

第11章 多线程CPU及其Verilog HDL设计

多核(Multi-Core)及多线程(Multithreading)技术已被世界知名计算机公司(比如Intel、IBM、AMD、Fujitsu等)广泛用于CPU的设计中。本章主要讨论多线程CPU的原理及设计。

11.1 多线程CPU概述

本节简要介绍多线程的基本概念以及能够并行执行多线程的CPU的基本结构。

11.1.1 多线程CPU的基本概念

一个线程是指一段程序在执行过程中不依赖其他线程产生的数据。见图11.1，线程D和线程E从理论上讲可以并行执行，但线程F只有等到线程D和线程E均执行完才能开始执行。

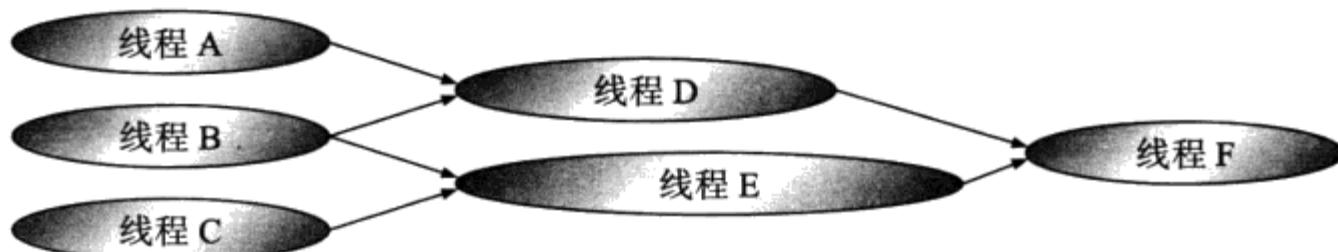


图11.1 线程的基本概念

虽然有些线程可以并行执行，但如果CPU不提供相应的硬件支持，多个本来可以并行执行的线程也只能轮流被执行，如图11.2(a)所示。传统的CPU就属于这种情况。从任意一个时间点来看，CPU实际上只在执行一个线程。我们称其为单线程CPU。依线程切换的方式不同，单线程CPU又可分为两种，这里不再详细描述。

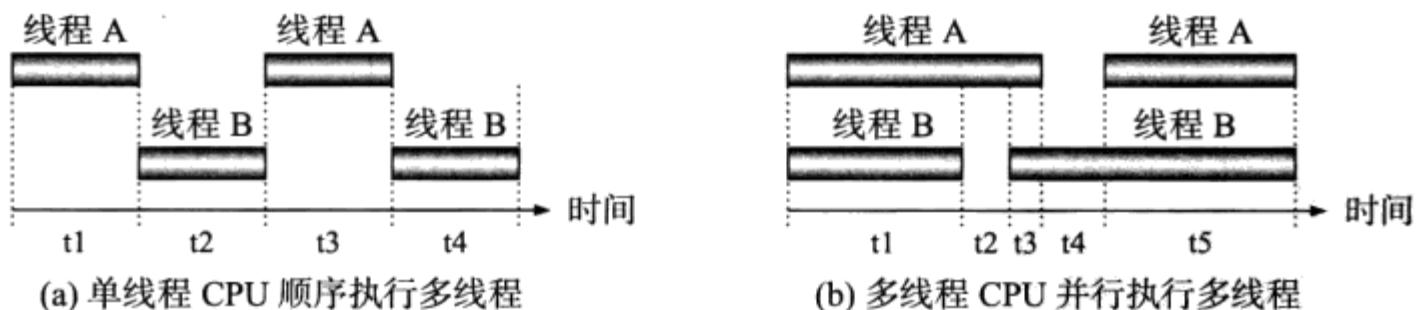


图11.2 多线程CPU的基本概念

而多线程CPU则不同。如图11.2(b)所示，在时间段t1、t3和t5中，两个线程可以并行地被执行。这样的CPU需要有多个程序计数器以及能够完成运算功能的多个功能部件(Functional Units)。与单纯的多核技术相比，多线程CPU的好处是功能部件可以得到充分的利用(多个线程共享功能部件)^[6, 7, 34, 2]。

11.1.2 多线程CPU的基本结构

图11.3示出的是最简单的双线程CPU的一个例子。图中有两个程序计数器(PC0和PC1)及两个寄存器堆(Register File 0和Register File 1)，它能同时执行两个线程，相当于有两个逻辑意义上的CPU。三个功能部件(两个ALU和一个FPU)被两个逻辑CPU所共享。当两个线程同时要求使用FPU时，其中一个的要求得到满足，而另一个必须等待。下面一节详细讨论线程的选择方法。

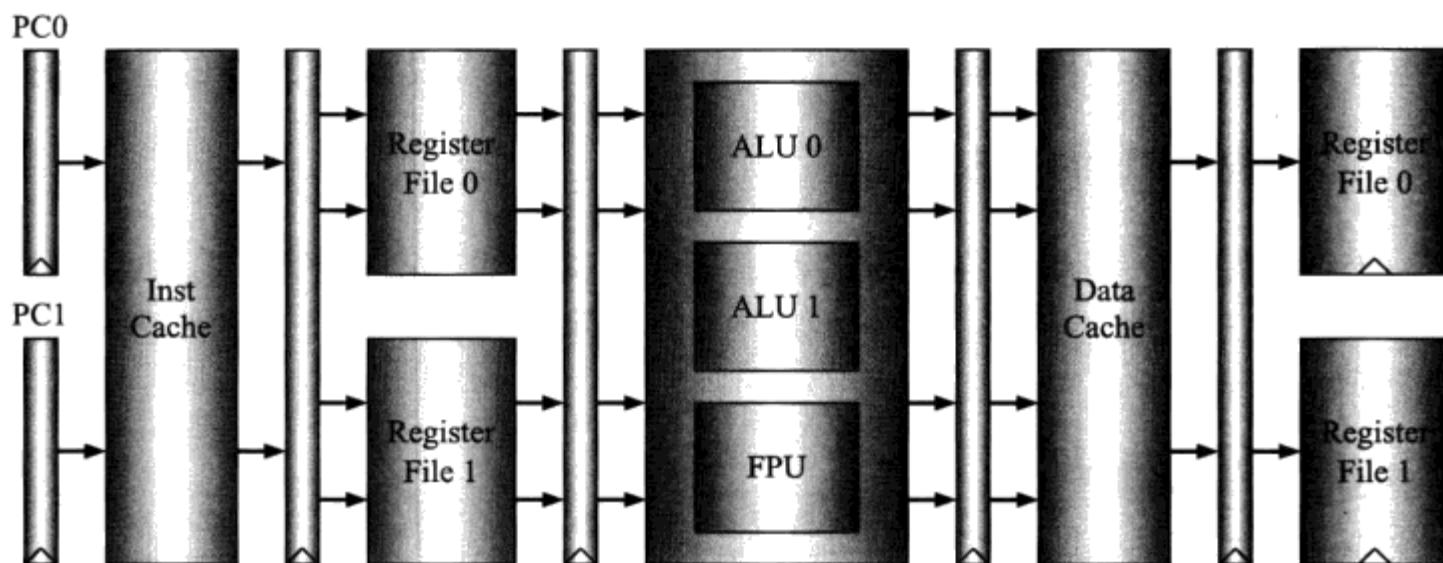


图11.3 多线程CPU的基本结构

11.2 多线程CPU设计

本节给出多线程CPU的线程选择方法、详细电路及具体的Verilog HDL代码。

11.2.1 线程的选择方法

假设多线程CPU中某种功能部件只有一个，问题就出现了：当多个线程同时要使用这个功能部件时，CPU到底选择哪个线程？我们可以用一个简单的计数器来选择一个线程。如果想让某些线程有较高的优先级得到执行，我们可以增大计数器的计数范围，多选一些计数值让那些线程有优先被执行的权力。比如一个3位的计数器可以使两个线程的优先级之比为5:3。本章给出双线程CPU的设计方法，而且令两个线程有相同的优先级。

图11.4示出的是双线程CPU选择线程的电路，模块名为selthread。信号fasmds0和fasmds1分别表示线程0和线程1是否有使用浮点部件的请求；信号dt表示在ID级被选中的线程；信号st0和st1分别表示是否要停线程0和线程1的流水线。左边的一个dff和一个非门构成一个计数器，输出为cnt。右边4个dff是流水线寄存器，它们的输出分别对应流水线的E1、E2、E3和WB级。中间的thread_sel模块是电路的主要部分，它的真值表在表11.1中给出。

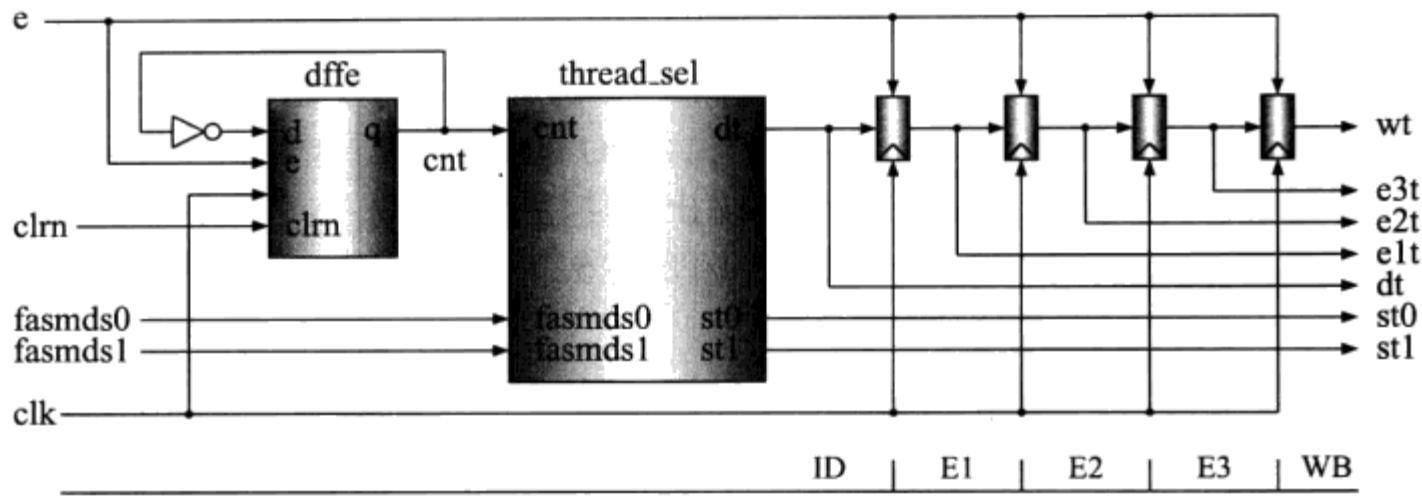


图 11.4 线程的选择电路 selthread

表 11.1 线程的选择方法 (图 11.4 中的 thread_sel 模块)

输入			输出		
cnt	fasmds0	fasmds1	dt	st0	st1
0	1	0	0	0	0
0	1	1	0	0	1
0	0	1	1	0	0
1	0	1	1	0	0
1	1	1	1	1	0
1	1	0	0	0	0
x	0	0	0	0	0

表 11.1 的内容应该不难理解。当 fasmds0 和 fasmds1 均为 1 时 (都有请求)，根据 cnt 选择线程 dt 并封锁线程 \bar{dt} 。st0 和 st1 分别是线程 0 和线程 1 的暂停信号。由此，我们得到 3 个输出信号的逻辑表达式：

```
dt = ~fasmds0 & fasmds1 | cnt & fasmds1;
st0 = cnt & fasmds0 & fasmds1;
st1 = ~cnt & fasmds0 & fasmds1;
```

11.2.2 多线程 CPU 的详细电路

我们已经在第 10 章介绍了带有 FPU 的流水线 CPU 的设计，以此为基础，本小节详细介绍双线程 CPU 的设计。

我们的双线程 CPU 有两个 ALU 和一个 FPU。本来两个 ALU 应该被两个线程所共享，为了简化设计，我们的双线程 CPU 为每个线程分配一个固定的 ALU。这相当于 CPU 中有两个独立的 IU 和一个共享的 FPU。

如果两个线程中的任何一个线程完全没有浮点指令时，两个线程可以互不干扰地并行执行。只有两个线程同时要执行浮点指令时，才会出现竞争 FPU 的情况。因此，多线程 CPU 设计的重点和难点在于如何解决对共享部件的竞争的问题。

我们已经在 11.2.1 小节介绍了当竞争出现时如何选择线程的电路。从被选中的线程的浮点寄存器堆读出的数据要被送到 FPU，因此我们需要在 FPU 的数据输入端使用一个二选一多路器 (Multiplexer)。另外，FPU 的计算结果也要被写入相应线程的浮点寄存器堆中，因此我们要在 FPU 的输出端使用一个类似于反向多路器的电路，我们暂且称其为分配器 (Demultiplexer)，见图 11.5。

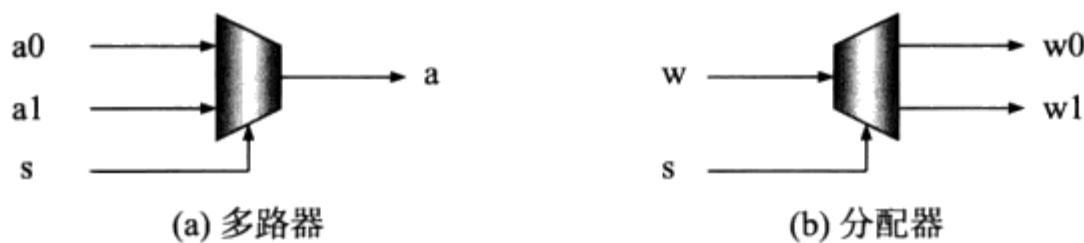


图 11.5 分配器与多路器的比较

比如浮点寄存器堆的写使能信号就要用到分配器。分配器输出信号的逻辑表达式如下，其中的信号 s 指明把输入信号 w 分配给谁： w_0 还是 w_1 。

w0 = ~s & w;
w1 = s & w;

有些信号并不需要分配，比如目的寄存器号和计算结果。有了多路器和分配器这两件神器，一个双线程的 CPU 的结构大致就出来了，见图 11.6。图中有两个整数部件 iu_0 和 iu_1 ；两个浮点寄存器堆以及相应的用于数据内部前推的多路器。图中 fpu 模块左边的是多路器 (mux)，右边的是分配器 (demux)。分配器的选择信号来自于 11.2.1 小节描述的 $selthread$ 模块，即图中右下角部分。

要被分配的信号处在不同的流水线级：stall、e1w、e2w、e3w 和 ww 分别处在流水线的 ID、E1、E2、E3 和 WB 级，相应的分配信号分别为 dt、e1t、e2t、e3t 和 wt。分配器的输出 ww0 和 ww1 分别接到线程 0 和线程 1 的浮点寄存器堆的写使能端，其余的接到相应的 iu。selthread 模块的输出信号 st0 和 st1 也接到相应的 iu，用于暂停流水线，暂停的原因是 FPU 的资源竞争。

多路器 mux 的选择信号是 ID 级的 dt。多路器的两路输入分别来自两个线程，其中除了浮点数据来自浮点寄存器堆右面的多路器，其余的均来自整数部件。整数部件 iu0 和 iu1 是等价的，与我们在第 10 章描述的相同；fpu 模块也与第 10 章描述的相同。

11.2.3 多线程 CPU 的 Verilog HDL 代码

以下是双线程 CPU 的 Verilog HDL 代码，模块名为 fpu_2_iu，它实现图 11.6 的电路。该模块的输出信号较多，完全是为了测试方便而设置的。

```
module fpu_2_iu (resetn,memclock,clock,  
pc0.inst0.ealu0.malu0.walu0.ww0,
```

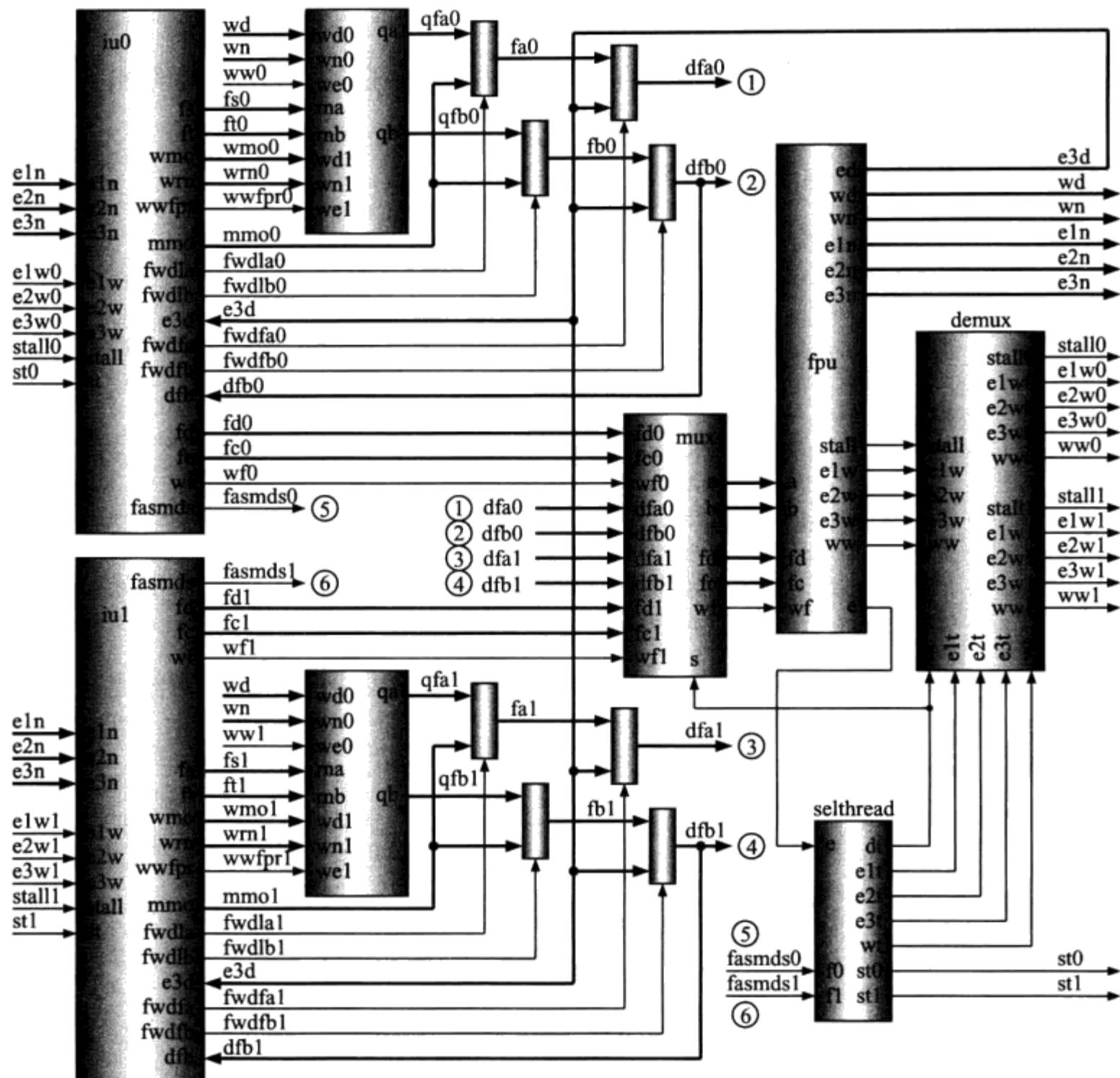


图 11.6 多线程 CPU 的模块图

```

    stall_lw0, stall_lwc10, stall_swc10, stall_fp0, stall0, st0,
    pcl, inst1, ealu1, malu1, walu1, ww1,
    stall_lw1, stall_lwc11, stall_swc11, stall_fp1, stall1, st1,
    wn, wd, count_div, count_sqrt, e1n, e2n, e3n, e3d, e);
input clock, memclock, resetn;
output [31:0] pc0, inst0, ealu0, malu0, walu0;
output [31:0] pc1, inst1, ealu1, malu1, walu1;
output ww0, stall_lw0, stall_lwc10, stall_swc10, stall_fp0, stall0, st0;
output ww1, stall_lw1, stall_lwc11, stall_swc11, stall_fp1, stall1, st1;
output e; // enable
output [31:0] e3d, wd;
output [4:0] e1n, e2n, e3n, wn;

```

```
output [4:0] count_div, count_sqrt;
```

以下是线程0(图11.6中的iu0)的代码，其中的iu模块已在第10章给出：

```
wire [31:0] qfa0,qfb0,fa0,fb0,dfa0,dfb0,mmo0,wmo0;
wire [4:0] fs0,ft0,fd0,wrn0;
wire [2:0] fc0;
wire      fwdla0,fwdlb0,fwdfa0,fwdfb0,wf0,fasmds0;
iu iu0 (eln,e2n,e3n, elw0,e2w0,e3w0, stall0,st0,
         dfb0,e3d, clock,memclock,resetn,
         fs0,ft0,wmo0,wrn0,wwfpr0,mmo0,fwdla0,fwdlb0,fwdfa0,fwdfb0,
         fd0,fc0,wf0,fasmds0,pc0,inst0,ealu0,malu0,walu0,
         stall_lw0,stall_fp0,stall_lwc10,stall_swc10);
regfile2w fpr0 (fs0,ft0,wd,wn,ww0,wmo0,wrn0,wwfpr0,
                  ~clock,resetn,qfa0,qfb0);
mux2x32 fwd_f_load_a0 (qfa0,mmo0,fwdla0,fa0); // forward lwcl to fp a
mux2x32 fwd_f_load_b0 (qfb0,mmo0,fwdlb0,fb0); // forward lwcl to fp b
mux2x32 fwd_f_res_a0  (fa0,e3d,fwdfa0,dfa0); // forward fp res to fp a
mux2x32 fwd_f_res_b0  (fb0,e3d,fwdfb0,dfb0); // forward fp res to fp b
```

以下是线程1(图11.6中的iu1)的代码，其中的iu模块已在第10章给出：

```
wire [31:0] qfa1,qfb1,fa1,fb1,dfa1,dfb1,mmol1,wmol1;
wire [4:0] fs1,ft1,fd1,wrn1;
wire [2:0] fc1;
wire      fwdla1,fwdlb1,fwdfa1,fwdfb1,wf1,fasmds1;
iu iu1 (eln,e2n,e3n, elw1,e2w1,e3w1, stall1,st1,
         dfb1,e3d, clock,memclock,resetn,
         fs1,ft1,wmol1,wrn1,wwfpr1,mmo1,fwdla1,fwdlb1,fwdfa1,fwdfb1,
         fd1,fc1,wf1,fasmds1,pc1,inst1,ealul,malul,walul,
         stall_lw1,stall_fp1,stall_lwc11,stall_swc11);
regfile2w fpr1 (fs1,ft1,wd,wn,ww1,wmol1,wrn1,wwfpr1,
                  ~clock,resetn,qfa1,qfb1);
mux2x32 fwd_f_load_a1 (qfa1,mmol1,fwdla1,fa1); // forward lwcl to fp a
mux2x32 fwd_f_load_b1 (qfb1,mmol1,fwdlb1,fb1); // forward lwcl to fp b
mux2x32 fwd_f_res_a1  (fa1,e3d,fwdfa1,dfa1); // forward fp res to fp a
mux2x32 fwd_f_res_b1  (fb1,e3d,fwdfb1,dfb1); // forward fp res to fp b
```

以下是共享的浮点部件(图11.6中的fpu)，代码已在第10章给出：

```
wire [1:0] e1c,e2c,e3c;
fpu fp_unit (dfa,dfb,fc,wf,fd,1'b1,clock,resetn,e3d,wd,wn,ww,
              stall,eln,elw,e2n,e2w,e3n,e3w,
              e1c,e2c,e3c,count_div,count_sqrt,e);
```

多路器，选择线程0还是线程1(图11.6中的mux)：

```
wire [31:0] dfa,dfb; // fp inputs a and b
wire [4:0] fd;       // fp destination register number
```

```

wire [2:0] fc;           // fp operation code
wire wf;                // fp register file write enable
assign dfa = dt? dfa1 : dfa0;
assign dfb = dt? dfb1 : dfb0;
assign fd = dt? fd1 : fd0;
assign wf = dt? wf1 : wf0;
assign fc = dt? fc1 : fc0;

```

分配器(图 11.6 中的 demux):

```

// demux: for thread 0;          for thread 1
wire stall0 = stall & ~dt;      wire stall1 = stall & dt; // ID stage
wire e1w0 = e1w & ~elt;        wire e1w1 = e1w & elt; // E1 stage
wire e2w0 = e2w & ~e2t;        wire e2w1 = e2w & e2t; // E2 stage
wire e3w0 = e3w & ~e3t;        wire e3w1 = e3w & e3t; // E3 stage
wire ww0 = ww & ~wt;          wire ww1 = ww & wt; // WB stage

```

线程选择信号及流水线暂停信号的产生(图 11.6 中的 selthread, 细节见图 11.4):

```

// thread selection
assign st0 = cnt & fasmds0 & fasmds1;           // stall thread 0
assign st1 = ~cnt & fasmds0 & fasmds1;           // stall thread 1
wire dt = ~fasmds0 & fasmds1 | cnt & fasmds1; // selected thread

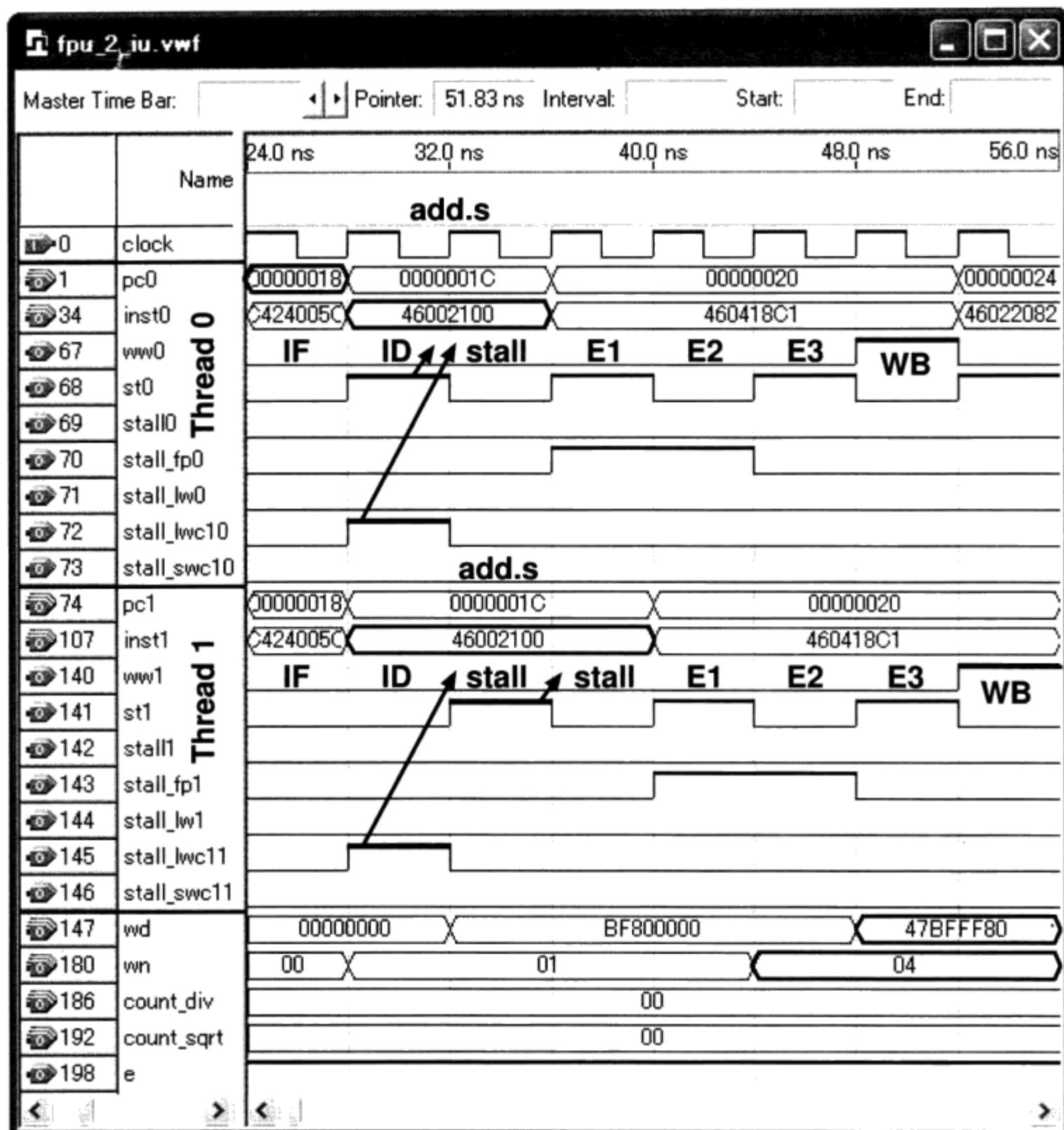
// count for thread selection
reg cnt;
always @(negedge resetn or posedge clock)
  if (resetn == 0) begin
    cnt <= 0;
  end else if (e) begin // enable
    cnt <= ~cnt;
  end

// pipelined thread info
reg elt,e2t,e3t,wt;
always @(negedge resetn or posedge clock)
  if (resetn == 0) begin
    elt <= 0;      e2t <= 0;      e3t <= 0;      wt <= 0;
  end else if (e) begin // enable
    elt <= dt;    e2t <= elt;    e3t <= e2t;    wt <= e3t;
  end
endmodule

```

11.3 多线程 CPU 的仿真波形

在本测试中, 两个线程执行同样的程序。我们使用第 10 章的测试程序, 重点给出浮点运算指令执行时的波形。以下图中最上部分是线程 0 的信号, 中间部分是线程 1 的信号, 最下部分是公用信号(FPU 部分)。执行的指令列在波形的下面。



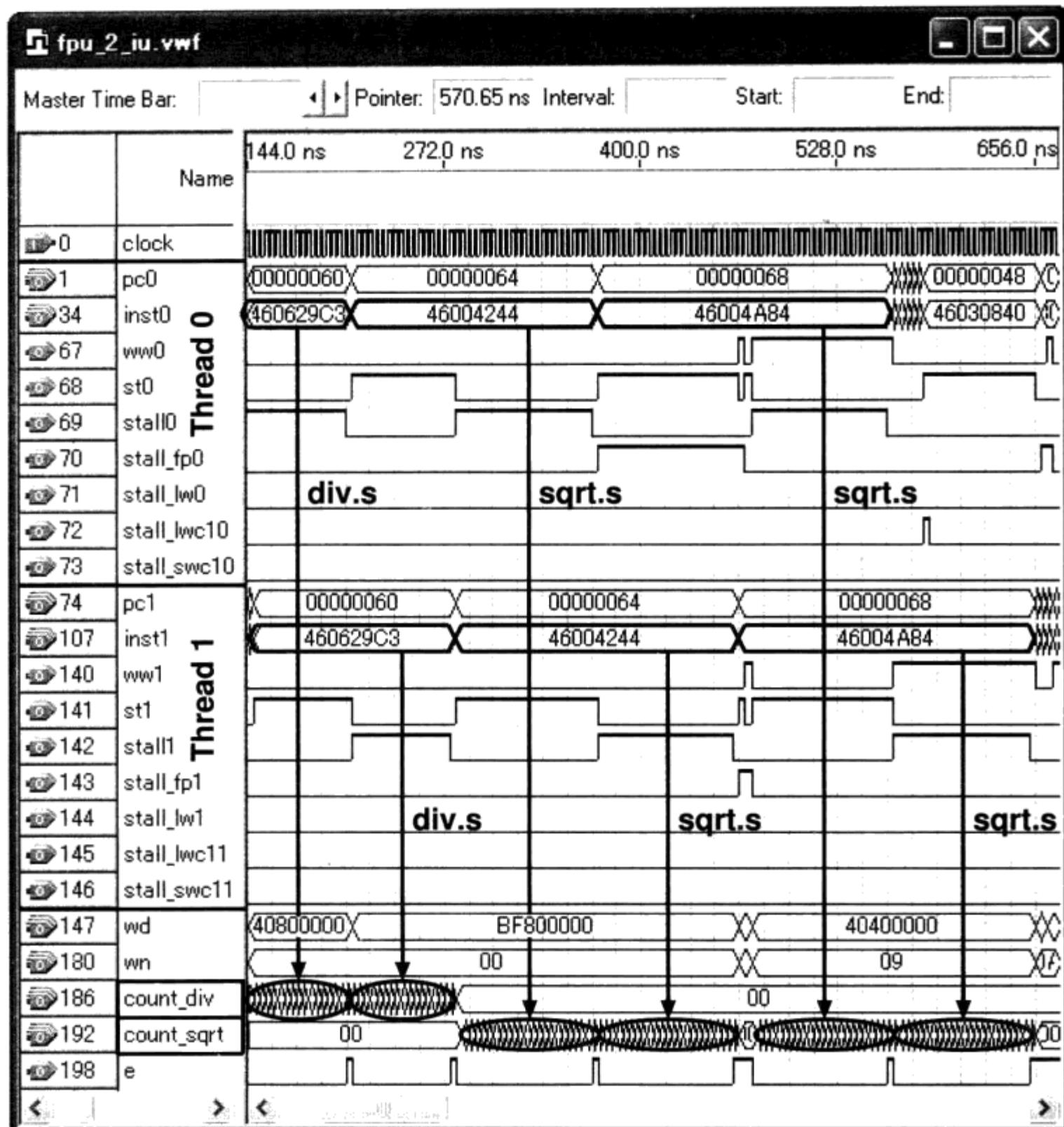
```

5 : C424005C; % (14)      lwc1    f4,   92(r1) # load fp data %
6 : 46002100; % (18)      add.s    f4,   f0 # f4: stall 1 %
7 : 460418C1; % (1C)      sub.s    f3,   f4 # f4: stall 2 %

```

图 11.7 多线程 CPU 仿真波形图 (线程 0 浮点加和线程 1 浮点加)

图 11.7 中线程 0 流水线的第一个暂停是由 add.s 指令与 lwc1 数据相关 (stall_lwc10) 或浮点部件资源冲突 (st0) 引起的。线程 1 的流水线暂停两个周期，一个是由 add.s 指令与 lwc1 数据相关引起的、另一个是由浮点部件资源冲突引起的。由于 sub.s 与 add.s 数据相关，流水线还要暂停两个周期。线程 0 和线程 1 分别在 48ns ~ 50ns (前半个周期) 和 52ns ~ 54ns 处把 wd = 47BFFF80 写入各自的 wn = 04 浮点寄存器。



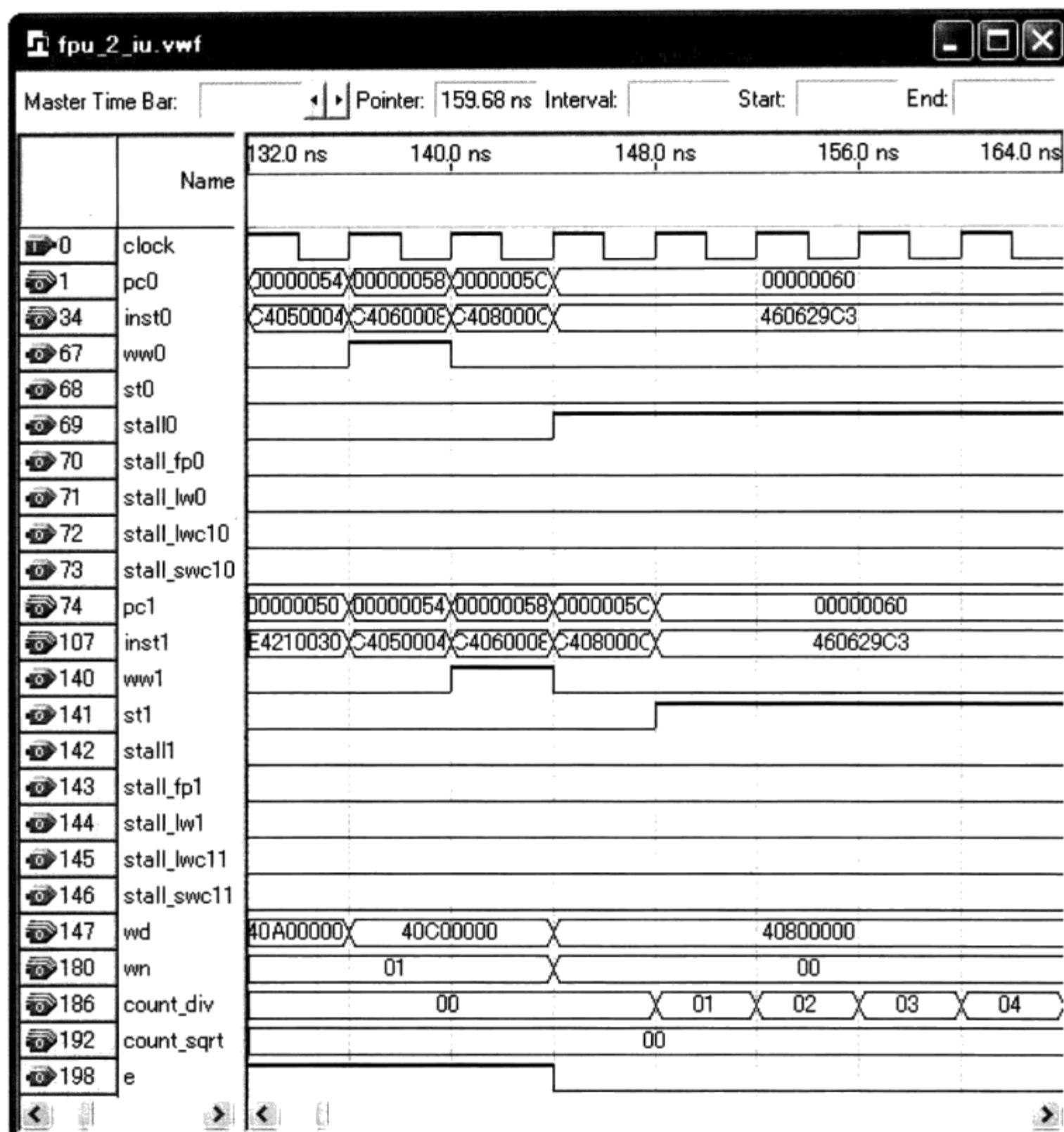
```

17 : 460629C3; % (5C)      div.s f7, f5, f6 # div %
18 : 46004244; % (60)      sqrt.s f9, f8     # sqrt %
19 : 46004A84; % (64)      sqrt.s f10, f9    # sqrt %

```

图 11.8 多线程 CPU 仿真波形图 (div.s、sqrt.s、sqrt.s 指令序列总体图)

图 11.8 示出的是执行 div.s、sqrt.s、sqrt.s 指令序列时的总体波形。下边密密麻麻的是除法计数器和开方计数器的值，在此期间完成 Newton-Raphson 迭代。两个线程轮流执行，先从线程 0 开始。线程 0 迭代时，stall0 为 1 (暂停自己的流水线等待迭代完成)、st1 为 1 (由于资源冲突而暂停对方的流水线)；线程 1 迭代时，stall1 为 1、st0 为 1。我们将陆续给出能够看清计数器值的详细的波形。



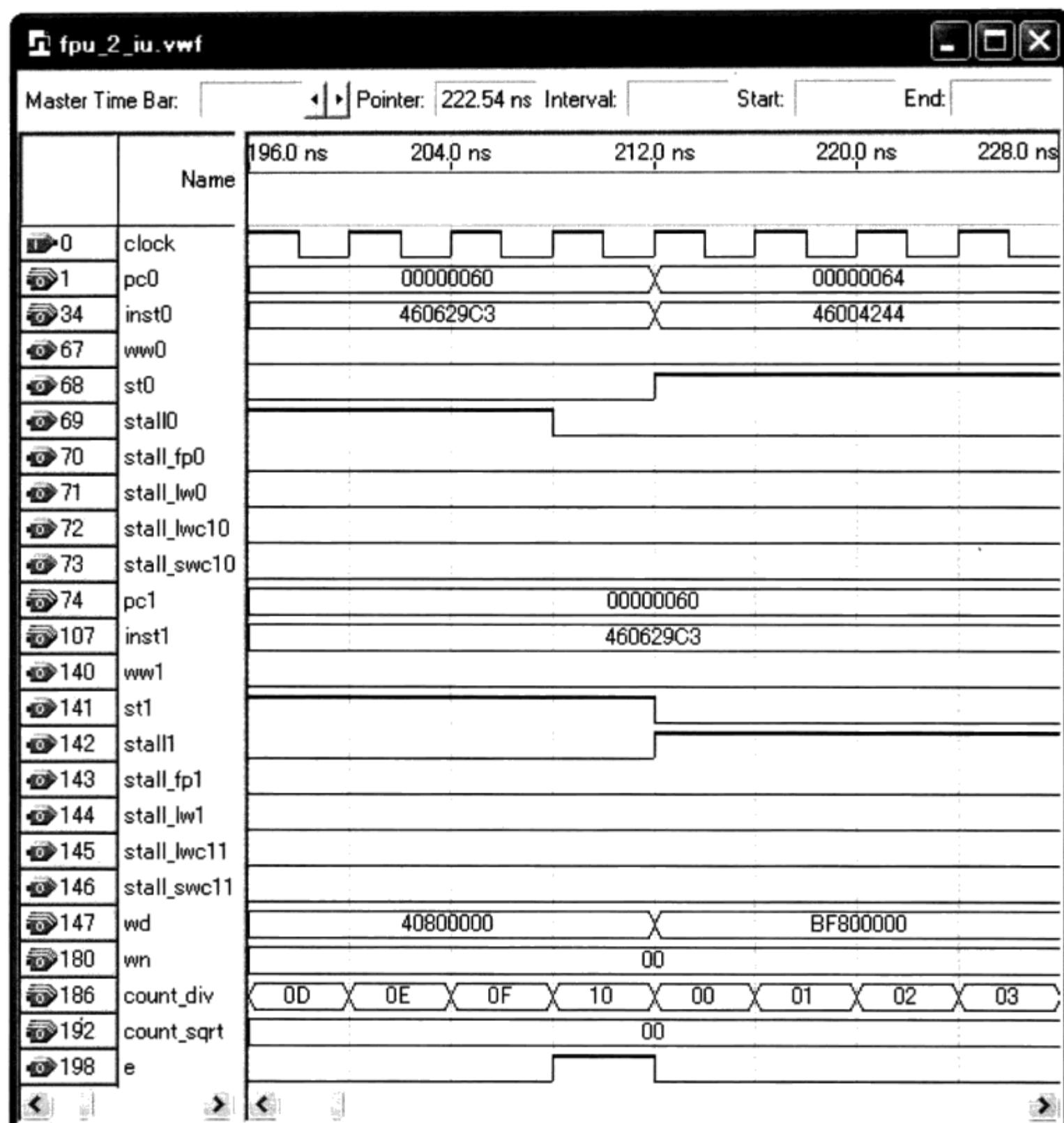
```

15 : C4060008; % (54)      lwc1    f6,  08(r0)  # load fp data %
16 : C408000C; % (58)      lwc1    f8,  12(r0)  # load fp data %
17 : 460629C3; % (5C)      div.s   f7,  f5, f6  # div %
18 : 46004244; % (60)      sqrt.s f9,  f8     # sqrt %

```

图 11.9 多线程 CPU 仿真波形图 (线程 0 浮点除开始)

图 11.9 示出的是线程 0 开始执行 div.s 指令时的波形。从 140ns ($pc0 = 0000005C$) 开始取 div.s $f7, f5, f6$ 指令 (IF); 从 144ns 开始译码并进行除法的 Newton-Raphson 迭代 (ID)。在译码期间, $stall0 = 1$ 、除法计数器开始从 0 计数。在 148ns 处, 线程 1 也对 div.s 指令译码, 但这时线程 0 正在使用浮点部件, 因此 $stall1 = 1$ 。



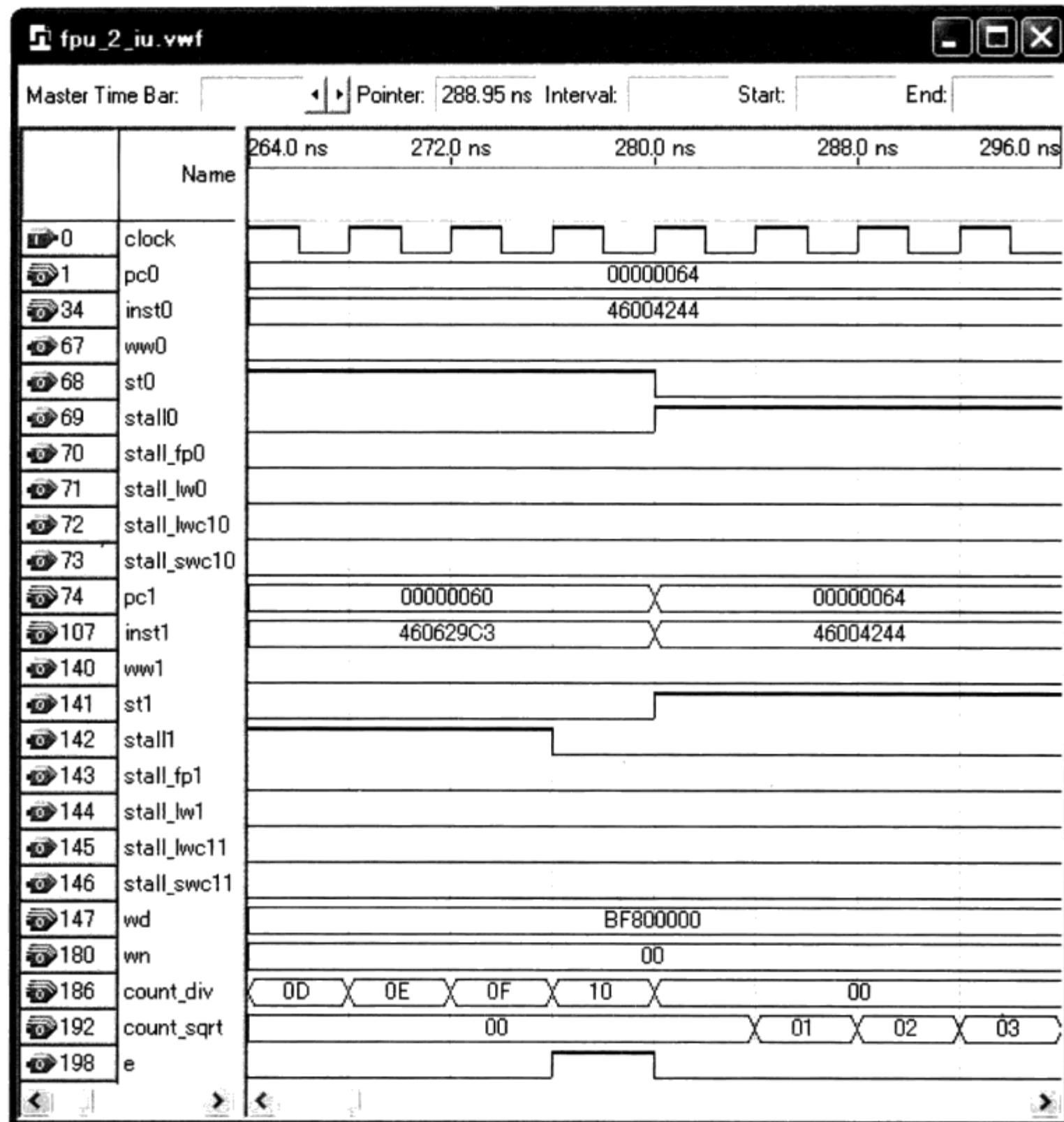
```

17 : 460629C3; % (5C)      div.s f7, f5, f6 # div %
18 : 46004244; % (60)      sqrt.s f9, f8      # sqrt %
19 : 46004A84; % (64)      sqrt.s f10, f9     # sqrt %

```

图 11.10 多线程 CPU 仿真波形图 (线程 1 浮点除开始)

图 11.10 示出的是浮点部件连续执行需要迭代的指令(除法指令或开方指令)时的情况。图中的 208ns 处, stall0 变 0, 导致使能信号 e 变 1, 进而结束线程 0 的 div.s 指令的 ID 级。从 212ns 处开始, 线程 1 进入 div.s 指令的 ID 级, stall1 = 1。与此同时, 线程 0 试图对 sqrt.s 指令进行译码, 但这时线程 1 正处在 div.s 指令的译码期间, 线程 0 必须等待 (st0 = 1)。



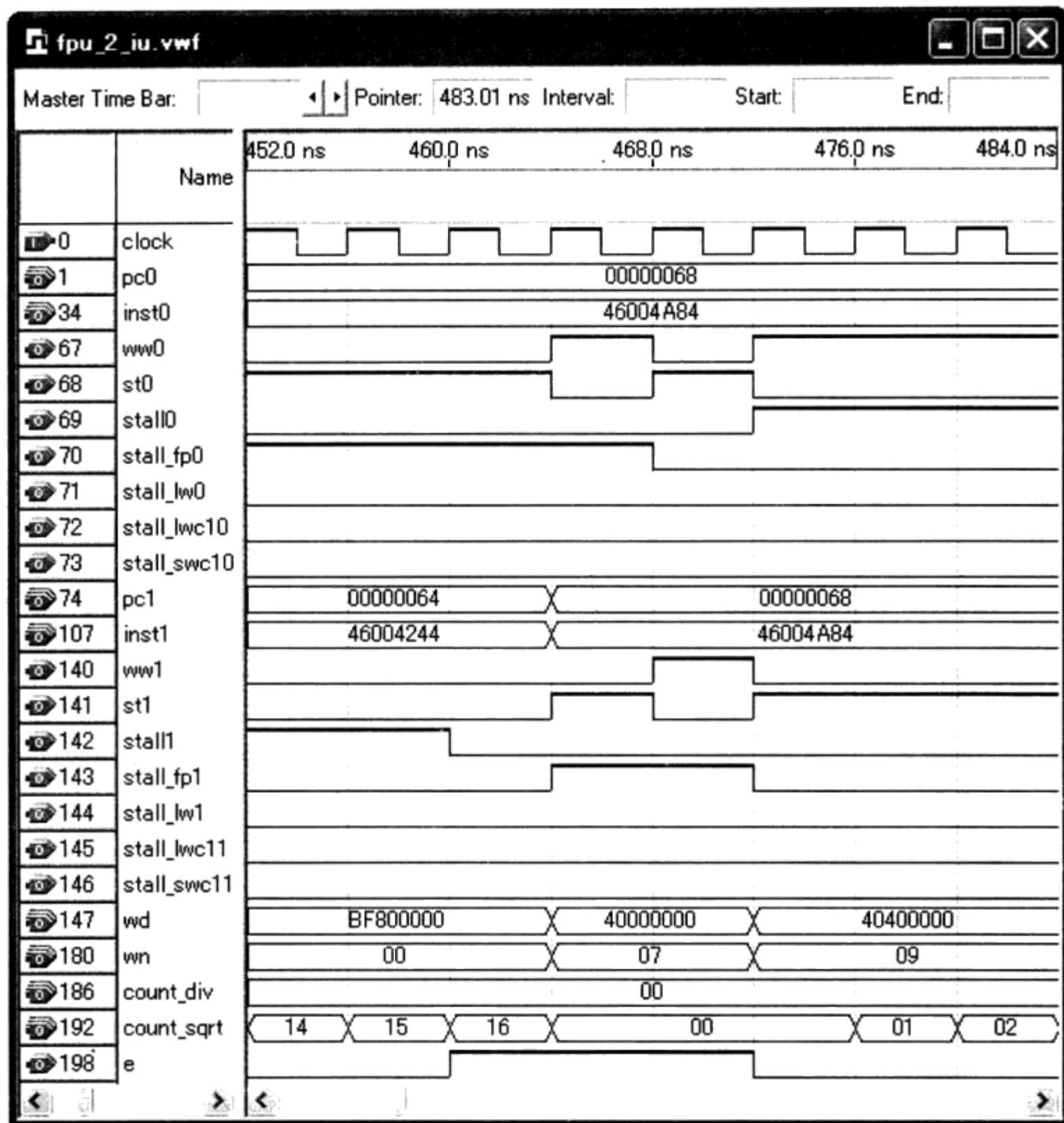
```

17 : 460629C3; % (5C)      div.s f7, f5, f6 # div %
18 : 46004244; % (60)      sqrt.s f9, f8      # sqrt %
19 : 46004A84; % (64)      sqrt.s f10, f9     # sqrt %

```

图 11.11 多线程 CPU 仿真波形图 (线程 0 浮点开方开始)

图 11.11 示出的也是浮点部件连续执行需要迭代的指令(除法指令或开方指令)时的情况。图中的 276ns 处, stall1 变 0, 导致使能信号 e 变 1, 进而结束线程 1 的 div.s 指令的 ID 级。从 280ns 处开始, 线程 0 进入 sqrt.s 指令的 ID 级, stall0 = 1。与此同时, 线程 1 试图对 sqrt.s 指令进行译码, 但这时线程 0 正处在 sqrt.s 指令的译码期间, 线程 1 必须等待(stall1 = 1)。



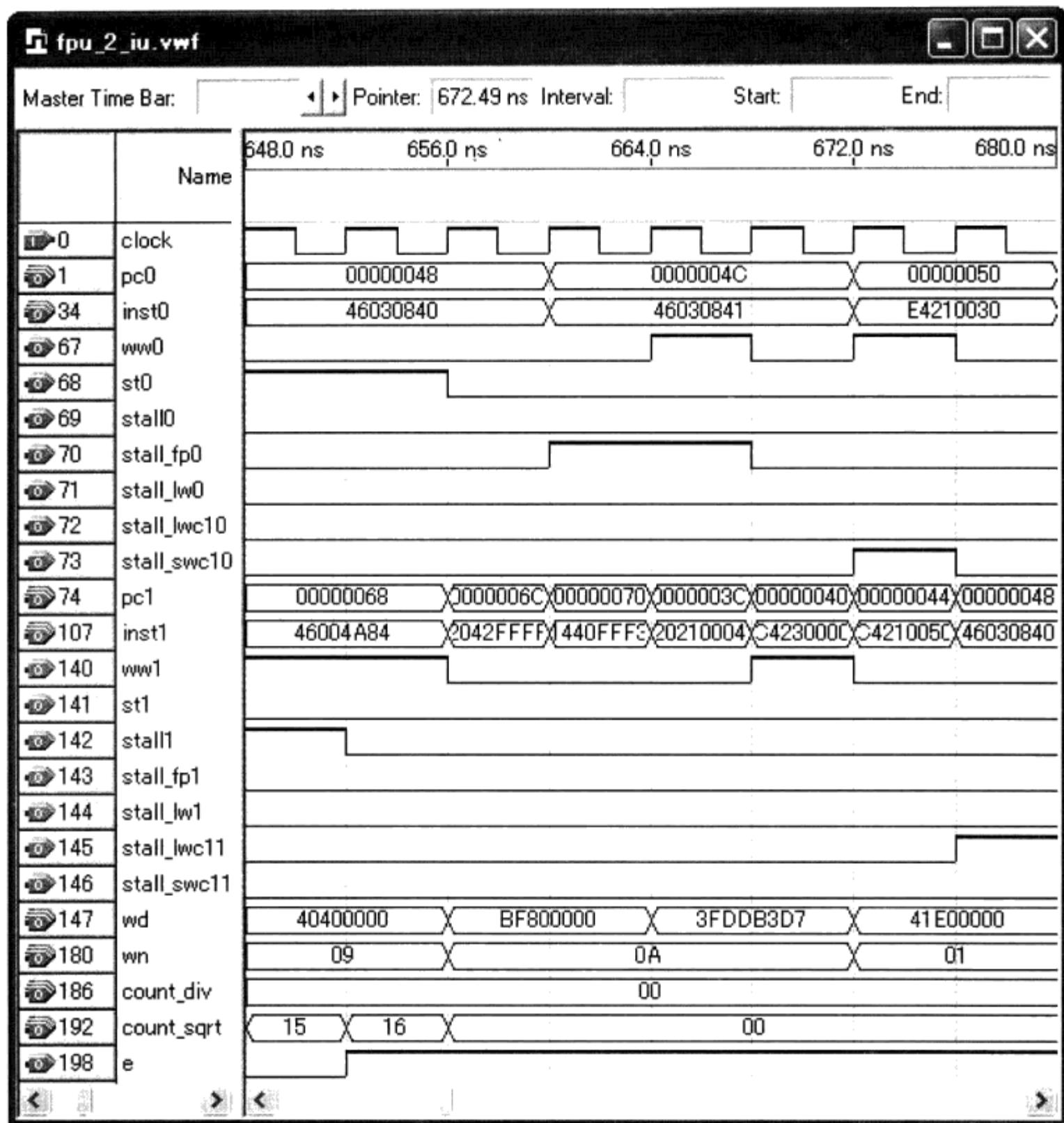
```

17 : 460629C3; % (5C)      div.s  f7,   f5,  f6  # div      %
18 : 46004244; % (60)      sqrt.s f9,   f8    # sqrt      %
19 : 46004A84; % (64)      sqrt.s f10,  f9    # sqrt      %

```

图 11.12 多线程 CPU 仿真波形图(保存浮点除法结果、线程 0 浮点开方开始)

图 11.12 示出的是线程 1 完成第一条 sqrt.s 的译码，然后线程 0 开始对第二条 sqrt.s 指令进行译码的波形。线程 1 也试图对第二条 sqrt.s 指令进行译码，但由于资源冲突，它必须等待 ($st1 = 1$)。由于第二条 sqrt.s 指令与第一条 sqrt.s 数据相关，因此 $stall_fp0 = 1$ 、 $stall_fp1 = 1$ 。从 464ns 开始，线程 0 和线程 1 相继把 div.s 指令的执行结果 (40000000) 写入各自的浮点寄存器 f7。



```

19 : 46004A84; % (64)      sqrt.s f10, f9      # sqrt      %
1A : 2042FFFF; % (68)      addi    r2, r2, -1   # counter - 1 %
1B : 1440FFF3; % (6C)      bne     r2, r0, 13   # finish? %

```

图 11.13 多线程 CPU 仿真波形图(保存浮点开方结果)

图 11.13 示出的是线程 1 保存第二条 sqrt.s 指令的执行结果前后的波形。当开方计数器值为十六进制的 16 时，ID 级将结束，然后经过 E1、E2、E3，线程 1 在 WB 级把浮点开方结果 (3FDDDB3D7) 写入浮点寄存器 f10。接下来的两个线程的指令都不是浮点除法或浮点开方指令，因此两个计数器的值均为 0、使能信号 e 为 1。其他指令的波形比较简单，就不列出了。

11.4 习题

1. 重新设计线程选择电路，使得线程 0 被执行的概率高于线程 1 被执行的概率，比如 $(5/8) : (3/8)$ 。
2. 重新设计 FPU 和线程控制部件：将 FPU 的加减、乘、除和开方分开：若各线程的操作不同，则可以并行执行，比如一个线程在做除法，而同时另一个线程在做开方。
3. 试设计一个 4 线程的 CPU。

第 12 章 存储器和虚拟存储器管理

总的来讲，存储器 (Memory)，特别是随机访问存储器 RAM (Random Access Memory)，是保存程序 (指令和数据) 的临时场所。当一个程序要投入运行时，操作系统把这个程序整个或部分地从硬盘调入到存储器，然后交由 CPU 去执行。CPU 执行程序时，要从存储器中取指令。存储器访问指令，比如 `lw` 或 `sw`，还要从存储器取数据或把数据写入存储器。因为存储器的速度比 CPU 慢很多，在 CPU 内部都设计有高速缓冲存储器 (以下简称 Cache)。

另外，所有编译好的程序都使用它自己的虚拟地址空间。一般来讲，虚拟地址空间都是从 0 开始。如果访问存储器时直接使用这个虚拟地址，则会造成不同的程序之间相互冲突。因此，当一个程序被调入到存储器时，还需要根据当前存储器的使用情况，为它安排空闲的存储器。这就需要有一个机制，把程序执行时的虚拟地址转换成实际的存储器地址。为了加速地址转换，CPU 内部通常设计有 TLB (Translation Lookaside Buffer)，它有与 Cache 非常类似的结构。

本章主要讨论各种存储器的原理、Cache 的结构与设计、虚拟存储器管理、用于快速地址转换的 TLB 以及 MIPS CPU 的基于 TLB 的地址转换机制。

12.1 存储器

我们知道，存储器是构成计算机的三剑客之一 (另外的两剑客是 CPU 和 I/O 接口)。存储器种类繁多，依用途或内部结构不同，我们把它大致分成以下 4 类。

- 1) 静态存储器 (Static Random Access Memory, SRAM)，主要用于 Cache 和 TLB 设计，有钱人用它来实现计算机主存；
- 2) 动态存储器 (Dynamic Random Access Memory, DRAM)，用于实现主存；
- 3) 只读存储器 (Read-Only Memory, ROM)，用于存放初始启动程序 (固化)；
- 4) 相联存储器 (Content Addressable Memory, CAM)，用于 Cache 和 TLB 设计。

除了 ROM，其他三种存储器都具有所谓的“挥发性”，意即电源关掉后，原来保存在存储器中的内容便消失得无影无踪。因此，刚开机时，RAM 中的内容是不可使用的。那么，开机时 CPU 执行的第一条指令从何而来？答案是从 ROM 中来。本节简要地描述这 4 种存储器的原理和结构。

12.1.1 静态存储器 (SRAM)

静态存储器是相对于动态存储器而言的，或者说动态存储器是相对于静态存储器而言的。简单地讲，动态存储器需要“输氧”，也就是所谓的刷新 (Refresh)。否则的话，动态存储器中的内容会渐渐消失。而静态存储器很“健康”，不需要刷新。大家应该还记得 D 锁存器吧。静态存储器的一位就类似于一个 D 触发器。

虽然没有谁这么干，但还是让我们来看看如何用 D 触发器来构成一个小容量的静态存储器^[26]。图 12.1 示出的是用 12 个 D 触发器设计的一个存储器模块。它一共有 4 个字，每个字 3 位。地址是 A[1:0]、数据输入端是 DI[2:0]、输出端是 DO[2:0]、WE 是写使能。数据输入端接到 D 触发器的 D 端。左侧是地址译码，用于选择 4 个字中的一个。译码的输出和 WE 相与 (AND)，接到 D 触发器的允许端 C。D 触发器的输出 Q 和地址译码的输出相与再送到一个或门，形成数据输出信号。

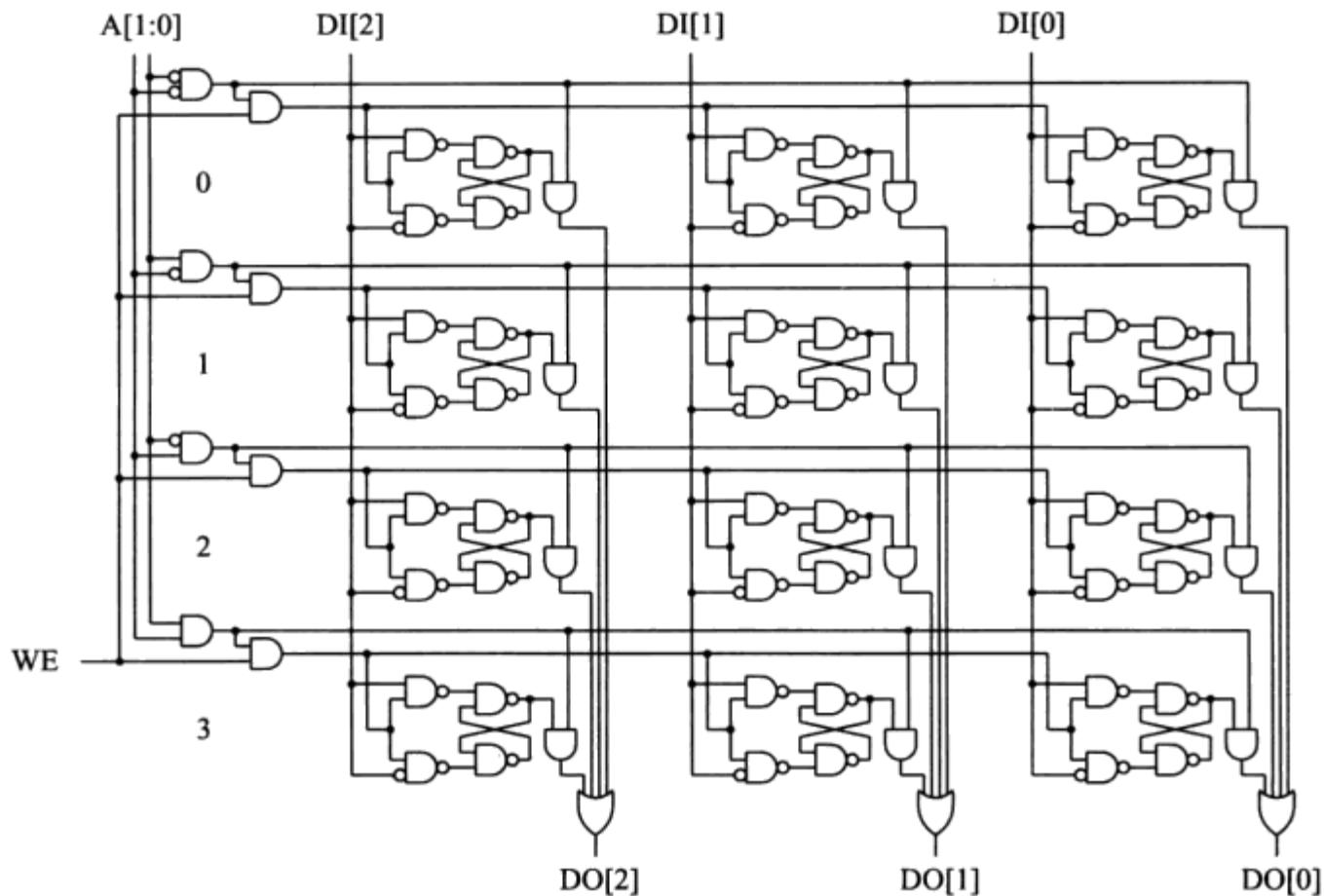


图 12.1 由 D 触发器构成的 4×3 位静态存储器 (供演示用)

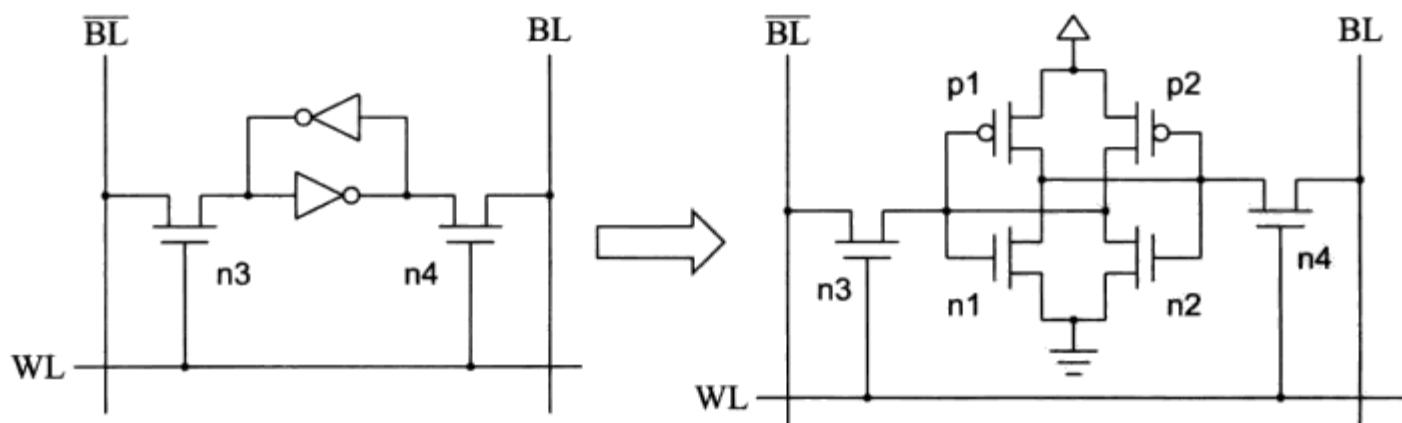


图 12.2 一种 6 晶体管静态存储器 SRAM 的存储单元

实际的静态存储器单元不是使用 D 触发器，而是使用类似于图 12.2 所示的电路。左侧是简化的电路图。两个非门构成一个双稳态的存储单元；BL 和 \overline{BL} 是数据线；WL 是一个字的选择信号，相当于 12.1 的地址译码输出。它控制 n3 和 n4 两个

晶体管，当它为高电平时，可对存储单元进行读写。右侧是详细的 CMOS 电路，p1 和 n1 组成一个非门；p2 和 n2 组成另一个非门。

静态存储器的特点是与 CPU 的接口简单且速度快，但价格高，耗电量也大。因此一般用于 Cache 和 TLB 设计，但有一些高性能计算机也拿它当主存用。

12.1.2 动态存储器 (DRAM)

与静态存储器的存储单元不同，动态存储器使用一个小容量的电容来保存信息，用电容中有无电荷(电平的高低)来表示 1 和 0。图 12.3 是一个 $1M \times 1$ 位 DRAM 芯片的内部结构示意图。“ $\times 1$ ”是指数据线外部接口的位数。 $1M$ 个单元(位)组成 $1K \times 1K$ 的二维阵列。10 位行地址译码器的输出用于选择阵列的一行 ($1K$ 位)。列地址译码器/多路选择器从一行的 $1K$ 位中选择一位。访问 $1M$ 个单元需要 20 位地址，但一般的 DRAM 芯片的地址线只有所需地址位数的一半。因此地址要分两次送给 DRAM 芯片，低电平有效的 RAS (Row Address Strobe) 和 CAS (Column Address Strobe) 分别用于选通行地址和列地址。

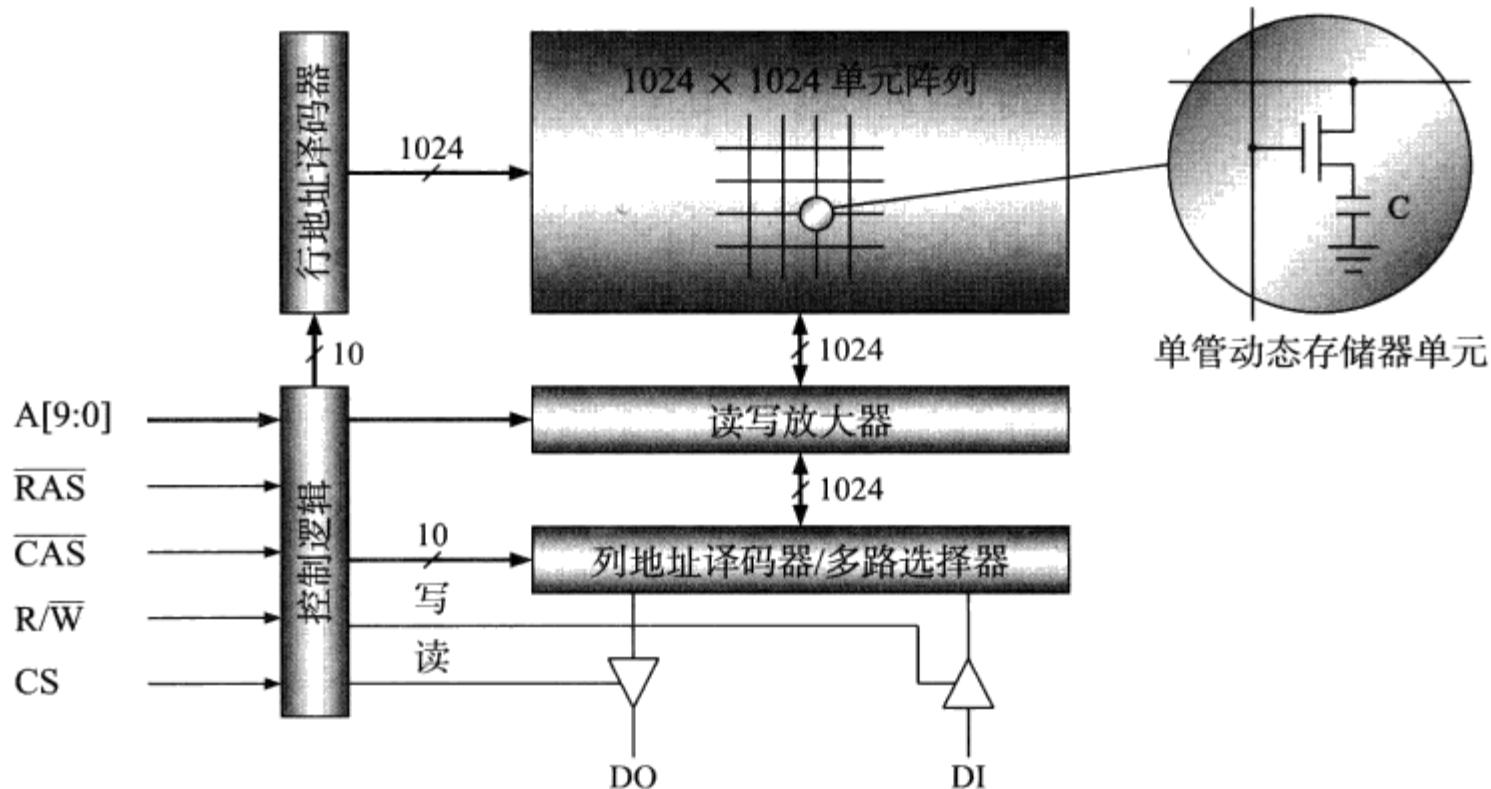


图 12.3 动态存储器 $1M \times 1$ 位 DRAM

图中的一位数据线有单独的输入和输出管脚 (DI 和 DO)，也有共用一个管脚的(双向数据线)。数据位数也有多于一位的。另外，电容会漏电，所以 DRAM 需要定期刷新。刷新不是一位一位地进行，而是一行一行地进行。

12.1.3 只读存储器 (ROM)

只读存储器也是可以随机访问的。与 RAM 不同，ROM 存储器的内容即使断了电也不会丢失。计算机系统中都有 ROM，用于存放操作系统的初始引导程序等。

ROM 有很多种。常见的有 MROM (Mask ROM)、PROM (Programmable ROM)、EPROM (Erasable PROM)、EEPROM (Electrically EPROM) 和 Flash Memory 等。Flash Memory 与一般的存储器不同，它不支持传统意义上的随机访问。与其说 Flash Memory 是一种存储器，倒不如说是一种类似于硬盘的外部存储设备。

12.1.4 相联存储器 (CAM)

还有一种比较特殊的存储器 (Content Addressable Memory, CAM)，按英文字面可译成“可按内容寻址的存储器”，但读起来不怎么顺。本书称其为相联存储器。不管名称如何，关键是要看它的内涵。

传统存储器的读操作是：给个存储器地址，相应的存储器数据就出来了。对相联存储器可以这样简单地理解：它与传统存储器刚好相反，给个存储器数据，该数据在存储器中的位置 (地址) 就出来了。当然，如果那个数据不在存储器中，也不可能有地址出来。即，相联存储器查找存储器中所有的内容，看看是否有一个或多个与输入数据匹配的单元。如果有，在哪儿？

图 12.4 是一个 4 字 3 位的相联存储器 CAM 的结构示意图。12 个 C 是 12 位存储器数据，3 位输入数据的每一位都有正反两个信号， SL_i 和 \overline{SL}_i ， $0 \leq i \leq 2$ ，接到所有字的相应的 C 位 (列)。SL 是 Search Line 的缩写。当某个字的 3 位 (行) 均与输入数据相同时，其 ML 信号为高电平。ML 是 Match Line 的缩写。如果有多个字匹配，那么就有多条 ML 线输出高电平。右边是一个优先级编码器，从中选出一个地址 Matched_Address。信号 Match_Found 表示是否有匹配。在有些电路的设计中可能并不需要这个编码器，而是直接输出全部的 ML。

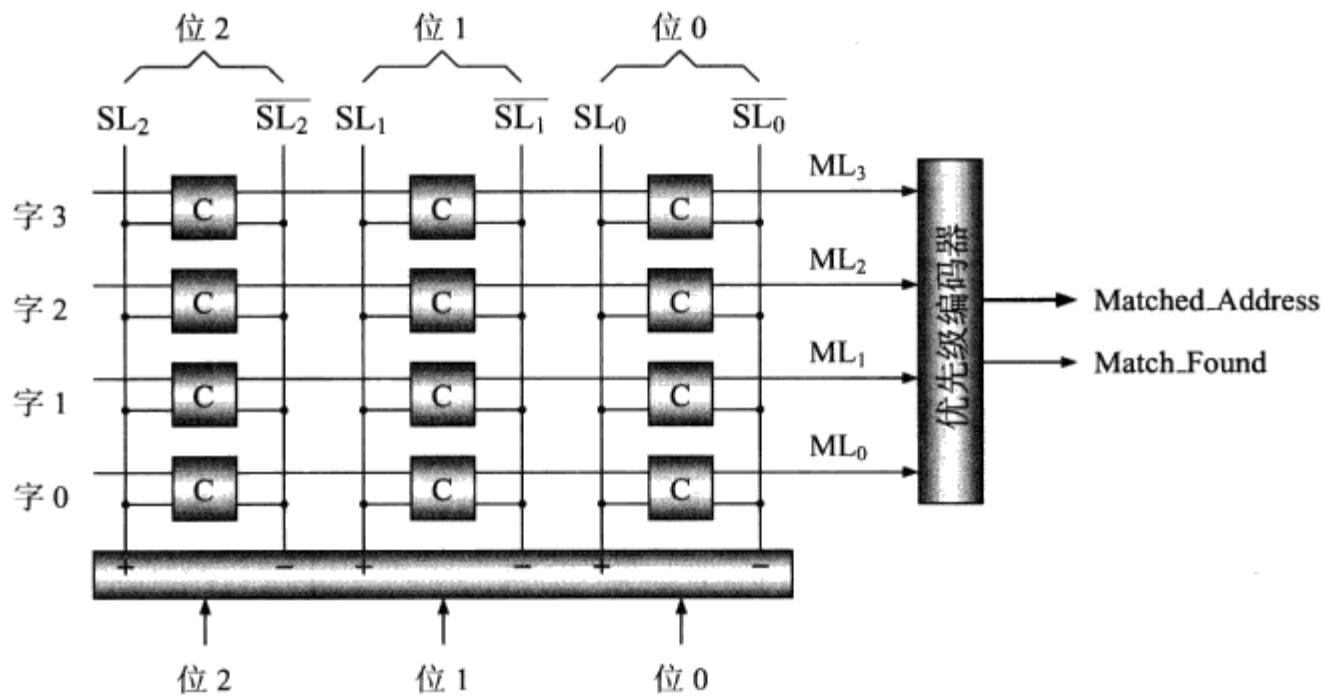


图 12.4 相联存储器 CAM 的结构图

那么存储单元 C 到底是一种怎样的结构，这么厉害，能实现匹配呢？这样的电路有许多种实现方法^[25]，图 12.5 给出的是一种非常朴素的电路。图中的存储单元部

分与 SRAM (见图 12.2) 相同, 它的输出用 D 和 \bar{D} 表示。不匹配时, \bar{D} 与 SL 相同, D 与 \bar{SL} 也相同, 两组晶体管 (n_1 和 n_2 一组、 n_3 和 n_4 一组) 总有一组导通, 导致 ML 变低。反之, 每组中都有一个晶体管截止, 维持 ML 的高电平 (匹配)。ML 连接到多位这样的存储单元, 只要有一位不匹配, ML 就被拉低。与图 12.4 相比, 图 12.5 中多出了 WL, 它和 SRAM 中的 WL 相同, 用于 SL 写入时的行选择。

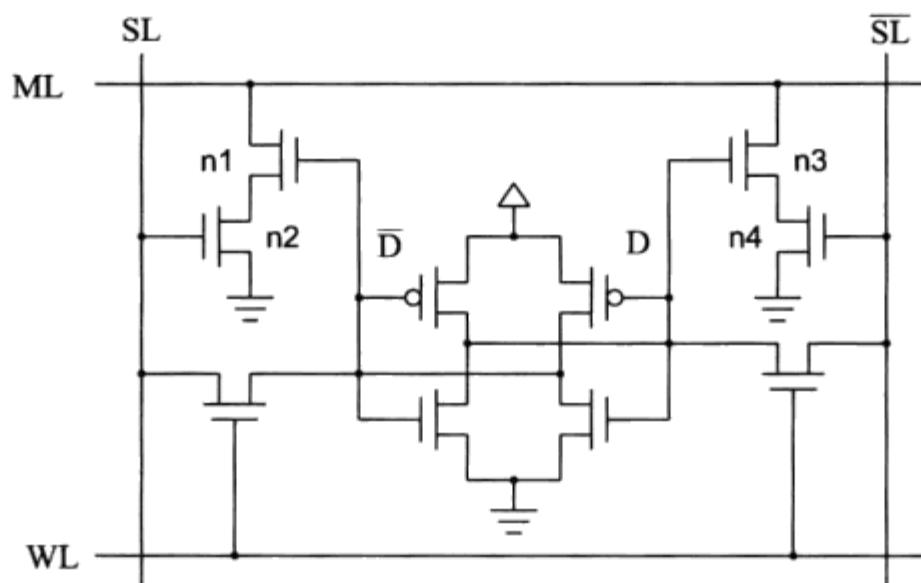


图 12.5 一种相联存储器 CAM 的存储单元

相联存储器可以用于各种有按位匹配要求的场合, 比如虚拟存储器管理、数据压缩、Cache、TCP/IP 中的 IP 地址匹配等。图 12.6 示出的是相联存储器的应用之一: Cache 的简单结构图。图中的左侧用存储器地址去匹配 CAM 相联存储器。如果有匹配发现, 其匹配地址被用来访问快速的 RAM, 从中得到指令或数据。图 12.6 是使用独立的 CAM 和 RAM 芯片设计 Cache 的例子。如果把它们集成在一个芯片中, 则可以省去编码器和译码器, 把 CAM 匹配线直接和 RAM 行选择线相连。但这时不允许有多个匹配同时出现。

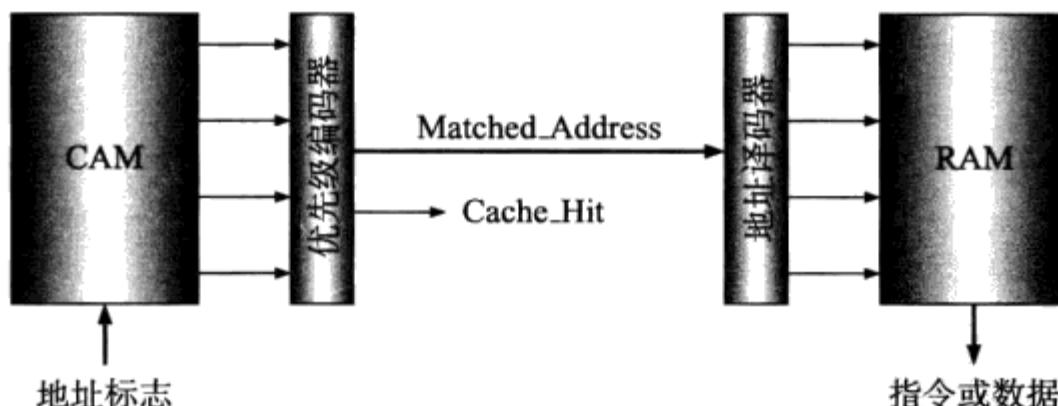


图 12.6 使用相联存储器的 Cache 示意图

12.1.5 存储层次

图 12.7 示出了目前计算机系统典型的存储层次结构。从左至右, 存储容量增大, 但速度变慢。使用这样一个存储层次的目的是为程序提供一个容量大、速度快且价格低的存储系统。

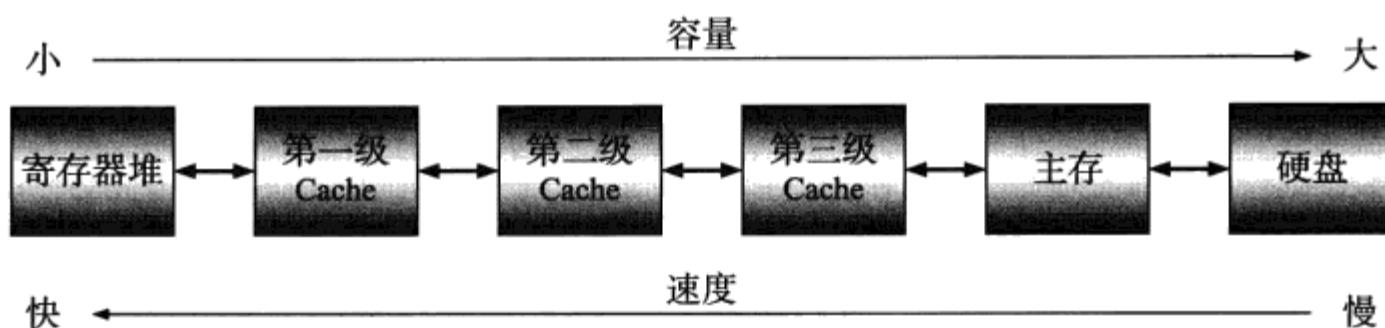


图 12.7 存储层次

寄存器堆是速度最快的存储部件，它在 CPU 芯片内部。CPU 使用指令中的寄存器号直接访问它。一般的 CPU 有 16 ~ 256 个寄存器。Cache 存放主存中的部分数据。第一级 Cache 和第二级 Cache 通常也在 CPU 片内 (On-Chip Cache)，而且第一级有分开的指令 Cache 和数据 Cache。第三级 Cache 在片外，使用 SRAM。Cache 对用户程序来讲是透明的，即用户看不到它，没有办法用程序对它进行控制或访问。在有些计算机的体系结构中，Cache 对操作系统也是透明的；而有些体系结构则提供了特权指令对 Cache 直接进行维护。主存 (Main Memory) 是用户能访问的存储器，存放指令和数据。硬盘的主要功能是存放文件，但它的另一功能也很重要，即与主存一起为用户提供一个相对大的虚拟存储器空间。如果我们把主存称为第一级存储器，则硬盘是第二级存储器。

12.2 高速缓存 (Cache)

Cache 的原意是一个隐蔽的场所，用来保存食物等贵重的东西。在计算机系统中，借用 Cache 来表示一个小容量高速度的缓冲区，用于存放 CPU 经常使用的原本在主存中的指令或数据以加快 CPU 访问存储器的速度。

前面我们已经讲过，Cache 对用户程序来讲是透明的，即用户看不到它。为什么呢？因为 Cache 是一个隐蔽的场所，让用户看到它就谈不上隐蔽了。实际的意义是：对 Cache 的管理是我们硬件的内政，不容你们软件干涉。

最初，Cache 中什么也没有。所以 CPU 必须访问主存。从主存拿到的指令或数据由硬件写入 Cache 中，以便 CPU 以后再访问相同的地址时，不用访问主存，直接从 Cache 中得到。

假设 Cache 的内容写入后就再也没有被 CPU 使用过，那么这个 Cache 就没有任何正面的意义。使用 Cache 是基于指令和数据访问的局部性 (Locality) 特性。局部性分空间局部性 (Spatial Locality) 和时间局部性 (Temporal Locality)。空间局部性是指当 CPU 访问某个存储单元时，该存储单元附近的存储单元最有可能被随后访问；时间局部性是指 CPU 访问某个存储单元后，该存储单元最有可能被再次访问。

如果 CPU 想要的指令或数据在 Cache 中找到了，我们说 Cache 命中 (Hit)。如果没命中 (Miss)，CPU 必须要访问主存。假设 Cache 的访问时间 $t_c = 1\text{ns}$ 、主存的访问时间 $t_m = 50\text{ns}$ 、Cache 的命中率 $h = 98\%$ ，则平均的存储器访问时间是

$$\begin{aligned}
 t &= h \times t_c + (1 - h) \times (t_c + t_m) \\
 &= t_c + (1 - h) \times t_m \\
 &= 1 + 0.02 \times 50 = 1 + 1 = 2\text{ns}
 \end{aligned}$$

Cache 的容量比主存小得多。那么主存的指令或数据放在 Cache 的什么地方？再者，当 Cache 已满，“新来的”要替换掉谁？如果往存储器写入数据时 Cache 没命中怎么办？以下各小节回答这些问题。

12.2.1 Cache 的映像机制

Cache 映像机制定义数据在 Cache 中存放的规则。主要的映像机制有三种：

- 1) 直接映像 (Direct Mapping);
- 2) 全相联映像 (Fully Associative Mapping);
- 3) 组相联映像 (Set Associative Mapping)。

1. 直接映像

直接映像使用地址的低位作为 Cache 存储器的地址直接访问 Cache。那么地址的高位怎么办？不能简单地忽略，否则会把不是属于你的东西拿来了。因此我们要在 Cache 中保存自己的高位地址作为一个标志 (Tag)。考虑到存储器访问的空间局部性的特点以及减少标志所占的存储单元的数量，Cache 通常是以块 (Block) 或行 (Line) 为单位与主存之间交换数据。这样，每个 Cache 块除了有数据，还要配备一个标志 Tag (地址的高位)。如果把低位地址看作“名”，那么标志就是“姓”，姓和名合起来就是姓名，能确定 Cache 中的东西到底是谁的。

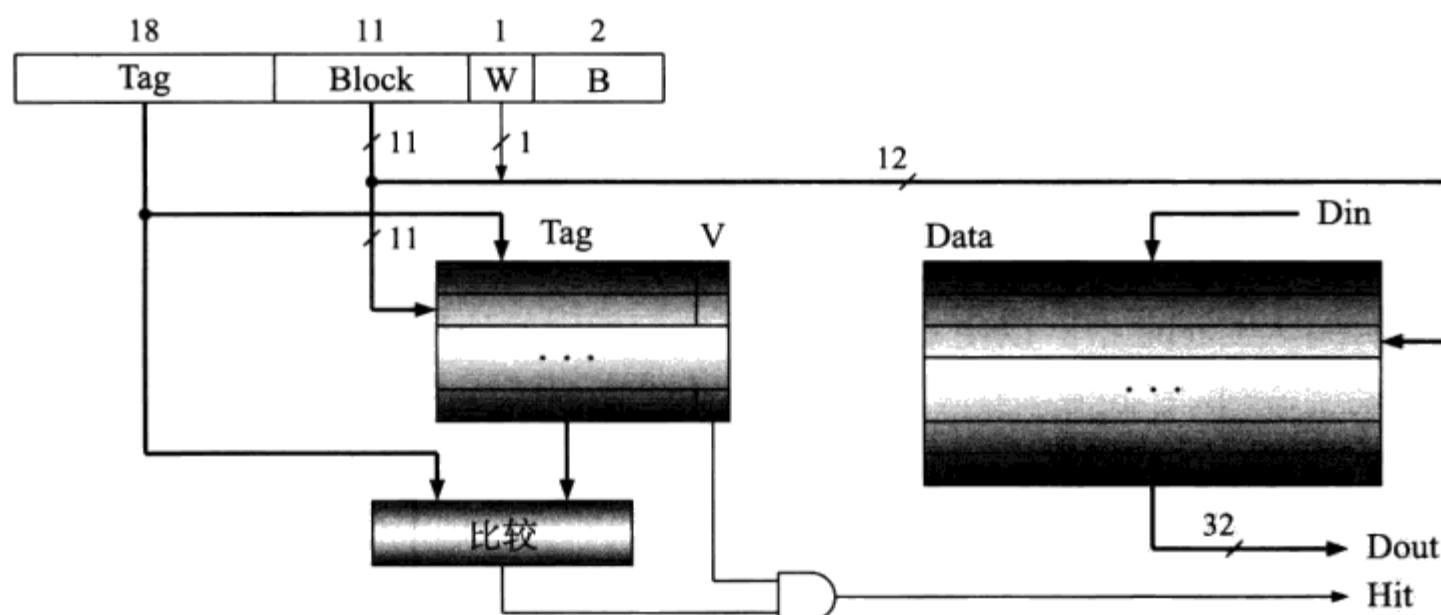


图 12.8 直接映像 Cache

图 12.8 给出的是 16KB (16×2^{10} 字节) 直接映像 Cache 的硬件结构。图中的地址是 32 位，其中的 Block 是 Cache 块号，或称块地址，W 是块内的字地址。一个字

有 4 个字节，B 是字节地址。由于 Cache 有 $16KB = 2^{14}$ 字节，需要地址的低 14 位作为 Cache 的地址。从图中可以看出，每块为 $8 = 2^3$ 个字节，或两个字。因此访问 Cache Tag 部分的地址为 $14 - 3 = 11$ 位(块地址)。而访问 Cache Data 部分的地址为 12 位(字地址)，输出的数据是 32 位(一个字)。另外图中的 V(Valid) 是相应块的有效位。当地址中的标志与 Cache 中的标志相等、并且有效位也为 1，我们说 Cache 命中(Hit = 1)。注意，图中数据部分的 Cache 存储器可以直接使用地址中的 W 位，而不用在外面加一个二选一多路器来选择一块(两个字)中的一个字。

2. 全相联映像

图 12.9 示出的是 16KB 全相联映像 Cache 的硬件结构。直接映像 Cache 的特点是存储器的数据在 Cache 中存放的地点是唯一的，一点迁徙的“自由”也没有。与此相对照，全相联映像 Cache 有最高的“自由度”，愿意住哪儿就住哪儿。多么和谐的社会啊。但直接映像 Cache 使用普通的静态存储器 SRAM 就行，而全相联映像 Cache 需要使用相联存储器 CAM。看来要自由是要付出代价的。

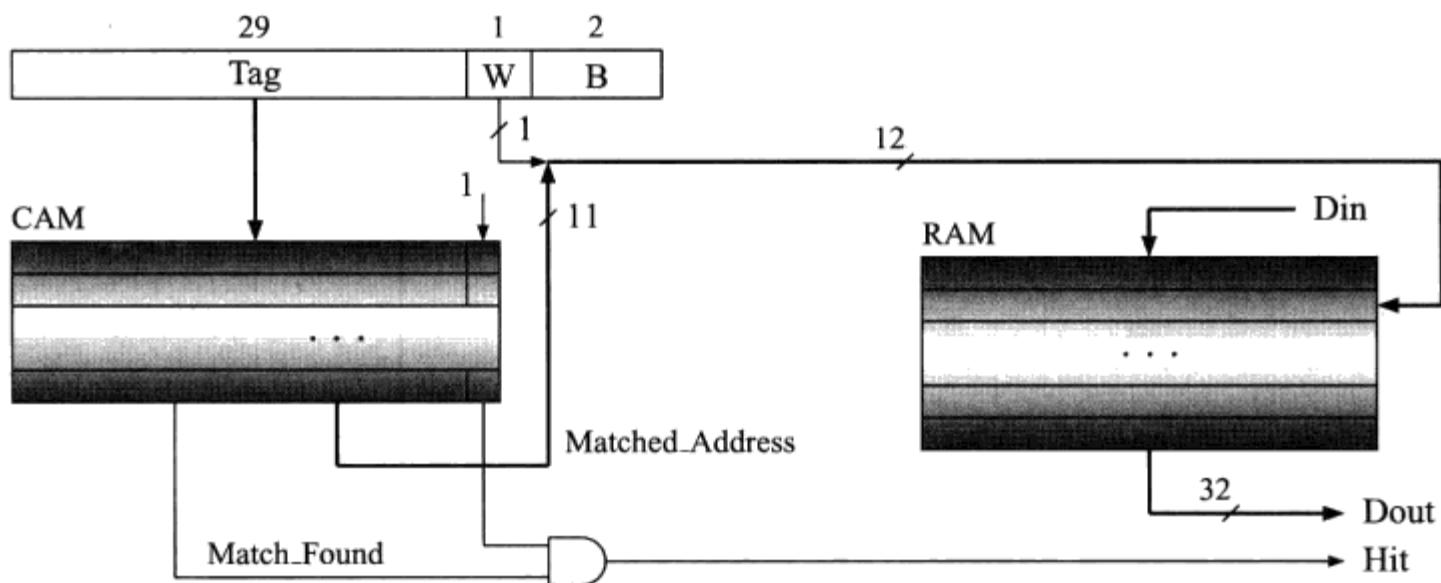


图 12.9 全相联映像 Cache

图中地址标志的比较电路使用 CAM。有关 CAM 的原理及内部单元的 CMOS 电路我们已经在本章开始处介绍了。若标志匹配，CAM 输出 11 位的匹配地址 Matched_Address，它与地址中的一位 W 一起，构成地址访问 Cache 数据部分的存储器 RAM。如前所述，CAM 和 RAM 可以集成在一起，省去地址编码器和地址译码器，用 CAM 匹配线直接选中 RAM 的一行(块)，再用 W 选出一个字。

3. 组相联映像

在“自由度”这个问题上，我们还是可以商量的。直接映像和全相联映像是两个极端，而组相联映像在二者之间搞“中庸”。尽管同样是中庸，但也有偏向谁的问题，其衡量指标是“路”(Way)的多少。二路组相联是把直接映像的 Cache 分成两路，每路的规模是总数的一半。存储器数据存入 Cache 时，用地址的低位对两路同

时访问，看看有没有一路空闲。如果有，先住进去再说。读 Cache 时也是一样，用地址的低位对两路同时访问，看看有没有一路命中。

二路组相联是最偏向直接映像的“中庸”，也就是随便应付一下。如果你觉得怠慢了全相联映像，你可以用“四路”、“八路”、“十六路”等。当路数增至与 Cache 总块数相同，就变成全相联映像了。你看，由量变到质变了不是。

图 12.10 示出的是 16KB 二路组相联映像 Cache 的硬件结构。Tag 部分一半一半分成两路，使用块地址同时访问这两路，因此块地址只要 10 位就够了（比直接映像少了一位，但标志多了一位）。同时被选中的两路合在一起称为一组（Set）。组内可以任意存放，即，选择一组时用直接映像，组内用全相联。访问 Cache 的数据部分仍用 12 位地址，其中 10 位来自于块地址 Block，另两位分别来自于 W 和 Hit1。

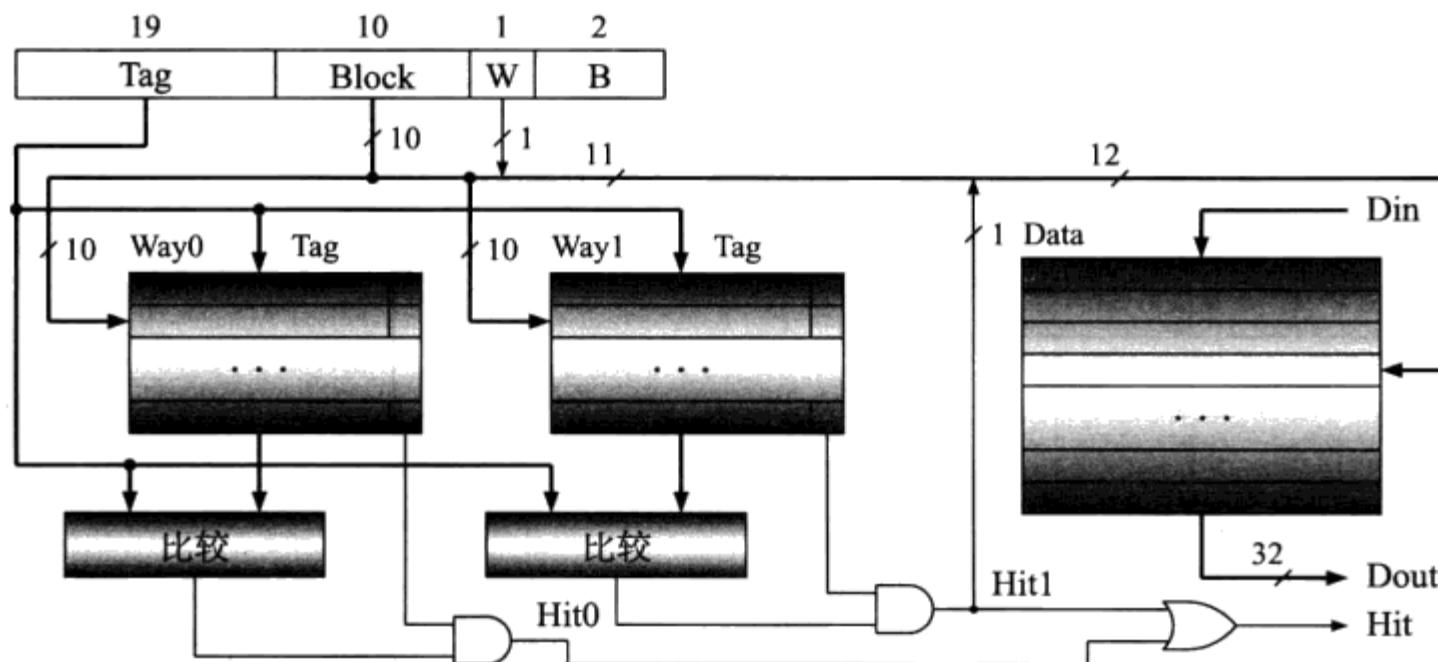


图 12.10 二路组相联映像 Cache

除了以上讲的三种映像，还有一些不太常用的其他映像方式，比如“扇区映像”（Sector Mapping）。映像方式不同，直接导致 Cache 的性能不同。我们对 Cache 的性能进行评价时，不但要看它们的命中率，也要看价格及访问速度。

12.2.2 Cache 块的替换算法

以全相联 Cache 为例。当一块数据被存入 Cache 时，首先查找有没有空闲的 Cache 块。如果有，就把数据存入该块。如果全相联 Cache 所有的块均被存入了数据，那么再有新的块又要存入 Cache 时怎么办呢？答案是只好把原来已在 Cache 中的块替换掉了。替换谁呢？这倒是一个伤脑筋的问题。同样，对于组相联 Cache 来讲，由于组内是全相联，也有替换掉哪一路的问题。直接映像的 Cache 有没有这个问题呢？没有，因为是直接映像，没有其他的选择。所以直接映像的 Cache 不用伤这个脑筋。

为了提高 Cache 的命中率，我们希望把“将来”不再被使用或很久很久以后才被使用的 Cache 块替换掉，而把那些近期将要被使用的块保留。但是，希望归希

望，“将来”的事情谁也说不准。

使用 Cache 的目的是缩短存储器的平均访问时间，因此 Cache 块的替换算法一般由硬件实现。以下我们介绍两种相对简单的替换策略：LRU 算法和随机算法。

1. LRU 替换算法

LRU (Least Recently Used) 是使用最为广泛的一种替换算法。LRU 的意思是“最近最少被使用”，即 LRU 替换算法不是“展望未来”，而是“回顾过去”。就像新招一个队员同时让一个“板凳队员”卷铺盖走人，实现 LRU 算法要求对每个“队员”做记录，看谁在板凳上坐的时间最长。

以组相联映像 Cache 为例。假设一组有八路，则每路要有一个三位计数器，通过各路的计数值来区分哪一路最近老没被教练派上场。规则如下：

- 1) 替换掉计数值为 0 的块 (如果有多个块的计数值为 0，随便替换哪一个)。
设置该块的计数器为最大值 7，其他块的计数器都减 1。但若计数值是 0 就不再减了。我们称这样的计数器为饱和计数器 (Saturated Counter)。
- 2) 命中时，假设命中块的计数值为 k ，则把计数值大于 k 的所有其他块的计数器减 1，并把命中块的计数器设置为最大值 7。
- 3) 计算机刚启动时，把所有的 Cache 块的计数器和有效位全部清零。这意味着该队连一个队员也还没有呢。

如果一组有四路，则需为每路安排一个两位计数器。如果一组只有两路，按推理需为每路安排一个一位计数器。但是，如果其中的一路是最近最少被使用的话，不言而喻，另一路一定是最近最多被使用的，因此只用一个一位计数器就足够了。

以上讨论的是一组中的情况。如果 Cache 总的组数为 S ，则总的计数器的数量是一组中计数器的个数乘以 S 。这是一笔不小的开销。

2. 随机替换算法

LRU 算法是通过总结历史经验来预测将来的发展方向。这种算法需要“投资”，即为每一块配备一个计数器。

如果你想省钱，随机 (Random) 算法倒是一个相当不错的选择。随机算法完全不管 Cache 块过去、现在及将来的使用情况，简单地根据一个随机数，选择一块替换掉。注意，整个 Cache 只需一个随机数产生器，所以比 LRU 省钱。随机数可由硬件产生，例如设置一个计数器，由系统时钟进行计数。是不是被替换掉就看你的运气了。虽然有些 Cache 设计者讨厌这种方法，但模拟结果证明，它的性能还是相当不错的，至少把钱先省了。

还有一种先进先出 FIFO (First-In First-Out) 算法：不管你在场上的表现如何，谁先入队谁先走人。与 LRU 算法一样，也需投资。虽然还有其他一些替换算法，但实现起来过于复杂。在大多数 Cache 设计中，二路和四路组相联映像 Cache 用 LRU 替换算法，其他的用随机替换算法 (直接映像 Cache 用不着替换算法)。

12.2.3 Cache 写策略

截至目前，我们都是在讨论读存储器时如何对 Cache 进行管理。那么写存储器时又是怎样的一种情况呢？看以下的一些 Cache 写策略。

1. 写透和写回

写存储器并且 Cache 命中时，有以下两种策略可供选择：写透 (Write Through) 和写回 (Write Back 或 Copy Back)。写透策略是指在写 Cache 的同时也写主存，也可译成写穿、写通、通写或统写 (统统写)，总之是 Cache 和主存“通吃”。它的优点是能够保持主存与 Cache 的一致性；缺点是增加数据传输量，并且写存储器要花费较长的时间 (可以把数据先放在快速的写缓冲区内，然后再由硬件慢慢写)。

写回策略是只写 Cache，不写主存。只有当数据块要被替换掉时，才将它写回主存。如果被替换掉的 Cache 块从来没被写过，即它的内容与主存相应块的内容是一样的，就不必写回主存了。因此，为了能够区分 Cache 块是否被写过，我们需要为每一块增加一位“修改位”(Updated Bit 或 Dirty Bit)。

当数据块首次被调入 Cache 时，清除修改位。一旦往数据块写数据时，把修改位置 1。在数据块被替换掉时，若它的修改位为 0，则简单地把新的数据写入该块；如果修改位是 1，则先要把数据块写回主存，然后再调入新的数据块。这种策略的优点是缩短了写操作所用的时间，减少了存储器访问量；缺点是主存中可能会存有过时的数据。当其他 CPU 或 DMA 控制器从存储器中读数据时，有可能读不到最新的数据，即出现所谓的 Cache 一致性 (Cache Coherence) 问题。

2. 写前读入和写不读入

写存储器但 Cache 不命中时，也有两种写策略：写前读入 (Write Allocate) 和写不读入 (No Write Allocate 或 Write No Allocate)。写前读入策略是先把数据块从主存读入 Cache，然后再写。为什么不直接写 Cache 而是先读再写呢？因为 Cache 块是一“大块”，要写入的只是一“小块”，地址标志是为整个“大块”而准备的，因此必须把整块内容读进来。写不读入策略是绕开 Cache，只写主存。一般地，写前读入与写回策略一起使用，写不读入和写透策略又是一对儿。为什么呢？

12.2.4 数据 Cache 电路设计及 Verilog HDL 代码

图 12.11 是 Cache 与 CPU 及存储器之间的一般连接示意图。所有的信号分成两组：与 CPU 连接的信号名称以 p_ 开始；与存储器连接的信号名称以 m_ 开始。信号 a 是地址线。dout 和 din 是数据线。strobe 是选通线，为 1 时表示要进行读或写操作。rw 为 0 时表示读，为 1 写。ready 为 1 时表示已经准备好了：m_ready 表示存储器准备好了；p_ready 是通知 CPU：外面的世界准备好了。

图 12.12 是一个具体的 Cache 电路的实现：它使用直接映像方式及写透策略。图中的三个 RAM 模块分别存放有效位 (Valid RAM)、高位地址标志 (Tag RAM) 和

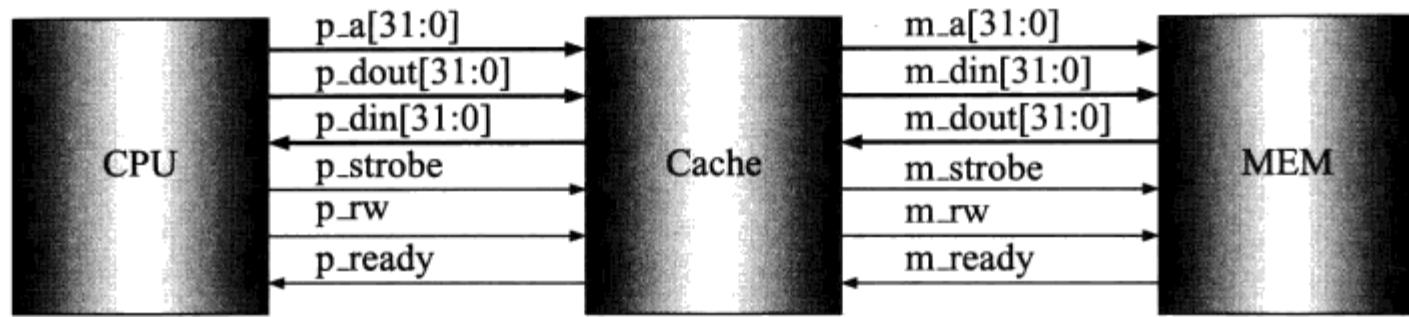


图 12.11 Cache 与 CPU 及存储器之间的接口信号

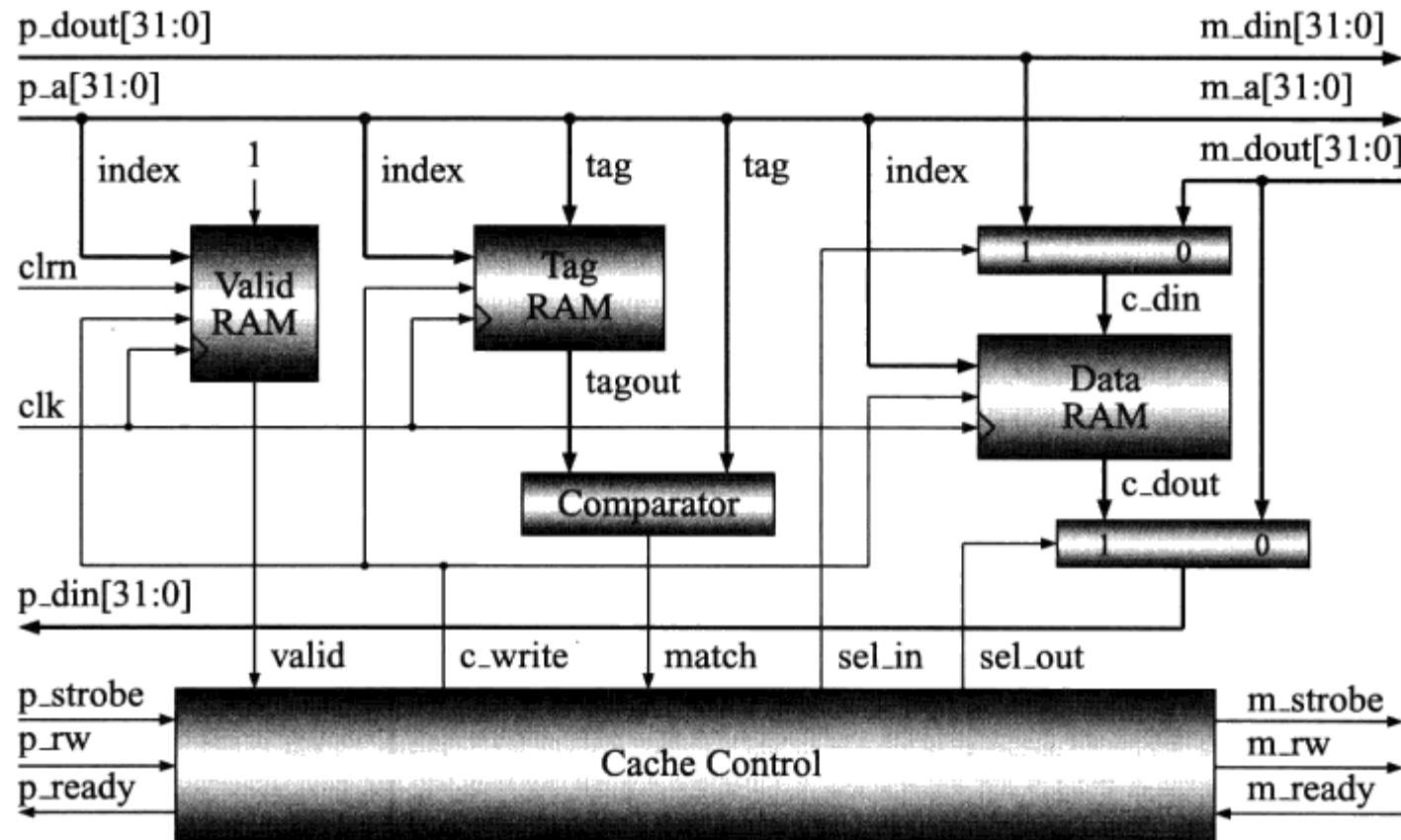


图 12.12 直接映像 Cache

Cache 数据 (Data RAM)，其中 Data RAM 的数据输入端和数据输出端各有一个二选一的多路器：写入 Cache 的数据有两个来源，一个是存储器、一个是 CPU。如果 Cache 没命中，从存储器取来的数据要写入 Cache 中。如果遇上存数据指令，则要把 CPU 的数据写入 Cache。送往 CPU 的数据 p_din 的来源也有两个，一个是 Cache 命中时的 Data RAM 数据，一个是没命中时从存储器取来的数据。两个多路器的选择信号由图中的控制电路产生。具体的实现见以下的 Verilog HDL 代码。

```
module d_cache #(parameter A_WIDTH = 32, parameter C_INDEX = 6)
(p_a, p_dout, p_din, p_strobe, p_rw, p_ready, clk, clrn,
m_a, m_dout, m_din, m_strobe, m_rw, m_ready);
input [A_WIDTH-1:0] p_a;
input [31:0] p_dout;
output [31:0] p_din;
input p_strobe;
input p_rw; // 0: read, 1: write
input
```

```
output          p_ready;
input           clk, clrn;
output [A_WIDTH-1:0] m_a;
input [31:0]      m_dout;
output [31:0]      m_din;
output           m_strobe;
output           m_rw;
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
wire  cache_miss = ~cache_hit;
assign m_din     = p_dout;
assign m_a       = p_a;
assign m_rw      = p_strobe & p_rw; // write through
assign m_strobe  = p_strobe & (p_rw | cache_miss);
assign p_ready   = ~p_rw & cache_hit |
                  (cache_miss | p_rw) & m_ready;
wire  c_write    = p_rw | cache_miss & m_ready;
wire  sel_in     = p_rw;
```

```

wire sel_out = cache_hit;
wire [31:0] c_din = sel_in ? p_dout : m_dout;
assign p_din = sel_out ? c_dout : m_dout;
endmodule

```

存放有效位的 RAM 开始时要清零，其他两个不用。由于使用写透策略，每次执行存数据指令 (sw 或 swc1) 时都要写存储器，不管 Cache 命中与否。因此 p_ready 信号中包含了一项 p_rw & m_ready (存储器写并且存储器准备好)。如果 p_ready 为 0，CPU 要等待并维持存储器访问信号。另外，由于该例中一个 Cache 块只有一个字，不需要写前读入，因此写不命中时也写 Cache。

图 12.13 和图 12.14 是以上代码的仿真结果。图 12.13 示出的是读操作没命中以及命中时的部分波形。图 12.14 示出的是写操作以及读命中时的部分波形。

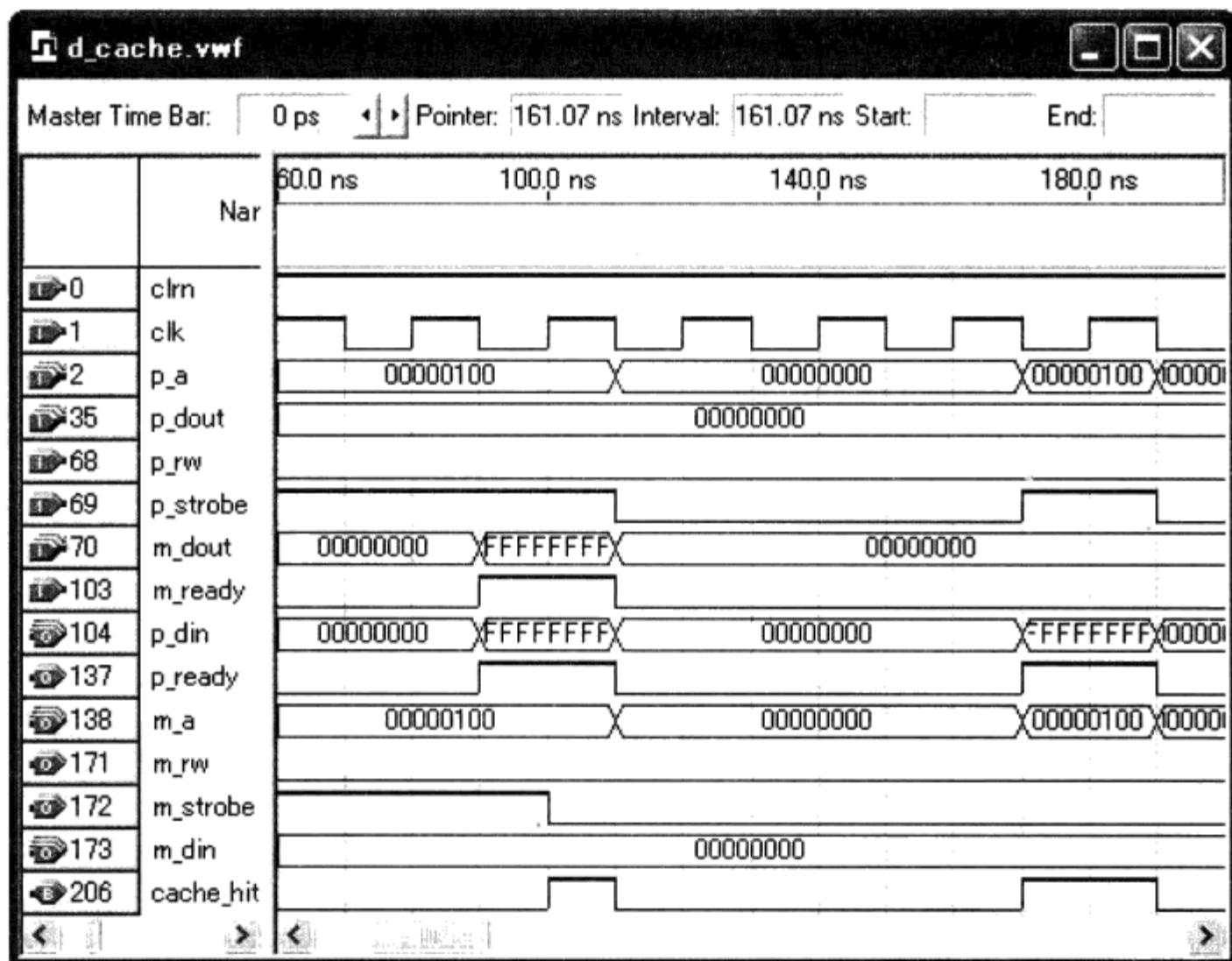


图 12.13 数据 Cache 仿真波形图 (读)

12.3 虚拟存储器管理及 TLB 设计

我们在本章开始处讲过，编译好的程序使用虚拟地址空间。当操作系统把程序调来执行时，为它分配存储器。在程序执行时，我们需要把程序中的虚拟地址转换成主存的实际地址。还有一些其他的名称，比如逻辑地址、处理机地址、程序地址

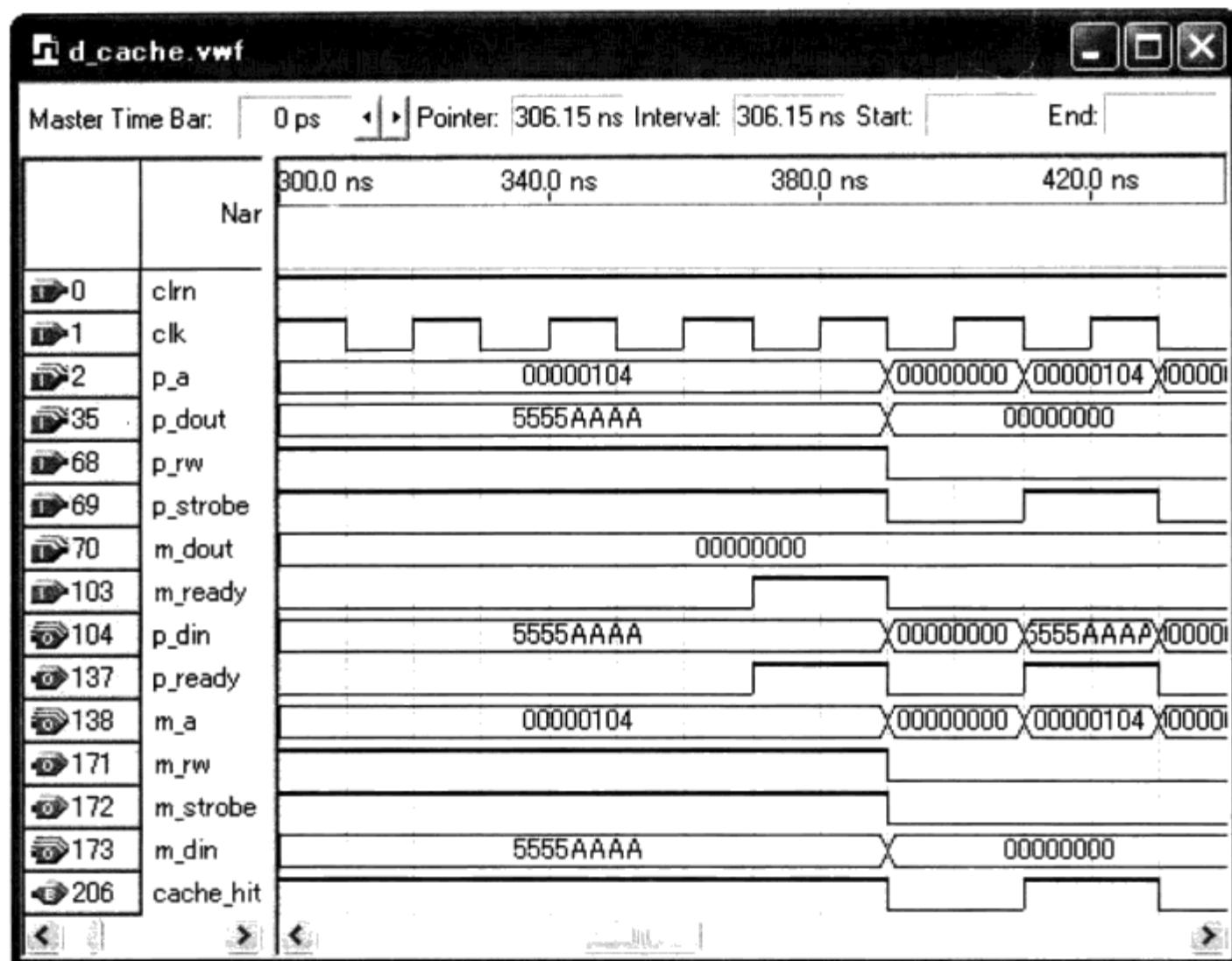


图 12.14 数据 Cache 仿真波形图 (写)

等，大致上它们的含义与虚拟地址相同。同样，存储器地址、物理地址、主存地址等又和实际地址有相同的含义。

12.3.1 虚拟存储器与主存的关系

我们知道，多个进程可以通过进程切换 (Context Switching) 的方式轮流运行，多线程 CPU 能同时运行多个进程。每个进程都使用从 0 开始的多至 4GB 的虚拟存储空间。图 12.15 示出的是两个进程的例子。由虚拟地址到实际地址的转换通过页表 (Page Table) 实现。页表的输入是进程号和虚拟页号，输出是主存的实际页号。图中主存的每页大小是 4KB，需要转换的是页号，页内偏移量不需转换。注意图中所示的是转换关系，实际上页表的输入只有一组信号，输出也只有一组信号。

把虚拟地址转换成实际地址这项工作由存储器管理部件 (Memory Management Unit, MMU) 完成。管理方法有分段管理 (Segmentation) 和分页管理 (Paging) 两种。图 12.15 示出的是分页管理。以下我们描述这两种管理方法。

12.3.2 分段管理

分段管理是把存储器分成若干段 (Segment)，为一个程序指定一个或几个“段寄存器” (Segment Registers)。这些寄存器指定当前段所在的主存的起始地址。每次访

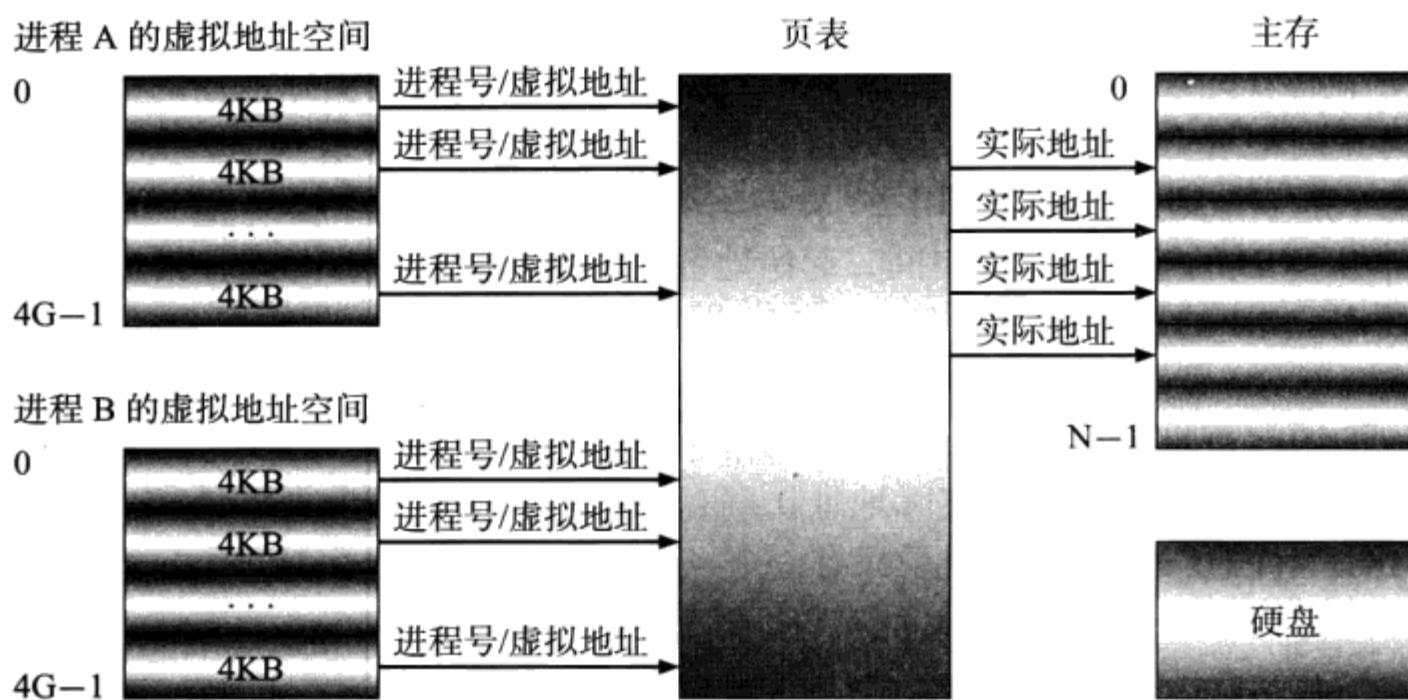


图 12.15 把虚拟地址转换成主存地址

问存储器时，主存的地址由段寄存器的内容与程序中的虚拟地址相加得到。比如 x86 就是这么做的。

除了主存的起始地址，段寄存器中可能还存有该段的大小以及对该段的访问控制信息。段的大小用于判断虚拟地址是否出界，而访问控制信息指出该段是否已在主存以及能否对其进行读写或执行等访问权限的信息。由此可见，分段管理中每段的大小是可以变化的。

12.3.3 分页管理

与分段管理不同，分页管理把存储器机械地分成若干页 (Page)，使用单一的虚拟地址，不需要段寄存器之类的东西。另外，虽然一页的大小是可以改变的，但是，一旦决定了，每页的大小就固定了，都是一样的。图 12.16 是分页管理的地址转换示意图，一页 4KB，页内地址有 12 位。

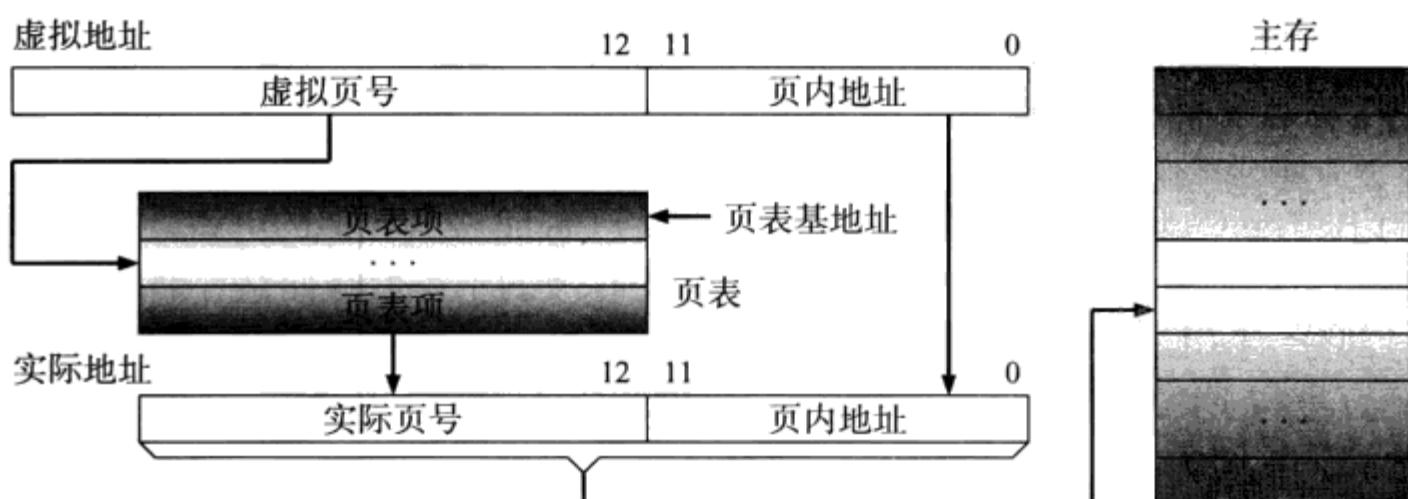


图 12.16 分页管理的地址转换

假设图 12.16 中的虚拟地址是 32 位。那么我们把高 20 位虚拟地址称做虚拟页号。使用虚拟页号作为地址访问页表 (Page Table)，从中得到实际地址的页号。实际页号与页内地址合起来就是实际地址。假设页表中的每一页表项 (Page Table Entry) 占用 4 个字节，则这个页表占用 $2^{20} \times 4 = 4\text{MB}$ 。问：页表在哪里？答：主存。页表需要占用 4MB 的连续的主存空间，起始地址由页表基地址 (实际地址) 指定。

使用 4MB 的连续的主存空间显然违背分页管理的精神。我们把 20 位的虚拟页号分成两部分：页目录地址 (10 位) 和页表地址 (10 位)，见图 12.17。我们首先使用页目录地址访问页目录，从中得到页表的起始地址，然后再使用页表地址访问页表，得到实际页号，最后再使用实际地址访问存储器。这样，一个页目录和一个页表都只占 4KB。注意，4KB 的页目录只有一个，但 4KB 的页表却有 1K 个。

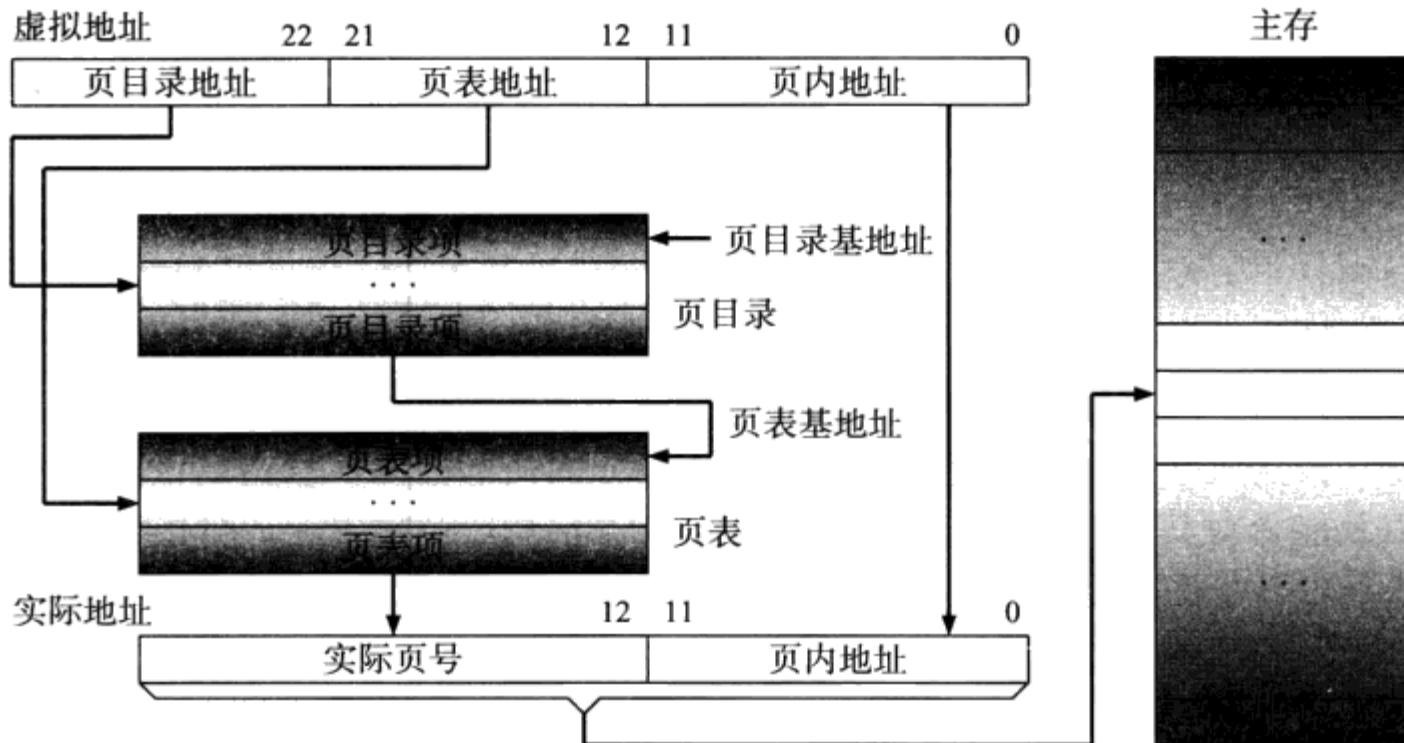


图 12.17 两级分页管理的地址转换

你看，为了访问一次数据存储器，先要访问一次页目录存储器，再要访问一次页表存储器，这时得到的才是实际地址。即，总共要访问三次主存。本来主存就比 CPU 的速度慢，乘以 3，就更慢了。有没有办法加快这个地址转换呢？有。

12.3.4 快速地址转换 TLB 及其电路设计

TLB (Translation Lookaside Buffer) 是一种与 Cache 极其相似的电路，只是它的目的是加快地址转换的速度。即，Cache RAM 中存放的不是数据而是存储器地址的实际页号部分。图 12.18 示出的是一种全相联映像的 TLB 的结构示意图。从 RAM 出来的实际页号与页内地址合起来是访问主存的实际地址。其他映像的 TLB 结构请参考相应的 Cache 结构。

图 12.19 示出的是 8 个 TLB 项的全相联 TLB 模块图。图中 CAM 是相联存储器，有 8 项，用虚拟页号来匹配 CAM 中的 vpn。若有一项匹配，vpn_found 输出 1，

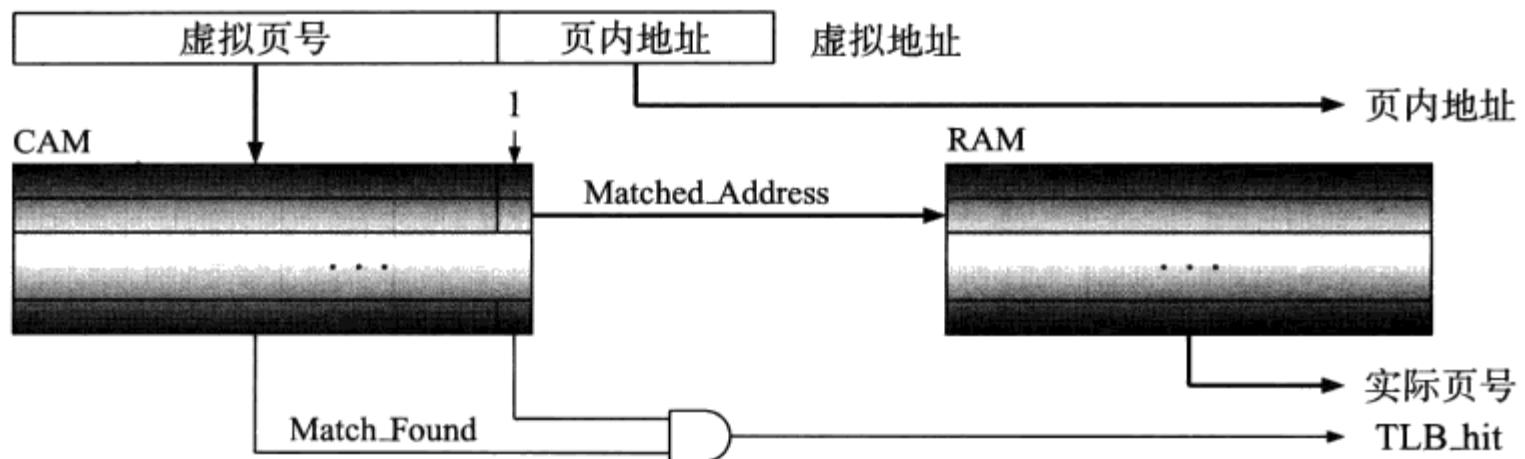


图 12.18 全相联映像的 TLB 示意图

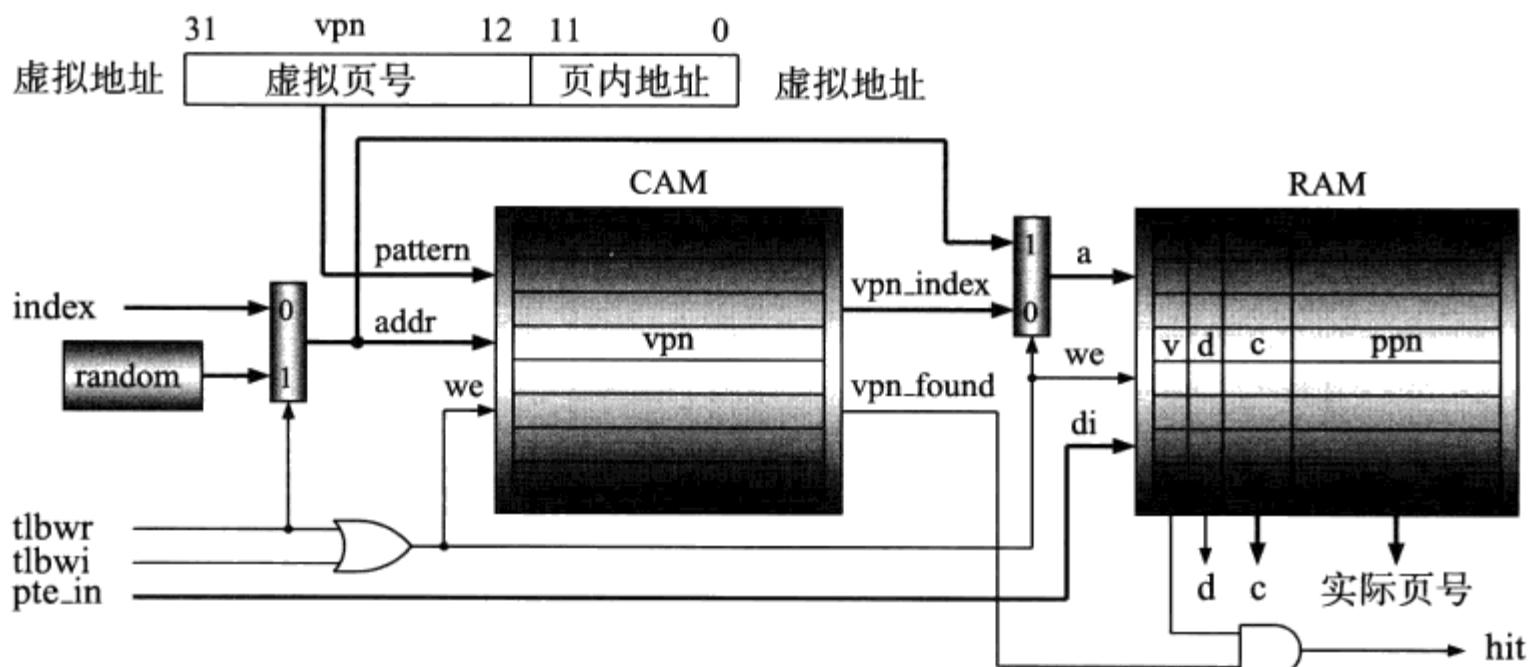


图 12.19 8 个 TLB 项的全相联 TLB 模块图

vpn_index 输出该项的地址，并作为访问 RAM 的地址，从 RAM 中得到实际页号。对 CAM 和 RAM 的写操作有两种方式：tlbwi 指令使用指定的地址 index；tlbwr 指令使用随机的地址 random。pte_in 是往 RAM 中写入的实际地址页号。hit 表示 TLB 命中。

图 12.20 是具体的电路实现。CAM 用 Altera 的 altcam 器件，RAM 用 lpm_ram_dq 器件。随机数使用一个 3 位的计数器。

以下是图 12.20 中的 ram8x24 模块的 Verilog HDL 代码。

```
module ram8x24 (address, data, inclock, outclock, we, q);
    input [2:0] address;
    input [23:0] data;
    input inclock;
    input outclock;
    input we;
    output [23:0] q;
    lpm_ram_dq ram_comp (.outclock (outclock),
                           .address (address),
                           .data (data),
                           .we (we),
                           .q (q));
endmodule
```

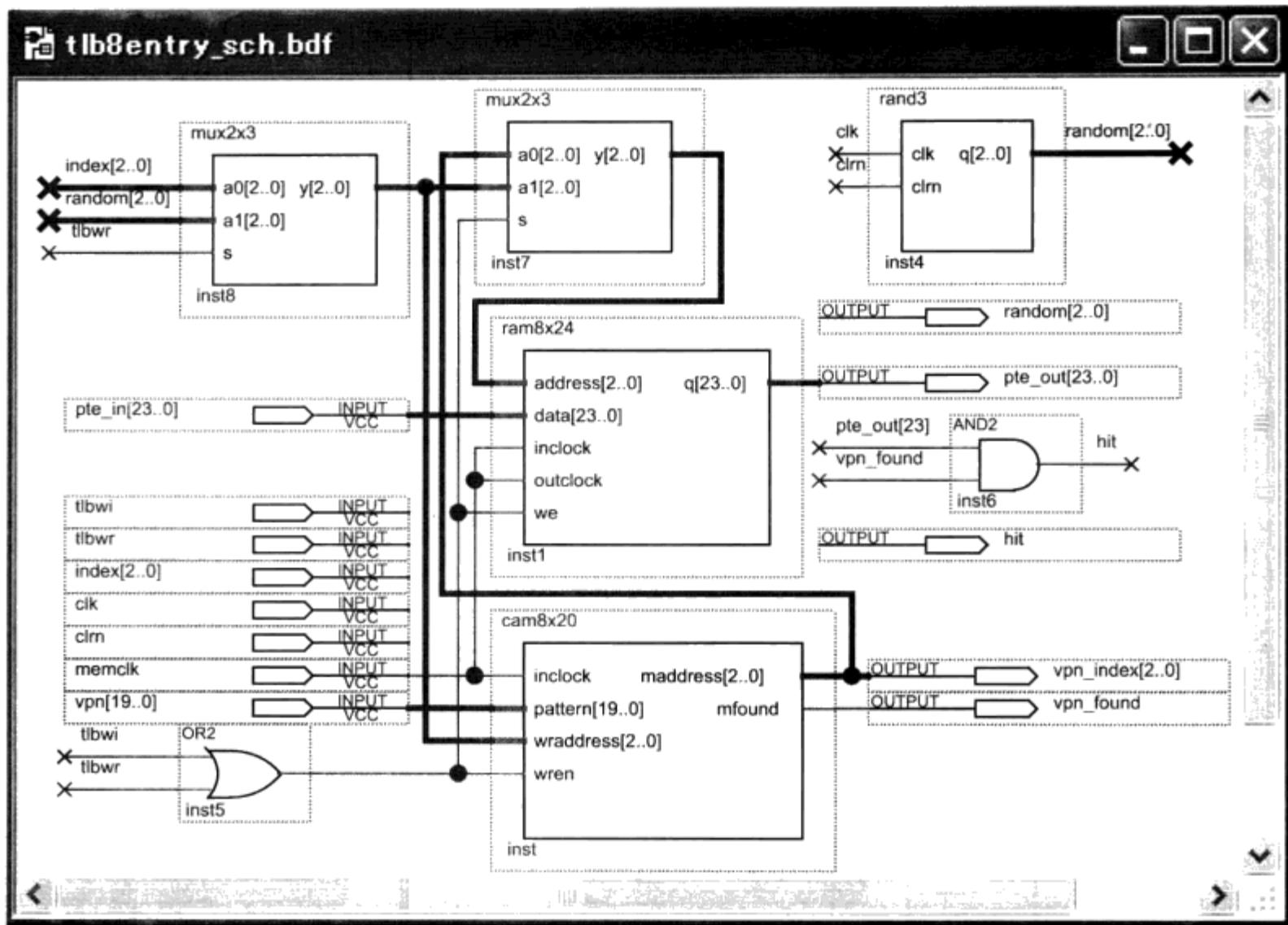


图 12.20 8 个 TLB 项的全相联 TLB 电路图

```

        .address (address),
        .inclock (inclock),
        .data (data),
        .we (we),
        .q (q));

defparam ram_comp.lpm_width      = 24,
      ram_comp.lpm_widthad = 3,
      ram_comp.lpm_indata = "registered",
      ram_comp.lpm_outdata = "registered",
      ram_comp.lpm_type    = "lpm_ram_dq",
      ram_comp.lpm_address_control = "registered";
endmodule

```

以下是图 12.20 中的 cam8x20 模块的 Verilog HDL 代码。

```

module cam8x20 (inclock,pattern,wraddress,wren,maddress,mfound);
  input inclock,wren;
  input [19:0] pattern;
  input [2:0] wraddress;
  output [2:0] maddress;
  output mfound;

```

```

altcam cam (.wren (wren),
    .inclock (inclock),
    .pattern (pattern),
    .wraddress (wraddress),
    .maddress (maddress),
    .mfound (mfound));
defparam cam.lpm_type      = "altcam",
    cam.match_mode     = "single",
    cam.numwords      = 8,
    cam.output_aclr   = "off",
    cam.output_reg    = "inclock",
    cam.pattern_aclr = "off",
    cam.pattern_reg   = "inclock",
    cam.width         = 20,
    cam.widthad       = 3,
    cam.wraddress_aclr= "off",
    cam.wrcontrol_aclr= "off";
endmodule

```

图 12.21 是使用 tlbwi 写入 TLB 时的仿真波形，图 12.22 是命中时的仿真波形及没命中时使用 tlbwr 写入 TLB 时的仿真波形。

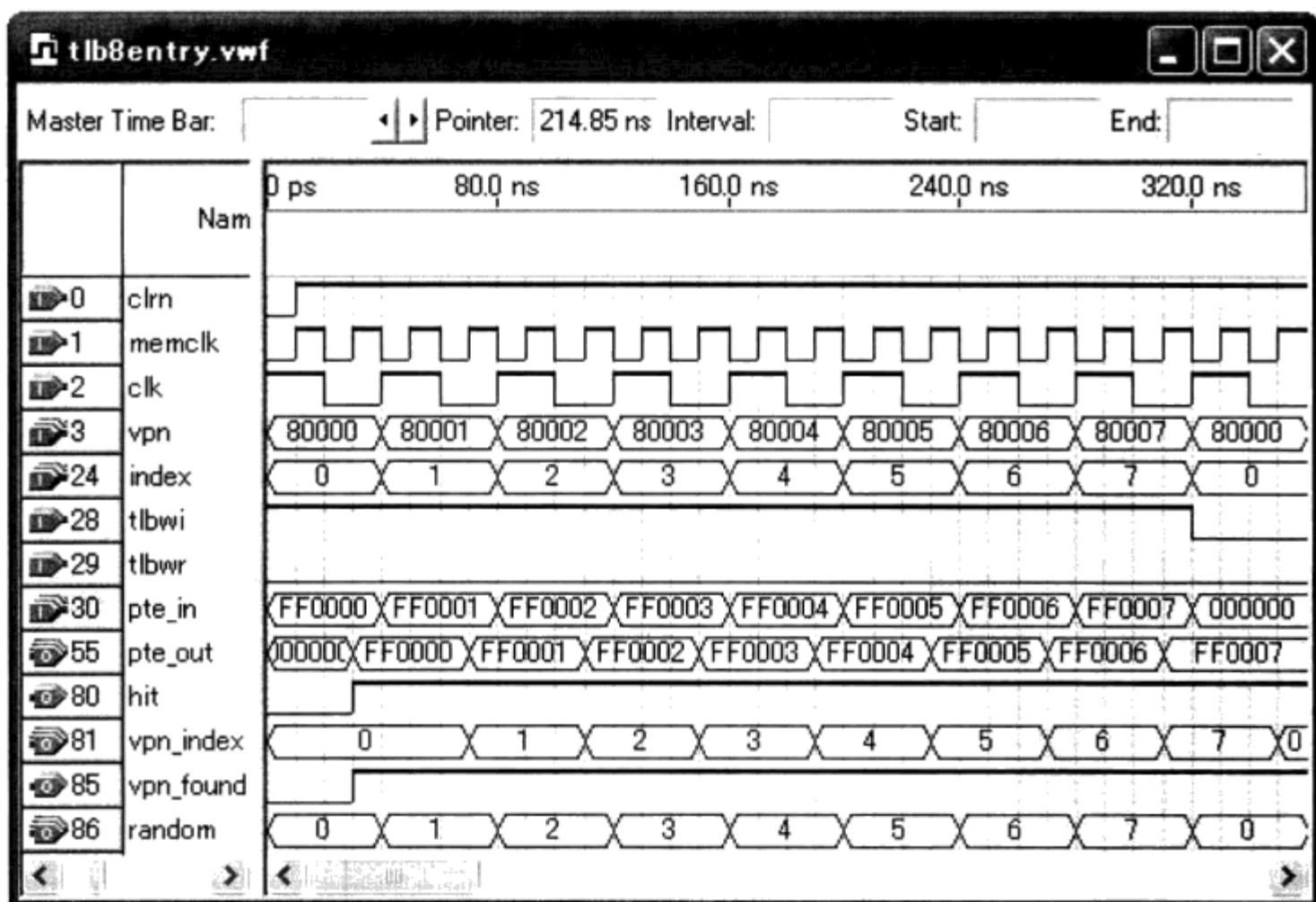


图 12.21 全相联 TLB 电路仿真波形图 (tlbwi 写入 TLB)

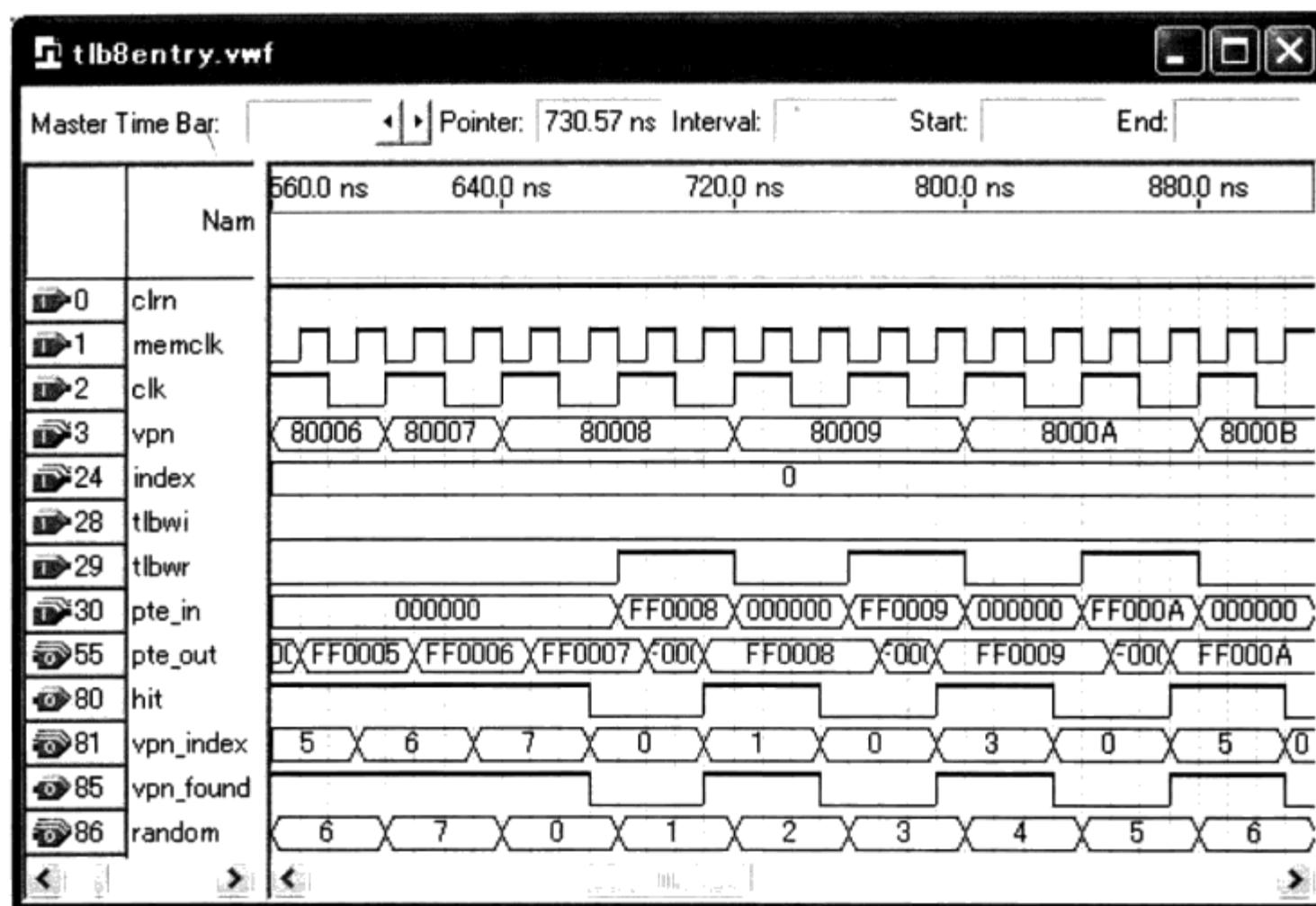


图 12.22 全相联 TLB 电路仿真波形图 (tlbwr 写入 TLB)

12.3.5 TLB 与 Cache 的并行访问

假设我们使用 k 路组相联映像的 Cache 且 Cache 的标志是实际地址的高位部分。在分页管理的情况下，假设页的大小为 2^m 字节。当 Cache 的容量不大于 $2^m \times k$ 字节时，访问 Cache 和访问 TLB 可以同时进行，见图 12.23。

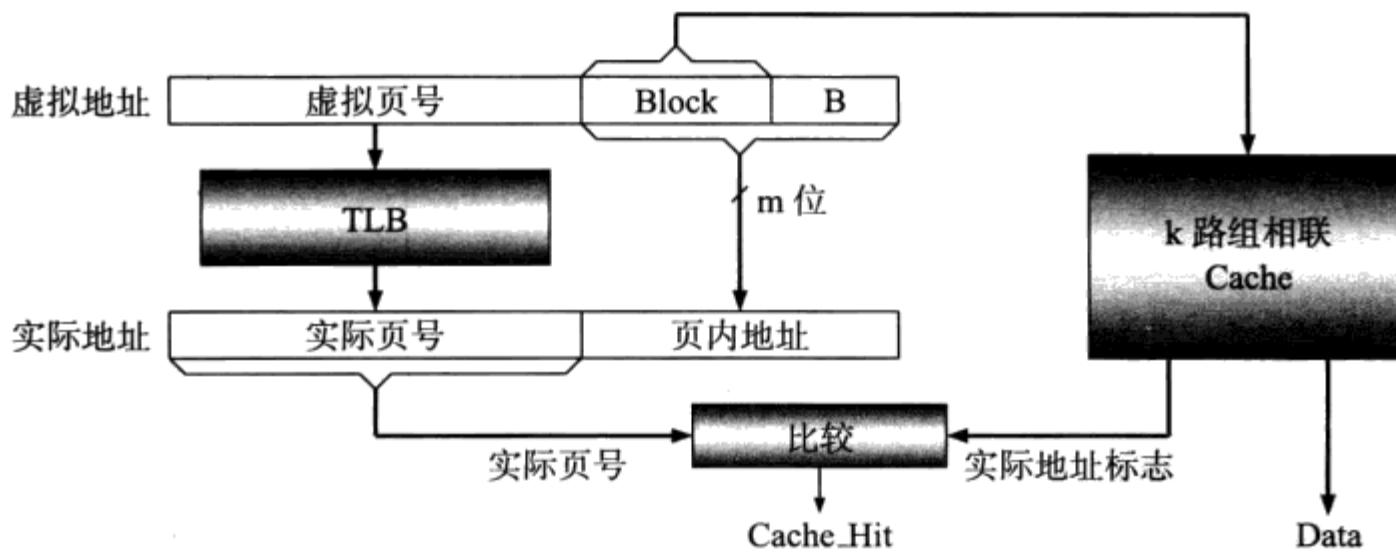


图 12.23 TLB 与 Cache 可以并行访问的条件

同时访问的意思是：使用虚拟页号查找 TLB 得到实际页号，使用不需转换的页内地址访问 Cache。假设二者的访问时间相同，则实际页号和 Cache 标志同时输出，

然后比较它们以确定 Cache 是否命中。例如， $m = 12$ 、 $k = 4$ 时，我们可以使用多至 16KB 的 Cache。如果 Cache 的容量超出这个界限，访问 Cache 时不得不使用 TLB 输出的实际页号的低若干位了，从而延长了 Cache 的访问时间。

以上是使用实际地址访问 Cache 的情况。那么访问 Cache 直接使用虚拟地址（地址标志也是虚拟地址的高位部分）是否可行呢？如果可行，它又会有怎样的结构呢？会出现什么问题吗？请读者思考并给出设计方案。

12.4 MIPS 基于 TLB 的虚拟地址转换机制

MIPS 使用 TLB 把虚拟地址转换成实际地址。特点是，MIPS 提供了用于 TLB 维护的指令和寄存器。

12.4.1 MIPS 的虚拟地址空间

MIPS 体系结构把 CPU 的运行状态分成三种：用户方式（User Mode）、管理方式（Supervisor Mode）和核心方式（Kernel Mode）。为了讨论方便，我们在这里忽略管理方式。图 12.24 是 MIPS 的虚拟地址空间与实际地址空间的对应关系。图中把 4GB 的虚拟地址空间分成了 4 段（Segments）：kseg2（2GB）、kseg0（512MB）、kseg1（512MB）和 kseg2（1GB）。核心方式可以使用所有的段，而用户方式只能使用 kuseg。虽然称做段，但 MIPS 的存储器管理用分页管理方式。

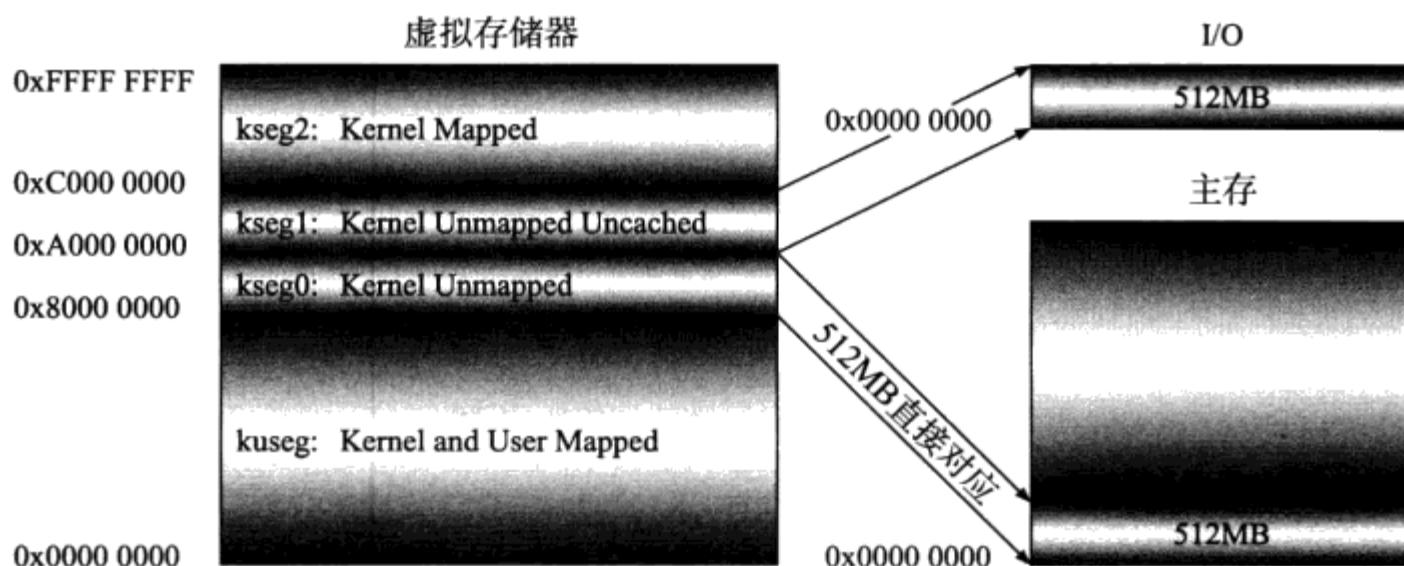


图 12.24 MIPS 虚拟地址空间映像

图中的“Mapped”表示虚拟地址要经过 TLB 转换（kseg2 和 kuseg），“Unmapped”不允许使用 TLB（kseg1 和 kseg0），“Uncached”表示不能往 Cache 里放（kseg1）。kseg0（0x8000 0000 ~ 0x9FFF FFFF）虽然不经过 TLB，但也不是直接使用，而是直接映像到主存的 0x0000 0000 ~ 0x1FFF FFFF。这相当于某些其他 CPU 的“Real Mode”。如果你有机会阅读为 MIPS 准备的操作系统的源程序，你会发现很多代码的起始地址是 0x8xxx xxx0（虚拟地址），把最高位的 1 改成 0 就是实际的主存地址。

注意，kseg1 也直接映像到实际地址空间的 (0x0000 0000 ~ 0x1FFF FFFF)。这岂不是要引发战争吗？不会，这段地址用于访问 I/O。这也是为什么它不被允许使用 Cache 的原因。你往存储器的某个单元写入一个数据，然后再读它：数据还是那个数据。但是，I/O 地址空间不具有这个特性：你往一个 I/O 地址写的可能是“控制”信息，而从相同的地址读来的可能是“状态”信息。所以禁止 kseg1 使用 Cache。MIPS 读写 I/O 也是使用诸如 lw 和 sw 之类的存储器访问指令。这就是所谓的“存储器映像的 I/O”。这一点与 x86 不同，x86 有专门的 I/O 指令。

你看，当机器刚启动时，TLB 没被初始化也没关系：操作系统既有从最低地址开始的 512MB 的主存可以使用，又有 512MB 的 I/O 地址可用。作为一个系统软件工程师，没什么好抱怨的了吧。接下来的工作是初始化页表以及 TLB，好让 kseg 和 kseg2 也能有存储器可用。

12.4.2 MIPS TLB 的构成

MIPS 所有的进程都有相同的虚拟地址空间。为了能够区分开它们，MIPS 为每个进程指定一个不同的“地址空间标示符” ASID (Address Space Identifier)。ASID 有 8 位，可以被看作是“扩展的虚拟地址”的高位部分。这样，不同的进程就有不同的“扩展的虚拟地址”空间了。在用 TLB 做地址转换时，ASID 也参加比较。但是，在某些情况下，操作系统又希望所有的进程“共享”相同的虚拟地址空间。为此，TLB 中增加了一位“全局”标志 G (Global)。如果 G 被设置为 1，则在转换时忽略 ASID。

使用 TLB 的目的是为了加快从虚拟地址到实际地址的转换。MIPS TLB 使用全相联映像结构，它的每一项由“虚”和“实”两部分组成，见图 12.25。

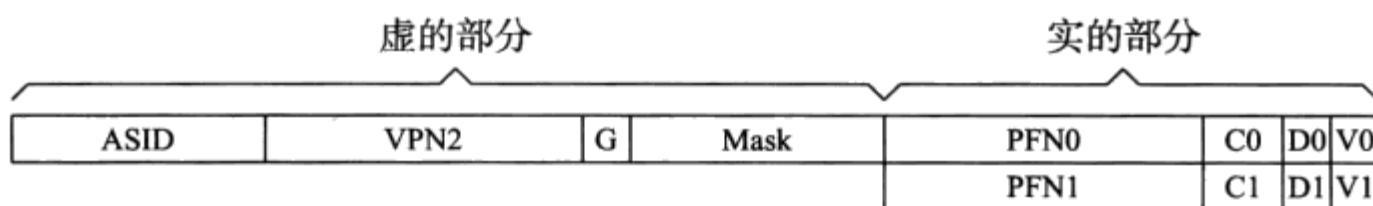


图 12.25 一个 TLB 项的内容

虚的部分包含有 ASID、全局标志 G、虚拟页号 (Virtual Page Number) VPN2 (实际的意义是 VPN/2，因为每一项对应两个实际的存储器页：偶页和奇页) 和用来指定一页大小的页屏蔽域 Mask。实的部分包含有偶页和奇页两组实际地址页号 PFN (Physical Page Frame Number) 及相关的控制和状态信息 (C、D 和 V)。偶页组编号为 0、奇页组编号为 1。V (Valid) 表示该实际地址页号是否有效；D (Dirty) 指出该页存储器的内容是否被修改过，比如执行 sw 指令；C (Cache Coherency) 是 Cache 一致性信息 (见表 12.1)。

CP0 寄存器 PageMask、EntryHi、EntryLo0 和 EntryLo1 中定义的每个域与 TLB 项中的每个域具有相同名称。这些寄存器被用来访问 TLB 项。TLB 偶页项来自于 EntryLo0 寄存器，奇页项来自于 EntryLo1 寄存器。有一处稍微不同：TLB 项中的 G

表 12.1 Cache 页一致性属性

C	Cache 页一致性属性
0 ~ 1	没有定义、具体实现时可由厂家自定义
2	该页存储器的内容不准往 Cache 里面放
3	该页存储器的内容可以往 Cache 里面放 (一般的存储器)
4 ~ 7	没有定义、具体实现时可由厂家自定义

是 EntryLo0 和 EntryLo1 中的两个 G 的逻辑“与”。

在表 12.1 中没有定义的 C 的值可以由 MIPS CPU 的生产厂家自己来定义，比如 IDT79RC32355 CPU^[4] 中对 C 域有如表 12.2 所示的定义。IDT79RC32355 手册中没有讲采用写回策略时 (C = 3) 对写不命中如何处理，但作者猜测是把相应的数据块取到 Cache 中 (Write Allocate)，以便下次访问到它时命中，反正它采用的是一次性写回的策略，不用每次执行 sw 指令时都要写入存储器 (Cache 命中时只写 Cache)。

表 12.2 IDT79RC32355 Cache 页一致性属性

C	Cache 页一致性属性
0	可以往 Cache 里面放、写透、写不读入 (Write Through, No Write Allocate)
1	可以往 Cache 里面放、写透、写前读入 (Write Through, Write Allocate)
2	不准往 Cache 里面放
3	可以往 Cache 里面放、写回 (Write/Copy Back)
4 ~ 7	保留

下面讲 TLB 项中的 Mask 域到底是干什么用的。Mask 对应于 CP0 第 5 号寄存器 PageMask (见图 12.26) 中的 Mask 域。它有 16 位，占用 PageMask 寄存器的 [28:13] 位。如果这些位全部为 0，则存储器页的大小为 4KB。即，实际地址的低 12 位 ([11:0]) 与虚拟地址的低 12 位相同。虚拟页号 (虚拟地址中的 [31:12] 位) 需要转换。但由于一个 TLB 项有奇偶两组实际页号，由虚拟地址的 [12] 位来选择，所以参与转换的虚拟地址位就变成了 [31:13]。MIPS CPU 可以通过设置适当的 Mask 值来改变存储器页的大小。当 Mask 位为 1 时，相应位置的虚拟地址就不需转换了，即一页存储器变大了。不像第 6 章中的中断屏蔽 IM，这里的 Mask 是真正意义上的“屏蔽”。



图 12.26 PageMask 寄存器 (CP0 寄存器 5) 的格式

表 12.3 列出了 Mask 的取值和与其相对应的存储器页的大小。最大的页就是 256MB 了。把 4GB 定为存储器页的大小不能说是愚蠢的，它意味着不需要做地址转换，比如在大部分嵌入式的应用中就不转换地址，而且存储器也不需要那么大。表中最右列的 S 用于选择实际地址页号 0 还是页号 1，它只有一位，括号中的数字是它

表 12.3 PageMask 寄存器中的 Mask 取值

页的 大小	位															S	
	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	
4KB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	[12]
16KB	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	[14]
64KB	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	[16]
256KB	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	[18]
1MB	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	[20]
4MB	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	[22]
16MB	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	[24]
64MB	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[26]
256MB	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[28]

在虚拟地址中所处的位号。在具体的 MIPS CPU 的设计中，虽然要做地址转换，但你也可以不使用 PageMask 寄存器。这时存储器页的大小就是固定的 4KB 了。

MIPS CP0 寄存器 EntryLo0、EntryLo1 和 EntryHi 的格式如图 12.27 和图 12.28 所示。EntryLo0 和 EntryLo1 的格式完全相同，主要内容是存储器地址的实际页号；EntryHi 的主要内容是虚拟页号。



图 12.27 EntryLo0 和 EntryLo1 寄存器 (CP0 寄存器 2 和 3) 的格式



图 12.28 EntryHi 寄存器 (CP0 寄存器 10) 的格式

两个图中所有的域的意义已经在对 TLB 项的描述中简单地提过了，这里再多讲几句。PFN (Physical Page Frame Number) 是存储器实际地址的页号，有 24 位。假设每页为 $4KB = 2^{12}B$ ，PFN 和页内偏移量两部分地址拼凑起来就是存储器的实际地址，有 $24 + 12 = 36$ 位，能访问 $2^{36} = 64GB$ 的存储器。你的机器有那么多的内存吗？没有吧。怎么办？简单！把地址的高位扔了。实际上，操作系统能侦察出你的机器的家底有多厚。没那么多存储器的话，操作系统也不会把进程的虚拟地址转换到你力所不能及的界外。

TLB 项中的 PFN 从哪里来？是从那寄存器 EntryLo0 或 EntryLo1 中来。寄存器 EntryLo0 或 EntryLo1 中的 PFN 又从哪里来？对面的 IU 写过来。写什么呢？IU 从哪里能得到它呢？答案是从我们已经讲过的用于分页存储器管理的动态页表中得到。

图 12.27 中有一位 V (Valid)，为 1 时表示可以使用该项做地址转换；为 0 时产生

TLBL 或 TLBS 异常(见第 6 章表 6.1 对 Cause 寄存器中 ExcCode 的定义)。

还有就是图 12.27 中的 D 又是负的什么责呢? D: Dirty, 有不干不净的意思。一页存储器不干净是什么意思呢? 存储器本来是干净的 ($D = 0$), 当 CPU 执行诸如 sw 类的存数据指令第一次往该页写数据时, 存储器就不干净了。这是一件再普通不过的事情了, 但还是要向 CPU 报警: 产生 Mod 异常(也见表 6.1)。如果该页存储器已经不干净了 ($D = 1$), 就可以随便写了, 再也不会报警了。

操作系统可以使用它来实现一些存储器分页管理的算法, 比如页替换算法等。当不干净的页被替换掉时, 要把它保存到硬盘的 Swap 区域(文件)。这一点与 Cache 的管理非常类似: 存储器管理使用全相联映像方式, 并且使用写回策略。其他 CPU 的 TLB 项中经常会有一位 Reference 或 Used 位, 用于实现类似于 LRU 的存储器页替换算法。但 MIPS 没有 Reference 位, 似乎只能使用随机替换策略了。

12.4.3 MIPS 虚拟地址转换

当对一个虚拟地址进行转换时, 把虚拟页号 VPN2 以及当前进程的 ASID 同时与 TLB 中的所有项进行比较(全相联)。如果以下条件全部满足时, TLB 命中, 从而得到实际地址页号。

- 1) 由虚拟地址中的 S 位选中的有效位 V0 或 V1 的值是 1(有效), S 位在虚拟地址中的位置由 PageMask 寄存器中的 Mask(也在 TLB 中)决定, 见表 12.3;
- 2) 当前进程的 ASID 与 TLB 项中的 ASID 匹配, 或者 TLB 项中的 G 为 1;
- 3) 去掉被屏蔽的位, 虚拟页号 VPN2 与 TLB 项中的 VPN2 相同。屏蔽哪些位由 PageMask 寄存器中的 Mask 域指定。MIPS 利用 PageMask 寄存器可以改变存储器页的大小。如果 CPU 中没有设计 PageMask 寄存器, 则认为 Mask 为 0, 虚拟页号 VPN2 的任何一位都不被屏蔽。这时存储器页的大小为 4KB。

见图 12.29, 如果 TLB 命中, 未经转换的页内地址与从 TLB 得到的实际地址页号合在一起, 形成访问存储器的实际地址。为了清楚地表示出虚拟地址中的 S 位, 屏蔽位 Mask 画在了虚拟地址的下面, 但要注意它是在 TLB 中。

12.4.4 MIPS TLB 维护指令

MIPS CPU 不能对 TLB 项直接访问。见图 12.30, CPU 要想读写 TLB 项, 必须使用 CP0 中的若干寄存器和一些特殊的指令(tlbp、tlbr、tlbwi 和 tlbwr)。

1. 与 TLB 维护指令有关的另外三个寄存器

我们已经介绍过了几个与 TLB 有关的 CP0 寄存器, 比如 PageMask、EntryLo0、EntryLo1 和 EntryHi 寄存器。以下再介绍另外三个与 TLB 维护指令有关的寄存器, 它们是 Index 寄存器(CP0 寄存器 0)、Random 寄存器(CP0 寄存器 1)和 Wired 寄存器(CP0 寄存器 6)。图 12.31 示出的是这三个寄存器的格式。

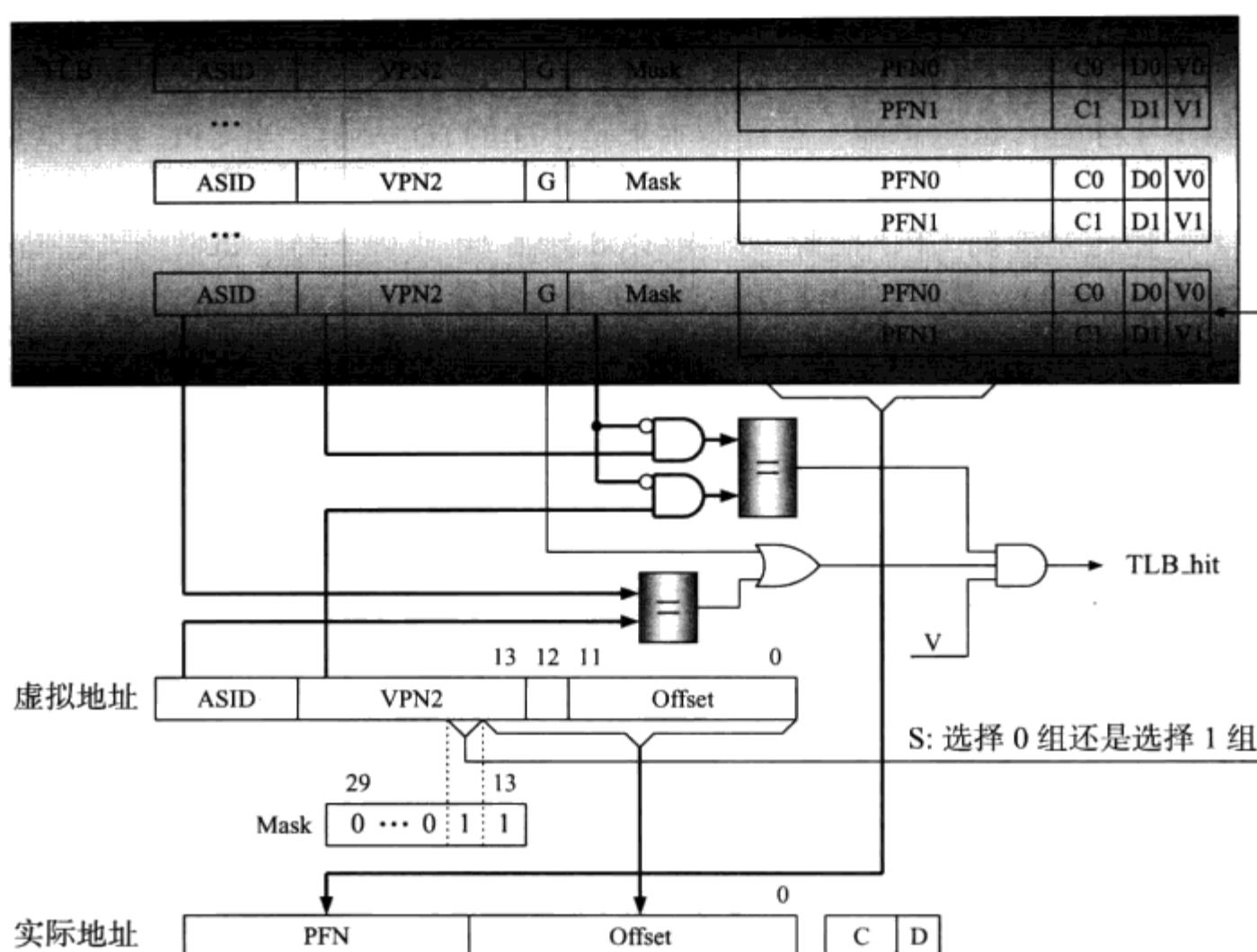


图 12.29 基于 TLB 的虚拟地址转换 (与所有的 TLB 项同时比较)

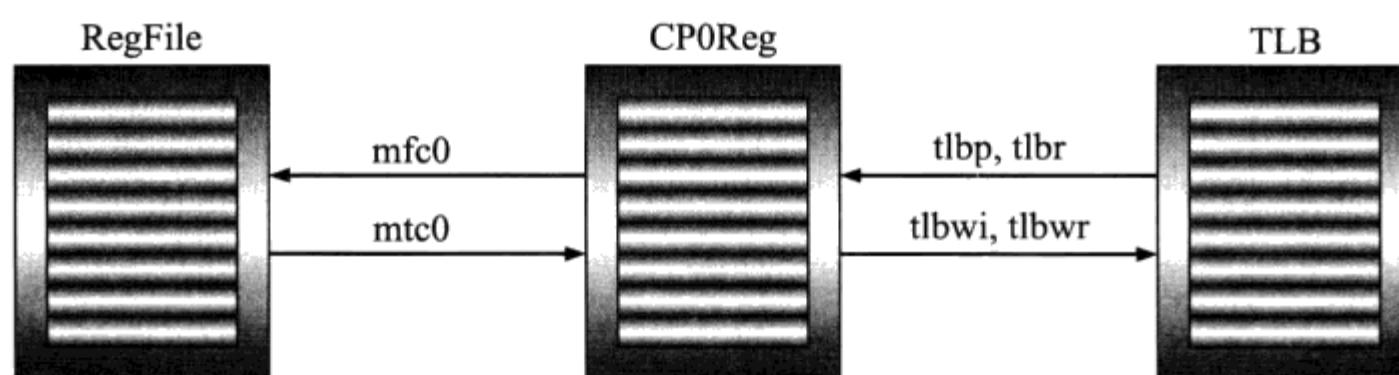


图 12.30 CPU 访问 TLB 必须经过 CP0 寄存器

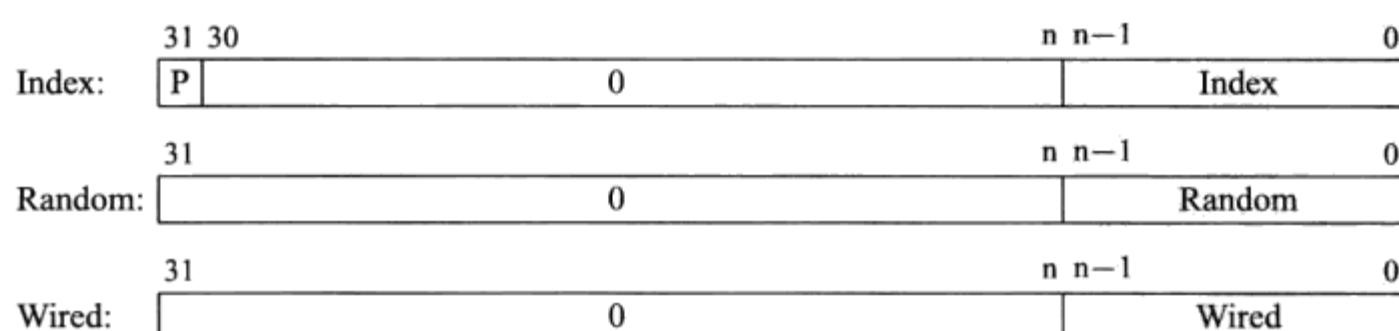


图 12.31 Index、Random 和 Wired 寄存器的格式

TLB 中有 $N = 2^n$ 个 TLB 项，每项都有一个 n 位的编号，相当于地址。Index

域就是被用来“指出”或“指定”这个地址的。比如 CPU 想修改某一 TLB 项中的内容，CPU 先要执行 mtc0 指令把那一项的地址写入 Index 寄存器，然后执行 tlbwi 指令。这是“指定”的意思。另外，CPU 有时也想调查一下 TLB 中有没有一项与 EntryHi 寄存器的内容匹配。如果有，硬件把那一项的地址写入 Index 域并把 P 位清零。这是“指出”的意思。CPU 可以使用 mfc0 指令把 Index 寄存器的内容读过来。

Random 寄存器是一个只能由 CPU 读但不能写的寄存器。我们知道 TLB 项的内容可能要被新的内容替换掉。替换掉哪一项呢？MIPS CPU 支持随机替换策略，即由硬件产生一个随机数，把这个随机数存放在 Random 寄存器。这个随机数的功效与 Index 相同，也是指定 TLB 项的地址。CPU 可以使用一条 tlbwr 指令来修改由 Random 寄存器指定的 TLB 项。

如果某些 TLB 项属于“重点保护单位”，不能被“随机”地替换掉，怎么办呢？答案是使用 Wired 寄存器。图 12.32 给出 Wired 寄存器的意义。假设 Wired 寄存器的内容是 i，则 TLB 的 $0 \sim i - 1$ 项是重点保护单位。

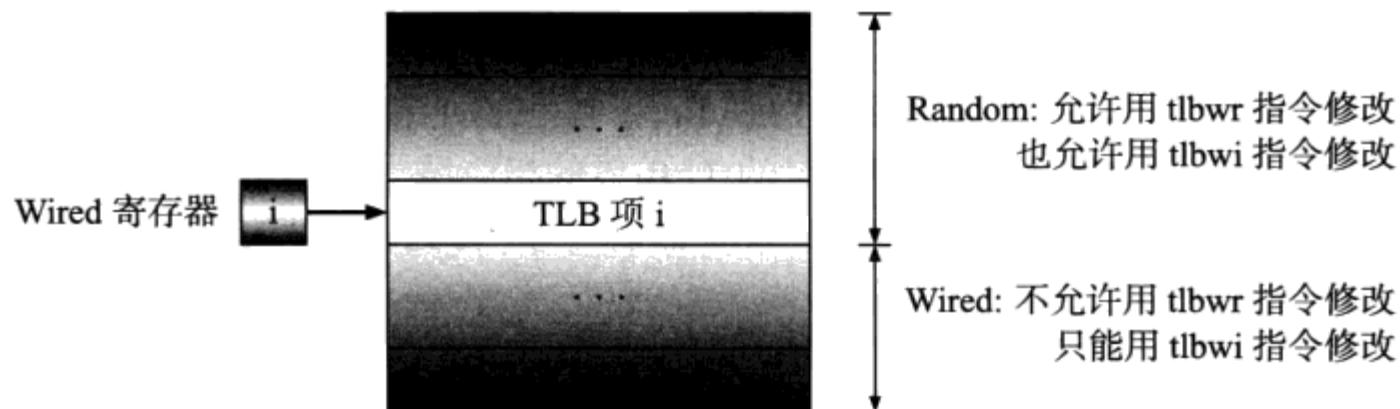


图 12.32 Wired 的意义

MIPS 用于 TLB 维护的指令有 4 条。使用 Index 寄存器修改 TLB 项的指令是 tlbwi；使用 Random 寄存器修改 TLB 项的指令是 tlbwr。还有两条指令是 tlbp 和 tlbp。它们的指令格式见图 12.33。

tlbp	31	26 25 24	6 5	0
	0 1 0 0 0 0 1	0 0 0 1 0 0 0		
tlbr	31	26 25 24	6 5	0
	0 1 0 0 0 0 1	0 0 0 0 0 0 1		
tlbwi	31	26 25 24	6 5	0
	0 1 0 0 0 0 1	0 0 0 0 0 1 0		
tlbwr	31	26 25 24	6 5	0
	0 1 0 0 0 0 1	0 0 0 0 1 1 0		

图 12.33 MIPS 4 条用于 TLB 管理的指令

2. tlbp (Probe TLB for Matching Entry) 指令

tlbp 指令检查是否有一 TLB 项与 EntryHi 寄存器的内容匹配(相同)。如果有，则把该 TLB 项的号码(地址)存入 Index 寄存器，最高位 P(Probe Failure)清零；如果没有，把 Index 寄存器的最高位 P 置 1。检查是否匹配时不能忘掉的事情有以下两个。一个是要考虑屏蔽位(PageMask 寄存器中的 Mask)，即只比较 VPN2 中的那些没被屏蔽的位；另一个是要考虑全局标志 G。如果 G = 0，还要比较二者的 ASID。

3. tlbr (Read Indexed TLB Entry) 指令

tlbr 指令读 TLB，把由 Index 寄存器中的 Index 域指定的 TLB 项的内容送到 EntryHi、EntryLo0、EntryLo1 和 PageMask 寄存器。注意 TLB 项中的一位 G 要写入 EntryLo0 和 EntryLo1 两个寄存器中的 G 位(相同了)，而当初写 TLB 时 EntryLo0 和 EntryLo1 两个寄存器中的 G 位可能不同，逻辑“与”后写入 TLB 项中的 G 位。

4. tlbwi (Write Indexed TLB Entry) 指令

tlbwi 指令写 TLB，把 EntryHi、EntryLo0、EntryLo1 和 PageMask 寄存器的内容送到由 Index 寄存器中的 Index 域指定的 TLB 项的相应域。注意 TLB 项中的一位 G 由 EntryLo0 和 EntryLo1 寄存器中的两位 G 逻辑“与”得到。

5. tlbwr (Write Random TLB Entry) 指令

tlbwr 指令也是写 TLB，完成与 tlbwi 类似的任务，不同点是它不使用 Index 寄存器，而是使用随机数寄存器 Random 来指定 TLB 项。

流水线 CPU 一定要有两个 TLB：指令 TLB 和数据 TLB，分别用于取指令和访问数据时的地址转换。而 MIPS 只提供了一套 TLB 读写指令，并且没有提供其他的手段能用软件分别读写这两个 TLB，导致 TLB 的设计变得有些麻烦。搞不懂设计者当初是怎么想的。

12.5 习题

1. 简要解释什么是 Cache、MMU 和 TLB 以及需要它们的原因。
2. 调查扇区映像(Sector Mapping)的 Cache 结构。
3. 设计并仿真一个两路组相联的 Cache 电路。注意要有与 CPU 和存储器的接口信号，其他策略自己决定。
4. 在写透策略中，不管 Cache 是否命中，每次写数据操作都要写存储器。本章给出的例子是 CPU 等待写存储器操作完成。试设计一个带有写缓冲区的 Cache，使得 CPU 不必等待。
5. 试调查虚地址 Cache 的原理与设计。

6. 试用 Verilog HDL 设计一个四路组相联的 TLB 电路并对电路进行仿真。
7. 试设计一个 CPU，使其能执行以下 MIPS 指令：mfc0、mtc0、tlbp、tlbr、tlbwi 和 tlbwr。
8. 用踪迹驱动模拟或执行驱动模拟的方法定量评价各种结构的 Cache，包括映像进制、Cache 的容量、块的大小、替换策略、写策略等。

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。

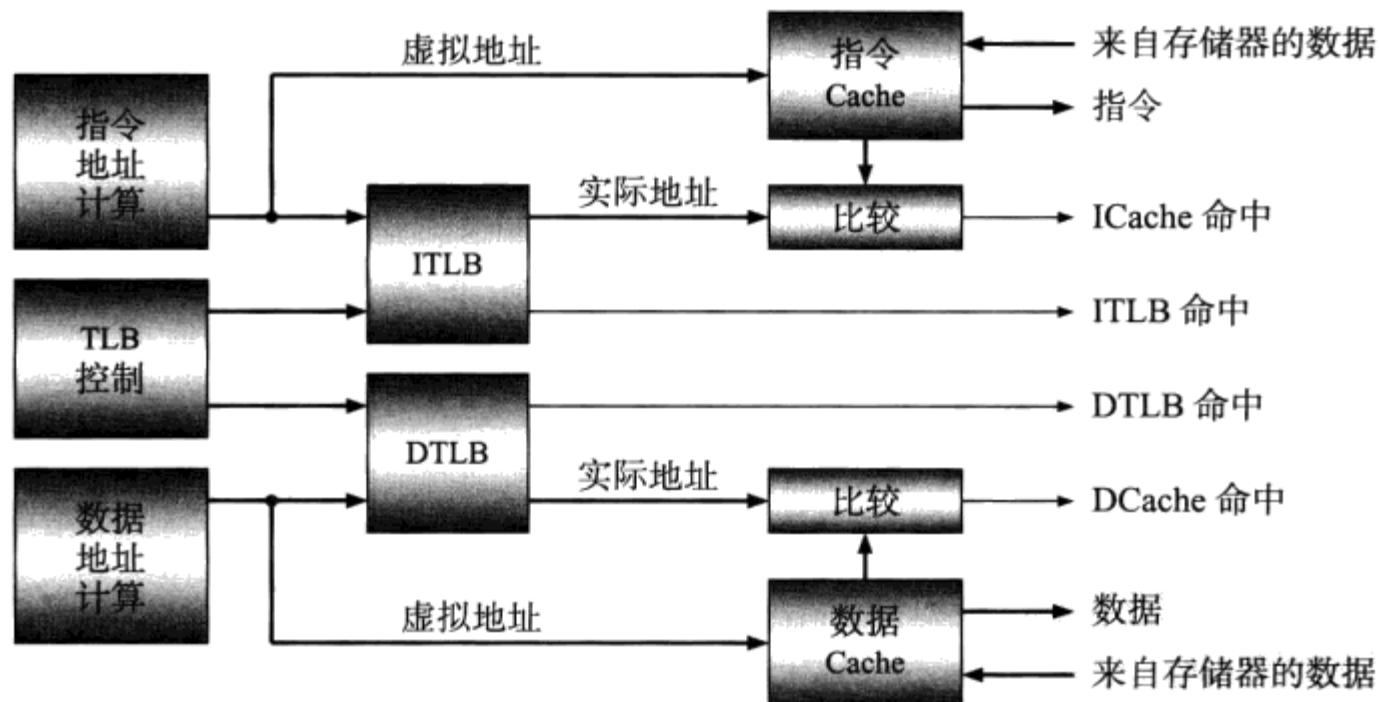


图 13.1 TLB 与 Cache 的连接

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。

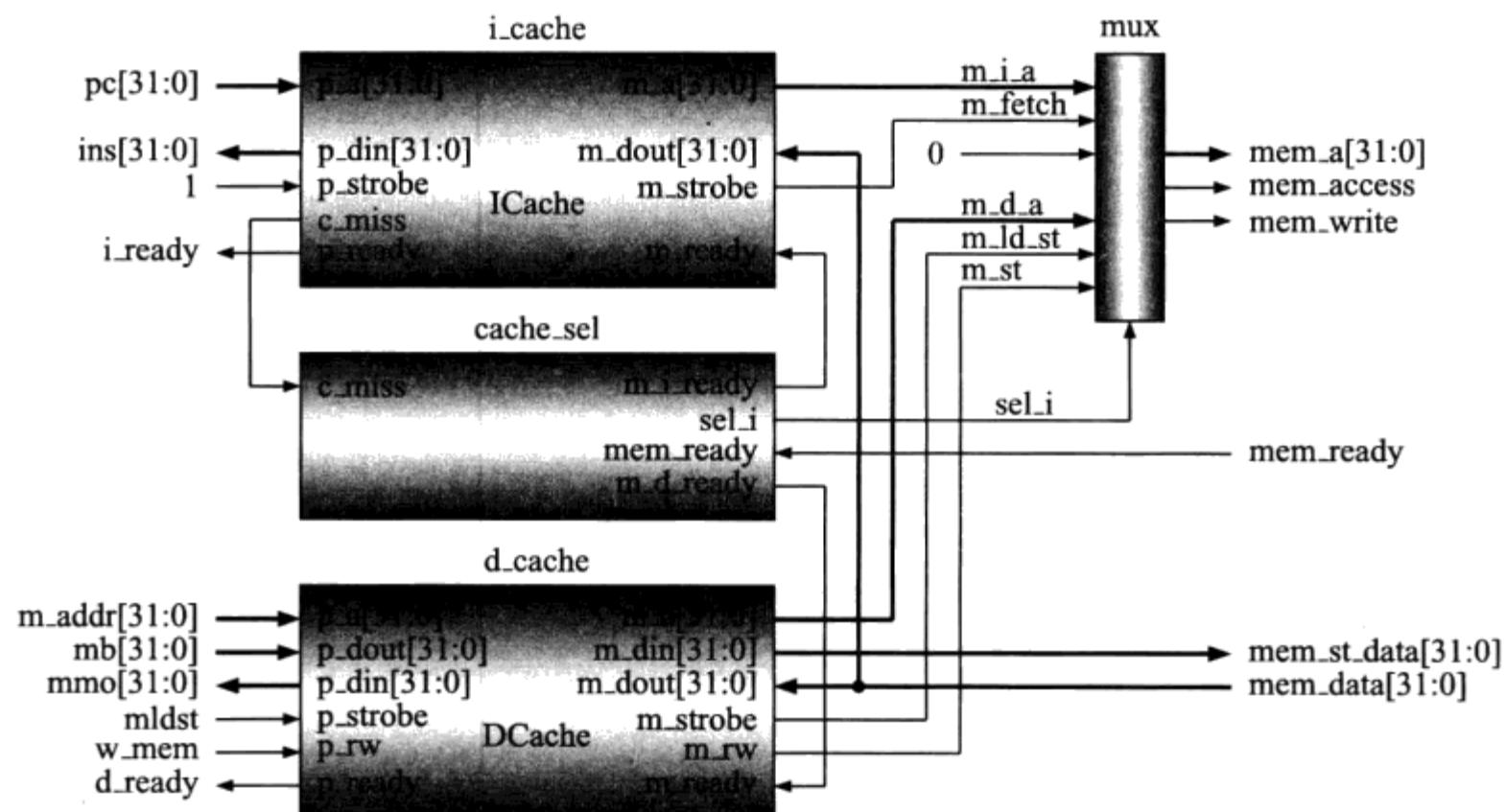


图 13.2 数据 Cache、指令 Cache 和存储器接口

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。

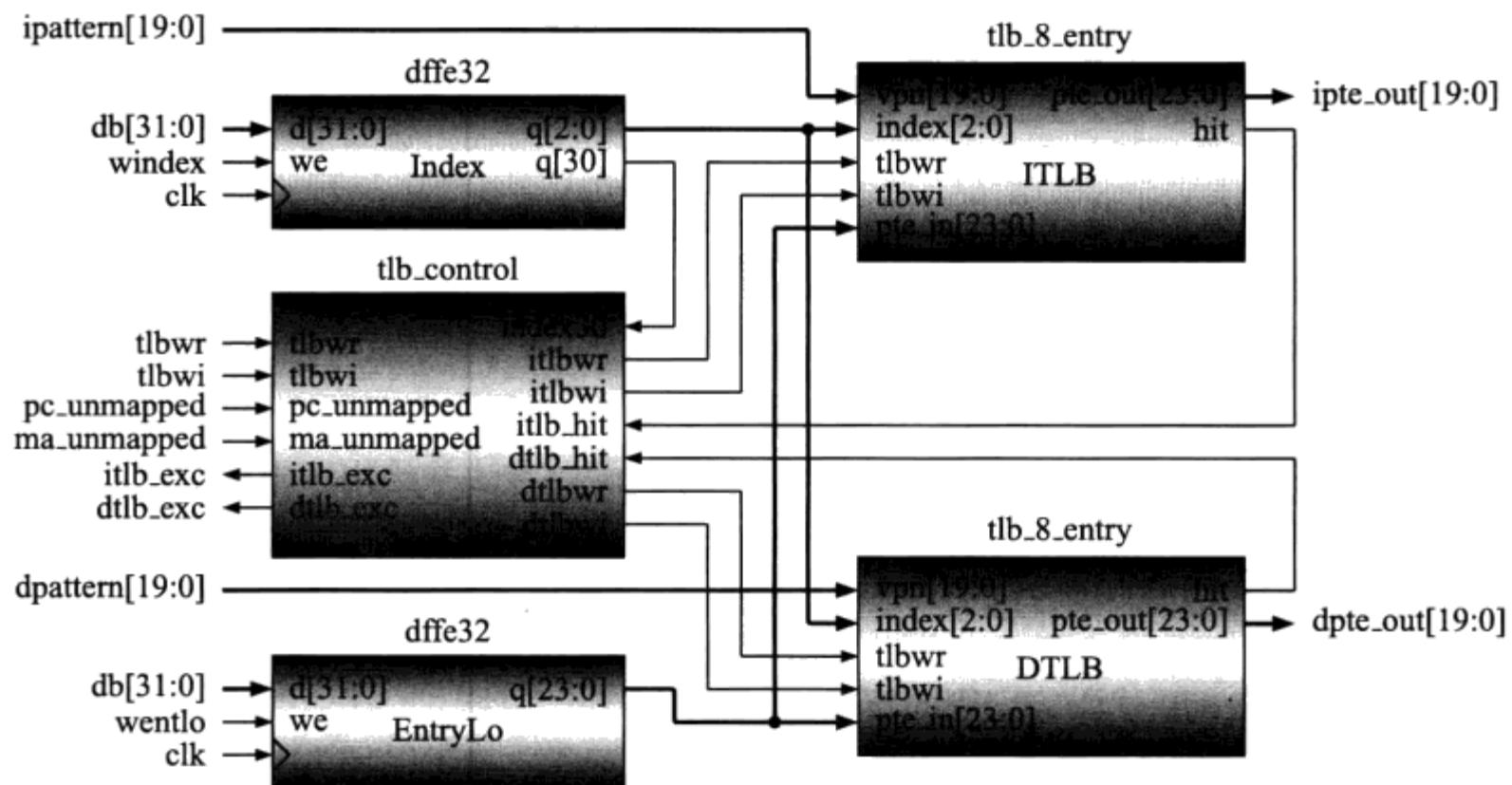


图 13.3 ITLB 和 DTLB

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。

Index (#0)	31 30 29	3 2 1 0
	D	Index
EntryLo (#2)	31 24 23 22 21 20 19	0
	V D C	PFN
Context (#4)	31 22 21	2 1 0
	PTEBase	BadVPN
EntryHi (#9)	31 20 19	0
	ProcessID	VPN
Status (#12)	31 8 7	0
		ExcEna
Cause (#13)	31 7 6	2 1 0
		ExcCode 0
EPC (#14)	31	0
	EPC	

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。

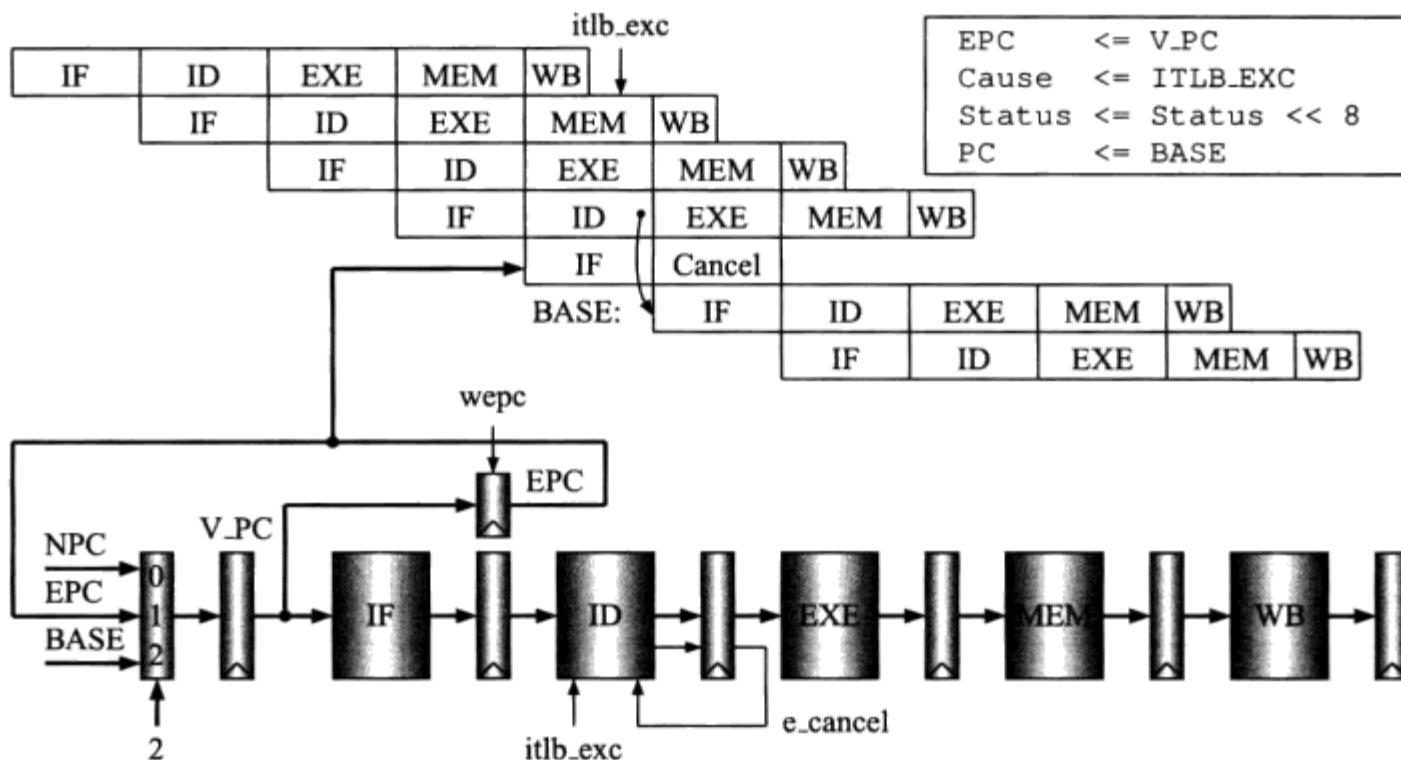


图 13.5 一般情况下的 ITLB_EXC

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。

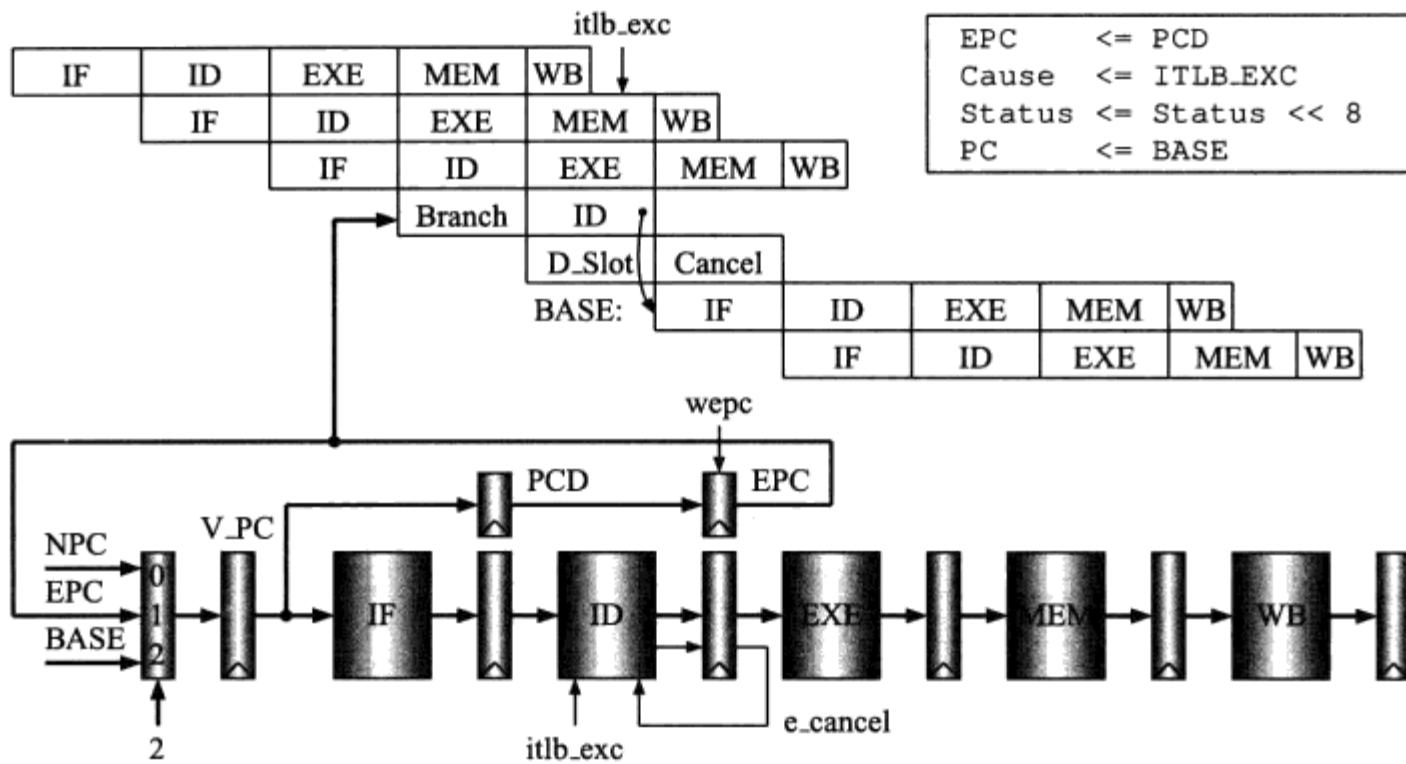


图 13.6 处在延迟槽的指令引起的 ITLB_EXC

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。

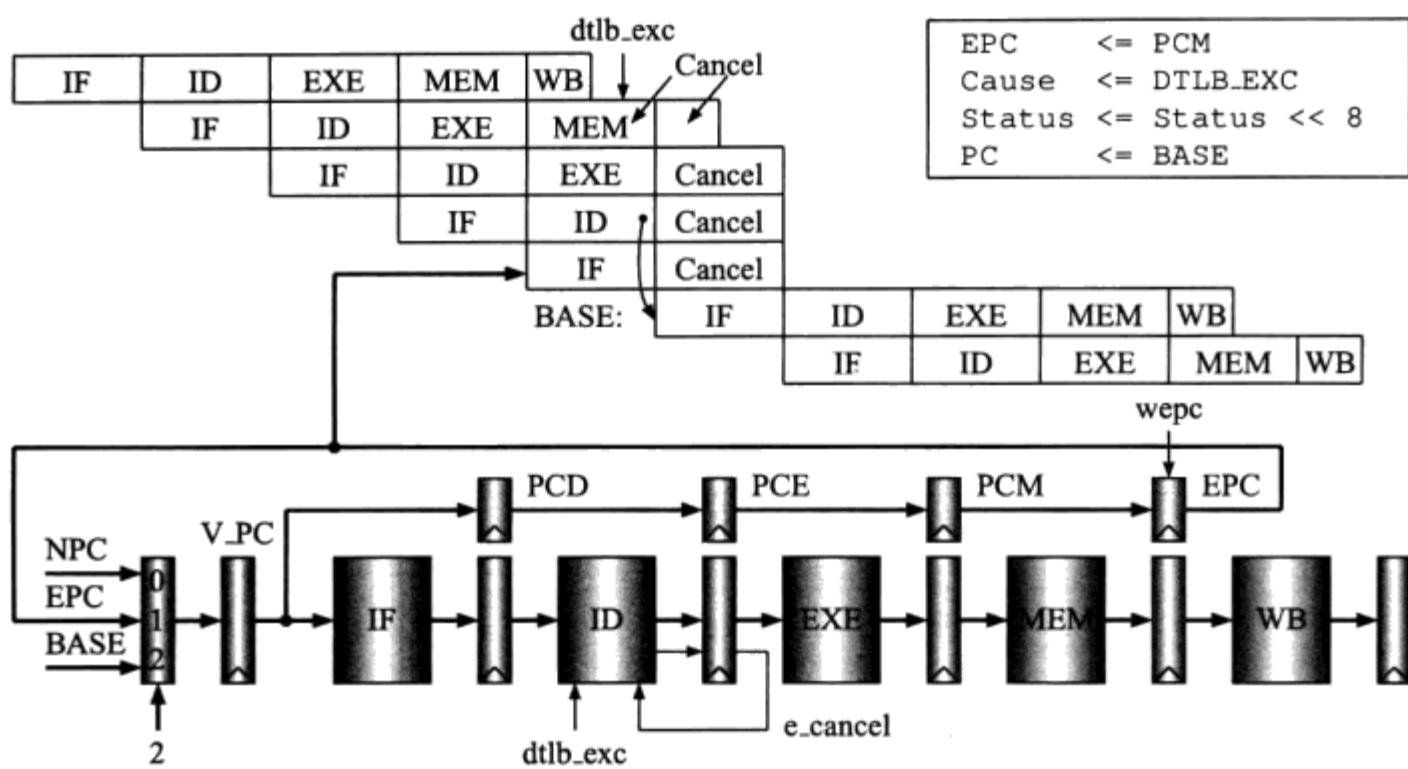


图 13.7 一般情况下的 DTLB_EXC

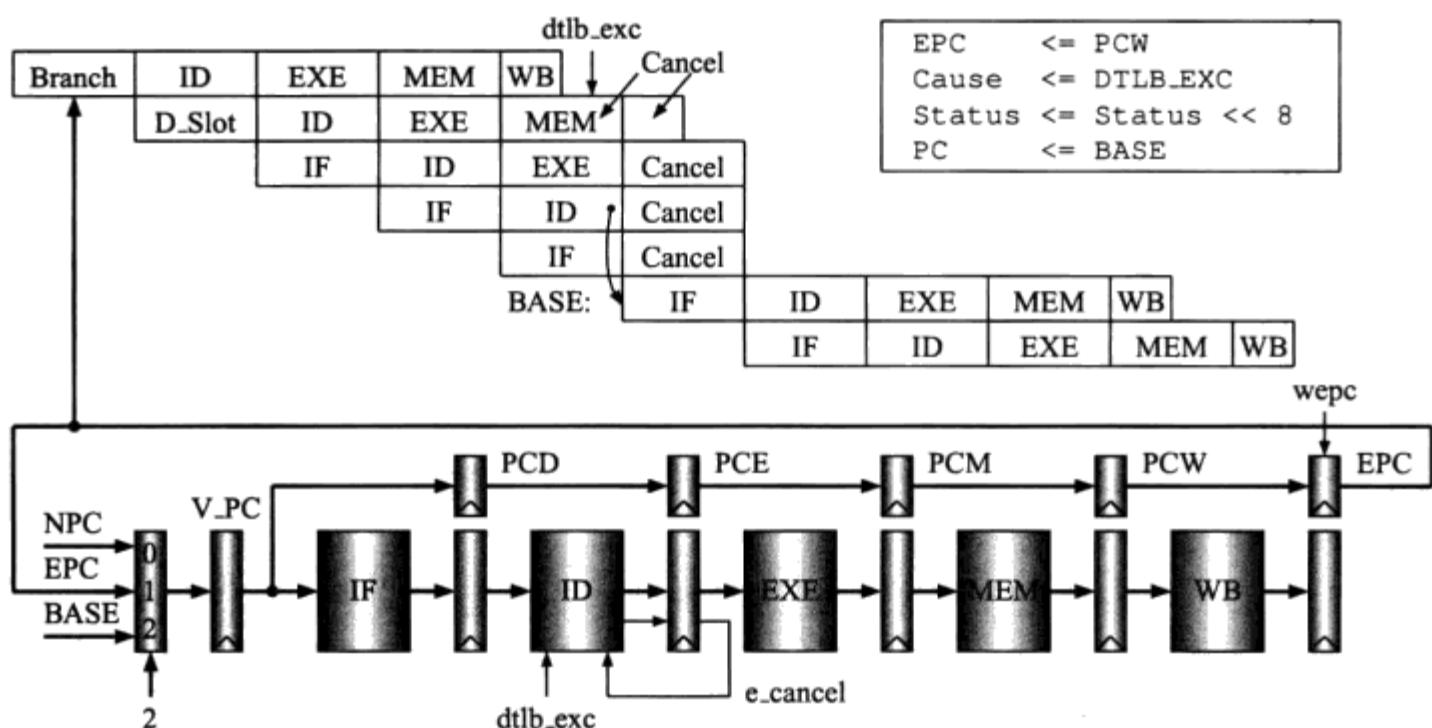


图 13.8 处在延迟槽的指令引起的 DTLB_EXC

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。

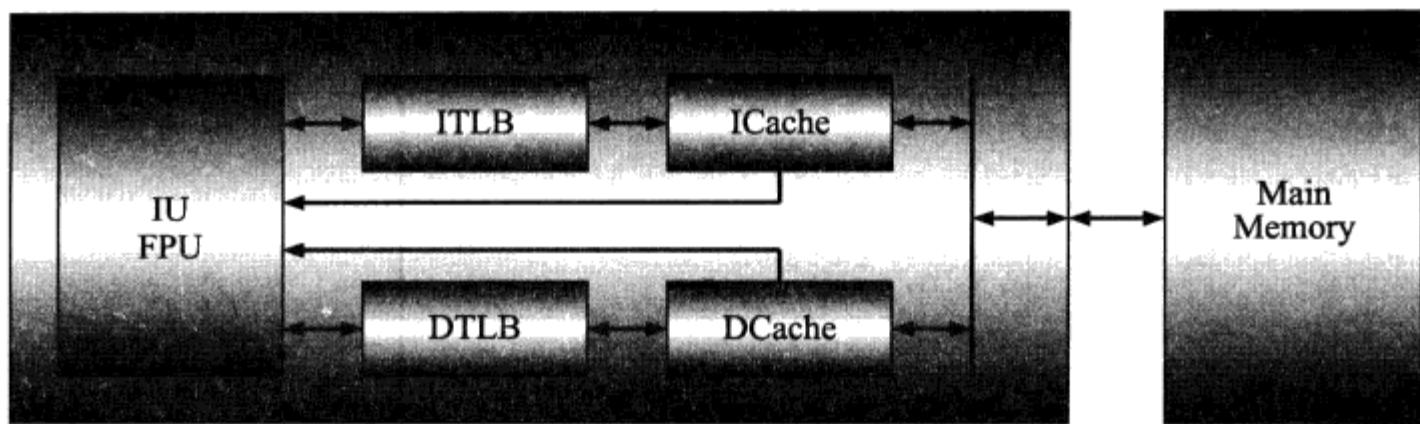


图 13.9 带有 Cache 及 TLB 的 CPU 模块图

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_swcl, 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_swcl, 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_swcl, 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_swcl, 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_swcl,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,fasmlds;
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,
                  fasmlds,v_pc,pc,inst,ealu,malu,walu,
                  stall_lw,stall_fp,stall_lwc1,stall_swcl,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,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,fasmlds,

```

```
v_pc,pc,inst,ealu,malu,walu,
stall_lw,stall_fp,stall_lwcl,stall_swcl,
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      elw,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,fasmlds;
output      stall_lw,stall_fp,stall_lwcl,stall_swcl;
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
dfffe32 pc_4_r (pc4, clock,resetn,wpcir&no_cache_stall,dpc4); // PC4
dfffe32 inst_r (ins, clock,resetn,wpcir&no_cache_stall,inst); // IR
dfffe32 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,fwdfe,fwdfe,wfpr,                      // used by iu
        fwdla,fwdlb,fwdfa,fwdfb,fc,wf,fasmnds,    // used by fpu
        stall_lw,stall_fp,stall_lwc1,stall_swcl,   // 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
dfffe32 c0_Index (db,clock,resetn,windex&no_cache_stall,index); // index
dfffe32 c0_Entlo (db,clock,resetn,wentlo&no_cache_stall,entlo); // entlo
dfffe32 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;
dfffe32 c0_Status (sta_in,clock,resetn,wsta&no_cache_stall,sta); // sta
dfffe32 c0_Cause  (cau_in,clock,resetn,wcau&no_cache_stall,cau); // cau
dfffe32 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;
  end

```

```

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/swcl 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 lwcl in wb stage
    wwreg    <= mwreg & ~dtlb_exc; // cancel lw in wb stage
    wm2reg   <= mm2reg;       wmo      <= mmo;
    walu     <= malu;         wrn      <= mrn;
    pcw      <= pcw;          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
                elw, 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, fwdf, wfpr,                          // for lwcl, swcl
                fwdla, fwdlb, fwdfa, fwdfb, fc, wf, fasmds,      // used by fpu
                stall_lw, stall_fp, stall_lwcl, stall_swcl,     // 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 elw, e1n, 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, fwdf, wfpr;
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_lwcl, stall_swcl; // 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_lwcl | i_swcl) & ~ecancel & ~dtlb_exc;
assign isbr = i_beq | i_bne | i_j | i_jal;
// itlb_exc dtlb_exc isbr 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_swcl,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_swcl, 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_swcl;
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
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

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_swcl;
assign sext    = i_addi | i_lw | i_sw | i_beq | i_bne | i_lwc1 | i_swcl;
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_swcl) & 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 hazards
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_lwcl & 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_lwcl = ewfpr & (i_fs & (ern == fs) | i_ft & (ern == ft));
assign swfp  = i_swcl; // select signal
assign fwdf  = swfp & e3w & (ft == e3n); // forward to id stage
assign fwdfa = swfp & e2w & (ft == e2n); // forward to exe stage
assign stall_swcl = swfp & elw & (ft == eln); // stall
assign wpcir = ~(stall_div_sqrt | stall_others);
wire stall_others = stall_lw | stall_fp | stall_lwcl | stall_swcl | 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
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 radices are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                        % otherwise specified, radices = 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: 3cla0000; % (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 radices are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                        % otherwise specified, radices = 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 radices are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                        % otherwise specified, radices = HEX %

CONTENT
BEGIN
    % physical address = 0x2000_0000 %
    0 : 20011100; %(20000000) addi r1,r0,0x1100 # address of data[0] %
    1 : C4200000; %(20000004) lwcl 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 radices are optional %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless %
                        % otherwise specified, radices = 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）。

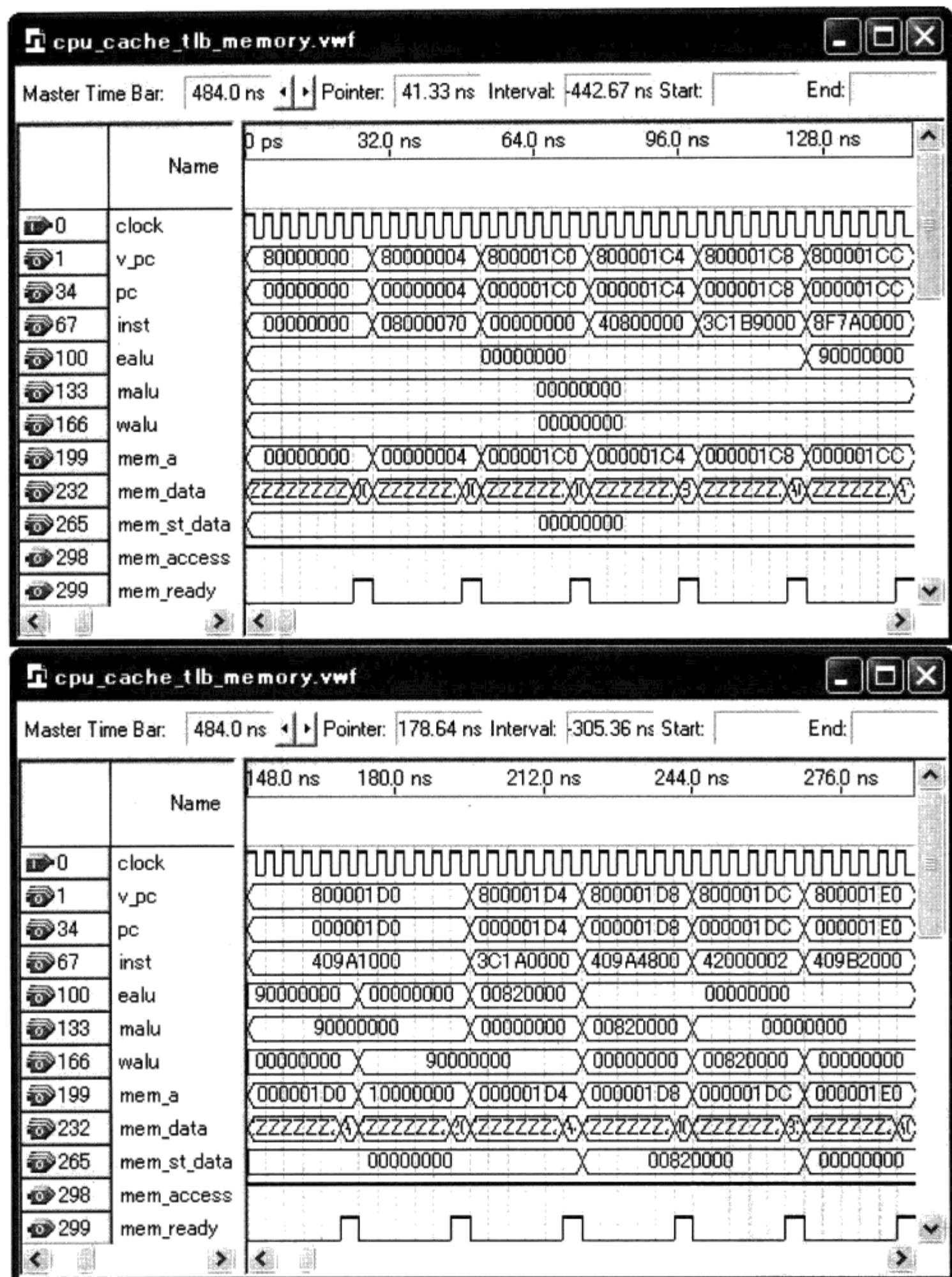


图 13.10 流水线 CPU 仿真波形图(复位后)

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。

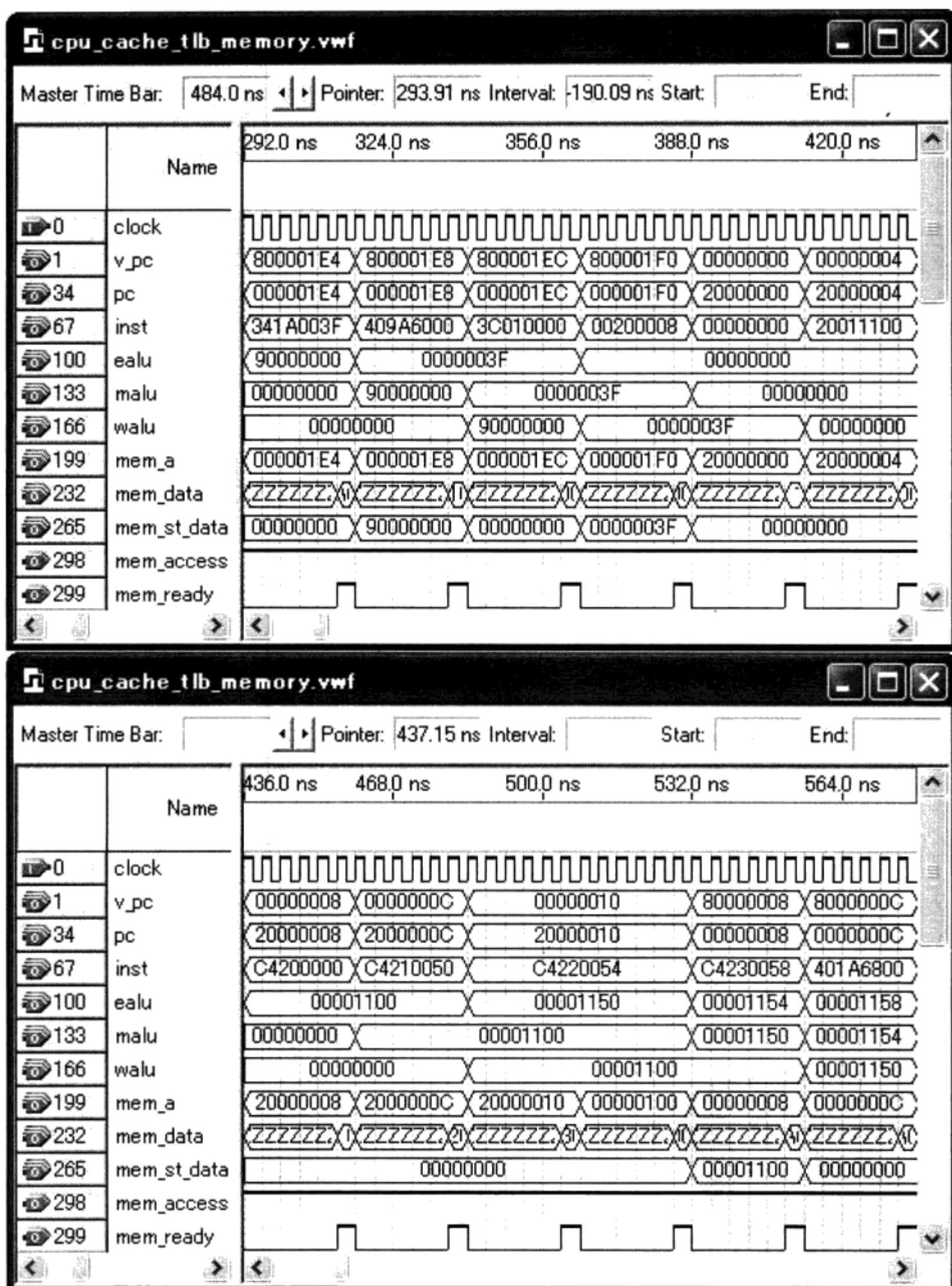


图 13.11 流水线 CPU 仿真波形图 (转用户程序)

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。

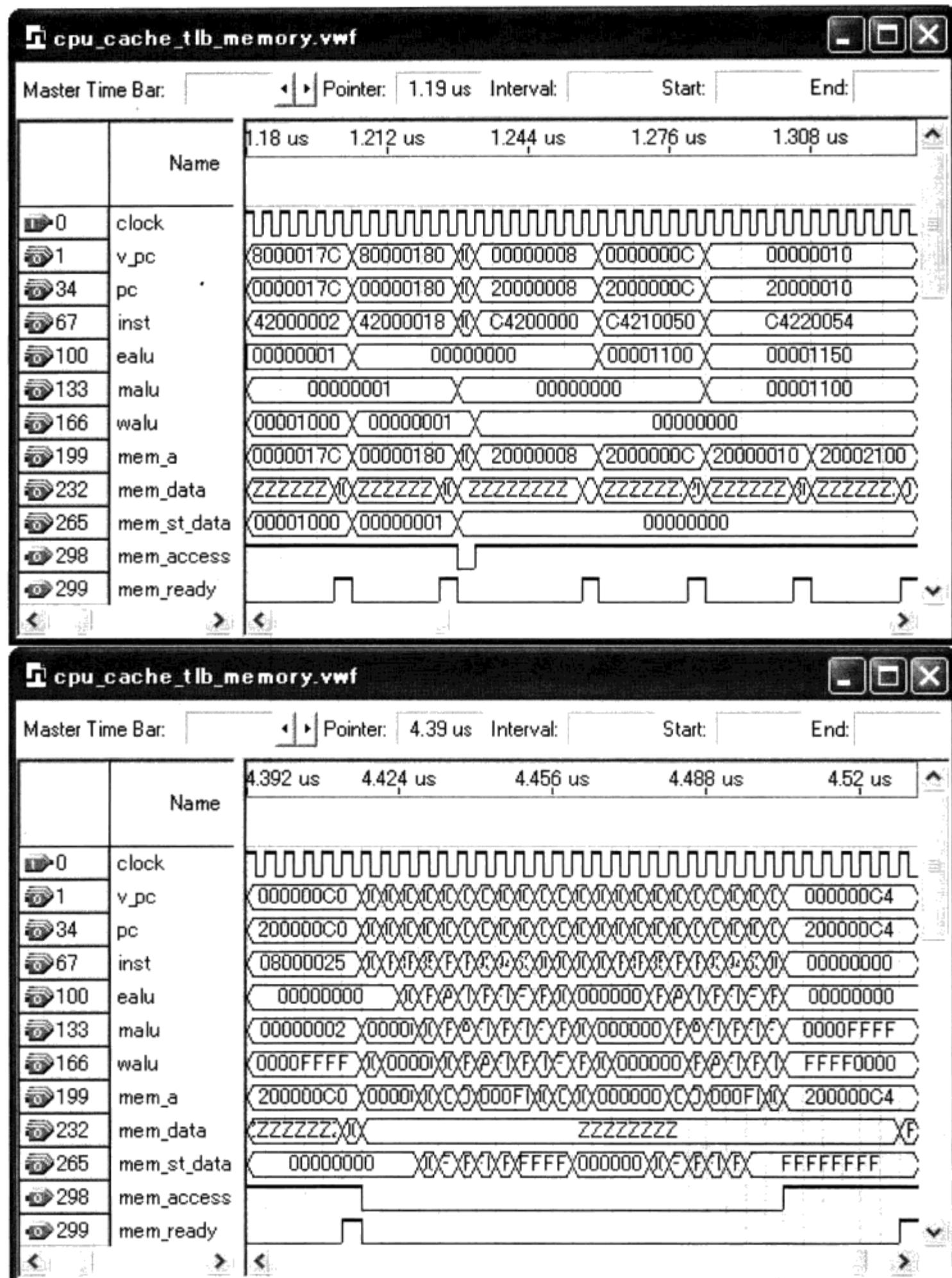


图 13.12 流水线 CPU 仿真波形图(从异常返回和指令 Cache 命中)

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。