浙江大学实验报告 Lab6

专业:计算机科学与技术　姓名:仇国智　学号:3220102181　日期:2024/6/11
课程名称:计算机组成与设计　实验名称:缓存设计　指导老师:刘海风　成绩:

操作方法与实验步骤

源文件编写

我的cache设计分为两部分,cache模块用于控制状态机,即空闲,读写,替换cache,写回内存之间的转换,cache_memory模块用于控制内存单元的读写的实现.

cache_memory模块

模块代码如下

```
`include "lab6.vh"
module cache_memory (
    input clk,
    input rst,
    input [`ADDR_WIDTH-1:0] addr,
    input [`WORD_WIDTH-1:0] write_data,
    input [`MEMORY_BLOCK_SIZE-1:0] mem_data,
    input [`CACHE_MEMORY_MODE_WIDTH-1:0] mode,
    output reg [`WORD_WIDTH-1:0] data,
    output reg [`ADDR_WIDTH-1:0] addr_replace,
    output reg [`MEMORY_BLOCK_SIZE-1:0] mem_data_replace,
    output reg dirty,
    output reg miss
);
  reg [`ENTRY_WIDTH-1:0] cache_data[`GROUP_SIZE-1:0][`WAY_SIZE-1:0];
  wire [`OFFSET_WIDTH-1:0] offset = addr[`OFFSET_WIDTH-1:0];
  wire [`INDEX_WIDTH-1:0] index = addr[`INDEX_WIDTH+`OFFSET_WIDTH-1:`OFFSET_WIDTH];
  wire [`TAG_WIDTH-1:0] tag = addr[`ADDR_WIDTH-1:`INDEX_WIDTH+`OFFSET_WIDTH];
  wire [`TAG_WIDTH-1:0]tag0 = cache_data[index][0][`TAG_WIDTH-1:0];
  wire [`TAG_WIDTH-1:0]tag1 = cache_data[index][1][`TAG_WIDTH-1:0];
  wire [`WORD_WIDTH-1:0] data0 ;
  wire [`WORD_WIDTH-1:0] data1 ;
  assign data0 =  (offset==`OFFSET_WIDTH'b0)?cache_data[index][0]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*1-1:`TAG_WIDTH+`LABEL_WIDTH]:
                (offset==`OFFSET_WIDTH'b1)?cache_data[index][0]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2-1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH]:
                (offset==`OFFSET_WIDTH'b10)?cache_data[index][0]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3-1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2]:
                cache_data[index][0][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*4-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3];
  assign data1 =  (offset==`OFFSET_WIDTH'b0)?cache_data[index][1]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*1-1:`TAG_WIDTH+`LABEL_WIDTH]:
```

```
                (offset==`OFFSET_WIDTH'b1)?cache_data[index][1]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2-1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH]:
                (offset==`OFFSET_WIDTH'b10)?cache_data[index][1]
[`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3-1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2]:
                cache_data[index][1][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*4-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3];
  wire dirty0 = cache_data[index][0][`D_BIT];
  wire dirty1 = cache_data[index][1][`D_BIT];
  wire valid0 = cache_data[index][0][`V_BIT];
  wire valid1 = cache_data[index][1][`V_BIT];
  wire lru0 = cache_data[index][0][`U_BIT];
  wire lru1 = cache_data[index][1][`U_BIT];
  wire replace=lru0?1:0;
  integer i, j;
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      data  <= 0;
      dirty <= 0;
      miss  <= 0;
      addr_replace <= 0;
      mem_data_replace <= 0;
      for (i = 0; i < `GROUP_SIZE; i = i + 1) begin
        for (j = 0; j < `WAY_SIZE; j = j + 1) begin
          cache_data[i][j] <= `ENTRY_WIDTH'b0;
        end
      end
    end else begin
      case (mode)
        `CACHE_MEMORY_IDLE: begin
          data <= 0;
          miss <= 0;
        end
        `CACHE_MEMORY_READ: begin
          if (tag0 == tag && valid0) begin
            data <= data0;
            miss <= 0;
            cache_data[index][0][`U_BIT] <= 1;
            cache_data[index][1][`U_BIT] <= 0;
          end else if (tag1 == tag && valid1) begin
            data <= data1;
            miss <= 0;
            cache_data[index][0][`U_BIT] <= 0;
            cache_data[index][1][`U_BIT] <= 1;
          end else begin
            data <= 0;
            miss <= 1;
          end
        end
        `CACHE_MEMORY_WRITE: begin
          if (tag0 == tag && valid0) begin
            case(offset)
              0: cache_data[index][0][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*1-
1:`TAG_WIDTH+`LABEL_WIDTH] <= write_data;
              1: cache_data[index][0][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH] <= write_data;
              2: cache_data[index][0][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2] <= write_data;
```

```
79              3: cache_data[index][0][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*4-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3] <= write_data;
80              endcase
81            cache_data[index][0][`D_BIT] <= 1;
82            cache_data[index][0][`U_BIT] <= 1;
83            cache_data[index][1][`U_BIT] <= 0;
84            miss <= 0;
85          end else if (tag1 == tag && valid1) begin
86            case(offset)
87              0: cache_data[index][1][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*1-
1:`TAG_WIDTH+`LABEL_WIDTH] <= write_data;
88              1: cache_data[index][1][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH] <= write_data;
89              2: cache_data[index][1][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*2] <= write_data;
90              3: cache_data[index][1][`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*4-
1:`TAG_WIDTH+`LABEL_WIDTH+`WORD_WIDTH*3] <= write_data;
91              endcase
92            cache_data[index][1][`D_BIT] <= 1;
93            cache_data[index][1][`U_BIT] <= 1;
94            cache_data[index][0][`U_BIT] <= 0;
95            miss <= 0;
96          end else begin
97            miss <= 1;
98            end
99        end
100       `CACHE_MEMORY_LOAD: begin
101         cache_data[index][replace][`TAG_WIDTH-1:0] <= tag;
102         cache_data[index][replace][`D_BIT] <= 0;
103         cache_data[index][replace][`V_BIT] <= 1;
104         cache_data[index][replace][`U_BIT] <= 0;
105         cache_data[index][replace][`ENTRY_WIDTH-1:`TAG_WIDTH+`LABEL_WIDTH] <= mem_data;
106         addr_replace <= {cache_data[index][replace][`TAG_WIDTH-1:0], index, 2'b00};
107         mem_data_replace <= cache_data[index][replace][`ENTRY_WIDTH-
1:`TAG_WIDTH+`LABEL_WIDTH];
108         dirty <= cache_data[index][replace][`D_BIT];
109       end
110     endcase
111   end
112 end
113 endmodule
```

cache_memory模块主要用于实现一个两路组相联的高速缓存,包含了对数据读取,写入以及加载操作的处理.模块包括以下几个部分:

- 输入和输出信号
  - 输入信号
    - clk: 时钟信号,用于同步操作.
    - rst: 复位信号,高电平时重置缓存.
    - addr: 地址信号,表示需要访问的内存地址.
    - write_data: 写入数据信号,当执行写操作时的数据.
    - mem_data: 从主存加载的内存块数据.
    - mode: 模式信号,指示当前的缓存操作模式(空闲,读取,写入,加载等).
  - 输出信号

    - `data`: 输出数据,当读取操作时的数据输出.
    - `addr_replace`: 替换地址,用于指示需要替换的缓存块的地址.
    - `mem_data_replace`: 替换的内存数据,用于指示需要替换的内存块数据.
    - `dirty`: 脏位标志,指示缓存块是否需要写回主存.
    - `miss`: 未命中标志,指示当前操作是否命中缓存.
- 内部结构
    - 缓存数据存储
      `cache_data`是一个二维寄存器数组,用于存储缓存数据.每个缓存条目包含标签,有效位,LRU位,脏位和实际数据.
    - 地址分解
        - `offset`: 从地址中提取的偏移量,用于定位数据块中的具体位置.
        - `index`: 从地址中提取的索引,用于选择缓存组.
        - `tag`: 从地址中提取的标签,用于验证缓存命中.
    - 缓存条目属性
        - `tag0`和`tag1`: 缓存组中两个缓存路的标签.
        - `data0`和`data1`: 根据偏移量提取的两个缓存路的实际数据.
        - `dirty0`和`dirty1`: 脏位,指示缓存条目是否被修改.
        - `valid0`和`valid1`: 有效位,指示缓存条目是否有效.
        - `lru0`和`lru1`: 最近最少使用位,用于替换策略.
- 操作模式
    - 复位
      当`rst`信号为高时,所有缓存条目和输出信号复位.
    - 空闲模式
      在`CACHE_MEMORY_IDLE`模式下,缓存处于空闲状态,不执行任何操作,输出信号清零.
    - 读取模式
      在`CACHE_MEMORY_READ`模式下,模块根据地址中的标签和索引查找缓存条目:
        - 如果缓存命中(标签匹配且有效位为高),输出数据,更新LRU位.
        - 如果未命中,设置`miss`信号为高.
    - 写入模式
      在`CACHE_MEMORY_WRITE`模式下,模块根据地址中的标签和索引查找缓存条目:
        - 如果缓存命中(标签匹配且有效位为高),将数据写入缓存,设置脏位,更新LRU位.
        - 如果未命中,设置`miss`信号为高.
    - 加载模式
      在`CACHE_MEMORY_LOAD`模式下,模块将从主存加载的数据写入缓存:
        - 选择需要替换的缓存路(根据LRU位).
        - 更新缓存条目的标签,有效位和数据.
        - 设置替换地址和替换数据信号,以便进行写回操作.
- 工作流程
    1. 初始化和复位
        - 当复位信号`rst`为高时,所有缓存条目清零,所有输出信号清零.
    2. 读取操作
        - 检查缓存是否命中(标签匹配且有效).

- 如果命中,输出对应的数据,并更新LRU位.
- 如果未命中,设置miss信号为高.

3. 写入操作
   - 检查缓存是否命中(标签匹配且有效).
   - 如果命中,根据偏移量写入数据,设置脏位,并更新LRU位.
   - 如果未命中,设置miss信号为高.
4. 加载操作
   - 选择需要替换的缓存路(根据LRU位).
   - 将从主存加载的数据写入缓存条目,更新标签,有效位和数据.
   - 设置替换地址和替换数据信号,用于写回主存.

这个模块通过以上步骤,实现了一个基本的两路组相联缓存的读写控制和替换策略.

cache模块

模块代码如下

```
`include "lab6.vh"
module cache (
    input clk,
    input rst,
    input [`ADDR_WIDTH-1:0] addr,
    input [`WORD_WIDTH-1:0] write_data,
    input [`MEMORY_BLOCK_SIZE-1:0] mem_data,
    input [`REQUIRE_MODE_WIDTH-1:0] require_mode,
    input memory_ready,
    output reg MemRW_read,
    output reg MemRW_write,
    output reg ready,
    output reg [`MEMORY_BLOCK_SIZE-1:0] mem_data_out,
    output reg [`ADDR_WIDTH-1:0] mem_addr_out,
    output reg [`WORD_WIDTH-1:0] data

    // output wire miss,
    // output wire [`REQUIRE_MODE_WIDTH-1:0] mode_inner
);
  reg [`CACHE_STATE_WIDTH-1:0] state;
  reg change;

  wire [`WORD_WIDTH-1:0] data_inner;
  wire [`ADDR_WIDTH-1:0] addr_replace;
  wire [`MEMORY_BLOCK_SIZE-1:0] mem_data_replace;
  wire dirty;
  wire miss;

  wire [`CACHE_MEMORY_MODE_WIDTH-1:0] mode_inner;

  assign mode_inner =   state==`CACHE_STATE_IDLE ? `CACHE_MEMORY_IDLE :
                        state==`CACHE_STATE_RW&&require_mode==`REQUIRE_MODE_READ ?
`CACHE_MEMORY_READ :
                        state==`CACHE_STATE_RW&&require_mode==`REQUIRE_MODE_WRITE ?
`CACHE_MEMORY_WRITE :
                        state==`CACHE_STATE_ALLOCATE&&memory_ready ? `CACHE_MEMORY_LOAD:
                        `CACHE_MEMORY_IDLE;
```

```
  cache_memory U1 (
      .clk(clk),
      .rst(rst),
      .addr(addr),
      .write_data(write_data),
      .mem_data(mem_data),
      .mode(mode_inner),
      .data(data_inner),
      .addr_replace(addr_replace),
      .mem_data_replace(mem_data_replace),
      .dirty(dirty),
      .miss(miss)
  );
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      state <= `CACHE_STATE_IDLE;
      change <= 1;
      MemRW_read <= 0;
      MemRW_write <= 0;
      ready <= 0;
      mem_data_out <= 0;
      mem_addr_out <= 0;
      data <= 0;
    end else begin
      case (state)
        `CACHE_STATE_IDLE: begin
          if (require_mode == `REQUIRE_MODE_READ || require_mode == `REQUIRE_MODE_WRITE)
begin
            state  <= `CACHE_STATE_RW;
            change <= 0;
            ready  <= 0;
          end
        end
        `CACHE_STATE_RW: begin
          if (change) begin
            if (miss) begin
              state <= `CACHE_STATE_ALLOCATE;
              MemRW_read <= 1;
              mem_addr_out <= addr;
              change <= 0;
            end else begin
              state  <= `CACHE_STATE_IDLE;
              data   <= data_inner;
              ready  <= 1;
              change <= 1;
            end
          end else begin
            change <= 1;
          end
        end
        `CACHE_STATE_ALLOCATE: begin
          if (change) begin
            if (dirty) begin
              state <= `CACHE_STATE_WB;
              MemRW_write <= 1;
              MemRW_read <= 0;
              mem_addr_out <= addr_replace;
              mem_data_out <= mem_data_replace;
```

```
            end else begin
              state <= `CACHE_STATE_RW;
              MemRW_read <= 0;
            end
            change <= 0;
          end else begin
            if (memory_ready) begin
              change <= 1;
            end
          end
        end
        `CACHE_STATE_WB: begin
          if (change) begin
            state <= `CACHE_STATE_RW;
            MemRW_write <= 0;
            change <= 0;
          end else begin
            if (memory_ready) begin
              change <= 1;
            end
          end
        end
      endcase
    end
  end
endmodule
```

cache模块实现了缓存控制器,用于管理缓存和主存之间的数据交互,并维护缓存状态机的转换.该模块的输入信号包括时钟信号clk、复位信号rst、表示访问内存地址的addr、用于写操作的数据write_data、从主存加载的内存块数据mem_data、指示当前操作是读取还是写入的请求模式require_mode以及指示主存数据是否准备好的memory_ready.输出信号则包括内存读使能信号MemRW_read、内存写使能信号MemRW_write、表示缓存操作完成的ready信号、用于写回操作的数据mem_data_out、用于读写操作的地址mem_addr_out以及用于读操作返回的数据data.

模块内部包含若干寄存器和信号.state寄存器表示缓存状态机的当前状态,change寄存器作为状态变化标志以控制状态机转换.内部信号还包括从cache_memory模块中读取的数据data_inner、需要替换的缓存地址addr_replace、需要替换的内存数据mem_data_replace、指示缓存块是否需要写回主存的脏位标志dirty、指示缓存操作是否未命中的未命中标志miss以及缓存内存模块的操作模式mode_inner.

在模块内部,cache_memory模块被实例化,并通过转换当前状态和请求模式为cache_memory模块的操作模式mode_inner来进行操作.实例化的cache_memory模块连接了相关信号,用于实际的缓存数据管理.

状态机的实现包括以下几个状态:

1. CACHE_STATE_IDLE状态:当请求模式为读取或写入时,状态机会转换到CACHE_STATE_RW状态,准备进行读写操作.
2. CACHE_STATE_RW状态:在此状态下,如果缓存命中,直接返回数据并转换到CACHE_STATE_IDLE状态,同时设置ready信号.如果缓存未命中,状态机会转换到CACHE_STATE_ALLOCATE状态,准备从主存加载数据.
3. CACHE_STATE_ALLOCATE状态:如果当前缓存块是脏的,需要写回主存,状态机会转换到CACHE_STATE_WB状态,并开始写回操作.如果缓存块不是脏的,则直接从主存加载数据,并转换到CACHE_STATE_RW状态.
4. CACHE_STATE_WB状态:在写回操作完成后,状态机会转换到CACHE_STATE_RW状态,继续进行读写操作.

通过复位信号rst,所有状态和输出信号将被复位.在空闲状态CACHE_STATE_IDLE,模块等待读写请求.当接收到读写请求时,状态机会进入读写状态CACHE_STATE_RW.在读写状态中,模块会判断缓存是否命中.如果命中,直接返回数据;如果未命中,进入分配状态CACHE_STATE_ALLOCATE,从主存加载数据.在分配状态中,如果当前缓存块是脏的,则需要写回主存,进入写回状态CACHE_STATE_WB.如果缓存块不是脏的,则直接从主存加载数据.完成写回操作后,继续进行读写操作.

通过这些状态机的设计,cache模块能够有效地管理缓存和主存之间的数据交互,确保缓存的一致性和操作的高效性.

头文件

下面是模块中用的宏定义

```
// require mode
`define REQUIRE_MODE_WIDTH 2
`define REQUIRE_MODE_IDLE `REQUIRE_MODE_WIDTH'b00
`define REQUIRE_MODE_READ `REQUIRE_MODE_WIDTH'b01
`define REQUIRE_MODE_WRITE `REQUIRE_MODE_WIDTH'b10

// cache state
`define CACHE_STATE_WIDTH 2
`define CACHE_STATE_IDLE `CACHE_STATE_WIDTH'b00
`define CACHE_STATE_RW `CACHE_STATE_WIDTH'b01
`define CACHE_STATE_ALLOCATE `CACHE_STATE_WIDTH'b10
`define CACHE_STATE_WB `CACHE_STATE_WIDTH'b11

// cache memory mode
`define CACHE_MEMORY_MODE_WIDTH 2
`define CACHE_MEMORY_IDLE `CACHE_MEMORY_MODE_WIDTH'b00
`define CACHE_MEMORY_READ `CACHE_MEMORY_MODE_WIDTH'b01
`define CACHE_MEMORY_WRITE `CACHE_MEMORY_MODE_WIDTH'b10
`define CACHE_MEMORY_LOAD `CACHE_MEMORY_MODE_WIDTH'b11

// cache parameters
`define WAY_SIZE 2
`define ADDR_WIDTH 32
`define WORD_WIDTH 32
`define TAG_WIDTH 23
`define OFFSET_WIDTH 2
`define INDEX_WIDTH `WORD_WIDTH-`TAG_WIDTH-`OFFSET_WIDTH
`define MEMORY_BLOCK_SIZE `WORD_WIDTH*(1<<`OFFSET_WIDTH)
`define GROUP_SIZE 1<<`INDEX_WIDTH

`define V_BIT `TAG_WIDTH
`define D_BIT `TAG_WIDTH+1
`define U_BIT `TAG_WIDTH+2
`define LABEL_WIDTH 3
`define ENTRY_WIDTH `MEMORY_BLOCK_SIZE+`TAG_WIDTH+`LABEL_WIDTH
```

仿真

仿真激励代码

编写仿真激励代码如下

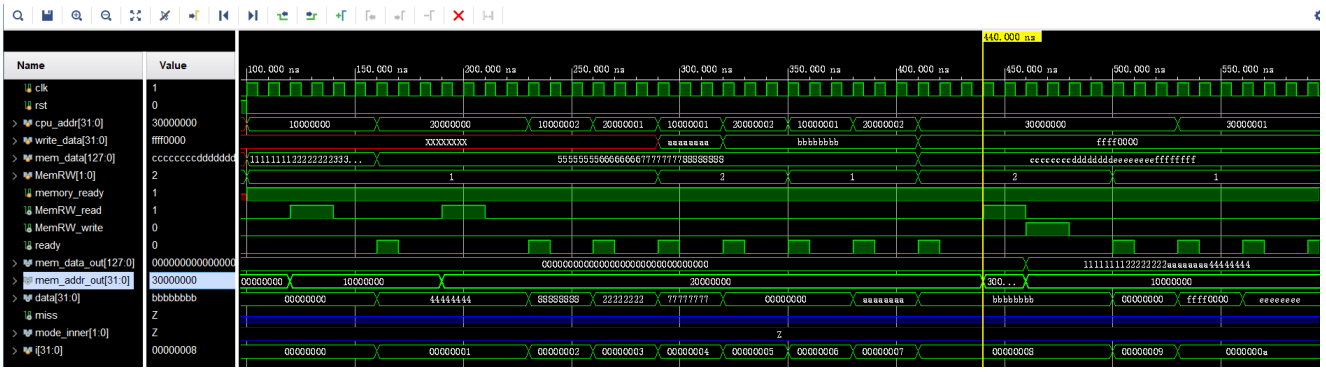
```
`timescale 1ns / 1ps
`include "../../sources_1/new/lab6.vh"
module cache_tb ();
  reg clk;
  reg rst;
  reg [31:0] cpu_addr;
```

```
reg [31:0] write_data;
reg [127:0] mem_data;
reg [1:0] MemRW;
reg memory_ready;
wire MemRW_read;
wire MemRW_write;
wire ready;
wire [127:0] mem_data_out;
wire [31:0] mem_addr_out;
wire [31:0] data;
wire miss;
wire [`REQUIRE_MODE_WIDTH-1:0] mode_inner;

// MemRW 0 idle
// MemRW 1 read
// MemRW 2 write
initial begin
  clk    = 1;
  rst    = 1;
  MemRW = 0;
  #100;
  rst = 0;
  memory_ready = 1;
  // Read Miss
  cpu_addr = 32'h10000000;
  MemRW = 1;
  mem_data = 128'h11111111222222223333333344444444;
end
integer i = 0;
always @(posedge ready) begin
  case (i)
    0: begin
      // Read Miss
      cpu_addr = 32'h20000000;
      mem_data = 128'h55555555666666667777777788888888;
      //#100;
      i = i + 1;
    end
    1: begin
      // Read Hit
      cpu_addr = 32'h10000002;
      //#100;
      i = i + 1;
    end
    2: begin
      cpu_addr = 32'h20000001;
      //#100;
      i = i + 1;
    end
    3: begin
      // Write Hit
      MemRW = 2;
      cpu_addr = 32'h10000001;  // 写第一个字
      write_data = 32'hAAAAAAAA;
      //#100;
      i = i + 1;
    end
    4: begin
```

```
        cpu_addr = 32'h20000002;  // 写第二个字
        write_data = 32'hBBBBBBBB;
        //#100;
        i = i + 1;
      end
      5: begin
        // Read Hit 检验刚刚写的内容是否被写进去了
        MemRW = 1;
        cpu_addr = 32'h10000001;
        //#100;
        i = i + 1;
      end
      6: begin
        cpu_addr = 32'h20000002;
        //#100;
        i = i + 1;
      end
      7: begin
        // Write miss, write back and allocate
        MemRW = 2;
        cpu_addr = 32'h30000000;  // 需要驱赶一个块
        write_data = 32'hFFFF0000;
        mem_data = 128'hCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFF;
        //#100;
        i = i + 1;
      end
      8: begin
        MemRW = 1;
        cpu_addr = 32'h30000000;
        //#100;
        i = i + 1;
      end
      9: begin
        cpu_addr = 32'h30000001;
        //#100;
        i = i + 1;
      end
    endcase
  end
  always #5 clk = ~clk;


  cache U1 (
      .clk(clk),
      .rst(rst),
      .addr(cpu_addr),
      .write_data(write_data),
      .mem_data(mem_data),
      .require_mode(MemRW),
      .memory_ready(memory_ready),
      .MemRW_read(MemRW_read),
      .MemRW_write(MemRW_write),
      .ready(ready),
      .mem_data_out(mem_data_out),
      .mem_addr_out(mem_addr_out),
      .data(data)
  );
endmodule
```

仿真图像

仿真图像如下



仿真分析

仿真结果符合预期.首先拉高rst位重置cache,然后降低rst进入测试点,每个测试点在cache输出ready后进入下一个测试点.以下是详细的测试点分析:

测试点0：读取未命中(Read Miss)

在时刻100ns,cpu_addr设置为0x10000000,MemRW设置为1表示读取操作,mem_data设置为0x11111111222222223333333344444444.由于是第一次访问该地址,cache未命中(miss信号为高),于是发起内存读取操作(MemRW_read为高),并将mem_addr_out设置为0x10000000.内存数据加载完成后,ready信号被拉高,表明读取操作完成,缓存被更新,data输出为0x44444444(对应0x10000000地址的数据).

测试点1：读取未命中(Read Miss)

cpu_addr设置为0x20000000,mem_data设置为0x55555555666666667777777788888888.由于这个地址也未被缓存,cache再次未命中,发起内存读取操作.mem_addr_out更新为0x20000000,内存数据加载完成后,ready信号被拉高,缓存更新,data输出为0x88888888(对应0x20000000地址的数据).

测试点2：读取命中(Read Hit)

cpu_addr设置为0x10000002.由于之前已经加载过地址0x10000000及其相邻地址,因此这次读取操作在缓存中命中.ready信号被拉高,数据从缓存中直接返回,data输出为0x22222222(对应0x10000002地址的数据).

测试点3：读取命中(Read Hit)

cpu_addr设置为0x20000001.由于之前已经加载过地址0x20000000及其相邻地址,因此这次读取操作在缓存中命中.ready信号被拉高,数据从缓存中直接返回,data输出为0x77777777(对应0x20000001地址的数据).

测试点4：写入命中(Write Hit)

MemRW设置为2表示写入操作,cpu_addr设置为0x10000001,write_data设置为0xAAAAAAAA.由于地址0x10000001在缓存中命中,写入操作直接在缓存中完成.ready信号被拉高,表明写入操作完成,缓存数据被更新.

测试点5：写入命中(Write Hit)

cpu_addr设置为0x20000002,write_data设置为0xBBBBBBBB.这个地址在缓存中命中,写入操作直接在缓存中完成.ready信号被拉高,缓存数据更新.

测试点6：读取命中(Read Hit)测试写入

MemRW设置为1表示读取操作,cpu_addr设置为0x10000001.读取操作命中缓存,检验之前写入的数据是否正确.ready信号被拉高,数据从缓存中直接返回,验证缓存写入正确性,data输出为0xAAAAAAAA,对应测试点4的写入.

测试点7：读取命中(Read Hit)测试写入

cpu_addr设置为0x20000002.读取操作命中缓存,检验之前写入的数据是否正确.ready信号被拉高,数据从缓存中直接返回,验证缓存写入正确性,data输出为0xBBBBBBBB,对应测试点5的写入.

测试点8：写入未命中(Write Miss)

MemRW设置为2表示写入操作,cpu_addr设置为0x30000000,write_data设置为0xFFFF0000,mem_data设置为0xCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFF.由于缓存需要替换一个块,cache发起写回操作(MemRW_write为高),将脏块写回内存,mem_addr_out设置为替换的块地址(根据LRU原则为0x10000000),mem_data_out为脏块数据.写回完成后,新的数据块从内存加载到缓存.

测试点9：读取命中(Read Hit)测试写入未命中的写入

MemRW设置为1表示读取操作,cpu_addr设置为0x30000000.读取操作命中缓存,ready信号被拉高,数据从缓存中直接返回,data输出为0xFFFF0000(对应测试点8).

测试点10：读取命中(Read Hit)测试写入未命中的读取

cpu_addr设置为0x30000001.读取操作命中缓存,ready信号被拉高,数据从缓存中直接返回,data输出为0xEEEEEEEE(对应测试点8)

通过这些测试点,可以验证缓存控制器在处理读写操作、缓存命中与未命中、缓存替换和写回等方面都能正确工作.每个测试点的操作结果与预期相符,表明缓存控制器的设计是正确和有效的.

思考题

- 实验只设计实现数据缓存,若实现指令缓存,设计方法是否一样？指令缓存也会存在写回,写分派现象吗？指令缓存的内容如果需要修改,如何操作?
  - 实现方法:实现指令缓存的方法与数据缓存基本相似,主要区别在于数据缓存处理的数据来自于程序的数据访问(读写),而指令缓存处理的是程序指令的存取(只读).指令缓存只需处理读取操作,而数据缓存则需处理读取和写入操作.
  - 写回和写分派:指令缓存一般不会存在写回和写分派现象.因为指令缓存主要是用来存储程序的指令,CPU从指令缓存中读取指令并执行.指令的内容通常不会在运行时修改,因此不需要写回到内存,也不需要处理写分派.
  - 修改:可以通过刷新清空指令缓存的方式来修改指令缓存的内容.先通过某种方式(比如说通过数据缓存)更改内存中的指令内容,然后刷新指令缓存,使得CPU重新从内存中读取指令并更新指令缓存.
- 带缓存的流水线CPU如何实现,当发生缺失的情况时CPU应该如何应对?
  - 暂停流水线:当CPU在执行过程中发现需要的数据或指令不在缓存中,暂停流水线操作.具体方法可以是暂停相应的流水线阶段,防止指令进入执行阶段.
  - 发起内存访问:向内存发起内存读取请求,以获取缺失的数据或指令.
  - 数据或指令填充缓存:当主存响应请求并返回数据或指令后,将其填充到缓存中,以便后续访问能直接从缓存中获取.
  - 恢复流水线:数据或指令填充完成后,CPU恢复暂停的流水线操作,并继续执行被暂停的指令.