RELATÓRIO LAOC2

Prática I (Parte III)

Dupla:

*Arthur Severo de Souza

*Victor Le Roy Matos

Na terceira parte da prática sobre hierarquia de memória fomos orientados à implementar um nível (L1) de memória cache totalmente associativa de quatro vias, este em vínculo com uma memória principal diretamente mapeada e atualizada por meio de *Write-Back*.

Os seguintes valores iniciais foram definidos:

Cache de dados (valores em decimal)

Válido?	Dirty?	LRU*	Tag	Valor
1	0	0	100	5
1	0	1	102	1
0	0	3	105	5
1	0	2	101	3

*LRU: 3 mais antigo, 0 mais recente

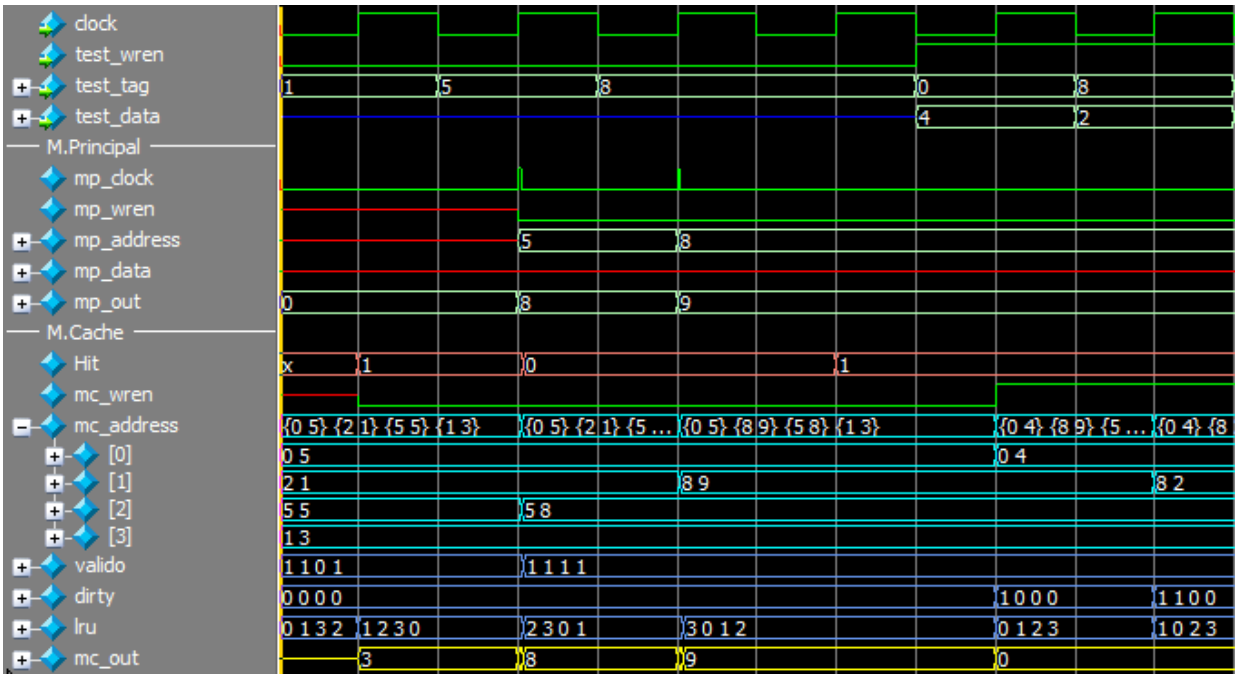
Memória (valores em decimal)

Endereço	Valor
100	5
101	3
102	1
103	0
104	1
105	8
106	3
107	4
108	9

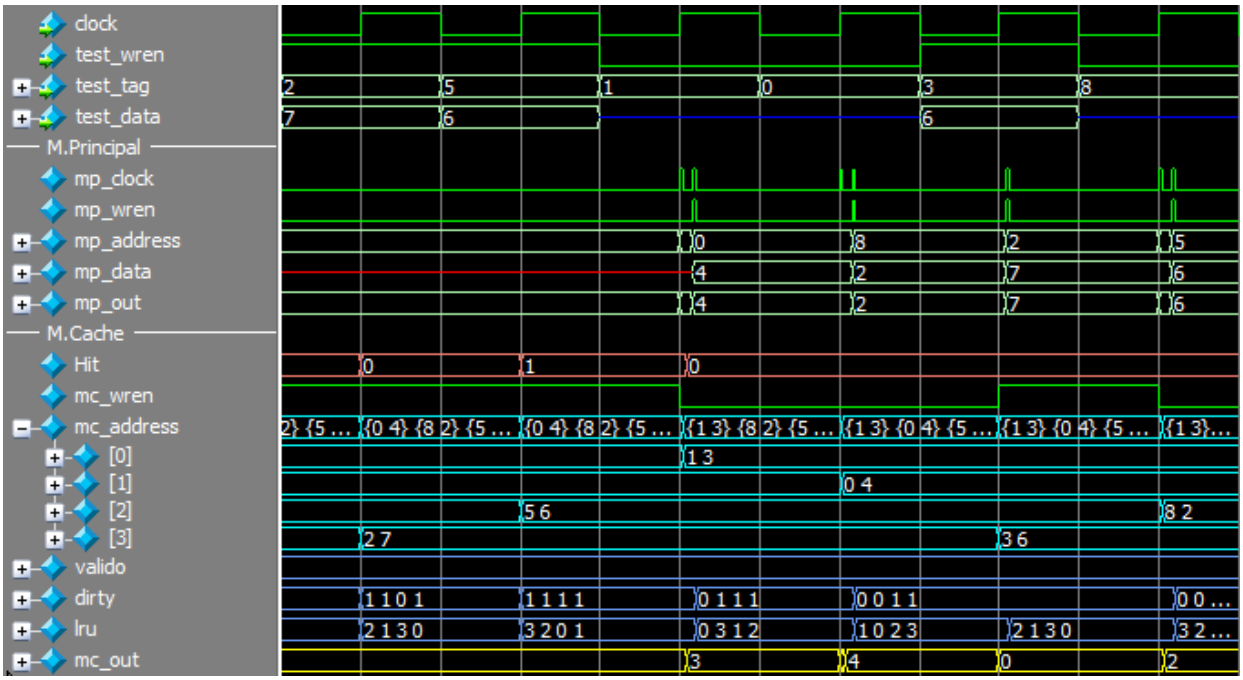
Apresentação das simulações

As operações requisitadas foram efetuadas em ordem de apresentação e acontecem em bordas de subida do clock.

Operações **1 -> 6**



Operações **7 -> 12**



Análise do código

No código, a tabela inicial da cache foi traduzida em uma matriz contendo as tags e os dados. É importante ressaltar que os códigos das tags foram traduzidos da tabela de valores iniciais, isto é, os códigos foram de (100-108) para (0-8). Os bits de validade, dirty e o LRU foram armazenados em um array de index equivalente às linhas da matriz.

```
mc_address[0][0] = 8'b00000000; mc_address[0][1] = 8'b00000101;
mc_address[1][0] = 8'b00000010; mc_address[1][1] = 8'b00000001;
mc_address[2][0] = 8'b00000101; mc_address[2][1] = 8'b00000101;
mc_address[3][0] = 8'b00000001; mc_address[3][1] = 8'b00000011;

valido[0] = 1'b1; valido[1] = 1'b1; valido[2] = 1'b0; valido[3] = 1'b1;
dirty[0] = 1'b0; dirty[1] = 1'b0; dirty[2] = 1'b0; dirty[3] = 1'b0;
lru[0] = 0; lru[1] = 1; lru[2] = 3; lru[3] = 2;
```

Então, a partir dos valores iniciais implementamos um bloco que identifique se houve hit ou miss ao acessar a memória cache. O loop percorre todos as quatro posições da cache e, caso encontre uma posição onde o bit de validade é de nível alto e a tag inserida seja equivalente à tag lida, identifica um hit no acesso, salva o index dos dados e abandona o laço.

```
// Verificar se houve hit
for (i=0; i<4; i=i+1) begin
   if(break == 0) begin
      if(test_tag == mc_address[i][0] && valido[i] == 1) begin // Hit
         hitIndex = i;
         hitTag = 1;
         break = 1;
      end
      else if(test_tag == mc_address[i][0] && valido[i] == 0) begin // Miss
         hitIndex = i;
         hitTag = 1;
         break = 1;
      end
      else begin // Miss
         hitTag = 0;
      end
   end
end
```

O laço seguinte é responsável por armazenar o index do acesso mais antigo para uso futuro.

```
// Procurar LRU mais antigo
for (i=0; i<4; i=i+1) begin
   if(lru[i] == 3) begin
      lastAccessedLRU = i;
   end
end
```

Implementamos as operações de read e write a depender do resultado do bloco de verificação de hit ou miss.

O bloco seguinte representa a operação de leitura. Primeiro, conferimos se o sinal wren está definido como leitura. Confirmamos também que as tags lida e recebida foram identificadas como equivalentes e se o bit de validade é de nível alto. Se ambos os sinais se mostrarem adequados, localizamos o index do dado procurado na matriz inicial e enviamos este para a saída da cache. Caso contrário, é preparado o ambiente para a possibilidade de um *Write-Back*.

```
// Operacao de read
if (mc_wren == 0) begin // Sinal setado para read
   if (hitTag == 1) begin // Tags equivalentes
      if (valido[hitIndex] == 1) begin // Bit de validade alto
         mc_out = mc_address[hitIndex][1]; // Hit, logo, dado encaminhado
                                           para a saída da cache
         Hit = 1;
      end
      else begin // Ocorreu a equivalencia entre as tags, porem o bit
         mc_address_aux[0] = mc_address[hitIndex][0]; de validade = 0,
         mc_address_aux[1] = mc_address[hitIndex][1];       logo, miss

         mp_address = test_tag;
         mp_clock = 1;
         mp_wren = 0;
         #2 mp_clock = 0; mp_wren = 0;

         mc_address[hitIndex][0] = mp_address;
         mc_address[hitIndex][1] = mp_out;
         #2 mc_out = mc_address[hitIndex][1];

         valido[hitIndex] = 1;
         Hit = 0;
      end
   end
   else begin  // Tags diferentes, logo, miss
      mc_address_aux[0] = mc_address[lastAccessedLRU][0];
      mc_address_aux[1] = mc_address[lastAccessedLRU][1];

      mp_address = test_tag;
      mp_clock = 1;
      mp_wren = 0;
      #2 mp_clock = 0; mp_wren = 0;

      mc_address[lastAccessedLRU][0] = mp_address;
      mc_address[lastAccessedLRU][1] = mp_out;
      #2 mc_out = mc_address[lastAccessedLRU][1];

      Hit = 0;
   end
end
```

Já o bloco de escrita depende do sinal wren em nível alto e basta que as tags inseridas e lida sejam equivalentes. Se for o caso, o dado inserido é escrito no index localizado durante a conferência do hit e o bit dirty tem seu valor definido como alto. Se ocorreu miss, o ambiente é preparado para a ocorrência de um *Write-Back*.

```
// Operacao de write
else begin // Sinal setado para write
   if (hitTag == 1) begin // Tags equivalente, logo hit
      mc_address[hitIndex][1] = test_data; // Escrita do dado de entrada
      dirty[hitIndex] = 1; // Bit dirty setado para nivel alto

      //#2 mc_out = mc_address[hitIndex][1];
      Hit = 1;
   end
   else begin // Tag nao encontrada, logo miss
      mc_address_aux[0] = mc_address[lastAccessedLRU][0];
      mc_address_aux[1] = mc_address[lastAccessedLRU][1];

      mc_address[lastAccessedLRU][0] = test_tag;
      mc_address[lastAccessedLRU][1] = test_data;

      if (dirty[lastAccessedLRU] == 0) begin
         dirty[lastAccessedLRU] = 1; skipWB = 1;
      end

      //#2 mc_out = mc_address[lastAccessedLRU][1];
      Hit = 0;
   end
end
```

O *Write-Back* então é efetuado pelo bloco a seguir, se foi identificado um miss no acesso. São tratados todos os possíveis casos responsáveis pela ocorrência do *Write-Back*. Os sinais auxiliares têm seus valores armazenados na posição mapeada da memória principal e, no caso de uma operação read, o bit dirty do index de acesso mais antigo é definido como baixo.

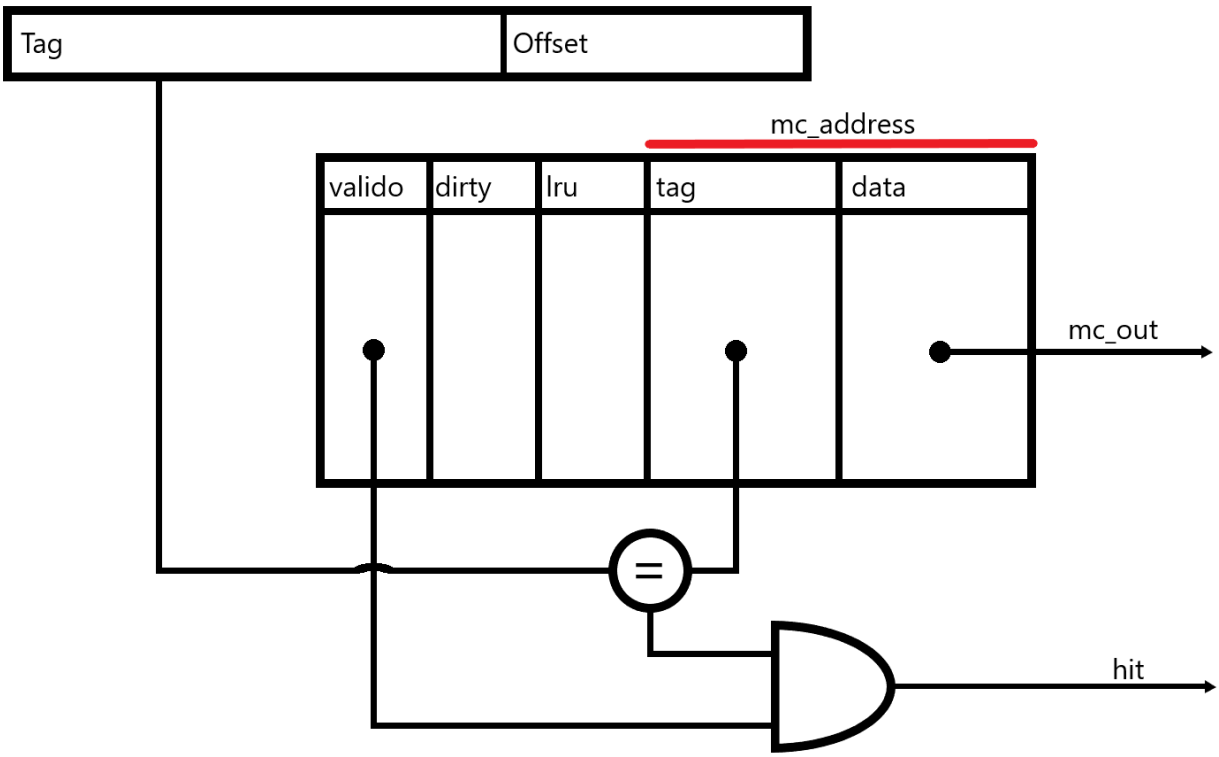
```
// Write back -> Ocorre apenas quando acontece miss
if (Hit == 0) begin // Se a cache gerou um miss
   if (dirty[lastAccessedLRU] == 1 && skipWB == 0) begin // Caso bloco esteja
      #4 mp_clock = 0;                                   sujo gera write back
      mp_address = mc_address_aux[0];
      mp_data = mc_address_aux[1];
      mp_wren = 1;
      mp_clock = 1;
      #2 mp_clock = 0; mp_wren = 0;

      if (mc_wren == 0) dirty[lastAccessedLRU] = 0; // Caso read, setamos o dirty para 0
   end
end
```

Por fim, condizente com o funcionamento em uma hierarquia implementada fisicamente, efetuamos a atualização do LRU. Os laços percorrem cada index da memória cache, incrementando os números menores que o LRU da posição acessada e zerando o valor na posição em questão. No caso de acesso à posição de LRU mais alto, esta é zerada.

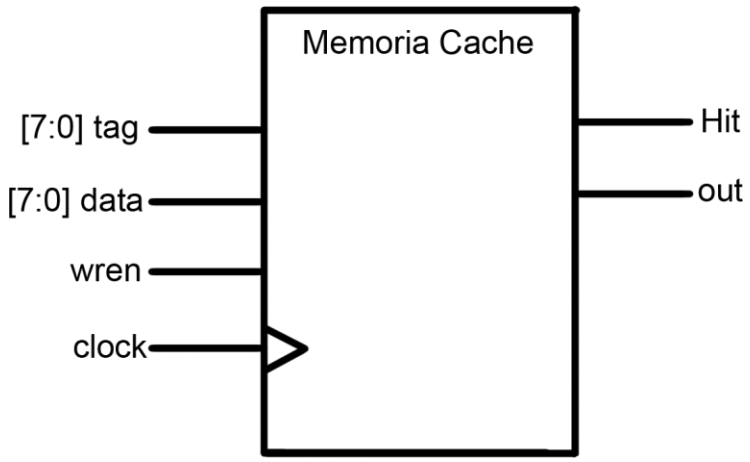
```
// Alteracoes do LRU
if (hitTag == 1) begin
   lruValue = lru[hitIndex];
   for (i=0; i<4; i=i+1) begin // Retirado valores de 0 a 3
      if (i != hitIndex && lru[i] < lruValue) lru[i] = lru[i] + 1;
      else lru[hitIndex] = 0;
   end
end
else // Retirado mais velho = 3
   for (i=0; i<4; i=i+1) begin
      if (i != lastAccessedLRU) lru[i] = lru[i] + 1;
      else lru[lastAccessedLRU] = 0;
   end
```

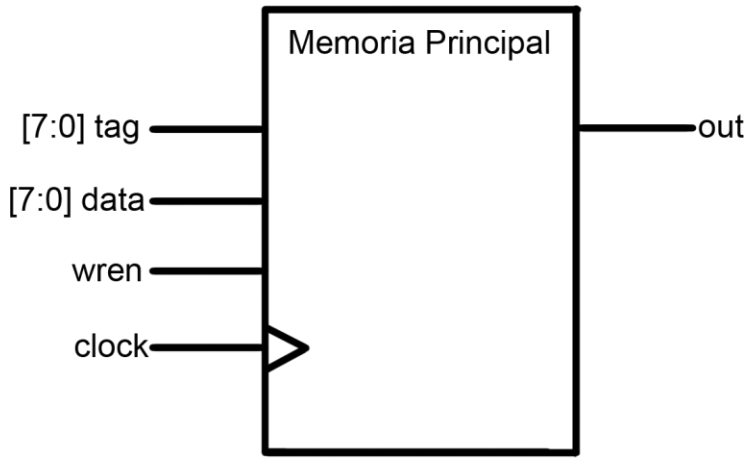
Projeto

Simplificação das conexões da cache (input e output)



Blocos presentes no projeto





Código completo da simulação

```
module m_cache (clock, test_tag, test_data, test_wren);

// Declarando inputs
input clock, test_wren;
input [7:0] test_tag, test_data;

// Memoria principal
reg mp_clock, mp_wren;           // Clock controlado para tornar mais facil a leitura da memoria principal
reg [7:0]  mp_address, mp_data;
wire [7:0] mp_out;

// Memoria cache
reg mc_wren;
reg [7:0] mc_address [0:3][0:1];
reg valido [0:3], dirty [0:3];
reg [1:0] lru [0:3];
reg [7:0] mc_out;
reg Hit;

// Intermediarias
integer i, lastAccessedLRU, hitIndex, hitTag, break, lruValue, skipWB;
reg [7:0] mc_address_aux [0:1];

// Valores iniciais da memoria cache definidos a partir do codigo teste
initial begin
   mc_address[0][0] = 8'b00000000; mc_address[0][1] = 8'b00000101;
   mc_address[1][0] = 8'b00000010; mc_address[1][1] = 8'b00000001;
   mc_address[2][0] = 8'b00000101; mc_address[2][1] = 8'b00000101;
   mc_address[3][0] = 8'b00000001; mc_address[3][1] = 8'b00000011;

   valido[0] = 1'b1; valido[1] = 1'b1; valido[2] = 1'b0; valido[3] = 1'b1;
   dirty[0] = 1'b0; dirty[1] = 1'b0; dirty[2] = 1'b0; dirty[3] = 1'b0;
   lru[0] = 0; lru[1] = 1; lru[2] = 3; lru[3] = 2;

   mp_clock = 0;
end

always@(posedge clock) begin
   i=0; lastAccessedLRU=0; hitIndex=0; hitTag=0; break=0; lruValue=0; skipWB=0; // Variaveis intermediarias
   mc_wren = test_wren; mc_out = 0; // Controle da cache

   // Verificar se houve hit
   for (i=0; i<4; i=i+1) begin
      if(break == 0) begin
         if(test_tag == mc_address[i][0] && valido[i] == 1) begin // Hit
            hitIndex = i;
            hitTag = 1;
            break = 1;
         end
         else if(test_tag == mc_address[i][0] && valido[i] == 0) begin // Miss
            hitIndex = i;
            hitTag = 1;
            break = 1;
         end
         else begin // Miss
            hitTag = 0;
         end
      end
   end


   // Procurar LRU mais antigo
   for (i=0; i<4; i=i+1) begin
      if(lru[i] == 3) begin
         lastAccessedLRU = i;
      end
   end
```

```
// Operacao de read
if (mc_wren == 0) begin // Sinal setado para read
   if (hitTag == 1) begin // Tags equivalentes
      if (valido[hitIndex] == 1) begin // Bit de validade alto
         mc_out = mc_address[hitIndex][1]; // Hit, logo, dado encaminhado para a saída da cache

         Hit = 1;
      end
      else begin // Ocorreu a equivalencia entre as tags, porem o bit de validade = 0, logo, miss
         mc_address_aux[0] = mc_address[hitIndex][0];
         mc_address_aux[1] = mc_address[hitIndex][1];

         mp_address = test_tag;
         mp_clock = 1;
         mp_wren = 0;
         #2 mp_clock = 0; mp_wren = 0;

         mc_address[hitIndex][0] = mp_address;
         mc_address[hitIndex][1] = mp_out;
         #2 mc_out = mc_address[hitIndex][1];

         valido[hitIndex] = 1;
         Hit = 0;
      end
   end
   else begin  // Tags diferentes, logo, miss
      mc_address_aux[0] = mc_address[lastAccessedLRU][0];
      mc_address_aux[1] = mc_address[lastAccessedLRU][1];

      mp_address = test_tag;
      mp_clock = 1;
      mp_wren = 0;
      #2 mp_clock = 0; mp_wren = 0;

      mc_address[lastAccessedLRU][0] = mp_address;
      mc_address[lastAccessedLRU][1] = mp_out;
      #2 mc_out = mc_address[lastAccessedLRU][1];

      Hit = 0;
   end
end
// Operacao de write
else begin // Sinal setado para write
   if (hitTag == 1) begin // Tags equivalente, logo hit
      mc_address[hitIndex][1] = test_data; // Escrita do dado de entrada
      dirty[hitIndex] = 1; // Bit dirty setado para nivel alto

      //#2 mc_out = mc_address[hitIndex][1];
      Hit = 1;
   end
   else begin // Tag nao encontrada, logo miss
      mc_address_aux[0] = mc_address[lastAccessedLRU][0];
      mc_address_aux[1] = mc_address[lastAccessedLRU][1];

      mc_address[lastAccessedLRU][0] = test_tag;
      mc_address[lastAccessedLRU][1] = test_data;

      if (dirty[lastAccessedLRU] == 0) begin
         dirty[lastAccessedLRU] = 1; skipWB = 1;
      end

      //#2 mc_out = mc_address[lastAccessedLRU][1];
      Hit = 0;
   end
end


// Write back -> Ocorre apenas quando acontece miss
if (Hit == 0) begin // Se a cache gerou um miss
   if (dirty[lastAccessedLRU] == 1 && skipWB == 0) begin // Caso bloco esteja sujo gera write back
      #4 mp_clock = 0;
      mp_address = mc_address_aux[0];
      mp_data = mc_address_aux[1];
      mp_wren = 1;
      mp_clock = 1;
      #2 mp_clock = 0; mp_wren = 0;

      if (mc_wren == 0) dirty[lastAccessedLRU] = 0; // Caso read, setamos o dirty para 0
   end
end
```

```
    // Alteracoes do LRU
    if (hitTag == 1) begin // Caso a tag seja encontrada
        lruValue = lru[hitIndex];
        for (i=0; i<4; i=i+1) begin // Retirado valores de 0 a 3
            if (i != hitIndex && lru[i] < lruValue) lru[i] = lru[i] + 1;
            else lru[hitIndex] = 0;
        end
    end
    else // Caso a tag nao seja encontrada
        for (i=0; i<4; i=i+1) begin // Retirado mais velho = 3
            if (i != lastAccessedLRU) lru[i] = lru[i] + 1;
            else lru[lastAccessedLRU] = 0;
        end
end

m_principal m_principal(mp_address, mp_clock, mp_data, mp_wren, mp_out);

endmodule
```

Configuração modelsim

Para realizar a simulação no modelsim, foi necessário configurar o clock:

Offset: 0

Duty: 50

Period: 100

First Edge: Falling

Além disso, para melhor visualização, adicionamos todas as variáveis de input/output, além das principais da memória cache e memória principal.