**《计算机体系结构》实验报告**

|  |
| --- |
| 二、实验项目内容  1. 最低要求：参考指导书中直接映射写直达 Cache 的实现，实现写回策略的 Cache 2. 替换实验环境中的 Cache 模块，并通过仿真测试 3. [选做][较高要求] 性能优化，实现 2 路组相联的 Cache 4. [选做][更高要求] 性能优化，使用伪 LRU 等替换策略实现 4 路以上组相联的 Cache 5. [选做][最高要求] 实现其它 Cache 性能优化方法，如 axi burst 传输 |
| 三，实验步骤  0，D\_cache.v , i\_cache.v文件和tb\_top\_behav.wcfg波形图以及完整工程见文件附件。  1，Cache  Cache存储器，高速缓冲存储器，是位于CPU和主存储器DRAM（Dynamic Random Access Memory）之间，规模较小，但速度很高的存储器，通常由SRAM（Static Random Access Memory 静态存储器）组成。它是位于CPU与内存间的一种容量较小但速度很高的存储器。CPU的速度远高于内存，当CPU直接从内存中存取数据时要等待一定时间周期，而Cache则可以保存CPU刚用过或循环使用的一部分数据，如果CPU需要再次使用该部分数据时可从Cache中直接调用，这样就避免了重复存取数据，减少了CPU的等待时间，因而提高了系统的效率。  如下为cache和cpu和主存之间的关系    2，本次实验需要重写d\_cache和i\_cache，d\_cache用于数据读写，涉及到对于数据的读和写操作。I\_cache用于读取指令,CPU不会写入指令，只会读取指令。  D\_cache接口如下   1. module d\_cache ( 2. input wire clk, rst, 3. //mips core 4. input         cpu\_data\_req     ,//CPU访问请求 5. input         cpu\_data\_wr      ,//写能端，为1表示写入数据 6. input  [1 :0] cpu\_data\_size    ,//数据宽度 7. input  [31:0] cpu\_data\_addr    ,//读写数据的地址 8. input  [31:0] cpu\_data\_wdata   ,//要写的数据 9. output [31:0] cpu\_data\_rdata   ,//读出的数据 10. output        cpu\_data\_addr\_ok ,//地址接收完成 11. output        cpu\_data\_data\_ok ,//数据操作完成 13. //axi interface 14. output         cache\_data\_req     ,//cache访问请求 15. output         cache\_data\_wr      ,//写能端，为1表示对主存写数据 16. output  [1 :0] cache\_data\_size    ,//数据宽度 17. output  [31:0] cache\_data\_addr    ,//数据操作地址 18. output  [31:0] cache\_data\_wdata   ,//写数据 19. input   [31:0] cache\_data\_rdata   ,//读出数据 20. input          cache\_data\_addr\_ok ,//主存地址接受完成 21. input          cache\_data\_data\_ok //主存数据处理完成 22. );   I\_cache接口如下   1. module i\_cache ( 2. input wire clk, rst, 3. //mips core 4. input         cpu\_inst\_req     ,//CPU访问请求 5. input         cpu\_inst\_wr      ,//写能端，为1表示写入数据 6. input  [1 :0] cpu\_inst\_size    ,//数据宽度 7. input  [31:0] cpu\_inst\_addr    ,//读写数据的地址 8. input  [31:0] cpu\_inst\_wdata   ,//要写的数据 9. output [31:0] cpu\_inst\_rdata   ,//读出的数据 10. output        cpu\_inst\_addr\_ok ,//地址接收完成 11. output        cpu\_inst\_data\_ok ,//数据操作完成 13. //axi interface 14. output         cache\_inst\_req     ,//cache访问请求 15. output         cache\_inst\_wr      ,//写能端，为1表示对主存写数据 16. output  [1 :0] cache\_inst\_size    ,//数据宽度 17. output  [31:0] cache\_inst\_addr    ,//数据操作地址 18. output  [31:0] cache\_inst\_wdata   ,//写数据 19. input   [31:0] cache\_inst\_rdata   ,//读出数据 20. input          cache\_inst\_addr\_ok ,//主存地址接受完成 21. input          cache\_inst\_data\_ok //主存数据处理完成 22. );   3,d\_cache的设计，对于数据缓存，本实验中实现了应用写回写分配策略和LRU替换算法的四路组相连的Cache。每路大小为4KiB，共16KiB。  D\_cache说明  大小：4 \* 4KiB，每路4KiB  相连度：4路组相连  替换算法：LRU算法  访存策略：写回写分配  D\_cache的地址说明  32到12位为tag标记（20位），11到2位为index索引（10位），1到0位为offset字内偏移位（2位），offset用于字节寻址  D\_cache的流程图如下：    D\_cache的状态转换图如下：    4，D\_Cache的实现  ①,数据存储，存储部分采用reg实现，LRU部分为LRU算法的饱和计数器。   1. //cache参数 2. parameter  INDEX\_WIDTH  = 10, OFFSET\_WIDTH = 2; 3. localparam TAG\_WIDTH    = 32 - INDEX\_WIDTH - OFFSET\_WIDTH; 4. localparam CACHE\_DEEPTH = 1 << INDEX\_WIDTH; 6. //Cach储存参数 7. reg                 cache\_valid\_1 [CACHE\_DEEPTH - 1 : 0]; 8. reg [TAG\_WIDTH-1:0] cache\_tag\_1   [CACHE\_DEEPTH - 1 : 0]; 9. reg [31:0]          cache\_block\_1 [CACHE\_DEEPTH - 1 : 0]; 10. reg           cache\_dirty\_1 [CACHE\_DEEPTH - 1 : 0]; 12. reg                 cache\_valid\_2 [CACHE\_DEEPTH - 1 : 0]; 13. reg [TAG\_WIDTH-1:0] cache\_tag\_2   [CACHE\_DEEPTH - 1 : 0]; 14. reg [31:0]          cache\_block\_2 [CACHE\_DEEPTH - 1 : 0]; 15. reg           cache\_dirty\_2 [CACHE\_DEEPTH - 1 : 0]; 17. reg                 cache\_valid\_3 [CACHE\_DEEPTH - 1 : 0]; 18. reg [TAG\_WIDTH-1:0] cache\_tag\_3   [CACHE\_DEEPTH - 1 : 0]; 19. reg [31:0]          cache\_block\_3 [CACHE\_DEEPTH - 1 : 0]; 20. reg           cache\_dirty\_3 [CACHE\_DEEPTH - 1 : 0]; 22. reg                 cache\_valid\_4 [CACHE\_DEEPTH - 1 : 0]; 23. reg [TAG\_WIDTH-1:0] cache\_tag\_4   [CACHE\_DEEPTH - 1 : 0]; 24. reg [31:0]          cache\_block\_4 [CACHE\_DEEPTH - 1 : 0]; 25. reg           cache\_dirty\_4 [CACHE\_DEEPTH - 1 : 0]; 27. //LRU 28. reg [1:0]          LRU\_1 [CACHE\_DEEPTH - 1 : 0]; 29. reg [1:0]          LRU\_2 [CACHE\_DEEPTH - 1 : 0]; 30. reg [1:0]          LRU\_3 [CACHE\_DEEPTH - 1 : 0]; 31. reg [1:0]          LRU\_4 [CACHE\_DEEPTH - 1 : 0];   ②，地址分割   1. //地址分割 2. wire [OFFSET\_WIDTH-1:0] offset; 3. wire [INDEX\_WIDTH-1:0] index; 4. wire [TAG\_WIDTH-1:0] tag; 6. assign offset = cpu\_data\_addr[OFFSET\_WIDTH - 1 : 0]; 7. assign index = cpu\_data\_addr[INDEX\_WIDTH + OFFSET\_WIDTH - 1 : OFFSET\_WIDTH]; 8. assign tag = cpu\_data\_addr[31 : INDEX\_WIDTH + OFFSET\_WIDTH];   ③分别提取四路的数据并判断命中和缺失   1. //分别提取4路cache 2. wire c\_valid\_1,c\_valid\_2,c\_valid\_3,c\_valid\_4; 3. wire [TAG\_WIDTH-1:0] c\_tag\_1,c\_tag\_2,c\_tag\_3,c\_tag\_4; 4. wire [31:0] c\_block\_1,c\_block\_2,c\_block\_3,c\_block\_4; 5. wire dirty\_1,dirty\_2,dirty\_3,dirty\_4;  8. assign c\_valid\_1 = cache\_valid\_1[index]; 9. assign c\_tag\_1   = cache\_tag\_1  [index]; 10. assign c\_block\_1 = cache\_block\_1[index]; 11. assign dirty\_1 =   cache\_dirty\_1[index]; 13. assign c\_valid\_2 = cache\_valid\_2[index]; 14. assign c\_tag\_2   = cache\_tag\_2  [index]; 15. assign c\_block\_2 = cache\_block\_2[index]; 16. assign dirty\_2 =   cache\_dirty\_2[index]; 18. assign c\_valid\_3 = cache\_valid\_3[index]; 19. assign c\_tag\_3   = cache\_tag\_3  [index]; 20. assign c\_block\_3 = cache\_block\_3[index]; 21. assign dirty\_3 =   cache\_dirty\_3[index]; 23. assign c\_valid\_4 = cache\_valid\_4[index]; 24. assign c\_tag\_4   = cache\_tag\_4  [index]; 25. assign c\_block\_4 = cache\_block\_4[index]; 26. assign dirty\_4 =   cache\_dirty\_4[index];   30. wire hit\_1, miss\_1,hit\_2, miss\_2,hit\_3, miss\_3,hit\_4, miss\_4; 31. assign hit\_1 = c\_valid\_1 & (c\_tag\_1 == tag); 32. assign hit\_2 = c\_valid\_2 & (c\_tag\_2 == tag); 33. assign hit\_3 = c\_valid\_3 & (c\_tag\_3 == tag); 34. assign hit\_4 = c\_valid\_4 & (c\_tag\_4 == tag); 36. //判断缺失和命中 37. wire hit, miss; 38. assign hit = hit\_1 | hit\_2 | hit\_3 | hit\_4; 39. assign miss = ~hit;   ④当缺失的时候找到需要替换的块，即在四路中此index位置的块中选择LRU计数器为0的块作为需要替换的块。并链接这个要替换的块的属性，供以后使用。   1. //选出替换块的地址和数据块(LRU) 2. wire [31:0] change\_addr,change\_data; 3. wire [1:0] change\_road; 4. wire dirty; 6. //LRU 7. assign change\_addr = LRU\_1[index]==0 ? {c\_tag\_1,index,offset} : 8. LRU\_2[index]==0 ? {c\_tag\_2,index,offset} : 9. LRU\_3[index]==0 ? {c\_tag\_3,index,offset} : 10. {c\_tag\_4,index,offset}; 11. assign change\_data = LRU\_1[index]==0 ? c\_block\_1 : 12. LRU\_2[index]==0 ? c\_block\_2 : 13. LRU\_3[index]==0 ? c\_block\_3 : 14. c\_block\_4; 15. assign change\_road = LRU\_1[index]==0 ? 2'b00 : 16. LRU\_2[index]==0 ? 2'b01 : 17. LRU\_3[index]==0 ? 2'b10 : 18. 2'b11; 19. assign dirty = LRU\_1[index]==0 ? dirty\_1 : 20. LRU\_2[index]==0 ? dirty\_2 : 21. LRU\_3[index]==0 ? dirty\_3 : 22. dirty\_4;   ⑤根据上述状态转移图设计d\_cache的状态机。   1. //FSM 2. parameter H = 2'b00, NHD = 2'b01, NHND = 2'b10; //H不需要读写存储器，NHD需要写和读，NHND只需要读 3. reg [1:0] state; 4. reg [31:0] mem\_addr; 5. wire write\_en; 6. reg mem\_write,mem\_read; 7. assign write\_en = mem\_write; 8. always @(posedge clk) begin 9. **if**(rst) begin 10. state <= H; 11. mem\_addr <= cpu\_data\_addr; 12. end 13. **else** begin 14. **case**(state) 15. H: begin 16. **if**((cpu\_data\_req & miss))begin 17. **if**(dirty)begin 18. state <= NHD; 19. mem\_write <= 1'b1; 20. mem\_read <= 1'b0; 21. mem\_addr <= change\_addr; 22. end **else** begin 23. state <= NHND; 24. mem\_write <= 1'b0; 25. mem\_read <= 1'b1; 26. mem\_addr <= cpu\_data\_addr; 27. end 29. end **else** begin 30. state <= H; 31. mem\_write <= 1'b0; 32. mem\_read <= 1'b1; 33. mem\_addr <= cpu\_data\_addr; 34. end 35. end 36. NHD:begin 37. state <= cache\_data\_data\_ok ? NHND : NHD; 38. mem\_write <= cache\_data\_data\_ok ? 1'b0: 1'b1; 39. mem\_read <= cache\_data\_data\_ok ? 1'b1 : 1'b0; 40. mem\_addr <= cache\_data\_data\_ok ? cpu\_data\_addr : change\_addr; 41. end 42. NHND:begin 43. state <= cache\_data\_data\_ok ? H : NHND; 44. mem\_write <= 1'b0; 45. mem\_read <= 1'b1; 46. mem\_addr <= cpu\_data\_addr; 47. end 49. endcase 50. end 51. end   H状态为命中态，当请求的地址未命中的时候，根据替换的块是否为脏块判断转移后的状态，若要替换的块为脏块，则转移为NHD状态，否则跳转到NHND状态。  NHND状态（读内存态）为缺失但不需要写入内存的状态，由于替换的块不为脏块，不需要写回内存，只需要将对应地址的块从内存读入cache并替换掉替换块，然后再根据是CPU读写请求读写此块。  NHND状态的读内存mem\_read=1，写内存mem\_write=0  NHD状态（写内存态）为缺失且需要写入内存的状态，由于替换的块为脏块，需要写回内存，需要将对应地址的块写入内存中，当写内存结束后，即脏块处理完毕，会将状态跳转为NHND状态处理读内存。  NHD状态的读内存mem\_read=0，写内存mem\_write=1  ⑥定义内存读写操作   1. //一次读内存操作 2. wire read\_req;      //读请求 3. reg addr\_rcv;       //内存地址接受成功到结束 4. wire read\_finish;  //读操作完毕 6. always @(posedge clk) begin 7. addr\_rcv <= rst ? 1'b0 : 8. mem\_read & cache\_data\_req & cache\_data\_addr\_ok ? 1'b1 : 9. read\_finish ? 1'b0 : addr\_rcv; 10. end 11. assign read\_req = state==NHND; 12. assign read\_finish = mem\_read & cache\_data\_data\_ok; 14. //一次写内存操作 15. wire write\_req; 16. reg waddr\_rcv; 17. wire write\_finish; 18. always @(posedge clk) begin 19. waddr\_rcv <= rst ? 1'b0 : 20. mem\_write & cache\_data\_req & cache\_data\_addr\_ok ? 1'b1 : 21. write\_finish ? 1'b0 : waddr\_rcv; 22. end 23. assign write\_req = state==NHD; 24. assign write\_finish = mem\_write & cache\_data\_data\_ok;   ⑦连接mips和axi   1. //output to mips core 2. assign cpu\_data\_rdata   = hit\_1 ? c\_block\_1 : 3. hit\_2 ? c\_block\_2 : 4. hit\_3 ? c\_block\_3 : 5. hit\_4 ? c\_block\_4 : cache\_data\_rdata; 6. assign cpu\_data\_addr\_ok = cpu\_data\_req & hit | cache\_data\_req & cache\_data\_addr\_ok & mem\_read; 7. assign cpu\_data\_data\_ok = cpu\_data\_req & hit | read\_finish ;   11. //output to axi interface 12. assign cache\_data\_req   = read\_req & ~addr\_rcv | write\_req & ~waddr\_rcv; 13. assign cache\_data\_wr    = write\_en; 14. assign cache\_data\_size  = cpu\_data\_size; 15. assign cache\_data\_addr  = mem\_addr; 16. assign cache\_data\_wdata = change\_data;   ⑧保存index的tag，防止以后地址变化导致的异常   1. //保存index和tag，防止之后发生改变 2. reg [TAG\_WIDTH-1:0] tag\_save; 3. reg [INDEX\_WIDTH-1:0] index\_save; 4. always @(posedge clk) begin 5. tag\_save   <= rst ? 0 : 6. cpu\_data\_req ? tag : tag\_save; 7. index\_save <= rst ? 0 : 8. cpu\_data\_req ? index : index\_save; 9. end 11. wire [31:0] write\_cache\_data\_1,write\_cache\_data\_2,write\_cache\_data\_3,write\_cache\_data\_4; 12. wire [3:0] write\_mask;   ⑨掩码操作，处理CPU的写数据   1. //掩码操作 2. assign write\_mask = cpu\_data\_size==2'b00 ? 3. (cpu\_data\_addr[1] ? (cpu\_data\_addr[0] ? 4'b1000 : 4'b0100): 4. (cpu\_data\_addr[0] ? 4'b0010 : 4'b0001)) : 5. (cpu\_data\_size==2'b01 ? (cpu\_data\_addr[1] ? 4'b1100 : 4'b0011) : 4'b1111);  8. assign write\_cache\_data\_1 = cache\_block\_1[index] & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} | 9. cpu\_data\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}}; 10. assign write\_cache\_data\_2 = cache\_block\_2[index] & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} | 11. cpu\_data\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}}; 12. assign write\_cache\_data\_3 = cache\_block\_3[index] & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} | 13. cpu\_data\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}}; 14. assign write\_cache\_data\_4 = cache\_block\_4[index] & ~{{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}} | 15. cpu\_data\_wdata & {{8{write\_mask[3]}}, {8{write\_mask[2]}}, {8{write\_mask[1]}}, {8{write\_mask[0]}}};   ⑩时序操作，执行命中和缺失处理完毕后的cache的更新和LRU饱和计数器的更新   1. integer t; 2. always @(posedge clk) begin 3. //初始化 4. **if**(rst) begin 5. **for**(t=0; t<CACHE\_DEEPTH; t=t+1) begin 6. cache\_valid\_1[t] <= 0; 7. cache\_dirty\_1[t] <= 0; 8. cache\_valid\_2[t] <= 0; 9. cache\_dirty\_2[t] <= 0; 10. cache\_valid\_3[t] <= 0; 11. cache\_dirty\_3[t] <= 0; 12. cache\_valid\_4[t] <= 0; 13. cache\_dirty\_4[t] <= 0; 14. LRU\_1[t] <= 0; 15. LRU\_2[t] <= 0; 16. LRU\_3[t] <= 0; 17. LRU\_4[t] <= 0; 18. end 19. end 20. **else** begin 21. //块替换后的处理（CPU发出读请求） 22. **if**(read\_finish & read) begin 23. **case**(change\_road) 24. 2'b00:begin 25. cache\_valid\_1[index\_save] <= 1'b1; 26. cache\_tag\_1  [index\_save] <= tag\_save; 27. cache\_block\_1[index\_save] <= cache\_data\_rdata; 28. cache\_dirty\_1[index\_save] <= 1'b0; 29. LRU\_1[index\_save] <= 3; 30. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 31. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 32. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 33. end 34. 2'b01:begin 35. cache\_valid\_2[index\_save] <= 1'b1; 36. cache\_tag\_2  [index\_save] <= tag\_save; 37. cache\_block\_2[index\_save] <= cache\_data\_rdata; 38. cache\_dirty\_2[index\_save] <= 1'b0; 39. LRU\_2[index\_save] <= 3; 40. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 41. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 42. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 43. end 44. 2'b10:begin 45. cache\_valid\_3[index\_save] <= 1'b1; 46. cache\_tag\_3  [index\_save] <= tag\_save; 47. cache\_block\_3[index\_save] <= cache\_data\_rdata; 48. cache\_dirty\_3[index\_save] <= 1'b0; 49. LRU\_3[index\_save] <= 3; 50. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 51. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 52. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 53. end 54. 2'b11:begin 55. cache\_valid\_4[index\_save] <= 1'b1; 56. cache\_tag\_4  [index\_save] <= tag\_save; 57. cache\_block\_4[index\_save] <= cache\_data\_rdata; 58. cache\_dirty\_4[index\_save] <= 1'b0; 59. LRU\_4[index\_save] <= 3; 60. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 61. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 62. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 63. end 64. endcase 65. end **else** **if**(read\_finish & write) begin 66. //块替换后的处理（CPU发出写请求） 67. **case**(change\_road) 68. 2'b00:begin 69. cache\_valid\_1[index\_save] <= 1'b1;             //灏咰ache line缃负鏈夋晥 70. cache\_tag\_1  [index\_save] <= tag\_save; 71. cache\_block\_1[index\_save] <= write\_cache\_data\_1; //鍐欏叆Cache line 72. cache\_dirty\_1[index\_save] <= 1'b1; 73. LRU\_1[index\_save] <= 3; 74. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 75. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 76. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 77. end 78. 2'b01:begin 79. cache\_valid\_2[index\_save] <= 1'b1;             //灏咰ache line缃负鏈夋晥 80. cache\_tag\_2  [index\_save] <= tag\_save; 81. cache\_block\_2[index\_save] <= write\_cache\_data\_2; //鍐欏叆Cache line 82. cache\_dirty\_2[index\_save] <= 1'b1; 83. LRU\_2[index\_save] <= 3; 84. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 85. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 86. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 87. end 88. 2'b10:begin 89. cache\_valid\_3[index\_save] <= 1'b1;             //灏咰ache line缃负鏈夋晥 90. cache\_tag\_3  [index\_save] <= tag\_save; 91. cache\_block\_3[index\_save] <= write\_cache\_data\_3; //鍐欏叆Cache line 92. cache\_dirty\_3[index\_save] <= 1'b1; 93. LRU\_3[index\_save] <= 3; 94. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 95. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 96. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 97. end 98. 2'b11:begin 99. cache\_valid\_4[index\_save] <= 1'b1;             //灏咰ache line缃负鏈夋晥 100. cache\_tag\_4  [index\_save] <= tag\_save; 101. cache\_block\_4[index\_save] <= write\_cache\_data\_4; //鍐欏叆Cache line 102. cache\_dirty\_4[index\_save] <= 1'b1; 103. LRU\_4[index\_save] <= 3; 104. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 105. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 106. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 107. end 108. endcase 109. end 110. **else** **if**(write & cpu\_data\_req & hit) begin 111. //块命中后的处理（CPU发出写请求） 112. **if**(hit\_1)begin 113. cache\_block\_1[index] <= write\_cache\_data\_1;      //鍐欏叆Cache line锛屼娇鐢╥ndex鑰屼笉鏄痠ndex\_save 114. cache\_dirty\_1[index] <= 1'b1; 115. LRU\_1[index\_save] <= 3; 116. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_1[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 117. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_1[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 118. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_1[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 119. end 120. **else** **if**(hit\_2)begin 121. cache\_block\_2[index] <= write\_cache\_data\_2;      //鍐欏叆Cache line锛屼娇鐢╥ndex鑰屼笉鏄痠ndex\_save 122. cache\_dirty\_2[index] <= 1'b1; 123. LRU\_2[index\_save] <= 3; 124. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_2[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 125. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_2[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 126. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_2[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 127. end 128. **else** **if**(hit\_3)begin 129. cache\_block\_3[index] <= write\_cache\_data\_3;      //鍐欏叆Cache line锛屼娇鐢╥ndex鑰屼笉鏄痠ndex\_save 130. cache\_dirty\_3[index] <= 1'b1; 131. LRU\_3[index\_save] <= 3; 132. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_3[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 133. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_3[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 134. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_3[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 135. end 136. **else** **if**(hit\_4)begin 137. cache\_block\_4[index] <= write\_cache\_data\_4;      //鍐欏叆Cache line锛屼娇鐢╥ndex鑰屼笉鏄痠ndex\_save 138. cache\_dirty\_4[index] <= 1'b1; 139. LRU\_4[index\_save] <= 3; 140. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_4[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 141. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_4[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 142. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_4[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 143. end 144. end **else** **if**(read & cpu\_data\_req & hit)begin 145. //块命中后的处理（CPU发出读请求） 146. **if**(hit\_1)begin 147. LRU\_1[index\_save] <= 3; 148. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_1[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 149. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_1[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 150. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_1[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 151. end **else** **if**(hit\_2)begin 152. LRU\_2[index\_save] <= 3; 153. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_2[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 154. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_2[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 155. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_2[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 156. end **else** **if**(hit\_3)begin 157. LRU\_3[index\_save] <= 3; 158. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_3[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 159. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_3[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 160. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_3[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 161. end **else** **if**(hit\_4)begin 162. LRU\_4[index\_save] <= 3; 163. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_4[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 164. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_4[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 165. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_4[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 166. end 167. end 168. end 169. end   5.I\_cache的设计。  对于指令缓存，由于不存在CPU的写指令，因此其可以被设计为只读缓存，不需要额外考虑数据的写入，也就没有脏块的概念，因此，在I\_cache的设计中不需要实现写入内存的逻辑。  I\_cache说明  大小：4 \* 4KiB，每路4KiB  相连度：4路组相连  替换算法：LRU算法  访存策略：只读（只涉及到读，不涉及脏块的写入）  I\_cache的流程图    I\_cache的状态转换图    6，I\_cache的实现。  ①，cache的参数和储存模块，使用reg实现。   1. //cache参数 2. parameter  INDEX\_WIDTH  = 10, OFFSET\_WIDTH = 2; 3. localparam TAG\_WIDTH    = 32 - INDEX\_WIDTH - OFFSET\_WIDTH; 4. localparam CACHE\_DEEPTH = 1 << INDEX\_WIDTH; 6. //Cach储存参数 7. reg                 cache\_valid\_1 [CACHE\_DEEPTH - 1 : 0]; 8. reg [TAG\_WIDTH-1:0] cache\_tag\_1   [CACHE\_DEEPTH - 1 : 0]; 9. reg [31:0]          cache\_block\_1 [CACHE\_DEEPTH - 1 : 0];  12. reg                 cache\_valid\_2 [CACHE\_DEEPTH - 1 : 0]; 13. reg [TAG\_WIDTH-1:0] cache\_tag\_2   [CACHE\_DEEPTH - 1 : 0]; 14. reg [31:0]          cache\_block\_2 [CACHE\_DEEPTH - 1 : 0];  17. reg                 cache\_valid\_3 [CACHE\_DEEPTH - 1 : 0]; 18. reg [TAG\_WIDTH-1:0] cache\_tag\_3   [CACHE\_DEEPTH - 1 : 0]; 19. reg [31:0]          cache\_block\_3 [CACHE\_DEEPTH - 1 : 0];  22. reg                 cache\_valid\_4 [CACHE\_DEEPTH - 1 : 0]; 23. reg [TAG\_WIDTH-1:0] cache\_tag\_4   [CACHE\_DEEPTH - 1 : 0]; 24. reg [31:0]          cache\_block\_4 [CACHE\_DEEPTH - 1 : 0];  27. //LRU 28. reg [1:0]          LRU\_1 [CACHE\_DEEPTH - 1 : 0]; 29. reg [1:0]          LRU\_2 [CACHE\_DEEPTH - 1 : 0]; 30. reg [1:0]          LRU\_3 [CACHE\_DEEPTH - 1 : 0]; 31. reg [1:0]          LRU\_4 [CACHE\_DEEPTH - 1 : 0];   ②，地址分割并读取四路组对应index的块   1. //地址分割 2. wire [OFFSET\_WIDTH-1:0] offset; 3. wire [INDEX\_WIDTH-1:0] index; 4. wire [TAG\_WIDTH-1:0] tag; 6. assign offset = cpu\_inst\_addr[OFFSET\_WIDTH - 1 : 0]; 7. assign index = cpu\_inst\_addr[INDEX\_WIDTH + OFFSET\_WIDTH - 1 : OFFSET\_WIDTH]; 8. assign tag = cpu\_inst\_addr[31 : INDEX\_WIDTH + OFFSET\_WIDTH]; 10. //分别提取4路cache 11. wire c\_valid\_1,c\_valid\_2,c\_valid\_3,c\_valid\_4; 12. wire [TAG\_WIDTH-1:0] c\_tag\_1,c\_tag\_2,c\_tag\_3,c\_tag\_4; 13. wire [31:0] c\_block\_1,c\_block\_2,c\_block\_3,c\_block\_4;   17. assign c\_valid\_1 = cache\_valid\_1[index]; 18. assign c\_tag\_1   = cache\_tag\_1  [index]; 19. assign c\_block\_1 = cache\_block\_1[index];  22. assign c\_valid\_2 = cache\_valid\_2[index]; 23. assign c\_tag\_2   = cache\_tag\_2  [index]; 24. assign c\_block\_2 = cache\_block\_2[index];  27. assign c\_valid\_3 = cache\_valid\_3[index]; 28. assign c\_tag\_3   = cache\_tag\_3  [index]; 29. assign c\_block\_3 = cache\_block\_3[index];  32. assign c\_valid\_4 = cache\_valid\_4[index]; 33. assign c\_tag\_4   = cache\_tag\_4  [index]; 34. assign c\_block\_4 = cache\_block\_4[index];    39. wire hit\_1, miss\_1,hit\_2, miss\_2,hit\_3, miss\_3,hit\_4, miss\_4; 40. assign hit\_1 = c\_valid\_1 & (c\_tag\_1 == tag); 41. assign hit\_2 = c\_valid\_2 & (c\_tag\_2 == tag); 42. assign hit\_3 = c\_valid\_3 & (c\_tag\_3 == tag); 43. assign hit\_4 = c\_valid\_4 & (c\_tag\_4 == tag);   ③，判断缺失和命中，并通过LRU获取替换块的属性   1. //选出替换块的地址和数据块(LRU) 2. wire [1:0] change\_road; 4. //LRU 5. assign change\_road = LRU\_1[index]==0 ? 2'b00 : 6. LRU\_2[index]==0 ? 2'b01 : 7. LRU\_3[index]==0 ? 2'b10 : 8. 2'b11;   12. //判断缺失和命中 13. wire hit, miss; 14. assign hit = hit\_1 | hit\_2 | hit\_3 | hit\_4; 15. assign miss = ~hit;   ④，状态机的实现   1. //FSM 2. parameter H = 1'b0,NH = 1'b1; //H不需要读存储器，NH需要读存储器 3. reg  state; 4. reg mem\_read; 5. always @(posedge clk) begin 6. **if**(rst) begin 7. state <= H; 8. end 9. **else** begin 10. **case**(state) 11. H: begin 12. **if**((cpu\_inst\_req & miss))begin 13. state <= NH; 14. end **else** begin 15. state <= H; 16. end 17. end 18. NH:begin 19. state <= cache\_inst\_data\_ok ? H : NH; 20. end 21. endcase 22. end 23. end   ⑤，读内存的实现   1. //一次读内存操作 2. wire read\_req;      //读请求 3. reg addr\_rcv;       //内存地址接受成功到结束 4. wire read\_finish;  //读操作完毕 6. always @(posedge clk) begin 7. addr\_rcv <= rst ? 1'b0 : 8. cache\_inst\_req & cache\_inst\_addr\_ok ? 1'b1 : 9. read\_finish ? 1'b0 : addr\_rcv; 10. end 11. assign read\_req = state==NH; 12. assign read\_finish = cache\_inst\_data\_ok;   ⑥，连接mips和axi模块   1. //output to mips core 2. assign cpu\_inst\_rdata   = hit\_1 ? c\_block\_1 : 3. hit\_2 ? c\_block\_2 : 4. hit\_3 ? c\_block\_3 : 5. hit\_4 ? c\_block\_4 : cache\_inst\_rdata; 6. assign cpu\_inst\_addr\_ok = cpu\_inst\_req & hit | cache\_inst\_req & cache\_inst\_addr\_ok; 7. assign cpu\_inst\_data\_ok = cpu\_inst\_req & hit | read\_finish ;   11. //output to axi interface 12. assign cache\_inst\_req   = read\_req & ~addr\_rcv ; 13. assign cache\_inst\_wr    = cpu\_inst\_wr; 14. assign cache\_inst\_size  = cpu\_inst\_size; 15. assign cache\_inst\_addr  = cpu\_inst\_addr; 16. assign cache\_inst\_wdata = cpu\_inst\_wdata;   ⑦，时序操作，更新缓存部分和LRU饱和计数器。   1. //保存index和tag，防止之后发生改变 2. reg [TAG\_WIDTH-1:0] tag\_save; 3. reg [INDEX\_WIDTH-1:0] index\_save; 4. always @(posedge clk) begin 5. tag\_save   <= rst ? 0 : 6. cpu\_inst\_req ? tag : tag\_save; 7. index\_save <= rst ? 0 : 8. cpu\_inst\_req ? index : index\_save; 9. end 11. integer t; 12. always @(posedge clk) begin 13. //初始化 14. **if**(rst) begin 15. **for**(t=0; t<CACHE\_DEEPTH; t=t+1) begin 16. cache\_valid\_1[t] <= 0; 17. cache\_valid\_2[t] <= 0; 18. cache\_valid\_3[t] <= 0; 19. cache\_valid\_4[t] <= 0; 20. LRU\_1[t] <= 0; 21. LRU\_2[t] <= 0; 22. LRU\_3[t] <= 0; 23. LRU\_4[t] <= 0; 24. end 25. end 26. **else** begin 27. //块替换后的处理（CPU发出读请求） 28. **if**(read\_finish) begin 29. **case**(change\_road) 30. 2'b00:begin 31. cache\_valid\_1[index\_save] <= 1'b1; 32. cache\_tag\_1  [index\_save] <= tag\_save; 33. cache\_block\_1[index\_save] <= cache\_inst\_rdata; 34. LRU\_1[index\_save] <= 3; 35. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 36. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 37. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 38. end 39. 2'b01:begin 40. cache\_valid\_2[index\_save] <= 1'b1; 41. cache\_tag\_2  [index\_save] <= tag\_save; 42. cache\_block\_2[index\_save] <= cache\_inst\_rdata; 43. LRU\_2[index\_save] <= 3; 44. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 45. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 46. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 47. end 48. 2'b10:begin 49. cache\_valid\_3[index\_save] <= 1'b1; 50. cache\_tag\_3  [index\_save] <= tag\_save; 51. cache\_block\_3[index\_save] <= cache\_inst\_rdata; 52. LRU\_3[index\_save] <= 3; 53. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 54. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 55. LRU\_4[index\_save] <= LRU\_4[index\_save]==0 ? 0 : LRU\_4[index\_save] - 1; 56. end 57. 2'b11:begin 58. cache\_valid\_4[index\_save] <= 1'b1; 59. cache\_tag\_4  [index\_save] <= tag\_save; 60. cache\_block\_4[index\_save] <= cache\_inst\_rdata; 61. LRU\_4[index\_save] <= 3; 62. LRU\_2[index\_save] <= LRU\_2[index\_save]==0 ? 0 : LRU\_2[index\_save] - 1; 63. LRU\_3[index\_save] <= LRU\_3[index\_save]==0 ? 0 : LRU\_3[index\_save] - 1; 64. LRU\_1[index\_save] <= LRU\_1[index\_save]==0 ? 0 : LRU\_1[index\_save] - 1; 65. end 66. endcase 67. end **else** **if**(cpu\_inst\_req & hit)begin 68. //块命中后的处理（CPU发出读请求） 69. **if**(hit\_1)begin 70. LRU\_1[index\_save] <= 3; 71. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_1[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 72. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_1[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 73. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_1[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 74. end **else** **if**(hit\_2)begin 75. LRU\_2[index\_save] <= 3; 76. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_2[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 77. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_2[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 78. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_2[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 79. end **else** **if**(hit\_3)begin 80. LRU\_3[index\_save] <= 3; 81. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_3[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 82. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_3[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 83. LRU\_4[index\_save] <= LRU\_4[index\_save] > LRU\_3[index\_save]?  LRU\_4[index\_save] - 1 : LRU\_4[index\_save]; 84. end **else** **if**(hit\_4)begin 85. LRU\_4[index\_save] <= 3; 86. LRU\_2[index\_save] <= LRU\_2[index\_save] > LRU\_4[index\_save]?  LRU\_2[index\_save] - 1 : LRU\_2[index\_save]; 87. LRU\_3[index\_save] <= LRU\_3[index\_save] > LRU\_4[index\_save]?  LRU\_3[index\_save] - 1 : LRU\_3[index\_save]; 88. LRU\_1[index\_save] <= LRU\_1[index\_save] > LRU\_4[index\_save]?  LRU\_1[index\_save] - 1 : LRU\_1[index\_save]; 89. end 90. end 91. end 92. end |
| 四，程序运行结果  1，仿真结果 |

备注：