ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Graduação em Engenharia Computação

PCS3612 - Organização e Arquitetura de Computadores I

Professora Cintia Borges

Arthur Pires da Fonseca — **NUSP: 10773096**

Guilherme Elias Setter Bauab — **NUSP: 9900383**

Sergio Ariel Gonzales Fuentes — **NUSP: 10770200**

Introdução

Este é o relatório de implementação do processador ARMv4 na versão pipeline. O projeto foi desenvolvido em VHDL 2008 com versões controladas em um repositório do GitHub. Os testes foram parcialmente realizados na máquina virtual disponibilizada pela disciplina e parcialmente com a ferramenta ModelSim do computador de um dos integrantes do grupo.

1. Implementação

Iniciamos a implementação do ARM pipeline partindo da implementação base de ARM ciclo único fornecida pelo livro, cujos entendimento e funcionamento detalhado estão descritos no planejamento do projeto. O diagrama completo do circuito ARM ciclo único utilizado está apresentado abaixo.

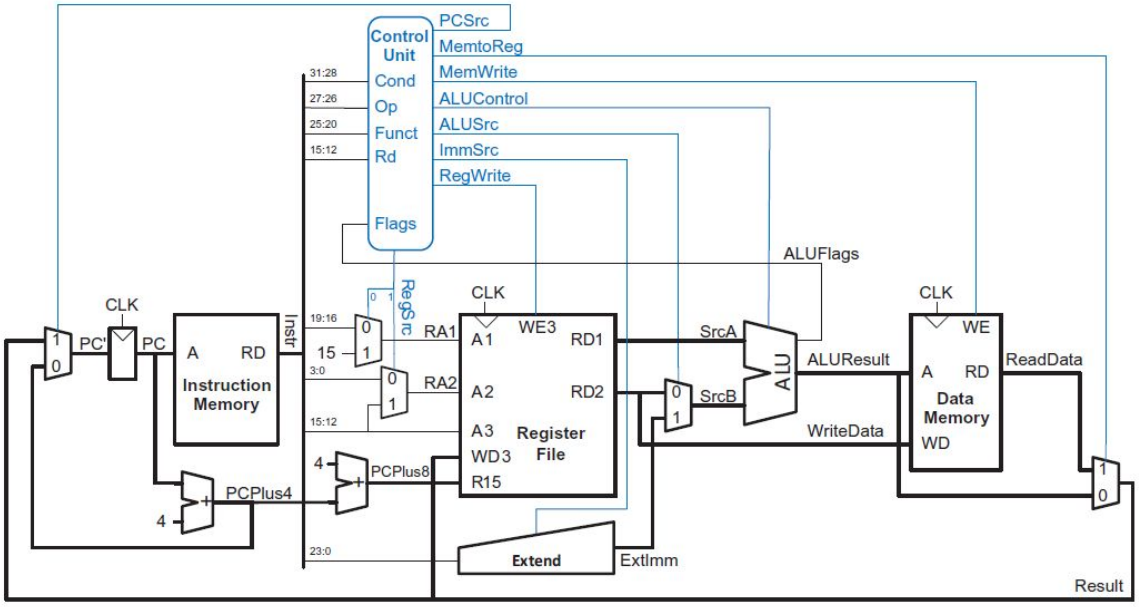


Figura 1 - Diagrama do circuito ARM ciclo único base

1.1. Breve descrição do circuito base

Esse circuito de base é composto por 3 partes principais: fluxo de dados, unidade de controle e memórias. Iremos alterar os componentes existentes e adicionar novos componentes nessas três partes para transformar o processador ARM em pipeline; além de adicionar uma quarta parte, a Hazard Unit, que está descrita em detalhes ao final deste tópico. O diagrama abaixo ilustra a relação entre essas 3 partes principais do circuito base.

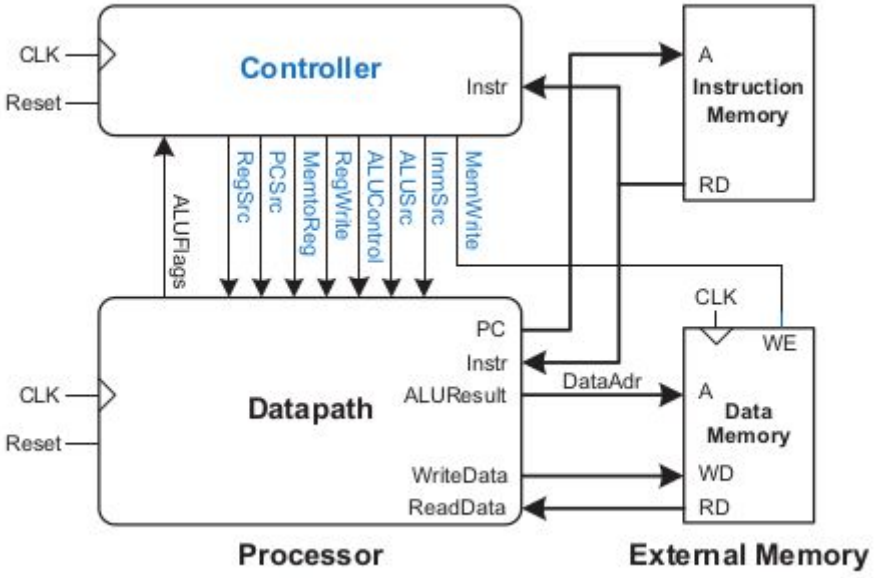


Figura 2 - Diagrama da relação entre fluxo de dados e unidade de controle

O fluxo de dados do circuito é formado por uma entidade ULA responsável pelas operações lógico-aritméticas, uma entidade regfile que é o banco de registradores responsável por armazenar os 16 registradores de 32 bits do circuito, um adder que é responsável por adições no program counter, um flopr que funciona como program counter determinando a posição na memória da instrução executada, um extend que é o componente responsável por fazer um padding nos sinais de tamanho menor que 32 bits que precisam ter esse tamanho e um mux que é responsável por multiplexar diferentes sinais que podem ser usados em uma entrada. O diagrama do fluxo de dados está ilustrado abaixo.

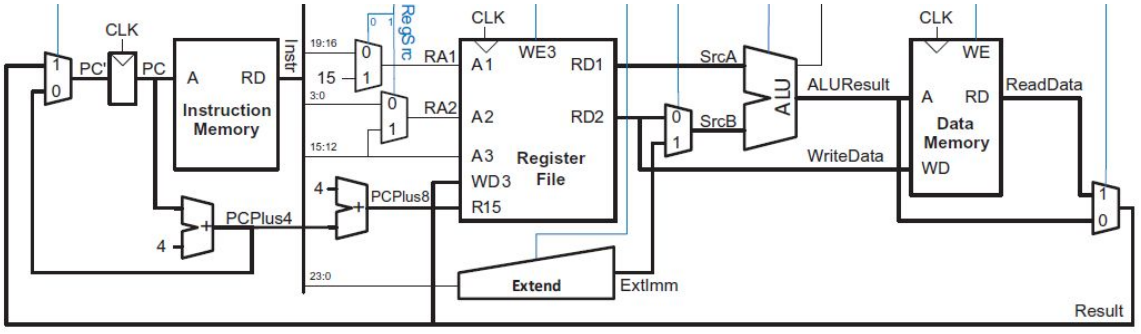


Figura 3 - Diagrama do fluxo de dados do circuito base

A unidade de controle é formada por dois componentes: decoder e condlogic. O decoder é responsável por determinar o tipo de instrução, tipo de operação da ULA e se a próxima instrução é sequencial ou branch. O condlogic determina as condições de execução da operação pelo campo condicional e valor de flag. O diagrama da unidade de controle está ilustrado abaixo.

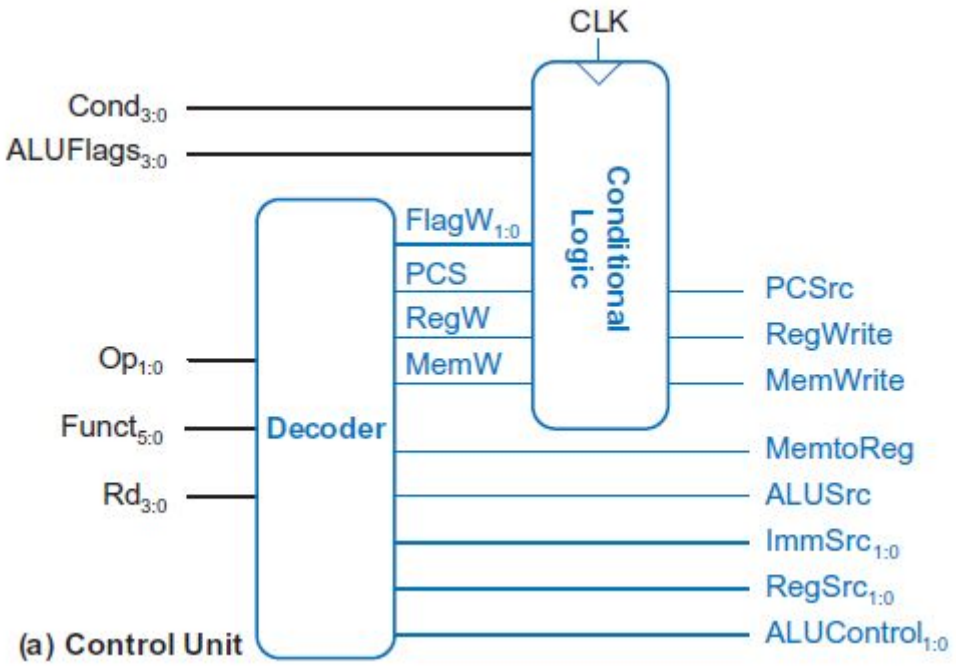


Figura 4 - Diagrama da unidade de controle do circuito base

1.2. Modificações no circuito base

O pipeline que desenvolvemos é dividido em 5 estágios que podem ser executados simultaneamente para computar instruções diferentes. Os estágios são:

1. Fetch - Ler instrução da *instruction memory*
2. Decode - Ler os operandos da *register flle* e decodificar a instrução para produzir os sinais de controle
3. Execute - Performar a computação da ULA
4. Memory - Ler ou escrever dados na memória de dados
5. Writeback - Escrever o resultado no registrador (quando aplicável)

Registradores de pipeline:

A primeira modificação que realizamos foi inserir 4 registradores de pipeline, que dividem o fluxo de dados entre os 5 estágios de pipeline, ilustrado pelo diagrama abaixo.

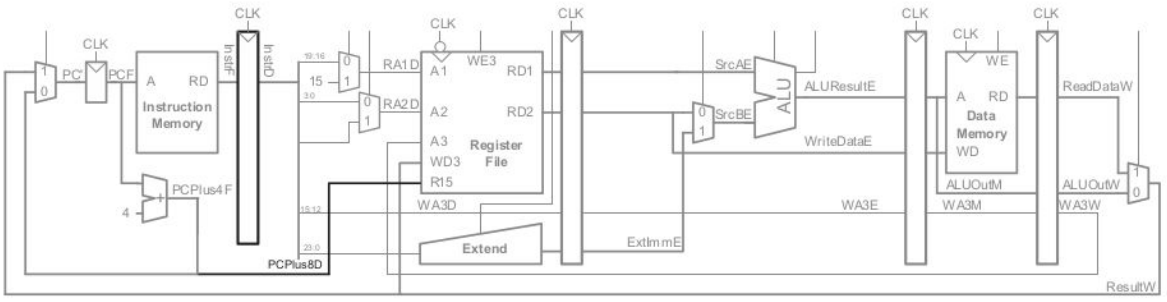


Figura 5 - Diagrama da unidade de controle do circuito base

Os registradores de pipeline têm a função de armazenar resultados parciais do estágio anterior (tanto do fluxo de dados quanto da unidade de controle) de forma que sinais referentes ao estágio atual não interfiram com os seguintes, permitindo, portanto, que instruções diferentes sejam executadas simultaneamente em estágios diferentes.
De forma a possibilitar testes intermediários, decidimos implementar os registradores de pipeline em 3 etapas:

1. Criação dos componentes e testes individuais garantindo o funcionamento perfeito de cada um.
2. Conexão de “caixas vazias” correspondentes aos registradores de pipeline no circuito e testes garantindo o funcionamento do circuito com essas conexões realizadas.
3. Inserção dos componentes desenvolvidos e já testados dentro das “caixas vazias” conectadas.

Como há muitos sinais conectados nos registradores, entendemos que seria bastante possível acontecer algum erro durante a conexão com o resto do sistema. Para evitar isso, decidimos realizar essa segunda etapa que consiste em conectar “caixas vazias” correspondentes aos registradores de pipeline no sistema e nos permite testar as conexões individualmente. Essas “caixas vazias” tem um sinal interno que liga diretamente seu input no output e portanto, se conectadas corretamente, não interferem no funcionamento do circuito original. Dessa forma, pode-se testar a conexão de todos os sinais internos executando o circuito com essas “caixas vazias”, possibilitando o diagnóstico e correção imediata dos erros de conexão.

Realizamos a primeira parte implementando cada um dos registradores de pipeline dentro do fluxo de dados, que os componentes:

1. partial_IF_ID - Entre o estágio Fetch e Decode
2. partial_ID_EX - Entre o estágio Decode e Execute
3. partial_EX_MEM - Entre o estágio Execute e Memory
4. partial_MEM_WB - Entre o estágio Memory e Writeback

Em seguida, testamos cada um dos componentes no ModelSim e confirmamos que todos funcionavam perfeitamente.

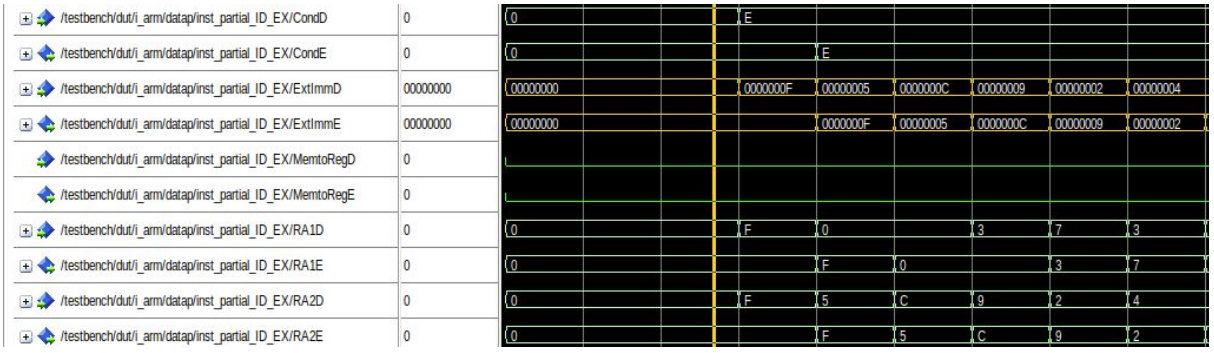


Figura 6 - Waveforms demonstrando o funcionamento dos registradores de pipeline

Em seguida, implementamos os componentes como “caixas vazias” no código de forma a testar as conexões. As “caixas vazias” tem mesmo nome e entidade que os registradores de pipeline implementados, porém sua arquitetura é apenas sinais ligando cada input no output, como visto no código da “caixa vazia” referente ao registrador partial_EX_MEM abaixo:

```
296  entity partial_EX_MEM is
297    port (
298      clock, reset : in std_logic;
299
300      PCSrcE, RegWriteE, MemtoRegE, MemWriteE : in std_logic; -- Sinais combinatorios
301      ALUResultE, WriteDataE : in std_logic_vector(31 downto 0);
302      WA3E : in std_logic_vector(3 downto 0);
303
304      PCSrcM, RegWriteM, MemtoRegM, MemWriteM : out std_logic; -- Sinais combinatorios
305      ALUResultM, WriteDataM : out std_logic_vector(31 downto 0);
306      WA3M : out std_logic_vector(3 downto 0)
307    );
308  end entity;
309
310  architecture arch of partial_EX_MEM is
311
312    signal s_PCSrc, s_RegWrite, s_MemtoReg, s_MemWrite : std_logic; -- Sinais combinatorios
313    signal s_ALUResult, s_WriteData : std_logic_vector(31 downto 0);
314    signal s_WA3 : std_logic_vector(3 downto 0);
315
316  begin
317
318    s_PCSrc <= PCSrcE;
319    s_RegWrite <= RegWriteE;
320    s_MemtoReg <= MemtoRegE;
321    s_MemWrite <= MemWriteE;
322    s_ALUResult <= ALUResultE;
323    s_WriteData <= WriteDataE;
324    s_WA3 <= WA3E;
325
326    PCSrcM <= s_PCSrc;
327    RegWriteM <=  s_RegWrite;
328    MemtoRegM <=  s_MemtoReg;
329    MemWriteM <= s_MemWrite;
330    ALUResultM <= s_ALUResult;
331    WriteDataM <= s_WriteData;
332    WA3M <= s_WA3;
333
334  end architecture;
```

Figura 6 - Entidade e arquitetura de uma das “caixas vazias”.

Testamos o circuito com as caixas vazias para identificar erros nas conexões. Após as devidas correções, o circuito funcionou perfeitamente, indicando que as conexões estavam corretas. A demonstração de funcionamento pode ser vista abaixo:

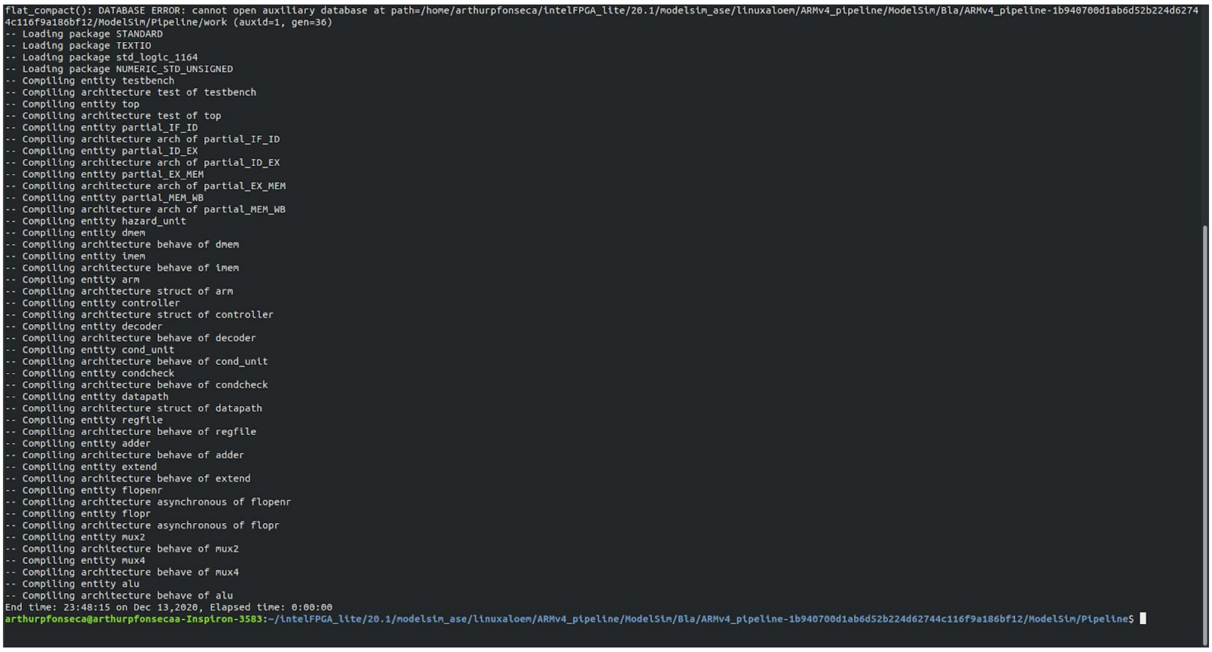


Figura 7 - Demonstração do funcionamento de testbench com as caixas vazias

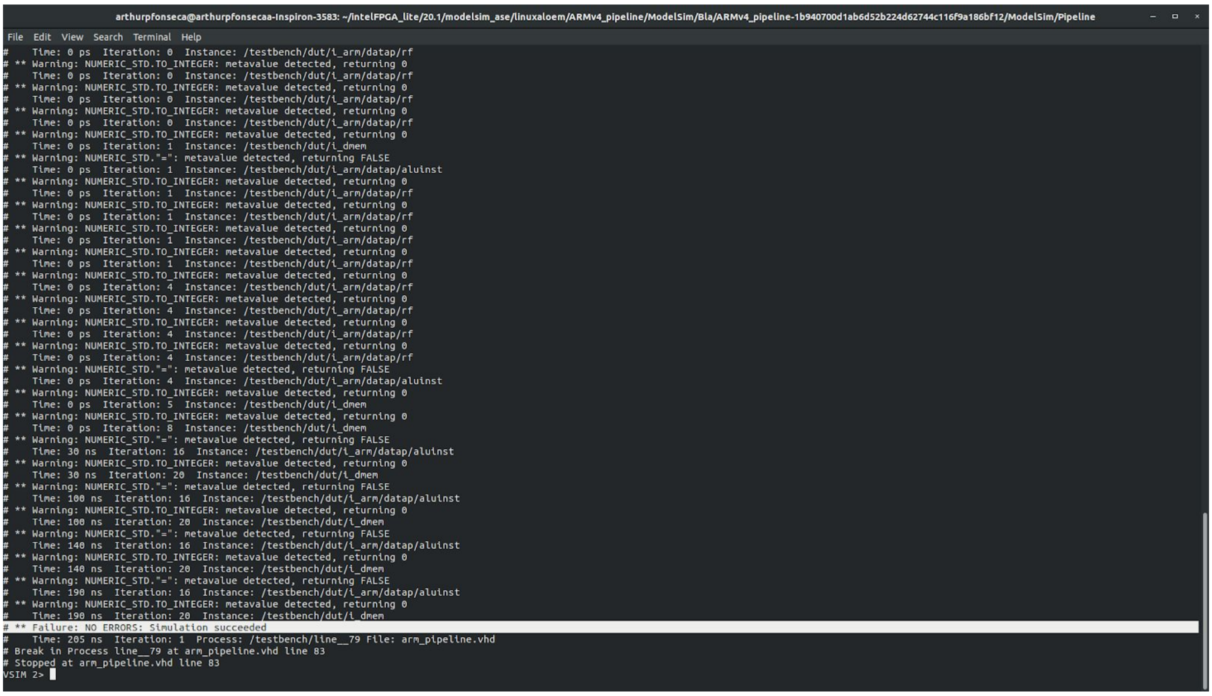


Figura 8 - Demonstração do funcionamento de testbench com as caixas vazias

Em seguida realizamos a terceira parte, inserindo os registradores de pipeline que desenvolvemos na primeira parte, já conectados corretamente pelas “caixas vazias”. Agora sabemos que os registradores de pipeline estão implementados corretamente no circuito, tendo em vista que confirmamos o funcionamento individual e a integração com o sistema de cada um deles.

Modificação no local do componente Conditional logic:

O componente Conditional Logic, que ficava na unidade de controle, agora precisa interagir com vários dos registradores de pipeline. Isso exigiria uma quantidade grande de adições nas saídas e entradas do fluxo de dados e unidade de controle, com sinais da unidade de controle interagindo com sinais e componentes de diferentes estágios de pipeline. Essa implementação é complexa e altamente suscetível a erros. Para evitar complicações desnecessárias, seguimos o exemplo do livro e transferimos esse componente para dentro do fluxo de dados, a partir de um componente novo chamado de “Cond Unit” ilustrado pela caixa vermelha no diagrama abaixo:

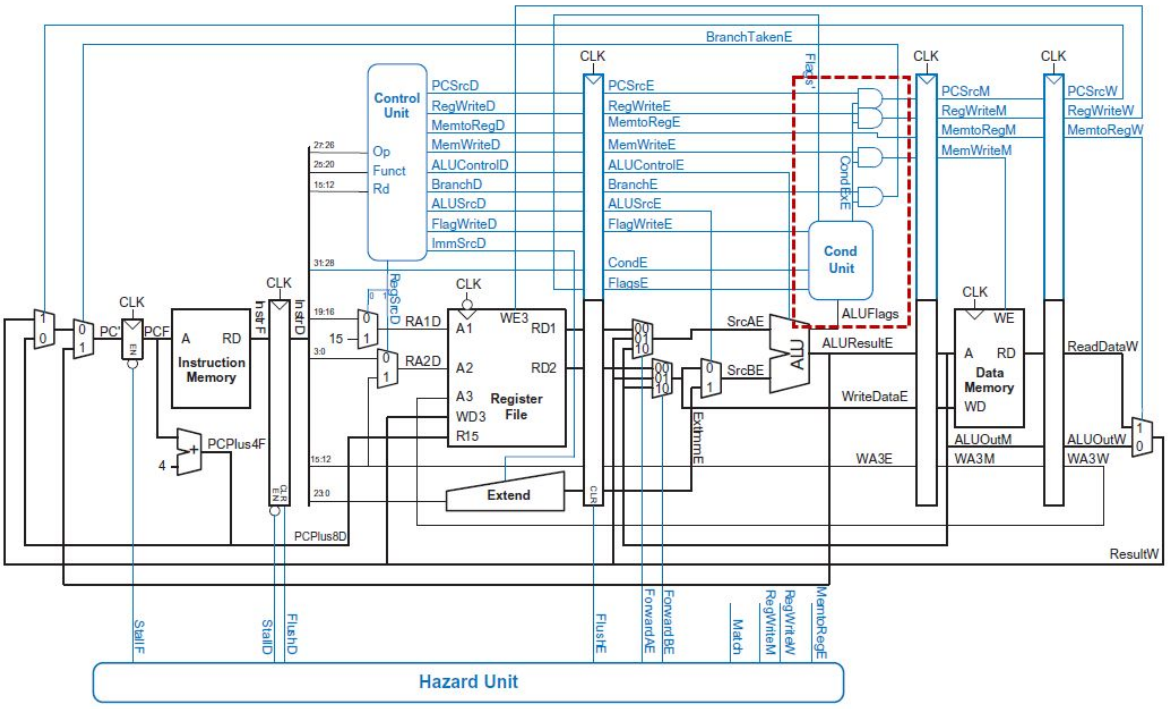


Figura 9 - Componente Cond Unit no fluxo de dados.

Implementamos o componente “cond_unit” baseado no componente Conditional Logic presente na unidade de controle, como visto no segmento VHDL abaixo:

```
845  library IEEE;
846  use IEEE.std_logic_1164.all;
847  entity cond_unit is -- Conditional logic
848    port (
849      clk, reset : in std_logic;
850
851      Cond : in std_logic_vector(3 downto 0);
852      ALUFlags : in std_logic_vector(3 downto 0);
853      FlagW : in std_logic_vector(1 downto 0);
854      PCS, RegW, MemW : in std_logic;
855      FlagsE: in std_logic_vector(3 downto 0);
856
857      Flags: out std_logic_vector(3 downto 0);
858      PCSrc, RegWrite : out std_logic;
859      MemWrite : out std_logic);
860  end;
861
862  architecture behave OF cond_unit is
863    component condcheck
864      port (
865        Cond : in std_logic_vector(3 downto 0);
866        Flags : in std_logic_vector(3 downto 0);
867        CondEx : out std_logic);
868    end component;
869    component flopenr generic (width : integer);
870      port (
871        clk, reset, en : in std_logic;
872        d : in std_logic_vector(width - 1 downto 0);
873        q : out std_logic_vector(width - 1 downto 0));
874    end component;
875
876    signal FlagWrite : std_logic_vector(1 downto 0);
877    --signal Flags : std_logic_vector(3 downto 0);
878    signal CondEx : std_logic;
```

```
879
880  begin
881    flagreg1 : flopenr generic map(2)
882    port map(
883      clk => clk,
884      reset => reset,
885      en => FlagWrite(1),
886      d => ALUFlags(3 downto 2),
887      q => Flags(3 downto 2)
888    );
889
890    flagreg0 : flopenr
891    generic map(width => 2)
892    port map(
893      clk => clk,
894      reset => reset,
895      en => FlagWrite(0),
896      d => ALUFlags(1 downto 0),
897      q => Flags(1 downto 0)
898    );
899
900    cc : condcheck port map(
901      Cond => Cond,
902      Flags => FlagsE,
903      CondEx => CondEx
904    );
905
906    FlagWrite <= FlagW AND (CondEx, CondEx);
907    RegWrite <= RegW AND CondEx;
908    MemWrite <= MemW AND CondEx;
909    PCSrc <= PCS AND CondEx;
910  end;
```

Figura 10

Testamos esse componente comparando com os sinais do componente “Conditional Logic” original. Os dois componentes têm a mesma função e portanto seus sinais internos são iguais. Dessa forma, confirmamos que o componente desenvolvido funciona como esperado executando a simulação no ModelSim e certificando que os sinais de input, output e internos atuam da mesma forma que no componente “conditional logic” do código de base.

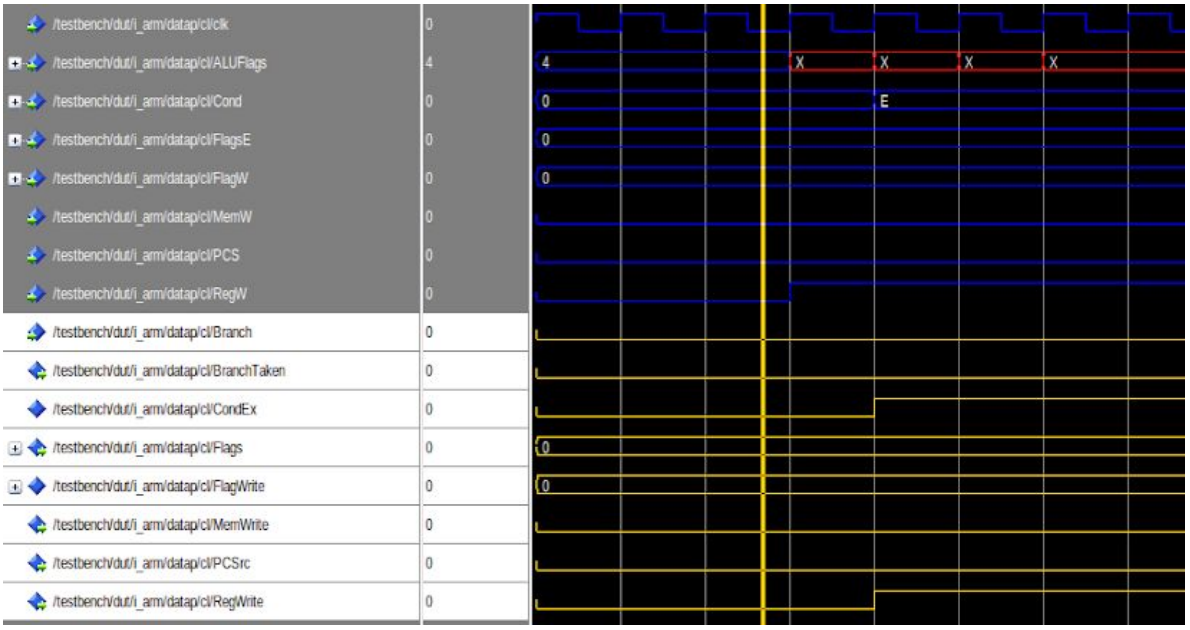


Figura 11 - Waveforms demonstrando o funcionamento do condunit

<u>Otimização sugerida:</u>

Realizamos a otimização sugerida pelo livro para economizar um registrador de pipeline e um somador de 32 bits: ligar a saída do primeiro somador no estágio de fetch diretamente

com a entrada R15 do banco de registradores, sem intermédio do registrador de pipeline e consequentemente do segundo somador. Isso é possível devido ao fato de que o valor que deve estar armazenado no registrador R15 no estágio ID de uma certa instrução é, quando não há um *branch* no *pipeline*, logicamente equivalente ao valor que será armazenado no registrador PC naquele mesmo ciclo (PC + 8 = (PC da instrução seguinte) + 4). A modificação é ilustrada pelo diagrama abaixo.

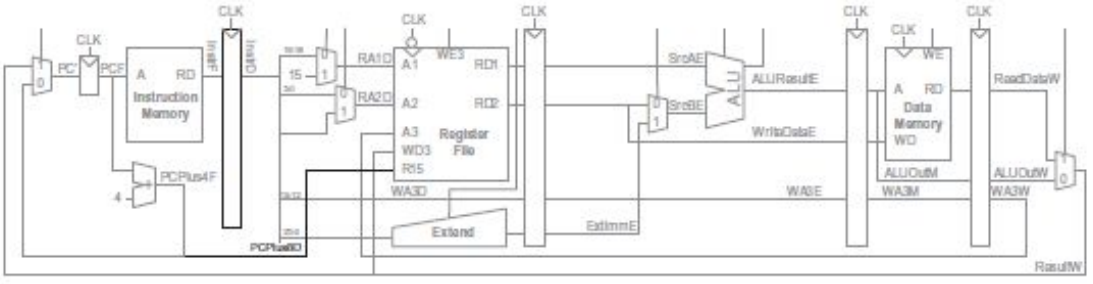


Figura 12 - Diagrama da otimização sugerida pelo livro.

Implementar essa otimização é realmente simples, apenas removemos a entrada e saída correspondentes ao sinal de PC do registrador de pipeline e o segundo somador. Em seguida, conectamos diretamente o output do primeiro somador com a entrada R15 da register file. O código com essa modificação é exibido abaixo, mostrando o portmap da register file recebendo o output do primeiro somador “PCPlus4” no input R15.

```
1292    rf : regfile port map
1293    (
1294      clk => clk,
1295      we3 => RegWriteW, --
1296      ra1 => RA1D,
1297      ra2 => RA2D,
1298      WA3 => WA3W  , -- [MUDAR PIPELINE] VEM DO WA3W
1299      wd3 => ResultW,
1300      r15 => PCPlus4, -- [MUDAR PIPELINE] VEM DO PCPlus4F ou PCPlus8D
1301
1302      rd1 => RD1D,--SrcAE, -- [FUNCIONA]
1303      rd2 => RD2D--WriteData [funciona]
1304    );
```

Figura 13 - Portmap do regfile recebendo o input direto do primeiro somador

Essa modificação foi testada através de uma simulação do circuito no ModelSim evidenciando os sinais internos de PC, saída do primeiro somador e entrada no R15 do regfile. Dessa forma foi possível confirmar que a modificação foi implementada corretamente pois o visualizamos que o input de R15 tem o mesmo valor do output do

primeiro somador e corresponde ao PC+4 da instrução mais recente, que é o mesmo que (PC+8) em relação à instrução anterior. A demonstração está exibida abaixo.

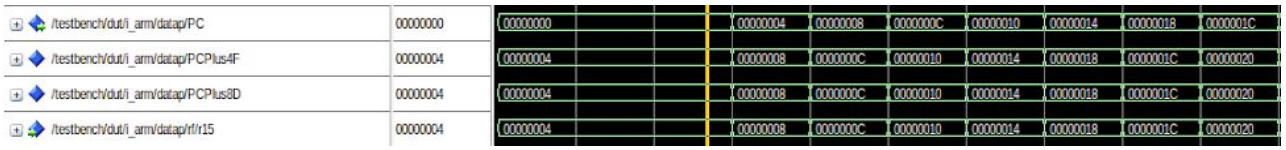


Figura 14

<u>Implementação dos multiplexadores para forwarding:</u>

O forwarding serve para resolver alguns conflitos de dados através de encaminhar um resultado do estágio de Memória ou Writeback para uma instrução no estágio de execução. Para isso, adicionamos dois MUX antes da ULA que selecionam os operandos dentre três sinais possíveis: valor armazenado em registrador (definido na instrução como de origem para operação), estágio de memória ou writeback. Esses dois MUX são controlados pelo sinal de controle ForwardAE e ForwardBE gerados pela hazard unit. O diagrama abaixo ilustra os Mux implementados assim como os sinais de controle provindos da hazard unit.

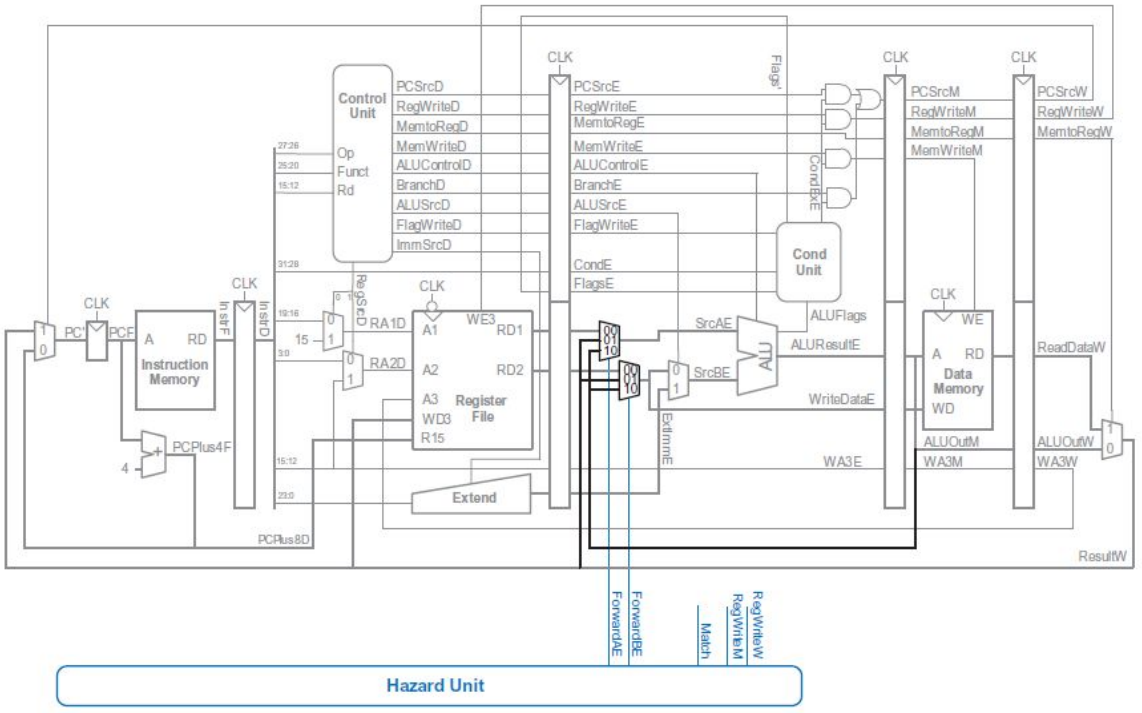


Figura 15 - Mux de encaminhamento e sinais da hazard unit

Como esses Mux comportam 3 entradas, implementamos um novo componente mux de 2 bits. Esse componente recebe até quatro sinais de entrada de tamanho genérico e seleciona uma dessas para a saída de acordo com o sinal de controle de 2 bits. O código do componente implementado está exibido abaixo.

```
1640  library IEEE;
1641  use IEEE.std_logic_1164.all;
1642  entity mux4 is --four-input multiplexer
1643    generic (width : integer);
1644    port (
1645      d0, d1, d2, d3 : in std_logic_vector(width - 1 downto 0);
1646      s : in std_logic_vector(1 downto 0);
1647      y : out std_logic_vector(width - 1 downto 0));
1648  end;
1649
1650  architecture behave OF mux4 is
1651  begin
1652    with s select
1653      y <=
1654        d0 when "00",
1655        d1 when "01",
1656        d2 when "10",
1657        d3 when "11",
1658        d0 when others;
1659
1660  end;
```

Figura 16 - Código do Mux de 4 bits implementado

Testamos o componente desenvolvido pelo ModelSim e confirmamos que funciona corretamente. Em seguida, implementamos os dois Mux no circuito e testamos a implementação como um todo, observando os sinais de input e output dos dois novos componentes. Dessa forma, confirmamos que estão funcionando corretamente e as conexões estão corretas.

Implementação do Mux para branch:

O processador desenvolvido utiliza o mecanismo de previsão nottaken na execução de instruções, ou seja, sempre supõe que não ocorrerá desvio. Esse mecanismo erra quando encontra uma instrução de desvio que deverá ser tomado. Neste caso, o processador faz um flush em todas as instruções que tinham começado a ser executadas depois do branch.

Esse procedimento foi implementado através de um Mux antes do registrador PC, que determina se o próximo endereço será sequencial (PC+4) ou um endereço calculado no estágio WB. Adicionalmente, adicionamos um enable no registrador de PC e no registrador de pipeline referente aos estágios fetch-decode e um reset nos registradores de pipeline dos estágios fetch-decode e decode-execute. O mux utilizado foi uma instanciação do componente já presente no processador de ciclo único utilizado como base. A implementação dos enables no registrador de PC e de pipeline, assim como os resets nos registradores de pipeline estão exibidos abaixo.

```
pcreg : flopENr --[MUDAR QUANDO FOR PIPELINE] torna-lo um registrador para por enable
generic map(width => 32)
port map(
  clk => clk,
  reset => reset,
  en => not_StallF,
  d => PCNext2,
  q => s_PC
);
```

Figura 17 - Enable no registrador de PC

```
189   library IEEE;
190   use IEEE.std_logic_1164.all;
191   use STD.TEXTIO.all;
192   use IEEE.NUMERIC_STD_UNSIGNED.all;
193
194   entity partial_IF_ID is
195     port (
196       clock, reset : in std_logic;
197       instrF : in std_logic_vector(31 downto 0);
198       stallD, flushD : in std_logic;
199
200       instrD : out std_logic_vector(31 downto 0)
201     );
202   end entity;
```

Figura 18 - Enable e reset de flush no registrador de pipeline fetch-decode

```
224  entity partial_ID_EX is
225    port (
226      clock, reset : in std_logic;
227
228      PCSrcD, RegWriteD : in std_logic;
229      MemtoRegD, MemWriteD : in std_logic;
230      ALUControlD, FlagWriteD : in std_logic_vector(1 downto 0);
231      BranchD, ALUSrcD : in std_logic;
232      RD1D, RD2D, ExtImmD : in std_logic_vector(31 downto 0);
233      WA3D : in std_logic_vector(3 downto 0);
234      CondD: in std_logic_vector(3 downto 0);
235      FlagsD : in std_logic_vector(3 downto 0);--[nao precisa mais ver tamanho]
236
237      FLushE : in std_logic;
238
239      PCSrcE, RegWriteE : out std_logic;
240      MemtoRegE, MemWriteE : out std_logic;
241      ALUControlE, FlagWriteE : out std_logic_vector(1 downto 0);
242      BranchE, ALUSrcE : out std_logic;
243      RD1E, RD2E, ExtImmE : out std_logic_vector(31 downto 0);
244      WA3E : out std_logic_vector(3 downto 0);
245      CondE : out std_logic_vector(3 downto 0);
246      FlagsE : out std_logic_vector(3 downto 0) -- [nao precisa mais ver o tamanho]
247    );
248  end entity;
```

Figura 19 - Reset de flush no registrador de pipeline decode-execute

Essas três alterações nos componentes foram testadas individualmente através do ModelSim e foi possível confirmar o funcionamento de todos. Em seguida, testamos o circuito com os três componentes modificados e novamente funcionou como esperado. Abaixo está exibido a demonstração de funcionamento dos três componentes.

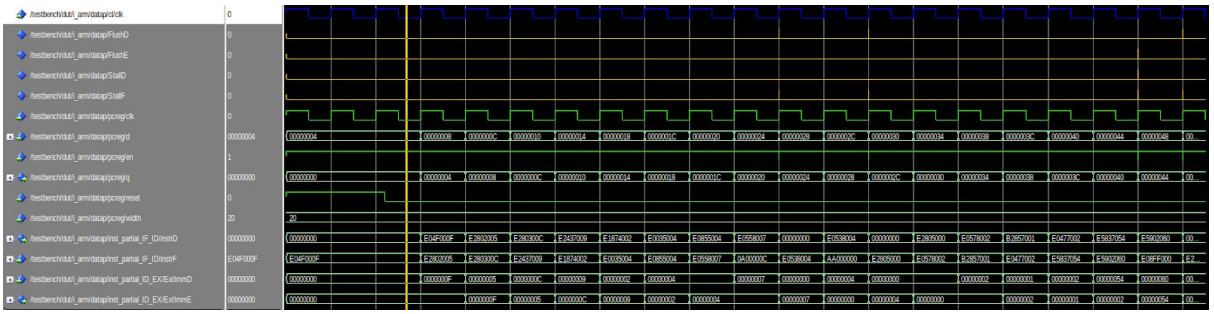


Figura 20 - Demonstração do funcionamento dos novos sinais adicionados

Hazard Unit:

A hazard unit é o componente responsável por gerar os sinais de controle que permitem o funcionamento do pipeline sem ocorrência de hazards.

A implementação desse componente possui 9 inputs:

1. **clock** - Sinal de clock do sistema.
2. **reset** - Sinal de reset do componente.
3. **Match** - Sinal de comparação entre o read address e o write address que identifica a necessidade de bolhas.
4. **PCWrPendingF** - Sinal quando um PC write está em progresso, habilita um stall no estágio de fetch e um flush no estágio de decode.
5. **RegWriteM** - Enable de escrita da register file no estágio de memória, usado pelo hazard unit para saber quando vai acontecer um write.
6. **RegWriteW** - Enable de escrita da register file no estágio de writeback, usado pelo hazard unit para saber quando vai acontecer um write.
7. **MemToRegE** - Sinal de controle do mux no estágio de writeback, usado pelo hazard para saber quando vai ocorrer uma escrita da memória
8. **PCSrcW** - Sinal de controle do mux no estágio de fetch que seleciona entre PC+4 e endereço na memória, usado pelo hazard unit para saber quando ocorre um branch para um endereço salvo na memória
9. **BranchTakenE** - Sinal de controle do mux no estágio de fetch que seleciona entre saída do outro mux e saída da ULA, usado pelo hazard unit para saber quando ocorre um branch.

E possui 6 outputs:

1. **StallF** - Sinal que controla o enable do registrador de PC e executa o stall no estágio de Fetch
2. **StallD** - Sinal que controla o enable do registrador de pipeline entre os estágios fetch-decode e executa o stall nesses estágios
3. **FlushD** - Sinal de reset no registrador de pipeline entre os estágios fetch-decode, que executa o stall nesses estágios
4. **FlushE** - Sinal de reset no registrador de pipeline entre os estágios decode-execute, que executa o stall nesses estágios
5. **ForwardAE** - Sinal de controle de um dos mux na entrada da ULA, selecionando entre um dos registradores do banco, a saída da ULA e a saída de leitura da memória
6. **ForwardBE** - Sinal de controle do outro mux na entrada da ULA, selecionando entre um dos registradores do banco, a saída da ULA e a saída de leitura da memória

Abaixo está ilustrado o diagrama do componente Hazard Unit desenvolvido. Ressaltamos que nesse diagrama os sinais PCSrcW, BranchTakenE e PCWrPendingF foram omitidos para facilitar a visibilidade.

Figura 21 - Diagrama da hazard unit

Abaixo está exibido o código da hazard unit implementada.

```
840  library IEEE;
841  use IEEE.std_logic_1164.all;
842  use STD.TEXTIO.all;
843  use IEEE.NUMERIC_STD_UNSIGNED.all;
844
845  entity hazard_unit is
846    port (
847      -- ENTRADAS
848      clock : in std_logic;
849      reset : in std_logic;
850      -- (Match_12D_E, Match_2E_W, Match_2E_M, Match_1E_W, Match_1E_M)
851      Match : in std_logic_vector(4 downto 0);
852      PCWrPendingF: in std_logic;
853
854      RegWriteM : in std_logic;
855      RegWriteW : in std_logic;
856      MemToRegE : in std_logic;
857
858      PCSrcW : in std_logic;
859      BranchTakenE : in std_logic;
860      -- SAIDAS
861      StallF : out std_logic;
862      StallD : out std_logic;
863      FlushD : out std_logic;
864      FlushE : out std_logic;
865      ForwardAE : out std_logic_vector(1 downto 0);
866      ForwardBE : out std_logic_vector(1 downto 0)
867    );
868  end entity;
869
```

```
870  architecture arch_hazard_unit OF hazard_unit is
871
872    signal Match_1E_M : std_logic;
873    signal Match_1E_W : std_logic;
874    signal Match_2E_M : std_logic;
875    signal Match_2E_W : std_logic;
876    signal Match_12D_E: std_logic;
877    signal LDRStall: std_logic;
878
879  begin
880    Match_1E_M <= Match(0);
881    Match_1E_W <= Match(1);
882    Match_2E_M <= Match(2);
883    Match_2E_W <= Match(3);
884    Match_12D_E <= Match(4);
885
886    -- Dar um stall quando instrucao LDR e o Reg de escrita em Execution e o mesmo que um dos operandos em Decode
887    LDRStall <= Match_12D_E and MemToRegE;
888
889    -- Saidas
890    StallD <= LDRStall;
891    StallF <= LDRStall or PCWrPendingF;
892    FlushE <= LDRStall or BranchTakenE;
893    FlushD <= PCWrPendingF or PCSrcW or BranchTakenE;
894
895    ForwardAE(1) <= '1' when (Match_1E_M and RegWriteM)
896                        else '0';
897    ForwardAE(0) <= '1' when (Match_1E_W and RegWriteW and (not ForwardAE(1)))
898                        else '0';
899    ForwardBE(1) <= '1' when (Match_2E_M and RegWriteM)
900                        else '0';
901    ForwardBE(0) <= '1' when (Match_2E_W and RegWriteW and (not ForwardBE(1)))
902                        else '0';
903  end;
```

Figura 22 - Código da hazard unit implementada

Realizamos o teste da hazard unit através de um testbench que desenvolvemos e posteriormente através de um teste integrado no circuito completo. O testbench que desenvolvemos testa simultaneamente a hazard unit e o componente hazard_logic, que é descrito em sequência, portanto a parte de testes da hazard unit está explicada junto com o próximo componente.

Hazard logic:

O hazard logic é o componente responsável por gerar os sinais Match e PCWrPendingF, utilizados pela hazard unit para gerenciar stalls durante a execução. Esse componente recebe os sinais clock, reset, RA1D, RA2D, RA1E, RA2E, WA3E, WA3M, WA3W, PCSrcD, PCSrcE e PCSrcM. Possui apenas duas saídas: Match e PCWrPendingF, ambas indo diretamente para o hazard unit. O código desse componente está exibido abaixo.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity hazard_logic is
  port (
    -- ENTRADAS
    clock : in std_logic;
    reset : in std_logic;
    RA1D : in std_logic_vector(3 downto 0);
    RA2D : in std_logic_vector(3 downto 0);
    RA1E : in std_logic_vector(3 downto 0);
    RA2E : in std_logic_vector(3 downto 0);
    WA3E : in std_logic_vector(3 downto 0);
    WA3M : in std_logic_vector(3 downto 0);
    WA3W : in std_logic_vector(3 downto 0);
    PCSrcD : in std_logic;
    PCSrcE : in std_logic;
    PCSrcM : in std_logic;
    -- SAIDAS
    -- (Match_12D_E, Match_2E_W, Match_2E_M, Match_1E_W, Match_1E_M)
    Match : out std_logic_vector(4 downto 0);
    PCWrPendingF: out std_logic
  );
end entity;

architecture arch_hazard_logic OF hazard_logic is

  signal Match_1E_M : std_logic;
  signal Match_1E_W : std_logic;
  signal Match_2E_M : std_logic;
  signal Match_2E_W : std_logic;

  signal Match_12D_E: std_logic;
  signal Match_12D_Ea: std_logic;
  signal Match_12D_Eb: std_logic;


```

```
947  begin
948    -- Comparar se Reg 1 de Execution e o mesmo que o reg de escrita em Memory
949    Match_1E_M <= '1'when RA1E = WA3M else '0';
950    -- Comparar se Reg 1 de Execution e o mesmo que o reg de escrita em WriteBack
951    Match_1E_W <= '1'when RA1E = WA3W else '0';
952    -- Comparar se Reg 2 de Execution e o mesmo que o reg de escrita em Memory
953    Match_2E_M <= '1'when RA2E = WA3M else '0';
954    -- Comparar se Reg 2 de Execution e o mesmo que o reg de escrita em WriteBack
955    Match_2E_W <= '1'when RA2E = WA3W else '0';
956
957    -- Comparar se Reg 1 de Decode e o mesmo que o reg de escrita em Execution
958    Match_12D_Ea <= '1'when RA1D = WA3E else '0';
959    -- Comparar se Reg 2 de Decode e o mesmo que o reg de escrita em Execution
960    Match_12D_Eb <= '1'when RA2D = WA3E else '0';
961    -- Formar Match_12D_E
962    Match_12D_E <= Match_12D_Ea or Match_12D_Eb;
963
964    -- Quando uma escrita de branch no PC estaria ocorrendo nos estagios Decode ou Execution ou Memory
965    PCWrPendingF <= PCSrcD or PCSrcE or PCSrcM;
966
967    Match(4) <= Match_12D_E;
968    Match(3) <= Match_2E_W;
969    Match(2) <= Match_2E_M;
970    Match(1) <= Match_1E_W;
971    Match(0) <= Match_1E_M;
972  end;
```

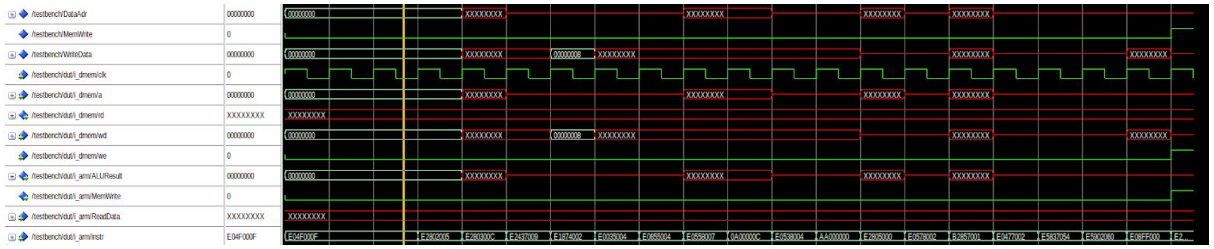
Figuras 23 - Código da hazard logic implementada

Esse componente foi testado de duas maneiras, a primeira é um testbench que desenvolvemos para testá-lo junto da hazard unit e a segunda é o teste da integração com o resto do circuito.

Após os testes individuais pelo testbench, os dois componentes foram integrados ao resto do circuito e novamente executamos o programa no ModelSim, confirmando que a integração também foi realizada corretamente.

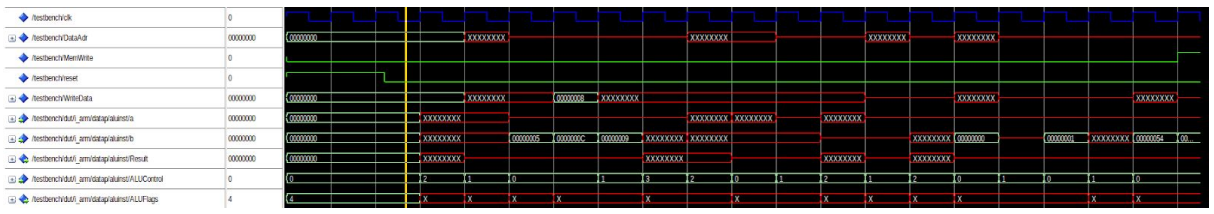
2. Simulação

Como pode ser visto, nem tudo deu certo.



3. Análise

O projeto de processador ARM versão pipeline foi implementado com sucesso. Todos os testes da versão final foram bem sucedidos e a execução correspondeu integralmente ao esperado. Utilizando a ferramenta ModelSim é possível analisar detalhadamente cada etapa da execução: observando uma instrução sendo acessada na memória, em seguida sendo decodificada enquanto é realizada a leitura do banco de registradores, executando a operação da ULA, acessando a memória ao fim e por último escrevendo o resultado no registrador. Também é possível observar os estágios de processamento sendo executados em paralelo com instruções diferentes, assim como observar um stall quando a sequência de instruções resulta em um hazard e observar um flush dos registradores quando ocorre um branch. Abaixo essa execução está ilustrada através de uma execução no ModelSim.



Como esperado, a versão pipeline do processador tem um desempenho significativamente superior à versão monociclo de base. Embora não haja melhoras em

latência, há um speedup pelo aumento na vazão em decorrência da execução com paralelismo. Idealmente a versão pipeline teria um throughput 5x maior, já que executa cinco instruções em paralelo em seus 5 estágios de pipeline. No entanto, a implementação em pipeline apresenta overheads e períodos de vazão reduzida como stalls e flushs, resultando em um ganho de vazão inferior ao valor ideal de 5x, porém ainda assim com ganho de performance considerável. Nossa implementação apresentou ganhos de aproximadamente 40% na performance.

Buscamos implementar o projeto de pipeline da maneira mais completa possível, seguindo as orientações do livro texto, incluindo modificações opcionais como forwarding. Dito isso, acreditamos que nossa implementação apresenta algumas limitações fundamentais como previsão de desvio menos eficiente e pouca variedade de instruções suportadas. Em relação à previsão de desvio, utilizamos o mecanismo “nottaken”, que sempre supõe que o desvio não será tomado, no entanto, vimos na atividade prática realizada que esse mecanismo de previsão normalmente não tem a melhor performance. Essa limitação pode ser resolvida empregando outro mecanismos de previsão mais eficientes como o “comb”, que combina os mecanismos de previsão “bi-modal” e “2-level predictor”. Em relação à variedade de instruções, nosso processador suporta um conjunto muito limitado de instruções: LDR, STR, processamento de dados (com operandos imediatos e de registradores) e B. Implementações mais completas como o conjunto de instruções ARM Assembly possui 232 instruções diferentes, suportando uma variedade significativamente maior de instruções. Essa limitação pode ser resolvida adicionando novos componentes que permitam uma maior variedade de instruções, complementando o projeto desenvolvido nessa etapa.

Infelizmente não conseguimos fazer o projeto funcionar completamente, ele funciona realizando encaminhamento geralmente, entretanto, não conseguimos descobrir o motivo pelo qual isso não funciona sempre, outro problema durante a implementação dos registradores de pipeline tivemos dificuldades em conectar todos os sinais corretamente, pois eram muitos e nenhum integrante tinha uma ferramenta do tipo “RTL viewer” disponível para visualizar as conexões em uma interface mais intuitiva. Resolvemos esse último problema incluindo uma etapa de testes intermediária conectando “caixas vazias” no lugar dos registradores de pipeline, de forma que era possível testar apenas as conexões e com isso completamos essa etapa com facilidade. Outra dificuldade, durante a integração da hazard unit, se dava com relação a erros no componente criado que, embora compilasse sem erros, não funcionava da forma esperada e tivemos dificuldade em identificar o erro. Para resolver isso criamos um testbench para os componentes hazard unit e hazard logic, com isso foi possível testá-los com mais facilidade, corrigir alguns dos erros.