Escola Politécnica da Universidade de São Paulo

PCS 3612 – Organização e Arquitetura de Computadores I

Projeto: MIPS Pipeline

Aluno:
Guilherme Rodrigues Ludescher 9833180

Curso: Engenhearia de Computação - Quadrimestral

Professora: Cíntia Borges Margi

Relatório do projeto de recuperação
Implementação Pipeline do processador MIPS

06 de Janeiro de 2020

Introdução **2**

Implementação **2**

Base do Pipeline 2

Estágios do Pipeline 4

Instruction Fetch 4

Instruction Decode 5

Execute 6

Memory Access 7

Writeback 8

Registradores de Pipe 8

Pipeline Incrementado 9

Instruções Adicionais 9

branch not equals 9

or immediate 10

Solução de Hazard 11

Forwarding Control 12

Hazard Unit 13

Execução **15**

Simulação 15

Depuração e Correções 16

Análise de formas de onda 16

Análise de schematics 17

Conclusão **18**

Dificuldades Enfrentadas 18

imem e dmem 18

estágios do pipeline 18

netlist view 18

desenvolvimento presencial 19

Resultados 19

Limitações do Projeto 19

Produto Final 20

Introdução

O MIPS (Microprocessor without Interlocked Pipelined Stages) é um microprocessador do tipo RISC desenvolvido pela MIPS Computer Systems utilizado em muitos sistemas embarcados como roteadores e já foi utilizado até em jogos eletrônicos como PS2 e N64. O MIPS Pipeline consiste em uma divisão de 5 estágios de execução (IF, ID, EX, MEM, WB) que serão abordados em breve. A implementação do Pipeline é feita de forma a permitir que instruções diferentes ocorram simultaneamente, ou em paralelo, aumento a capacidade do processador.

Implementação

O projeto tomou como base a implementação já existente do processador MIPS de ciclo único[1], disponibilizada pela professora. Assim, toda o desenvolvimento do MIPS Pipeline foi feito com adições e modificações realizadas ao código já existente.

O desenvolvimento foi feito seguindo uma lógica incremental, ou de forma radial. Com isso, tendo-se o MIPS de ciclo único, primeiramente foram implementados os elementos básicos para a aplicação do pipeline. Em sequência, foram aplicadas as modificações necessárias para abrigar a unidade de detecção de *hazard*, ou Hazard Unit (HU) e o *forwarding*.

Além disso, a estruturação do código seguiu os diagramas e sequências apresentados no livro-texto[2] da bibliografia da disciplina, mantendo a maior parte dos nomes de sinais e variáveis para facilitar o referenciamento com o texto-fonte.

Vale ressaltar que o projeto não tomou como base a entrega da Atividade 4, sendo uma implementação totalmente nova. O projeto foi realizado em conjunto com o colega de turma e de grupo da Atividade 4, Matheus da Silva Sato, número USP 9833200.

Base do Pipeline

O desenvolvimento se iniciou implementando-se a base do Pipeline de acordo com o diagrama abaixo, disponível na página 414 do livro-texto da disciplina.

---

[1] Arquivo .vhd do MIPS ciclo único disponível em:
https://github.com/guiludescher/mips_pipeline/blob/master/mipssingle.vhd

[2]David M. Harris, Sarah L. Harris - Digital Design and Computer Architecture,-Morgan Kaufmann (2012)

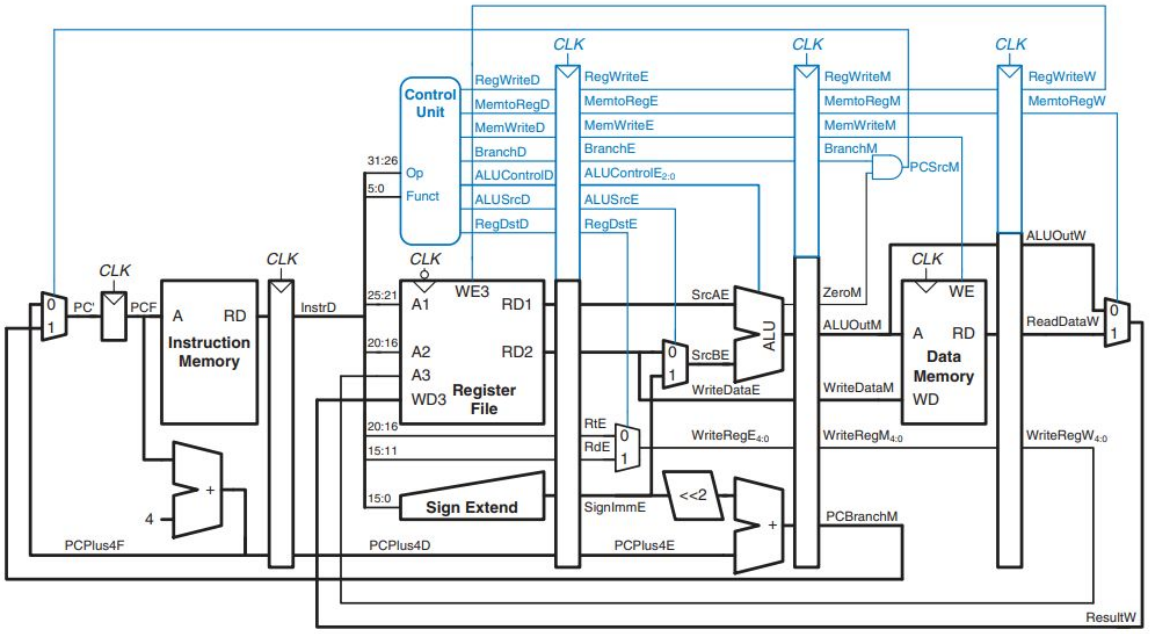

Imagem 1 - Diagrama MIPS *pipeline* sem Hazard Unit

Ao ser finalizada a implementação dos elementos necessários para a base do pipeline, foram adicionados componentes e sinais que permitiriam a atuação de uma Hazard Unit, que será descrita posteriormente.

Assim, a implementação ficou semelhante ao que se vê no diagrama abaixo.

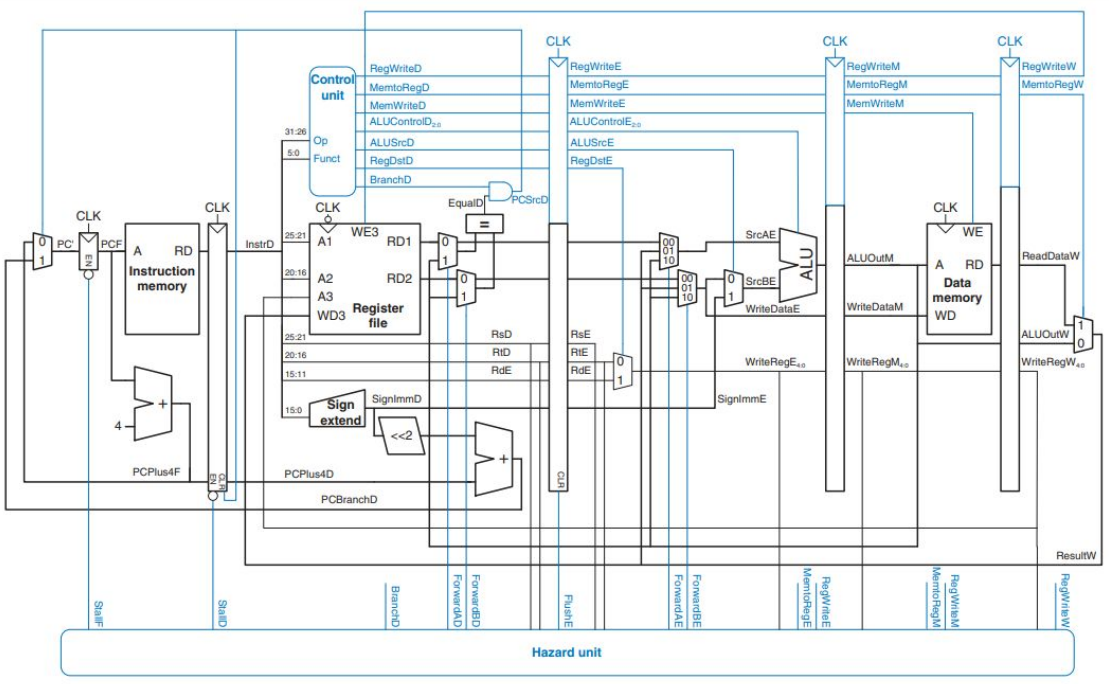

Figure 7.58 Pipelined processor with full hazard handling

Imagem 2 - Diagrama MIPS *pipeline* com Hazard Unit

Pode-se também obter uma visão geral do projeto com o RTL View abaixo, no qual podemos ver a entidade mips encapsulando a unidade de controle e o fluxo de dados.

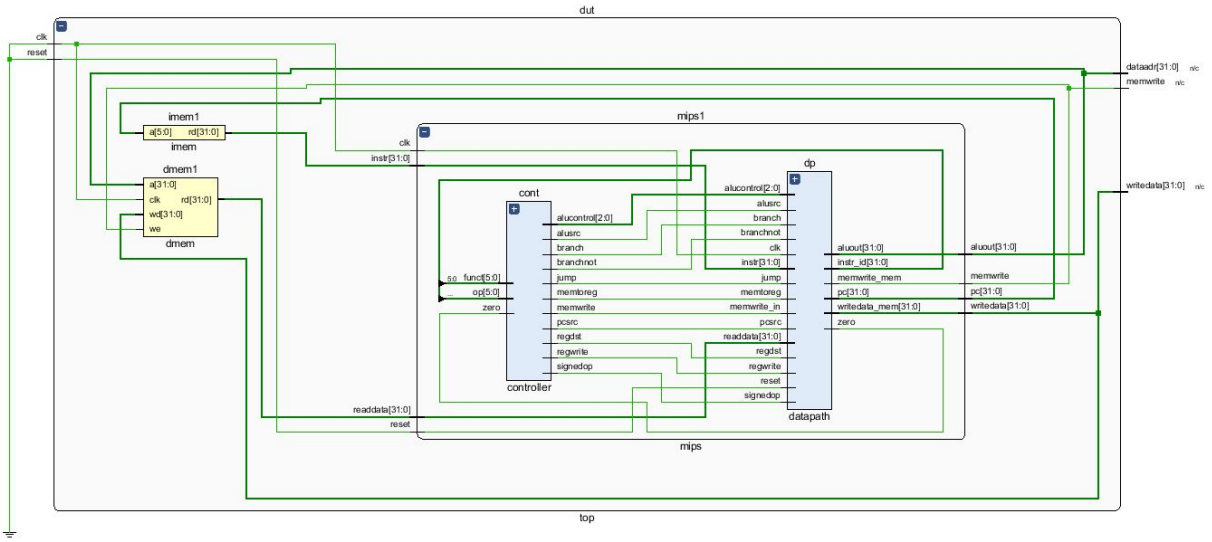

Imagem 3 - RTL View: Visão geral do MIPS *pipeline*

Estágios do Pipeline

Para implementar a estrutura desejada, foi necessário implementar os 5 estágios do *pipeline* como parte do fluxo de dados. Usou-se, então, setores no código para melhor identificar os estágios. Os estágios são: *instruction fetch* (**IF**), *instruction decode* (**ID**), *execute* (**EX**), *memory access* (**MEM**) e *write back* (**WB**).

É válido notar que, no código, os sinais relativos a cada um dos estágios foram referenciados com o uso de um sufixo (F, D, E, M e W, respectivamente) para indicar mais facilmente a qual estágio pertenciam.

Instruction Fetch

Este estágio é o primeiro do *pipeline* e é responsável por buscar a instrução que deve ser executada na memória de instrução (**imem**). Para isso, é necessário calcular o próximo *program count* (**PC**) somando-se 4 ao último realizado. Soma-se 4 pelo fato de que todas as instruções tem 32 bits = 4 bytes de comprimento.

Além disso, é preciso saber se houve uma ramificação (*branch*), ou redirecionamento direto (*jump*). Então, sabendo-se exatamente qual PC utilizar, basta passá-lo ao componente externo imem, que terá como saída a instrução respectiva.

Assim, são necessários um somador para incrementar o PC e multiplexadores (**MUX**) para tomar as decisões sobre qual PC utilizar. Os sinais que

escolhem as saídas dos MUX são originados na unidade de controle (**UC**). Como o PC é retroalimentado, é necessário um registrador simples que permita a passagem do sinal somente com o *clock*.

```
   -- ---- instruction_fetch ----
 reg_PC: flopr generic map(32) port map(clk, reset, pcnext, pc);

   mux_branch: mux2 generic map(32) port map(pcplus4, pc_branch, pcsrc,
pcnextbr);

   mux_jump: mux2 generic map(32) port map(pcnextbr, x"00000044", jump,
pcnext_not_stall);

    mux_stall: mux2 generic map(32) port map(pcnext_not_stall, pc,
s_stall, pcnext);

   add4_PC: adder port map(pc, X"00000004", pcplus4);

   id_flush <= (pcsrc or jump) and not s_stall;

  ifid: if_id port map(clk, reset, id_flush, pcplus4, instr, s_stall,
pcplus4_id, s_instr_id);
```

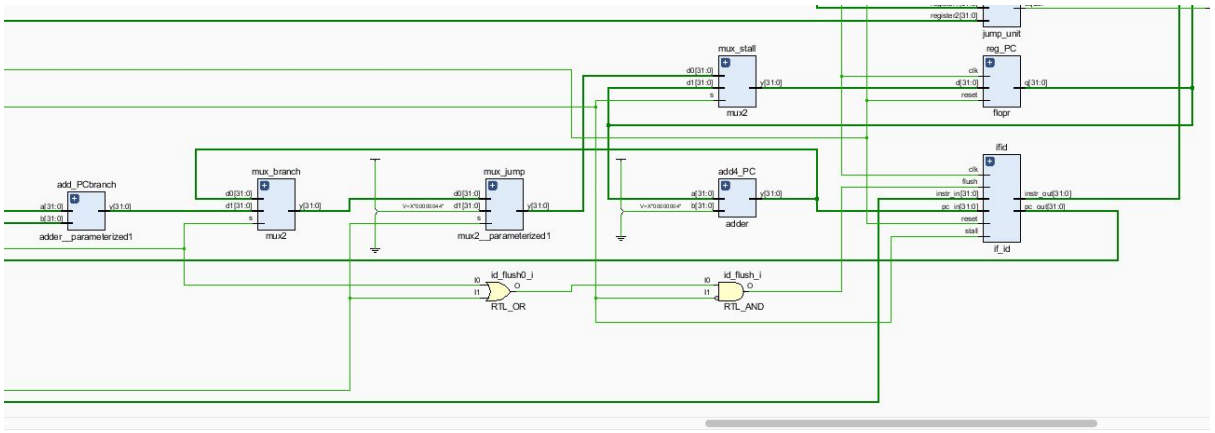

Imagem 4 - RTL View: Estágio Instruction Fetch do *pipeline*

Instruction Decode

Neste estágio, a instrução coletada é decodificada utilizando-se o arquivo de registradores (**regfile**). São entradas do regfile os endereços dos registradores com os quais a instrução será realizada e a saída são os valores presentes nesses

registradores referenciados. Neste momento, também são passados à UC os valores de opcode e funct, que identificam exatamente a instrução.

Os 32 bits de uma instrução são normalmente divididos da seguinte forma:

opcode	register source (rs)	register target (rt)	register destination (rd)	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
31:26	25:21	20:16	15:11	10:6	5:0

Entretanto, para instruções com operandos imediatos, por exemplo, os últimos 16 bits são utilizados para esses operandos. Ainda assim, é necessário que estes tenham 32 bits para realizar as operações do fluxo de dados. Por isso, aplica-se o extensor de sinal (**signext**) aos últimos 16 bits da instrução. Esse componente também é utilizado para determinar o PC da próxima instrução caso tenha havido *branch*.

```
   -- ---- instruction_decode ----
   pcjump <= pcplus4_id(31 downto 28) & s_instr_id(25 downto 0) & "00";
     sign_ext:  signext  port  map(s_instr_id(15  downto  0),  signedop,
signimm);

   shift_imm: sl2 port map(signimm, signimmsh);

 add_PCbranch: adder port map(pcplus4_id, signimmsh, pc_branch);

   reg_file: regfile port map(clk, wb_wb(1), s_instr_id(25 downto 21),
s_instr_id(20 downto 16), writereg_wb, result, srca, writedata);

   ju: jump_unit port map(srca, writedata, zero);

   instr_id <= s_instr_id;
```

Execute

O terceiro estágio do *pipeline* é o que de fato executa a instrução, realizando operações matemáticas entre os valores presentes nos registradores acionados na instrução.

Para executar as operações, está presente a Unidade Lógica Aritmética (**ULA**), cujo sinal de controle vem da UC. As entradas da ULA são sempre um dos

registradores e a saída de um MUX, que decide entre a saída do outro registrador ou o sinal imediato, que é saída do signext no estágio ID.

```
   -- ---- execute ----
 wrmux:  mux2  generic  map(5)  port  map(writereg0_ex,  writereg1_ex,
regdstE, writereg_ex);

    mux_forw1:  mux3  generic  map(32)  port  map  (srca_ex,  aluout_mem,
result, selfwdmux1, ulain0);

  mux_forw2: mux3 generic map(32) port map (writedata_ex, aluout_mem,
result, selfwdmux2, srcb);

  mux_ALU: mux2 generic map(32) port map (srcb, signimm_ex, alusrcE,
ulain1);

   ULA_main: alu port map
     (
       a => ulain0,
       b => ulain1,
       alucontrol => alucontrolE,
       result => aluout_ex,
       zero => open
     );
```

Memory Access

O estágio de Memory Access é responsável por ler ou escrever na memória de dados (**dmem**) sempre que necessário. Como o componente dmem foi implementado externo ao fluxo de dados, basta passar a esse componente os sinais adequados.

Esses sinais são o sinal de controle que vem da UC e indica se há uma escrita, a saída da ALU no estágio EXE - indica o endereço de escrita ou leitura - e o valor que deve ser escrito, quando houver.

```
  -- ---- memory_access ----
  aluout <= aluout_mem;
  memwrite_mem <= m_mem(0);
```

Writeback

O último estágio do *pipeline* tem como função atualizar o regfile com as alterações que tenham ocorrido durante a execução da instrução. Para isso, um sinal de controle que funciona como *write enable* no regfile é passado a esse componente, juntamente com o resultado da operação do *pipeline* e o endereço do registrador a ser atualizado.

O resultado da operação é a saída de um MUX que decide entre a saída da ULA e a saída do *read data* do componente dmem. Ou seja, o resultado será o dado que acabou de ser escrito ou lido na memória, de acordo com a instrução atual.

```
   -- ---- write_back ----
   mux_result:  mux2  generic  map(32)  port  map(aluout_wb,  readdata_wb,
wb_wb(0), result);
```

Registradores de Pipe

Outro elemento importante da implementação *pipeline* é o Registrador de Pipe. Para que as informações não sejam perdidas entre um estágio e outro da execução, é necessário que haja um registrador intermediário, que recebe os dados de um estágio e os transmite ao próximo estágio.

Assim, foi necessário adicionar um registrador entre cada um dos estágios do *pipeline*, totalizando 4 Registradores de Pipe. Esses registradores são simplesmente componentes que “seguram” os dados por um ciclo de *clock* antes de transferi-los para o próximo estágio do *pipeline*, garantindo assim que cada sinal seja lido ou alterado no momento correto.

Posteriormente, durante a implementação da Hazard Unit, sinais de *flush* ou *stall* foram adicionados a alguns desses registradores para fazer o controle dos dados. O *flush* atua como *clear*, tornando “zero” todos os sinais que são passados adiante. Já o *stall* atua como *not enable*, impedindo a passagem dos dados enquanto estiver ativo.

Os registradores são nomeados de acordo com suas posições no pipeline:

- **if_id:** entre *instruction fetch* e *instruction decode*;
- **id_ex:** entre *instruction decode* e *execute*;
- **ex_mem:** entre *execute* e *memory access*;
- **mem_wb:** entre *memory access* e *writeback*.

Pipeline Incrementado

Instruções Adicionais

Foi necessária a implementação de instruções adicionais para que o projeto se adequasse aos requerimentos. As novas instruções implementadas foram *branch not equals* (**bne**) e *or immediate* (**ori**). Então, foram criados sinais no decodificador principal (**maindec**), componente da UC, para lidar com essas novas instruções.

```
      architecture behave of maindec is
   signal controls: STD_LOGIC_VECTOR(10 downto 0); -- Added extra bit
(signedop)
 begin
   process(all) begin
     case op is
       when "000000" => controls <= "10110000010"; -- RTYPE
       when "100011" => controls <= "10101001000"; -- LW
       when "101011" => controls <= "10001010000"; -- SW
       when "000100" => controls <= "10000100001"; -- BEQ
       when "001000" => controls <= "10101000000"; -- ADDI
       when "000010" => controls <= "10000000100"; -- J
         when "001101" => controls <= "00101000011"; -- ORI   -- aluop
altered -- CORRECAO DE REGDST
       when "000101" => controls <= "11000000001"; -- BNE aqui
       when others   => controls <= "-----------"; -- illegal op
     end case;
   end process;

   (signedop, branchnot, regwrite, regdst,
   alusrc, branch, memwrite, memtoreg, jump) <= controls(10 downto 2);

   aluop <= controls(1 downto 0);

 end;
```

branch not equals

Essa instrução é bem parecida com a instrução *branch equals* (**beq**), já implementada no MIPS ciclo único que serviu como base para o projeto. A instrução

bne compara os valores de dois registradores e, caso sejam diferentes, a execução do programa deve ser ramificada, ou seja, criada uma *branch*. A execução então continua a partir do ponto definido na instrução.

Exemplo[3]:

```
...
bne $t3, $0, here
...
here: sub $t2, $t2, $t0
```

Neste caso, o valor de `$t3` é diferente de `$0`, fazendo com que a *branch* seja tomada. O próximo PC então deve ser o referente à instrução presente no endereço `here`.

Para implementar a instrução, foi necessário adicionar uma verificação na UC, onde já era verificada a ocorrência da instrução de *branch*. Utiliza-se então o sinal de *branchnot*, criado no maindec para esta instrução. O sinal resultante é importante para decidir qual PC utilizar: o de *branch* ou o simplesmente acrescido de 4.

```
-- branch op
pcsrc <= (branch and zero) or (branchnot and not zero);
branch <= s_branch;
branchnot <= s_branchnot;
```

or immediate

Essa instrução realiza uma operação “or *bitwise*” de um valor de um registrador com um operando imediato. Ou seja, cada bit do valor vindo do registrador passa por um “or” lógico com o operando imediato da instrução.

Exemplo:

```
ori $t0, $0, 0x8000
```

[3]Exemplos baseados no programa “test2.asm”, disponível em: https://github.com/guiludescher/mips_pipeline/blob/master/instrucoes_para_teste/test2.asm

Aqui, o registrador `$t0` receberá o resultado da operação “or *bitwise*” entre `$0` e `0x8000`, o que equivale a atribuir a `$t0` o valor `0x8000`. Como 0x8000 é um operando imediato, precisa passar pelo extensor de sinal. Entretanto, esse operando é do tipo *unsigned*. Por isso, precisou-se criar um sinal no maindec, o *signedop*, que é ‘0’ nesta instrução.

Com esse sinal, alterou-se a arquitetura do signext, que agora tem a seguinte forma:

```
architecture behave of signext is
begin
-- Adding unsigned operation
  y <= X"ffff" & a when (a(15) AND signedop) else X"0000" & a;
end;
```

Na forma de onda a seguir, pode-se ver a instrução ori em ação. Os sinais exibidos são, em ordem, o *clock*; a instrução que sai do estágio IF; a saída do signext; o resultado da operação como saída do estágio WB; e o sinal de *writedata* que é entrada do *regfile*.

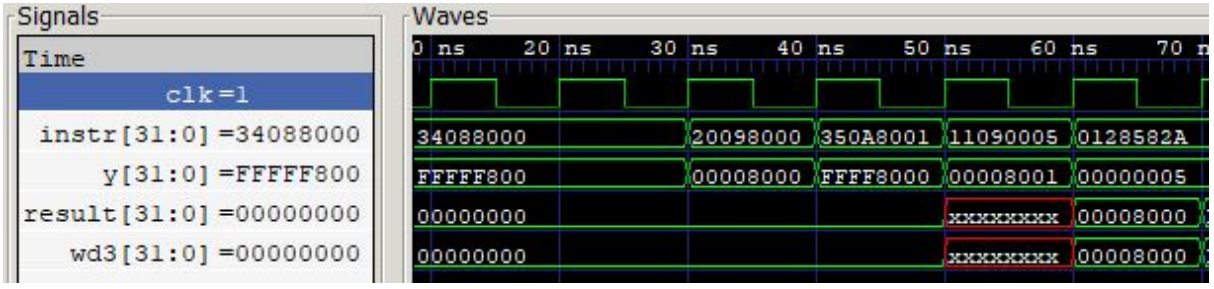

Imagem 5 - gtkwave: Formas de onda evidenciando execução da instrução ‘ori’

É possível ver que a primeira instrução (34088000) é do tipo ori e, em seguida, a saída do signext assume `0x00008000`. Alguns ciclos depois, o resultado do WB é retornado para ser escrito no registrador `$t0` como `0x00008000`.

Solução de Hazard

Com a implementação do *pipeline*, inserem-se também no processador alguns *hazards*, que devem ser solucionados para que as instruções sejam executadas corretamente. Para isso, foram implementados o *Forwarding Control* e a *Hazard Unit*.

Forwarding Control

Este componente é responsável por resolver os *data hazards*. O *forwarding*, ou encaminhamento, é necessário quando o registrador do qual uma instrução pegará os dados ainda não foi escrito - mas precisa ser - por uma instrução anterior.

Assim, os valores, que já foram calculados, são encaminhados do estágio MEM para o EXE. Componentes MUX no estágio EXE são responsáveis por escolher o valor encaminhado ou o valor regular do fluxo. Os sinais que controlam esses MUX são as saídas da unidade de *forwarding*.

```
architecture control of forwarding_control is

 signal selA , selB : std_logic_vector(2 downto 0);
 signal equal, rs_not_zero, rs_eq_rt : std_logic;
begin

  process(all) begin
    -- conditions for alu first source
    if read_register1E = write_registerM and regwriteM = '1' then
      forward_a <= "01";
    elsif read_register1E = write_registerW and regwriteW = '1' then
      forward_a <= "10";
    else forward_a <= "00";
    end if;
    -- conditions for alu second source
    if read_register2E = write_registerM and regwriteM = '1' then
      forward_b <= "01";
    elsif read_register2E = write_registerW and regwriteW = '1' then
      forward_b <= "10";
    else forward_b <= "00";
    end if;
  end process;
end;
```

Pode-se ver na imagem a seguir o RTL View que demonstra a ligação da unidade de *forwarding* com o estágio EXE do *pipeline.*

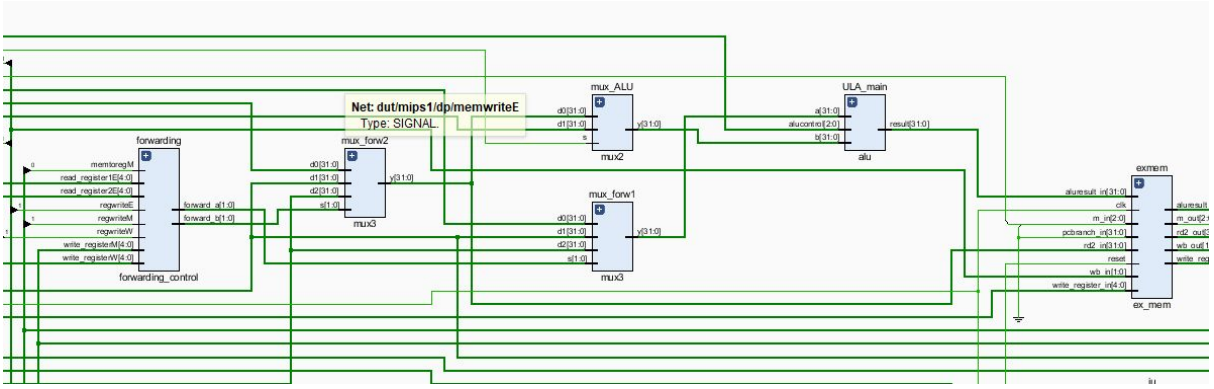

Imagem 6 - RTL View: Estágio Instruction Fetch do *pipeline*

Hazard Unit

A *Hazard Unit* tem como função impedir os *data* e *control hazards* com a introdução de *stalls* e *flushes* na execução do *pipeline.*

Assim, a unidade de *hazard* deve inserir um *stall* quando alguma informação necessária para uma instrução ainda não foi calculada - *data hazard*. Então, essa instrução deve aguardar o término da instrução anterior que está calculando esse valor para prosseguir.

Há também os casos de *control hazard*, quando a decisão de uma *branch* ainda não foi tomada no momento de realizar o *fetch* na próxima instrução. Para resolver isso, a *branch* é calculada anteriormente no *pipeline* e o *stall* passa a depender também do sinal de *branch*.

Neste caso, o *stall* age como um *not enable* nos registradores de *pipe* e, quando há esse *stall*, deve haver também o *flush* nos registradores, agindo como *clear* e evitando que dados indevidos prossigam no fluxo de dados.

```
architecture arch_hazard_unit of hazard_unit is
  -- stall for each condition
    signal branchStall, lwStall, swStall : STD_LOGIC;
     -- D = decode, E = ex, M = mem, W = wb. rs and rt are read regs,
rd is read.
    signal rsD, rtD, rdE, rdM, rdW : STD_LOGIC_VECTOR(4 downto 0);
    -- branch signal
    signal branchD : STD_LOGIC;
begin
    rsD <= ID_EX_rd1;
    rdE <= ID_EX_writeReg;
    rdM <= EX_MEM_writeReg;
    rdW <= MEM_WB_writeReg;
```

```
    branchD <= (ID_EX_branch or ID_EX_branchnot or ID_EX_jump);

    stall <= lwStall or branchStall;

    process (all) begin
             if regdst = '1' or (regdst='0' and alusrc = '0') or
ID_EX_memwrite = '1' then -- there is no second operand
            rtD <= ID_EX_rd2;
        else rtD <= ID_EX_rd1;
        end if;

        -- stall logic for branch
         if branchD = '1' and ID_EX_regWrite ='1' and (rdE = rsD or rdE
= rtD) then
          branchStall <= '1';
         elsif branchD = '1' and EX_MEM_regWrite ='1' and (rdM = rsD or
rdM = rtD) then
          branchStall <= '1';
        else branchStall <= '0';
        end if;

        -- stall logic for lw
        if EX_MEM_memToReg = '1' and ((rsD = rdE) or (rtD = rdE)) then
            lwStall <= '1';
        else lwStall <= '0';
        end if;
    end process;
 end;
```

Na imagem da forma de onda, pode-se ver o *stall* entrando em ação.

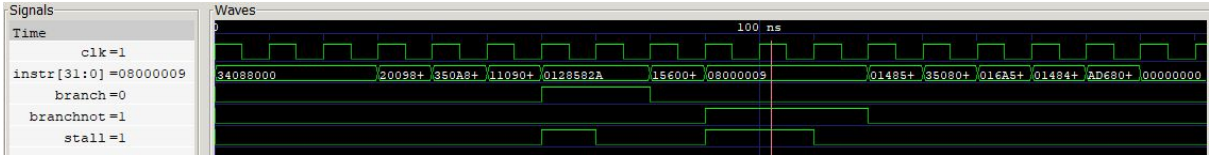

Imagem 7 - gtkwave: Formas de onda evidenciando ativação do sinal '*stall*'

Execução

Simulação

A simulação do circuito foi realizada usando o software ghdl, instalado em uma máquina virtual executando Linux. A máquina virtual foi fornecida pelo Professor Bruno Albertini, juntamente com um tutorial detalhado para sua instalação e utilização[4]. Os comandos utilizados para realizar a análise, elaboração e simulação do projeto foram, nessa ordem, os seguintes:

```
ghdl -a --std=08 --ieee=synopsys mips_pipeline.vhd

ghdl -e --std=08 --ieee=synopsys testbench

ghdl -r --std=08 --ieee=synopsys testbench --vcd=mips_wave.vcd
```

A adição da opção "`--vcd=mips.vcd`" gera um arquivo de saída com formas de onda de todos os sinais do projeto, podendo ser lido por softwares específicos para fins de depuração e análise.

O projeto foi executado com dois arquivos diferentes do tipo ".dat", que servem como entrada das instruções. Isso permitiu testar o projeto contra programas diferentes. As linhas de código que precisam ser alternadas para poder alternar entre os arquivos "memfile.dat" e "memfile2.dat"[5] (gerado a partir do arquivo test2.asm) são as seguintes:

```
    -- if (to_integer(dataadr) = 84 and writedata="00000111") then --
teste com memfile.dat

    if (to_integer(dataadr) = 84 and
writedata="11111111111111110111111100000010") then -- teste com
memfile2.dat
```

e

```
  --FILE_OPEN(mem_file, "memfile.dat", READ_MODE);

  FILE_OPEN(mem_file, "memfile2.dat", READ_MODE);
```

[4]Máquina virtual e tutorial disponíveis em: https://balbertini.github.io/
[5]Arquivos de saída da execução .vcd e .txt disponíveis em: https://github.com/guiludescher/mips_pipeline/tree/master/saidas_simulacao

Vale notar que o primeiro desses trechos de código indica a condição de sucesso da simulação para cada um dos programas.

As formas de ondas[6] geradas pela execução do código com esses arquivos podem ser vistas abaixo, de acordo com seus estados no final da execução do programa. É possível ver que as saídas *writedata* e *dataadr* estão de acordo com os valores esperados quando o *memwrite* torna-se 1.

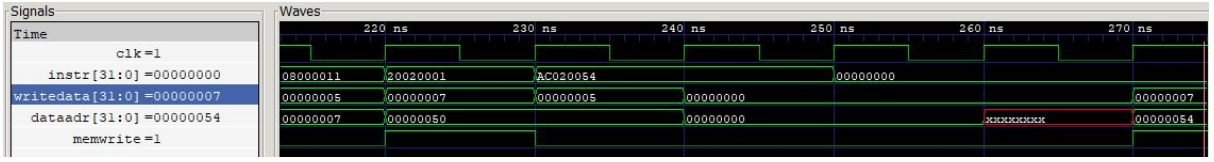

Imagem 8 - gtkwave: Forma de onda com o resultado da simulação com o arquivo “memfile.dat”

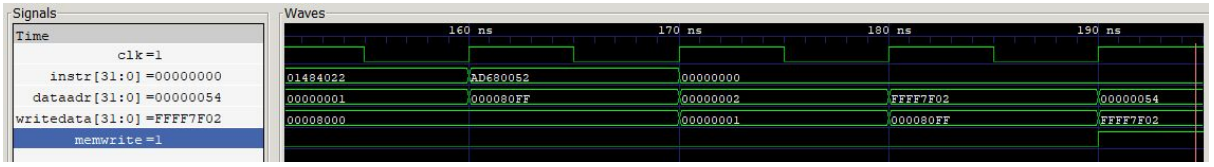

Imagem 9 - gtkwave: Forma de onda com o resultado da simulação com o arquivo “memfile2.dat”

Depuração e Correções

Durante a execução do projeto, em vários momentos, a simulação não ocorreu conforme o esperado. Para encontrar os problemas, foi necessário encontrar meios de depurar o projeto.

Os dois métodos focais de depuração foram a análise de forma de ondas, utilizando-se o software gtkwave e a análise de *schematics* do projeto com um *netlist viewer*, sintetizado pelo software Xilinx Vivado.

Análise de formas de onda

Aqui, vale ressaltar dois momentos notáveis da depuração.

O primeiro deles ocorreu quando percebemos, com o auxílio das formas de onda, que quase todos os sinais do projeto estavam “*undefined*” ou “*don’t care*”. Não havia, a princípio, motivo óbvio para isso. Analisando mais a fundo as ondas e forçando-se alguns valores em sinais específicos, percebemos que havia um problema nos registradores de *pipe,* que não estavam passando a informação adiante no *pipeline*.

[6]memfile.dat, memfile2.dat disponíveis em:
https://github.com/guiludescher/mips_pipeline/tree/master/instrucoes_para_teste

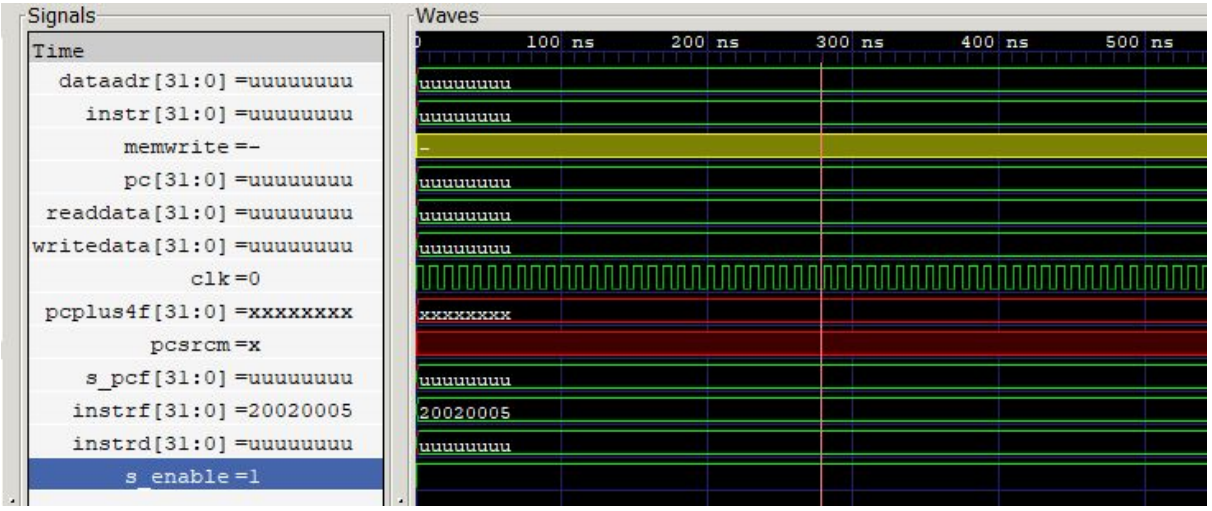

Imagem 10 - gtkwave: Forma de onda com sinais "*undefined*"

O outro momento ocorreu quando o projeto conseguia percorrer as instruções, mas não chegava no final esperado. Percebeu-se então uma decisão incorreta de *branch*, que levava a erros posteriores. Foram então realizadas correções na lógica da instrução de *branch* e no seu controle, levando a melhorias na execução do projeto.

Análise de schematics

A análise de *schematics* gerados pelo *netlist viewer* do Vivado foi importante durante todo o projeto para garantir que os sinais estivessem percorrendo o caminho correto. Com essa ferramenta, foi possível identificar, em vários momentos, sinais conectados incorretamente ou simplesmente não conectados.

Além disso, visualizar os componentes permite tornar um pouco menos abstrato o projeto e ajuda a compreensão geral da funcionalidade e ligação de cada elemento.

Vale notar que foi necessário comentar dois trechos do código[7] - as arquiteturas dos componentes dmem e imem - para que o Vivado conseguisse sintetizar o projeto.

[7]Arquivo pronto para ser sintetizado em projeto do Vivado disponível em: https://github.com/guiludescher/mips_pipeline/blob/master/mips_vivado.vhd

Conclusão

Dificuldades Enfrentadas

O desenvolvimento do projeto enfrentou uma série de dificuldades que podem ser citadas.

imem e dmem

A implementação, inicialmente, seguiu à risca os diagramas disponíveis no livro-texto da matéria, colocando os componentes imem e dmem dentro do fluxo de dados. Entretanto, isso diferia da estrutura do MIPS de ciclo único fornecido e isso gerou incertezas quanto à validade de manter os componentes no fluxo de dados.

O grupo optou então por separar os componentes e manter a estrutura de forma mais similar ao projeto base. Os sinais que eram ligados a esses componentes, até então sinais internos, tornaram-se saídas ou entradas do fluxo de dados.

estágios do pipeline

Inicialmente, optou-se por encapsular cada um dos estágios do pipeline como componentes do fluxo de dados. Assim, haveria a entidade *instruction_fetch*, a entidade *instruction_decode*, e assim por diante.

Contudo, como, novamente, isso diferia da estrutura adotada pelo MIPS de ciclo único que, por sua vez, deixava os componentes internos de cada estágio “soltos” no fluxo de dados, o grupo optou adotar a segunda estratégia. Assim, os componentes foram setorizados no código para facilitar a leitura e compreensão, mas não foram encapsulados em componentes que representassem os estágios do *pipeline*.

netlist view

Como, em outros projetos realizados na universidade, já havia se utilizado o VHDL, já havia algum conhecimento a respeito da linguagem e era de acordo do grupo que a primeira depuração de problemas deveria ser feita usando um *netlist viewer*. Isso possibilitaria identificar sinais não conectados ou conectados incorretamente com facilidade.

A ferramenta para realizar isso que o grupo conhecia era o Intel Quartus Prime. Entretanto, essa ferramenta não é compatível com diversas funcionalidades do VHDL 2008, versão usada no código base fornecido. O grupo tentou adaptar o código, mas, mesmo após inúmeras alterações para adequar a sintaxe, a síntese

ainda não era possível. Percebeu-se então que, dado o número de alterações realizadas, quaisquer mudanças no código original gerariam grandes dificuldades para que as mudanças se propagassem ao projeto do Quartus.

Decidiu-se finalmente por instalar o Xilinx Vivado. Este software, apesar de bastante pesado e de ter necessitado mais de 3 horas totais para instalação, serviu ao seu propósito e ajudou a encontrar diversos pequenos problemas, além de gerar diagramas úteis.

desenvolvimento presencial

A maior parte do projeto foi desenvolvido sem a necessidade de encontro dos membros do grupo. Para isso, foi utilizado o git para controle do código e houve constante e intensa comunicação para alinhamento das tarefas. Entretanto, nos dias finais do projeto viu-se a necessidade de uma reunião para o desenvolvimento, visando facilitar a comunicação e interação.

Como é de praxe dos alunos da universidade, o local escolhido para realizar a atividade foi a Escola Politécnica. Entretanto, como a data da reunião - a única viável - era 03/01, a universidade estava de recesso e todos os prédios estavam fechados, não permitindo a nossa entrada. Ainda buscamos alternativas, procurando na internet informações sobre bibliotecas de outros institutos, mas não foi encontrada nenhuma aberta.

Como precisávamos de tomada, mesa e acesso à internet, fomos até uma cafeteria Starbucks para poder realizar o trabalho. Apesar de não ser um ambiente tão adequado quanto o da universidade, a reunião correu bem e foi produtiva.

Resultados

Limitações do Projeto

O projeto do MIPS *pipeline* desenvolvido apresenta algumas limitações que podem ser evidenciadas. É fácil perceber que nem todas as instruções do *instruction set* padrão do MIPS[8] foram implementadas. Assim, um programa que contivesse uma instrução não prevista não poderia ser executado.

Além disso, por mais que a ideia toda por trás do *pipeline* seja paralelizar a execução de instruções e, aumentando a vazão delas, diminuir também o tempo de execução do programa, a adição de elementos para que isso seja possível causa um *overhead* que pode gerar um aumento no tempo para certos tipos de programas.

[8] *Instruction Set* padrão do MIPS disponível em: https://github.com/guiludescher/mips_pipeline/blob/master/MIPS%20-%20Instruction%20Set.pdf

Outro ponto a ser levantado é a forma como foi implementado o *testbench*, responsável por verificar o fim da execução e validar se as saídas finais estão corretas é bastante específico. Assim, precisa ser alterado diretamente no código cada vez que o programa a ser executado também for.

Produto Final

O produto final do desenvolvimento foi bem sucedido, realizando as simulações com sucesso em ambos os programas teste fornecidos. Entendemos que há pontos que poderiam ser melhorados no projeto, possivelmente tornando-o mais completo e eficiente. Ainda assim, sentimos que o resultado foi muito positivo.