

实验报告成绩:	成绩评定日期:
---------	---------

2021~2022 学年秋季学期
A3705060050 《计算机系统》必修课
课程实验报告



班级：人工智能 1901

组长：孙平炜

组员：孙翰文 刘寒

报告日期：2021.12.18

目录

- 项目基本情况 2
 - 完成情况 2
 - 组内分工 2
- 总体设计 2
- 流水段设计 6
 - IF 段 6
 - ID 段 6
 - 译码器 6
 - 寄存器的读写和数据相关 8
 - 操作数和操作类型的选择 11
 - 指令回写控制 12
 - 分支跳转和暂停请求 13
 - EX 段 14
 - 乘法器模块 14
 - 访存控制 17
 - MEM 段 19
 - WB 段 19
- 总结与改进 22
- 参考资料 22

项目基本情况

完成情况

本次实验采用 Vivado 2019.2 平台编译运行，在程序编写过程中使用 VScode。

按照程序内部检测机制的反馈，共计通过 64 个测试点位，实现了 52 条指令。此外，我们组使用了组内自行编写的乘法器并进行了上板验证。

组内分工

孙平炜：解决数据相关、添加指令、解决 **stall** 请求相关问题、解决乘除法寄存器的相关问题。

孙瀚文：解决跳转指令相关问题、添加指令、实现本组的乘法器、解决访存相关问题。

刘寒：解决访存相关问题、添加指令。

总体设计

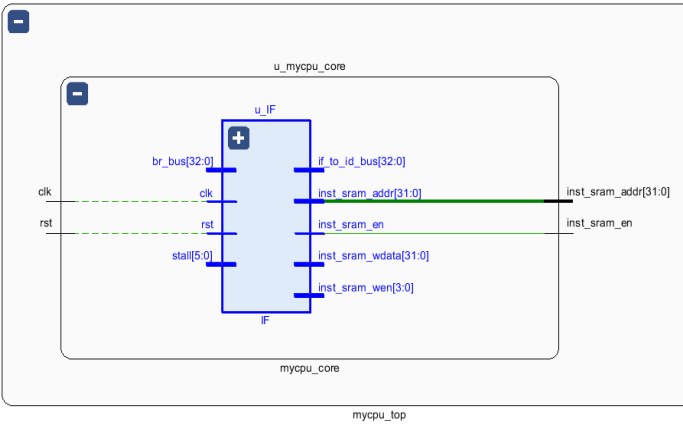
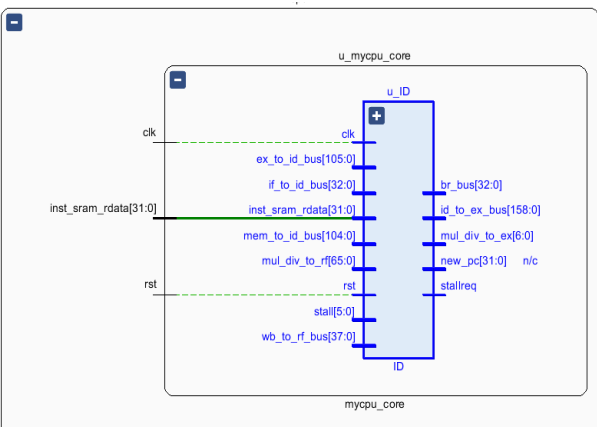
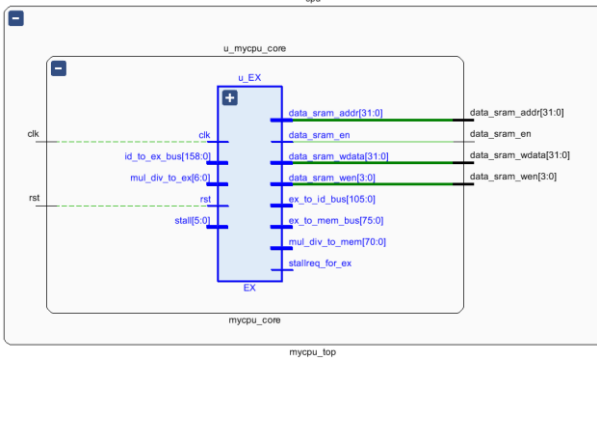
该 cpu 内核共涉及五个流水模块：IF、ID、EX、MEM、WB。cpu 的核心部分与外界的连线主要是与指令访存和数据访存相关，此外还有一些总体控制信号。在核内部除了五段流水的模块还设有 **crtl** 模块负责管理流水线上的 **stall** 请求。

流水段之间通过各种总线进行数据传输，每个流水段都内置了一个寄存器，在时钟上沿对输入的数据进行存储记录。每个流水段内部的主要数据流动都遵循“输入-寄存器-处理模块-输出”的过程。这样，各个流水段就能在时钟信号的控制下，周期性地运行流水线上自己对应的待处理指令。

另外，在本组的设计方案中，为满足乘除法及其相关指令的需求，在各流水段之间单独开辟了一路总线用于传递乘除法相关信息，并且使用组内自行编写的补码移位相加乘法器实现乘法运算。

以上涉及的 CPU 流水段总体设计在下一页附有原理图。

下表列出了各个流水段之间互相交互的 IO 接口

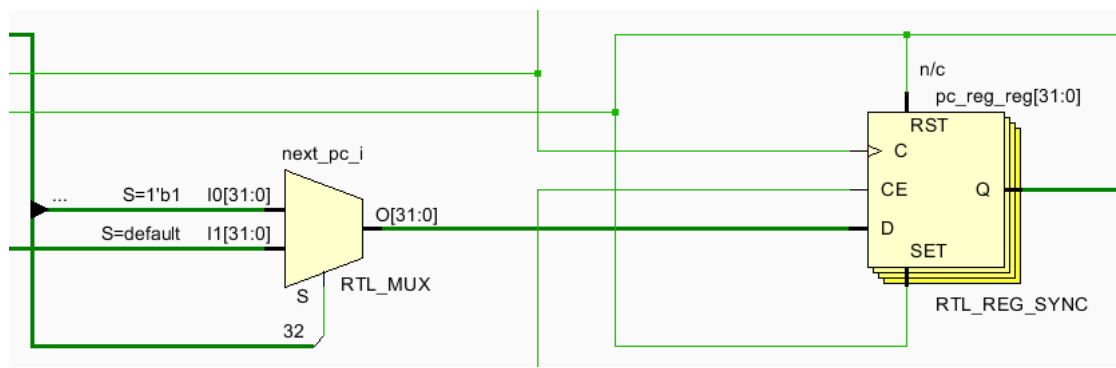
IF		<p>输入:</p> <ul style="list-style-type: none"> br_bus: 跳转指令控制线 stall: 暂停流水线的控制信号 (所有模块均相同) rst: 复位信号 (所有模块均相同) clk: 时钟信号 (所有模块均相同) <p>输出:</p> <ul style="list-style-type: none"> if_to_id_bus: 给 ID 段传递流水信息 inst_sram_addr: 目标访存 (指令) 地址 inst_sram_en: 指令寄存器访问控制 inst_sram_wdata: 指令寄存器写入值 inst_sram_wen: 指令寄存器写入控制
ID		<p>输入:</p> <ul style="list-style-type: none"> if_to_id_bus: 接收 IF 段传递的流水信息 ex_to_id_bus: EX 段 forwarding 回路 mem_to_id_bus: MEM 段 forwarding 回路 wb_to_id_bus: WB 段 forwarding 回路 inst_sram_data: 待执行指令 wb_to_rf_bus: 写回寄存器堆的相关数据 mul_div_to_rf: 乘除法写回寄存器堆的相关数据 <p>输出:</p> <ul style="list-style-type: none"> br_bus: 跳转指令控制线 id_to_ex_bus: 给 EX 段传递流水信息 mul_div_to_ex: 给 EX 段传递乘除法相关信息 stallreq: 发出 stall 请求 new_pc: 传递需要临时更改的 pc 值
EX		<p>输入:</p> <ul style="list-style-type: none"> id_to_ex_bus: 接收 ID 段传递的流水信息 mul_dic_to_ex: 接收乘除法的相关信息 <p>输出:</p> <ul style="list-style-type: none"> data_sram_addr: 目标访存 (数据) 地址 data_sram_en: 访存 (数据) 控制信号 data_sram_wdata: 要写入 ram 的数据 data_sram_wen: ram 读写控制信号 ex_to_id_bus: EX 段 forwarding 回路 ex_to_mem_bus: 给 MEM 段传递流水信息 mul_div_to_mem: 给 MEM 传递乘除法相关信息 stallreq_for_ex: 发出 stall 请求

MEM		<p>输入:</p> <p>data_sram_rdata: 访问 ram 的返回结果</p> <p>ex_to_mem_bus: 接收 EX 段传递的流水信息</p> <p>mul_div_to_mem: 接收乘除法相关信息</p> <p>输出:</p> <p>mem_to_id_bus: MEM 段 forwarding 回路</p> <p>mem_to_wb_bus: 给 WB 传递流水信息</p> <p>mul_div_to_wb: 给 WB 传递乘除法相关信息</p>
WB		<p>输入:</p> <p>mem_to_wb_bus: 接收 MEM 段传的流水线信息</p> <p>mul_div_to_wb: 接收乘除法相关信息</p> <p>输出:</p> <p>四条 debug 线留给测试程序使用</p> <p>wb_to_rf_bus: 寄存器回写的相关数据</p> <p>mul_div_to_rf: 乘除法相关指令回写的数据</p>

流水段设计

IF 段

此段主要负责取址操作，即对指令寄存器（inst_ram）发起访问，使 ID 段能够拿到当前需要执行的目标指令。此外，IF 段还包含了对于分支跳转指令的响应操作，通过一个由分支跳转使能信号控制的选择器来选择访问指令寄存器时的目标地址。



CPU 设计原理图 2

ID 段

此段主要工作是对指令进行译码，并为后续的操作准备好操作数和控制信号。因此，本段的内部集成了译码器模块、寄存器模块、还有多个用于判断和生成控制信号的组合逻辑电路，下面将按照一条指令的译码流程依次进行说明。

译码器

在此实验中一共设置了四个译码器，分别对应一条 32 位指令中的四段位置。

解码方式采用独热编码，即通过 decoder 模块将 n 位操作码对应的十进制数转换成 2^n 位的 one-hot 表示，随后在外部访问独热码的对应位来判断是否激活对应的指令。

例如 ORI 指令的 opcode 为 001101 (13_{10})，将其传入 opcode 段所用的 6bit 解码器，解码器会传出一个 64 位的二进制串 op_d，而 op_d[13] 为 1，其余位均为 0。

这样一来，通过 assign inst_ori = op_d[6'b00_1101];即可在正确的时候激活 ORI 指令。对于需要多个解码器共同判断是否激活的指令，将多个解码器的结果做“&”运算即可。

此处附各条指令的译码实现：

```
1.    assign inst_ori      = op_d[6'b00_1101];
2.    assign inst_lui      = op_d[6'b00_1111];
3.    assign inst_addiu    = op_d[6'b00_1001];
4.    assign inst_beq      = op_d[6'b00_0100];
5.    assign inst_subu     = op_d[6'b00_0000]&func_d[6'b10_0011];
6.    assign inst_j        = op_d[6'b00_0010];
7.    assign inst_jr       = op_d[6'b00_0000]&func_d[6'b00_1000];
8.    assign inst_jal      = op_d[6'b00_0011];
9.    assign inst_addu     = op_d[6'b00_0000]&func_d[6'b10_0001];
10.   assign inst_sll      = op_d[6'b00_0000]&func_d[6'b00_0000];
11.   assign inst_or       = op_d[6'b00_0000]&func_d[6'b10_0101];
12.   assign inst_xor      = op_d[6'b00_0000]&func_d[6'b10_0110];
13.   assign inst_sltu     = op_d[6'b00_0000]&func_d[6'b10_1011];
14.   assign inst_sltiu    = op_d[6'b00_1011];
15.   assign inst_bne      = op_d[6'b00_0101];
16.   assign inst_slt      = op_d[6'b00_0000]&func_d[6'b10_1010];
17.   assign inst_slti     = op_d[6'b00_1010];
18.   assign inst_add      = op_d[6'b00_0000]&func_d[6'b10_0000];
19.   assign inst_addi     = op_d[6'b00_1000];
20.   assign inst_sub      = op_d[6'b00_0000]&func_d[6'b10_0010];
21.   assign inst_and      = op_d[6'b00_0000]&func_d[6'b10_0100];
22.   assign inst_andi     = op_d[6'b00_1100];
23.   assign inst_nor      = op_d[6'b00_0000]&func_d[6'b10_0111];
24.   assign inst_xori     = op_d[6'b00_1110];
25.   assign inst_sllv     = op_d[6'b00_0000]&func_d[6'b00_0100];
26.   assign inst_sra      = op_d[6'b00_0000]&func_d[6'b00_0011];
27.   assign inst_srav     = op_d[6'b00_0000]&func_d[6'b00_0111];
28.   assign inst_srl      = op_d[6'b00_0000]&func_d[6'b00_0010];
29.   assign inst_srlv     = op_d[6'b00_0000]&func_d[6'b00_0110];
30.   assign inst_bgez     = op_d[6'b00_0001]&rt_d[5'b00_001];
31.   assign inst_bgtz     = op_d[6'b00_0111]&rt_d[5'b00_000];
32.   assign inst_blez     = op_d[6'b00_0110]&rt_d[5'b00_000];
33.   assign inst_bltz     = op_d[6'b00_0001]&rt_d[5'b00_000];
34.   assign inst_bltzal   = op_d[6'b00_0001]&rt_d[5'b10_000];
35.   assign inst_bgezal   = op_d[6'b00_0001]&rt_d[5'b10_001];
36.   assign inst_jalr     = op_d[6'b00_0000]&func_d[6'b00_1001];
37.   assign inst_div      = op_d[6'b00_0000]&func_d[6'b01_1010];
38.   assign inst_divu     = op_d[6'b00_0000]&func_d[6'b01_1011];
39.   assign inst_mult     = op_d[6'b00_0000]&func_d[6'b01_1000];
40.   assign inst_multu    = op_d[6'b00_0000]&func_d[6'b01_1001];
41.   assign inst_mfhi     = op_d[6'b00_0000]&func_d[6'b01_0000];
42.   assign inst_mflo     = op_d[6'b00_0000]&func_d[6'b01_0010];
```



```

43.    assign inst_mthi    = op_d[6'b00_0000]&func_d[6'b01_0001];
44.    assign inst_mtlo    = op_d[6'b00_0000]&func_d[6'b01_0011];
45.    assign inst_lb      = op_d[6'b10_0000];
46.    assign inst_lbu     = op_d[6'b10_0100];
47.    assign inst_lh      = op_d[6'b10_0001];
48.    assign inst_lhu     = op_d[6'b10_0101];
49.    assign inst_sb      = op_d[6'b10_1000];
50.    assign inst_sh      = op_d[6'b10_1001];
51.    assign inst_sw      = op_d[6'b10_1011];
52.    assign inst_lw      = op_d[6'b10_0011];

```

寄存器的读写和数据相关

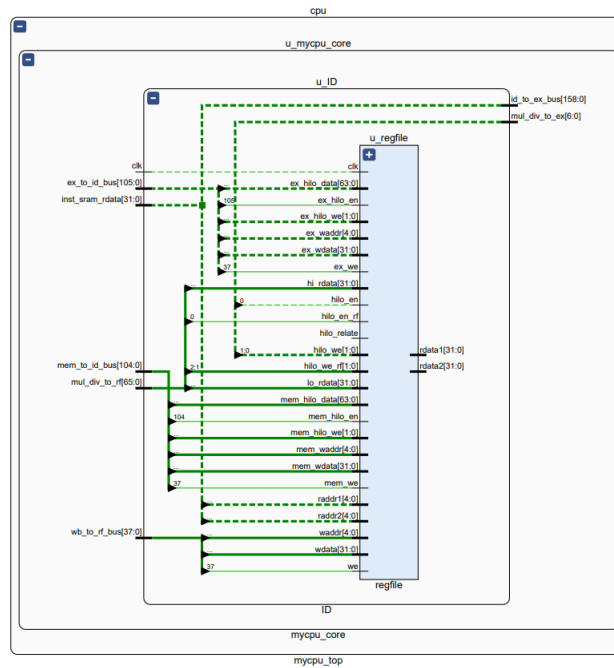
ID段内部集成的寄存器模块包含了32个通用寄存器和服务于乘除法相关指令的hilo寄存器，由于原理相同，在此仅以通用寄存器的为例进行说明。

关于寄存器的写回，这一操作总是发生在时钟周期的上沿，它受使能信号和目标地址信号共同控制，只有地址不为0并且使能信号被激活时才能向目标寄存器写入wb段传来的数据。

关于寄存器的读取，我们设定无论该指令是否需要取两个操作数，寄存器都会始终按照当前指令的rs和rt段所表示的地址输出对应寄存器内的数据，至于是否使用将在ID里进行选择，我们在下一部分中会进行说明。

流水线所涉及的数据相关问题我们也一并归纳到寄存器模块内部解决。

优先级 ex>mem>wb	EX				MEM				WB			
是否写寄存器	Y		N		Y		N		Y		N	
目标寄存器是否与 ID 段指令要读取的寄存器相同	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
是否采用 forwarding 的值	Y	N	N		Y	N	N		Y	N	N	



```

1.         reg [31:0] reg_array [31:0];
2.         reg [31:0] hilo_reg [1:0]; //hi,lo
3.         // write
4.         always @ (posedge clk) begin
5.             if (we && waddr!=5'b0) begin
6.                 reg_array[waddr] <= wdata;
7.             end
8.             if (hilo_we_rf == 2'b11 && hilo_en_rf)begin
9.                 hilo_reg[0]<=lo_rdata;
10.                hilo_reg[1]<=hi_rdata;
11.            end
12.            if (hilo_we_rf == 2'b10 && hilo_en_rf)begin
13.                hilo_reg[1]<=wdata;
14.            end
15.            if (hilo_we_rf == 2'b01 && hilo_en_rf)begin
16.                hilo_reg[0]<=wdata;
17.            end
18.        end
19.        reg [31:0] temp [1:0];
20.        //assign rdata1 = (raddr1 == 5'b0) ? 32'b0 : reg_array[raddr1];
21.        always @ (*) begin
22.            if((ex_we ==1'b1) && (raddr1 ==ex_waddr) && ~hilo_relate)
23.                temp[0] <= ex_wdata;
24.            else if((mem_we ==1'b1) && (raddr1 ==mem_waddr) && ~hilo_relate)
25.                temp[0] <= mem_wdata;
26.            else if((we ==1'b1) && (raddr1 == waddr) && ~hilo_relate)
27.                temp[0] <= wdata;

```

```

28.     else if (raddr1 != 5'b0)
29.         temp[0] <= reg_array[raddr1];
30.     else if(ex_hilo_we[1] && ~hilo_we[1] && ex_hilo_en && ~hilo_en && hilo
        _relate)
31.         temp[0] <= ex_hilo_we==2'b11 ? ex_hilo_data[63:32]:ex_wdata;
32.     else if(ex_hilo_we[0] && ~hilo_we[0] && ex_hilo_en && ~hilo_en && hilo
        _relate)
33.         temp[0] <= ex_hilo_we==2'b11 ? ex_hilo_data[31:0]:ex_wdata;
34.     else if(mem_hilo_we[1] && ~hilo_we[1] && mem_hilo_en && ~hilo_en && hi
        lo_relate)
35.         temp[0] <= mem_hilo_we==2'b11 ? mem_hilo_data[63:32]:mem_wdata;
36.     else if(mem_hilo_we[0] && ~hilo_we[0] && mem_hilo_en && ~hilo_en && hi
        lo_relate)
37.         temp[0] <= mem_hilo_we==2'b11 ? mem_hilo_data[31:0]:mem_wdata;
38.     else if(hilo_we_rf[1] && ~hilo_we[1] && hilo_en_rf && ~hilo_en && hilo
        _relate)
39.         temp[0] <= hilo_we_rf==2'b11 ? hi_rdata:wdata;
40.     else if(hilo_we_rf[0] && ~hilo_we[0] && hilo_en_rf && ~hilo_en && hilo
        _relate)
41.         temp[0] <= hilo_we_rf==2'b11 ? lo_rdata:wdata;
42.     else if (~hilo_we[0] && ~hilo_en && hilo_relate)
43.         temp[0] <= hilo_reg[0];
44.     else if (~hilo_we[1] && ~hilo_en && hilo_relate)
45.         temp[0] <= hilo_reg[1];
46.     else
47.         temp[0] <= 32'b0;
48. end
49. assign rdata1 = temp[0];
50. // read out2
51. //assign rdata2 = (raddr2 == 5'b0) ? 32'b0 : reg_array[raddr2];
52. always @ (*) begin
53.     if((ex_we == 1'b1) && (raddr2 == ex_waddr))
54.         temp[1] <= ex_wdata;
55.     else if((mem_we == 1'b1) && (raddr2 == mem_waddr))
56.         temp[1] <= mem_wdata;
57.     else if((we == 1'b1) && (raddr2 == waddr))
58.         temp[1] <= wdata;
59.     else if (raddr2 != 5'b0)
60.         temp[1] <= reg_array[raddr2];
61.     else
62.         temp[1] <= 32'b0;
63. end
64.     assign rdata2 = temp[1];

```

操作数和操作类型的选择

在我们的设计中，ID 段在这一步不会直接选出操作数的具体数据并传给 EX 段，而是得出选择信号，在 EX 段中再根据选择信号取操作数以及进行预处理。

这里我们设置了 sel_alu_src1[2:0]和 sel_alu_src[3:0]两个信号分别控制两个操作数的来源。

```
// rs(base) to reg1
assign sel_alu_src1[0] = inst_ori | inst_addiu | inst_subu | inst_addu | inst_or | inst_sw | inst_lw | inst_xor | inst_sltu | inst_slt
                        | inst_slti | inst_sltiu | inst_bne | inst_beq | inst_add | inst_addi | inst_sub | inst_and | inst_andi | inst_nor
                        | inst_xori | inst_sllv | inst_srav | inst_srlv | inst_jalr | inst_div | inst_divu | inst_mult | inst_multu
                        | inst_mfhi | inst_mflo | inst_mthi | inst_mtlo | inst_lb | inst_lbu | inst_lh | inst_lhu | inst_sb | inst_sh
                        | inst_lsa;

// pc to reg1
assign sel_alu_src1[1] = 1'b0;

// sa_zero_extend to reg1
assign sel_alu_src1[2] = inst_sll | inst_sra | inst_srl ;

// rt to reg2
assign sel_alu_src2[0] = inst_subu | inst_addu | inst_sll | inst_or | inst_xor | inst_sltu | inst_slt | inst_bne | inst_beq
                        | inst_add | inst_sub | inst_and | inst_nor | inst_sllv | inst_sra | inst_srav | inst_srl | inst_srlv
                        | inst_div | inst_divu | inst_mult | inst_multu | inst_lsa;

// imm_sign_extend to reg2
assign sel_alu_src2[1] = inst_lui | inst_addiu | inst_sw | inst_lw | inst_slti | inst_sltiu | inst_addi | inst_lb | inst_lbu | inst_lh | inst_lhu |
inst_sb | inst_sh;

// 32'b8 to reg2
assign sel_alu_src2[2] = 1'b0;

// imm_zero_extend to reg2
assign sel_alu_src2[3] = inst_ori | inst_andi | inst_xori;
```

同样的，ALU 对应的十二种操作也会在 ID 段被对应的指令激发，在 EX 段调用 ALU 模块时才会激发某一种操作单元。

```
assign op_add = inst_addiu | inst_sw | inst_jal | inst_addu | inst_sw | inst_lw | inst_add | inst_addi | inst_bltzal | inst_bgezal | inst_jalr
                | inst_mfhi | inst_mflo | inst_mthi | inst_mtlo | inst_lb | inst_lbu | inst_lh | inst_lhu | inst_sb | inst_sh | inst_lsa;
```

```

assign op_sub = inst_subu | inst_sub;
assign op_slt = inst_slt | inst_slti;
assign op_sltu = inst_sltu | inst_sltiu;
assign op_and = inst_and | inst_andi;
assign op_nor = inst_nor;
assign op_or = inst_ori | inst_or;
assign op_xor = inst_xor | inst_xori;
assign op_sll = inst_sll | inst_sllv;
assign op_srl = inst_srl | inst_srlv;
assign op_sra = inst_sra | inst_srav;
assign op_lui = inst_lui;
assign alu_op = {op_add, op_sub, op_slt, op_sltu,
                op_and, op_nor, op_or,
                op_xor,
                op_sll, op_srl, op_sra, op_lui};

//mul_div inst
wire [3:0] mul_div_inst_onehot;
assign mul_div_inst_onehot = inst_div ? 4'b1000 :
                                inst_divu ? 4'b0100 :
                                inst_mult ? 4'b0010 :
                                inst_multu ? 4'b0001 :
                                inst_mfhi ? 4'b1001 :
                                inst_mflo ? 4'b0110 :
                                inst_mthi ? 4'b1010 :
                                inst_mtlo ? 4'b0101 : 4'b0000;

// hilo store enable
assign hilo_we = inst_div | inst_divu | inst_mult | inst_multu ? 2'b11 :
                inst_mthi ? 2'b10 :
                inst_mtlo ? 2'b01 :
                inst_mfhi ? 2'b01 :
                inst_mflo ? 2'b10 : 2'b00;
assign hilo_en = inst_div | inst_divu | inst_mult | inst_multu | inst_mthi | inst_mtlo ;//1 write enable
assign hilo_relate=inst_mfhi | inst_mflo;
assign mul_div_to_ex={ mul_div_inst_onehot, hilo_we, hilo_en};

```

指令回写控制

与之前的操作数选择类似，指令写回时的目标寄存器也需要进行选择，有些指令的目标写回地址由指令的 rt 段表示，有的由 rd 段表示，有的还需要写回特定的寄存器。因此，我们需要在每条指令的 ID 段进行解码后就设置好它的写回控制信号，包括使能信号和目标地址。

```
// regfile store enable
assign rf_we = inst_ori | inst_lui | inst_addiu | inst_subu | inst_jal | inst_addu | inst_sll | inst_or | inst_lw | inst_xor
              | inst_sltu | inst_slt | inst_slti | inst_sltiu | inst_add | inst_addi | inst_sub | inst_and | inst_andi
              | inst_nor | inst_xori | inst_sllv | inst_sra | inst_srav | inst_srl | inst_srlv | inst_bltzal | inst_bgezal
              | inst_jalr | inst_mfhi | inst_mflo | inst_lb | inst_lbu | inst_lh | inst_lhu | inst_lsa;

// store in [rd]
assign sel_rf_dst[0] = inst_subu | inst_addu | inst_sll | inst_or | inst_xor | inst_sltu | inst_slt | inst_add | inst_sub | inst_and
                    | inst_nor | inst_sllv | inst_sra | inst_srav | inst_srl | inst_srlv | inst_jalr | inst_mfhi | inst_mflo | inst_lsa;

// store in [rt]
assign sel_rf_dst[1] = inst_ori | inst_lui | inst_addiu | inst_lw | inst_slti | inst_sltiu | inst_addi | inst_andi | inst_xori | inst_lb | inst_lbu |
inst_lh | inst_lhu;

// store in [31]
assign sel_rf_dst[2] = inst_jal | inst_bltzal | inst_bgezal;

// sel for regfile address
assign rf_waddr = {5{sel_rf_dst[0]}} & rd
                  | {5{sel_rf_dst[1]}} & rt
                  | {5{sel_rf_dst[2]}} & 32'd31;
```

分支跳转和暂停请求

对于分支跳转请求，主要的控制信号有使能信号和目标地址。

使能信号和之前的回写寄存器操作一样，由对应的指令直接激发即可，有些条件跳转请求则需要在此之前做出判断来判断 `br_enable` 是否可以被激活。而跳转的目标地址则一般是由当前指令的某一段进行扩展并与 `pc` 运算所得，例如 `inst_bne` 对应的目的地址为 $(pc_plus_4 + \{14\{inst[15]\}, inst[15:0], 2'b0\})$ 。此外，有的指令还要求将当前 `pc` 保留到 31 号寄存器中，因此要同时激发当前指令的回写相关信号。

在涉及到访存相关指令时，有一些数据相关问题是采用 forwarding 技术也无法解决的，这时就需要我们在流水线中插入气泡（发出暂停请求）来等待访存阶段的数据成功取回。具体地，当前 EX 段指令的访存使能信号被激活且该指令的回写目标寄存器为当前 ID 段指令需要访问的操作数所在的寄存器，则激活暂停请求信号。CTRL 模块受到 ID 段的暂停请求后会将前三段流水线暂停一个周期，它们仍然按照各自内部的段间寄存器所存储的信息运行，而后两段流水线会将当前的工作继续做完。这样，一个周期过后，mem 段会从内存受到程序所期望的数据，并通过 forwarding

传给 ID 段，再利用数据相关通路实现正确运行。

```
assign rs_eq_rt = (rdata1 == rdata2);
assign rs_ge_z = (rdata1[31]==0);
assign rs_gt_z = (rdata1>32'h0 & rdata1[31]!=1);
assign rs_le_z = (rdata1==32'h0 | rdata1[31]==1);
assign rs_lt_z = (rdata1[31]!=1);
assign br_e = (inst_beq & rs_eq_rt) | (inst_bne & ~rs_eq_rt) | inst_jr | inst_jal | inst_j | inst_jalr
              | (inst_bgez & rs_ge_z) | (inst_bgtz & rs_gt_z) | (inst_blez & rs_le_z) | (inst_bltz & rs_lt_z)
              | (inst_bltzal & rs_lt_z) | (inst_bgezal & rs_ge_z);
assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_bne ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_jal ? ({pc_plus_4[31:28],inst[25:0],2'b0}) :
                  inst_j    ? ({pc_plus_4[31:28],inst[25:0],2'b0}) :
                  inst_jr ? rdata1 :
                  inst_bgez ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_bgtz ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_blez ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_bltz ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_bltzal ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_bgezal ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
                  inst_jalr ? rdata1 : 32'b0;

assign br_bus = {
    br_e,
    br_addr
};
```

EX 段

此段主要负责对操作数进行运算处理，内部集成了 ALU 运算模块、乘法器、除法器以及对数据寄存器（data_ram）进行访问控制的一些组合逻辑电路。

对于使用 ALU 模块的操作，首先会进行操作数的选取和预处理（扩展至相同位数），随后传入 ALU 模块进行运算，最后的运算结果随着 ex_to_mem 总线继续流水运行。

乘法器模块

我们自己设计了一个 32 周期的补码移位相加乘法器，电路原理图附在下页。模块输入信号包括两个进行运算的数，有无符号乘法信号等。输出包括完成信号 done 和最终的 64 位结果。

对于有符号乘法，我们的思路是将有符号乘法先转换为无符号乘法，无符号乘法计算完后再对符号进行转换判断。首先将负数都转换为正数

```
if(mul_signed==1'b1&&ain[31]==1'b1) begin
    multiplier<=~ain+1;
end
```

完成计算后，最后判断两个操作数的符号(最高位)决定结果的符号。

```
result<=((ain[31]^bin[31])&mul_signed)?(~result_r+1):result_r;
```

无符号乘法器采用时序逻辑控制，每个时钟周期进行加法和移位操作。核心包括一个计数器，一个加法器和两个移位器。

计数器 $i=0$ 时，进行数据符号的变换和载入。接着，每一个时钟周期，在时序控制逻辑内进行两个操作数与结果的更新，并在时序逻辑外进行数据移位和相加的并行操作。

```
assign multiplicand_next={multiplicand[63:0],1'b0};    //被乘数左移一位

assign multiplier_next={1'b0,multiplier[31:1]};        //乘数右移一位

assign result_r_next = multiplier[0]? (multiplicand+result_r):result_r;//乘法结果的更新。
```

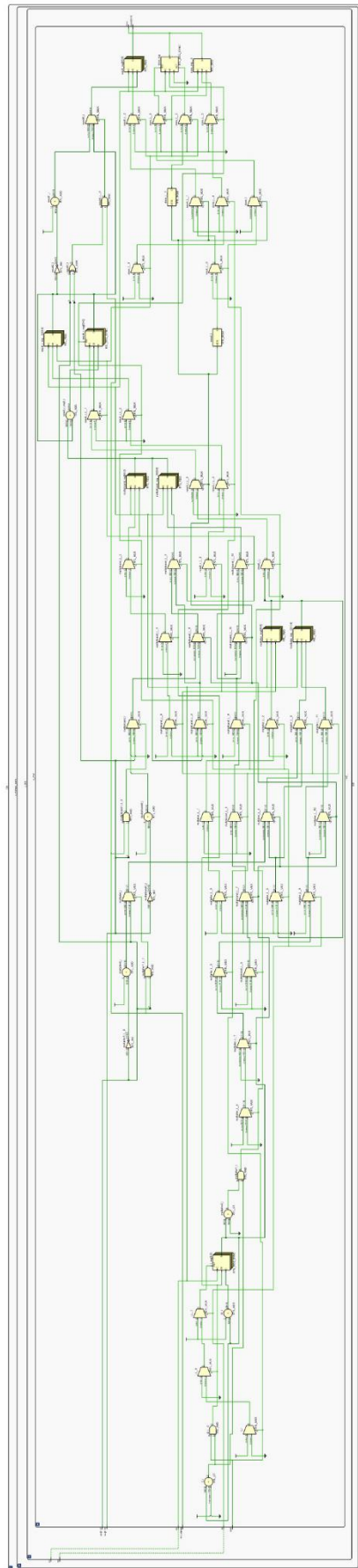
由于第 0 个周期进行数据的加载，对于有符号乘法， $i=32$ 时可以完成计算，而对于无符号乘法， $i=33$ 时可以完成计算。

```
else if(i==6'd33)    //统一在 i=33 时完成计算;

begin
    result<=((ain[31]^bin[31])&mul_signed)?(~result_r+1):result_r;
    done<=1'b1; end
```

注意乘法计算中流水线需要 stall，stall 信号通过乘法 done 信号控制:

```
assign stallreq_for_mul=~mul_done&&(inst_mult|inst_multu);
```

CPU 设计原理图 3

访存控制

本实验与访存相关的指令一共有八条，由于 ID 段在解码时已经获得了要执行的指令名，所以后续不希望再对 inst 进行解码，而是在 ID 段利用 4bit 线宽的 data_ram_wen 对这八条指令进行编码，当 EX 段和 MEM 段需要获取当前是哪种存储指令时就直接按照 wen 码比对即可。

编码规则如下：[store 1/load 0 , signed 1/unsigned 0 , 2 bit? 1:0, 4 bit? 1:0]

MIIPS 指令集中关于 store 的操作指令一共有三条，对应操作如下：

指令	对应 wen 编码	写入的位置				
SB	1100	目标地址后两位	00	01	10	11
		字节位置	1	2	3	4
		wen 控制信号	0001	0010	0100	1000
SH	1110	目标地址后两位		00		10
		写入字节位置		1、2		3、4
		wen 控制信号		0011		1100
SW	1101	所有字节都写入				

ID 段：

```
// load and store enable
```

```
assign data_ram_en = inst_sw | inst_lw | inst_lb | inst_lbu | inst_lh | inst_lhu | inst_sb | inst_sh;
```

```
// write enable [s/l,signed/unsigned,2 bit?,4 bit?]
```

```
assign data_ram_wen = inst_lb ? 4'b0100 :
```

```
inst_lbu ? 4'b0000 :
```

```
inst_lh ? 4'b0110 :
```

```
inst_lhu ? 4'b0010 :
```

```
inst_lw ? 4'b0101 :
```

```
inst_sb ? 4'b1100 :
```

```
inst_sh ? 4'b1110 :
```

```
inst_sw ? 4'b1101 : 4'b0;
```

EX 段:

```
wire inst_sb, inst_sh, inst_sw;
assign inst_sb  = data_ram_wen == 4'b1100 ? 1:0;
assign inst_sh  = data_ram_wen == 4'b1110 ? 1:0;
assign inst_sw  = data_ram_wen == 4'b1101 ? 1:0;
assign data_sram_en=data_ram_en;
assign data_sram_wen= inst_sw ? 4'b1111:
    inst_sh && ex_result[1:0]==2'b00 ? 4'b0011:
    inst_sh && ex_result[1:0]==2'b10 ? 4'b1100:
    inst_sb && ex_result[1:0]==2'b00 ? 4'b0001:
    inst_sb && ex_result[1:0]==2'b01 ? 4'b0010:
    inst_sb && ex_result[1:0]==2'b10 ? 4'b0100:
    inst_sb && ex_result[1:0]==2'b11 ? 4'b1000: 4'b0;
assign data_sram_addr=alu_result;
assign data_sram_wdata = inst_sh && ex_result[1:0]==2'b10 ? {rf_rdata2[15:0],16'b0}:
    inst_sb && ex_result[1:0]==2'b01 ? {16'b0,rf_rdata2[7:0],8'b0}:
    inst_sb && ex_result[1:0]==2'b10 ? {8'b0,rf_rdata2[7:0],16'b0}:
    inst_sb && ex_result[1:0]==2'b11 ? {rf_rdata2[7:0],24'b0}: rf_rdata2;
```

MEM 段:

```
//load from ram
wire inst_lb, inst_lbu, inst_lh, inst_lhu, inst_lw;
assign inst_lb  = data_ram_wen == 4'b0100 ? 1:0;
assign inst_lbu = data_ram_wen == 4'b0000 ? 1:0;
assign inst_lh  = data_ram_wen == 4'b0110 ? 1:0;
assign inst_lhu = data_ram_wen == 4'b0010 ? 1:0;
assign inst_lw  = data_ram_wen == 4'b0101 ? 1:0;
wire [31:0] byte_ram_data;
wire [31:0] half_ram_data;
assign byte_ram_data = inst_lb  && ex_result[1:0]==2'b00 ? {{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}:
    inst_lb  && ex_result[1:0]==2'b01 ? {{24{data_sram_rdata[15]}},data_sram_rdata[15:8]}:
    inst_lb  && ex_result[1:0]==2'b10 ? {{24{data_sram_rdata[23]}},data_sram_rdata[23:16]}:
    inst_lb  && ex_result[1:0]==2'b11 ? {{24{data_sram_rdata[31]}},data_sram_rdata[31:24]}:
    inst_lbu && ex_result[1:0]==2'b00 ? {24'b0,data_sram_rdata[7:0]}:
    inst_lbu && ex_result[1:0]==2'b01 ? {24'b0,data_sram_rdata[15:8]}:
    inst_lbu && ex_result[1:0]==2'b10 ? {24'b0,data_sram_rdata[23:16]}:
    inst_lbu && ex_result[1:0]==2'b11 ? {24'b0,data_sram_rdata[31:24]}: 32'b0;
assign half_ram_data = inst_lh  && ex_result[1:0]==2'b00 ? {{16{data_sram_rdata[15]}},data_sram_rdata[15:0]}:
    inst_lh  && ex_result[1:0]==2'b10 ? {{16{data_sram_rdata[31]}},data_sram_rdata[31:16]}:
    inst_lhu && ex_result[1:0]==2'b00 ? {16'b0,data_sram_rdata[15:0]}:
    inst_lhu && ex_result[1:0]==2'b10 ? {16'b0,data_sram_rdata[31:16]}: 32'b0;
assign rf_wdata = sel_rf_res ? mem_result :
```

```

data_ram_en && inst_lw ? data_sram_rdata :
data_ram_en && (inst_lb | inst_lbu) ? byte_ram_data:
data_ram_en && (inst_lh | inst_lhu) ? half_ram_data: ex_result;

```

MEM 段

此段主要负责接收和处理访存得到的数据，并将其传给 WB 段。

MIIPS 指令集中关于 load 的操作指令一共有五条，对应操作如下：

指令	对应 wen 编码	读取的数据	读取的位置
LB	0100	单字节有符号扩展	目标地址后两位 00、01、10、11 分别对应从低位开始第 1、2、3、4 字节
LBU	0000	单字节无符号扩展	
LH	0110	半字有符号扩展	目标地址后两位 00、10 分别对应低位两个字节和高位两个字节
LHU	0010	半字无符号扩展	
LW	0101	字	所有字节都取走

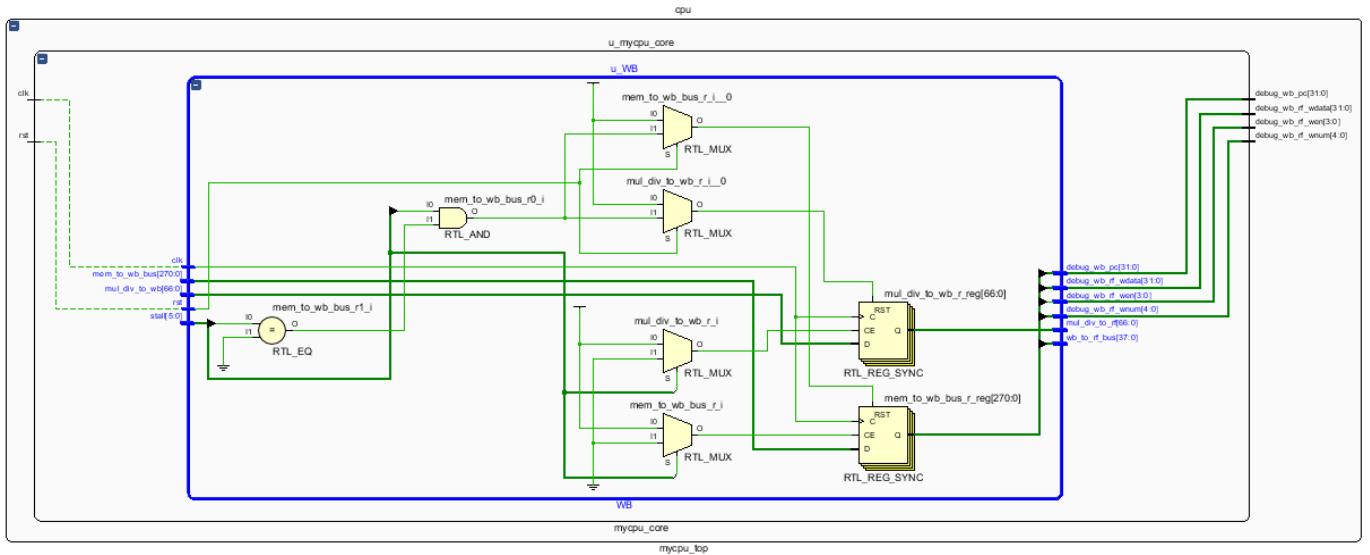
按照上述规则以及 MEM 段接收到的 wen 编码，我们即可在 MEM 段从读入的数据中选出需要的部分，按照指令要求的形式写回寄存器。

以 LB 指令的一种情况为例进行说明：

- (1) assign inst_lb = data_ram_wen == 4'b0100 ? 1:0;
- (2) assign byte_ram_data = inst_lb && ex_result[1:0] == 2'b00 ?
{{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}: 32'b0;
- (3) assign rf_wdata = data_ram_en && (inst_lb | inst_lbu) ? byte_ram_data : ex_result;

WB 段

此段主要负责对一条指令的操作进行收尾整合，把需要写回寄存器的数据回传给位于 ID 段的寄存器模块中。此外，WB 段在回写之前还为程序 debug 留出了接口。



cpu 设计原理图 4

```

1.    reg [`MEM_TO_WB_WD-1:0] mem_to_wb_bus_r;
2.    reg [66:0] mul_div_to_wb_r;
3.    always @ (posedge clk) begin
4.        if (rst) begin
5.            mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
6.            mul_div_to_wb_r <= 66'b0;
7.        end
8.        // else if (flush) begin
9.        //     mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
10.       // end
11.       else if (stall[4]==`Stop && stall[5]==`NoStop) begin
12.           mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
13.           mul_div_to_wb_r <= 66'b0;
14.       end
15.       else if (stall[4]==`NoStop) begin
16.           mem_to_wb_bus_r <= mem_to_wb_bus;
17.           mul_div_to_wb_r <= mul_div_to_wb;
18.       end
19.   end
20.   wire [31:0] wb_pc;
21.   wire rf_we;
22.   wire [4:0] rf_waddr;
23.   wire [31:0] rf_wdata;
24.   assign {
25.       wb_pc,
26.       rf_we,
27.       rf_waddr,

```

```

28.         rf_wdata
29.     } = mem_to_wb_bus_r;
30.     // assign wb_to_rf_bus = mem_to_wb_bus_r[`WB_TO_RF_WD-1:0];
31.     assign wb_to_rf_bus = {
32.         rf_we,
33.         rf_waddr,
34.         rf_wdata
35.     };
36.
37.     wire [31:0] hi_rdata;
38.     wire [31:0] lo_rdata;
39.     wire hi_we;
40.     wire lo_we;
41.     wire hilo_en;
42.     assign {
43.         hilo_en,
44.         hi_rdata,
45.         lo_rdata,
46.         hi_we,
47.         lo_we
48.     } = mul_div_to_wb_r;
49.     assign mul_div_to_rf = {
50.         hi_rdata,
51.         lo_rdata,
52.         hi_we,
53.         lo_we,
54.         hilo_en
55.     }
56.     assign debug_wb_pc = wb_pc;
57.     assign debug_wb_rf_wen = {4{rf_we}};
58.     assign debug_wb_rf_wnum = rf_waddr;
59.     assign debug_wb_rf_wdata = rf_wdata;

```

总结与改进

孙平炜：这次自己动手做 CPU 的系统实验使我对课上所学的理论知识有了更加深入地理解和认识。通过自己的亲身实践操作，我对 CPU 各个流水段之间的工作模式以及控制方式掌握的也更加全面，做到了理论与实践相结合的学习效果。但是我们毕竟还是第一次进行尝试，对硬件和编程语言的掌握还不够全面，设计出的 CPU 体系也还有进一步改进的空间。

孙翰文：这次 cpu lab 对我来说是一次新的尝试，接触了面向硬件的编程语言，编程思想，掌握了面向波形图的调试方法。同时，亲自设计和编写 cpu 让我对课上学习的知识有了新的理解，尤其是数据在流水线间各个模块的流动和运算方式。

刘寒：这次的实验对我来说是一次很好的锻炼机会，通过亲手编写硬件相关的控制程序，第一次接触到硬件语言 verilog 和环境 ivivado，这次计算机系统实验课使得我对 cpu 的架构和五级流水线有了更深刻的认识，同时是对所学课上内容的很好补充，并且进一步熟悉了 mips 指令，受益良多。

参考资料

- 【1】 自己动手做 cpu_雷思磊
- 【2】 [Verilog 教程 | 菜鸟分类 | 菜鸟教程 \(runoob.com\)](#)
- 【3】 <https://github.com/fluctlight001/SampleCPU.git>