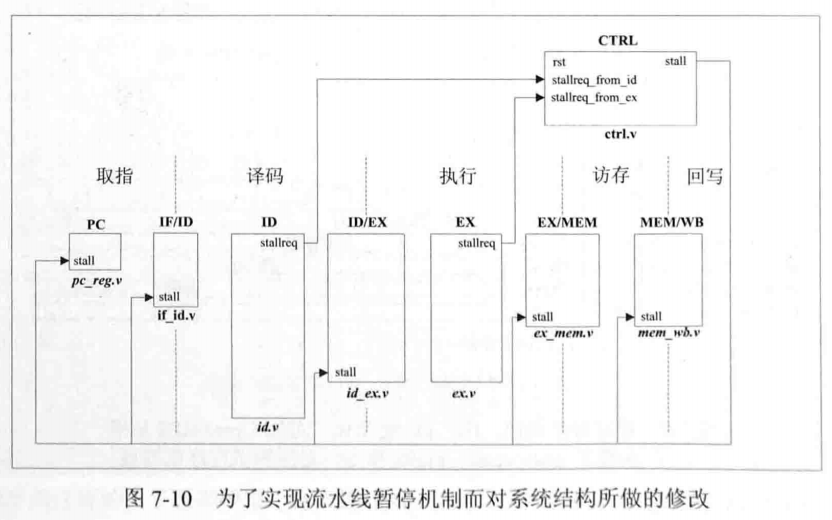
**流水线暂停机制的设计与实现**

# 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：



# 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：



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

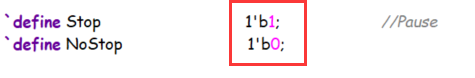
stall <= 6'b000000;

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~



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。