8 带有中断控制的流水线CPU

1. **SC CPU的时钟周期由最复杂的指令所决定**，每个指令的执行时间均为一个相等的时钟周期。这样的逻辑会使得最简单的指令也需要花费最复杂的指令的执行时间去完成工作

> *The single-cycle (SC) CPU (central processing unit), described in Chapter 5, executes each instruction in one clock cycle, regardless of the complexity of instructions. The cycle time must be determined so that the slowest instruction, lw in our CPU's instruction set, can complete the execution correctly in one clock cycle. Once the cycle time is determined, every instruction uses the same length of time for the execution. This means that, although fast instructions, jump instructions for instance, can complete their executions early, they have to take one whole cycle for the executions.*

2. MC CPU将指令的执行划分为多个阶段，指令按需经过这些阶段（**不需要全经过**）。每个阶段的时间即为一个时钟周期，**时钟周期由最缓慢的功能部件决定**。这样的逻辑思想使得MC CPU通过尽可能地缩短每一条指令的执行延迟来提高性能

> *Compared to the SC CPU, the multiple-cycle (MC) CPU, described in Chapter 7, divides the execution of an instruction into several small steps (short cycles). By using the finite state machine, we can use more or less number of cycles to execute each instruction. For example, we can use five cycles to execute the lw instruction, but we can use only two cycles to execute a jump instruction (jr, j, or jal). That is, the MC CPU improves performance by making the execution latency of each instruction as short as possible. The cycle time of the MC CPU is determined by the slowest functional unit, the arithmetic logic unit (ALU) or memory, for instance.*

3. **SC、MC CPU的指令均必须在上一条指令流出之后才可流入**

4. 带有流水线的CPU虽然使得每条指令的执行均需要经过5个节点，但是流水线的存在使得每个时钟周期均可以流入一条指令，从而使得每个时钟周期均有结果流出➞**利用ISP增大了吞吐率**

> *The common feature of both SC CPU and MC CPU is that the execution of an instruction can be started only after the execution of its predecessor (prior instruction) is completed. This chapter introduces how to design a five-stage pipelined CPU which allows overlapping execution of multiple instructions. In the five-stage pipelined CPU, although an instruction takes five clock cycles to pass through the pipeline, a new instruction can enter the pipeline during every clock cycle. Under ideal circumstances, the pipelined CPU can produce a result in every clock cycle. In contrast, the MC CPU produces a result in every 2 - 5 clock cycles. That is, the pipelined CPU improves the throughput, instead of shortening the execution latency of each instruction*

英语

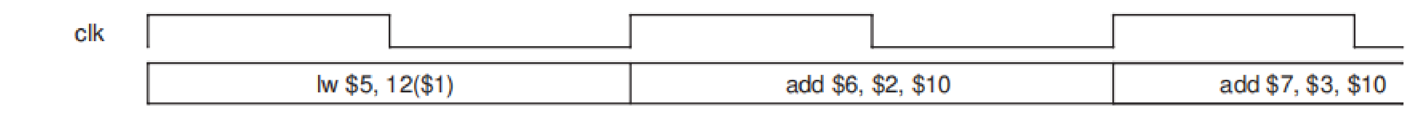
1. *latency*潜伏、延迟
2. *predecessor*祖先
3. *assembly*装配线
4. *recrute*招收
5. *successive*连续的、相继的
6. *state-of-the-art* 最新的

1. *throughput*吞吐量
2. overlapping重叠
3. *dedictaed*专用的
4. *hazard*冒险
5. *trade-off*折中
6. *in addition to*除了...

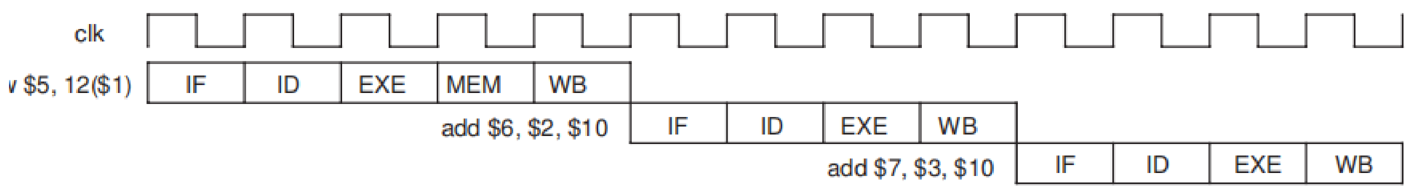
8.1 流水线 CPU的基本组成

流水线的一些基本概念这里不再赘述，可以查阅流水线的基本概念

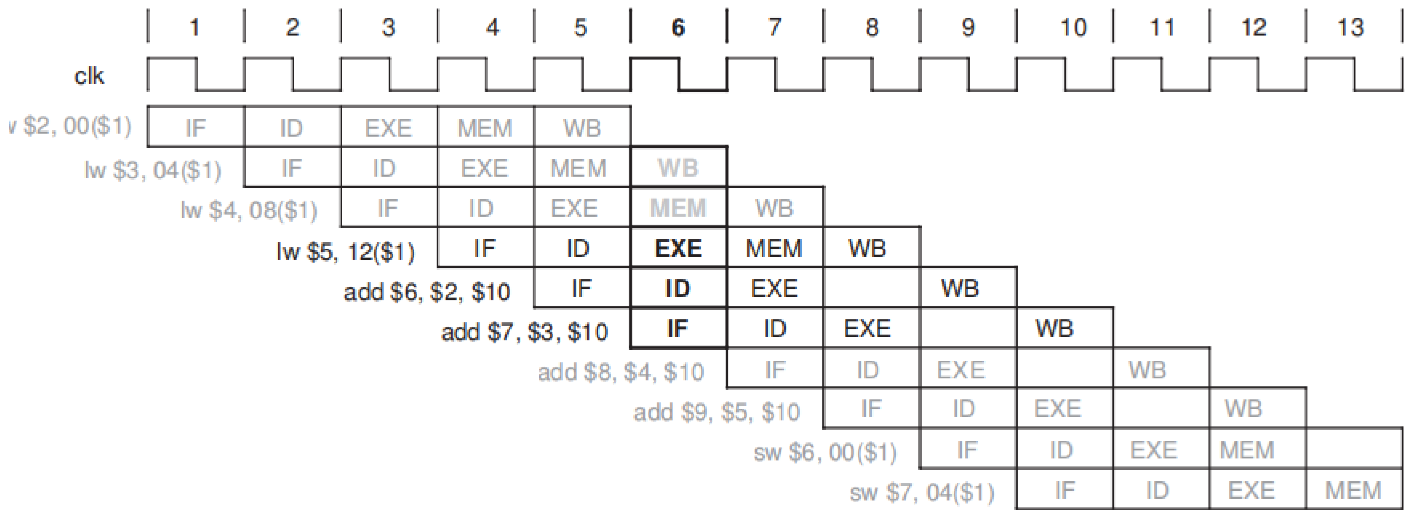
8.1.1 CPU性能分析——吞吐量



(a) Timing chart of single cycle CPU



(b) Timing chart of multiple cycle CPU



(c) Timing chart of pipelined CPU

Figure 8.2 Timing comparison between the three types of CPUs

左图为三种类型的CPU执行同样一个指令序列所花费的时间对比图

对于一个具有m段流水的CPU而言，执行n条指令，需要的总时间是(n+m-1)*TPC；对于MC CPU而言执行的时间是n*k*TPC,k为平均时钟周期数；对于SC CPU而言执行时间是n*m*TPC

因此流水线CPU对于SC CPU而言的加速比是$\frac{n*m}{n+m-1}$，对于MC CPU而言的加速比是$\frac{n*k}{n+m-1}$

当n→∞时，前者值为m后者值为k

8.1.2流水线CPU的Baseline

因为流水线的存在，使得在一个时钟周期内，多个操作在重叠执行。为了使接下来的流水段用到之前所产生的数据，需要使用流水线寄存器来暂存之前流水段所产生的结果

流水线CPU也将指令的执行划分为IF、ID、EXE、MEM、WB这五个阶段，流水线段与段之间存在着流水线寄存器，PC和WB的写回寄存器也被视为流水线寄存器，一个是表示开始，一个是表示结束。因此共有6个流水线寄存器：PC、IF/ID、ID/EXE、EXE/MEM、MEM/WB、writeRegAddr。除了PC和writeRegAddr外，其余流水线寄存器是透明的，不能直接读取到其中的内容

接下来以下图的指令序列，介绍五个阶段的工作情况

```
# address   instruction           comment
  100:      lw   $2, 00($1)       # $2 <-- memory[$1+00];  load x[0]
  104:      lw   $3, 04($1)       # $3 <-- memory[$1+04];  load x[1]
  108:      lw   $4, 08($1)       # $4 <-- memory[$1+08];  load x[2]
  112:      lw   $5, 12($1)       # $5 <-- memory[$1+12];  load x[3]
  116:      add $6, $2, $10       # $6 <-- $2 + $10;     x[0] = x[0] + s
  120:      add $7, $3, $10       # $7 <-- $3 + $10;     x[1] = x[1] + s
  124:      add $8, $4, $10       # $8 <-- $4 + $10;     x[2] = x[2] + s
  128:      add $9, $5, $10       # $9 <-- $5 + $10;     x[3] = x[3] + s
  132:      sw   $6, 00($1)       # memory[$1+00] <-- $6; store x[0]
  136:      sw   $7, 04($1)       # memory[$1+04] <-- $7; store x[1]
  140:      sw   $8, 08($1)       # memory[$1+08] <-- $8; store x[2]
  144:      sw   $9, 12($1)       # memory[$1+12] <-- $9; store x[3]
```

8.1.2.1取指阶段示意图

取指阶段，**在PC寄存器和IF/ID寄存器之间存在着指令存储器insROM和专用的加法器Adder**。PC的值是100，在时钟上升沿，根据地址100从指令存储器中取出指令并写入IF/ID寄存器，同时NPC(可能是Adder的结果PC+4，也可能是BPC、JPC、RPC、Base、EPC等)也被写入PC

> *In the IF stage, there is an instruction memory module and an adder between two pipeline registers. The leftmost pipeline register is the PC; it holds 100. In the end of the first cycle (at the rising edge of clk), the instruction fetched from instruction memory is written into the IF/ID register (the first-year students are promoted to second year). Meanwhile, the output of the adder (PC + 4, the next PC) is written into PC (for recruiting new first-year students).*

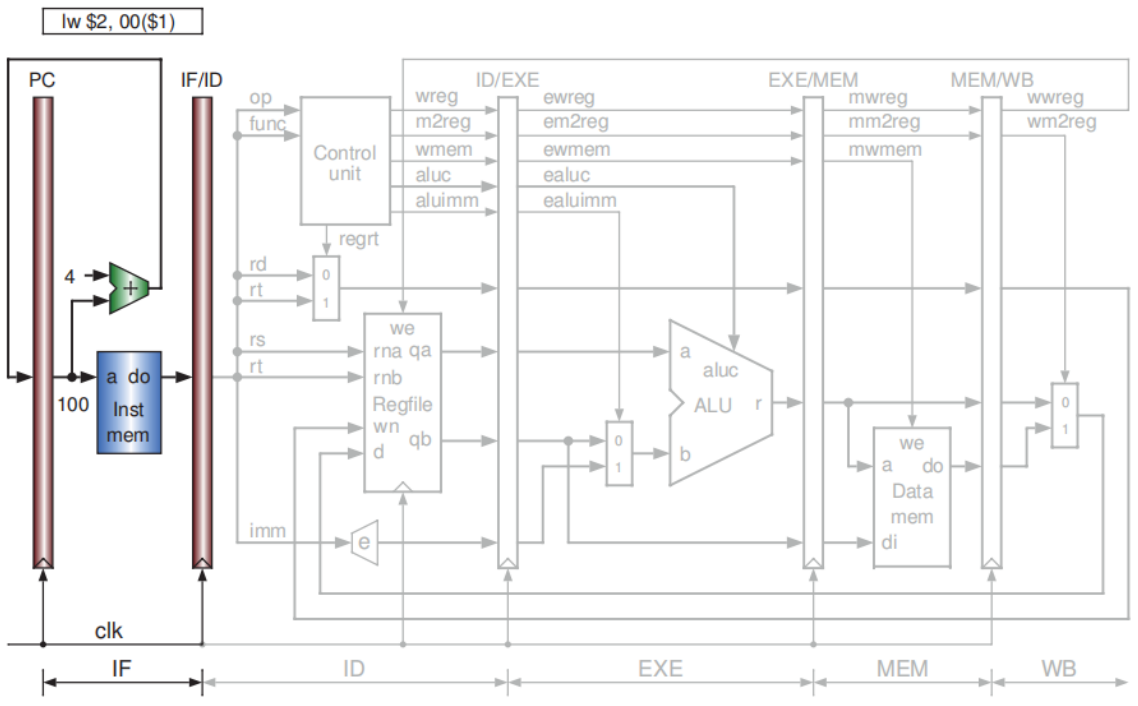


Figure 8.3 Pipeline instruction fetch (IF) stage

8.1.2.2译码阶段示意图

在第二个时钟周期，104地址处的指令进入了IF阶段，100地址处的指令进入了译码阶段。根据这条进入了ID阶段的指令，将控制信号送至CU进行译码，并从寄存器文件中读取rs、rt地址的操作数（尽管lw指令不用rt地址上的操作数）。立即数被sext信号进行扩展，并使用regdst、jal控制信号进行写寄存器的地址选择。其他之后会被用到的控制信号、数据均写入ID/EXE寄存器

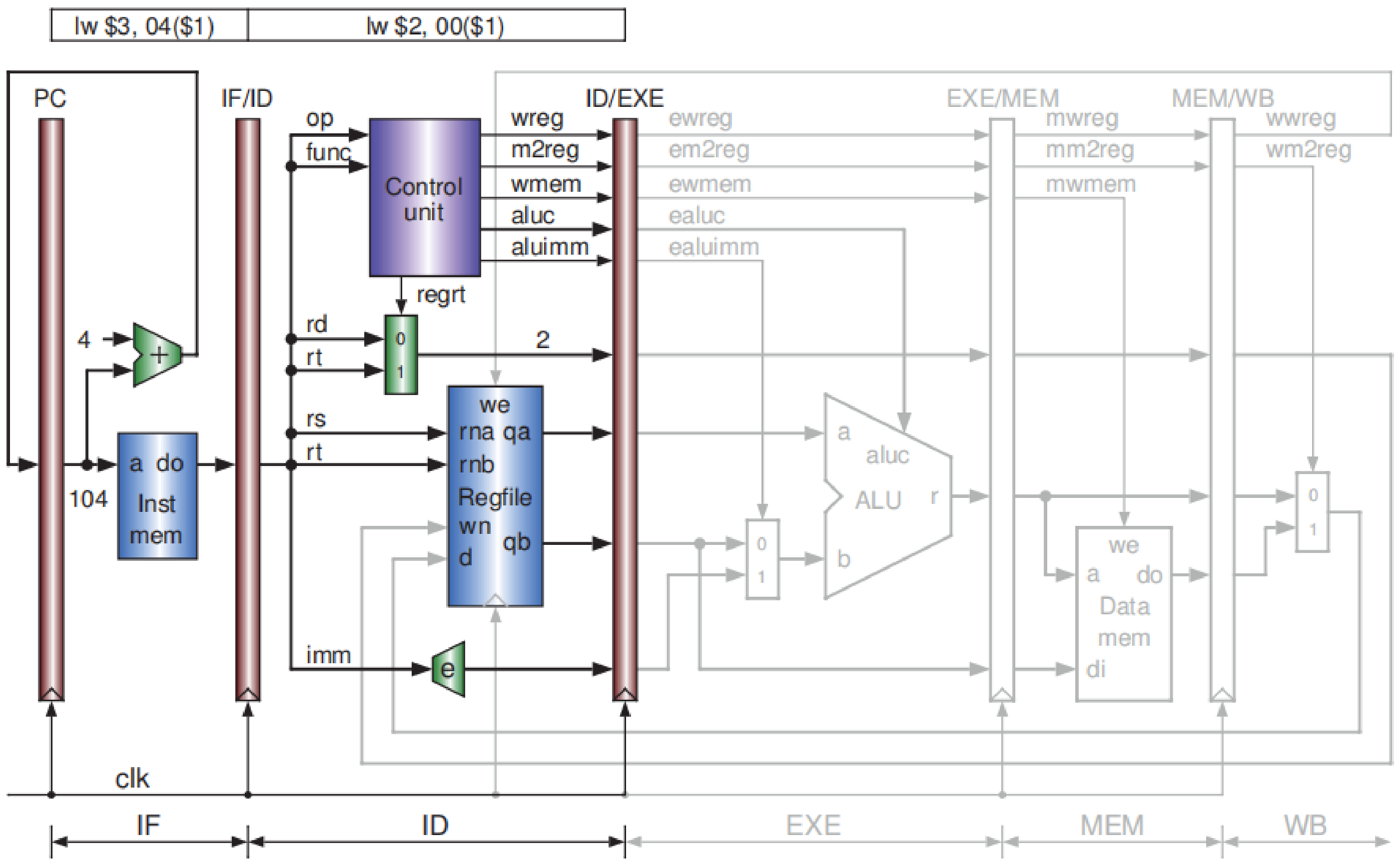


Figure 8.4 Pipeline instruction decode (ID) stage

8.1.2.3执行阶段示意图

在ID/EXE和EXE/MEM之间的流水线的主要部件是多个多路选择器以及ALU，所有的控制信号加前缀“e”

这些多个多路选择器主要是根据ID/EXE流水线寄存器所传至执行阶段的操作数以及控制信号进行选择，选择结果送至ALU的操作数输入端，根据控制信号aluc进行对应的运算

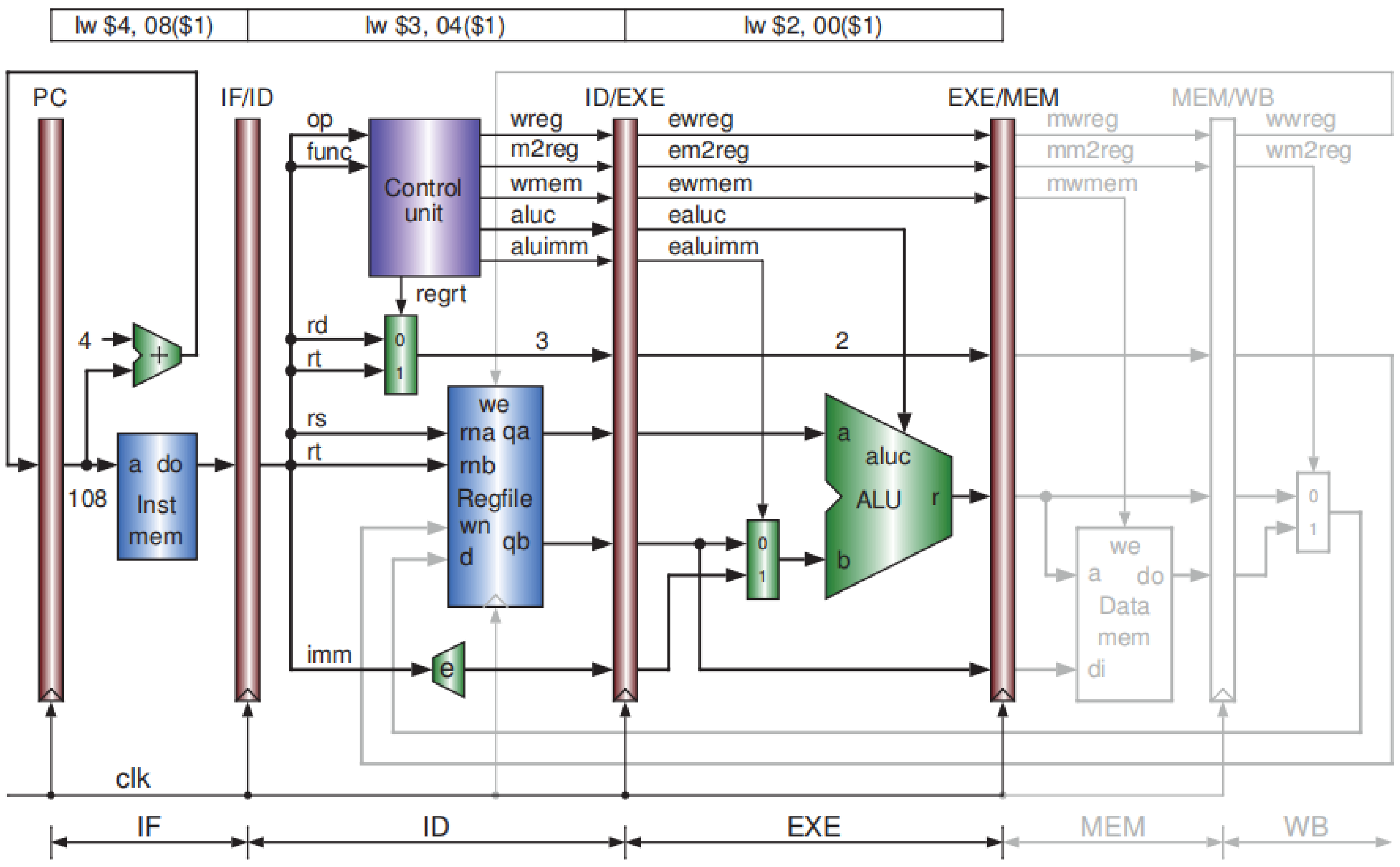


Figure 8.5 Pipeline execution (EXE) stage

8.1.2.4访存阶段

访存阶段主要是lw、sw两条指令，之间的部件主要是**数据存储器dataRAM**，所有的控制信号前缀加“m”

lw、sw均根据EXE/MEM流水线寄存器所传递的ALU的运算结果作为访存的地址，lw取出对应地址上的单元输出，sw则将流水线寄存器传递的qb在wmem的作用下写至存储器

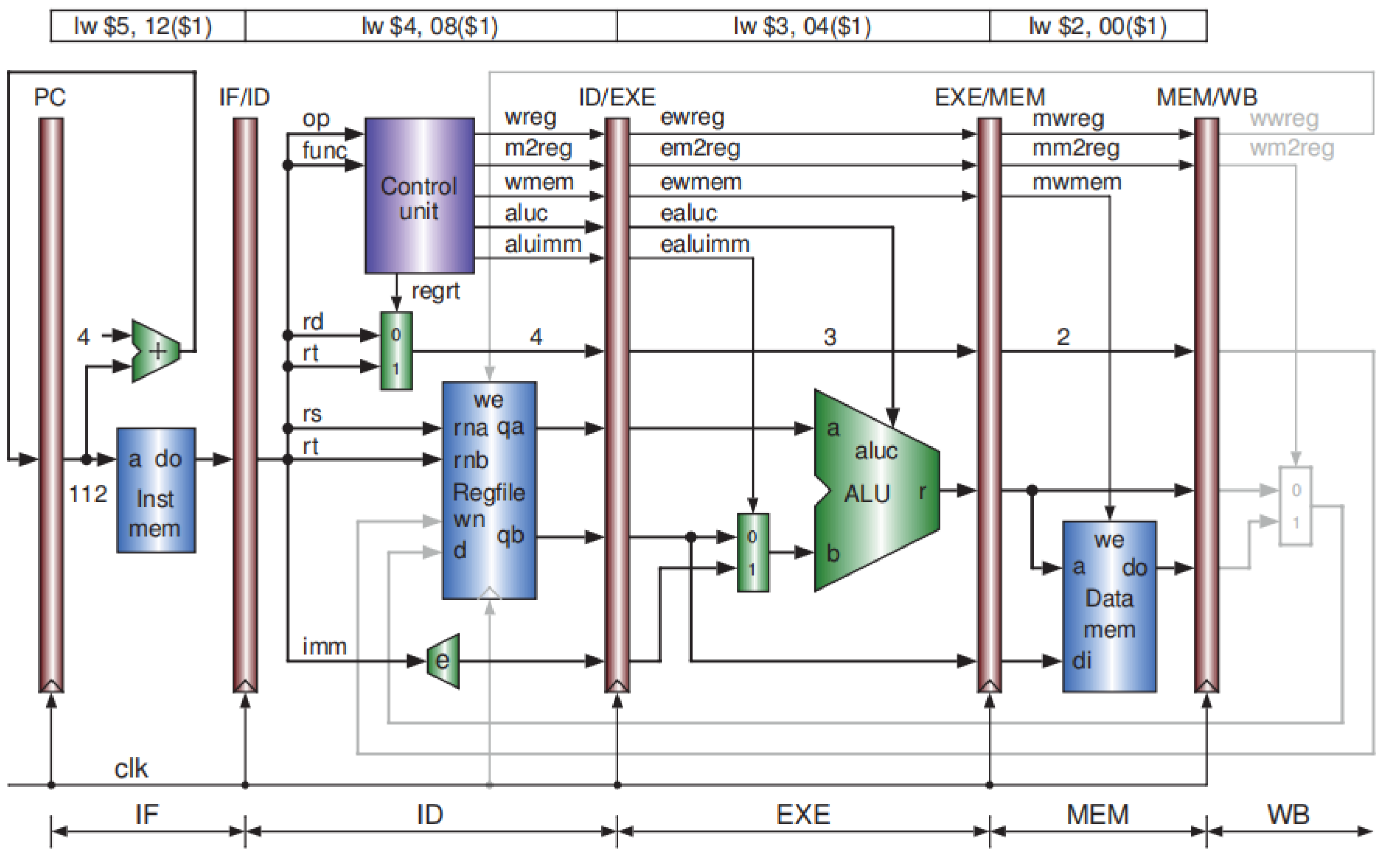


Figure 8.6 Pipeline memory access (MEM) stage

8.1.2.5写回阶段

写回阶段的主要部件即为多路选择器，所有的控制信号前缀加“w”，根据m2reg信号选择写入寄存器数据，然后传送至ID阶段的寄存器文件写

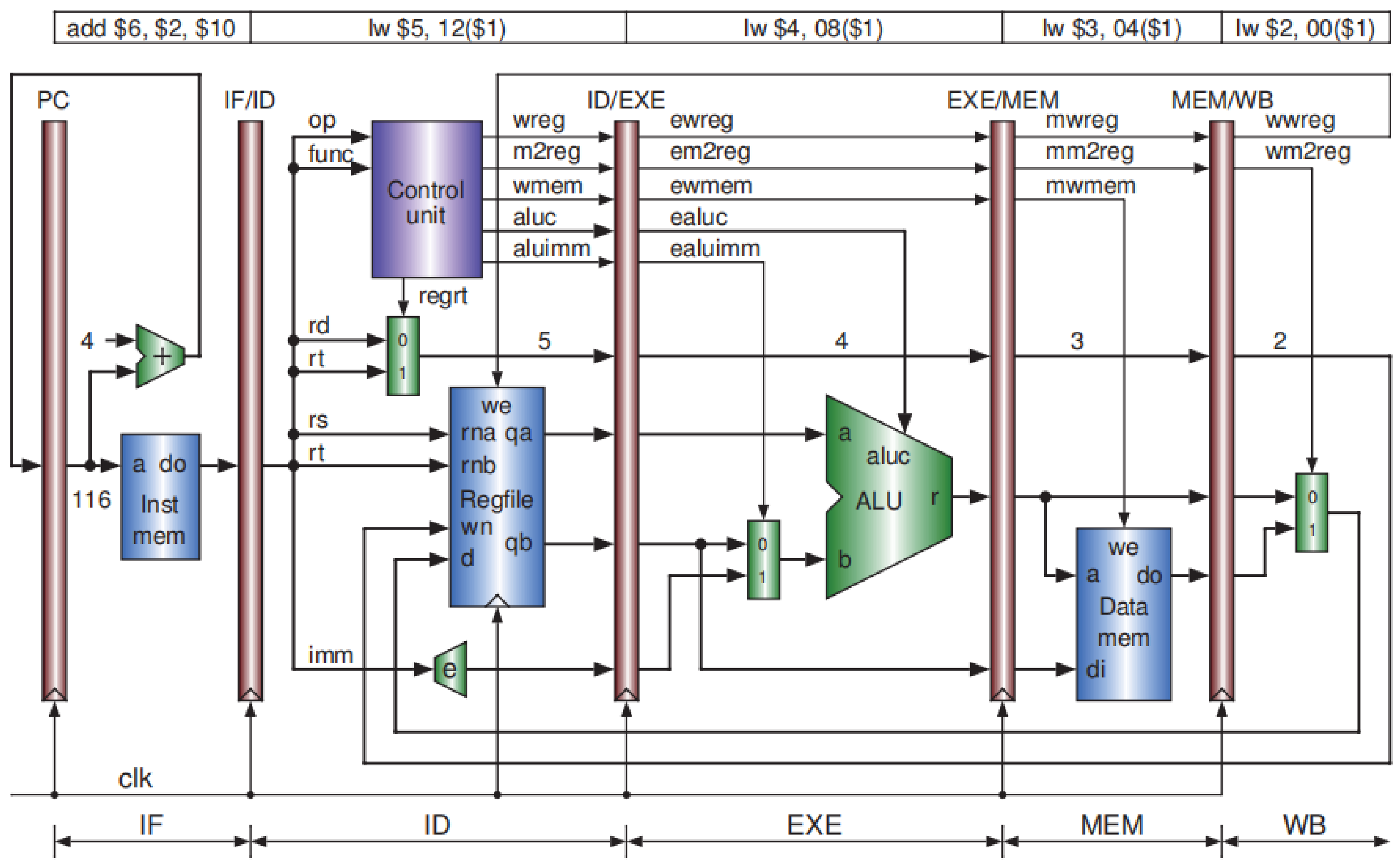


Figure 8.7 Pipeline write back (WB) stage

📌所有的六个流水线寄存器均在时钟上沿更新

8.2流水线冒险和解决方法

在理想的条件下，流水线化的CPU可以在每个时钟周期中执行一条有用的指令。由于多个指令的重叠执行导致的指令级并行问题，流水线化的CPU可能会遇到一些指令无法被处理或它将无法正确执行的情况。这些情况被称为冒险。在流水线化的CPU中有三种类型的冒险：结构冒险、数据冒险和控制冒险

> *Under ideal conditions, the pipelined CPU can execute a useful instruction during every clock cycle. Because of the overlapping execution of multiple instructions, the pipelined CPU may encounter some situations where an instruction cannot be processed or it will not execute correctly. These situations are called hazards. There are three types of hazards in a pipelined CPU: structural hazards, control hazards, and data hazards.*

8.2.1结构冒险和解决方法

结构冒险通常发生在两个或两个以上指令尝试同时使用一个只能服务一个请求的硬件的情况。结构冒险的例子比如：当只有一个存储单元时指令访存和数据访存冲突；当执行阶段用时较长，ALU被之前的指令始终占用导致的ALU使用冲突；同时两个指令写寄存器的端口冲突

📌一般来说，任何缺乏足够的硬件资源导致流水线阻塞都是一种结构冒险。由于目前的集成电路（IC）技术允许制造商在芯片上制造大量的晶体管，因此大多数的结构冒险都可以得到解决。

> *Generally, any lack of sufficient hardware resources that results in pipeline stalls is a structural hazard. Because the current integrated circuit (IC) technology allows makers to fabricate a large number of transistors on a chip, most of the structural hazards can be solved.*

例如上述的例子，可以使用分离指令存储和数据存储、实现多个ALU/流水线乘法器、除法器[注释1]、寄存器文件两个写端口

8.2.2数据冒险RAW和前递技术

数据冒险也被称为数据依赖。它意味着一个指令i使用指令i-1的执行结果。因为指令i-1的执行结果在WB阶段被写入了对应的寄存器中，但是指令i是在ID阶段就读操作数，由于WB在ID的后两个阶段，所以指令i可能会读出一个错误的数据，可能会导致结果的执行错误，这样的情况称之为“数据冒险”

> A data hazard is also known as data dependency. It means that an instruction i uses the execution result of its predecessor (instruction i − 1). Because the execution result of instruction i − 1 is written into register file at the end of its WB stage, while the instruction i reads the register contents in its ID stage, a data word read from the register file is not the state-of-the-art result produced by instruction i − 1. Such situations are called data hazards.

如下图的数据冒险的例子：

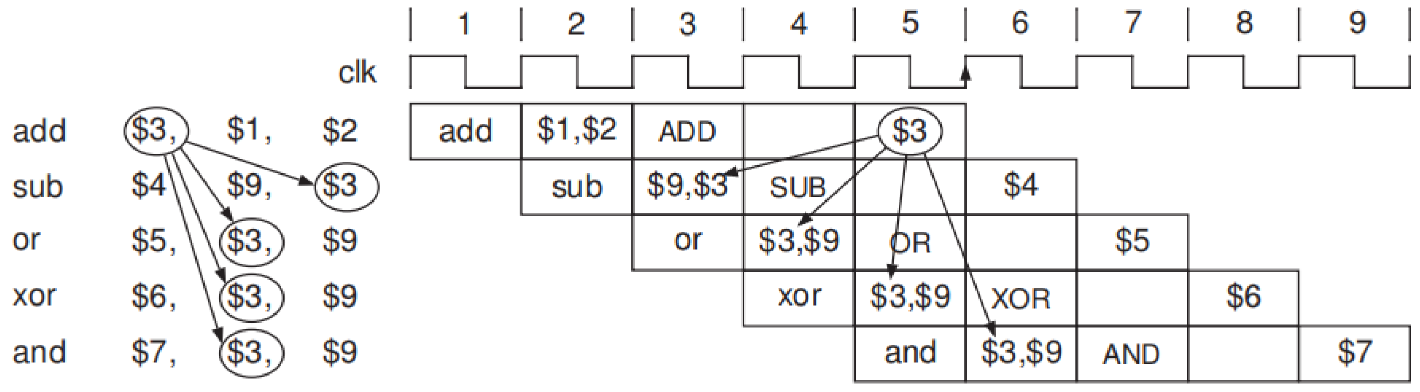


Figure 8.10 Data hazard examples

sub、or、xor、and指令均使用到了add的结果，按照甘特图，sub、or、xor均不能正确的读出数据

流水线阻塞技术

一个解决数据冒险的方法是流水线阻塞技术，即在检测到流水线数据冒险后，人为的插入若干个空时钟周期至数据被写入寄存器之后（图中的T6）再读。这样的做法比较简单，但是会延长指令的执行时间，降低性能

前递技术

另一个解决数据冒险的方法是前递技术，即不必等到写回阶段写入寄存器后再读，而是产生结果后就将这个结果送至需要的指令处，如下图所示

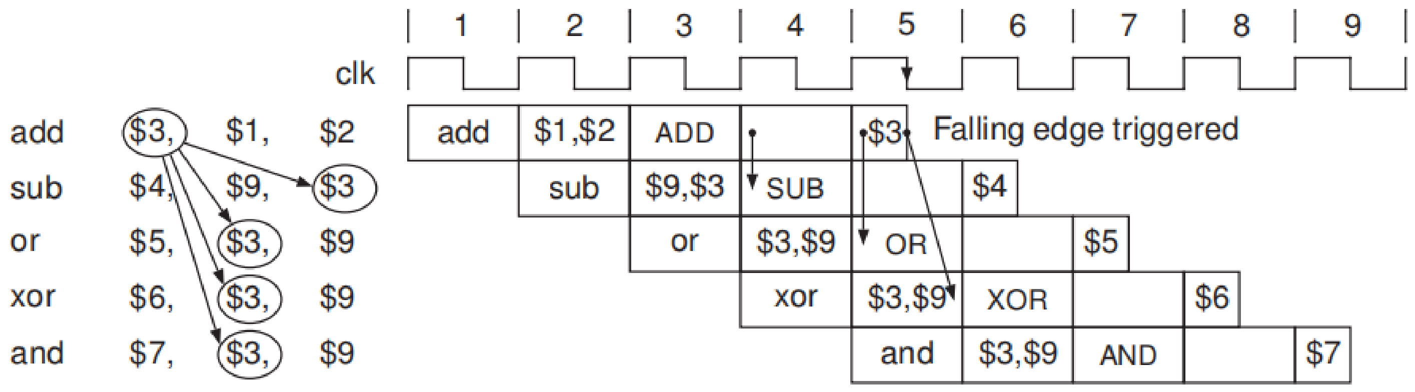


Figure 8.11 Internal forwarding to the EXE stage

当sub开始做减法时（进入EXE阶段），add指令已经完成了算术工作，其结果已保存至EXE/MEM流水线寄存器中，那么就可以搭建一条通路，将EXE/MEM中的结果送至sub的EXE阶段；

同样or指令在进入EXE阶段时，add指令已经在WB阶段，算术结果保存在MEM/WB寄存器中，那么同样可以搭建一条通路将MEM/WB中的结果送至or的EXE阶段；

xor在进入EXE阶段时，add指令已完成WB阶段，我们也可以设置写寄存器在第一个时钟半沿，那么在xor的EXE上沿时就可以直接从寄存器中读出最新的数据。

如上所述，使用前递技术解决了这三条指令的数据冒险

📌**但是结合后面的分支指令提前判断是否分支时要在ID阶段就读取寄存器数进行判断，这也涉及到了数据冒险的问题，所以我们整合两种情况，将数据冒险的前递设置在ID阶段**，设计如下：

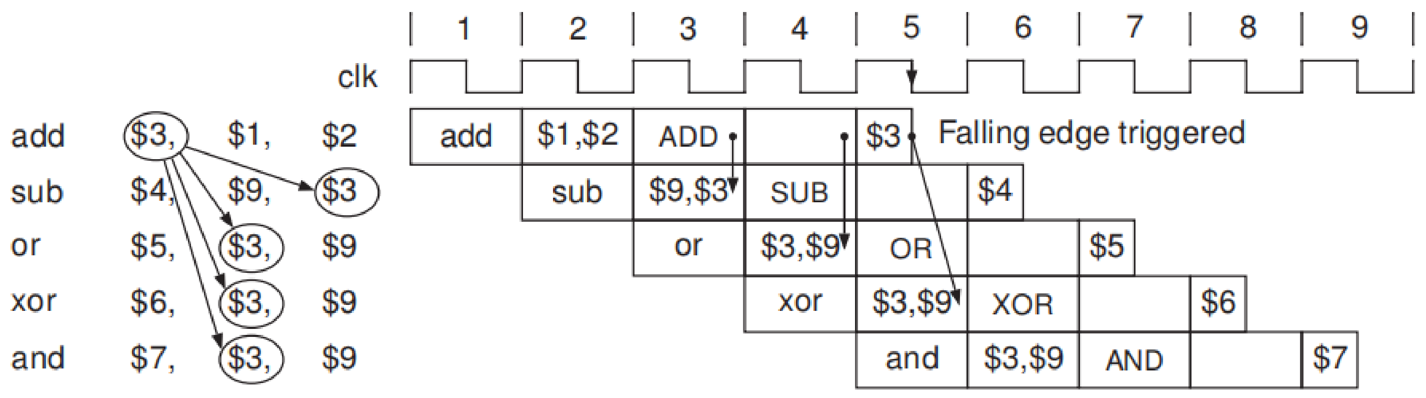


Figure 8.12 Internal forwarding to the ID stage

检测数据冒险，将EXE的ALU执行结果、MEM的ALU执行结果和MEM的访存结果送至需要的地方

但是会有load指令的特殊情况，“load-delay”需要特殊对待

如下，load指令的结果产生在MEM阶段的memread，这会导致紧邻load的下一条指令无法从EXE的aluout获取到需要的值（其他后续指令均可正常获取），这时候就必须采用流水线阻塞技术，将紧邻的这条指令延迟一个时钟周期，然后将MEM阶段的memread送至ID/EXE流水线寄存器

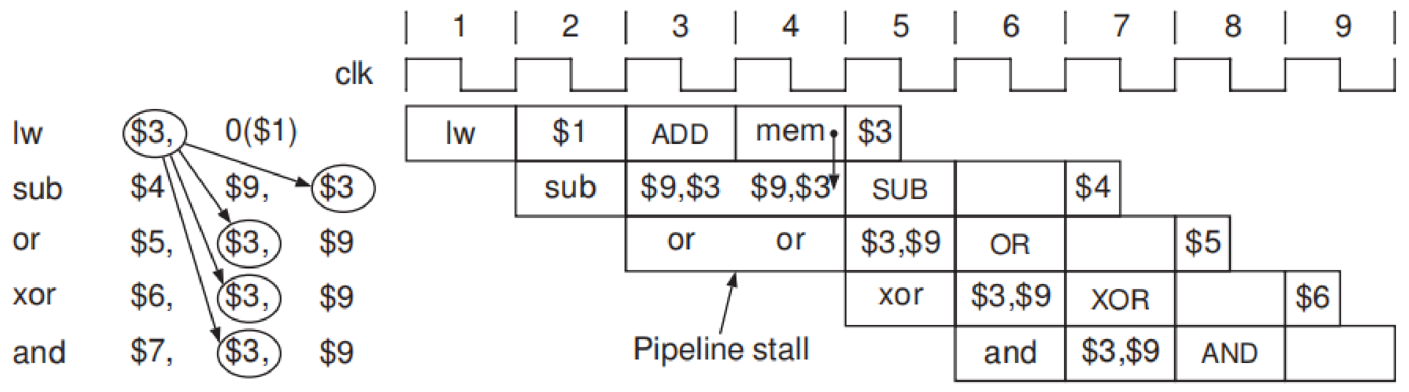


Figure 8.13 Data hazard with load instruction

前递技术+阻塞技术的实现

1. clk反向接寄存器文件的脉冲输入→写寄存器在第一个半沿完成
2. 判断执行阶段是否是load指令，如果是且指令需要使用rs、rt寄存器且rs、rt与写寄存器地址有相等，则PC、IF/ID流水线寄存器写使能无效，并暂停当前指令的执行[注释2]
3. 如果执行阶段不是load指令，但写寄存器信号有效，那么判断当前译码阶段rs、rt与写寄存器地址是否有相等，如果有则将执行阶段ALUout送至ID/EXE
4. 判断MEM阶段是否是写寄存器，如果是则判断当前译码阶段rs、rt与写寄存器地址是否有相等，如果有相等且不是load，则将mem阶段的ALUout送至ID/EXE，如果是load则送MEM阶段的memread

5. 没有数据冒险，则正常将寄存器rs、rt地址上的数据送ID/EXE流水线寄存器

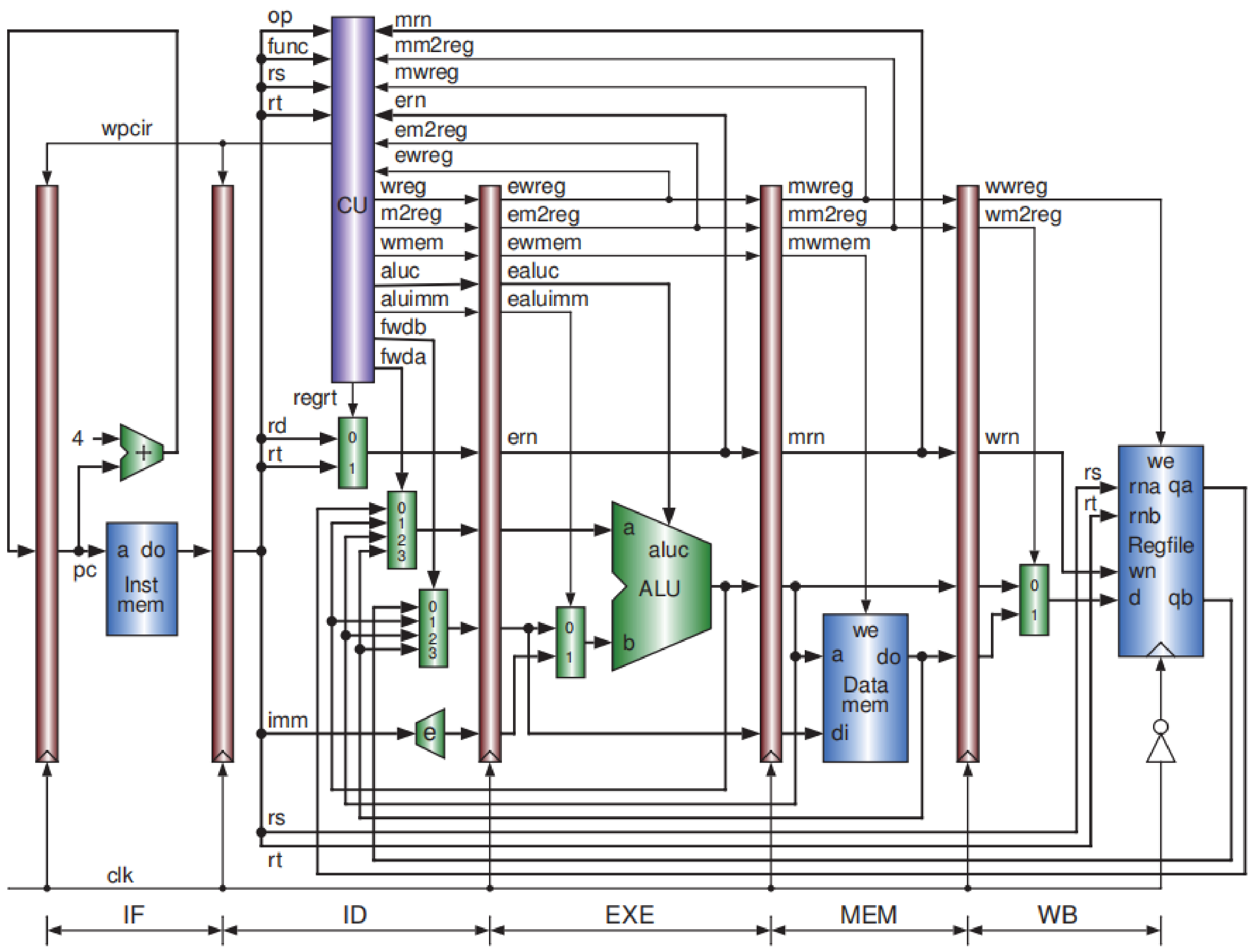


Figure 8.14 Mechanism of internal forwarding and pipeline stall

```
//where i_rs and i_rt indicate that an instruction uses the contents of the rs register and the rt
register,
//respectively.stall inverse as the wpcir
stall = ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) | i_rt & (ern == rt));
// forward control signal for alu input a
fwda = 2'b00; // default: 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
    fwba = 2'b00; // default: no hazards
    if (ewreg & (ern != 0) & (ern == rt) & ~em2reg) begin
        fwda = 2'b01; // select exe_alu
    end else begin
        if (mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg) begin
            fwda = 2'b10; // select mem_alu
        end else begin
            if (mwreg & (mrn != 0) & (mrn == rt) & mm2reg) begin
                fwda = 2'b11; // select mem_lw
            end
        end
    end
```

Verilog

📌**stall信号反向作为PC、IF/ID寄存器的写使能信号**，但是之前的指令已被保存在IF/ID寄存器中，这样会使得它被执行了一次，然后stall信号无效后，又会再执行一次，如下图：

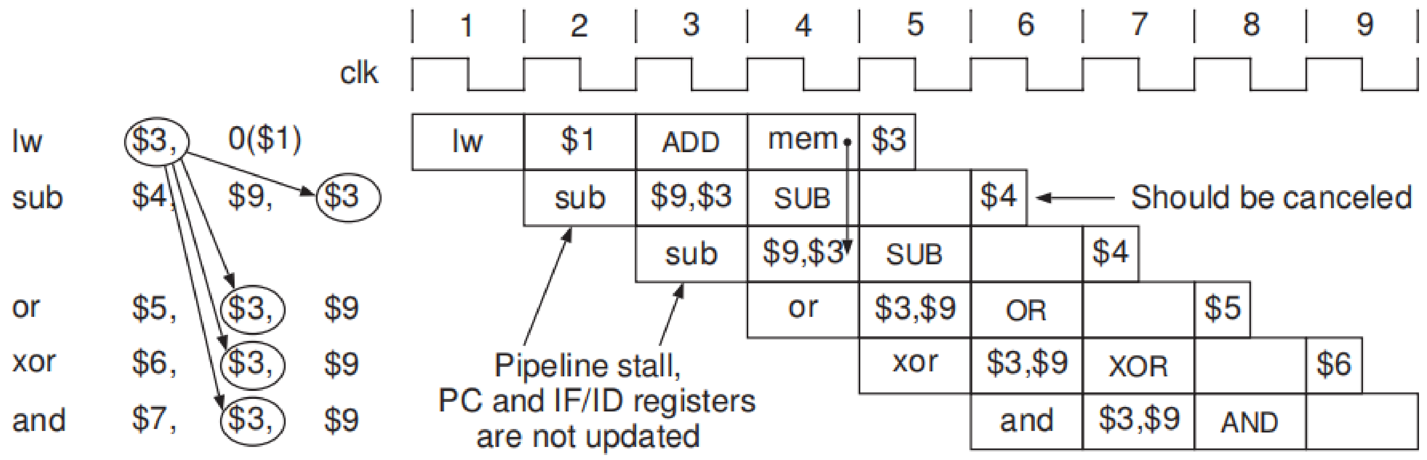


Figure 8.16 Pipeline stall causing an instruction to be executed twice

因此我们需要取消第一次的执行，可以使用stall信号生成所有的写控制信号，通过将写控制信号置0来取消指令的执行，得到正确的执行甘特图如下：

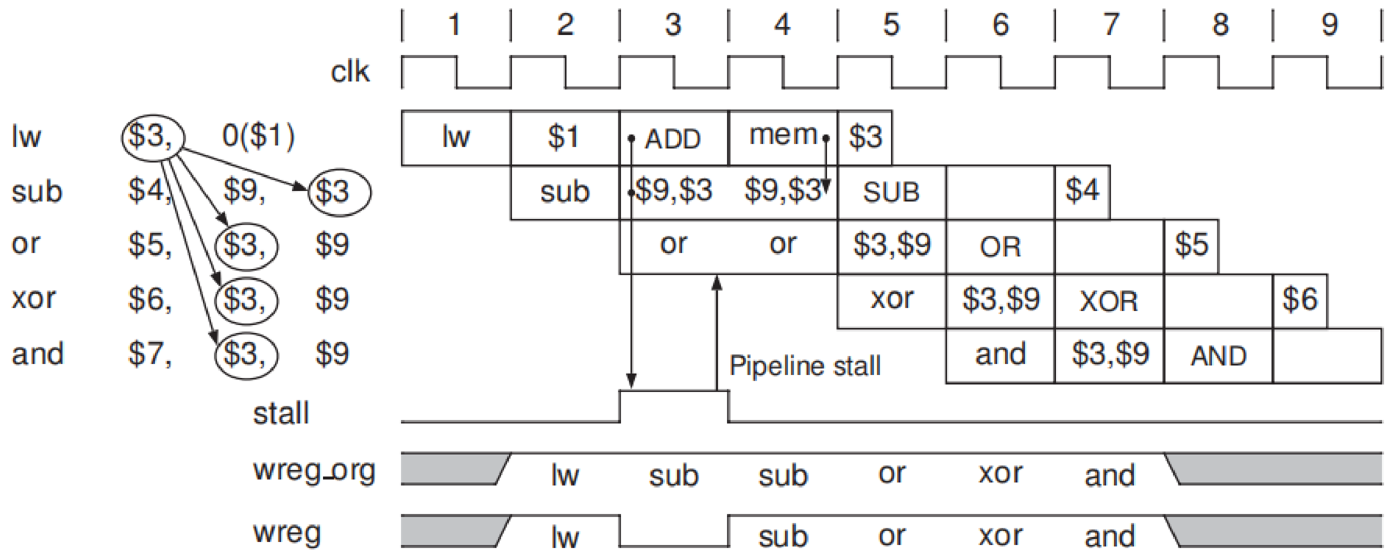


Figure 8.15 Implementing pipeline stall and instruction cancellation

8.2.3控制冒险和分支延迟

当流水化的CPU执行一个分支、跳转指令时就会发生控制冒险。跳转指令较分支指令的处理更为简单，因为跳转指令在ID阶段即可更新PC，只需要解决ID阶段取出来的下一条指令即可；而分支指令的判断是在EXE阶段，此时下一条指令已经进入了ID阶段，下下条指令已被取出

因此首先需要将分支指令处理成跳转指令相同的情况，然后再对取出的下一条指令进行处理：

提前分支判断到ID阶段→涉及到了数据冒险，已在8.2.2解决（将前递提前到ID的原因）

```
    rsrtequ   = ~|(da^db);                          // (da == db)
```

对下一条指令的处理有两种方法:

取消指令的执行

如果当前指令是分支指令/跳转指令，那么就产生一个stall信号，置0所有的写信号，并阻塞PC、IF/ID的更新

分支延迟

我们将跳转/分支指令和跳转/分支目标地址之间的位置叫做延迟槽

MIPS指令集采用的是延迟槽的机制：无论是否进行分支，位于延迟槽中的指令将始终被执行，如下图8.18。也因为延迟槽的存在，jal写$31的数据将是PC+8，如下图8.20

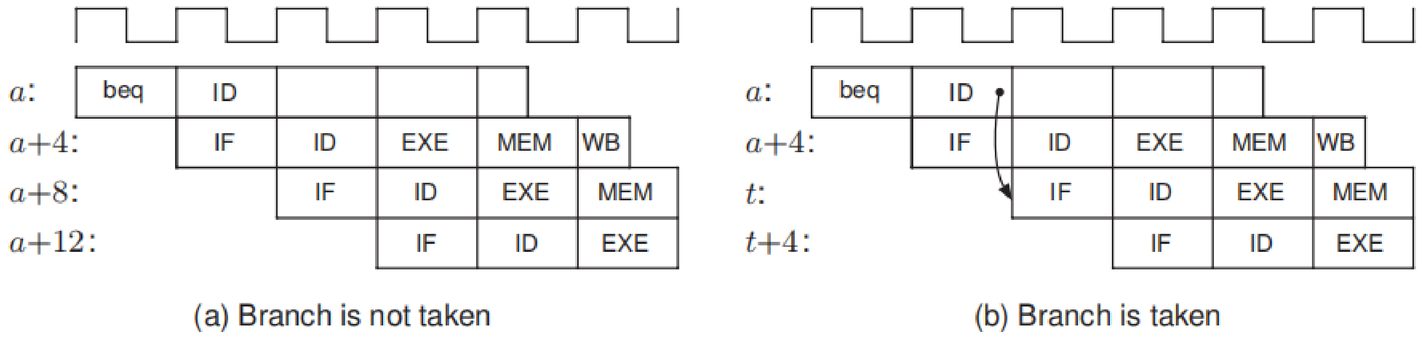


Figure 8.18 Delayed branch

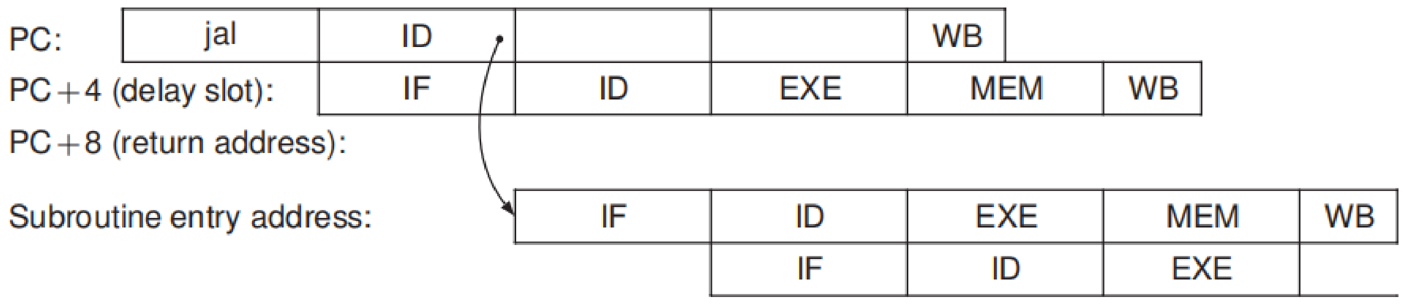


Figure 8.20 Return address of the function call instruction

使用分支延迟技术解决控制冒险的示意图如下:

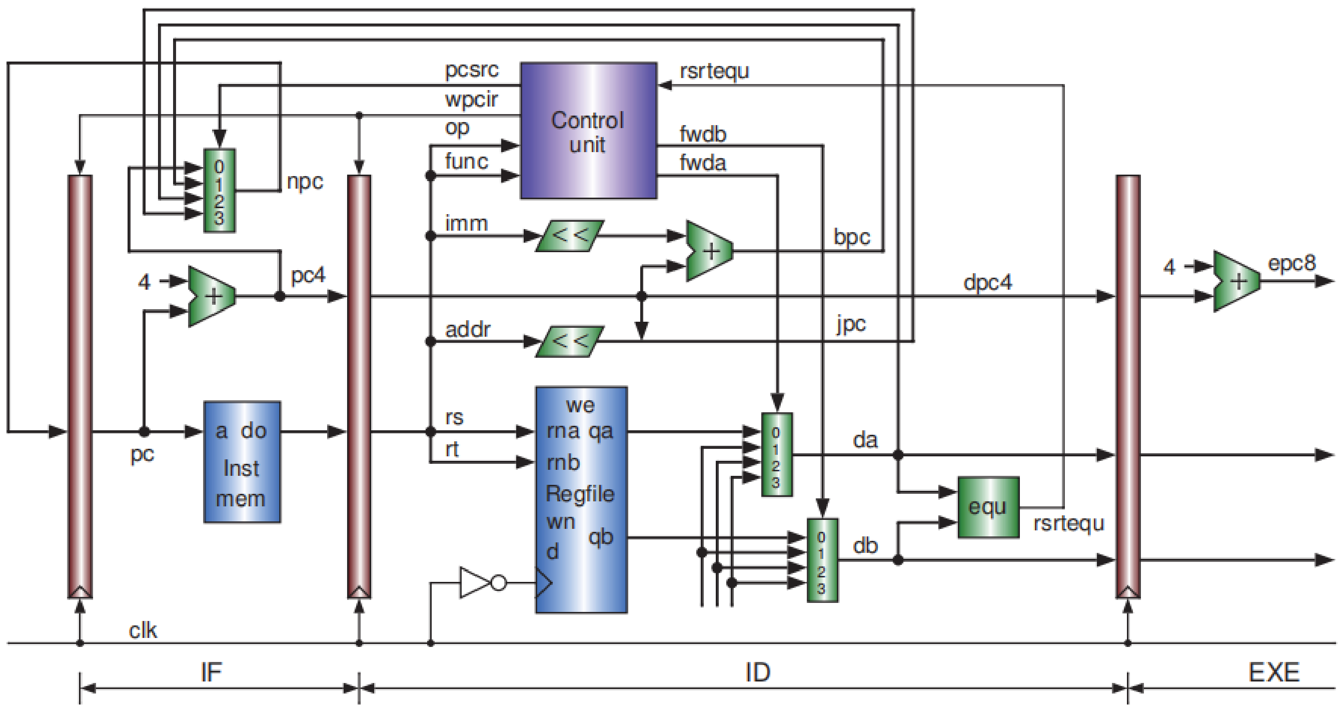


Figure 8.19 Implementation of delayed branch with one delay slot

8.3流水线化CPU示意图及实现

8.3.1流水化的CPU示意图

实现的流水化的CPU可以执行前几章所实现的20条指令格式的指令（注意jal指令的不同处理，采用分支延迟技术的流水化CPU是写PC+8到$31,采用取消执行的是写PC+4）

下图即为流水化的CPU的综合实现图：

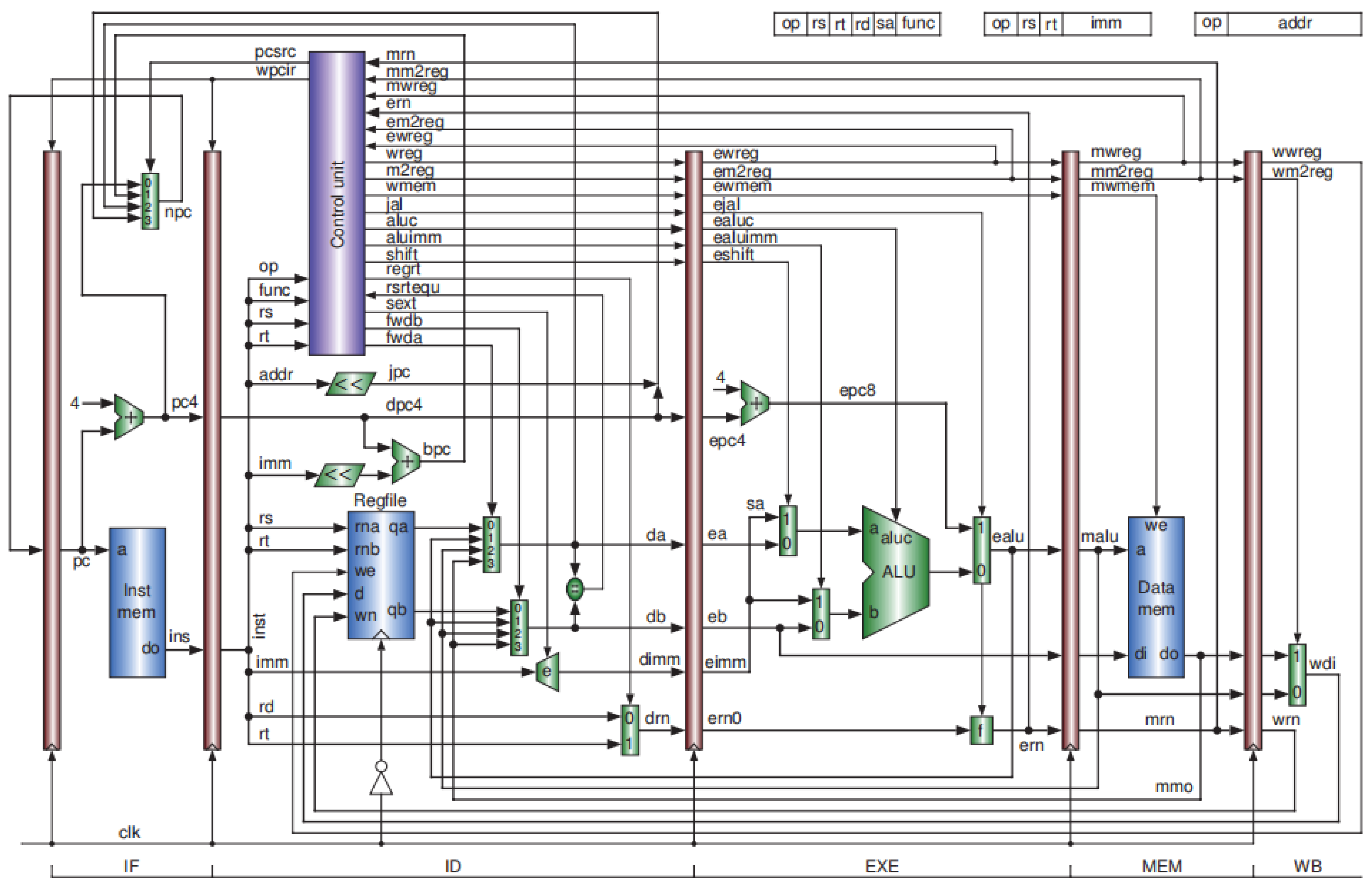


Figure 8.21 Detailed circuit of the pipelined CPU

8.3.2实现代码

数据通路

专用加法器、多选器、寄存器文件、均采用之前的实现

```
`timescale 1ns / 1ps

module regDesign #(parameter  WIDTH= 32)(
    input clk,enable,clrn,
    input [WIDTH-1:0]i_data,
    output reg[WIDTH-1:0]o_data
);
    always @(posedge clk or posedge clrn) begin
        if (clrn) begin
            o_data<=0;
        end else if (enable) begin
            o_data=i_data;
        end
    end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module datapath(
    input clk,clrn,

    //PC更新
    input wpcir,
    input [1:0]pcsrc,
    output [31:0]pc,
    input [31:0]inst,

    //IF/ID
    output [5:0]op,funct,
    output [4:0]rs,rt,
    input sext,regdst,
    input [1:0]forwardAD,forwardBD,
    output z,

    //ID/EX
    input eshift,ealusrc,ejal,
    input [3:0]ealuc,
    output [4:0]ewn,

    //EX/MEM
    output mwmem,
    output [31:0]aluoutM,memWriteData,
    input [31:0]memreadM,
    output [4:0]mwn,

    //MEM/WB
    input wm2reg,wwreg,

    output [31:0]wd,aluoutE
);

    //PC寄存器的更新
    wire [31:0]pcAdder4,bpc,jpc,rpc,npc;
    mux4to1 generate_npc(pcAdder4,bpc,rpc,jpc,pcsrc,npc);
    regDesign #(.WIDTH(32))pc_init(clk,wpcir,clrn,npc,pc);
    adder generate_pcAdder4(pc,32'd4,pcAdder4);

    //IF/ID
    wire [31:0]pcAdder4D,instrD;
    regDesign #(.WIDTH(64))if_id_inst(clk,wpcir,clrn,{pcAdder4,inst},{pcAdder4D,instrD});//当要先清0
    wire [4:0]rd;
    wire [15:0]imm=instrD[15:0];
    wire [31:0]imm_sext={{16{imm[15]&sext}},imm[15:0]};
    assign op=instrD[31:26];
    assign funct=instrD[5:0];//写的时候忘记标注了
    assign rs=instrD[25:21];
    assign rt=instrD[20:16];
    assign rd=instrD[15:11];
    adder generate_bpc(pcAdder4D,imm_sext<<2,bpc);
    assign jpc={pcAdder4D[31:28],instrD[25:0],2'b0};
    wire [31:0]qa,qb;
    wire [4:0]wn_temp,wwn;
    //用WB阶段的信号写回
    regfile regfile_init(~clk,wwreg,clrn,rs,rt,wwn,wd,qa,qb);
    mux2to1 generate_wn_temp(rd,rt,regdst,wn_temp);

    wire [31:0]A,B;
    mux4to1 generateA(qa,aluoutE,aluoutM,memreadM,forwardAD,A);
    mux4to1 generateB(qb,aluoutE,aluoutM,memreadM,forwardBD,B);
    assign rpc=A;
    assign z=~|(A^B);//判断0时, 刚开始写成了~(A^B)是对A^B所有位取反,需要先连续|得到的结果再取反~|

    //ID/EXE
    wire [31:0]pcAdder4E,eA,eB,eimmSext;
    wire [4:0]ewn_temp;
    regDesign
#(.WIDTH(133))id_ex_init(clk,1'b1,clrn,{pcAdder4D,A,B,imm_sext,wn_temp},{pcAdder4E,eA,eB,eimmSext,ew
n_temp});//位宽刚开始数错了, 也要清0
    wire [31:0]pcAdder8,aluA,aluB;
    adder generate_pcAdder8(pcAdder4E,32'd4,pcAdder8);
    mux2to1 generate_ALUa(eA,{27'b0,eimmSext[10:6]},eshift,aluA);
    mux2to1 generate_ALUb(eB,eimmSext,ealusrc,aluB);
    wire zero,overflow;
    wire [31:0]alures;
    alu alu_init(aluA,aluB,ealuc,alures,zero,overflow);
    mux2to1 generate_aluout(alures,pcAdder8,ejal,aluoutE);
    assign ewn=ewn_temp|{5{ejal}};
    //EX/MEM
    regDesign #(.WIDTH(69))
ex_mem_init(clk,1'b1,clrn,{aluoutE,eB,ewn},{aluoutM,memWriteData,mwn});//也要清0
    //MEM/WB
    wire [31:0]aluoutW,memreadW;
    regDesign
```

```
#(.WIDTH(69))mem_wb_init(clk,1'b1,clrn,{aluoutM,memreadM,mwn},{aluoutW,memreadW,wwn});//也要清0
    mux2to1 generate_wd(aluoutW,memreadW,wm2reg,wd);

endmodule
```
Verilog

控制单元

```
`timescale 1ns / 1ps

module harzard(
    input is_rs,is_rt,
    input ewreg,em2reg,mwreg,mm2reg,
    input [4:0]rs,rt,ewn,mwn,

    output reg[1:0]forwardAD,forwardBD,
    output stall
);
    assign stall=ewreg&em2reg&(ewn!=5'b0)&(is_rs&(rs==ewn)|is_rt&(rt==ewn));
    always @(*) begin
        forwardAD=2'b00;
        if (ewreg&(ewn!=5'b0)&(ewn==rs)) begin
            forwardAD=2'b01;
        end else begin
            if (mwreg&~mm2reg&(mwn!=5'b0)&(mwn==rs)) begin
                forwardAD=2'b10;
            end else if (mwreg&mm2reg&(mwn!=5'b0)&(mwn==rs)) begin
                forwardAD=2'b11;
            end
        end
    end
    always @(*) begin
```

```verilog
        forwardBD=2'b00;
        if (ewreg&(ewn!=5'b0)&(ewn==rt)) begin
            forwardBD=2'b01;
        end else begin
            if (mwreg&~mm2reg&(mwn!=5'b0)&(mwn==rt)) begin
                forwardBD=2'b10;
            end else if (mwreg&mm2reg&(mwn!=5'b0)&(mwn==rt)) begin
                forwardBD=2'b11;
            end
        end
    end
endmodule
```
Verilog

```verilog
`timescale 1ns/1ps

module cu(
    input clk,clrn,
    input [5:0]funct,op,
    input [4:0]rs,rt,ewn,mwn,
    input z,

    //IF
    output  wpcir,
    output [1:0]pcsrc,

    //ID
    output sext,regdst,
    output [1:0]forwardAD,forwardBD,

    //EX
    output ejal,eshift,ealusrc,
    output [3:0]ealuc,

    //MEM
    output mwmem,

    //WB
    output wwreg,wm2reg

);
    wire rtype=~op[5]&~op[4]&~op[3]&~op[2]&~op[1]&~op[0];//op 000000
    wire i_add=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 100000
    wire i_sub=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0 funct 100010
    wire i_and=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&~funct[0];//op 0 funct 100100
    wire i_or=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&funct[0];//op 0 funct 100101
    wire i_xor=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&~funct[0];//op 0 funct 100110
    wire i_sll=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 000000
    wire i_srl=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0  funct 000010
    wire i_sra=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&funct[0];//op 0 funct 000011
    wire i_jr=rtype&~funct[5]&~funct[4]&funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 001000
    wire i_addi=~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];//op 001000
    wire i_andi=~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];//op 001100
    wire i_ori=~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];//op 001101
    wire i_xori=~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];//op 001110
    wire i_lw=op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 100011
    wire i_sw=op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];////op 101011
    wire i_beq=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];//op 000100
    wire i_bne=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];//op 000101
    wire i_lui=~op[5]&~op[4]&op[3]&op[2]&op[1]&op[0];//op 001111
    wire i_j=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];//op 000010
    wire i_jal=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 000011


    wire stall;
    assign wpcir=~stall;
    assign regdst=i_addi|i_andi|i_ori|i_xori|i_lw|i_lui;
    assign sext=i_addi|i_lw|i_sw|i_beq|i_bne;
    assign pcsrc[1]=i_j|i_jal|i_jr;
    assign pcsrc[0]=i_beq&z|i_bne&~z|i_j|i_jal;

    wire
wreg=(i_add|i_sub|i_add|i_or|i_xor|i_sll|i_srl|i_sra|i_addi|i_andi|i_ori|i_xori|i_lw|i_lui|i_jal)&~s
tall;
    wire jal=i_jal;
    wire m2reg=i_lw;
    wire shift=i_sll|i_srl|i_sra;
    wire alusrc=i_addi|i_andi|i_ori|i_xori|i_lw|i_sw+i_lui;
    wire [3:0]aluc;
    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;
    wire wmem=i_sw&~stall;

    //ID/EX
    wire ewreg,em2reg,ewmem;
    regDesign #(.WIDTH(10))
id_ex_cu(clk,1,0,{wreg,jal,m2reg,shift,alusrc,aluc,wmem},{ewreg,ejal,em2reg,eshift,ealusrc,ealuc,ewm
em});
    //EX/MEM
    wire mwreg,mm2reg;
    regDesign #(.WIDTH(3)) ex_mem_cu(clk,1,0,{ewreg,em2reg,ewmem},{mwreg,mm2reg,mwmem});
    //MEM/WB
    regDesign #(.WIDTH(2)) mem_wb_cu(clk,1,0,{mwreg,mm2reg},{wwreg,wm2reg});

    wire i_rs=i_add|i_sub|i_and|i_or|i_xor|i_jr|i_addi|i_andi|i_ori|i_xori|i_beq|i_bne|i_lw|i_sw;
    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;
    harzard
hazard_init(i_rs,i_rt,ewreg,em2reg,mwreg,mm2reg,rs,rt,ewn,mwn,forwardAD,forwardBD,stall);
endmodule
```

Verilog

存储器

```
module p1_inst_mem (
    input [31:0]a,// rom address
    output [31:0]inst// rom content = rom[a]
); // instruction memory, rom

    wire [31:0] rom [0:63]; // rom cells: 64 words * 32 bits
    // rom[word_addr] = instruction // (pc) label instruction
    assign rom[6'h00] = 32'h3c010000; // (00) main: lui $1, 0
    assign rom[6'h01] = 32'h34240050; // (04) ori $4, $1, 80
    assign rom[6'h02] = 32'h0c00001b; // (08) call: jal sum
    assign rom[6'h03] = 32'h20050004; // (0c) dslot1: addi $5, $0, 4
    assign rom[6'h04] = 32'hac820000; // (10) return: sw $2, 0($4)
    assign rom[6'h05] = 32'h8c890000; // (14) lw $9, 0($4)
    assign rom[6'h06] = 32'h01244022; // (18) sub $8, $9, $4
    assign rom[6'h07] = 32'h20050003; // (1c) addi $5, $0, 3
    assign rom[6'h08] = 32'h20a5ffff; // (20) loop2: addi $5, $5, -1
    assign rom[6'h09] = 32'h34a8ffff; // (24) ori $8, $5, 0xffff
    assign rom[6'h0a] = 32'h39085555; // (28) xori $8, $8, 0x5555
    assign rom[6'h0b] = 32'h2009ffff; // (2c) addi $9, $0, -1
    assign rom[6'h0c] = 32'h312affff; // (30) andi $10,$9,0xffff
    assign rom[6'h0d] = 32'h01493025; // (34) or $6, $10, $9
    assign rom[6'h0e] = 32'h01494026; // (38) xor $8, $10, $9
    assign rom[6'h0f] = 32'h01463824; // (3c) and $7, $10, $6
    assign rom[6'h10] = 32'h10a00003; // (40) beq $5, $0, shift
    assign rom[6'h11] = 32'h00000000; // (44) dslot2: nop
    assign rom[6'h12] = 32'h08000008; // (48) j loop2
```

```verilog
    assign rom[6'h13] = 32'h00000000; // (4c) dslot3: nop
    assign rom[6'h14] = 32'h2005ffff; // (50) shift: addi $5, $0, -1
    assign rom[6'h15] = 32'h000543c0; // (54) sll $8, $5, 15
    assign rom[6'h16] = 32'h00084400; // (58) sll $8, $8, 16
    assign rom[6'h17] = 32'h00084403; // (5c) sra $8, $8, 16
    assign rom[6'h18] = 32'h000843c2; // (60) srl $8, $8, 15
    assign rom[6'h19] = 32'h08000019; // (64) finish: j finish
    assign rom[6'h1a] = 32'h00000000; // (68) dslot4: nop
    assign rom[6'h1b] = 32'h00004020; // (6c) sum: add $8, $0, $0
    assign rom[6'h1c] = 32'h8c890000; // (70) loop: lw $9, 0($4)
    assign rom[6'h1d] = 32'h01094020; // (74) stall: add $8, $8, $9
    assign rom[6'h1e] = 32'h20a5ffff; // (78) addi $5, $5, -1
    assign rom[6'h1f] = 32'h14a0fffc; // (7c) bne $5, $0, loop
    assign rom[6'h20] = 32'h20840004; // (80) dslot5: addi $4, $4, 4
    assign rom[6'h21] = 32'h03e00008; // (84) jr $31
    assign rom[6'h22] = 32'h00081000; // (88) dslot6: sll $2, $8, 0
    assign inst = rom[a[7:2]]; // use 6-bit word address to read rom
endmodule
```
Verilog

```verilog
`timescale 1ns / 1ps

module p1_data_mem(
    input clk,we,
    input [31:0]addr,datain,
    output [31:0]dataout
);
    reg [31:0] ram [0:31]; // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]]; // use 5-bit word address
    always @ (posedge clk) begin
        if (we)
            ram[addr[6:2]] = datain; // write ram
    end
    integer i;
    initial begin // ram initialization
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data // (byte_addr) item in data array
        ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + a3 = a3
        ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca
        ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143
        ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```
Verilog

8.4流水化CPU中的中断/异常机制

流水化的CPU的中断/异常机制较SC、MC CPU更为复杂，因为在SC、MC CPU中一个时钟周期仅仅只有一条指令处于执行状态，在指令执行完成后响应中断控制的时间点较为容易去判断，但是在流水化的CPU中一个时钟周期存在多条指令执行，并不能找到一个所有指令都执行完毕的时间点；而且流水化的CPU还采用了分支延迟技术，使得中断、异常处理更为繁琐

如果一个流水化的CPU可以以SC CPU的方式完美地处理中断，那么这个流水化的CPU就具备精确的异常/中断机制

> *If a pipelined CPU can deal with interrupts in the same beautiful way as an SC CPU does, this pipelined CPU is said to have a precise interrupt mechanism*

更具体一点，当中断发生时，流水化的CPU中当前ID阶段和它的先辈指令均能正确的安全的执行完毕，但是它之后的指令必须被停止，中断返回后，继续从第一条被停止的指令开始执行

> *More specifically, when an interrupt occurs, the instruction that is staying at the ID stage and its predecessors must be executed safely, and its follow-up instructions must be canceled. After the interrupt request is serviced, the control is returned to the place of the first canceled instruction: the program then continues as if nothing happened.*

8.4.1中断和异常的类型以及相关的寄存器

同C6的假设，我们所实现的流水化CPU仅仅处理3种异常和一个外部中断

Table 8.1 Stage where exception or interrupt may occur

ExcCode	Mnemonic	Mask	Description	Occur in
0	Int	IM[0]	External interrupt	Any stage
1	Sys	IM[1]	System call	ID stage
2	Unimpl	IM[2]	Unimplemented instruction	ID stage
3	Ov	IM[3]	Arithmetic overflow	EXE stage

使用CP0的12号Cause寄存器去寻找服务程序的入口地址

Cause寄存器的Excode反映中断/异常源，当中断响应时，由硬件写Excode，汇编指令读取Excode，并结合入口地址表基址得到跳转地址。Cause寄存器的BD位用于后面算术溢出指令是位于延迟槽还是普通情况

使用CP0的13号Status寄存器禁止中断的响应

Status寄存器的IM字段存储是否能够响应对应位的中断，S字段用于保存响应中断/异常时的IM字段

IM[0]➝int, IM[1]➝syscall, IM[2]➝Unimplement, IM[3]➝overflow

当响应中断/异常时，Status左移4位，将IM保存在S并禁止多层嵌套；

当返回时，Status右移4位，恢复IM

使用CP0的14号EPC寄存器存储中断返回地址，进行中断返回

当中断发生时，如果非分支、跳转指令，那么IF阶段的指令地址送至EPC；如果是分支、跳转指令，那么ID阶段的指令地址送EPC；

当异常发生时，当前ID阶段的地址送至EPC

> *When an interrupt occurs, the address of the instruction in the IF stage will be saved into the EPC (the instruction in the ID stage will be executed). However, if the instruction in the ID stage is a branch or a jump instruction when an interrupt occurs, the branch or jump instruction should be canceled and its address saved in the EPC.* When an instruction generates anexception, the address of the instruction is saved into the EPC so that the instruction is capable of being executed again. If there is no need to execute it again, the return address must be updated by adding a 4 to the EPC before executing the eret instruction.

Note that the eret instruction itself has no delay slot

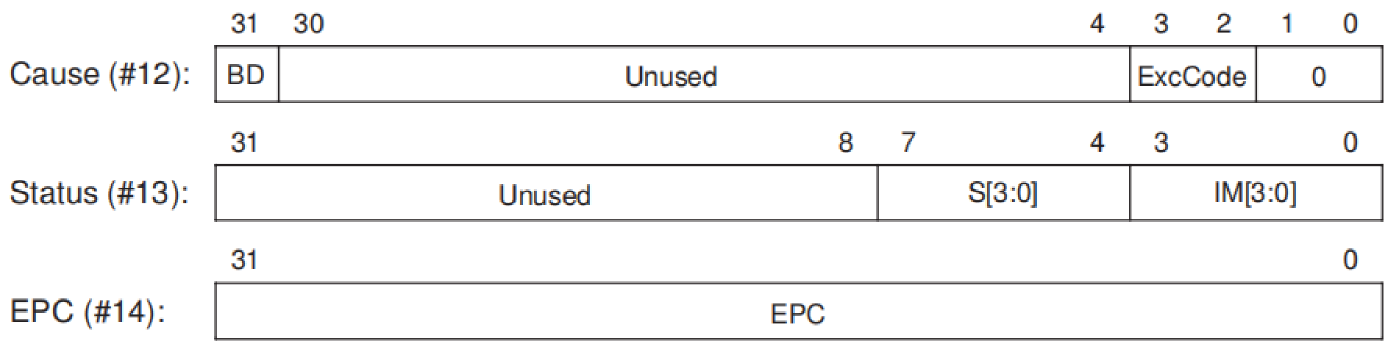


Figure 8.25 Three registers for implementing precise interrupt

📌虽然中断请求可以发生在任何时期，但是中断响应均需要经过ID阶段的CU处理，因此可以将中断的发生时间分为以下三种情况：

1. When an interrupt occurs, the instruction in the ID stage is a jump or a branch;
2. When an interrupt occurs, the ID stage is in a delay slot;
3. Other cases (ordinary cases) than cases 1 and 2.

8.4.1.1中断时位于ID阶段的指令是普通情况

当中断时位于ID阶段的指令是普通情况，ID阶段的指令可以继续执行，但是IF阶段的指令需要被取消，根据发生的中断/异常类型写Exccode，IF阶段的指令地址写EPC，中断服务程序的入口地址写PC，Status寄存器左移4位

取消IF指令的流程如下：ID阶段产生取消信号cancel，保存至ID/EXE流水线寄存器中，在下一个时钟周期，当IF阶段的指令进入ID阶段后[注释3]，使用ecancel信号取消它的执行➞**下一个时钟周期写信号为0**

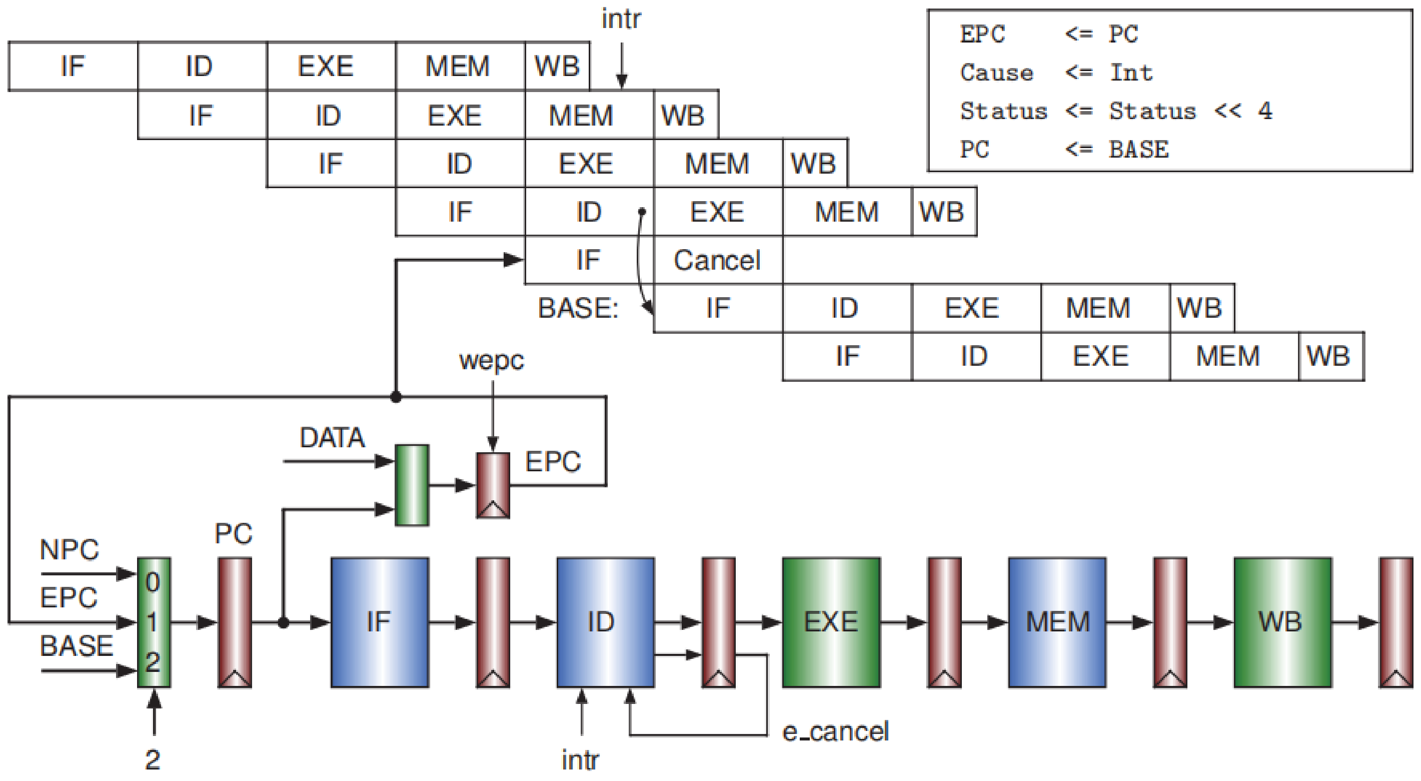


Figure 8.26 Interrupt occurring in an ordinary case

8.4.1.2中断时位于ID阶段的指令是分支/跳转指令

当中断时位于ID阶段的指令是分支/跳转指令时，ID阶段的跳转/分支指令和IF阶段取出的延迟槽指令均被取消，然后ID阶段的PC写入EPC，Status左移4位，根据异常/中断类型写exccode从而得到服务程序的入口地址送PC

ID阶段产生的控制信号会立刻取消当前指令，然后传送至ID/EXE的ecancel会在下一个时钟周期取消IF阶段的指令

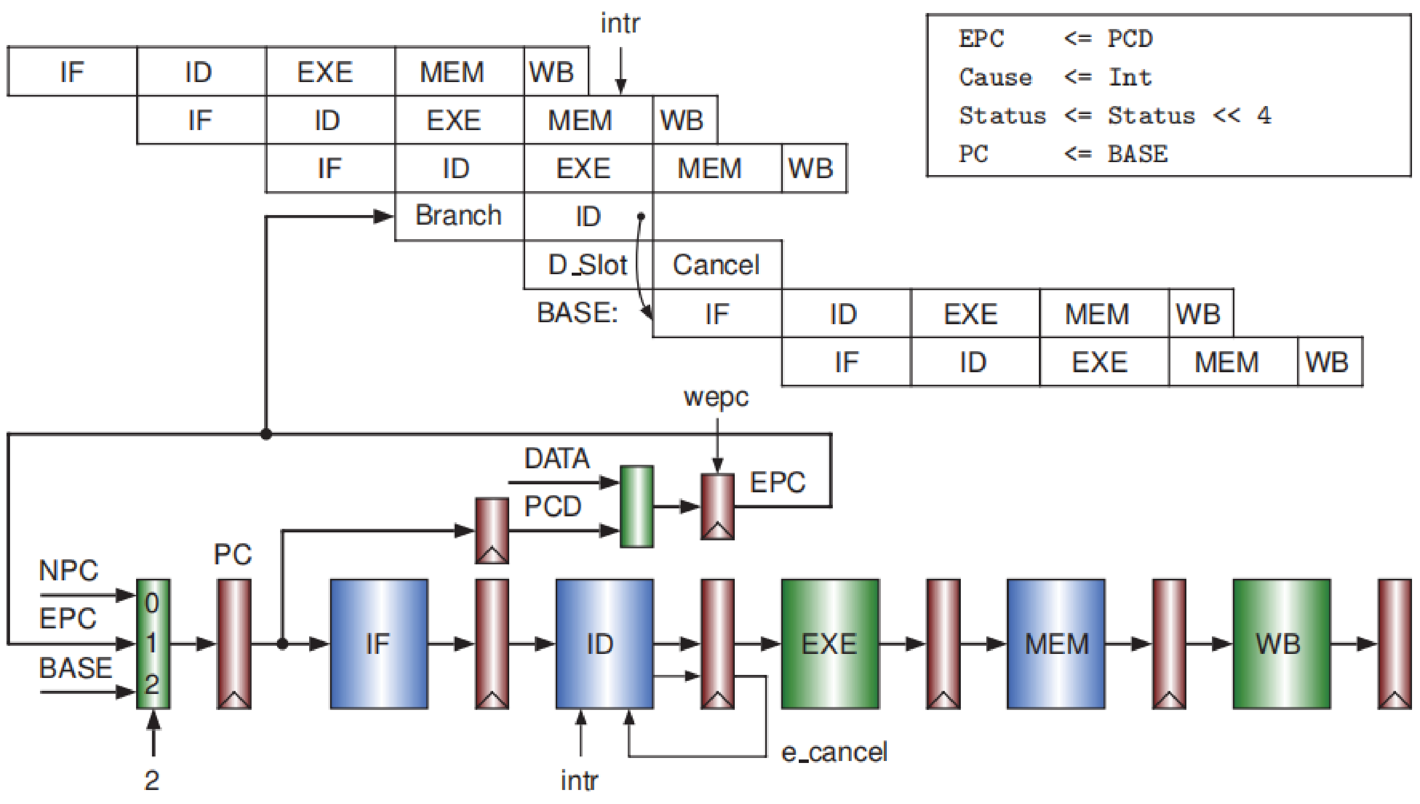


Figure 8.27 Interrupt occurring when decoding the branch instruction

8.4.1.3中断时位于ID阶段的指令是延迟槽指令

当中断时位于ID阶段的指令是延迟槽指令，ID阶段的指令会继续执行，IF阶段的指令会被取消，因此将IF阶段的PC写至EPC，根据中断/异常类型写Exccode且BD位置1，从而得到服务程序的入口地址并将Status左移4位

ID阶段产生的cancel会传送至ID/EXE寄存器以取消IF阶段的指令

📌当中断时位于ID阶段的指令是延迟槽指令的处理方式与是普通情况指令的处理方式基本一致，唯一的区别是前者写Cause寄存器的BD位为1（基本对于中断BD位是无用的）

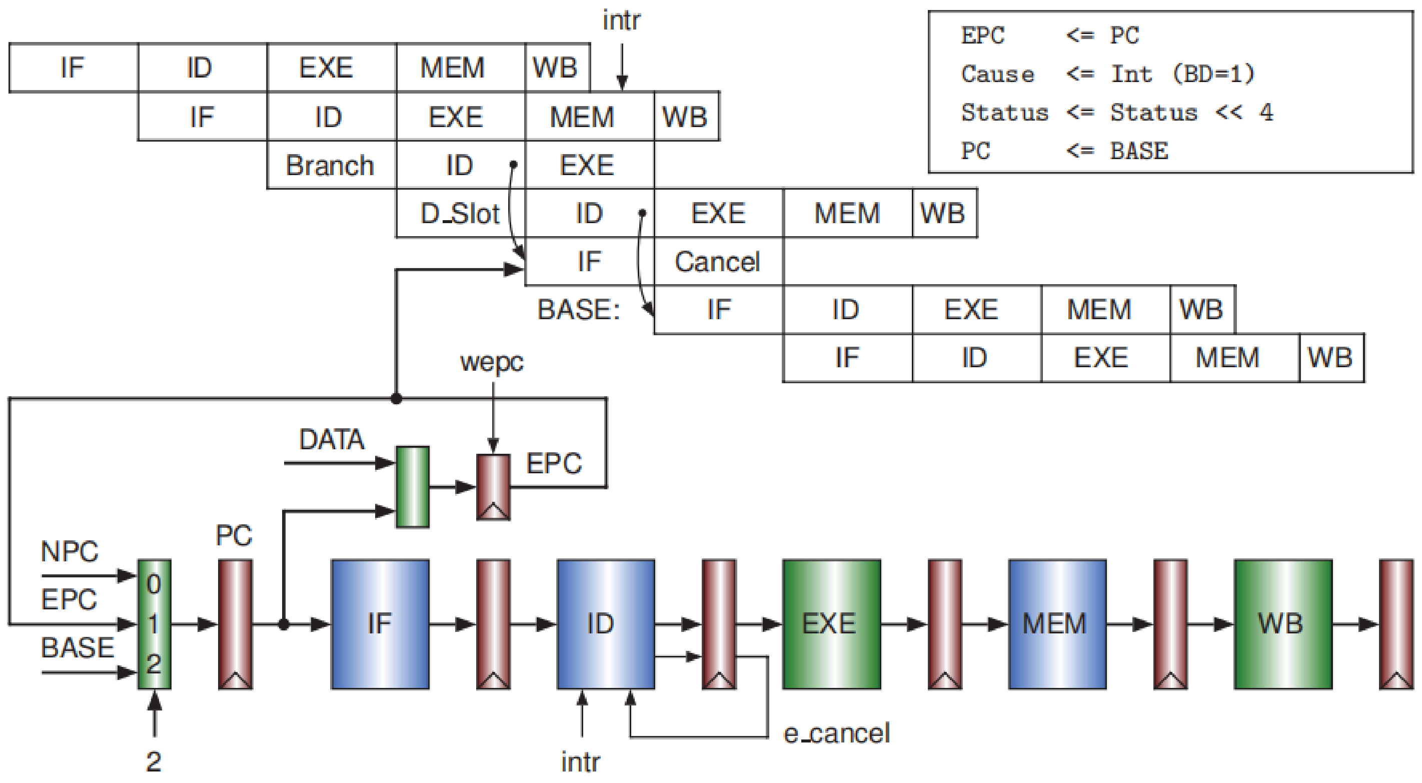


Figure 8.28 Interrupt occurring in the ID stage of delay slot

8.4.2流水线化的CPU中异常的处理机制

8.4.2.1系统调用异常

当CPU执行系统调用指令[注释4]时发生系统调用异常

当译码指令时检测到该条指令是一个系统调用指令，则在其未被屏蔽的前提下，会触发系统调用异常。首先会根据异常类型设置Exccode字段，并将Status左移4位，将当前ID阶段的PC写入EPC，然后根据Exccode查询中断服务程序的入口地址送至PC。ID阶段生成的cancel信号需要保存至ID/EXE寄存器用于停止systemcall的下一条指令

系统调用异常是将译码阶段也就是系统调用指令所在的PC送至EPC，异常返回时为了不继续执行系统调用指令陷入死循环，需要将EPC+4→EPC，相关的汇编代码如下：

```
epc_plus4:
  mfc0 $k0, c0_epc # load the content of epc to register 26
  addiu $k0, $k0, 4 # increment by four
  mtc0 $k0, c0_epc # save it back to epc
  eret # pc <-- epc (return from the exception handler)
  nop # will be canceled
```

```
WebAssembly
```

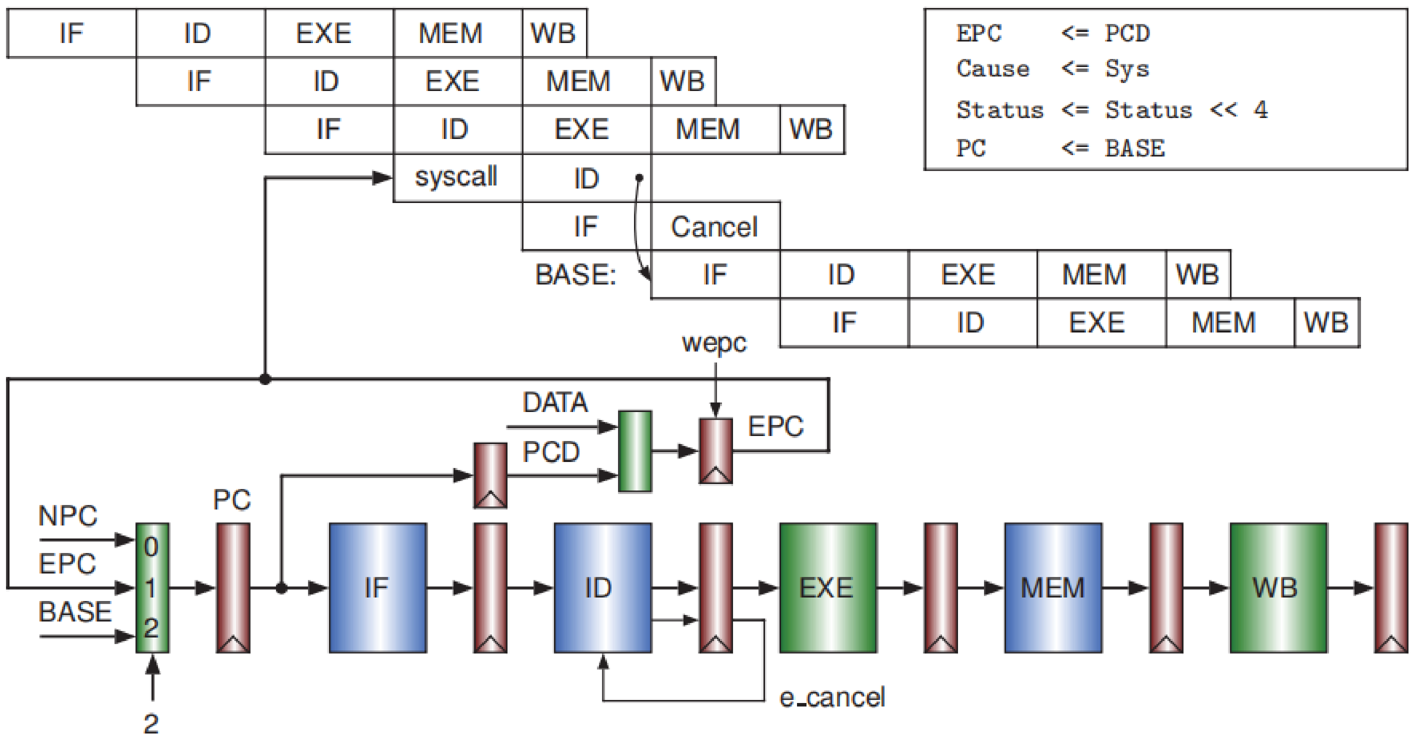


Figure 8.29 Execution of system call instruction

📌 在我们的实现中，是不允许将系统调用指令放在延迟槽中的

8.4.2.2未实现的指令的异常

当CPU执行一条合法的在ISA内规定了的指令，但CPU硬件没有实现该指令时，未实现的指令就会出现异常。此异常不同于未定义指令的异常，未定义的指令不是ISA中的合法指令

处理未实现的指令的异常的方法同处理系统调用指令异常：首先根据异常类型写Exccode，写译码阶段的PC至EPC，并将Status左移4位，停止IF阶段指令的执行，并将Exccode所寻址到的服务程序的地址送至PC

📌 在我们的实现中，是不允许未实现的指令放在延迟槽中的

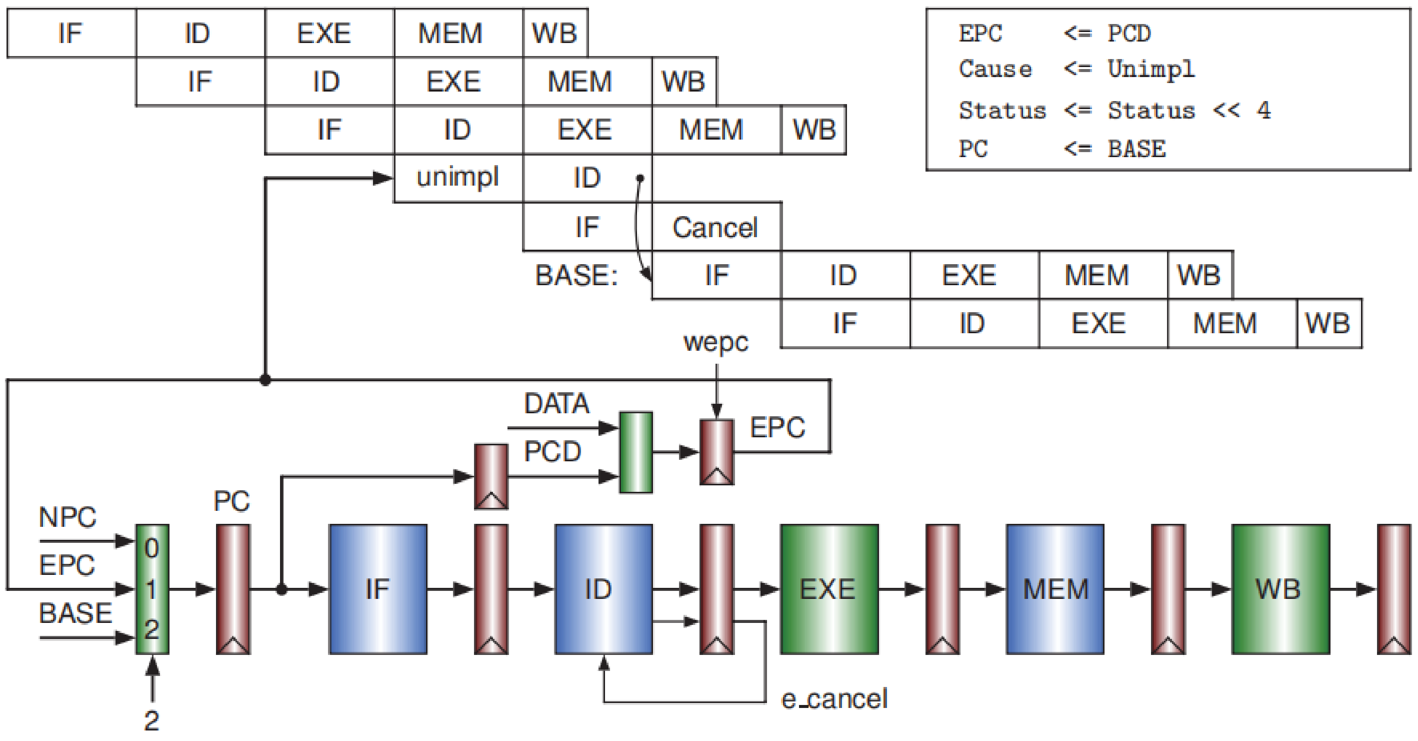
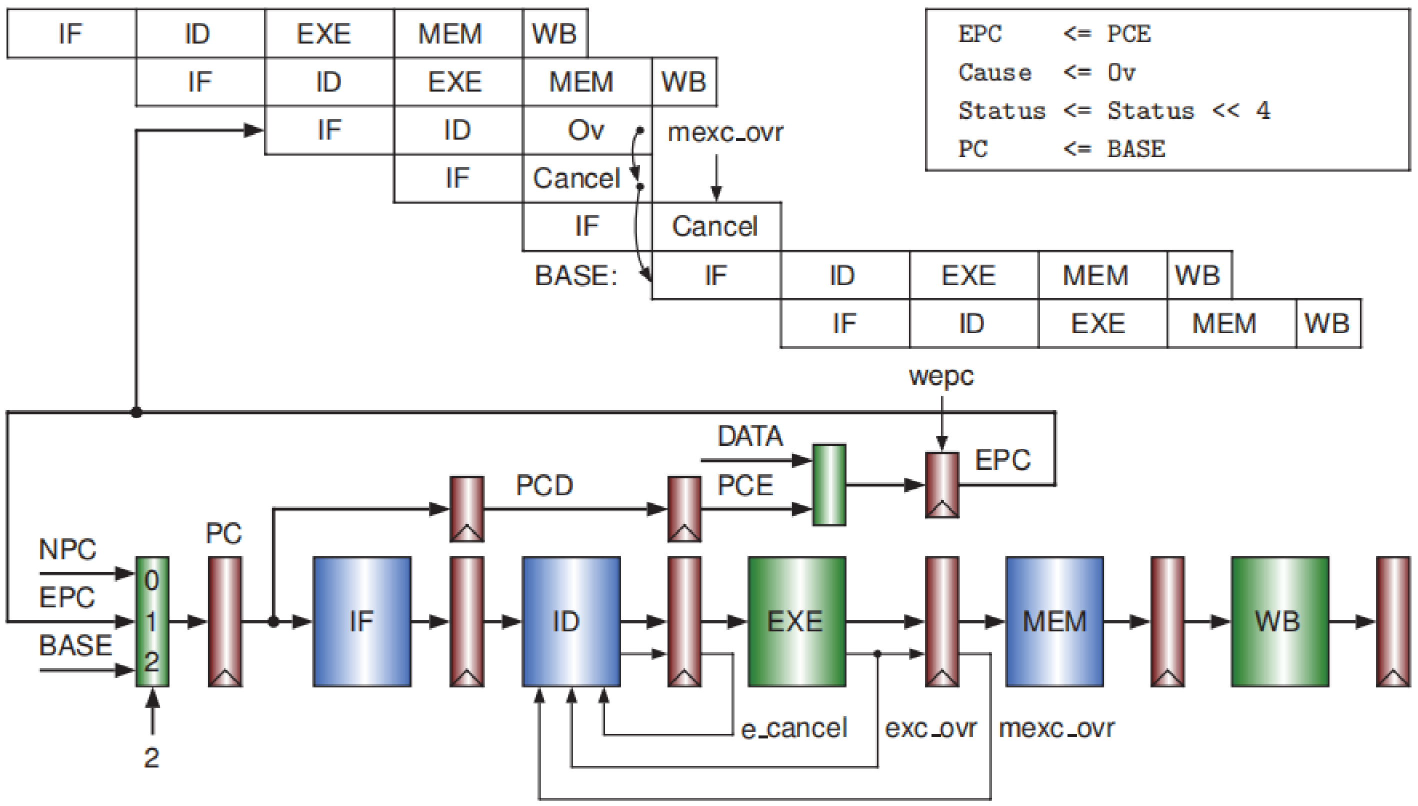


Figure 8.30 Unimplemented instruction exception

8.4.2.3算术溢出异常

之前所描述的异常、中断的检查都是在ID阶段的，但是算术溢出异常是发生在EXE阶段的，但是也在ID段检查是否允许，算术溢出异常的发生分为两个情况：普通情况和指令在延迟槽中的情况

普通情况下的算术溢出



上图即为普通情况下算术溢出的处理方式，产生算术溢出异常的指令的PC需要送至EPC，同时根据异常的类型写Exccode，Status左移4位，并根据Exccode查找服务程序的入口地址写PC。算术溢出指令以及它后面的两条指令均需要被取消

EXE阶段如果有溢出，则出现exc_ovr信号，该信号传送至ID阶段生成cancel信号

之后的两条指令的取消，可以使用exc_ovr和mexc_ovr传至对应的ID阶段传至控制单元内部，前者取消异常指令的下一条指令，后者取消异常指令的下两条指令。当前指令的取消可以使用exc_ovr置0；

延迟槽指令的算术溢出

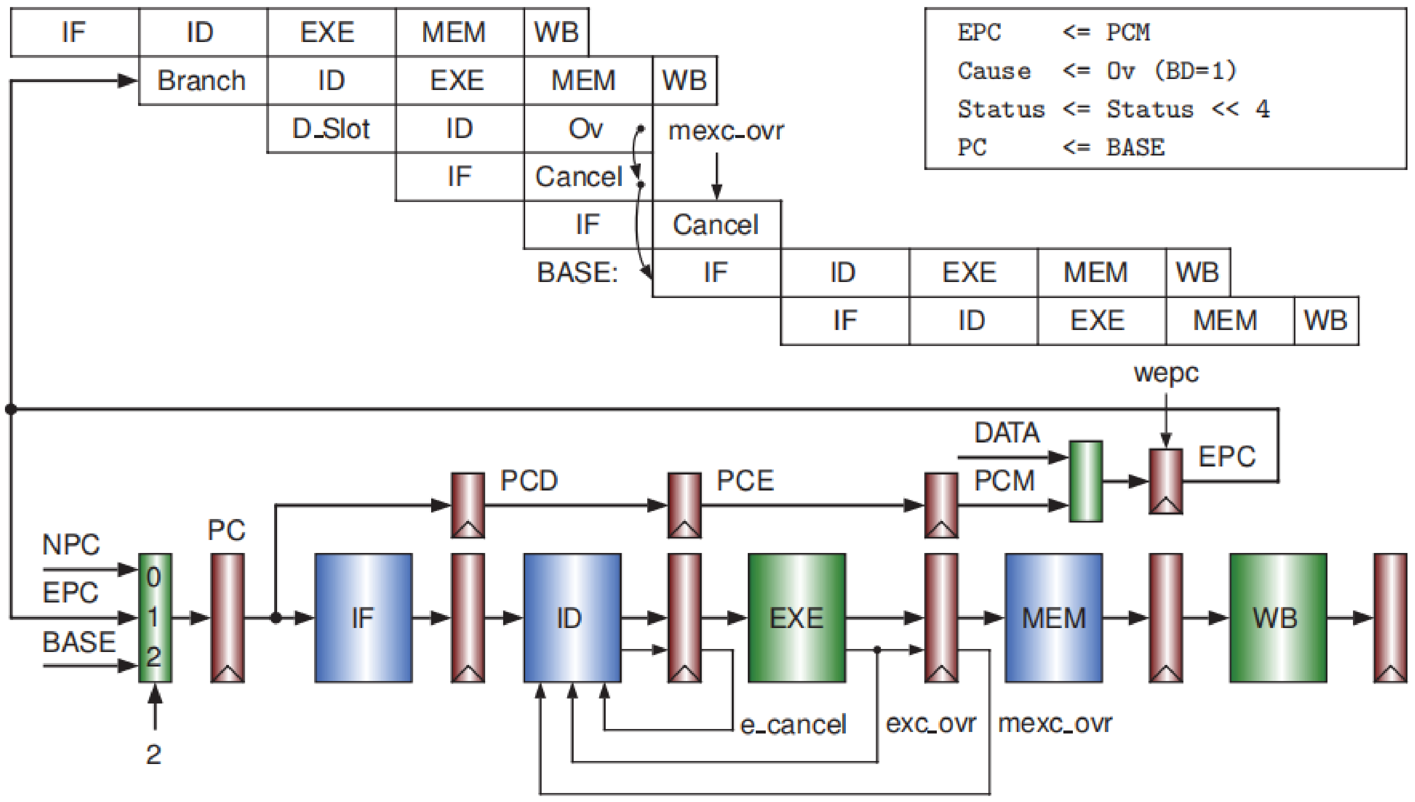


Figure 8.32 Overflow exception occurring in the delay slot

当延迟槽中的指令出现算术溢出时，必须将延迟槽的上一条指令——分支指令的地址[注释5]写EPC，即PCM➞EPC，然后正常的异常响应处理：写exccode、移位Status、服务地址写PC，并将BD置1

异常处理程序读EPC和Cause寄存器的BD位判断哪条指令造成了算术溢出：

BD为1时，表示当前EPC地址的下一条地址上的指令产生了算术溢出

BD为0时，表示当前EPC地址的指令产生了算术溢出

但是，没有办法恢复程序的执行，因为一旦恢复执行，溢出异常将再次发生。通常，当这种异常发生时，异常处理程序会根据EPC或EPC + 4所指向的指令显示一些消息，并停止程序

📌 因为异常、中断可能会在ID阶段产生也可能会在EXE阶段产生，所以会有同时产生的情况，此时需要根据优先级处理：exccode值越大优先级越高

📌 异常/中断处理的难点主要是：**保存什么地址去EPC；怎么取消后续指令的执行；**

Table 8.2 Writing EPC on exception or interrupt

Mnemonic	Branch/jump in ID stage	Occurs in delay slot	Others
`Int`	EPC ⇐ PCD	EPC ⇐ PC	EPC ⇐ PC
`Sys`	Impossible	Not allowed	EPC ⇐ PCD
`Unimpl`	Impossible	Not allowed	EPC ⇐ PCD
`Ov`	Not occur	EPC ⇐ PCM	EPC ⇐ PCE

8.5带有精确的异常/中断机制的流水化CPU

8.5.1 CPU的结构

CPU中的CU除了译码实现之前的20条基本指令外，也需要对“mfc0、mtc0、eret、syscall”指令进行译码实现(mfc0指令也需要使regdst、wreg有效)；mfc0、mtc0操作的CP0的寄存器**Status、EPC、Cause寄存器放在ID段和EXE段之间**➞使得mfc0的执行周期为五个周期——WB阶段时将取得的数据写至rt寄存器，但mtc0只需要两个周期——ID阶段将rt寄存器的数据写至CP0的某个寄存器中，**既然mtc0要读rt那么也需要使用到前递技术**

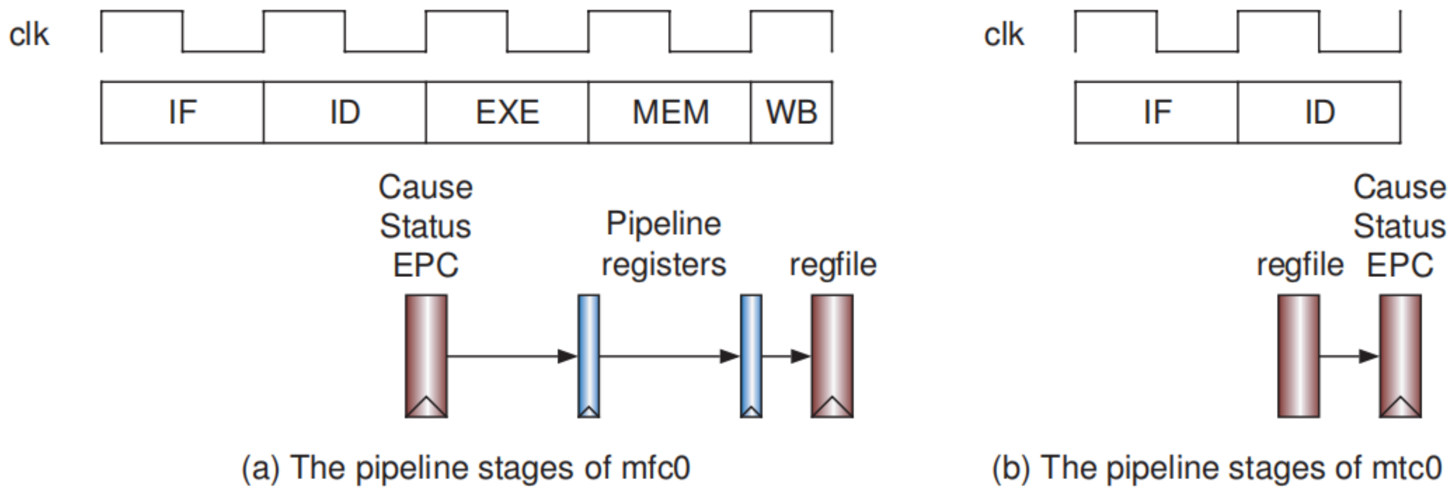


Figure 8.34 Pipeline stages of `mfc0` and `mtc0` instructions

下图是在流水化CPU的基础上增加一些部件所实现的带有精确中断/异常机制的流水化CPU

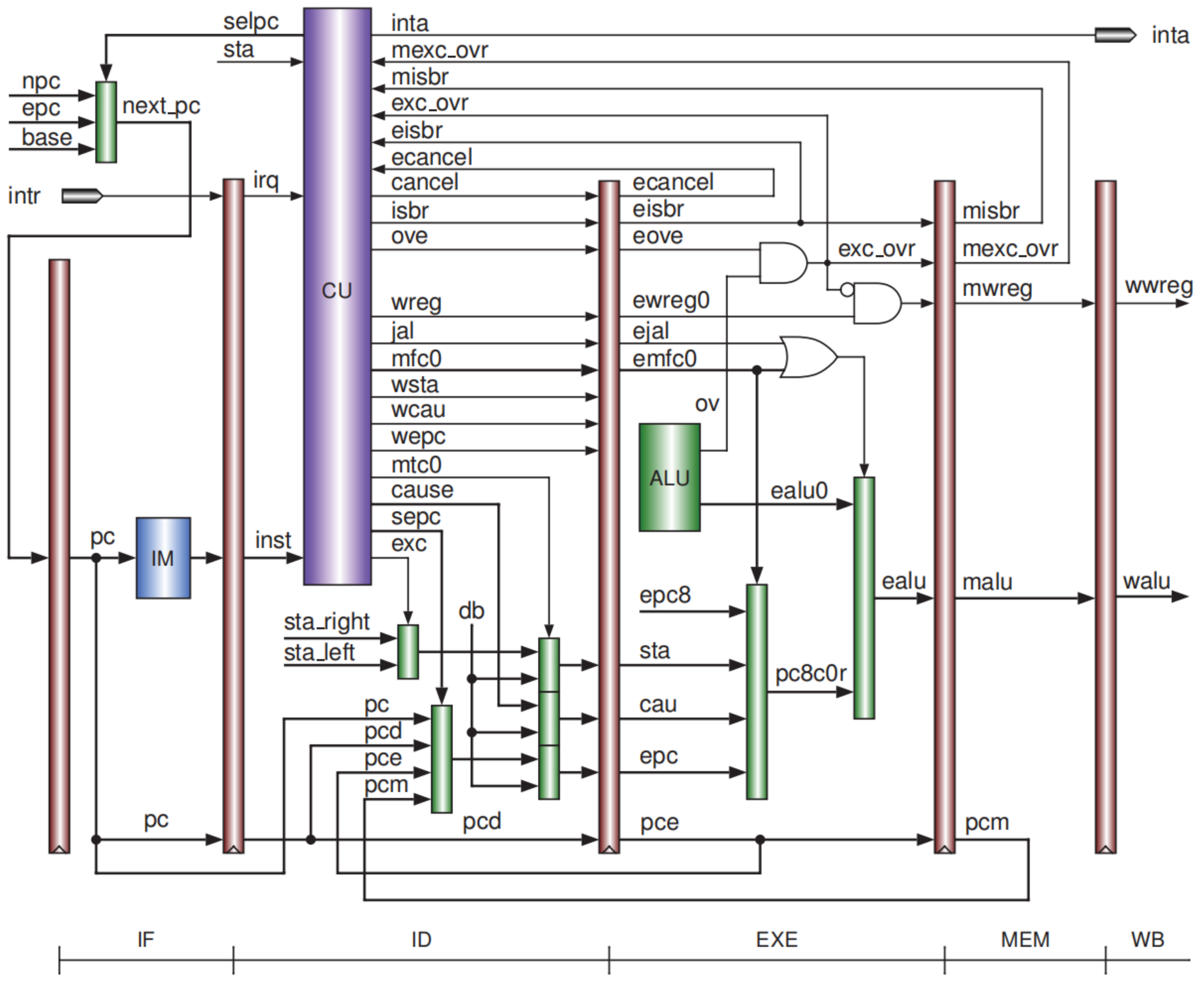


Figure 8.36 The circuit of pipelined CPU with precise interrupt

从左往右看，相较于不具有中断的情况：

1. 多了一个多路选择器，需要在npc、epc、base之间根据选择信号selpc生成next_pc（**eret写epc、exc写base**）
2. 外部输入intr，经IF/ID流水线寄存器传递产生irq输入CU
3. 多了CP0协处理器的Cause、Status、EPC三个寄存器的读写

   写：Cause寄存器**可以用mtc0写数据，也可以在中断/异常响应时，写服务程序表的基址**

   Status寄存器**可以用mtc0写数据，也可以在中断响应时写Status状态字左移4位的值，也可以在中断返回时写Status状态字右移4位的值**

EPC寄存器根据中断的分类：**“普通情况、延迟槽写PC，分支跳转写PCD”，根据异常的种类“Syscall写PCD、Unimplemented写PCD、发生在普通情况下的overflow写PCE，发生在延迟槽中的overflow写PCM”**

写的二级选择统一采用mtc0信号控制，Status寄存器的一级写数据选择采用exc控制信号，EPC寄存器的一级写则结合下表写sepc控制信号

```
// sel epc:     id is_branch    eis_branch    mis_branch     others
// exc_int      PCD (01)        PC  (00)      PC  (00)       PC  (00)
// exc_sys      x               x             PCD (01)       PCD (01)
// exc_uni      x               x             PCD (01)       PCD (01)
// exc_ovr      x               x             PCM (11)       PCE (10)
```

可得知CU内部需要is_br信号的传递

📌写操作的执行还需要写信号wcau、wsta、wepc，所以需要判断mtc0所操作的rd字段是表示哪个寄存器——12是cause、13是status、14是epc

wcau=产生响应/mfc0指令有效且写12号寄存器

wsta=产生响应/返回指令有效/mfc0有效且写13号寄存器

wepc=产生响应/mfc0指令有效且写14号寄存器

读：读是利用mfc0指令，首先也需要选读哪个数据写寄存器——因为在EXE阶段写寄存器的数据可能是aluout也可能是流水中的jal写pc+8，这两个是结合ejal控制信号选择的，这里再加上sta、cau和epc，那么就可以设置emfc0信号在“PC+8、sta、cause、epc”中四选一，然后jal和emfc0再或选择四选一的结果和aluout

📌当读sta时选中01，当读cause时选中10，当读epc时选中11

那么结合rd字段以及mfc0指令共同产生mfc0[1:0]的控制信号

4. 执行阶段溢出异常的产生

   译码阶段需要输出是否允许溢出异常响应的ove信号，该信号传送至EX阶段产生eove信号，若ALU计算出溢出使得overflow为1，那么overflow与eove的相与结果即作为产生的eov_exc异常，该eov_exc信号用于清除当前EX阶段执行的指令以及ID阶段执行的指令，传送至MEM阶段产生的meov_exc用于清除IF阶段的指令

5. cancel信号只有在当前中断发生在跳转/分支指令时还需要停止当前在ID阶段的指令，其余都是传送到EX阶段，用ecancel信号停止下一条指令

   cancel信号的产生是要么有中断/异常要么是eret指令，因为eret指令没有延迟槽，那么eret的下一条指令也需要被取消

📌综合cancel、ecancel、stall、eov_exc、mov_exc信号，对指令的取消——取消wreg、wmem表达式如下：

```Verilog
wire
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_j
al|i_mfc0)&~stall&~(isbj&inta&cancel)&~ecancel&~exc_ovr&~mexc_ovr;
```

```Verilog
wire wmem=i_sw&~stall&~(is_bj&inta&cancel)&~ecancel&~exc_ovr&~mexc_ovr;
```

8.5.2 实现

数据通路

```
`timescale 1ns / 1ps

module datapath(
    input clk,clrn,

    //PC更新

    input wpcir,intr,//多intr表示中断请求

    input [1:0]pcsrc,selpc,//多selpc去在npc、base、epc中选择

    output [31:0]pc,
    input [31:0]inst,

    //IF/ID
    output [5:0]op,funct,
    output [4:0]rs,rt,rd,

    output [31:0]sta,//多了给CU的状态字

    input sext,regdst,wcau,wsta,wepc,exc,mtc0,//多了写CP0寄存器的信号,还有选择信号

    input [1:0]forwardAD,forwardBD,sepc,

    input [31:0]cause,//写Cause寄存器的字
    output z,irq,

    //ID/EX
    input eshift,ealusrc,ejal,
    input [3:0]ealuc,
    input [1:0]emfc0,//选择CP0寄存器的读出数据
    output overflow,
    output [4:0]ewn,

    //EX/MEM
    output mwmem,
    output [31:0]aluoutM,memWriteData,
    input [31:0]memreadM,
    output [4:0]mwn,

    //MEM/WB
    input wm2reg,wwreg,

    output [31:0]wd,aluoutE
);

    //PC寄存器的更新
    wire [31:0]pcAdder4,bpc,jpc,rpc,npc,next_pc,epc;
    mux4to1 generate_npc(pcAdder4,bpc,rpc,jpc,pcsrc,npc);
    mux4to1 generate_nextPC(npc,epc,32'd8,32'd0,selpc,next_pc);
    regDesign #(.WIDTH(32))pc_init(clk,wpcir,clrn,next_pc,pc);
    adder generate_pcAdder4(pc,32'd4,pcAdder4);

    //IF/ID
    wire [31:0]pcAdder4D,instrD,pcD;
    regDesign
#(.WIDTH(97))if_id_inst(clk,wpcir,clrn,{pcAdder4,inst,pc,intr},{pcAdder4D,instrD,pcD,irq});//当要先
```

```
清0

    wire [15:0]imm=instrD[15:0];
    wire [31:0]imm_sext={{16{imm[15]&sext}},imm[15:0]};
    assign op=instrD[31:26];
    assign funct=instrD[5:0];//写的时候忘记标注了
    assign rs=instrD[25:21];
    assign rt=instrD[20:16];
    assign rd=instrD[15:11];
    adder generate_bpc(pcAdder4D,imm_sext<<2,bpc);
    assign jpc={pcAdder4D[31:28],instrD[25:0],2'b0};
    wire [31:0]qa,qb;
    wire [4:0]wn_temp,wwn;

    //用WB阶段的信号写回
    regfile regfile_init(~clk,wwreg,clrn,rs,rt,wwn,wd,qa,qb);
    mux2to1 generate_wn_temp(rd,rt,regdst,wn_temp);

    wire [31:0]A,B;
    mux4to1 generateA(qa,aluoutE,aluoutM,memreadM,forwardAD,A);
    mux4to1 generateB(qb,aluoutE,aluoutM,memreadM,forwardBD,B);
    assign rpc=A;
    assign z=~|(A^B);//判断0时, 刚开始写成了~(A^B)是对A^B所有位取反,需要先连续|得到的结果再取反~|

    wire [31:0]sta_left={sta[27:0],4'b0};
    wire [31:0]sta_right={4'b0,sta[31:4]};
    wire [31:0]pcE,pcM,writeEPCtemp,writeStatemp;
    mux4to1 generate_writeEPCtemp(pc,pcD,pcE,pcM,sepc,writeEPCtemp);
    mux2to1 generate_writeStatemp(sta_right,sta_left,exc,writeStatemp);
    wire [31:0]epcD,staD,cauD;

    //会有数据冒险
    wire [31:0]cau;
    mux2to1 gstaD(writeStatemp,B,mtc0,staD);
    mux2to1 gepcD(writeEPCtemp,B,mtc0,epcD);
    mux2to1 gcauD(cause,B,mtc0,cauD);
    regDesign #(.WIDTH(32))sta_reg(clk,wsta,clrn,staD,sta);
    regDesign #(.WIDTH(32))epc_reg(clk,wepc,clrn,epcD,epc);
    regDesign #(.WIDTH(32))cau_reg(clk,wcau,clrn,cauD,cau);

    //ID/EXE
    wire [31:0]pcAdder4E,eA,eB,eimmSext;
    wire [4:0]ewn_temp;
    regDesign
#(.WIDTH(165))id_ex_init(clk,1'b1,clrn,{pcAdder4D,A,B,imm_sext,wn_temp,pcD},{pcAdder4E,eA,eB,eimmSex
t,ewn_temp,pcE});//位宽刚开始数错了，也要清0
    wire [31:0]pcAdder8,aluA,aluB;
    adder generate_pcAdder8(pcAdder4E,32'd4,pcAdder8);
    mux2to1 generate_ALUa(eA,{27'b0,eimmSext[10:6]},eshift,aluA);
    mux2to1 generate_ALUb(eB,eimmSext,ealusrc,aluB);
    wire zero;
    wire [31:0]alures,pc8c0r;
    alu alu_init(aluA,aluB,ealuc,alures,zero,overflow);

    mux4to1 generate_pc8c0r(pcAdder8,sta,cau,epc,emfc0,pc8c0r);
    mux2to1 generate_aluout(alures,pc8c0r,ejal|(|emfc0),aluoutE);
    assign ewn=ewn_temp|{5{ejal}};


    //EX/MEM
    regDesign #(.WIDTH(101))
ex_mem_init(clk,1'b1,clrn,{aluoutE,eB,ewn,pcE},{aluoutM,memWriteData,mwn,pcM});//也要清0
    //MEM/WB
    wire [31:0]aluoutW,memreadW;
    regDesign
#(.WIDTH(69))mem_wb_init(clk,1'b1,clrn,{aluoutM,memreadM,mwn},{aluoutW,memreadW,wwn});//也要清0
    mux2to1 generate_wd(aluoutW,memreadW,wm2reg,wd);

endmodule
```

Verilog

控制器

指令存储器

```
module p1_inst_mem (
    input [31:0]a,// rom address
    output [31:0]inst// rom content = rom[a]
); // instruction memory, rom

    wire [31:0] rom [0:63]; // rom cells: 64 words * 32 bits
    // rom[word_addr] = instruction // (pc) label instruction
    assign rom[6'h00] = 32'h0800001d; // (00) main: j start
    assign rom[6'h01] = 32'h00000000; // (04) nop
    // common entry of exc and intr
    assign rom[6'h02] = 32'h401a6000; // (08) exc_base: mfc0 $26, c0_cause
    assign rom[6'h03] = 32'h335b000c; // (0c) andi $27, $26, 0xc
    assign rom[6'h04] = 32'h8f7b0020; // (10) lw $27,j_table($27)
    assign rom[6'h05] = 32'h00000000; // (14) nop
    assign rom[6'h06] = 32'h03600008; // (18) jr $27
    assign rom[6'h07] = 32'h00000000; // (1c) nop
    // 0x00000030: intr handler
    assign rom[6'h0c] = 32'h00000000; // (30) int_entry: nop
    assign rom[6'h0d] = 32'h42000018; // (34) eret
    assign rom[6'h0e] = 32'h00000000; // (38) nop
    // 0x0000003c: syscall handler
    assign rom[6'h0f] = 32'h00000000; // (3c) sys_entry: nop
    assign rom[6'h10] = 32'h401a7000; // (40) epc_plus4: mfc0 $26, c0_epc
    assign rom[6'h11] = 32'h235a0004; // (44) addi $26, $26, 4
    assign rom[6'h12] = 32'h409a7000; // (48) mtc0 $26, c0_EPC
    assign rom[6'h13] = 32'h42000018; // (4c) e_return: eret
    assign rom[6'h14] = 32'h00000000; // (50) nop
    // 0x00000054: unimpl handler
    assign rom[6'h15] = 32'h00000000; // (54) uni_entry: nop
    assign rom[6'h16] = 32'h08000010; // (58) j epc_plus4
    assign rom[6'h17] = 32'h00000000; // (5c) nop
    // 0x00000068: overflow handler
    assign rom[6'h1a] = 32'h00000000; // (68) ovf_entry: nop
    assign rom[6'h1b] = 32'h0800002f; // (6c) j exit
    assign rom[6'h1c] = 32'h00000000; // (70) nop
    // start: enable exc and intr
    assign rom[6'h1d] = 32'h2008000f; // (74) start: addi $8, $0, 0xf
    assign rom[6'h1e] = 32'h40886800; // (78) exc_ena: mtc0 $8, c0_status
    // unimplemented instruction
    assign rom[6'h1f] = 32'h0128001a; // (7c) unimpl: div $9, $8
    assign rom[6'h20] = 32'h00000000; // (80) nop
    // system call
    assign rom[6'h21] = 32'h0000000c; // (84) sys: syscall
    assign rom[6'h22] = 32'h00000000; // (88) nop
    // loop code for testing intr
    assign rom[6'h23] = 32'h34040050; // (8c) int: ori $4, $1, 0x50
    assign rom[6'h24] = 32'h20050004; // (90) addi $5, $0, 4
    assign rom[6'h25] = 32'h00004020; // (94) add $8, $0, $0
    assign rom[6'h26] = 32'h8c890000; // (98) loop: lw $9, 0($4)
    assign rom[6'h27] = 32'h01094020; // (9c) add $8, $8, $9
    assign rom[6'h28] = 32'h20a5ffff; // (a0) addi $5, $5, -1
    assign rom[6'h29] = 32'h14a0fffc; // (a4) bne $5, $0, loop
    assign rom[6'h2a] = 32'h20840004; // (a8) addi $4, $4, 4 # DS
    assign rom[6'h2b] = 32'h8c080048; // (ac) ov: lw $8, 0x48($0)
    assign rom[6'h2c] = 32'h8c09004c; // (b0) lw $9, 0x4c($0)
    // jump to start forever
    assign rom[6'h2d] = 32'h0800001d; // (b4) forever: j start
    // overflow in delay slot
    assign rom[6'h2e] = 32'h01094020; // (b8) add $9, $9, $8 #ov
    // if not overflow, go to start
    // exit, should be jal $31 to os
    assign rom[6'h2f] = 32'h0800002f; // (bc) exit: j exit
    assign rom[6'h30] = 32'h00000000; // (c0) nop
    assign inst = rom[a[7:2]]; // use 6-bit word address to read rom
endmodule
```

Verilog

数据存储器

```
`timescale 1ns / 1ps

module p1_data_mem(
    input clk,we,
    input [31:0]addr,datain,
    output [31:0]dataout
);
    reg [31:0] ram [0:31]; // ram cells: 32 words * 32 bits
    assign dataout = ram[addr[6:2]]; // use 5-bit word address
    always @ (posedge clk) begin
        if (we)
            ram[addr[6:2]] = datain; // write ram
    end
    integer i;
    initial begin // ram initialization
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data // (byte_addr) item in data array
        ram[5'h08] = 32'h00000030; // (20) 0. int_entry
        ram[5'h09] = 32'h0000003c; // (24) 1. sys_entry
        ram[5'h0a] = 32'h00000054; // (28) 2. uni_entry
        ram[5'h0b] = 32'h00000068; // (2c) 3. ovr_entry
        ram[5'h12] = 32'h00000002; // (48) for testing overflow
        ram[5'h13] = 32'h7fffffff; // (4c) 2 + max_int -> overflow
        ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + a3 = a3
        ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca
        ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143
        ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258
    end
endmodule
```
Verilog

8.2.3仿真

顶层模块

```
`timescale 1ns / 1ps



module top(
    input clk,clrn,intr,
    output [31:0]pc,inst,aluoutM,aluoutE,wd,
    output inta
);
    //PC更新
    wire wpcir;
    wire [1:0]pcsrc,selpc;
    //IF/ID
    wire [5:0]op,funct;
    wire [4:0]rs,rt,ewn,mwn,rd;
    wire [31:0]sta,cause;
    wire sext,regdst,wcau,wsta,wepc,exc,mtc0;
    wire [1:0]forwardAD,forwardBD,sepc;
    wire z,irq;
    //ID/EX
    wire eshift,ealusrc,ejal;
    wire [3:0]ealuc;
    wire [1:0]emfc0;
    wire overflow;
    //EX/MEM
    wire mwmem;
    wire [31:0]memWriteData;
    wire [31:0]memreadM;
    //MEM/WB
    wire wm2reg,wwreg;

    datapath  datapath_inst (
        .clk(clk),
        .clrn(clrn),
        .wpcir(wpcir),
        .intr(intr),
        .pcsrc(pcsrc),
        .selpc(selpc),
        .pc(pc),
        .inst(inst),
        .op(op),
        .funct(funct),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .sta(sta),
        .sext(sext),
        .regdst(regdst),
        .wcau(wcau),
        .wsta(wsta),
        .wepc(wepc),
        .exc(exc),
        .mtc0(mtc0),
        .forwardAD(forwardAD),
        .forwardBD(forwardBD),
        .sepc(sepc),
        .cause(cause),
        .z(z),
        .irq(irq),
        .eshift(eshift),
        .ealusrc(ealusrc),
        .ejal(ejal),
        .ealuc(ealuc),
        .emfc0(emfc0),
        .overflow(overflow),
        .ewn(ewn),
        .mwmem(mwmem),
        .aluoutM(aluoutM),
        .memWriteData(memWriteData),
        .memreadM(memreadM),
        .mwn(mwn),
        .wm2reg(wm2reg),
        .wwreg(wwreg),
        .wd(wd),
        .aluoutE(aluoutE)
      );
      cu  cu_inst (
        .clk(clk),
        .clrn(clrn),
        .funct(funct),
        .op(op),
        .rs(rs),
        .rt(rt),
        .ewn(ewn),
        .mwn(mwn),
        .rd(rd),
        .z(z),
        .overflow(overflow),
        .irq(irq),
        .sta(sta),
        .wpcir(wpcir),
        .pcsrc(pcsrc),
        .selpc(selpc),
        .sext(sext),
        .regdst(regdst),
        .wcau(wcau),
        .wsta(wsta),
        .wepc(wepc),
        .exc(exc),
        .mtc0(mtc0),
        .forwardAD(forwardAD),
        .forwardBD(forwardBD),
        .sepc(sepc),
        .cause(cause),
        .inta(inta),
        .ejal(ejal),
        .eshift(eshift),
        .ealusrc(ealusrc),
        .ealuc(ealuc),
        .emfc0(emfc0),
        .mwmem(mwmem),
        .wwreg(wwreg),
        .wm2reg(wm2reg)
    );
```

```verilog
    p1_inst_mem rom(pc,inst);
    p1_data_mem ram(clk,mwmem,aluoutM,memWriteData,memreadM);
endmodule
```

Verilog

测试模块

```verilog
`timescale 1ns / 1ps

module top_test();
    reg clk,clrn,intr;
    wire inta;
    wire [31:0]pc;
    wire [31:0]inst;
    wire [31:0]aluoutM,aluoutE,wd;
    initial begin
        clk=0;clrn=0;intr=0;
        #3;clrn=1;
        #1;clrn=0;
        #516;intr=1;
        #10;intr=0;
    end
    always #5 clk=~clk;
    top  top_inst (
    .clk(clk),
    .clrn(clrn),
    .intr(intr),
    .pc(pc),
    .inst(inst),
    .aluoutM(aluoutM),
    .aluoutE(aluoutE),
    .wd(wd),
    .inta(inta)
```

```verilog
  );
endmodule
```

Verilog

[注释1] We can even design a fully pipelined divider that can accept a new divide instruction in every clock cycle. The problem is the utilization of such expensive functional units. A trade-off must be made between the hardware cost and the performance (the inverse of program execution time).

[注释2] 通过将所有的写信号置为低电平去取消指令的执行

[注释3] 在流水线处理器的设计中，IF阶段负责指令的取指和预取操作，而ID阶段负责指令的译码和操作数的准备。当发生中断时，正在IF阶段取指的指令可能已经进入了流水线并且无法立即取消。

因此，将取消指令的操作放在ID阶段更为合适。在ID阶段生成的cancel信号会被存储到ID/EXE流水线寄存器中，以便在下一个时钟周期中生效。当被取消的指令进入ID阶段时，取消信号会被使用，禁止ID阶段生成的所有写入信号，从而实现指令的取消。

至于e_cancel信号，它是用于EXE阶段的取消信号。当该信号为高电平时，会阻止ID阶段生成的所有写入信号，因此可以将ID/EXE寄存器清零，以取消指令的执行。具体来说，e_cancel信号会使ID/EXE流水线寄存器中的相关位清零，并且阻止ID阶段将其写入EXE阶段。

[注释4] 请求操作系统内核提供某种服务

[注释5] 此时分支指令在MEM阶段