

例 3.4 竞争条件

Ben 设计了一个新的 D 锁存器，并且他宣布这个设计比图 3-17 中的 D 锁存器更好。因为在这个电路中门的数量更少。他写出了真值表来确定输出 Q ，给定两个输入 D 和 CLK 以及锁存器的原来状态 Q_{prev} 。根据这个真值表，Ben 得出布尔表达式。他通过将输出 Q 反馈得到 Q_{prev} 。他的设计如图 3-18 所示。他的锁存器是否能正确工作，不考虑每个门的延迟？

解：如图 3-19 所示，当某些门比其他门的速度慢时，有竞争的条件（race condition）电路将导致电路故障。假设 $\text{CLK} = D = 1$ 。锁存器是透明的，将 D 传送到 Q 使 $Q = 1$ 。现在 CLK 下降。锁存器应该记住它原来的值，保持 $Q = 1$ 。但是，假设从 CLK 到 $\overline{\text{CLK}}$ 通过反相器的延迟比与门和或门的延迟长。那么在 $\overline{\text{CLK}}$ 上升前，结点 N1 和 Q 可能同时下降。在这种情况下，N2 将不能上升， Q 值将停留在 0。

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

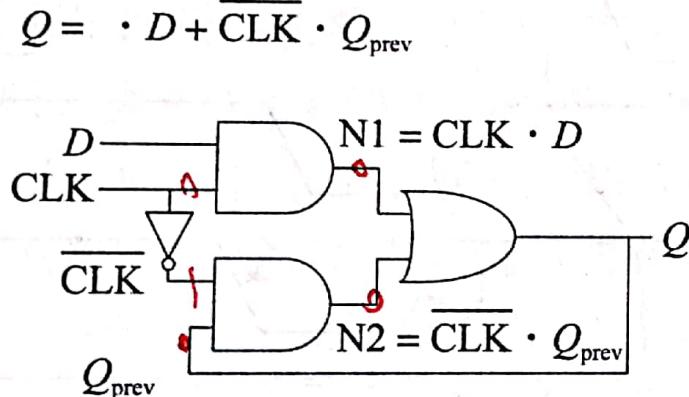


图 3-18 一个看似改进的 D 锁存器

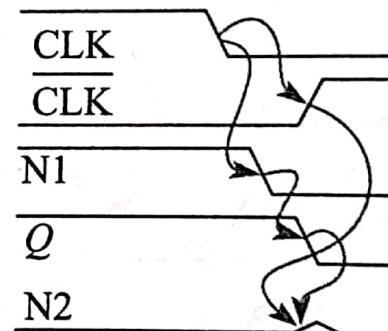


图 3-19 描述竞争条件的锁存器的波形



扫描全能王 创建

例 3.5 同步时序电路

~~图 3-21 中的哪些电路是同步时序电路?~~

解: 电路 a) 是组合逻辑电路, 不是时序逻辑电路, 因为它没有寄存器。电路 b) 是一个不带反馈回路的简单时序电路。电路 c) 既不是组合电路也不是时序电路, 因为它有一个锁存器, 这个锁存器既不是寄存器也不是组合逻辑电路。电路 d) 和 e) 是同步时序逻辑电路。它们是有有限状态机的两种形式, 将在 3.4 节中讨论。电路 f) 既不是组合电路也不是时序电路, 因为它有一个从组合逻辑电路的输出端反馈到同一逻辑电路的输入端的回路, 但是在回路上没有寄存器。电路 g) 是同步时序逻辑电路的流水线形式, 将在 3.6 节中讨论。电路 h) 严格地说不是同步时序电路, 因为两个寄存器的时钟信号不同, 它们有 2 个反相器的延迟。

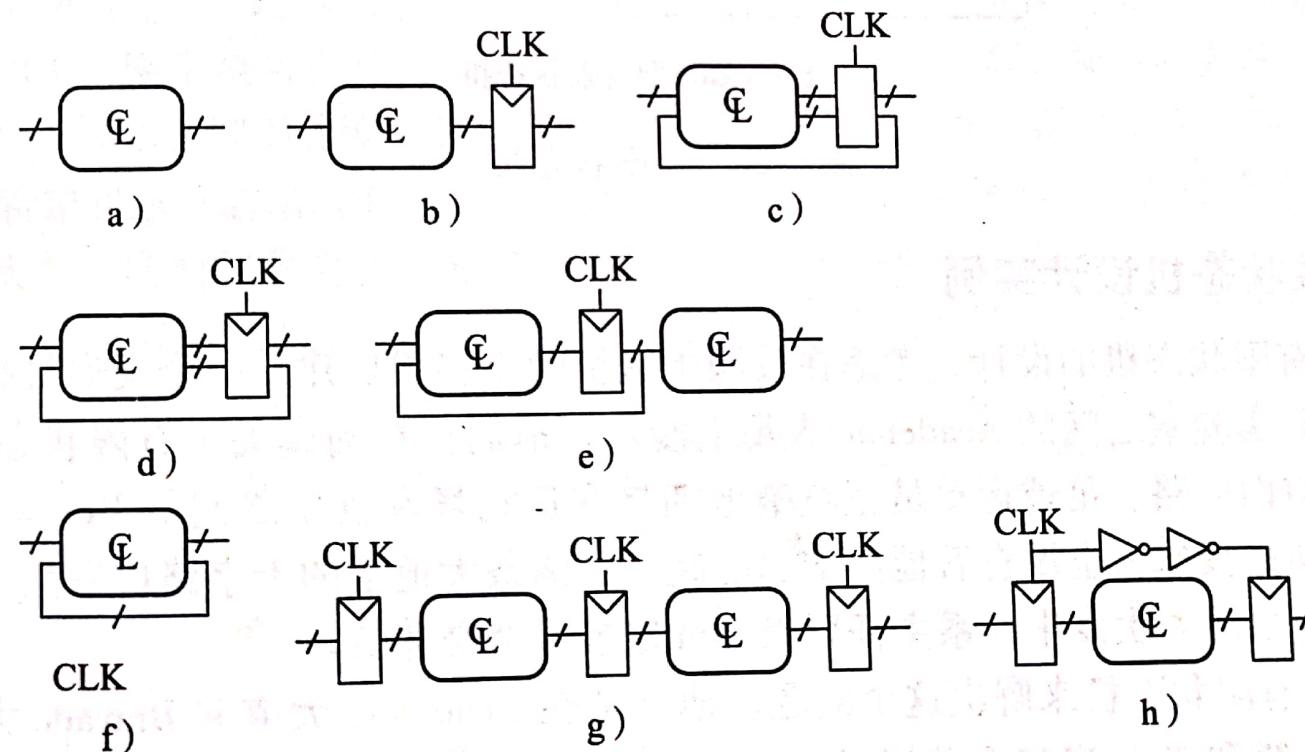


图 3-21 例子电路



扫描全能王 创建

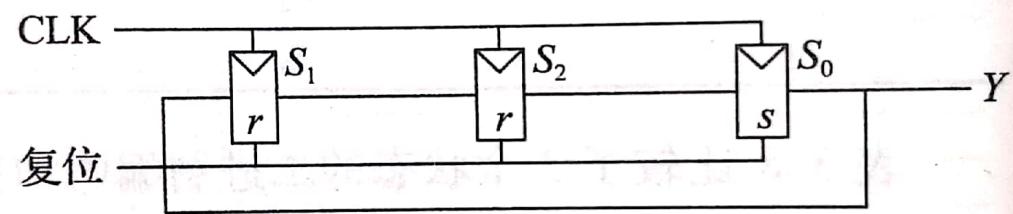
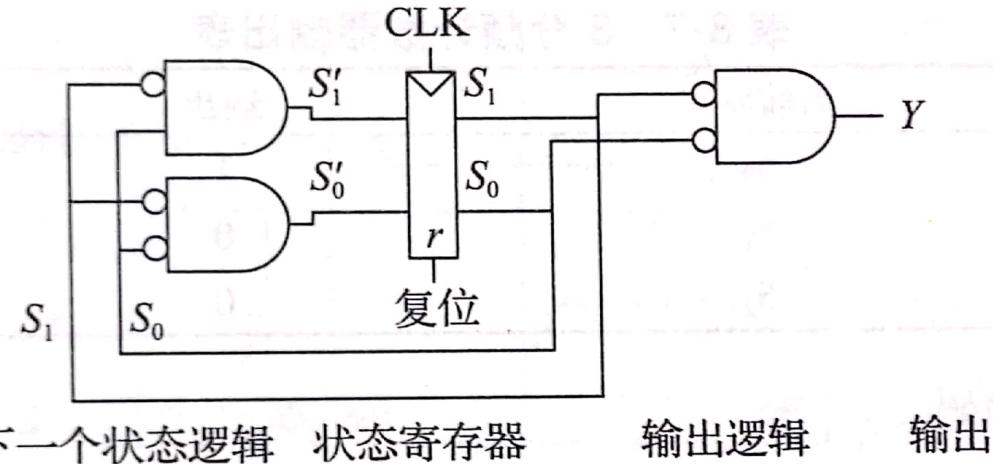


图 3-29 3 分频计数器电路



扫描全能王 创建

阻塞和非阻塞赋值准则

SystemVerilog

1) 使用 `always_ff @ (posedge clk)` 和非阻塞赋值描述同步时序逻辑。

```
always_ff @(posedge clk)
begin
    n1 <= d; // 非阻塞
    q <= n1; // 非阻塞
end
```

2) 使用连续赋值描述简单组合逻辑。

```
assign y = s ? d1 : d0;
```

3) 使用 `always_comb` 和阻塞赋值描述复杂组合逻辑将很有帮助。

```
always_comb
begin
    p = a ^ b; // 阻塞
    g = a & b; // 非阻塞
    s = p ^ cin;
    cout = g | (p & cin);
end
```

4) 不要在多于 1 个 `always` 语句或者连续赋值语句中对同一个信号赋值。

VHDL

1) 使用 `process (clk)` 和非阻塞赋值描述同步时序逻辑。

```
process(clk) begin
    if rising_edge(clk) then
        n1 <= d; -- 非阻塞
        q <= n1; -- 非阻塞
    end if;
end process;
```

2) 使用 `process` 语句外的并发赋值描述简单组合逻辑。

```
y <= d0 when s = '0' else d1;
```

3) 使用 `process (all)` 描述复杂组合逻辑将会有帮助。使用阻塞赋值对内部变量进行赋值。

```
process(all)
    variable p, g: STD_LOGIC;
begin
    p := a xor b; -- 阻塞
    g := a and b; -- 阻塞
    s <= p xor cin;
    cout <= g or (p and cin);
end process;
```

4) 不要在多于 1 个 `process` 语句或者并发赋值语句中对同一个信号赋值。



扫描全能王 创建

Moore 型状态机和 Mealy 型状态机的电路原理图如图 3-31 所示。每种状态机的时序图如图 3-32 所示。两种状态机的状态序列不同。然而，Mealy 型状态机的输出上升要早一个周期。这是因为它的输出直接响应输入，而不需要等待状态的变化。如果 Mealy 型状态机的输出通过触发器产生延迟，那么它的输出将与 Moore 型状态机一样。在选择有限状态机设计类型时，需要考虑何时需要到输出响应。

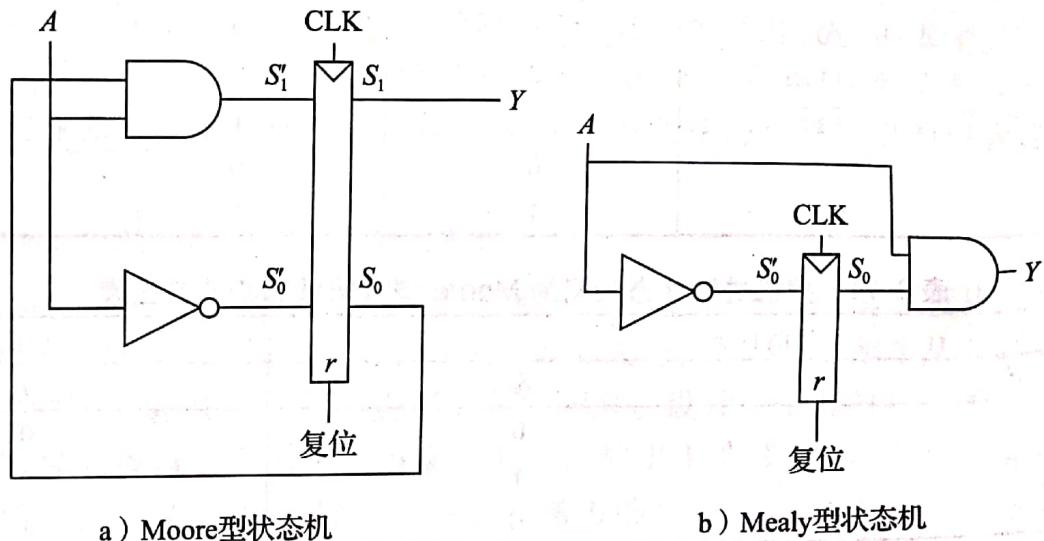


图 3-31 有限状态机的电路原理图

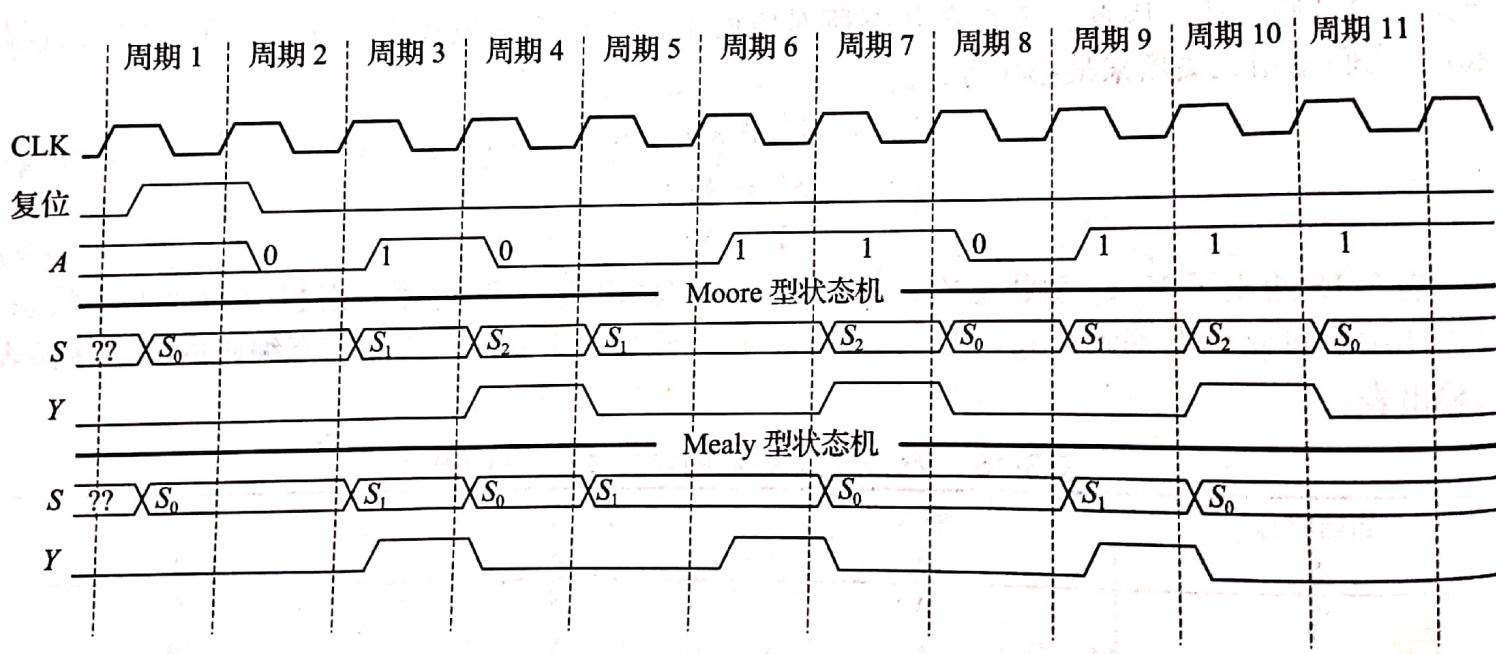


图 3-32 Moore 型状态机和 Mealy 型状态机的时序图



扫描全能王 创建

HDL 例 4.35 参数化的 $N:2^N$ 译码器

SystemVerilog

```
module decoder
#(parameter N=3)
  (input logic [N-1:0] a,
   output logic [2**N-1:0] y);

  always_comb
    begin
      y=0;
      y[a]=1;
    end
endmodule
```

$2^{**} N$ 代表 2^N 。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in STD_LOGIC_VECTOR(N-1 downto 0);
        y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

$2^{**} N$ 代表 2^N 。



扫描全能王 创建

产生一个由 2 输入与门级联构成的 N 输入 AND 功能。当然，对于这个应用中，使用缩位运算符将更简单明了，但该例子阐述了硬件生成器的通用原理。

使用 generate 语句必须注意，它很容易不经意地生成大量的硬件。

HDL 例 4.36 参数化的 N 输入与门

SystemVerilog

```
module andN
  #(parameter width=8)
  (input logic [width-1:0] a,
   output logic         y);
  genvar i;
  logic [width-1:0] x;
  generate
    assign x[0]=a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i]=a[i] & x[i-1];
    end
  endgenerate
  assign y=x[width-1];
endmodule
```

for 语句循环通过 $i = 1, 2, \dots, width - 1$ 以便产生许多连续的与门。在 generate for 循环中的 begin 后面必须有 “:” 和一个任意的标识(在这个例子中是 forloop)。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity andN is
  generic(width: integer := 8);
  port(a: in STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;
architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
  x(0) <= a(0);
  gen: for i in 1 to width-1 generate
    x(i) <= a(i) and x(i-1);
  end generate;
  y <= x(width-1);
end;
```

生成循环变量 i 不需声明。

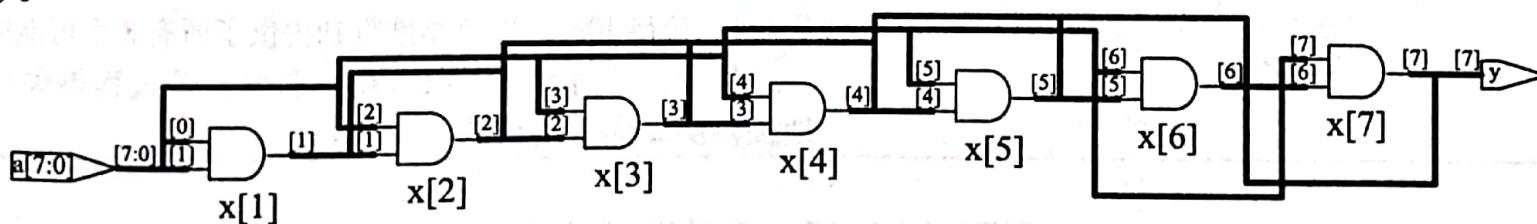


图 4-30 andN 综合后的电路



扫描全能王 创建

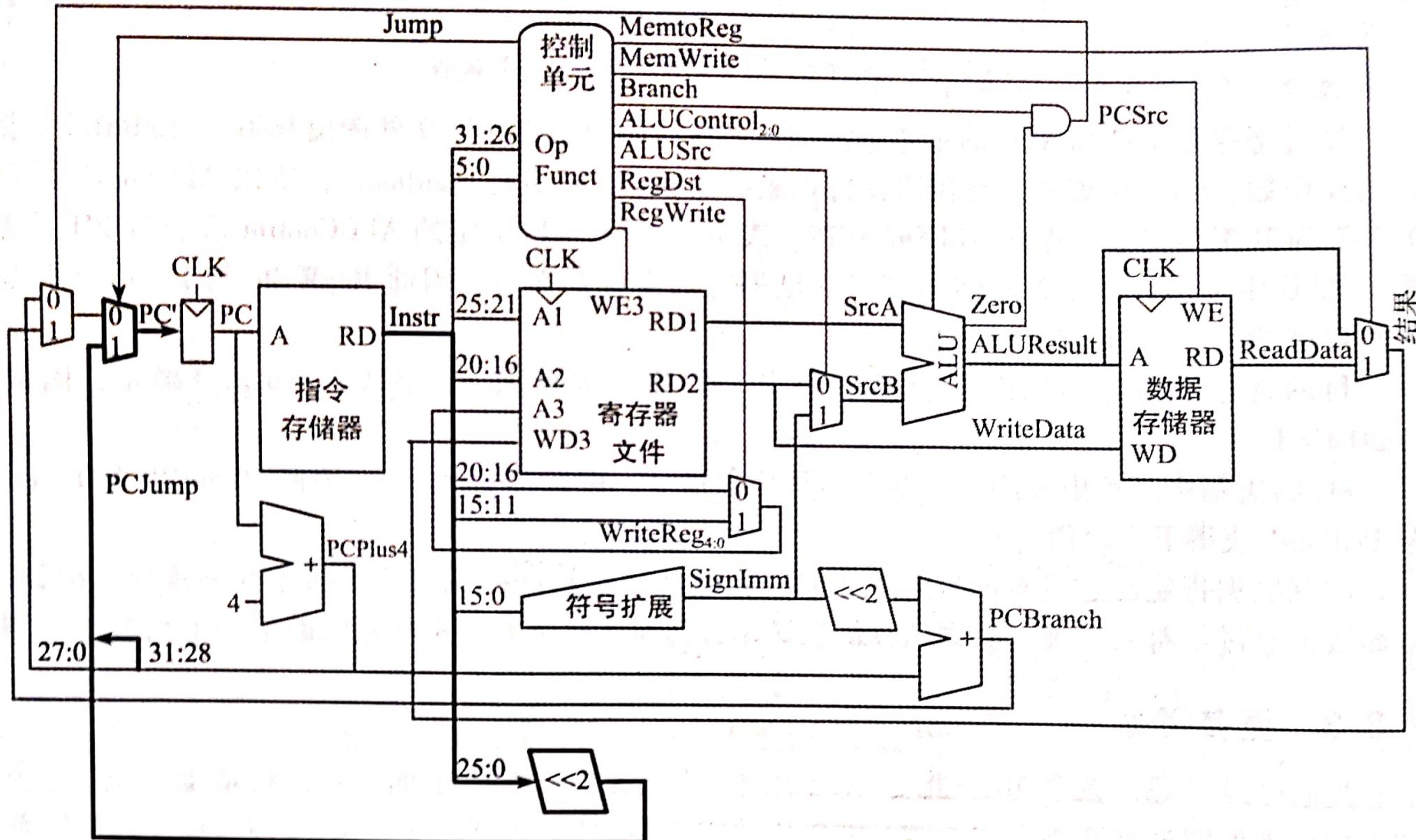


图 7-14 扩展单周期 MIPS 数据路径来支持 `j` 指令



扫描全能王 创建

表 7-5 扩展主译码器真值表来支持 j 指令

指令	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R 类型	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1



扫描全能王 创建

表 7-1 ALUOp 编码

ALUOp	含义	ALUOp	含义
00	加法	10	依赖于 funct 字段
01	减法	11	无定义

表 7-2 为 ALU 译码器的真值表。3 个 ALUControl 信号的含义如表 5-1 所示。由于 ALUOp 不可能为 11，所以真值表中不可能使用无关项 X1 和 1X，而使用 01 和 10 来简化逻辑。当 ALUOp 为 00 或 01 时，ALU 应该为加法或减法。当 ALUOp 为 10 时，译码器检查 funct 字段来确定 ALUControl 信号。注意我们实现的 R 类型指令中，funct 字段的最高两位总是 10，因此可以忽略它们来简化设计。

表 7-2 ALU 译码器真值表

ALUOp	Funct	ALUControl
00	X	010(加)
X1	X	110(减)
1X	100000(add)	010(加)
1X	100010(sub)	110(减)
1X	100100(and)	000(与)
1X	100101(or)	001(或)
1X	101010(slt)	111(小于置位)



扫描全能王 创建

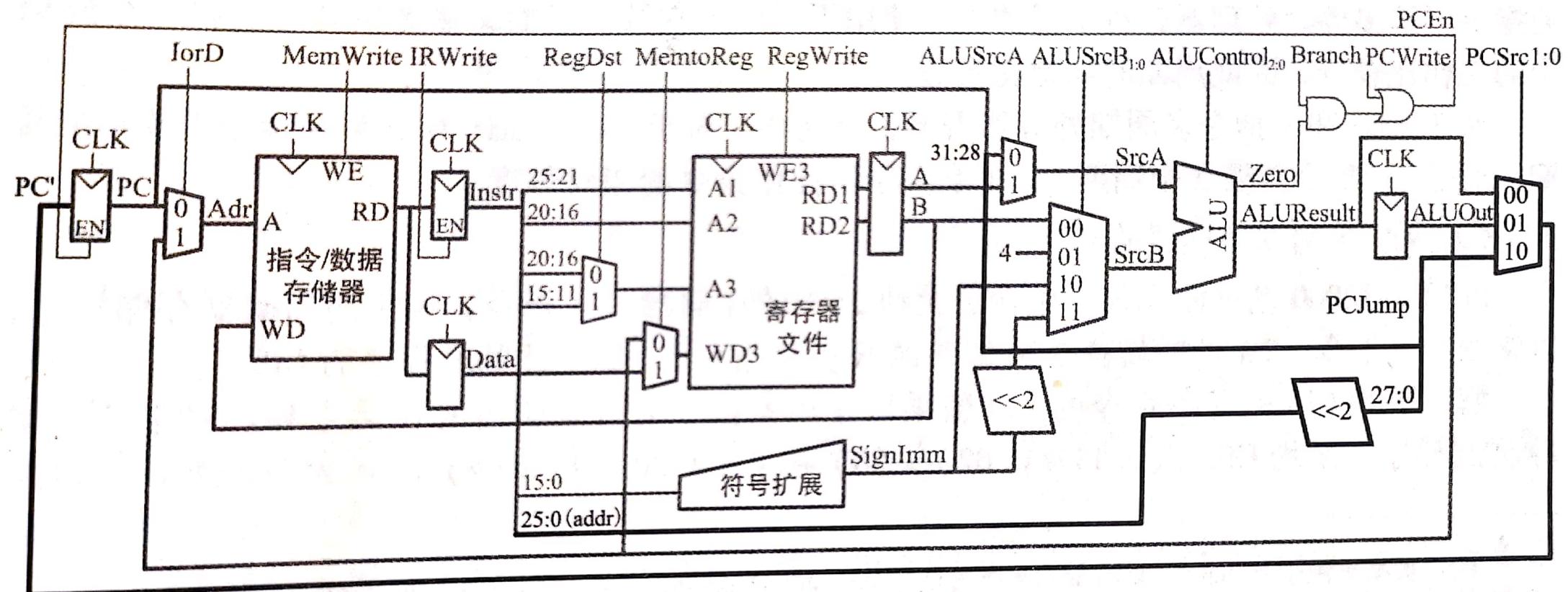
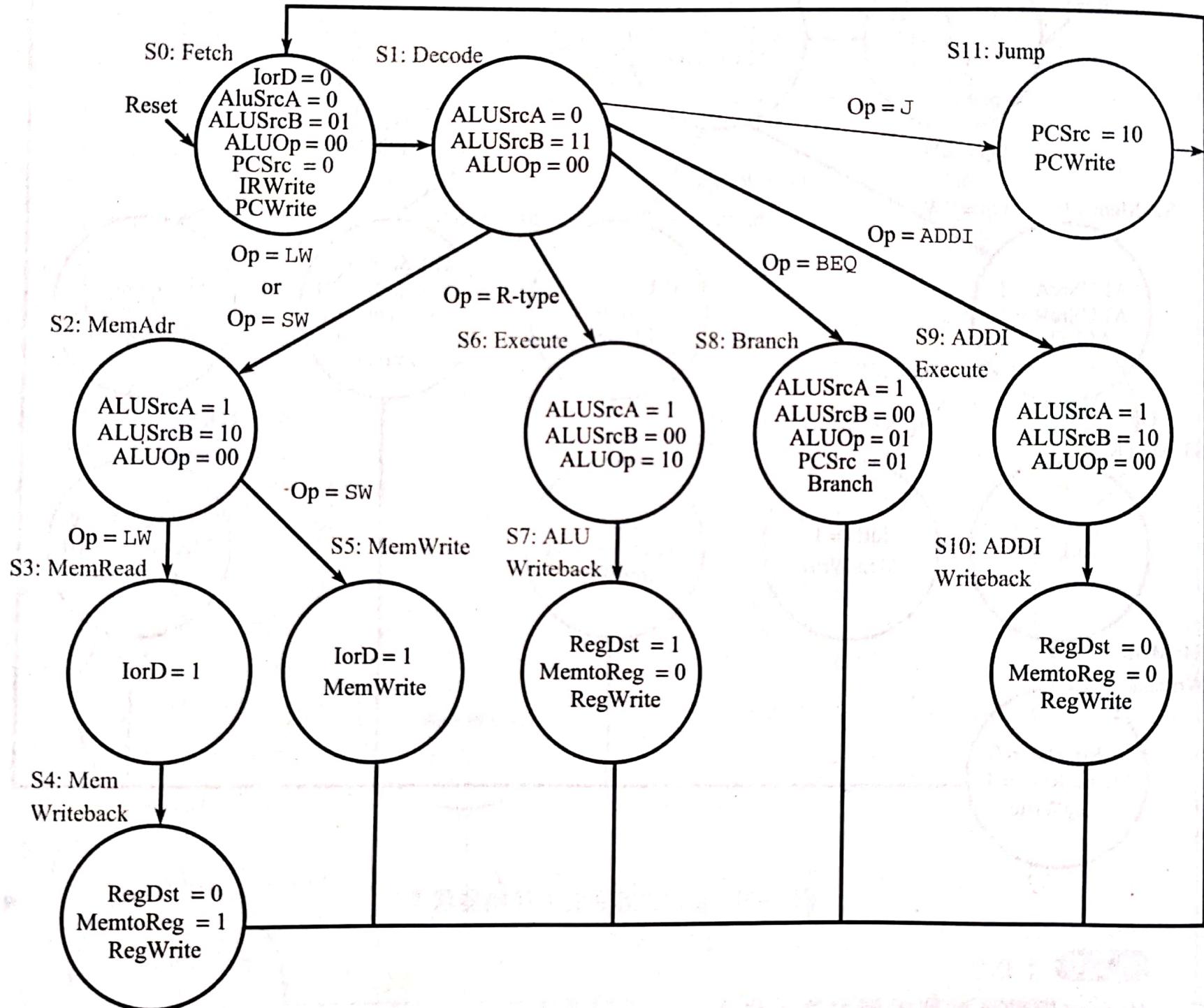


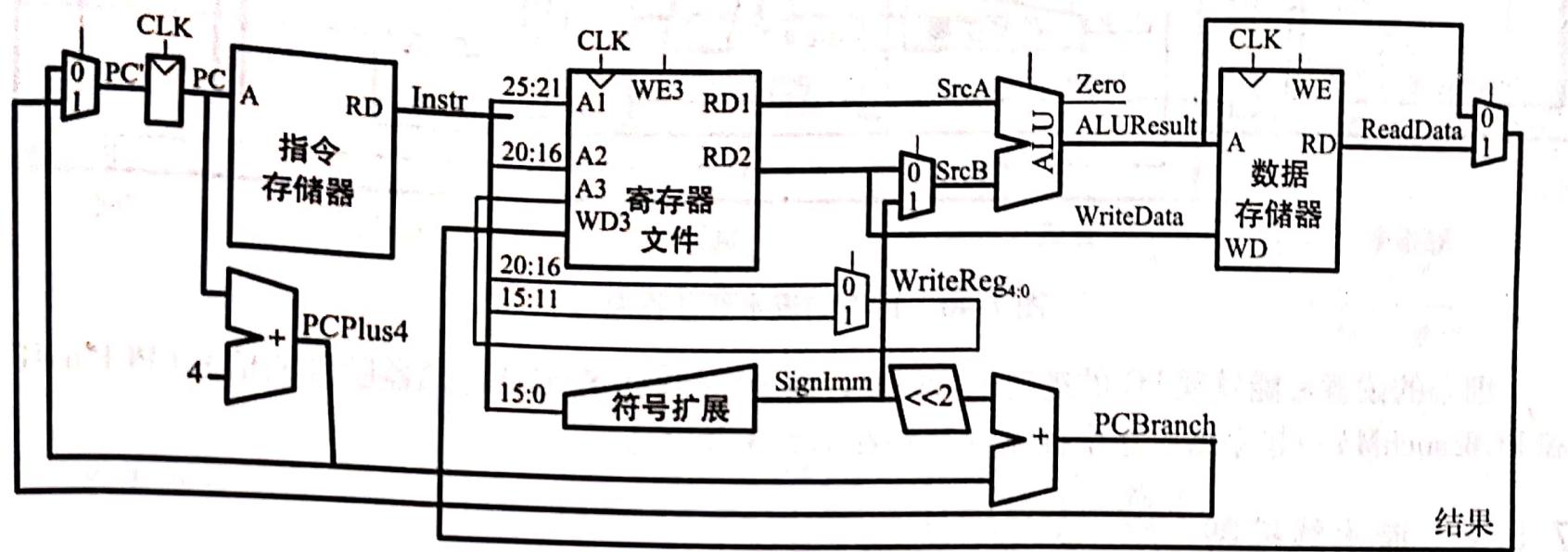
图 7-41 支持 j 指令的扩展的多周期 MIPS 数据路径



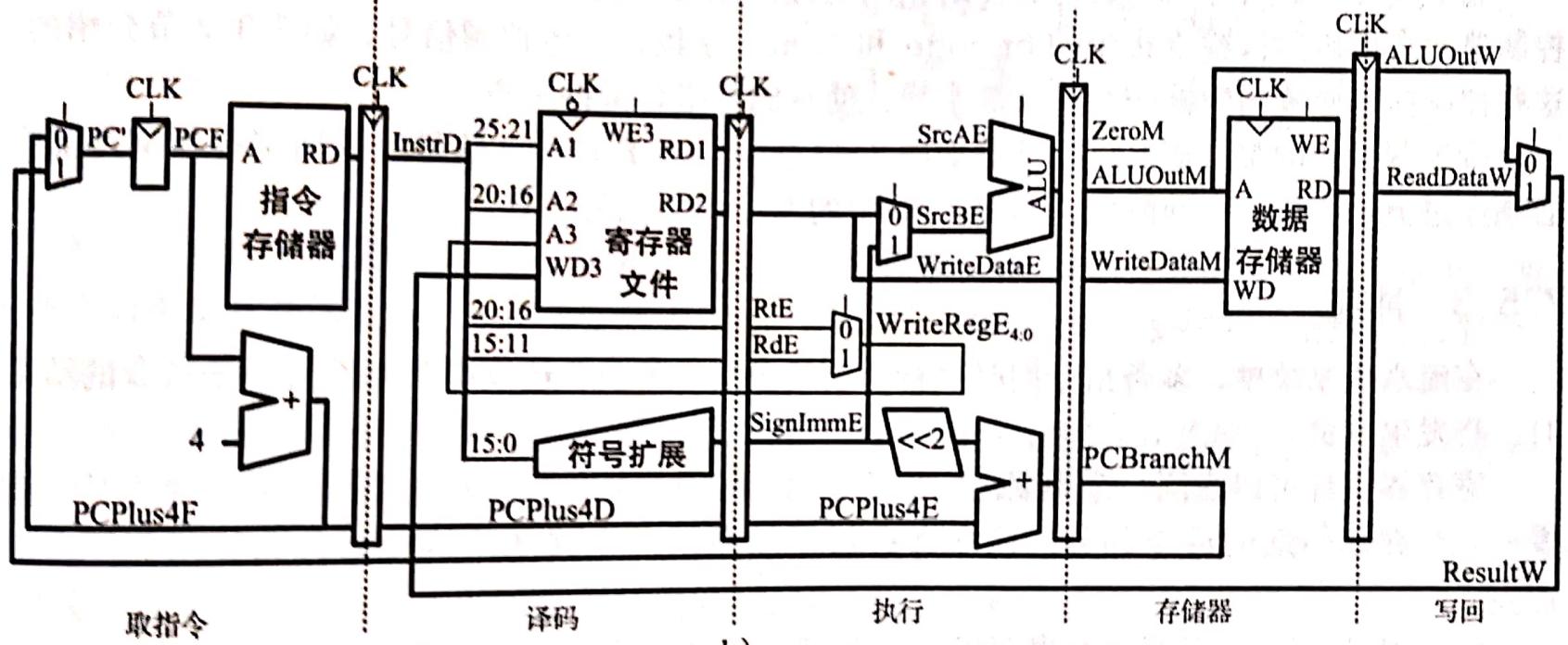
扫描全能王 创建



扫描全能王 创建



a)



b)

图 7-45 单周期和流水线数据路径



扫描全能王 创建

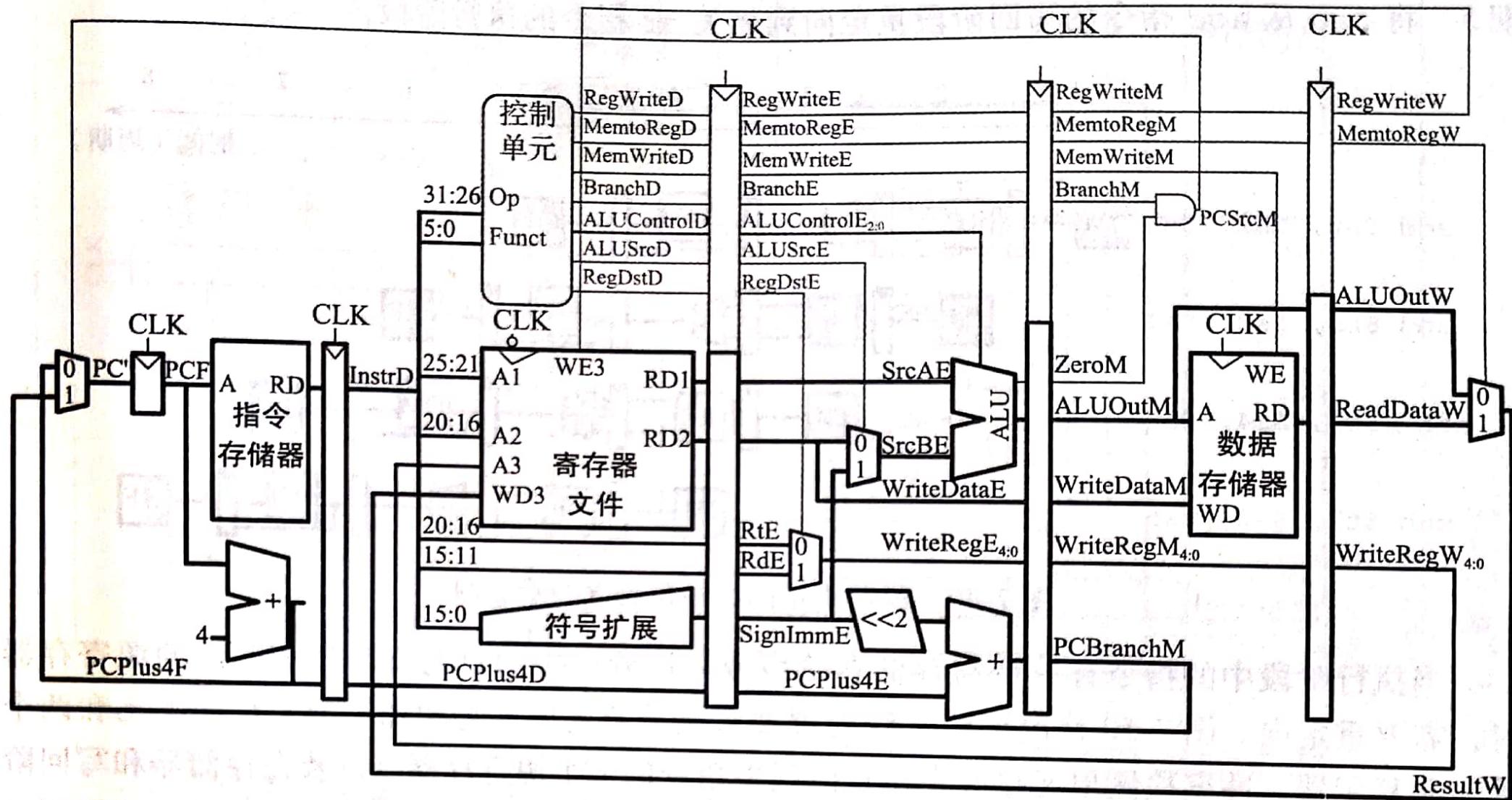


图 7-47 带控制信号的流水线处理器



扫描全能王 创建

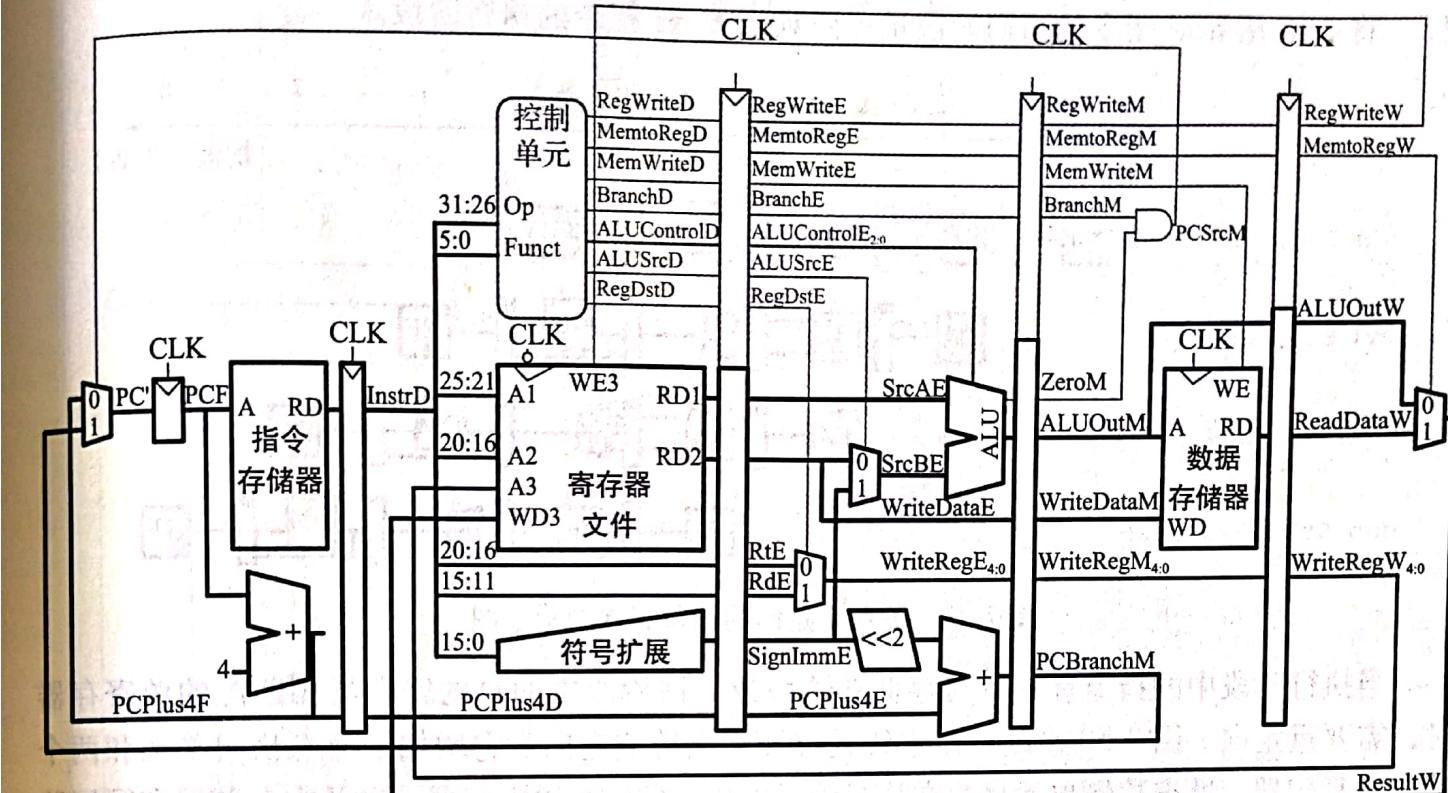


图 7-47 带控制信号的流水线处理器

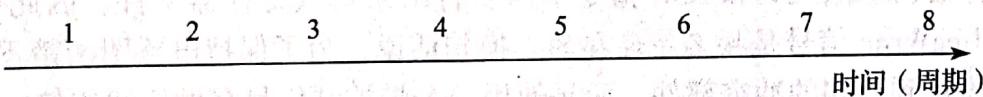


图 7-48 说明冲突的抽象流水线图



扫描全能王 创建

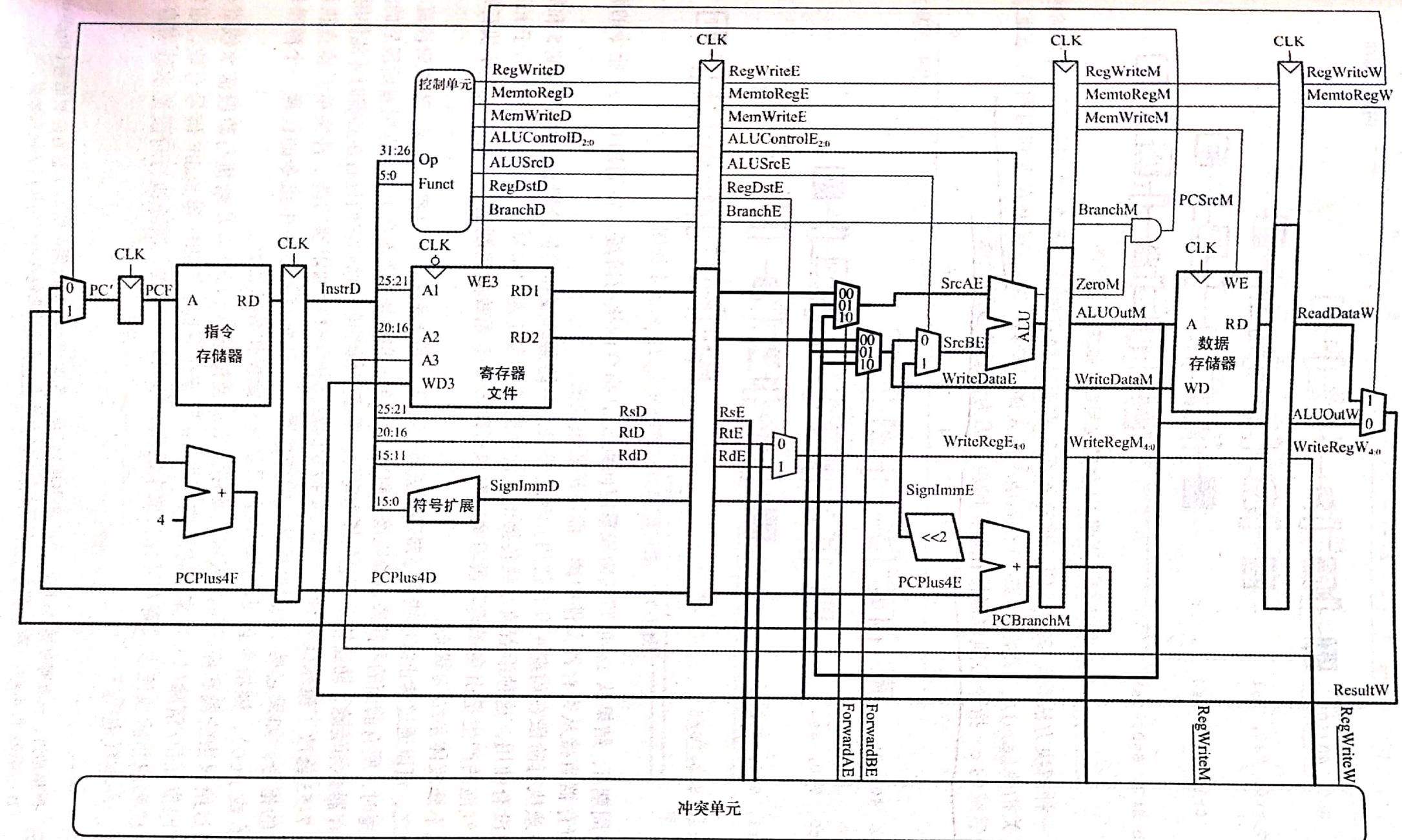


图7-50 利用重定向解决冲突的流水线处理器



扫描全能王 创建

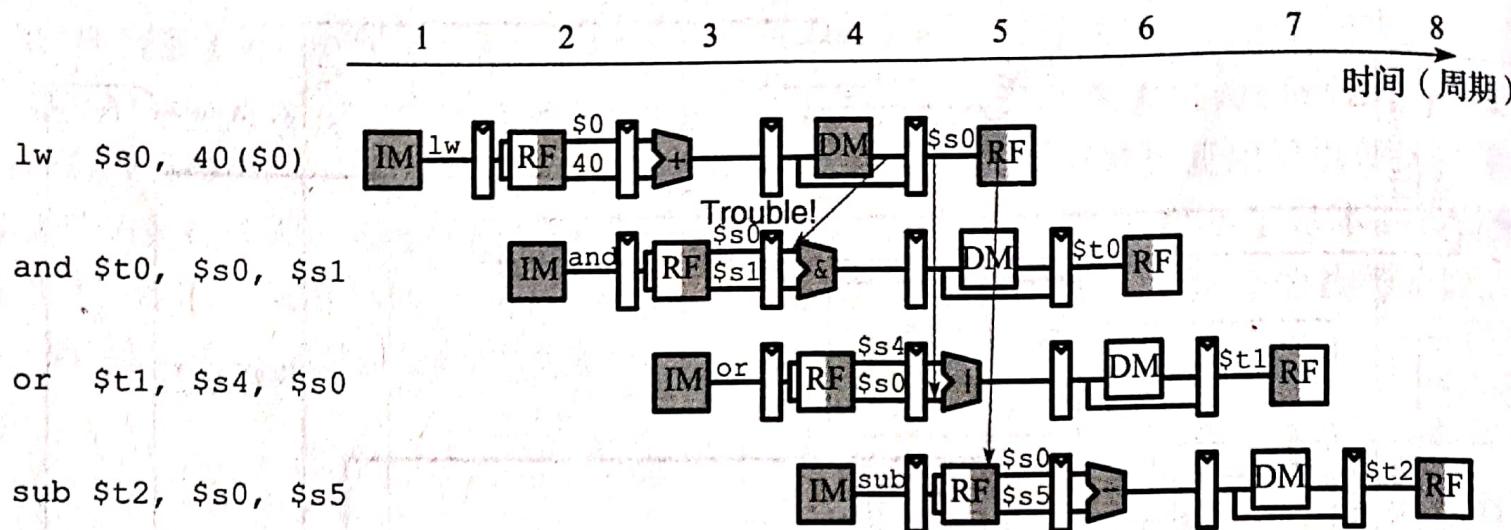


图 7-51 说明 `lw` 重定向问题的抽象流水线图

另一种解决方法是阻塞流水线，将操作挂起直至数据有效时。图 7-52 显示了在译码阶段阻塞相关指令(`and`)。该指令(`and`)在周期 3 进入译码阶段，并一直阻塞直到周期 4。在此过程中，后续的 `or` 指令必须也保持在取指阶段，因为译码阶段已经满了。

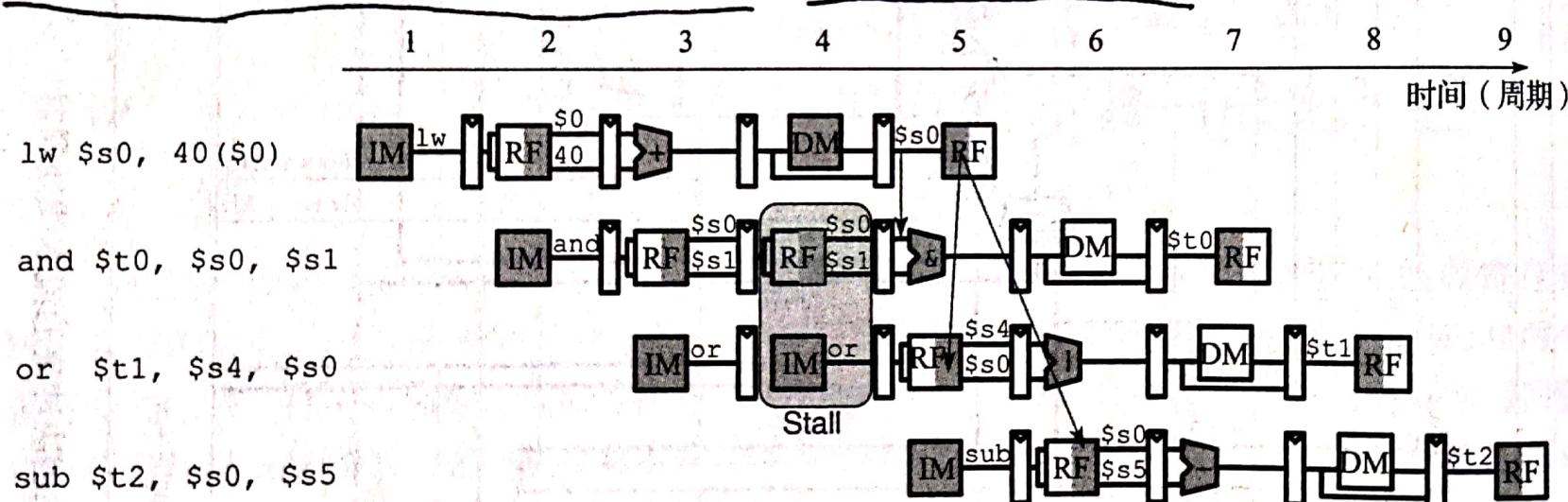


图 7-52 通过阻塞方式解决冲突的抽象流水线图



扫描全能王 创建

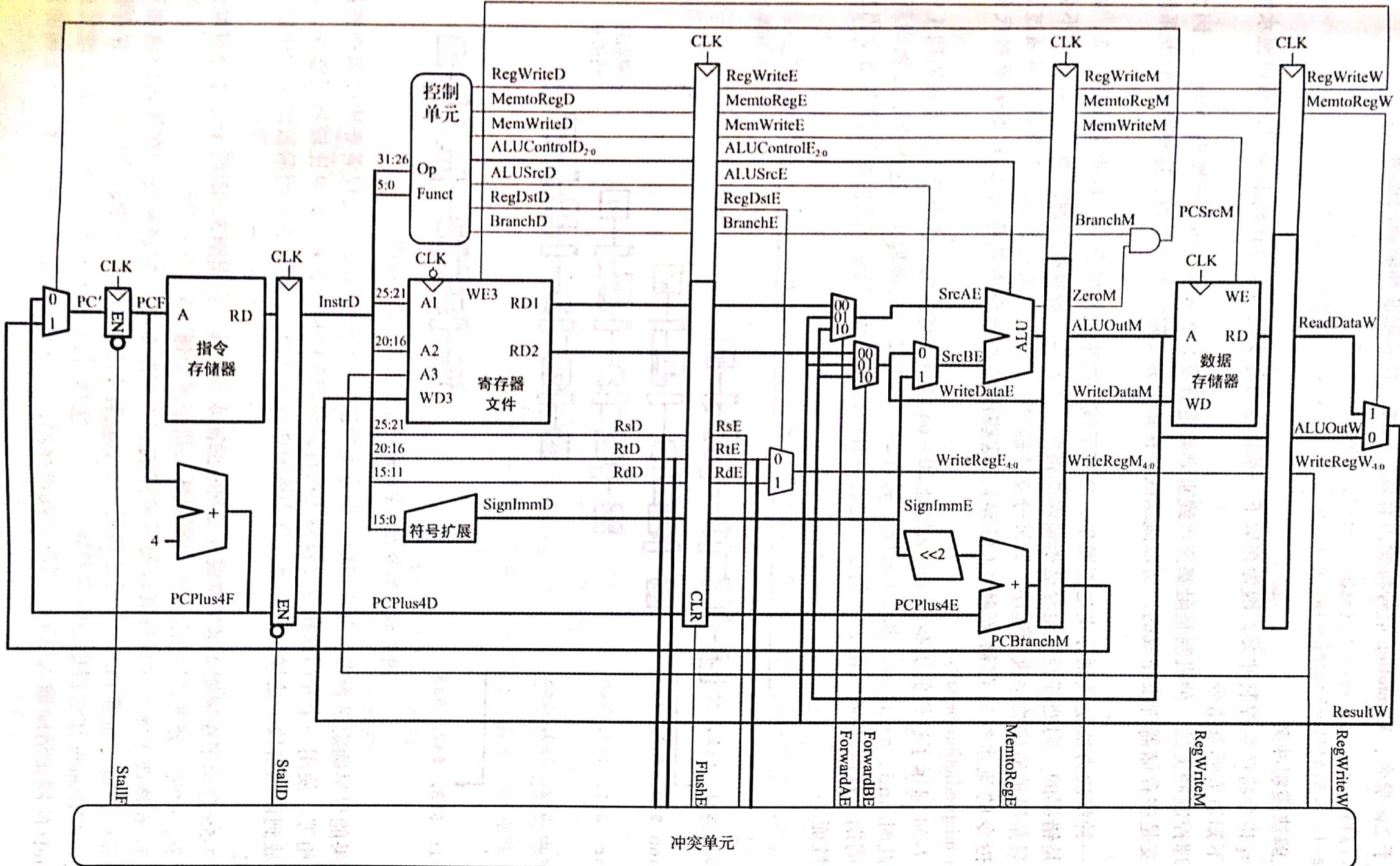


图7-53 通过阻塞方式解决1w指令数据冲突的流水线处理器



扫描全能王 创建

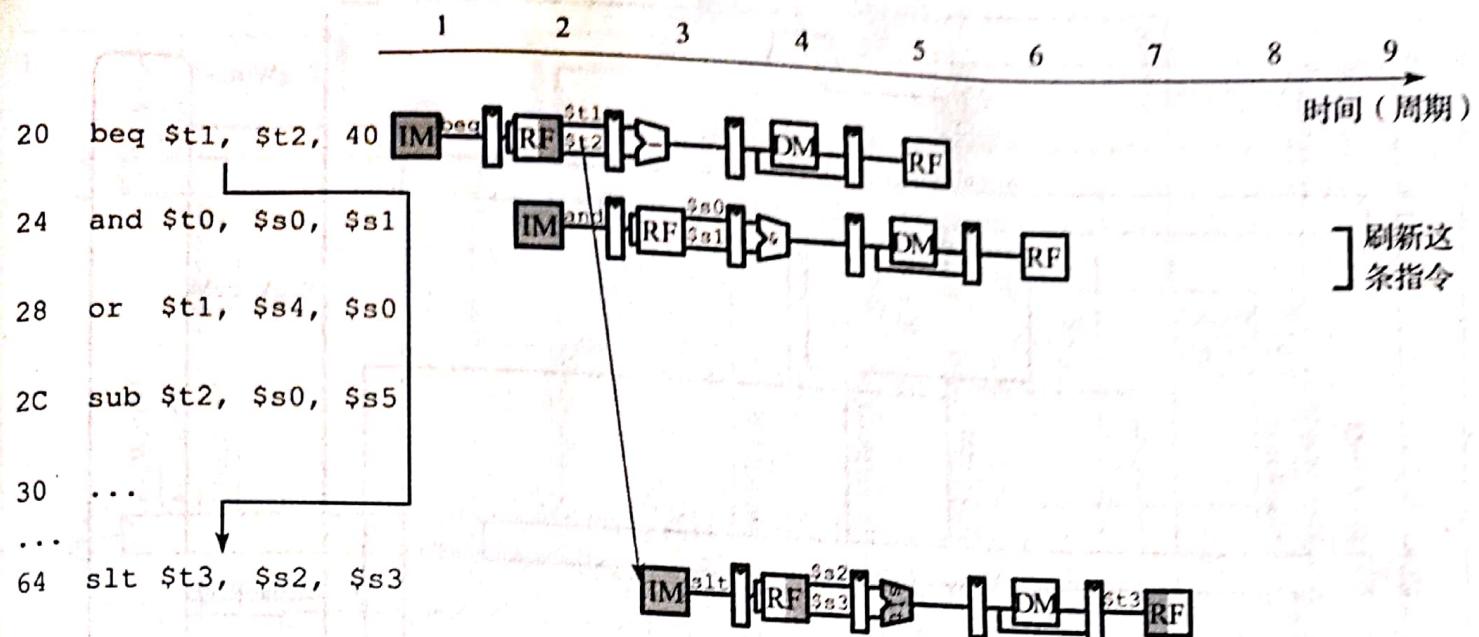


图 7-55 尽早确定分支的抽象流水线图

不幸的是，提前确定分支的硬件会产生新的 RAW 数据冲突。特别是，如果分支指令的一个源操作数由前一条指令计算得到且还没有写入寄存器文件，分支指令将从寄存器文件中读取错误的操作数值。我们可以采用前面所介绍的方法来解决数据冲突（如果数据有效则进行转发，或者阻塞流水线直至数据准备好）。

图 7-57 显示了对流水线处理器的修改以便解决在译码阶段的数据相关性。如果结果在写回阶段，它将在前半周期写入寄存器，而在后半周期进行读操作，所以此时不存在冲突。如果 ALU 指令的结果在存储器阶段，可以将它通两个新的复用器重定向到相等比较器。如果 ALU 指令的结果在执行阶段或者 lw 指令的结果在存储器阶段，则流水线必须在译码阶段阻塞直至结果准备好。

译码阶段重定向逻辑如下式给出。

```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

分支的阻塞检测逻辑如下式所示。处理器必须在译码阶段完成分支判断。如果分支指令的源寄存器依赖于处于执行阶段的 ALU 指令，或者依赖与存储器阶段的 lw 指令，处理器必须阻塞直至源操作数准备好。

```
branchstall =
  BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
  OR
  BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

现在处理器阻塞可以会由于装入或分支冲突：

```
StallF = StallD = FlushE = lwstall OR branchstall
```



扫描全能王 创建

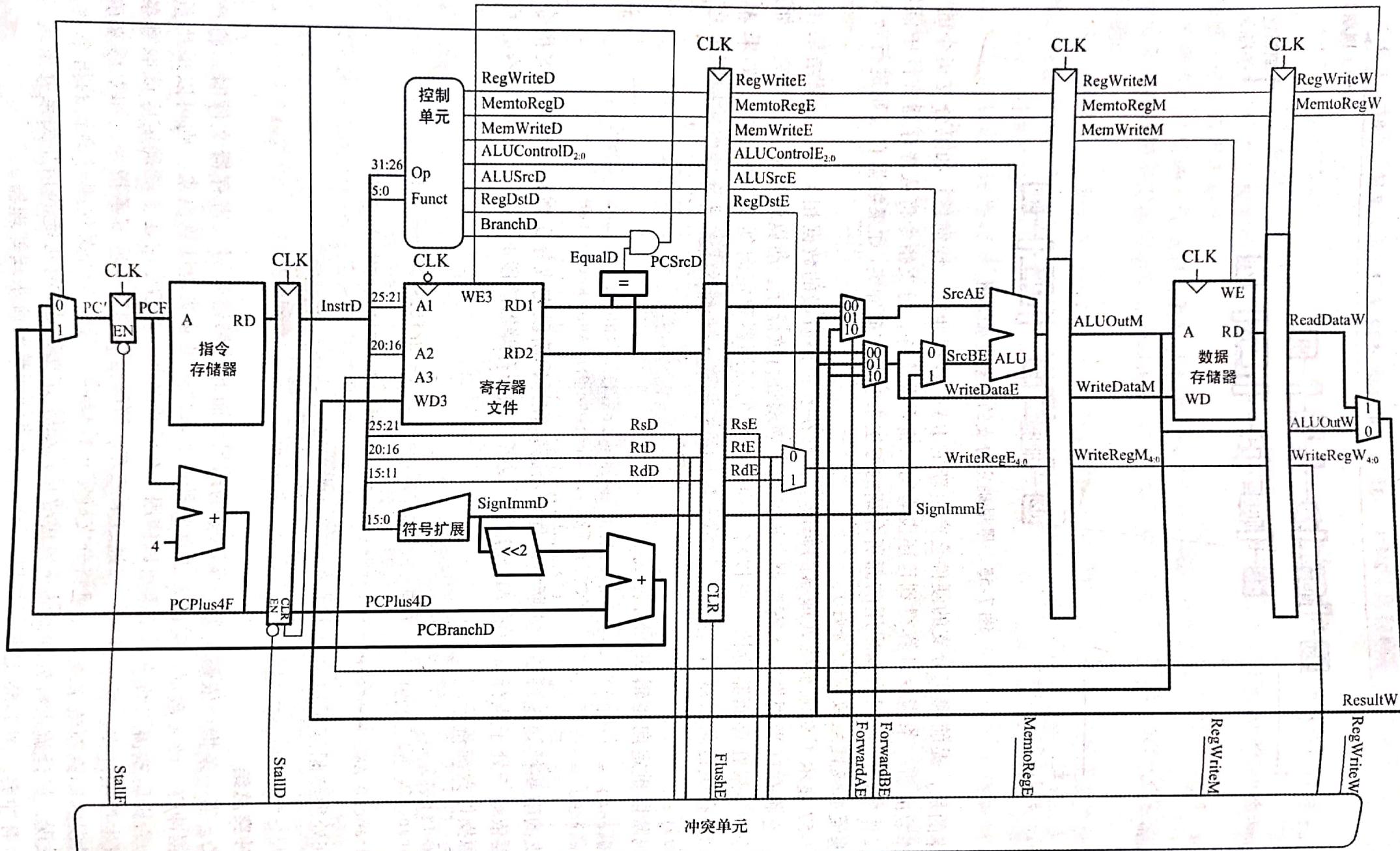


图7-56 处理分支控制冲突的流水线处理器



扫描全能王 创建

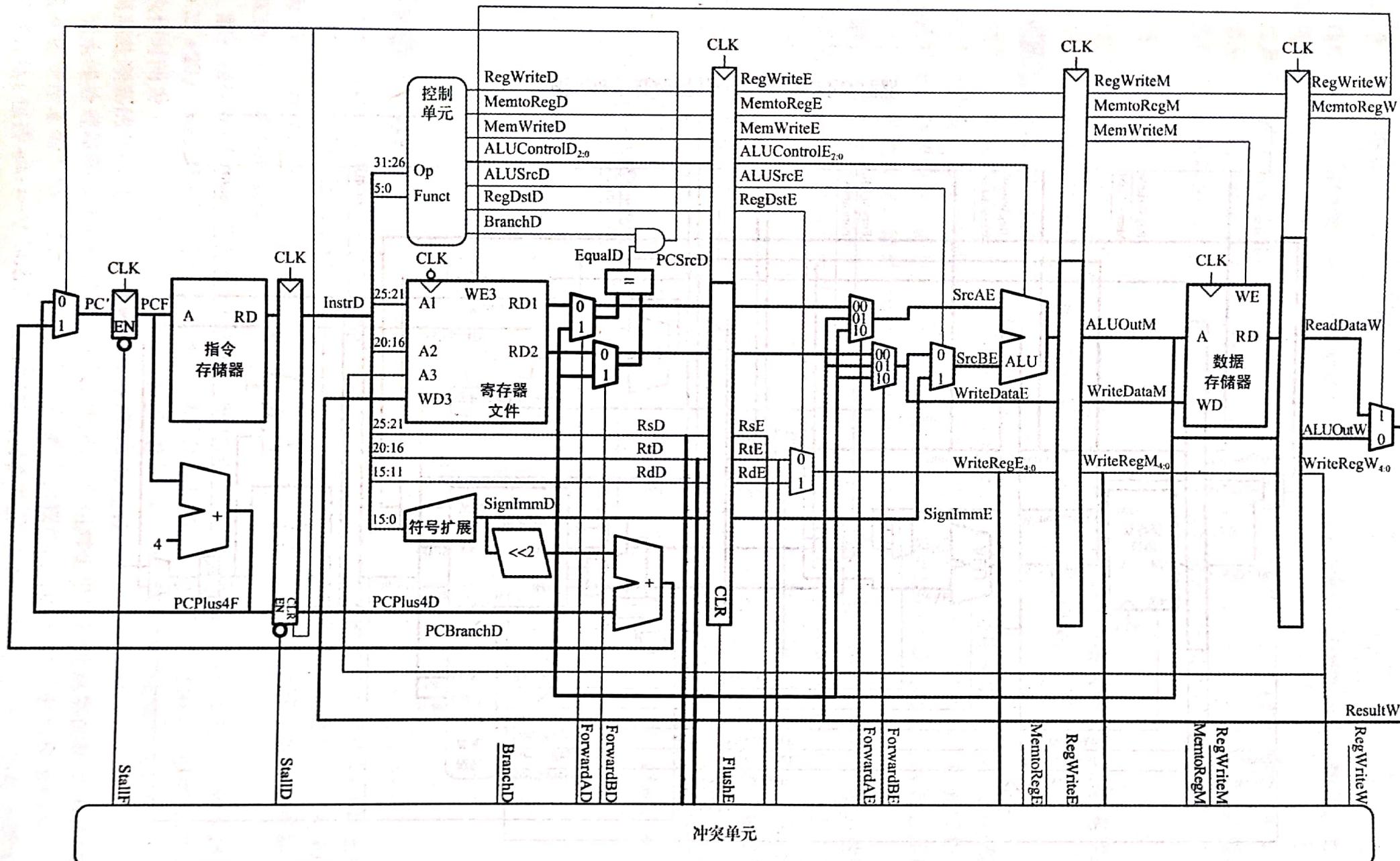


图7-57 处理分支指令数据相关性的流水线处理器



扫描全能王 创建

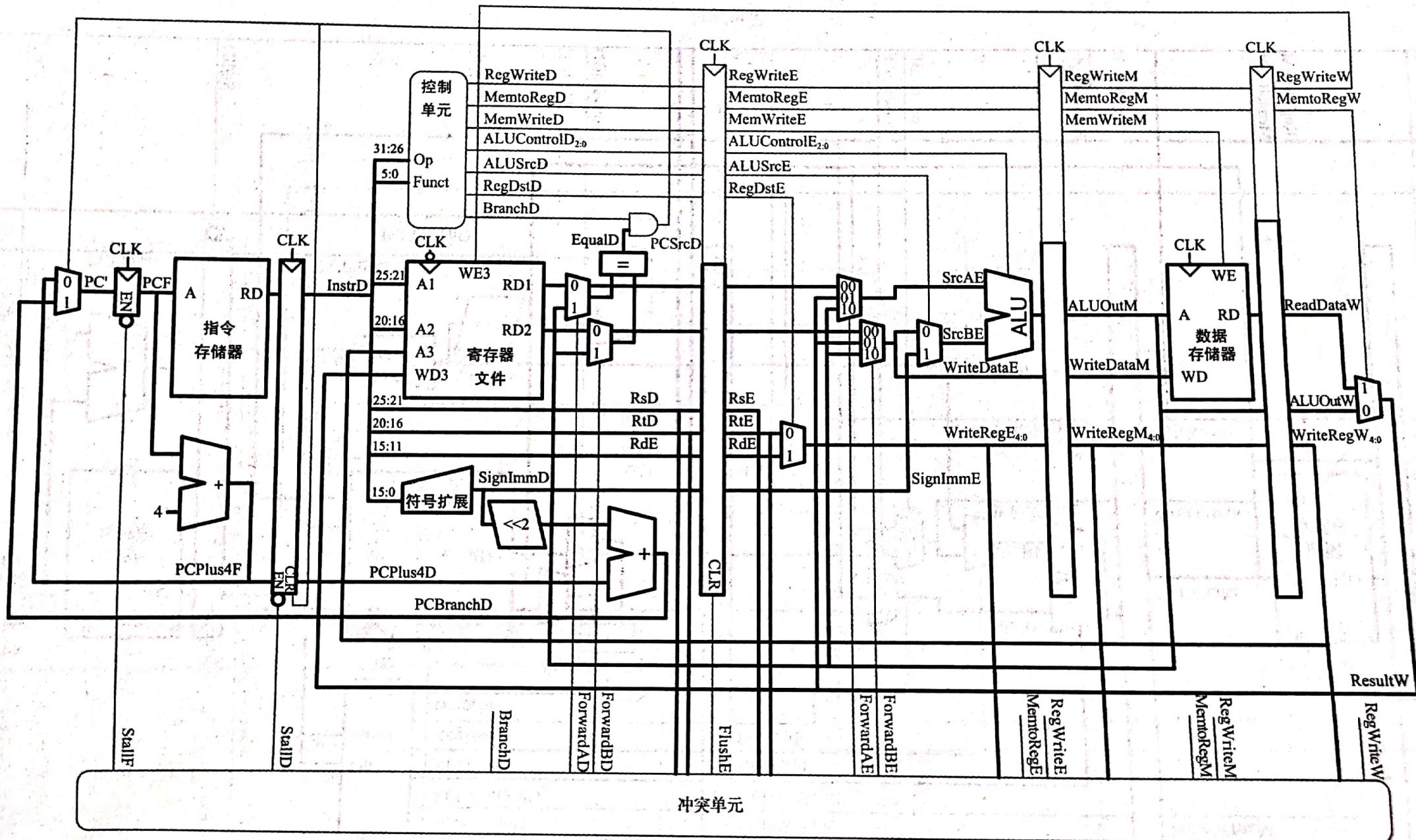


图7-58 处理所有冲突的流水线处理器



扫描全能王 创建

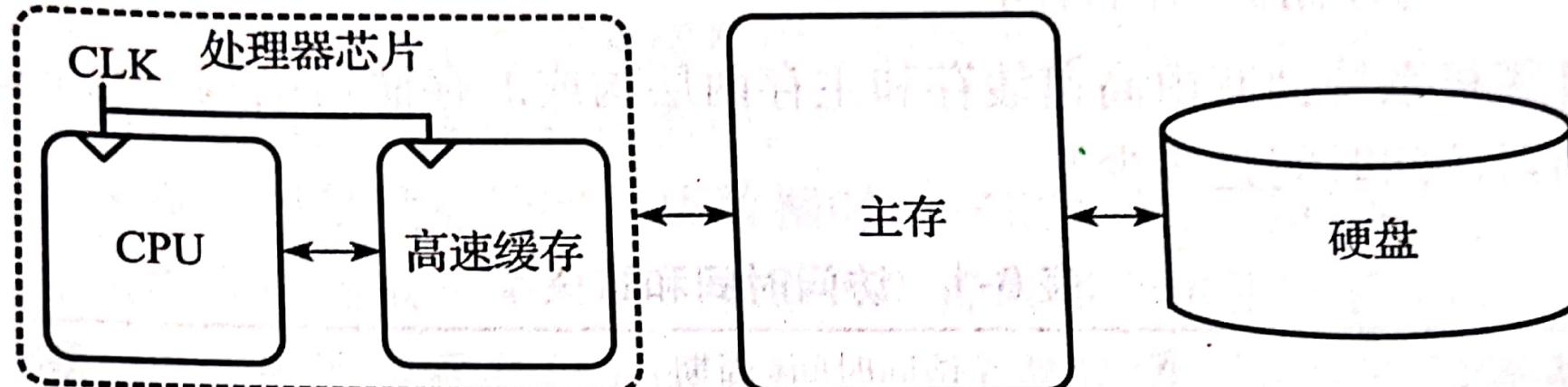


图 8-3 典型的存储器层次结构

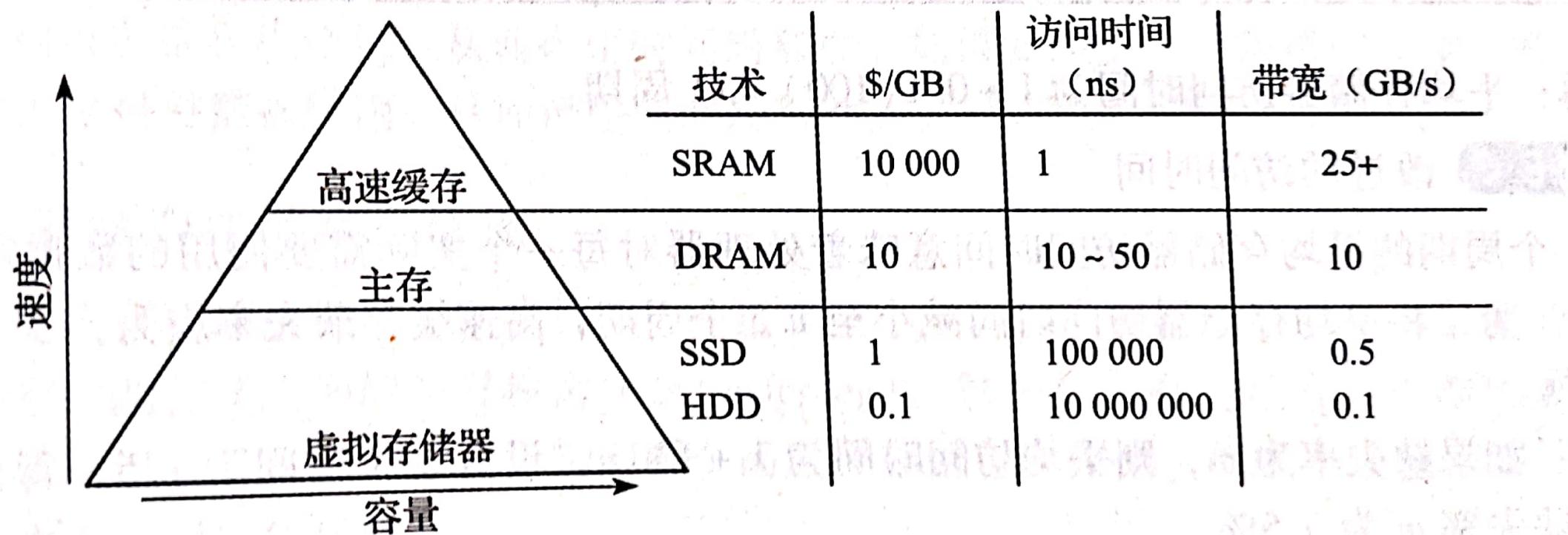
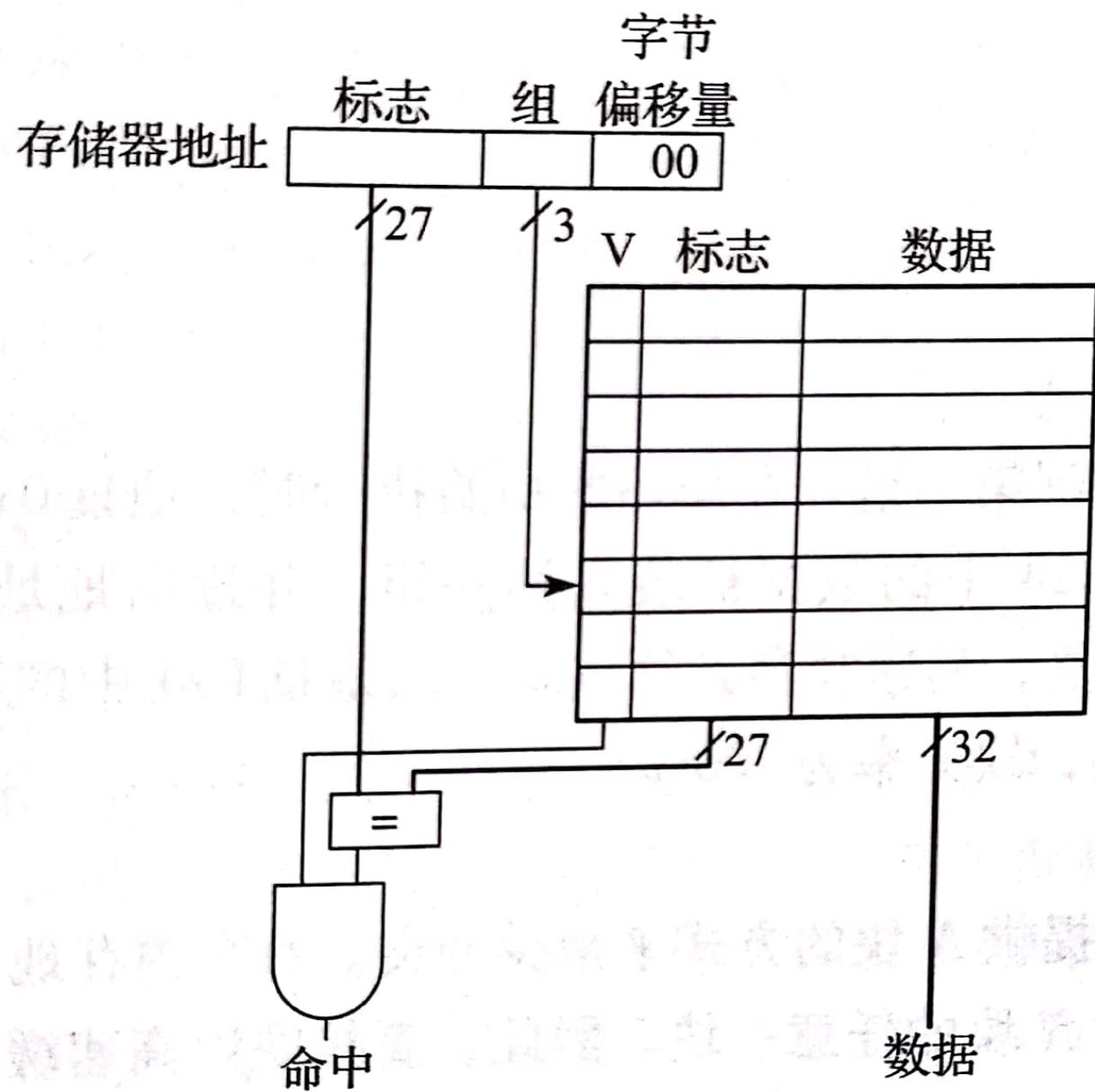


图 8-4 2012 年，存储器层次结构中各组成部分的典型特征



扫描全能王 创建



8表项SRAM,
其中每个表项包括
 $(1 + 27 + 32)$ 位

图 8-7 8 组的直接映射高速缓存



扫描全能王 创建

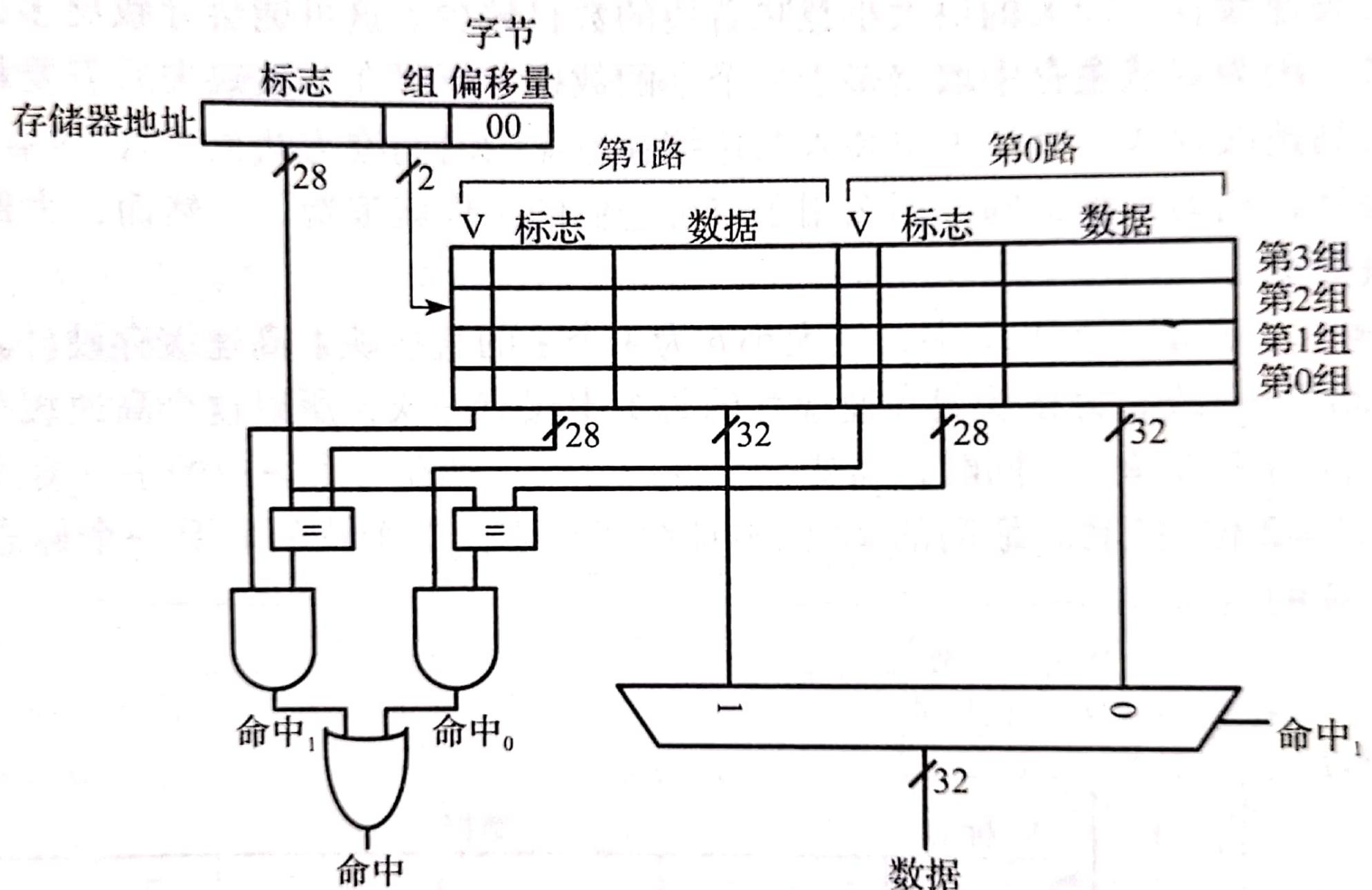


图 8-9 2 路组相联高速缓存



扫描全能王 创建

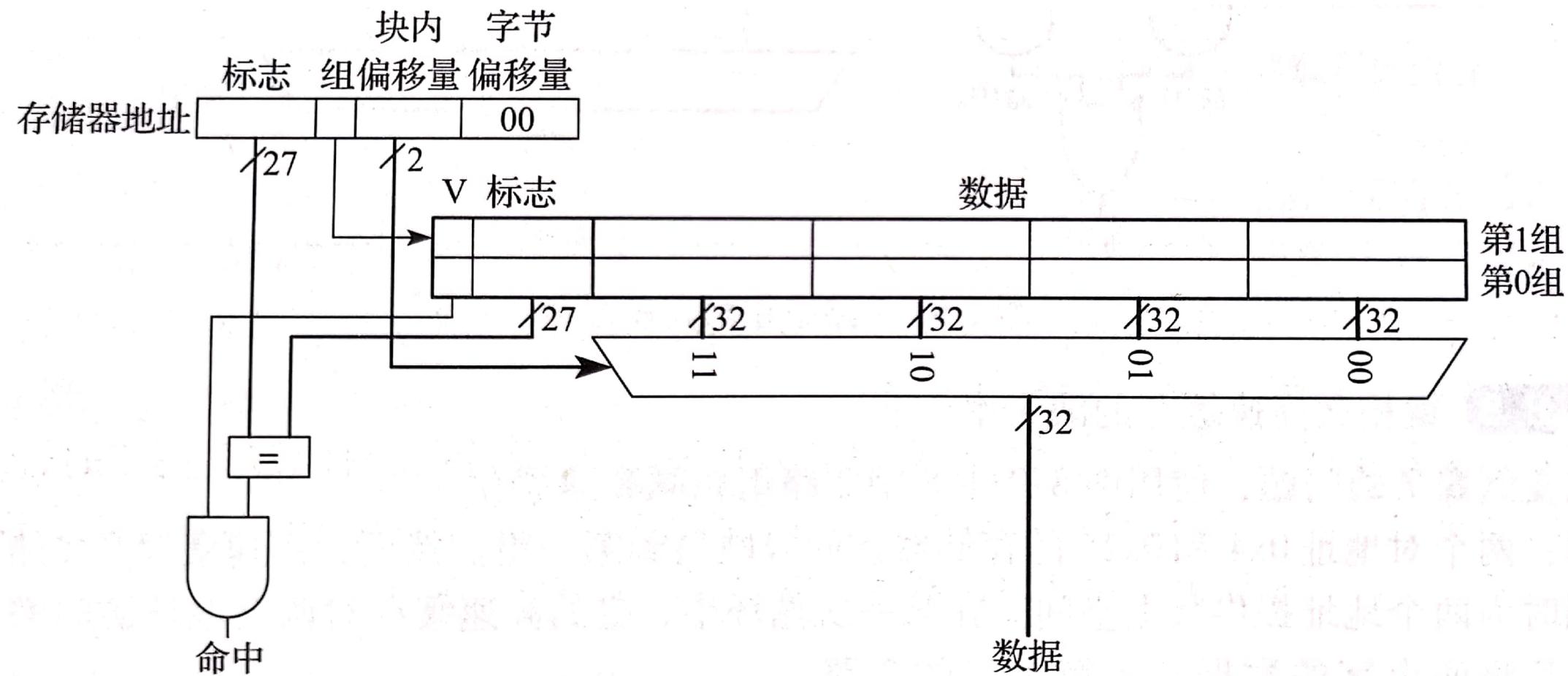


图 8-12 组数为 2，块大小为 4 字的直接映射高速缓存



扫描全能王 创建

VHDL 的 IEEE 标准；最近更新在 2008 年。相关内容请见：ieeexplore.ieee.org。
Wakerly J., *Digital Design: Principles and Practices*, 4th ed., Prentice Hall, 2006.

一本关于数字设计方面的全面、易读的教材，和很好的参考书。
Weste N., and Harris D., *CMOS VLSI Design*, 4th ed., Addison-Wesley, 2011.

超大规模集成电路(VLSI)是构造包含很多晶体管的芯片的一门艺术和科学。本书的内容覆盖从初开始的基本知识到用于商业产品的最先进的技术。

补充阅读

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

微芯片的发明者之一，仙童半导体公司和英特尔共同创立者之一 Robert Noyce 的精彩传记。对于任何想要工作在硅谷的人来说，这本书可以让他们了解硅谷这个地方的文化，一种相比其他硅谷风云人物，被 Noyce 加以更深影响的文化。

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.

Intel 几代奔腾芯片开发的传奇故事，由这个项目的负责人之一所著。对于那些考虑从事这个领域的人来说，这本书提供了了解这个巨大设计项目管理的多个视角，透露了这个最重要的商业微处理器产品线的幕后新闻。

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

关于计算机运算系统的最全面的教材。构架高质量计算机运算单元的优秀资源。

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011.

高级计算机体系结构的权威教材。如果你很有兴趣了解最前沿的微处理器的内部工作原理，这本书正是适合你的书。

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

一个计算机系统设计的经典故事。30 年后，这本书依然让你欲罢不能，书中关于项目管理和技术的观点和看法在今天仍然适用。

Pedroni V., *Circuit Design and Simulation with VHDL*, 2nd ed., MIT Press, 2010.

一本展示如何用 VHDL 设计电路的参考书。

Ciletti M., *Advanced Digital Design with the Verilog HDL*, 2nd ed., Prentice Hall, 2010.

一本关于 Verilog 2005(而不是 System Verilog)的较好的参考书。

SystemVerilog IEEE Standard (IEEE STD 1800).

System Verilog Hardware Description Language 的 IEEE 标准，最近更新在 2009 年。相关内容请见：ieeexplore.ieee.org。

VHDL IEEE Standard (IEEE STD 1076).



扫描全能王 创建

HDL 例 4.37 测试

SystemVerilog

```
module testbench1();
    logic a, b, c, y;

    // instantiate device under test
    sillyfunction dut(a, b, c, y);

    // apply inputs one at a time
    initial begin
        a=0; b=0; c=0; #10;
        c=1;           #10;
        b=1; c=0;      #10;
        c=1;           #10;
        a=1; b=0; c=0; #10;
        c=1;           #10;
        b=1; c=0;      #10;
        c=1;           #10;
    end
endmodule
```

在模拟开始时 Initial 语句执行该段内的语句。在本例中，它首先提供输入模式 000，然后等待 10 个时间单位。然后提供 001，等待 10 个时间单位，以此类推，直到提供了所有 8 个可能的输入。Initial 语句只能在测试程序上用于模拟，不能用于综合为实际硬件的模块中。第一次启动时，硬件无法执行一系列特殊的步骤。

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
    component sillyfunction
        port(a, b, c: in STD_LOGIC;
              y:     out STD_LOGIC);
    end component;
    signal a, b, c, y: STD_LOGIC;
begin
    -- instantiate device under test
    dut: sillyfunction port map(a, b, c, y);

    -- apply inputs one at a time
    process begin
        a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
        c <= '1';           wait for 10 ns;
        b <= '1'; c <= '0'; wait for 10 ns;
        c <= '1';           wait for 10 ns;
        a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
        c <= '1';           wait for 10 ns;
        b <= '1'; c <= '0'; wait for 10 ns;
        c <= '1';           wait for 10 ns;
        wait; -- wait forever
    end process;
end;
```

process 语句最先提供输入模式 000，等待 10ns。然后它提供 001，等待 10ns，以此类推直到提供了所有 8 个可能的输入。最后，该过程将无限等待；否则，该过程再次开始，重复地提供测试向量的模式。



扫描全能王 创建

HDL 例 4.38 自检测试程序

SystemVerilog

```
module testbench2();
    logic a, b, c, y;
    // instantiate device under test
    sillyfunction dut(a, b, c, y);
    // apply inputs one at a time
    // checking results
    initial begin
        a=0; b=0; c=0; #10;
        assert (y === 1) else $error("000 failed.");
        c=1; #10;
        assert (y === 0) else $error("001 failed.");
        b=1; c=0; #10;
        assert (y === 0) else $error("010 failed.");
        c=1; #10;
        assert (y === 0) else $error("011 failed.");
        a=1; b=0; c=0; #10;
        assert (y === 1) else $error("100 failed.");
        c=1; #10;
        assert (y === 1) else $error("101 failed.");
        b=1; c=0; #10;
        assert (y === 0) else $error("110 failed.");
        c=1; #10;
        assert (y === 0) else $error("111 failed.");
    end
endmodule
```

SystemVerilog 的 assert 语句检查特定条件是否成立。如果不成立，则执行 else 语句。else 语句中的 \$error 系统任务用于输出描述 assert 错误的错误信息。在综合过程中 assert 将被忽略。

在 SystemVerilog 中，可以在不包括 x 和 z 值的

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench2 is -- no inputs or outputs
end;
architecture sim of testbench2 is
component sillyfunction
    port(a, b, c: in STD_LOGIC;
         y:      out STD_LOGIC);
end component;
signal a, b, c, y: STD_LOGIC;
begin
    -- instantiate device under test
    dut: sillyfunction port map(a, b, c, y);
    -- apply inputs one at a time
    -- checking results
    process begin
        a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
        assert y = '1' report "000 failed.";
        c <= '1'; wait for 10 ns;
        assert y = '0' report "001 failed.";
        b <= '1'; c <= '0'; wait for 10 ns;
        assert y = '0' report "010 failed.";
        c <= '1'; wait for 10 ns;
        assert y = '0' report "011 failed.";
        a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
        assert y = '1' report "100 failed.";
        c <= '1'; wait for 10 ns;
        assert y = '1' report "101 failed.";
        b <= '1'; c <= '0'; wait for 10 ns;
        assert y = '0' report "110 failed.";
        c <= '1'; wait for 10 ns;
        assert y = '0' report "111 failed.";
        wait; -- wait forever
    end process;
end;
```



扫描全能王 创建

HDL 例 4.39 说明了这种测试程序。测试程序使用没有敏感信号列表的 always/process 语句产生一个时钟，这样它会连续不断地重复运行。在模拟的开始，它从一个文本文件读取测试向量，提供两个周期的 reset 脉冲。虽然时钟信号和复位信号在组合逻辑测试中不是必需的，但它们也包含在代码中，因为它们在测试时序 DUT 中是很重要的。example.tv 是包含二进制格式输入和期待输出的文本文件：

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL 例 4.39 带测试文件的测试程序

SystemVerilog

```
module testbench3();
    logic clk, reset;
    logic a, b, c, y, yexpected;
    logic [31:0] vectornum, errors;
    logic [3:0] testvectors[10000:0];

    // instantiate device under test
    sillyfunction dut(a, b, c, y);

    // generate clock
    always @ (posedge clk)
        begin
            clk = 1; #5; clk = 0; #5;
        end

    // at start of test, load vectors
    // and pulse reset
    initial
        begin
            $readmemh("example.tv", testvectors);
            vectornum = 0; errors = 0;
            reset = 1; #27; reset = 0;
        end

    // apply test vectors on rising edge of clk
    always @(posedge clk)
        begin
            #1; {a, b, c, yexpected} = testvectors[vectornum];
        end

    // check results on falling edge of clk
    always @(negedge clk)
        if (~reset) begin // skip during reset
            if (y != yexpected) begin // check result
                $display("Error: inputs=%b", {a, b, c});
                $display(" outputs=%b (%b expected)", y, yexpected);
                errors = errors + 1;
            end
            vectornum = vectornum + 1;
            if (testvectors[vectornum] === 4'bxx) begin
                $display("%d tests completed with %d errors",
                        vectornum, errors);
                $finish;
            end
        end
endmodule
```

\$readmem 将二进制数字文件读入 testvectors 数组中。\$readmemh 与之相似，但它读取

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
    component sillyfunction
        port(a, b, c: in STD_LOGIC;
             y: out STD_LOGIC);
    end component;
    signal a, b, c, y: STD_LOGIC;
    signal y_expected: STD_LOGIC;
    signal clk, reset: STD_LOGIC;
begin
    -- instantiate device under test
    dut: sillyfunction port map(a, b, c, y);

    -- generate clock
    process begin
        clk <= '1'; wait for 5 ns;
        clk <= '0'; wait for 5 ns;
    end process;

    -- at start of test, pulse reset
    process begin
        reset <= '1'; wait for 27 ns; reset <= '0';
        wait;
    end process;

    -- run tests
    process is
        file tv: text;
        variable L: line;
        variable vector_in: std_logic_vector(2 downto 0);
        variable dummy: character;
        variable vector_out: std_logic;
        variable vectornum: integer := 0;
        variable errors: integer := 0;
    begin
        FILE_OPEN(tv, "example.tv", READ_MODE);
        while not endfile(tv) loop
            -- change vectors on rising edge
            wait until rising_edge(clk);
            -- read the next line of testvectors and split into pieces
            readline(tv, L);
            read(L, vector_in);
            read(L, dummy); -- skip over underscore
            read(L, vector_out);
            if (vector_out /= y) then
                errors := errors + 1;
            end if;
            vectornum := vectornum + 1;
        end loop;
    end process;
end;
```



扫描全能王 创建

十六进制数字的文件。

代码的下一块在时钟的上沿后等待一个时间单位(以防止时钟和数据同时改变造成的混乱),然后根据当前测试向量中的4位设置3位输入(a、b和c)和期望的输出(yexpected)。

测试程序将期望的输出yexpected与生成的输出y比较,如果它们不相等,则输出一条错误信息。%b和%d分别表示以二进制或者十进制输出值。例如,\$display("%b %b",y,yexpected);表示以二进制输出y和yexpected两个值。%h以十六进制输出数值。

这个进程重复直到tesevector数组中没有更多可用的测试向量。\$finish结束模拟。

注意,即使SystemVerilog模块最多支持10 001个测试向量,但是它在执行文件中8个的测试向量后就结束模拟。

```
read(L, vector_out);
(a, b, c) <= vector_in(2 downto 0) after 1 ns;
y_expected <= vector_out after 1 ns;

-- check results on falling edge
wait until falling_edge(clk);

if y /= y_expected then
    report "Error: y=" & std_logic'image(y);
    errors := errors+1;
end if;

vectornum := vectornum+1;
end loop;

-- summarize results at end of simulation
if (errors=0) then
    report "NO ERRORS -- " &
        integer'image(vectornum) &
        " tests completed successfully."
    severity failure;
else
    report integer'image(vectornum) &
        " tests completed, errors=" &
        integer'image(errors)
    severity failure;
end if;
end process;
end;
```

VHDL代码使用文件读取指令已经超出了本章的范围,但这里也给出了一个VHDL自检测试程序的概况。

在时钟的上升沿向被测设备提供新的输入,在时钟的下降沿检查输出。测试程序可以在发生错误时报告错误。可以在模拟结束时,测试程序输出应用的全部测试向量数和检测的错误数。

对这样简单的电路,使用HDL例4.39中的测试程序有点过分了。然而,经过简单的修改,它可以测试更复杂的电路,修改的内容主要包括:修改example.tv、实例化新的DUF、修改一些代码行来设置输入和检查输出。



扫描全能王 创建

代码示例 6.19 while 循环

高级语言代码

```
int pow = 1;
int x = 0;
while (pow != 128)
{
    pow = pow * 2;
    x = x + 1;
}
```

MIPS 汇编代码

```
# $s0 = pow, $s1 = x
addi $s0, $0, 1      # pow = 1
addi $s1, $0, 0      # x = 0
addi $t0, $0, 128    # t0 = 128 for comparison
while:
beq $s0, $t0, done  # if pow == 128, exit while loop
sll $s0, $s0, 1      # pow = pow * 2
addi $s1, $s1, 1      # x = x + 1
j    while
done:
```



扫描全能王 创建

代码示例 6.20 for 循环

高级语言编码

```
int sum = 0;  
  
for (i = 0; i != 10; i = i + 1) {  
    sum = sum + i ;  
}  
  
// equivalent to the following while loop  
int sum = 0;  
int i = 0;  
while (i != 10) {  
    sum = sum + i;  
    i = i + 1;  
}
```

MIPS 汇编编码

```
# $s0 = i, $s1 = sum  
add $s1, $0, $0      # sum = 0  
addi $s0, $0, 0       # i = 0  
addi $t0, $0, 10      # $t0 = 10  
  
for:  
beq $s0, $t0, done   # if i == 10, branch to done  
add $s1, $s1, $s0      # sum = sum + i  
addi $s0, $s0, 1       # increment i  
j for  
  
done:
```



扫描全能王 创建

3. 量值比较

目前为止，例子使用 `beq` 和 `bne` 指令执行相等或不相等的比较和分支。MIPS 为量值比较提供了小于设置指令(`slt`)。当 $rs < rt$ 时，`slt` 将 rd 设置为 1，否则， rd 为 0。

例 6.6 使用 `slt` 指令的循环

下述高级语言代码将对从 1 ~ 100 中的 2 的整数次幂求和。将其翻译为汇编语言程序。

```
// high-level code
```

```
int sum = 0;  
for (i = 1; i < 101; i = i * 2)  
    sum = sum + i;
```

解：汇编语言代码使用小于设置指令(`slt`)执行 `for` 循环中的小于比较操作。

```
# MIPS assembly code
```

```
# $s0 = i, $s1 = sum  
addi  $s1, $0, 0      # sum = 0  
addi  $s0, $0, 1      # i = 1  
addi  $t0, $0, 101    # $t0 = 101  
  
loop:  
    slt  $t1, $s0, $t0    # if (i < 101) $t1 = 1, else $t1 = 0  
    beq  $t1, $0, done    # if $t1 == 0 (i >= 101), branch to done  
    add   $s1, $s1, $s0    # sum = sum + i  
    sll   $s0, $s0, 1      # i = i * 2  
    j     loop  
  
done:
```



扫描全能王 创建

代码示例 6.22 使用 for 循环来访问数组

高级语言编码

```
int i;
int array[1000];

for (i = 0; i < 1000; i = i + 1)

array[i] = array[i] * 8;
```

MIPS 汇编编码

```
# $s0 = array base address, $s1 = i
# initialization code
    lui  $s0, 0x23B8          # $s0 = 0x23B80000
    ori  $s0, $s0, 0xF000      # $s0 = 0x23B8F000
    addi $s1, $0, 0            # i = 0
    addi $t2, $0, 1000         # $t2 = 1000

loop:
    slt  $t0, $s1, $t2        # i < 1000?
    beq  $t0, $0, done         # if not, then done
    sll  $t0, $s1, 2           # $t0 = i*4 (byte offset)
    add  $t0, $t0, $s0          # address of array[i]
    lw   $t1, 0($t0)           # $t1 = array[i]
    sll  $t1, $t1, 3           # $t1 = array[i] * 8
    sw   $t1, 0($t0)           # array[i] = array[i] * 8
    addi $s1, $s1, 1            # i = i + 1
    j    loop                  # repeat

done:
```



扫描全能王 创建

代码示例 6.27 factorial 递归函数调用

高级语言代码

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n * factorial(n - 1));  
}
```

汇编语言代码

0x90	factorial:	addi \$sp, \$sp, -8	# make room on stack
0x94		sw \$a0, 4(\$sp)	# store \$a0
0x98		sw \$ra, 0(\$sp)	# store \$ra
0x9C		addi \$t0, \$0, 2	# \$t0 = 2
0xA0		slt \$t0, \$a0, \$t0	# n <= 1 ?
0xA4		beq \$t0, \$0, else	# no: goto else
0xA8		addi \$v0, \$0, 1	# yes: return 1
0xAC		addi \$sp, \$sp, 8	# restore \$sp
0xB0		jr \$ra	# return
0xB4	else:	addi \$a0, \$a0, -1	# n = n - 1
0xB8		jal factorial	# recursive call
0xBC		Iw \$ra, 0(\$sp)	# restore \$ra
0xC0		Iw \$a0, 4(\$sp)	# restore \$a0
0xC4		addi \$sp, \$sp, 8	# restore \$sp
0xC8		mul \$v0, \$a0, \$v0	# n * factorial(n-1)
0xCC		jr \$ra	# return

factorial 函数可能修改 \$a0 和 \$ra，所以它将这两个寄存器保存在栈中。然后它检查 n 是否小于 2，如果 n 小于 2，就返回 1 并保存在 \$v0 中，恢复栈指针，返回到调用函数。这种情况下，不需要重新装入 \$ra 和 \$a0，因为它们没有被修改。如果 n 大于 1，函数将递归调用 factorial(n - 1)，然后它从栈中恢复 n(\$a0) 的值和返回地址(\$ra)，执行乘法，返回结果。乘法指令(mul \$v0, \$a0, \$v0) 将 \$a0 和 \$v0 相乘，将结果存入 \$v0 中。

30

图 6-26 显示了执行 factorial(3) 时栈的情况。假定 \$sp 最初指向 0xFC，如图 6-26a 所示。函数创建两个字的栈空间来保存 \$a0 和 \$ra。在第一次调用时，factorial 将 \$a0 (\$a0 中保存着 n = 3) 保存在 0xF8 中，将 \$ra 保存在 0xF4 中，如图 6-26b 所示。然后函数将 \$a0 中的内容改变为 n = 2 并递归调用 factorial(2)，使 \$ra 保存 0xBC。在第二次调用时，factorial 将 \$a0 (\$a0 中保存着 n = 2) 保存在 0xF0 中，将 \$ra 保存在 0xEC 中。这时，我



扫描全能王 创建

们知道 \$ra 中存储了 0xBC。然后函数将 \$a0 中的内容改变为 $n = 1$ 并递归调用 factorial(1)。在第三次调用时, fuctorial 将 \$a0(\$a0 中保存这 $n = 1$) 保存在 0xE8 中, 将 \$ra 保存在 0xE4 中。这时, \$ra 存储的还是 0xBC。fuctorial 的第三次调用返回保存在 \$v0 中的 1, 并且在返回到第二次调用前回收栈空间。第二次调用将 n 恢复为 2, 将 \$ra 恢复为 0xBC (\$ra 中已经是这个值了), 然后回收栈帧, 返回 $\$v0 = 2 \times 1 = 2$ 给第一次调用。第一次调用将 n 恢复为 3, 将 \$ra 恢复为调用函数的返回地址, 回收栈帧, 返回 $\$v0 = 3 \times 2 = 6$ 。图 6-26c 显示了递归调用函数返回时栈的情况。当 factorial 返回到调用函数时, 栈指针指向它的初始位置(0xFC), 指针之上的栈空间的内容没有变化, 而且所有受保护寄存器保存它们的初始值, \$v0 保存返回值 6。

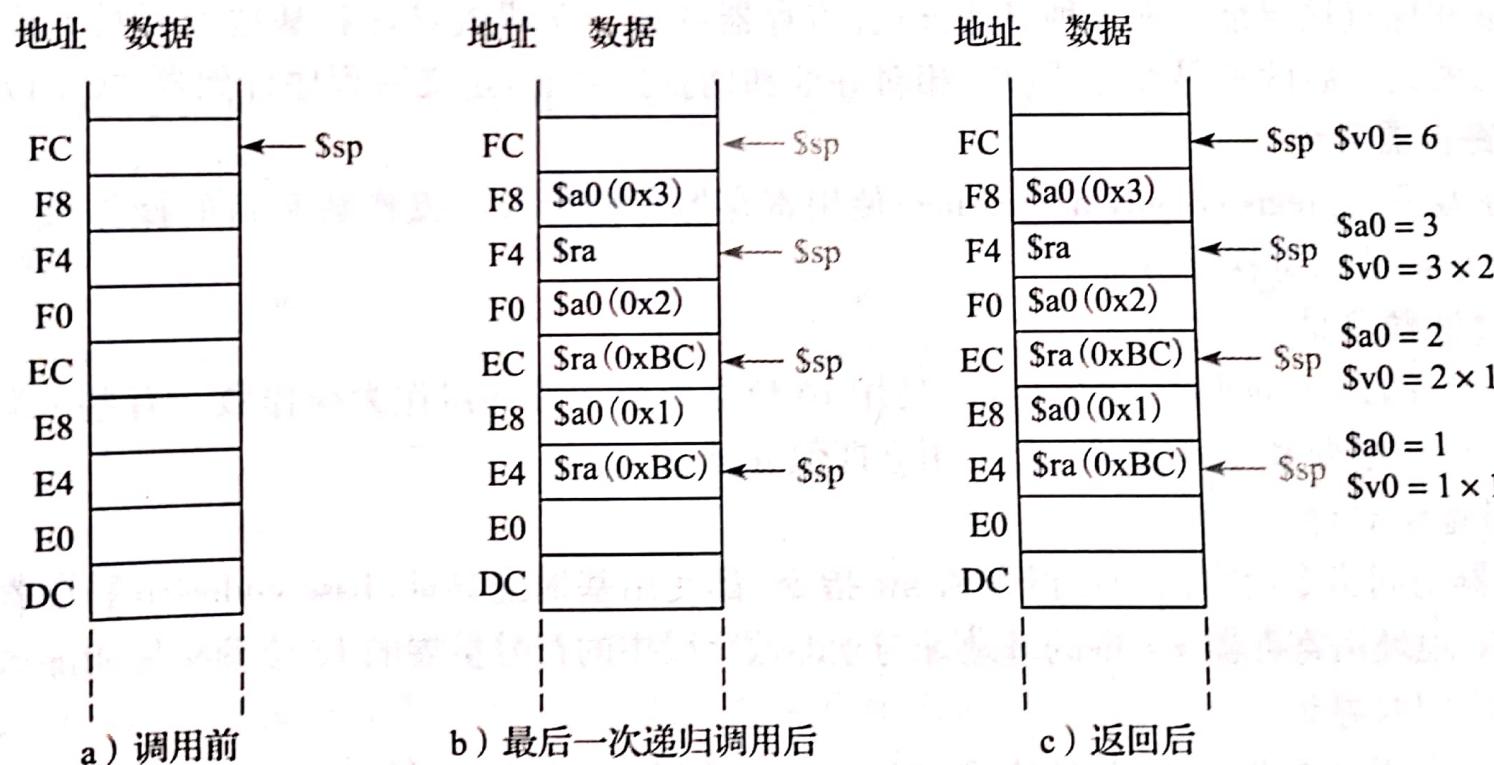


图 6-26 在 factorial 函数调用($n = 3$)期间栈的变化



扫描全能王 创建

面试问题

下述问题在数字设计工作的面试中曾经被问到(但对于其他汇编语言也适用)。

6. 1 写一段 MIPS 汇编代码，交换两个寄存器 \$t0 和 \$t1 中的内容，但不允许使用其他寄存器。
6. 2 假设给定一个存有正数和负数的整型数组。写一段 MIPS 汇编代码寻找具有最大和的数组子集假定数组的基地址存储在 \$a0 中，数组长度存储在 \$a1 中，产生的数组的子集的起始地址存储在 \$a2 中。编写代码，使之运行得越快越好。
6. 3 数组中保存着一个 C 语言字符串。设计算法来反转字符串并将新字符串存储在原来的数组中。使用 MIPS 汇编代码完成该算法。
6. 4 设计算法来计算一个 32 位数字中‘1’的个数，使用 MIPS 汇编代码完成。
6. 5 编写 MIPS 汇编代码，反转寄存器中的位。使用的指令越少越好，假设寄存器是 \$t3。
6. 6 编写 MIPS 汇编代码，测试 \$t2 和 \$t3 相加时是否有溢出，使用的指令越少越好。
6. 7 设计算法，测试给定的字符串是否为回文(回文就是从前面读和从后面读是一样的，例如，wow 和 racecar 就是回文)。使用 MIPS 汇编代码完成算法。



扫描全能王 创建

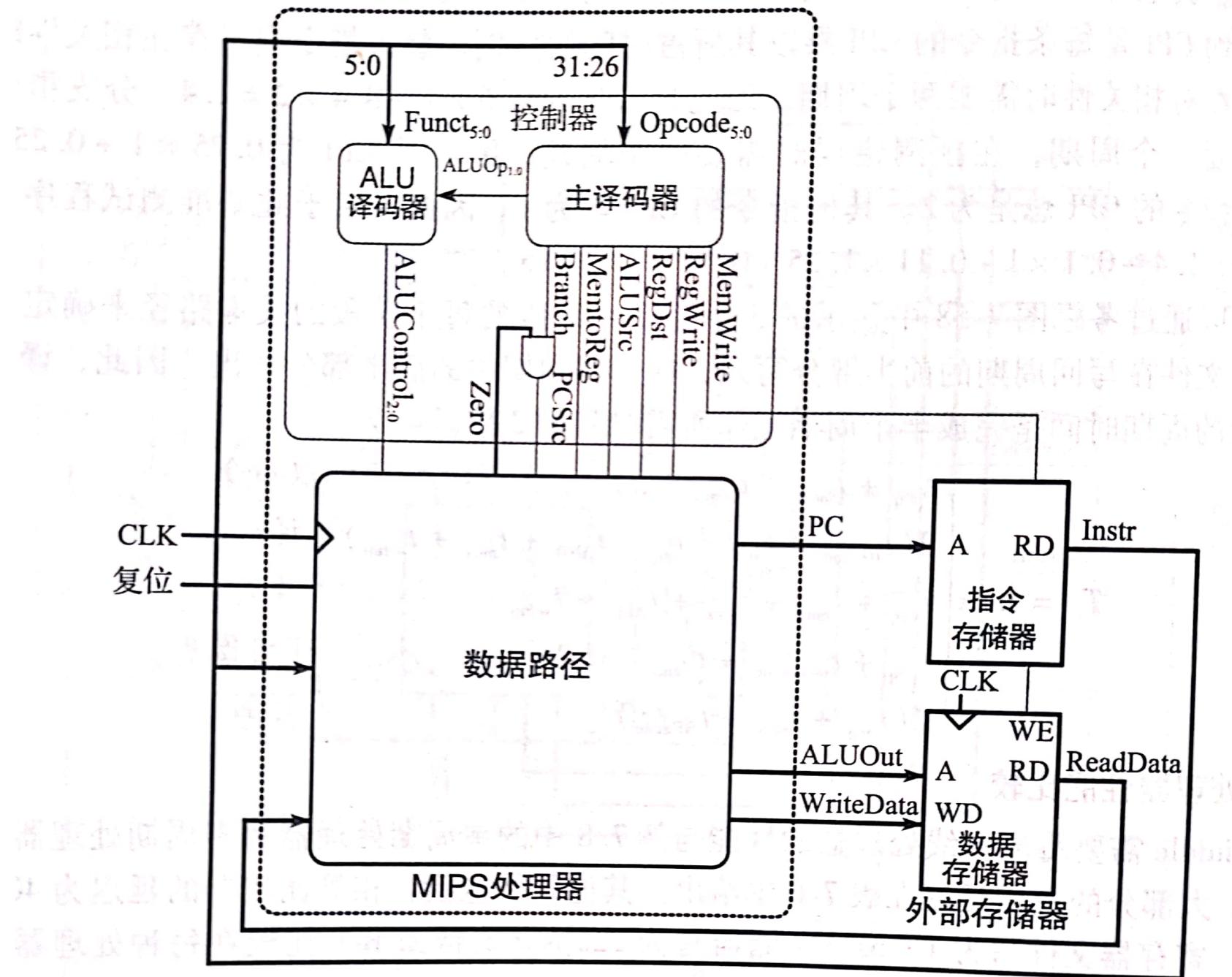


图 7-59 具有外部存储器接口的 MIPS 单周期处理器



扫描全能王 创建

7.6.3 基准测试程序

MIPS 基准测试程序将一段程序装入存储器中。图 7-60 中的程序通过计算检查所有指令，只有当所有指令都正确运行时才能得到正确的结果。具体地，如果该程序运行完全正确，则应向地址 84 写入值 7，如果硬件有问题就不可能这么做。这种测试访问称为随机测试 (ad hoc testing)。

#	Assembly	Description	Address	Machine
#			0	20020005
# Test the MIPS processor.			4	2003000c
# add, sub, and, or, slt, addi, lw, sw, beq, j			8	2067ffff7
# If successful, it should write the value 7 to address 84			c	00e22025
main:	addi \$2, \$0, 5	# initialize \$2 = 5	10	00642824
	addi \$3, \$0, 12	# initialize \$3 = 12	14	00a42820
	addi \$7, \$3, -9	# initialize \$7 = 3	18	10a7000a
	or \$4, \$7, \$2	# \$4 = (3 OR 5) = 7	1c	0064202a
	and \$5, \$3, \$4	# \$5 = (12 AND 7) = 4	20	10800001
	add \$5, \$5, \$4	# \$5 = 4 + 7 = 11	24	20050000
	beq \$5, \$7, end	# shouldn't be taken	28	00e2202a
	slt \$4, \$3, \$4	# \$4 = 12 < 7 = 0	2c	00853820
	beq \$4, \$0, around	# should be taken	30	00e23822
	addi \$5, \$0, 0	# shouldn't happen	34	ac670044
around:	slt \$4, \$7, \$2	# \$4 = 3 < 5 = 1	38	8c020050
	add \$7, \$4, \$5	# \$7 = 1 + 11 = 12	3c	08000011
	sub \$7, \$7, \$2	# \$7 = 12 - 5 = 7	40	20020001
	sw \$7, 68(\$3)	# [80] = 7	44	ac020054
	lw \$2, 80(\$0)	# \$2 = [80] = 7		
	j end	# should be taken		
	addi \$2, \$0, 1	# shouldn't happen		
end:	sw \$2, 84(\$0)	# write mem[84] = 7		

图 7-60 MIPS 测试程序的汇编代码和机器代码

机器代码存储在十六进制文件 memfile.dat 中(见图 7-61)，这个文件在模拟时由基准测试程序装入。这个文件包含了指令的机器代码，其中每条指令一行。

20020005
2003000c
2067ffff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054

图 7-61 memfile.dat 的内容

基准测试程序、顶层 MIPS 模块和外部存储器 HDL 代码由下例子给出。该例子中的存储器均包含了 64 个字。



扫描全能王 创建

HDL 例 7.12 MIPS 基准测试程序

SystemVerilog

```
module testbench();
    logic clk;
    logic reset;
    logic [31:0] writedata, dataaddr;
    logic memwrite;
    // instantiate device to be tested
    top dut (clk, reset, writedata, dataaddr, memwrite);
    // initialize test
    initial
    begin
        reset <= 1; # 22; reset <= 0;
    end
    // generate clock to sequence tests
    always
    begin
        clk <= 1; # 5; clk <= 0; # 5;
    end
    // check results
    always @ (negedge clk)
    begin
        if (memwrite) begin
            if (dataaddr == 84 & writedata == 7)
                $display("Simulation succeeded");
            else
                $display("Simulation failed");
        end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
component top
    port(clk, reset:      in STD_LOGIC;
          writedata, dataaddr: out STD_LOGIC_VECTOR(31 downto 0);
          memwrite:           out STD_LOGIC);
end component;
signal writedata, dataaddr: STD_LOGIC_VECTOR(31 downto 0);
signal clk, reset, memwrite: STD_LOGIC;
begin
    -- instantiate device to be tested
    dut: top port map(clk, reset, writedata, dataaddr, memwrite);
    -- Generate clock with 10 ns period
    process begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;
    -- Generate reset for first two clock cycles
    process begin
        reset <= '1';
        wait for 22 ns;
        reset <= '0';
        wait;
    end process;
    -- check that 7 gets written to address 84 at end of program
    process(clk) begin
        if (clk'event and clk = '0' and memwrite = '1') then
            if (to_integer(dataaddr) = 84 and to_integer(writedata) = 7) then
                report "NO ERRORS: Simulation succeeded" severity failure;
            elsif (dataaddr /= 80) then
                report "Simulation failed" severity failure;
            end if;
        end if;
    end process;
end;
```



扫描全能王 创建

7.7 异常*

6.7.2 节中介绍了异常，它引起程序流的意外变化。在本节中，我们将扩展多周期处理器以便支持两类异常：未定义的指令和算术溢出。在其他微结构中支持的异常也遵循类似的原则。

如 6.7.2 节所述，当异常发生时，处理器将 PC 复制到 EPC 寄存器并将异常代码存储在标识异常来源的原因寄存器中。异常原因中，0x28 表示未定义的指令，0x30 表示溢出（见表 6-7）。然后，处理器跳转到存储器地址 0x80000180 处的异常处理程序。异常处理程序是响应异常的代码。它是操作系统的一个部分。

6.7.2 节还指出异常寄存器是协处理器 0 的一个部分，该协处理器也是 MIPS 处理器中用于处理系统功能的一部分。协处理器 0 最多可定义 32 个专用寄存器，包括 EPC 寄存器和原因寄存器。异常处理程序可以使用 `mfc0`（从协处理器 0 移动数据）指令将这些专用寄存器复制到寄存器文件中的通用寄存器中。原因寄存器是协处理器 0 的寄存器 13，EPC 寄存器是协处理器 0 的寄存器 14。

为了处理异常，必须在数据路径中增加 EPC 寄存器和原因寄存器，扩展 PCSrc 复用器来接收异常处理程序地址，如图 7-62 所示。这两个新的寄存器具有写使能（`EPCWrite` 和 `CauseWrite`），在发生异常时存储 PC 和异常原因。为异常选择合适代码的复用器产生异常发生的原因。ALU 还必须产生溢出信号，如 5.2.4 节所述[⊕]。

为支持 `mfc0` 指令，应该增加一路来选择协处理器 0 寄存器并将它们写入寄存器文件，如图 7-63 所示。`mfc0` 指令通过 $Instr_{15:11}$ 来指明协处理器 0 寄存器，在图中仅支持了原因寄存器和 EPC 寄存器。我们为 MemtoReg 复用器增加了另一个输入以便从协处理器 0 中选择值。



扫描全能王 创建

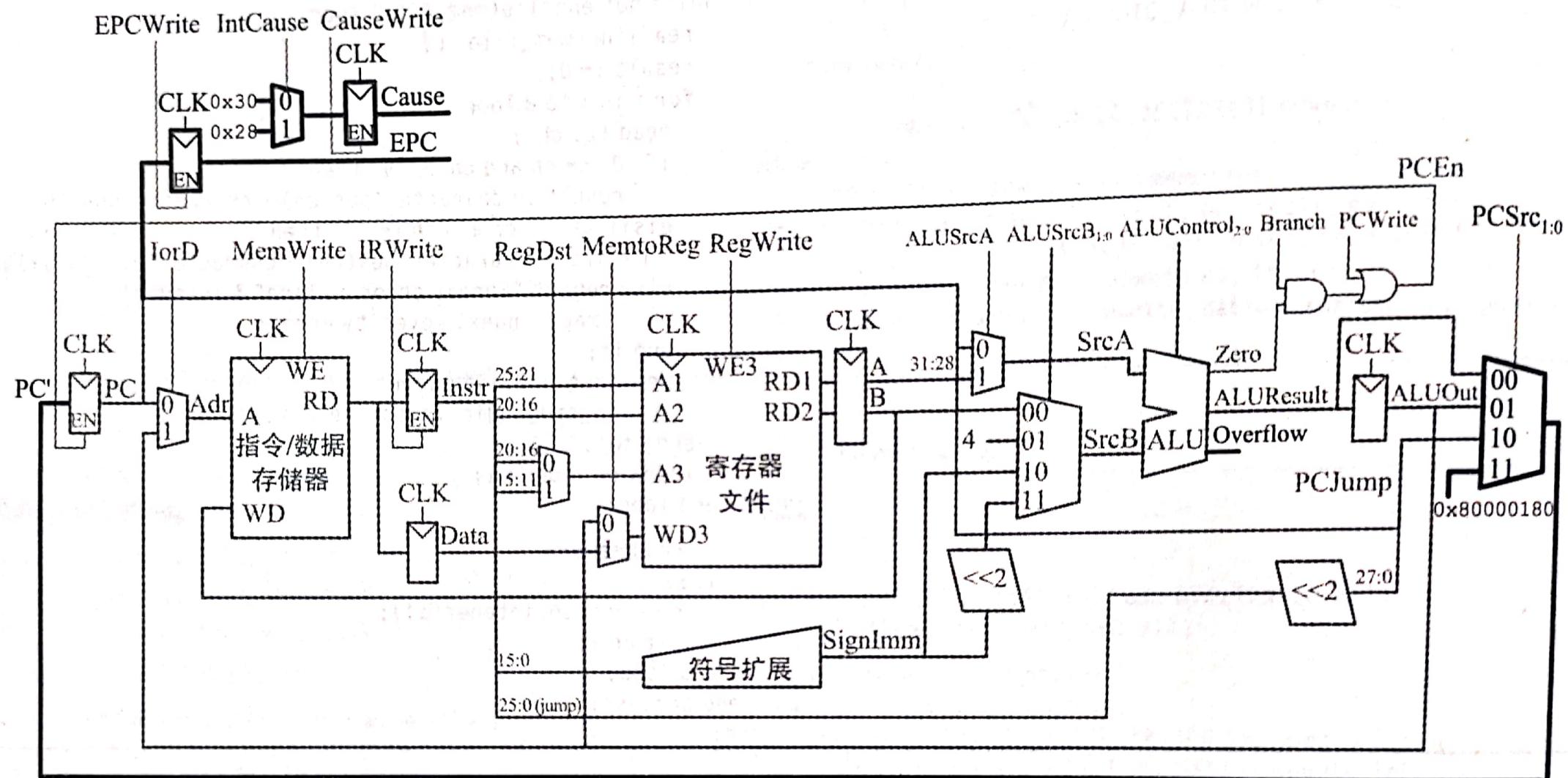


图 7-62 支持溢出和未定义指令异常的数据路径



扫描全能王 创建

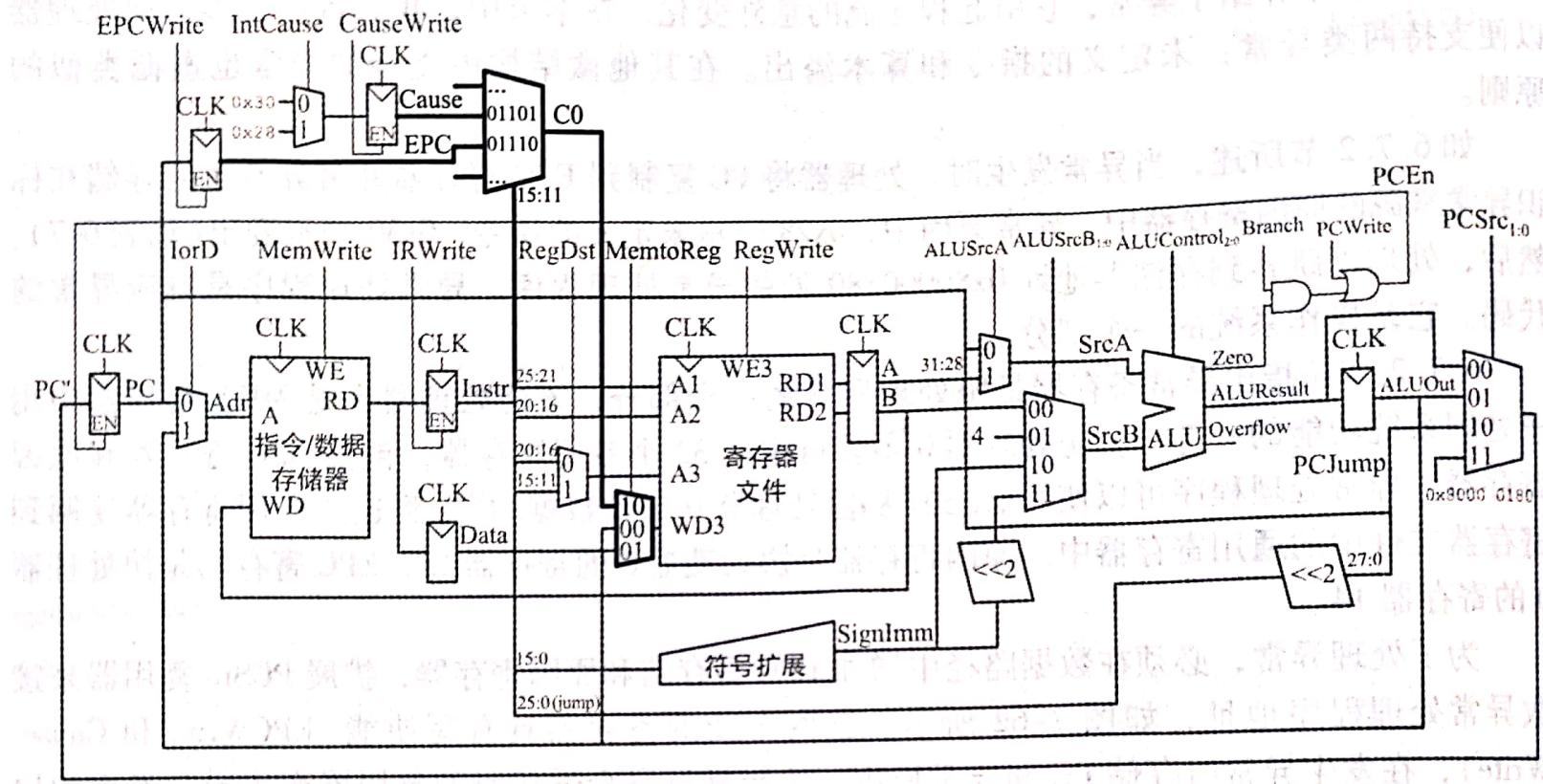


图 7-63 支持 mfc0 指令的数据路径

修改后的控制器如图 7-64 所示。控制器从 ALU 接收溢出标志。它产生 3 个新的控制信号：一个是写 EPC；一个是写原因寄存器；最后一个选择原因寄存器。它还包括支持两个异常的两个新的状态和处理 mfc0 指令的另一个状态。

如果控制器接收到一个未定义的指令（不知道应该如何处理的指令），它转至 S12，将 PC 保存到 EPC 寄存器，向原因寄存器写入 0x28，并跳转到异常处理程序。同样，如果控制器检测到 add 或 sub 指令的算术溢出，它转至 S13，将 PC 保存到 EPC 寄存器，向原因寄存器写入 0x30，并跳转到异常处理程序。需要注意的是，当异常发生时，将抛弃发生异常的指令，其结果也不写入寄存器文件。当译码 mfc0 指令时，处理器进入 S14，并选择合适的协处理器 0 寄存器写入主寄存器文件中。



扫描全能王 创建

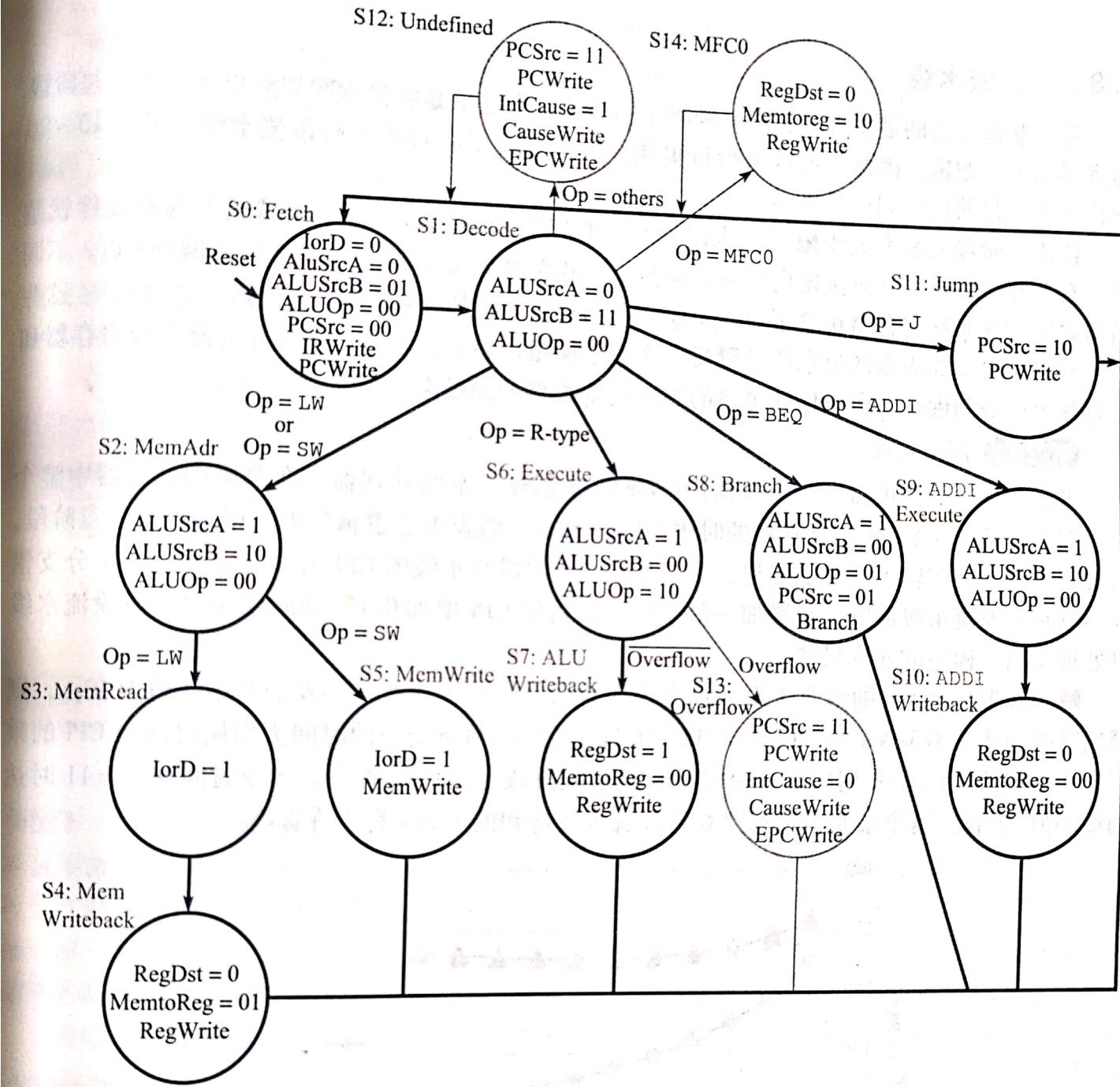


图 7-64 支持异常和 mfc0 的控制器



扫描全能王 创建

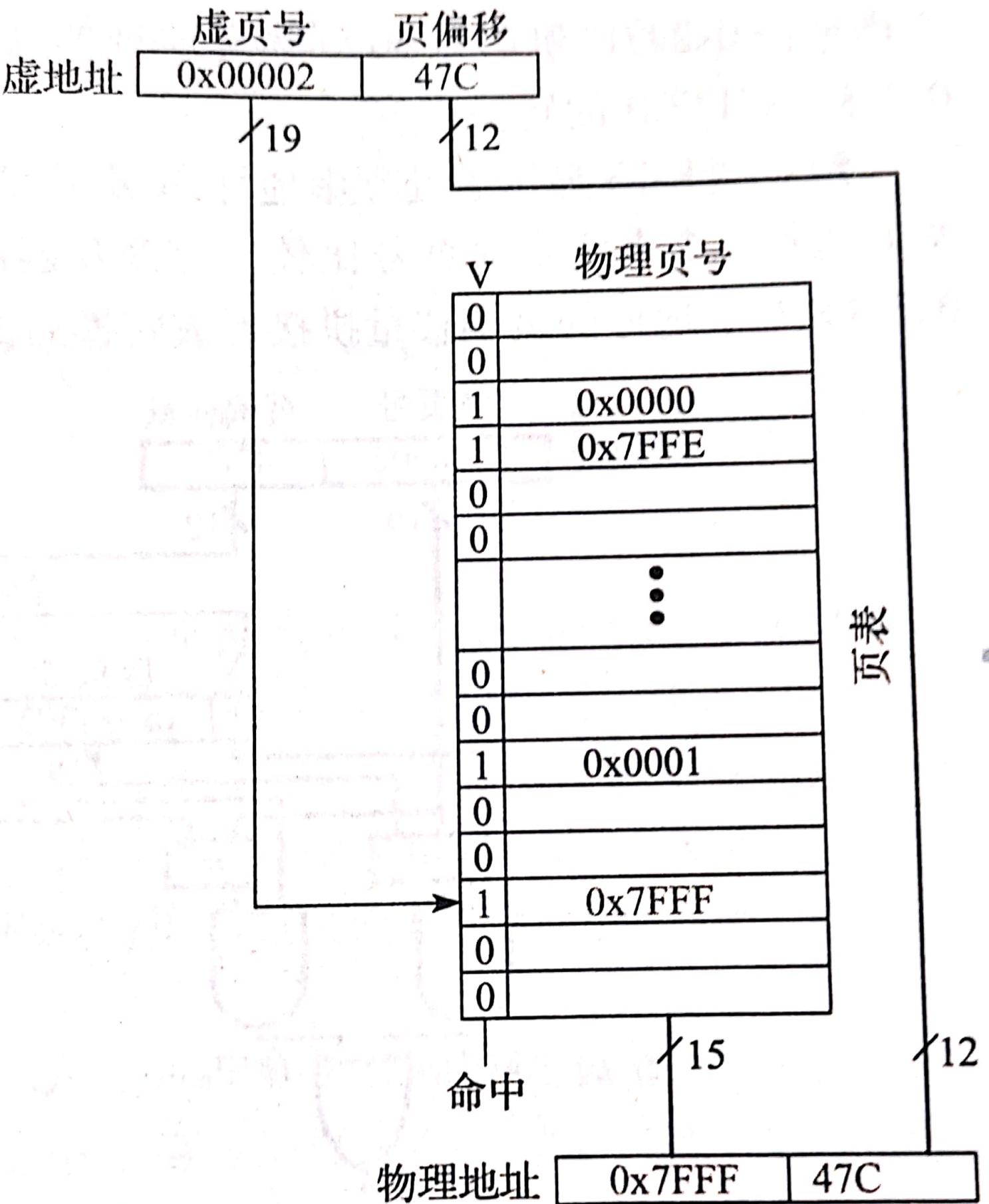


图 8-24 使用页表进行地址转换



扫描全能王 创建

