东北大学2022秋

计算机组成原理

（基于RISC-V架构）

课程设计报告

班级：物联网2002班

小组成员：陆晓晓、阮雨荷、郎敏杰

学号：20203578、20203289、20204719

（排名顺序不分先后）

指导老师：刘学

2023年1月

**目录**

[一、 实验目的 3](#_Toc10646)

[1.1集成电路发展概述 3](#_Toc5897)

[1.2 CPU设计概述 3](#_Toc10660)

[二、 实验任务 4](#_Toc16396)

[三、 实验原理与设计方案 6](#_Toc1470)

[3.1 单周期CPU设计概述 6](#_Toc30082)

[3.2 38条指令的RTL描述 7](#_Toc26713)

[3.3 数据通路图 12](#_Toc31578)

[3.4 ALU设计 13](#_Toc10745)

[3.4.1 ALUctr设计 13](#_Toc10407)

[3.4.2 控制器设计 15](#_Toc2219)

[四、 实验验证与分析 29](#_Toc8549)

[4.1 R- 型指令 29](#_Toc1561)

[4.1.0 R- 型指令的数据通路 29](#_Toc32711)

[4.1.1 R- add指令 29](#_Toc15890)

[4.1.2 R- sub指令 31](#_Toc22816)

[4.1.3 R- sll指令 32](#_Toc15956)

[4.1.4 R- srl指令 33](#_Toc780)

[4.1.5 R- slt指令 34](#_Toc20893)

[4.1.6 R- sltu指令 36](#_Toc25264)

[4.1.7 R- xor指令 38](#_Toc79)

[4.1.8 R- or指令 39](#_Toc8029)

[4.1.9 R- and指令 40](#_Toc23337)

[4.2 I- 型指令 41](#_Toc2283)

[4.2.0 I- 型指令的数据通路 41](#_Toc25823)

[4.2.1 I- addi指令 41](#_Toc5590)

[4.2.2 I- xori指令 43](#_Toc19822)

[4.2.3 I- ori指令 44](#_Toc15458)

[4.2.4 I- andi指令 45](#_Toc3772)

[4.2.5 I- slli指令 46](#_Toc22711)

[4.2.6 I- srli指令 47](#_Toc7070)

[4.2.7 I- srai指令 48](#_Toc1172)

[4.2.8 I- slti指令 49](#_Toc14372)

[4.2.9 I- sltiu指令 51](#_Toc1998)

[4.3 B- 型指令仿真结果及分析 53](#_Toc30896)

[4.3.0 B- 型指令数据通路 53](#_Toc27035)

[4.3.1 B- beq指令 54](#_Toc25234)

[4.3.2 B- bne指令 55](#_Toc4407)

[4.3.3 B- blt指令 57](#_Toc667)

[4.3.4 B- bge指令 58](#_Toc6309)

[4.3.5 B- bltu指令 60](#_Toc12625)

[4.3.6 B- bgeu指令 61](#_Toc19107)

[4.4 J- 型指令仿真结果及分析 63](#_Toc14751)

[4.4.0 J- 型指令的数据通路 63](#_Toc19715)

[4.4.1 J- jalr指令 64](#_Toc32318)

[4.4.2 J- jal指令 65](#_Toc17160)

[4.5 U- 型指令仿真结果及分析 66](#_Toc20349)

[4.5.0 U- 型指令的数据通路 66](#_Toc11215)

[4.5.1 U- lui指令 66](#_Toc23703)

[4.5.2 U- auipc指令 68](#_Toc25589)

[4.6 L- 型指令仿真结果及分析 69](#_Toc22612)

[4.6.0 L- 型指令的数据通路 69](#_Toc17135)

[4.6.1 L- lw指令 69](#_Toc32121)

[4.6.2 L- lb指令 71](#_Toc7420)

[4.6.3 L- lh指令 72](#_Toc22141)

[4.6.4 L- lbu指令 73](#_Toc4496)

[4.6.5 L- lhu指令 74](#_Toc23170)

[4.7 S- 型指令仿真结果及分析 75](#_Toc3869)

[4.7.1 S- sw指令 75](#_Toc16018)

[4.7.2 S- sb指令 76](#_Toc18200)

[4.7.3 S- sh指令 77](#_Toc13573)

[五、 电路仿真 78](#_Toc7575)

[六、 总结与讨论 78](#_Toc28584)

[七、 心得与体会 79](#_Toc17853)

[7.1 陆晓晓心得体会 79](#_Toc19732)

[7.2 郎敏杰心得体会 80](#_Toc3838)

[7.3 阮雨荷心得体会 81](#_Toc9058)

# 实验目的

## 1.1集成电路发展概述

在集成电路设计与生产发展到今天，人们设计数字电路系统的方法也发生了

巨大的变化，以前设计数字电路均使用小规模（SSI）、中规模（MSI）电路。现

在设计数字电路用大规模（LSI）或超大规模（VLSI）电路。原 Intel 公司的总裁

戈登·摩尔(Gordon Moore)先生，大约在三十年前就观察到，集成电路技术正在以单个芯片上集成的晶体管数量每一年半或两年就翻一番的惊人速度发展着。这种现象，俗称摩尔定律，直到今天仍在延续。无疑，这种技术将对人们的生活各方面产生巨大的影响。对大多数的硬件产品来说，设计者有二种选择，一种是设计能在单个芯片上实现的电路，一种是设计安装在一块印制电路板(PCB)上由多个芯片实现的电路。实现这样的电路常用到的芯片主要有三种：标准芯片、可编

程逻辑器件、定制芯片。

## 1.2 CPU设计概述

在计算机设计中，CPU 的设计是十分重要的部分，但如何设计是摆在每一个

设计者面前着重要考虑的问题。一般来讲，要设计 CPU，首先要确定是设计通

用 CPU，还是设计专用 CPU。一旦被确定，接下来就要考虑是要 CPU 与编译软

件除指令编译外相对独立（软硬件可相对独立设计，软件在 CPU 上运行效率低

一些），还是要编译软件除指令编译外，还要协助完成 CPU 对指令在运行中的一些优化工作（软件在 CPU 上运行效率提高，软硬件设计相互牵制较大）。Intel

8086CPU 属于前者，MIPS CPU 属于后者。指令系统的设计，可根据实际任务的要求来选择所需的指令，确定使用何种指令结构、指令格式和操作码。确定后，

再对每一条指令进行仔细分析，把每条指令执行过程分成若干个状态或段，把所

有指令的状态或段进行综合，并归纳成最终需要的若干个状态或段，以便后面的

设计。

在单周期 CPU 设计中，因是单周期设计，所以不考虑把每条指令分成若干

个状态或段，而只关注在 1 个时钟周期内能不能完成就可以了。画出带控制信号的单周期数据通路图，列出指令译码表，画出指令译码图，并与带控制信号的单周期数据通路图的信号对应相连。单周期 CPU 在实际的计算机设计中基本不使用。

在多周期 CPU 设计中，首先用把所有指令的状态进行综合，并归纳成最终

需要的若干个状态，画出指令流程的状态图（这里我们给出一个实际状态图的例

子见图 1.2.1），根据状态图设计状态机。然后画出带控制信号的多周期数据通路

图，列出指令译码表，最后设计控制译码器，将控制译码器信号与多周期数据通

路图上的控制信号对应相连。注意在多周期 CPU 设计中，有可能会出现不同状

态下，做的是同一种操作。

在流水线CUP设计中，首先对每一条指令进行仔细分析，把每条指令执行过

程分成若干个段，把所有指令的段进行综合，并归纳成最终需要的若干个段。例

如：

取指(IF) : 从指令存储器取指令并进行顺序 PC 下地址计算

取数和译码 (ID) : 寄存器取数，同时对指令进行译码

执行 (EX) : 指令执行或计算内存单元地址

读写存储器(MEM) : 从数据存储器中读取数据或将数据存入存储器中

写寄存器 (WB): 将数据写到寄存器中

# 实验任务

掌握设计单周期CPU的具体流程。

具体完成的指令：

addi、slti、sltiu、xori、slli、srli、srai、ori、andi、lb、lh、lw、lbu、lhu、add、sub、sll、slt、sltu、xor、srl、sra、or、and、lui、auipc、sb、sh、sw、beq、bne、blt、bge、bltu、bgeu、jal、Jalr

开发工具：Vivado

采用的语言：verilog

涉及的指令类型： I型， R型，B型，U型，J型，S型。

详细：

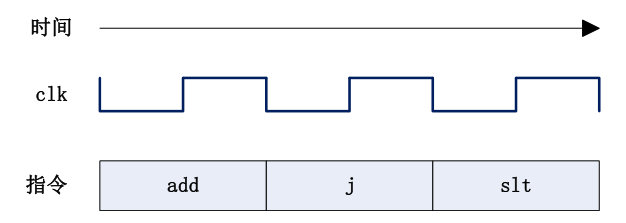
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |
| 31 25 24 20 19 15 14 12 | | | | | 11 | 0 |  |
| imm[31:12] | | | | | rd | 0110111 | U lui |
| imm[31:12] | | | | | rd | 0010111 | U auipc |
| imm[20|10:1|11|19:12] | | | | | rd | 1101111 | J jal |
| imm[11:0] | | | rs1 | 000 | rd | 1100111 | I jalr |
| imm[12|10:5] | | rs2 | rs1 | 000 | imm[4:1|11] | 1100011 | B beq |
| imm[12|10:5] | | rs2 | rs1 | 001 | imm[4:1|11] | 1100011 | B bne |
| imm[12|10:5] | | rs2 | rs1 | 100 | imm[4:1|11] | 1100011 | B blt |
| imm[12|10:5] | | rs2 | rs1 | 101 | imm[4:1|11] | 1100011 | B bge |
| imm[12|10:5] | | rs2 | rs1 | 110 | imm[4:1|11] | 1100011 | B bltu |
| imm[12|10:5] | | rs2 | rs1 | 111 | imm[4:1|11] | 1100011 | B bgeu |
| imm[11:0] | | | rs1 | 000 | rd | 0000011 | I lb |
| imm[11:0] | | | rs1 | 001 | rd | 0000011 | I lh |
| imm[11:0] | | | rs1 | 010 | rd | 0000011 | I lw |
| imm[11:0] | | | rs1 | 100 | rd | 0000011 | I lbu |
| imm[11:0] | | | rs1 | 101 | rd | 0000011 | I lhu |
| imm[11:5] | | rs2 | rs1 | 000 | imm[4:0] | 0100011 | S sb |
| imm[11:5] | | rs2 | rs1 | 001 | imm[4:0] | 0100011 | S sh |
| imm[11:5] | | rs2 | rs1 | 010 | imm[4:0] | 0100011 | S sw |
| imm[11:0] | | | rs1 | 000 | rd | 0010011 | I addi |
| imm[11:0] | | | rs1 | 010 | rd | 0010011 | I slti |
| imm[11:0] | | | rs1 | 011 | rd | 0010011 | I sltiu |
| imm[11:0] | | | rs1 | 100 | rd | 0010011 | I xori |
| imm[11:0] | | | rs1 | 110 | rd | 0010011 | I ori |
| imm[11:0] | | | rs1 | 111 | rd | 0010011 | I andi |
| 0000000 | | shamt | rs1 | 001 | rd | 0010011 | I slli |
| 0000000 | | shamt | rs1 | 101 | rd | 0010011 | I srli |
| 0100000 | | shamt | rs1 | 101 | rd | 0010011 | I srai |
| 0000000 | | rs2 | rs1 | 000 | rd | 0110011 | R add |
| 0100000 | | rs2 | rs1 | 000 | rd | 0110011 | R sub |
| 0000000 | | rs2 | rs1 | 001 | rd | 0110011 | R sll |
| 0000000 | | rs2 | rs1 | 010 | rd | 0110011 | R slt |
| 0000000 | | rs2 | rs1 | 011 | rd | 0110011 | R sltu |
| 0000000 | | rs2 | rs1 | 100 | rd | 0110011 | R xor |
| 0000000 | | rs2 | rs1 | 101 | rd | 0110011 | R srl |
| 0100000 | | rs2 | rs1 | 101 | rd | 0110011 | R sra |
| 0000000 | | rs2 | rs1 | 110 | rd | 0110011 | R or |
| 0000000 | | rs2 | rs1 | 111 | rd | 0110011 | R and |
| 0000 | pred | succ | 00000 | 000 | 00000 | 0001111 | I fence |
| 0000 | 0000 | 0000 | 00000 | 001 | 00000 | 0001111 | I fence.i |
| 000000000000 | | | 00000 | 00 | 00000 | 1110011 | I ecall |
| 000000000000 | | | 00000 | 000 | 00000 | 1110011 | I ebreak |
| csr | | | rs1 | 001 | rd | 1110011 | I csrrw |
| csr | | | rs1 | 010 | rd | 1110011 | I csrrs |
| csr | | | rs1 | 011 | rd | 1110011 | I csrrc |
| csr | | | zimm | 101 | rd | 1110011 | I csrrwi |
| csr | | | zimm | 110 | rd | 1110011 | I cssrrsi |
| csr | | | zimm | 111 | rd | 1110011 | I csrrci |

# 实验原理与设计方案

## 3.1 单周期CPU设计概述

计算机的性能由三个关键因素决定：指令数目，时钟周期，CPI。其中，指令数目由编译器和指令集决定；时钟周期和 CPI 由处理器的设计和实现决定。单周期处理器每条指令在一个时钟周期内完成，所以 CPI 为 1，而时钟周期往往径长，通常取最长的指令周期。

单周期 CPU 的特点是每条指令的执行叧需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由二每个时钟上升沿时更新地址，因此要在上升沿到来乊前完成所有运算，而这所有的运算除可以利用一个下降沿外，叧能通过组吅逡辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。



图：CPU指令执行时序图

## 3.2 38条指令的RTL描述

每条指令的RTL描述如下表：

|  |  |  |
| --- | --- | --- |
| **R I U S B J 型指令功能的RTL描述** | | |
| **指令** | **功能** | **说明** |
| add rd, rs1, rs2 | M[PC], PC <== PC+4   R[rd] <== R[rs1] + R[rs2] | 从PC所指的内存单元中取指令，并PC加4从rs1、rs2中取数后相加，结果送rd(不进行溢出判断) |
| slt rd, rs1, rs2 | if ( R[rs1]<R[rs2] ) R[rd] <== 1 else R[rd] <== 0 | 从rs1、rs2中取数后按带符号数来判断两数大小， 小于则rd置1， 否则rd中清0 |
| sltu rd, rs1, rs2 | if ( R[rs1]<R[rs2] ) R[rd] <== 1 else R[rd] <== 0 | 从rs1、rs2中取数后按带符号数来判断两数大小， 小于则rd置1， 否则rd中清0 |
| sub rd,rs1,rs2 | R(rd)=R(rs1)-R(rs2) | 减 |
| xor rd,rs1,rs2 | R(rd)=R(rs1)^SEXT(imm) | 异或 |
| sra rd,rs1,rs2 | R(rd)=R(rs1)>>R(rs2) | 算数右移，空位最高位填充 |
| and rd,rs1,rs2 | R(rd)=R(rs1) & R(rs2) | 与 |
| or rd,rs1,rs2 | R(rd)=R(rs1) | R(rs2) | 或 |
| sll rd,rs1,rs2 | R(rd)=R(rs1)<<R(rs2) | 逻辑左移，空位0填充 |
| srl rd,rs1,rs2 | R(rd)=R(rs1)>>R(rs2) | 逻辑右移，空位0填充 |
| addi rd, rs1, imm12 | M[PC], PC <== PC+4   R[rd] <== R[rs1] + SEXT(imm12) | 从PC所指的内存单元中取指令，并PC加4从rs1中取数后加imm12符号扩展，结果送rd(不进行溢出判断) |
| slti rd, rs1, imm12 | if ( R[rs1]<SEXT(imm12) ) R[rd] <== 1 else R[rd] <== 0 | 从rs1中取数 与 有符号扩展的imm12 进行带符号大小判断， 小于则rd置1， 否则rd中清0 |
| sltiu rd, rs1, imm12 | if ( R[rs1]<ZEXT(imm12) ) R[rd] <== 1 else R[rd] <== 0 | 从rs1中取数 与 有符号扩展的imm12 进行无符号大小判断， 小于则rd置1， 否则rd中清0 |
| xori rd, rs1, imm12 | R[rd] <== R[rs1]^SEXT(imm12) | 从rs1中取数 与 有符号扩展的imm12 进行按位或运算 |
| ori rd, rs1, imm12 | R[rd] <== R[rs1]|SEXT(imm12) | 从rs1中取数 与 有符号扩展的imm12 进行按位异或运算 |
| andi rd, rs1, imm12 | R[rd] <== R[rs1]&SEXT(imm12) | 从rs1中取数 与 有符号扩展的imm12 进行按位与运算 |
| slli rd, rs1, shamt | R[rd] <== R[rs1] << SEXT(shamt) | 逻辑左移 低位补0 把寄存器 R[rs1]左移 shamt 位，空出的位置填入 0，结果截为 32 位，进行有符号扩展后写入x[rd]。仅当 shamt[5]=0 时，指令才是有效的。 |
| srli rd, rs1, shamt | R[rd] <== R[rs1] >> SEXT(shamt) | 逻辑右移 高位补0 |
| srai rd, rs1, shamt | R[rd] <== R[rs1] >> SEXT(shamt) | 算术右移 高位补符号位 |
| lb rd, rs1, imm12 | Addr <== R[rs1] + SEXT(imm12) R[rd] <== M[Addr] | M[Addr] 取后8位进行有符号扩展 |
| lh rd, rs1, imm12 | Addr <== R[rs1] + SEXT(imm12) R[rd] <== M[Addr] | M[Addr] 取后16位进行有符号扩展 |
| lw rd, rs1, imm12 | Addr <== R[rs1] + SEXT(imm12) R[rd] <== M[Addr] | 从rs1取数、将imm12进行符号扩展，然后将两者相加 结果作为访存地址Addr，将Addr中取数并送rd |
| lbu rd, rs1, imm12 | Addr <== R[rs1] + SEXT(imm12) R[rd] <== M[Addr] | M[Addr] 取后8位进行无符号扩展 |
| lhu rd, rs1, imm12 | Addr <== R[rs1] + SEXT(imm12) R[rd] <== M[Addr] | M[Addr] 取后16位进行无符号扩展 |
| beq rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond == 0) PC += 2 X SEXT(imm12) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经符号扩展后值的2倍 |
| bne rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond != 0) PC += 2 X SEXT(imm12) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经符号扩展后值的2倍 |
| blt rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond < 0) PC += 2 X SEXT(imm12) (有符号) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经符号扩展后值的2倍 |
| bge rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond >= 0) PC += 2 X SEXT(imm12) (有符号) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经符号扩展后值的2倍 |
| bltu rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond < 0) PC += 2 X SEXT(imm12) (无符号) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经无符号扩展后值的2倍 |
| bgeu rs1, rs2, imm12 | Cond = R[rs1] - R[rs2] if (Cond >= 0) PC += 2 X SEXT(imm12) (无符号) | 做减法以比较rs1和rs2中内容大小，并计算下条指令地址， 根据比较结果修改PC，采用相对寻址， 基准地址为当前指令地址（即PC）， 偏移量为立即数imm12经无符号扩展后值的2倍 |
| jalr rd，rs1，imm12 | PC = R[rs1]+SEXT[imm12]; R[rd] = PC+4 | PC+4送寄存器，实现32位地址空间的相对跳转和绝对跳转 |
| jal rd, imm20 | R[rd] = PC + 4 PC += 2 X SEXT(imm12) | PC+4结果送寄存器，然后计算下条指令地址。 转移地址采用相对寻址基准地址为当前PC， 偏移量为立即数经过符号扩展后的两倍 |
| lui rd, imm20 | R[rd] = imm20 || 000H | 将立即数imm20存到寄存器rd的高20位上 |
| auipc rd，imm20 | R[rd] = imm20 || 000H+PC | 将20位立即数加到PC的高20位上 |
| sw rs1, rs2, imm12 | Addr = R[rs1] + SEXT(imm12) M[Addr] = R[rs2] | 从rs1取数、将imm12进行符号扩展，然后将两者相加 结果作为访存地址Addr，将rs2送Addr中 |
| sb rs1,rs2,imm12 | Addr <== R[rs1] + SEXT(imm12) M[Addr] <== R[rs2] | R[rs2] 取低8位写入存储单元，采用小端方式存放。 |
| sh rs1,rs2,imm12 | Addr <== R[rs1] + SEXT(imm12) M[Addr] <== R[rs2] | R[rs2] 取低16位写入存储单元，采用小端方式存放。 |
| sw rs1,rs2,imm12 | Addr <== R[rs1] + SEXT(imm12) M[Addr] <== R[rs2] | 从rs1取数、将imm12进行符号扩展，然后两者相加， 结果作为访存地址Addr,将rs2送Addr中 |

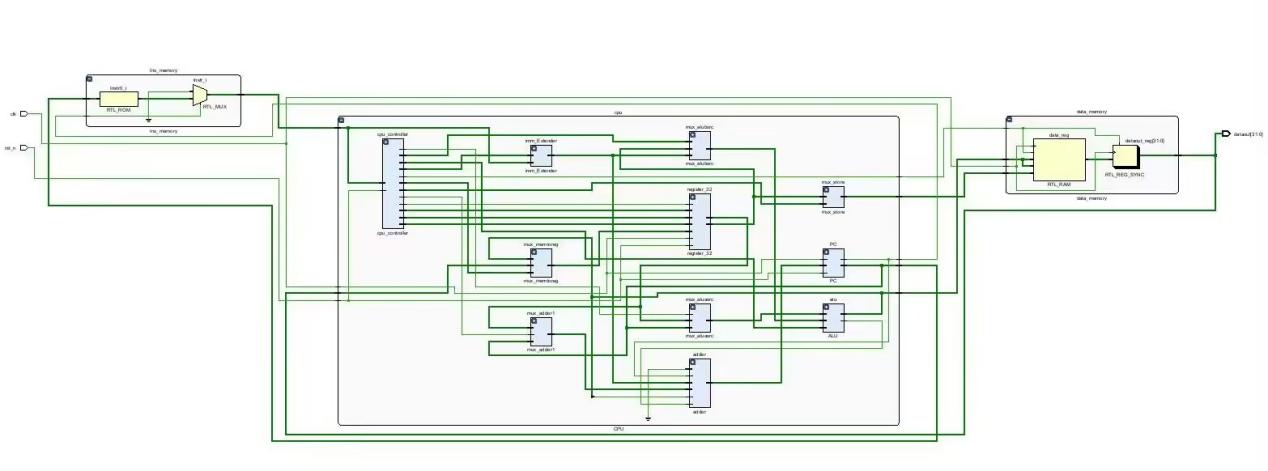
## 3.3 数据通路图

清楚了每条指令的功能后，我们就知道了执行这条指令需要的那些原件。首先必须要有的是指令寄存器。既然有指令寄存器，就要有从寄存器里取出具体指令的器件，即取指令部件。

指令取出来之后我们要怎样执行它，这还需要一个翻译指令的过程，即一条指令他的每一小段对应的具体意义是什么。于是就需要一个指令译码器来执行这个任务。

我们都知道整个数据通路错综复杂，每一条指令怎么样走出属于他们自己的数据通路呢？没对每个多路选怎的的时候需要有哪些信号来控制呢？控制单元部件就解决了这一难题。他根据最初输入的指令选取opcode, func3, func7指令段。最终会得到每条指令对应的一串各个控制信号的值。

在执行的过程中，我么们还需要对一些运算结果进行存储。这个功能就交给了寄存器和存储器。还有一些指令是和立即数进行运算。而我们进行的运算都是32位的形式。这就需要有一个立即数扩展部件。



图：数据通路图（可放大查看）

在执行指令时会遇到不同的运算，这些运算数据大部分是在算术逻辑部件(ALU)中进行的。

在一个cop中，会遇到很多种多个输入单个输出的情况。这需要多路选择器部件来完成这一任务。我们是从简单的R型指令的数据通路开始构建其数据通路，在此基础上再构建I型指令的数据通路。因为I型指令和R型指令和相近，就只许添加一个扩展部件。接着我们添加U型指令的数据通路，这个指令执行的时候再R, I指令的基础上就可以运行。只需改变一些信号的值。接着我们添加了S型指令，分为load和store两类。再接着添加B型指令。组后添加J型指令。

## 3.4 ALU设计

### 3.4.1 ALUctr设计

ALUctr设计为6位控制。在ALU内部的小信号的控制OPctr设计为4位控制。统计之后进行编码然后计算通过总信号ALUctr计算各分信号的逻辑表达式。

设计表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ALUctr[5:0]** | **操作类型** | **SUBctr** | **SIGctr** | **Opctr[3:0]** | **Opctr 的含义** |
| 0 0 0 0 0 0 | add | 0 | x | 0 0 0 0 | 选择加法器的输出结果 |
| 0 0 0 0 0 1 | 未用 |  |  |  |  |
| 0 0 0 0 1 0 | slt | 1 | 1 | 0 0 1 1 | 选择小于置位结果输出 |
| 0 0 0 0 1 1 | sltu | 1 | 0 | 0 0 1 1 | 选择小于置位结果输出(无符号) |
| 0 0 0 1 0 0 | 未用 |  |  |  |  |
| 0 0 0 1 0 1 | 未用 |  |  |  |  |
| 0 0 0 1 1 0 | or | x | x | 0 0 0 1 | 选择“按位或”结果输出 |
| 0 0 0 1 1 1 | 未用 |  |  |  |  |
| 0 0 1 0 0 0 | sub | 1 | x | 0 0 0 0 | 选择加法器的输出结果 |
| 1 1 0 0 0 0 | bge | 1 | 1 | 1 1 0 0 | 选择大于等于置位结果输出 |
| 1 1 1 0 0 0 | bgeu | 1 | 0 | 1 1 1 0 | 选择大于等于置位结果输出(无符号) |
| 其余 | 未用 |  |  |  |  |
| 0 1 0 0 0 0 | xor | x | x | 0 1 0 0 | 选择“按位异或”结果输出 |
| 0 1 0 1 0 0 | and | x | x | 0 1 0 1 | 选择“按位与”结果输出 |
| 0 1 1 0 0 0 | sll | x | x | 0 1 1 0 | 逻辑左移 |
| 0 1 1 1 0 0 | srl | x | x | 0 1 1 1 | 逻辑右移 |
| 1 0 0 0 0 0 | sra | x | x | 1 0 0 0 | 算数右移 |
| 0 0 1 1 1 1 | srcB | x | x | 0 0 1 0 | 直接输出B |

**逻辑表达式：**

SUBctr = (~ALUctr[3] & ~ALUctr[2] & ALUctr[1]) | ALUctr[3] | ALUctr[5]

SIGctr = (ALUctr[1] & ~ALUctr[0]) | (~ALUctr[3] & ~ALUctr[0])

Opctr[0] = (ALUctr[2] & ALUctr[1] & ~ALUctr[0]) | (ALUctr[2] & ~ALUctr[1] & ~ALUctr[0]) |(~ALUctr[2] & ALUctr[1])

Opctr[1] = (ALUctr[2] & ALUctr[1] & ALUctr[0]) | (~ALUctr[3] & ~ALUctr[2] & ALUctr[1]) |ALUctr[4] & ALUctr[3] & ~ALUctr[1]) | (ALUctr[4] & ALUctr[3] & ALUctr[1])

Opctr[2] = ALUctr[4]

Opctr[3] = ALUctr[5]

**关键代码设计：**

1. `timescale 1ns / 1ps
2. //////////////////////////////////////////////////////////////////////////////////
3. // Company:
4. // Engineer:
5. //
6. // Create Date: 2022/11/12 00:51:30
7. // Design Name:
8. // Module Name: alu\_control
9. // Project Name:
10. // Target Devices:
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////////////////////////////////////////////////

23. module alu\_control(
24. input [5:0] ALUctr,
25. output [3:0] OPctr,
26. output subctr,
27. output sigctr
28. );
29. assign subctr = (~ALUctr[3] & ~ALUctr[2] & ALUctr[1]) | ALUctr[3] | ALUctr[5];
30. assign SIGctr = (ALUctr[1] & ~ALUctr[0]) | (~ALUctr[3] & ~ALUctr[0]);
31. assign OPctr[0] = (ALUctr[2] & ALUctr[1] & ~ALUctr[0]) | (ALUctr[2] & ~ALUctr[1] & ~ALUctr[0]) |
32. (~ALUctr[2] & ALUctr[1]);
33. assign OPctr[1] = (ALUctr[2] & ALUctr[1] & ALUctr[0]) | (~ALUctr[3] & ~ALUctr[2] & ALUctr[1]) |
34. (ALUctr[4] & ALUctr[3] & ~ALUctr[1]) | (ALUctr[4] & ALUctr[3] & ALUctr[1]);
35. assign OPctr[2] = ALUctr[4];
36. assign OPctr[3] = ALUctr[5];
37. endmodule

### 3.4.2 控制器设计

根据数据通路设计，对不同指令的各控制信号的取值进行统计、编码。

详细设计表如下，可放大查看：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **funct7** | **funct3** | **控制信号**   op | **指令** | Branch[1:0] | Jump | ALUASrc | ALUBSrc[1:0] | ALUctr  [5:0] | MemtoReg  [2:0] | Reg  Wr | MemWr | ExtOp  [2:0] | MUX\_  pcOrbusA | MUX\_Store  <1:0> |
|  | 000 | 0010011 | addi | 00 | 0 | 0 | 10 | 000000 （add） | 000 | 1 | 0 | 000 immI | X | X |
|  | 010 | 0010011 | slti | 00 | 0 | 0 | 10 | 000010 (slt) | 000 | 1 | 0 | 000 immI | X | X |
|  | 011 | 0010011 | sltiu | 00 | 0 | 0 | 10 | 000011 (sltu) | 000 | 1 | 0 | 000 immI | X | X |
|  | 100 | 0010011 | xori | 00 | 0 | 0 | 10 | 010000 (xor) | 000 | 1 | 0 | 000 immI | X | X |
|  | 001 | 0010011 | slli | 00 | 0 | 0 | 10 | 011000 (sll) | 000 | 1 | 0 | 000 immI | X | X |
| 0000000 | 101 | 0010011 | srli | 00 | 0 | 0 | 10 | 011100 (srl) | 000 | 1 | 0 | 000 immI | X | X |
| 0100000 | 101 | 0010011 | srai | 00 | 0 | 0 | 10 | 100000 (sra) | 000 | 1 | 0 | 000 immI | X | X |
|  | 110 | 0010011 | ori | 00 | 0 | 0 | 10 | 000110 (or) | 000 | 1 | 0 | 000 immI | X | X |
|  | 111 | 0010011 | andi | 00 | 0 | 0 | 10 | 010100 (and) | 000 | 1 | 0 | 000 immI | X | X |
|  | 000 | 0000011 | lb | 00 | 0 | 0 | 10 | 000000 （add） | 010 | 1 | 0 | 000 immI | X | X |
|  | 001 | 0000011 | lh | 00 | 0 | 0 | 10 | 000000 （add） | 011 | 1 | 0 | 000 immI | X | X |
|  | 010 | 0000011 | lw | 00 | 0 | 0 | 10 | 000000 （add） | 001 | 1 | 0 | 000 immI | X | X |
|  | 100 | 0000011 | lbu | 00 | 0 | 0 | 10 | 000000 （add） | 100 | 1 | 0 | 000 immI | X | X |
|  | 101 | 0000011 | lhu | 00 | 0 | 0 | 10 | 000000 （add） | 101 | 1 | 0 | 000 immI | X | X |
| 0000000 | 000 | 0110011 | add | 00 | 0 | 0 | 00 | 000000 （add） | 000 | 1 | 0 | X | X | X |
| 0100000 | 000 | 0110011 | sub | 00 | 0 | 0 | 00 | 001000 (sub) | 000 | 1 | 0 | X | X | X |
|  | 001 | 0110011 | sll | 00 | 0 | 0 | 00 | 011000 (sll) | 000 | 1 | 0 | X | X | X |
|  | 010 | 0110011 | slt | 00 | 0 | 0 | 00 | 000010 (slt) | 000 | 1 | 0 | X | X | X |
|  | 011 | 0110011 | sltu | 00 | 0 | 0 | 00 | 000011 (sltu) | 000 | 1 | 0 | X | X | X |
|  | 100 | 0110011 | xor | 00 | 0 | 0 | 00 | 010000 (xor) | 000 | 1 | 0 | X | X | X |
| 0000000 | 101 | 0110011 | srl | 00 | 0 | 0 | 00 | 011100 (srl) | 000 | 1 | 0 | X | X | X |
| 0100000 | 101 | 0110011 | sra | 00 | 0 | 0 | 00 | 100000 (sra) | 000 | 1 | 0 | X | X | X |
|  | 110 | 0110011 | or | 00 | 0 | 0 | 00 | 000110 (or) | 000 | 1 | 0 | X | X | X |
|  | 111 | 0110011 | and | 00 | 0 | 0 | 00 | 010100 (and) | 000 | 1 | 0 | X | X | X |
|  | 无关 | 0110111 | lui | 00 | 0 | X | 10 | 001111 (srcB) | 000 | 1 | 0 | 001 immU | X | X |
|  | 无关 | 0010111 | auipc | 00 | 0 | 1 | 10 | 000000 (add) | 000 | 1 | 0 | 001 immU | X | X |
|  | 000 | 0100011 | sb | 00 | 0 | 0 | 10 | 000000 (add) | X | 0 | 1 | 010 immS | X | 01 |
|  | 001 | 0100011 | sh | 00 | 0 | 0 | 10 | 000000 (add) | X | 0 | 1 | 010 immS | X | 11 |
|  | 010 | 0100011 | sw | 00 | 0 | 0 | 10 | 000000 （add） | X | 0 | 1 | 010 immS | X | 00 |
|  | 000 | 1100011 | beq | 01 | 0 | 0 | 00 | 001000 (sub) | X | 0 | 0 | 011 immB | 0 | X |
|  | 001 | 1100011 | bne | 10 | 0 | 0 | 00 | 001000 (sub) | X | 0 | 0 | 011 immB | 0 | X |
|  | 100 | 1100011 | blt | 11 | 0 | 0 | 00 | 000010 (slt) | X | 0 | 0 | 011 immB | 0 | X |
|  | 101 | 1100011 | bge | 11 | 0 | 0 | 00 | 110000 (bge) | X | 0 | 0 | 011 immB | 0 | X |
|  | 110 | 1100011 | bltu | 11 | 0 | 0 | 00 | 000011 (sltu) | X | 0 | 0 | 011 immB | 0 | X |
|  | 111 | 1100011 | bgeu | 11 | 0 | 0 | 00 | 111000 (bgeu) | X | 0 | 0 | 011 immB | 0 | X |
|  | 无关 | 1101111 | jal | 00 | 1 | 1 | 01 | 000000 （add） | 000 | 1 | 0 | 100 immJ | 0 | X |
|  | 000 | 1100111 | jalr | 00 | 1 | X | 01 | 000000 （add) | 000 | 1 | 0 | 000 immI | 1 | X |

**逻辑表达式：**

Branch[1] = (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[0]);

Branch[0] = (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[0]);

Jump = (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0]) | (op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0])

ALUASrc = (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0]); | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])

ALUBSrc[1] = (~op[6] & ~op[5] & op[4]&~op[3] & ~op[2] & op[1] & op[0]) //I-type-ALU

| (~op[6]&op[5]&op[4]&~op[3]&op[2]&op[1]&op[0]) // lui

| (~op[6]&~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0]) // Load

| (~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0]); // Store

ALUBSrc[0] = (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0]) // J-type

| (op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0]);

MUX\_pcOrbusA = op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0]

MUX\_Store[1] = ~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0] & f3[0]

MUX\_Store[0] = ~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0] & ~f3[1]

ALUctr[5] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & f7[5]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & f7[5]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[0])

ALUctr[4] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & f3[0]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & ~f3[0]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & ~f3[0])

ALUctr[3] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & ~f3[0] & f7[5]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[1] & f3[0])

ALUctr[2] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1]) | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])

ALUctr[1] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1]) | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1])

| (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])

ALUctr[0] = (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1] & f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1] & f3[0]) | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0]) | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])

MemtoReg[2] = ~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2]

MemtoReg[1] = ~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1]

MemtoReg[0] = (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[1]) | (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[0]);

RegWr = (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0])  // R - type

            | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0])// I - type - ALU

             | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])                // lui

             | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])

             | (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])         // Load

          | (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0]);      // J - type

MemWr = ~op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] ;  // Store

ExtOP[2] = op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];       // J - type

ExtOP[1] = (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])    // B - type

                | (~op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0]);  // Store

ExtOP[0] = (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])     // lui

                | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])   // B - type

                | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]);

**原理阐述：**

对于每个控制信号，根据其取值范围的不同定义了控制信号的位数，并且通过查阅资料，得到了每种指令的控制信号的取值（如上表所示）。我们设定控制信号取值为0时无效，并且控制信号取值为x时，说明该信号的取值对于指令的执行没有影响，因此控制信号取值为x的情况我们不作考虑，因此我们单独考虑控制信号每一位取值与操作码op取值之间的联系。

对于Branch信号，我们设定的是2位，依次分析Branch[1]和Branch[0]的取值为1时的逻辑表达式，通过观察所有指令的Branch各位的取值，对于Branch[1]，可以看到只有当操作码取值为1100011并且f3[2]=1或者操作码取值为1100011并且f3[0]=1时，Branch[1]=1；对于Branch[0]，只有当操作码取值为1100011并且f3[2]=1或者操作码取值为1100011并且f3[0]=0时，Branch[0]=0.

对于Jump信号，只有1位，通过观察所有指令的Jump取值，可以看到只有当操作码为1101111或1100111时，Jump=1.

对于ALUASrc信号，只有1位，通过观察所有指令的ALUASrc取值，可以看到只有当操作码取值为1101111或者00101111时，ALUASrc=1.

对于ALUBSrc信号，设定的是2位，依次分析ALUBSrc[1]和ALUBSrc[0]的取值为1时的逻辑表达式，通过观察所有指令的ALUBSrc各位的取值，对于ALUBSrc[1]，可以看到只有当操作码为0010011或者0110111或者0000011或者0100011时，ALUBSrc[1]=1；对于ALUBSrc[0]，只有当操作码为1101111或者1100111时，ALUBSrc[0]=1.

对于MUX\_pcOrbusA信号，只有1位，通过观察所有指令的MUX\_pcOrbusA取值，可以看到只有当操作码为1100111时，MUX\_pcOrbusA=1.

对于MUX\_Store信号，设定的是2位，依次分析MUX\_Store[1]和MUX\_Store[0]的取值为1时的逻辑表达式，通过观察所有指令的MUX\_Store各位的取值，对于MUX\_Store[1]，可以看到只有当操作码为0100011并且f3[0]=1时，MUX\_Store[1]=1；对于MUX\_Store[0]，只有当操作码为0100011并且f3[1]=0时，MUX\_Store[0]=1.

对于ALUctr信号，设定的是6位，依次分析ALUctr[5]、ALUctr[4]、ALUctr[3]、ALUctr[2]、ALUctr[1]和ALUctr[0]的取值为1时的逻辑表达式，通过观察所有指令的ALUctr各位的取值，对于ALUctr[5]，可以看到只有当操作码为0010011并且f3[2]=1f3[1]=0,f3[0]=1,f7[5]=1或者操作码为0110011并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=1或者操作码为1100011并且f3[2]=1,f3[0]=1时，ALUctr[5]=1；对于ALUctr[4]，只有当操作码为0010011并且f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0010011并且f3[2]=1,f3[1]=1,f3[0]=1或者操作码为0010011并且f3[2]=0,f3[1]=0,f3[0]=1或者操作码为0010011并且f3[2]=1,f3[1]=0,f3[0]=0或者操作码为1100011并且f3[2]=1,f3[0]=1或者操作码为0110011并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0110011并且f3[2]=1,f3[1]=1,f3[0]=1或者操作码为0110011并且f3[2]=0,f3[1]=0,f3[0]=1或者操作码为0110011并且f3[2]=1,f3[1]=0,f3[0]=0时，ALUctr[4]=1；对于ALUctr[3]，只有当操作码为00100111并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0010011并且f3[2]=0,f3[1]=0,f3[0]=1或者操作码为0110011并且f3[2]=0,f3[1]=0,f3[0]=0,f7[5]=1或者操作码为0110011并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0110011并且f3[2]=0,f3[1]=0,f3[0]=1或者操作码为0110111或者1100011并且f3[2]=0或者操作码为1100011并且f3[1]=1,f3[0]=1时，ALUctr[3]=1；对于ALUctr[2]，只有当操作码为0010011并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0010011并且f3[2]=1,f3[1]=1或者操作码为0110011并且f3[2]=1,f3[1]=0,f3[0]=1,f7[5]=0或者操作码为0110011并且f3[2]=1,f3[1]=1或者操作码为0110111时，ALUctr[2]=1；对于ALUctr[1]，只有当操作码为0010011并且f3[2]=0,f3[1]=1或者操作码为0010011并且f3[2]=1,f3[1]=1,f3[0]=0或者操作码为0110011并且f3[2]=1,f3[1]=1,f3[0]=0或者操作码为0110011并且f3[2]=0,f3[1]=1或者操作码为1100011并且f3[2]=1,f3[0]=0或者操作码为0110111时，ALUctr[1]=1；对于ALUctr[0]，只有当操作码为0010011并且f3[2]=0,f3[1]=1,f3[0]=1或者操作码为0110011并且f3[2]=0,f3[1]=1,f3[0]=1或者操作码为1100011并且f3[2]=1,f3[1]=1,f3[0]=0或者操作码为0110111时，ALUctr[0]=1.

对于MemtoReg信号，设定为3位，依次分析MemtoReg[2]、MemtoReg[1]、MemtoReg[0]的取值为1时的逻辑表达式，通过观察所有指令的MemtoReg各位的取值，对于MemtoReg[2]，可以看到只有当操作码为0000011并且f3[2]=1时，MemtoReg[2]=1；对于MemtoReg[1]，只有当操作码为0000011并且f3[2]=0,f3[1]=0时，MemtoReg[1]=1；对于MemtoReg[0]，只有当操作码为0000011并且f3[1]=1或者操作码为0000011并且f3[0]=1时，MemtoReg[0]=1.

对于RegWr信号，只有1位，通过观察所有指令的RegWr取值，可以看到只有当操作码为0110011或者0010011或者0110111或者0010111或者0000011或者1101111时，RegWr=1.

对于MemWr信号，只有1位，通过观察所有指令的MemWr取值，可以看到只有当操作码为0100011时，MemWr=1.

对于ExtOP信号，设定为3位，依次分析ExtOP[2]、ExtOP[1]、ExtOP[0]的取值为1时的逻辑表达式，通过观察所有指令的ExtOP各位的取值，对于ExtOP[2]，可以看到只有当操作码为1101111时，ExtOP[2]=1；对于ExtOP[1]，只有当操作码为1100011或者0100011时，ExtOP[1]=1；对于ExtOP[0]，只有当操作码为0110111或者1100011或者0010111时，ExtOP[0]=1.

**关键代码设计：**

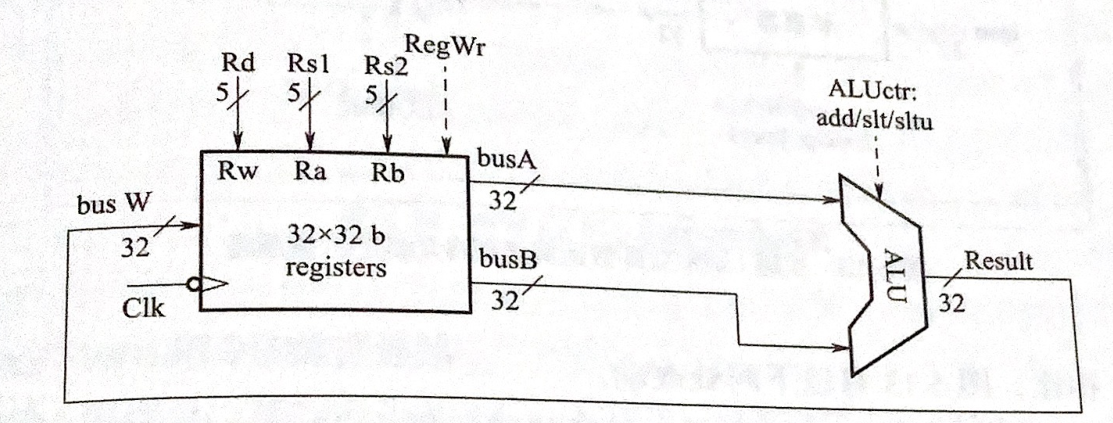
1. `timescale 1ns / 1ps
2. //////////////////////////////////////////////////////////////////////////////////
3. // Company:
4. // Engineer:
5. //
6. // Create Date: 2022/11/16 23:43:46
7. // Design Name:
8. // Module Name: cpu\_controller
9. // Project Name:
10. // Target Devices:
11. // Tool Versions:
12. // Description:
13. //
14. // Dependencies:
15. //
16. // Revision:
17. // Revision 0.01 - File Created
18. // Additional Comments:
19. //
20. //////////////////////////////////////////////////////////////////////////////////

23. module cpu\_controller(
24. input             rst\_n,
25. input [31:0]      inst\_i,
27. // 输出到regfile
28. output reg [4:0]  reg1\_addr\_o,
29. output reg [4:0]  reg2\_addr\_o,
30. output reg [4:0]  wd\_o,
32. // 单值控制信号
33. output reg [1:0]  branch\_o,       // pc
34. output reg  jump\_o,         // pc
35. output reg [2:0] memto\_reg\_o,   // ALU to Reg
36. output reg  reg\_wr\_o,       // Reg
37. output reg  mem\_wr\_o,       // Mem
38. output reg  alu\_asrc\_o,     // Reg to ALU
39. output reg muxpc\_busa\_o,
41. // 多值控制信号
42. output reg [1:0]  alu\_bsrc\_o,   // imm and Reg to ALU
43. output reg [2:0]  ext\_op\_o,     // imm
44. output reg [5:0]  alu\_ctr\_o,     // ALU
45. output reg [1:0]  mux\_store\_o
46. );
47. wire [6:0] op  = inst\_i[6:0];            //运算类型
48. wire [2:0] f3 = inst\_i[14:12];          //具体运算方式
49. wire [6:0] f7 = inst\_i[31:25];
51. always @ (\*) begin
52. **if**(~rst\_n) begin    // 清零重置
53. wd\_o <= 0;
54. reg1\_addr\_o <= 0;
55. reg2\_addr\_o <= 0;
56. end
57. **else** begin      // 译码
58. wd\_o <= inst\_i[11:7];                     // 写寄存器地址
59. reg1\_addr\_o <= inst\_i[19:15];             // 读寄存器 A 地址
60. reg2\_addr\_o <= inst\_i[24:20];             // 读寄存器 B 地址
62. // 单值控制信号
63. branch\_o[1]<=(op[6]& op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2])
64. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[0]);
65. branch\_o[0]<=(op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2])
66. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[0]);    //B-type
68. jump\_o<=(op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0])
69. | (op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0]);            // J-type
71. memto\_reg\_o[2]<= ~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2];
72. memto\_reg\_o[1]<= ~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1];
73. memto\_reg\_o[0]<= (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[1])
74. | (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[0]);    // Load
76. reg\_wr\_o<= (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0])           // R - type
77. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0])      // I - type - ALU
78. | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])        // lui
79. | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])
80. | (~op[6] & ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])     // Load
81. | (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0]);        // J - type
83. mem\_wr\_o<= ~op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];  // Store
84. alu\_asrc\_o<= (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0])
85. | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])
86. | (op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0]);  // J-type
88. muxpc\_busa\_o<= op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0];
89. mux\_store\_o[1] = ~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0] & f3[0];
90. mux\_store\_o[0] = ~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0] & ~f3[1];
92. // 多值控制信号
93. alu\_bsrc\_o[1]<= (~op[6] & ~op[5] & op[4]&~op[3] & ~op[2] & op[1] & op[0])   // I-type-ALU
94. | (~op[6]&op[5]&op[4]&~op[3]&op[2]&op[1]&op[0])              // lui
95. | (~op[6]&~op[5]&op[4]&~op[3]&op[2]&op[1]&op[0])             // auipc
96. | (~op[6]&~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0])           // Load
97. | (~op[6]&op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0]);           // Store
98. alu\_bsrc\_o[0]<= (op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0])
99. | (op[6] & op[5] & ~op[4] & ~op[3] & op[2] & op[1] & op[0]); // J-type
101. ext\_op\_o[2]<= op[6] & op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];   // J - type
102. ext\_op\_o[1]<= (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])        // B - type
103. | (~op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0]);   // Store
104. ext\_op\_o[0]<= (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])         // lui
105. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])     // B - type
106. | (~op[6] & ~op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]);
108. alu\_ctr\_o[5]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & f7[5])
109. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & f7[5])
110. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[0]);
111. alu\_ctr\_o[4]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
112. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & f3[0])
113. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0])
114. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & ~f3[0])
115. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[0])
116. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
117. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & f3[0])
118. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0])
119. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & ~f3[0]);
120. alu\_ctr\_o[3]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
121. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0])
122. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & ~f3[0] & f7[5])
123. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
124. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & ~f3[1] & f3[0])
125. | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0])
126. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2])
127. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[1] & f3[0]);
128. alu\_ctr\_o[2]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
129. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1])
130. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[1] & f3[0] & ~f7[5])
131. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1])
132. | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]);
133. alu\_ctr\_o[1]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1])
134. | (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0])
135. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0])
136. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1])
137. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & ~f3[0])
138. | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]);
139. alu\_ctr\_o[0]<= (~op[6] & ~op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1] & f3[0])
140. | (~op[6] & op[5] & op[4] & ~op[3] & ~op[2] & op[1] & op[0] & ~f3[2] & f3[1] & f3[0])
141. | (op[6] & op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0] & f3[2] & f3[1] & ~f3[0])
142. | (~op[6] & op[5] & op[4] & ~op[3] & op[2] & op[1] & op[0]);
143. end
144. end
145. endmodule

# 实验验证与分析

## 4.1 R- 型指令

### 4.1.0 R- 型指令的数据通路



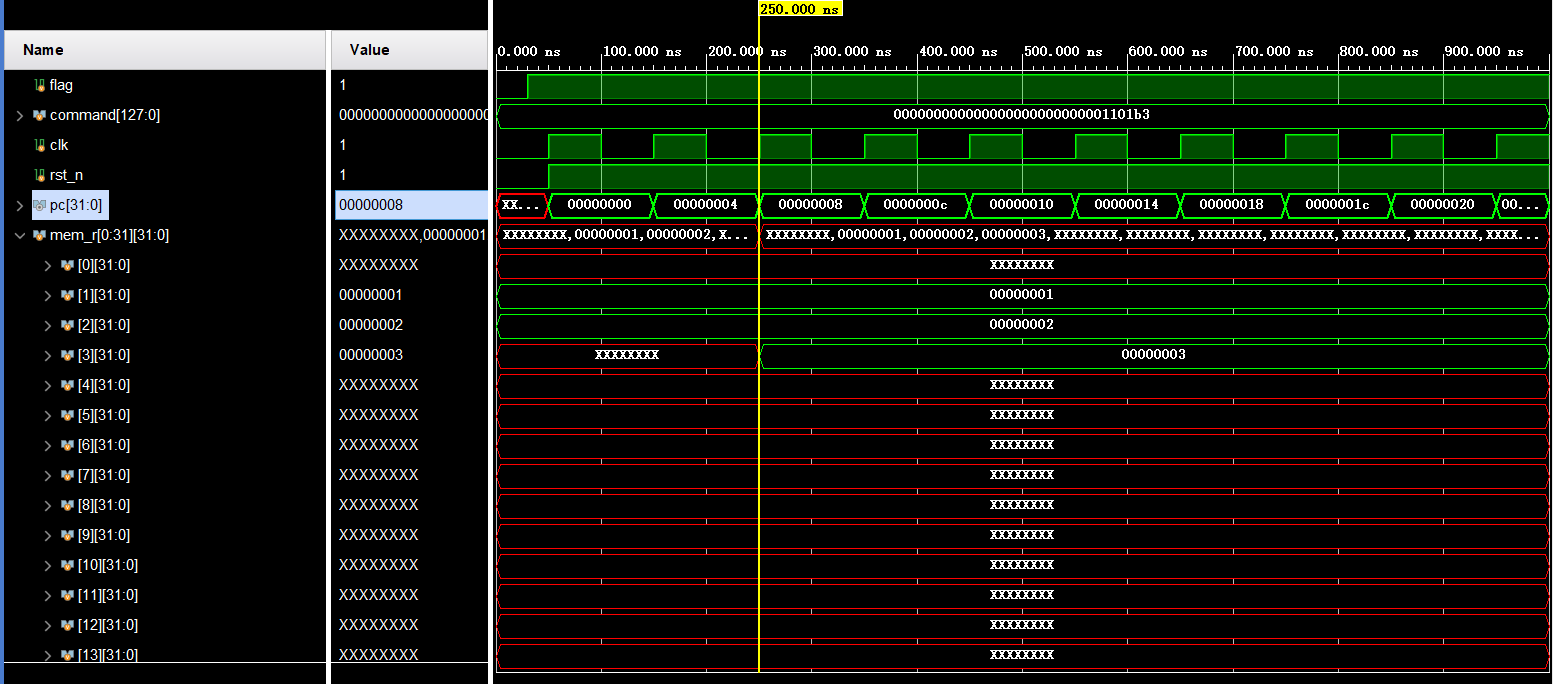
### 4.1.1 R- add指令

**仿真代码与注释：**

command={{32'b0000\_0000\_0001\_0001\_0000\_0001\_1011\_0011}}

取寄存器中地址为 2 的数据 1 和地址为 1 的数据 2 相加后为 3 存储至寄存器地址为 3 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，与mem\_r[2]处的数据0000 0002相加，结果为0000 0003，存入mem\_r[3]中。

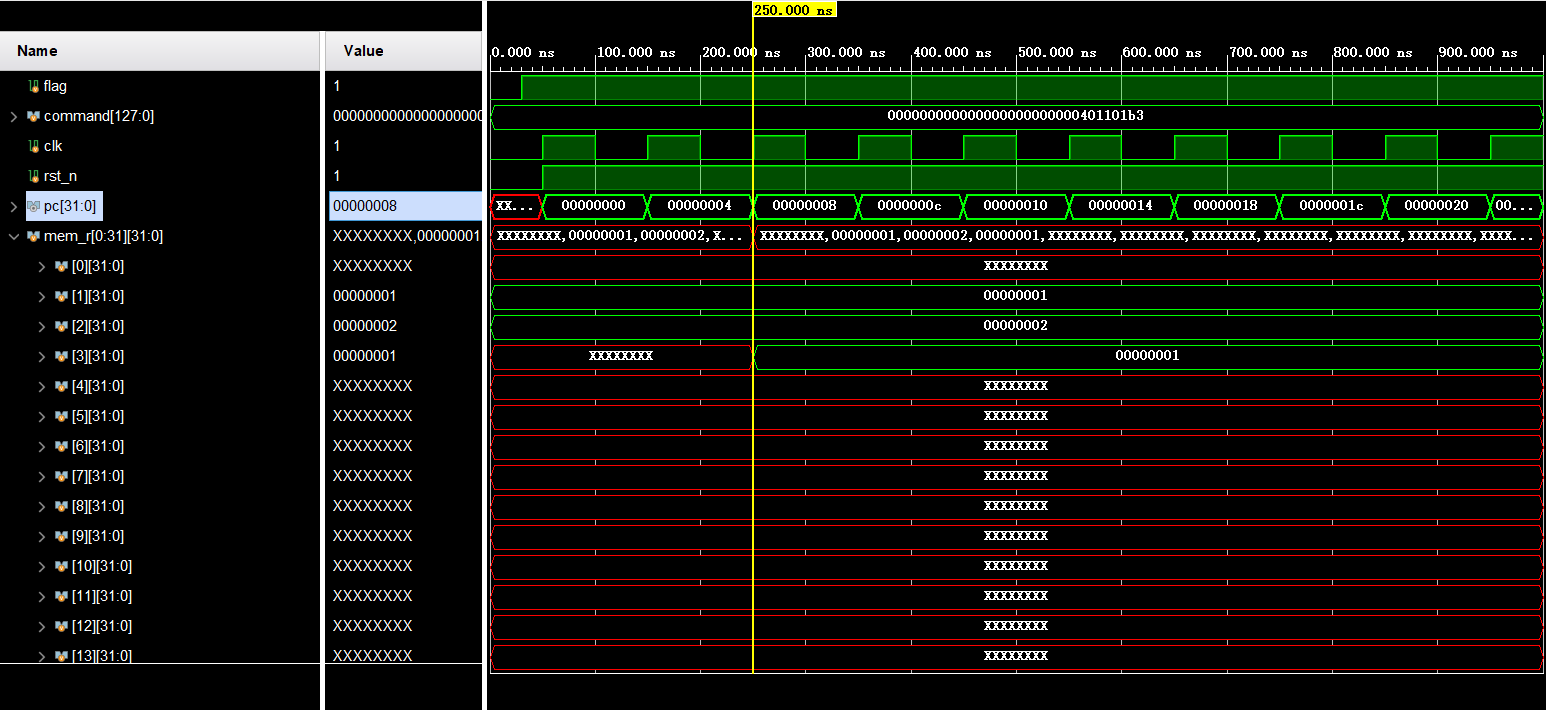
### 4.1.2 R- sub指令

**仿真代码与注释：**

command={{32'b0100\_0000\_0001\_0001\_0000\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 2 和地址为 1 的数据 1 相减后为 1 存储至寄存器地址为 3 处

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001作为减数，与mem\_r[2]处的数据0000 0002作为被减数，结果为0000 0001，存入mem\_r[3]中。

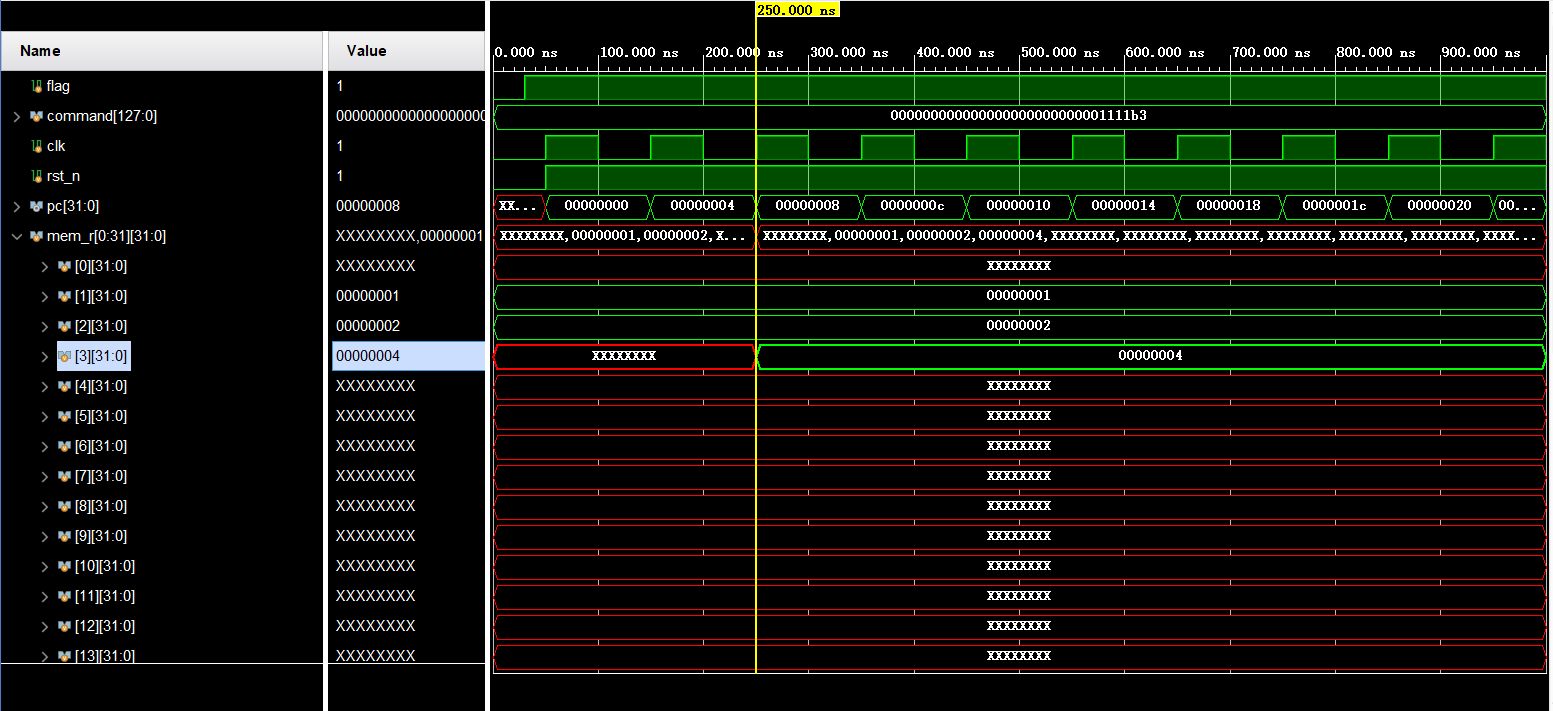
### 4.1.3 R- sll指令

**仿真代码与注释：**

command={{32'b0000\_0000\_0001\_0001\_0001\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 1 左移寄存器地址为 1 的数据 2 后为 4 存储至寄存器地址为 3 处，低位补0。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，左移mem\_r[2]处的数据0000 0002位，二进制1左移两位后结果为4，结果为0000 0004，存入mem\_r[3]中。

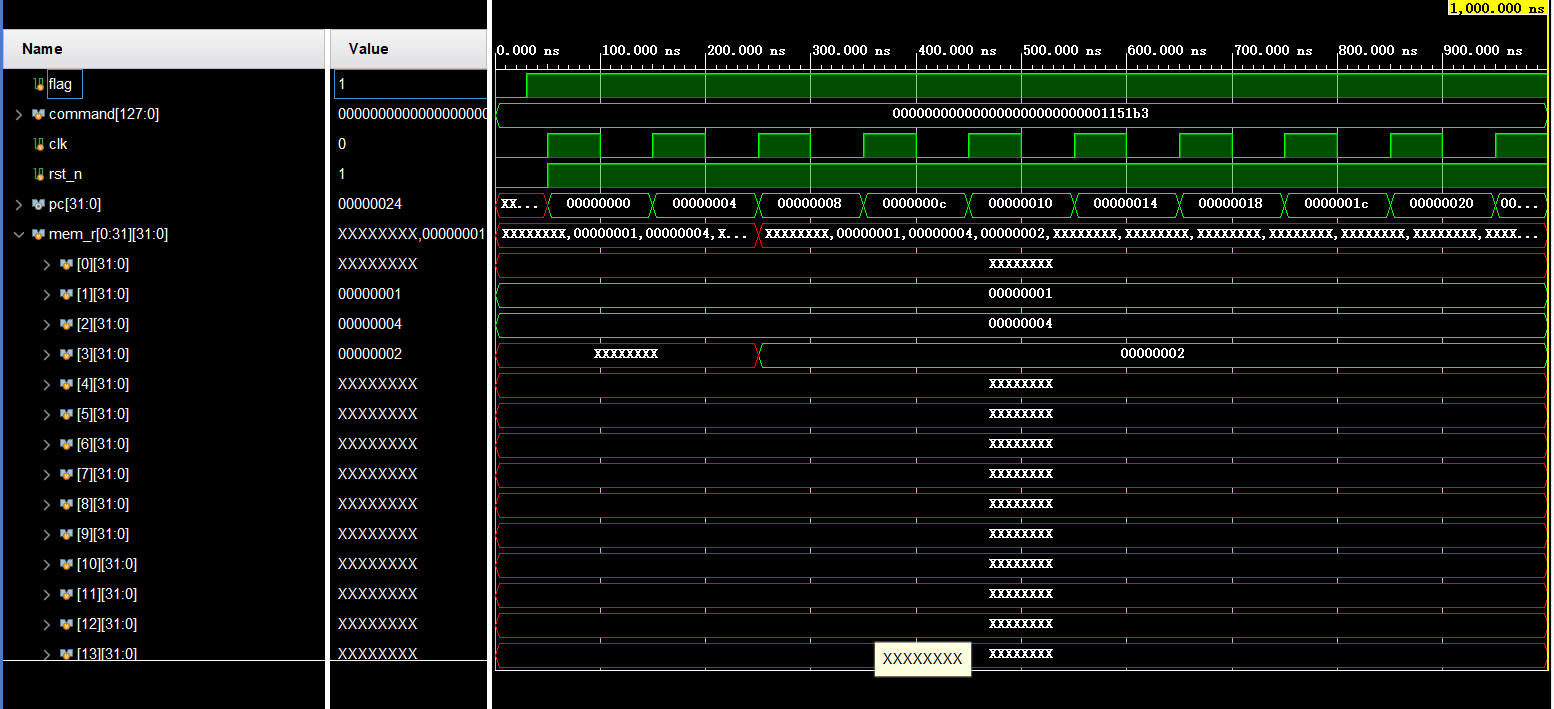
### 4.1.4 R- srl指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0001\_0001\_0101\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 4 右移寄存器地址为 1 的数据 1 后为 2 存储至寄存器地址为 3 处，低位补0。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[2]处的数据：0000 0004，左移mem\_r[1]处的数据0000 0001位，二进制4右移一位后结果为2，结果为0000 0002，存入mem\_r[3]中。

### 4.1.5 R- slt指令

**仿真代码与注释：**

1. 正数-负数：

command={{32'b0000\_0000\_0001\_0001\_0010\_0001\_1011\_0011}};

寄存器地址为 1 处的数据为任意正数，地址为 2 的地方数据为任意负数，如果地址为 1 处的数据大于地址为 2 处的数据，应将 1 存入寄存器地址 3 处。

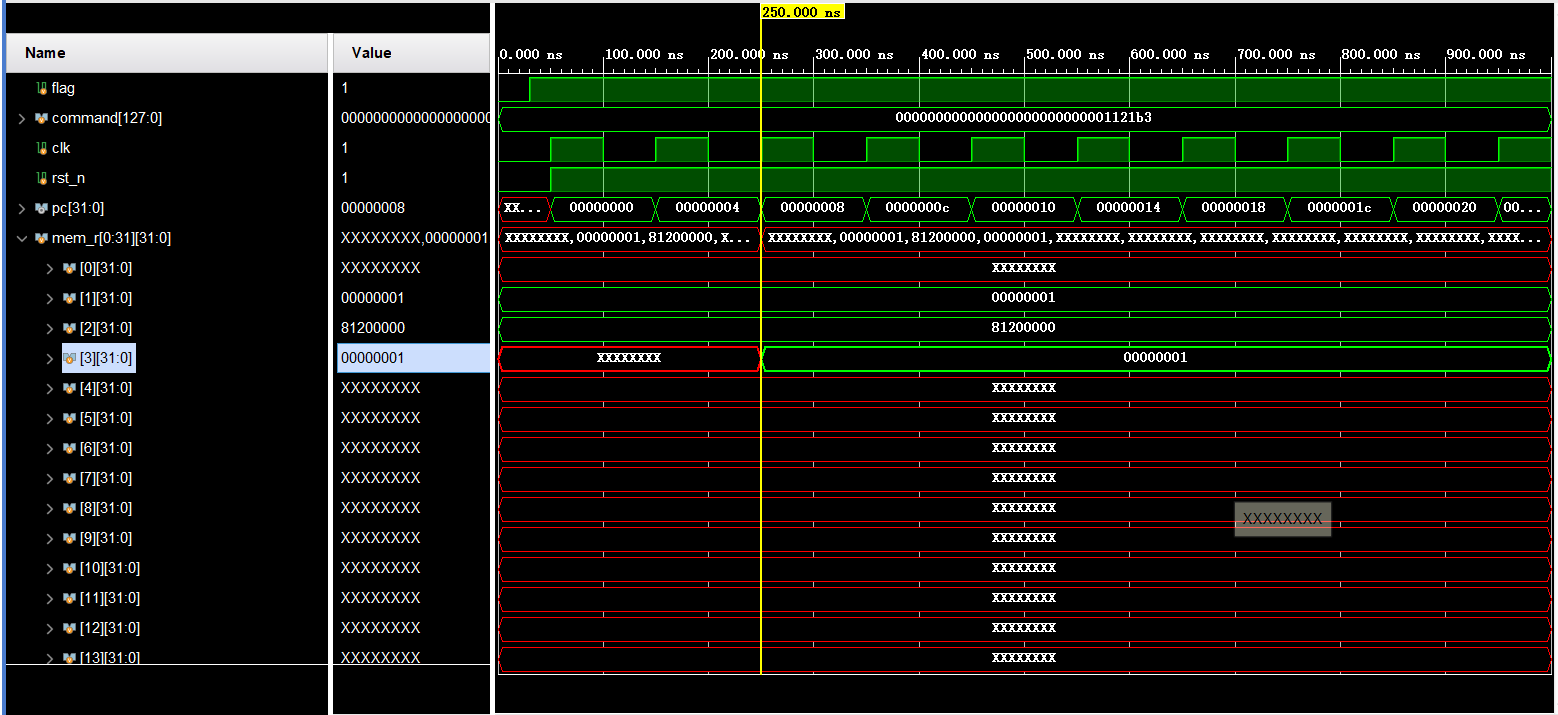
1. 负数-正数：

command={{32'b0000\_0000\_0001\_0001\_0010\_0001\_1011\_0011}};

寄存器地址为 1 处的数据为任意负数，地址为 2 的地方数据为任意正数，如果地址为 1 处的数据不大于地址为 2 处的数据，应将 0 存入寄存器地址 3 处。

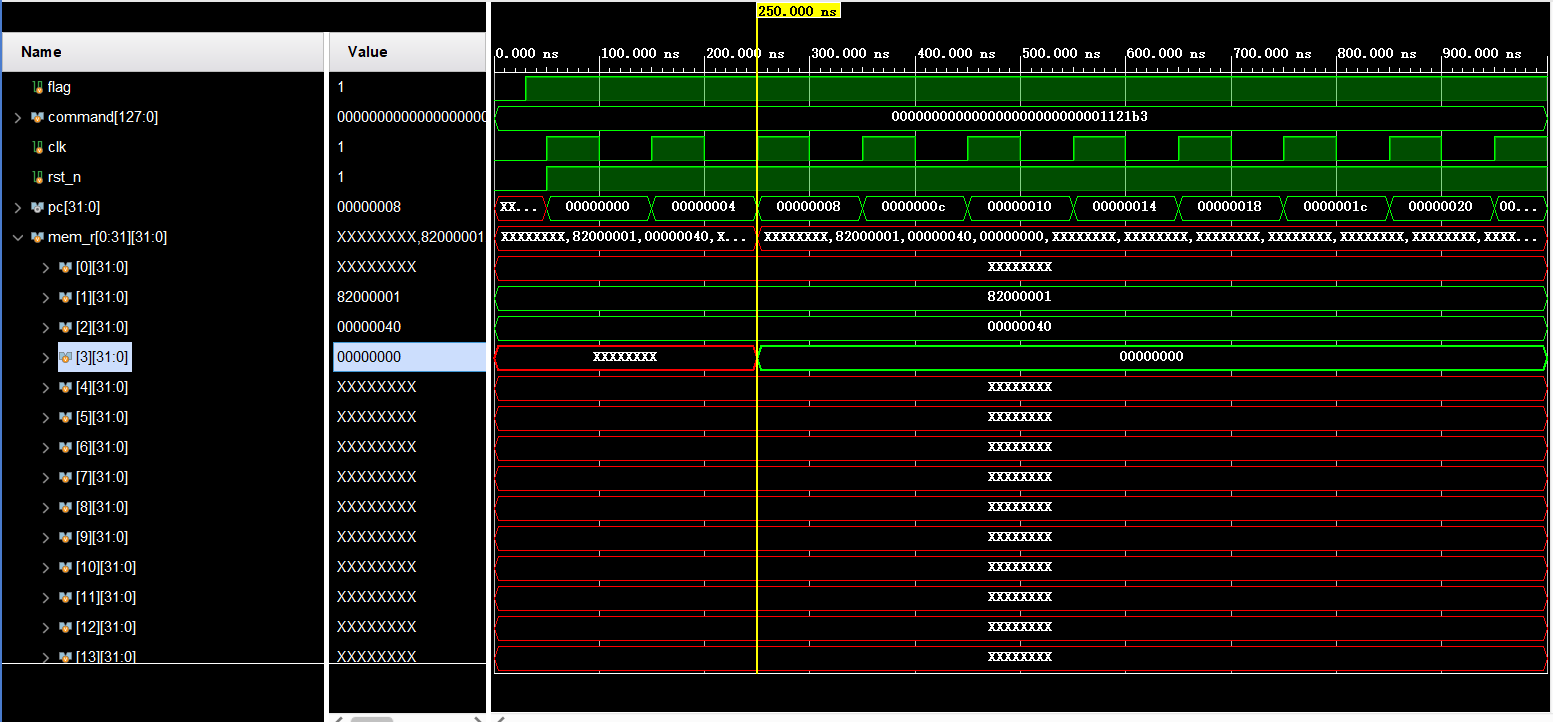
**验证结果与结果分析：**

（1）正数-负数：



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，mem\_r[2]处的数据B120 0000，比较大小，正数必然大于负数，将mem\_r[1]处数据存入mem\_r[3]处。

（2）负数-正数：



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：B200 0001，mem\_r[2]处的数据0000 0040，比较大小，负数必然小于正数，将0存入mem\_r[3]处。

注：这里正负数据取值为任意的。

### 4.1.6 R- sltu指令

**仿真代码与注释：**（涉及到的关键的）

（1）大于情况（结果为1）：

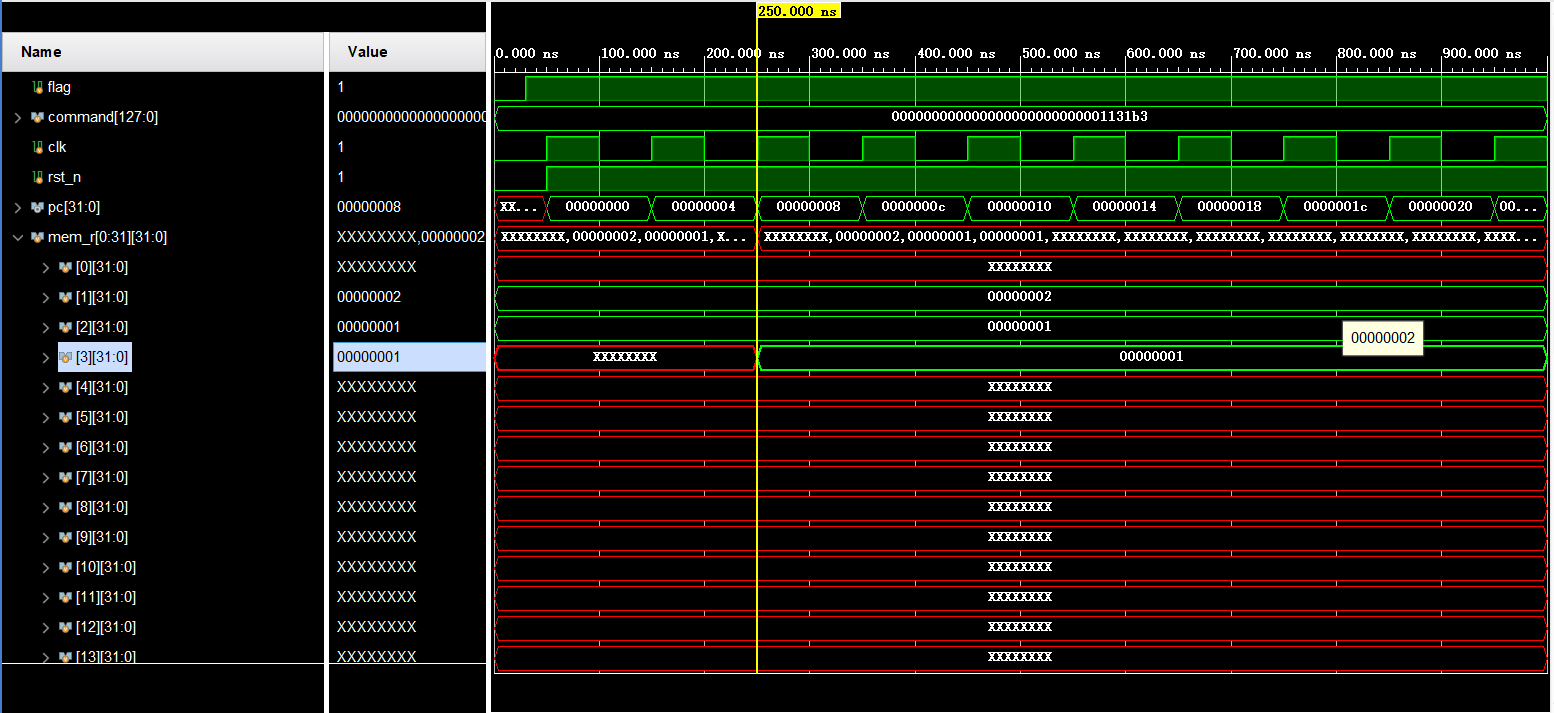
command={{32'b0000\_0000\_0001\_0001\_0011\_0001\_1011\_0011}};//寄存器地址为 1 处的数据为 2，地址为 2 的地方数据为 1，如果地址为 1 处的数据大于地址为2 处的数据，应将 1 存入寄存器地址 3 处。

（2）小于情况（结果为0）：

command={{32'b0000\_0000\_0001\_0001\_0011\_0001\_1011\_0011}};//寄存器地址为 1 处的数据为 1，地址为 2 的地方数据为 2，如果地址为 1 处的数据大于地址为2 处的数据，应将 0 存入寄存器地址 3 处。

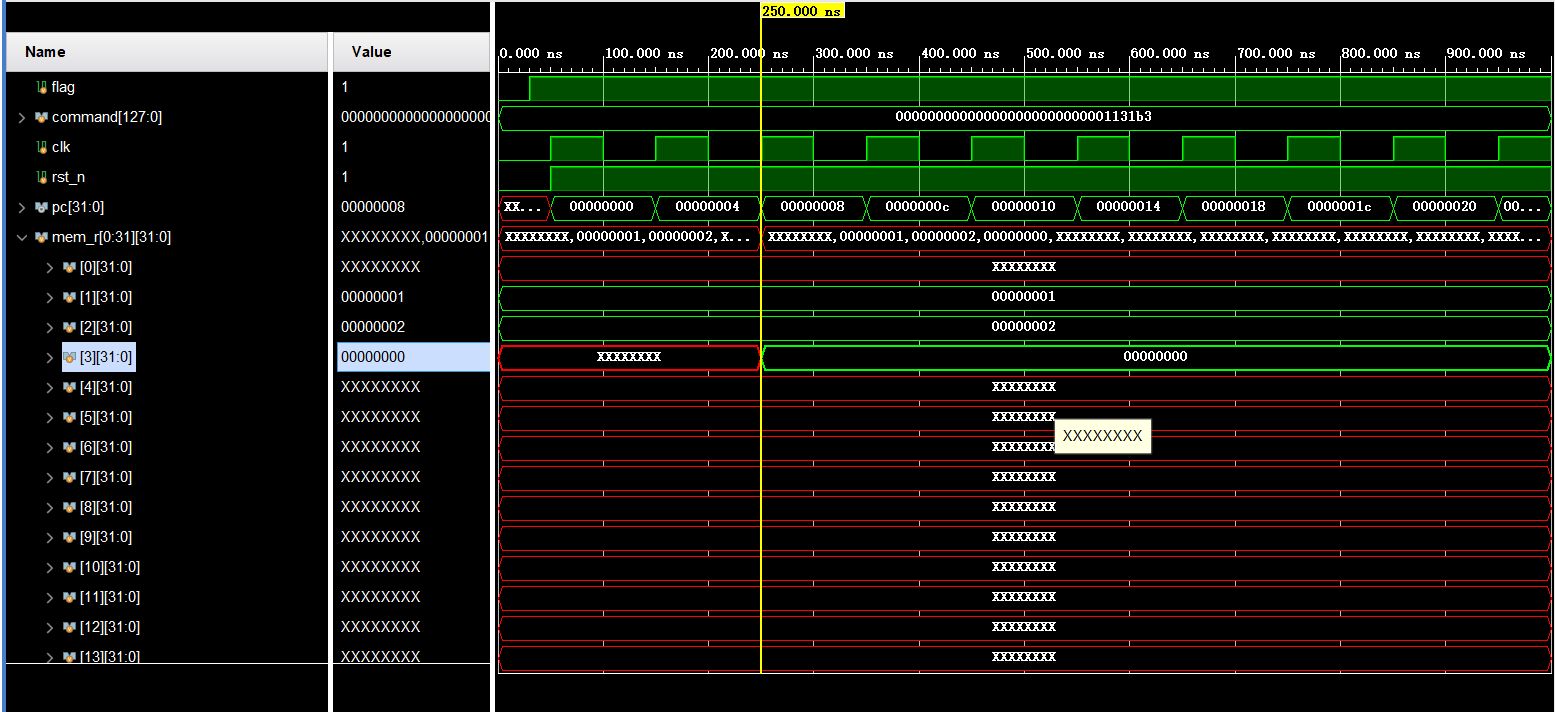
**验证结果与结果分析：**

1. 大于情况（结果为1）：



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，mem\_r[2]处的数据0000 0001，比较大小，mem\_r[1]处的数据大于mem\_r[2]处的数据，将1存入mem\_r[3]处，即mem\_r[3] = 0000 0001。

（2）小于情况（结果为0）：



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，mem\_r[2]处的数据0000 0002，比较大小，mem\_r[1]处的数据小于mem\_r[2]处的数据，将0存入mem\_r[3]处，即mem\_r[3] = 0000 0000。

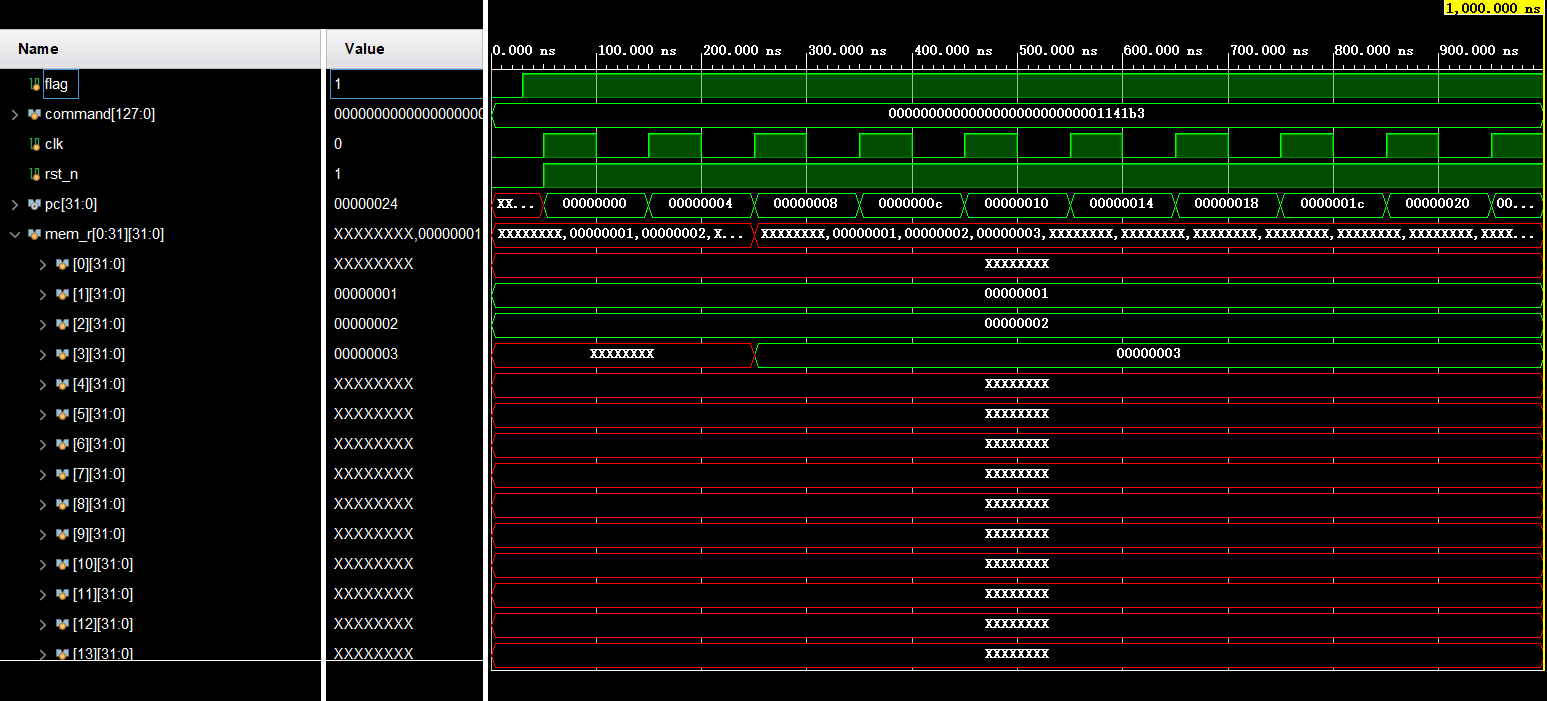
### 4.1.7 R- xor指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0001\_0001\_0100\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 2 和地址为 1 的数据 3 异或后为 1 存储至寄存器地址为 3 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，mem\_r[2]处的数据0000 0002，二进制为0000 0001和0000 0010，两个数进行异或，结果是0000 0011，即十进制数3，将结果存入mem\_r[3]处，即mem\_r[3] = 0000 0003。

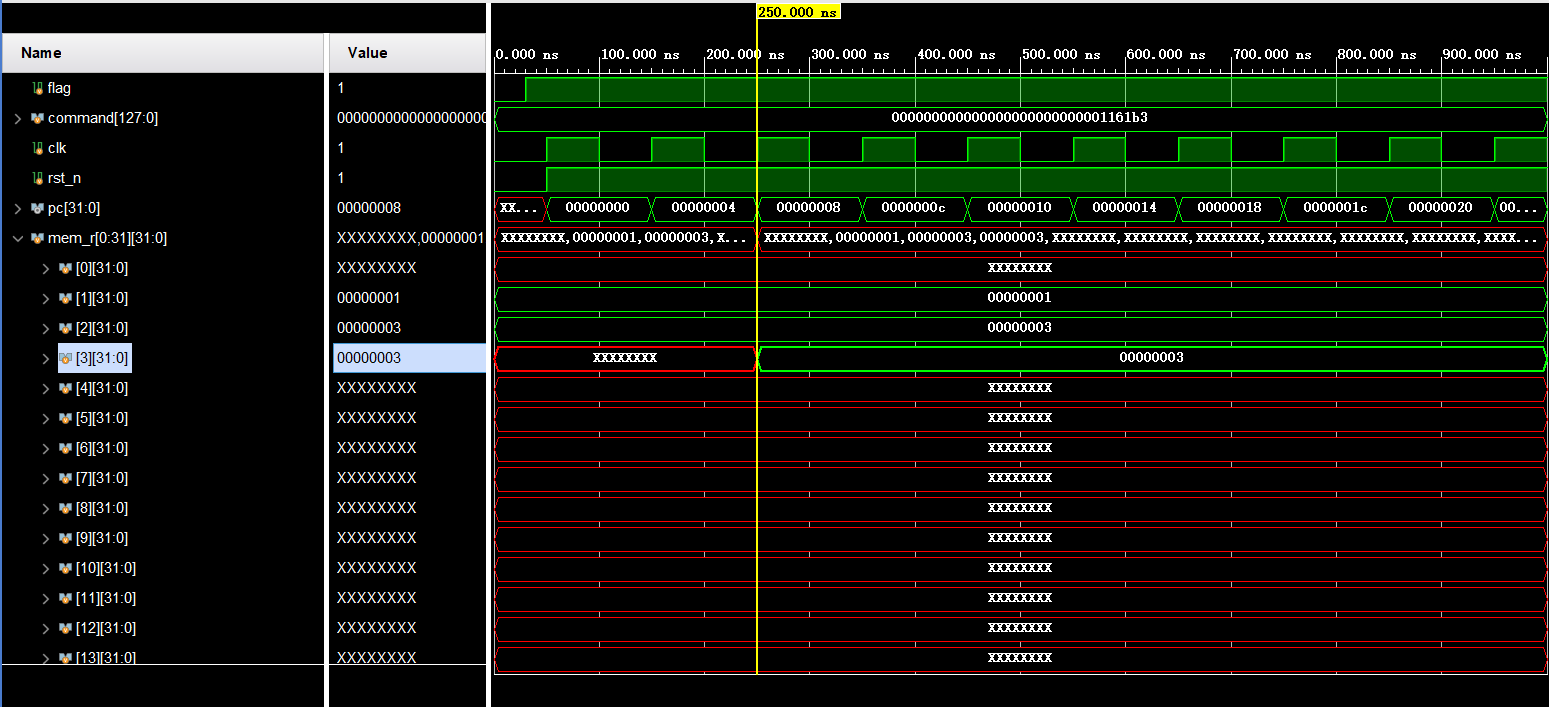
### 4.1.8 R- or指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0001\_0001\_0110\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 2 和地址为 1 的数据 3 位或运算后为 3 存储至寄存器地址为 3 处。

**验证结果：**



**结果分析：**

位或运算：只要对应的二个二进位有一个为1时，结果位就为1。

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，mem\_r[2]处的数据0000 0003，二进制为0000 0001和0000 0011，两个数进行按位或，结果是0000 0011，即十进制数3，将结果存入mem\_r[3]处，即mem\_r[3] = 0000 0003。

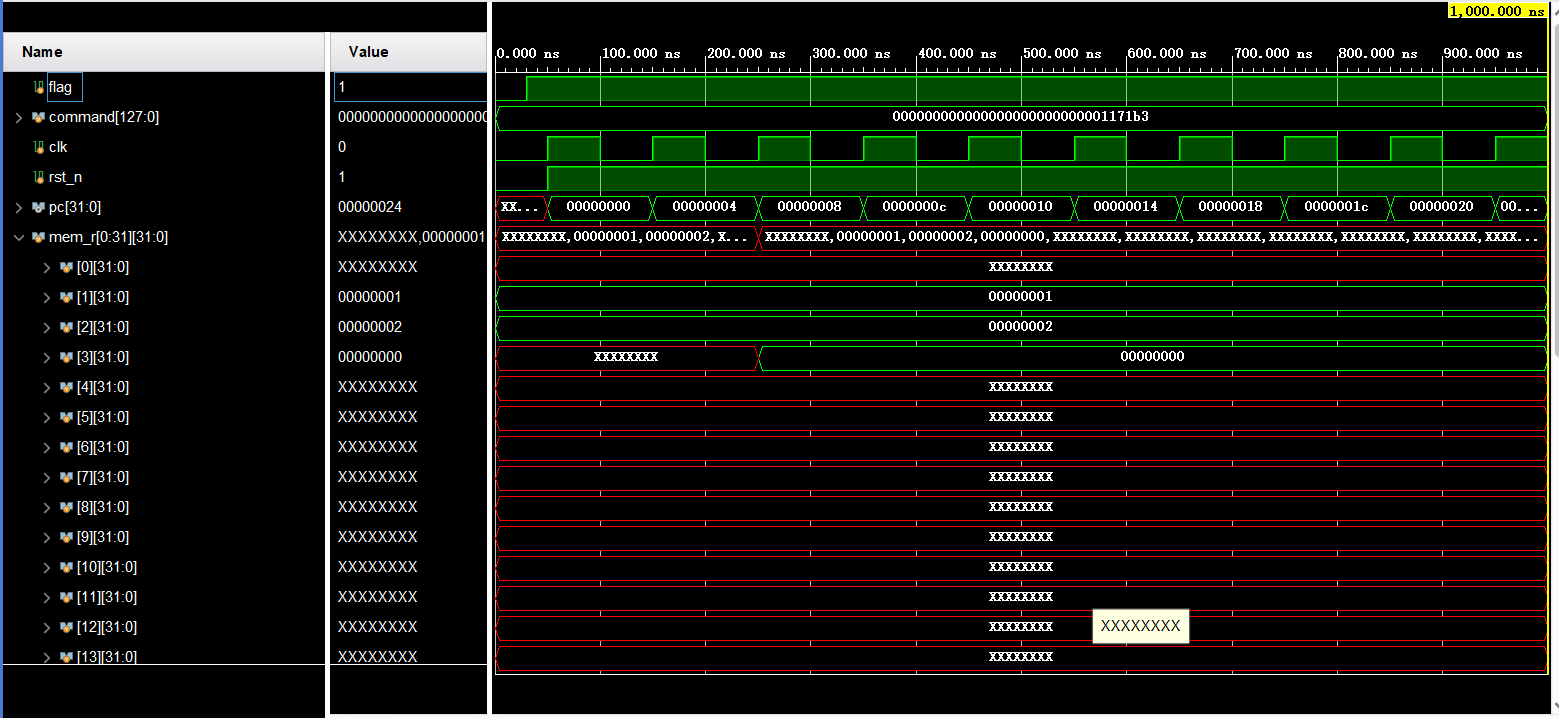
### 4.1.9 R- and指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0001\_0001\_0111\_0001\_1011\_0011}};

取寄存器中地址为 2 的数据 2 和地址为 1 的数据 3 位与运算后为 2 存储至寄存器地址为 3 处。

**验证结果：**



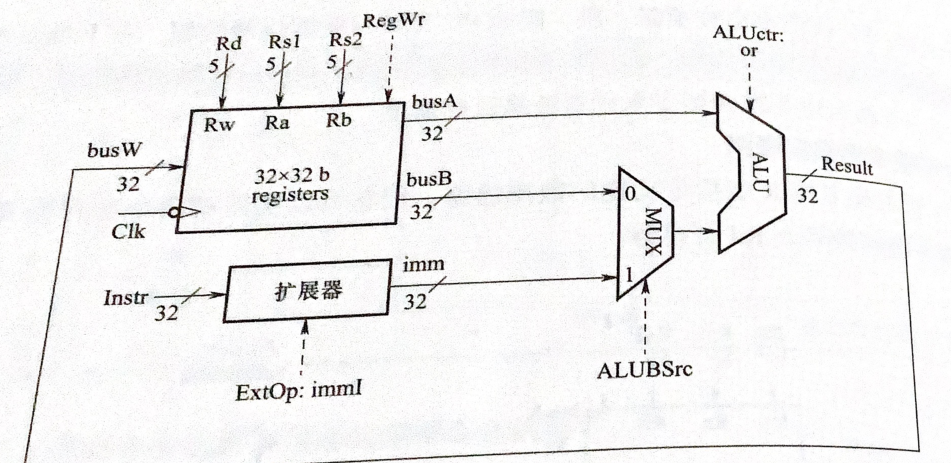
**结果分析：**

按位与运算，两个都相同且为1的时候结果为1。

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，mem\_r[2]处的数据0000 0002，二进制为0000 0001和0000 0010，两个数进行按位与，结果是0000 0000，即十进制数0，将结果存入mem\_r[3]处，即mem\_r[3] = 0000 0000。

## 4.2 I- 型指令

### 4.2.0 I- 型指令的数据通路



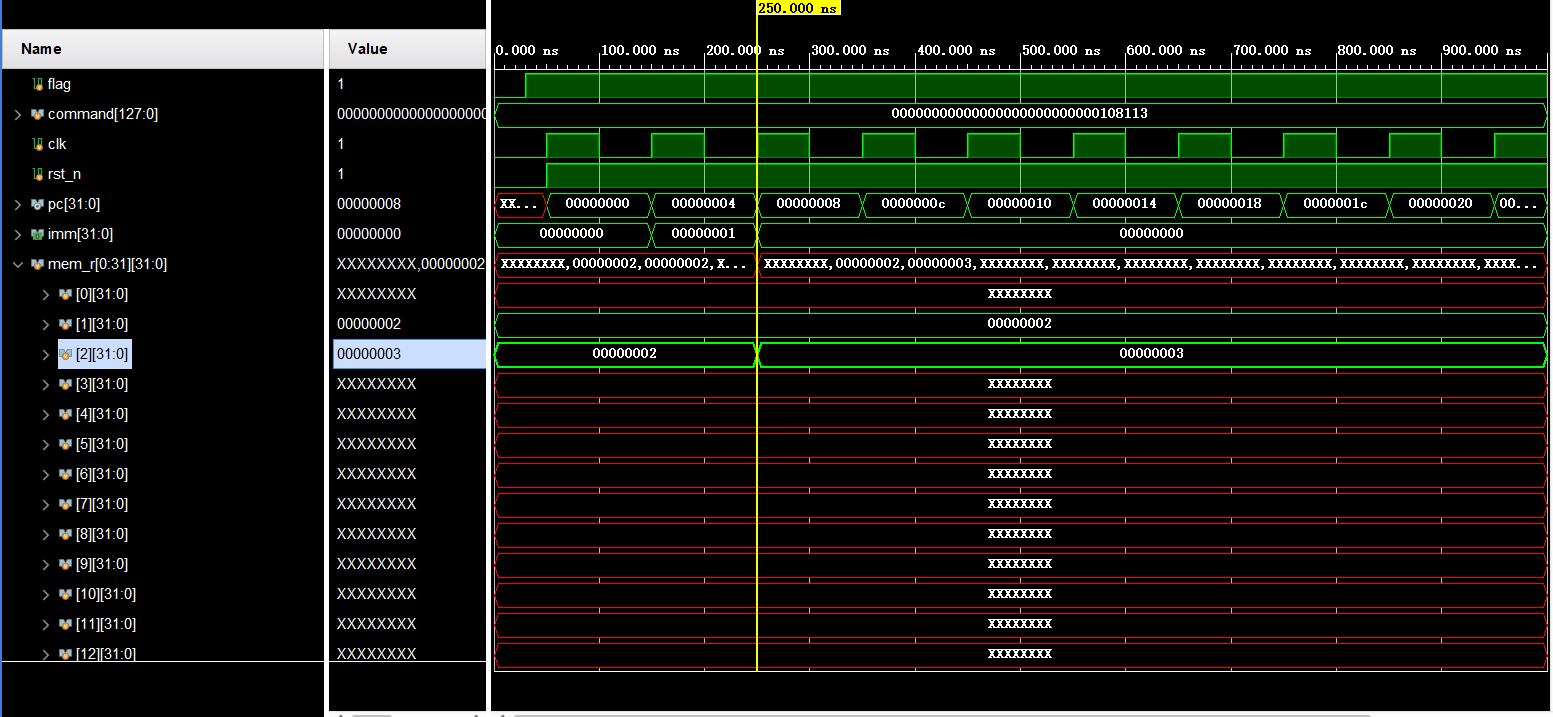
### 4.2.1 I- addi指令

**仿真代码与注释：**

command={{32'b0000\_0000\_0001\_0000\_1000\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 1 和立即数数据 1 做加法运算后为 2 存储至寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，与imm中的立即数1，进行相加，结果是0000 0003，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0003。

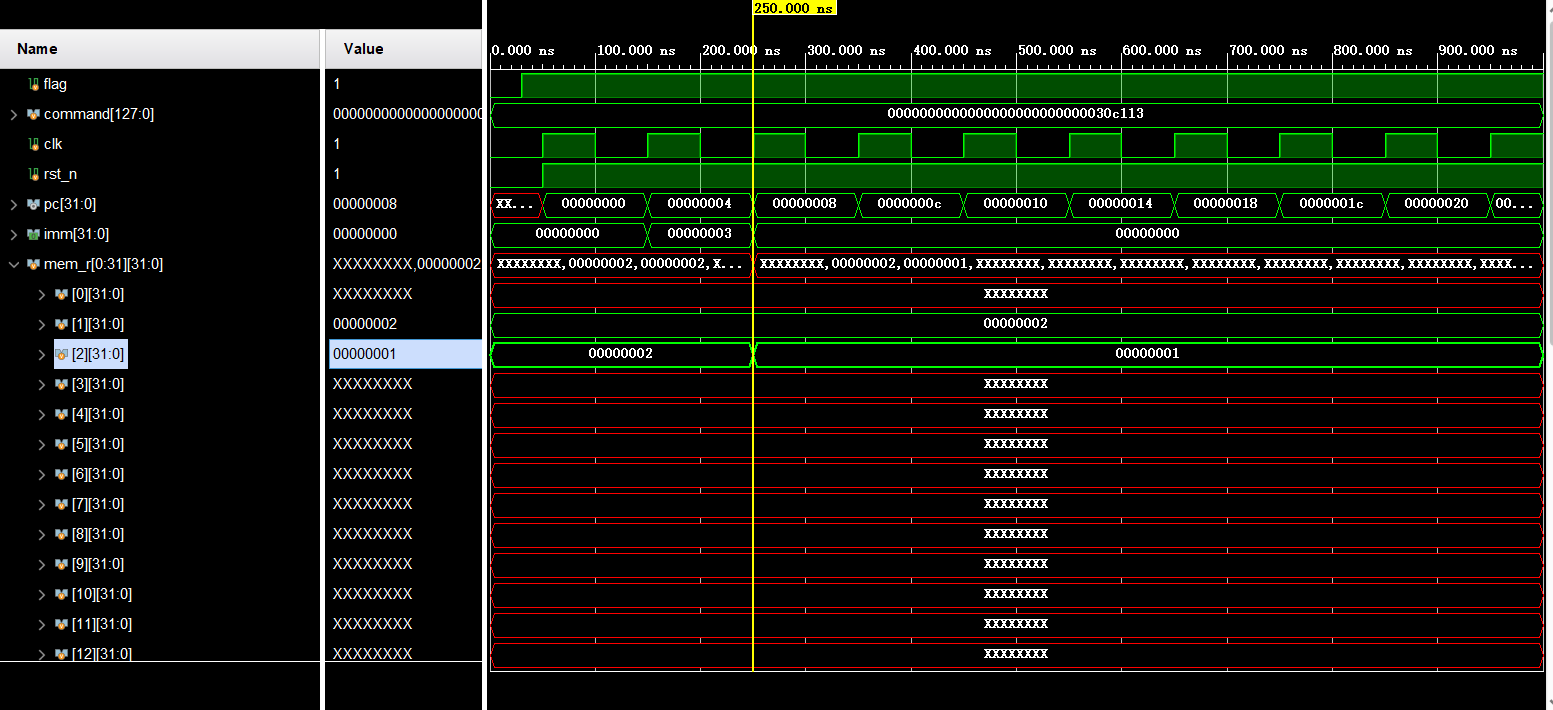
### 4.2.2 I- xori指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0011\_0000\_1100\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 2 和立即数数据 3 做位异或运算后为 1 存储至寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，二进制为0000 0010，与imm中的立即数3，二进制为0000 0011，进行按位异或，结果是0000 0001，十进制为1，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0001。

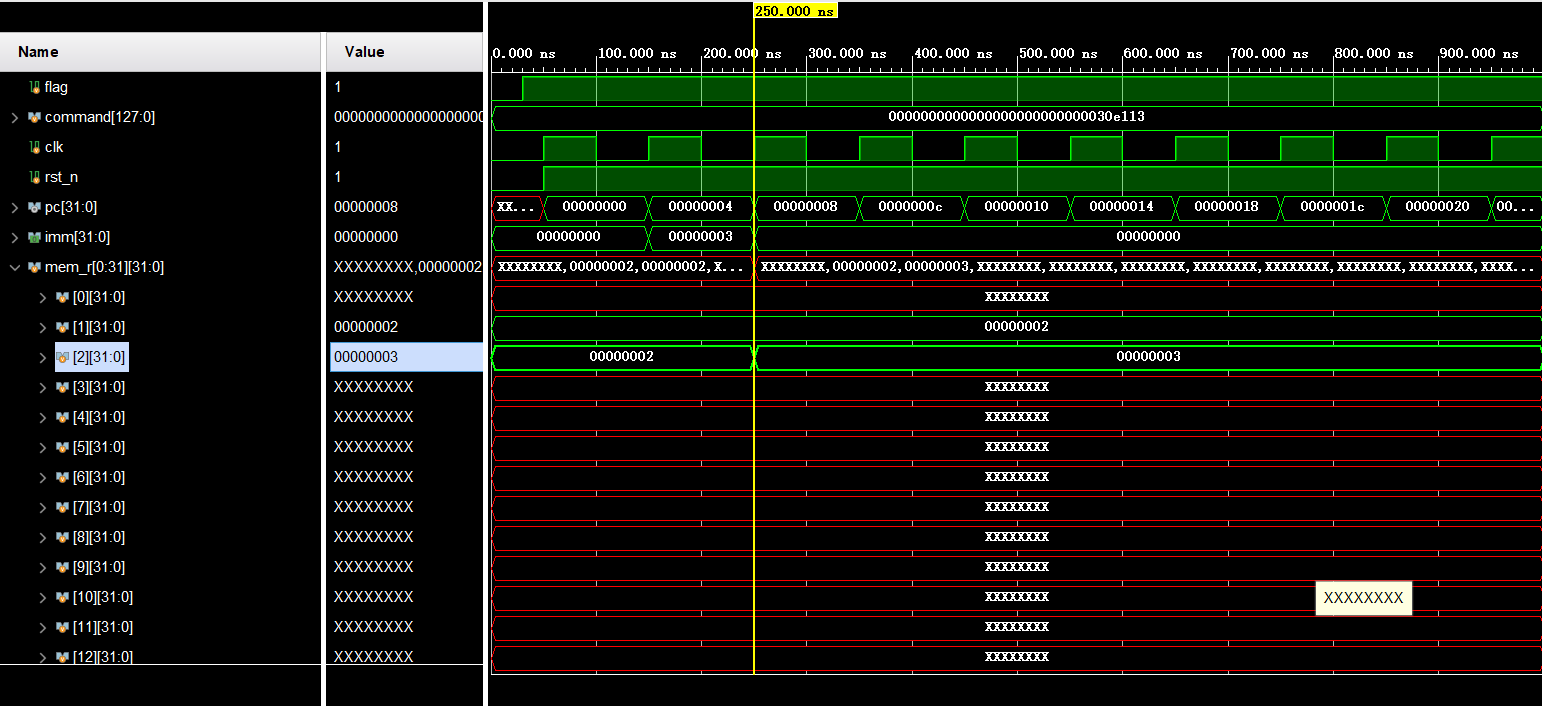
### 4.2.3 I- ori指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0011\_0000\_1110\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 2 和立即数数据 3 做位或运算后为 3 存储至寄存器地址为 2 处。

**验证结果：**



**结果分析：**

位或运算：只要对应的二个二进位有一个为1时，结果位就为1。

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，立即数处的数据0000 0003，二进制为0000 0001和0000 0011，两个数进行按位或，结果是0000 0011，即十进制数3，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0003。

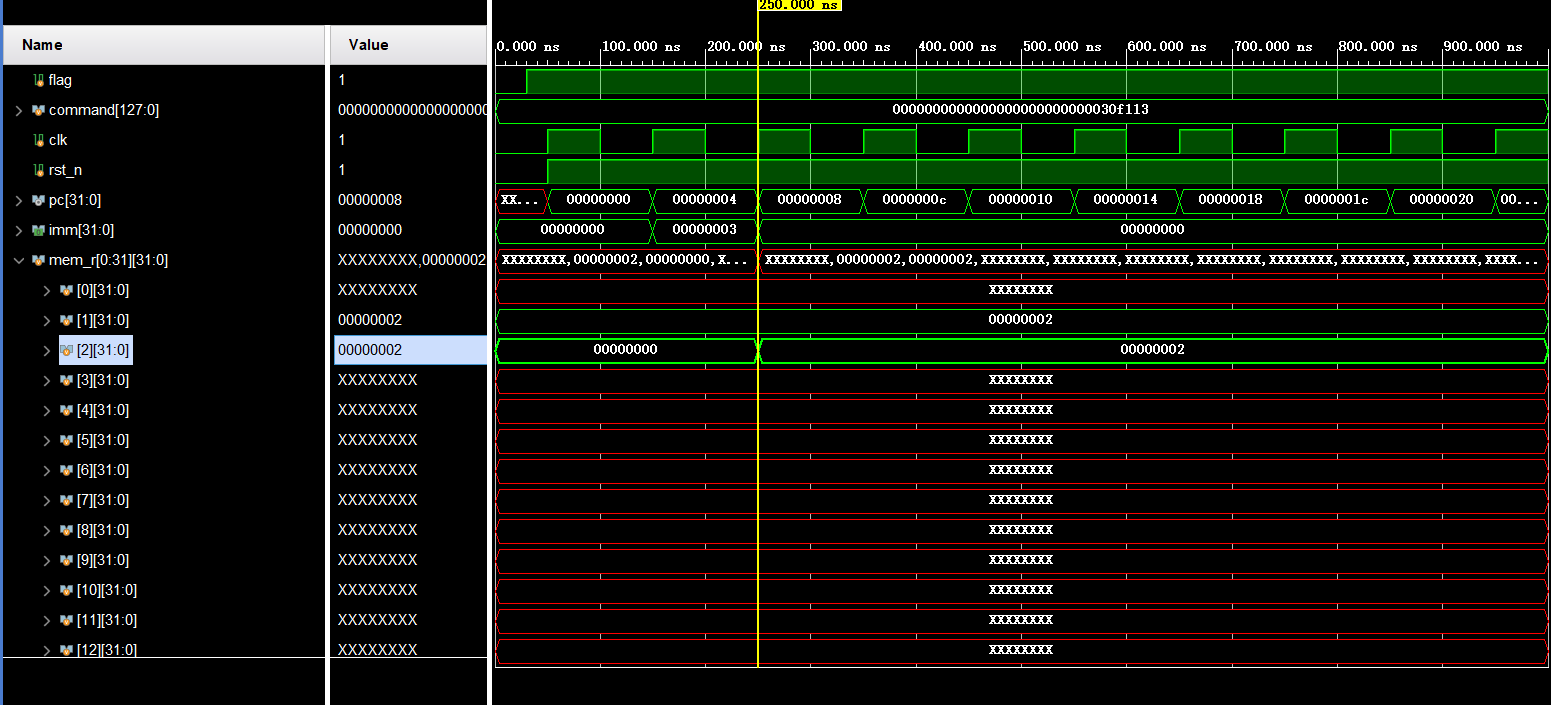
### 4.2.4 I- andi指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0011\_0000\_1111\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 2 和立即数数据 3 做位与运算后为 2 存储至寄存器地址为 2 处。

**验证结果：**



**结果分析：**

按位与运算，两个都相同且为1的时候结果为1。

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，立即数处的数据0000 0003，二进制为0000 0010和0000 0011，两个数进行按位与，结果是0000 0010，即十进制数2，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0010。

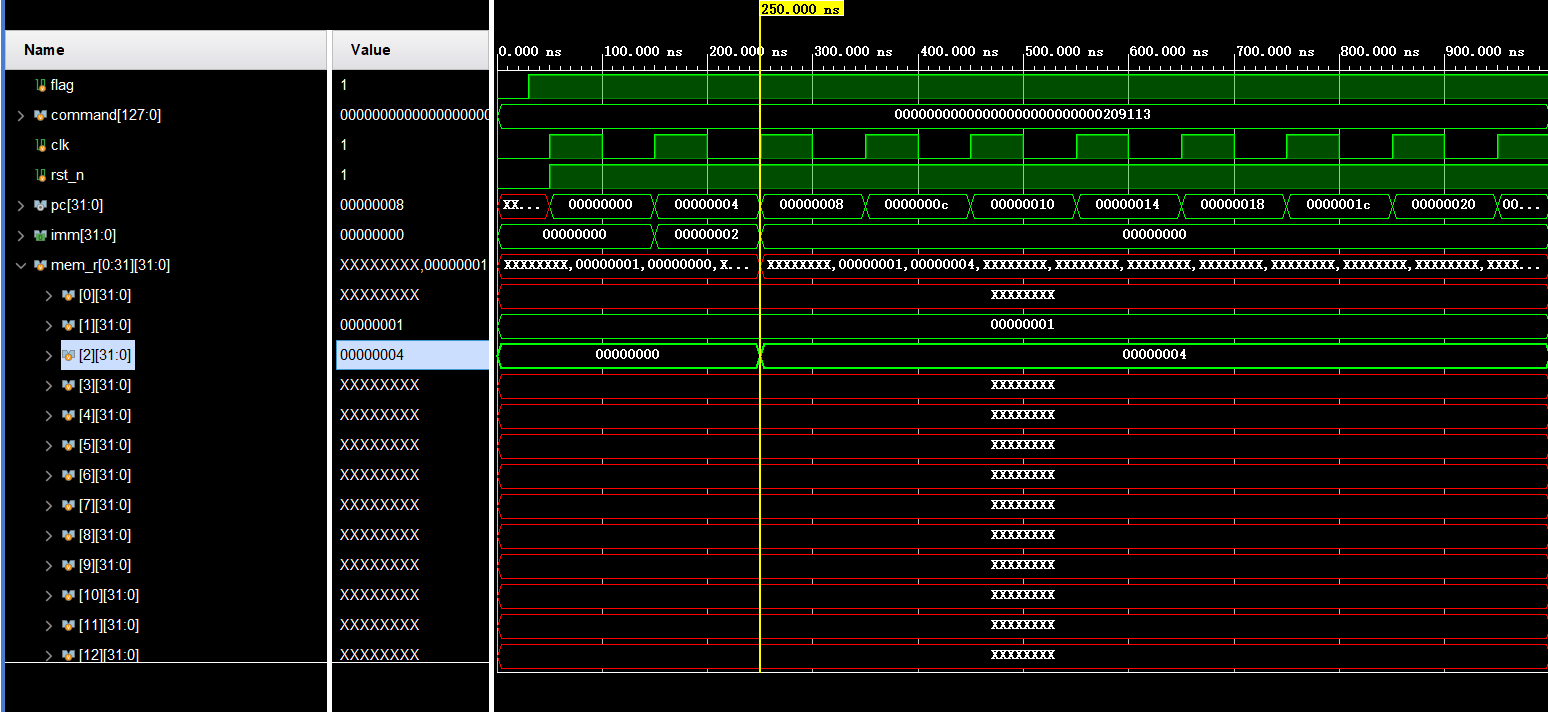
### 4.2.5 I- slli指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1001\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 1 和左移 shamt 数据 2 后为 4 存储至寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，左移立即数处的数据0000 0002，即左移2位，0000 0100，结果为十进制数4，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0100。

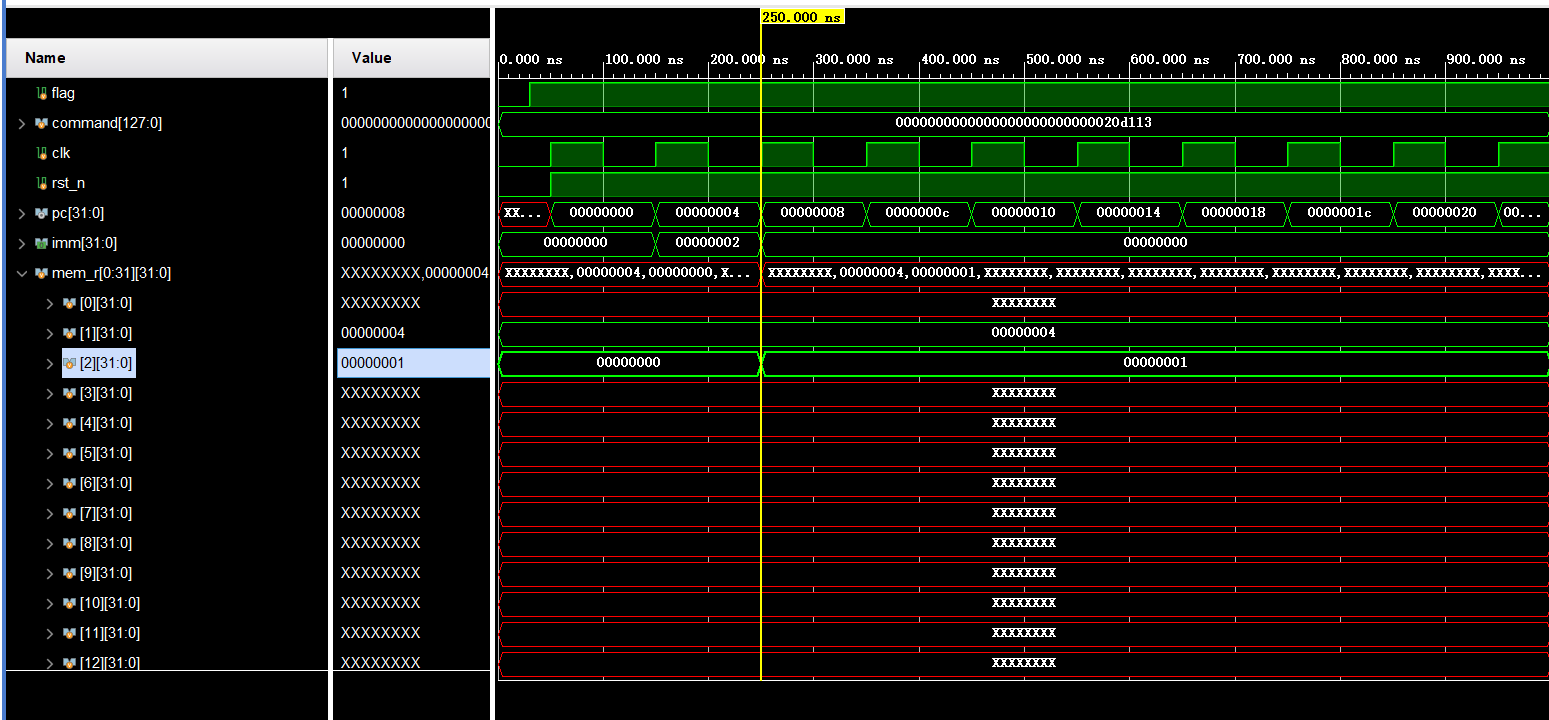
### 4.2.6 I- srli指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1101\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 4 和右移 shamt 数据 2 后为 1 存储至寄存器地址为 2 处。

**验证结果：**

**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0100，右移立即数处的数据0000 0002，即右移2位，0000 0001，结果为十进制数1，将结果存入mem\_r[2]处，即mem\_r[2] = 0000 0001。

### 4.2.7 I- srai指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0100\_0000\_0010\_0000\_1101\_0001\_0001\_0011}};

取寄存器中地址为 1 的数据 80000007 算数右移 shamt 数据 2 后为 e0000001 存储至寄存器地址为 2 处。

**验证结果：**

**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：8000 0007，右移立即数处的数据的位数，将结果存入mem\_r[2]处，即mem\_r[2] = e0000001。

### 4.2.8 I- slti指令

**仿真代码与注释：**（涉及到的关键的）

1. 寄存器负数立即数正数

command={{32'b0000\_0000\_0001\_0000\_1010\_0001\_0001\_0011}};

寄存器地址为 1 处的数据为任意负数，imm 数据为任意正数，如果地址为 1 处的数据大于 imm的数据，应将 1 存入寄存器地址 2 处。

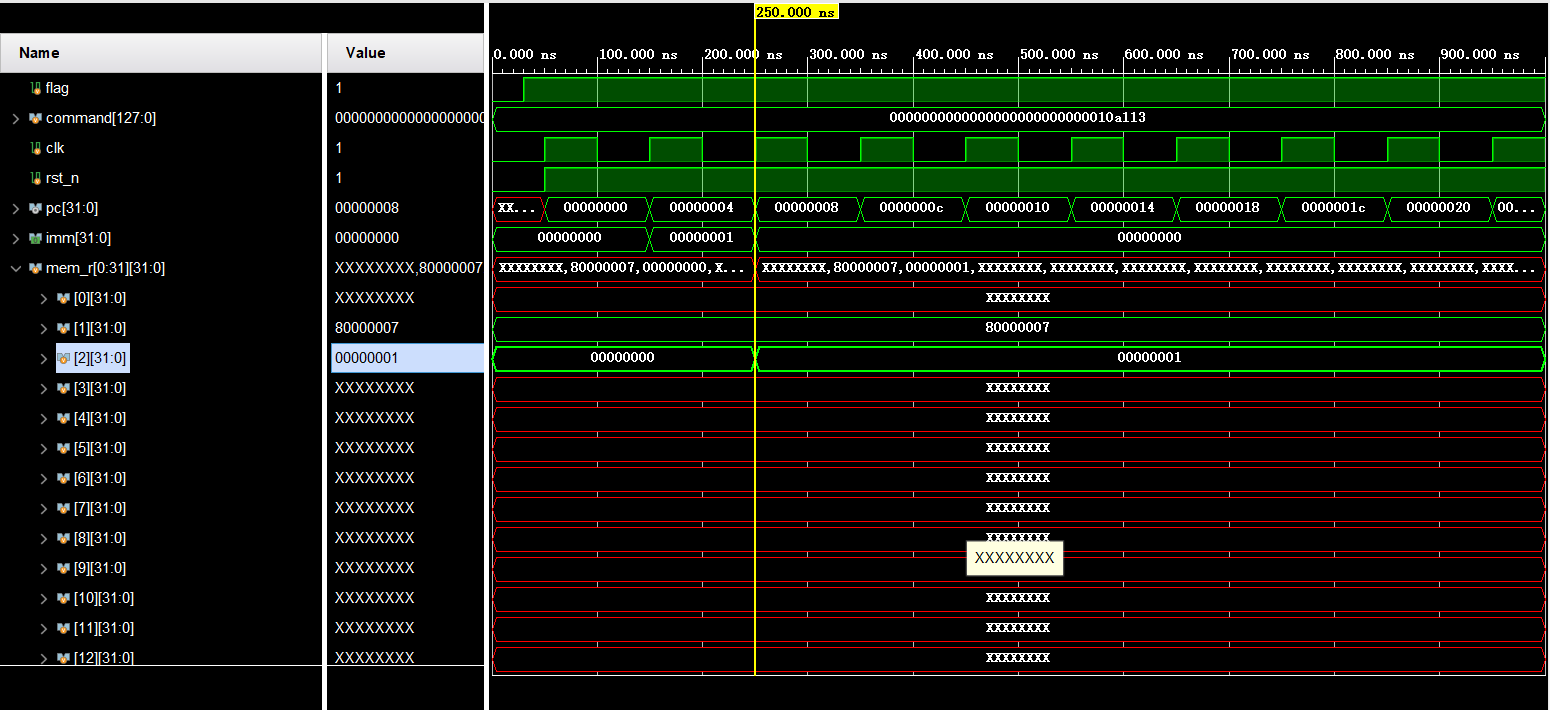
1. 立即数负数寄存器正数

command={{32'b1000\_0000\_0001\_0000\_1010\_0001\_0001\_0011}};

寄存器地址为 1 处的数据为任意正数，imm 数据为任意负数，如果地址为 1 处的数据不大于 imm 的数据，应将 0 存入寄存器地址 2 处。

**验证结果、结果分析：**

1. 寄存器负数立即数正数



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：8000 0007，imm 数据为任意正数 0000 0001，如果地址为 1 处的数据小于 imm的数据，将结果1存入mem\_r[2]处，即mem\_r[2] = 00000001。

（2）立即数负数寄存器正数



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，imm 数据为任意正数 ffff f801，如果mem\_r[1]处的数据大于（显然正数大于负数）imm的数据，将结果0存入mem\_r[2]处，即mem\_r[2] = 00000000。

### 4.2.9 I- sltiu指令

**仿真代码与注释：**（涉及到的关键的）

1. 寄存器数<imm数，结果1

command={{32'b0000\_0000\_0010\_0000\_1011\_0001\_0001\_0011}};

寄存器地址为 1 处的数据为 1，imm 数据为 2，如果 imm 的数据大于寄存器地址为 1 处的数据，应将 1 存入寄存器地址 2 处。

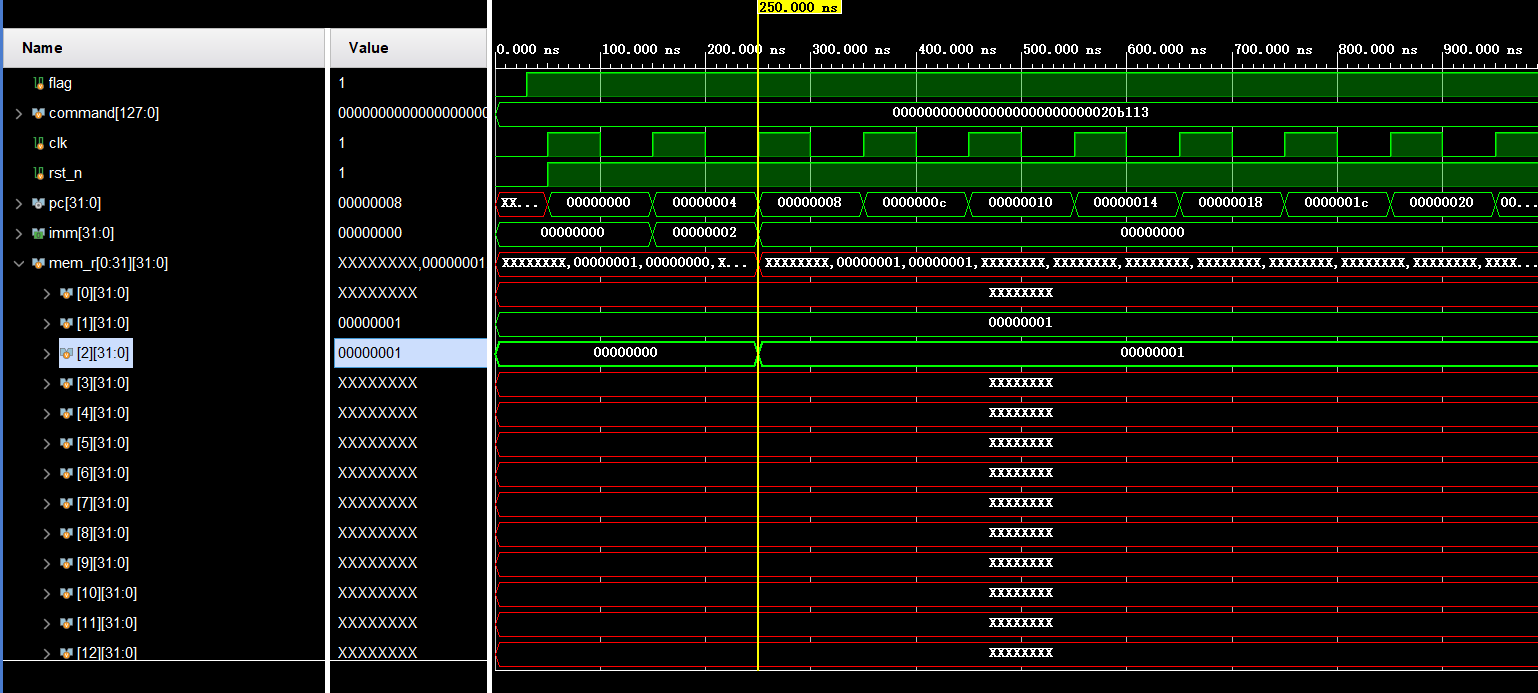
1. 寄存器数≥imm数，结果0

command={{32'b0000\_0000\_0001\_0000\_1011\_0001\_0001\_0011}};

寄存器地址为 1 处的数据为 2，imm 数据为 1，如果 imm 的数据不大于寄存器地址为 1 处的数据，应将 0 存入寄存器地址 2 处。

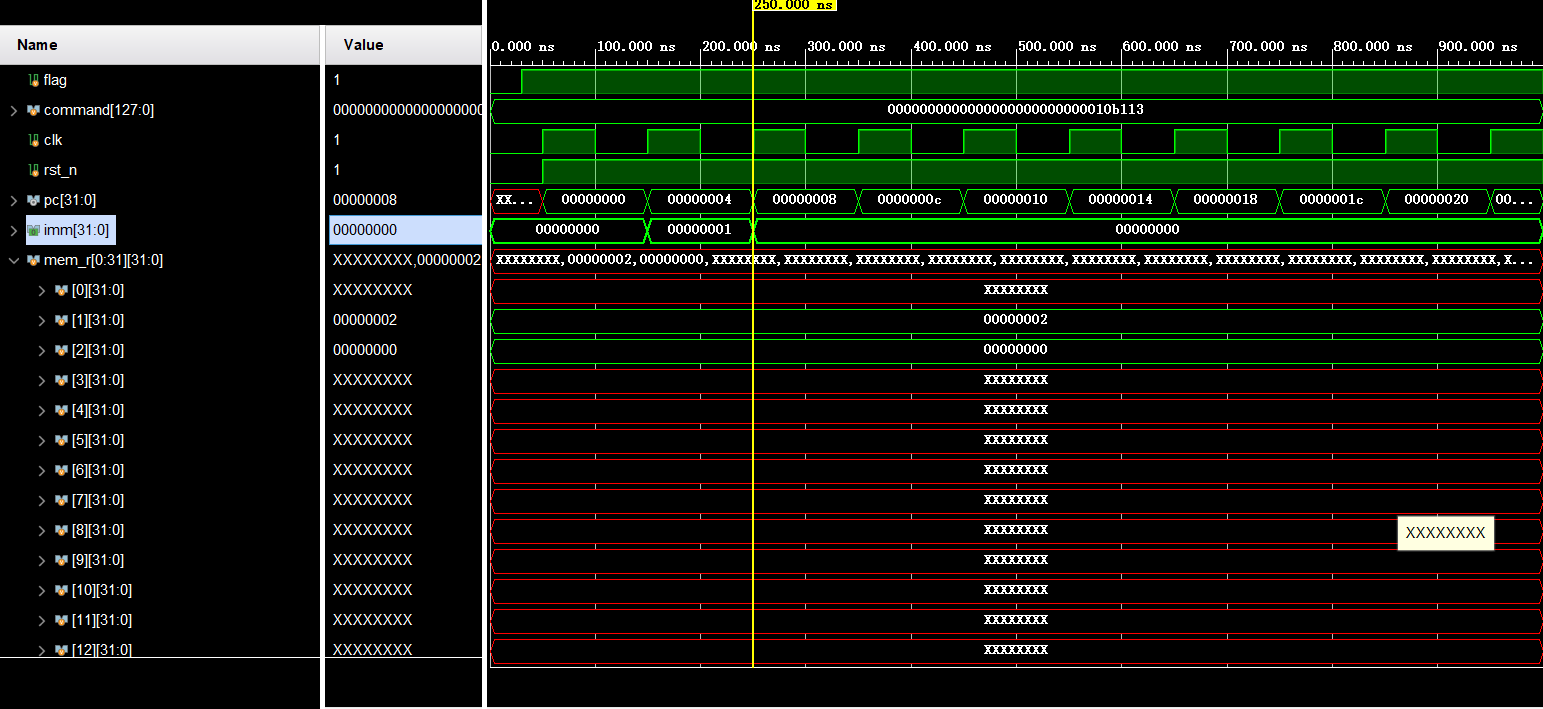
**验证结果、结果分析：**

1. 寄存器数<imm数，结果1



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，imm 数据为任意0000 0002，如果mem\_r[1]处的数据小于imm的数据，将结果1存入mem\_r[2]处，即mem\_r[2] = 00000001。

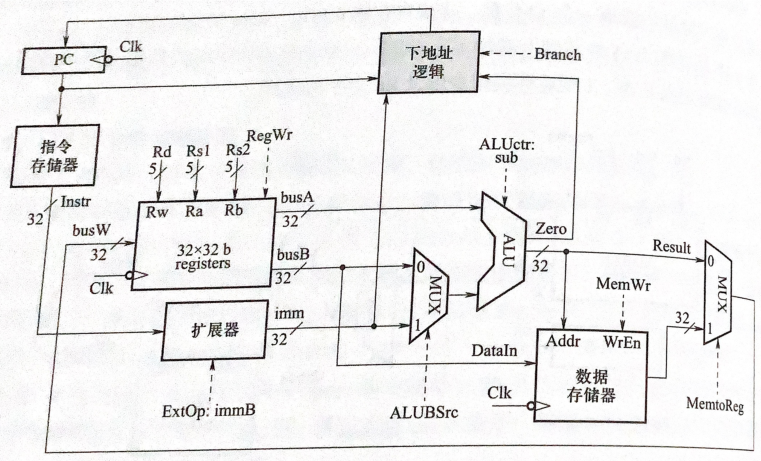
（2）寄存器数≥imm数，结果0



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，imm 数据为任意0000 0001，如果mem\_r[1]处的数据大于imm的数据，将结果0存入mem\_r[2]处，即mem\_r[2] = 00000001。

## 4.3 B- 型指令仿真结果及分析

### 4.3.0 B- 型指令数据通路



图：B- 型指令的数据通路

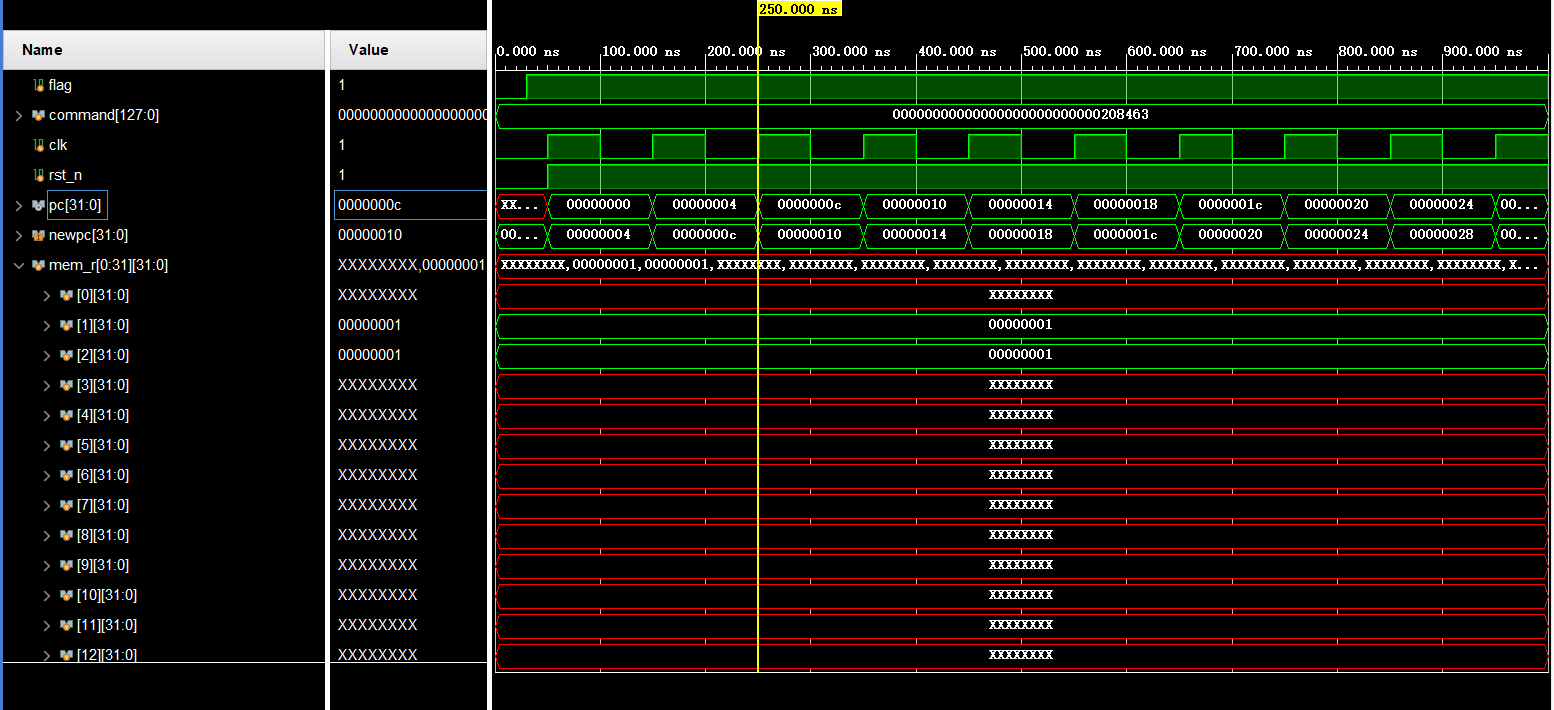
### 4.3.1 B- beq指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1000\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 2，因为 rs1 地址处的数据小于rs2 地址处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2）的值12。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0002，mem\_r[1]处的数据小于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2）的值12。

### 4.3.2 B- bne指令

**仿真代码与注释：**（涉及到的关键的）

1. 两数相等，PC跳转

command={{32'b0000\_0000\_0010\_0000\_1001\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 1，因为两地址处的数据相等，因此下周期 PC 应为当前 PC 加上 4 的值。

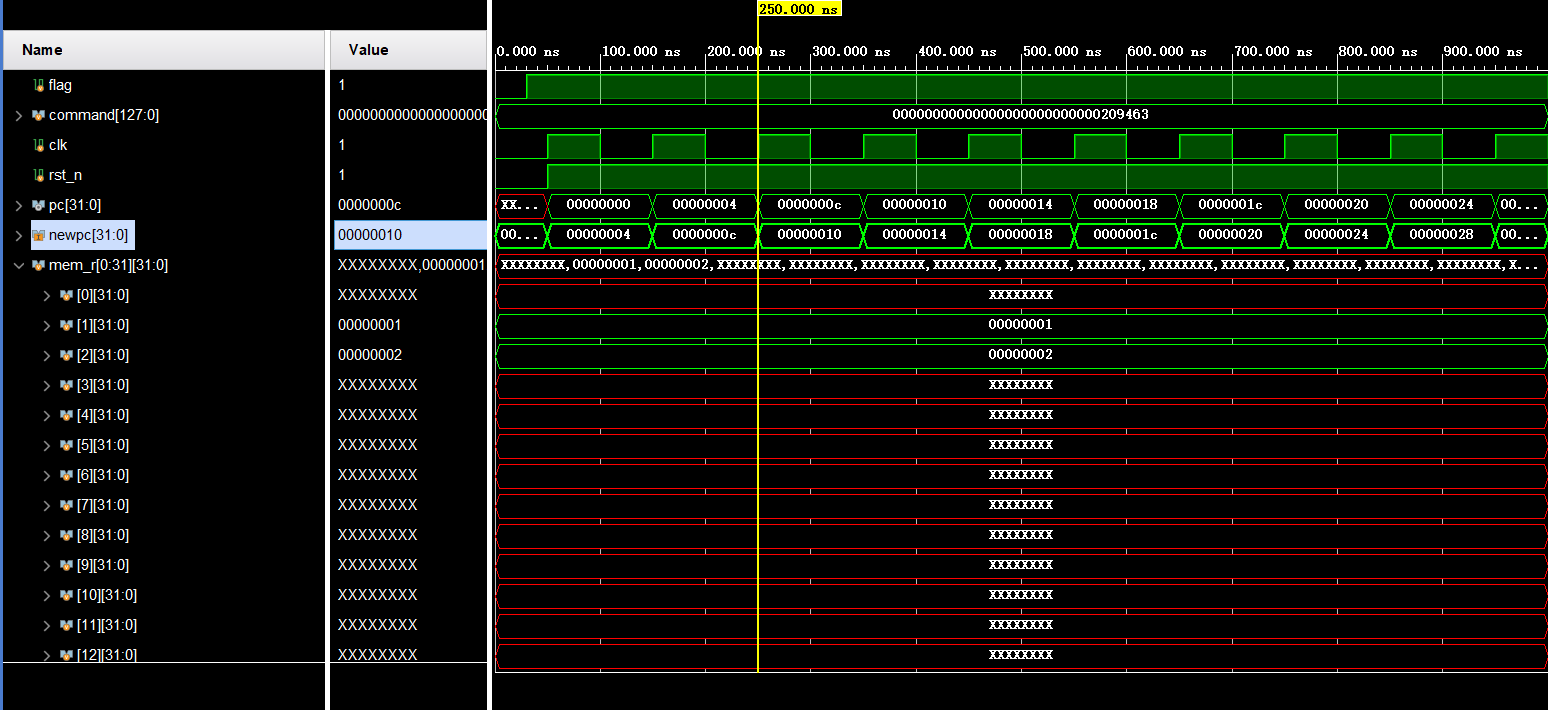
1. 两数不等，PC跳转

command={{32'b0000\_0000\_0010\_0000\_1001\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 2，因为两地址处的数据不相等，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

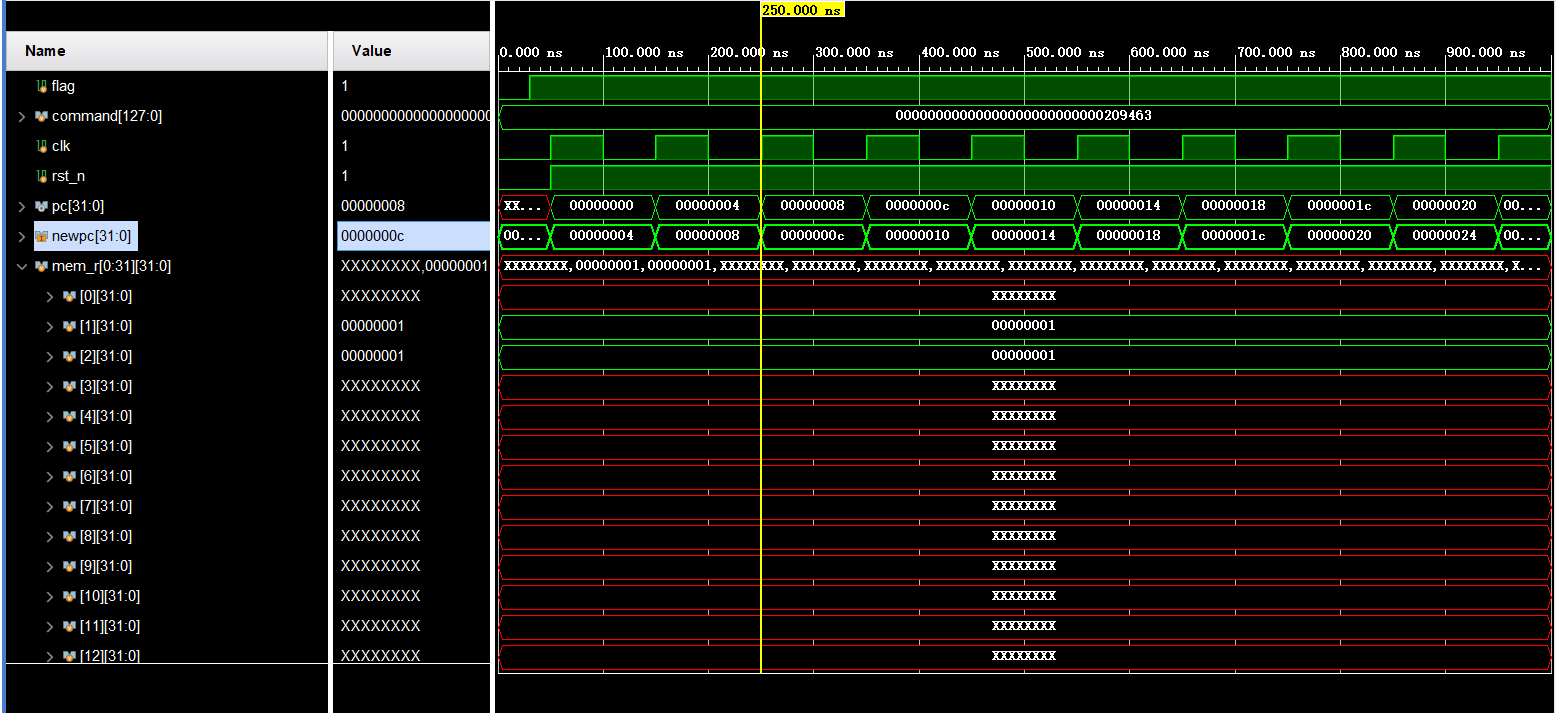
**验证结果、结果分析：**

1. 两数不等，PC跳转



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0002，mem\_r[1]处的数据不等于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8。

（2）两数相等，PC跳转



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0001，mem\_r[1]处的数据等于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上 4 的值。

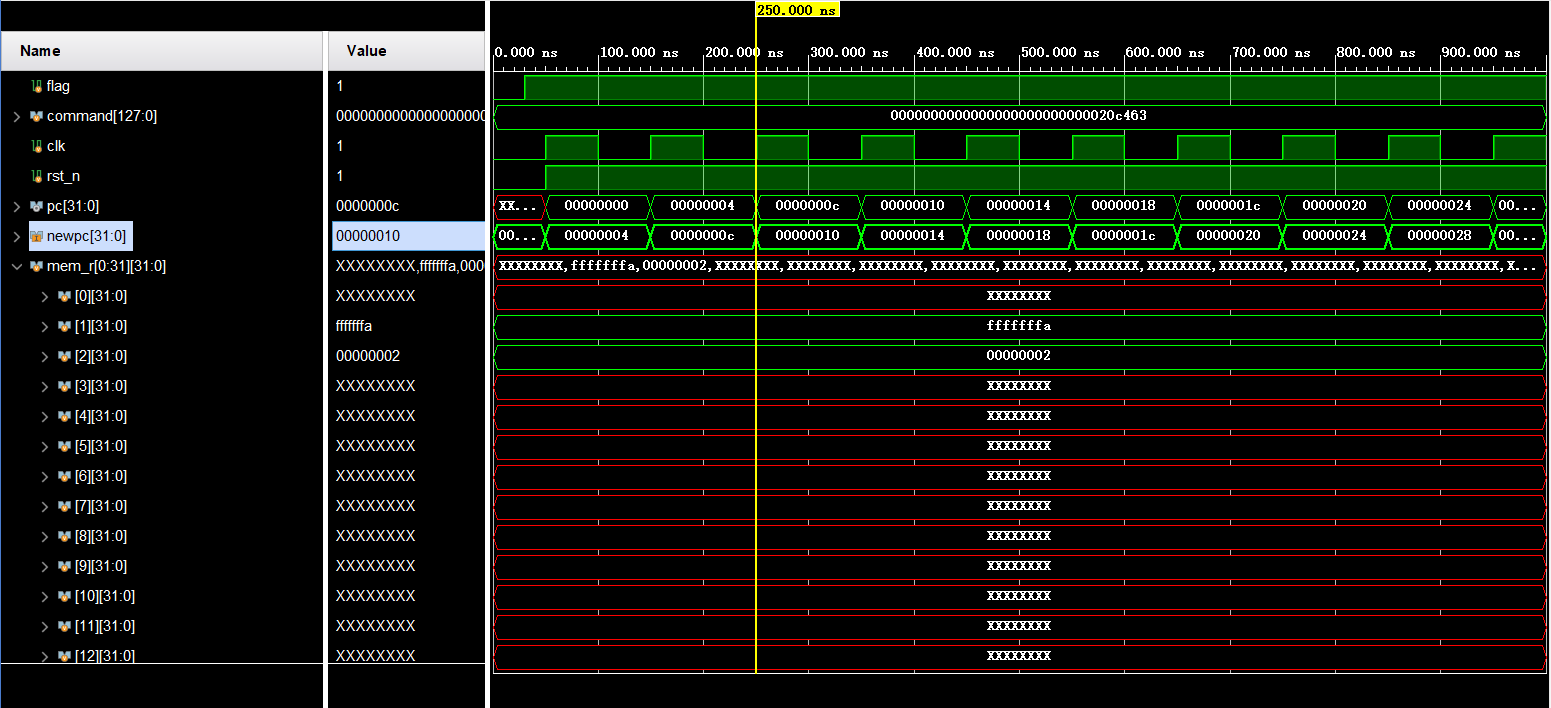
### 4.3.3 B- blt指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1100\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为任意负数，地址为 2 处的数据为任意正数，因为地址 rs1 处的负数应小于地址 rs2 处的正数，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：ffff fffa，取mem\_r[2]处的数据：0000 0002，mem\_r[1]处的数据小于mem\_r[2]处的数据（负数必然小于正数），因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

### 4.3.4 B- bge指令

**仿真代码与注释：**（涉及到的关键的）

（1）寄存器地址为 1 处的数据为任意正数，地址为 2 处的数据为任意负数

command={{32'b0000\_0000\_0010\_0000\_1101\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为任意正数，地址为 2 处的数据为任意负数，因为地址 rs1 处的正数应大于地址 rs2 处的负数，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

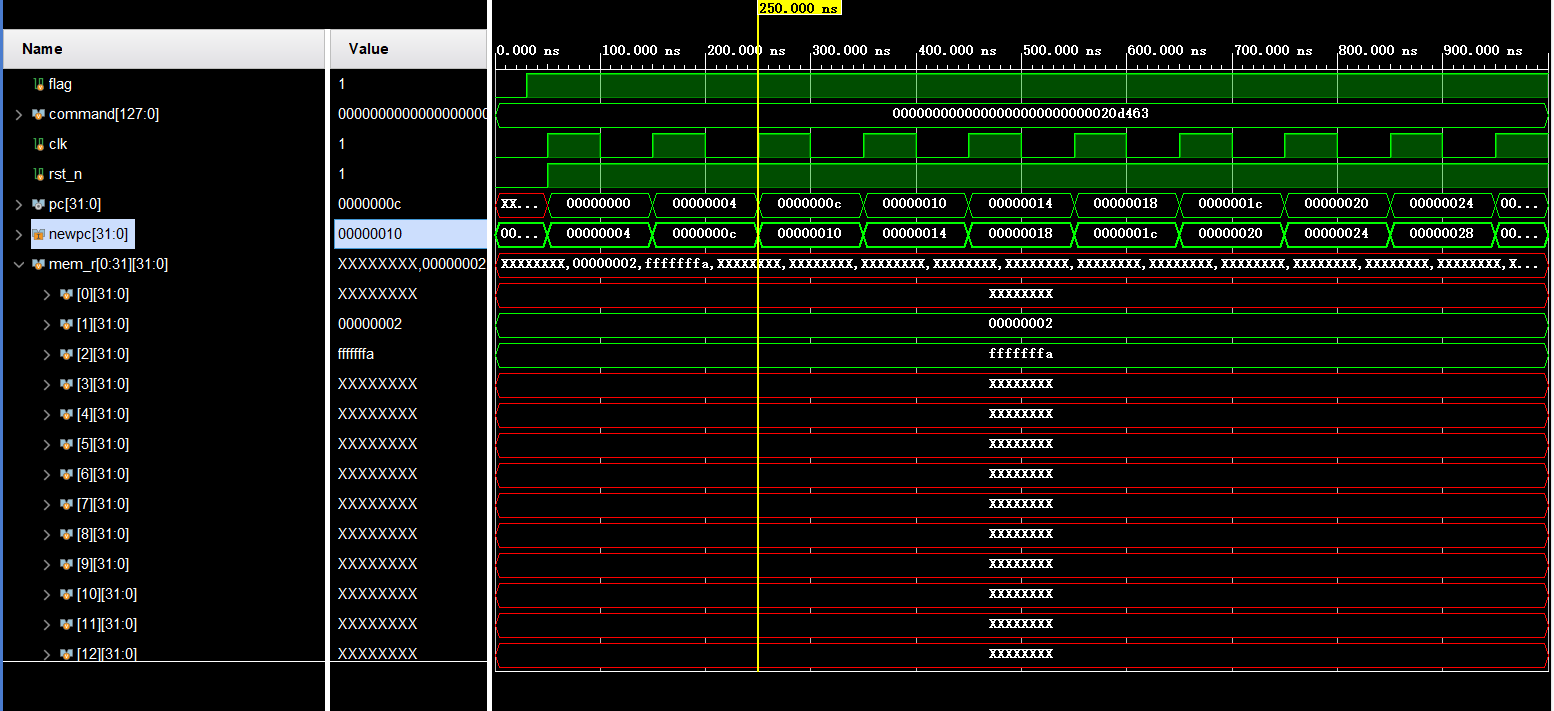
（2）寄存器地址为 1 处的数据为负数 -6，地址为 2 处的数据为负数 -6

command={{32'b0000\_0000\_0010\_0000\_1101\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为负数 -6，地址为 2 处的数据为负数 -6，因为地址 rs1 处的负数应等于地址 rs2 处的负数，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

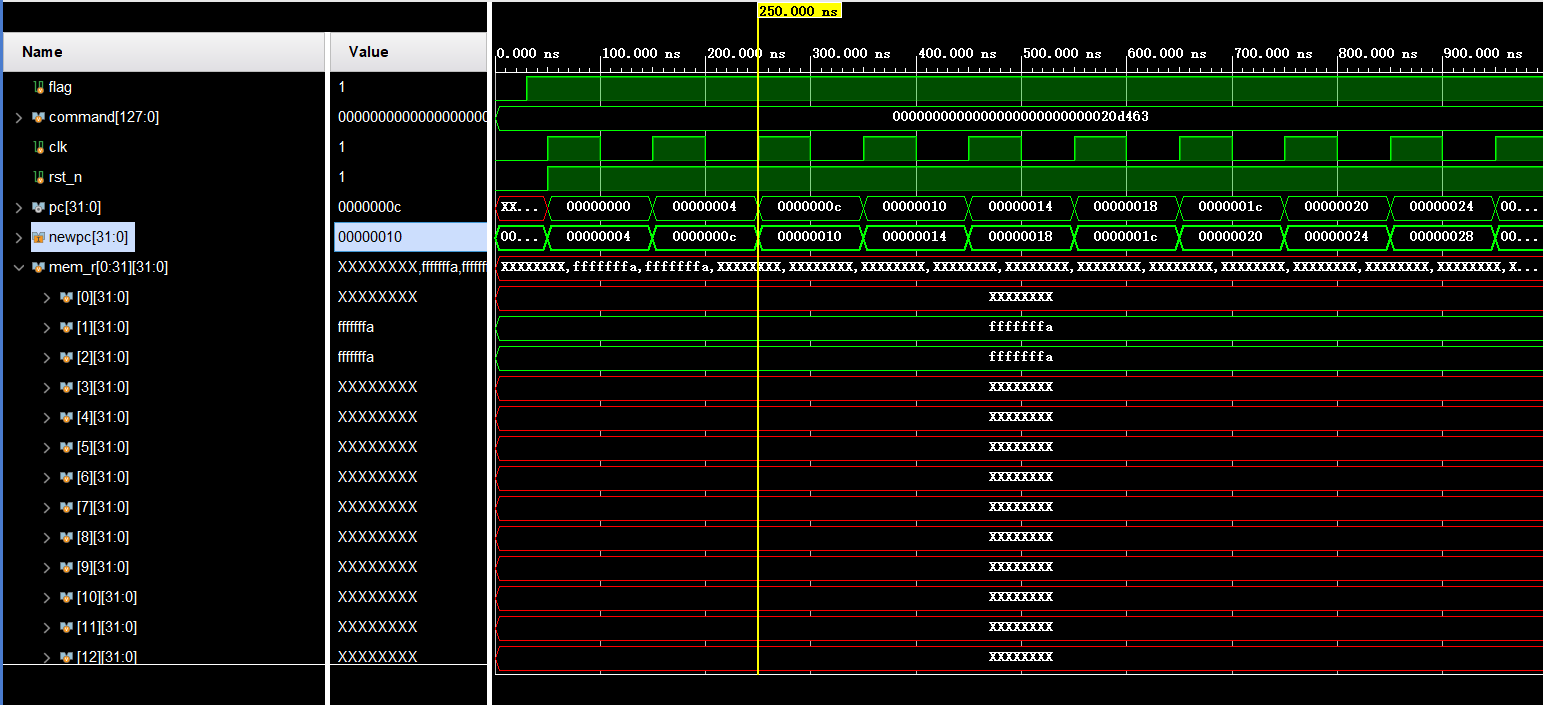
**验证结果、结果分析：**

（1）寄存器地址为 1 处的数据为任意正数，地址为 2 处的数据为任意负数



这里是mem\_r[]寄存器中，取mem\_r[2]处的数据：ffff fffa，取mem\_r[1]处的数据：0000 0002，mem\_r[1]处的数据大于mem\_r[2]处的数据（正数必然大于负数），因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

（2）寄存器地址为 1 处的数据为负数 -6，地址为 2 处的数据为负数 -6



这里是mem\_r[]寄存器中，取mem\_r[2]处的数据：ffff fffa，取mem\_r[1]处的数据：ffff fffa，mem\_r[1]处的数据等于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

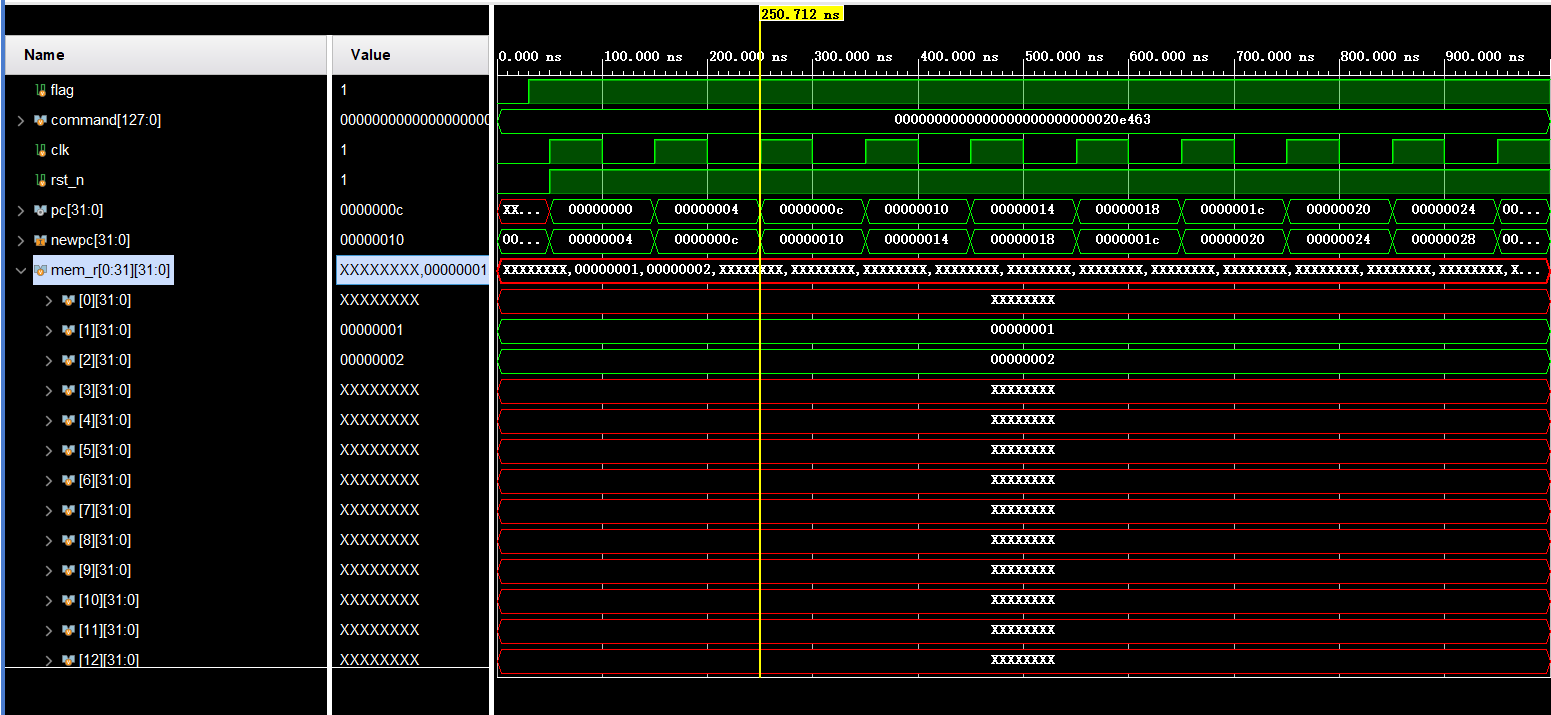
### 4.3.5 B- bltu指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1110\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 2，因为 rs1 地址处的数据小于 rs2 地址处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0002，mem\_r[1]处的数据小于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

### 4.3.6 B- bgeu指令

**仿真代码与注释：**（涉及到的关键的）

1. 两数相等，PC跳转

command={{32'b0000\_0000\_0010\_0000\_1111\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 1，因为 rs1 地址处的数据等于 rs2 地址处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

1. 寄存器1值>寄存器2值，PC跳转

command={{32'b0000\_0000\_0010\_0000\_1111\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 2，地址为 2 处的数据为 1，因为 rs1 地址处的数据大于 rs2 地址处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

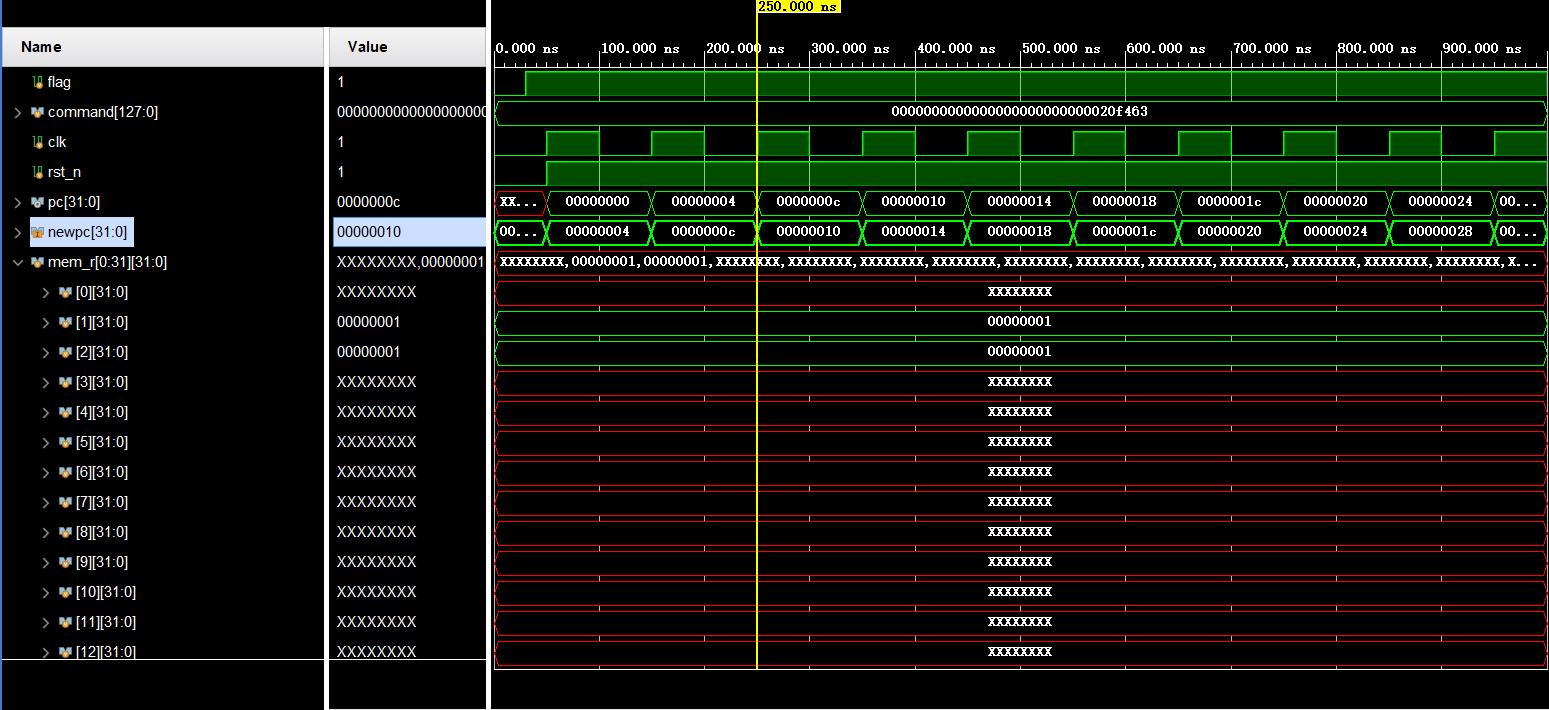
1. 寄存器1值<寄存器2值，PC跳转

command={{32'b0000\_0000\_0010\_0000\_1111\_0100\_0110\_0011}};

寄存器地址为 1 处的数据为 1，地址为 2 处的数据为 2，因为 rs1 地址处的数据等于 rs2 地址处的数据，因此下周期 PC 应为当前 PC 加上 4 的值 4 。

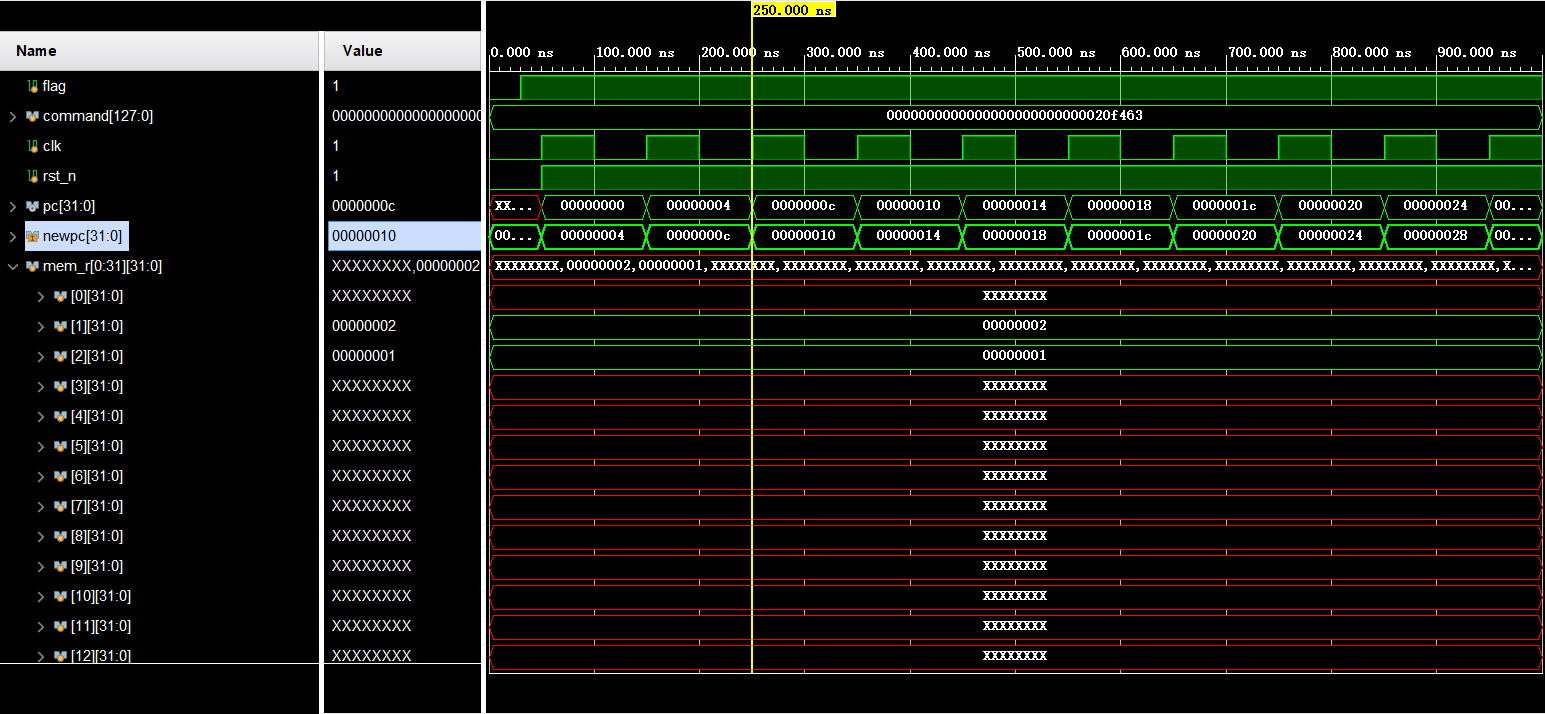
**验证结果、结果分析：**

1. 两数相等，PC跳转



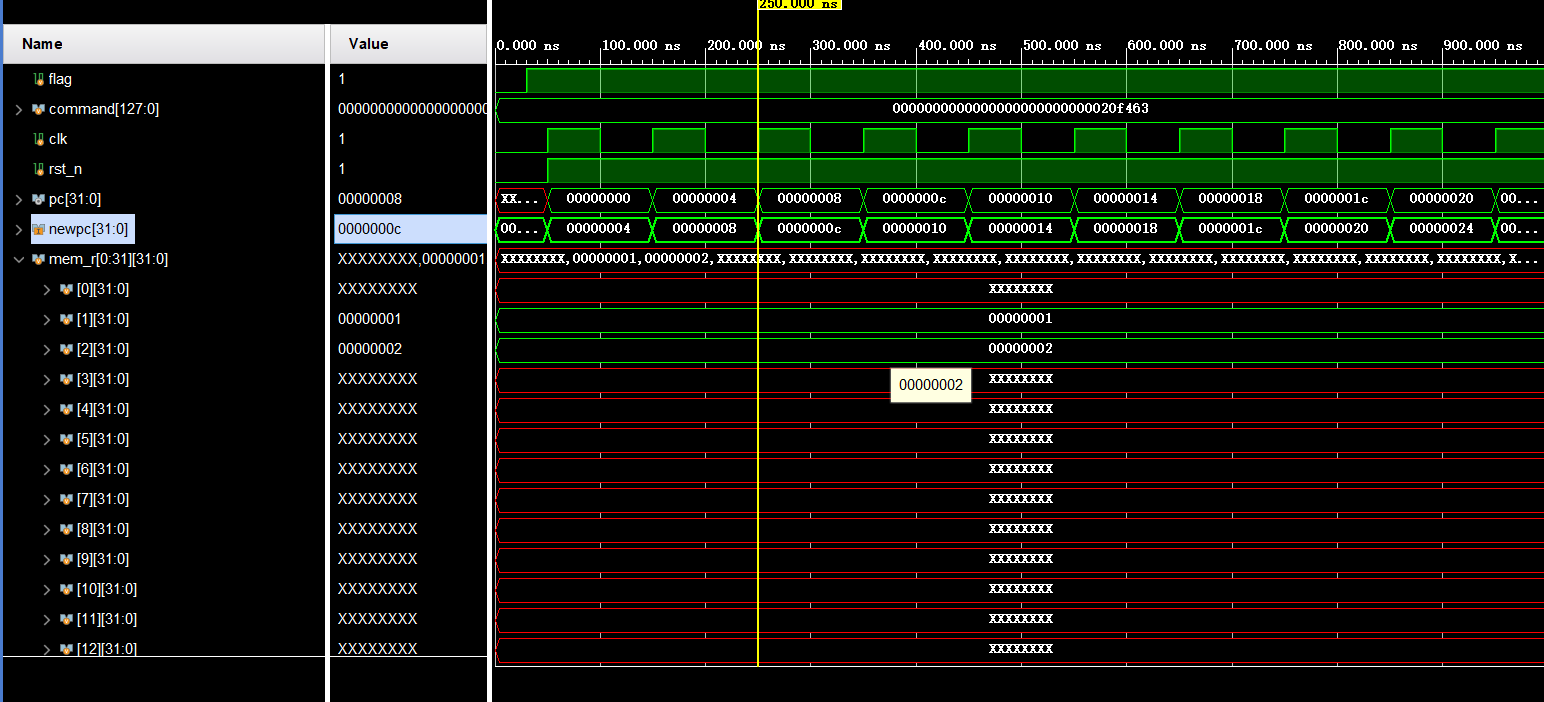
这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0001，mem\_r[1]处的数据等于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

1. 寄存器1值>寄存器2值，PC跳转



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0002，取mem\_r[2]处的数据：0000 0001，mem\_r[1]处的数据大于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上立即数（imm\*2） 的值 8 。

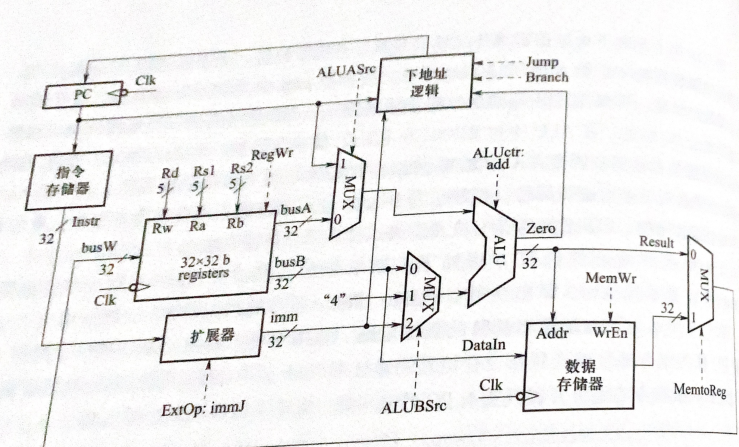
（3）寄存器1值<寄存器2值，PC跳转



这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0001，取mem\_r[2]处的数据：0000 0002，mem\_r[1]处的数据小于mem\_r[2]处的数据，因此下周期 PC 应为当前 PC 加上 4 的值 4 。

## 4.4 J- 型指令仿真结果及分析

### 4.4.0 J- 型指令的数据通路



图：J- 型指令的数据通路

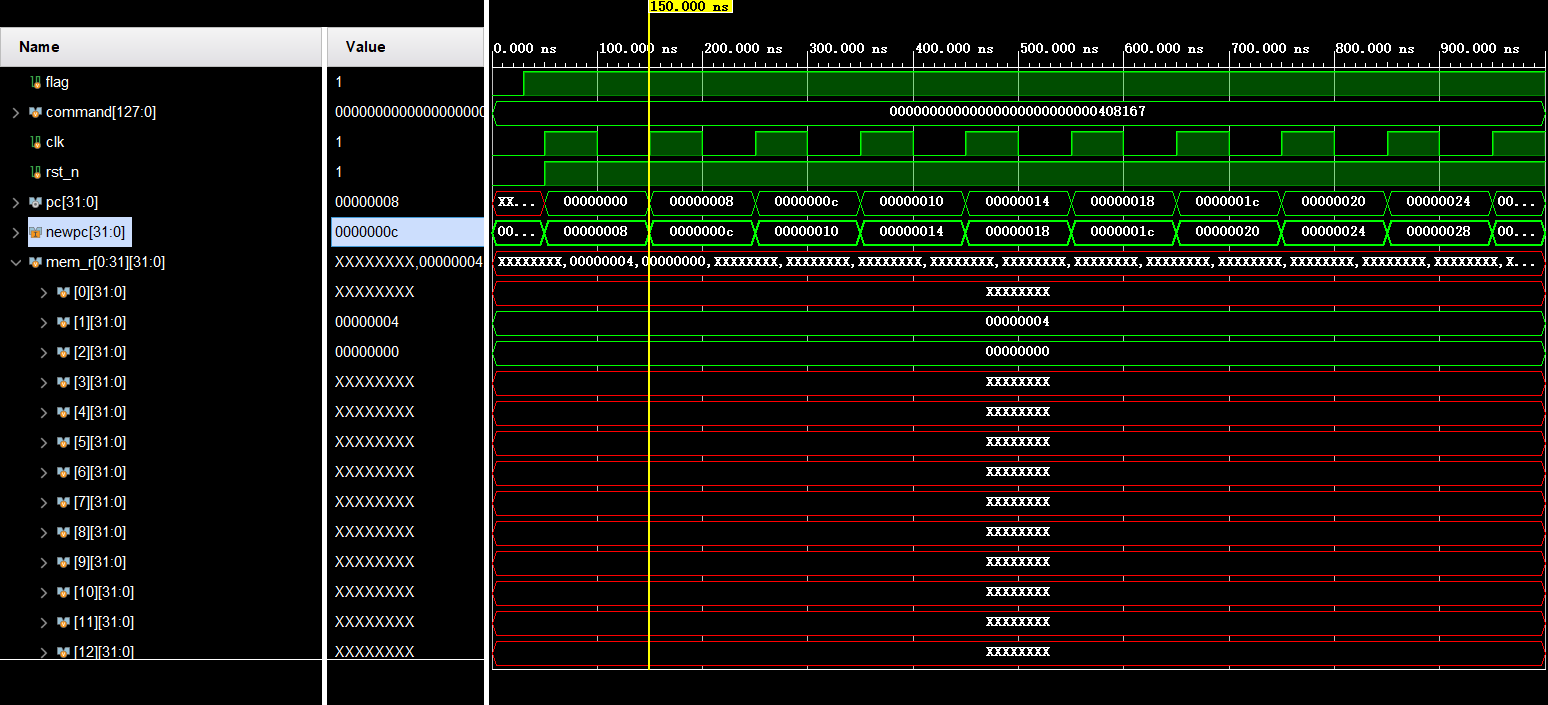
### 4.4.1 J- jalr指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0100\_0000\_1000\_0001\_0110\_0111}};

将当前 PC 值为 0 加 4 后送入寄存器地址 2 处，然后将寄存器地址 1 处数据 4 加上立即数扩展后的值 4 最终为 8，因此最终 R[1] 因为 4，下一周期 PC 应为 8。

**验证结果：**

**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0004，取mem\_r[2]处的数据：0000 0000，当前 PC 值为 0 加 4 后送入寄存器地址 2 处，然后将寄存器地址 1 处数据 4 加上立即数扩展后的值 4 最终为 8，因此最终 mem\_r[1] 为 4，下一周期 PC 应为 8。

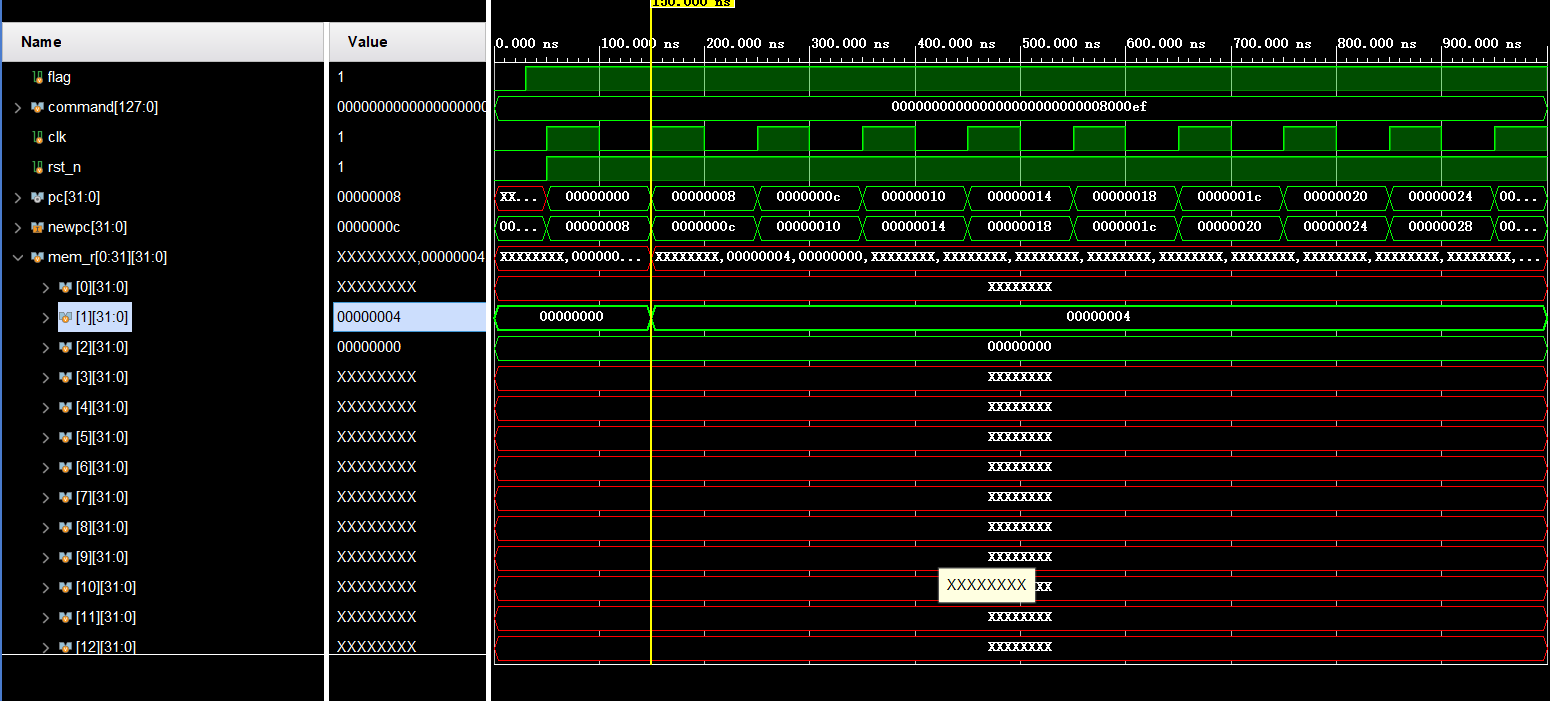
### 4.4.2 J- jal指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_1000\_0000\_0000\_0000\_1110\_1111}};

将当前 PC 值为 0 加 4 后送入寄存器地址 1 处，然后将当前 PC 加上立即数扩展后乘 2 的值为 8，因此最终 R[1] 因为 4，下一周期 PC 应为 8。

**验证结果：**

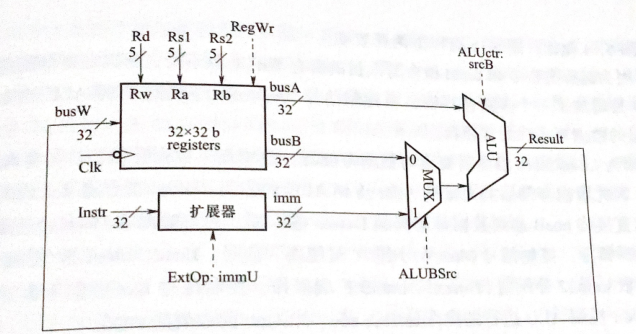


**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0000，取mem\_r[2]处的数据：0000 0000，将当前 PC 值为 0 加 4 后送入寄存器地址 1 处，mem\_r[1]处的数据：0000 0004，然后将当前 PC 加上立即数扩展后乘 2 的值为 8，因此最终mem\_r[1]为 4，下一周期 PC 应为 8。

## 4.5 U- 型指令仿真结果及分析

### 4.5.0 U- 型指令的数据通路



### 4.5.1 U- lui指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0000\_0000\_0001\_0000\_1011\_0111}};

将立即数扩展后为 00001000 并将其存储至寄存器地址为 1 处，因此最终 R[1] 应为 00001000 。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0000，取mem\_r[2]处的数据：0000 0000，将立即数扩展后为 0000 1000 并将其存储至寄存器地址为 1 处，因此最终mem\_r[1]处的数据应为 0000 1000。

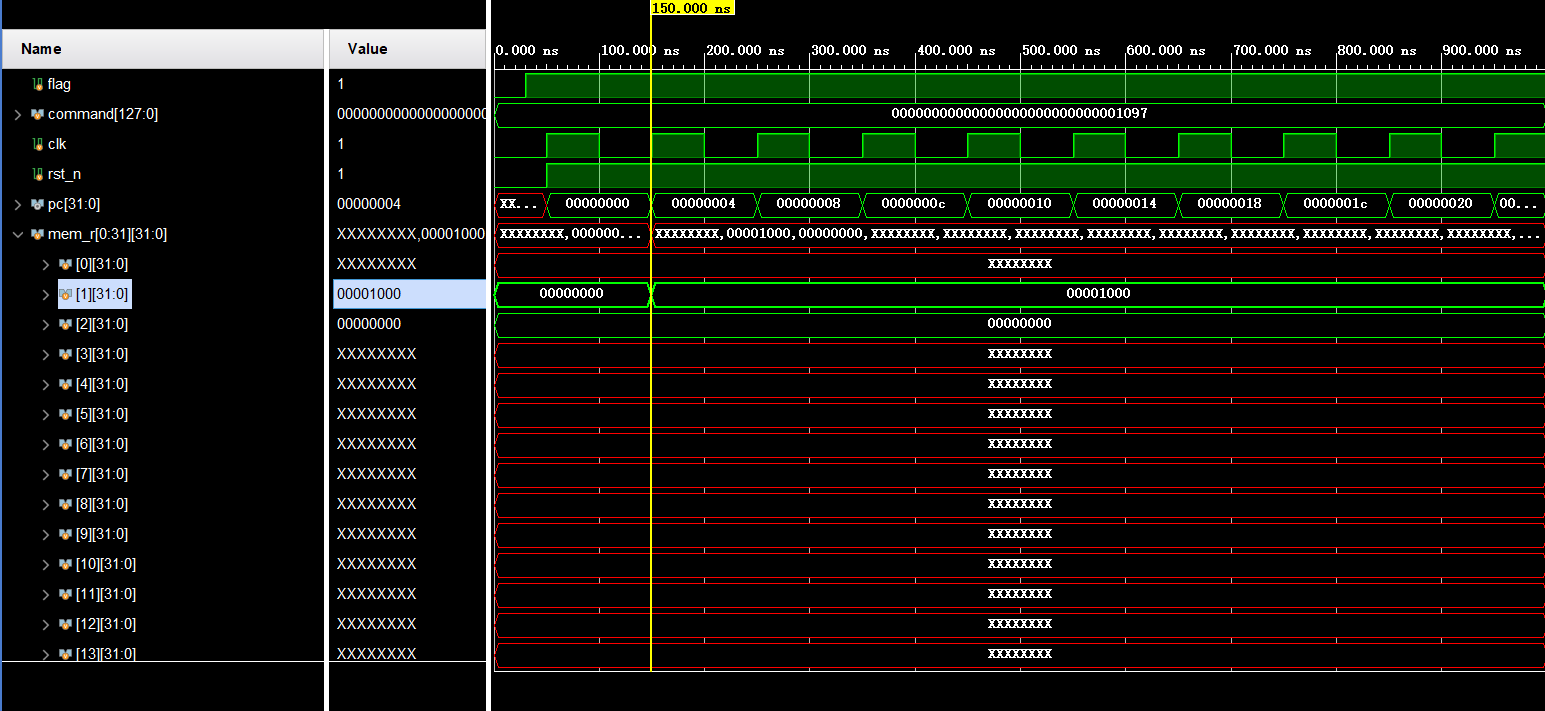
### 4.5.2 U- auipc指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0000\_0000\_0001\_0000\_1001\_0111}};

将立即数扩展后为 00001000 并将其与当前 PC 相加后存储至寄存器地址为 1 处，因此最终 R[1] 应为 00001000 。

**验证结果：**

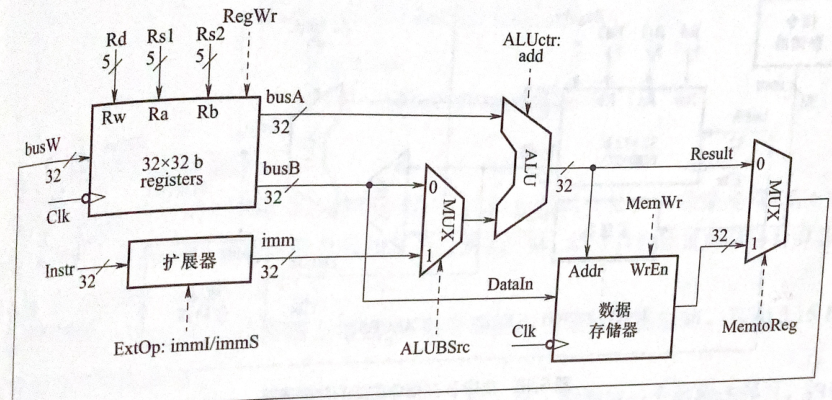


**结果分析：**

这里是mem\_r[]寄存器中，取mem\_r[1]处的数据：0000 0000，取mem\_r[2]处的数据：0000 0000，将立即数扩展后为 0000 1000 并将其与当前 PC 相加后存储至寄存器地址为 1 处，因此最终mem\_r[1]应为 00001000 。

## 4.6 L- 型指令仿真结果及分析

### 4.6.0 L- 型指令的数据通路



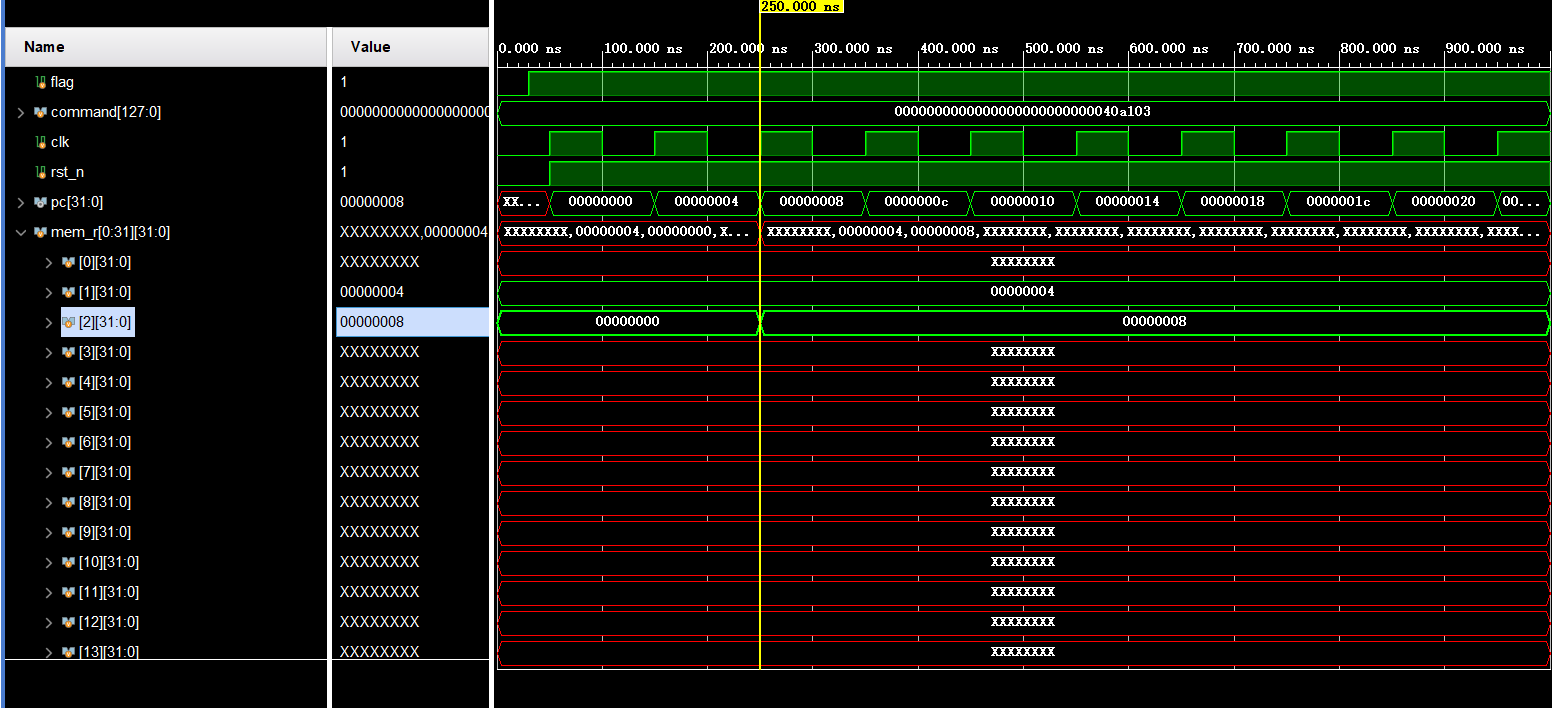
### 4.6.1 L- lw指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0100\_0000\_1010\_0001\_0000\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化数据存储器中地址为 2 处的数据为 00000008 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据 00000008 后直接保存到寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，mem\_r[2]处的数据为 00000008，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据 00000008 后直接保存到mem\_r[2]处。

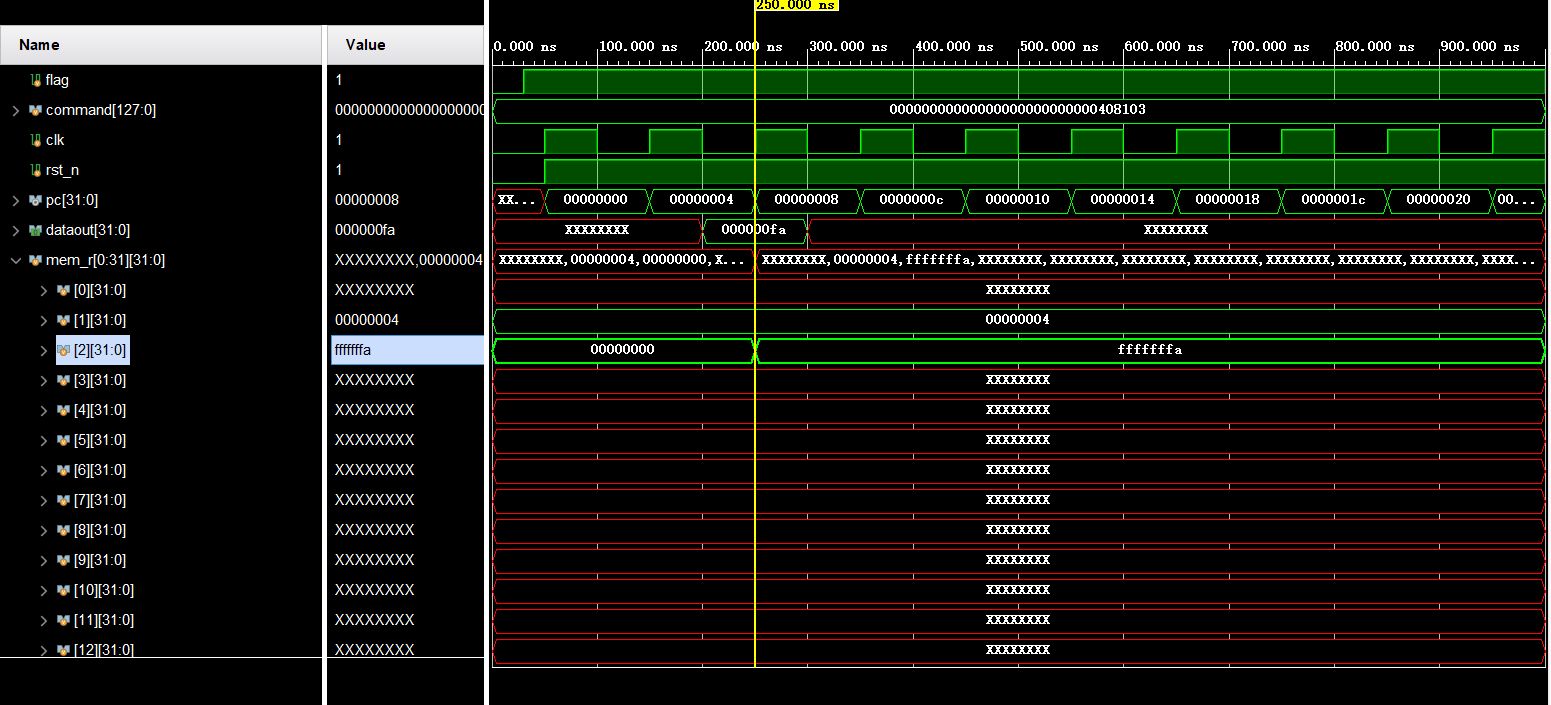
### 4.6.2 L- lb指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b 0000\_0000\_0100\_0000\_1000\_0001\_0000\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化数据存储器中地址为 2 处的数据为 00000008 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低八位进行有符号扩展后为 fffffffa 保存到寄存器地址为 2 处。

**验证结果：**

**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低八位进行有符号扩展后为 fffffffa 保存到mem\_r[2]处。

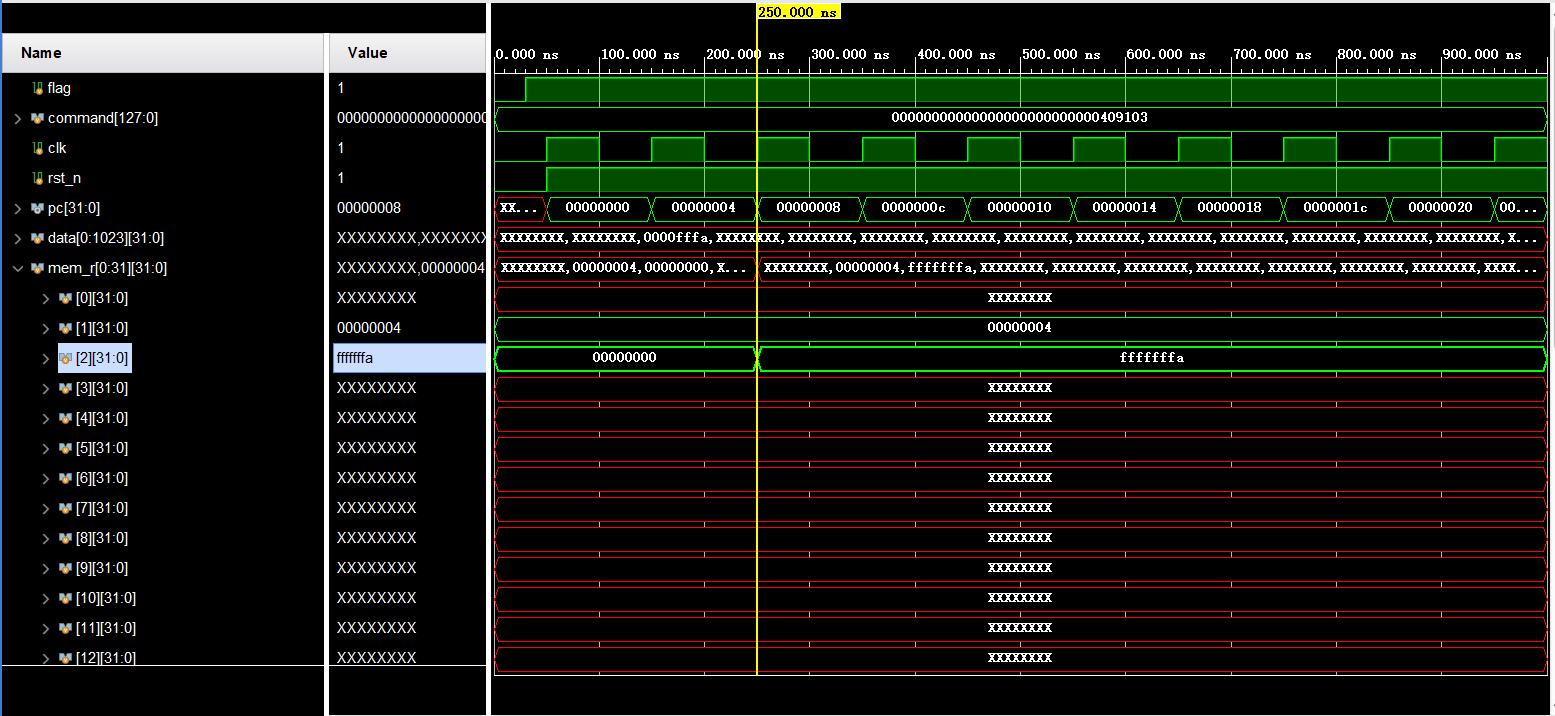
### 4.6.3 L- lh指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0100\_0000\_1001\_0001\_0000\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化数据存储器中地址为 2 处的数据为 00000008 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低十六位进行有符号扩展后为 fffffffa 保存到寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低十六位进行有符号扩展后为 fffffffa 保存到mem\_r[2]处。

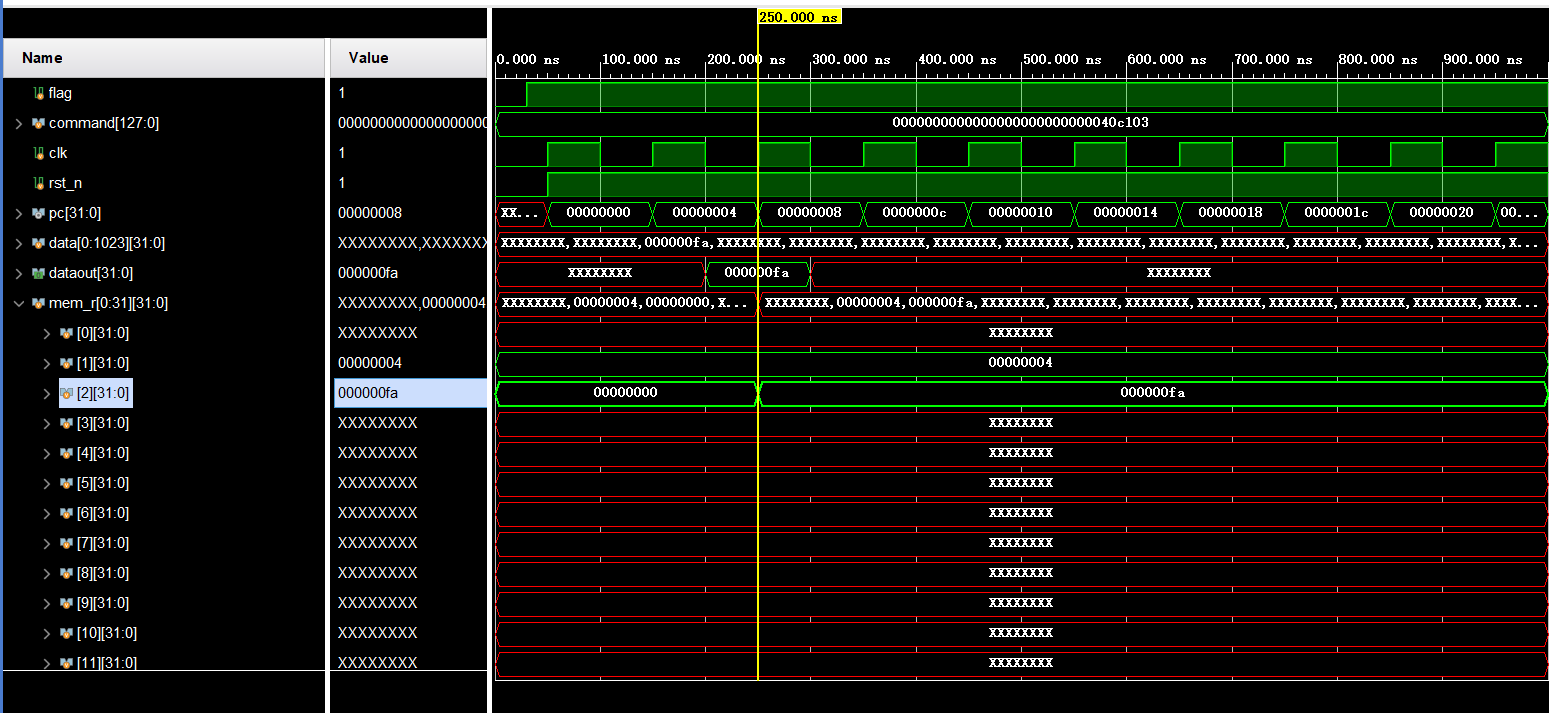
### 4.6.4 L- lbu指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0100\_0000\_1100\_0001\_0000\_0011}}

初始化寄存器地址为 1 处的数据为 4 并初始化数据存储器中地址为 2 处的数据为 00000008 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低八位进行无符号扩展后为 000000fa保存到寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低十六位进行有符号扩展后为 fffffffa保存到mem\_r[2]处。

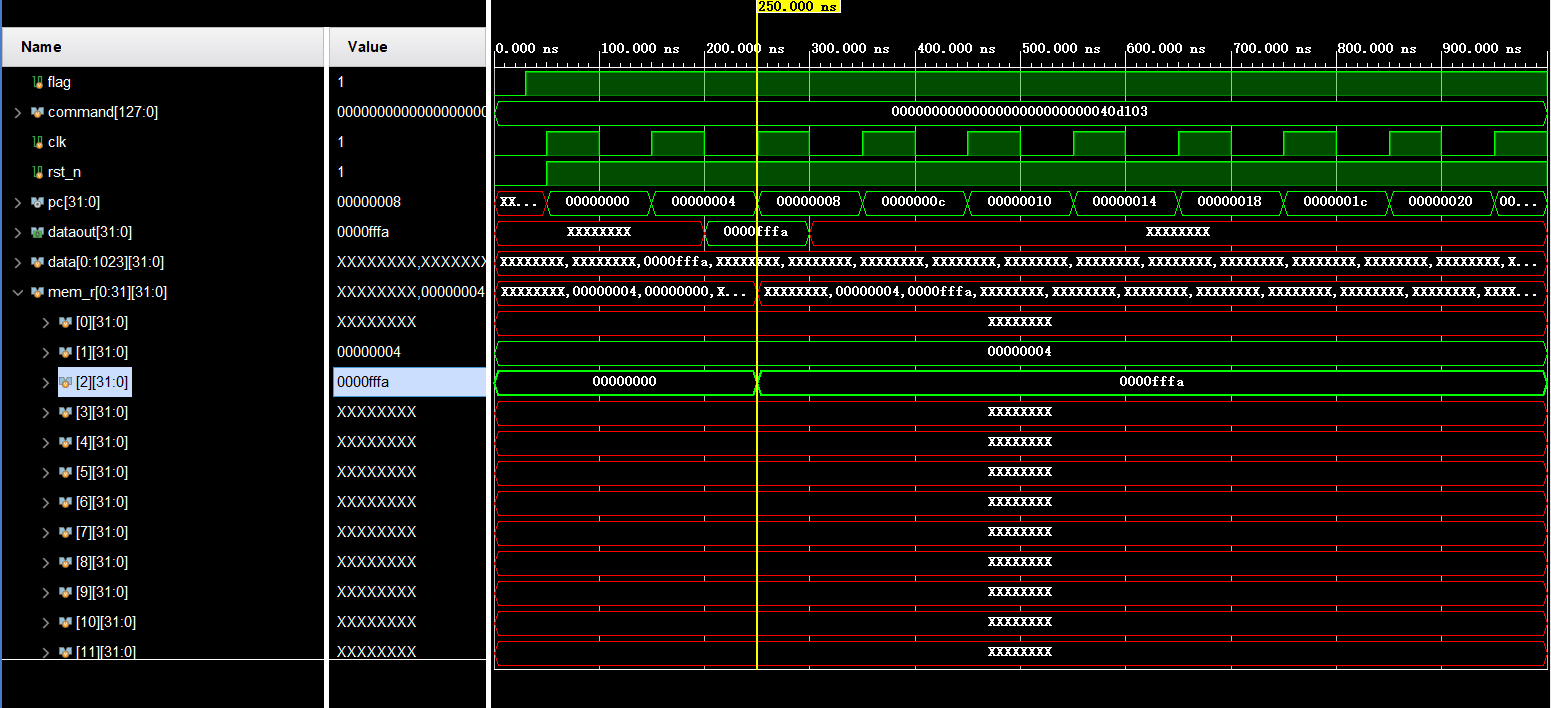
### 4.6.5 L- lhu指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0100\_0000\_1101\_0001\_0000\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化数据存储器中地址为 2 处的数据为 00000008 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 0000fffa 取低十六位进行无符号扩展后为 0000fffa 保存到寄存器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并在从存储器获取到数据后 000000fa 取低十六位进行有符号扩展后为 fffffffa 保存到寄存器地址为 mem\_r[2] 处。

## 4.7 S- 型指令仿真结果及分析

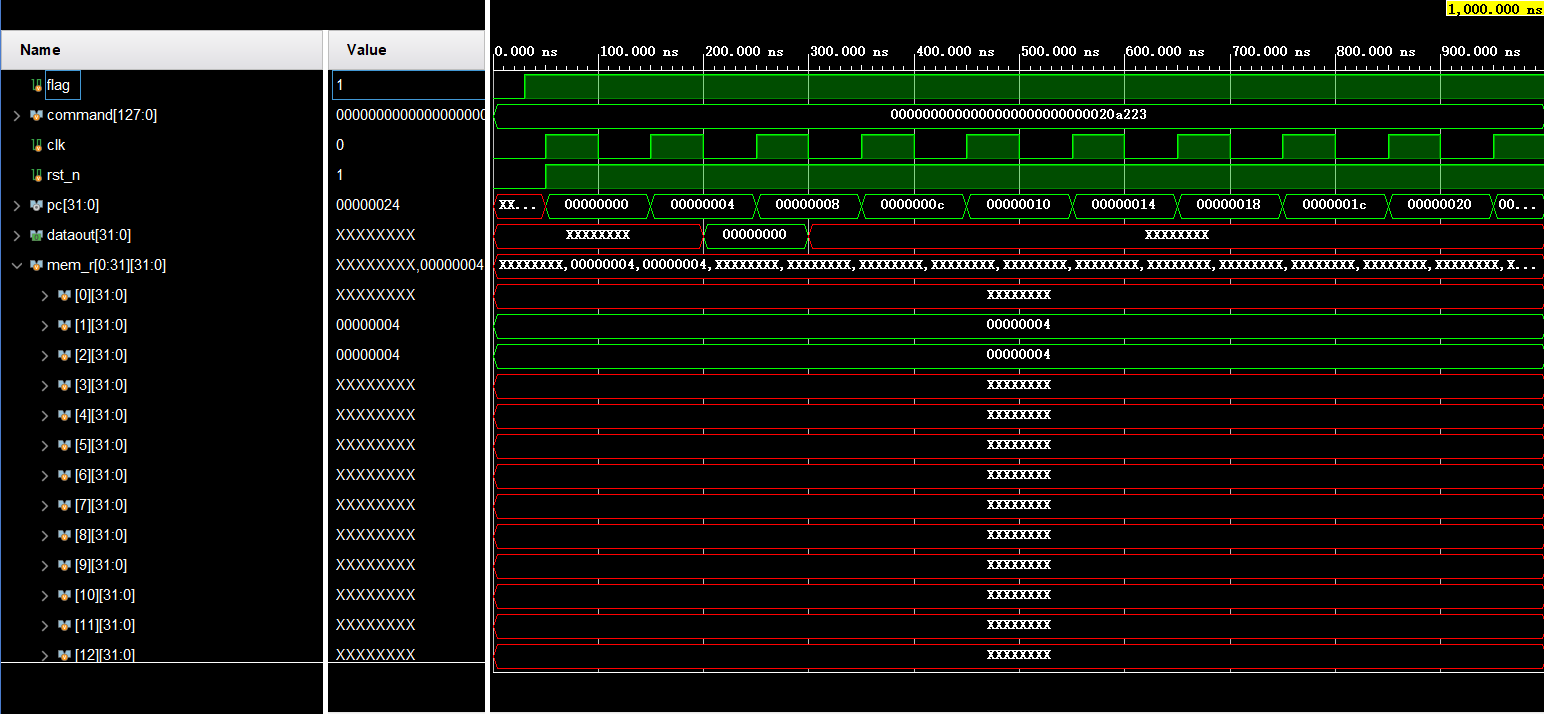
### 4.7.1 S- sw指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1010\_0010\_0010\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化地址为 2 处的数据为 8 ，初始化 imm 为 4 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将寄存器中地址为 2 处的数据 4 保存到存储器地址为 2 处。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将mem\_r[2]处的数据 4 保存到存储器地址为 2 处。

### 4.7.2 S- sb指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1000\_0010\_0010\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化地址为 2 处的数据为 8 ，初始化 imm 为 4 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将寄存器中地址为 2 处的数据 000000f0 取低八位小端存放至数据存储器地址为 2 处，且应带符号扩展为 fffffff0 。

**验证结果：**



**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，初始化 imm 为 4 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将mem\_r[2]处的数据 000000f0 取低八位小端存放至数据存储器地址为 2 处，且应带符号扩展为 fffffff0 。

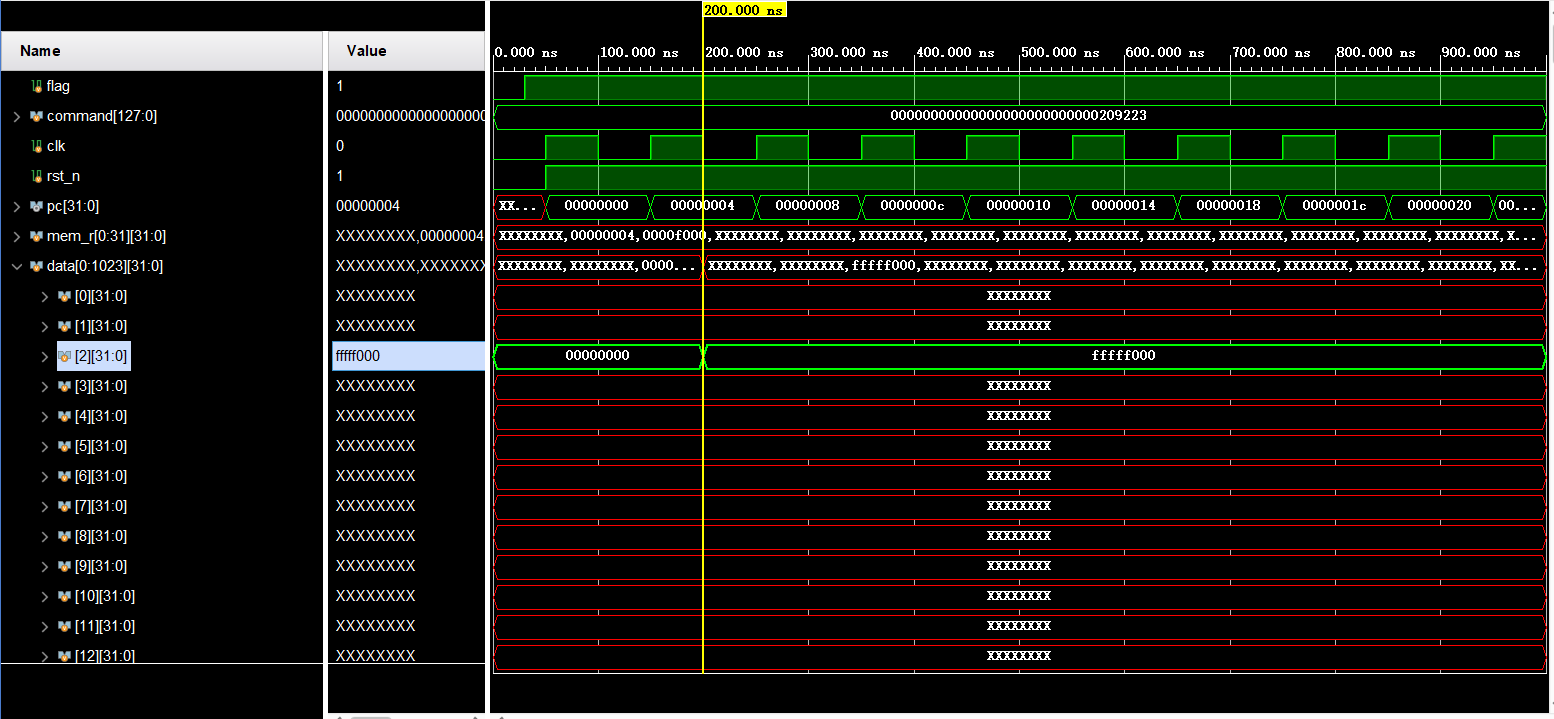
### 4.7.3 S- sh指令

**仿真代码与注释：**（涉及到的关键的）

command={{32'b0000\_0000\_0010\_0000\_1001\_0010\_0010\_0011}};

初始化寄存器地址为 1 处的数据为 4 并初始化地址为 2 处的数据为 8 ，初始化 imm 为 4 ，按照 RTL 逻辑应当寄存器地址为 1 处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将寄存器中地址为 2 处的数据 0000f000 取低十六位小端存放至数据存储器地址为 2 处，且应带符号扩展为 fffff000 。

**验证结果：**

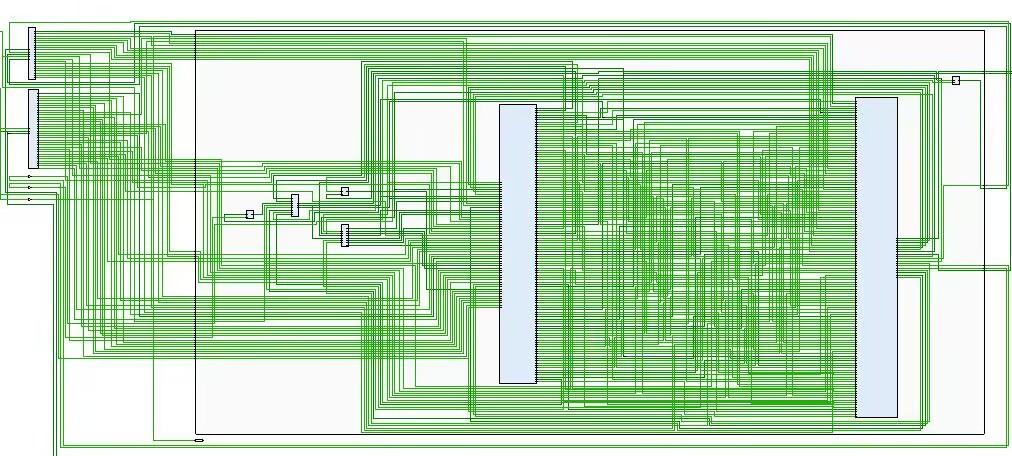


**结果分析：**

这里是mem\_r[]寄存器中，初始化mem\_r[1]处的数据为0000 0004，并初始化数据存储器中mem\_r[2]的数据为 00000008 ，初始化 imm 为 4 ，按照 RTL 逻辑应当mem\_r[1]处的数据跟 imm 的数据相加后获取到存储器地址的偏移量，并进行除 4 操作转化为存储器地址，并将mem\_r[2]处的数据 0000f000 取低十六位小端存放至数据存储器地址为 2 处，且应带符号扩展为 fffff000 。

## 电路仿真

可放大查看：



图：电路仿真图

由于我们组的板子没有带回来，故没有上板验证。

## 总结与讨论

由于时间的紧张和经验的不足，这次我们组进行的仅仅是单周期CPU设计。

从指令的执行来看，单周期 CPU 每条指令在一个时钟周期内完成，一条指令执行完再执行下一条指令。在一个时钟周期内完成上述五个阶段的操作，每个时钟下降沿更新地址。因此，要依照最长延迟的指令（lw）的时间来确定时钟周期的时间长度，无论指令的类型和它实际需要的执行时间如何，每条指令都要执行一个时钟周期。

从执行效率来看，单周期 CPU 时钟周期远远大于许多指令实际所需执行时间，R 型指令和 I 型指令都不需要访问数据存储器，sw 指令不需要写回寄存器，blez 指令不需要访问内存和写寄存器，j 指令不需要 ALU 运算，不需要读内存，也不需要写寄存器。受时钟周期宽度的影响，单周期 CPU 的效率低下，性能较差。

从信号竞争来看，首先实际的寄存器堆和存储器在单周期通路里不可能可靠工作，因为不能保证地址和数据能在“写使能”信号有效前稳定，也就是说，地址、数据和“写使能”之间存在竞争问题。

从控制单元来看，单周期 CPU 每条指令的控制信号由控制单元产生，在指令执行期间是不变的。

## 心得与体会

### 7.1 陆晓晓心得体会

刘学老师的《计算机组成原理》（基于RISC-V架构）这门课程给我们初步构建了计算机组成理论、技术的研究视图，通过这门课程的学习，我深入了解了计算机系统、计算机内部数据的运算逻辑、CPU设计等等，尤其是了解了新兴的开放指令集架构RISC-V，深入了解了整个计算机系统中最基础的部分，为之后的深入学习打下了很好的基础。在日后的学习与科研中，我会牢记这些知识，并用来解决实际问题，力求学以致用。

本次计算机组成原理课程设计难度不小，相对于其他课程而言，《计算机组成原理》这门课也几乎是本学期最难的课程，课程难度大、内容多，时间紧、任务重，课程内容与时俱进，主要围绕新兴的RISC-V指令集，它追求简约，丢弃历史包袱，具有模块化和可扩展化的特点，并且开放免费，完全是当下的主流热点。而《计算机组成原理课程设计》的要求就是实现RISC-V指令集，由于我们组同学基础都比较薄弱，所以我们初步计划是先按照书上，实现九条指令，之后在进行进一步设计、讨论扩展指令集，最终完成了单周期38条指令。完成指令后，我们对于每条指令都进行了验证，（具体您可以查阅本报告第四小节）。验收时也获得了老师的一定认可，并且完美回答了老师所有的提问。

我在本次课程设计中负责的任务是设计控制器、编写控制器部分的代码以及部分指令代码和部分文书工作，这次课程设计让我回顾了上学期学习的verilog，主要难度还是在设计部分，当然代码方面难度其实也不小。从初步完成9条指令到一步步扩展完成了38条指令，小组的每个成员都花费了不少时间和精力。本次课设虽然没有完成流水线CPU的实现，但在之后我们也会利用课余时间，讨论流水线CPU的设计，以及冒险、冲突等等，做到有始有终，尽量掌握好计组这门课程的所有知识，提升自己的专业能力。

### 7.2 郎敏杰心得体会

在我国信息产业流域，一直都存在着“缺芯少魂”的大难题，华为和中兴也遭遇了美国芯片“制裁”，芯片技术和芯片产业具有极其重大的价值，在数字领域中最复杂的CPU芯片目前有两种主流架构，X86架构和ARM架构，我们希望有一种新的CPU架构出现，能够适合中国市场大多数中小企业和团队，而RISC-V指令集就是答案。它支持高效实现微架构、大量定制和加速功能，与现有软件及编程语言良好适配。而计算机组成原理课程设计也要求我们“造”一个自己的RISC-V架构的CPU，从设计到完成，我们也查阅了不少资料，但主要是依托于《计算机组成原理（基于RISC-V架构）》这本书。

本次计组课设，我主要负责完成的任务是整个CPU总的设计，总的代码编写以及调试和部分报告工作，由于转专业，我没有参与到verilog这门课的学习中，所以我只能从头学起，一步步完成本次课设的内容，经常是课设需要我会写什么我就会去学对应的verilog的代码编写，比如要写个多路选择，我就要百度一下多路选择怎么写。在课程设计的整个过程中，刘学老师也给了我们组非常多的指导，按照书上的内容梳理了整个CPU设计的逻辑，并且利用了一节课的时间回顾了一下verilog语言，使我快速入门课程设计并且参与到课程设计中来。在课程设计验收中，我作为主讲人，也清晰的阐述了我们组在本次课程设计中的所有工作。

整个课设的过程中也遇到了许多问题，一开始并没有熟练的掌握调试技巧，往往是写了很多部件才来测试，结果功能不对，再去查错的时候非常煎熬，因为部件数目太多一时间并不易发现出错的地方在哪，后来经过与组员同学们的学习，知道了在写每个部件的时候就将其功能放在testbech中进行测试，这样保证了基本单元的功能工作都是正常的，最后只需要检查顶层模块逻辑连接是否正确即可。还有即是硬件编程与普通的程序设计并不是一样的，很多时候逻辑正确了，但是编写的代码会出现无法仿真的情况，因为所写的代码并没有办法仿真成某个部件，所以以后要注意等等问题。

总的来说，这次课设对自己的硬件设计能力有了极大的提升，但是由于基础和时间问题，没能实现流水线CPU，在以后的学习过程中，相信自己也会去实现更难，更好的东西！

### 7.3 阮雨荷心得体会

踉踉跄跄地忙碌了两个月，这次计算机组成原理课程设计是课上理论的“实战”，在学完了教材第五章的内容之后，开始了本学期的计组课设，前两周小组成员一起讨论，进行大致分工，每个人去学习自己部分对应的内容，之后再交流，前两周，我们的目标是先按照书上的内容实现九条指令，能够通过测试。之后再进行位扩展，逐步实现更多的指令。第三周，我们完善了38条指令的RTL描述、控制器设计和ALU设计，第四周，开始进行29条指令的实现。由于这期间计组课设延期了不少时间，所以我们战线也拖的比较长，中间经历了期末考，期末考之后我们小组才进行了整个单周期处理器的仿真，并且逐条指令进行测试。整体上完成了老师的基本要求。

在本次课程设计中我负责的任务是设计控制器、编写控制器部分的代码以及部分指令代码和部分文书工作，整体上这次课程设计需要我们完成的任务难度不小，尤其是在整体的设计上，代码编写也给我们带来了不少的困扰，小组成员的verilog基础都相对薄弱，不过中间由于计组课设验收推迟了不少时间，所以给了我们充分的时间去学习相关的知识，再完成这次课程设计的全部内容。课程设计报告的完成对于这门课来说是个句号，但对于我的大学生活来说仅仅是一个逗号，随着报告的提交，我的大三上学期也完全结束了，一时间，感慨万千。在此，我想感谢一下指导过我们小组课程设计的刘学老师以及助教学长，助教学长撰写的本次课设的指导书（放在github仓库的那本）在我复习verilog以及入门RISC-V指令集的过程中，让我少走了不少弯路；刘学老师认真严谨，在课程设计过程中也讲述了好几遍这次课程设计任务的完成的主体思路。如果没有他们的帮助，或许我们的课程设计没有那么顺利地完成。