Report 2 — September 20

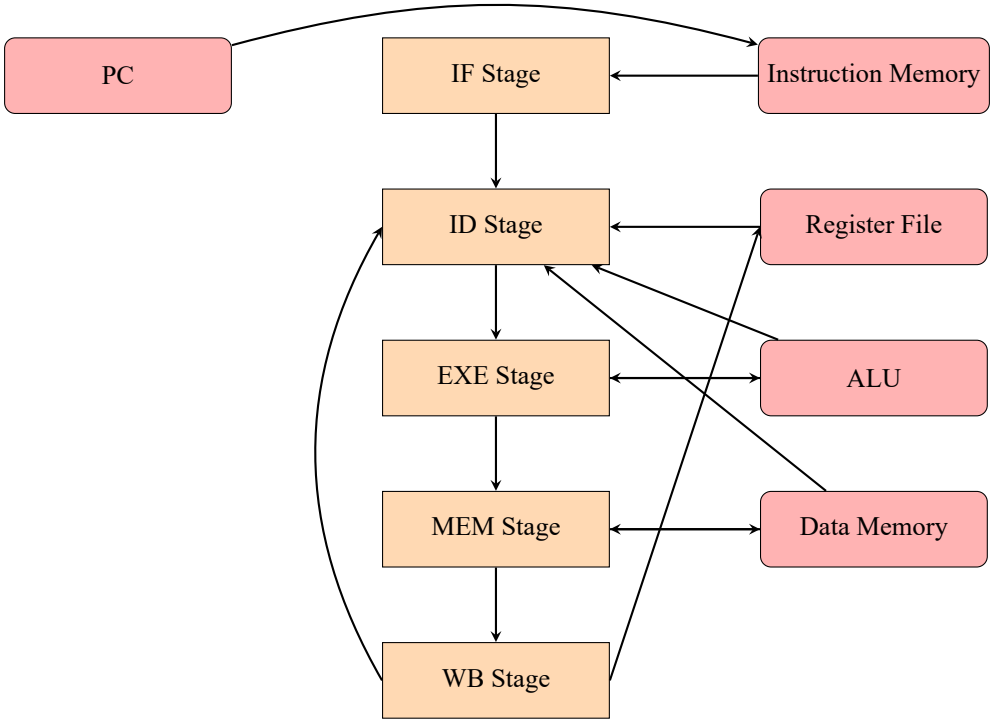
Lecturer: Wang Wenxiang *Completed by: Zhang Jiawei*

2.1 实验任务

1. exp7: 通过修改单周期 CPU 代码，实现一个不考虑数据冲突的简单流水线处理器，包括取指、译码、执行、访存和写回五个阶段。
2. exp8: 在流水线处理器的基础上，实现冲突时阻塞流水级，解决数据冲突。
3. exp9: 在流水线处理器的基础上，实现冲突时进行数据前递，解决数据冲突，提高流水线效率。

2.2 实验设计

下面给出大致设计框图：



IF 阶段从指令存储器中读取指令，ID 阶段进行译码，获取各个控制信号，并从寄存器堆中读取数据（可能是前递的数据），EXE 阶段进行运算，MEM 阶段访问数据存储器，WB 阶段将结果写回寄存器堆，这便是设计思路。当然，具体实现要比这个复杂得多，需要考虑各种情况。数据怎么在流水级之间传递？怎么判定数据冲

突？数据冲突的时候应该怎么办？分支指令怎么处理？应该怎样用控制信号来控制流水线的暂停和恢复？这些问题就是实验过程中我们需要解决的问题，也是这三次实验的重点。

2.2.1 流水线所使用的控制信号

我们的流水线总共有五个阶段，每个阶段都需要各自的控制信号。valid 指示当前阶段是否有效，allowin 指示是否允许该个阶段接收数据，readygo 指示是否允许该阶段向下一个阶段传递数据，pass 指示数据是否能够通过该阶段。以 EXE 阶段为例，其控制信号如下：

```
// EXE stage
reg   exe_valid;
wire  exe_allowin;
wire  exe_readygo;
wire  exe_pass;
always @(posedge clk) begin
   if (reset) begin
      exe_valid <= 1'b0;
   end
   else begin
      exe_valid <= id_pass & exe_allowin;
   end
end
assign exe_allowin = ~exe_valid | mem_allowin & exe_pass;
assign exe_readygo = 1'b1;
assign exe_pass = exe_valid & exe_readygo;
```

EXE 阶段的 valid 信号在复位时为 0，在 ID 阶段数据通过并且 EXE 阶段允许接收数据时为 1。EXE 阶段的 allowin 信号在 EXE 阶段无效、或者 MEM 阶段允许接收数据并且 EXE 阶段允许数据通过时为 1，否则为 0。EXE 阶段的 readygo 信号始终为 1，表示 EXE 阶段始终允许数据通过。EXE 阶段的 pass 信号在 EXE 阶段有效并且 EXE 阶段允许数据通过时为 1，否则为 0。

其余阶段的控制信号赋值类似，但也有不同，对控制流水线的暂停和恢复起着重要的作用，在之后的篇幅中会详细讨论。

2.2.2 流水级间缓存的实现

由于所有内存都是理想内存，不需要考虑内存握手等待，故而我们可以认定，一个时钟周期之内就可以完成一个流水级的操作，这是三次实验的大前提。依流水级顺序向下传递数据在不考虑数据冲突的 exp7 中就得以实现。与单周期处理器不同的是，一条指令在某个阶段产生的数据或控制信号，可能在后续的阶段中还会用到，倘若我们都使用组合逻辑，那么这些需要复用的数据或控制信号就消失了。因此，我们需要在流水级之间加入缓存，以便进行数据的传递。一个 IF 与 ID 阶段间的缓存示例如下：

```
// IF --> ID
reg [31:0] inst_id;
```

```
reg [31:0] pc_id;
always @(posedge clk) begin
   if (if_pass & id_allowin) begin
      inst_id <= inst;
      pc_id  <= pc;
   end
end
```

`inst` 和 `pc` 是 IF 阶段产生的指令和 PC 值，显然，指令需要进行后续译码、PC 可能需要进行后续跳转地址的计算，自然需要加入缓存中，传到 ID 阶段。由于一个时钟周期就是一个流水级的长度，在时钟上升沿更新寄存器，恰好 ID 阶段就获得了当前指令和 PC 值。这样，我们就可以在 ID 阶段使用 `inst_id` 和 `pc_id`，而不用担心这两个信号在 IF 阶段被覆盖。

后续缓存的实现原理都是一样的，只是需要传递的数据不同。比如 ID 与 EXE 阶段间的缓存，就需要传递各个控制信号、寄存器堆的读取数据、立即数等等，下面给出代码（其中有些数据可能并不需要传递）：

```
// ID --> EXE
reg [31:0] inst_exe;
reg [31:0] pc_exe;
reg [31:0] alu_op_exe;
reg [ 4:0] dest_exe;
reg [31:0] rj_value_exe;
reg [31:0] rkd_value_exe;
reg [31:0] imm_exe;
reg [31:0] br_offs_exe;
reg [31:0] jirl_offs_exe;
reg        src1_is_pc_exe;
reg        src2_is_imm_exe;
reg        res_from_mem_exe;
reg        dst_is_r1_exe;
reg        gr_we_exe;
reg        mem_we_exe;
reg        ld_w_exe;
always @(posedge clk) begin
   if (reset) begin
      gr_we_exe <= 1'b0;
      dest_exe <= 5'b0;
      ld_w_exe <= 1'b0;
   end
   else if (id_pass & exe_allowin) begin
      inst_exe   <= inst_id;
      pc_exe     <= pc_id;
      alu_op_exe <= alu_op;
      dest_exe   <= dest;
```

```
        rj_value_exe <= rj_value;
        rkd_value_exe<= rkd_value;
        imm_exe    <= imm;
        br_offs_exe <= br_offs;
        jirl_offs_exe<= jirl_offs;
        src1_is_pc_exe<= src1_is_pc;
        src2_is_imm_exe<= src2_is_imm;
        res_from_mem_exe<= res_from_mem;
        dst_is_r1_exe<= dst_is_r1;
        gr_we_exe  <= gr_we;
        mem_we_exe <= mem_we;
        ld_w_exe   <= inst_ld_w;
    end
end
```

之后的流水级间缓存不再赘述。

2.2.3 数据冲突的判定与解决方案

数据冲突是个非常棘手的问题。当一条指令需要在 ID 阶段从寄存器堆读数据，但前面一条指令还没有到达 WB 阶段写回数据，倘若这条指令的读地址和前一条指令的写地址相同时，这条指令要是再从寄存器堆读数据，就会读到错误的数据，这就是数据冲突。exp7 不考虑数据冲突，是因为反汇编代码在编写时，在所有可能引发冲突的指令之后都加入了四条“nop”指令（即 `addi.w $r0,$r0,0`），来让前一条指令的数据写回：

```
1c05fa1c: 15125710  lu12i.w $r16,-486728(0x892b8)
1c05fa20: 02800000  addi.w $r0,$r0,0
1c05fa24: 02800000  addi.w $r0,$r0,0
1c05fa28: 02800000  addi.w $r0,$r0,0
1c05fa2c: 02800000  addi.w $r0,$r0,0
1c05fa30: 02a95e10  addi.w $r16,$r16,-1449(0xa57)
1c05fa34: 14f47191  lu12i.w $r17,500620(0x7a38c)
1c05fa38: 02800000  addi.w $r0,$r0,0
1c05fa3c: 02800000  addi.w $r0,$r0,0
1c05fa40: 02800000  addi.w $r0,$r0,0
1c05fa44: 02800000  addi.w $r0,$r0,0
1c05fa48: 02968231  addi.w $r17,$r17,1440(0x5a0)
```

虽然这么做确实避免了数据冲突，但是这么做大大降低了流水线的效率，如果每条指令都有冲突，那么流水线的效率就会降到和单周期处理器一样。因此，我们需要解决数据冲突的问题。exp8 和 exp9 各自提供了一种解决方案：

- exp8: 取到某一条指令，在该指令译码阶段，判断是否有数据冲突，如果有，就暂停流水线（将 `id_readygo` 赋 0），直到与之冲突的指令写回数据。

- exp9: 取到某一条指令，在该指令译码阶段，判断是否有数据冲突，如果有，就进行数据前递，将数据从后面的流水级传递到前面的流水级。仅在 Load 指令与紧邻下一条指令之间存在数据冲突时，才会暂停流水线。

• exp8 的数据冲突解决方案

讲义中写到：“处于译码流水级的指令具有来自非 0 号寄存器的源操作数，那么如果这些源操作数中任何一个的寄存器号与当前时刻处于执行级、访存级或写回级的指令的目的操作数的寄存器号（非 0 号）相同，则表明处于译码级的指令与执行级、访存级或写回级的指令存在会引发冲突的‘写后读’相关关系。”也就是说，我们需要在代码中判断这种情况。如果符合这种情况，就需要暂停流水线。

```
assign id_readygo = ~(gr_we_exe & exe_valid & (inst_src_is_r1 & (rf_raddr1 == dest_exe)
                                             | inst_src_is_r2 & (rf_raddr2 == dest_exe))
                    | gr_we_mem & mem_valid & (inst_src_is_r1 & (rf_raddr1 == dest_mem)
                                             | inst_src_is_r2 & (rf_raddr2 == dest_mem))
                    | gr_we_wb & wb_valid & (inst_src_is_r1 & (rf_raddr1 == dest_wb)
                                             | inst_src_is_r2 & (rf_raddr2 == dest_wb)));

assign inst_src_is_r1 = inst_add_w | inst_sub_w | inst_slt | inst_sltu
                      | inst_nor | inst_and | inst_or | inst_xor
                      | inst_slli_w | inst_srli_w | inst_srai_w | inst_addi_w
                      | inst_ld_w | inst_st_w | inst_beq | inst_bne | inst_jirl;

assign inst_src_is_r2 = inst_add_w | inst_sub_w | inst_slt | inst_sltu
                      | inst_nor | inst_and | inst_or | inst_xor
                      | inst_beq | inst_bne | inst_st_w;
```

这段代码里，`inst_src_is_r1` 判断指令是否需要从 `rf_raddr1` 地址读取数据，`inst_src_is_r2` 判断指令是否需要从 `rf_raddr2` 地址读取数据。如果这两个地址与前面的某条有效指令的写地址相同且写使能为 1，就说明存在数据冲突，需要暂停流水线。

• exp9 的数据冲突解决方案

exp8 中需要结果的指令在译码级等待的过程中，前面产生结果的指令其实已经生成出结果了，只不过是还没有写入到寄存器堆中。让前面的指令直接把已经生成出来的结果直接转给后面的指令，这样后面的指令不就不再需要等待了，这就是数据前递的思想。在执行级就能产生结果的指令可以直接把结果传递给译码级，但是像 Load 指令这种需要访存的指令，就需要等到访存级才能产生结果，当它与后一条指令产生数据冲突时，就需要暂停流水线（将 `id_readygo` 赋 0）。

数据前递的方法如下：路径起点位于执行级 ALU 的结果输出处/访存级结果输出处/写回级结果输出处，终点位于译码级寄存器堆读出结果生成逻辑处；遇见 Load 指令与紧邻下一条指令之间存在数据冲突时，暂停流水线：

```
assign id_readygo = ~((rf_raddr1 == dest_exe | rf_raddr2 == dest_exe) && ld_w_exe) |
    ~exe_valid;
```

```
assign rj_value = (rf_raddr1 == dest_exe && dest_exe != 5'b0 && gr_we_exe && !ld_w_exe) ?
    alu_result_forward :
                (rf_raddr1 == dest_mem && dest_mem != 5'b0 && gr_we_mem) ?
                    res_from_mem_mem ? mem_result_forward : alu_result_mem_forward :
                (rf_raddr1 == dest_wb && dest_wb != 5'b0 && gr_we_wb ) ?
                    final_result_forward : rf_rdata1;

assign rkd_value = (rf_raddr2 == dest_exe && dest_exe != 5'b0 && gr_we_exe && !ld_w_exe)
    ? alu_result_forward :
                (rf_raddr2 == dest_mem && dest_mem != 5'b0 && gr_we_mem) ?
                    res_from_mem_mem ? mem_result_forward : alu_result_mem_forward :
                (rf_raddr2 == dest_wb && dest_wb != 5'b0 && gr_we_wb ) ?
                    final_result_forward : rf_rdata2;

assign alu_result_forward = alu_result;
assign mem_result_forward = mem_result;
assign alu_result_mem_forward = alu_result_mem;
assign final_result_forward = final_result;
```

这段代码中，rj_value 和 rkd_value 是 ID 阶段从寄存器堆读取的数据/发生数据冲突前递得到的数据。这里的数据需要进行多路选择，原因在于一条指令的读地址可能和之前多条指令的写地址相同，这时就需要选择最新的数据，即流水级优先顺序为执行级、访存级、写回级。这里的 alu_result_forward、mem_result_forward、alu_result_mem_forward、final_result_forward 是前递得到的数据。

ID 阶段阻塞流水线的条件是：前一条指令的写地址与当前指令的读地址相同，且前一条指令是 Load 指令。其中 exe_valid 用于将 id_readygo 复位为 1，以便在流水线暂停后恢复。

2.2.4 分支指令控制相关处理

分支指令可能会改变 PC 的值，因此在流水线中，分支指令需要特殊处理。我们的处理方式是，当译码阶段识别到分支指令时，将 PC 的值传递给 IF 阶段，此时 IF 阶段已经误取了下一条指令，我们把误取的指令取消掉，代码如下：

```
always @(posedge clk) begin
    if (reset) begin
        if_valid <= 1'b0;
    end
    else if (if_allowin) begin
        if_valid <= 1'b1;
    end
    else if (br_taken_cancel) begin
        if_valid <= 1'b0;
    end
end
```

```
always @(posedge clk) begin
    if (reset) begin
        id_valid <= 1'b0;
    end
    else if (br_taken_cancel) begin
        id_valid <= 1'b0;
    end
    else if (id_allowin) begin
        id_valid <= if_pass;
    end
end
```

当要分支时，把 IF 阶段取到的指令取消掉，把从取指阶段到译码阶段的 `valid` 置为 0，再将取值阶段的 `valid` 置为 0 以免其再次取出错误指令即可。这样，处理器就不会执行这条指令，而是继续取分支指令的目标地址。

2.3 实验结果

exp8 顺利通过：

```
==============================================================
Test end!
----PASS!!!
$finish called at time : 1322405 ns : File "C:/Users/ZhangJiawei/Desktop/CAlab_init/prj2_exp8/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:06 ; elapsed = 00:00:22 . Memory (MB): peak = 968.613 ; gain = 7.855
```

exp9 顺利通过：

```
==============================================================
Test end!
----PASS!!!
$finish called at time : 836225 ns : File "C:/Users/ZhangJiawei/Desktop/CAlab_init/prj2_exp9/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:05 ; elapsed = 00:00:19 . Memory (MB): peak = 1105.059 ; gain = 0.000


==============================================================
Test end!
----PASS!!!
$finish called at time : 604585 ns : File "C:/Users/ZhangJiawei/Desktop/CAlab_init/prj2_exp9/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:00:03 ; elapsed = 00:00:13 . Memory (MB): peak = 1019.676 ; gain = 0.000
```

可以看出 exp9 的速度要显著高于 exp8，这是因为 exp9 在数据冲突时进行了数据前递，避免了流水线的暂停。exp9 中，判断流水线是否阻塞是一个影响性能的巨大因素：在 `assign id_readygo = ((rf_raddr1 == dest_exe | rf_raddr2 == dest_exe) && ld_w_exe) | exe_valid;` 这一行代码中，倘若去掉 `ld_w_exe`，那么流水线就会在每一条与执行阶段冲突的指令后都暂停，这样流水线的效率就会大大降低。如图所示，这一点足足造成了有 23 万纳秒的差距。

下面是同一段汇编代码在 exp8 和 exp9 中的运行对比：

```
1c010000: 157f5fe4  lu12i.w $r4,-263425(0xbfaff)
1c010004: 02810084  addi.w $r4,$r4,64(0x40)
1c010008: 157f5fe5  lu12i.w $r5,-263425(0xbfaff)
1c01000c: 0280c0a5  addi.w $r5,$r5,48(0x30)
1c010010: 157f5fe6  lu12i.w $r6,-263425(0xbfaff)
```

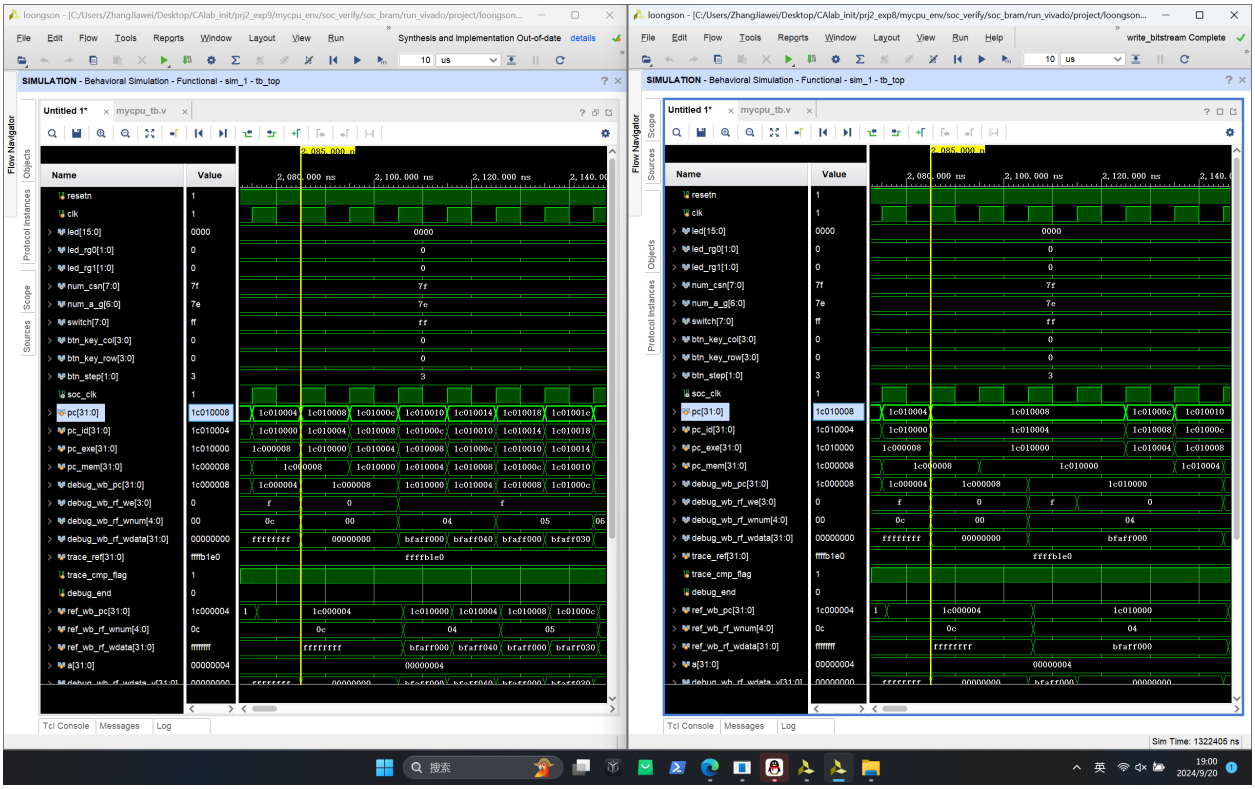


图 **2.1.** exp8 和 exp9 的运行对比，左边为 exp9，右边为 exp8

虽然两条 `addi` 指令均与前一条指令存在数据冲突，但 exp9 中的数据前递使得流水线不会暂停，而 exp8 中的流水线在每一条冲突指令后都会暂停。在波形图中可以看到，exp9 中的流水线依然保持一个时钟周期一个流水级的速度，而 exp8 中的流水线在每一条冲突指令后都会多等待三个时钟周期，明显看出 exp9 的速度会快得多。

2.4 实验总结

本次实验是关于流水线处理器的实验，通过不断修改，最终实现了一个考虑控制相关和数据相关的流水线处理器。在实验过程中，我学习了流水线处理器的设计思想，学会了如何在流水线中处理数据冲突的方法，学会了如何在流水线中处理分支指令。这次实验是一个很好的机会，让我对流水线处理器有了更深的理解，对处理器性能的提高有了更深的认识。