2021 春季学期
计算机组成原理
32 位 Cache
设
计
报
告

Hollow Man

32 位 Cache 设计报告

一、 课程设计简述

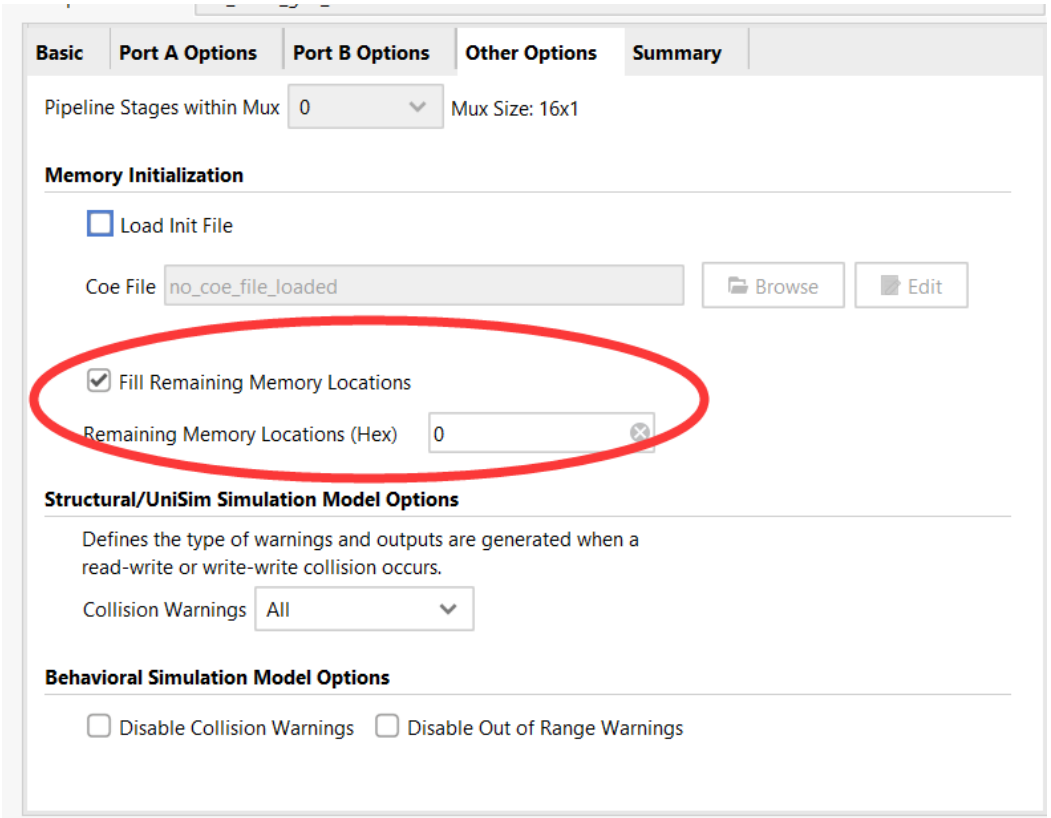
设计语言：Verilog 硬件描述语言

仿真环境：Vivado

实现 Cache 映射方式：直接映射、四路组相联映射

二、 设计原理

令内存的物理地址为 32 位，按字编址，则内存的后两位为块内地址。Cache 被分成了 2^{16} = 65536 块，则内存物理地址的前 14 位被划分为 tag 号。为了节省 Cache 空间，同时方便构建，所以本次设计没有给 Cache 中分配实际的数据位，而仅仅模拟 Cache 分配内存地址。设置 BRAM 地址为 16 位，初始化 BRAM 中的内存默认值均为 0。



内存物理地址划分：

31 Tag 18	17 Index 2	1 块内地址 0

三、 设计内容

Cache 模块均接受一个时钟信号,一个复位信号(低电平有效)和一个 32 位地址值作为输入

直接映射

设置 BRAM 的块宽为 15 位，存储一个有效位值和 Tag 值。

模块中使用如下端口和寄存器：

```
reg[15:0] miss;          //未命中
reg[15:0] hit;           //命中
reg[15:0]     w_addr;       //RAM PORT A 写地址
reg[14:0]    w_data;      //RAM PORT  A 写数据
reg          ena;         //RAM 使能，高电平有效
reg          wea;         //RAM PORT  A 写使能，高电平有效
reg          enb;         //RAM PORT  B 读使能，高电平有效
reg[15:0]     r_addr;      //RAM PORT  B 读地址
wire[14:0]   r_data;     //RAM PORT   B 读数据
```

模块首先根据 address 中的 Index 部分读出缓存中数据,然后通过判断有效位的值看出数据块是否有效。如果数据无效，则为未命中，自动模拟从内存装入改数据，即更改缓存中数据为当前物理地址 Tag 号，置有效位为 1。如果数据有效，则判断 Tag 号是否相符，不相符则未命中，替换为当前物理地址 Tag 号，置有效位为 1；相符则命中。

4 路组相联映射

设置 BRAM 的块宽为 92 位，分别存储四路地址数据的有效位，

最近一次访问的序列号以及 Tag 号。

92 位地址划分如下：

[91]　第一组的有效位
[90:83] 第一组的最近一次访问序列号
[82:69] 第一组的 Tag 号

[68]　第二组的有效位
[67:60] 第二组的最近一次访问序列号
[59:46] 第二组的 Tag 号

[45]　第三组的有效位
[44:37] 第三组的最近一次访问序列号
[36:23] 第三组的 Tag 号

[22]　第四组的有效位
[21:14] 第四组的最近一次访问序列号
[13:0]　第四组的 Tag 号

模块中使用如下端口和寄存器：

```
reg not_valid;          //记录当前记录的组号是否存放着无效数据
reg[1:0] least_set;     //LRU 算法完成后最远的组号
reg[7:0] temp;          //运行 LRU 算法时的临时寄存器
reg[7:0] count;         //当前查询的编号，用于 LRU 算法
reg[15:0] miss;         //未命中
reg[15:0] hit;          //命中
reg[15:0]     w_addr;       //RAM PORT A 写地址
reg[91:0]    w_data;      //RAM PORT  A 写数据
reg          ena;         //RAM 使能，高电平有效
reg          wea;         //RAM PORT  A 写使能，高电平有效
reg          enb;         //RAM PORT  B 读使能，高电平有效
reg[15:0]     r_addr;      //RAM PORT  B 读地址
wire[91:0]   r_data;    //RAM PORT   B 读数据
```

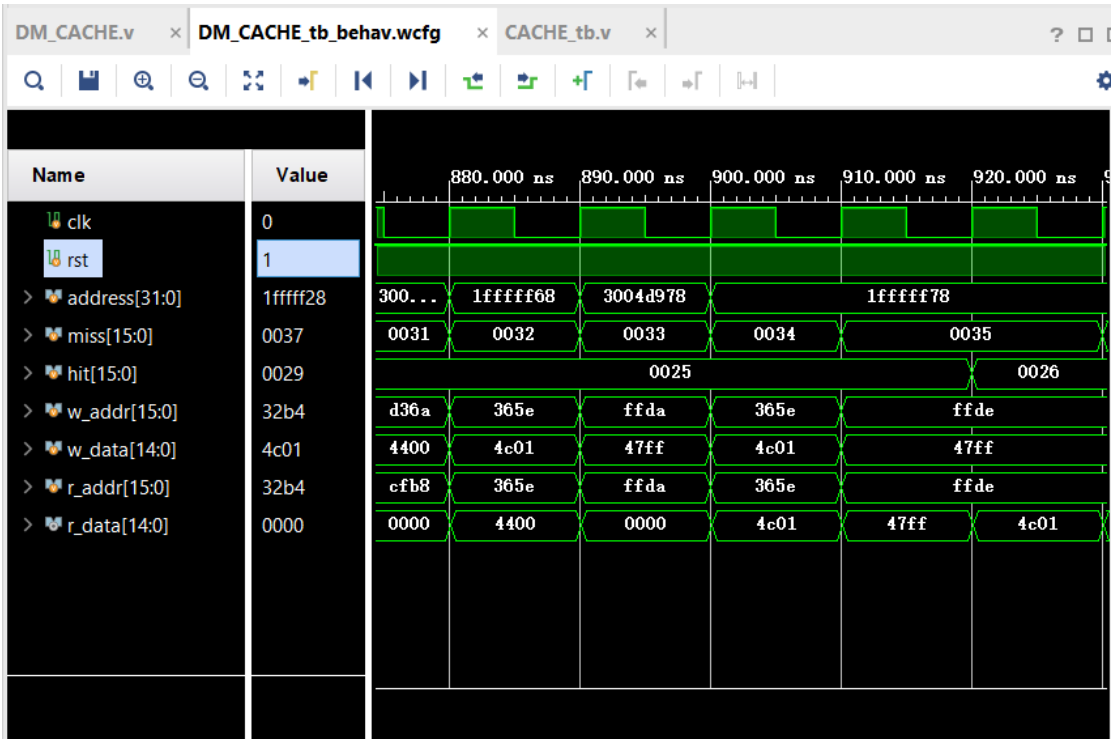
模块首先根据 address 中的 Index 部分读出缓存中数据，然后遍历四个组，通过判断有效位的值和 Tag 值来判断是否命中该组，不命中则继续遍历。在遍历的过程中获取当前组的访问序列号，挑选出最小值，以便最终需要替换时使用 LRU 算法进行替换。若

所有组都未命中，则选择数据有效位的值为 0 的组进行替换，若全部有效则使用 LRU 选择最近没被使用的组进行替换。

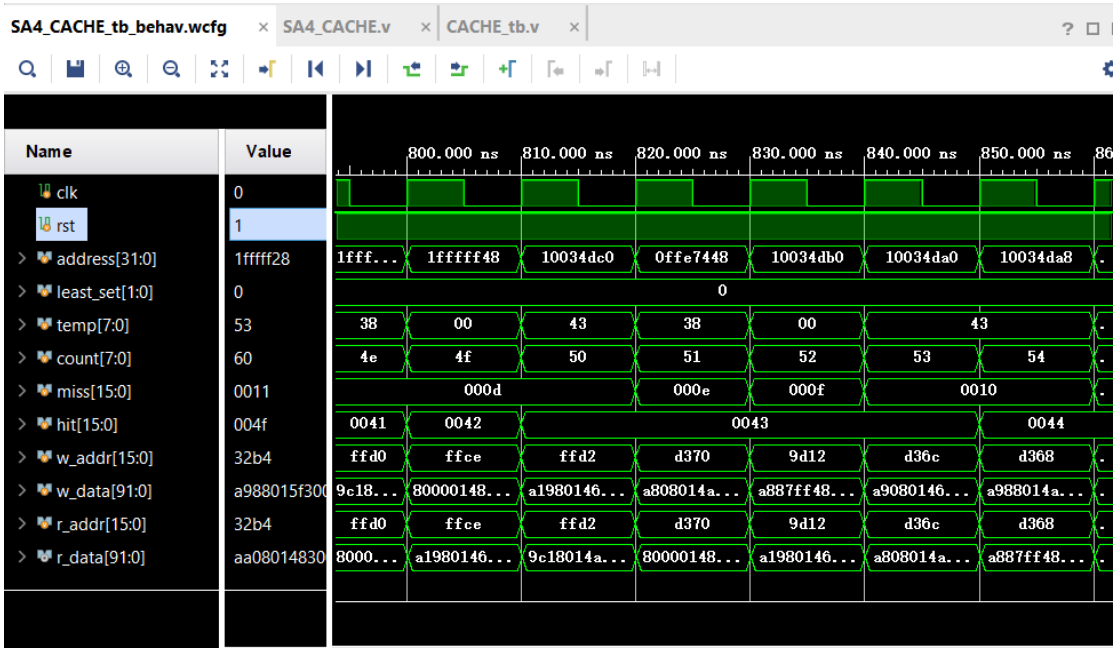
四、 测试结果

使用 96 条内存地址进行测试。数据来源为 gcc 程序运行时真实的前 96 条内存访问地址。最终得到如下结果：

直接映射



4 路组相联映射



通过 miss 寄存器和 hit 寄存器的值来计算命中比率：

	全相联	四路组相联
命中次数	41	79
不命中次数	55	17
命中率	0.427083	0.82291667
不命中率	0.572917	0.17708333

可以看出在同样的条件下，组相联方式在命中率上要明显优于全相联方式。

五、 附源代码

直接映射

```
`timescale 1ns / 1ps
module DM_CACHE(
input clk,                  //时钟
input rst,                  //复位信号，低电平有效
input [31:0] address
    );

reg[15:0] miss;           //未命中
reg[15:0] hit;            //命中
reg[15:0]      w_addr;       //RAM PORT A 写地址
reg[14:0]     w_data;      //RAM PORT  A 写数据
reg           ena;         //RAM 使能，高电平有效
reg           wea;         //RAM PORT  A 写使能，高电平有效
```

```
reg          enb;        //RAM PORT  B 读使能，高电平有效
reg[15:0]     r_addr;     //RAM PORT  B 读地址
wire[14:0]   r_data;    //RAM PORT   B 读数据

//全相联映射缓存
always@(posedge clk or negedge rst) begin
      if(!rst) begin  //重设相关数值
          ena <= 1'b0;
          enb <= 1'b0;
          r_addr <= 16'd0;
          wea <= 1'b0;
          w_addr <= 16'd0;
          w_data <= 15'd0;
          miss <= 16'd0;
          hit <= 16'd0;
      end
      else begin
            ena <= 1'b1;
            enb <= 1'b1;
            r_addr <= address[17:2];            //根据 Index 读出缓存中数据
            if(r_data[14]==0)                   //数据无效,自动模拟从内存装入数
据
            begin
                w_data[14]<=1;                  //有效位置为 1
                w_data[13:0]<=address[31:18];   //Tag
                w_addr<=address[17:2];      //Index
                miss<=miss+1'b1;                    //未命中
                wea <= 1'b1;
            end
            else
            begin
                if(r_data[13:0]==address[31:18]) //数据有效并且 tag 号相同
                begin
                    hit<=hit+1'b1;                  //命中
                end
                else
                begin                           //数据有效但未装入,直接替换装入
                    w_data[14]<=1;                  //有效位
                    w_data[13:0]<=address[31:18];   //Tag
                    w_addr <= address[17:2];        //Index
                    miss<=miss+1'b1;                //未命中
                    wea <= 1'b1;
                end
            end
```

```
      end
end

////////////////////////////////////////////////
//实例化 RAM
blk_mem_gen_0 ram_ip_test (
  .clka      (clk          ),          // input clka
  .ena       (ena          ),          // input [1 : 0] ena
  .wea       (wea          ),          // input [1 : 0] wea
  .addra     (w_addr       ),          // input [16 : 0] addra
  .dina      (w_data       ),          // input [15 : 0] dina
   .clkb     (clk          ),          // input clkb
   .enb      (enb          ),          // input [1 : 0] ena
   .addrb    (r_addr       ),          // input [16 : 0] addrb
   .doutb    (r_data       )           // output [15 : 0] doutb
  );

endmodule
```

4 路组相联映射

```
`timescale 1ns / 1ps
module SA4_CACHE(
input clk,                //时钟
input rst,                //复位信号，低电平有效
input [31:0] address
    );

reg not_valid;
reg[1:0] least_set;     //LRU 算法完成后最远的组号
reg[7:0] temp;          //运行 LRU 算法时的临时寄存器
reg[7:0] count;         //当前查询的编号，用于 LRU 算法
reg[15:0] miss;         //未命中
reg[15:0] hit;          //命中
reg[15:0]     w_addr;      //RAM PORT A 写地址
reg[91:0]    w_data;     //RAM PORT  A 写数据
reg          ena;        //RAM 使能，高电平有效
reg          wea;        //RAM PORT  A 写使能，高电平有效
reg          enb;        //RAM PORT  B 读使能，高电平有效
reg[15:0]     r_addr;     //RAM PORT  B 读地址
wire[91:0]   r_data;    //RAM PORT   B 读数据

//4 路组相联映射缓存
always@(posedge clk or negedge rst) begin
      if(!rst) begin  //重设相关数值
```

```
            ena <= 1'b0;
            enb <= 1'b0;
            r_addr <= 16'd0;
            wea <= 1'b0;
            w_addr <= 16'd0;
            w_data <= 15'd0;
            miss <= 16'd0;
            hit <= 16'd0;
            count <= 8'd0;
            temp <= 8'd0;
            least_set <= 2'b0;
            not_valid <= 1'b0;
        end
        else begin
              least_set <= 2'b0;
              ena <= 1'b1;
              enb <= 1'b1;
              r_addr <= address[17:2];            //根据 Index 读出缓存中数据
              if(r_data[91]==1 && r_data[82:69]==address[31:18])  //匹配到第一组
数据有效并且 tag 号相同
              begin
                  hit<=hit+1'b1;
                  w_data<=r_data;
                  w_data[90:83]<=count;       //更新被使用
                  w_addr<=address[17:2];      //Index
                  wea <= 1'b1;
              end
              else
              begin
                  if (r_data[91]==0)          //如果是无效位直接替换
                  begin
                      not_valid <= 1'b1;
                  end
                  else
                  begin
                      temp <= r_data[90:83];
                  end

                  if(r_data[68]==1 && r_data[59:46]==address[31:18])  //匹配到第
二组数据有效并且 tag 号相同
                  begin
                      hit<=hit+1'b1;
                      w_data<=r_data;
                      w_data[67:60]<=count;       //更新被使用
```

```
                w_addr<=address[17:2];      //Index
                wea <= 1'b1;
            end
            else
            begin
                if (r_data[68]==0)          //如果是无效位直接替换
                begin
                    not_valid <= 1'b1;
                    least_set <= 2'b1;
                end
                else
                begin
                    if(not_valid == 0 && temp > r_data[67:60])  // LRU 寻找
最远访问过的组
                    begin
                        least_set <= 2'b1;
                        temp <= r_data[67:60];
                    end
                end

                if(r_data[45]==1 && r_data[36:23]==address[31:18])  // 匹 配
到第三组数据有效并且 tag 号相同
                begin
                    hit<=hit+1'b1;
                    w_data<=r_data;
                    w_data[44:37]<=count;       //更新被使用
                    w_addr<=address[17:2];      //Index
                    wea <= 1'b1;
                end
                else
                begin
                    if (r_data[45]==0)          //如果是无效位直接替换
                    begin
                        not_valid <= 1'b1;
                        least_set <= 2'b10;
                    end
                    else
                    begin
                        if(not_valid == 0 && temp > r_data[44:37])  //  LRU
寻找最远访问过的组
                        begin
                            least_set <= 2'b10;
                            temp <= r_data[44:37];
                        end
```

```
                    end

                    if(r_data[22]==1 && r_data[13:0]==address[31:18])   //
匹配到第四组数据有效并且 tag 号相同
                    begin
                        hit<=hit+1'b1;
                        w_data<=r_data;
                        w_data[21:14]<=count;       //更新被使用
                        w_addr<=address[17:2];      //Index
                        wea <= 1'b1;
                    end
                    else
                    begin
                        if (r_data[22]==0)          //如果是无效位直接替换
                        begin
                            not_valid <= 1'b1;
                            least_set <= 2'b11;
                        end
                        else
                        begin
                            if(not_valid == 0 && temp > r_data[21:14])  //
LRU 寻找最远访问过的组

                            begin
                                least_set <= 2'b11;
                            end
                        end
                        w_data<=r_data;
                        //全部未命中，进行替换
                        case(least_set)
                            2'b0:
                            begin
                                w_data[91]<=1;              //有效位置为 1
                                w_data[90:83]<=count;       //更新被使用
                                w_data[82:69]<=address[31:18];//Tag
                            end
                            2'b1:
                            begin
                                w_data[68]<=1;              //有效位置为 1
                                w_data[67:60]<=count;       //更新被使用
                                w_data[59:46]<=address[31:18];//Tag
                            end
                            2'b10:
                            begin
                                w_data[45]<=1;              //有效位置为 1
```

```
                                w_data[44:37]<=count;       //更新被使用
                                w_data[36:23]<=address[31:18];//Tag
                            end
                            2'b11:
                            begin
                                w_data[22]<=1;              //有效位置为 1
                                w_data[21:14]<=count;       //更新被使用
                                w_data[13:0]<=address[31:18];//Tag
                            end
                        endcase
                        w_addr<=address[17:2];      //Index
                        wea <= 1'b1;
                        miss<=miss+1'b1;
                    end
                end
            end
        end
        count<=count+1'b1;                  //LRU 计数
    end
end

////////////////////////////////////////////
//实例化 RAM
blk_mem_gen_0 ram_ip_test (
  .clka      (clk          ),            // input clka
  .ena       (ena          ),            // input [1 : 0] ena
  .wea       (wea          ),            // input [1 : 0] wea
  .addra     (w_addr       ),            // input [16 : 0] addra
  .dina      (w_data       ),            // input [92 : 0] dina
   .clkb     (clk          ),            // input clkb
   .enb      (enb          ),            // input [1 : 0] ena
   .addrb    (r_addr       ),            // input [16 : 0] addrb
   .doutb    (r_data       )             // output [92 : 0] doutb
  );

endmodule
```

TestBench 激励文件(全相联，组相联把模块名称改了就好，大同小异)

```
`timescale 1ns / 1ps

module DM_CACHE_tb(
```

```
   );
reg clk;
reg rst;
reg[31:0] address;

DM_CACHE DM_CACHE_test(.clk(clk),.rst(rst),.address(address));

initial begin
rst = 1'b1;
clk = 1'b0;
address = 1'b0;
#5 rst = 1'b0;
#10 rst = 1'b1;
#5 clk =~clk;
   address <= 32'h1fffff50;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff58;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff88;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff90;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff98;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffffa0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffffa8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffffb0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffffb8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffffc0;
#5 clk =~clk;
#5 clk =~clk;
```

```
   address <= 32'h1fffffc8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffffd0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffffd8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffffe0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffffe8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffff0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1ffffff8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000000;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000008;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000010;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000018;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000020;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000028;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000030;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000038;
#5 clk =~clk;
```

```
#5 clk =~clk;
   address <= 32'h20000040;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000048;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000050;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000058;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000060;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000068;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000070;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000078;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000080;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000088;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000090;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h20000098;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000a0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000a8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000b0;
```

```
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000b8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000c0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000c8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000d0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000d8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000e0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000e8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000f0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h200000f8;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h30031f10;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d960;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d968;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004caa0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d970;
#5 clk =~clk;
#5 clk =~clk;
```

```
   address <= 32'h3004d980;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h30000008;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d970;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d960;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d968;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004caa0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff58;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff68;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff68;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d980;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h30000008;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d970;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d960;
#5 clk =~clk;
```

```
#5 clk =~clk;
   address <= 32'h3004d968;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004caa0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff58;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff40;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff38;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff40;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff38;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff48;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h10034dc0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h0ffe7448;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h10034db0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h10034da0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h10034da8;
```

```
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h10033ee0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff68;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004d978;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff78;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff78;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff78;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff78;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff30;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff28;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h3004cad0;
#5 clk =~clk;
#5 clk =~clk;
   address <= 32'h1fffff28;
#5 clk =~clk;
end

endmodule
```