recursos: *Pipelining* de 5 estágios, resolução automática de conflitos com bolha, predição de desvios estática considerando desvio não tomado, *forwarding* e JTB. Esse circuito montado no Logisim fica com o aspecto mostrado na Figura 4.

A implementação deste trabalho é disponibiliza na página do projeto Aperture, que também contém códigos de teste para cada uma das unidades funcionais. Assim, há material complementar para facilitar o uso em aulas ou individualmente por alguém que apenas esteja tentando entender o funcionamento de cada componente.

Uma atividade possível de se realizar é a comparação da performance de cada arquitetura ao executar um programa. Para exemplificar esse caso neste trabalho, foi feito um programa que soma os elementos de um vetor. Esse código foi escolhido por ser simples e explorar todas as capacidades da arquitetura, como operações aritméticas, busca em memória, desvios condicionais e não condicionais. Os resultados desse experimento podem ser vistos na Figura 5.

O impacto de cada recurso no desempenho geral é notório, com a última implementação gastando menos ciclos que a arquitetura monociclo. Apesar disso, pode parecer que apenas essa última teve melhor desempenho que a monociclo. Contudo, o período de *clock* dessa arquitetura é longo, e executa apenas uma instrução por vez, enquanto as arquiteturas que implementam *pipelining* possuem uma vazão maior.

Se mantivermos as suposições de valores do exemplo mostrado na seção III-B, poderíamos calcular que arquitetura monociclo levou 800ps*64 para finalizar a execução do código, totalizando 51200ps. E para as implementações com *pipelining*, se considerarmos o pior caso de que todas as instruções executadas passaram por todos os estágios, teríamos os valores mostrados na Figura 6, dessa forma ficando bem mais evidente o ganho de performance em cada arquitetura.

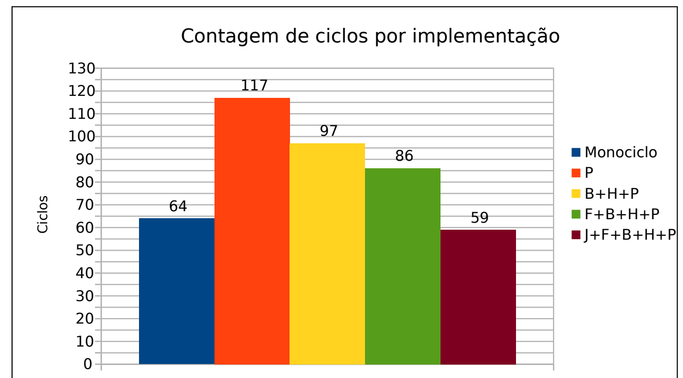

Figura 5. Número de ciclos da execução de uma soma em vetor em cada implementação em que: P - *pipelining*, B - previsão de desvios estática, H - resolução automática de conflitos, F - *forwarding* e J - JTB.

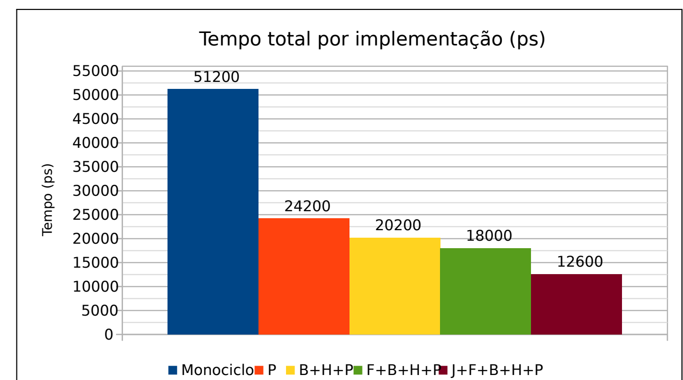

Figura 6. Tempo de execução da soma em vetor para cada implementação da arquitetura, considerando os mesmos valores de III-B.

Apesar desses resultados, é importante lembrar que ele se aplica para esse caso, com um programa com um comportamento específico. Outros algoritmos com diferentes características podem se aproveitar melhor ou não se aproveitar completamente das estruturas implementadas na arquitetura. Por exemplo, um algoritmo com poucos desvios incondicionais não se aproveitaria tanto do JTB. Essa situação pode ser explorada em aula pedindo que os alunos testem

códigos com diferentes características e então realizarem uma atividade descrevendo ou analisando como esses códigos se aproveitaram de cada unidade funcional.

V. CONCLUSÃO

Este trabalho apresentou a implementação incremental de uma arquitetura baseadas em MIPS, em um simulador visual. Essas implementações e especificações dão suporte ao ensino e pesquisa sobre conceitos relacionados à arquitetura e organização de computadores, para professores, estudantes e entusiastas. O projeto pode ser usado por esses estudantes ou pesquisadores que possuem intenções de aprender mais sobre a área ou então testar a implementação de algum unidade funcional em um processador MIPS, como uma unidade de exceção ou uma unidade de ponto flutuante.

Como trabalhos futuros, visamos a implementação de outras versões da técnica de previsão de desvios, como um previsor de desvio estático que tem um comportamento diferente caso o desvio seja para frente ou para trás, e a implementação de uma técnica de previsão de desvio dinâmica.

Além disso, técnicas de paralelismo a nível de instrução, como VLIW e superescalar, podem ser implementadas para estudos avançados. Também, foram iniciadas pesquisas relacionadas a migração da arquitetura para a ISA RISC-V visto o recente interesse do mercado nesse padrão. Também esperamos explorar uma ferramenta do Logisim que permite exportar o circuito para VHDL, contudo, são necessários testes para validar a aplicação desse recurso.

As implementações descritas estão disponíveis publicamente na página do projeto[3] para o impulsionamento de sua utilização. Esperamos colaborar com o ensino e a pesquisa na área de Arquitetura e Organização de Computadores, fornecendo um material com ênfase na visualização e interação com os circuitos de um processador de computador.

AGRADECIMENTOS

Os autores agradecem à PROREC/UTFPR pelo apoio financeiro (Edital 03/2024).

REFERÊNCIAS

[1] Henk W. de Regt. ”Visualization as a Tool for Understanding,” in Perspectives on Science, vol. 22, no. 3, pp. 377-396, 2014.

[2] R. Tennant. ”Visualizing Mathematics: Imagery Techniques for Learning Abstract Concepts,” in Math Pad Online Journal of the MiPad Research Group, vol. 13, 2006.

[3] N. Dahal, B. P. Pant, I. M. . Shrestha, and N. K. Manandhar, “Use of GeoGebra in Teaching and Learning Geometric Transformation in School Mathematics”, Int. J. Interact. Mob. Technol., vol. 16, no. 08, pp. pp. 65–78, Apr. 2022.

[4] “PhET Interactive Simulations.” PhET. https://phet.colorado.edu/ (accessed Aug. 12, 2024).

[5] B. Ferreira and C. Martins. ” Arduino virtual no Tinkercad Circuits como motivação ao aprendizado prévio de Arquitetura de Computadores”, in Anais Estendidos do XXI Simpósio em Sistemas Computacionais de Alto Desempenho, Evento Online, 2020.

[6] S. Fernandes, I. Silva. ”Relato de experiência interdisciplinar usando MIPS,” in International Journal of Computer Architecture Education (IJCAE), vol. 6, no. 1, pp. 52–61, 2017.

[7] K. Vollmar, P. Sanderson. ”MARS: an education-oriented MIPS assembly language simulator,” in SIGCSE Bull., vol. 38, no. 1, pp. 239–243, 2006.

[8] L. Reis, L. Duenha. ”TFSim: um simulador do algoritmo de Tomasulo para apoio ao ensinode arquiteturas superescalares,” in International Journal of Computer Architecture Education (IJCAE), vol. 8, no. 1, pp. 17–27, 2019.

[9] C. Burch. ”Logisim: a graphical system for logic circuit design and simulation,” in J. Educ. Resour. Comput., vol. 2, no. 1, pp. 5–16, 2002.

[10] D. Patterson, J. Hennessy, Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. Morgan Kaufmann Publishers Inc., 2013.

[11] C. Rodrigues, R. Gonçalves, J. Fabrício Filho, ” Implementações de Processadores MIPS em Simulador Visual,” in Anais da XV Escola Regional de Alto Desempenho de São Paulo, 2024, pp. 57–60.

[12] H. Baranda, J. Penha, R. Ferreira. ”Implementacão de um Preditor de Desvio no MIPS 5 Estágios,” in International Journal of Computer Architecture Education (IJCAE), vol. 6, no. 1, pp. 18–26, 2017.

[13] Duenha, L., et al. ”Avaliação de preditores de desvios por meio de simuladores como parte do processo de ensino eaprendizagem de Arquitetura de Computadores,” in International Journal of Computer Architecture Education (IJCAE), vol. 6, no. 1, pp. 1–9, 2017.

[14] C. Koliver, C. Meinhardt, M. Silva. ”Por um Ensino de Arquitetura de Computadores para Cursos de Sistemas de Informação,” in International Journal of Computer Architecture Education (IJCAE), vol. 11, no. 1, pp. 1–9, 2022.

[15] John Hennessy, ”MIPS reference card,” , 1996.

[16] Waterman, A., et al, ”The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0,” EECS Department, University of California, Berkeley, Tech. Rep., 2014.

[17] E. Blem, J. Menon, and K. Sankaralingam, ‘Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures’, in 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), 2013, pp. 1–12.

[18] P. Higa, ”O que é Apple Silicon? Saiba quais são os modelos de chips da Apple.” tecnoblog.net. https://tecnoblog.net/responde/o-que-e-apple-silicon-processadores (accessed Aug. 1, 2024).

[19] ”Raspberry Pi 5 é anunciado com primeiro processador próprio da marca.” tudocelular.com. https://www.tudocelular.com/tech/noticias/n212234/raspberry-pi-5-e-anunciado-com-primeiro-processador-proprio-da-marca.html (accessed Aug. 1, 2024).

[20] I. Cutress, ”New #1 Supercomputer: Fugaku in Japan, with A64FX, take Arm to the Top with 415 PetaFLOPs.” anandtech.com. https://www.anandtech.com/show/15869/new-1-supercomputer-fujitsus-fugaku-and-a64fx-take-arm-to-the-top-with-415-petaflops (accessed Aug. 1, 2024).

[21] C. Burch, ”Schools using Logisim.” cburch.com. http://www.cburch.com/logisim/usage.html (accessed Aug. 12, 2024).

[22] Patti, D., et al. ”Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator,” in IEEE Transactions on Education, vol. 55, no. 3, pp. 406-411, 2012.

[23] M. Petersen, ”Ripes: A Visual Computer Architecture Simulator,” in 2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE), 2021, pp. 1-8.

[3] https://christoferlv.github.io/ProjetoAperture/