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。

1. 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：

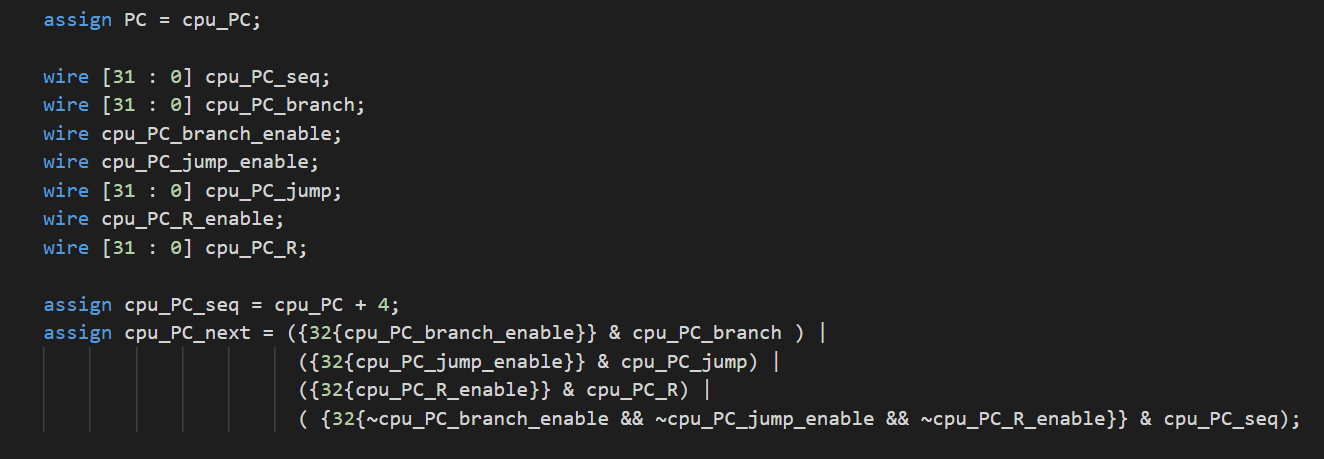
1. IF阶段

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。

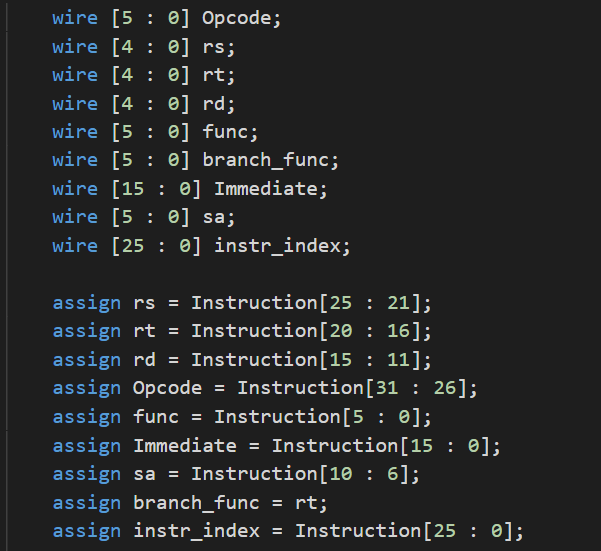
然后分别施加判断信号。



1. 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。



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

1. EXE阶段

执行阶段需要例化 alu 和 shifter

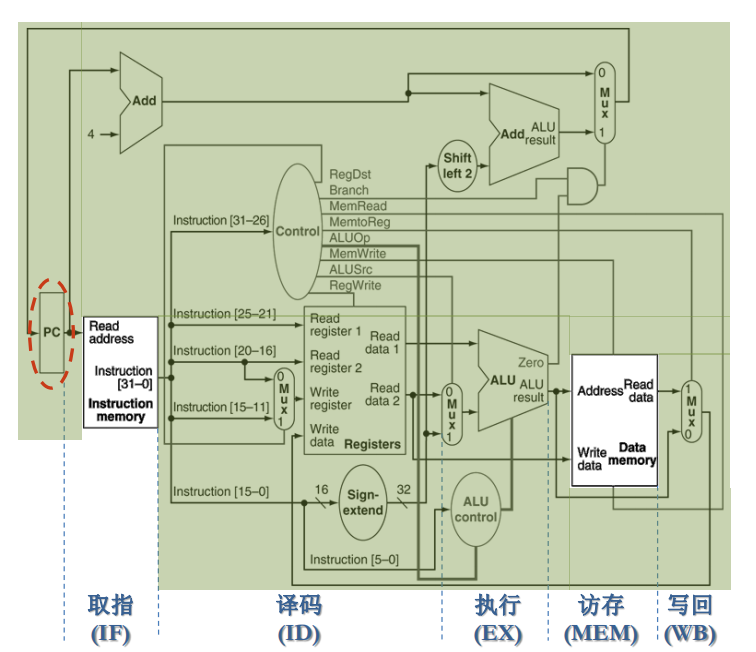
1. MEM阶段

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。

1. 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。

1. 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

1. 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。

1. 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.

1. 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。

1. 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

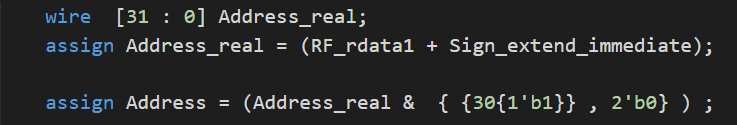
对于load指令，我们需要从内存mem中读取对应地址的数据：

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：



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]。

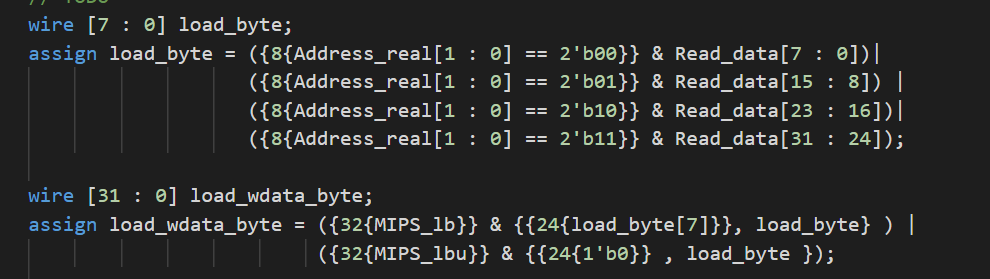
写内存，需要写4个字节，reg |3 | 2 | 1|0|

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。

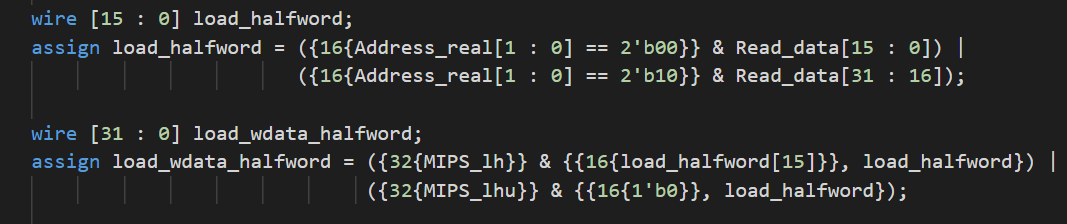
将上述的描述性语句转化为代码，也就是说:



1. 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



1. lw 最直接，直接写入即可。
2. 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)

1. 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|

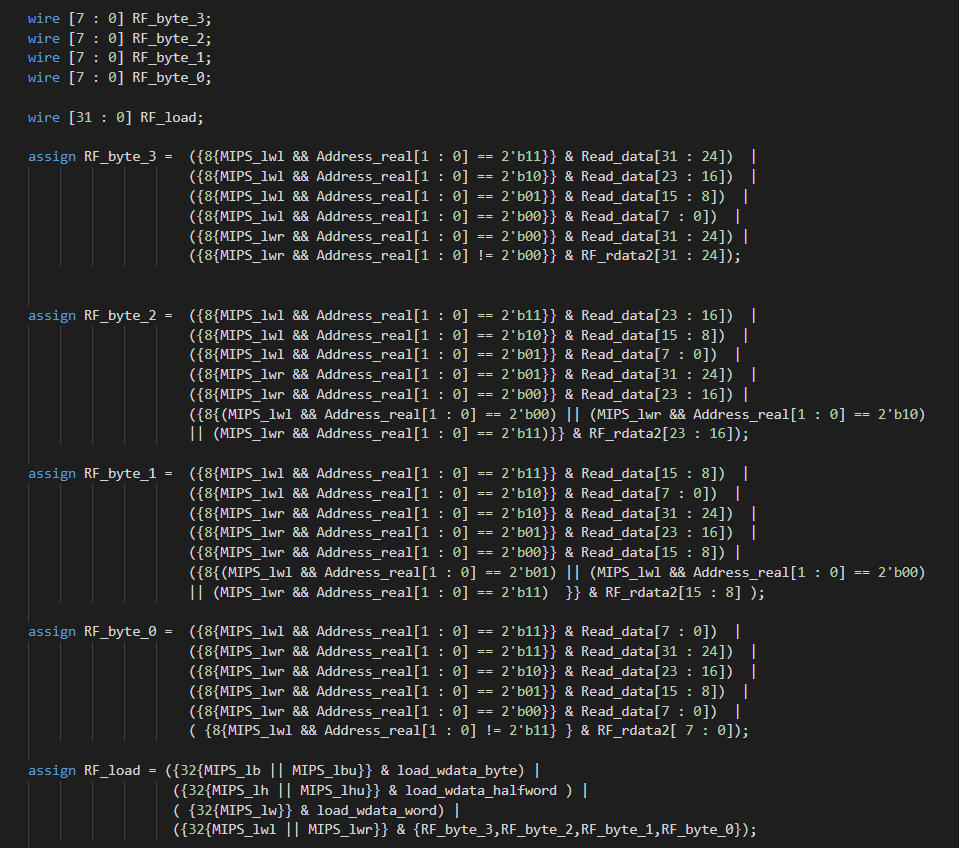
Address\_real[1 : 0] = 2’b11

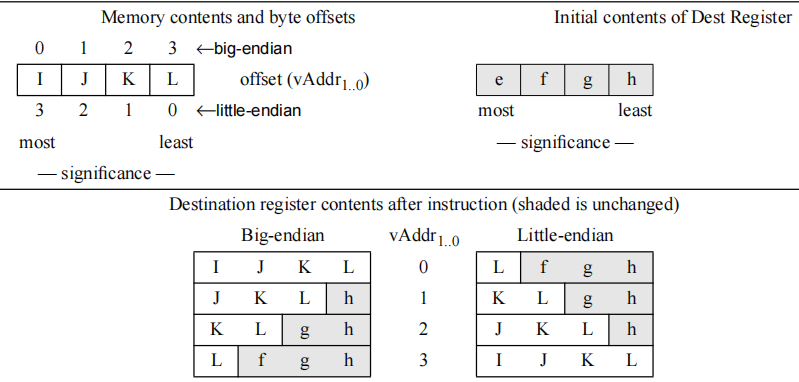
Mem |3| ---------> reg |0|

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：

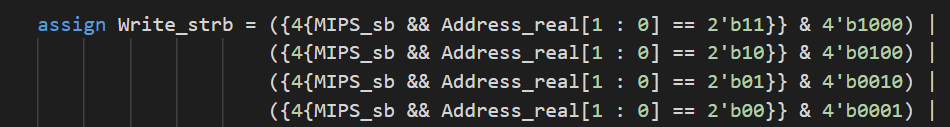
这一部分对应于MIPS\_vol2中的示意图，上面的示意图以大端序，具有误导性。小端序的示意图如下，略抽象：

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：

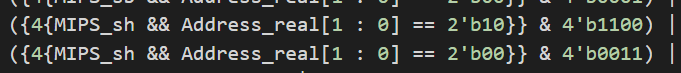
1. sb

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。



1. 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。



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:

1. 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|

1. swr

示意图如下:

1. 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|

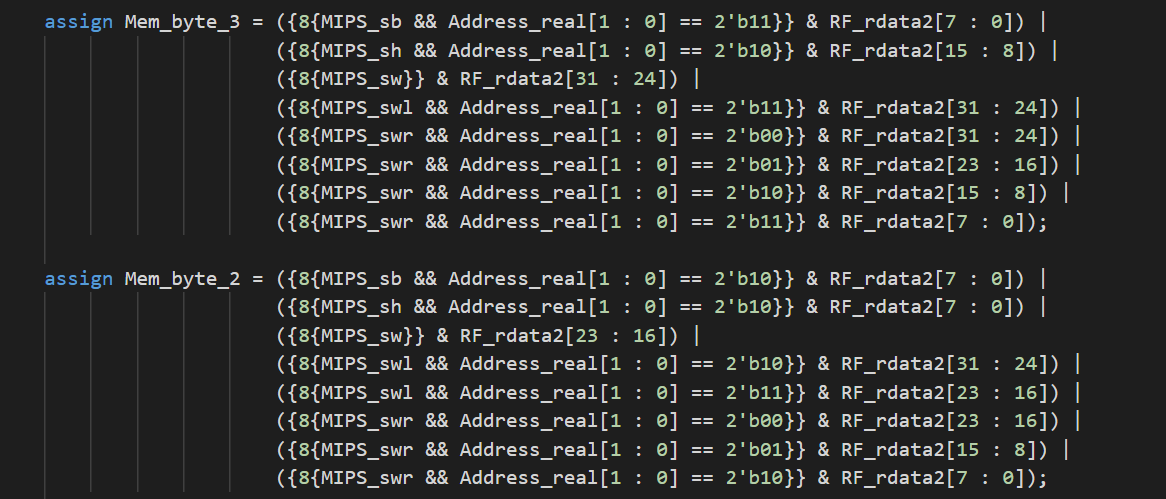
(iv)Address\_real[1 : 0] = 2’b11

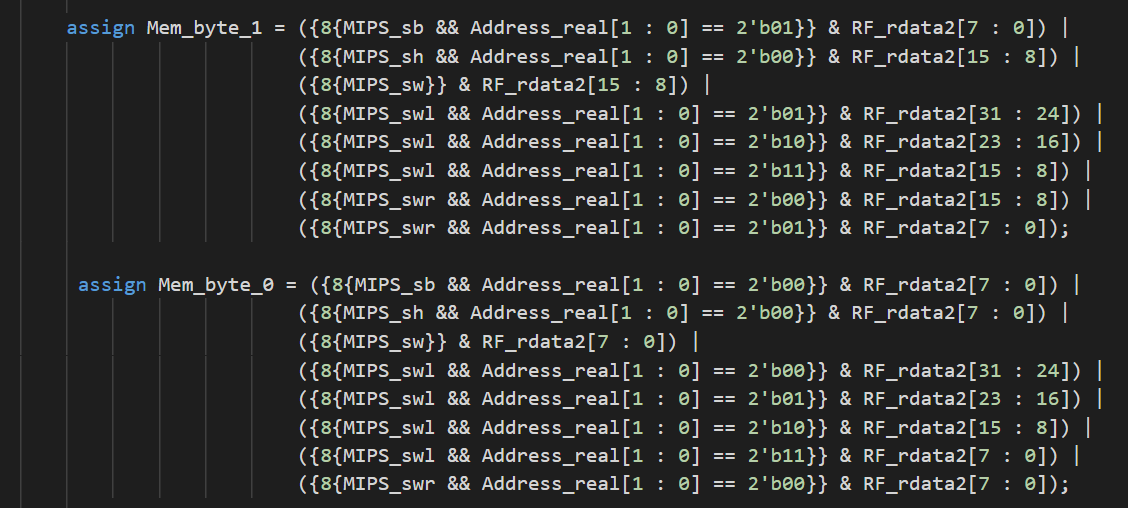
Reg |0| -------------> mem |3|

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。





1. 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。

1. 对讲义中思考题（如有）的理解和回答
2. 在课后，你花费了大约\_\_\_\_\_\_50\_\_\_\_\_小时完成此次实验。
3. 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。