Lab7 存储器体系实验

存储器是计算机系统非常重要的部件，用来存放程序和数据。程序运行时需要的数据、中间值和最终运行结果都保存在存储器中。存储器容量指它能存放的二进制位数或字节数。存储器的访问时间也称为存取时间，是指访问一次数据所用的时间。存储器容量和访问时间应能随着处理器速度的提高而同步提高，以保持系统性能的平衡。然而，随着时间的推移，处理器和存储器在性能上的差异越来越大。

计算机系统对存储器的需求是大容量、高速度和低成本，但从现有技术看，某一种存储元件制造的存储器很难同时满足这些要求。计算机中的存储器不仅有主存，还有寄存器、高速缓存、磁盘、磁带、光盘等，它们各自有不同的速度、容量和价格。在计算机中把各种不同容量和不同存取速度的存储器按一定的结构有机地组织在一起，以形成层次化存储结构，使得整个存储系统在速度、容量和价格等方面具有较好的综合性能指标。

本次实验将学习内部存储器和高速缓冲存储器的设计方法。数据存储器在 CPU 运行中存储全局变量、堆栈等数据。要求实现至少 128kB 大小的数据存储器容量。并且，该数据存储器需要支持在边沿上进行读取和写入操作。RV32I 的字长是 32bit，但是，数据存储器不仅要支持 32bit 数据的存取，同时也需要支持按字节（8bit) 或半字(16bit)大小的读取。由于单周期 CPU 需要在一个周期内完成一条指令的所有操作，需要数据 RAM 有独立的读时钟和写时钟。其中读取操作在系统时钟的上升沿进行（即一个时钟周期的一半时刻），写操作在系统时钟的下降沿进行（即一个时钟周期的结束时刻）。建议使用双端口 RAM（RAM 2 PORT）来实现数据存储器。

一、实验目的

1. 掌握队列、栈的设计方法。
2. 掌握数据存储器的设计方法和应用。
3. 掌握高速缓冲存储器Cache设计方法。

二、实验环境

1. Vivado 开发环境

2. Nexys A7-100T 实验板

三、实验原理

存储器的核心部件是一个由能存储 0 或 1 的记忆单元构成的存储阵列，以及 I/O 电路、地址译码和控制电路等组成部件。存储器的端口包括输入端、输出端和控制端口。输入端口包括：读/写地址端口、数据输入端口等；输出端口一般指的是输出数据端口；控制端口包括时钟端、读/写使能端口和读写数据长度等。存储器的工作过程如下：

写数据：在时钟有效沿时，如果写使能有效，则读取数据输入端口上的数据，将其存储到写入地址所指定的存储单元中。

读数据：存储器的读取输出可以受时钟和使能端的控制，也可以不受时钟和使能端的控制。如果输出受时钟的控制，则在时钟有效沿，将读取地址所指示的单元中的数据，传输到数据输出端口上；如果不受时钟的控制，则只要读取地址有效，就立即将此地址所指的单元中的数据送到数据输出端口上。

实验板上的 FPGA 芯片中实现存储器有两种方式：一种是分布式 RAM，FPGA 芯片中有 15,850 个逻辑片，最大分布式 RAM 容量为 1188Kb。另一种是专用集成块 BRAM，FPGA 芯片中有 135 个块 RAM，每个块 RAM 容量 36Kb，共计 4,860Kb 的 BRAM。在容量方面，BRAM 可用的容量比分布式 RAM 大的多。在时序方面，BRAM 读和写操作都采用同步方式，需要 1 个时钟周期，而分布式 RAM 写操作采用同步方式，而读操作则可以采用异步方式，这个特性有利于实现单周期 CPU 的指令存储器。

在实例化一个 BRAM 后，即使只是占用到该 RAM 的一小部分，而综合后，可能会消耗一整块 RAM 的资源（每个块 RAM 可以配置为两个独立的 18 Kb RAM），BRAM 在 FPGA 中是按列分布，可能造成用户逻辑和 BRAM 之间的布线较长和延时增加。

在 Verilog HDL 中，可以用多维数组定义存储器。定义一个存储器有 1024 个存储单元，每个存储单元有 32 位。在 Verilog 语言中可以作如下变量声明：

reg [31:0] mem4K [1023:0];

存储单元 mem4K [0]~ mem4K [1023]，每个存储单元都是 32 位的存储空间。

默认情况下，综合工具软件会自动选择存储器的最佳实现方法，RAM_STYLE 属性则指示综合工具如何实现 RAM 存储器。RAM_STYLE 属性设置为“block”，使用块 RAM 来实现；“distributed”，使用 LUT 搭建分布式 RAM；“registers”，使用寄存器组来替代 RAM；“ultra”，使用 UltraScale 中的 URAM。如果该属性在定义 RAM 的信号处声明，它仅适用于该信号；如果在某一层次结构处声明，则会应用于当前层次中的所有 RAM。例如：

(* ram_style=" distributed " *) reg [31:0] mem4K [1023:0];

表示使用分布式存储器来实现 mem4K 存储器。

初始化存储器 RAM 的方法，一种方法是在 Initial 语句中使用数组赋值语句来实现，如：

```
integer i;
initial for (i=0;i<1024;i=i+1) mem4K[i]=32'b0;
```

第二种方法是使用 Verilog 中的文件读取函数从外部数据文件中获取 RAM 或 ROM 的初始化数据。数据文件必须是 ASCII 文本文件，每一行表示 RAM 中的一个地址，文件中数据采用二进制或十六进制表示。二进制数据文件使用 Verilog 的系统任务$readmemb 来读取，十六进制数据文件使用$readmemh 来读取，使用格式如下所示：

$readmemh(“数据文件名”，存储器名[，起始地址[，结束地址]])；

数据文件的内容只能包含：空白位置(空格、回车、制表符等)，注释行，二进制或十六进制的数字。数据文件被读取时，读取的数据都被存放到存储器地址连续的单元中。存放地址范围可以通过系统任务声明语句中的起始地址和结束地址来说明，如果声明语句中未定义，则从地址 0 单元开始存储。如果需要数据存放到指定的存储器地址中，需要在数据文件中定义指定的地址，其格式为：@地址 数据。

假设文本文件 ram.data 内容如下：

```
01234567
89ABCDEF
5E5A6553
63594149
47480000
```

则，使用系统任务$readmemh 初始化存储器 0-4 个存储单元的语句如下：

```
Initial begin    $readmemh("ram.data",mem4k,0,4);    end     //初始化存储器内容
```

1、寄存器堆

寄存器堆在 CPU 中用于暂存指令执行过程中用到的中间数据，由许多寄存器组成，每个寄存器有一个编号，CPU 可以对指定编号的寄存器进行读写。一个带时钟控制的双口寄存器堆有两个读口和一个写口，每个读口包括一个读取寄存器编号输入端和一个数据输出端，写口包括一个吸入寄存器编号输入端和一个写入数据端。此外，还有一个写使能输入端 WE，当它有效时，才能在时钟触发边沿到来时，将写入数据端口 busW 上的数据写入指定寄存器中。

支持 RV32I 指令集的 CPU 中的寄存器堆包含 32 个 32 位寄存器，0 号寄存器始终为 0。 则采用寄存器使用的寄存器堆的代码参考设计如下：

```
module registerfile(
    output wire [31:0] busa,     //寄存器 ra 输出数据
    output wire [31:0] busb,     //寄存器 rb 输出数据
    input clk,
    input [4:0] ra,              //读寄存器编号 ra
```

```
    input [4:0] rb,              //读寄存器编号 rb
    input [4:0] rw,              //写寄存器编号 rw
    input [31:0] busw,           //写入数据端口
    input we                     //写使能端，为 1 时，可写入
    );
  (* ram_style="registers" *) reg [31:0] regfiles[0:31];      //综合时使用寄存器实现寄存器堆
    always @(posedge clk)   begin
        if((we==1'b1))  regfiles[rw]<=busw;                   //写端口
     end
    assign     busa=(ra==5'b0) ? 32'b0 : regfiles[ra];        //读端口 ra
    assign     busb=(rb==5'b0) ? 32'b0 : regfiles[rb];        //读端口 rb
endmodule
```

2、主要存储器

在计算机体系中，主要存储器（主存）用来存储在 CPU 运行中的程序二进制代码和数据，以及全局变量和堆栈等数据。主存容量一般比较大，支持在时钟边沿上进行读取和写入操作。RV32I 的字长是 32 位，读取指令时都是 32 位；而存取数据的长度不仅仅是 32 位，还要支持 8 位和 16 位数据的存取。

在数据存取过程中，通过 MemOP 信号来控制读写数据的长度，RV32I 中的存储访问指令与 Memop 对应关系如表 7.1 所示：

表 7.1 MemOp 控制信号含义

MemOp	指令	含义
000	lb,sb	存取 1 个字节，在读取时，按符号位扩展到 4 字节
001	lh,sh	存取半字，在读取时，按符号位扩展到 4 字节
010	lw,sw	存取 4 字节
100	lbu	读取 1 个字节数据，0 扩展到 4 字节
101	lhu	读取半字，0 扩展到 4 字节

对于读取操作，可以直接读取 32 位的数据，然后根据 MemOP 来判断是需要 8 位，16 位 还是 32 位的数据，再选择合适的数据拼接成所需的结果。对于写入操作，由于需要对 32 位中特定字节写入数据，而不能破坏其他字节的数据。

一种方法是在写入之前先读取原有 32 位数据，修改后再写入 32 位数据。对于单周期 CPU 而言，需要在一个周期内完成一条指令的所有操作，因而可设置读取操作在时钟的上升沿进行，写操作在系统时钟的下降沿进行。此时要求地址信号要在时钟上升沿到达前就准备好，并增加 Memop 控制信号。

考虑存储器地址边界对齐，采用先读后写、按字编址（字长 32 位）的存储容量为 256KB 的数据存储器参考代码设计如下：

```
module DataMem(
    output reg [31:0] dataout,          //数据输出
    input clk,                          //时钟信号
    input we,                           //存储器写使能信号，高电平时允许写入数据
    input [2:0] MemOp,                  //读写字节数控制
    input [31:0] datain,                //下输入数据
    input [15:0] addr                   //存储器地址
);
(* ram_style="block" *)    reg [31:0] ram [2**16-1:0];    //设置使用块 RAM 综合成存储器
reg [31:0] intmp;
reg [31:0] outtmp;
always @(posedge clk)     begin
          outtmp <= ram[addr[15:2]]; //上升沿读取存储器数据，先读取地址中数据
end
always @(negedge clk)    begin
     if (we) ram[addr[15:2]] <= intmp; //下降沿写入存储器数据，写使能有效时，写入数据
end
always @(*)
begin
    if (~we)    begin    //读取操作
                case (MemOp)
                  3'b000: //lb
                    begin dataout = {{24{outtmp[7]}}, outtmp[7:0]}; end
                  3'b001: //lh
                    begin dataout = {{16{outtmp[15]}}, outtmp[15:0]}; end
                  3'b010: begin dataout = outtmp; end
                  3'b100: begin dataout = {24'h000000, outtmp[7:0]}; end
                  3'b101: begin dataout = {16'h0000, outtmp[15:0]}; end
                  default:dataout = outtmp;
                endcase
          end
     else    begin    //写入操作
                case (MemOp)
                     3'b000: begin intmp = {outtmp[31:8], datain[7:0]}; end
                     3'b001: begin intmp = {outtmp[31:16], datain[15:0]}; end
                     3'b010: begin intmp = datain; end
                    default:intmp = datain;
                endcase
          end
 end
endmodule
```

3、高速缓冲存储器

高速缓冲存储器 cache 是一种小容量高速缓冲存储器，直接制作在 CPU 芯片内，速度几乎与 CPU 一样快。

大量典型程序运行情况分析的结果表明，在较短时间间隔内，程序产生的地址往往集中在存储器的一个很小范围，这种现象称为程序访问的局部性。因为程序由指令和数据组成，指令在主存按顺序存放，其地址连续，循环程序段或子程序段通常被重复执行，因此，指令的访问具有明显的局部化特性；而数据在主存一般也是连续存放，特别是数组元素，常常被按序重复访问，因此，数据也具有明显的访问局部化特征。

在 CPU 和主存之间设置 Cache 的作用是把主存中被频繁访问的程序块和数据块复制到 Cache 中。为了便于 Cache 和主存间交换信息，Cache 和主存空间都被划分为大小相等的区域，是 Cache 和主存之间的信息交换单位。主存中的区域称为块（block），Cache 中存放一个主存块的区域称为 Cache 行。

CPU 执行程序过程中，需要从主存取指令或读数据时，先检查 Cache 中有没有要访问的信息，若有，就直接从 Cache 中读取，而不用访问主存储器；若没有，再从主存中把当前访问信息所在的一个主存块复制到 Cache 中，因此，Cache 中的内容是主存中部分内容的副本。

在将主存块复制到 Cache 行时，主存块和 Cache 行之间必须遵循一定的映射规则，这样，CPU 要访问某个主存单元时，可以依据映射规则，到 Cache 对应的行中查找要访问的信息，而不用在整个 Cache 中查找。

根据不同的映射规则，主存块和 Cache 行之间有以下三种映射方式。

（1）直接（direct）映射：每个主存块映射到 cache 的固定行中。

（2）全相联（fully associate）映射：每个主存块映射到 Cache 的任意行中。

（3）组相联（set associate）映射：每个主存块映射到 Cache 的固定组的任意行中。

直接映射的基本思想是把每一个主存块映射到固定的 Cache 行中，也称**模映射**，其映射关系如下：

cache 行号=主存块号 mod cache 行数

例如，假定 cache 共有 16 行，根据 100 mod 16 = 4，可知，主存第 100 块应映射到 cache 的第 4 行中。

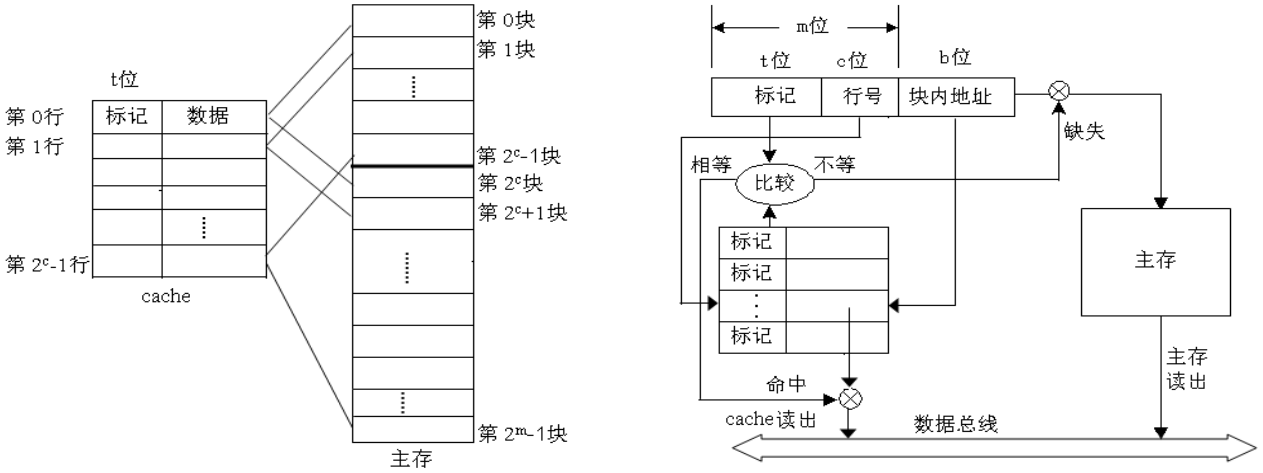
通常 cache 的行数是 2 的幂次，假定 cache 有 2^c 行，主存有 2^m 块，这个映射函数的含义就是以 m 位主存块号中后 c 位作为对应的 cache 行号来进行 cache 映射。也就是说，m 位块号中低 c 位相同的那些内存块，即“同余”内存块，将被映射到同一个 cache 行，形成一个“多对一”的映射关系，如图 7.1(a)所示。由映射函数可看出，主存块号的低 c 位正好是它要装入的 cache 行号。在 cache 中，给每一个行设置一个 t 位

长的标记（tag），此处 $t = m–c$，主存某块调入 cache 后，就将其块号的高 t 位设置在对应 cache 行的标记中，表示该行中存放的信息来自主存中哪个对应主存块。

根据以上分析可知，主存地址被分成以下三个字段：

标记 tag	cache 行号	块内地址

其中，高 t 位为标记，中间 c 位为 cache 行号（也称行索引），剩下的低位地址为块内地址，若一个主存块占 2^b 个单元，则块内地址占 b 个二进位。



(a) cache 和主存间的映射关系　　(b) CPU 访存过程

图 7.1　cache 和主存之间的直接映射方式

访存过程如下：首先将访存地址中间 c 位作为索引，直接找到对应的 cache 行，将对应 cache 行中的标记和主存地址高 t 位标记进行比较，若相等并有效位为 1，则访问 cache 命中，此时，根据主存地址中最低 b 位的块内地址，在对应的 cache 行中存取信息；若不相等或有效位为 0，则 cache 缺失，此时，CPU 从主存中读出该地址所在的一块信息，送到对应的 cache 行中，将有效位置 1，并将标记设置为地址中的高 t 位，同时将该地址中的内容送 CPU，如图 7.1(b)所示。

CPU 访存时，读操作和写操作的过程有一些不同，相对来说，读操作比写操作简单。因为 cache 行中的信息是主存某块的副本，所以，在写操作时会出现 cache 行和主存块数据的一致性问题。

全相联映射的基本思想是：一个主存块可装入 cache 任意一行中。全相联映射 cache 中，每行的标记用于指出该行的信息来自哪个主存块。因为一个主存块可能在任意一行中，所以需要比较所有 cache 行的标记。因此，主存地址中只有标记和块内地址两个字段。全相联映射方式下，只要有空闲 cache 行，就不会发生冲突，因而块冲突概率低。

组相联映射的主要思想是：将 cache 所有行分成 2^q 个大小相等的组，每组有 2^s 行。每个主存块被映射到 cache 固定组中的任意一行，即组间模映射、组内全映射。映射关系如下。

cache 组号=主存块号 mod cache 组数

例如，假定 8K 字的 cache 划分为 2^3 组×2^1 行/组×512 字/行，则主存第 100 块应映射到 cache 第 4 组的任意一行中，因为 100 mod 2^3= 4。

如此设置的 2^q 组×2^s 行/组的 cache 映射方式称为 2^s 路组相联映射，即 s=1 为 2 路组相联；s=2 为 4 路组相联；以此类推。通过对主存块号取模，使得每 2^q 个主存块与 2^q 个 cache 组一一对应，主存地址空间实际上被分成了若干组群，每个组群中有 2^q 个主存块对应于 cache 的 2^q 个组。假设主存地址有 n 位，块内地址占 b 位，有 2^m 个组群，则 n=m+q+b，主存地址被划分为以下三个字段。

标记	cache 组号	块内地址

其中，高 m 位为标记，中间 q 位为组号（也称组索引），剩下的 b 位低位地址部分为块内地址。标记字段的含义表示当前地址所在的主存块位于主存哪个组群。

图 7.2 所示的是采用 2 路组相联映射的 cache，整个访存过程如下。① 根据主存地址中的 cache 组号找到对应组；② 将地址中的标记与对应组中每个行的标记 Tag 进行比较；③ 将比较结果和有效位 V 相“与”；④ 若有一路比较相等并有效位为 1，则 Hit 为 1，并选中这一路 cache 行中的主存块；⑤ 在 Hit 为 1 的情况下，根据主存地址中的块内地址从选中的一块内取出对应单元的信息，若 Hit 不为 1，则 CPU 要到主存去读一块信息到 cache 行中。

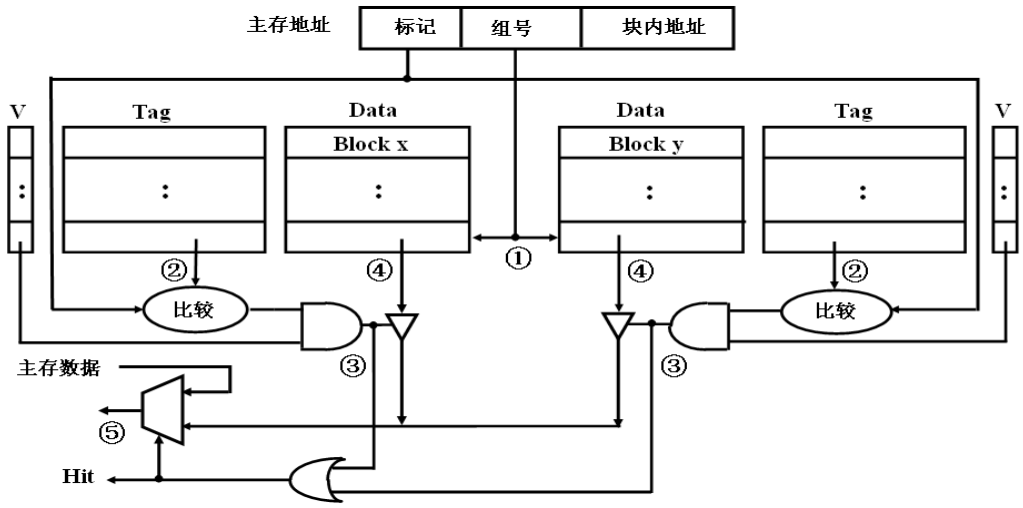


图 7.2 组相联映射方式的硬件实现

主存地址划分以及主存块和 cache 行的对应关系如图 7.3 所示。

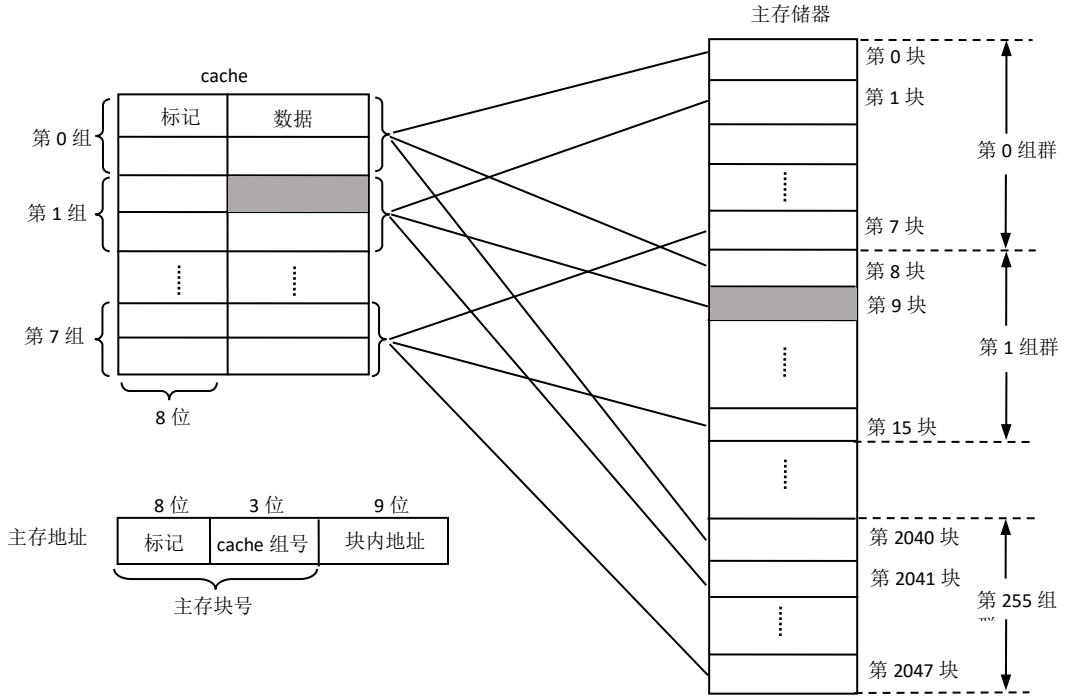


图 7.3 组相联映射方式下主存块和 cache 行对应关系

组相联映射方式结合了直接映射和全相联映射的优点。当 cache 的组数为 1 时，变为全相联映射；当每组只有一个 cache 行时，则变为直接映射。组相联映射的冲突概率比直接映射低，由于只有组内各行采用全相联映射，所以比较器的位数和个数都比全相联映射少，易于实现，查找速度也快得多。

Cache 替换算法：cache 行数比主存块数少很多，因此，往往多个主存块会映射到同一个 cache 行中。当一个新主存块复制到 cache 时，cache 中对应行可能已经全部被占满，此时，必须选择淘汰掉一个 cache 行中的主存块，使得该行中能存放新主存块。常用的替换算法有：先进先出、最近最少用、最不经常用和随机替换算法等。

Cache 一致性：因为 cache 中内容是主存块的副本，当对 cache 中的内容进行更新时，就存在 cache 和主存如何保持一致的问题。通常有两种写操作方式。

1．通写法

通写法的基本做法是：写操作时，若写命中，则同时写 cache 和主存；若写不命中，则有以下两种处理方式。

（1）**写分配法**。先在主存块中更新相应存储单元，然后分配一个 cache 行，将更新后的主存块装入到分配的 cache 行中。这种方式可以充分利用空间局部性，但每次写不命中都要从主存读一个块到 cache 中，增加了读主存块的开销。

（2）**非写分配法**。仅更新主存单元而不装入主存块到 cache 中。这种方式可以减少读入主存块的时间，但没有很好利用空间局部性。

显然，采用通写法使得 cache 和主存的一致性能得到充分保证。但是，这种方法会大大增加写操作的开销。

2．回写法

回写法的基本做法是：当 CPU 执行写操作时，若写命中，则信息只被写入 cache 而不被写入主存；若写不命中，则在 cache 中分配一行，将主存块调入该 cache 行中并更新相应单元的内容。因此，该方式下在写不命中时，通常采用写分配法进行写操作。

由此可见，该方式实际上采用的是回头再写回或最后一次性写的做法，因此，该方式通常被称为**一次性写方式**或**写回法**。

在 CPU 执行写操作时，回写法不会更新主存单元，只有当 cache 行中的主存块被替换时，才将该块内容一次性写回主存。这种方式的好处在于减少了写主存的次数，因而大大降低了主存带宽需求。为了减少主存块回写的开销，每个 cache 行设置了一个**修改位**（dirty bit，也称为**“脏位”**）。若修改位为 1，则说明对应 cache 行中的主存块被修改过，替换时需要写回主存；若修改位为 0，则说明对应主存块未被修改过，替换时无须写回主存。

由于回写法没有同步更新 cache 和主存内容，所以存在 cache 和主存内容不一致而带来的潜在隐患。通常需要其他的同步机制来保证存储信息的一致性。

决定系统访存性能的重要因素包括 cache 命中率和缺失损失，它们与 cache 设计的许多方面有关。显然，cache 容量越大，命中率就越高。此外，cache 命中率还与主存块大小有一定关系。采用大的交换单位能很好地利用空间局部性，但是，较大的主存块需要花费较多的时间来存取，因此，缺失损失会变大。由此可见，主存块大小必须适中，不能太大，也不能太小。当然，缺失损失还与写策略有关。

除了上述提到的这些问题外，设计 cache 时，还要考虑采用单级还是多级 cache、数据 cache 和指令 cache 是分开还是合并、主存–总线–cache–CPU 之间采用什么架构等问题，在 L1 cache 采用分离 cache，即数据 cache（Dcache）和指令 cache（Icache）分开设置。

更多内容参考《数字逻辑与计算机组成》教材第 9 章的相关内容、威斯康辛大学的计算机体系结构课程 CS552 中的 Cache 设计实验和在线 Cache 设计资源。

四、实验内容

1、存储器读写实验

利用 4 片 8 位 64KB 的 RAM 级联成一个 32 位 256KB 的存储器，每片 RAM 对应着 32 位存储器的特定字节数据。按字节编制，利用存储器地址的最低两位来选择对应的 8 位 RAM 片，其余的地址位连接到 4 片 RAM 的地址引脚上，将 4 片 RAM 的 8 位数据输入输出引脚拼接成 32 位数据位。

根据表 7.1 所示的 Memop 控制信定义实现该存储器，并在实验开发板上进行验证。具体要求如下：

1）、根据 MemOp 信号能够按字节、半字和字读写数据。

2）、32 位输出数据显示在 8 个七段数码管上。

3）、利用按钮 BTNC 实现清零复位功能。

4）、利用开关 SWTICH[2:0]实现 MemOp 信号的输入，开关 SWTICH[3]表示写入使能信号，开关 SWTICH[7:4]表示地址信号的最低 4 位，开关 SWTICH[15:8]表示输入 1 个字节数据。

5）、利用字节写入功能，写入 32 位数据，要求每一个字节数据不同，然后按照字节、半字和字进行读取。

6）、利用半字写入功能，写入 32 位数据，要求每半字数据不同，然后按照字节、半字和字进行读取。

假设 8 位存储器模块 mem8b 的接口定义如下：

```
module mem8b(
    output reg [7:0] dataout,          //输出数据
    input cs,                          //片选信号，高电平有效。有效时，存储器正常工作
    input clk,                         //时钟信号
    input we,                          //存储器写使能信号，高电平时允许写入数据
    input [7:0] datain,                //输入数据
    input [15:0] addr                  //16 位存储器地址，存储容量 64KB
);
  reg [7:0] ram [2**16-1:0];   //设置使用块 RAM 综合成存储器
// Add your code here
endmodule
```

假设 32 位存储器模块 mem32b 的接口定义如下：

```
module mem32b(
    output reg [31:0] dataout,         //输出数据
```

```
    input clk,                      //时钟信号
    input we,                       //存储器写使能信号，高电平时允许写入数据
    input [2:0] MemOp,              //读写字节数控制信号
    input [31:0] datain,            //输入数据
    input [15:0] addr               //16 位存储器地址
);
// Add your code here
endmodule
```

假设 32 位存储器上层实验接口模块 mem32b-top 的接口定义如下：

```
module mem32b_top(
     output [6:0]SEG,
     output [7:0]AN,                 //显示 32 位输出数值
     output [15:0] dataout_L8b,      //输出数据低 16 位
     input CLK100MHZ,                //系统时钟信号
     input BTNC,                     //复位清零信号
     input [2:0] MemOp,              //读写字节数控制信号
     input we,                       //存储器写使能信号，高电平时允许写入数据
     input [3:0] addr_L4b,           //地址位低 4 位，高位可以指定为 0 或其他任意值
     input [7:0] datain_L8b          //输入数据低 8 位，可重复 4 次，或高位指定为任意值
 );
 // Add your code here
endmodule
```

请根据上述描述，按照下列步骤完成实验。

1、 使用 Vivado 创建一个新工程。

2、 点击添加设计源码文件，加入 lab7.zip 里的 mem8b.v、mem32b.v、mem32b_top.v 等文件。

3、 点击添加测试文件，加入 lab7.zip 里的 datamen_tb .v 文件。

4、 点击添加约束文件，加入 lab7.zip 里的 mem32b_top.xdc 文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成位流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

2、高速缓冲存储器实验（选做）

处理器访问内存通常需要花费较长时间，为了尽可能降低访存时间，需要在 CPU 设计中实现了 Cache 处理机制。Cache 存储了内存中部分数据的副本，当 CPU 访问内存时，会优先查看访问时间较短的 Cache，只有当 Cache 中不存在需要访问的数据时，才会访问内存。Cache 和 CPU、主存之间的接口如图 7.4 所示。

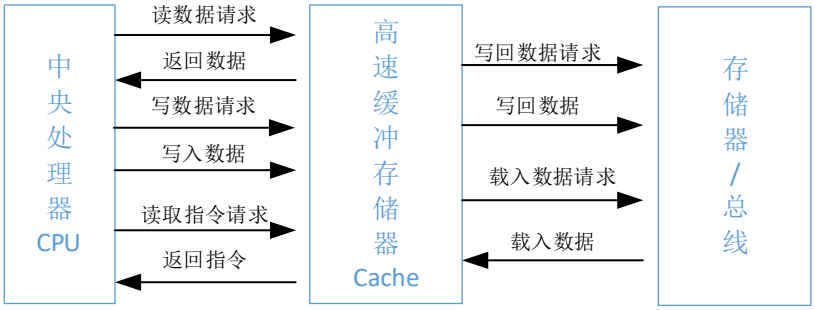


图 7.4 CPU、Cache 和主存之间接口

Cache 的状态机如图 7.5 所示，在 Cache 空闲时，状态机处于 Idle 状态。当有读写请求输入时，进入查找状态，首先检查 Cache 是否命中，如果命中，则返回空闲状态，并依据请求类型返回相应的数据，并更新 dirty 位信息；当未命中时，如果脏位 dirty 为真，则进入写回状态，存储器写回 Cache 中被更新的内存块，然后进入载入状态；如果脏位 dirty 为假，则进入载入状态，从存储器中读取内存块，替换 Cache 中块数据，返回查找状态。

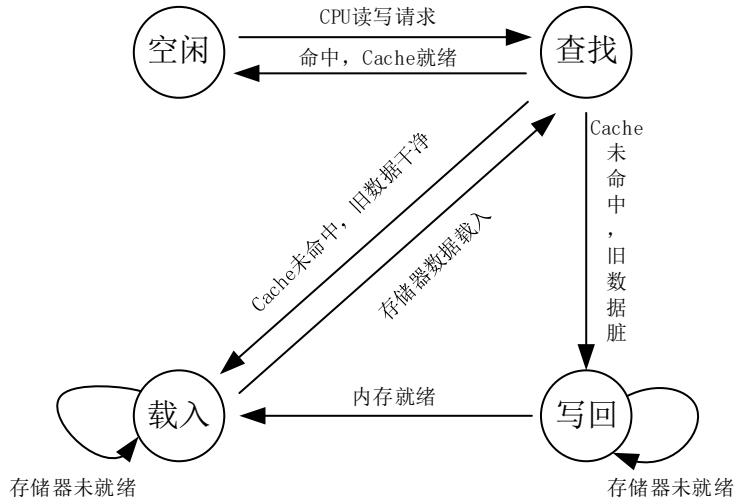


图 7.5 Cache 状态机

空闲状态 IDEL：等待来自处理器的有效读写请求。

查找比对状态 COMPARE：根据地址比较读写请求是否命中，根据 Cache 的映射方式，比较请求地址数据是否在 cache 中。如果当前 cache 块的 valid 有效，请求地址段的 Tag 部分和 Cache 块的 Tag 相同；则命中 hit 信号有效，无论读写数据请求都返回 Cache 就绪信号。如果是写入请求，则当前 Cache 块的脏位 Dirty 信号有效。如果未命中，则需要替换 cache 块内容，如果脏位 dirty 信号有效，则进入写回状态，否则进入载入状态。

写回状态 WRITEBACK：当存储器处于就绪状态后，根据 Cache 块的 tag 和 index 字段组合成地址信息，把 Cache 块的数据写回到存储器中。

载入状态 LOADMEM：当存储器处于就绪状态后，读取存储器的内容，替换 Cache 块，返回比对状态。

这是一个简单的 Cache 状态，在比对状态中把地址比较和读写合在一个单周期中，实际应用中，可以分开以便提升单周期的频率。另一个可以优化的方法是，建立一个写缓冲区，保存脏块数据。这样遇到脏位信号有效时，可以直接读取新的存储器数据，而不需要等待两个存储器访问周期。当 CPU 在处理请求数据时，cache 可以把缓冲区的数据写回存储器。

实验要求：假设计算机的主存地址空间大小为 256KB，采用字节编址方式。Cache 的大小为 4KB，采用 4 路组相联映射、随机替换算法和回写策略，每个 Cache 行包括 128 个字节和对应一个标签字段 tag、有效位 valid 和脏位(dirty)。指令 Cache 只支持 4 个字节整体读写，数据 Cache 支持 1、2、4 个字节的读写。

提示：32 位的地址可以分割成如下 5 个部分：

word_addr[1:0] ：字节地址，即指定字节是字（word）中的第几个字节，2 位。

offset_addr：块内地址，其长度由每个行内包括的 word 数来决定。

index_addr：分组索引，其长度由 Cache 中分组数来决定。

tag_addr: 标签块，32 位存储地址在 Cache 中有效对应地址段 TAG。当发生读写请求时，cache 应该把 32 位地址中的 tag_addr 取出，与 cache 中的 TAG 比较，如果相等则命中。如果不等则缺失。

unused_addr: 32 位地址中的高位，恒为 0。

首先创建 Cache 管理模块的接口参考定义如下：

```
module cache #(
    parameter OFFSET_WIDTH = 3,          // 块内（偏移）地址宽度，需根据实验要求进行修改
    parameter INDEX_WIDTH    = 6,        // Cache 组数地址宽度，需根据实验要求进行修改
    parameter ADDR_WIDTH     = 32,               // 存储器地址宽度
    parameter DATA_WIDTH     = 32,               // 数据位宽度
    // Local parameters
    parameter BLOCK_SIZE     = 1 << OFFSET_WIDTH,                  // 块内地址
    parameter CACHE_DEPTH    = 1 << INDEX_WIDTH,                   // Cache 组数
    parameter BLOCK_WIDTH    = DATA_WIDTH * BLOCK_SIZE,        // Cache 块的存储
    parameter TAG_WIDTH      = ADDR_WIDTH - OFFSET_WIDTH - INDEX_WIDTH     // 标签 Tag 的宽度
) (
    // From CPU
    input                               clk,                //时钟信号
    input                               rst,                //复位信号
    input                               read_in,            //读取 Cache 信号
    input                               write_in,           //写入 Cache 信号
    input [3 : 0]                       byte_w_en_in,       //读写字节数写使能信号，单热点编码
    input [ADDR_WIDTH - 1 : 0]      addr,               //指令 Cache 地址
    input [DATA_WIDTH - 1 : 0]      data_from_reg,      //写入数据
    // To CPU
    output                              mem_stall,
    output [DATA_WIDTH - 1 : 0]     data_out,           //读取 Cache 的数据
     // From RAM
    input                               ram_ready,          // 存储器就绪信号
    input [BLOCK_WIDTH - 1 : 0]     block_from_ram,     //   当缺失时载入的内存块
    // To RAM
    output                              ram_en_out,         // 存储器输出使能信号
    output                              ram_write_out,      // 存储器写使能输出信号
    output [ADDR_WIDTH - 1    : 0] ram_addr_out,
```

```
    output [BLOCK_WIDTH - 1 : 0] data_wb,                    // 回写 Cache 块数据

    // debug
    output reg   [2:0] status,      // Cache 状态，表明 cache 是否*已经*发生缺失以及缺失类型，长度可修改
    output reg   [2:0] counter      // 块内偏移指针/迭代器，用于载入块时逐字写入，可修改。
);
 // Add your code here
endmodule
```

可通过顶层模块文件 cache_top，设计输入输出接口，实现实验开发板验证。输入开关来设置 index、tag 和部分输入数据，rst 信号连接到 BTNC 按钮，通过数码管和 Led 指示灯来输出读取的 cache 数据、当前 index 和 cache 状态位如 valid、dirty 和 hit 等，具体对应关系可自行设置。

```
module cache_top
(
    input           rst,      //复位
    input           clk,
    //------gpio-------
    output      [15:0] led,
    input       [15 :0] switch,
    output reg [7 :0] AN,
    output reg [6 :0] SEG
);
// Add your code here
endmodule
```

仿真测试文件要求针对每一个 index 组，生成 4 组随机的 tag 和 data 对。然后生成写请求将这 4 组数据写进到 cache 组中，再进行读取这 4 组数据。如期间没有发生错误，则对下一个 index 组重新随机生成 tag 和 data 对，进行同样的读取测试，直到完成所有 index 组的验证测试。对于写 cache 请求，要求写请求发出后会出现 Cache 缺失状态，cache 模块会发生回写状态和载入状态，期间对替换的数据进行读写比较，如果发生错误，则记录组号、tag 标签和数据，并显示错误数据。写操作完成后，再进行读取操作，对写入的数据和读取的数据进行比对，如果生错误，则记录组号、tag 标签和数据，并显示错误数据。如果所有的 index 组的 cache 初始化和读写操作都没有发生错误，则显示“Test Pass！”。

请根据上述描述，按照下列步骤完成实验。

1、 使用 Vivado 创建一个新工程。

2、 点击添加设计源码文件，加入 lab7.zip 里的 cache.v、cache_top 等文件。

3、 点击添加测试文件，加入 lab7.zip 里的 cache_tb .v 文件。

4、 点击添加约束文件，加入 lab7.zip 里的 cache.xdc 文件。

5、 根据实验要求，完成源码文件的设计。

6、 对工程进行仿真测试。

7、 仿真通过后，进行综合、实现并生成比特流文件。

8、 生成位流文件后，加载到实验开发板，进行调试验证，并记录验证过程。

五、思考题

1、分别采用分布式 RAM 和块 RAM 实现存储器，通过仿真程序分析异步读和同步读的时序状态。

2、分析 Cache 大小对命中率的影响。

3、当指令 Cache 和数据 Cache 独立实现时，如何设计顶层 Cache 的有限状态机。