Trabalho 3 de Arquitetura de Computadores I
Turma 590 - 2020/2
Professor: Cesar Augusto Missio Marcon
Alunos: Diego Henrique Oliveira; Ícaro Stumpf

Projeto – - - *Implementar a segunda versão do MIPS pipeline*

Descrição: O objetivo desta etapa é implementar a arquitetura do processador MIPS empregando pipeline com tratamento do conflito de dados e controle.

Para o conflito de dados, deverá ser incluído o mecanismo de adiantamento e o pipeline deverá ter uma solução de inserção automática de bolha em situações em que o adiantamento não resolva. O tratamento de conflito de controle deve feito de forma automática através do emprego da política **"salto nunca ocorre".** Assim, deve ser excluído do pipeline (*flush*) as instruções que tenham sido inseridas incorretamente, quando um desvio ocorrer.

Para o desenvolvimento do presente trabalho, foi utilizada, como base, a implementação em VHDL da MIPS pipeline desenvolvida por esse mesmo grupo de alunos, na ocasião de realização do trabalho prático 2, que, por sua vez, foi desenvolvida a partir da implementação em VHDL da MIPS multiciclo pelos professores Fernando Moraes e Ney Calazans, versão junho de 2020[1].

[1] Disponível em http://www.inf.pucrs.br/~calazans/undergrad/orgcomp_EC/mips_multi/Files_MIPS-MC_SingleEdge.7z, acesso em 24 de Outubro de 2020

Trabalho 3 de Arquitetura de Computadores I
Turma 590 - 2020/2
Professor: Cesar Augusto Missio Marcon
Alunos: Diego Henrique Oliveira; Ícaro Stumpf

I – Questões preliminares

Primeiramente, criamos, com base nas orientações dadas em aulas, dois objetos do tipo composto *record*, os quais nos serão úteis para detecção de conflitos:

```
type DataConflict is record
        opA : std_logic_vector(1 downto 0);
        opB : std_logic_vector(1 downto 0);
        bubble: std_logic;
end record;
```

opA: para detecção de conflitos com o registrador Rs

opB: para detecção de conflitos com o registrador Rt

bubble: para quando for necessário parar o *pipeline*

Com relação à primeira implementação da MIPS *pipeline*, foram adicionados os seguintes sinais (em vermelho):

```
entity datapath is
      port(  ck, rst :      in std_logic;
             ce, rw, bw : out std_logic; -- used in the forth stage to control memory access
             d_address :   out std_logic_vector(31 downto 0);
             data :        inout std_logic_vector(31 downto 0);
             uins :        in microinstruction;
             IR_OUT :      out std_logic_vector(31 downto 0);
             i_address :   out std_logic_vector(31 downto 0);
             instruction : in std_logic_vector(31 downto 0)
          );
end datapath;

architecture datapath of datapath is
    signal pc, incpc, IR, result, R1, R2, RB, RIN, sign_extend, op1, op2, RALU, mdr_int,
           dtpc, cte_im: std_logic_vector(31 downto 0) := (others=> '0');
    signal adD, adS : std_logic_vector(4 downto 0) := (others=> '0');
    signal salta : std_logic := '0';
    signal conflict : DataConflict;
```

Mais adiante ficará evidente sua necessidade.

Trabalho 3 de Arquitetura de Computadores I
Turma 590 - 2020/2
Professor: Cesar Augusto Missio Marcon
Alunos: Diego Henrique Oliveira; Ícaro Stumpf

A unidade de controle de conflitos segue abaixo (seguimos as orientações disponíveis nos *slides*):

```
-------------------------------------------------------------------------------------------------
-- opA  | Meaning - adD = RS
-------------------------------------------------------------------------------------------------
 -- "00" | No conflict
 -- "01" | Instruction on the forth stage conflicts with the instruction on the second stage
 -- "10" | Instruction on the third stage conflicts with the instruction on the second stage
 -- "11" | Load word/Immediate instructions conflict
 -------------------------------------------------------------------------------------------------
conflict.opA <= "11" when ((memer.uins.ce = '1') and (memer.uins.rw = '1') and (memer.uins.wreg = '1') and (memer.adD = diex.RS)) else
                "10" when (exmem.uins.wreg = '1') and (exmem.adD /= "00000") and (exmem.adD = diex.RS) else
                "01" when (memer.uins.wreg = '1') and (memer.adD /= "00000") and (exmem.adD /= diex.RS) and (memer.adD = diex.RS) else
                "00";


 -------------------------------------------------------------------------------------------------
 -- opB  | Meaning - adD = Rt
 -------------------------------------------------------------------------------------------------
 -- "00" | No conflict
 -- "01" | Instruction on the forth stage conflicts with the instruction on the second stage
 -- "10" | Instruction on the third stage conflicts with the instruction on the second stage
 -- "11" | Load word/Immediate instructions conflict
 -------------------------------------------------------------------------------------------------
 conflict.opB <= "11" when (memer.uins.ce = '1') and (memer.uins.rw = '1') and (memer.uins.wreg = '1') and (memer.adD = diex.RT) and
                           (diex.uins.i /= LBU) and (diex.uins.i /= LW) and (diex.uins.i /= LUI) else
                 "10" when (exmem.uins.wreg = '1') and (exmem.adD /= "00000") and (exmem.adD = diex.RT) and
                           (diex.uins.i /= LBU) and (diex.uins.i /= LW) and (diex.uins.i /= LUI) else
                 "01" when (memer.uins.wreg = '1') and (memer.adD /= "00000") and (exmem.adD /= diex.RT) and (memer.adD = diex.RT) and
                           (diex.uins.i /= LUI) and (diex.uins.i /= LBU) and (diex.uins.i /= LW) else
                 "00";

-------------------------------------------------------------------------------------------------
-- bubble | Meaning - Detects load word/Immediate instructions conflict, stops Pipeline for a clock cycle
-------------------------------------------------------------------------------------------------
 -- '1' | Conflict
 -- '0' | No conflict
 -------------------------------------------------------------------------------------------------

conflict.bubble <= '1' when (((diex.uins.ce = '1') and (diex.uins.rw = '1') and (diex.uins.wreg = '1') and (diex.RT = bidi.IR(20 downto 16)))
                            or ((diex.uins.ce = '1') and (diex.uins.rw = '1') and (diex.uins.wreg = '1') and (diex.RT = bidi.IR(25 downto 21))))
                   else '0';
```

Os *records* opA e opB prescindem de maiores explicações – os comentários acima dão conta de todos os casos do MUX em questão [ressalte-se que estávamos tendo problemas com as instruções LUI, LBU e LW (nelas, o registrador RT sofre escrita, não leitura, diferentemente das outras instruções), então, foi criada a condição acima para contornar o problema, evitando que um conflito fosse erroneamente detectado]. A lógica utilizada na implementação da unidade de detecção de conflitos foi baseada nas considerações feitas nas lâminas “Aula 7 – Tratamento de Conflitos no Pipeline”.

Quanto ao *bubble*, tipo criado para parar o *pipeline* quando houver conflito em que não seja possível o adiantamento, a ideia é a seguinte: é necessário saber se será feita leitura na memória e escrita em algum registrador (instruções como LW, por exemplo, entram nesse contexto). Além disso, é necessário saber se o registrador em que será feita a escrita é o mesmo que está na barreira BIDI. Ou seja, caso o registrador *target* da instrução que está na barreira DIEX seja igual ao registrador que está na barreira BIDI (que vai ser lido), é necessário parar o pipeline, porque não há adiamento possível para ser feito (sendo, na nossa implementação, bubble= ‘1’).

Cumpre fazer mais um esclarecimento quanto ao caso em que temos opA ou opB recebendo “11”: esse caso só acontece quando o *pipeline* para e, nesse caso, temos que a instrução que está na barreira MEMER leu da memória e vai escrever em algum registrador, então, é possível fazer o adiantamento da barreira MEMER para a barreira DIEX.

II – Tratamento de conflitos

Alterações necessárias relativas ao *program conter:*

```
dtpc <= exmem.RALU when (exmem.uins.inst_branch='1' and exmem.salta='1') or
                        exmem.uins.i=J or exmem.uins.i=JAL or exmem.uins.i=JALR or exmem.uins.i=JR
                  else pc when (conflict.bubble = '1')
                        else incpc;
```

Ou seja, em havendo uma bolha, o PC recebe o próprio valor, o que significa, na prática, que ele não avança uma instrução.

A barreira BIDI foi modificada da seguinte forma:

```
--================================================================================
-- BI/DI Pipeline Register
--================================================================================
--process(ck, rst)
process(ck, rst)
begin
   if (rst = '1') then
       bidi.npc <= (others=>'0');
       bidi.IR <= (others=>'0');
   elsif ck'event and ck = '0' then
      if (diex.uins.i = J or diex.uins.i = JAL or salta = '1') then
         bidi.npc <= bidi.npc;
         bidi.IR <= (others=>'0');
      elsif (conflict.bubble = '1') then
         bidi.npc <= bidi.npc;
         bidi.IR <= bidi.IR;
      else
       bidi.npc <= incpc; -- store in the pipeline register pc + 4
       bidi.IR <= instruction; -- store in the pipeline register instruction code
      end if;
   end if;
end process;

--================================================================================
```

Assim, em havendo alguma instrução J, JAL ou um *branch* válido, ocorre um *flush*.

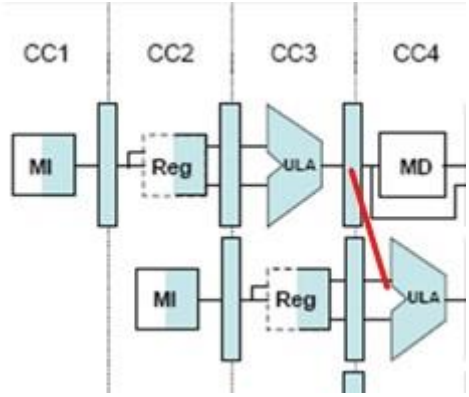
Caso haja uma bolha (bubble=’1’), não é feito o *flush*; o pipeline apenas para por um ciclo de relógio.

Quanto à barreira DIEX, foi acrescentada a seguinte condição:

```
elsif ck'event and ck = '0' then
   if (conflict.bubble = '1' or diex.uins.i = J or diex.uins.i = JAL or salta = '1') then
      diex.npc <= (others=>'0');
      diex.RD <= (others=>'0');
      diex.RT <= (others=>'0');
      diex.RS <= (others=>'0');
      diex.RA <= (others=>'0');
      diex.RB <= (others=>'0');
      diex.IMED <= (others=>'0');
      -- uins rst
      diex.uins.inst_branch <= '0';
      diex.uins.inst_grupol <= '0';
      diex.uins.inst_grupoI <= '0';
      diex.uins.wreg <= '0';
      diex.uins.ce <= '0';
      diex.uins.rw <= '0';
      diex.uins.bw <= '1';
      diex.uins.i <= NOP;
```

Veja-se que as instruções J, JAL, *branch* e a condição bubble=’1’ estão todas na mesma condição, pois, em todos os casos, propaga-se um NOP (tanto quanto o pipeline para, como quando há *flush*).

Quanto à ULA, antes de entrar no código em si, faz-se necessário ilustrar a inspiração por trás do código:



Ou seja, “pegamos” o sinal da barreira temporal que sucede o estágio de execução e o colocamos na entrada da própria ULA, obedecendo às seguintes condições:

```
            -- select the first ALU operand
op1 <=   diex.npc  when (diex.uins.inst_branch='1'
            or diex.uins.i=J or diex.uins.i=JAL) else
         exmem.RALU when ((conflict.opA = "10") and (diex.uins.inst_branch='0')) else
         memer.RALU when ((conflict.opA = "01") and (diex.uins.inst_branch='0')) else
         memer.MDR when (conflict.opA = "11") else
         diex.RA;

            -- select the second ALU operand
op2 <=   diex.RB when (diex.uins.inst_grupoI='1' or diex.uins.i=SLTU or diex.uins.i=SLT or diex.uins.i=JR
          or diex.uins.i=SLLV or diex.uins.i=SRAV or diex.uins.i=SRLV) and (conflict.opB = "00") else
         exmem.RALU when ((conflict.opB = "10") and (diex.uins.inst_branch='0') and (diex.uins.i /= SW) and (diex.uins.inst_grupoI = '0')) else
         memer.RALU when ((conflict.opB = "01") and (diex.uins.inst_branch='0') and (diex.uins.i /= SW) and (diex.uins.inst_grupoI = '0')) else
         memer.MDR when ((conflict.opB = "11") and (diex.uins.i /= SW) and (diex.uins.inst_grupoI = '0')) else
      diex.IMED;
```

Os operandos foram adaptados de modo a ficarem coerentes com as detecções de conflito elaboradas acima. Quanto ao op2, é importante fazer alguns esclarecimentos: como ele recebe o valor do registrador *target*, foram feitas restrições quanto a alguns tipos de instruções, uma vez que, na arquitetura MIPS, o registrador *target* contém o dado imediato. Ou seja, sem essas condições, não teria sentido a execução das instruções do tipo I.

Foi necessário, portanto, criar um sinal para armazenar o conteúdo das instruções do tipo I. Para esse fim, criamos o sinal RB:

```
            -- saves RT register's content
RB   <=    memer.MDR when (conflict.opB = "11") else
           exmem.RALU when ((conflict.opB = "10") and (diex.uins.inst_branch='0')) else
           memer.RALU when ((conflict.opB = "01") and (diex.uins.inst_branch='0')) else
           diex.RB;
```

Para o tratamento do conflito de controle, como se pede para empregar a política de “o salto nunca ocorre”, elaboramos uma lista (que supomos ser exaustiva) de todas as situações de conflito em que pode haver uma instrução de *branch*, englobando todos os possíveis cenários de conflitos, fazendo a análise correta dos registradores em cada caso:

```
 -- evaluation of conditions to take the branch instructions
salta <=  '1' when (
                        (conflict.opA = "10" and diex.RB=exmem.RALU and diex.uins.i=BEQ) or
                        (conflict.opB = "10" and diex.RA=exmem.RALU and diex.uins.i=BEQ) or
                        (conflict.opA = "01" and diex.RB=memer.RALU and diex.uins.i=BEQ) or
                        (conflict.opB = "01" and diex.RA=memer.RALU and diex.uins.i=BEQ) or
                        (conflict.opA = "00" and conflict.opB = "00" and diex.RA=diex.RB  and diex.uins.i=BEQ) or
                        (conflict.opA = "10" and exmem.RALU>=0 and diex.uins.i=BGEZ) or
                        (conflict.opA = "01" and memer.RALU>=0 and diex.uins.i=BGEZ) or
                        (conflict.opA = "00" and diex.RA>=0 and diex.uins.i=BGEZ) or
                        (conflict.opA = "10" and exmem.RALU<=0 and diex.uins.i=BLEZ) or
                        (conflict.opA = "01" and memer.RALU<=0 and diex.uins.i=BLEZ) or
                        (conflict.opA = "00" and diex.RA<=0 and diex.uins.i=BLEZ) or
                        (conflict.opA = "10" and diex.RB/=exmem.RALU and diex.uins.i=BNE) or
                        (conflict.opB = "10" and diex.RA/=exmem.RALU and diex.uins.i=BNE) or
                        (conflict.opA = "01" and diex.RB/=memer.RALU and diex.uins.i=BNE) or
                        (conflict.opB = "01" and diex.RA/=memer.RALU and diex.uins.i=BNE) or
                        (conflict.opA = "00" and conflict.opB = "00" and diex.RA/=diex.RB and diex.uins.i=BNE)
                       )
                else '0';
```

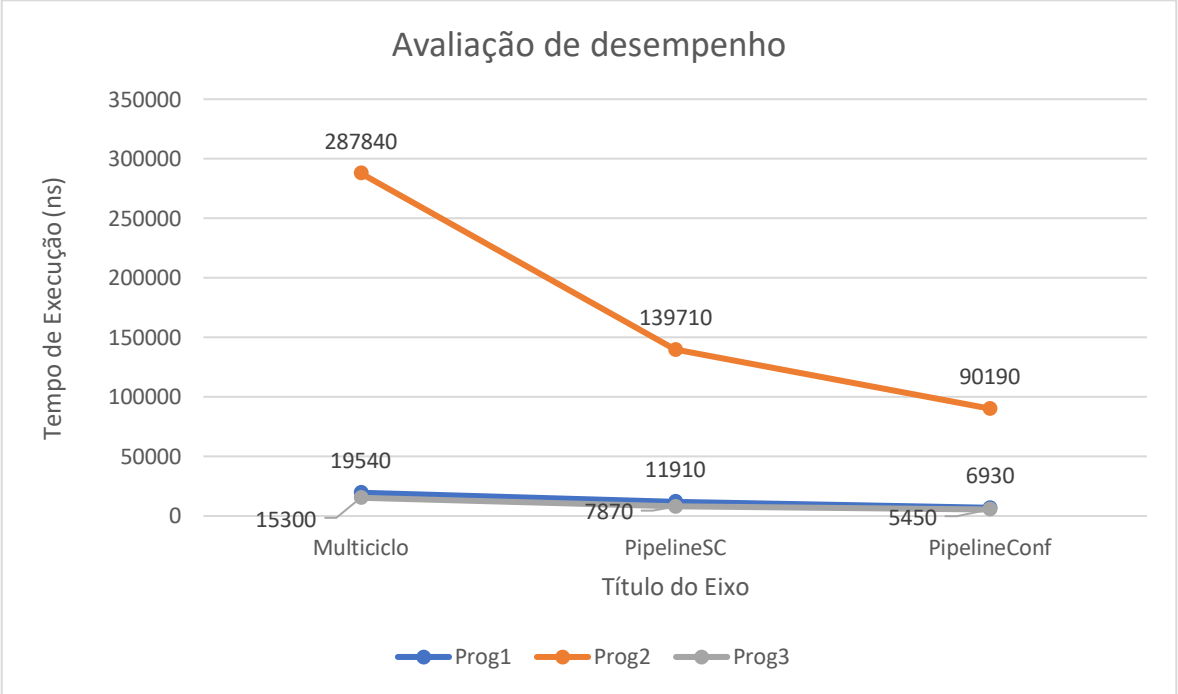
Enfim, a barreira EXMEM teve uma única modificação:

```
-- EX/MEM Pipeline Register
--=============================================================================

process(ck, rst)
begin
    if rst = '1' then
        exmem.npc <= (others=>'0');
        exmem.salta <= '0';
        exmem.RALU <= (others=>'0');
        exmem.adD <= (others=>'0');
        exmem.RB <= (others=>'0');
        -- uins rst
        exmem.uins.inst_branch <= '0';
        exmem.uins.inst_grupol <= '0';
        exmem.uins.inst_grupoI <= '0';
        exmem.uins.wreg <= '0';
        exmem.uins.ce <= '0';
        exmem.uins.rw <= '0';
        exmem.uins.bw <= '1';
        exmem.uins.i <= NOP;

    elsif ck'event and ck = '0' then
        exmem.npc <= diex.npc;
        exmem.salta <= salta;
        exmem.RALU <= RALU;
        exmem.adD <= adD;
        exmem.uins <= diex.uins;
        exmem.RB <= RB;
    end if;
end process;
```

exmem.RB <= RB; em vez de exmem.RB <= diex.RB (lembrando que o RB foi criado por nós para guardar o valor do registrador *target*), pois, em havendo algum conflito com instruções do tipo I na ULA, teremos o valor original do registrador *target* armazenado.

III – Avaliação de Desempenho

Executados os programas na MIPS multiciclo e nas MIPS pipeline, versões 1 e 2, obtivemos o seguinte gráfico:



Percebe-se que essa segunda versão da MIPS pipeline possui desempenho superior (menor tempo de execução) à primeira em todos os programas estudados. Os programas 1, 2 e 3 tiveram uma queda de aproximadamente 42%, 35% e 31%, em seus tempos de execução, respectivamente, em comparação aos tempos obtidos na versão anterior da MIPS Pipeline.