Lab 2 report

PB21020485 吴敌

实验目的

1. 掌握五级流水线 CPU 的设计方法
2. 熟悉RISC-V的指令集和数据通路，了解其设计背后的思想
3. 进一步提高使用verilog编码和调试的能力，学会使用仿真调试

实验内容

阶段1，2：

实现如下指令：

SLLI、SRLI、SRAI、ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND、ADDI、SLTI、 SLTIU、XORI、ORI、ANDI、LUI、AUIPC 和

JALR、LB、LH、LW、LBU、LHU、SB、SH、SW、BEQ、BNE、BLT、BLTU、BGE、BGEU、JAL

实验过程

1. ControllerDecoder.v

对指令进行译码。首先做这个因为先确定整体指令执行的逻辑，然后通过这个再补充其余的各部分。

2. BranchDecision.v

根据 br_type 和 reg1，reg2 判断是否跳转，分别编写每种条件跳转的判断逻辑，如果需要跳转那么 $br = 1$，否则 $br = 0$.

3. ALU.v

根据 ALU_func 和Parameter对 op1，op2 进行不同逻辑运算操作（为简单实现，我选择了使用verilog的操作符使得vivado自动综合），得到 ALU_out.

4. ImmExtend.v

对立即数进行不同方式扩展，根据输入的 imm_type 判断立即数的类型，将指令中的立即数扩展成32位即可。

5. DataExtend.v

针对load指令的不同类型，将load得到的数据扩展。对于 Byte 类型，输入的 addr 是地址的最后两位，00就将此数做第一个字节扩展，01就将其做第二个字节扩展，10就将其做第三个字节扩展，11就将其做第四个字节扩展，同时根据要求做符号or零扩展；对于 Half 类型指令，类似Byte将其作为第1or2个半字做扩展；Word 类型指令不需要扩展。

6. NPCGenerator.v

根据输入的 jalr， jal，br 控制信号，确定下一条指令的地址 NPC；不跳转时NPC+4，jalr时NPC是 $jalr\_target$, 其他也根据其target跳转。

跳转信号的优先级很重要，需要先判断 $jalr$ 和 br 是否为 1 再判断 jal，因为 jal 的跳转在ID段，而 $ja1r$ 和 br 的判断和跳转在EX段。

7. Hazard.v

bubble 和 flush：

如果有 br 或 $jalr$ 的跳转信号，需要将前面的（ID段和EX段）的控制信号冲刷。 jal 的跳转信号，需要将ID段控制信号冲刷。

如果存在 load -use需要增加一条空指令。这种相关是在ID段和EX段判断出来的，所以在ID段和EX段 bubble 置1，IF段 flush 置1。

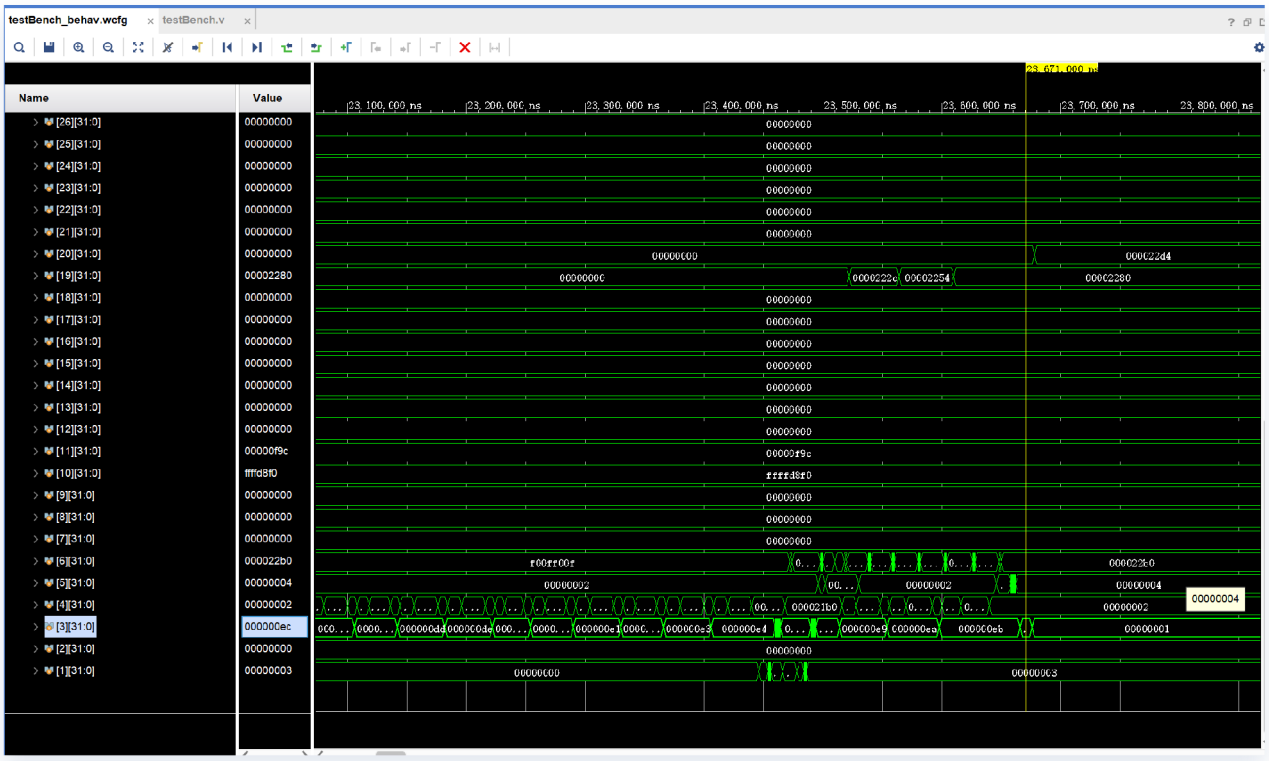
op1 or op2_se1：

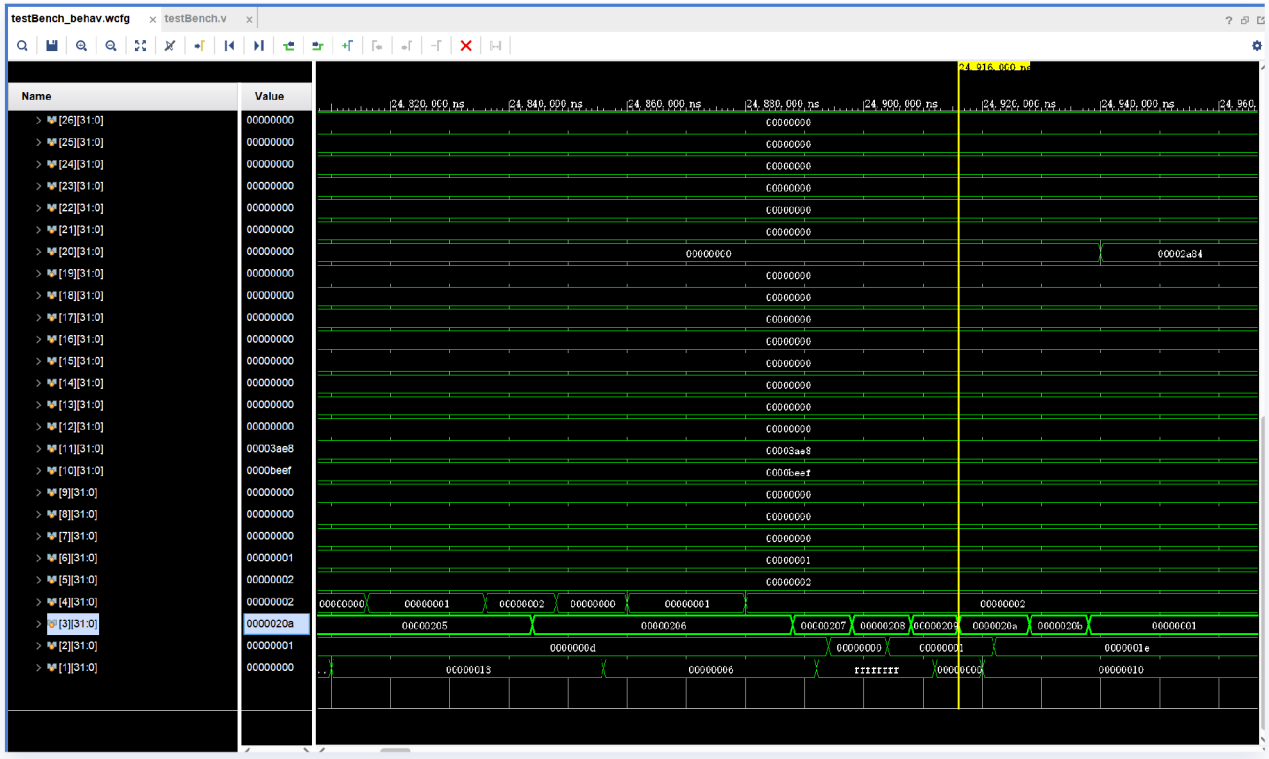
如果EX段的源寄存器和MEM段的目的（写入）寄存器相同，且读的寄存器不是 $r0$ ，则需要将该寄存器直接重定向到EX段。

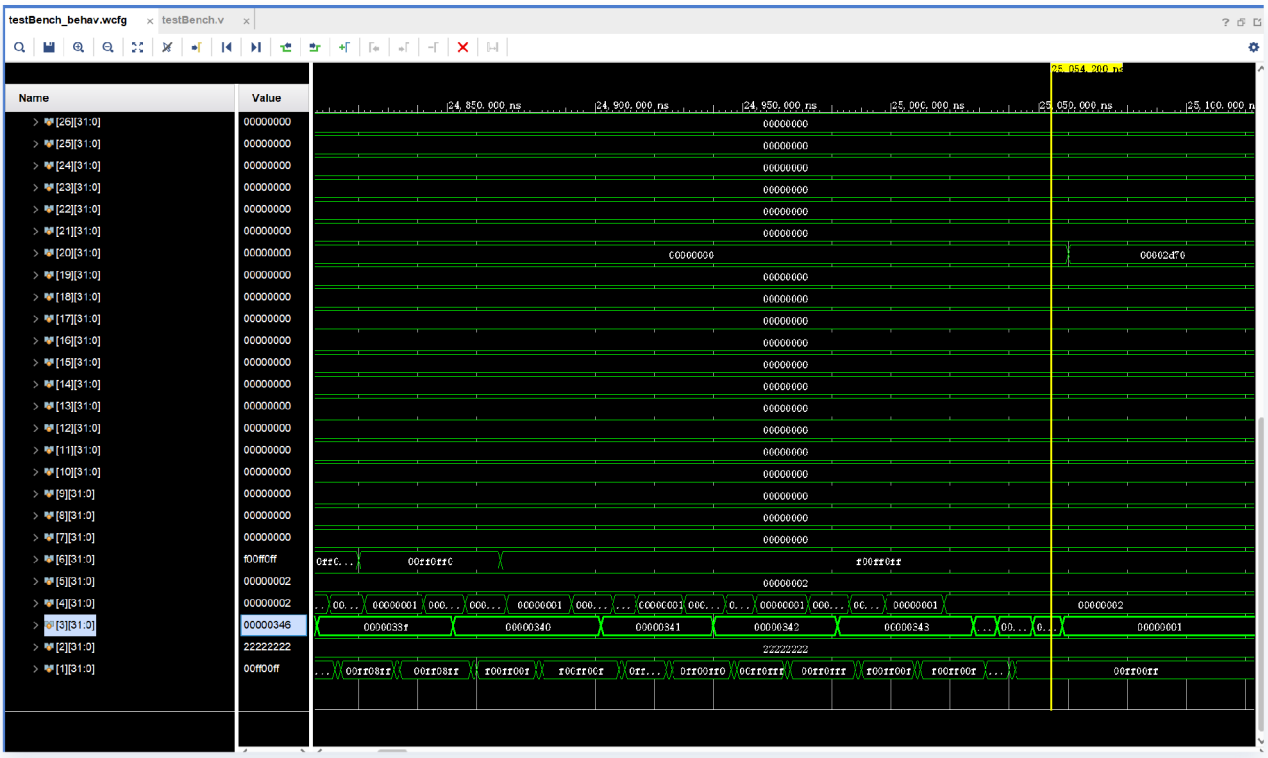
如果EX段的源寄存器和WB段的目的寄存器相同，从WB段重定向到EX段。

实验结果

1，2，3测试样例的仿真结果分别如下:


testBench_behav.wcfg
testBench.v
Name
Value
23, 671, 000 ns
23, 100, 000 ns
23, 200, 000 ns
23, 300, 000 ns
23, 400, 000 ns
23, 500, 000 ns
23, 600, 000 ns
23, 700, 000 ns
23, 800, 000 ns
[26][31:0] 00000000
[25][31:0] 00000000
[24][31:0] 00000000
[23][31:0] 00000000
[22][31:0] 00000000
[21][31:0] 00000000
[20][31:0] 00000000
[19][31:0] 00002280
[18][31:0] 00000000
[17][31:0] 00000000
[16][31:0] 00000000
[15][31:0] 00000000
[14][31:0] 00000000
[13][31:0] 00000000
[12][31:0] 00000000
[11][31:0] 0000f9c
[10][31:0] ffffd8f0
[9][31:0] 00000000
[8][31:0] 00000000
[7][31:0] 00000000
[6][31:0] 000022b0
[5][31:0] 00000004
[4][31:0] 00000002
[3][31:0] 000000ec
[2][31:0] 00000000
[1][31:0] 00000003
00000004


testBench_behav.wcfg
testBench.v
Name
Value
24, 916, 000 ns
24, 820, 000 ns
24, 840, 000 ns
24, 860, 000 ns
24, 880, 000 ns
24, 900, 000 ns
24, 920, 000 ns
24, 940, 000 ns
24, 960,
[26][31:0] 00000000
[25][31:0] 00000000
[24][31:0] 00000000
[23][31:0] 00000000
[22][31:0] 00000000
[21][31:0] 00000000
[20][31:0] 00000000
[19][31:0] 00000000
[18][31:0] 00000000
[17][31:0] 00000000
[16][31:0] 00000000
[15][31:0] 00000000
[14][31:0] 00000000
[13][31:0] 00000000
[12][31:0] 00000000
[11][31:0] 00003aa8
[10][31:0] 0000beef
[9][31:0] 00000000
[8][31:0] 00000000
[7][31:0] 00000000
[6][31:0] 00000001
[5][31:0] 00000002
[4][31:0] 00000002
[3][31:0] 0000020a
[2][31:0] 00000001
[1][31:0] 00000000



3号寄存器逐个累加，每加一说明通过一个test，最后寄存器的结果是1，说明通过全部的测试样例。

阶段3

实现CSR指令：CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI

实验过程

1. CSR_EX.V

段间寄存器，将CSR控制信号的值从ID段传递到EX段。

2. CSR_Regfile.v

异步读，同步写，相关计算已经在ALU中处理，这里只需正常读写，无额外处理。

3. ALU.V

增加一种计算类型。

CSRRS 指令源通用寄存器中所有为 1 的位所对应的CSR中的那一位都会被写1，用到 OR 操作。

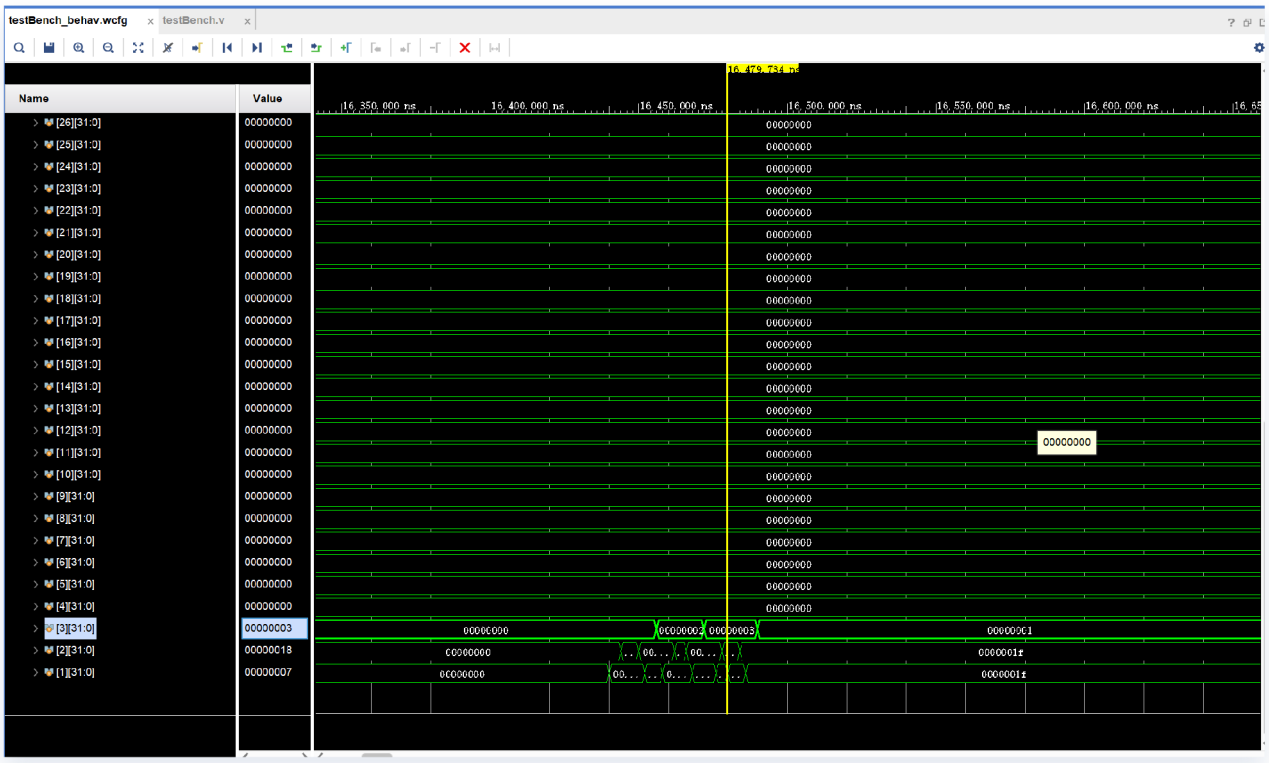
CSRRC 指令需要源通用寄存器中为 1 的位在CSR中所对应的位会被清零，用到的操作是 ALU_out = ~op1 & op2。

I类型指令只是把从寄存器读换为立即数。

4. ControllerDecoder.v

对CSR指令相关也需要进行译码，分别控制ALU操作类型以及一些读写寄存器信号。

实验结果



3号寄存器从0累加到3，然后回到1，说明通过全部样例。

实验难点

本次实验在调试过程中遇到的困难有：

1. 没有意识到CSR段寄存器没有实现，导致指令都没有正常执行，找了较长时间。
2. 实现CSR类指令开始没想在ALU特殊运算以及需要补充操作。

实验时间

阶段1和2：一天。

阶段3：半天。

报告：半天。

实验收获

经过本次实验，我对流水线CPU有了更深入的理解。

另外，不用从0开始完全手写CPU比COD舒服太多。

改进意见

希望可以告知一下在前两个阶段工作时要自行先补好CSR段寄存器代码。我以为这是第三阶段内容，而产生了一些不必要的bug。