中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009915025　姓名：马月骁 专业：计算机科学与技术

实验序号：2　　实验名称：单周期处理器设计

一、 逻辑电路结构与仿真波形的截图及说明

单周期处理器实验中共完成四个模块（Register File，ALU，Shifter 以及 MIPS CPU）的设计与编写。其中 Register File 模块直接复用实验一中的 Register File 代码，ALU 模块在复用实验一中 ALU 代码之外添加了 xor、nor、sltu 三类操作及其对应代码，Shifter 与 MIPS CPU 模块在本次实验中完成设计与编写。具体代码见 Github 仓库。

整体逻辑电路结构图如下：

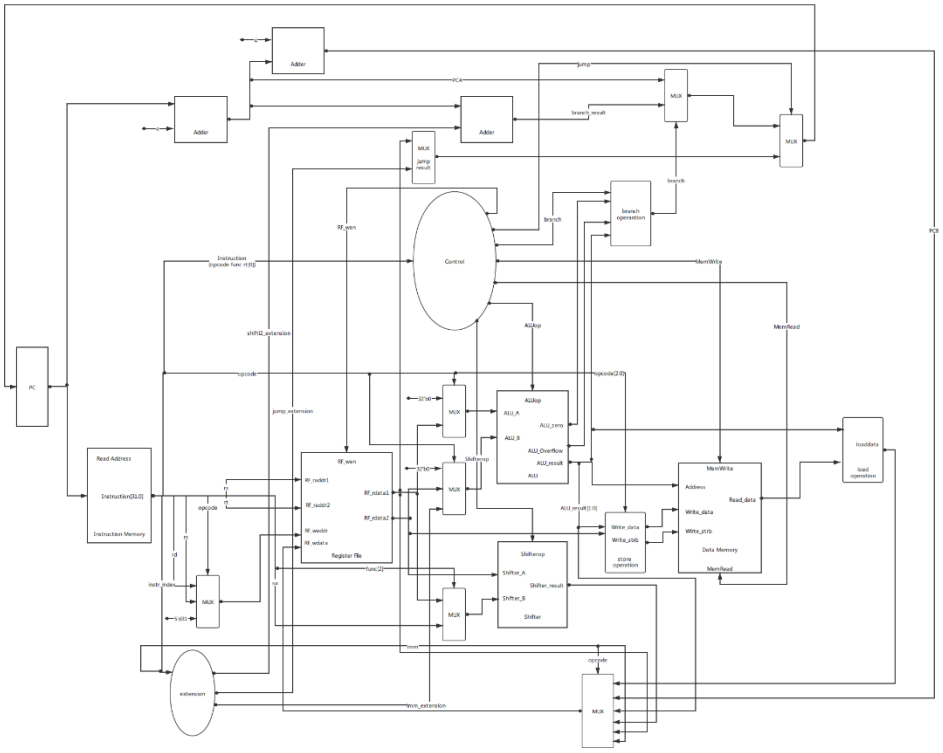


图 1 整体逻辑电路结构图

注：结构图中并未包含时序电路信号：clk，rst。部分图中未注明信号如下：

```
opcode = Instruction[31:26];
rs = Instruction[25:21];
rt = Instruction[20:16];
rd = Instruction[15:11];
sa = Instruction[10:6];
func = Instruction[5:0];
imm = Instruction[15:0];
instr_index = Instruction[25:0];
```

本次单周期处理器设计实验的逻辑电路依据信号上的相关性大致可以分为五个部分：Control 部分（根据指令生成控制信号），PC 部分（根据控制信号确定 PC 值），Register File 部分（根据指令与控制信号确定 Register File 的输入输出信号），Calculator 部分（包括 ALU 与 Shifter）以及 Memory 部分（主要为 Data Memory，根据指令与控制信号确定输入输出）

在具体实现时，为了便于编写代码，将 Control 部分相关的逻辑电路依据信号上的相关性分散至其余四个部分中分别实现相关控制信号生成。

1．PC 部分：

PC 部分逻辑电路图如下：

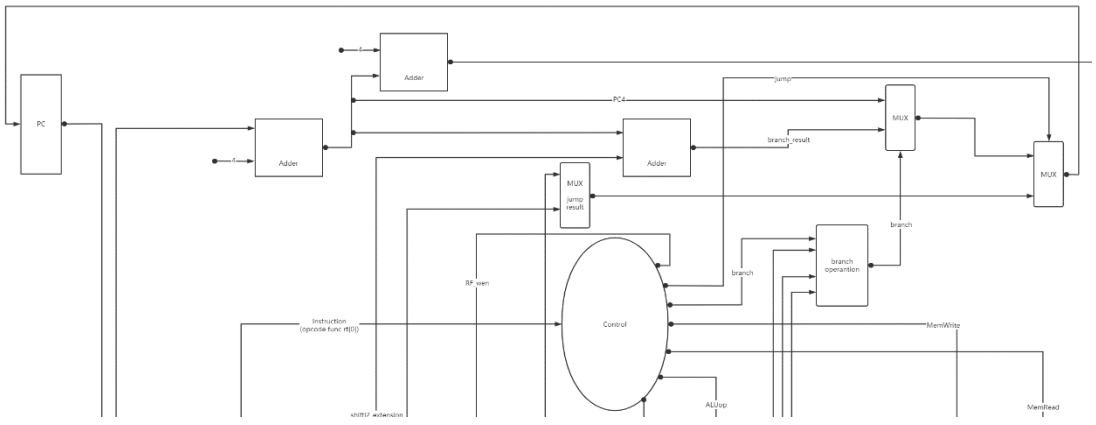


图 2 PC 部分逻辑电路图

对应代码段：

```
        //PC singals
            assign PC4 = PC + 4;
            assign PC8 = PC + 8;

            //define branch siganls
            //beq, bne, blez, bgtz: opcode[5:2] = 0001 , opcode[1:0] = 00, 01, 10, 11
            //bltz, bgez:                : opcode[5:0] = 000001,    rt[0] = 0, 1
            assign branch = (((~|opcode[5:3]) & opcode[2]) & (((opcode[1:0] == 2'b00) &
ALU_zero)                                                    |
                                                                ((opcode[1:0] == 2'b01) &
~ALU_zero)                                                   |
                                                                ((opcode[1:0] == 2'b10) &
(ALU_zero | (ALU_Overflow ^ ALU_result[31]))) |
                                                                ((opcode[1:0] == 2'b11) &
(~ALU_zero & (ALU_Overflow ^ (~ALU_result[31]))))
                                                               )
                                    ) |
                                   (((~|opcode[5:1]) & opcode[0]) & ((~rt[0] & (ALU_Overflow ^
ALU_result[31])) |
                                                                (rt[0] & (ALU_Overflow ^
(~ALU_result[31])))
                                                               )
                                   ) ;
            assign branch_result = PC4 + shiftl2_extension;

            //define jump signals
            //R-Type jump: opcode[5:0] = 000000, {func[5:3], func[1]} == 0010
            //J-type          : opcode[5:1] = 00001
            assign jump = ((~|opcode) & ({func[5:3], func[1]} == 4'b0010)) | ((~|opcode[5:2]) &
opcode[1]);
            assign jump_result = ({32{(~|opcode) & ({func[5:3], func[1]} == 4'b0010)}} &
RF_rdata1)            |
                                        ({32{(~|opcode[5:2]) & opcode[1]}}
& jump_extension)    ;

            always @(posedge clk) begin
                if(rst) begin
                    PC <= 32'b0;
                end
                else begin
                    PC <= branch ? branch_result : jump ? jump_result : PC4;
                end
            end
```

主要实现功能：PC 寄存器正常“+4”取得下一指令地址；branch 或 jump 指令下 PC 寄存器根据寄存器中值或计算值进行跳转，取得下一指令地址。

Branch operations：根据 opcode 与 ALU 运算结果确定 branch 指令类型以及是否需要跳转和跳转地址。

Jump operations：根据 opcode 与 func 确定 jump 指令类型以及是否需要和跳转地址。

对应仿真波形：

下以第 25 个 benchmark 中对应部分仿真波形为例说明：

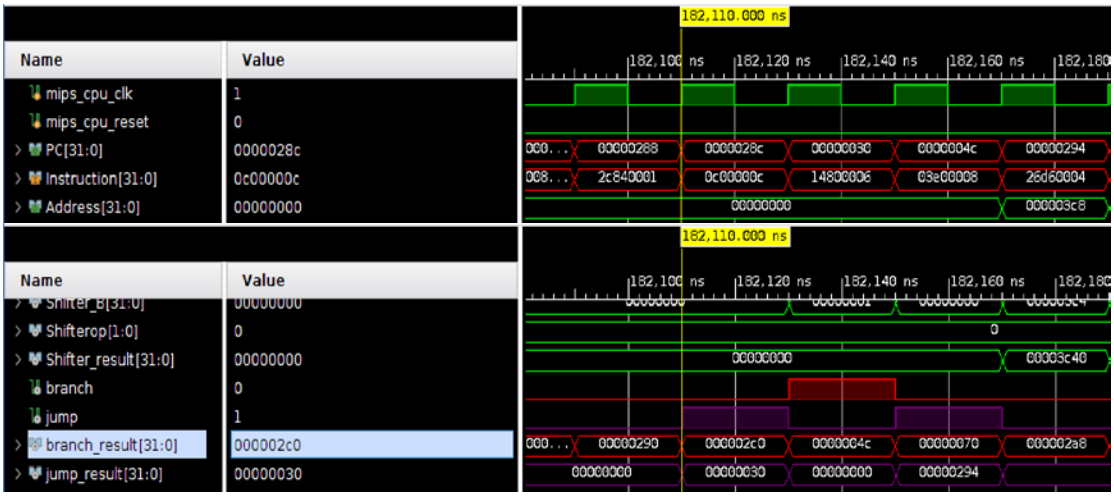


图 3 25_mul-longlong 部分仿真波形（182110ns）

由图三可见，在 182110ns 处 Instruction 为 0c00000c（十六进制），对应指令为 jal，根据逻辑电路结果 jump 信号为 1，PC 寄存器应当根据当前 jump_result 值 00000030 跳转。故在下一时钟周期中 PC 值为 00000030。

在 182120ns 处 Instruction 为 148000006（十六进制），对应指令为 bne，根据 ALU 运算结果 branch 信号为 1，PC 寄存器应当根据当前 branch_result 值 0000004c 跳转。故在下一时钟周期中 PC 值为 0000004c。

2. Register File 部分：

Register File 部分对应逻辑电路图如下：

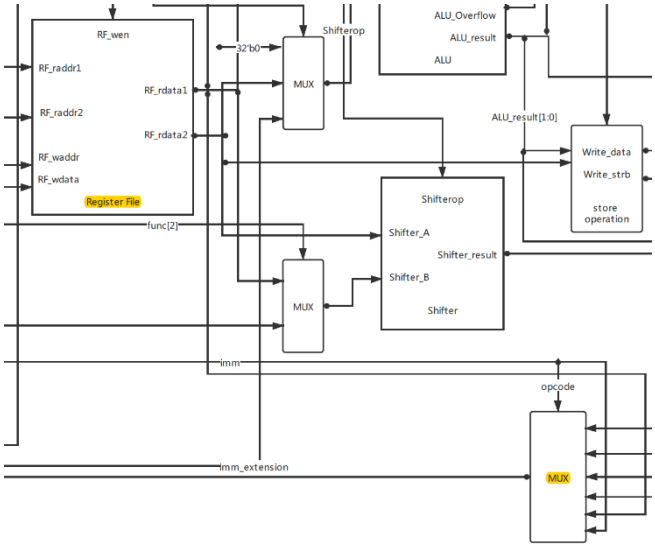


图 4 Register File 部分对应逻辑电路图（荧光部分）

对应代码段：（Register File 部分代码完全复用实验一，不重复说明）

```
        //define signals concerning register file
        assign RF_raddr1 = rs;
        assign RF_raddr2 = rt;
        //jal: opcode[5:0] = 000011                    r31
        //I-type calculate: opcode[5:3] = 001          rt
        //I-type load: opcode[5] = 1, opcode[3] = 0    rt
        //other operations                             rd (or not cared)
        assign RF_waddr = (opcode[5:0] == 6'b000011)                    ?
5'd31 :
                          ((opcode[5:3] == 3'b001) | (opcode[5] & ~opcode[3])) ? rt    :
                          rd ;
        //consider operations not write register file
        //jr:                opcode[5:0] = 000000, func[5:0] = 001000
        //movz:              opcode[5:0] = 000000, func[5:0] = 001010
        //movn:              opcode[5:0] = 000000, func[5:0] = 001011
        //REGIMM:            opcode[5:0] = 000001
        //j:                 opcode[5:0] = 000010
        //I-type branch:     opcode[5:2] = 0001
        //store:             opcode[5] = opcode[3] = 1
        assign RF_wen = ~(((~|opcode) & ((func[5:0] == 6'b001000) | ((func[5:0] == 6'b001010)
& ~ALU_zero) | ((func[5:0] == 6'b001011) & ALU_zero))) |
```

```
                                ((~|opcode[5:1]) & opcode[0])
|
                                (opcode[5:0] == 6'b000010)
|
                                (opcode[5:2] == 4'b0001)
|
                                (opcode[5] & opcode[3])
                               );
        //R-type calculate: opcode[5:0] = 000000, func[5] = 1
        //R-type shift:       opcode[5:0] = 000000, func[5:3] = 000
        //R-type mov:         opcode[5:0] = 000000, func[5:0] = 001010, 001011
        //R-type jalr:        opcode[5:0] = 000000, func[5:0] = 001001
        //I-type calculate: opcode[5:3] = 001 (except lui)
        //jal:                opcode[5:0] = 000011
        //lui:                opcode[5:0] = 001111
        //load:               opcode[5] = 1, opcode[3] = 0
        assign RF_wdata = ((~|opcode) & ((func[5:0] == 6'b001010) & ALU_zero | (func[5:0]
== 6'b001011) & ~ALU_zero)) ? RF_rdata1          :
                                (((~|opcode) &   (func[5:0] == 6'b001001)) | (opcode[5:0] ==
6'b000011))                          ? PC8                :
                                ((~|opcode) &
(~|func[5:3]))                                                                  ?
Shifter_result :
                                (opcode[5:0] ==
6'b001111)                                                                      ?
{imm, 16'b0}     :
                                (opcode[5] &
~opcode[3])                                                                     ?
load_data        :
                                ALU_result;
```

RF_wen（寄存器堆写使能信号）：考虑补集，当指令为 jr、movz、movn、REGIMM、j、I-type branch、I-type store 时寄存器堆不写。

RF_waddr（寄存器堆写地址）：当为 jal 指令时为 31 号寄存器，当为 I-type calculate 或 load 类指令时为 rt 所指寄存器，其余指令默认设为 rd 所指寄存器。

RF_wdata(寄存器堆写数据):当为 R-type mov 类指令时,输入为 RF_rdata1；当为 jalr、jal 指令时，输入为 PC+8；当为 lui 指令时，输入为{imm,16'b0}；当为 R-type shift 类指令时，输入为 Shifter-result；当为 I-type load 类指令时，输入为 loaddata；其余情况默认输入为 ALU-result。

对应仿真波形：

下以第 25 个 benchmark 中对应部分仿真波形为例说明：

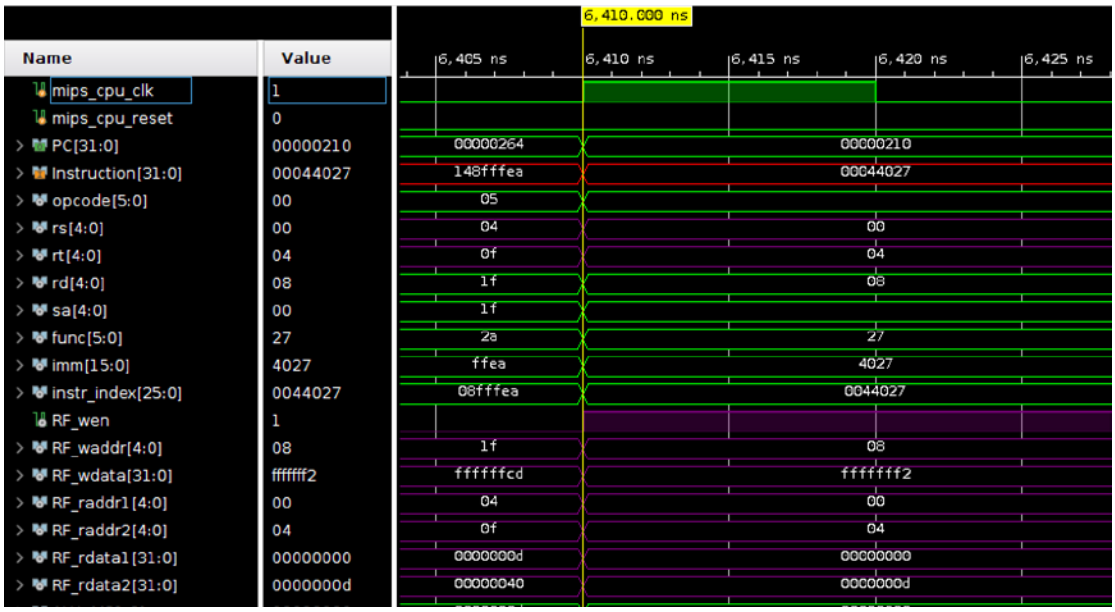
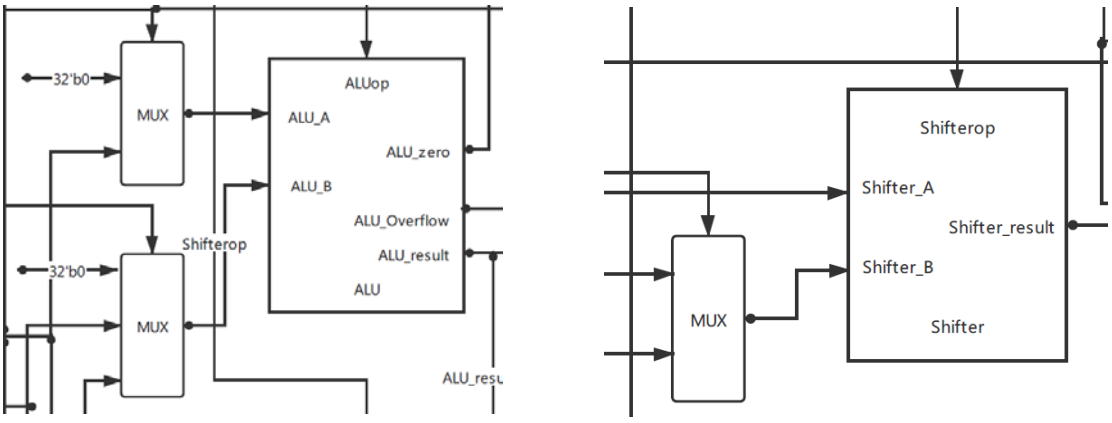


图 5 25_mul-longlong 部分仿真波形（6410ns）

由图五可见，于 6410ns 处，Instruction 为 00044027（十六进制），为 nor 指令。根据逻辑电路结果 RF_wen 为 1，应当向 8(RF_waddr)号寄存器中写入 ffffffd2 (RF_wdata)。而 RF_raddr1 于 RF_raddr2 则默认与 rs 和 rt 分别保持一致。

3. Calculator 部分：

Calculator 部分对应逻辑电路图如下：



（a）ALU　　（b）Shifter

图 5 Calculator 部分对应逻辑电路图

对应代码：

（1） ALU：

ALU 模块内部代码（ALU 部分代码在复用实验一的基础上加以修改，仅列出增改部分）：

```
        assign R_XOR = A ^ B;
        assign R_NOR = ~R_OR;
        assign R_SLTU = {31'b0,~COUT};
    //output
    //final result according to ALUop
        assign Result = ({`DATA_WIDTH{ALUop == AND}} & R_AND) |
                        ({`DATA_WIDTH{ALUop == OR}} & R_OR) |
                        ({`DATA_WIDTH{ALUop == XOR}} & R_XOR) |
                        ({`DATA_WIDTH{ALUop == NOR}} & R_NOR) |
                        ({`DATA_WIDTH{ALUop == ADD || ALUop == SUB}} &
R_ADD_SUB) |
                        ({`DATA_WIDTH{ALUop == SLT}} & R_SLT) |
                        ({`DATA_WIDTH{ALUop == SLTU}} & R_SLTU) ;
```

增加了三种操作：XOR（异或），NOR（或非）以及 SLTU（无符号数比较）

ALU 控制信号、输入输出信号：

```
        //define ALU control signals
        //R-Type calculate: opcode[5:0] = 000000 &&    func[5] = 1
        //I-Type calculate: opcode[5:3] = 001
        //mov                   : opcode[5:0] = 000000 && {func[5:3], func[1]} = 0011
        //REGIMM                : opcode[5:0] = 000001
        //I-type branch     : opcode[5:2] = 0001
        //load store        : opcode[5]     = 1
        assign ALUop = (~|opcode)                   ? (func[5] == 1             ? (func[3:2]
== 2'b00 ? {func[1] , 2'b10}            :
3'b0) :
                                                    opcode[5:3] == 3'b001       ? (opcode[2:1] ==
2'b00               ? {opcode[1] , 2'b10}               :
                                                    opcode[5:0] == 6'b000001 ?    3'b110 :
                                                    opcode[5:2] == 4'b0001      ?
(opcode[1]                              ? 3'b110                                :
3'b0) :
                                                    opcode[5]                     ?    3'b010 :
                                                    3'b0;
         assign ALU_A = ((~|opcode) & ({func[5:3], func[1]} == 4'b0011)) ? 32'b0 :
RF_rdata1;
```

```
        assign ALU_B = ((opcode[5:3] == 3'b001) & (opcode[2:0] != 3'b111)) | (opcode[5] ==
1'b1) ? imm_extension :
                                        ((opcode[5:1] == 5'b00011) | (opcode[5:0] ==
6'b000001))                 ? 32'b0         :
                                        RF_rdata2;
```

ALUop：根据指令类型：R-type calculate、I-type calculate、mov、REGIMM、I-type branch、I-type load store，依据 opcode 以及 func 确定。

ALU_A：仅当指令为 movn 时为 32’b0，其余情况默认为 RF_rdata1。

ALU_B:当指令为 REGIMM 或 I-type branch 类型时为 32’b0;当指令为 I-type calculate（除 lui)、I-type branch、I-type load store 时为 imm_extention；其余情况默认为 RF_rdata2。

（2） Shifter:

Shifter 模块内部代码:

```
`timescale 10 ns / 1 ns
`define DATA_WIDTH 32
    module shifter (
        input [`DATA_WIDTH - 1:0] A,
        input [`DATA_WIDTH - 1:0] B,
        input [1:0] Shiftop,
        output [`DATA_WIDTH - 1:0] Result
);
//define Shifterop operations: left shift; arithmatic right shift, logic right shift
    parameter L = 2'b00,
              AR = 2'b11,
              LR = 2'b10;
//define intermediate reg
//results
wire [`DATA_WIDTH - 1:0] R_L;
wire [`DATA_WIDTH - 1:0] R_AR;
wire [`DATA_WIDTH - 1:0] R_LR;
//shift digits
    assign R_L = A << B[4:0];
    assign R_LR = A >> B[4:0];
    assign R_AR = ($signed(A)) >>> B[4:0];
    assign Result = ({`DATA_WIDTH{Shiftop == L}} & R_L) |
```

```
                ({`DATA_WIDTH{Shiftop == AR}} & R_AR) |
                ({`DATA_WIDTH{Shiftop == LR}} & R_LR);
endmodule
```

为保证算数右移结果正确，算数右移数 A 前加上$signed()。

Shifter 控制信号、输入输出信号：

```
    //define Shifter control signals
    //R-Type shift: opcode[5:0] = 000000, func[5:3] = 000
    //sa: func[2] = 0; rs: func[2] = 1
    assign Shifterop = {2{(~|opcode) & (~|func[5:3])}} & func[1:0];
    assign Shifter_A = RF_rdata2;
    assign Shifter_B = func[2] ? RF_rdata1 : {{27'b0}, sa};
```

Shifterop：依据 opcode 与 func 确定。

Shifter_A：默认为 RF_rdata2。

Shifter_B：sll，sra，srl 指令为 sa，sllv，srav，srlv 指令为 RF_rdata1。

对应仿真波形：

下以第 25 个 benchmark 中对应部分仿真波形为例说明：

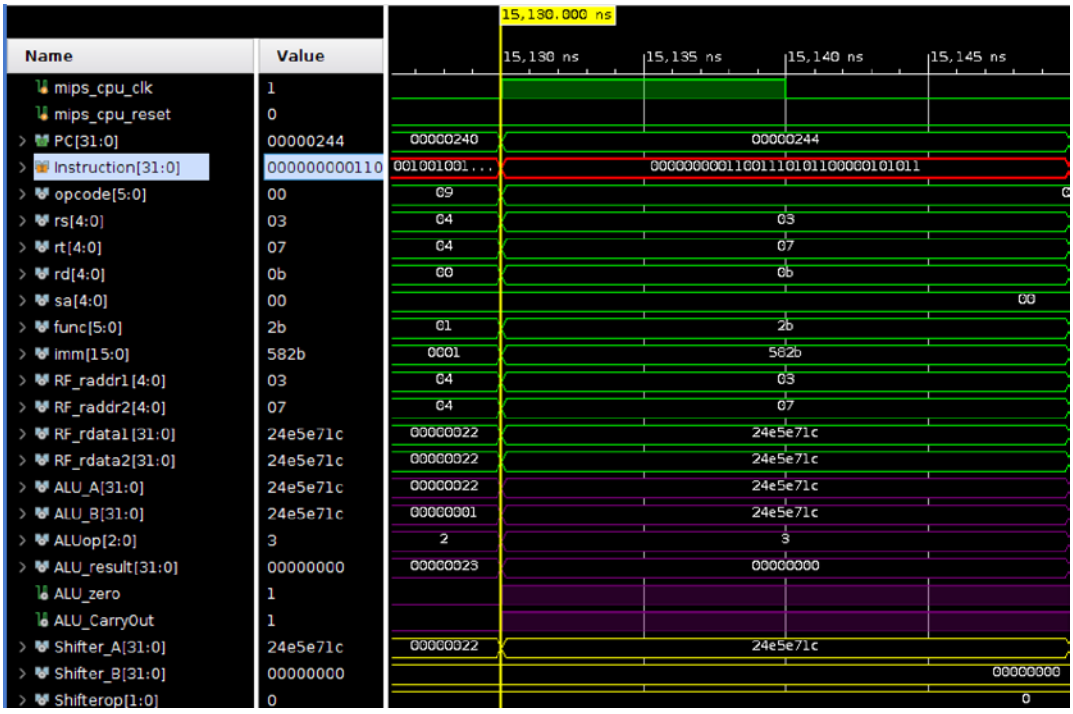


图 6 25_mul-longlong 部分仿真波形（15130ns）

如图 6，15130ns 处 Instruction 为 0067582b（十六进制），为 sltu 指令。根据输入信号以及 ALU 运算，结果为 ALU_result=0

4. （Data）Memory 部分

（Data）Memory 部分对应逻辑电路图如下：

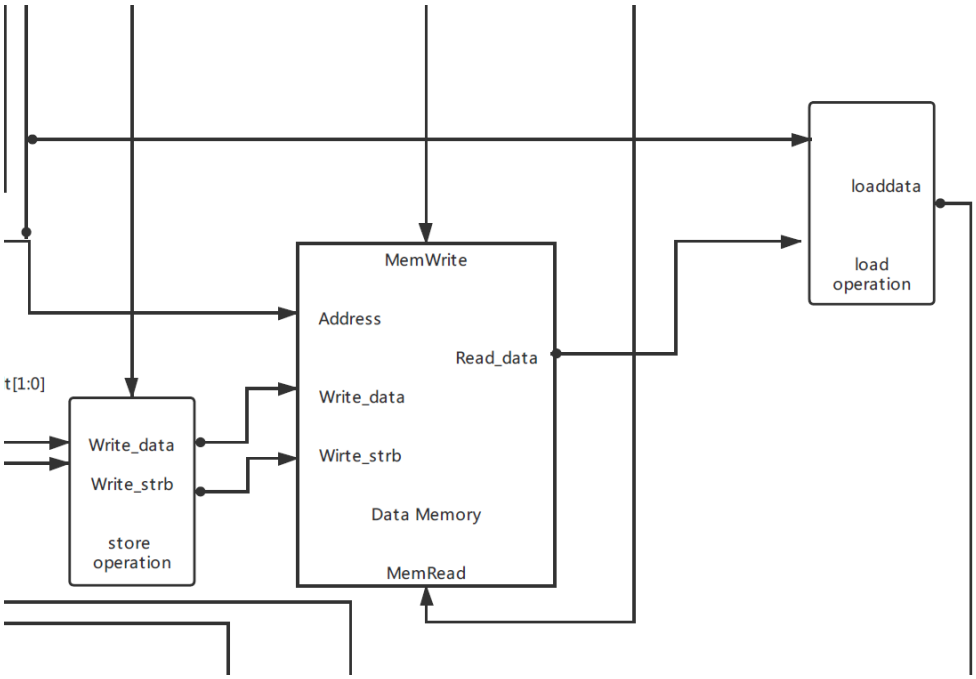


图 7 （Data）Memory 部分对应逻辑电路图

对应代码：

```
//define mem control signals
assign MemWrite = opcode[5] & opcode[3];
assign MemRead  = opcode[5] & ~opcode[3];
assign Address  = {ALU_result[31:2], 2'b0};

//define Write_str & Write_data
//store(opcode[5:3] = 101), sb(opcode[2:0] = 000), sh(001), sw(011), swl(010), swr(110)
assign swl_shift_digits = ({5{~ALU_result[1] & ~ALU_result[0]}} & 5'd24) |
                          ({5{~ALU_result[1] & ALU_result[0]}}  & 5'd16) |
                          ({5{ALU_result[1]  & ~ALU_result[0]}} & 5'd8)  ;
assign swr_shift_digits = ({5{ALU_result[1]  & ALU_result[0]}}  & 5'd24) |
                          ({5{ALU_result[1]  & ~ALU_result[0]}} & 5'd16) |
                          ({5{~ALU_result[1] & ALU_result[0]}}  & 5'd8)  ;
```

```
        assign Write_strb = ({4{opcode[2:0] == 3'b000}} & {ALU_result[1] & ALU_result[0],
ALU_result[1] & ~ALU_result[0], ~ALU_result[1] & ALU_result[0], ~ALU_result[1] &
~ALU_result[0]}) |
                                ({4{opcode[2:0] == 3'b001}} & {ALU_result[1],
ALU_result[1], ~ALU_result[1], ~ALU_result[1]}) |
                                ({4{opcode[2:0] == 3'b011}} & 4'b1111) |
                                ({4{opcode[2:0] == 3'b010}} & {ALU_result[1] &
ALU_result[0], ALU_result[1], ALU_result[1] | ALU_result[0], 1'b1}) |
                                ({4{opcode[2:0] == 3'b110}} & {1'b1, ~(ALU_result[1] &
ALU_result[0]), ~ALU_result[1], ~(ALU_result[1] | ALU_result[0])}) ;
        assign Write_data = ({32{opcode[2:0] == 3'b000}} & {4{RF_rdata2[7:0]}}) |
                                ({32{opcode[2:0] == 3'b001}} & {2{RF_rdata2[15:0]}}) |
                                ({32{opcode[2:0] == 3'b011}} & RF_rdata2) |
                                ({32{opcode[2:0] == 3'b010}} & (RF_rdata2 >>
swl_shift_digits)) |
                                ({32{opcode[2:0] == 3'b110}} & (RF_rdata2 <<
swr_shift_digits)) ;

        //define load data
        //lb(000), lh(001), lw(011), lbu(100), lhu(101), lwl(010), lwr(110)
        assign lb_data      = ({32{~ALU_result[1] & ~ALU_result[0]}} &
{{24{Read_data[7]}} , Read_data[7:0]})     |
                                ({32{~ALU_result[1] & ALU_result[0]}}    &
{{24{Read_data[15]}}, Read_data[15:8]})    |
                                ({32{ALU_result[1]    & ~ALU_result[0]}} &
{{24{Read_data[23]}}, Read_data[23:16]}) |
                                ({32{ALU_result[1]    & ALU_result[0]}}    &
{{24{Read_data[31]}}, Read_data[31:24]}) ;
        assign lh_data      = ({32{~ALU_result[1]}} & {{16{Read_data[15]}},
Read_data[15:0]})    |
                                ({32{ALU_result[1]}}    & {{16{Read_data[31]}},
Read_data[31:16]}) ;
        assign lw_data      = Read_data[31:0];
        assign lbu_data     = {24'b0, lb_data[7:0]};
        assign lhu_data     = {16'b0, lh_data[15:0]};
        assign lwl_data     = ({32{~ALU_result[1] & ~ALU_result[0]}} & {Read_data[7:0] ,
RF_rdata2[23:0]}) |
                                ({32{~ALU_result[1] & ALU_result[0]}}    &
{Read_data[15:0], RF_rdata2[15:0]}) |
                                ({32{ALU_result[1]    & ~ALU_result[0]}} &
{Read_data[23:0], RF_rdata2[7:0]})    |
                                ({32{ALU_result[1]    & ALU_result[0]}}    & Read_data) ;
        assign lwr_data     = ({32{~ALU_result[1] & ~ALU_result[0]}} & Read_data) |
                                ({32{~ALU_result[1] & ALU_result[0]}}    &
{RF_rdata2[31:24], Read_data[31:8]})    |
```

```
                                ({32{ALU_result[1]    & ~ALU_result[0]}} &
{RF_rdata2[31:16], Read_data[31:16]}) |
                                ({32{ALU_result[1]    & ALU_result[0]}}    &
{RF_rdata2[31:8] , Read_data[31:24]}) ;
        assign load_data = ({32{opcode[2:0] == 3'b000}} & lb_data)    |
                                ({32{opcode[2:0] == 3'b001}} & lh_data)    |
                                ({32{opcode[2:0] == 3'b011}} & lw_data)    |
                                ({32{opcode[2:0] == 3'b100}} & lbu_data) |
                                ({32{opcode[2:0] == 3'b101}} & lhu_data) |
                                ({32{opcode[2:0] == 3'b010}} & lwl_data) |
                                ({32{opcode[2:0] == 3'b110}} & lwr_data) ;
```

Write_strb、Write_data、load_data：根据指令手册中的数据位数选择确定。

对应仿真波形：

下以第 26 个 benchmark 中对应部分仿真波形为例说明：

图 8 26_load-store 对应部分仿真波形

由图 8 可见，在 5010ns 处，Instruction 为 8cc40000（十六进制），为 lwl 指令。此时将 f7000001（lwl_data）写入 Register File。

在 5090ns 处，Instruction 为 a4e20000（十六进制），为 sb 指令。此时根据逻辑电路将 0000001（Write_data）存入地址为 000001b4（Address）内存中。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1．RTL 代码：

（1）语法错误：

问题 1：语句末尾缺少“;”

解决方案：在外部有自动检错软件中编写代码，防止出现低级语法错误。

问题 2：由于复制代码未检查，模块末尾出现多余 endmodule

解决方案：复用、复制代码后需检查一遍，是否产生重复语句。

（2）逻辑错误：

问题 1：编写代码时由于笔误设置信号宽度出错、选取信号位数出错

解决方案：根据波形图，定位出错信号位置，逐行排查逻辑，加以修改；编写代码时小心谨慎，防止书写错误。

问题 2：运算符优先级混淆，导致逻辑错误

解决方案：遇到不熟悉运算符时，比对运算优先级表再进行代码编写；编写代码时，合理使用括号，防止运算符优先级导致的逻辑错误。

2．仿真、云平台调试中的难点：

问题：每次查看波形图，需要添加新信号时，需要重新提交、运行，耗费时间较长。

解决方案：直接一次性将所有信号添加，减少因多次添加信号造成的浪费时间。

三、 对讲义中思考题（如有）的理解和回答

本次实验无思考题

四、 在课后，你花费了大约 20 小时完成此次实验。

五、 对于此次实验的心得、感受和建议

实验心得、感受：

1．仔细阅读指令手册，完善译码表有助于后续实验代码的快速、稳定编写。

2．遇到问题，长时间未能完成 debug，积极与同学交流，往往能够快速解决。

（感谢冯浩原同学愿意与我交流病情）

3．记录每次代码编写进度，防止遗漏部分逻辑的代码编写。

实验建议：

1．希望能完善云平台测试功能：波形信号增改可以不用反复提交，重复运行。

2．希望增加拓展知识讲解（如 SoC）。