Disclaimer: Esse manual é um material complementar fornecido com a arquitetura Aperture e estará em constante modificação. Visite a página do projeto no github quando possível e verifique se a versão do manual que você possúi é a mais recente. Modificações sobre o processo de implementação serão recorrentes com o texto sendo continualmente atualizado e aprimorado de acordo com o atual estágio da implementação da arquitetura.
`https://github.com/ChristoferLv/Aperture-Releases`

`Versão: 0.1`

Manual para Implementação de uma Arquitetura MIPS com Pipeline de 5 Estágios

Christofer Daniel[1], **João Fabrício Filho**[1]

[1]Universidade Técnológica Federal do Paraná (UTFPR)

Abstract. *The purpose of this paper is to discuss the development of a MIPS-style data path with a five-stage pipeline, the intention being to enlighten the reader with digital logic concepts that cover the workings of a computer processor, with the ultimate aim of getting the reader interested in the field of computing. The way in which this is achieved is by explaining the assembly of the five-stage MIPS in the Logisim visual digital circuit simulator. This has resulted in a manual that explains the operation of all the basic components for assembling a five-stage data path, explaining their operation and input and output signals in such a way that a reader can use the material to help with a data path implementation.*

Resumo. *A intenção desse trabalho de discorrer sobre o desenvolvimento de um caminho de dados estilo MIPS com um pipeline de cinco estágios, a intenção disso é elucidar o leitor com conceitos de lógica digital que abrangem o funcionamento de um processador de computador, com o propósito final de causar um interesse no leitor sobre a área da computação. A maneira com que isso é atingido é explicando a montagem do MIPS de cinco estágios no simulador visual de circuitos digitais Logisim. Com isso, foi alcançado um manual que contém a explicação do funcionamento de todos os componentes básicos para a montagem de um caminho de dados de cinco estágios, explicando seus funcionamentos e sinais de entrada e saída de forma que um leitos pode usar o material para auxiliar em uma implementação de um caminho de dados.*

1. Introdução

Este manual é um artefato para auxiliar no processo de ensino de arquitetura de computadores, especificamente o conceito de caminho de dados e pipeline de múltiplos estágios que é considerado complexo e muitas vezes não explorado de forma adequada em sala de aula.

O objetivo do material é elucidar o leitor com os conceitos necessários para a implementação de um caminho de dados ou de um pipeline básico da arquitetura MIPS. A maneira com que isso é feito com a construção de um processador de 5 estágios em um simulador visual de circuitos digitais conhecido como Logisim.

Com este trabalho espera-se descrever o processo de montagem de um caminho de dados com um pipeline de 5 estágios no Logisim capaz de executar algumas das instruções mais essenciais descritas no MIPS ref card [Patterson and Hennessy 1984], mostrando a construção das unidades funcionais para montagem de caminho de dados, como o Arquivo de Registradores, Unidade de Controle, os Bancos de Registradores Intermediários e a Unidade de Controle de Hazards.

Com esse processo, é espera-se causar um interesse no leitor sobre o funcionamento do computador e o campo da arquitetura de computadores, com a intenção de capturar seu interesse pela área, pendendo leva-lo a produzir pesquisa ou artefatos que ajudem na evolução do estado da arte do campo.

2. Principais Componentes da Arquitetura

A arquitetura MIPS tem uma característica que faz ela ser chamada de simples, que é seu conjunto reduzido de instruções. Além disso, suas instruções possuem um formato conhecido e que não varia, e usam quase sempre os mesmos operadores, que são os registradores do arquivo de registradores. Por causa disso, seus componentes também são simples, pois precisam lidar com pouca ou nenhuma variação na forma com que eles lidam com as intruções.

Essa homogeneidade no conjunto de instruções é característica de arquiteturas do tipo RISC (Reduced Instruction Set Computer) (Computador de Conjunto de Instruções Reduzidos) que é o oposto de arquiteturas CISC (Complex Instruction Set Computer) (Computador de Conjunto de Instruções Complexas) que podem ter instruções com diferentes tamanhos, e que permitem operações tanto em registradores quanto em memória, o que causa uma maior complexidade no circuito para lidar com essa variações.

2.1. Contador de Programa

Chamado de Program Counter (PC), que é seu nome em inglês, esse componente pode ser entendido como um registrador de 32 bits que armazena qual a instrução que está sendo executada pelo processador. Além disso, a memória do MIPS tem a característica de ser alinhada por endereços, e cada unidade de endereço tem 1 byte, e como estamos lidando com uma arquitetura de 32 bits, cada instrução tem 4 bytes, portanto, o PC incrementa de 4 em 4 a cada ciclo.

Por exemplo, quando ele está com o valor 4, significa que a próxima instrução que será executada é a segunda, se seu valor for 8, a instrução a ser executada é a terceira (iniciando a contagem das instruções por 0). Normalmente o pino de saída do contador é conectado à porta de endereço da memória de instruções.

A figura 1 mostra o componente pronto fornecido pelo Logisim [Fiori 2018], e na Figura 2 uma implementação do PC usando os componentes básicos, como um registrador e um somador.

O funcionamento das conexões são as seguintes:

- Clock: Pequeno triângulo na lateral do componente, representa o pino para entrada do sinal de clock. O Logisim permite configurar o componente para responder a diferentes posições do clock: Borda de subida, borda de descida, nível baixo e nível alto.
- R: Sinal usado para limpar o registrador. Quando o valor 1 é enviado nessa porta, o valor do PC é colocado em zero.
- Q: Pino para leitura do valor no registrador, equivalente à saída na direita inferior no componente pronto.
- D: Pino para entrada do valor no registrador. Equivalente à conexão inferior na esquerda do componente pronto

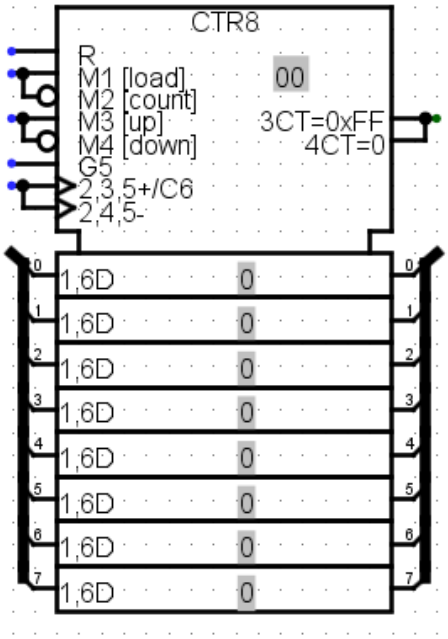

Figura 1. Componente do contador de programa fornecido no Logisim.

- WE: Habilita a escrita no registrador na borda de clock. Quando o WE é 1 e o clock acontecer, o registrador é atualizado para o valor entrando no pino D. Equivalente ao pino M1 no componente fornecido pelo Logisim, ambos podem ser conectados a ao componente constante setado para o valor 1.

2.2. ROM (Read Only Memory) (Memória Somente de Leitura)

O componente ROM normalmente é usado como memória de instruções do processador. Como as instruções de um programa não variam durante a execução, não é necessário uma memória que permita regravações para guardar as instruções do programa.

Na Figura 3 vemos o componente pronto da ROM no Logisim.

O funcionamento das suas conexões são os seguintes:

- Conexão mais à esquerda: Endereço do dado a ser lido. Esse pino recebe um valor de 32 bits normalmente vindo do contador de programa. O valor colocado nesse pino indicará qual instrução o processador irá acessar.
- Conexão mais à direita: Dado que estava armazenado na posição de memória informada na conexão da esquerda. Como esse valor de saída é uma instrução para a arquitetura MIPS, então ela também tem 32 bits.

2.3. Arquivo de Registradores

O arquivo de registradores, mostrado na Figura 4, é uma unidade funcional da arquitetura do MIPS que organiza e centraliza os 32 registradores de um MIPS de 32 bits. Um processador baseado em MIPS só pode realizar operações em registradores, portanto, toda operação precisa ter seus valores em algum registrador do arquivo de registradores. Sempre que os registradores precisam ser lidos ou um registrador precisa ter seu valor atualizado, alguns sinais de controle devem ser enviados para o arquivo de registradores identificando a ação a ser realizada.

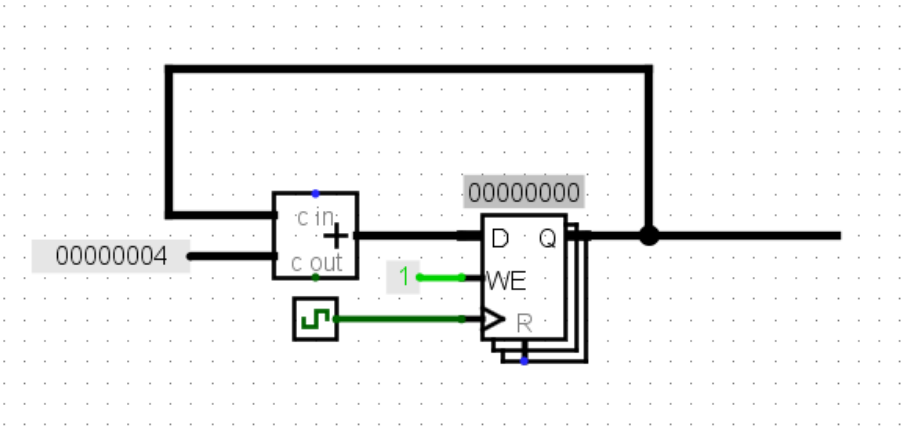

Figura 2. Implementação de um PC usando componentes básicos.

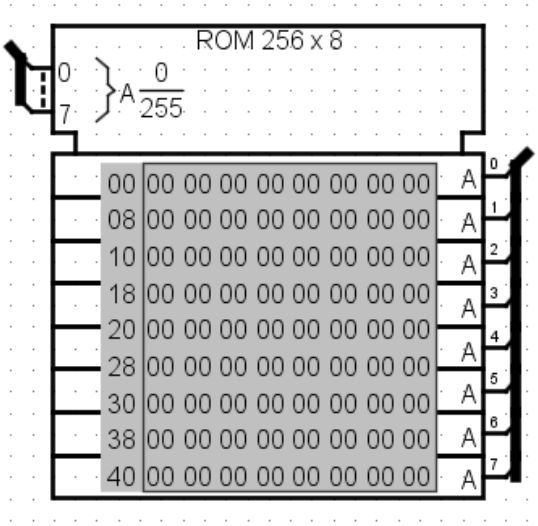

Figura 3. Componente ROM do Logisim.

O componente possui 8 conexões, e suas funcionalidades e usos são os seguintes:

- CLK: Indicado por um pequeno triângulo é a conexão para o sinal de clock do processador.
- A e rA: A conexão rA serve para entrada do endereço de qual registrador terá seu valor lido para a conexão A. A entrada na conexão A é um valor de 5 bits.
- B e rB: São análogos ao funcionamento de A e rA.
- W e rW : rW é uma conexão de 5 bits que identifica qual registrador terá seu valor alterado para o valor que está sendo colocado na conexão W, que recebe um valor de 32 bits.
- WE: Conexão usada para indicar se uma escrita no banco de registradores pode acontecer.

2.4. Decodificador de Instrução

O MIPS possui 3 formatos de instrução, todas de 32 bits, com os 6 bits mais à esquerda representando o código de operação (opcode), os bits restantes são os parâmetros da

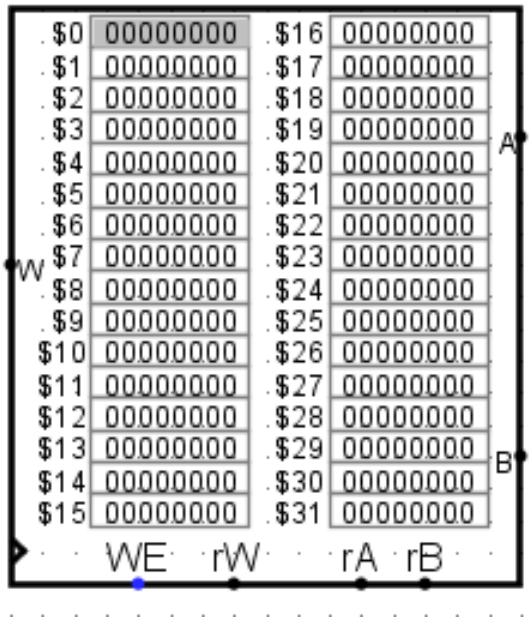

Figura 4. Banco de Registradores do MIPS.

instrução. Esses serão usados de forma diferente pelo processador de acordo com o valor do opcode, na Figura 1 são mostrados os três tipos de instruções e quais são seus campos.

Instrução	Campos					
Formato	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Tipo R	op	rs	rt	rd	shamt	funct
Tipo I	op	rs	rt	Endereço/Imediato		
Tipo J	op	Endereço de Destino				

Tabela 1. Formatos de instruções do MIPS.

Os três formatos de instrução são chamados de R-Format (Register Format), I-Format (Immediate Format) e J-Format (Jump Format).

O formato R é normalmente utilizado por instruções que usam até 3 registradores em sua execução. Além disso, o opcode utilizado para instruções desse tipo é 0x0, que indica instruções lógicas e aritméticas, qual operação será realizada é indicada pelo campo funct, que são os 6 últimos bits da instrução.

No formato *Immediate*, normalmente temos dois campos de registradores mais um campo de 16 bits para um valor constante que é usado de acordo com a instrução. Exemplos de instruções desse tipo são load word, store word e branch. Nas instruções load e store o campo immediate é usado como um offset, e para instruções de branch o immediate é o número de instruções para trás ou para frente que o branch vai saltar em relação a PC.

As instruções do tipo J são usadas para desvios incondicionais, com o campo de

endereço de destino sendo o valor da instrução de destino. Um detalhe das instruções do tipo J é que elas possuem apenas 26 bits para o campo do endereço de destino, e as instruções são endereçadas com 32 bits. Apesar disso, duas técnicas são usadas para compensar esses bits perdidos, uma delas é saber que o endereço das instruções são sempre múltiplos de 4, ou seja, os dois últimos bits são sempre 00, portanto, para calcular o endereço de destino basta dividir por 4 antes de colocar no campo. E os outros 4 bits restantes são os 4 bits mais significativos do contador de programa, que são desconsiderados pois normalmente quando queremos fazer um jump é para uma posição próxima do código que estamos executando, no momento da reconstituição do endereço de destino são concatenados os 4 bits mais significativos do PC no momento atual.

Para tentarmos entender melhor como essa divisão dos parâmetros da instrução acontece, vamos usar como exemplo a seguinte instrução já em binário: OPCODE10010010001000000010000010. Seguindo o esquema da Figura 7 podemos separar a instrução conforme a Tabela 2.4.

Tipo R	OPCODE	10010	01000	10000	00010	000010
Tipo I	OPCODE	10010	01000	1000 0000 1000 0010		
Tipo J	OPCODE	10 0100 1000 1000 0000 1000 0010				

Tabela 2. Exemplo de sparação de uma instrução do MIPS

Essa tarefa de separar os campos da instrução é papel da unidade de decodificação de instrução. Como entrada o componente recebe um valor de 32 bits recebido da memória de instruções, e deve separar todos os possíveis campos da instrução distribuídos em diferentes pinos de saída, mesmo que para determinada instrução um campo não seja utilizado, como por exemplo em uma instrução de tipo R ter um valor de imediato, o decodificador de instruções ainda assim vai gerar o valor de imediato. Neste momento não vamos tratar como os valores são usados, a unidade apenas vai separar os diferentes campos. A Figura 5 mostra a unidade de decodificação criada para o caminho de dados.

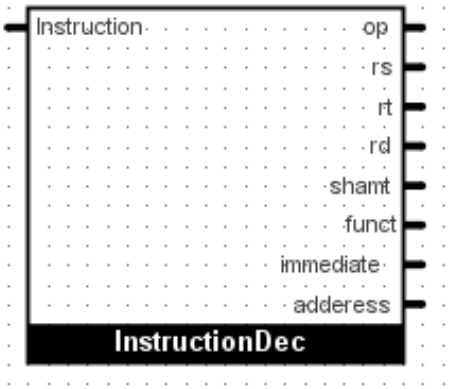

Figura 5. Componente Decodificador de Instruções.

2.5. Unidade de Controle

A Unidade de Controle (UC) tem o papel de, a partir dos campos opcode e funct fornecidos pelo Decodificador de Instruções, gerar todos os sinais de controle possíveis para

que a instrução seja corretamente executada pelo processador. A UC gera sinais como quais registradores serão lidos, sinal de Write Enable do arquivo de registradores e qual operação será realizada na ULA. A UC também gera flags indicando se os valores que precisam ser utilizados são imediatos, se acontecerá um jump incondicional ou se o resultado da operação será gravado em memória ou em um registrador de destino. A Figura 6 apresenta um esquema de entradas e saídas da UC.

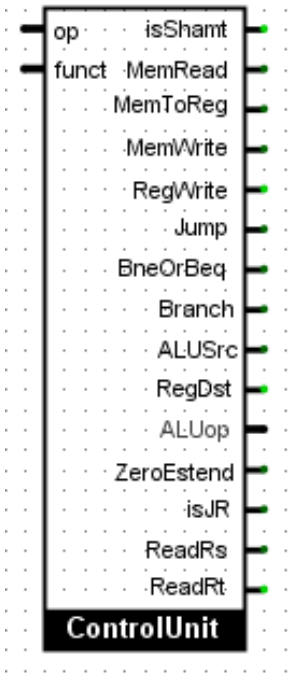

Figura 6. Unidade de Controle.

2.6. Unidade Lógica e Aritmética (ULA)

A unidade lógica e aritmética de um processador, mostrada na Figura 7, é o local onde acontecem todas as possíveis operações lógicas ou aritméticas suportadas por uma arquitetura MIPS. Exemplos dessas operações são, soma e subtração, que seriam operações aritméticas, e operações AND, OR ou XOR bitwise que seriam exemplos de operações lógicas suportadas por uma ULA de um MIPS. Suas conexões são:

- X: Entrada do valor do registrador A que vem do banco de registradores.
- Y: Entrada do valor do registrador B que vem do banco de registradores.
- Result 1: Resultado da operação realizada entre X e Y.
- Result 2: Devolve o carry out das operações de multiplicação e divisão.
- OP: Entrada de 4 bits usada para indicar qual operação será realizada com os valores de entrada.
- isEqual: Sinal 0 ou 1 que indica se o valor fornecido em X é igual o valor fornecido em Y.
- Unsigned Overflow indica se existiu um estouro de representação após realizada a operação.

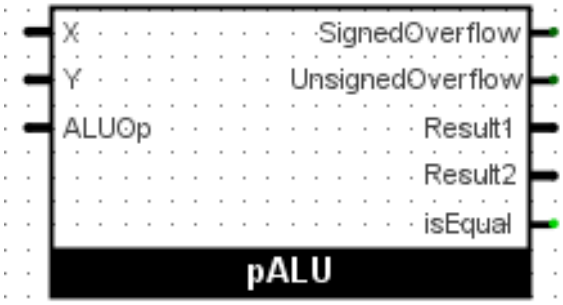

Figura 7. Representação de uma ULA de um MIPS.

3. E o que Podemos Fazer com Todas Essas Partes?

A partir desse ponto, com esses componentes, conseguimos montar uma versão do caminho de dados chamada de “monociclo”, mostrado na imagem 8. O motivo desse nome é que ele executa toda instrução em apenas um ciclo, a instrução e os resultados “atravessam” o processador de um lado para o outro em um único passo.

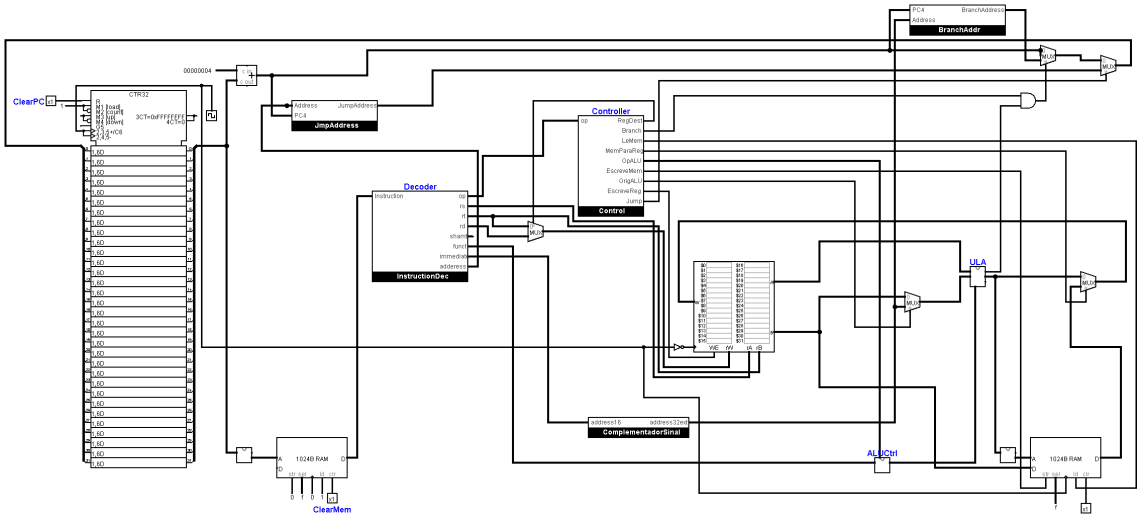

Figura 8. Imagem de um datapath monociclo montado.

A primeira vista isso parece um funcionamento bem interessante, entretanto, é possível melhorar esse funcionamento implementando um conceito chamado de “pipelining”.

3.1. Porque Utilizar um Pipeline?

No contexto de um processador de computador, o que chamamos de pipeline é a divisão do caminho de dados em diferentes estágios, e então, agora cada instrução será executada em partes com cada parte executada em um ciclo diferente. a primeira vista essa implementação parece ser sem sentido, pois, agora uma instrução que demorava um ciclo para ser executada vai demorar o número total de estágios em ciclos.

Em um primeiro momento pode parecer estranho querermos que nossas instruções demorem mais ciclos para serem executadas, mas o que acontece nessa implementação é

que conseguimos colocar mais de uma instrução no processador ao mesmo tempo, então, apesar de diminuirmos o tempo de resposta para um instrução individual, a vazão de um conjunto maior de instruções aumenta. Veja o seguinte exemplo.

Suponha a atividade de lavar roupas em uma lavanderia, e que isso é feito em quatro etapas: lavar, secar, dobrar e guardar. Se dissermos que cada etapa demora 30 minutos para ser concluída, uma pessoa conseguiria terminar de usar o equipamento da lavanderia após duas horas, e se tivermos quatro pessoas para lavar suas roupas naquele dia, o tempo total é de oito horas, veja o gráfico na imagem 9.

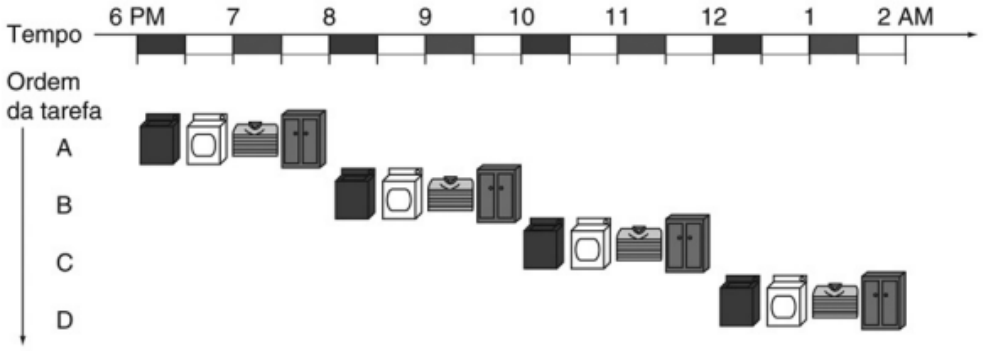

Figura 9. Lavanderia sem o conceito de pipeline.

Agora, vamos supor que queiramos implementar a técnica de pipelining na lavanderia. Isso significa que assim que uma pessoa terminar de usar um dos equipamentos, uma outra pessoa está autorizada a usar ele também, ou seja, teremos mais de uma pessoa realizando a atividade de lavar as roupas ao mesmo tempo. Essa mudança na dinâmica da lavanderia permitiu que a mesma quantidade de pessoas terminassem de lavar suas roupas em menos tempo, pois não precisam mais esperar alguém terminar o processo completo antes de começarem o seu.

Nesse caso, o tempo total em minutos para quatro pessoas lavarem suas roupas segue a fórmula 30x(E+(P-1)), onde E é o número de estágios na lavanderia e P e o número de pessoas. Nesse caso ainda, nas mesmas oito horas que levou o exemplo anterior e ainda com 4 etapas, conseguimos um total de 13 pessoas com roupas lavadas, isso é um aumento de mais de três vezes na vazão total, vejá o gráfico na imagem 10.

3.2. Convertendo o Datapath para Pipeline

Após esta descoberta de o quanto um pipeline pode melhorar o desempenho de nosso processador, certamente vamos querer implementar isso no nosso caminho de dados. Entretanto, para fazer isso, teremos que construir mais alguns componentes, e esses são chamados de bancos de registradores intermediários, e também precisaremos de uma unidade de controle mais sofisticada, e também de um outro componente chamado de “unidade de hazards” que será explicado mais à frente.

4. Os 5 Estágios do Pipeline do MIPS

O pipeline implementado neste trabalho possui 5 estágios, esses são chamados de Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM),

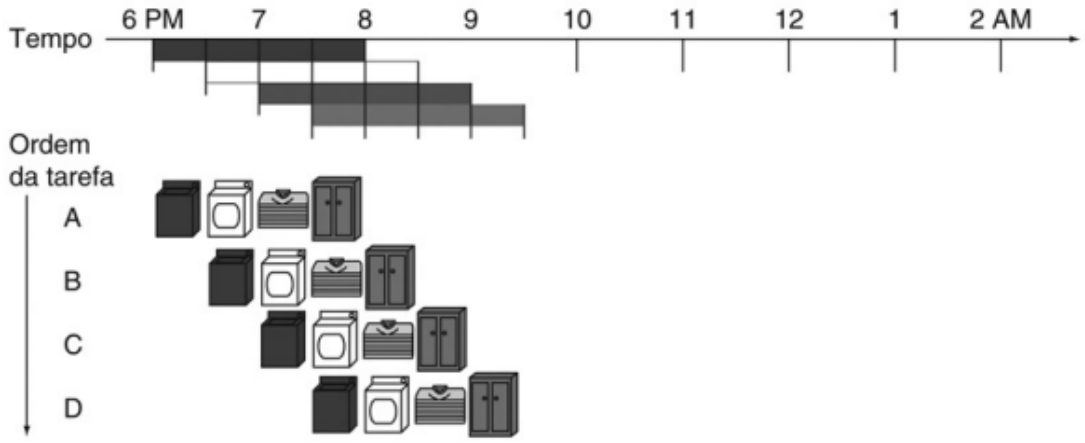

Figura 10. Lavanderia com o conceito de pipeline.

WriteBack (WB). Os 5 estágios são separados por 4 unidades funcionais que são os bancos de registradores, que levam os nomes dos estágios que eles estão separando, e eles são: IF/ID, ID/EX, EX/MEM, MEM/WB. Os bancos de registradores são responsáveis por passar as informações de um estágio para outro. A forma com que eles fazem isso é tendo internamente alguns registradores que são capazes de guardar as informações do seu estágio à esquerda que podem ser necessárias em algum dos outros estágios à direita. Essas informações vão sendo passadas e consumidas de acordo com a passagem da instrução pelo pipeline.

As funções de cada um dos cinco estágios são as seguintes:

1. Instruction Fetch (IF): Esse é o primeiro estágio do pipeline, ele é responsável por buscar a instrução que será executada, e também cálcula o endereço da próxima instrução que é PC+4.
2. Instruction Decode (ID): Esse estágio consome a instrução recuperada por IF e utilizando o decodificador e a unidade de controle, gera todos os sinais que serão necessário por todo o restante do caminho de dados para que a instrução seja executada de forma correta e seu resultado seja armazenado no lugar correto. Além disso, esse estágio também é responsável por acessar o arquivo de registradores, então, além de gerar os sinais, esse estágio também produz os dados que serão usado durante a execução.
3. Execute (EX): Nesse estágio a operação indicada pela instrução é executada na unidade lógica aritmética. Nesse estágio também podem ser calculados os endereços de desvio.
4. Memory Access (MEM): Nesse estágio está a unidade funcional da RAM, então nele acontecem os acessos a memória em intruções que realizam essa operação, como *load* e *store*.
5. Write Back (WB): Nesse estágio acontece a escrita de resultados de volta no arquivo de registradores. Esses resultados podem ser de operações realizadas na ULA ou um dado buscado na memória.

4.1. Bancos de Registradores Intermediários

Para conseguirmos executar instruções em um caminho de dados que implementa pipeline, precisaremos de algumas unidades funcionais adicionais, nesse caso os bancos de

registradores que ficam entre cada estágio.

O papel desses componentes é armazenar os resultados e sinais gerados em um estágio do caminho de dados para que esses sejam acessados posteriormente pelo estágio seguinte. E como dito antes, os sinais e resultados produzidos vão sendo encaminhados por cada estágio e consumidos em cada um deles de acordo com o que os estágios precisam de informação para realizar sua ação.

Por exemplo, se nossa instrução necessita de um acesso a memória, é preciso que no banco de registradores que fica entre os estágios EX e MEM tenha armazenado nele qual o endereço que será acessado na memória.

Outro exemplo é para o estágio de execução. Para que essa etapa consiga realizar sua tarefa, ela precisa dos operadores e do sinal que indica qual a operação a ula deve realizar com eles. Então, essa informação precisa estar armazenada de alguma forma no banco de registradores que fica entre o estágio de ID e EXE.

No caso dessa implementação, a grande maioria dos sinais e endereços que são necessários para orquestrar o comportamento do caminho de dados durante o processamento de uma instrução são gerados pelo estágio dois, pois nele é onde se encontra a unidade de decodificação e a unidade de controle, que são as unidades que ”dão significado”a instrução buscada pelo estágio IF.

5. Consequências do Estilo Pipeline

Os benefícios da utilização de uma arquitetura com pipeline não vem de graça, uma das consequências negativas da construção desse tipo de arquitetura, além do aumento da complexidade do circuito, são chamados de “hazards”. E eles são de três possíveis tipos: Estruturais, de dados, e de controle.

5.1. Hazards Estruturais

Os hazards estruturais ocorrem quando duas instruções diferentes precisam acessar uma mesma unidade funcional, por exemplo se de alguma forma uma instrução de soma e outra de adição tentassem utilizar a Unidade Lógica e Aritmética ao mesmo tempo. Caso isso aconteça, pode ocorrer uma mistura dos sinais de controle ou dos resultados gerados pela unidade funcional, o que acaba por gerar dados inúteis para todas as instruções que tentaram o acesso.

Essa solução funciona no MIPS pois a construção do pipeline garante que não ocorra de duas instruções tentarem acessar a mesma unidade funcional ao mesmo tempo. O motivo disso é que toda instrução sempre precisa passar por todo o caminho de dados mesmo que não utilize algum dos estágios, obrigando então a todas as instruções levarem a mesma quantidade de tempo para serem executadas.

Por conta desse comportamento do funcionamento do MIPS, é garantido que uma instrução use uma unidade funcional apenas quando ela não está mais sendo usada para outra tarefa. Isso também é garantido pelo funcionamento dos bancos de registradores intermediários, nos quais a gravação dos dados ocorre no sinal de subida do relógio e a leitura no sinal de descida.

5.2. Dependência de Dados

Antes da explicação sobre os *Hazards* de Dados, é necessário que tenhamos entendimento sobre o conceito de dependência de dados.

Como normalmente na execução de um algoritmo realizamos operações para produzir alguns dados para que eles sejam usados depois, é esperado que dependências entre esses resultados ocorram. Um pequeno código de exemplo para dependência de dados pode ser como mostrado no código 1.

Código 1. Exemplo de código com dependência de dados

```
1 addi $t1, $s1, 5
2 sub $t2, $t1, $t3
3 or $s0, $t2, $t4
```

Nesse pequeno trecho de código é possível ver duas dependências de dados. A primeira, é a dependência que a segunda instrução tem do resultado produzido na instrução 1, que vai ter seu resultado salvo em t1 e logo já é usado pela instrução 2. E na instrução 3 temos um caso análogo, em que ela precisa de um dado que será produzido pela instrução 2 e salvo em t2. Na imagem 11 é mostrada uma representa]ao de como essa dependência aconteceria no pipeline de acordo com a execução das instruções.

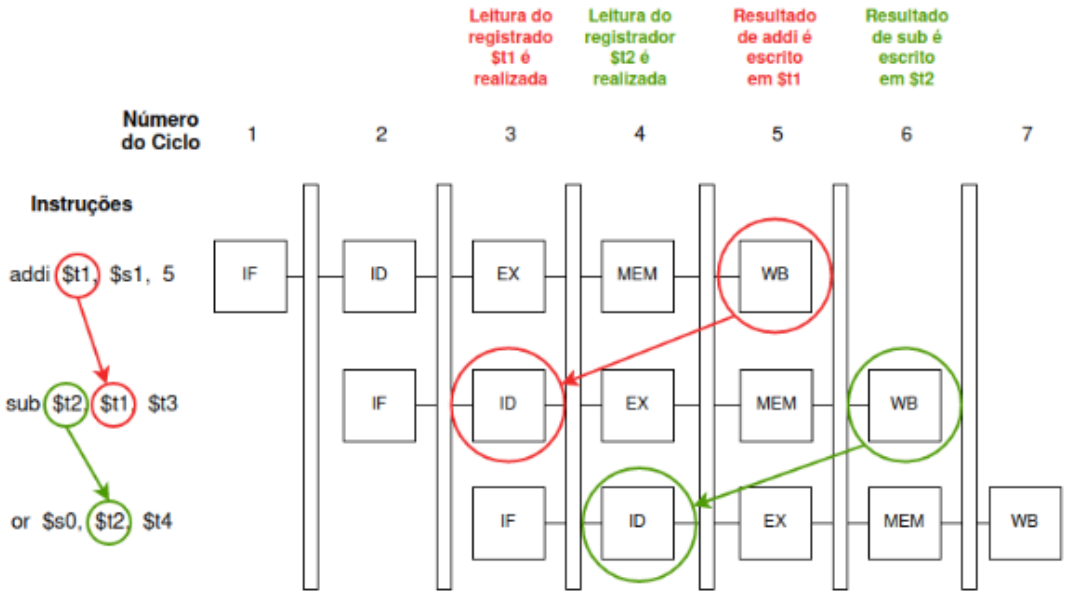

Figura 11. Dependêncai de Dados no Pipeline.

5.3. Hazards de Dados

Agora, com os conceitos de dependência de dados explicados, é possível entender porque os *hazards* de dados aparecem quando uma arquitetura é convertida para pipeline. Como a execução de uma instrução é separada em etapas, e ela avança uma etapa por ciclo, quer dizer que em um pipeline de 5 estágios, um dado que for gerado por uma instrução só poderá ser acessado 5 ciclos depois que a instrução entrou para ser processada.

Assim, no caso do exemplo mostrado no código 1, esse código não seria executado corretamente em um *datapath* com pipeline, o dado gerado pela primeira instrução não estaria disponível ainda no arquivo de registradores para que a instrução dois pudesse buscá-lo e usá-lo

5.3.1. Formas de Tratamento

As possíveis formas de tratamento para os *hazards* de dados não são tão triviais como as dos *hazards* estruturais. Para lidarmos com esse problema podemos usar tanto a técnica de bolha ou de *forwarding*.

A técnica da bolha é a mais simples estruturalmente de ser implementada, e também resolve todo tipo de *hazard* de dados. A forma com que ela funciona é colocando instruções vazias, chamadas de bolhas, no pipeline, até que o dado necessário para a execução da instrução que está para ser executada esteja finalmente disponível, assim podendo ser buscado para a correta execução da instrução.

A segunda forma de tratamento desse problema é chamada de *forwarding*. A proposta dessa técnica é de, logo que a informação produzida ficar pronta, ela já pode ser reencaminhada para o estágio que esteja necessitando da informação para a execução de uma outra instrução.

No caso do exemplo mostrado em 1, a unidade de *forwarding* removeria a necessidade de bolhas, pois, assim que a primeira instrução tivesse seu resultado produzido pela ULA, esse dado poderia ser reencaminhado para a própria ULA no pino que estivesse requerendo aquele dado. Não esquecendo que o dado ainda precisa "caminhar" pelo resto do caminho de dados para ser devidamente gravado na memória ou no arquivo de registradores.

5.4. Hazards de Controle

Esse tipo de *hazard* ocorre quando temos alguma instrução que precise alterar o contador de programa. O problema causado por isso é que não temos como saber para qual valor o contador será alterado até que a instrução de desvio grave a nova posição no quarto estágio, assim não temos como saber se devemos buscar a próxima instrução (PC+4), ou se na verdade vamos buscar a instrução para qual o desvio colocar o contador.

A forma mais direta de se lidar com isso é colocando quatro bolhas após a instrução de desvio. Entretanto, isso é bem ruim, pois ficaremos três ciclos sem fazer execução útil, o que é uma situação bem ruim, e torna uma arquitetura ineficiente.

A outra forma seria mover o circuito de desvio para o segundo estágio e adicionar apenas uma bolha. Isso funciona pois comparar dois valores é uma tarefa fácil, podemos fazer isso usando apenas uma operação de XOR, caso os valores sejam corretos a saída da operação é 0. E como as instruções de desvio condicional do MIPS trabalham com a comparação de igual ou diferente, essa implementação se torna possível.

A terceira forma de lidar com essa situação é mover o circuito de desvio para o segundo estágio e implementar uma técnica de predição, entretanto esse tópico não será abordado nesse texto, mais informações podem ser encontradas em [Hennessy and Patterson 2014].

5.5. Unidade de Hazard

Uma das maneiras de lidar com os *hazards* é a adição de bolhas pelo compilador. Para fazer isso, o compilador precisa gerar o código de máquina pronto com esse tratamento de erros. Um complicante é o conhecimento de detalhes arquiteturais pelo compilador, para identificar os pontos para a inclusão das bolhas, principalmente se considerarmos a possibilidade de termos diferentes arquiteturas com diferentes quantidades de estágios em um pipeline.

Implementar na arquitetura uma unidade capaz de detectar a ocorrência desses *hazards* evita esse complicante, pois inclui as bolhas de forma transparente para o compilador, e também garante que a bolha sempre será incluída quando necessário, sem intervenção do software.

5.5.1. Funcionamento da Bolha no Pipeline

O nosso processador é composto por elementos de estado e combinacionais. Elementos de estado são aqueles que têm a capacidade de de armazenar informações e manter um estado interno. Eles são caracterizados por poderem ter diferentes saídas para mesmas entradas. Isso permite que esses elementos ditem comportamentos futuros de acordo com acontecimentos passados. No nosso processador, os elementos de estado são o contador de programa (PC), o arquivo de registradores, a memória RAM e os bancos de registradores intermediários.

Bolhas desativam sinais de escrita dos elementos de estado para garantir a preservação do contexto dos componentes, com a intenção de aguardar o momento que o dado esteja disponível para a instrução prosseguir corretamente.

O seu comportamento é da seguinte forma: Primeiro, precisamos evitar que uma nova instrução seja buscada, pois já existe uma instrução no pipeline incapaz de prosseguir. Para alcançarmos isso, negamos o sinal de *clock* que chega ao contador de programa, fazendo com que ele não mude o endereço da instrução a ser buscada. Além disso, precisamos evitar o avanço dos sinais guardados nos bancos de registradores intermediários, para que não percamos as informações que estão ali, já que a instrução vai ficar parada por alguns ciclos de acordo com a distância da instrução que requisita o dado para a instrução que produz o dado, até que ela esteja disponível. A maneira que fazemos isso é desativando o sinal de *escrita* de todos os registradores dos bancos de registradores IF/ID e ID/EX.

Já para as unidades funcionais do arquivo de registradores e da memória, e também para o estágio da ULA, que precisa estar executando alguma operação, é possível apenas desabilitar os sinais de *WriteMem* e *WriteReg*. Assim, independente do que seja produzido nos estágios após EX, nenhuma alteração será feita pela instrução da bolha no estado do processador.

5.5.2. Como Implementar uma Bolha no Pipeline?

Agora que sabemos como a bolha funciona, precisamos descobrir como implementar um hardware capaz de executar seu funcionamento de forma autônoma.

O fundamento da bolha é inserir uma pausa no pipeline quando requisitado um dado no estágio EX que ainda está sendo produzido por outra instrução. Assim, precisamos pedir para a instrução que está chegando para ”ter calma”, até que tudo esteja pronto para sua execução.

Os condicionais que a unidade de *hazard* deve cobrir estão descritos no Código 2.

Código 2. Condicionais para Inserção da Bolha no Pipeline

```
1  if (EX/MEM.WriteReg
2  and (EX/MEM.RegistradorRd != 0)
3  and (EX/MEM.RegistradorRd == ID/EX.RegistradorRs))
4          Insere uma bolha.
5
6  if (EX/MEM.RegWrite
7  and (EX/MEM.RegistradorRd != 0)
8  and (EX/MEM.RegistradorRd == ID/EX.RegistradorRt))
9          Insere uma bolha
10
11 if (MEM/WB.WriteReg
12 and (MEM/WB.RegistradorRd != 0)
13 and (MEM/WB.RegistradorRd == ID/EX.RegistradorRs))
14         Insere uma bolha.
15
16 if (MEM/WB.WriteReg
17 and (MEM/WB.RegistradorRd != 0)
18 and (MEM/WB.RegistradorRd == ID/EX.RegistradorRt))
19         Insere uma bolha
```

5.5.3. Tratamento Para os Hazards de Controle

Além dos *hazards* de dados, ainda precisamos tratar os *hazards* de controle. Como explicado anteriormente, esses acontecem quando temos instruções que podem alterar o valor do PC de forma condicional quando executadas, como por exemplo um desvio. Isso faz com que não saibamos qual vai ser o próximo endereço do PC até que a instrução termine de passar pelos estágios do pipeline que ela precisa para ser concluída.

Uma maneira de tratar esse problema de forma eficiente é utilizando de previsão de desvio, situação que é melhor explicada no livro [Hennessy and Patterson 2014], entretanto não é coberta neste trabalho por conta da complexidade de implementação.

A forma de resolver esse problema proposta neste trabalho é de aguardar o resultado do desvio, e para isso, é possível colocar bolhas no pipeline. E nesse caso, existem duas opções: (1) caso a comparação de desvio esteja sendo feita no estágio EX, serão necessária três bolhas no pipeline; e (2) colocar o hardware necessário para comparação no estágio dois, e nesse caso precisamos de duas bolhas no pipeline apenas. Essa segunda alternativa é possível pois no conjunto de instruções do MIPS, os desvios são normalmente baseados em **beq** ou **bne**, e a comparação ode igualdade é uma ação que pode ser executada facilmente com uma operação lógica XOR. Como no segundo estágio já temos

os valores dos registradores a serem comparados, e também sabemos se a instrução é um desvio ou não, é possível realizar essa operação nele.

6. Conclusão

Nesse trabalho, descrevemos o processo de implementação de um processador MIPS com pipeline de cinco estágios. Com certo grau de conhecimento em computação e interesse por arquitetura de computadores, este manual possibilita ao leitor entender melhor como é o processo de um computador executar uma instrução.

Nossa intenção é criar no leitor uma curiosidade por essa área da computação, gerando interesse em trabalhos técnicos ou pesquisa científica na área, visto a importância do ramo para a computação e a sociedade, principalmente no momento histórico que nos encontramos em que vários pesquisados mostram que estamos tendo grandes dificuldades de manter o grau de evolução do processadores seguindo os modelos atuais de arquitetura de computadores. Assim, esperamos trazer o interesse de novas mentes com novas ideias que pode acabar gerando avanços científicos no campo.

Trabalhos futuros podem cobrir técnicas não abordadas aqui como predição de desvio e encaminhamento conhecidas na literatura.

Referências

Fiori, P. (2018). logisim-evolution. `https://github.com/PabloFiori/Logisim3410Hacked`. Acesso em: 29/11/2023.

Hennessy, J. and Patterson, D. (2014). Organização e Projeto de Computadores: A Interface Hardware/Software. pages 525–539. Elsevier Brasil.

Patterson and Hennessy (1984). *M I P S Reference Data*. Elsevier.