PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

DIEGO HENRIQUE OLIVEIRA
ÍCARO STUMPF
LEONARDO BARBOSA DA ROSA

OTIMIZAÇÃO DO MECANISMO DE TRATAMENTO DE CONFLITO DE CONTROLE

Porto Alegre, Rio Grande do Sul - Brasil
2020

1. DESCRIÇÃO

O objetivo da quarta etapa é implementar um mecanismo de predição dinâmica com dois bits partindo inicialmente do estado “salto não ocorre” (not taken). A política deve ser mantida para cada instrução de salto condicional que venha a ser encontrada durante a execução de um programa qualquer. Note que para manter a política para toda e qualquer instrução de salto condicional, deve existir um mecanismo que associa endereços de salto com os históricos. Implemente o histórico para 4 instruções de desvio. Caso o programa tenha mais de 4 instruções de desvios, as demais devem ser implementadas com a política default not taken.

2. INSERÇÃO DE PACKAGE

Inicialmente foi adicionado um novo package, com a finalidade de criarmos um tipo de estrutura que contenha as seguintes informações:

- VB, do tipo std_logic para guardar o dado do bit de validação;
- FSM, do tipo std_logic(1 downto 0), com a finalidade de conter o valor da maquina de estado finita;
- Address, do tipo std_logic(31 downto 0), com a finalidade de conter o valor do endereço da instrução que contenha um branch;
- b_address, do tipo std_logic(31 downto 0), com a finalidade de conter o valor do endereço da instrução em que seria a próxima de o branch analisado for válido.

Esta estrutura supracitada pode ser visualizada na imagem abaixo:

```
type DynamicPrediction is record
        VB : std_logic; -- validation bit
        FSM : std_logic_vector(1 downto 0); -- Finite State Machine
        address : std_logic_vector(31 downto 0); -- branch instruction address
        b_address : std_logic_vector(31 downto 0); -- branch result address
end record;
```

3. MODIFICAÇÃO NO DATAPATH

Nesta parte do código foi adicionado um buffer com tamanho de 32, com a finalidade de incluir a possibilidade de enfileirar 32 instruções condicionais.

Essa estrutura, que é basicamente uma estrutura de dados do tipo vetor, com tamanho 32, e do tipo definido na inserção de package, DynamicPrediction, é inicialmente inicializado com todas as suas variáveis em zero, afim de não ocorrer o estado “U” (não inicializado), durante a execução.

Essa inserção pode ser visualizada no trecho de código abaixo:

```
-- Dynamic Prediction Signals
constant array_range : integer:=31;
type buff is array(0 to array_range) of DynamicPrediction;
constant initialize_array : DynamicPrediction := (VB => '0',
                                    FSM => (others=>'0'),
                                address => (others=>'0'),
                                b_address => (others=>'0'));
signal branch_buffer : buff := (others => initialize_array);
```

Além da inserção desta estrutura de dados, foram também adicionadas variáveis com a finalidade de serem utilizadas para salvarem os valores da posição de qual condicional está sendo lidada na estrutura de dados acima, sendo a variável id_tb responsável por guardar o índice da instrução que possui o branch tomado e a variável id_wtb responsável por guardar o índice da instrução que possui o branch não tomado, para que assim seja possível obter o endereço correto da próxima instrução a ser executada, mantendo salvo tanto a posição da instrução que seria executada se o resultado da condicional for verdadeira, bem como a posição da instrução se o resultado do branch for falso.

Essas variáveis, id_tb e id_wtb, podem ser visualizadas abaixo:

```
signal id_tb, id_wtb : integer :=0;
```

Foram também adicionadas flags, taken_branch e wrongly_taken_branch. Sendo que a primeira, quando em nível lógico alto, informa que um branch foi tomado e, se o estado é nível lógico baixo, branch não tomado, já a segunda funciona da seguinte forma: quando carregado com o valor 0 informa que o branch foi tomado corretamente e quando em 1 informa que o branch foi tomado erroneamente.

```
signal taken_branch : std_logic := '0';
signal wrongly_taken_branch : std_logic := '0';
```

4. DEFINIÇÃO DA MAQUINA DE ESTADOS FINITOS

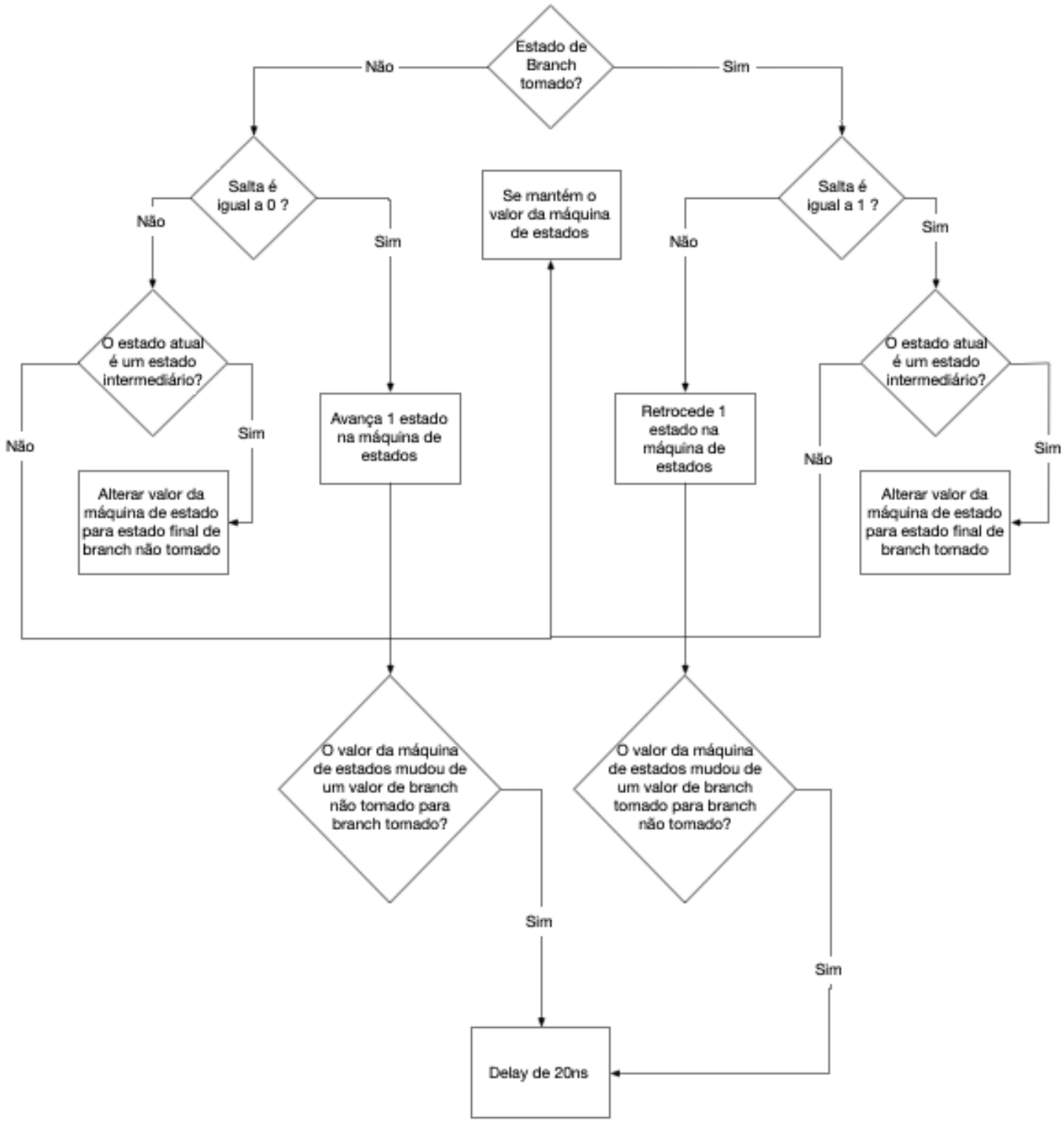
Neste trecho de código é estruturado a lógica da máquina de estados. Os estados 00 e 01 são responsáveis pelas hipóteses de condicionais não tomadas e os estados 10 e 11 responsáveis por condicionais que serão executadas.

Ao verificar a barreira DIEX, se a instrução nela contida for uma instrução condicional, o trecho de código abaixo será executado.

```
process(ck, rst)
begin
   if (rst = '1') then
      branch_buffer <= (others => initialize_array);
   elsif ck'event and ck = '1' then
      if (diex.uins.inst_branch='1') then
         for i in 0 to array_range loop
            if (branch_buffer(i).address = diex.pc) then
               if (branch_buffer(i).FSM = "00" and salta = '1') then
                  branch_buffer(i).FSM <= "01";
                  exit;
               elsif (branch_buffer(i).FSM = "01" and salta = '1') then
                  branch_buffer(i).FSM <= "11" after 20ns;
                  exit;
               elsif (branch_buffer(i).FSM = "01" and salta = '0') then
                  branch_buffer(i).FSM <= "00";
                  exit;
               elsif (branch_buffer(i).FSM = "11" and salta = '1') then
                  branch_buffer(i).FSM <= "11";
                  exit;
               elsif (branch_buffer(i).FSM = "11" and salta = '0') then
                  branch_buffer(i).FSM <= "10";
                  exit;
               elsif (branch_buffer(i).FSM = "10" and salta = '1') then
                  branch_buffer(i).FSM <= "11" after 20ns;
                  exit;
               elsif (branch_buffer(i).FSM = "10" and salta = '0') then
                  branch_buffer(i).FSM <= "00" after 20ns;
               else
                  branch_buffer(i).FSM <= "00";
                  exit;
               end if;
            elsif (branch_buffer(i).VB = '0') then
               branch_buffer(i).VB <= '1';
               branch_buffer(i).address <= diex.pc;
               branch_buffer(i).b_address <= RALU;
               exit;
            end if;
         end loop;
      end if;
   end if;
end process;
```

Neste trecho de código é possível identificar que está sendo executado um loop, com a finalidade de percorrer todas as instruções contidas no array de 32 instruções definido no tópico “Modificação no Datapath”.

Para cada posição desse vetor, é analisado o endereço da instrução, comparado com o endereço contido em DIEX, e, quando eles forem iguais, executa-se o seguinte algoritmo:



Resumidamente, este algoritmo possui a finalidade de verificar se o estado da máquina de estados está condizente com o valor da variável responsável por identificar o saldo da instrução.

Se a máquina estiver em estado de branch tomado e o valor de salta for 0, significa que esse branch não deveria ter sido tomado, e, por isso, retrocede-se um estado. Caso o valor de salta esteja como verdadeiro e esteja em um estado transitório, a máquina de estados será avançada para um estado fixo, e, caso esteja em um estado fixo, nele se manterá.

Em contrapartida, se a máquina de estados estiver em um estado de branch não tomado e o valor de salta não estiver condizente, isto é, estiver em 1, significa que este branch deveria ter sido tomado e, por essa razão, avança-se um estado. Na situação em que o valor de salta esteja correto, isto é, esteja em 0, e esteja em um estado intermediário, o estado da máquina irá para o estado fixo de branch não tomado, e em caso de já estar neste estado fixo, neste será mantido.

Para cada transição entre estado de branch tomado para não tomado, não necessariamente nesta ordem, será aguardado 1 ciclo de clock com a finalidade de manter o correto funcionamento do processador.

Caso a instrução de branch analisada não esteja contida no vetor de instruções condicionais, ela será incluída na estrutura de dados, através do código abaixo:

```
elsif (branch_buffer(i).VB = '0') then
   branch_buffer(i).VB <= '1';
   branch_buffer(i).address <= diex.pc;
   branch_buffer(i).b_address <= RALU;
   exit;
end if;
```

5. IDENTIFICAÇÃO DE CONDICIONAIS

Como descrito no item 3, foram adicionadas 2 variáveis responsáveis por avisar o restante dos processos assíncronos do processador que foi identificada uma condicional tomada correta ou erroneamente. Essas variáveis são taken_branch e wrongly_taken_branch.

A variável taken_branch terá seu valor alterado para nível lógico alto por 1 ciclo de clock se for verificado que o incpc está com o endereço de uma condicional, conforme o código abaixo:

```
-------------------------------------------------------------------------------
-- Taken branch flag (FSM = "11" or FSM = "10")
-------------------------------------------------------------------------------

process(ck)
begin
   if ck'event and ck = '1' then
      for i in 0 to array_range loop
         if (branch_buffer(i).address = incpc) then
            if (branch_buffer(i).FSM = "11" or branch_buffer(i).FSM = "10") then
               id_tb <= i;
               taken_branch <= '1';
               taken_branch <= '0' after 20ns;
               exit;
            else
               id_tb <= 0;
               taken_branch <= '0';
               exit;
            end if;
         end if;
      end loop;
   end if;
end process;
```

Já a variável wrongly_taken_branch é responsável por informar que está sendo processada uma condicional cujo resultado é falso, isto é, que este branch não foi tomado, e, portanto, manterá seu valor em nível lógico alto por 1 ciclo de clock, conforme trecho de código abaixo:

```
-------------------------------------------------------------------------------
-- Wrongly taken branch flag ((FSM = "11" or FSM = "10") and salta = '0')
-------------------------------------------------------------------------------
process(ck)
begin
   if ck'event and ck = '0' then
      for i in 0 to array_range loop
         if (branch_buffer(i).address = diex.pc) then
            if ((branch_buffer(i).FSM = "10" or branch_buffer(i).FSM = "11") and salta = '0') then
               id_wtb <= i;
               wrongly_taken_branch <= '1';
               wrongly_taken_branch <= '0' after 20ns;
               exit;
            else
               id_wtb <= 0;
               wrongly_taken_branch <= '0';
               exit;
            end if;
         end if;
      end loop;
   end if;
end process;
```

6. MODIFICAÇÃO DA INCREMENTAÇÃO DO PC

A variável incpc, responsável por ditar o andamento do código conforme os endereços do assembly carregado no processador, foi modificada para receber o endereço resultante do salto a ser executado em caso de branch tomado e, em caso de branch não tomado, receber o endereço da próxima instrução a ser realizada. Se a instrução analisada não for uma condicional, mantém-se o funcionamento de incrementar o pc em 4 unidades, para direcioná-lo à próxima instrução. Assim, temos:

```
incpc <= branch_buffer(id_tb).b_address when (taken_branch = '1')
         else (branch_buffer(id_wtb).address + 4) when (wrongly_taken_branch = '1')
         else pc + 4;
```

7. MODIFICAÇÃONAS BARREIRAS TEMPORAIS

Nas barreiras BI/DI e DI/EX foram adicionadas descrições para executar o flush do processador, isto é uma limpeza do Pipeline, em caso de identificação de uma predição errada, ou seja, se o salta estiver em 1 e a máquina de estados estiver com valores de condicionais não tomadas, 00 e 01, ou ao contrário, se o valor de salta estiver em 0 e a máquina de estados estiver valorada com 10 ou 11.

Flush da barreira BI/DI:

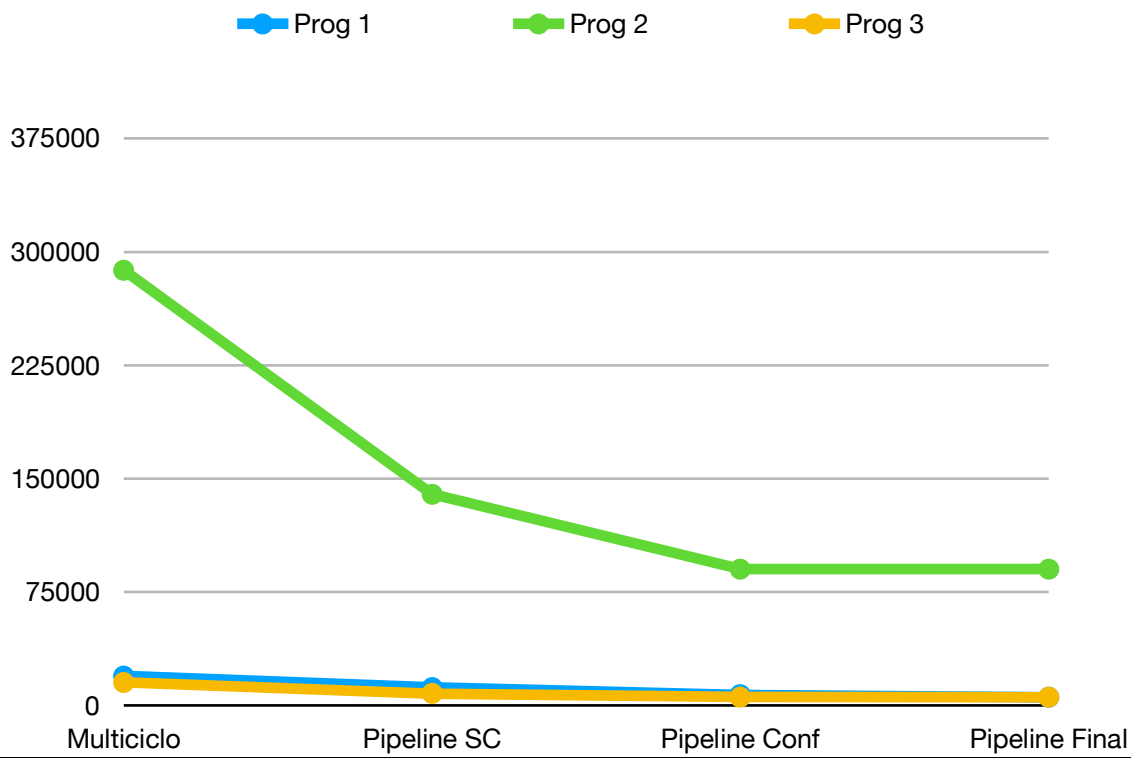
```
if (branch_buffer(i).address = diex.pc) then
   if ((branch_buffer(i).FSM = "00" or branch_buffer(i).FSM = "01") and salta = '1') then
      bidi.npc <= bidi.npc;
      bidi.IR <= (others=>'0');
      bidi.pc <= (others=>'0');
      exit;
   elsif ((branch_buffer(i).FSM = "10" or branch_buffer(i).FSM = "11") and salta = '0') then
      bidi.npc <= bidi.npc;
      bidi.IR <= (others=>'0');
      bidi.pc <= (others=>'0');
      exit;
```

Flush da barreira temporal DI/EX:

```
if (branch_buffer(i).address = diex.pc) then
   if ((branch_buffer(i).FSM = "00" or branch_buffer(i).FSM = "01") and salta = '1') then
      diex.npc <= (others=>'0');
      diex.pc <= (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;
      exit;
   elsif ((branch_buffer(i).FSM = "10" or branch_buffer(i).FSM = "11") and salta = '0')  then
      diex.npc <= (others=>'0');
      diex.pc <= (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;
      exit;
```

8. AVALIAÇÃO DE DESEMPENHO

Neste gráfico estão contidas as seguintes informações:



Programa	Tempo de Execução no Multiciclo	Tempo de Execução no Pipeline SC	Tempo de Execução no Pipeline Conf	Tempo de Execução no Pipeline Final
1	19540	11910	6930	5390
2	287840	139710	90190	90190
3	15300	7870	5450	5390

A partir do gráfico e da tabela acima, é possível verificar que a versão final do Pipeline, com a otimização do mecanismo de tratamento de conflito de controle, é capaz de diminuir ainda mais o tempo de execução dos programas testados.