Pontifícia Universidade Católica do
Rio Grande do Sul

Arquitetura de Computadores II

Trabalho III

Alunos	Diego Henrique Oliveira
	Ícaro Stumpf
	Leonardo Barbosa
Professor	Dr. Cesar Augusto Missio Marcon

Porto Alegre, Junho de 2021

Conteúdo

1 Introdução

O objetivo deste trabalho é implementar uma hierarquia de dados com três níveis (L1, L2 e MP), de forma a pôr em prática conhecimentos acerca do funcionamento de caches abordados em aula.

Ao final do trabalho, espera-se que os alunos tenham desenvolvido uma arquitetura semelhante à vista abaixo, além de cinco programas que validem o seu devido funcionamento.

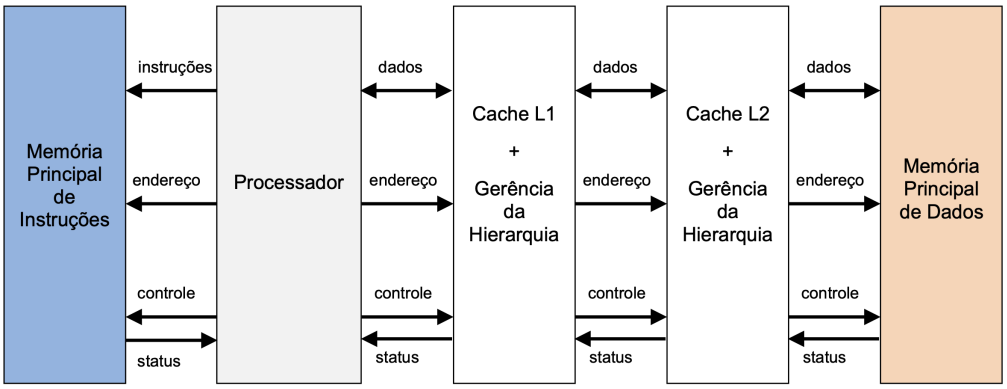


Figura 1: Arquitetura descrita na especificação do trabalho.

Para que o desenvolvimento da arquitetura solicitada fosse possível, os alunos optaram por fazer o uso, como base, da arquitetura MIPS Multiciclo, utilizada como objeto de ensino na disciplina de Organização de Computadores e desenvolvida pelo professor Ney Calazans. Foram feitas modificações na última versão da arquitetura original disponibilizada[1] pelo professor, de forma a suportar o uso e implementação das caches, além do trabalho necessário para que o acesso à memória de dados se tornasse síncrono. Além disso, foram criados dois programas em assembly, onde ambos fazem acesso a uma matriz, um que privilegiando a localidade espacial/temporal e outro operando de forma inversa, prejudicando a localidade espacial/temporal. Maiores detalhes acerca dos programas criados e modificações feitas na arquitetura multiciclo serão descritos nos tópicos abaixo.

[1] https://www.inf.pucrs.br/calazans/orgcomp_mat.html

2 Desenvolvimento

2.1 Arquiteturas alvo

Para cada arquitetura alvo, foram feitas modificações que melhor atendiam às especificações apresentadas para o desenvolvimento do trabalho. A arquitetura MP-0 se baseia, sumariamente, na arquitetura original produzida pelo professor Ney Calazans, sem modificações. Por conta disso, essa arquitetura não será abordada em um subtópico específico. As arquiteturas MP, MP+L1 e MP+L1+L2 sofreram modificações, que serão abordadas e descritas nos subtópicos abaixo. O acesso aos códigos-fonte dessas arquiteturas pode ser feito na íntegra ao acessar a pasta Arquiteturas, que está dentro da pasta compactada entregue junto deste relatório.

2.1.1 Arquitetura MP

Conforme descrito pelo professor César Marcon, a arquitetura MP é, de forma resumida, "A arquitetura MP-0 inserido atraso na memória principal." ... "Adicionalmente, toda a vez que o processador acessar a hierarquia de memória de dados, deve aguardar uma resposta de que o dado está disponível.". Portanto, com base nessa descrição, foram feitas modificações no código VHDL da MP-0, as quais serão vistas a seguir.

De forma a suportar o atraso na memória principal (MP), foi criado um novo sinal de relógio. Conforme especificação, a MP possui tempo de acesso de quatro ciclos de relógio do processador. Dessa forma, o *clock* da MP foi descrito conforme o código abaixo:

```
process
    begin
    mem_ck <= '1', '0' after 40 ns;
    wait for 80 ns;
end process;
```

Ademais, de forma a possibilitar a comunicação entre o processador e a MP, foram criados sinais de controle que fazem o papel de indicar ao processador que a operação de leitura ou escrita feita na memória foi finalizada e que a execução da instrução pode ser seguida. Esses sinais foram declarados como saídas na estrutura da MP e como entradas na estrutura do processador (mais especificamente, na unidade de controle). A lógica que faz o processador operar conforme o estado lógico desses sinais é descrita no código VHDL abaixo:

```
 -- fourth stage: data memory operation
when Sld => if (end_fetch = '0') then
                NS <= Sld;
            else
                NS <= Swbk;
            end if;

-- forth or fifth cycle: last for most instructions
when Sst => if (end_write = '0') then
                NS <= Sst;
            else
                NS <= Sfetch;
            end if;
```

É possível observar que, enquanto a operação de escrita/leitura não estiver finalizada, o processador não partirá para o próximo estado, ou seja, ficará no mesmo estado, em uma espécie de *loop*. Isso impede que o processador "atropele" o processamento da MP, de forma a "esperar" que ela termine o seu processamento.

Por fim, o uso do relógio da MP e da parametrização do sinal de controle end_write nas operações de escrita na MP pode ser visto no seguinte trecho de código VHDL:

```
process (ce_n, we_n, low_address, mem_ck, rst)
begin
  if (ce_n='0' and we_n='0') then
    if rst = '1' then
      if CONV_INTEGER(low_address)>=0 and CONV_INTEGER(low_address)
                                                <=MEMORY_SIZE-3 then
        if bw='1' then
              RAM(CONV_INTEGER(low_address+3)) <= data(31 downto 24);
              RAM(CONV_INTEGER(low_address+2)) <= data(23 downto 16);
              RAM(CONV_INTEGER(low_address+1)) <= data(15 downto  8);
            end if;
            RAM(CONV_INTEGER(low_address)) <= data(7 downto  0);
      end if;
    else
      if (mem_ck'event and mem_ck = '1') then
        if CONV_INTEGER(low_address)>=0 and CONV_INTEGER(low_address)
                                                <=MEMORY_SIZE-3 then
              if bw='1' then
                RAM(CONV_INTEGER(low_address+3)) <= data(31 downto 24);
                RAM(CONV_INTEGER(low_address+2)) <= data(23 downto 16);
                RAM(CONV_INTEGER(low_address+1)) <= data(15 downto  8);
              end if;
              RAM(CONV_INTEGER(low_address)) <= data(7 downto  0);
              end_write <= '1';
              end_write <= '0' after 25ns;
            end if;
      end if;
    end if;
  end if;
end process;
```

Pode-se notar que foi criado um processo sensível aos sinais de controle de escrita na memória, ao sinal de *reset* da CPU e ao sinal de *clock* da MP. As

condicionais estão dispostas da forma vista acima porque a escrita de dados na MP durante a inicialização da arquitetura, enquanto o sinal de *reset* está em nível lógico alto, deve somente ser feita de forma assíncrona, já que essa inicialização e manipulação da MP não é feita pelo processador em si, mas sim pelo próprio *testbench*, em um tempo muito curto. Por outro lado, caso a inicialização do sistema já tenha sido concluída, sabe-se que o acesso deve ser feito conforme o relógio parametrizado para a MP, como pode ser visto no código. Ao fim da escrita na memória, o sinal end_write é acionado e o processador, em seu próximo ciclo de relógio, detecta essa mudança no sinal e a execução da instrução é prosseguida.

De forma análoga, o uso do relógio da MP e da parametrização do sinal de controle end_fetch nas operações de leitura na MP pode ser visto no seguinte trecho de código:

```
process(mem_ck, rst)
begin
  if rst = '1' then
   data(31 downto 24) <= (others=>'Z');
   data(23 downto 16) <= (others=>'Z');
   data(15 downto  8) <= (others=>'Z');
   data( 7 downto  0) <= (others=>'Z');
   end_fetch <= '0';
  elsif mem_ck'event and mem_ck = '1' then
    if ce_n='0' and oe_n='0' and CONV_INTEGER(low_address)>=0 and
                                CONV_INTEGER(low_address)<=MEMORY_SIZE-3 then
      data(31 downto 24) <= RAM(CONV_INTEGER(low_address+3));
      data(23 downto 16) <= RAM(CONV_INTEGER(low_address+2));
      data(15 downto  8) <= RAM(CONV_INTEGER(low_address+1));
      data(7 downto  0) <= RAM(CONV_INTEGER(low_address));
      end_fetch <= '1';
      end_fetch <= '0' after 25ns;
    else
      data(31 downto 24) <= (others=>'Z');
      data(23 downto 16) <= (others=>'Z');
      data(15 downto  8) <= (others=>'Z');
      data( 7 downto  0) <= (others=>'Z');
    end if;
  end if;
end process;
```

Nota-se a criação um processo sensível aos sinais de *reset* da CPU e ao

sinal de *clock* da MP. Caso haja um evento de relógio da MP e os sinais de controle de leitura na memória estejam em '1', a leitura da MP é feita, assim como a escrita para o barramento de dados do processador. Logo após, o sinal end_fetch é acionado, possibilitando que o processador continue a execução da instrução em seu próximo ciclo de relógio.

Essa arquitetura foi validada por completo através da execução completa de cinco programas assembly solicitados pelo professor.

2.1.2 Arquitetura MP+L1

Conforme descrito pelo professor César Marcon, a arquitetura MP+L1 é, de forma resumida, "A arquitetura MP com a interposição de uma cache de nível 1 entre o processador e a memória principal." ... "A cache L1 deve operar na mesma frequência do processador (acesso com um ciclo de relógio). Para implementação das caches de dados devem ser consideradas um tipo de mapeamento e um mecanismo para manter a integridade de dados, tal como descrito a seguir: (i) Cache L1 - mapeamento direto como write-through." Visualmente, a arquitetura MP+L1 deve ser algo semelhante ao que pode ser visto na figura abaixo:

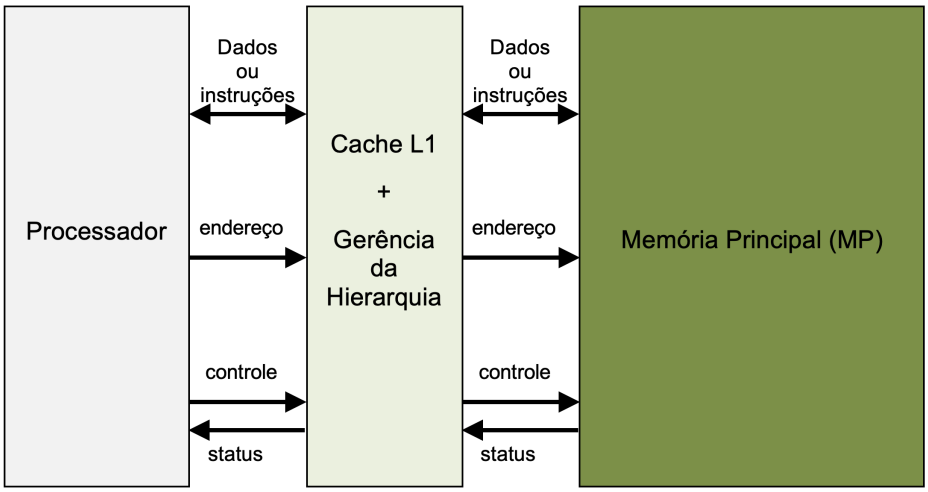


Figura 2: Arquitetura MP+L1.

Portanto, com base nessa descrição, foram feitas modificações no código VHDL da MP, as quais serão vistas a seguir.

De forma a descrever programaticamente a cache L1 com base nas aulas dadas, foram criados os seguintes tipos (*type*):

```
type word is array(0 to 15) of wires8;

type wt_memitem is record
    validation_bit: std_logic;
        tag: std_logic_vector(25 downto 0);
        words : word;
end record;

type L1_Cache is array(0 to 3) of wt_memitem;
```

O tipo word foi criado de forma a representar os blocos de palavras de cada linha da cache. Foi especificado na descrição do trabalho que cada linha das caches L1 e L2 comporta 4 palavras de 32 bits. Ao longo do desenvolvimento do trabalho, após consulta na documentação[2] da MIPS Multiciclo, foi descoberto que o endereçamento de memória da arquitetura é feito a byte, ou seja, cada palavra de 32 bits ocupa um conjunto de 4 posições de memória consecutivas. Com isso em mente, optou-se por fazer com que os blocos das linhas da cache comportassem um total de 16 bytes, que, no fim das contas, são as 4 palavras de 32 bits exigidas na descrição do trabalho divididas em bytes. Isso foi feito de forma a alinhar o endereçamento da cache com o da MP.

O tipo wt_memitem deriva de *write-through* (técnica para manter a integridade de dados escolhida pelo professor para a cache L1) *item*. Ademais, a cache L1 possui o mapeamento direto como forma de mapeamento de endereços. Ou seja, esse tipo representa uma linha, de forma individual, da cache L1. Esse tipo comporta um sinal que representa o bit de validação, um sinal de 26 bits que representa a TAG e uma instância do tipo word, que representa o bloco de palavras daquela linha.

O tipo L1_Cache é, em suma, um vetor de 4 instâncias do tipo anterior, portanto, é uma cache L1 (conforme especificação, a cache L1 possui 4 linhas). Dessa forma, se tem que um endereço transmitido pelo processador à cache L1 é traduzido da seguinte forma: os 26 bits mais altos do endereço seriam mapeados na TAG, os 2 bits seguintes na respectiva linha da cache e os seus últimos 4 bits na posição do bloco de palavras da linha da cache.

Além dos sinais vistos na descrição da arquitetura MP, foi criado um sinal de relógio para a cache L1 (que opera na mesma frequência do processador, com acesso em um ciclo de relógio) e os seguintes sinais de controle:

```
signal L1Cache : L1_Cache := (others => initialize_wt_memitem);
```

[2] https://www.inf.pucrs.br/calazans/undergrad/orgcomp_EC/mips_multi/Arq_MIPS_Multiciclo_Spec.pdf

```
signal searchRAM, writeRAM, writeCPU : std_logic := '0';
```

O sinal L1Cache representa a instância do tipo L1_Cache descrito anteriormente, com todos os seus sinais inicializados em zero. Os sinais seguintes serão descritos ao longo da descrição do algoritmo de funcionamento da cache L1.

De forma a simplificar o entendimento do código VHDL que descreve o algoritmo de mapeamento direto como *write-through* da cache L1, é válido relembrar como é dado o funcionamento do algoritmo *write-through*:

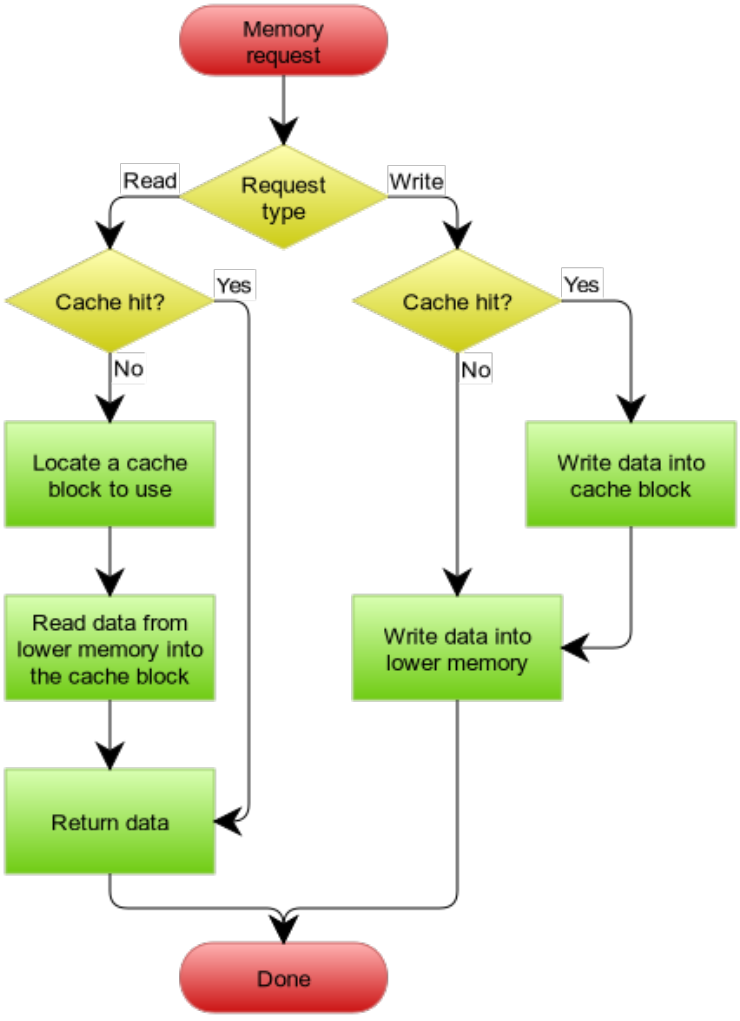


Figura 3: Diagrama explicativo do método write-through.

Portanto, cabe trazer os trechos de código:

```
...
elsif l1_ck'event and l1_ck = '1' then
  if ce_n='0' and oe_n='0' and
       CONV_INTEGER(low_address)>=0 and CONV_INTEGER(low_address)<=MEMORY_SIZE-3 then
    if writeCPU = '1' then
      data(31 downto 24) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+3));
      data(23 downto 16) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+2));
      data(15 downto 8) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+1));
      data(7 downto 0) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)));
      end_fetch <= '1';
      end_fetch <= '0' after 25ns;
    elsif L1Cache(CONV_INTEGER(low_address(5 downto 4))).validation_bit = '1' then
      if L1Cache(CONV_INTEGER(low_address(5 downto 4))).tag = tmp_address(31 downto 6) then
        data(31 downto 24) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+3));
        data(23 downto 16) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+2));
        data(15 downto 8) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)+1));
        data(7 downto 0) <= L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
                                        (CONV_INTEGER(low_address(3 downto 0)));
        end_fetch <= '1';
        end_fetch <= '0' after 25ns;
      else
        searchRAM <= '1';
        searchRAM <= '0' after 45ns;
      end if;
    else
      searchRAM <= '1';
      searchRAM <= '0' after 45ns;
    end if;
else
  data(31 downto 24) <= (others=>'Z');
  data(23 downto 16) <= (others=>'Z');
  data(15 downto  8) <= (others=>'Z');
  data( 7 downto  0) <= (others=>'Z');
end if;
```

...

Como é possível observar, o trecho de código exibido acima é extenso e engloba apenas o cenário de leitura na cache. Em um evento de *clock* da cache, é verificado se os sinais de controle de leitura estão ativos para, posteriormente verificar se o sinal writeCPU está em '1'. Esse sinal é setado quando a cache sofre um *miss* de leitura e os dados já foram buscados no nível abaixo dela (neste caso, a MP) para, por fim, serem escritos no barramento de dados do processador. Nota-se que o barramento de dados do processador possui 32 bits, então a escrita é feita byte a byte nesse barramento (da forma como já era feito na arquitetura original, MP-0). Caso writeCPU não esteja em '1', é verificado se o bit de validação da respectiva linha endereçada pelo processador é igual à '1'. Se positivo, é verificado se a TAG endereçada pelo processador é a mesma que está armazenada na linha da cache. Se positivo, tem-se um *hit* de leitura. Com isso, o dado é escrito no barramento de dados do processador e o sinal end_fetch é setado para '1', permitindo que o processador continue a execução da instrução. Caso negativo, houve um *miss* de leitura. Dessa forma, o sinal searchRAM é setado para '1'. O nome do sinal por si só já o descreve, mas é válida a explicação: este sinal "avisará" para a MP que ela sofrerá busca, já que houve *miss* e a cache deve procurar os dados no nível abaixo dela.

O trecho de código abaixo descreve a busca de dados na MP, após um *miss* de leitura na cache:

```
...
elsif mem_ck'event and mem_ck = '1' then
  if searchRAM = '1' then
    tmp16(15 downto 4) := tmp_address(15 downto 4);
    tmp16(3 downto 0) := "0000";
    for i in 0 to L1Cache(CONV_INTEGER(low_address(5 downto 4))).words'length-1 loop
      L1Cache(CONV_INTEGER(low_address(5 downto 4))).words(i) <= RAM(CONV_INTEGER(tmp16));
      tmp16:= tmp16+1;
    end loop;
    L1Cache(CONV_INTEGER(low_address(5 downto 4))).validation_bit <= '1';
    L1Cache(CONV_INTEGER(low_address(5 downto 4))).tag <= tmp_address(31 downto 6);
    writeCPU <= '1';
    writeCPU <= '0' after 20ns;
  end if;
end if;
...
```

É possível observar que, em uma borda de relógio da MP, é verificado se o sinal searchRAM está em '1' (indicando que houve *miss* de leitura na cache). Caso positivo, um sinal temporário é criado para que seja possível iterar sobre o bloco de palavras da linha da cache e atribuir os dados corretos da MP para as devidas posições desse bloco. Dessa forma, a cache é escrita com os dados da MP, o bit de validação da linha é setado para '1' e a TAG da linha é atribuída com o respectivos bits endereçados pela MP. Ademais, o sinal writeCPU é setado para '1', de forma a possibilitar que a cache escreva o dado solicitado pelo processador em seu barramento de dados.

O trecho de código a seguir descreve o cenário onde uma solicitação de escrita é vinda para a cache:

```
...
elsif l1_ck'event and l1_ck = '1' then
  if ce_n='0' and we_n='0' and CONV_INTEGER(low_address)>=0 and
                                CONV_INTEGER(low_address)<=MEMORY_SIZE-3 then
    if L1Cache(CONV_INTEGER(low_address(5 downto 4))).validation_bit = '1' then
      if L1Cache(CONV_INTEGER(low_address(5 downto 4))).tag = tmp_address(31 downto 6) then
        if bw='1' then
          L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
             (CONV_INTEGER(low_address(3 downto 0)+3)) <= data(31 downto 24);
          L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
             (CONV_INTEGER(low_address(3 downto 0)+2)) <= data(23 downto 16);
          L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
             (CONV_INTEGER(low_address(3 downto 0)+1)) <= data(15 downto 8);
        end if;
        L1Cache(CONV_INTEGER(low_address(5 downto 4))).words
            (CONV_INTEGER(low_address(3 downto 0))) <= data(7 downto 0);
        writeRAM <= '1';
        writeRAM <= '0' after 45ns;
      else
        writeRAM <= '1';
        writeRAM <= '0' after 45ns;
      end if;
    else
      writeRAM <= '1';
      writeRAM <= '0' after 45ns;
    end if;
  end if;
end if;
```

...

De forma breve, o código acima verifica, em um evento de relógio da cache, se os sinais de controle de escrita estão em nível lógico alto. Caso positivo, são feitas as mesmas verificações vistas no trecho de código anterior: bit de validação da linha em '1', TAG endereçada pelo processador coincidente com a armazenada na respectiva linha da cache. Caso positivo, tem-se um *hit* de escrita. Como a técnica para manter a integridade de dados para essa cache é a *write-through*, o dado a ser escrito é atualizado na cache e passado para o nível de baixo (através do sinal writeRAM, que é setado para '1'), ou seja, o dado será escrito na memória principal independentemente se haja *miss* ou *hit* de escrita.

Por fim, não convém exibir o trecho de código onde a escrita na MP é feita, já que este é extremamente similar ao código exibido na descrição da arquitetura MP, com exceção do sinal writeRAM, que é o responsável por informar à memória principal se ela sofrerá escrita ou não. Ou seja, em um evento de relógio da MP, é verificado se o sinal writeRAM está em '1', para, em caso positivo, efetuar a escrita do dado na MP.

Essa arquitetura foi validada por completo através da execução completa de cinco programas assembly solicitados pelo professor.

2.1.3 Arquitetura MP+L1+L2

Conforme descrito pelo professor César Marcon, a arquitetura MP+L1+L2 é, de forma resumida, "A arquitetura MP+L1 com a interposição de uma cache de nível 2 entre a cache de nível 1 e a memória principal." ... "A cache L2 tem tempo de acesso de 2 ciclos de relógio. Para implementação das caches de dados devem ser consideradas um tipo de mapeamento e um mecanismo para manter a integridade de dados, tal como descrito a seguir: (ii) Cache L2 - mapeamento associativo como write-back, e regra de substituição com contador no caso de miss com cache tendo todas as posições ocupadas." O desenvolvimento dessa arquitetura foi considerado o mais desafiador para o grupo. Visualmente, a arquitetura MP+L1+L2 deve ser algo semelhante ao que pode ser visto na figura abaixo:

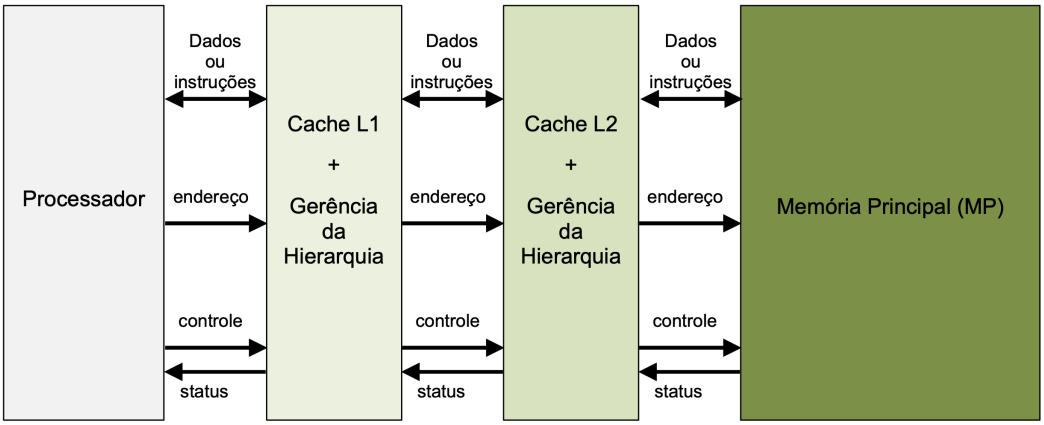


Figura 4: Arquitetura MP+L1+L2.

Portanto, com base nessa descrição, foram feitas modificações densas e extensas no código VHDL da MP, as quais serão vistas a seguir.

De forma a suportar o atraso na cache nível 2 (L2), foi criado um novo sinal de relógio. Conforme especificação, a cache L2 possui tempo de acesso de dois ciclos de relógio do processador. Dessa forma, o *clock* da cache L2 foi descrito conforme o código abaixo:

```
process
    begin
    l2_ck <= '1', '0' after 20 ns;
    wait for 40 ns;
end process;
```

De forma a descrever programaticamente a cache L2 com base nas aulas dadas, foram criados os seguintes tipos (*type*):

```
type wb_memitem is record
    validation_bit: std_logic;
    dirty_bit: std_logic;
    tag: std_logic_vector(27 downto 0);
    words : word;
end record;

type L2_Cache is array(0 to 7) of wb_memitem;
```

O tipo wb_memitem deriva de *write-back* (técnica para manter a integridade de dados escolhida pelo professor para a cache L2) *item*. Ademais, a

cache L2 possui o mapeamento associativo como forma de mapeamento de endereços, além de possuir regra de substituição com contador no caso de miss com cache tendo todas as posições ocupadas. Ou seja, esse tipo representa uma linha, de forma individual, da cache L2. Esse tipo comporta um sinal que representa o bit de validação, um sinal que representa o bit de sujeira, um sinal de 28 bits que representa a TAG e uma instância do tipo word, que representa o bloco de palavras daquela linha.

O tipo L2_Cache é, em suma, um vetor de 8 instâncias do tipo anterior, portanto, é uma cache L2 (conforme especificação, a cache L2 possui 8 linhas). Dessa forma, se tem que um endereço transmitido pelo processador à cache L2 é traduzido da seguinte forma: os 28 bits mais altos do endereço seriam mapeados na TAG e os 4 bits seguintes na posição do bloco de palavras da linha da cache.

Além dos sinais vistos na descrição da arquitetura MP+L1, foram criados os seguintes sinais de controle:

```
...
writeRAM_WB: out std_logic := '0';
...
signal L2Cache : L2_Cache := (others => initialize_wb_memitem);
signal writeback, writeL1, writeL2, writeL2_WB, searchL2,
         searchRAM_WB, write_done : std_logic := '0';
signal next_to_replace : integer := 0;
```

O sinal L2Cache representa a instância do tipo L2_Cache descrito anteriormente, com todos os seus sinais inicializados em zero. Os sinais seguintes serão descritos ao longo da descrição do algoritmo de funcionamento da cache L2. Cabe pontuar, porém, que o sinal next_to_replace representa o contador da regra de substituição na cache L2, descrita anteriormente.

De forma a simplificar o entendimento do código VHDL que descreve o algoritmo de mapeamento associativo como *write-back* da cache L2, é válido relembrar como é dado o funcionamento do algoritmo *write-back*:

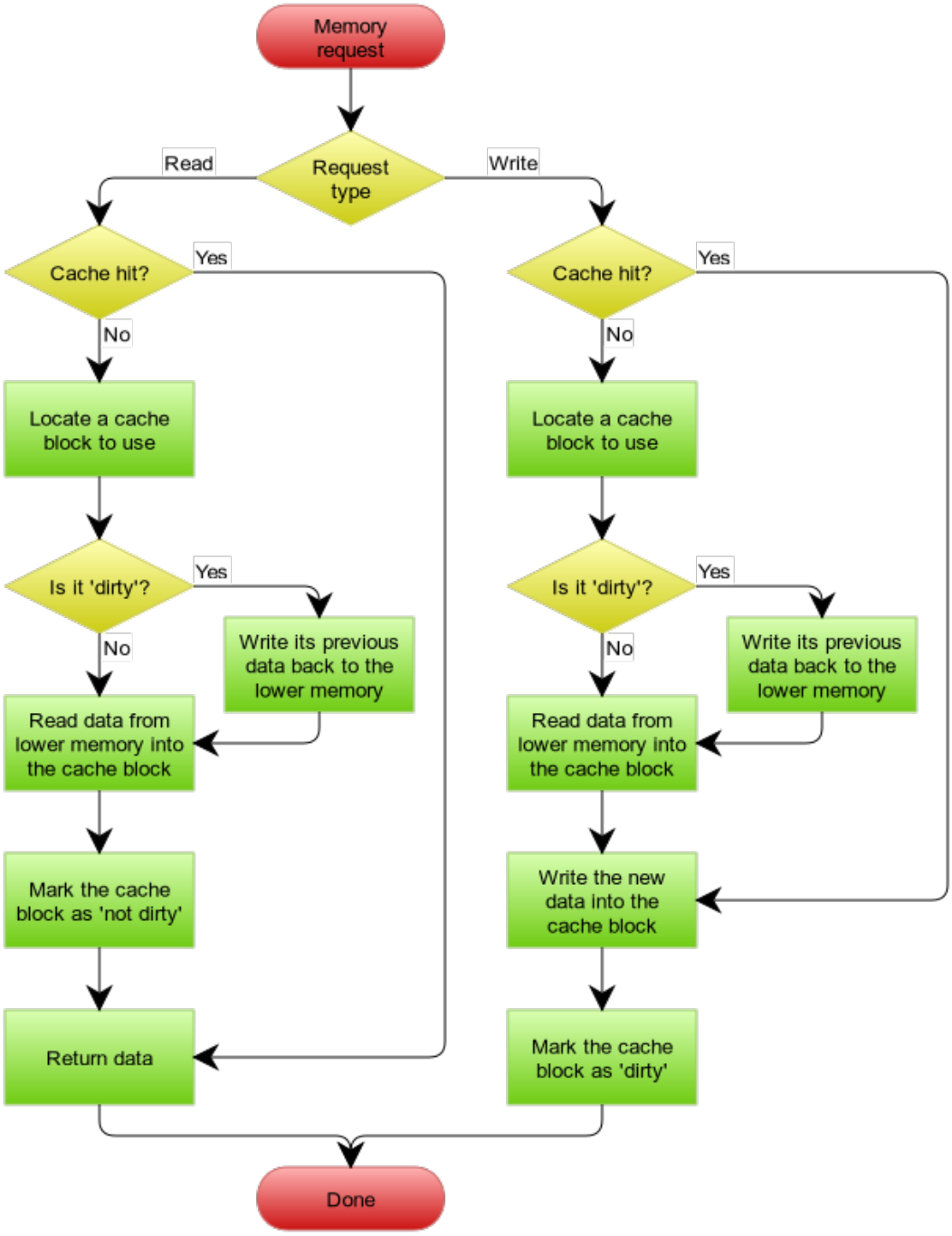


Figura 5: Diagrama explicativo do método write-back.

Portanto, cabe trazer os (extensos) trechos de código:

O processo de leitura na cache L1 é idêntico ao processo de leitura visto na arquitetura anterior. Entretanto, caso haja *miss*, ao invés da procura ser na MP, nessa arquitetura a procura será na cache L2 (que está um nível abaixo dela). Com isso, o sinal searchL2 será setado em '1' e a cache L2 será "avisada" de que uma procura irá ocorrer nela em seu próximo ciclo de relógio. Além disso, uma variável chamada searchL2_done assegura de que a cache L1 não sofra múltiplas leituras de forma desnecessária na mesma instrução (isso poderia acontecer em caso de *miss*, enquanto níveis abaixo dela seriam lidos). Além da inclusão desses dois sinais descritos, o processo de leitura na cache L1 segue sendo de forma similar ao visto anteriormente. Portanto, não há necessidade de reexibir aqui o extenso trecho de código que já foi explicado.

Entretanto, cabe exibir o trecho de código que ilustra a busca de leitura na cache L2, que ocorre em caso de *miss* de leitura na cache L1:

```
...
elsif l2_ck'event and l2_ck = '1' then
...
  elsif (searchL2 = '1' and searchRAM = '0') then
    tmp4 := "0000";
    for i in 0 to L2Cache'length-1 loop
      if L2Cache(i).validation_bit = '1' then
        if L2Cache(i).tag = tmp_address(31 downto 4) then
          for j in 0 to L1Cache(CONV_INTEGER(low_address(5 downto 4))).words'length-1 loop
            L1Cache(CONV_INTEGER(low_address(5 downto 4))).words(j)
                                              <= L2Cache(i).words(CONV_INTEGER(tmp4));
            tmp4:= tmp4+1;
          end loop;
          L1Cache(CONV_INTEGER(low_address(5 downto 4))).validation_bit <= '1';
          L1Cache(CONV_INTEGER(low_address(5 downto 4))).tag <= tmp_address(31 downto 6);
          found := true;
          exit;
        end if;
      end if;
    end loop;
    if found = true then
      writeCPU <= '1';
      writeCPU <= '0' after 20ns;
      found := false;
```

```
    else
      searchRAM <= '1';
      searchRAM <= '0' after 85ns;
    end if;
  end if;
end if;
...
```

O algoritmo é simples: em um evento de relógio da cache L2 é verificado se o endereço do dado transmitido pelo processador se encontra na cache L2 (através de um laço que percorre toda a cache L2, que verifica o bit de validade da linha e se a TAG é coincidente com a endereçada pelo processador). Caso positivo, tem-se um *hit* e o bloco inteiro de palavras da respectiva linha da cache L1 é escrito com os dados que estão no bloco da linha encontrada da cache L2 (através de um laço que escreve palavra a palavra do bloco da cache L1 e, ao fim do laço, escreve a TAG endereçada e seta o bit da validade em '1' da respectiva linha). Com isso, graças ao auxílio de um booleano que, quando em *true*, indica que houve *hit* e, em *false*, indica que houve *miss*, é decidido qual sinal de controle será setado. Em caso de *hit*, sabe-se que a cache L1 foi escrita com os valores buscados e o sinal writeCPU é setado para '1', possibilitando que o barramento de dados do processador seja escrito com o valor agora armazenado na cache L1 (de forma idêntica à ocorrida no trecho de código visto na arquitetura anterior). Em caso de *miss*, sabe-se que a cache L2 deverá buscar dados no nível abaixo dela, neste caso, a MP. Portanto, o sinal searchRAM é setado para '1', "avisando", à MP que esta sofrerá busca por parte da cache L2, de forma similar à vista na arquitetura passada.

O trecho de código que faz a busca na MP por parte da cache L2 é bastante similar ao trecho visto na arquitetura anterior, apenas mudando o fato de que a cache L2 que sofrerá escrita por parte da MP, e não a cache L1. Com isso, supondo que a cache L2 havia espaço para escrita e esta foi feita com sucesso, o sinal writeL1 é setado para '1', avisando à cache L1 que essa pode ser escrita com os dados recém escritos na cache L2, para, dessa forma, poder escrever o dado correto no barramento de dados do processador.

Entretanto, cabe exibir o trecho de código que lida com uma situação típica do mapeamento associativo: o fato de, caso a cache L2 esteja cheia, a substituição nela ser feita com base na posição que o contador next_to_replace está apontando. Vejamos:

```
...
elsif mem_ck'event and mem_ck = '1' then
  ...
  elsif (searchRAM = '1' or searchRAM_WB = '1') then
  ...
    else
      if L2Cache(next_to_replace).dirty_bit = '1' then
        writeRAM_WB <= '1';
        writeRAM_WB <= '0' after 85ns;
        writeRAM <= '1';
        writeRAM <= '0' after 85ns;
      else
        for i in 0 to L2Cache(next_to_replace).words'length-1 loop
          L2Cache(next_to_replace).words(i) <= RAM(CONV_INTEGER(tmp16));
          tmp16:= tmp16+1;
        end loop;
        L2Cache(next_to_replace).validation_bit <= '1';
        L2Cache(next_to_replace).tag <= tmp_address(31 downto 4);
        if searchRAM = '1' then
          writeL1 <= '1';
          writeL1 <= '0' after 45ns;
        else
          writeL2_WB <= '1';
          writeL2_WB <= '0' after 45ns;
        end if;
        if next_to_replace = 7 then
          next_to_replace <= 0;
        else
          next_to_replace <= next_to_replace + 1;
        end if;
      end if;
    end if;
  end if;
end if;
```

Como pode ser notado, há uma certa complexidade nesse cenário: antes de fazer a substituição na posição da cache L2 apontada pelo contador next_to_replace, deve ser verificado se o bit de sujeira dessa posição é igual a '1'. Caso positivo, tem-se uma situação que causará maior atraso ainda: todos os dados que estão no bloco dessa linha da cache L2 deverão obrigatoriamente serem escritos na MP, de forma a não perder de forma definitiva

os dados escritos e a integridade deles seja mantida. Portanto, dois sinais são setados para '1', que operam de forma similar: o sinal writeRAM_WB faz a comunicação necessária ao mundo externo para que o sinal de controle de escrita seja ativado, já que, neste cenário, o processador está executando uma instrução de leitura e apenas os sinais de controle de leitura estão ativos. O sinal writeRAM, já conhecido da arquitetura anterior, faz o mesmo que fazia anteriormente: avisa à MP que ela sofrerá escrita, mas, nessa arquitetura, ela sofrerá escrita da cache L2. Por outro lado, caso o bit de sujeira da posição apontada por next_to_replace seja '0', a substituição na cache L2 já poderá ser feita sem que haja comprometimento dos dados escritos na respectiva linha. Portanto, os dados são escritos na respectiva linha da cache L2, o contador next_to_replace é apontado para a próxima posição que deve substituir futuramente e o sinal writeL1 é setado para '1', indicando à cache L1 que esta poderá sofrer escrita, para, enfim, poder escrever o dado correto no barramento de dados do processador.

Para manter a coerência com a descrição anterior, é valido comentar sobre o trecho de código que efetua a escrita de dados da cache L2 na MP. É valido ressaltar que isso ocorre apenas em cenários onde o bit de sujeira da linha a ser lida ou escrita da cache L2 esteja em '1'. Dessa forma, para manter a integridade dos dados, estes são escritos na MP, que é o conjunto abaixo da cache L2. Entretanto, o trecho de código no qual essa escrita na MP é feita é extremamente similar ao visto na arquitetura anterior, com exceção da fonte dos dados a serem escritos, que nessa arquitetura é a cache L2. Além disso, o sinal writeback é setado para '1' após a escrita dos dados na MP ser efetuada, "avisando" à MP que esta escreverá dados na cache L2 em seu próximo ciclo de relógio. Por conta dos motivos descritos, não convém mostrar o trecho de código que efetua a escrita na MP, dada a similaridade com o código visto anteriormente, salvas as exceções já explicadas neste parágrafo. Cabe, porém, exibir o trecho de código no qual ocorre o passo seguinte ao da escrita na MP, onde o sinal writeback está em '1':

```
...
elsif mem_ck'event and mem_ck = '1' then
  if (writeback = '1') then
    for i in 0 to L2Cache(next_to_replace).words'length-1 loop
          L2Cache(next_to_replace).words(i) <= RAM(CONV_INTEGER(tmp16));
      tmp16:= tmp16+1;
    end loop;
    L2Cache(next_to_replace).validation_bit <= '1';
    L2Cache(next_to_replace).tag <= tmp_address(31 downto 4);
```

```
    L2Cache(next_to_replace).dirty_bit <= '0';
    if searchRAM_WB = '1' then
      writeL2_WB <= '1';
      writeL2_WB <= '0' after 45ns;
    else
      writeL1 <= '1';
      writeL1 <= '0' after 45ns;
    end if;
    if next_to_replace = 7 then
      next_to_replace <= 0 after 10ns;
    else
      next_to_replace <= next_to_replace + 1 after 10ns;
    end if;
    ...
  end if;
end if;
```

Fica evidente a similaridade deste trecho de código com o trecho de código no qual a cache L2 faz a busca de dados na MP em caso de *miss* de leitura, exibido anteriormente. A diferença neste cenário é que há a certeza de que o dado será escrito na cache L2, já que estará havendo a substituição na linha da cache apontada por next_to_replace. Portanto, os dados são escritos na respectiva linha da cache L2, assim como o contador next_to_replace é atualizado. Como essa substituição na cache L2 pode ocorrer em dois cenários diferentes (tanto em instruções de leitura, quanto em instruções de escrita na memória), é verificado em qual cenário se está inserido. Caso o sinal searchRAM_WB esteja em '1', significa que está se executando uma instrução de escrita, então o sinal writeL2_WB é setado para '1', indicando à cache L2 que, em seu próximo ciclo de relógio, o dado enviado pelo processador será escrito em sua estrutura. Caso contrário, sabe-se que se trata da execução de uma instrução de leitura. Com isso, o sinal writeL1 é setado para '1', indicando à cache L2 que, em seu próximo ciclo de relógio, os dados recém escritos em sua estrutura serão escritos na cache L1, para que, posteriormente, o barramento de dados do processador possa ler o dado solicitado na cache L1 de forma correta.

Os trechos de código mostrados até o momento englobam o fluxo de instruções de leitura vindas do processador, seus acessos nas caches/MP e seus possíveis cenários de desvio (*miss* de leitura na L1, *miss* de leitura na L2, L2 cheia para busca na MP e posterior escrita, linha da L2 a ser substituída com o bit de sujeira em '1', etc.), que são muitos e, também, extensos. A

partir deste trecho, será mostrado o fluxo de instruções de escrita vindas do processador e seus acessos nas caches/MP:

Quando uma solicitação de escrita nas memórias chega, a primeira memória que terá acesso ao dado a ser escrito é a cache L1. Assim como na arquitetura passada, por conta da técnica para manter a integridade de dados escolhida para a cache L1 (*write-back*, vale lembrar), o dado a ser escrito irá ser passado para o nível abaixo dela independente de *miss* ou *hit*. Caso haja *hit*, o dado será atualizado na respectiva posição da cache L1 e será passado para o nível abaixo (sabe-se que a cache L2 fica no nível abaixo ao da cache L1). Portanto, o trecho de código que descreve essa situação não será exibido por ser muito similar ao trecho visto na arquitetura passada. A sutil diferença fica na presença do sinal de controle writeL2, que é setado para '1', de forma a informar à cache L2 que ela sofrerá escrita, seja após a atualização do dado na cache L1, quanto em caso de *miss*.

Assim, convém apresentar o trecho de código no qual a busca e escrita (em caso de *hit*) na cache L2 é feita:

```
...
elsif l2_ck'event and l2_ck = '1' then
  if ce_n='0' and we_n='0' and CONV_INTEGER(low_address)>=0
        and CONV_INTEGER(low_address)<=MEMORY_SIZE-3 and writeRAM = '0' then
    if (writeL2 = '1' or writeL2_WB = '1') then
      for i in 0 to L2Cache'length-1 loop
        if L2Cache(i).validation_bit = '1' then
          if L2Cache(i).tag = tmp_address(31 downto 4) then
            if bw='1' then
              L2Cache(i).words(CONV_INTEGER(low_address(3 downto 0)+3))
                                                    <= data(31 downto 24);
              L2Cache(i).words(CONV_INTEGER(low_address(3 downto 0)+2))
                                                    <= data(23 downto 16);
              L2Cache(i).words(CONV_INTEGER(low_address(3 downto 0)+1))
                                                    <= data(15 downto 8);
            end if;
            L2Cache(i).words(CONV_INTEGER(low_address(3 downto 0)))
                                                    <= data(7 downto 0);
            L2Cache(i).dirty_bit <= '1';
            found := true;
            exit;
          end if;
        end if;
      end loop;
```

```
      if found = true then
        end_write <= '1';
        end_write <= '0' after 25ns;
        write_done <= '1';
        write_done <= '0' after 25ns;
        found := false;
      else
        searchRAM_WB <= '1';
        searchRAM_WB <= '0' after 45ns;
      end if;
    end if;
end if;
```

O algoritmo do código acima é simples: em um evento de relógio da cache L2 é verificado se os sinais de controle de escrita estão ativos e, com isso, verifica-se o estado do sinal writeL2 ou do sinal writeL2_WB é igual a '1'. Caso sim, é iniciada a busca na cache L2: caso haja *hit*, o booleano found é setado para "true", caso contrário, é setado para "false". Sabendo o estado do booleano found, é definido qual será o próximo passo: caso seja "true", sabe-se que a cache L2 foi atualizada com o dado a ser escrito e o processador pode continuar a execução da instrução (sinal end_write é setado para '1', possibilitando que isso aconteça). Caso contrário, sabe-se que houve um *miss* de escrita. Com isso, o sinal searchRAM_WB é setado para '1', de forma a avisar à MP que ela sofrerá busca por parte da cache L2.

Assim como dito anteriormente, o trecho de código que faz a busca na MP por parte da cache L2 é bastante similar ao trecho visto na arquitetura anterior, apenas mudando o fato de que a cache L2 que sofrerá escrita por parte da MP, e não a cache L1. Com isso, supondo que a cache L2 havia espaço para escrita e esta foi feita com sucesso, o sinal writeL2_WB é setado para '1', avisando à cache L2 que ela pode ser atualizada com o dado contido no barramento de dados do processador, de forma a finalizar o ciclo de escrita e a continuação da execução da instrução pelo processador.

Há, também, a possibilidade de não haver espaço para armazenamento dos dados na cache L2 e, por conta disso, a posição apontada pelo sinal next_to_replace será a posição escolhida para ser substituída pelos dados novos na cache. Os passos seguintes já são conhecidos: são os mesmos passos vistos no fluxo de leitura, com os mesmos trechos de código, algoritmos e decisões tomadas. O que muda é que, no fim da substituição dos dados na posição da cache L2 apontada pelo sinal next_to_replace, o sinal writeL2_WB é setado para '1', de forma a avisar à cache L2 de que ela pode escrever em

sua estrutura o dado enviado pelo processador através do seu barramento de dados, podendo assim seguir com a execução da instrução. Por conta do trecho de código que descreve o que foi dito acima já se encontrar na presente subseção, não há a necessidade de reapresentá-lo.

Essa arquitetura foi validada por completo através da execução completa de cinco programas assembly solicitados pelo professor. Assim, é finalizada a apresentação das arquiteturas alvo desenvolvidas no presente trabalho. No tópico seguinte, serão apresentados os programas utilizados para teste das arquiteturas, junto com um gráfico comparativo dos seus tempos de execução em cada arquitetura.

2.2 Programas desenvolvidos

De forma a validar o funcionamento das arquiteturas alvo desenvolvidas e apresentadas no tópico anterior, foram desenvolvidos cinco programas assembly que fizessem uso de instruções de leitura e escrita na memória. Foi solicitado que dois desses programas deveriam explorar a localidade temporal/espacial de forma a privilegiá-la e outro prejudicando-a. Os outros três programas poderiam ser de qualquer fonte, desde que explorassem o acesso às memórias. O código fonte dos programas pode ser conferido ao acessar a pasta Programas, que está dentro da pasta compactada entregue junto deste relatório. Tais programas serão brevemente explicados a seguir.

2.2.1 prog1.asm

O primeiro programa foi desenvolvido pelos alunos que estão desenvolvendo o presente trabalho, quando cursaram Arquitetura de Computadores I, como um entregável para um trabalho da disciplina. Conforme descrição feita pelo professor César Marcon na especificação dos trabalhos a serem feitos na já citada disciplina, o programa faz o seguinte:

"Um arquivo descrevendo um algoritmo que a partir de dois vetores de entrada ENT_1 e ENT_2, de mesmo tamanho, gera um terceiro vetor de saída SAI, igualmente de mesmo tamanho. Cada elemento do vetor SAI deve ser o resultado de uma operação entre elementos de mesma posição dos vetores ENT_1 e ENT_2. Esta operação será uma soma sempre que os elementos dos dois vetores forem igualmente pares ou igualmente ímpares, caso contrário, será uma subtração.".

2.2.2 prog2.asm

O segundo programa foi desenvolvido por Germano Mergel e Bruno Campos em 2004 e foi obtido através da página[3] de Material de Apoio da disciplina de Organização de Computadores, hospedada pelo professor Ney Calazans. O programa faz o cálculo dos elementos comuns de dois vetores, armazenados na memória, e armazena em um terceiro vetor, também na memória.

2.2.3 prog3.asm

O terceiro programa é um dos solicitados para desenvolvimento no presente trabalho. Sua ideia é a de fazer acesso a um vetor/matriz de forma

[3]https://www.inf.pucrs.br/calazans/orgcomp_mat.html

a privilegiar a localidade temporal/espacial. Foi optado o uso de uma matriz de 10 linhas por 10 colunas, armazenada em memória. O programa se comporta de forma similar ao visto no seguinte código em C++:

```
int m[10][10];
int c = 0;

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        c = c + m[i][j];
        m[i][j] = c;
    }
}
```

2.2.4 prog4.asm

O quarto programa é, também, um dos solicitados para desenvolvimento no presente trabalho. Sua ideia é a de fazer acesso a um vetor/matriz de forma a prejudicar a localidade temporal/espacial. Foi optado o uso de uma matriz de 10 linhas por 10 colunas, armazenada em memória, assim como no prog3. O programa se comporta de forma similar ao visto no seguinte código em C++:

```
int m[10][10];
int c = 0;

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        c = c + m[j][i];
        m[j][i] = c;
    }
}
```

2.2.5 prog5.asm

O quinto programa, assim como o primeiro, foi desenvolvido pelos alunos que estão desenvolvendo o presente trabalho, quando cursaram Arquitetura de Computadores I, como um entregável para um trabalho da disciplina. Conforme descrição feita pelo professor César Marcon na especificação dos trabalhos a serem feitos na já citada disciplina, o programa faz o seguinte:

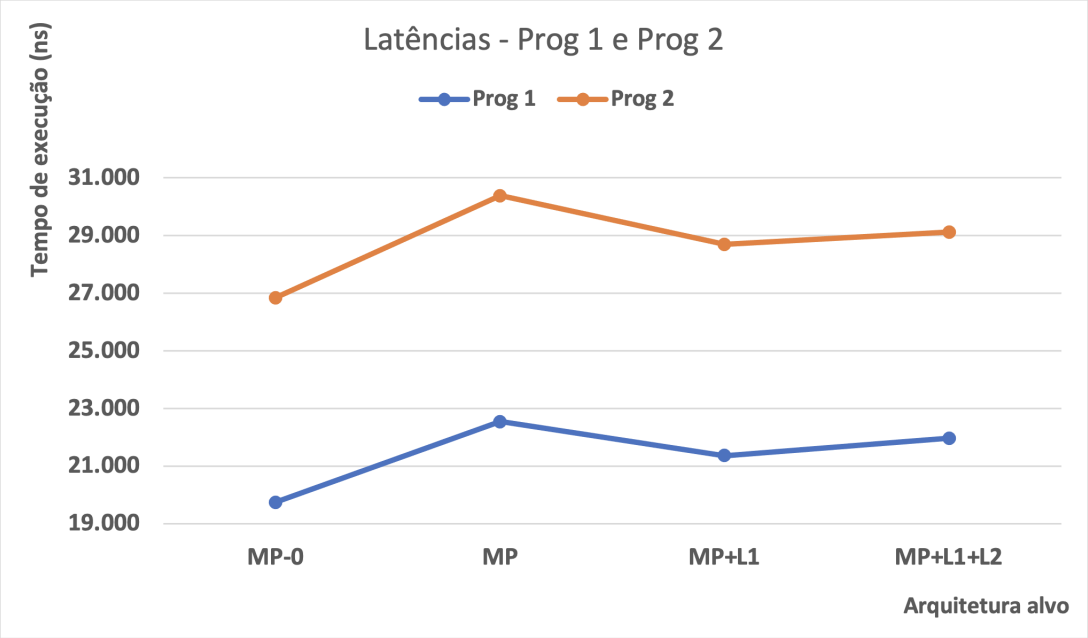
"Um algoritmo de ordenação de vetores que receba 20 inteiros quaisquer e, reordene os mesmos. O vetor original e o ordenado devem estar na memória".

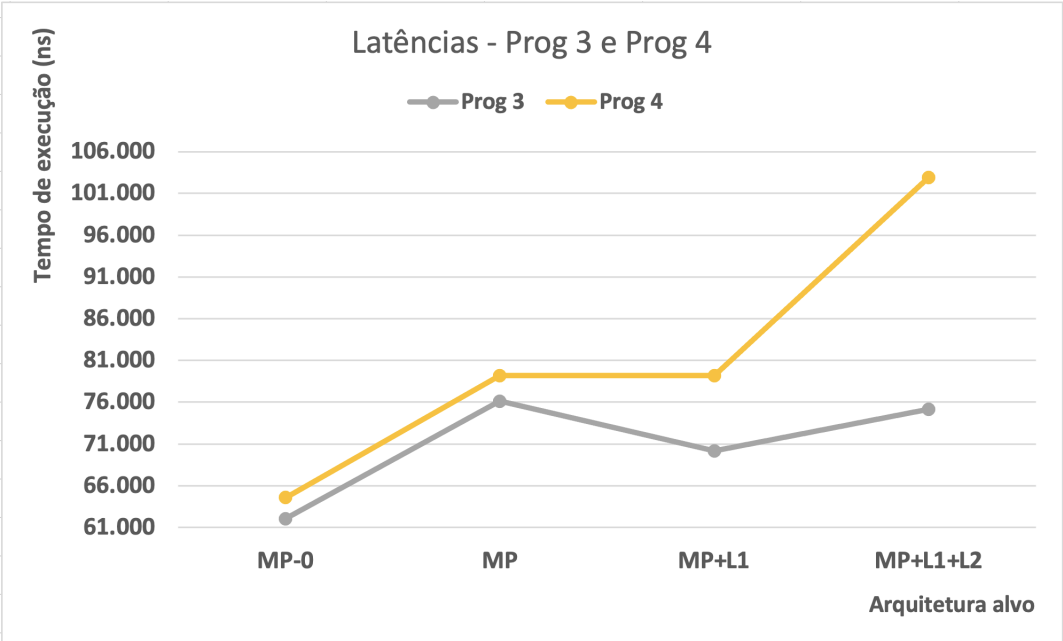
3 Resultados

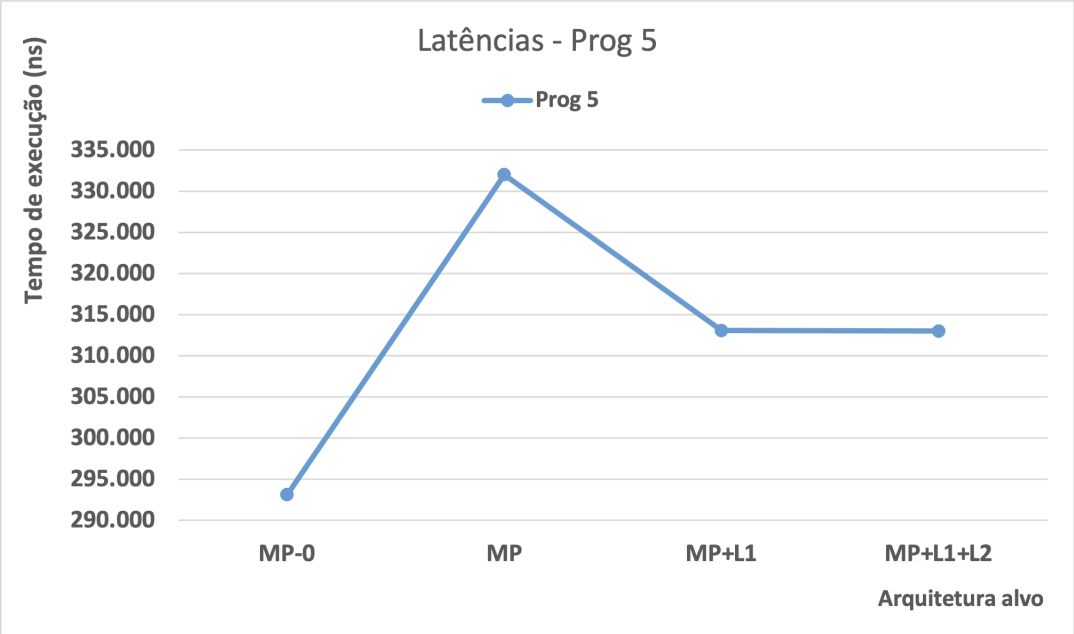
De forma a validar a implementação das arquiteturas e o funcionamento dos programas apresentados, foram medidas as latências dos cinco programas solicitados em cada uma das arquiteturas. O resultado pode ser visto a seguir:

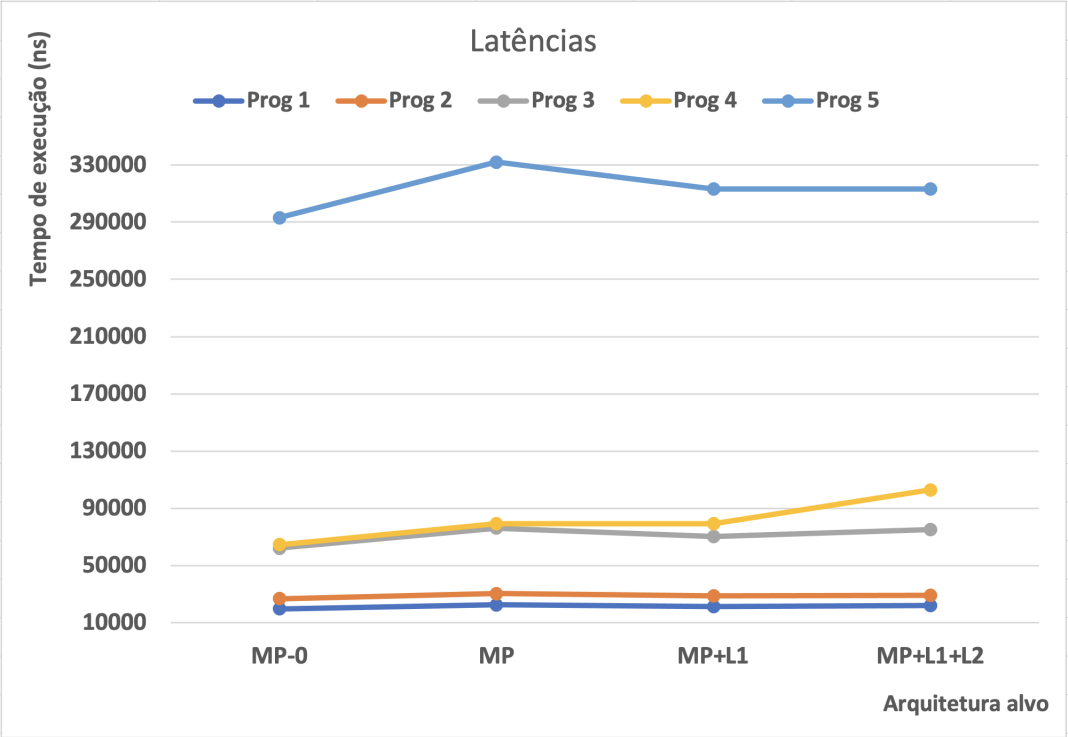
Programa	MP-0	MP	MP+L1	MP+L1+L2
Prog 1	19.740 ns	22.540 ns	21.360 ns	21.960 ns
Prog 2	26.840 ns	30.380 ns	28.680 ns	29.120 ns
Prog 3	62.080 ns	76.140 ns	70.160 ns	75.160 ns
Prog 4	64.580 ns	79.180 ns	79.200 ns	102.920 ns
Prog 5	293.080 ns	331.980 ns	313.040 ns	313.000 ns

Por conta das discrepâncias entre as latências dos programas, optou-se por plotar gráficos de programas cujas latências fossem próximas, podendo assim ter maior visibilidade dos pontos por conta do *range* dos eixos.




Latências - Prog 3 e Prog 4
Prog 3
Prog 4
Tempo de execução (ns)
106.000
101.000
96.000
91.000
86.000
81.000
76.000
71.000
66.000
61.000
MP-0
MP
MP+L1
MP+L1+L2
Arquitetura alvo


Latências - Prog 5
Prog 5
Tempo de execução (ns)
335.000
330.000
325.000
320.000
315.000
310.000
305.000
300.000
295.000
290.000
MP-0
MP
MP+L1
MP+L1+L2
Arquitetura alvo



Ao observar os gráficos e os resultados na tabela, é possível concluir que as arquiteturas que possuem cache(s) possibilitam com que os programas executem, geralmente, de forma consideravelmente mais rápida em comparação à arquitetura MP, que possui o atraso para acesso à MP. A exceção fica no Prog 3, que, como descrito anteriormente, é o programa que prejudica a localidade espacial/temporal, tendo por consequência uma sequência bem grande de cache *miss*. Por conta disso, a latência desse programa na arquitetura MP+L1+L2 é consideravelmente maior que a latência do mesmo programa na arquitetura MP. Entretanto, esse é o único programa no qual o uso das caches não pareceu favorável. Nas latências coletadas de todos os outros programas, é nítida a diminuição do tempo de execução deles nas arquiteturas com caches em comparação aos tempos de execução na arquitetura MP.

Cabe pontuar, também, que a arquitetura MP+L1 possui, em geral, menores latências se comparadas com as obtidas com a arquitetura MP+L1+L2. Isso se deve, principalmente, aos programas onde há muitos cache *miss* na cache L2, já que, além do seu tempo de acesso ser de 2 ciclos, a mesma deve fazer operações na memória, que leva 5 ciclos para ser acessada. Isso causa um atraso maior que o obtido em casos de cache *miss* na arquitetura

MP+L1. Todavia, cabe observar que as diferenças entre as latências dessas duas arquiteturas, em geral, não são expressivas.

Por fim, é visível que a arquitetura original MP-0 possui latências consideravelmente menores que às obtidas com as outras arquiteturas. Isso se deve unicamente ao fato de a arquitetura MP-0 não ter atraso algum durante o acesso à MP, sendo esse acesso feito de forma "ìnstantânea", o que não reflete a realidade do mundo computacional.

4 Conclusão

Através do extenso conteúdo abordado e dissertado neste trabalho, concluise que o conhecimento sobre memórias cache, suas técnicas de integridade de dados, modos de endereçamento, desenvolvimento em linguagem VHDL, interpretação de simulações e manutenção de códigos assembly foi grandiosamente ampliado durante seu desenvolvimento, sendo considerado pelos integrantes um conhecimento enriquecedor e essencial para as suas futuras vivências como engenheiros de computação.