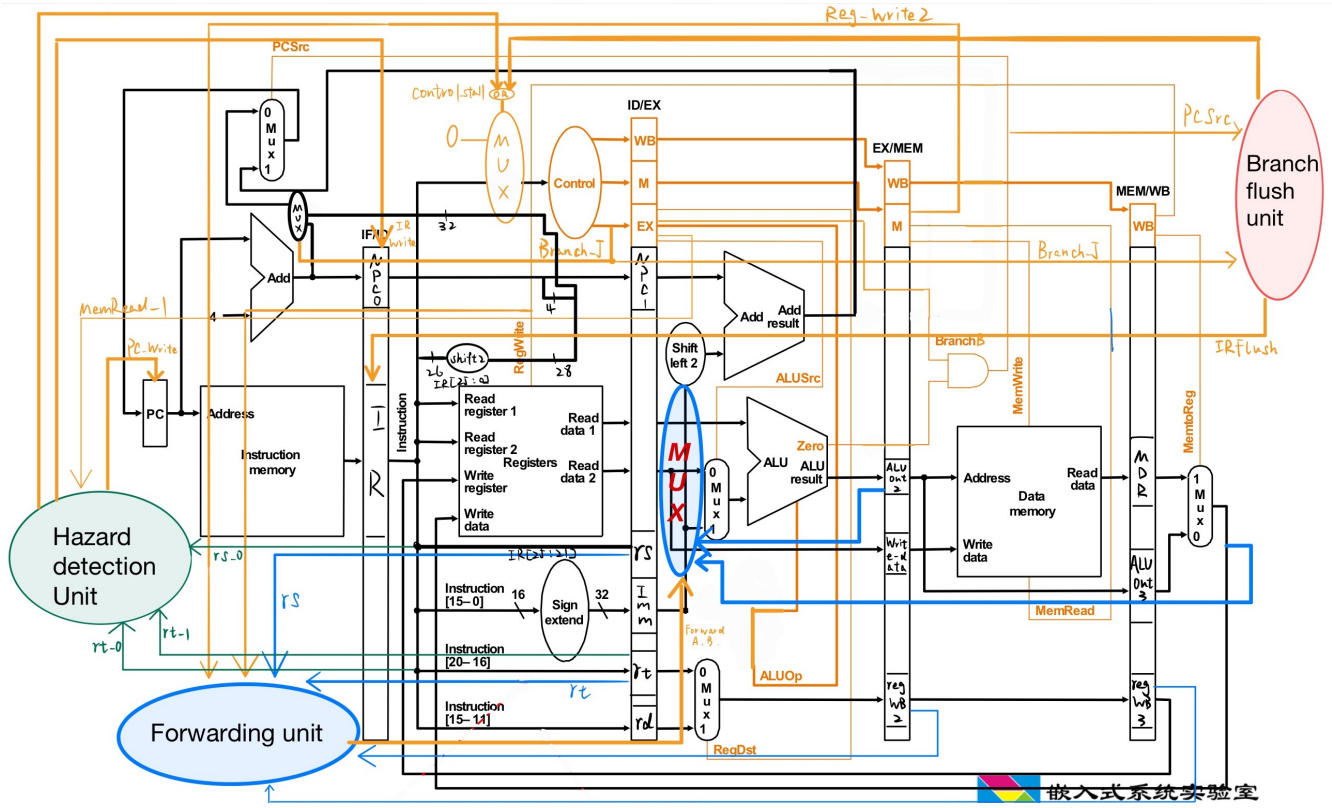
实验六 综合设计

- 实验时间：2019. 5. 17 —— 2019. 5. 30
- **袁旷 PB17081543**
- 实现内容：利用支持完全转发、阻塞、静态分支预测的 32 位流水线 MIPS-CPU 实现带 I/O 的冒泡排序

一、概述

CPU 的完全电路图：



主要亮点：

1. 实现了支持包含 R 类指令、I 类指令、访存指令、分支跳转指令在内的 17 条经典 MIPS 指令的流水线 CPU，相比于课堂上与教材中的版本增加了 I 类指令和 jump 指令。
2. 实现了完全转发（forwarding），可将 EX/MEM 段与 MEM/WB 段的相关数据转发至 EX 段，以及寄存器堆可在同一周期内完成读写。
3. 利用硬件方案实现了 load-use 阻塞（interlock）。若汇编程序出现 load-use 情况时，不需要人为添加空指令，机器可自动阻塞一个周期。
4. 实现了静态分支预测。若分支跳转成功，可将已进入流水线的错误指令 flush 掉，不需要人为添加空指令。
5. 实现了 I/O , 在运行冒泡排序之前，可以修改 Data_memory 中的待排序的数据。

二、各模块功能与核心代码介绍

1. Forwarding_unit

若当前位于 EX/MEM 段和 MEM/WB 段的指令会涉及写寄存器操作，那么将它们将要写回的寄存器地址与 ID/EX 段的操作数寄存器相比较。如果有涉及数据相关，那么将 MEM 段和 WB 段的寄存器数据转发回 EX 段，利用数据选择器，取代 EX 段的源数据。这个模块可以解决两个周期内的大部分数据相关。

代码：

```
module Forwarding_unit(
    input RegWrite_2,
    input RegWrite_3,
    input [4:0]rs,
    input [4:0]rt,
    input [4:0]reg_WB_2,
    input [4:0]reg_WB_3,
    output reg [1:0]ForwardA,
    output reg [1:0]ForwardB
    );
    wire c1_A,c1_B,c2_A,c2_B;
    assign c1_A = (RegWrite_2==1 && reg_WB_2==rs)? 1:0;
    assign c1_B = (RegWrite_2==1 && reg_WB_2==rt)? 1:0;
    assign c2_A = (RegWrite_3==1 && reg_WB_3==rs && reg_WB_2!=rs)? 1:0;
    assign c2_B = (RegWrite_3==1 && reg_WB_3==rt && reg_WB_2!=rt)? 1:0;

    always@(c1_A or c2_A)
    begin
        if(c1_A == 1) ForwardA = 2'b10;
        else if(c2_A == 1) ForwardA = 2'b01;
        else ForwardA = 2'b00;
    end

    always@(c1_B or c2_B)
    begin
        if(c1_B == 1) ForwardB = 2'b10;
        else if(c2_B == 1) ForwardB = 2'b01;
        else ForwardB = 2'b00;
    end

endmodule
```

其中 C1 转发是转发 EX/MEM 段的数据，C2 转发是转发 MEM/WB 段的数据。两种转发只会执行其中一个，其中 C1 转发优先于 C2 转发，因为 EX/MEM 段的数据较新。

A 和 B 分别对应 EX 段的两个操作数。ForwardA ， ForwardB 是控制 EX 段数据选择器的控制信号。

2. 寄存器堆

若设计数据相关的两条指令相差三个周期，则需要通过修改 ID 段的时序来解决。让寄存器堆在 ID 段前半周期写入数据，后半周期再将数据读出到寄存器。可以解决该类数据相关。

需要注意的是，具体实现时，不应该将 ID/EX 段寄存器改为下降沿触发（延后半个周期），因为这么做会导致 CPU 其他部分的时序也受到影响。正确做法是将寄存器堆内的写时序改为下降沿触发（提前半个周期）。

3. Hazard_detection_unit

上述两个模块可以解决绝大部分的数据相关问题，但是 load-use 型数据相关仍然不能解决。解决 load-use 型数据相关可以通过在指令中加气泡或硬件阻塞来解决。此处采用硬件阻塞方案。

利用 EX 段的 Memread 控制信号判断当前指令是否为 lw 指令。若为 lw 指令，且要写回的寄存器地址与当前 ID 段操作数寄存器地址相比较。若相同，判定为 load-use 型数据相关。此时采取的操作为将当前下一周期 EX 段的控制信号清空，相当于把 lw 后的一条指令改为空指令。同时把 IRwrite 和 PCwrite 信号置为 0，即让该指令不被下一条指令覆盖，达到延后一个周期再执行的效果。

代码：

```
module Hazard_detection_unit(
    input MemRead,
    input [4:0]rt_1,
    input [4:0]rs_0,
    input [4:0]rt_0,
    output reg control_stall,
    output reg IRWrite,
    output reg PCWrite
    );
    always@(*)
    begin
        if(MemRead==1 && ((rt_1 == rs_0)||(rt_1 == rt_0)))
        begin
            control_stall = 1;
            IRWrite = 0;
            PCWrite = 0;
        end
        else
        begin
            control_stall = 0;
            IRWrite = 1;
            PCWrite = 1;
        end
    end
endmodule
```

至此，所有数据相关都被解决。

4. Branch_flush_unit

按照原基本流水线的 CPU 方案，beq 指令的执行需要 4 个周期，jump 指令的执行需要 3 个周期。而分支跳转指令时，因不能判断下一条指令是什么，需要插入 2~3 个 nop 指令，导致流水线的停滞，降低效率。

为了解决这个控制相关的问题，采用静态分支预测的思想，总是预测分支跳转不执行。若真的执行，那么将错误进入流水线的指令清空。

为了优化分支跳转指令的完成时间，减少错误进入流水线的指令数，通过修改数据通路将 beq 指令改为 3 个周期，jump 指令改为 2 个周期。

将控制 beq 跳转与 jump 指令跳转的控制信号传入此模块。如果 beq 指令需要跳转，那么将 ID/EX 段的控制信号清空，同时将 IF/ID 段已经取指的 IR 清空，相当于将已经错误进入流水线的两条指令清除。jump 指令只需将 IR 清空即可。

代码：

```
module Branch_flush_unit(
    input Branch_J,
    input PCSrc,
    output IRFlush,
    output control_stall
    );
    assign IRFlush = (Branch_J==1 || PCSrc==1)? 1:0;
    assign control_stall = (PCSrc==1)? 1:0;
endmodule
```

解决了控制相关。

三、结果分析

通过将 Data_memory 与 Instruction_memory 分离的哈佛结构，解决了结构相关。通过转发和阻塞，解决了控制相关。通过静态分支预测，解决了控制相关。自此，该流水线 CPU 可以执行任意一个**不需人为添加空指令**的汇编程序。性能优秀。若后续需要继续提升性能，可以加入动态分支预测，如果需要处理异常，则实现中断。

将冒泡排序的指令导入指令存储器，将生成的 100 个随机数导入数据存储器。在排序之前，可以利用开关作为 I/O 修改数据存储器内任意一个待排序数据的值。

通过测试，排序结果正确。

四、总结与感想

本次实验通过实现一个比较完整的流水线 CPU 与应用，深刻地理解了流水线 CPU 的工作原理，以及各种相关的产生的原因和具体的处理方法。听课和看书往往都是纸上谈兵，只有一步一步的实现才能真正发现具体的问题所在。

此次实验不仅帮助我更好的掌握了组成原理的知识，完成如此大型的硬件电路也帮助我提升了工程能力、统筹组织全局的能力以及对具体问题进行分析的能力。计算机硬件的内容虽然比较繁琐，但是真正实现起来还是发现硬件的体系结构充满了工程的美感，成功完成也是让人成就感满满。另外，对 CPU 工作原理有了深刻的理解也会帮助我更好的掌握其他方面的知识，比如流水线的各种相关就与编译原理结合紧密。

作为一个 CS 专业的学生，我对这次实验感到很满意，完整地实现这样一个 CPU 相信会是大有裨益的。