# 中国科学院大学计算机组成原理实验课

# 实 验 报 告

学号：_____2018K8009909006_____ 姓名：__唐宇菲_____ 专业：数学与应用数学_____

实验序号：__2__ 实验名称：___简单功能型处理器设计

_____

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在~/COD-Lab/reports 目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 GitLab 远程仓库 master 分支（具体命令详见实验报告）。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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图、相应信号的仿真波形和信号变化的说明等）

首先实验的整体流程是：根据 45 条指令整理出了一个 excel 表格，然后根据表格中对于指令的大类，进行代码写作。后来发现这样写比较乱，整体还是需要按照指令的 取指、译码、执行、访存、写回这 5 个步骤来进行。

整体的结构与 ppt 上的类似：

下面对信号进行说明：

（1）IF 阶段

定义了 cpu_PC,cpu_PC_next,分别表示当前拍的 PC 和下一拍的 PC，

用 cpu_PC_seq 表示无跳转下的 PC+ 4，cpu_PC_branch 表示属于 I-Type-branch 或者 Regimm 指令，cpu_PC_jump 表示属于 J-Type 类型指令。

cpu_PC_R 表示属于 jr 或者 jalr 类型指令。

然后分别施加判断信号。

```
assign PC = cpu_PC;

wire [31 : 0] cpu_PC_seq;
wire [31 : 0] cpu_PC_branch;
wire cpu_PC_branch_enable;
wire cpu_PC_jump_enable;
wire [31 : 0] cpu_PC_jump;
wire cpu_PC_R_enable;
wire [31 : 0] cpu_PC_R;

assign cpu_PC_seq = cpu_PC + 4;
assign cpu_PC_next = ({32{cpu_PC_branch_enable}} & cpu_PC_branch ) |
                     ({32{cpu_PC_jump_enable}} & cpu_PC_jump) |
                     ({32{cpu_PC_R_enable}} & cpu_PC_R) |
                     ( {32{~cpu_PC_branch_enable && ~cpu_PC_jump_enable && ~cpu_PC_R_enable}} & cpu_PC_seq);
```

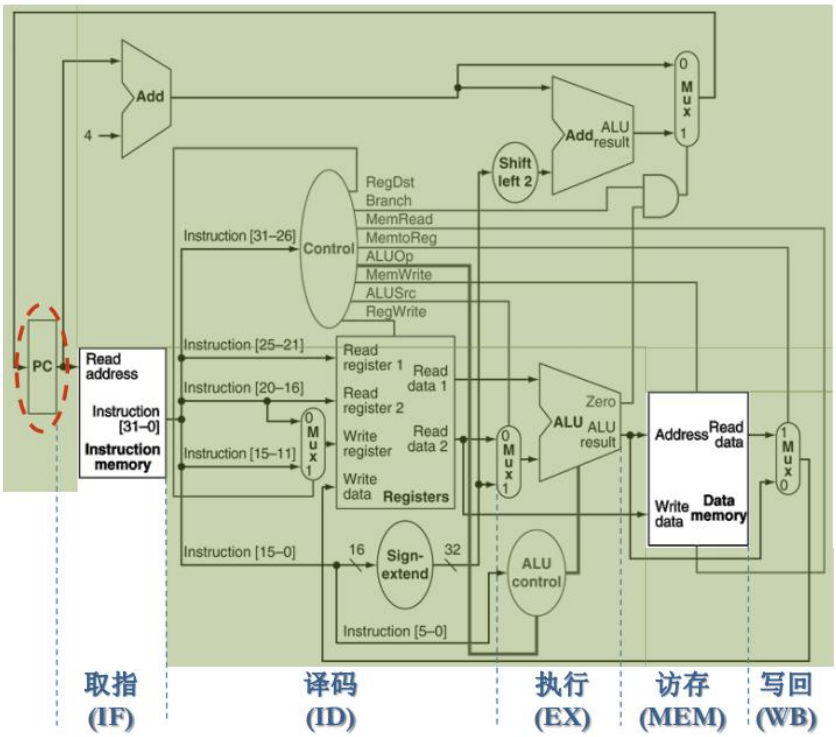
（2）ID 阶段

对于指令各个字段进行判断，将 Instruction 赋值给几个字段
rs,rt,rd,Opcode,func,Immediate,sa,instr_index 等，都是直接根据指令集得到。

```
wire [5 : 0] Opcode;
wire [4 : 0] rs;
wire [4 : 0] rt;
wire [4 : 0] rd;
wire [5 : 0] func;
wire [5 : 0] branch_func;
wire [15 : 0] Immediate;
wire [5 : 0] sa;
wire [25 : 0] instr_index;

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

然后判断属于
R_Type\I_Type_Compute\J_Type\load\store\I_Type_Branch\Regimm 等类型。
接下来对于所有 R_Type 的指令进行译码，根据 Opcode 和 func 确定是哪一种。
由于 and,or 等在 Verilog 语法中属于特定含义，所以在命名时，
统一加上前缀 MIPS_(例如 MIPS_addu,MIPS_subu 等等)
在译码阶段需要确定 ALU 的输入和 ALUop,尽管我们的 ALU 是在执行阶段例化的。
同时在执行阶段需要例化一个移位器 Shifter,但在译码阶段同样需要确定移位器的输入和 Shiftop.
以及在 ID 阶段就可以确定 是否为 J-Type 类型，所以可以对
cpu_PC_jump_enable 赋值，也可以确定是否为 jr 或者 jalr 类型，所以对
cpu_PC_R 赋值，对 cpu_PC_jump ，cpu_PC_branch 和 cpu_PC_R_enable 赋值
（3）EXE 阶段
执行阶段需要例化 alu 和 shifter
（4）MEM 阶段
这一阶段的 load 和 store 指令在逻辑上略复杂，在问题二中给出详细的解释。
（5）WB 阶段

对封装的 RF_wen,RF_waddr,RF_wdata 进行赋值。
这里对几乎所有的指令进行了译码，这里的操作主要是使得代码的可读性增强，相应带来了代码行数的增加，缺少了一定的整合性。




波形变化说明：
（实验跑通之后就看不到波形图了）
记录之前查看波形检查到的问题：
1. Instruction[31 : 0]= 04c1ffb3
写寄存器的值不等于 Reference.
检查得到这是一条 bgez 指令，其中的 >=0 结果，对于两个操作数 00000000 和 ffffffff，相减结果本应该是 00000001,但实际上对于无符号数而言，应该是 00000000 < ffffffff,所以在 alu 的基础上，将操作数附加符号位，将 32 位操作数变为{1’b0,32 位操作数},当作 33 位有符号数运算。
2. Instruction[31 : 0] = 00063043
得到这是一条 sra 指令，最终发现写入寄存器值错误的原因是移位错误，进而发现 Shifter 的判断条件应该使用 func，笔误写成了 Opcode
3. Instruction[31:0]=0031880
发现 Shifter_shifterop 的判断应该使用 {32{}},而不是一位的 func 结果进行按位与操作。
4. Instruction[31 : 0]= a0450000
发现访存地址需要对齐的问题。写的地址是 00003ff1,Reference 是 0003ff0.
5. Instruction[31:0]=00c3302b
判断是 sltu 指令，发现 sltu 需要对 ALU 进行修改，并得到最终的结果。

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，逻辑仿真和 FPGA 调试过程中的难点等）

解：(一)本实验的难点在于理解 load 和 store 类型的指令。首先需要考虑 load 和 store 实际的功能，其主要目的在于将运算得到的数据，暂时存储在寄存器中，方便进行下一步的运算。
所以从功能角度考虑，load，将内存 mem 中的数据写入存储器 reg 时，应该写入 reg 的低位。这里我们以小端序为例，一个字(word)大小的数据，我们表示成
| 3 | 2 | 1 | 0 |
这里 |3 | 表示数据所在地址的最后两位为 2’b11。
下面分别介绍(1)load 中的 7 条指令 和 (2) store 中的 5 条指令
对于 load 指令，我们需要从内存 mem 中读取对应地址的数据：
实际上的原理是这样的，我们的实验二完成的是 除了理想内存(ideal_mem)之外的部分，整个实验二可以看作一个大的模块。
实验二需要给出 output [31 : 0]Address,即实验二的 simple_cpu 传递需要读\写内存的地址，然后 simple_cpu_top 的顶层模块，帮助完成根据地址读取内存的操作。
首先输出一个 32 位的地址（这里后期调代码的时候，发现需要实现对齐操作，即地址的最后两位必须为 0 ，但实际上我们进行 load 的写内存判断 和 write_strb 的写地址判断时，还是必须根据真实地址给出控制信号。
也就是写成下面的形式：

```
wire  [31 : 0] Address_real;
assign Address_real = (RF_rdata1 + Sign_extend_immediate);

assign Address = (Address_real &  { {30{1'b1}} , 2'b0} ) ;
```

处于对齐的考虑，在对 Address 赋值时，需要对最后两位置零，但实际进行控制时，利用 Address_real 来进行控制。
lb 说明，不管从内存 mem 中读取什么样的数据，都需要最后将其写入 reg 的最低位。
可以从加法计算的角度考虑，如果我们想要计算 1 + 1 = 2 ,并且把计算的结果 2 放入寄存器，后续进行 2 + 3 = 5 的操作。
如果一开始把 2 放到寄存器非最低位，即对于
Reg |3 | 2 | 1 | 0| ,我们把数字 2 放入 1 号地址处，相当于自动左移一位，在取出运算后，还需要进行右移恢复，这样的操作是麻烦且不必要的，指令集在设计的时候，就考虑到直接放在最低位的方便之处。
(1) lb 和 lbu
我们考虑地址，地址是相对于 mem 而言的，由于统一采用小端序，所以实际上 Mem |3 | 2 | 1 | 0|这四段，分别对应 Read_data[31 : 24]、Read_data[23 : 16]、Read_data[15 : 8]、Read_data[7 : 0]。
写内存，需要写 4 个字节，reg |3 | 2 | 1|0|
只有 reg 地址|0|处是从 mem 内存得到的 Read_data，其余高位为符号位扩展，或者为零扩展。
这里的逻辑就是：load 指令中 lb 和 lbu,（i）对于寄存器 reg 的每一个字节(这里专门指一个字中 reg |3 | 2| 1| 0|字节，都需要写。
(ii) 只有 reg |0| 处写的是从 mem 中读到的数，Read_data 具体取哪一个字节，是根据 Address_real 来确定的。

将上述的描述性语句转化为代码，也就是说:

```
wire [7 : 0] load_byte;
assign load_byte = ({8{Address_real[1 : 0] == 2'b00}} & Read_data[7 : 0])|
                   ({8{Address_real[1 : 0] == 2'b01}} & Read_data[15 : 8]) |
                   ({8{Address_real[1 : 0] == 2'b10}} & Read_data[23 : 16])|
                   ({8{Address_real[1 : 0] == 2'b11}} & Read_data[31 : 24]);

wire [31 : 0] load_wdata_byte;
assign load_wdata_byte = ({32{MIPS_lb}} & {{24{load_byte[7]}}, load_byte} ) |
                         ({32{MIPS_lbu}} & {{24{1'b0}} , load_byte });
```

(2) lh 和 lhu

在理解 lb 和 lbu 的基础上，也就是 mem 的内容必须写到 reg 的最低两个字节，但是写的内容依靠 Address_real 来确定 Read_data 的位数。

Reg 的高两个字节，同样需要写，但是是根据最低两个字节进行符号位扩展或者零扩展

```
wire [15 : 0] load_halfword;
assign load_halfword = ({16{Address_real[1 : 0] == 2'b00}} & Read_data[15 : 0]) |
                       ({16{Address_real[1 : 0] == 2'b10}} & Read_data[31 : 16]);

wire [31 : 0] load_wdata_halfword;
assign load_wdata_halfword = ({32{MIPS_lh}} & {{16{load_halfword[15]}}, load_halfword}) |
                             ({32{MIPS_lhu}} & {{16{1'b0}}, load_halfword});
```

（3）lw 最直接，直接写入即可。

（4）Lwl 需要注意

我们画出一个示意图，来表达 lwl 的真实含义。

这里代码的写法借鉴了 ideal_mem 中 byte_0,byte_1,byte_2,byte_3 的定义。

Address_real[1 : 0]=2’ b00

Mem |0| ----------------> reg |3|

Address_real[1 : 0] = 2’ b01

Mem |1|0| ----------------> reg |3|2|

Address_real[1 : 0] = 2 ‘b10

Mem |2|1|0| ------------> reg |3|2|1|

Address_real[1 : 0] = 2’ b11

Mem |3|2|1|0| ----------> reg |3|2|1|0|

就是说，从 Address_real 开始，取出所有 mem 中地址右边的数据，然后依次放到 reg 的最左边(最高位地址)

(5) lwr 的示意图

Address_real[1 : 0]=2’ b00

Mem |3|2|1|0| ----------------> reg |3|2|1|0|

Address_real[1 : 0] = 2’ b01

Mem |3|2|1| ----------------> reg |2|1|0|

Address_real[1 : 0] = 2 ‘b10

Mem |3|2| ------------> reg |1|0|

Address_real[1 : 0] = 2’ b11

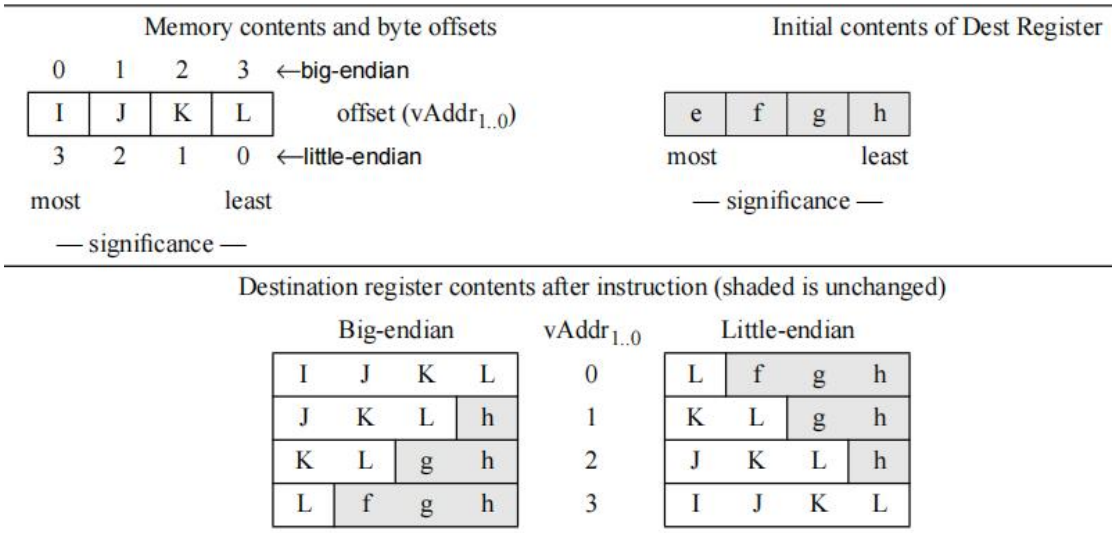
Mem |3| ----------> reg |0|

也就是从 Address_real 开始，取出所有 mem 中地址左边的数据，然后依次放到

reg 的最右边(从最低位的地址开始放）
即 Mem |3| ----------> reg |0|
表示把寄存器[31 : 24]字节的内容，写到 reg 的第[7 : 0]位。
判断逻辑如下：

```
wire [7 : 0] RF_byte_3;
wire [7 : 0] RF_byte_2;
wire [7 : 0] RF_byte_1;
wire [7 : 0] RF_byte_0;

wire [31 : 0] RF_load;

assign RF_byte_3 =  ({8{MIPS_lwl && Address_real[1 : 0] == 2'b11}} & Read_data[31 : 24])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b10}} & Read_data[23 : 16])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b01}} & Read_data[15 : 8])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b00}} & Read_data[7 : 0])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b00}} & Read_data[31 : 24]) |
                    ({8{MIPS_lwr && Address_real[1 : 0] != 2'b00}} & RF_rdata2[31 : 24]);


assign RF_byte_2 =  ({8{MIPS_lwl && Address_real[1 : 0] == 2'b11}} & Read_data[23 : 16])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b10}} & Read_data[15 : 8])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b01}} & Read_data[7 : 0])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b01}} & Read_data[31 : 24])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b00}} & Read_data[23 : 16]) |
                    ({8{(MIPS_lwl && Address_real[1 : 0] == 2'b00) || (MIPS_lwr && Address_real[1 : 0] == 2'b10)
                    || (MIPS_lwr && Address_real[1 : 0] == 2'b11)}} & RF_rdata2[23 : 16]);

assign RF_byte_1 =  ({8{MIPS_lwl && Address_real[1 : 0] == 2'b11}} & Read_data[15 : 8])  |
                    ({8{MIPS_lwl && Address_real[1 : 0] == 2'b10}} & Read_data[7 : 0])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b10}} & Read_data[31 : 24])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b01}} & Read_data[23 : 16])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b00}} & Read_data[15 : 8]) |
                    ({8{(MIPS_lwl && Address_real[1 : 0] == 2'b01) || (MIPS_lwl && Address_real[1 : 0] == 2'b00)
                    || (MIPS_lwr && Address_real[1 : 0] == 2'b11)  }} & RF_rdata2[15 : 8] );

assign RF_byte_0 =  ({8{MIPS_lwl && Address_real[1 : 0] == 2'b11}} & Read_data[7 : 0])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b11}} & Read_data[31 : 24])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b10}} & Read_data[23 : 16])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b01}} & Read_data[15 : 8])  |
                    ({8{MIPS_lwr && Address_real[1 : 0] == 2'b00}} & Read_data[7 : 0])  |
                    ( {8{MIPS_lwl && Address_real[1 : 0] != 2'b11} } & RF_rdata2[ 7 : 0]);

assign RF_load = ({32{MIPS_lb || MIPS_lbu}} & load_wdata_byte) |
                 ({32{MIPS_lh || MIPS_lhu}} & load_wdata_halfword ) |
                 ( {32{MIPS_lw}} & load_wdata_word) |
                 ({32{MIPS_lwl || MIPS_lwr}} & {RF_byte_3,RF_byte_2,RF_byte_1,RF_byte_0});
```

这一部分对应于 MIPS_vol2 中的示意图，上面的示意图以大端序，具有误导性。
小端序的示意图如下，略抽象：




介绍 store 命令，和 load 命令是完全相同的逻辑，是同一种操作的两方面描述。Store 需要从寄存器的低位开始读数据，然后写入 Mem 根据地址判断得到的不同位置：

(1) sb

从 reg 的最低位 RF_rdata2[7:0]读出数据，写到对应地址的位置。读数据都是从统一的 reg 最低位读取。

```
assign Write_strb = ({4{MIPS_sb && Address_real[1 : 0] == 2'b11}} & 4'b1000) |
                    ({4{MIPS_sb && Address_real[1 : 0] == 2'b10}} & 4'b0100) |
                    ({4{MIPS_sb && Address_real[1 : 0] == 2'b01}} & 4'b0010) |
                    ({4{MIPS_sb && Address_real[1 : 0] == 2'b00}} & 4'b0001) |
```

（2）sh

从 reg 的最低两位 RF_rdata[15 : 0] 读出数据，写到对应地址的位置。

```
({4{MIPS_sh && Address_real[1 : 0] == 2'b10}} & 4'b1100) |
({4{MIPS_sh && Address_real[1 : 0] == 2'b00}} & 4'b0011) |
```

半字的写(最高两位，地址为 2’b10,最低两位，地址为 2’b00)

（3）sw

直接进行读写

(4) swl

示意图如下:

(i) Address_real[1 : 0] = 2’b00

Reg |3| ----------> mem |0|

(ii)Address_real[1 : 0] = 2’b01

Reg |3|2| ----------> mem |1|0|

(iii)Address_real[1 : 0] = 2’b10

Reg |3|2|1| ----------> mem |2|1|0|

(iv)Address_real[1 : 0] = 2’b11

Reg |3|2|1|0| ----------> mem |3|2|1|0|

（5）swr

示意图如下:

(ii) Address_real[1 : 0] = 2’b00

Reg |3|2|1|0| ----------> mem |3|2|1|0|
(ii)Address_real[1 : 0] = 2’b01
Reg |2|1|0| ----------> mem |3|2|1|
(iii)Address_real[1 : 0] = 2’b10
Reg |1|0| ----------> mem |3|2|
(iv)Address_real[1 : 0] = 2’b11
Reg |0| ----------> mem |3|
表示从寄存器的最右侧开始拿数据，写到 reg 地址及左侧的所有地址内。
分为 4 个字节分别讨论：
这里再次强调，如果 Write_strb 为零，则按照 ideal_mem 中的控制，需要保持数据为原来内存中的数据不变。这些功能在 ideal_mem 中都封装好了，不需要在 simple_cpu 中再进行重复描述。

```
assign Mem_byte_3 = ({8{MIPS_sb && Address_real[1 : 0] == 2'b11}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sh && Address_real[1 : 0] == 2'b10}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_sw}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b11}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b00}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b01}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b10}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b11}} & RF_rdata2[7 : 0]);

assign Mem_byte_2 = ({8{MIPS_sb && Address_real[1 : 0] == 2'b10}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sh && Address_real[1 : 0] == 2'b10}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sw}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b10}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b11}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b00}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b01}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b10}} & RF_rdata2[7 : 0]);
```

```
assign Mem_byte_1 = ({8{MIPS_sb && Address_real[1 : 0] == 2'b01}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sh && Address_real[1 : 0] == 2'b00}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_sw}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b01}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b10}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b11}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b00}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b01}} & RF_rdata2[7 : 0]);

assign Mem_byte_0 = ({8{MIPS_sb && Address_real[1 : 0] == 2'b00}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sh && Address_real[1 : 0] == 2'b00}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_sw}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b00}} & RF_rdata2[31 : 24]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b01}} & RF_rdata2[23 : 16]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b10}} & RF_rdata2[15 : 8]) |
                    ({8{MIPS_swl && Address_real[1 : 0] == 2'b11}} & RF_rdata2[7 : 0]) |
                    ({8{MIPS_swr && Address_real[1 : 0] == 2'b00}} & RF_rdata2[7 : 0]);
```

（二）跳转指令
对于 I-Type-branch 类型的指令，和 Regimm 类型的指令，以及 R-type 类型中和跳转相关的 jr、jlar 指令，都需要在原有 PC 上 + 4，再进行 PC + offset，即鉴于分支延迟槽，需要写成 PC + 4 + offset。

三、 对讲义中思考题（如有）的理解和回答

四、 在课后，你花费了大约_____50_____小时完成此次实验。

五、 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，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

感受：理论课老师的思路讲得非常清晰，指令译码表在写代码的过程中起了非常重要的作用。

建议：感觉平台上跑测试仍然比较慢，在语法错误检查阶段，每次语法错误检查需要大概 10min，时间略长。

一开始写代码按照 5 大类指令进行写，最后合起来。由于是纯组合逻辑，所以当初并没有非常明确 5 个阶段的过程，但后续发现这样写比较混乱。最终感觉按照取指、译码、执行、访存、写回，来分层级（多周期 CPU 的思想），更加清晰。

感谢张老师非常细致的讲解（几乎是手把手说明单周期 cpu 的设计逻辑）。以及 ppt 的几页表格，非常清晰。