流水线暂停机制的设计与实现

一、 流水线暂停机制的设计

按照自己的理解，若某条指令在某个阶段（比如在执行阶段）需要暂停，那么我们需要在执行（EX）阶段进行流水线暂停，在此之前的阶段都要暂停，而之后的阶段可以继续执行，因为若之前的阶段也继续执行，那么新的数据经过执行阶段时便会覆盖原先的暂停时候的数据导致数据错误。

我们添加了一个 CTRL 模块，其作用是接收各阶段传递过来的流水线暂停请求信号，从而控制流水线各阶段的运行。

我们对系统结构图做如下修改：

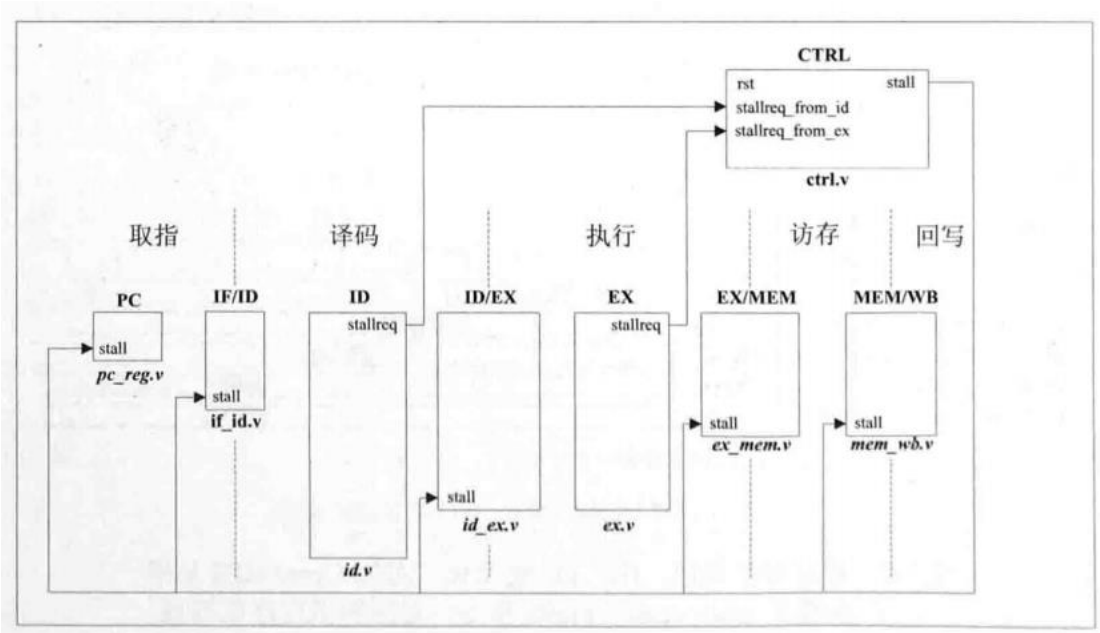

图 7-10　为了实现流水线暂停机制而对系统结构所做的修改

二、 流水线暂停机制的实现

2.1 CTRL 模块的实现

模块的接口如图：

表 7-1　CTRL 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	stallreq_from_id	1	输入	处于译码阶段的指令是否请求流水线暂停
3	stallreq_from_ex	1	输入	处于执行阶段的指令是否请求流水线暂停
4	stall	6	输出	暂停流水线控制信号

因为在我们现在所做的简易版的 CPU 只有在译码阶段和执行阶段会有流水线暂停，其他阶段例如访存和回写阶段不会出现某条指令需要多个时钟周期才能完成。

其中 stall 信号是一个 6 位信号，每一位含义如下：

- stall[0]: 表示取地址 PC 是否表示不变，为 1 表示不变；
- stall[1]: 表示流水线取值阶段是否暂停，为 1 表示暂停；
- stall[2]: 表示流水线译码阶段是否暂停，为 1 表示暂停；
- stall[3]: 表示流水线执行阶段是否暂停，为 1 表示暂停；
- stall[4]: 表示流水线访存阶段是否暂停，为 1 表示暂停；
- stall[5]: 表示流水线回写阶段是否暂停，为 1 表示暂停。

源代码 **ctrl.v：**

```
module ctrl(
    input wire rst,
    input wire stallreq_from_id,
    input wire stallreq_from_ex,
    output reg[5:0] stall
    );
    always @ (*) begin
        if (rst == `RstEnable) begin
            stall <= 6'b000000;
        end else if (stallreq_from_ex == `Stop) begin
            stall <= 6'b001111;
        end else if (stallreq_from_id == `Stop) begin
            stall <= 6'b000111;
        end else begin
            stall <= 6'b000000;
        end
    end
endmodule
```

有个小插曲，我在宏定义的时候不小心在后面添加了分号，结果提示语法错误，改了十分钟才发现~

```
`define Stop      1'b1;      //Pause
`define NoStop    1'b0;
```

代码很好理解，不多说了。

2.2 修改取指阶段

2.1.1 PC 模块的修改

PC 模块新增了 stall 接口，该接口来自 ctrl 模块，PC 模块修改如下：

源代码 **pc.v:**

```
module pc_reg(
    input wire clk,
    input wire rst,
    input wire [5:0] stall,
    output reg [`InstAddrBus] pc,
    output reg ce
);

always @(posedge clk) begin
    if (rst == `RstEnable) ce <= `ChipDisable;
    else ce <= `ChipEnable;
end

always @(posedge clk) begin
    if (ce == `ChipDisable) pc <= 32'h0000_0000;
    //PC equals 0 when instruction memory is prohibited
    else if (stall[0] == `NoStop) begin
        pc <= pc + 4'h4;
    //PC add 4 per clock period when instrution is enabled
    end
end
endmodule
```

2.2.2 IF/ID 模块的修改

源代码 **if_id.v**：

```
always @(posedge clk) begin
    if (rst == `RstEnable) begin
        id_pc <= `ZeroWord;
        id_inst <= `ZeroWord;
    end else if (stall[1] == `Stop && stall[2] == `NoStop)begin
         id_pc <= `ZeroWord;
         id_inst <= `ZeroWord;
    end else if (stall[1] == `NoStop) begin
        id_pc <= if_pc;
        id_inst <= if_inst;
    end
end
```

当 stall[1]为 Stop，stall[2]为 NoStop 时，表示取指阶段暂停，而译码阶段继续，因此使用空指令作为下一个周期进入译码阶段的指令。

2.3 修改译码阶段

2.3.1 ID 模块的修改

增加一个输出接口 stallreq

2.3.2 ID/EX 模块的修改

源代码 **id_ex.v**：

```
always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            …
        end else if (stall[2] == `Stop && stall[3] == `NoStop) begin
            ex_aluop <= `EXE_NOP_OP;
            ex_alusel <= `EXE_RES_NOP;
            ex_reg1 <= `ZeroWord;
            ex_reg2 <= `ZeroWord;
            ex_wd <= `NOPRegAddr;
            ex_wreg <= `WriteDisable;
        end else if (stall[2] == `NoStop) begin
           …
        end
    end
```

2.4 修改执行阶段

2.4.1 EX 模块的修改

增加一个输出接口 stallreq

2.4.2 EX/MEM 模块的修改

源代码 **ex_mem.v：**

```
always @(posedge clk) begin
        if (rst == `RstEnable) begin
            …
        end else if (stall[3] == `Stop && stall[4] == `NoStop) begin
            mem_wd <= `NOPRegAddr;
            mem_wreg <= `WriteDisable;
            mem_wdata <= `ZeroWord;
            mem_hi <= `ZeroWord;
            mem_lo <= `ZeroWord;
            mem_whilo <= `WriteDisable;
        end else if (stall[3] == `NoStop) begin
            mem_wdata <= ex_wdata;
            mem_wd <= ex_wd;
            mem_wreg <= ex_wreg;
            mem_hi <= ex_hi;
            mem_lo <= ex_lo;
            mem_whilo <= ex_whilo;
        end
    end
```

2.5 修改访存阶段

只需修改 MEM/WB 模块即可：

源代码 **mem_wb.v：**

```
always @(posedge clk) begin
    if (rst == `RstEnable) begin
        …
    end else if (stall[4] == `Stop && stall[5] == `NoStop) begin
        wb_wd <= `NOPRegAddr;
        wb_wdata <= `ZeroWord;
        wb_wreg <= `WriteDisable;
        wb_hi <= `ZeroWord;
        wb_lo <= `ZeroWord;
```

```
            wb_whilo <= `WriteDisable;
        end else if (stall[4] == `NoStop) begin
            wb_wd <= mem_wd;
            wb_wdata <= mem_wdata;
            wb_wreg <= mem_wreg;
            wb_hi <= mem_hi;
            wb_lo <= mem_lo;
            wb_whilo <= mem_whilo;
        end
    end
```

2.6 修改顶层模块

源代码 **openmips.v：**

```
wire [5:0] stall;
wire stallreq_from_id;
wire stallreq_from_ex;

//CTRL
ctrl ctrl0(
    .rst(rst),
    .stall_from_id(stallreq_from_id),
    .stall_from_ex(stallreq_from_ex),
    .stall(stall)
);
```

最后记住在 openmips 模块中将上述提到的接口实例化，并且要将 stall_from_id 和 stall_from_ex 初始化，否则将会出现 stall_from_id 和 stall_from_ex 高阻态情况（不要问我怎么知道的）。

再总结一下上一节乘法指令的实现，乘法指令有 mul、mult 和 multu 几种，其中 mul 是将乘积结果的低 32 位存入通用寄存器，而 mult 和 multu 则是将乘积结果的低 32 位存入 LO 寄存器，高 32 位存入 HI 寄存器，因此他们属于两种不同的运算，尽管都属于乘法，但是却不能使用同一种运算类型。

我们重新定义 mul 运算类型 alusel 为 mul，定义 mult 和 multu 为 nop（空操作），前者在通用寄存器处操作，后者在 HILO 寄存器中单独判断。