中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2015K8009929045　姓名：张远航　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

注 1：本实验报告请以 PDF 格式提交。文件命名规则：[学号]-PRJ[实验序号]-RPT.pdf，其中文件名字母大写，后缀名小写。例如：[2014K8009959088]-PRJ[1]-RPT.pdf

注 2：实验报告模板以下部分的内容供参考，可包含但不限定如下条目内容。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段 {包含注释} 及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

最后结果：Data Path Delay = 11.403ns，理论最高频率 87.7MHz。

Summary

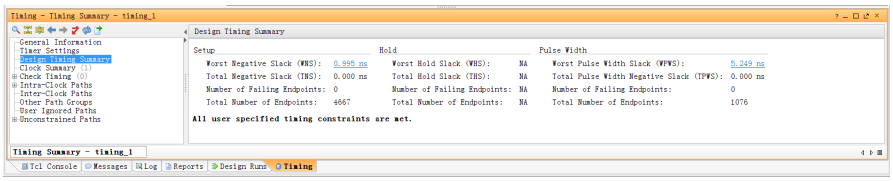
Name	Path 1
Slack	0.995ns
Source	u_mips_cpu_top/u_mips_cpu/mips_cpu_datapath/mips_PC_reg[2]/C (rising edge-triggered cell FDRE clocked by clk_fpga_0 (rise@0.00
Destination	u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r1_0_31_12_17/RAMA_D1/I (rising edge-triggered cell RAMD32 clocked by clk_fpg
Path Group	clk_fpga_0
Path Type	Setup (Max at Slow Process Corner)
Requirement	12.999ns (clk_fpga_0 rise@12.999ns - clk_fpga_0 rise@0.000ns)
Data Path Delay	11.403ns (logic 2.467ns (21.635%) route 8.936ns (78.365%))
Logic Levels	9 (CARRY4=1 LUT5=2 LUT6=3 RAMD32=1 RAMD64E=2)
Clock Path Skew	-0.145ns
Clock Uncertainty	0.198ns

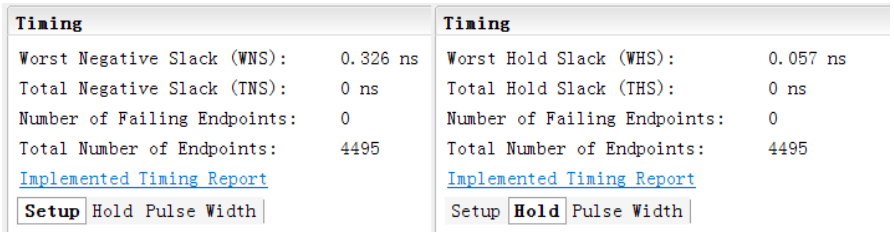
最长路径 9 层逻辑，寄存器堆最多 3 层逻辑，ALU 最多 3 层逻辑，基本已经压平，大部分延时来自 `ideal_mem`，很难再进一步优化。

Data Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
FDRE (Prop fdre C Q)	(r) 0.478	2.178		u_mips_cpu_top/u_mips_cpu/mips_cpu_datapath/mips_PC_reg[2]/Q
net (fo=131, unplaced)	1.071	3.249		u_mips_cpu_top/u_ideal_mem/mem_reg_r1_128_191_21_23/ADDRB0
				u_mips_cpu_top/u_ideal_mem/mem_reg_r1_128_191_21_23/RAMB/RADR0
RAMD64E (Prop ramd64e RADR0 O)	(r) 0.295	3.544		u_mips_cpu_top/u_ideal_mem/mem_reg_r1_128_191_21_23/RAMB/O
net (fo=1, unplaced)	1.111	4.655		u_mips_cpu_top/u_ideal_mem/mem_reg_r1_128_191_21_23_n_1
				u_mips_cpu_top/u_ideal_mem/r_reg_r2_0_31_0_5_i_4/I1
LUT6 (Prop lut6_I1_O)	(r) 0.124	4.779		u_mips_cpu_top/u_ideal_mem/r_reg_r2_0_31_0_5_i_4/O
net (fo=36, unplaced)	1.185	5.964		u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r2_0_31_0_5/ADDRA1
				u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r2_0_31_0_5/RAMA_D1/RADR1
RAMD32 (Prop ramd32_RADR1_O)	(r) 0.124	6.088		u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r2_0_31_0_5/RAMA_D1/O
net (fo=5, unplaced)	1.139	7.227		u_mips_cpu_top/u_mips_cpu/mips_cpu_alu/rdata1[1]
				u_mips_cpu_top/u_mips_cpu/mips_cpu_alu/mem_reg_r1_0_63_0_2_i_24/I0
LUT5 (Prop lut5_I0_O)	(r) 0.124	7.351		u_mips_cpu_top/u_mips_cpu/mips_cpu_alu/mem_reg_r1_0_63_0_2_i_24/O
net (fo=1, unplaced)	0.000	7.351		u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/S[1]
				u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/mem_reg_r1_0_63_0_2_i_17/S[1]
CARRY4 (Prop carry4_S[1]_O[3])	(r) 0.643	7.994		u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/mem_reg_r1_0_63_0_2_i_17/O[3]
net (fo=3, unplaced)	0.636	8.630		u_zynq_soc_wrapper/r_reg_r1_0_31_0_5_i_45_0[1]
				u_zynq_soc_wrapper/mem_reg_r2_0_63_0_2_i_5/I5
LUT6 (Prop lut6_I5_O)	(r) 0.307	8.937		u_zynq_soc_wrapper/mem_reg_r2_0_63_0_2_i_5/O
net (fo=128, unplaced)	1.217	10.154		u_mips_cpu_top/u_ideal_mem/mem_reg_r2_64_127_12_14/ADDRB1
				u_mips_cpu_top/u_ideal_mem/mem_reg_r2_64_127_12_14/RAMB/RADR1
RAMD64E (Prop ramd64e RADR1 O)	(r) 0.124	10.278		u_mips_cpu_top/u_ideal_mem/mem_reg_r2_64_127_12_14/RAMB/O
net (fo=1, unplaced)	1.111	11.389		u_mips_cpu_top/u_ideal_mem/mem_reg_r2_64_127_12_14_n_1
				u_mips_cpu_top/u_ideal_mem/r_reg_r1_0_31_12_17_i_8/I0
LUT5 (Prop lut5_I0_O)	(r) 0.124	11.513		u_mips_cpu_top/u_ideal_mem/r_reg_r1_0_31_12_17_i_8/O
net (fo=2, unplaced)	1.122	12.635		u_mips_cpu_top/u_ideal_mem/r_reg_r1_0_31_12_17_i_8_n_0
				u_mips_cpu_top/u_ideal_mem/r_reg_r1_0_31_12_17_i_1/I1
LUT6 (Prop lut6_I1_O)	(r) 0.124	12.759		u_mips_cpu_top/u_ideal_mem/r_reg_r1_0_31_12_17_i_1/O
net (fo=2, unplaced)	0.344	13.103		u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r1_0_31_12_17/DIA1
RAMD32				u_mips_cpu_top/u_mips_cpu/mips_cpu_reg_file/r_reg_r1_0_31_12_17/RAMA_D1/I
Arrival Time		13.103		

在 80MHz 运行时，综合后 WNS = 0.995ns，实现后 Setup WNS = 0.326ns，Hold WHS = 0.057ns。





Timing		Timing	
Worst Negative Slack (WNS):	0.326 ns	Worst Hold Slack (WHS):	0.057 ns
Total Negative Slack (TNS):	0 ns	Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4495	Total Number of Endpoints:	4495
Implemented Timing Report		Implemented Timing Report	
Setup Hold Pulse Width		Setup Hold Pulse Width	

ALU 与 RF 已经在实验一中完成，但本次实验后半程进行优化时，我发现上次写的 RF 虽然功能没有问题，但综合后的物理实现是错误的。具体解释及修改后的代码见报告第四部分第一节“在修正中前进”。修正后的寄存器堆逻辑电路结构见图 1 。

数据通路代码如下，实现遵照 Patterson 在 COD 中给出的结构，参见图 2 。

```
1  module datapath(
2      input  rst,
3      input  clk,
4
5      // signals from Control
6      output wire[5:0] Op,
7      input  [1:0] ALUOp,
8      input  RegWrite, RegDst, MemtoReg, ALUSrc, MemWrite, MemRead
           , Branch,
9
10     // signals from ALU Control
11     output wire [5:0] func,
12
13     // signals from ALU
14     output [31:0] A, B,
15     input  [31:0] Result,
16     input  Zero,
17
18     // signals from register file
19     output wire [4:0] waddr, raddr1, raddr2,
20     output wire wen,
21     output wire [31:0] wdata,
22     input  [31:0] rdata1, rdata2,
23
24     // MIPS core I/O
25     output reg [31:0] mips_PC,
26     input  [31:0] mips_Instruction,
```

```
27      output wire [31:0] mips_Address,
28      output wire mips_MemWrite,
29      output wire mips_MemRead,
30      output wire [31:0] mips_Write_data,
31      input  [31:0] mips_Read_data
32  );
33
34      // note: no R-type ops specified so far
35      wire [4:0] rs, rt, rd;
36      wire [15:0] immediate;
37
38      wire PCsrc, SignExtend;
39      reg  [31:0] nPC; // PC and nextPC (nPC)
40      wire [31:0] PC_p4, PC_jump;
41
42      wire [31:0] sext_imm;
43
44      initial begin
45          nPC = 32'b0;
46      end
47
48      assign Op = mips_Instruction[31:26];
49      assign rs = mips_Instruction[25:21];
50      assign rt = mips_Instruction[20:16];
51      assign rd = mips_Instruction[15:11];
52      assign immediate = mips_Instruction[15:0];
53      assign func = mips_Instruction[5:0];
54
55      // multiplexors
56      assign SignExtend = mips_Instruction[15];
57      assign PCsrc = ({Branch, Zero} == 2'b10);
58
59      // sign-extended immediate
60      assign sext_imm = SignExtend ? {16'hFFFF, immediate} : {16'
            h0000, immediate};
61
62      assign PC_p4 = mips_PC + 4;
63      assign PC_jump = PC_p4 + (sext_imm << 2);
64
```

```
    // ALU
    assign A = rdata1;
    assign B = ALUSrc ? sext_imm : rdata2;

    // register file
    assign waddr = RegDst ? rd : rt;
    assign raddr1 = rs;
    assign raddr2 = rt;
    assign wen = RegWrite;
    assign wdata = MemtoReg ? mips_Read_data : Result;

    // MIPS
    assign mips_MemWrite = MemWrite;
    assign mips_MemRead = MemRead;
    assign mips_Address = Result;
    assign mips_Write_data = rdata2;

    always @(posedge clk) begin
        if (rst) mips_PC <= 32'b0;
        else mips_PC <= nPC;
    end

    always @(*) begin
        nPC <= PCsrc ? PC_jump : PC_p4;
    end

endmodule
```

然后是 ALU 的控制单元（采用多级译码）：

```
module alu_control(
    input  [5:0] func,

    input  [1:0] ALUOp,
    output reg [2:0] ALUctr
);

    parameter AND = 3'b000;
    parameter ADD = 3'b010;
    parameter SUB = 3'b110;
```

```
11
12      always @(*) begin
13          casex ({ALUOp, func})
14              8'b00xxxxxx: ALUctr <= ADD; // LW or SW
15              8'b01xxxxxx: ALUctr <= SUB; // BNE
16              8'b11xxxxxx: ALUctr <= ADD; // I-type
17              default: ALUctr <= 3'b000;  // exceptions
18          endcase
19      end
20
21  endmodule
```

以及 CPU 的控制单元：

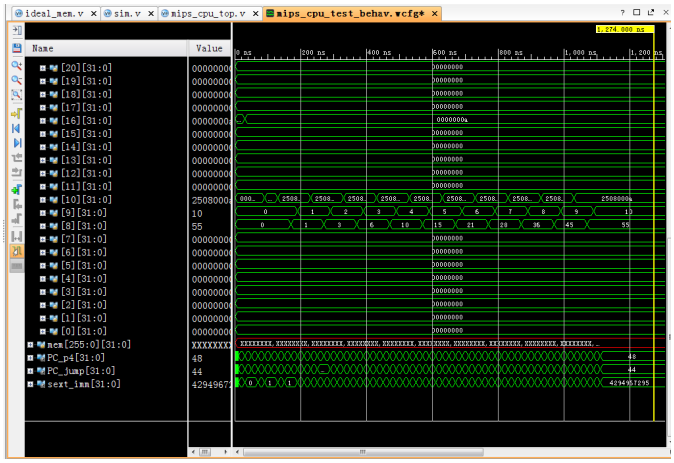
```
1   module control(
2       input  [5:0] Op,
3       output reg [1:0] ALUOp,
4
5       output reg RegDst, Branch, MemRead, MemtoReg, MemWrite,
            ALUSrc, RegWrite
6   );
7
8       parameter ADDIU = 6'b001001;
9       parameter LW = 6'b100011;
10      parameter SW = 6'b101011;
11      parameter BNE = 6'b000101;
12      parameter SPECIAL = 6'b000000;
13
14      always @(*) begin
15          case (Op)
16              LW: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b00_0011011;
17              SW: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b00_0000110;
18              BNE: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b01_0100000;
19              ADDIU: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b11_0000011;
20              SPECIAL: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b10_1000000;
```

```
21              default: {ALUOp, RegDst, Branch, MemRead, MemtoReg,
                    MemWrite, ALUSrc, RegWrite} <= 9'b00_0000000;
22          endcase
23      end
24
25  endmodule
```

最后在 mips_cpu.v 中对各组件简单进行实例化即可，由于基本是端口定义没有任何内涵，该文件代码略去。CPU 的完整 RTL 图见图 3 。

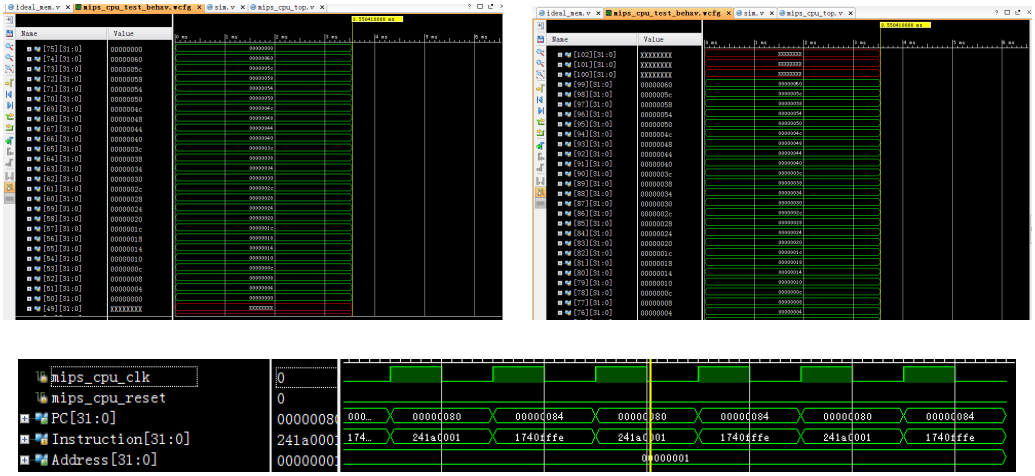
下面是一些仿真波形和上板时的截图。测试用到的程序一代码如下，它从 1 加到 10。里面的寄存器号定义和使用遵循 MIPS 规范。（感谢这极其有限的指令集，让我充分开动脑筋想到了改写指令实现内层循环，也学会了如何用 BNE 指令实现指令和数据的区分以及停机操作）：

```
mem[0] = `ADDIU(`_s0, `_zero, 16'd10);  // const max = 10;
mem[1] = `ADDIU(`_t1, `_zero, 16'd0);
// initialize loop counter b
mem[2] = `ADDIU(`_t0, `_zero, 16'd0);   // initialize a
mem[3] = `LW(`_t2, `_zero, 16'd20);
// initialize command register with mem[5]
mem[4] = `BNE(`_t2, `_zero, 16'd1);
// jump to loop body mem[6]
mem[5] = `ADDIU(`_t0, `_t0, 16'd0);     // initial: "a = a + 0"
mem[6] = `ADDIU(`_t2, `_t2, 16'd1);     // increment command
mem[7] = `SW(`_t2, `_zero, 16'd32);
// update command in mem[8]
mem[8] = `NOP;
// to be replaced by command "a = a + x", x = 1...b
mem[9] = `ADDIU(`_t1, `_t1, 16'd1);
// increment loop counter
mem[10] = `BNE(`_s0, `_t1, -16'd5);
// if (b != max) jump to mem[6] and increment command
mem[11] = `BNE(`_zero, `_s0, -16'd1);   // halt
```



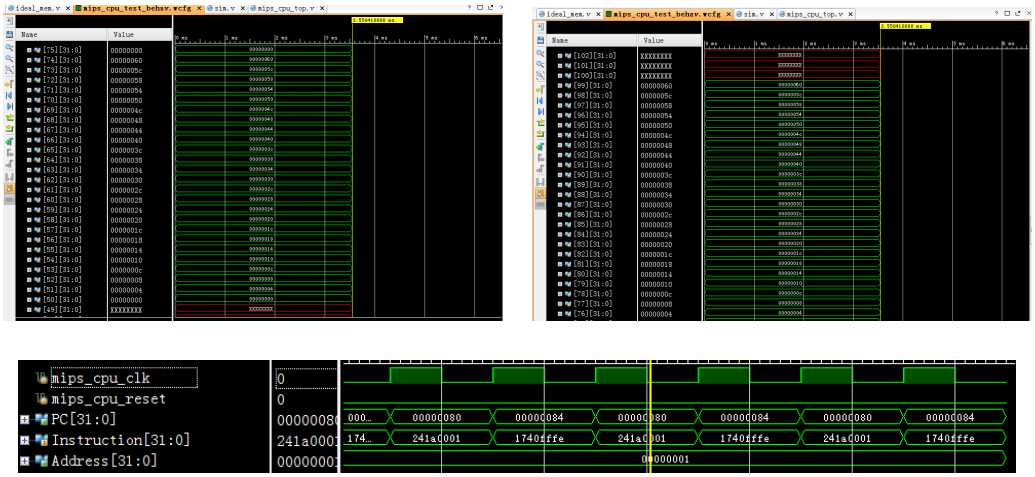
测试程序 1：$1+2+\cdots+10=55$

（8 号寄存器存放和，9 号寄存器存放循环变量）



测试程序 2：memcpy

（最后 PC 在 80 和 84 之间反复，进入汇编代码的 spin 段）



上板验收阶段

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真及上板调试过程中的难点等）

Bug 合集

(a) 字长是 4!! 字长是 4!! 字长是 4!!

(b) 细枝末节的问题：译码器手滑敲错一位，`ALUOp`（ALU 控制单元的多级译码信号）和 `ALUop`（运算类型）搞混之类。

(c) 第二节课助教哥哥告诉了我们怎么用静态时序分析提高频率，人总要有点追求嘛，我就开始了漫漫优化路，结果之前验收通过的代码突然出不来正确结果了……原因是这样的：最开始，我的 `PC` 是用两个实例化的 ALU 计算的，运算结果也没有问题，上板验收都过了：

```
alu add32_0(.A(mips_PC), .B(4), .ALUop(3'b010), .Overflow(),
    .CarryOut(), .Zero(), .Result(PC_p4));
alu add32_1(.A(PC_p4), .B(sext_imm << 2), .ALUop(3'b010), .
   Overflow(), .CarryOut(), .Zero(), .Result(PC_jump));
```

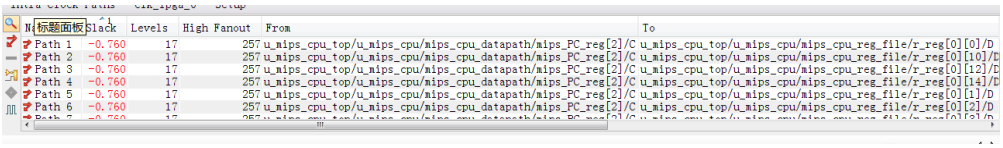
实验课验收完成后当晚，想到 ALU 的使用会增加逻辑层数，我随手将其改成了：

```
assign PC_p4 = mips_PC + 4;
assign PC_jump = PC_p4 + sext_imm << 2; // wrong
```

可是加号的优先级是比逻辑左移的优先级高的，因此 `sext_imm << 2` 外侧应该有一对括号。由于这个修改做的时间距离我开始做第二次优化比较远，我一直没对数据通路的正确性产生过怀疑，最后排查 n 遍无果回来盯着 `PC` 的波形看，才意识到问题所在。

结论：（1）不常用的运算符多打个括号没坏处。（2）做好版本控制。

在修正中前进 第一次完成了寄存器堆后要撰写实验报告，当时看到寄存器堆的 schematic 非常复杂，心底已经有点疑问，但也没细究。这次为了让 CPU 跑的更快，仔细分析了关键路径，才意识到寄存器堆的实现是有问题的：我对寄存器做了写零复位操作，这不但是硬件不支持的，也不是 MIPS 所要求的！结果就是，本来应该被映射到 RAM 的存储器变成了一大把触发器，也给 CPU 带来了巨大的 delay……最后的解决方法是，用 initial 语句，这个不占用额外资源。



延时基本全部来自寄存器堆，最长路径一度竟有 20 层逻辑！

修改过程中，我尝试过很多不同的组合，比如只给 `$zero` 寄存器写 0 之类，效果如旧。但是这其实是可以猜到的，因为综合花了整整七分半（它大概是在用运行时间表示抗议），而通常代码没什么问题的话综合一分半以内就结束了。或许可以说，这也正验证了一句真理：

Simplicity prefers regularity.

改成逆否命题：If it's not regular, clearly it's not simple enough.

此外，我之前没有必要的写回操作也带来了额外延时，本来应该只有两个大选择器，结果产生了三个：

```
// bad
r[waddr] <= (wen && waddr)? wdata : r[waddr];
// revised
if (wen && waddr) r[waddr] <=  wdata;
```

下面是修改后的寄存器堆代码：

```
... (port definition, omitted)
    parameter REG_UNITS = 1 << `ADDR_WIDTH;
    reg [`DATA_WIDTH – 1:0] r[REG_UNITS – 1:0];

    integer i;
    initial begin
        for (i = 0; i < REG_UNITS; i = i + 1)
            r[i] = `DATA_WIDTH'b0;
    end

    always @(posedge clk) begin
        // zero register ($zero) always gives a value of zero
           and should not be overwritten
        if (wen && waddr) r[waddr] <= wdata;
    end

    assign rdata1 = r[raddr1];
```

```
17      assign rdata2 = r[raddr2];
18
19  endmodule
```

三、 对讲义中思考题的理解和回答

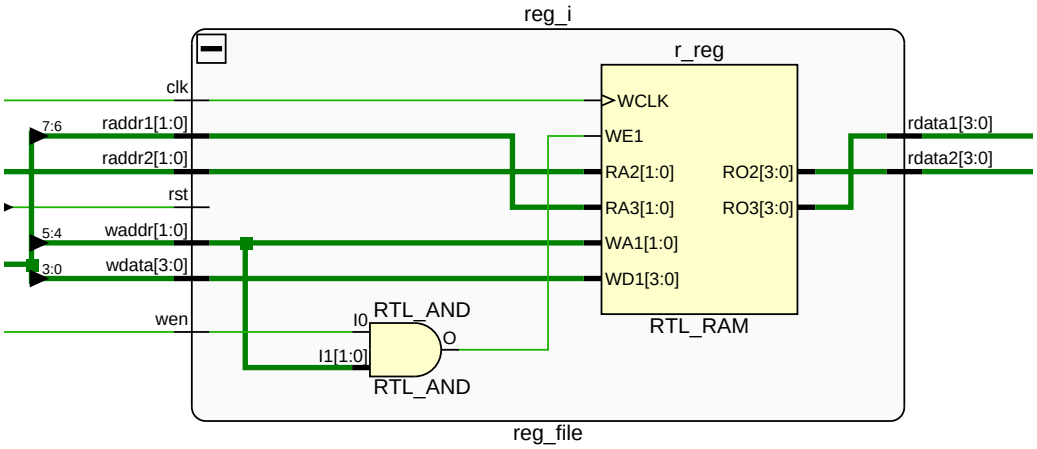
本次实验没有思考题。（其实我觉得调试和优化中的思考汲取的经验才是真正的财富，纸上得来终觉浅，绝知此事要躬行，写硬件代码更是这样。）

四、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，以及其他想与任课老师交流的内容等）

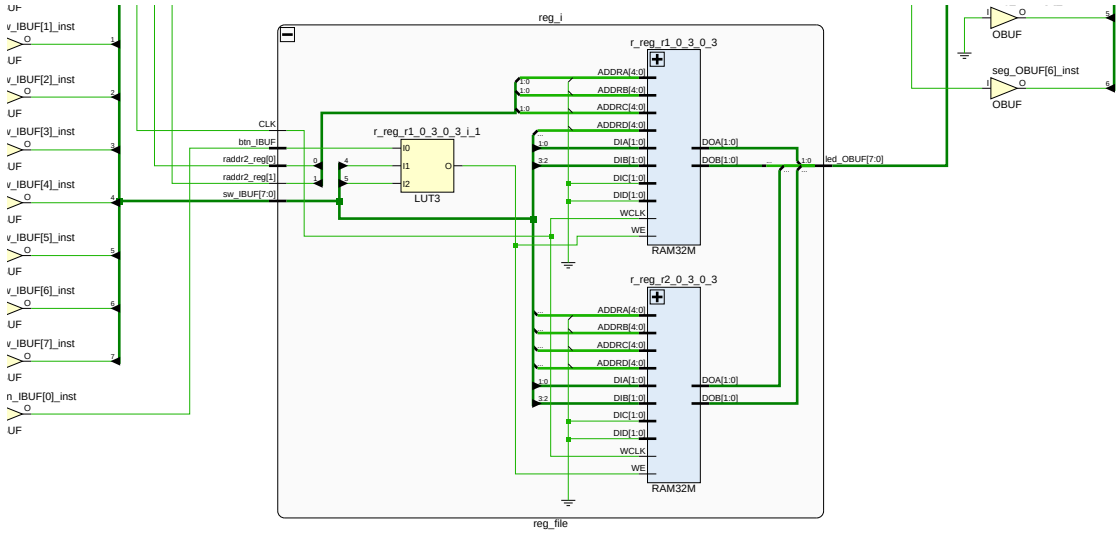
- 学会的新技能：`casex` 语句（可以用通配符），`initial` 语句（不需要占用额外资源）。
- 我觉得助教老师应该提醒我们关注一些部件综合后的物理实现，比如寄存器堆。一旦做了硬件上不支持的操作（比如初始化时写 0），它就不再会被映射为一块 RAM，而是变成了用触发器拼凑而成的 state element。
- 这次实验让我充分领会到了写 Verilog 代码与写 C 代码的差别。编程语言写多了以后我们往往关注算法而不重视里面的实现细节，把优化工作都交给编译器来做；而硬件描述语言则不同，在写的时候如果你脑子里对自己的代码综合以后出来的电路是个什么样子没有概念，那肯定写不出最好的代码。一不小心少写个括号，可能就多了一层逻辑。最近找了两本国外关于 Verilog 的书，专门讲这种细节问题的。
- 上板三分钟，台下十年功。（听起来好像没有“奋战三小时做个 CPU”振奋人心？）
- 我觉得下一届可以加一个思考题，如何用这几条基本指令（可以认为是图灵完备的）实现一些其他操作？
- 虽然还是没有实现跑上 100MHz 的理想，也算是尽力了～还帮另两个同学解决了一些问题:-)

插图

1 修正后寄存器堆的逻辑电路结构 12
2 数据通路 13
3 CPU 完整的 RTL 图 14

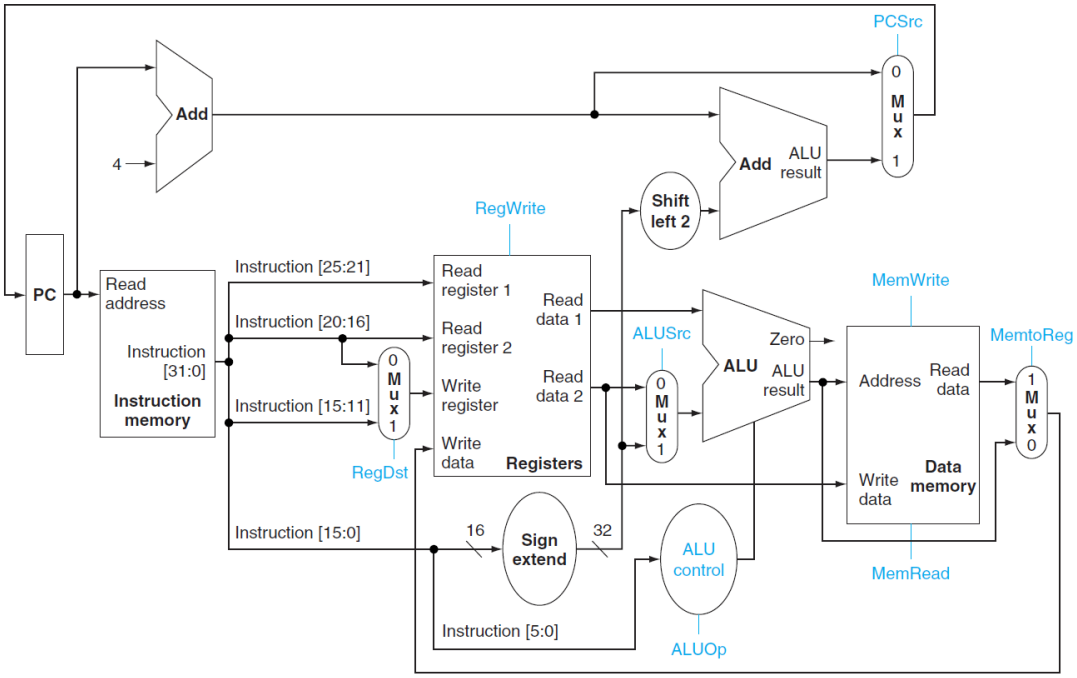


(a) RTL 图

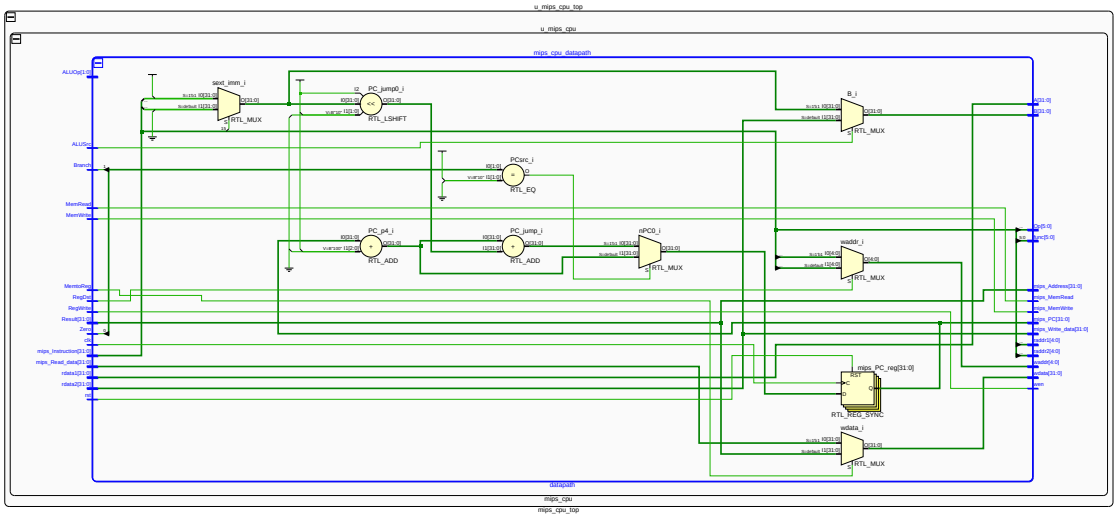


(b) 综合后

图 1: 修正后寄存器堆的逻辑电路结构



(a) 期望的数据通路（来源：Patterson & Hennessey, 2005）



(b) 实现的 RTL 图

图 2: 数据通路

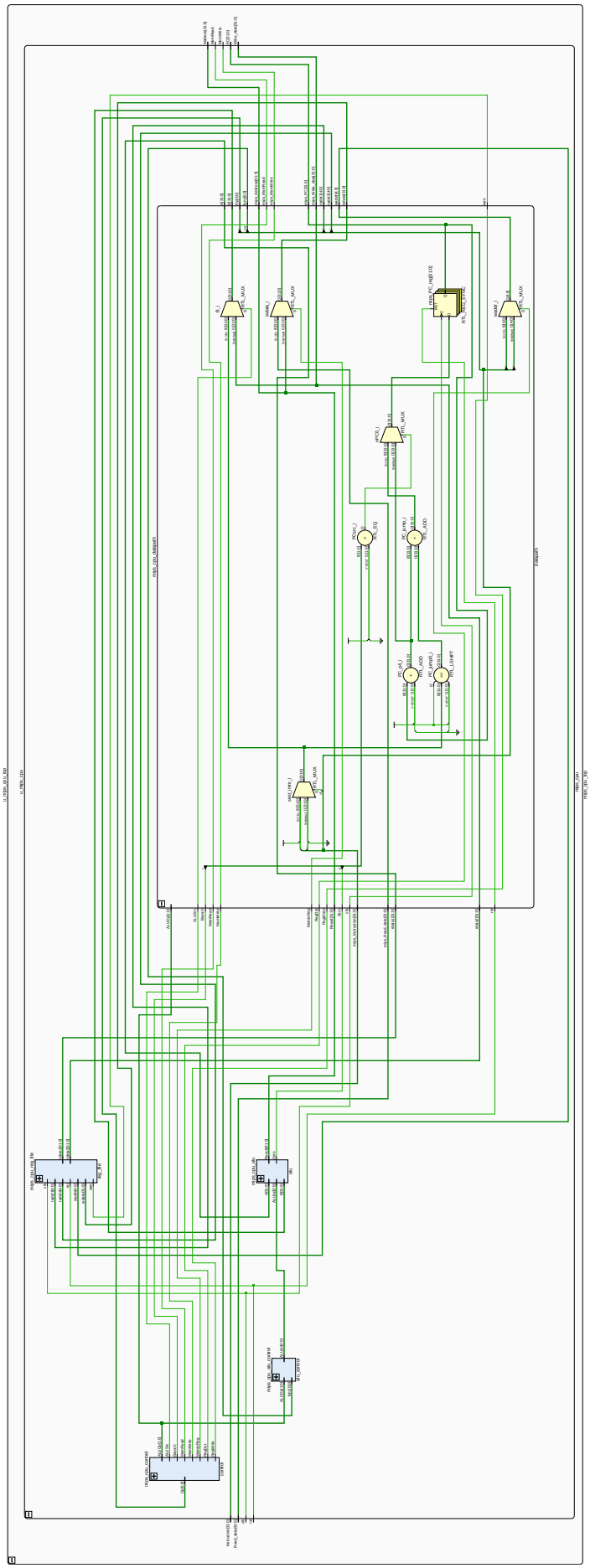


图 3: CPU 完整的 RTL 图