第 13 章　带有 Cache 及 TLB 和 FPU 的 CPU 设计

本章描述一个完整的流水线 CPU 的设计，包括整数部件、浮点部件、分开的指令 Cache 和数据 Cache 以及分开的指令 TLB 和数据 TLB。浮点部件能完成加减乘除和开方运算，Cache 完全由硬件控制，而 TLB 的维护需要软件介入。

TLB 不命中时产生异常信号，因此本章给出的 CPU 也包含了异常处理。TLB 维护指令只实现 tlbwi 和 tlbwr 两条指令。另外，我们在 Index 寄存器的第 30 位增设了一位 D (Data TLB)。当 D = 1 时，tlbwi 和 tlbwr 写数据 TLB；为 0 时写指令 TLB。本章给出 CPU 的 Verilog HDL 设计代码以及仿真波形。

13.1　Cache 和 TLB 的总体结构

我们已经在第 12 章介绍了 Cache 和 TLB 的工作原理及它们的 Verilog HDL 代码。本节的重点是如何使用它们，与整数部件和浮点部件以及主存有机地结合在一起，为 CPU 提供有效的存储器层次的管理。图 13.1 是简化的 TLB 与 Cache 之间的连接概念图。

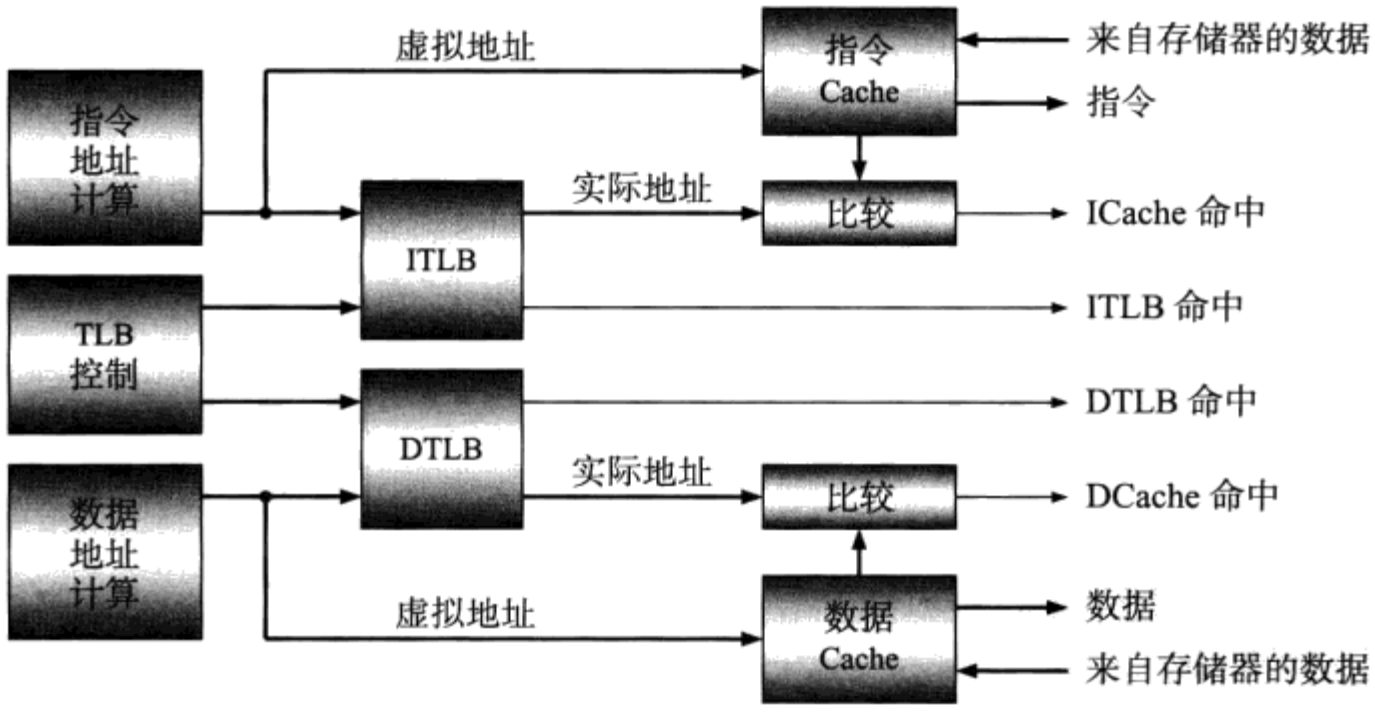

图 13.1　TLB 与 Cache 的连接

TLB 的作用是把 CPU 产生的虚拟地址快速地转换成存储器的实际地址。我们使用两个 TLB：指令 TLB (简称 ITLB) 和数据 TLB (简称 DTLB)，分别用于指令地址和数据地址的转换。为了设计简单起见，我们使用固定大小的 4KB 页面，TLB 的映像方式为全相联映像。

Cache 的作用是为 CPU 快速地提供指令和数据。我们也使用两个 Cache：指令

Cache 和数据 Cache。Cache 的映像方式为最简单的直接映像，标志位使用经过 TLB 转换过的实际地址。

13.2 与 Cache 有关的电路设计

13.2.1 指令 Cache 的 Verilog HDL 代码

我们已经在第 12 章介绍了数据 Cache 的设计方法并给出了它的 Verilog HDL 代码。指令 Cache 比数据 Cache 简单，因为 CPU 并不往指令 Cache 写任何东西。以下是指令 Cache 的 Verilog HDL 代码。请与数据 Cache 的代码进行比较。

```
module i_cache #(parameter A_WIDTH = 32, parameter C_INDEX = 6)
  (p_a,p_din, p_strobe,p_ready,cache_miss,clk,clrn,
   m_a, m_dout,m_strobe,m_ready);
    input [A_WIDTH-1:0]  p_a;
    output [31:0]        p_din;
    input                p_strobe;
    output               p_ready;
    output               cache_miss;
    input                clk, clrn;
    output [A_WIDTH-1:0] m_a;
    input [31:0]         m_dout;
    output               m_strobe;
    input                m_ready;
localparam T_WIDTH = A_WIDTH - C_INDEX - 2; // 1 block = 1 word
reg                      d_valid [0:(1<<C_INDEX)-1];
reg [T_WIDTH-1:0]        d_tags  [0:(1<<C_INDEX)-1];
reg [31:0]               d_data  [0:(1<<C_INDEX)-1];
wire [C_INDEX-1:0]       index = p_a[C_INDEX+1:2];
wire [T_WIDTH-1:0]       tag = p_a[A_WIDTH-1:C_INDEX+2];
// write to cache
always @ (posedge clk or negedge clrn)
    if (clrn == 0) begin
        integer i;
        for (i = 0; i < (1 << C_INDEX); i = i + 1)
            d_valid[i] <= 1'b0;
    end else if (c_write)
        d_valid[index] <= 1'b1;
always @ (posedge clk)
    if (c_write) begin
        d_tags[index] <= tag;
        d_data[index] <= c_din;
    end
// read from cache
wire                 valid = d_valid[index];
```

```
    wire [T_WIDTH-1:0] tagout = d_tags[index];
    wire [31:0]        c_dout = d_data[index];
    // cache control
    wire   cache_hit  = valid & (tagout == tag); // hit
    assign cache_miss = ~cache_hit;
    assign m_a        = p_a;
    assign m_strobe   = p_strobe & cache_miss ; // read on miss
    assign p_ready    = cache_hit | cache_miss & m_ready;
    wire   c_write    = cache_miss & m_ready;
    wire   sel_out    = cache_hit;
    wire [31:0] c_din = m_dout;
    assign      p_din = sel_out? c_dout : m_dout;
endmodule
```

13.2.2 数据 Cache 和指令 Cache 与外部存储器的接口

因为有分开的指令 Cache 和数据 Cache，CPU 从指令 Cache 取指令的同时，也可以访问数据 Cache。但与存储器的接口只有一套，见图 13.2。

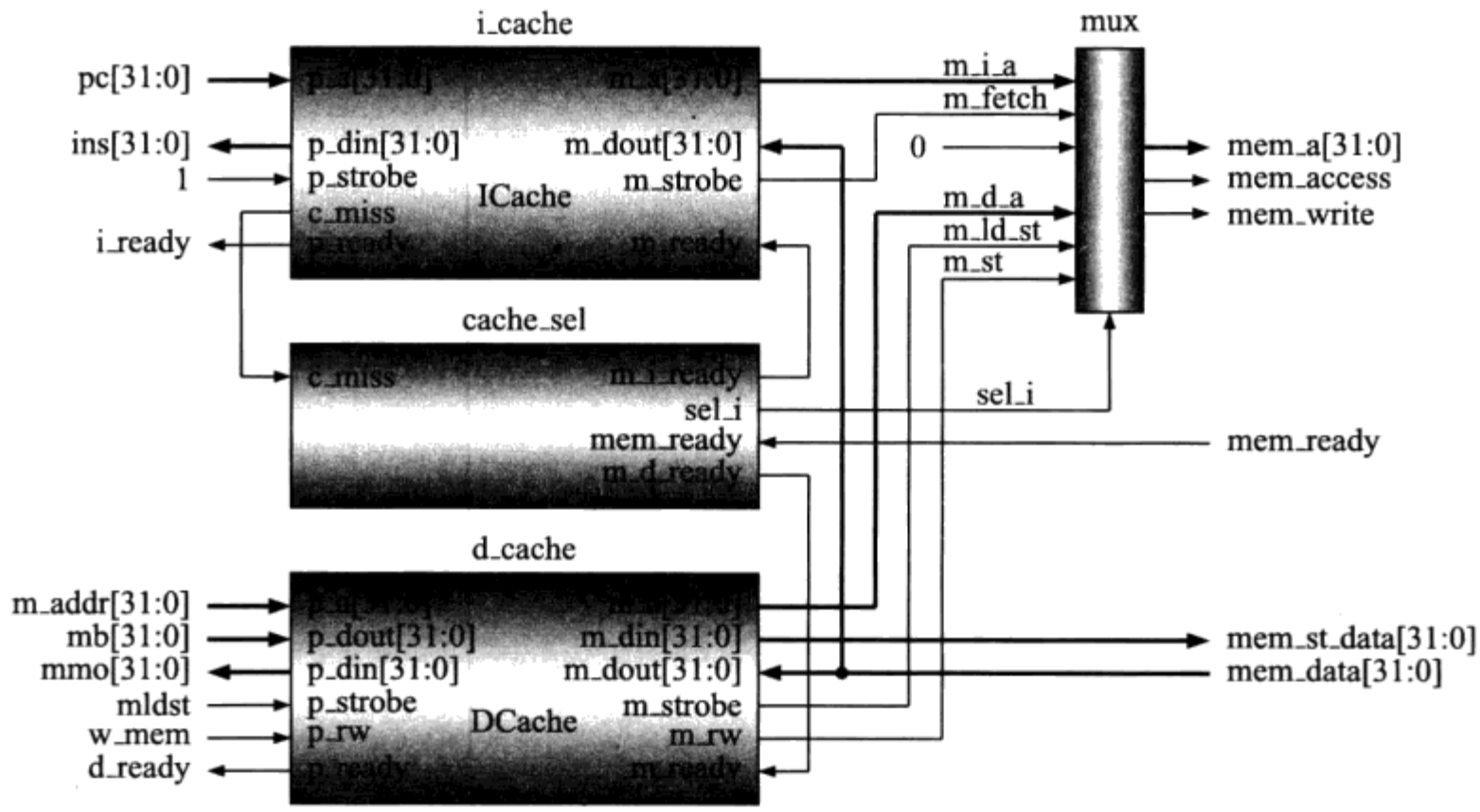

图 13.2 数据 Cache、指令 Cache 和存储器接口

图中左边的信号连接到 CPU 中的 IU 部分，右边的信号连接到主存。我们规定指令 Cache 的优先级比数据 Cache 高，即当两个 Cache 都不命中时，首先从存储器中取出指令。图中的 mux 和 cache_sel (demux) 电路的代码如下。

```
    // mux, i_cache has higher priority than d_cache
    sel_i       = i_cache_miss;
```

```
mem_a      = sel_i ? m_i_a    : m_d_a;
mem_access = sel_i ? m_fetch  : m_ld_st;
mem_write  = sel_i ? 1'b0     : m_st;
// demux
m_i_ready  = mem_ready &  sel_i;
m_d_ready  = mem_ready & ~sel_i;
```

13.2.3 Cache 不命中时流水线暂停的电路

Cache 不命中时，要访问存储器。在访问存储器期间，我们采用最简单的处理方法：暂停流水线。以下是 Cache 没有暂停流水线的条件：

```
no_cache_stall = ~(~i_ready | mldst & ~d_ready);
```

式中的 mldst 是流水线 MEM 级的信号，它表示当前在 MEM 级的指令是存储器访问指令。原有的流水线暂停信号 wpcir 还是控制 PC 和 IR，而 no_cache_stall 控制包括 PC 和 IR 在内的所有的流水线寄存器。

13.3 与 TLB 有关的电路设计

我们已经在第 12 章介绍了 TLB 模块本身的设计方法并给出了电路图。本节介绍如何使用它来实现对指令虚拟地址和数据虚拟地址的转换。与 Cache 的处理方法不同，当 ITLB 或 DTLB 不命中时，产生相应的异常信号，CPU 执行异常处理程序来填充 TLB。ITLB 和 DTLB 具有相同的结构，都使用 tlb_8_entry 模块。我们首先给出它的 Verilog HDL 代码，它等同于第 12 章的电路图 (见图 12.20)。

```
module tlb_8_entry (pte_in,tlbwi,tlbwr,index,vpn,memclk,clk,clrn,
                    random,pte_out,hit,vpn_index,vpn_found);
    input  [23:0] pte_in;
    input         tlbwi, tlbwr;
    input   [2:0] index;
    input  [19:0] vpn;
    input         memclk, clk, clrn;
    output  [2:0] random;
    output [23:0] pte_out; // v d c c ppn
    output        hit;
    output  [2:0] vpn_index;
    output        vpn_found;
    wire    [2:0] w_idx, ram_idx;
    wire          tlbw = tlbwi | tlbwr;
    rand3   rdm (clk,clrn,random);
    mux2x3  w_address (index,random,tlbwr,w_idx);
    mux2x3  ram_address (vpn_index,w_idx,tlbw,ram_idx);
    ram8x24 pte (ram_idx,pte_in,memclk,memclk,tlbw,pte_out);
```

```
    cam8x20 valid_tag (memclk,vpn,w_idx,tlbw,vpn_index,vpn_found);
    assign  hit = pte_out[23] & vpn_found;
endmodule
```

13.3.1 指令 TLB (ITLB) 和数据 TLB (DTLB)

图 13.3 给出了两个 TLB 及周边电路的模块图。右边的两个输出信号 ipte_out 和 dpte_out 分别是指令和数据实际存储器地址的页号。寄存器 Index 主要存放 TLB 项的号码，另外第 30 位指出是写 ITLB 还是写 DTLB (作者定义的)。EntryLo (只有一个) 主要存放存储器实际地址的页号，它的内容是从存储器页表读来的，当 TLB 不命中时被写入 TLB。

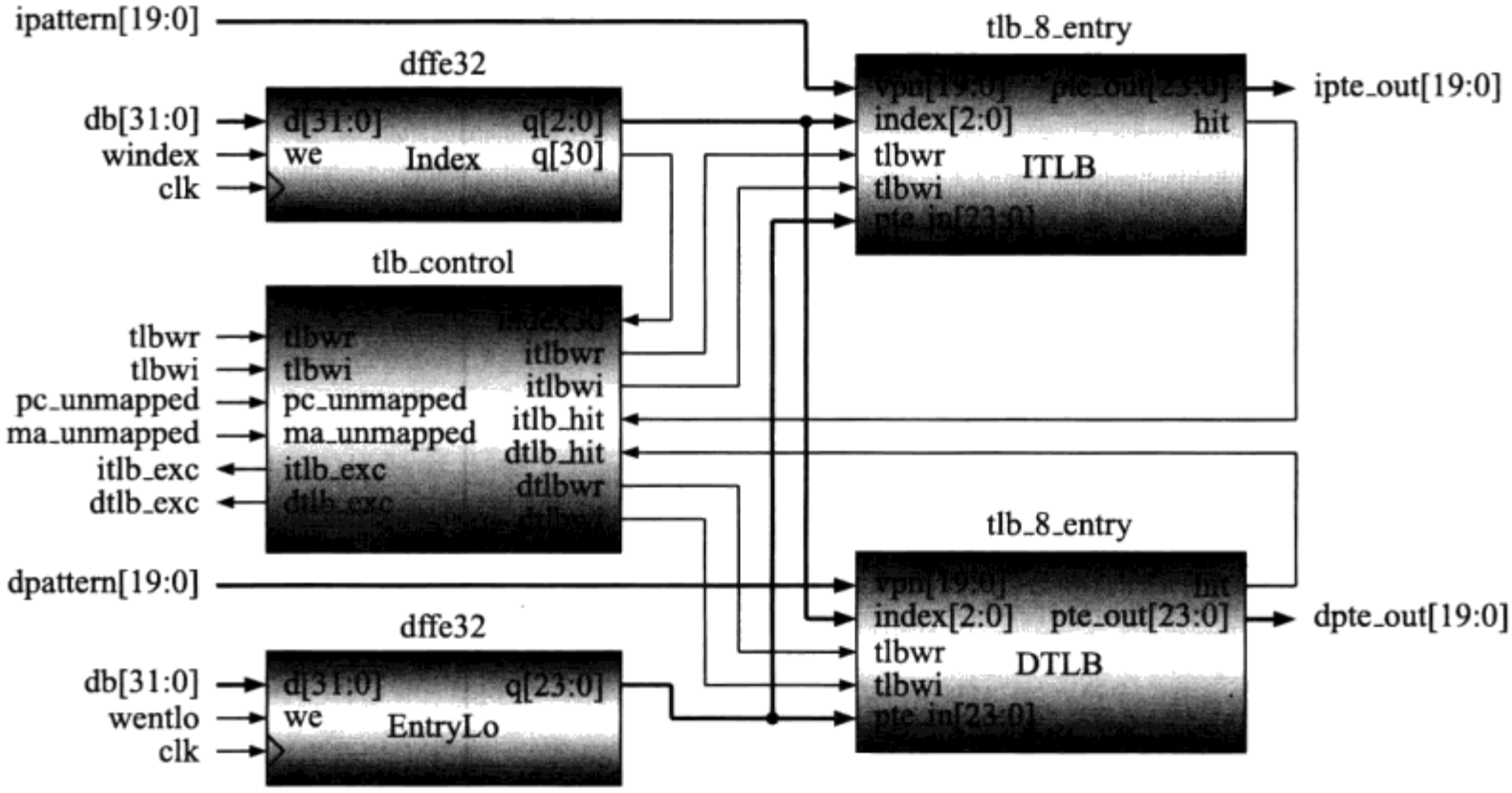

图 13.3 ITLB 和 DTLB

输入信号 ipattern 和 dpattern 各自都有两个来源：通常情况下来自于 CPU 虚拟地址的页号，用于匹配 TLB 来做地址转换；TLB 不命中时来自于 ENTRY_HI 寄存器，它的内容要被写入 TLB。ENTRY_HI 的内容实际上也是存储器虚拟地址的页号。两个信号的产生方法如下，其中 v_pc 是指令的虚拟存储器地址、malu 是数据的虚拟存储器地址。

```
ipattern = (itlbwi | itlbwr) ? enthi[19:0] : v_pc[31:12];
dpattern = (dtlbwi | dtlbwr) ? enthi[19:0] : malu[31:12];
```

13.3.2 TLB 不命中时异常信号的产生

并不是所有的存储器访问都要用 TLB 做地址转换。见图 12.24，当虚拟地址的最高两位为 10 时，只要把最高位的 1 改为 0，就得到实际地址。因此，当虚拟地址的最高两位为 10 时，要封锁 TLB 不命中时产生的异常信号。这部分的代码如下。

```
// mapped or unmapped
pc_unmapped = v_pc[31] & ~v_pc[30]; // 10x; v_pc: va of inst
ma_unmapped = malu[31] & ~malu[30]; // 10x; malu: va of data
// real addresses
pc      = pc_unmapped ? {1'b0,v_pc[30:0]} :
                        {ipte_out[19:0],v_pc[11:0]};
m_addr = ma_unmapped ? {1'b0,malu[30:0]} :
                        {dpte_out[19:0],malu[11:0]};
// exceptions
itlb_exc = ~itlb_hit & ~pc_unmapped;
dtlb_exc = ~dtlb_hit & ~ma_unmapped & mldst;
```

其中，itlb_exc 是 ITLB 不命中的异常信号，dtlb_exc 是 DTLB 不命中的异常信号。我们规定 itlb_exc 的优先级比 dtlb_exc 高，即当二者同时为 1 时，首先处理 dtlb_exc。

13.3.3 与 TLB 不命中异常有关的寄存器

对 TLB 不命中异常的处理要用到一些特殊的寄存器，见图 13.4。注意有些寄存器的定义与 MIPS 不同。这些寄存器的意义如下所述。

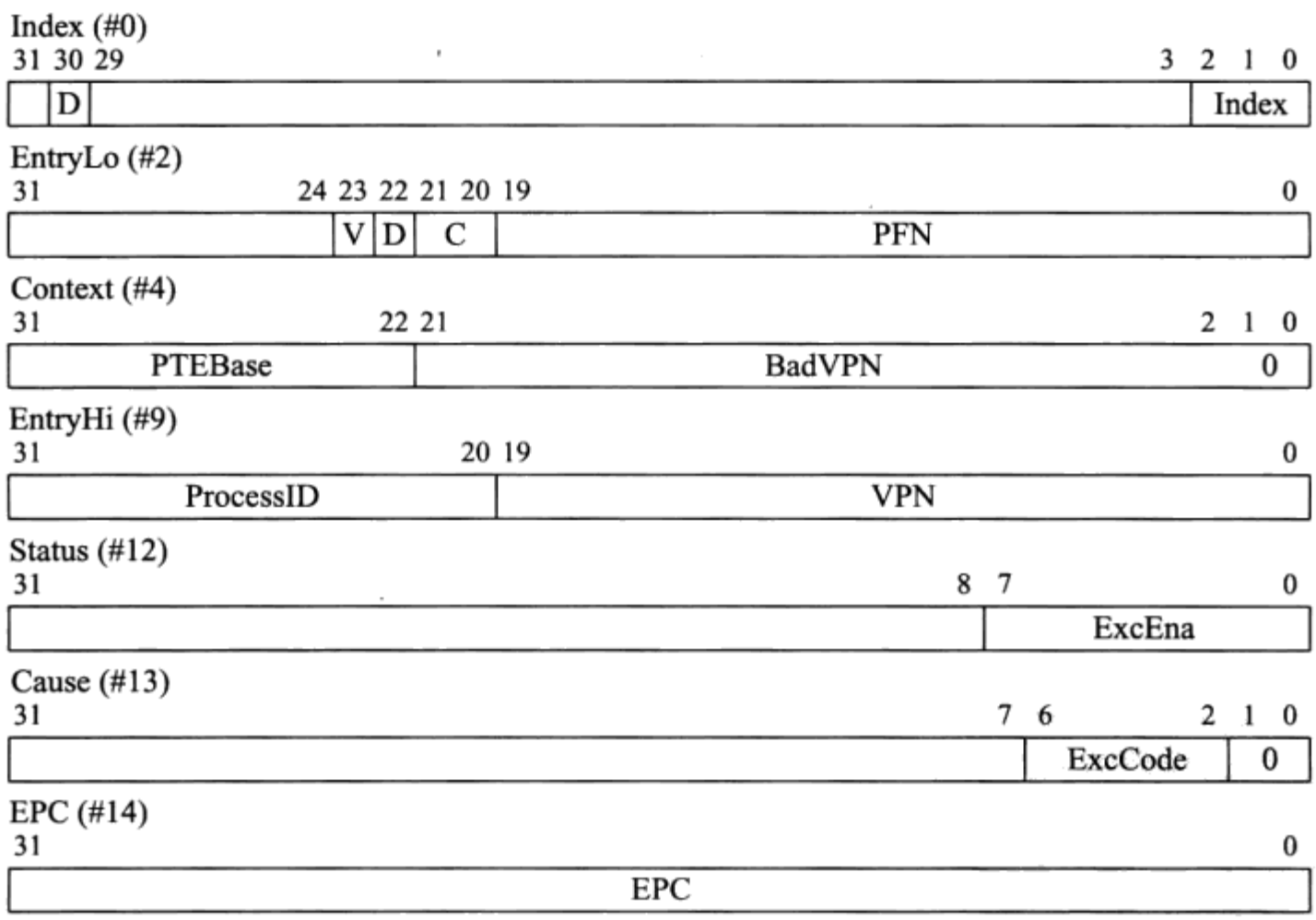

图 13.4 与 TLB 不命中异常有关的寄存器

1) CPU 通过执行 tlbwi 指令，可以修改某个 TLB 项。寄存器 Index 的最低 3 位用来指定这个 TLB 项 (在我们的设计中，TLB 总共有 8 项)。第 30 位 D 用来指定是写 ITLB 还是写 DTLB。

2) 寄存器 EntryLo 只有一个，其中的 PFN 是存储器实际地址的页号；V 是有效位；D 和 C 没有使用。
3) 寄存器 Context 的内容是一个页表项在存储器中的实际地址。当 TLB 不命中时，CPU 使用这个地址从页表中读出一个页表项，将其写入 TLB。Context 中的高位部分 PTEBase 由 CPU 执行 mtc0 指令写入；低位部分 BadVPN 由硬件自动写入，它是引起 TLB 不命中异常的虚拟地址的页号。
4) 寄存器 EntryHi 中的 VPN 是虚拟地址的页号，由 CPU 设置。CPU 执行 tlbwi 或 tlbwr 指令时，把它写入 TLB。ProcessID 没有使用。
5) 寄存器 Status 中的 ExcEna 是 8 位异常允许位，其中第 4 位和第 5 位分别对应 itlb_exc 和 dtlb_exc。为 0 时，屏蔽相应的异常。当 CPU 响应异常时，把 Status 寄存器的内容左移 8 位以屏蔽进一步的异常 (作者的做法)。
6) 寄存器 Cause 中的 ExcCode 指出当前发生的是哪种异常：itlb_exc 和 dtlb_exc 的 ExcCode 分别定义为 4 和 5。
7) 寄存器 EPC 用来保存异常返回地址，由硬件自动写入。

以下描述 ITLB 和 DTLB 不命中产生异常时硬件需要完成的动作。图 13.5 所示的是在通常情况下出现 itlb_exc 时的流水线时序和保存返回地址的电路。

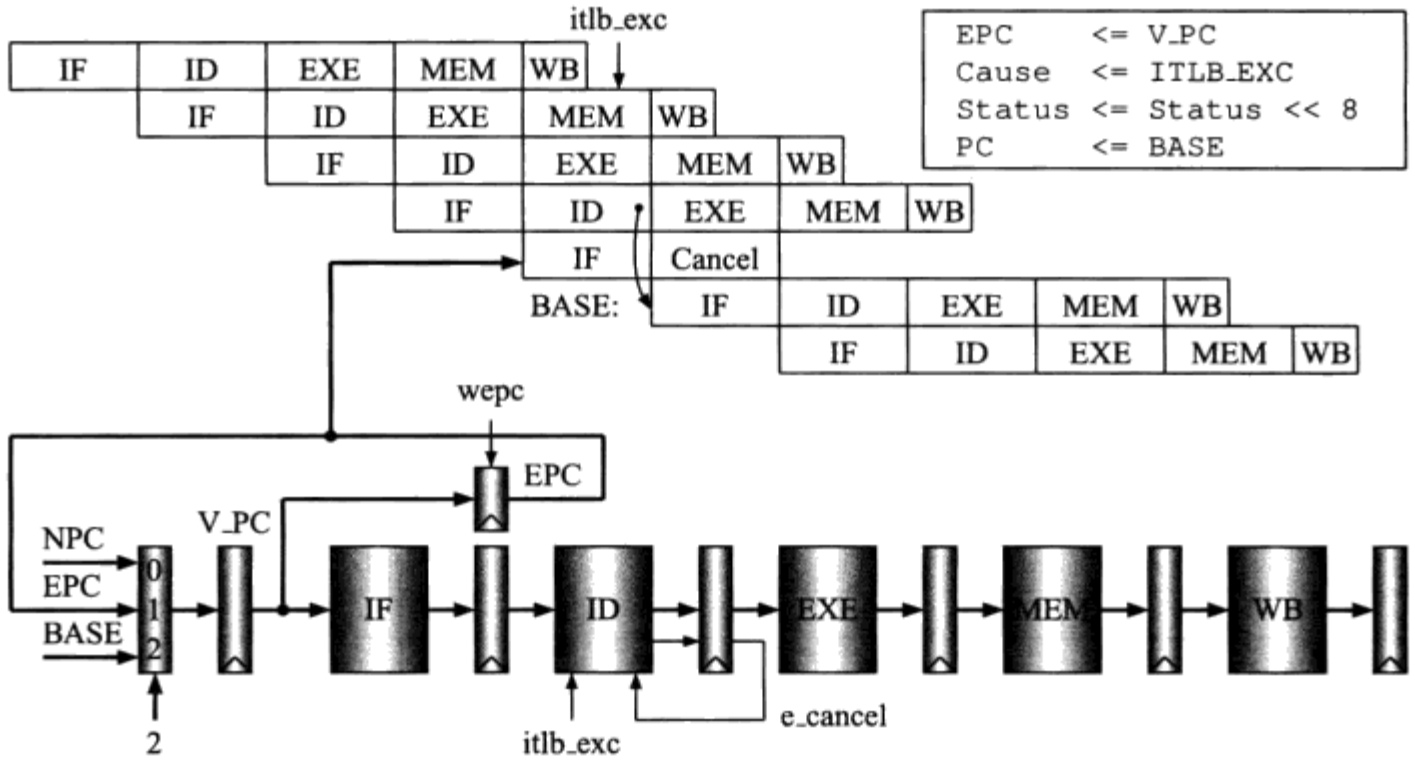

图 13.5 一般情况下的 ITLB_EXC

ITLB 不命中异常 itlb_exc 出现在取指令的 IF 级。此时的 ID 级要对该异常进行处理。处理动作包括：(1) 把引起异常的指令的虚拟地址保存到 EPC；(2) 把 4 写入 Cause 寄存器的 ExcCode 域；(3) Status 寄存器左移 8 位；(4) 把异常处理程序的入口地址写入 V_PC；(5) 产生废弃指令的信号 cancel。以上五个动作同时完成。

因为已经把异常处理程序的入口地址写入了 V_PC，所以 CPU 将执行程序以实现对 ITLB 的修改。具体的做法稍后讨论。我们还是接着讲 ITLB 和 DTLB 异常。

图 13.6 所示的是在取延迟槽指令的情况下出现 itlb_exc 时的流水线时序和保存返回地址到 EPC 的电路。此时处在 ID 级的指令是转移指令，本来好好地要转移到目标地址，但由于出现了异常，必须要转移到异常处理程序的入口，因此保存到 EPC 的返回地址必须是转移指令的地址，即返回后重新执行转移指令。注意此时的 V_PC 指向的是延迟槽指令。为了能够得到转移指令的地址，我们增加了一个流水线寄存器 PCD。此时 PCD 中的内容就是转移指令的地址，把它保存到 EPC 中就行了。

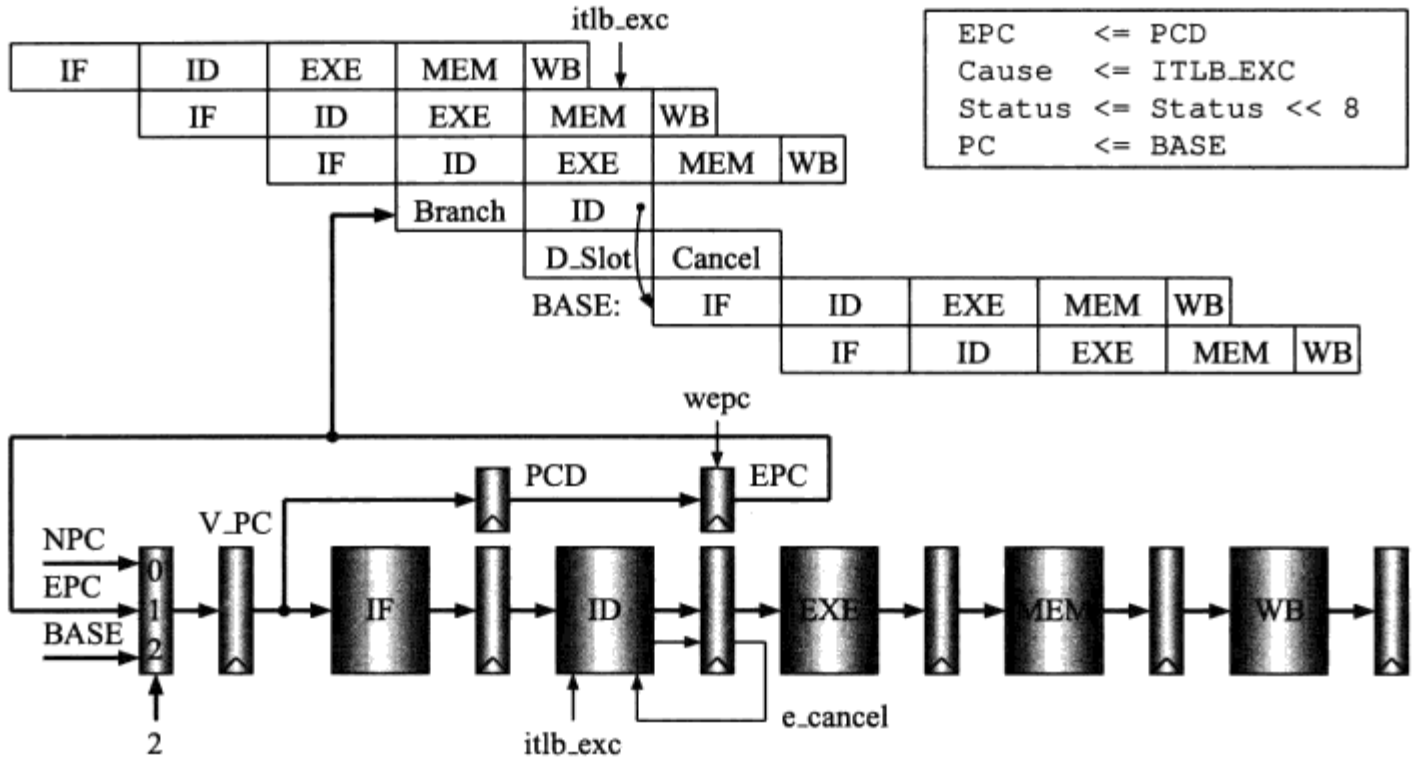

图 13.6 处在延迟槽的指令引起的 ITLB_EXC

DTLB 出现不命中异常就有一点点小麻烦了，它出现在流水线的 MEM 级，与 ITLB 的不命中异常出现在 IF 级相比，整整晚了 3 级。倒不是说保存返回地址有多麻烦，真正麻烦的是要废弃好多指令。

图 13.7 所示的是在通常情况下出现 dtlb_exc 时的流水线时序和保存返回地址的电路。要废弃的指令总共有 4 条，它们分别处在 IF、ID、EXE 和 MEM 级。废弃信号在 ID 级产生，“自废”没问题，但除此之外还要忙前忙后。废弃下一条 IF 级的指令也没问题，因为已经在 ITLB 不命中时演练过了。废弃 EXE 级指令的办法是把 EXE 级的控制信号在写入流水线寄存器之前封锁掉 (指令在 EXE 级不改变 CPU 状态)。废弃 MEM 级的指令时，不仅要把写入流水线寄存器的控制信号封锁掉，也要把在 MEM 级使用的写存储器信号封锁掉。写入 EPC 的返回地址在 PCM 中，即引起 DTLB 不命中异常的指令的地址。

图 13.8 所示的是处在延迟槽的指令引起 dtlb_exc 时的流水线时序和保存返回地址的电路。与 ITLB 的情况类似，返回地址应该是转移指令的地址，它在 PCW 寄存器中。废弃指令的动作与上述一般情况下的废弃指令的动作相同。

我们把以上 4 种情况加以总结，列在表 13.1 中。由此我们得到选择信号 sepc 的逻辑表达式如下。它选择不同的返回地址，选中的地址被写入 EPC 中。

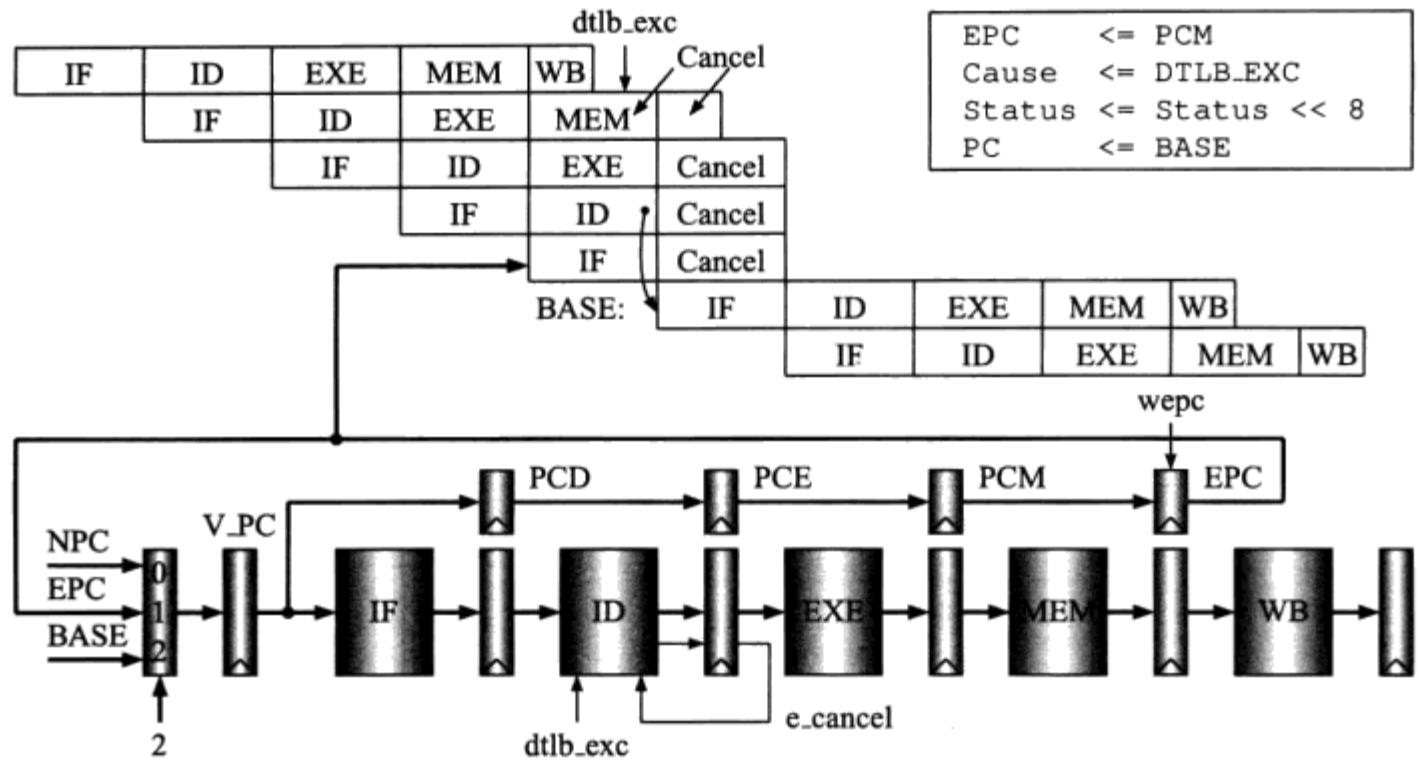

图 13.7 一般情况下的 DTLB_EXC

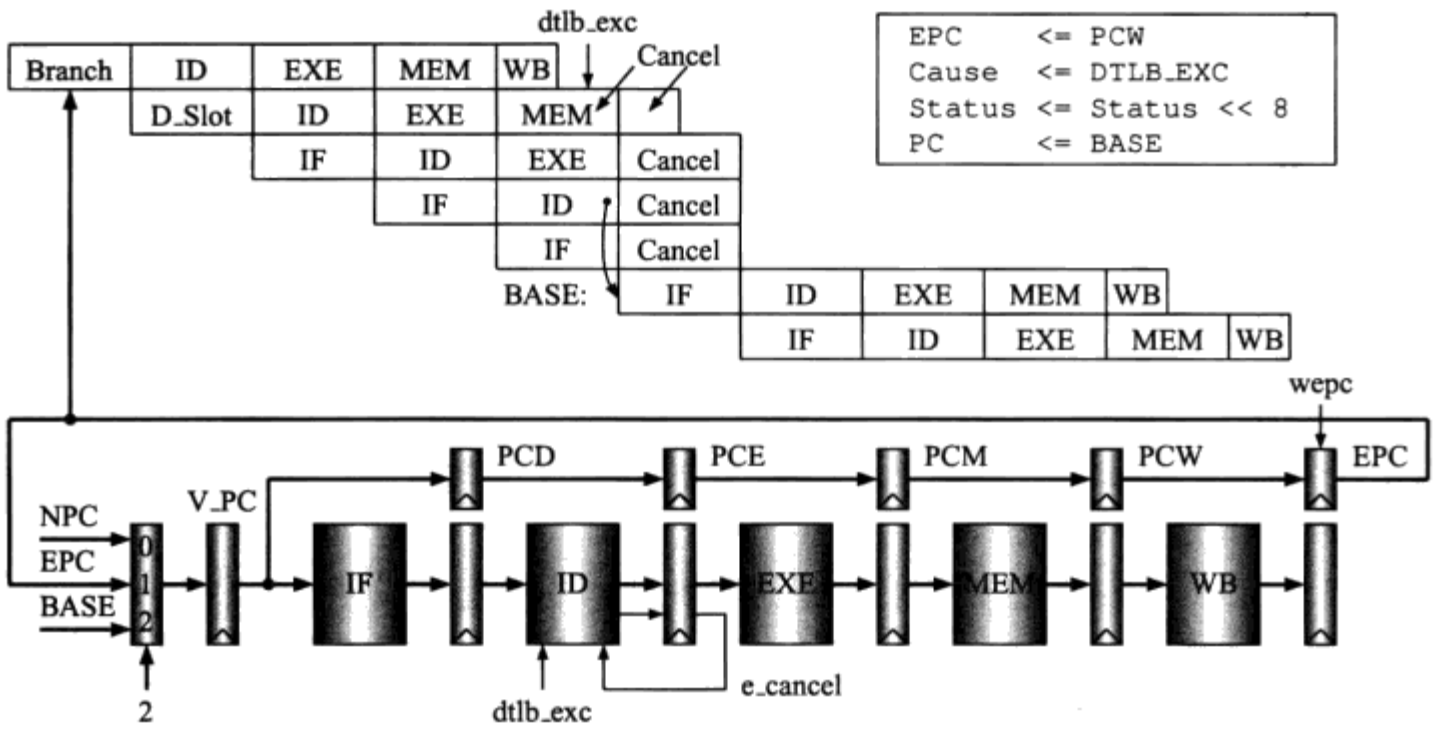

图 13.8 处在延迟槽的指令引起的 DTLB_EXC

表 13.1 为 EPC 选择返回地址

itlb_exc	dtlb_exc	isbr	wisbr	EPC	sepc[1:0]
1	x	0	x	V_PC	0 0
1	x	1	x	PCD	0 1
0	1	x	0	PCM	1 0
0	1	x	1	PCW	1 1

```
sepc[1] = ~itlb_exc & dtlb_exc;
sepc[0] =  itlb_exc & isbr | ~itlb_exc & dtlb_exc & wisbr;
```

13.3.4 对 TLB 不命中异常的处理

当 TLB 不命中异常发生且异常没被屏蔽时，CPU 跳转到固定的地址 (BASE) 去执行异常处理程序。在我们的设计中，这个地址是 0x80000008。当 CPU 处理完异常，执行 eret 指令时，EPC 的内容要写入程序计数器中，以便实现从异常处理程序返回。因此我们在程序计数器的输入端加了一个多路器。多路器的输入及输入选择信号如下。其中 exc 是异常信号，i_eret 为 1 表示当前指令是 eret。

```
// selpc[1:0]:  00: npc;  01: epc;  10: EXC_BASE
selpc[1] = exc;
selpc[0] = i_eret;
```

当异常发生时，由 CPU 硬件将异常源写入 Cause 寄存器的 ExcCode 域。如前所述，我们把 itlb_exc 和 dtlb_exc 的 ExcCode 分别定义为 4 和 5。当 CPU 进入异常处理程序后，可根据这个 ExcCode 转入相应的程序去处理。我们的做法是设置一个跳转表 (j_table)，把 ExcCode 作为地址偏移量从跳转表得到入口地址，然后跳转到这个地址。以下的汇编程序演示这个过程 (程序中的 EXC_BASE = BASE)。

```
EXC_BASE:                                  # exception/interrupt entry
0x80000008: mfc0 r26, C0_CAUSE     # read cp0 Cause reg
0x8000000c: andi r26, r26, 0x1c    # get ExcCode, 3 bits here
0x80000010: lui  r27, 0x8000       # j_table address high
0x80000014: or   r27, r27, r26     # j_table address low
0x80000018: lw   r27, j_table(r27) # get address from table
0x8000001c: nop                    #
0x80000020: jr   r27               # jump to that address
0x80000024: nop                    #

.data
j_table:            # address table for exception and interrupt
0x80000040: 0x80000030 # 0. int_entry, addr. for interrupt
0x80000044: 0x8000003c # 1. sys_entry, addr. for Syscall
0x80000048: 0x80000054 # 2. uni_entry, addr. for Unimpl. inst.
0x8000004c: 0x80000068 # 3. ovf_entry, addr. for Overflow
0x80000050: 0x800000c0 # 4. itlb_entry, addr. for itlb miss
0x80000054: 0x80000140 # 5. dtlb_entry, addr. for dtlb miss
0x80000058: 0x80000000 # 6.
0x8000005c: 0x80000000 # 7.
```

比如当前的异常是 dtlb_exc，则跳转到地址 0x80000140。处理 dtlb_exc 异常的主要工作是为没命中的虚拟地址在 DTLB 中准备一个 TLB 项，填上实际地址的页号。

该页号从页表中得到。修改 TLB 的指令有两条： tlbwi 和 tlbwr。我们的演示程序使用了 tlbwi，该方法需要有一个 index 号码，用来指出写哪个 TLB 项。为此我们准备了一个计数器，每次写 TLB 时，都把它加 1 (用软件实现先进先出替换策略)。由于我们的 TLB 只有 8 项，因此只使用计数器的低 3 位。这 3 位计数器值连同 DTLB 标志 D 一起写入 Index 寄存器。寄存器 Context 中的内容实际上就是页表的存储器地址，我们可以从该地址得到实际地址的页号，把它写入 EntryLo 寄存器。我们还要设置 EntryHi 寄存器，它的内容应该是引起 DTLB 不命中的虚拟地址的页号。以上寄存器都设置好之后，执行 tlbwi 指令修改 TLB，然后返回。这部分程序如下。

```
0x80000140: lui  r27, 0x8000       # 0x800001fc: counter
0x80000144: lw   r26, 0x1fc(r27)   # load dtlb index counter
0x80000148: addi r26, r26, 1       # index + 1
0x8000014c: andi r26, r26, 7       # 3-bit index
0x80000150: sw   r26, 0x1fc(r27)   # store index
0x80000154: lui  r27, 0x4000       # dtlb tag D (bit 30)
0x80000158: or   r26, r27, r26     # dtlb tag and index
0x8000015c: mtc0 r26, C0_INDEX     # move to c0 index
0x80000160: mfc0 r27, C0_CONTEXT   # move from c0 context
0x80000164: lw   r26, 0x0(r27)     # get pte
0x80000168: mtc0 r26, C0_ENTRY_LO # move to c0 entry_lo
0x8000016c: sll  r26, r27, 10      # get bad vpn
0x80000170: srl  r26, r26, 12      # for c0 entry_hi
0x80000174: mtc0 r26, C0_ENTRY_HI # move to entry_hi
0x80000178: tlbwi                  # update dtlb
0x8000017c: eret                   # return from exception
0x80000180: nop                    #
```

13.4 带有 Cache 及 TLB 的 CPU 设计

13.4.1 带有 Cache 及 TLB 的 CPU 总体结构

图 13.9 给出的是带有 Cache 及 TLB 的 CPU 总体结构的示意图。图中的 IU/FPU 模块的基本结构与第 10 章描述的 CPU 相同，但增加了对 TLB 不命中异常的处理电路，包括跳转到异常处理程序及从中返回的电路、与 TLB 有关的寄存器和对相关指令的译码电路等。

与第 10 章的 CPU 不同的是当 CPU 复位时，程序计数器的初始值为 0x80000000，而不是 0x00000000。另外，为了测试 TLB 不命中异常，我们在测试程序中扩大了数据存储器的使用范围。为此，我们使用了不连续的 4 段物理存储器，它们分别存放：(1) 系统复位时的初始化程序和异常处理程序；(2) 用于虚拟存储器管理的页表；(3) 用于测试 IU/FPU 的用户程序；(4) 用户程序所使用的数据。以下给出 CPU 的 Verilog HDL 源代码。

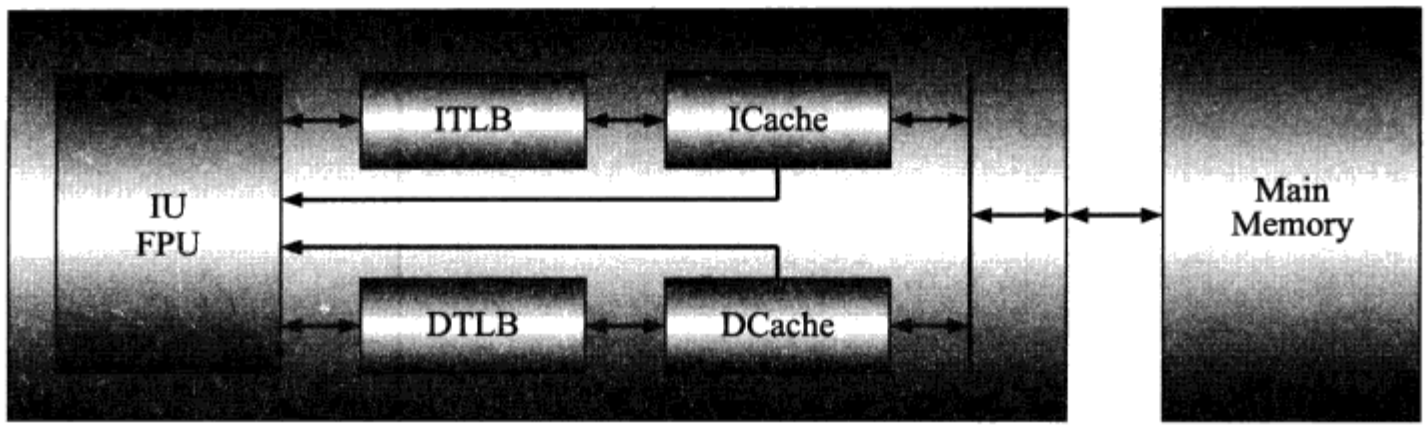

图 13.9 带有 Cache 及 TLB 的 CPU 模块图

13.4.2 带有 Cache 及 TLB 的 CPU 的 Verilog HDL 代码

以下是最顶层模块 cpu_cache_tlb_memory，包括 CPU 和存储器。

```
module cpu_cache_tlb_memory (
    clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,ww,
    stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,
    mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    input  clock,memclock,resetn;
    output [31:0] v_pc,pc,inst,ealu,malu,walu;
    output [31:0] wd;
    output  [4:0] wn;
    output ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall;
    output [31:0] mem_a;
    output [31:0] mem_data;
    output [31:0] mem_st_data;
    output        mem_access;
    output        mem_write;
    output        mem_ready;
    // cpu
    cpu_cache_tlb cpucachetlb (
       clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,
       ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,mem_a,
       mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    // main memory
    physical_memory mem (mem_a,mem_data,mem_st_data,mem_access,
                         mem_write,mem_ready,clock,memclock,resetn);
endmodule
```

以下是 CPU 模块 cpu_cache_tlb，包括 IU、Cache、TLB 和 FPU。

```
module cpu_cache_tlb
   (clock,memclock,resetn,v_pc,pc,inst,ealu,malu,walu,wn,wd,ww,
    stall_lw,stall_fp,stall_lwc1,stall_swc1,stall,
    mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
```

```
input  clock,memclock,resetn;
output [31:0] v_pc,pc,inst,ealu,malu,walu;
output [31:0] wd;
wire   [31:0] e3d;
output  [4:0] wn;
wire    [4:0] e1n,e2n,e3n;
output ww,stall_lw,stall_fp,stall_lwc1,stall_swc1,stall;
wire   e; // for multithreading CPU, not used here
wire    [4:0] count_div,count_sqrt; // for testing
output [31:0] mem_a;
input  [31:0] mem_data;
output [31:0] mem_st_data;
output        mem_access;
output        mem_write;
input         mem_ready;
wire [31:0] qfa,qfb,fa,fb,dfa,dfb,mmo,wmo; // for iu
wire [4:0]  fs,ft,fd;
wire [2:0]  fc;
wire        fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
wire        e1w,e2w,e3w,wwfpr;
wire        no_cache_stall;
iu_cache_tlb i_u (e1n,e2n,e3n, e1w,e2w,e3w, stall,1'b0,
    dfb,e3d, clock,memclock,resetn,no_cache_stall,
        fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,
        fasmds,v_pc,pc,inst,ealu,malu,walu,
        stall_lw,stall_fp,stall_lwc1,stall_swc1,mem_a,
        mem_data,mem_st_data,mem_access,mem_write,mem_ready);
    wire [4:0] wrn;
    regfile2w fpr (fs,ft,wd,wn,ww,wmo,wrn,wwfpr,~clock,resetn,
                   qfa,qfb);
    mux2x32 fwd_f_load_a (qfa,mmo,fwdla,fa);
    mux2x32 fwd_f_load_b (qfb,mmo,fwdlb,fb);
    mux2x32 fwd_f_res_a  (fa,e3d,fwdfa,dfa);
    mux2x32 fwd_f_res_b  (fb,e3d,fwdfb,dfb);
    wire [1:0] e1c,e2c,e3c; // for fpu
    fpu fp_unit (dfa,dfb,fc,wf,fd,no_cache_stall,clock,resetn,
                 e3d,wd,wn,ww,stall,e1n,e1w,e2n,e2w,e3n,e3w,
                 e1c,e2c,e3c,count_div,count_sqrt,e);
endmodule
```

以下是IU模块iu_cache_tlb，包括IU、Cache和TLB。

```
module iu_cache_tlb (
    e1n,e2n,e3n, e1w,e2w,e3w, stall,st,
    dfb,e3d, clock,memclock,resetn,no_cache_stall,
    fs,ft,wmo,wrn,wwfpr,mmo,fwdla,fwdlb,fwdfa,fwdfb,fd,fc,wf,fasmds,
```

```
v_pc,pc,inst,ealu,malu,walu,
stall_lw,stall_fp,stall_lwc1,stall_swc1,
mem_a,mem_data,mem_st_data,mem_access,mem_write,mem_ready);
input [31:0] dfb,e3d;
input [4:0] e1n,e2n,e3n;
input       e1w,e2w,e3w, stall,st, clock,memclock,resetn;
output      no_cache_stall;
output [31:0] v_pc,pc,inst,ealu,malu,walu;
output [31:0] mmo,wmo;
output [4:0]  fs,ft,fd,wrn;
output [2:0]  fc;
output        wwfpr,fwdla,fwdlb,fwdfa,fwdfb,wf,fasmds;
output        stall_lw,stall_fp,stall_lwc1,stall_swc1;
output [31:0] mem_a;
input  [31:0] mem_data;
output [31:0] mem_st_data;
output        mem_access;
output        mem_write;
input         mem_ready;
parameter   EXC_BASE = 32'h80000008; // = base = BASE
wire [31:0] bpc,jpc,npc,pc4,ins,dpc4,inst,qa,qb,da,db,dimm,dc,dd;
wire [31:0] simm,epc8,alua,alub,ealu0,ealu1,ealu,sa,eb,mmo,wdi;
wire [5:0]  op,func;
wire [4:0]  rs,rt,rd,fs,ft,fd,drn,ern;
wire [3:0]  aluc;
wire [1:0]  pcsource,fwda,fwdb;
wire        wpcir;
wire        wreg,m2reg,wmem,aluimm,shift,jal;
wire [31:0] qfa,qfb,fa,fb,dfa,dfb,efb,e3d;
wire [4:0]  e1n,e2n,e3n,wn;
wire [2:0]  fc;
wire [1:0]  e1c,e2c,e3c;
reg ewfpr,ewreg,em2reg,ewmem,ejal,efwdfe,ealuimm,eshift;
reg mwfpr,mwreg,mm2reg,mwmem;
reg wwfpr,wwreg,wm2reg;
reg [31:0] epc4,ea,ed,eimm,malu,mb,wmo,walu;
reg [4:0] ern0,mrn,wrn;
reg [3:0] ealuc;
// IF
p_c vpc (next_pc,clock,resetn,wpcir&no_cache_stall,v_pc); // VPC
cla32 pc_plus4 (v_pc,32'h4,1'b0,pc4); // VPC+4
mux4x32 nextpc (pc4,bpc,da,jpc,pcsource,npc); // Next PC
wire tlbwi,tlbwr;
wire itlbwi = tlbwi & ~index[30]; // itlb write
wire itlbwr = tlbwr & ~index[30];
wire dtlbwi = tlbwi &  index[30]; // dtlb write
wire dtlbwr = tlbwr &  index[30];
```

```
wire [19:0] ipattern = (itlbwi | itlbwr) ? enthi[19:0] : v_pc[31:12];
wire pc_unmapped = v_pc[31] & ~v_pc[30]; // 10x
assign pc = pc_unmapped?{1'b0,v_pc[30:0]}:{ipte_out[19:0],v_pc[11:0]};
wire  [2:0] irandom;
wire [23:0] ipte_out;
wire        itlb_hit;
wire  [2:0] ivpn_index;
wire        ivpn_found;
tlb_8_entry itlb (entlo[23:0], itlbwi, itlbwr, index[2:0], ipattern,
                  memclock, clock, resetn,
                  irandom, ipte_out, itlb_hit, ivpn_index, ivpn_found);
wire itlb_exc = ~itlb_hit & ~pc_unmapped;
wire      i_ready,i_cache_miss;
i_cache icache (pc,ins,1'b1,i_ready,i_cache_miss,clock,resetn,
                m_i_a,mem_data,m_fetch,m_i_ready);
// IF-ID pipeline registers
dffe32 pc_4_r (pc4, clock,resetn,wpcir&no_cache_stall,dpc4); // PC4
dffe32 inst_r (ins, clock,resetn,wpcir&no_cache_stall,inst); // IR
dffe32 pcd_r  (v_pc,clock,resetn,wpcir&no_cache_stall,pcd);  // PCD
wire [31:0] pcd;
// ID
assign op   = inst[31:26];
assign rs   = inst[25:21];
assign rt   = inst[20:16];
assign rd   = inst[15:11];
assign ft   = inst[20:16];
assign fs   = inst[15:11];
assign fd   = inst[10:6];
assign func = inst[5:0];
assign simm = {{16{sext&inst[15]}},inst[15:0]};
assign jpc  = {dpc4[31:28],inst[25:0],2'b00}; // jump target
cla32 br_addr (dpc4,{simm[29:0],2'b00},1'b0,bpc); // branch target
regfile rf (rs,rt,wdi,wrn,wwreg,~clock,resetn,qa,qb); // reg file
mux4x32 alu_a (qa,ealu,malu,mmo,fwda,da); // forward A
mux4x32 alu_b (qb,ealu,malu,mmo,fwdb,db); // forward B
wire swfp,regrt,sext,fwdf,fwdfe,wfpr;
mux2x32 store_f (db,dfb,swfp,dc); // swc1
mux2x32 fwd_f_d (dc,e3d,fwdf,dd); // forward fp result
wire rsrtequ = ~|(da^db);          // rsrtequ = (da == db)
mux2x5 des_reg_no (rd,rt,regrt,drn); // destination reg
wire wepc,wcau,wsta,isbr,cancel,exc,ldst;
wire [1:0] sepc,selpc;
control cu (op,func,rs,rt,rd,fs,ft,rsrtequ,            // control unit
            ewfpr,ewreg,em2reg,ern,                    // from iu
            mwfpr,mwreg,mm2reg,mrn,                    // from iu
            e1w,e1n,e2w,e2n,e3w,e3n,stall,st,          // from fpu
            pcsource,wpcir,wreg,m2reg,wmem,jal,aluc,   // iu
```

```
            sta,aluimm,shift,sext,regrt,fwda,fwdb,      // iu
            swfp,fwdf,fwdfe,wfpr,                       // used by iu
            fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds,       // used by fpu
            stall_lw,stall_fp,stall_lwc1,stall_swc1,    // for testing
            windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr, // for tlb
            c0rn,wepc,wcau,wsta,isbr,sepc,cancel,cause,exc,selpc,ldst,
            wisbr,ecancel,itlb_exc,dtlb_exc);
wire [31:0] index; // cp0 reg 0: index
wire [31:0] entlo; // cp0 reg 2: entry lo
reg  [31:0] contx; // cp0 reg 4: context
wire [31:0] enthi; // cp0 reg 9: entry hi
wire        windex,wentlo,wcontx,wenthi; // write enables
wire        rc0,wc0; // read,write c0 res
wire [1:0]  c0rn;    // c0 reg # for mux
dffe32 c0_Index (db,clock,resetn,windex&no_cache_stall,index); // index
dffe32 c0_Entlo (db,clock,resetn,wentlo&no_cache_stall,entlo); // entlo
dffe32 c0_Enthi (db,clock,resetn,wenthi&no_cache_stall,enthi); // enthi
always @(negedge resetn or posedge clock)                      // contx
    if (resetn == 0) begin
        contx <= 0;
    end else begin
        if (wcontx) contx[31:22] <= db[31:22];              // PTEBase
        if (itlb_exc) contx[21:0] <= {v_pc[31:12],2'b00}; // BadVPN
        else if (dtlb_exc) contx[21:0] <= {malu[31:12],2'b00};
    end
wire [31:0] sta,cau,epc,sta_in,cau_in,epc_in, // for exception
            stalr,epcin,epc10,cause,c0reg,next_pc;
dffe32  c0_Status (sta_in,clock,resetn,wsta&no_cache_stall,sta); // sta
dffe32  c0_Cause  (cau_in,clock,resetn,wcau&no_cache_stall,cau); // cau
dffe32  c0_EPC    (epc_in,clock,resetn,wepc&no_cache_stall,epc); // epc
mux2x32 sta_mx (stalr,db,wc0,sta_in); // mux for Status reg
mux2x32 cau_mx (cause,db,wc0,cau_in); // mux for Cause reg
mux2x32 epc_mx (epcin,db,wc0,epc_in); // mux for EPC reg
mux2x32 sta_lr ({8'h0,sta[31:8]},{sta[23:0],8'h0},exc,stalr);
mux4x32 epc_04 (v_pc,pcd,pcm,pcw,sepc,epcin); // epc source
mux4x32 irq_pc (npc,epc,EXC_BASE,32'h0,selpc,next_pc); // for PC
mux4x32 fromc0 (contx,sta,cau,epc,ec0rn,c0reg);  // for mfc0
// ID-EXE pipeline registers
reg [31:0] pce;
reg  [1:0] ec0rn;
reg        erc0,ecancel,eisbr,eldst;
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        ewfpr   <= 0;          ewreg   <= 0;
        em2reg  <= 0;          ewmem   <= 0;
        ejal    <= 0;          ealuimm <= 0;
        efwdfe  <= 0;          ealuc   <= 0;
```

```
        eshift  <= 0;          epc4    <= 0;
        ea      <= 0;          ed      <= 0;
        eimm    <= 0;          ern0    <= 0;
        erc0    <= 0;          ec0rn   <= 0;
        ecancel <= 0;          eisbr   <= 0;
        pce     <= 0;          eldst   <= 0;
    end else if (no_cache_stall) begin
        ewfpr   <= wfpr;       ewreg   <= wreg;
        em2reg  <= m2reg;      ewmem   <= wmem;
        ejal    <= jal;        ealuimm <= aluimm;
        efwdfe  <= fwdfe;      ealuc   <= aluc;
        eshift  <= shift;      epc4    <= dpc4;
        ea      <= da;         ed      <= dd;
        eimm    <= simm;       ern0    <= drn;
        erc0    <= rc0;        ec0rn   <= c0rn;
        ecancel <= cancel;     eisbr   <= isbr;
        pce     <= pcd;        eldst   <= ldst;
    end
// EXE
cla32 ret_addr (epc4,32'h4,1'b0,epc8); // PC+8
assign sa = {eimm[5:0],eimm[31:6]}; // shift amount
mux2x32 alu_ina (ea,sa,eshift,alua); // ALU input A
mux2x32 alu_inb (eb,eimm,ealuimm,alub); // ALU input B
mux2x32 save_pc8 (ealu0,epc8,ejal,ealu1); // PC+8 if jal
mux2x32 read_cr0 (ealu1,c0reg,erc0,ealu); // read c0 regs
wire z;
alu al_unit (alua,alub,ealuc,ealu0,z); // ALU
assign ern = ern0 | {5{ejal}}; // r31 for jal
mux2x32 fwd_f_e (ed,e3d,efwdfe,eb); // forward fp result
// EXE-MEM pipeline registers
reg [31:0] pcm;
reg        misbr,mldst;
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        mwfpr   <= 0;          mwreg   <= 0;
        mm2reg  <= 0;          mwmem   <= 0;
        malu    <= 0;          mb      <= 0;
        mrn     <= 0;          misbr   <= 0;
        pcm     <= 0;          mldst   <= 0;
    end else if (no_cache_stall) begin
        mwfpr   <= ewfpr & ~dtlb_exc; // cancel
        mwreg   <= ewreg & ~dtlb_exc; // cancel
        mwmem   <= ewmem & ~dtlb_exc; // cancel
        mldst   <= eldst & ~dtlb_exc; // cancel
        mm2reg  <= em2reg;
        malu    <= ealu;       mb      <= eb;
        mrn     <= ern;        misbr   <= eisbr;
```

```
        pcm     <= pce;
    end
// MEM
wire [19:0] dpattern = (dtlbwi | dtlbwr) ? enthi[19:0] : malu[31:12];
wire ma_unmapped     =  malu[31] & ~malu[30]; // 10x; malu: va of data
wire [31:0] m_addr   =  ma_unmapped? {1'b0,malu[30:0]} :
                                     {dpte_out[19:0],malu[11:0]};
wire  [2:0] drandom;
wire [23:0] dpte_out;
wire        dtlb_hit;
wire  [2:0] dvpn_index;
wire        dvpn_found;
tlb_8_entry dtlb (entlo[23:0], dtlbwi, dtlbwr, index[2:0], dpattern,
                  memclock, clock, resetn,
                  drandom, dpte_out, dtlb_hit, dvpn_index, dvpn_found);
wire dtlb_exc = ~dtlb_hit & ~ma_unmapped & mldst;
wire d_ready;
wire w_mem = mwmem & ~dtlb_exc; // cancel sw/swc1 in mem stage
d_cache dcache (m_addr,mb,mmo,mldst,w_mem,d_ready,clock,resetn,
                m_d_a,mem_data,mem_st_data,m_ld_st,m_st,m_d_ready);
// MEM-WB pipeline registers
reg [31:0] pcw;
reg        wisbr;
always @(negedge resetn or posedge clock)
    if (resetn == 0) begin
        wwfpr   <= 0;           wwreg   <= 0;
        wm2reg  <= 0;           wmo     <= 0;
        walu    <= 0;           wrn     <= 0;
        pcw     <= 0;           wisbr   <= 0;
    end else if (no_cache_stall) begin
        wwfpr   <= mwfpr & ~dtlb_exc; // cancel lwc1 in wb stage
        wwreg   <= mwreg & ~dtlb_exc; // cancel lw   in wb stage
        wm2reg  <= mm2reg;      wmo     <= mmo;
        walu    <= malu;        wrn     <= mrn;
        pcw     <= pcm;         wisbr   <= misbr;
    end
// WB
mux2x32 wb_sel (walu,wmo,wm2reg,wdi);
// for main_memory access
wire m_fetch,m_ld_st,m_st;
wire [31:0] m_i_a,m_d_a;
// mux, i_cache has higher priority than d_cache
wire        sel_i = i_cache_miss;
assign      mem_a = sel_i? m_i_a   : m_d_a;
assign mem_access = sel_i? m_fetch : m_ld_st;
assign mem_write  = sel_i? 1'b0    : m_st;
// demux
```

```
    wire   m_i_ready  = mem_ready &  sel_i;
    wire   m_d_ready  = mem_ready & ~sel_i;
    assign no_cache_stall = ~(~i_ready | mldst & ~d_ready);
endmodule
```

以下是控制部件模块control。

```
module control (op,func,rs,rt,rd,fs,ft,rsrtequ,                  // control unit
                ewfpr,ewreg,em2reg,ern,                          // from iu
                mwfpr,mwreg,mm2reg,mrn,                          // from iu
                e1w,e1n,e2w,e2n,e3w,e3n,stall_div_sqrt,st,       // from fpu
                pcsource,wpcir,wreg,m2reg,wmem,jal,aluc,         // iu
                sta,aluimm,shift,sext,regrt,fwda,fwdb,           // iu
                swfp,fwdf,fwdfe,wfpr,                            // for lwc1,swc1
                fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmds,            // used by fpu
                stall_lw,stall_fp,stall_lwc1,stall_swc1,         // for testing
                windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr, // for tlb
                c0rn,wepc,wcau,wsta,isbr,sepc,cancel,cause,exc,selpc,ldst,
                wisbr,ecancel,itlb_exc,dtlb_exc);
    input  rsrtequ, ewreg,em2reg,ewfpr, mwreg,mm2reg,mwfpr;
    input  e1w,e2w,e3w,stall_div_sqrt,st;
    input  [5:0] op,func;
    input  [4:0] rs,rt,rd,fs,ft,ern,mrn,e1n,e2n,e3n;
    input  [31:0] sta; // IM[7:0] : x,x,dtlb_exc,itlb_exc,ov,unimpl,sys,int
    output wpcir,wreg,m2reg,wmem,jal,aluimm,shift,sext,regrt;
    output swfp,fwdf,fwdfe;
    output fwdla,fwdlb,fwdfa,fwdfb;
    output wfpr,wf,fasmds;
    output [1:0] pcsource,fwda,fwdb;
    output [3:0] aluc;
    output [2:0] fc;
    output stall_lw,stall_fp,stall_lwc1,stall_swc1; // for testing
    output windex,wentlo,wcontx,wenthi,rc0,wc0,tlbwi,tlbwr; // for tlb
    output [1:0] c0rn,sepc,selpc;
    output wepc,wcau,wsta,isbr,cancel,exc,ldst;
    output [31:0] cause;
    input  wisbr,ecancel,itlb_exc,dtlb_exc;

    assign ldst = (i_lw | i_sw | i_lwc1 | i_swc1) & ~ecancel & ~dtlb_exc;
    assign isbr = i_beq | i_bne | i_j | i_jal;
    // itlb_exc dtlb_exc isbr wisbr EPC   sepc[1:0]
    // 1        x        0    x     V_PC  0 0
    // 1        x        1    x     PCD   0 1
    // 0        1        x    0     PCM   1 0
    // 0        1        x    1     PCW   1 1
    assign sepc[1] = ~itlb_exc & dtlb_exc;
    assign sepc[0] =  itlb_exc & isbr | ~itlb_exc & dtlb_exc & wisbr;
    assign exc     =  itlb_exc & sta[4] | dtlb_exc & sta[5]; // mask
```

```
assign cancel  =  exc;
// selpc
// 0 0 : npc
// 0 1 : epc
// 1 0 : EXC_BASE
// 1 1 : x
assign selpc[1] = exc;
assign selpc[0] = i_eret;
// op     rs    rt    rd          func
// 010000 00100 xxxxx xxxxx 00000 000000 mtc0 rt, rd; c0[rd] <-- gpr[rt]
// 010000 00000 xxxxx xxxxx 00000 000000 mfc0 rt, rd; gpr[rt] <-- c0[rd]
// 010000 10000 00000 00000 00000 000010 tlbwi
// 010000 10000 00000 00000 00000 000110 tlbwr
// 010000 10000 00000 00000 00000 011000 eret
wire   i_mtc0  = (op==6'h10) & (rs==5'h04) & (func==6'h00);
wire   i_mfc0  = (op==6'h10) & (rs==5'h00) & (func==6'h00);
wire   i_eret  = (op==6'h10) & (rs==5'h10) & (func==6'h18);
assign tlbwi   = (op==6'h10) & (rs==5'h10) & (func==6'h02);
assign tlbwr   = (op==6'h10) & (rs==5'h10) & (func==6'h06);
assign windex  = i_mtc0 & (rd==5'h00); // write index
assign wentlo  = i_mtc0 & (rd==5'h02); // write entry_lo
assign wcontx  = i_mtc0 & (rd==5'h04); // write context
assign wenthi  = i_mtc0 & (rd==5'h09); // write entry_hi
assign wsta    = i_mtc0 & (rd==5'h0c) | exc | i_eret; // write status
assign wcau    = i_mtc0 & (rd==5'h0d) | exc; // write cause
assign wepc    = i_mtc0 & (rd==5'h0e) | exc; // write epc
//wire rcontx  = i_mfc0 & (rd==5'h04); // read  context
wire rstatus   = i_mfc0 & (rd==5'h0c); // read  status
wire rcause    = i_mfc0 & (rd==5'h0d); // read  cause
wire repc      = i_mfc0 & (rd==5'h0e); // read  epc
assign c0rn[1] = rcause  | repc; // c0rn 00    01  10  11
assign c0rn[0] = rstatus | repc; //      contx sta cau epc
assign rc0     = i_mfc0; // read  c0 regs
assign wc0     = i_mtc0; // write c0 regs
wire [2:0] exccode;
// 100 00 itlb_exc
// 101 00 dtlb_exc
assign exccode[2] = itlb_exc | dtlb_exc;
assign exccode[1] = 1'b0;
assign exccode[0] = dtlb_exc;
assign cause      = {27'h0,exccode,2'b00};
wire r_type,i_add,i_sub,i_and,i_or,i_xor,i_sll,i_srl,i_sra,i_jr;
and(r_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]); // r format
and(i_add,r_type, func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_sub,r_type, func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_and,r_type, func[5],~func[4],~func[3], func[2],~func[1],~func[0]);
and(i_or, r_type, func[5],~func[4],~func[3], func[2],~func[1], func[0]);
```

```
and(i_xor,r_type, func[5],~func[4],~func[3], func[2], func[1],~func[0]);
and(i_sll,r_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_srl,r_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_sra,r_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_jr, r_type,~func[5],~func[4], func[3],~func[2],~func[1],~func[0]);
wire i_addi,i_andi,i_ori,i_xori,i_lw,i_sw,i_beq,i_bne,i_lui;
and(i_addi,~op[5],~op[4], op[3],~op[2],~op[1],~op[0]);
and(i_andi,~op[5],~op[4], op[3], op[2],~op[1],~op[0]);
and(i_ori, ~op[5],~op[4], op[3], op[2],~op[1], op[0]);
and(i_xori,~op[5],~op[4], op[3], op[2], op[1],~op[0]);
and(i_lw,   op[5],~op[4],~op[3],~op[2], op[1], op[0]);
and(i_sw,   op[5],~op[4], op[3],~op[2], op[1], op[0]);
and(i_beq, ~op[5],~op[4],~op[3], op[2],~op[1],~op[0]);
and(i_bne, ~op[5],~op[4],~op[3], op[2],~op[1], op[0]);
and(i_lui, ~op[5],~op[4], op[3], op[2], op[1], op[0]);
wire i_j,i_jal;
and(i_j,   ~op[5],~op[4],~op[3],~op[2], op[1],~op[0]);
and(i_jal, ~op[5],~op[4],~op[3],~op[2], op[1], op[0]);
wire f_type,i_lwc1,i_swc1,i_fadd,i_fsub,i_fmul,i_fdiv,i_fsqrt;
and(f_type,~op[5], op[4],~op[3],~op[2],~op[1], op[0]); // f format
and(i_lwc1, op[5], op[4],~op[3],~op[2],~op[1], op[0]);
and(i_swc1, op[5], op[4], op[3],~op[2],~op[1], op[0]);
and(i_fadd,f_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);
and(i_fsub,f_type,~func[5],~func[4],~func[3],~func[2],~func[1], func[0]);
and(i_fmul,f_type,~func[5],~func[4],~func[3],~func[2], func[1],~func[0]);
and(i_fdiv,f_type,~func[5],~func[4],~func[3],~func[2], func[1], func[0]);
and(i_fsqrt,f_type,~func[5],~func[4],~func[3],func[2],~func[1],~func[0]);
wire i_rs = i_add  | i_sub | i_and  | i_or  | i_xor | i_jr  | i_addi |
            i_andi | i_ori | i_xori | i_lw  | i_sw  | i_beq | i_bne  |
            i_lwc1 | i_swc1;
wire i_rt = i_add  | i_sub | i_and  | i_or  | i_xor | i_sll | i_srl  |
            i_sra  | i_sw  | i_beq  | i_bne | i_mtc0;
assign stall_lw = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) |
                                                 i_rt & (ern == rt));
reg [1:0] fwda, fwdb;
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or
          rt) begin
    fwda = 2'b00; // default forward a: no hazards
    if (ewreg & (ern != 0) & (ern == rs) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
```

```
            end
        end
        fwdb = 2'b00; // default forward b: no hazards
        if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
            fwdb = 2'b01; // select exe_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
                fwdb = 2'b10; // select mem_alu
            end else begin
                if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                    fwdb = 2'b11; // select mem_lw
                end
            end
        end
    end
end
assign wreg   =(i_add  | i_sub  | i_and  | i_or   | i_xor | i_sll  |
                i_srl  | i_sra  | i_addi | i_andi | i_ori | i_xori |
                i_lw   | i_lui  | i_jal  | i_mfc0) &
                wpcir  & ~ecancel & ~dtlb_exc;
assign regrt  = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui |
                i_lwc1 | i_mfc0;
assign jal    = i_jal;
assign m2reg  = i_lw;
assign shift  = i_sll  | i_srl  | i_sra;
assign aluimm = i_addi | i_andi | i_ori | i_xori | i_lw | i_lui |
                i_sw | i_lwc1 | i_swc1;
assign sext   = i_addi | i_lw | i_sw | i_beq | i_bne | i_lwc1 | i_swc1;
assign aluc[3] = i_sra;
assign aluc[2] = i_sub | i_or  | i_srl | i_sra | i_ori  | i_lui;
assign aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq  |
                 i_bne | i_lui;
assign aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
assign wmem    = (i_sw | i_swc1) & wpcir & ~ecancel & ~dtlb_exc;
assign pcsource[1] = i_jr | i_j | i_jal;
assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
// fop:  000: fadd  001: fsub  01x: fmul  10x: fdiv  11x: fsqrt
wire [2:0]  fop;
assign fop[0] = i_fsub; // fpu operation control code
assign fop[1] = i_fmul | i_fsqrt;
assign fop[2] = i_fdiv | i_fsqrt;
// stall caused by fp data harzards
wire i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt; // use fs
wire i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;           // use ft
assign stall_fp = (e1w & (i_fs & (e1n == fs) | i_ft & (e1n == ft))) |
                  (e2w & (i_fs & (e2n == fs) | i_ft & (e2n == ft)));
assign fwdfa = e3w & (e3n == fs); // forward fpu e3d to fp a
assign fwdfb = e3w & (e3n == ft); // forward fpu e3d to fp b
```

```
    assign wfpr  = i_lwc1 & wpcir & ~ecancel & ~dtlb_exc; // fp regfile we
    assign fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
    assign fwdlb = mwfpr & (mrn == ft); // forward mmo to fp b
    assign stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
    assign swfp  = i_swc1; // select signal
    assign fwdf  = swfp & e3w & (ft == e3n); // forward to id  stage
    assign fwdfe = swfp & e2w & (ft == e2n); // forward to exe stage
    assign stall_swc1 = swfp & e1w & (ft == e1n); // stall
    assign wpcir = ~(stall_div_sqrt | stall_others);
    wire stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1 | st;
    assign fc = fop & {3{~stall_others}};
    assign wf = i_fs & wpcir;
    assign fasmds = i_fs;
endmodule
```

以下是主存模块，分为4个区间。

```
module physical_memory #(parameter A_WIDTH = 32)
    (a, dout, din, strobe, rw, ready, clk, memclk, clrn);
    input [A_WIDTH-1:0]  a;
    output [31:0]        dout;
    input [31:0]         din;
    input                strobe;
    input                rw;
    output               ready;
    input                clk, memclk, clrn;
    // for memory ready
    reg [2:0]            wait_counter;
    reg                  ready;
    always @ (negedge clrn or posedge clk) begin
        if (clrn == 0) begin
            wait_counter <= 3'b0;
        end else begin
            if (strobe) begin
                if (wait_counter == 3'h5) begin
                    ready <= 1'b1;
                    wait_counter <= 3'b0;
                end else begin
                    ready <= 1'b0;
                    wait_counter <= wait_counter + 3'b1;
                end
            end else begin
                ready <= 1'b0;
                wait_counter <= 3'b0;
            end
        end
    end
```

```
// 31 30 29 28 ... 15 14 13 12 ... 3  2  1  0
//  0  0  0  0      0  0  0  0     0  0  0  0 (0) 0x0000_0000
//  0  0  0  1      0  0  0  0     0  0  0  0 (1) 0x1000_0000
//  0  0  1  0      0  0  0  0     0  0  0  0 (2) 0x2000_0000
//  0  0  1  0      0  0  1  0     0  0  0  0 (3) 0x2000_2000
wire [31:0] m_out32 = a[13] ? mem_data_out3 : mem_data_out2;
wire [31:0] m_out10 = a[28] ? mem_data_out1 : mem_data_out0;
wire [31:0] mem_out = a[29] ? m_out32       : m_out10;
assign      dout    = ready ? mem_out       : 32'hzzzz_zzzz;

// (0) 0x0000_0000- (virtual address 0x8000_0000-)
wire [31:0] mem_data_out0;
wire        write_enable0 = ~a[29] & ~a[28] & rw & ~clk;
lpm_ram_dq ram0 (.data(din),.address(a[8:2]),
                 .we(write_enable0),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out0));
defparam   ram0.lpm_width   = 32;
defparam   ram0.lpm_widthad =  7;
defparam   ram0.lpm_indata  = "registered";
defparam   ram0.lpm_outdata = "registered";
defparam   ram0.lpm_file    = "cpu_cache_tlb_0.mif";
defparam   ram0.lpm_address_control = "registered";

// (1) 0x1000_0000- (virtual address 0x9000_0000-)
wire [31:0] mem_data_out1;
wire        write_enable1 = ~a[29] &  a[28] & rw & ~clk;
lpm_ram_dq ram1 (.data(din),.address(a[8:2]),
                 .we(write_enable1),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out1));
defparam   ram1.lpm_width   = 32;
defparam   ram1.lpm_widthad =  7;
defparam   ram1.lpm_indata  = "registered";
defparam   ram1.lpm_outdata = "registered";
defparam   ram1.lpm_file    = "cpu_cache_tlb_1.mif";
defparam   ram1.lpm_address_control = "registered";

// (2) 0x2000_0000- (mapped va 0x0000_0000-)
wire [31:0] mem_data_out2;
wire        write_enable2 = a[29] & ~a[13] & rw & ~clk;
lpm_ram_dq ram2 (.data(din),.address(a[8:2]),
                 .we(write_enable2),.inclock(memclk),
                 .outclock(memclk&strobe),.q(mem_data_out2));
defparam   ram2.lpm_width   = 32;
defparam   ram2.lpm_widthad =  7;
defparam   ram2.lpm_indata  = "registered";
```

```
    defparam   ram2.lpm_outdata = "registered";
    defparam   ram2.lpm_file    = "cpu_cache_tlb_2.mif";
    defparam   ram2.lpm_address_control = "registered";

    // (3) 0x2000_2000- (mapped va 0x0000_0000-)
    wire [31:0] mem_data_out3;
    wire        write_enable3 = a[29] & a[13] & rw & ~clk;
    lpm_ram_dq ram3 (.data(din),.address(a[8:2]),
                     .we(write_enable3),.inclock(memclk),
                     .outclock(memclk&strobe),.q(mem_data_out3));
    defparam   ram3.lpm_width   = 32;
    defparam   ram3.lpm_widthad = 7;
    defparam   ram3.lpm_indata  = "registered";
    defparam   ram3.lpm_outdata = "registered";
    defparam   ram3.lpm_file    = "cpu_cache_tlb_3.mif";
    defparam   ram3.lpm_address_control = "registered";
endmodule
```

13.5 带有Cache及TLB的CPU的测试程序和仿真波形

本节给出CPU的测试程序及数据。以下是初始化和异常处理程序。

```
DEPTH = 128;           % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
    % physical address = 0x0000_0000                             %
    % reset entry,  va = 0x8000_0000                             %
 0: 08000070; % (00) j    init             # jump to init        %
 1: 00000000; % (04) nop                                        %
   % EXC_BASE:                             # exception/interrupt entry  %
 2: 401a6800; % (08) mfc0 r26, C0_CAUSE    # read cp0 Cause reg  %
 3: 335a001c; % (0c) andi r26, r26, 0x1c   # get ExcCode, 3 bits here   %
 4: 3c1b8000; % (10) lui  r27, 0x8000      #                     %
 5: 037ad825; % (14) or   r27, r27, r26    #                     %
 6: 8f7b0040; % (18) lw   r27, j_table(r27)# get address from table  %
 7: 00000000; % (1c) nop                   #                     %
 8: 03600008; % (20) jr   r27              # jump to that address  %
 9: 00000000; % (24) nop                   #                     %
[a..f] : 0;
    % j_table:             # address table for exception and interrupt  %
10: 80000030; % (40)  int_entry  # 0. address for interrupt          %
11: 8000003c; % (44)  sys_entry  # 1. address for Syscall            %
```

```
12: 80000054; % (48)  uni_entry  # 2. address for Unimpl. inst.          %
13: 80000068; % (4c)  ovf_entry  # 3. address for Overflow               %
14: 800000c0; % (50)  itlb_entry # 4. address for itlb miss              %
15: 80000140; % (54)  dtlb_entry # 5. address for dtlb miss              %
16: 80000000; % (58)                                                     %
17: 80000000; % (5c)                                                     %
[18..2f] : 0;
    % itlb_entry:                                                       %
30: 3c1b8000; % (c0)  lui  r27, 0x8000      # 0x800001f8: counter        %
31: 8f7a01f8; % (c4)  lw   r26, 0x1f8(r27)  # load itlb index counter    %
32: 235a0001; % (c8)  addi r26, r26, 1      # index + 1                  %
33: 335a0007; % (cc)  andi r26, r26, 7      # 3-bit index                %
34: af7a01fc; % (d0)  sw   r26, 0x1fc(r27)  # store index                %
35: 3c1b0000; % (d4)  lui  r27, 0x0000      # itlb tag                   %
36: 037ad025; % (d8)  or   r26, r27, r26    # itlb tag and index         %
37: 409a0000; % (dc)  mtc0 r26, C0_INDEX    # move to c0 index           %
38: 401b2000; % (e0)  mfc0 r27, C0_CONTEXT  # move from c0 context       %
39: 8f7a0000; % (e4)  lw   r26, 0x0(r27)    # get pte                    %
3a: 409a1000; % (e8)  mtc0 r26, C0_ENTRY_LO # move to c0 entry_lo       %
3b: 001bd280; % (ec)  sll  r26, r27, 10     # get bad vpn                %
3c: 001ad302; % (f0)  srl  r26, r26, 12     # for c0 entry_hi            %
3d: 409a4800; % (f4)  mtc0 r26, C0_ENTRY_HI # move to entry_hi          %
3e: 42000002; % (f8)  tlbwi                # update itlb                %
3f: 42000018; % (fc)  eret                 # return from exception      %
40: 00000000; % (100) nop                  #                            %
[41..4f] : 0;
    % dtlb_entry:                                                       %
50: 3c1b8000; % (140) lui  r27, 0x8000      # 0x800001fc: counter        %
51: 8f7a01fc; % (144) lw   r26, 0x1fc(r27)  # load dtlb index counter    %
52: 235a0001; % (148) addi r26, r26, 1      # index + 1                  %
53: 335a0007; % (14c) andi r26, r26, 7      # 3-bit index                %
54: af7a01fc; % (150) sw   r26, 0x1fc(r27)  # store index                %
55: 3c1b4000; % (154) lui  r27, 0x4000      # dtlb tag                   %
56: 037ad025; % (158) or   r26, r27, r26    # dtlb tag and index         %
57: 409a0000; % (15c) mtc0 r26, C0_INDEX    # move to c0 index           %
58: 401b2000; % (160) mfc0 r27, C0_CONTEXT  # move from c0 context       %
59: 8f7a0000; % (164) lw   r26, 0x0(r27)    # get pte                    %
5a: 409a1000; % (168) mtc0 r26, C0_ENTRY_LO # move to c0 entry_lo       %
5b: 001bd280; % (16c) sll  r26, r27, 10     # get bad vpn                %
5c: 001ad302; % (170) srl  r26, r26, 12     # for c0 entry_hi            %
5d: 409a4800; % (174) mtc0 r26, C0_ENTRY_HI # move to entry_hi          %
5e: 42000002; % (178) tlbwi                # update dtlb                %
5f: 42000018; % (17c) eret                 # return from exception      %
60: 00000000; % (180) nop                  #                            %
[61..6f] : 0;

    % init                                                              %
```

```
70: 40800000; % (1c0) mtc0 r0,  C0_INDEX    # C0_INDEX <-- 0 (itlb[0])  %
71: 3c1b9000; % (1c4) lui  r27, 0x9000      # page table base           %
72: 8f7a0000; % (1c8) lw   r26, 0x0(r27)    # 1st entry of page table   %
73: 409a1000; % (1cc) mtc0 r26, C0_ENTRY_LO # C0_ENTRY_LO <-- v,d,c,pfn %
74: 3c1a0000; % (1d0) lui  r26, 0x0         # va (=0) for C0_ENTRY_HI   %
75: 409a4800; % (1d4) mtc0 r26, C0_ENTRY_HI # C0_ENTRY_HI <-- vpn (0)   %
76: 42000002; % (1d8) tlbwi                 # write itlb for user prog  %
77: 409b2000; % (1dc) mtc0 r27, C0_CONTEXT  # C0_CONTEXT <-- PTEBase    %
78: 341a003f; % (1e0) ori  r26, r0, 0x3f    # enable exceptions         %
79: 409a6000; % (1e4) mtc0 r26, C0_STATUS   # C0_STATUS <-- 0..00111111 %
7a: 3c010000; % (1e8) lui  r1, 0x0          # va = 0x0000_0000          %
7b: 00200008; % (1ec) jr   r1               # jump to user program      %
7c: 00000000; % (1f0) nop                  #                           %
7d: 00000000; % (1f4) nop                  #                           %
7e: 00000000; % (1f8) .data 0              # itlb index counter        %
7f: 00000000; % (1fc) .data 0              # dtlb index counter        %
END ;
```

页表，此处很小，实际很大。系统软件可以维护这个页表：

```
DEPTH = 128;           % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
    % physical address = 0x1000_0000 %
    % page table,   va = 0x9000_0000 %
 0: 00820000; % (00) va: 0000_0000 --> pa: 20000000 ; 1 of 8: valid bit %
 1: 00820002; % (04) va: 0000_1000 --> pa: 20002000 ; 1 of 8: valid bit %
 2: 00820001; % (08) va: 0000_2000 --> pa: 20001000 ; 1 of 8: valid bit %
 3: 008200f0; % (0c) va: 0000_3000 --> pa: 200f0000 ; 1 of 8: valid bit %
[4..7F] : 00000000;
END ;
```

IU / FPU 测试程序：

```
DEPTH = 128;           % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
   % physical address = 0x2000_0000 %
 0 : 20011100; %(20000000)   addi  r1,r0,0x1100  # address of data[0]    %
 1 : C4200000; %(20000004)   lwc1  f0,  0x0(r1)  # load fp data          %
```

```
 2 : C4210050; %(20000008)   lwc1  f1, 0x50(r1)  # load fp data          %
 3 : C4220054; %(2000000C)   lwc1  f2, 0x54(r1)  # load fp data          %
 4 : C4230058; %(20000010)   lwc1  f3, 0x58(r1)  # load fp data          %
 5 : C424005C; %(20000014)   lwc1  f4, 0x5c(r1)  # load fp data          %
 6 : 46002100; %(20000018)   add.s f4,  f4, f0   # f4: stall 1           %
 7 : 460418C1; %(2000001C)   sub.s f3,  f3, f4   # f4: stall 2           %
 8 : 46022082; %(20000020)   mul.s f2,  f4, f2   # mul                   %
 9 : 46040842; %(20000024)   mul.s f1,  f1, f4   # mul                   %
 A : E4210070; %(20000028)   swc1  f1, 0x70(r1)  # f1: stall 1           %
 B : E4220074; %(2000002C)   swc1  f2, 0x74(r1)  # store fp data         %
 C : E4230078; %(20000030)   swc1  f3, 0x78(r1)  # store fp data         %
 D : E424007C; %(20000034)   swc1  f4, 0x7c(r1)  # store fp data         %
 E : 20020004; %(20000038)   addi  r2,  r0,  4   # counter               %
 F : C4230000; %(2000003C) 13: lwc1  f3, 0x0(r1) # load fp data          %
10 : C4210050; %(20000040)   lwc1  f1,  0x50(r1) # load fp data          %
11 : 46030840; %(20000044)   add.s f1,  f1, f3   # stall 1               %
12 : 46030841; %(20000048)   sub.s f1,  f1, f3   # stall 2               %
13 : E4210030; %(2000004C)   swc1  f1,  0x30(r1) # stall 1               %
14 : C4051104; %(20000050)   lwc1  f5,0x1104(r0) # load fp data          %
15 : C4061108; %(20000054)   lwc1  f6,0x1108(r0) # load fp data          %
16 : C408110C; %(20000058)   lwc1  f8,0x110c(r0) # load fp data          %
17 : 460629C3; %(2000005C)   div.s f7,  f5, f6   # div                   %
18 : 46004244; %(20000060)   sqrt.s f9,  f8      # sqrt                  %
19 : 46004A84; %(20000064)   sqrt.s f10, f9      # sqrt                  %
1A : 2042FFFF; %(20000068)   addi  r2,  r2, -1   # counter - 1           %
1B : 1440FFF3; %(2000006C)   bne   r2,  r0, 13   # finish?               %
1C : 20210004; %(20000070)   addi  r1,  r1,  4   # address+4, DELAY SLOT %
1D : 3c010000; %(20000074) iu_test:lui  r1, 0    # address of data[0]    %
1E : 34241150; %(20000078)   ori  r4, r1, 0x1150 # address of data[0]    %
1F : 0c000038; %(2000007C) call:   jal  sum      # call function         %
20 : 20050004; %(20000080) dslot1: addi r5,r0,4  # DELYED SLOT(DS)       %
21 : ac820000; %(20000084) return: sw  r2, 0(r4) # store result         %
22 : 8c890000; %(20000088)   lw   r9, 0(r4)      # check sw              %
23 : 01244022; %(2000008C)   sub  r8, r9, r4     # sub: r8 <-- r9 - r4   %
24 : 20050003; %(20000090)   addi r5, r0,  3     # counter               %
25 : 20a5ffff; %(20000094) loop2: addi r5,r5,-1  # counter - 1           %
26 : 34a8ffff; %(20000098)   ori  r8, r5, 0xffff # zero-extend: 0000ffff %
27 : 39085555; %(2000009C)   xori r8, r8, 0x5555 # zero-extend: 0000aaaa %
28 : 2009ffff; %(200000A0)   addi r9, r0, -1     # sign-extend: ffffffff %
29 : 312affff; %(200000A4)   andi r10, r9,0xffff # zero-extend: 0000ffff %
2A : 01493025; %(200000A8)   or   r6, r10, r9    # or:  ffffffff         %
2B : 01494026; %(200000AC)   xor  r8, r10, r9    # xor: ffff0000         %
2C : 01463824; %(200000B0)   and  r7, r10, r6    # and: 0000ffff         %
2D : 10a00003; %(200000B4)   beq  r5, r0, shift  # if r5 = 0, goto shift %
2E : 00000000; %(200000B8) dslot2: nop           # DS                    %
2F : 08000025; %(200000BC)   j    loop2          # jump loop2            %
30 : 00000000; %(200000C0) dslot3: nop           # DS                    %
```

```
31 : 2005ffff; %(200000C4) shift:  addi r5,r0,-1 # r5   = ffffffff       %
32 : 000543c0; %(200000C8)   sll  r8, r5, 15     # <<  15 = ffff8000     %
33 : 00084400; %(200000CC)   sll  r8, r8, 16     # <<  16 = 80000000     %
34 : 00084403; %(200000D0)   sra  r8, r8, 16     # >>> 16 = ffff8000     %
35 : 00084c32; %(200000D4)   srl  r8, r8, 15     # >>  15 = 0001ffff     %
36 : 08000036; %(200000D8) finish: j finish      # dead loop             %
37 : 00000000; %(200000DC) dslot4: nop           # DS                    %
38 : 00004020; %(200000E0) sum: add  r8, r0, r0  # sum                   %
39 : 8c890000; %(200000E4) loop: lw r9, 0(r4)    # load data             %
3A : 01094020; %(200000E8)   add  r8, r8, r9     # sum                   %
3B : 20a5ffff; %(200000EC)   addi r5, r5, -1     # counter - 1           %
3C : 14a0fffc; %(200000F0)   bne  r5, r0, loop   # finish?               %
3D : 20840004; %(200000F4) dslot5: addi r4,r4,4  # address + 4, DS       %
3E : 03e00008; %(200000F8)   jr   r31            # return                %
3F : 00081000; %(200000FC) dslot6: sll  r2,r8,0  # move res. to v0, DS   %
[40..7F] : 0;
END ;
```

IU/FPU 测试数据：

```
DEPTH = 128;           % Memory depth and width are required    %
WIDTH = 32;            % Enter a decimal number                 %
ADDRESS_RADIX = HEX;   % Address and value radixes are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless    %
                       % otherwise specified, radixes = HEX     %
CONTENT
BEGIN
    % physical address = 0x2000_2000 %
[0..3F] :   0; % (20002000..200020FC) 0 %
40 : BF800000; % (20002100) 1 01111111 00..0 fp -1 %
41 : 40800000; % (20002104) %
42 : 40000000; % (20002108) %
43 : 41100000; % (2000210C) %
[44..53] :  0; % (20002110..2000214C) 0 %
54 : 40C00000; % (20002150) 0 10000001 10..0 data[0] 4.5%
55 : 41C00000; % (20002154) 0 10000011 10..0 data[1] %
56 : 43C00000; % (20002158) 0 10000111 10..0 data[2] %
57 : 47C00000; % (2000215C) 0 10001111 10..0 data[3] %
[58..7F] :  0; % (20002160..200021FC) 0 %
END ;
```

图 13.10 ～ 图 13.12 是执行测试程序时的部分波形。复位后，CPU 从虚拟地址 0x80000000 开始执行程序，主要工作是为用户程序初始化一个 ITLB 项。然后转去执行用户程序。当用户程序引起 DTLB 不命中异常时，转去执行异常处理程序，填充 DTLB 项，然后返回。为了看波形图方便，我们假设 Cache 不命中时需要 6 个周期访问存储器 (实际不止 6 个周期)。

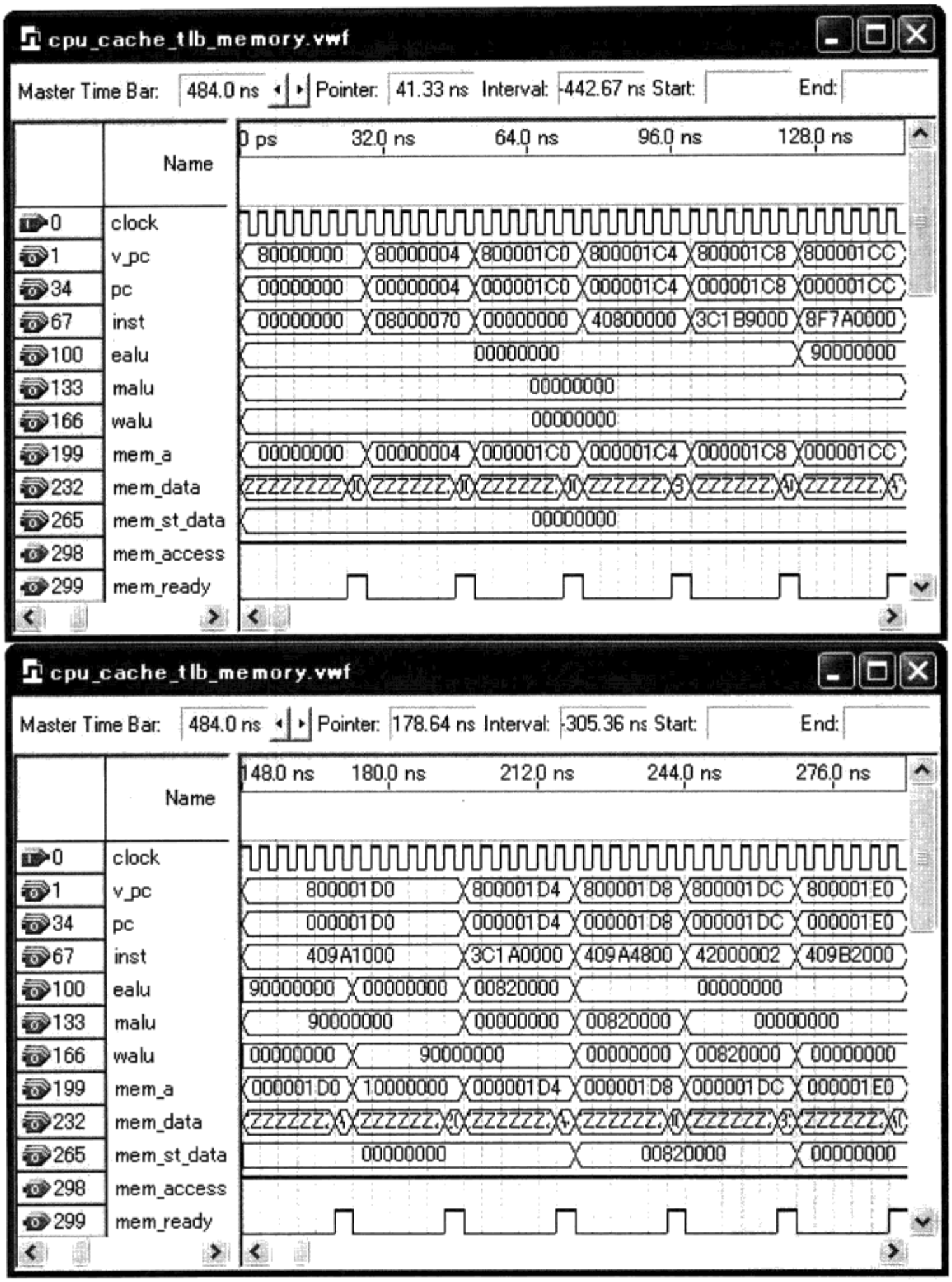

图 13.10 流水线 CPU 仿真波形图 (复位后)

图 13.10 中的 v_pc 是指令的虚拟地址，pc 是实际地址。这个区域的地址转换不经过 ITLB。由于是刚开始，指令 Cache 一定是不命中。

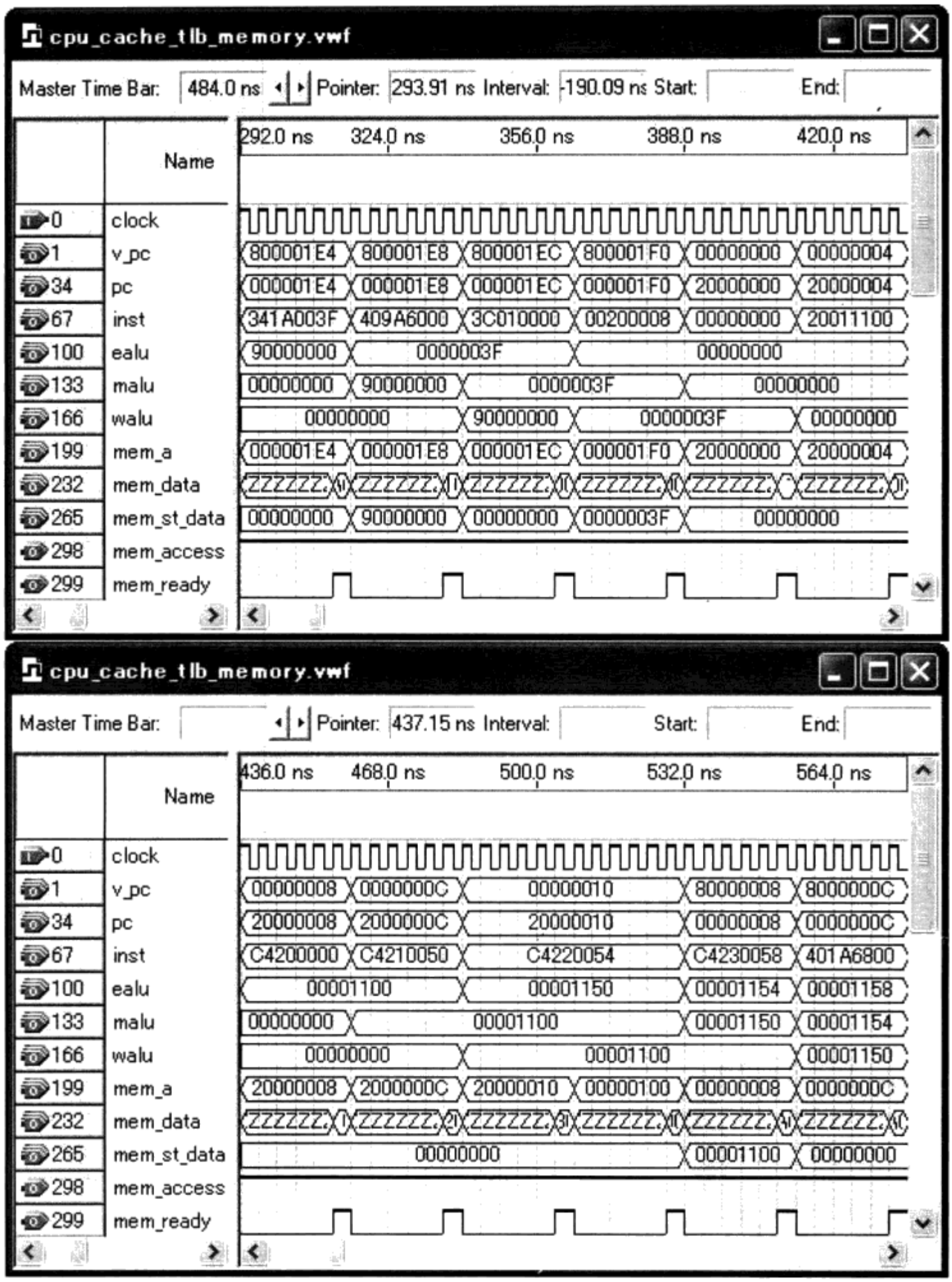

图 13.11 流水线 CPU 仿真波形图 (转用户程序)

图 13.11 上半部分示出的是初始化完成后转去执行用户程序的波形。下半部分示出的是 lwc1 指令执行到 MEM 级时 DTLB 不命中而转去执行异常处理程序的波形。

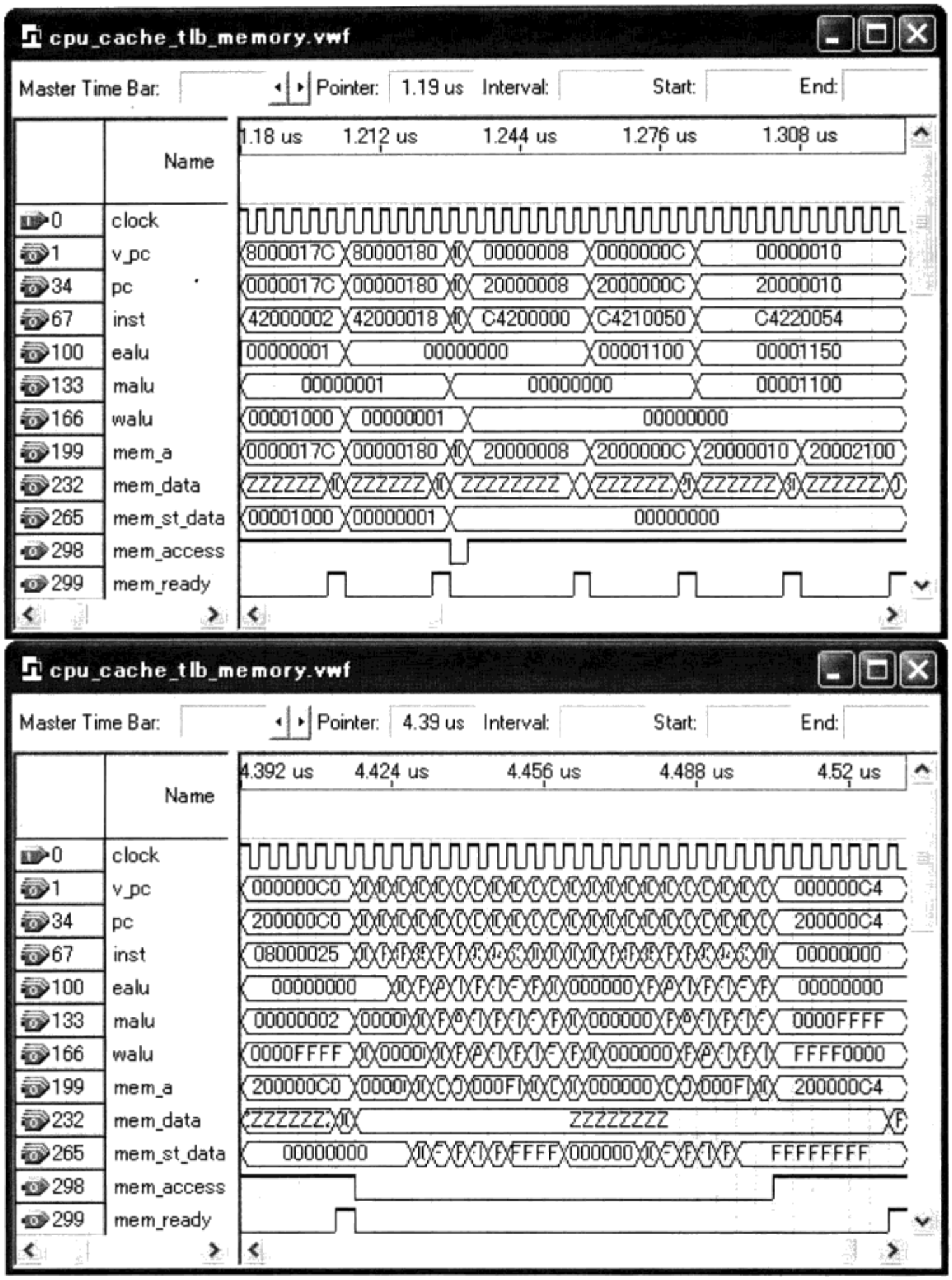

图 13.12 流水线 CPU 仿真波形图 (从异常返回和指令 Cache 命中)

图 13.12 上半部分是从异常处理返回时的波形。返回地址是 0x00000004，即重新执行 lwc1 指令 (Cache 命中)。下半部分是指令 Cache 命中时的波形。