Desenvolvimento da Unidade Operativa Pipeline MIPS

Departamento de Ciência da Computação - Universidade de Brasília
Disciplina: Organização e Arquitetura de Computadores
Professor: Flávio de Barros Vidal
Turma: B

Pedro Saman Cesarino - 15/0144890
Faculdade de Tecnologia - UnB
pedrosaman96@gmail.com

Rafael Augusto Torres - 15/0145365
Faculdade de Tecnologia - UnB
rafael.a.t97@gmail.com

Pedro Victor Galieta Tomaz - 15/0144938
Faculdade de Tecnologia - UnB
pedrogalieta@hotmail.com

Brasília
21 de junho de 2018

Abstract

In modern days processors are tools largely used in the technology field. A common strategy used to implement those devices is called Pipeline, which allows multiple instructions to be processed simultaneously, increasing instructions throughput. MIPS architecture is usually studied and used as a first contact with computer architecture and organization due its simplicity and intuitive implementation. In this project we developed a theoretical processor utilizing Altera Quartus II software applying the pipeline strategy. Through Waveform tool it was simulated the implementation for different precompiled code aiming to verify the project functionality. With the simulated processor fully functional to a subset of the native MIPS instructions, we were able to note important characteristics of this kind of processor organization, such as increased instruction throughput, data and control hazards.

Resumo

Processadores são ferramentas amplamente aplicadas atualmente no ramo da tecnologia. Uma estratégia comum para implementação desses dispositivos é o Pipeline, que permite que múltiplas instruções sejam processadas simultaneamente, aumentando a vazão do processo. A arquitetura MIPS é comumente estudada e usada como primeiro contato com arquitetura e organização de computadores, fato que se deve à sua simplicidade e implementação intuitiva. Nesse trabalho, desenvolveu-se um processador teórico utilizando o software Altera Quartus II aplicando a estratégia de Pipeline. Através da ferramenta "Waveform", simulou-se a implementação para diferentes códigos pré-compilados com o intuito de verificar a funcionalidade do projeto. Com o processador simulado se mostrando funcional para um subconjunto de instruções MIPS nativas, foi possível observar algumas características importantes deste tipo de organização, como aumento da vazão de instruções e controle de hazards, de dados ou de instruções.

1. Objetivos

Esta atividade teve como objetivo central a familiarização com o projeto, controle e execução de instruções da unidade operativa *Pipeline* MIPS. Propôs-se o desenvolvimento completo de um processador MIPS com organização do tipo *Pipeline*, contemplando todos módulos desde tratamento de exceções, *hazards* de dados ou controle e overflow.

O produto final desse projeto deve operar de forma automática por meio do ambiente Altera Quartus II, e ser capaz de processar um subconjunto de instruções pré-compiladas. A cada estágio do *Pipeline*, demonstrar claramente a instrução que está sendo executada e os valores dos registradores no dado momento.

2. Introdução

Para a implementação de um processador funcional utilizou-se a plataforma Quartus, que permite simular o funcionamento do circuito lógico-funcional desenvolvido e observar quaisquer erros de implementação. A fim de entender a organização *Pipeline* implementada, uma breve explicação será discorrida logo abaixo, assim como uma comparação com os métodos uniciclo e multiciclo. A escolha da arquitetura MIPS foi influenciada pela sua facilidade de entendimento e poucos estágios quando utilizada em *Pipeline*.

2.1. Arquitetura MIPS

A arquitetura MIPS aplicada no projeto é uma das mais estudadas por ser extremamente didática e simples quando comparada com outras, a exemplo da IA32. Ela é baseada em registradores, isto é, processadores com essa implementação utilizam registradores para armazernar e posteriormente acessar informações pertinentes. Neste caso, são trinta e dois registradores de trinta e dois bits cada, porém nem todos são acessíveis aos programadores, uma vez que há aqueles reservados para o hardware.

O MIPS segue uma filosofia baseada em 4 pilares: "Simplicidade favorece regularidade; o menor é (quase sempre) mais rápido; um bom projeto demanda compromissos e o caso comum deve ser mais rápido". Esses princípios foram responsáveis por determinar vários aspectos de funcionamento da implementação, dentre eles, o tamanho e a organização das instruções.

Outra característica importante consiste no MIPS ser uma arquitetura RISC (Reduced Instruction Set Computer), o que significa que existe um conjunto pequeno de instruções nativas que demoram, aproximadamente, o mesmo tempo para serem executadas. Em contrapartida, existem também as arquiteturas CISC (Complex Instruction Set Computer), que contam com amplo número de instruções nativas especializadas que possuem uso extremamente específico. A redução no número de instruções do MIPS permite que seja consumida uma quantidade menor de recursos na confecção do processador (grasnistores, por exemplo), barateando o desenvolvimento de um processador que segue essa arquitetura.

2.2. Pipeline

A filosofia *pipeline* consiste em "Melhorar o desempenho aumentando a vazão das instruções, ao invés de diminuir o tempo de execução de uma instrução individual". Isso quer dizer que, quando se implementa essa técnica, as instruções individualmente não melhoram em desempenho, porém sua vazão aumenta já que um ciclo do *pipeline* é menor que um ciclo de uma implementação uniciclo, implicando que a quantidade de operações por tempo seja maior [1].

Dessa forma, a implementação *Pipeline* é uma técnica de projeto que permite várias instruções estão sendo processadas simultaneamente no hardware sem que seja necessário o término de uma instrução para o início da próxima. Para que tal estratégia seja possível, é preciso dividir o processamento de cada instrução em diferentes estágios que sejam independentes. Por exemplo, na arquitetura MIPS, comumente são implementados cinco estágios: busca de instruções, decodificação da instrução, execução, acesso à memória de dados e escrita do resultado em um registrador de destino [1].

Por outro lado, apesar de apresentar vantagens sobre as demais técnicas, o *pipeline* tem a complicação de ser suscetível a conflitos denominados *hazards*, que podem ser de três tipos: estrutural, controle e de dados. O primeiro acontece quando duas instruções tentam utilizar, simultaneamente, um mesmo recurso. Já o *hazard* de controle acontece quando é necessário tomar uma decisão com base em uma instrução, um desvio por exemplo, enquanto as seguintes já foram iniciadas. O último deles, ocorre quando uma instrução depende do resultado de outra que ainda não terminou de ser executada [1].

2.3. Quartus

Altera Quartus II é um software voltado a dispositivos lógicos programáveis, que permite a síntese de circuitos digitais e sua posterior análise funcional e temporal. Proporciona integração com placas FPGA e, consequentemente, implementações físicas de circuitos com alta complexidade. Além do mais, inclui suporte a implementação em linguagem de descrição de hardware, como VHDL ou Verilog, e em diagrama de blocos auxiliadas da simulação por meio de *Waveform*.

3. Materiais

- Computadores;
- Software Altera Quartus II versão 9.1 sp1 ou superior.

4. Métodos

Esse trabalho foi feito pensando na modularização de cada parte fundamental de um processador com arquitetura RISC MIPS, isso facilitou o desenvolvimento visto que os alunos envolvidos puderam dividir a carga de trabalho, designando para cada o desenvolvimento de certos módulos. Entre os principais blocos que deveriam ser desenvolvidos estão: ULA (unidade lógico aritmética), controle da ULA, controle principal, banco de registradores, barreira de registradores, memória de instrução e de dados (Estilo Harvard), extensão de sinal e outros. Nas próximas subseções serão discutidas como foram implementadas cada uma dessas unidades e como foi feita a integração delas.

4.1. Controle

O controle do *pipeline* é quem orquestra toda a atividade do processamento e pode ser dividido em três unidades separadas: controle principal, controle da ULA e controle de hazards.

4.1.1 Controle Principal

O controle principal é responsável por comandar 18 sinais que controlam diversas multiplexações ao longo da execução de instruções e também por auxiliar no controle da ULA.

Esse bloco foi desenvolvido utilizando-se uma técnica semelhante àquela aplicada em PLA's (Programable Logic Array), realizando operações básicas do tipo NOT, AND e OR nas entradas. As entradas desse bloco consistem nos 6 bits de 'OpCode' da instrução e os 6 bits do campo 'funct' da instrução, os campos da funct foram utilizados para controlar o tipo de extensão de sinal aplicada, sinalizada ou não sinalizada, a permissão de escrita nos registradores 'HI' e 'LO', sinais do tipo salto não condicional (jump) e outros. Os sinais de controle implementados foram:

- BEQ = Sinal necessário para realizar salto condicional de igualdade;
- BEQn = Sinal necessário para realizar salto condicional de desigualdade;
- BEQez = Sinal necessário para realizar salto condicional com comparação de maior ou igual com zero;
- Regdst[1..0] = Controle do registrador onde o resultado da instrução será escrito:
  - 00 → RT
  - 01 → RD
  - 10 → $31, caso do 'jal'
- OrigALU[1..0] = Controle de qual sinal é realmente enviado à ULA:
  - 00 → Conteúdo de RT ou conteúdo proveniente do forwarding
  - 01 → Imediato estendido (32 bits)
  - 1X → Upper = 16 bits imediato + 16 zeros
- Escreve_mem = Sinal que permite a escrita na memória de dados;
- Escreve_reg = Sinal que permite ou não a escrita no banco de registradores;
- EHILO = Sinal que permite ou não escrever nos registradores especiais 'HI' e 'LO';
- ULAorHILO[1..0] = Seleciona se a saída do que vai ser enviado para ser escrito no registrador é o resultado da ULA, ou algum valor armazenado em HI ou LO:
  - 00 → Resultado da ULA
  - 01 → Valor armazenado 'HI'
  - 10 → Valor armazenado em 'LO'
- Ex_sel = Seleciona qual tipo de extensão de sinal irá ocorrer: 0 sinalizada e 1 extensão não sinalizada;
- PC_sel[1] = Seleciona se o endereço da próxima instrução será selecionado por PC + 4, Branch ou Jump;
- OpALU[1..0] = Mapeamento de qual tipo de instrução será operada na ULA:
  - 00 → Acesso a memória
  - 01 → Salto condicional
  - 10 → Tipo R
  - 11 → Operação aritmética com imediato
- Jr = Sinal que diferencia os pulos não condicionais: 0 salto para o endereço representado pelo imediato e 1 para o endereço guardado no registrador;
- Le_mem = Sinal que permite a leitura da memória de dados.

4.1.2 Controle da ULA

A unidade que define qual operação será realizada na ULA é o controle da ULA. Esse controle recebe o sinal vindo do controle principal "OpALU" que define qual tipo de instrução será operada, os bits do campo "funct", que auxilia a diferenciar qual operação aritmética será realizada em primeiro plano na ULA, e por último o sinal "ins" são os 3 LSB's do campo opcode da instrução.

As operações na ULA são comandadas seguindo o seguinte mapeamento:

- 000000 → A and B
- 000001 → A or B
- 000010 → A + B
- 001010 → A - B
- 001011 → A $<$ B ? 1 , 0
- 011000 → A nor B
- 000100 → A xor B
- 100000 → A $<<$ B

- 100001 → A >> B
- XXXX00 → A x B sinalizado
- XXXX01 → A / B sinalizado
- XXXX10 → A x B não sinalizado
- XXXX11 → A / B não sinalizado

4.1.3 Controle de Hazards

O módulo em questão é responsável pela detecção dos hazards de controle e de dados, e a partir dela, define sinais que aplicam, ao processador, processos de *stall* ou *flush*.

O *stall*, igualmente denominado bolha, é requisitado se, e somente se, a instrução imediatamente abaixo de um *Load Word* utiliza o registrador carregado da memória como um de seus operandos, como apresentado a seguir:

```
lw  $t0, 4($s1)
add $t2, $t1, $t0
```

Quaisquer outros casos de hazard de dados podem ser solucionados pela Unidade de Forwarding.

Levando em consideração que a instrução LW é a única que implica no sinal de controle LeMem == 1, pode-se verificar a ocorrência desse hazard em seu estágio de Execução, da seguinte forma:

```
if(EX.LeMem == 1)
and[(EX.Rt == ID.Rt) or (EX.Rt == ID.Rt)]
```

Em caso afirmativo, deve-se manter o estágio de Identificação por mais um ciclo de Clock. Para tal, bloqueia-se a atualização tanto do valor de PC quanto da barreira IF/ID, e os sinais do Controle principal são todos substituídos por 0 (através de um mux denominado *Mux_Cont* na implementação) para que se execute uma instrução *Nop*.

Por outro lado, o *Flush* é aplicado unica e exclusivamente em casos de salto nas instruções, seja por conta de *Branch* ou *Jump*. A razão que o faz necessário se deve ao fato do sinal de desvio ser identificado apenas no estágio de Memória e, portanto, as três instruções seguintes terem sido iniciadas. Um exemplo, destacado em vermelho, pode ser verificado na figura 1.

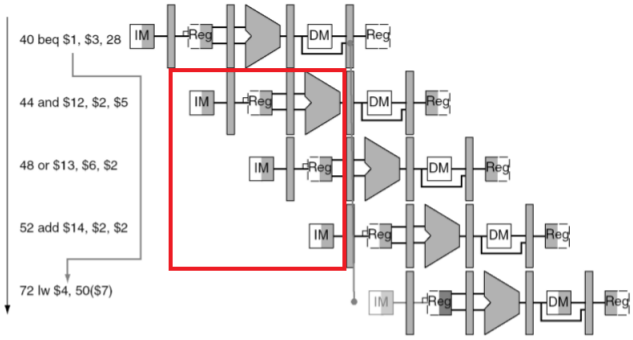

Figura 1. Exemplo de fluxo de Pipeline.

Assim sendo, caso seja constatado que se deva executar, de fato, um desvio, substitui-se o conteúdo das barreiras IF/ID e ID/EX por 0 (Clear), fazendo o mesmo com os sinais do Controle principal. Na subida de clock seguinte, correspondente ao estágio Write-back da instrução de origem, se dá a busca da instrução de destino.

4.2. Unidade Lógico Aritmética - ULA

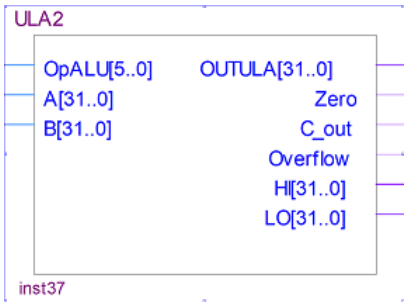

Figura 2. Unidade Lógico Aritmética.

A unidade lógica aritmética, ou ULA, é um circuito que realiza operações aritméticas e lógicas, sendo fundamental para o funcionamento dos processadores já que é ela que efetivamente realiza os cálculos e mudanças nos valores dos registradores ou memória. Para essa implementação foi desenvolvida uma ULA de 32 bits, ou seja, as operações têm duas entradas de 32 bits e a saída tem o mesmo tamanho.

A ULA foi implementada juntando unidades básicas que realizam operações lógicas bit a bit, que, organizadas de modo correto, são capazes de operar sobre sinais de entrada com vários bits. As únidades básicas realizam operações de 'and', 'or', soma com carry_in e carry_out e 'xor' sobre as entradas. Para realizar a combinação de soma de vários bits utilizou-se uma mesclagem entre as técnicas de *ripple carry* e *carry lookahead*, foram criados blocos de 4 unidades básicas com *carry lookahead*, estes blocos foram então cascateados com a técnica de *ripple carry*. Essa junção busca reduzir o tempo total necessário para se realizar as operações de soma e subtração na ULA.

Nesta unidade foi incluída blocos capazes de realizar as operações lógicas de AND, OR, NOR, XOR, SLL e SRL além das operações aritméticas de soma, subtração, multiplicação e divisão. Dentro da ULA foi implementado também a detecção de overflow para as operações de soma e subtração além de um circuito capaz de perceber se o resultado de alguma operação foi zero, que foi utilizado para as operações de *Branch equal* e *Branch not equal*.

O sinal de "ALUctr"vem do controle da ULA e ele seleciona qual operação será realizada e cujo resultado será enviado para a saída do bloco como o sinal "OUTULA". O sinal "Zero" é o bit que acusa a igualdade ou a não igualdade explicada anteriormente. Por fim, os sinais de 32 bits 'HI' e 'LO' são as words do resultado da divisão ou multiplicação que vão para registradores especiais, em que para a divisão LO recebe o quociente e HI o resto enquanto que para multiplicação os 32 LSB vão para o LO e os 32 MSB vão para o HI.

As operações na ULA são sempre realizadas paralelamente e somente a saída correta é escolhida através de multiplexadores, a unidade de *shift* utiliza como parâmetro de distância os 5 bits do campo "shamt" que chega ne entrada B da ULA.

No processador foram utilizados três blocos de ULA sendo um deles a unidade principal, aquela que realmente realiza as operações, uma que sempre realiza soma de 4 ao valor de PC, e uma que soma ao valor de PC+4 o offset do imediato para realizar os pulos condicionais.

4.3. Banco de Registradores

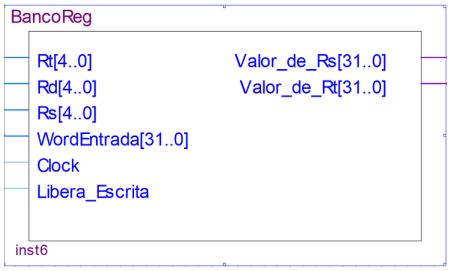

Figura 3. Banco de registradores.

O banco de registradores é um bloco que contém 32 registradores internos, que são constituídos de 32 *flipflops* tipo D com *enable* e clock, além de um *decode* de 5 bits e dois mux com 32 entradas de 32 bits e uma saída de 32 bits. Esse bloco recebe três entradas de 5 bits dois para selecionar nos mux internos quais registradores serão lidos nas saídas "Valor de Rs" e "Valor de Rt" e um que entra no decode e seleciona juntamente com o sinal de "Libera Escrita" qual registrador será escrito com a word "WordEntrada". O sinal de clock serve para permitir, na borda de subida, a escrita dos *flipflops* do registrador selecionado.

Para o bloco de registradores usou-se um clock diferente do clock geral do processador, com o período sendo metade do período original, para garantir que a escrita e leitura no banco ocorram em momentos separados. Esta característica do clock garante que a escrita realizada durante a etapa de *Write Back* ocorra antes da leitura que ocorre em uma instrução futura que esteja na etapa de *Instruction Decoding*.

4.4. Registradores HI e LO

Os registradores HI e LO são dois registradores localizados durante a etapada de execução do *Pipeline* que armazenam os resultados da multiplicação e divisão enviadas à ULA. A escrita nesses registradores é permitida através do sinal de controle EHILO, como explicado anteriormente.

4.5. Barreira de Registradores

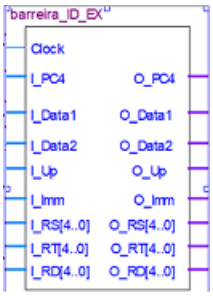

Figura 4. Barreira de registradores.

As barreiras de registradores são componentes que caracterizam a implementação do Pipeline, tornando possível a utilização de seus recursos. Essas barreiras preservam todos os sinais de words e controle que são definidas por cada instrução entre os cinco estágios do pipeline descritos na introdução. São constituídas por uma série de registradores que impedem que esses dados sejam sobrescritos no fluxo da execução.

4.6. Blocos de Memória

Visto a necessidade de se testar a implementação feita, fez-se necessário criar blocos de memória para as instruções e para os dados utilizando a arquitetura Harvard de memória, isto é, uma organização independente e separada fisicamente para os sinais e armazenamento de dados e instruções do programa. Essa característica permite que o processador acesse simultaneamente as duas memórias tornando o processo mais rápido. Foi utilizada a megafunction do Quartus II de criação de cada bloco de memória citado.

4.6.1 Memória de Instruções

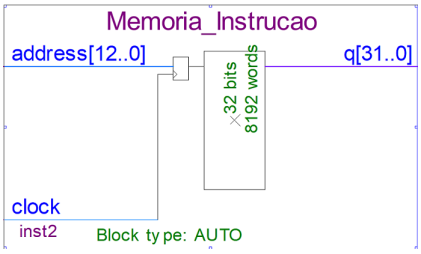

Figura 5. Memória de instruções.

A memória de instruções, como não há necessidade de escrita e que foi considerado que o programa a rodar foi pré-compilado, foi criada como uma memória ROM de uma porta com clock de leitura apenas na entrada. Foi escolhido apenas os 13 bits menos significativos como entrada para o endereço de memória, como pode ser visto na figura 5, pois esse era o valor máximo de endereçamento disponível e como não há motivo para mais espaço foi decido que esse valor era adequado. Já para a saída, ela precisa ser uma word de 32 bits, já que é esse o tamanho da instrução na arquitetura utilizada. O clock dessa memória precisa ser diferente do clock geral do processador, pois acessos a memória costumam ser demorados.

4.6.2 Memória de Dados

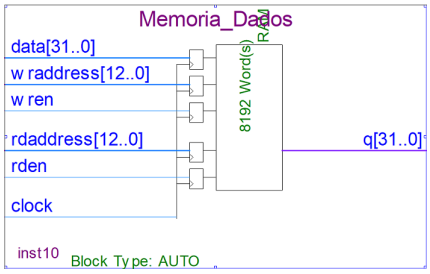

Figura 6. Memória de dados.

Já a memória de dados que tem a necessidade de leitura e escrita foi escolhida o bloco de memória RAM com uma porta para leitura e uma para escrita e da mesma forma que a memória de instruções foi satisfatório usar apenas 13 bits para o endereçamento nessa memória. Nesse caso foi necessário incluir ao bloco sinais de permissão para leitura e escrita, que devem ser antagônicos, isto é, se o sinal de leitura estiver em um o de escrita deve ser zero e vice-versa, se não há um congelamento da memória para preservar os dados. Essa memória, se em leitura, devolve no barramento 'q' o valor da word armazenada no endereço dado em "rdaddress" e, se em modo de escrita, escreve na posição dada por "wraddress" a word no barramento "data".

4.7. Extensão de Sinal

Essa unidade é responsável por fazer com que instruções que utilizam imediato sejam operadas corretamente na ULA. O que ela faz é basicamente copiar os 16 bits menos significativos do sinal de imediato da operação para a saída enquanto todos os outros bits serão iguais ao 16º bit, isto para os casos onde deseja-se realizar extensão de sinal sinalizada. Desta forma o valor numérico se mantém o mesmo, mas agora com 32 bits. Já a extensão de sinal não sinalizada puxa sempre o valor zero para os bits 16 da parte alta, esta extensão não sinalizada é utilizada para as instruções do tipo R e unsigned, addu e subu, e para a instrução ori.

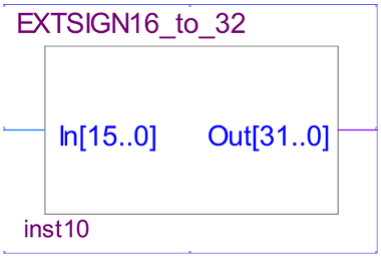

Figura 7. Extensão de sinal.

4.8. Unidade de Forwarding

Conforme citado anteriormente, uma implementação Pipeline é afetada pela ocorrência de conflitos denominados Hazards. Dentre eles, os hazards de dados se dão quando o operando de uma instrução tem dependência com o resultado de outra, anterior a ela. Cabe à unidade em questão detectar tais ocorrências e, assim, evitar que *stalls* sejam necessários.

O primeiro tipo de hazard de dado, tratado por 1.a, acontece quando o registrador Rs da instrução que se encontra na Execução corresponde ao registrador de destino da intrução a um estágio de distância a frente (Memória). Por outro lado, 1.b ocorre em situação similar, mas com Rt.

Analogamente, o tipos 2.a e 2.b configuram dependência entre Rs ou Rt da instrução na Execução e o registrador de destino daquela dois estágios a frente (Write-back).

A detecção dos casos acima descritos é realizada comparando o campo dos registradores nos casos em que a instrução anterior acessa, para escrita, o banco (EscreveReg==1). Ou seja, exemplificando o caso 1.a:

```
if(EX/MEM.escreveReg == 1)
and(EX/MEM.Rd == ID/EX.Rs)
```

Os resultados de quatro comparadores como esse são combinados para gerar sinais que selecionam dois mux posicionados imediatamente antes das entradas da ULA. Quando o seletor é 00, ambos habilitam o conteúdo dos registradores no banco para a operação. 01, por sua vez, seleciona o conteúdo proveniente do estágio WB e 10, do

estágio MEM. A imagem 8 ilustra como esse sinal é gerado e 9 contempla o bloco e todos os sinais, sejam input ou output, envolvidos no processo.

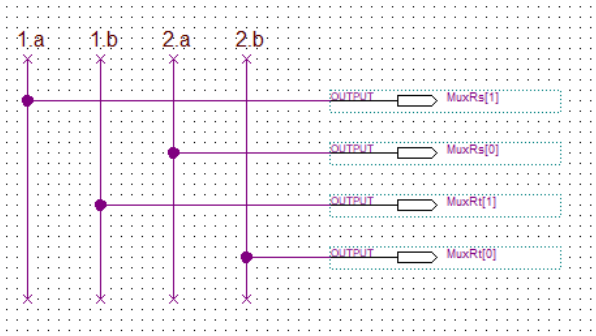

Figura 8. Determinação dos seletores dos mux

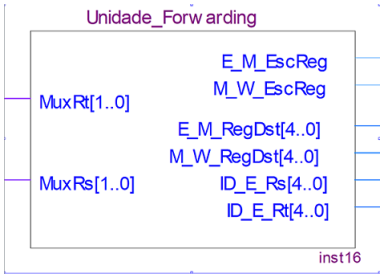

Figura 9. Unidade de *forwarding*

5. Funcionamento geral do processador

Pela característica do *Pipeline* MIPS, podem chegar a ocorrer até 5 instruções diferentes no processador em um mesmo período de tempo, cada uma em uma etapa diferente. Todo o caminho que uma instrução percorre no processador foi definido de tal modo:

5.1. Etapa Instruction Fetch - IF

Na etapa de Instruction Fetch, o registrador PC informa à memória qual deve ser a próxima instrução a ser executada, e os 32 bits dessa instrução são então enviados ao próximo estágio. O registrador PC é então incrementado em 4 e enviado ao próximo estágio, para que no próximo ciclo a próxima instrução possa ser capturada.

Nesta etapa é possível travar o update da barreira de registradores que faz a comunicação com o próximo estágio e o próprio registrador PC, para os casos onde se deseja realizar um *stall* no registrador. Nos casos onde se deseja zerar os valores armazenados na barreira, quando se realiza o *flush*, ainda existe uma entrada de clear.

Obs: Em um processador comum, o registrador PC é incrementado de 4 em 4, nesse projeto, no entanto, o valor contido em PC foi dividido por 4 na hora de ser enviado ao bloco de memória de instruções, pois neste projeto as instruções ficam em posições na memórias com diferença de 1, ou seja, as instruções são mapeadas por word, e não por byte.

5.2. Etapa Instruction Decoding - ID

Nesta etapa os 32 bits referentes à instrução são separados de acordo com os seguites campos:

- Para instruções do tipo R: opcode (6 bits) + rs (5 bits) + rt (5 bits) + rd (5 bits) + shamt (5 bits) + funct (5 bits)
- Para instruções do tipo I: opcode (6 bits) + rs (5 bits) + rt (5 bits) + imediato (16 bits)
- Para instruções do tipo J: opcode (6 bits) + address (26 bits)

Os campos de rs e rt são enviados ao banco de registradores para que os valores contidos neles possam ser acessados, o campos opcode e funct são sempre enviados ao bloco de controle para que o processador possa orquestrar como será realizada essa instrução ao longo dos outros estágios.

Nesta etapa também existe uma entrada de clear na barreira de comunicação com o próximo estágio, para garantir que o *flush* seja realizado de maneira correta.

Para o próximo estágio são enviados então os valores dentro de rs e rt, o valor de PC + 4 e os 28 últimos bits da instrução.

5.3. Etapa Execution - EX

Como o nome desta etapa indica, nela ocorre a maioria dos cálculos realizados ao longo do processamento da instrução. Vários sinais de controle são utilizados nesta etapada, como por exemplo os sinais que controlam a multiplexação que escolhe a entrada do segundo canal da ULA, diferenciando entre o valor contido em rt, o imediato extendido e o Upper (16 bits imediato + 16 zeros). A extensão de sinal sinalizada ou não também é controlada nesse estágio.

Nesta etapa também é controlada a escrita nos registradores HI e LO, também como a multiplexação que escolhe entre a saída da ULA, o valor de PC+4, ou o valor contido em HI ou LO para ser enviada ao próximo estágio. O endereço do registrador destino também é escolhido neste estágio, para diferenciar entre instruções do tipo R, tipo I ou jal.

O controle da ULA é realizado durante este estágio, garantindo que a ULA, um dos blocos essenciais para o funcionamento de qualquer processador, também ocorra de forma ótima.

O cálculo dos endereços destinos para o caso de saltos, condicionais ou não condicionais, também é realizado durante este estágio, e estes também enviados ao próximo

estágio, para que caso o salto ocorra, este seja realizado corretamente.

A unidade de forwarding age sobre essa etapa, evitando que valores contidos em registradores ainda não atualizados sejam utilizados nos cálculos necessários.

Os stalls ou flushs são decididos nesta etapa, utilizando o sinal de controle "Le_mem".

5.4. Etapa de acesso a memória - MEM

Nesta etapa se dá o acesso à memória de dados. O resultado do cálculo realizado na ULA corresponde ao endereço a ser acessado no caso das instruções *lw* e *sw*. Para os casos de escrita, o valor registrado será aquele provindo da multiplexação realizada pela Unidade de Forwarding na etapa anterior, EX.

Assim como na etapa IF, o endereço aqui passado é dividido por 4, uma vez que a memória de dados também é indexada por words ao invés de bytes.

Durante esse estágio, determina-se como será atualizado o valor contido em PC. O endereço pode ser proveniente de um registrador (caso de jr), pode ter sido calculado no estágio anterior para salto não condicional ou condicinal ou simplesmente o valor de PC+4.

5.5. Etapa Write Back - WB

Última etapa pela qual passam as instruções no *Pipeline*, responsável por decidir se algum registrador será atualizado e qual será o valor enviado a ele.

6. Resultados

A fim de verificar o correto funcionamento do processador desenvolvido, definiu-se uma série de programas a serem usados como teste. Tais testes buscaram apurar o funcionamento de cada tipo de instrução e operação a que se deu suporte no projeto. Embora os resultados sejam um pouco confusos para uma boa interpretação, é interessante observar alguns valores específicos, como por exemplo as últimas 3 linhas das simulações, que indicam claramente se haverá escrita em algum registrador, em qual registrador será feita essa escrita e qual dado será escrito, representados respectivamente pelos campos 'Escreve_reg_WB', 'OUTPUT_WB' e 'RD_WB'.

É necessário explicar que devido a alguns problemas com a plataforma Quartus utilizada nas simulações, não foi possível mostrar muitos dados nas onda geradas, por motivos de limitações do software em si e também para tentar deixar as respostas um pouco mais legíveis.

6.1. Primeiro Programa

O código em assembly do primeiro programa pode ser visualizado na figura 10.

```
1  li $t0, 50
2  sw $t0, 0($0)
3  addi $t1, $zero, 100
4  lw $t2, 0($0)
5  subu $t2, $t1, $t2
6  div $t2, $t0
7  mflo $t2
8  sll $t2, $t2, 2
```

Figura 10. Código do primeiro programa de testes.

De maneira sucinta, pode-se comparar a saída do estágio WB no *Waveform* da figura 11 ao resultado esperado em cada instrução, encontrado com a execução *step by step* do Mars:

- 1: $8 = 50
- 2: instrução de acesso à memória que não conta com escrita registrador de destino
- 3: $9 = 100
- 4: $10 = 50
- 5: $10 = 50
- 6: não conta com registrador de destino, apenas com os intermediários HI e LO
- 7: $10 = 1
- 8: $10 = 4

Existe, entre as instruções 4 e 5, hazard de dados que não pode ser solucionado pela Unidade de *Forwarding*. Assim sendo, o procedimento esperado é um *stall* de um ciclo, caracterizado pelo valor zero nos controles. Verifica-se sua execução pelo valor 0 atribuído a EscreveReg entre instruções que demandam escrita.

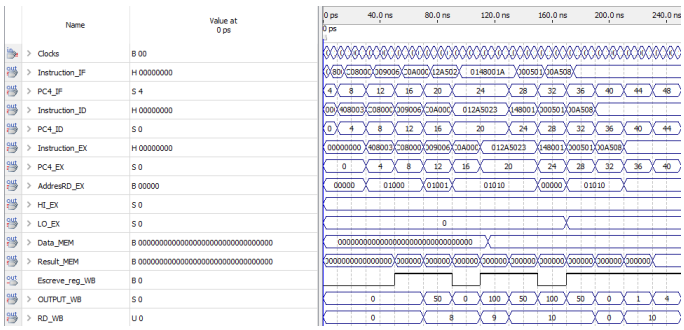

Figura 11. "Waveform" do primeiro programa de testes.

6.2. Segundo Programa

O código da figura 12 contempla instruções distintas das usadas anteriormente.

```
1  addi $t5, $0, -1
2  addi $t6, $t5 , 2
3  slt $t7, $t6, $t5
4  addi $s0, $t7, 400
5  sll $s1, $s0, 4
6  srl $s2, $s1, 4
7  mult $s1, $s2
8  mfhi $s3
9  mflo $s4
```

Figura 12. Código do segundo programa de testes.

Analogamente à seção 6.1, obtêm-se:

- 1: $13 = -1
- 2: $14 = 1
- 3: $15 = 0
- 4: $16 = 400
- 5: $17 = 6400
- 6: $18 = 400
- 7: não conta com registrador de destino, apenas os intermediários LO e HI
- 8: $19 = 0
- 9: $20 = 2.560.000

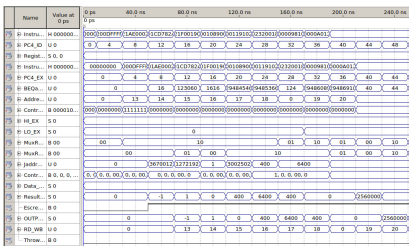

Figura 13. *Waveform* do segundo programa de testes.

6.3. Terceiro Programa

Por fim, apresenta-se o terceiro código, na figura 14.

```
1  li $t1, 100
2  li $t2, 4
3  add $t1, $t1, $t2
4  sub $t1, $t2, $t1
5  and $t3, $t1, $t2
6  andi $s3, $t3, 4
7  or $t4, $t1, $t2
8  ori $s4, $t4, 100
9  nor $t5, $t1,$t2
10 xor $t6, $t1, $t2
11 xori $s1, $t6, 150
12 addu $t7, $t1, $t1
```

Figura 14. Código do terceiro programa de testes.

Seus resultados abaixo podem ser comparados ao *Waveform* na figura 15.

- 1: $9 = 100
- 2: $10 = 4
- 3: $9 = 104
- 4: $9 = -100
- 5: $11 = 4
- 6: $19 = 4
- 7: $12 = -100
- 8: $20 = -4
- 9: $13 = 99
- 10: $14 = -104
- 11: $17 = -242
- 12: $15 = -200

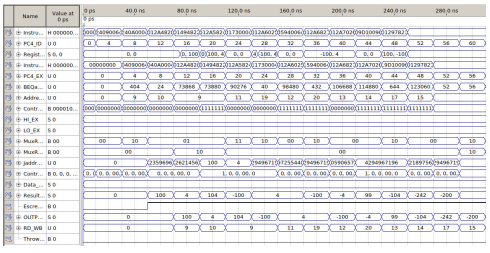

Figura 15. “Waveform” do terceiro programa de testes.

6.4. Quarto Programa

Para que desse para ver cada estágio do *Pipeline* acontecendo ao mesmo tempo, seria feito mais um programa 16 destacando cada um dos estágios na *Waveform* 17. Cada instrução está separada por cores sendo que existe um estágio de *Stall* graças a instrução de *load word* que esta destacada de preto.

```
1  addi $t0, $0, 42
2  sw $t0, 0($0)
3  ori $t1, $t0, 0x0005
4  lw $t2, 0($0)
5  and $t3, $t2, $t1
6  mult $t0, $t1
7  mflo $t4
8  add $0, $t3, $t4
9  add $0, $t3, $t4
10 add $0, $t3, $t4
```

Figura 16. Código do quarto programa de testes.

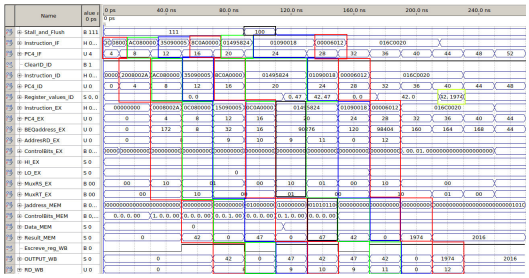

Figura 17. "Waveform" do quarto programa de testes.

6.5. Controle

Para verificar o controle geral e controle da ULA foram inseridos no meio das ondas geradas os bits de controle respectivos a cada etapa do Pipeline, analisando esses valores, conseguimos garantir o controle estava gerando o conjunto de bits correto para todas as instruções pedidas. O controle foi implementado como explicado na seção 4.1.

7. Discussão e Conclusões

A atividade propunha a concepção e o desenvolvimento de um processador voltado à Arquitetura MIPS estruturado em cinco estágios de *pipeline*. Para tal, o grupo deveria fazer uso do Altera Quartus II, software de projeto e simulação de circuitos digitais.

Trabalhando de acordo com as metodologias anteriormente explanadas, concluiu-se o projeto, que suporta todas as instruções previamente solicitadas. Além disso, Unidade de Forwarding e Controle de Hazards estão implementados e funcionando de maneira eficiente.

Ao propor e executar a bateria de testes, o grupo pôde observar o impacto gerado, diretamente no processador, pelas técnicas do programador responsável. Isto é, o efeito causado por negligenciar hazards de dados, principalmente aqueles derivados de uma instrução lw, que não são contornados com *forwarding*.

Notou-se pela realização dos testes que as formas de onda geradas não são um meio muito intuitivo de se verificar a corretude dos resultados fornecidos pelo processador, mas que, ao longo do tempo, os alunos desenvolveram certa afinidade com o modelo gerado e portanto conseguiam analisar mais facilmente o que ocorria em cada estágio do processo de cada instrução.

Uma característica notável do Pipeline observada ao longo da realização do projeto foi sua capacidade de manter o formato em que as instruções são organizadas ao longo da execução, no formato de escada decrescente, garantindo que as instruções estarão sempre próximas no tempo, com uma leve defasagem de uma instrução "nop" quando realmente necessário.

Neste projeto, as instruções jump, 'j', e jump and link, 'jal' foram implementadas como sendo hazards, no entanto, um teste inicial de sua detecção na etapa IF foi realizado, e que acabou comprovando que essa escolha manteria uma melhor vazão de instruções do que manter a sua detecção na etapa MEM. No entanto, como este era um projeto inicial, optou-se por inicialmente dar suporte às instruções de um jeito mais simples, e após sua finalização, iniciar melhorias deste tipo. Esse teste e escolha demonstra outra observação feita frequentemente durante a realização do projeto, o fato de que facilidade de implementação e eficiência muitas vezes se demonstram estar inversamente proporcionais, ou seja, para solucionar um problema de uma maneira um pouco mais eficiente, como no caso dos forwardings, foi necessário incluir muito mais portas lógicas e talvez até alterar o controle para garantir que as instruções sejam realizadas de maneira suave, sem interrupções.

Além do mais, foram notáveis as consequências da delimitação prévia do número de estágios, uma vez que, buscando manter a estrutura proposta, foram necessárias alterações que demandaram certo trabalho. No entanto, foi o preço a se pagar para que, além de tudo, se assegurasse que "simplicidade favorece regularidade". Uma melhoria possível de ser incluida no projeto seria a criação de sub-estágios em algumas etapas durante a execução do processamento, o estágio EX por exemplo está levemente poluído e com muitas portas lógicas, incluindo um sub-estágio garantiríamos que o clock geral dessa etapa poderia ser reduzido, dado que seria necessário apenas o tempo para realizar o cálculo útil na ULA, desconsiderando tempos de propagação de muitas outras portas.

Referências

[1] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.