《数字逻辑与计算机组成》实验报告五. 存储器及数据通路设计

1 实验

1.1 只读存储器实验

1.1.1 实验内容

利用 ASCII 码可显示字符点阵字库,在 Logisim 的 LED 点阵组件上显示 ASCII 码字符形状。

1.1.2 实验整体方案设计

ASCII 码可见字符点阵字库 ascii8-16.zk 的文件内容如下:1. v2.0 raw 2. 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3. 00 00 18 3C 3C 3C 18 18 18 00 18 18 00 00 00 00 4. 00 63 63 63 22 00 00 00 00 00 00 00 00 00 00 00 5. 00 00 00 36 36 7F 36 36 36 7F 36 36 00 00 00 00 6. …首行是 Logisim 数据文件的标识,不存储到存储器中;第二行表示字符空格的字形点阵,第三行表示字符“!”的字形点阵,第四行表示字符“"”的字形点阵以此类推;载入存储器中时从第二行开始。每个字符的字形点阵都是 16 个字节,表示 16 行 8 列矩阵,如果某 1 位为 1,则表示对应矩阵点是可显示的。ASCII 码可见字符共有 96 个,每个字符用 16 个字节的数据来表示,点阵字库共有 1536 个字节。实验设计的关键一是在于如何根据输入的 ASCII 码数值,查找到字符点阵字库中的初始位置。在点阵字库中 ASCII 码可见字符点阵从地址 0 开始存储,因此在根据 ASCII 码来读取点阵字库时,需要先把 ASCII 码值减去十六进制 0x20,然后根据每个字符的点阵数据是 16 个字节,把得到的差值左移 4 位在低位补 4 个 0,则得到该字符点阵在字库中的起始位置,字符点阵数据为连续存放的 16 个字节。实验设计的关键二是如何同步输出某个字符 16 个字节的点阵数据。首先需要将点阵字库文件加载的只读存储器,由于提供的点阵字库文件是以字节为单位,为了能够同时输出 16 个字节的点阵数据,则需要复制 16 个加载相同点阵字库的只读存储器,读取点阵数据时每个只读存储器的地址数据按点阵行的次序加 1 递增,则可以同时读取第 1 行、第 2 行、…、第 16 行的点阵数据,然后再同步输出到 LED 点阵组件的输入端中。根据点阵字库中的数据宽度为 8 位,则设置只读存储器的数据位宽度为 8 位;点阵字库文件大小为 1536 个字节,则只读存储器的地址位宽度至少需设置为 11 位。如果输入的 ASCII 码不在可见字符范围内,则输出错误标志位 CodeErr。实验要求根据输入的 ASCII 码值,在 LED 点阵中显示该 ASCII 码的字形;如果 ASCII 值不在可显示字符范围内,则输出 ASCII 码错误标志位。
本实验不需要顶层模块设计图。

row,led 灯	将存入 rom 的数据显示出来
code	输入的 ASCII 编码
codeerror	输入溢出或者太小(小于 20)
全加器,扩展器,比较器	实现功能的工具

表 1: 功能表

1.1.3 实验原理图和电路图

本实验不需要原理图，只需要思路。

思路就是，用比较器比较 0x020 和 0x600，保证输入的数据合法，否则输出 codeerror；减法器减去 0x020，16 个 ram 分别加常量 1,2,3……来实现地址的递增，剩下的工作就是连线和存入镜像了。

电路图实现：

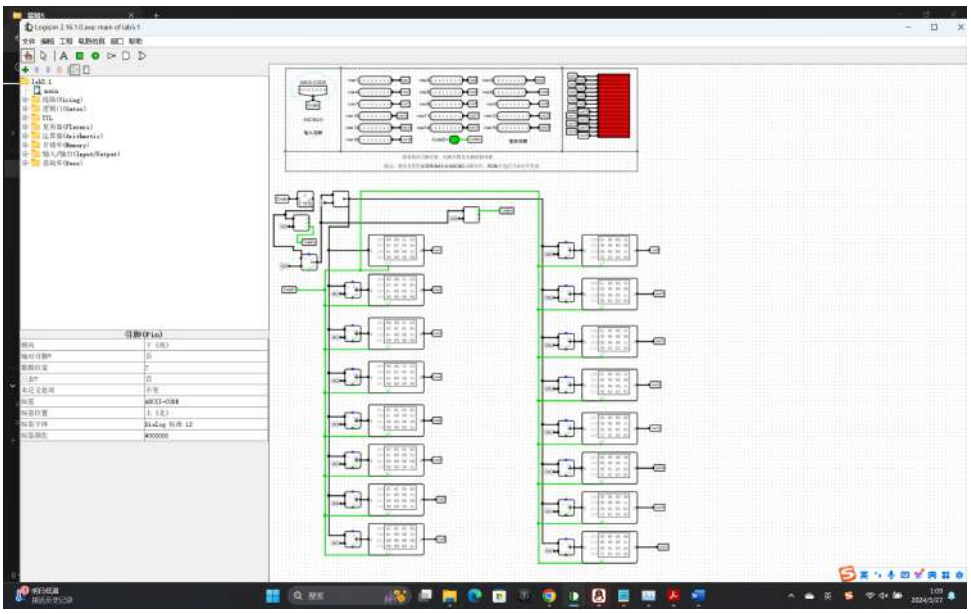


图 1: 电路图 1

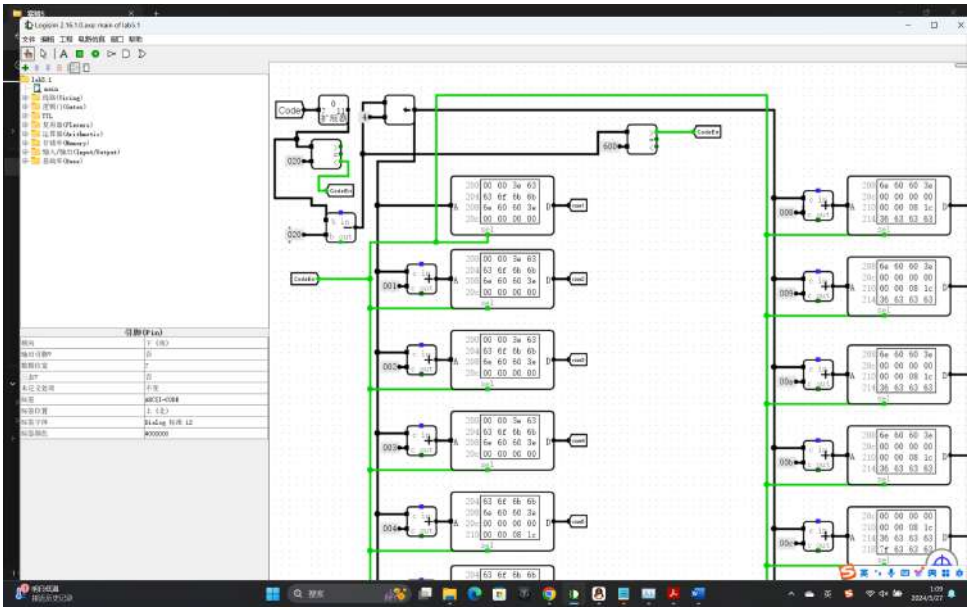


图 2: 电路图 2

1.1.4 实验数据仿真测试图

展示了初始状态和载入两个 ASCII 码的 LED。

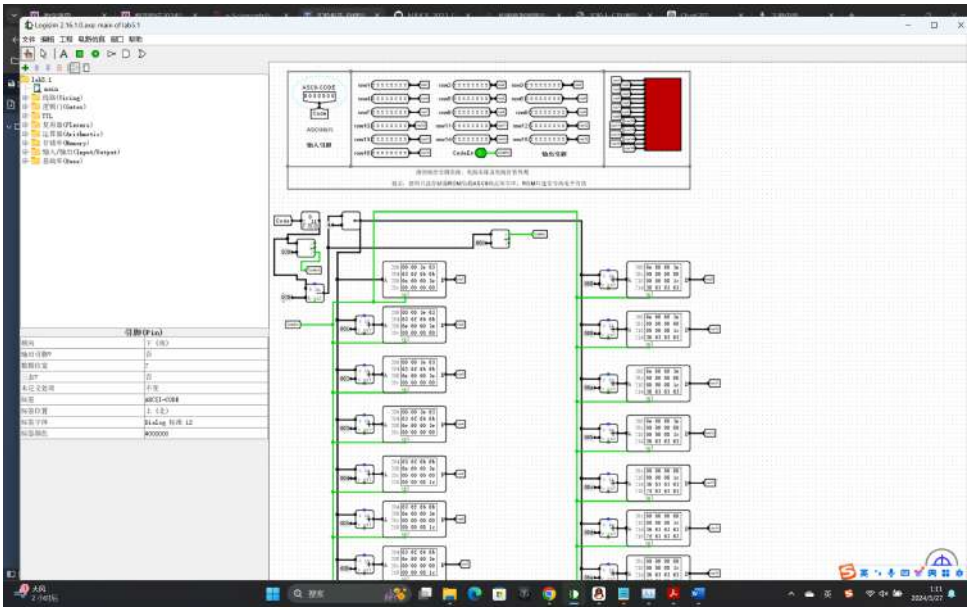


图 3: 模拟 1

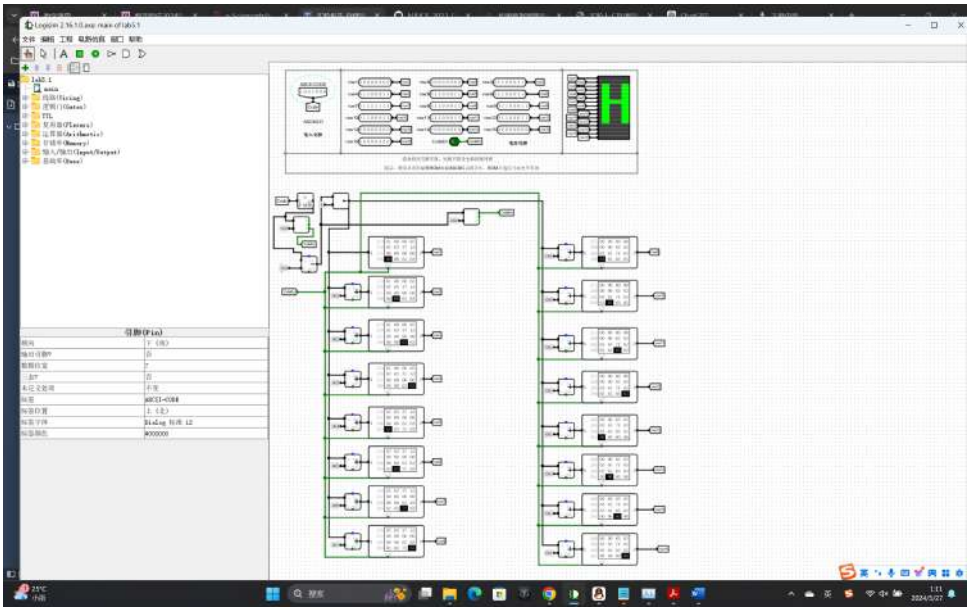


图 4: 模拟 2

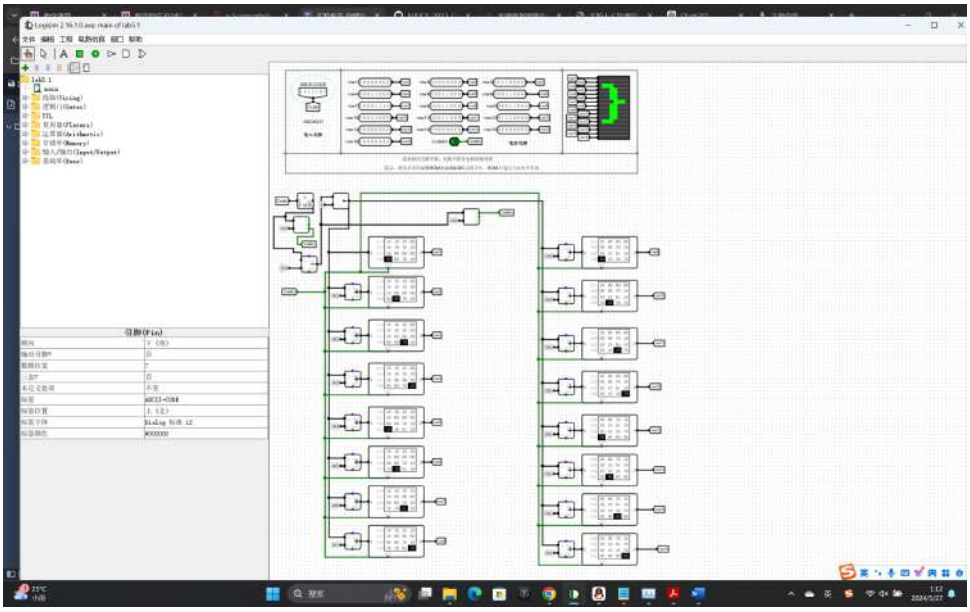


图 5: 模拟 3

功能表在前面。

1.1.5 错误现象及分析

第一次测试时，只有少数的样例的 codeerror 没过，考虑了一下原因，是没有只对小于 0x20（大于 0x0）的数用比较器，对大于 0x600 的数也没有检测，后来加上了两个比较器后通过了实验。不是大问题，只是忘记边界比较了。

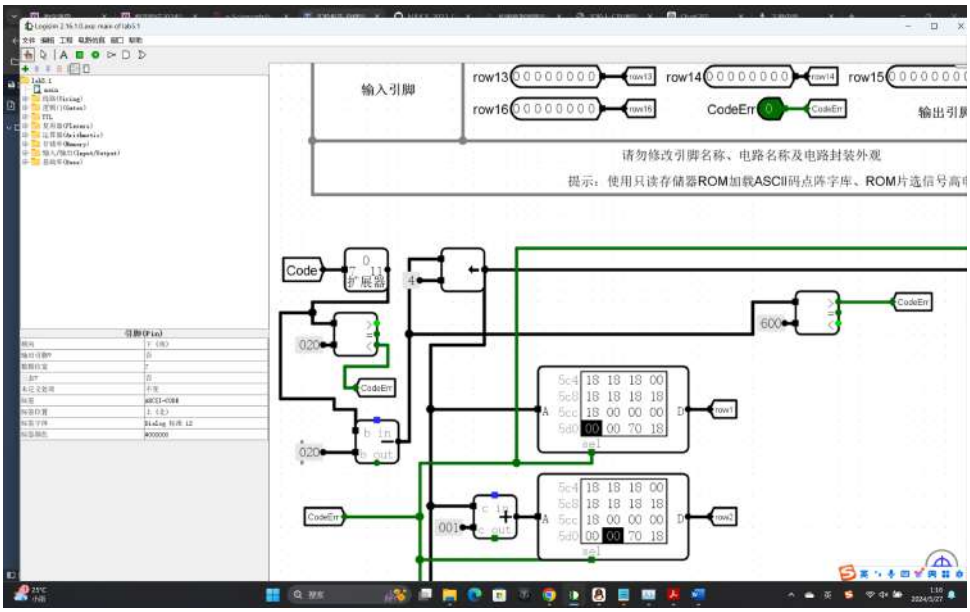


图 6: 后加的比较器

1.2 数据存储器实验

1.2.1 实验内容

数据存储器实验。实验要求设计一个 256KB 的数据存储器，可按照字节进行存取操作，数据字长为 32 位，则地址位为 18 位。提示：为了方便按照字节进行存取操作设计，可使用 4 片数据位宽为 8 位 RAM，分别存储 32 位数据的不同字节段数据。

1.2.2 实验整体方案设计

本实验的顶层模块设计图将通过原理图给出，这里先提出一个我实现的思路。（讲解在实验五 pdf 里，就不附了，反正是无效字数）

我们知道，ROM 的控制分为 SEL（片选）和 STR（存储），对于 SEL，我们要按照对应的真值表的最小项进行连线，生成 SEL0 SEL3(一个 ROM 能存 8 位，因此需要四个不同的 SEL 来控制不同的可能性（是否存储））（如图），这里顺便把 ALignErr 也连上了。

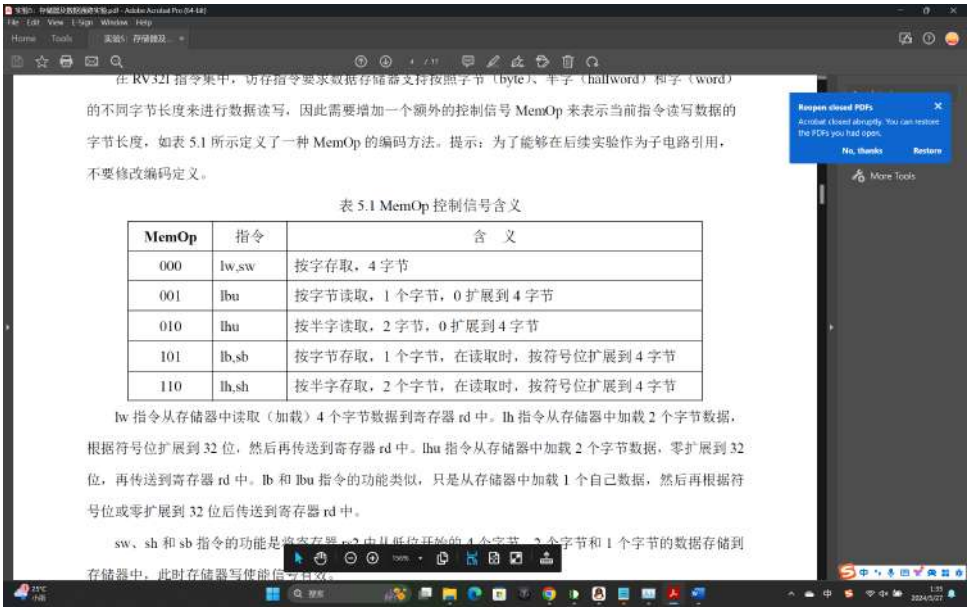
端。可根据 MemOp 控制信号和存储器地址 Addr 最低 2 位来确定片选信号的数值，当存取多字节数据的有效地址不是自然对齐时，显示错误信号有效 AlignErr=1。出现未定义的 MemOp 状态时，错误信号也有效 AlignErr=1，如表 5.2 所示。

表 5.2 存储器控制信号、地址低 2 位和片选信号、地址对齐的对应关系

MemOp[2][1][0]	Addr[1][0]	SEL3	SEL2	SEL1	SEL0	AlignErr
000	00	1	1	1	1	0
*00	01	0	0	0	0	1
*00	10	0	0	0	0	1
*00	11	0	0	0	0	1
*01	00	0	0	0	1	0
*01	01	0	0	1	0	0
*01	10	0	1	0	0	0
*01	11	1	0	0	0	0
*10	00	0	0	1	1	0
*10	01	0	0	0	0	1
*10	10	1	1	0	0	0
*10	11	0	0	0	0	1
其它	**	0	0	0	0	1

根据片选信号确定每一片 RAM 在存储时的写入字节，例如当片选信号 1111 全部有效时，存储高字节的 RAM3 写入输入数据 Din 的

图 7: SEL，ALignErr 的真值表



在 RV32I 指令集中，访存指令要求数据存储器支持按照字节（byte）、半字（halfword）和字（word）的不同字节长度来进行数据读写，因此需要增加一个额外的控制信号 MemOp 来表示当前指令读写数据的字节长度，如表 5.1 所示定义了一种 MemOp 的编码方法。提示：为了能够在后续实验作为子电路引用，不要修改编码定义。

表 5.1 MemOp 控制信号含义

MemOp	指令	含 义
000	lw,sw	按字存取，4 字节
001	lbu	按字节读取，1 个字节，0 扩展到 4 字节
010	lhu	按半字读取，2 字节，0 扩展到 4 字节
101	lb,sb	按字节存取，1 个字节，在读取时，按符号位扩展到 4 字节
110	lh,sh	按半字存取，2 个字节，在读取时，按符号位扩展到 4 字节

lw 指令从存储器中读取（加载）4 个字节数据到寄存器 rd 中。lh 指令从存储器中加载 2 个字节数据，根据符号位扩展到 32 位，然后再传送到寄存器 rd 中。lhu 指令从存储器中加载 2 个字节数据，零扩展到 32 位，再传送到寄存器 rd 中。lb 和 lbu 指令的功能类似，只是从存储器中加载 1 个自己数据，然后再根据符号位或零扩展到 32 位后传送到寄存器 rd 中。

sw、sh 和 sb 指令的功能是将寄存器 rs2 中从低位开始的 4 个字节、2 个字节和 1 个字节的数据存储到存储器中，此时存储器写使能信号有效。

图 8: memop 含义

连完了 SEL，接下来是 STR。STR 很简单，用 WE 控制即可。接下来是 ADDR 的问题。ADDR 有 18 位，而我们需要的只是从第二位往上的位数（前两位是计算 SEL 的，不应该控制 RAM 的地址）（这个是我后来才意识到的，这个后面在错误分析会说），因此用分线器截断；
D1，D2，D3，D4 的四个输入也需要用多路选择器选择，在 SEL 的控制下，RAM 可能会得到不同的输入，也可能得不到输入。
RAM0 RAM3 也需要用 SEL 进行控制，控制传到哪个 Dout 上，最后才能得到正确的输出。
最后的总 DOUt，也需要 SEL 控制如何扩展，以得到正确的结果。

SEL(四个)	不同 RAM 的片选控制
DIN–Di	由原先输入得到的 RAM 的输入值
WE	STR
Ram0-3	RAM 输出的结果
Douti	经过分拣的真正的输出结果

表 2: 功能表

1.2.3 实验原理图和电路图

原理图：

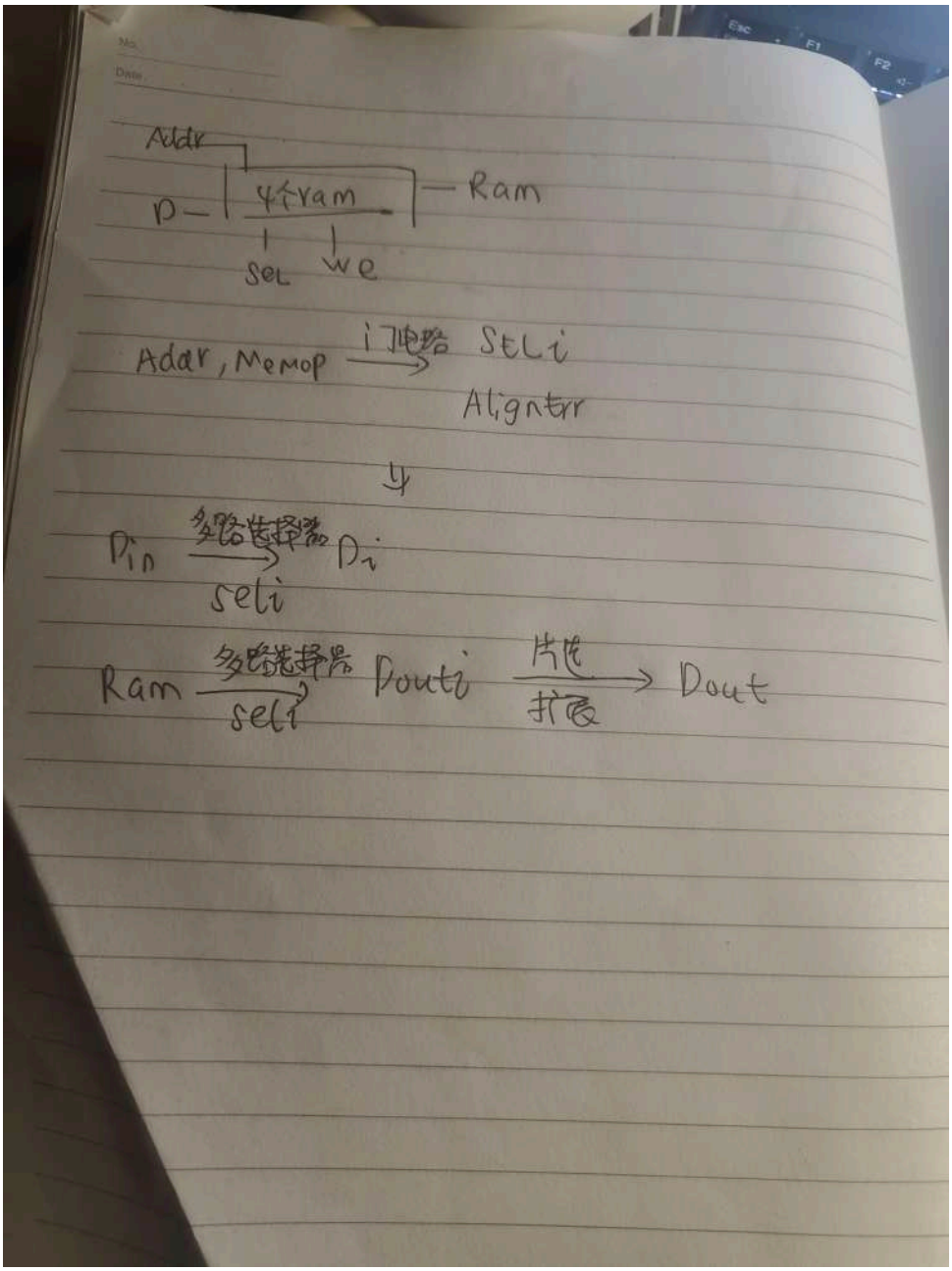

Addr
Ram
Sel
we
Adar, MemOp
SELi
AlignErr
Din
seli
Di
Ram
seli
Douti
Dout

图 9: 原理图

电路图实现：

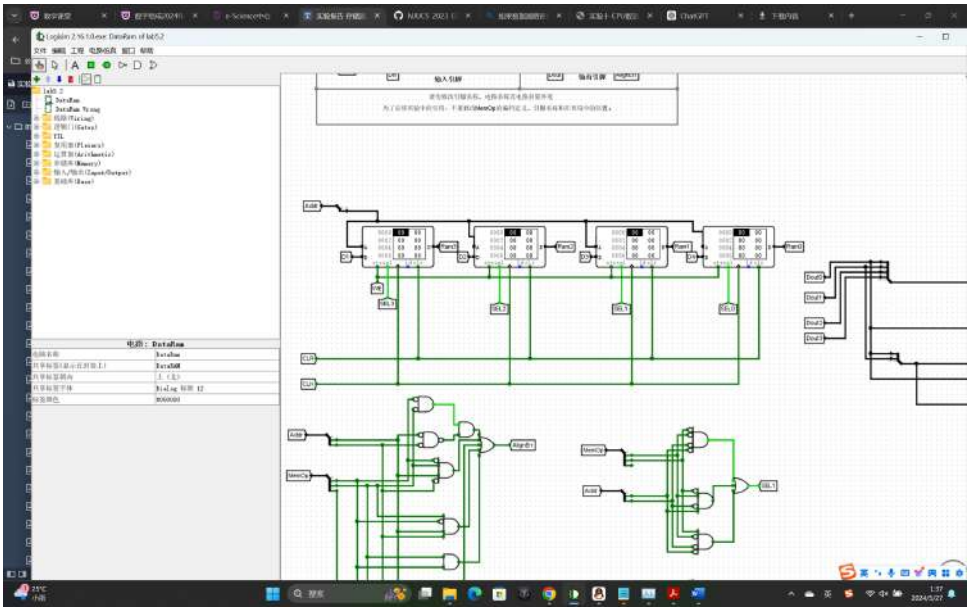


图 10: 主电路

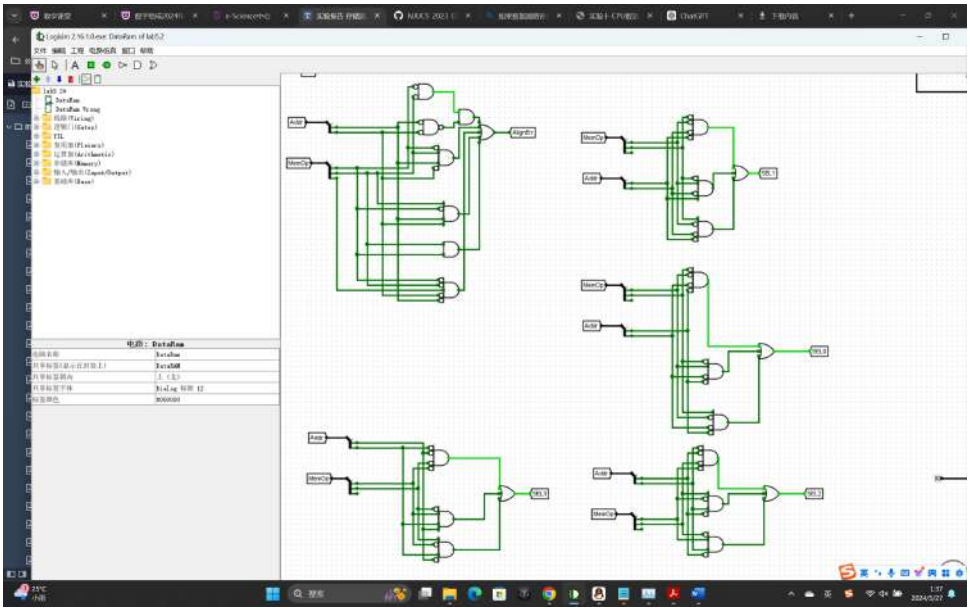


图 11: SEL，AlignErr

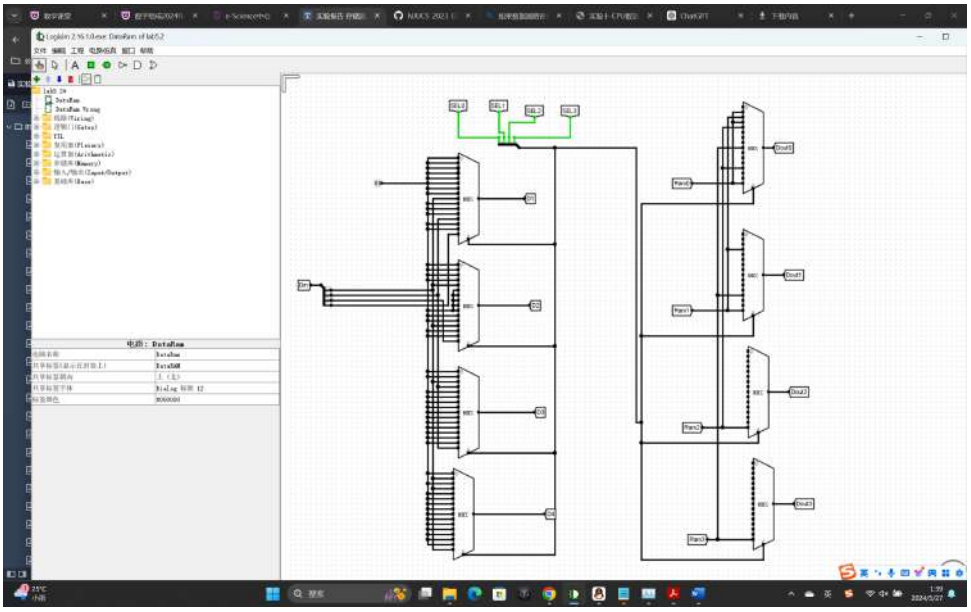


图 12: Di,Douti

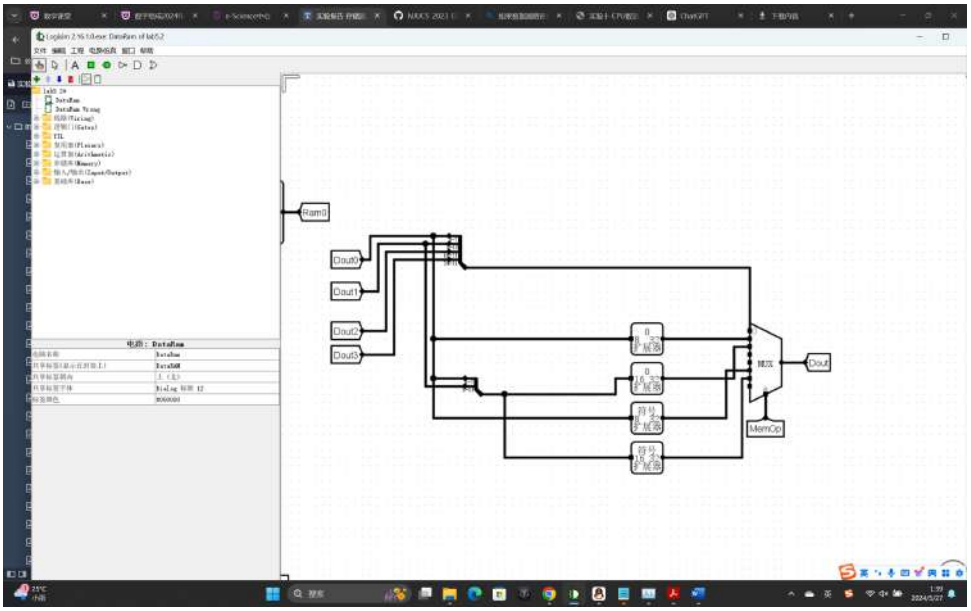


图 13: 最终 Dout

1.2.4 实验数据仿真测试图

该实验仿真测试图特别复杂,这里只展示前六个时钟周期。(和测试集思想一样)

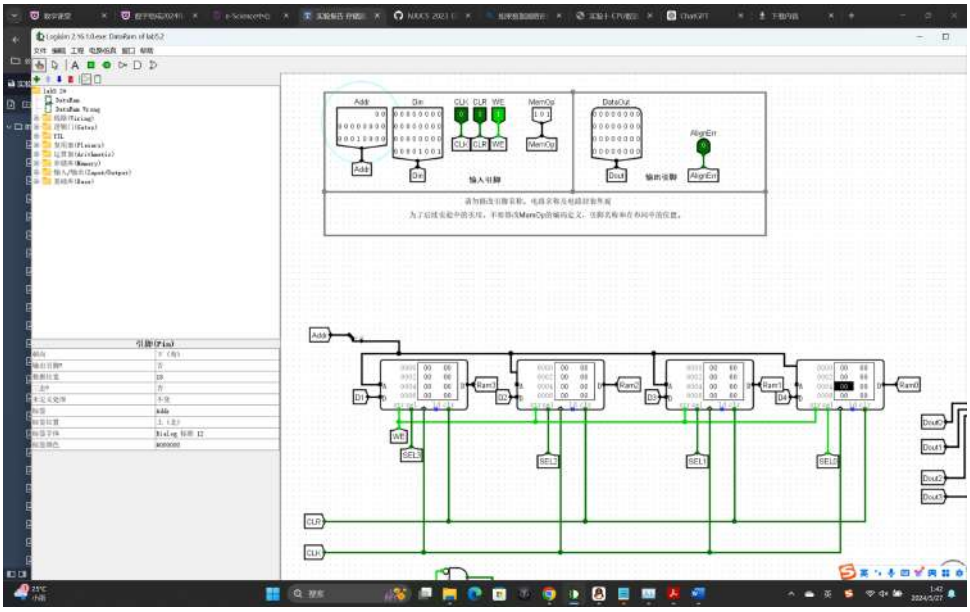


图 14: 1

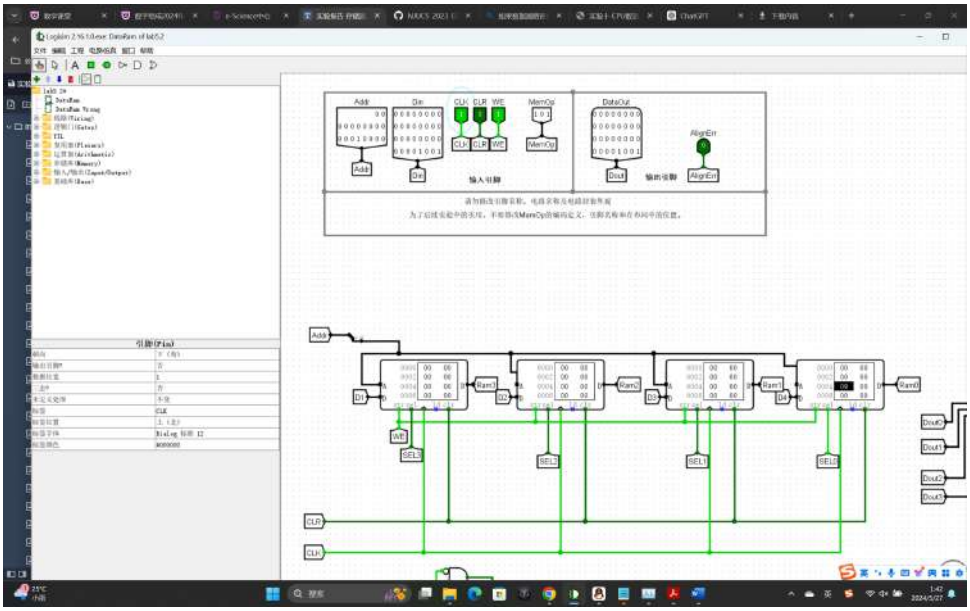


图 15: 2

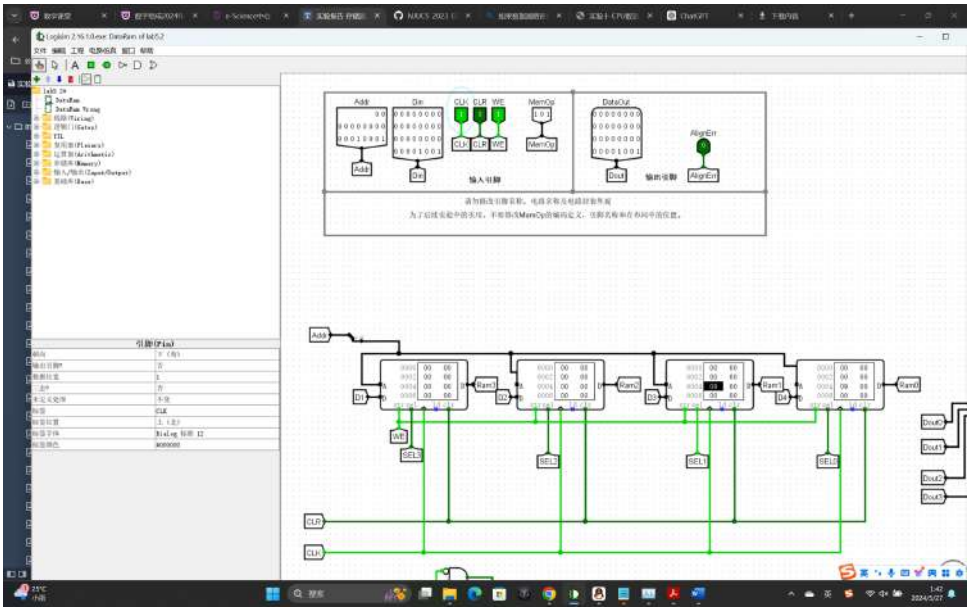


图 16: 3

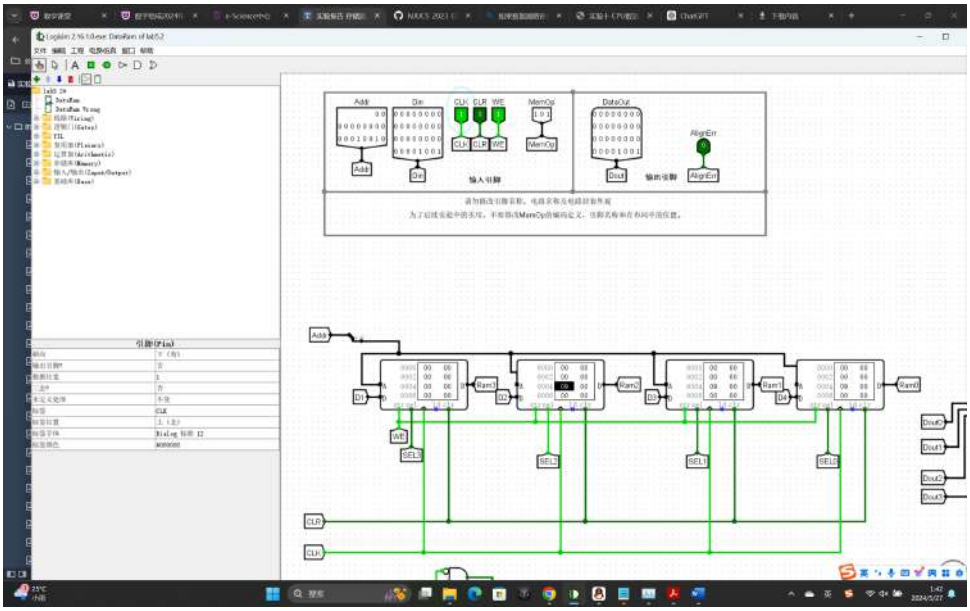


图 17: 4

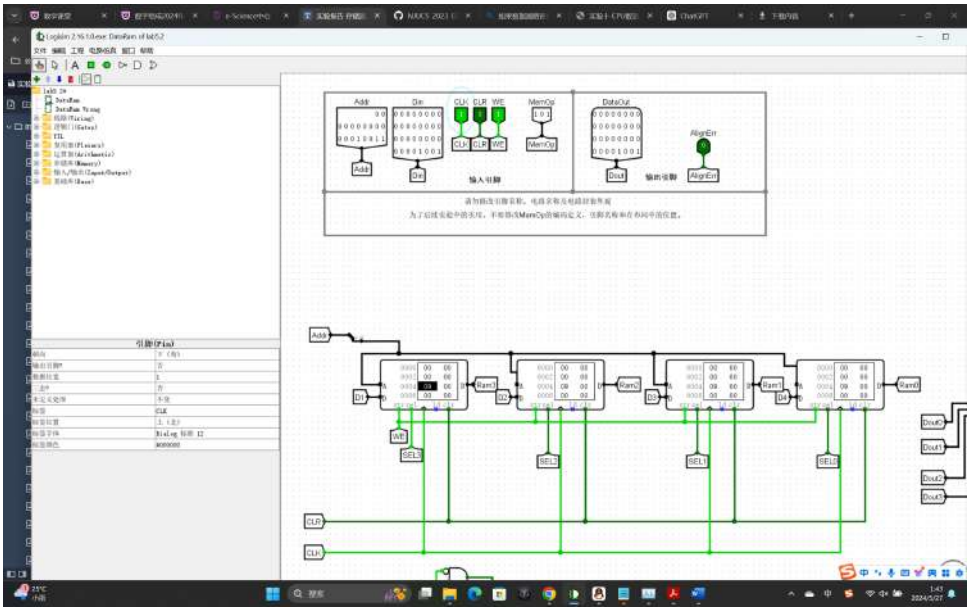


图 18: 5

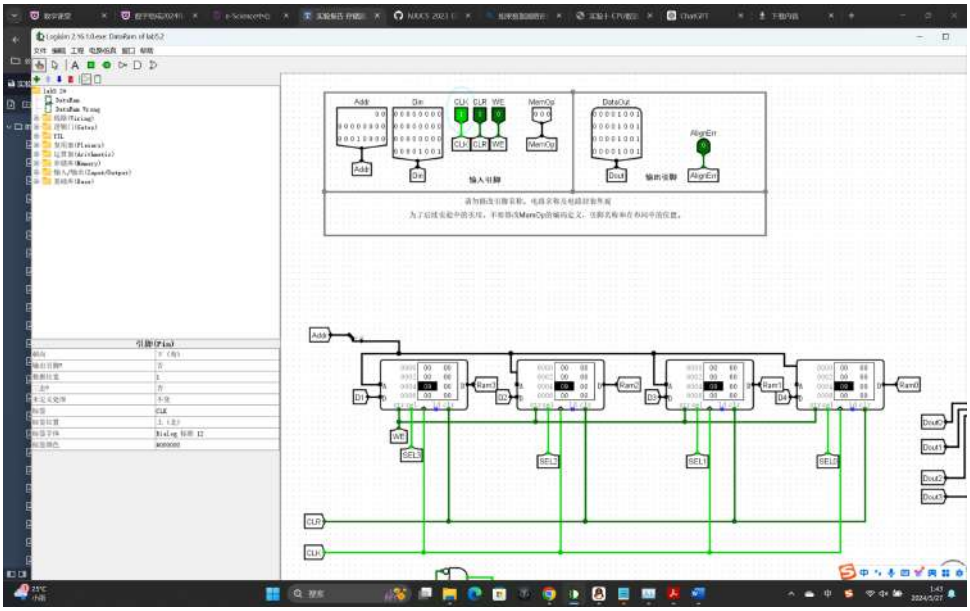


图 19: 6

1.2.5 错误现象及分析

其实最初过第二关时，电路图并非这样的。最初过不去，我以为是因为没有考虑 ADDR 的变化问题，因为当时我的 ADDR 连的是 18 位，没有排除后两位，所以地址有问题，于是采取了面向测试集编程，使用了一个寄存器，如果 SEL 为 0 就不更新 ADDR 对应的值，这样过了 5.2，但是 5.5 由于寄存器使时钟周期混乱而过不去了，于是最后改成了现在的样子。

其实是没看题仔细的问题，下面贴的是之前虽然过了但是有问题的电路的部分（其他部分没改不贴了）

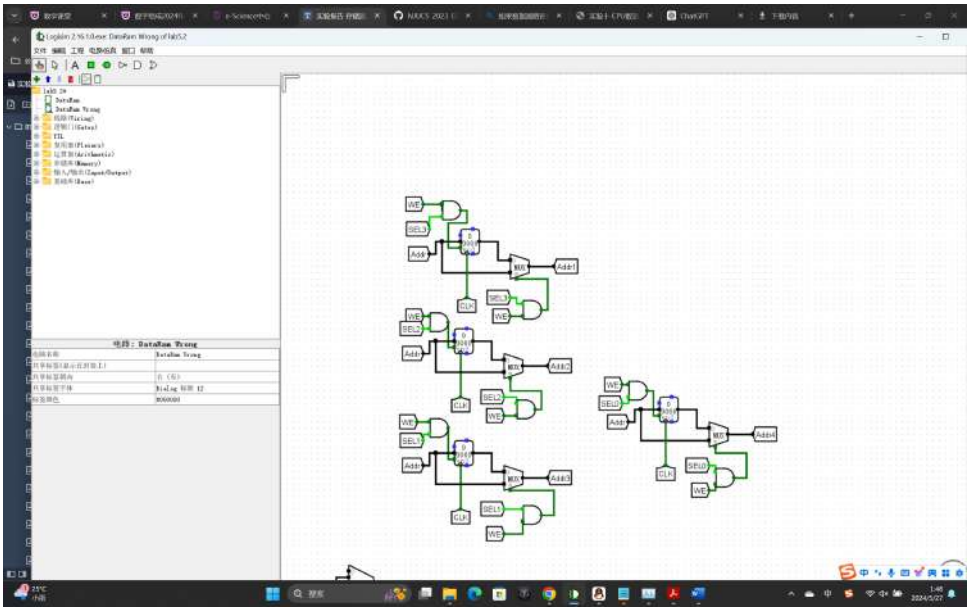


图 20: 多加了这个控制 ADDR

1.3 取指令部件实验

1.3.1 实验内容

根据初始地址 Initial Address、立即数寄存器 Imm 和 rs1 寄存器的数据 BusA，以及控制信号 Reset、NxtASrc 和 NxtBSrc 的赋值输出当前指令的地址寄存器 PC 和指令存储器的输出内容 IR。按

1.3.2 实验整体方案设计

由于 Logisim 存储器组件的存储单元是按照数据位宽为单位，当定义数据位宽为 32 位时，每个地址中包含 4 个字节内容因而 Logisim 中的一个存储单元相当于 RISC-V 程序中的 4 个字节。因而，将 RISC-V 中指令地址转换为 Logisim 中指令存储器的地址时，需要将把指令地址中的最低两位舍弃。

1、顺序执行指令，则 PC=PC+4。2、无条件跳转指令，jal 指令，PC = PC + 立即数 imm jalr 指令，PC = R[rs1] + 立即数 imm。3、分支转移指令，根据比较运算的结果和 Zero 标志位来判断，如果条件成立则 PC = PC + 立即数 imm 否则 PC=PC+4。这些指令用多路选择器和全加器实现，逻辑很清晰。reset 通过多路选择器选择是否输出 init address.

存入寄存器后 PC 更新，传回 BUSA 对应的多路选择器。

1.3.3 实验原理图和电路图

思路很简单，不需要原理图。电路图实现：

Init addr	初始地址
PC	程序计数器
NextAsrc , NextBsrc	控制信号
寄存器	存地址
Imm	立即数
Reset	控制输入初始地址
BusA	RS1 寄存器值
IR	有效地址

表 3: 功能表

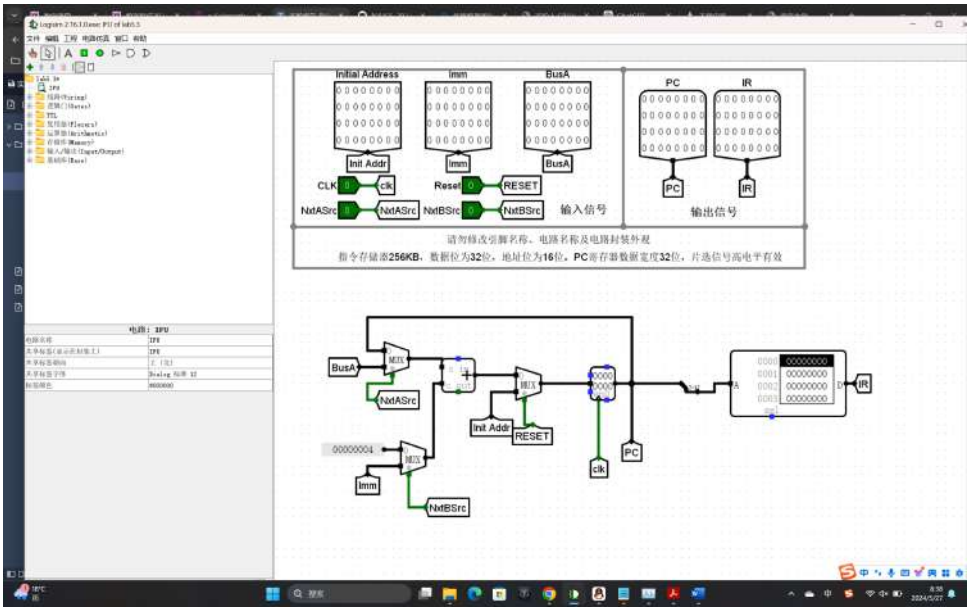
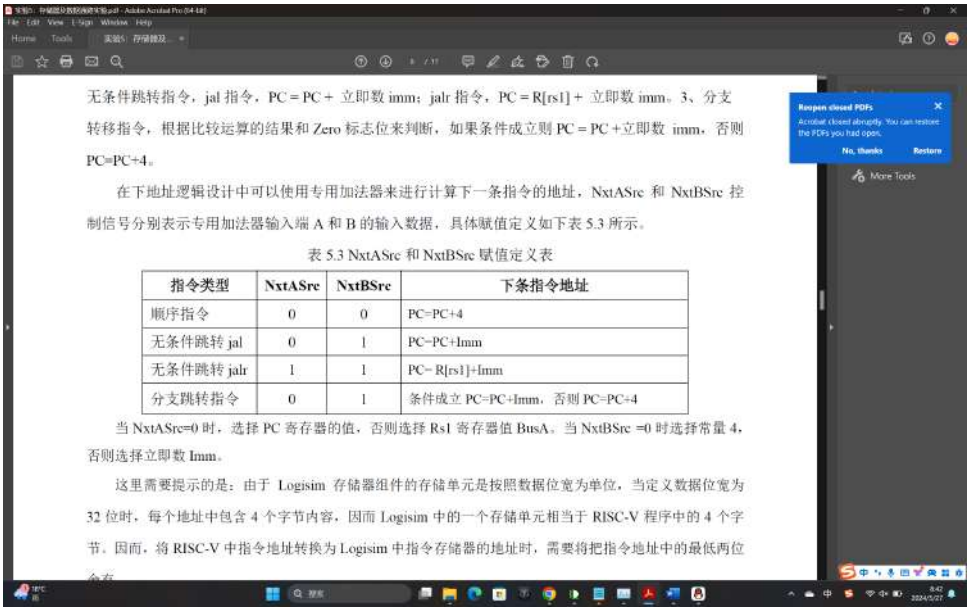


图 21: 电路图

1.3.4 实验数据仿真测试图

测试了 reset , 三种指令类型。



无条件跳转指令，jal 指令，PC = PC + 立即数 imm；jalr 指令，PC = R[rs1] + 立即数 imm。3、分支转移指令，根据比较运算的结果和 Zero 标志位来判断，如果条件成立则 PC = PC +立即数 imm，否则 PC=PC+4。

在下地址逻辑设计中可以使用专用加法器来进行计算下一条指令的地址，NxtASrc 和 NxtBSrc 控制信号分别表示专用加法器输入端 A 和 B 的输入数据，具体赋值定义如下表 5.3 所示。

表 5.3 NxtASrc 和 NxtBSrc 赋值定义表

指令类型	NxtASrc	NxtBSrc	下条指令地址
顺序指令	0	0	PC=PC+4
无条件跳转 jal	0	1	PC=PC+Imm
无条件跳转 jalr	1	1	PC= R[rs1]+Imm
分支跳转指令	0	1	条件成立 PC=PC+Imm，否则 PC=PC+4

当 NxtASrc=0 时，选择 PC 寄存器的值，否则选择 Rs1 寄存器值 BusA。当 NxtBSrc =0 时选择常量 4，否则选择立即数 Imm。

这里需要提示的是：由于 Logisim 存储器组件的存储单元是按照数据位宽为单位，当定义数据位宽为 32 位时，每个地址中包含 4 个字节内容，因而 Logisim 中的一个存储单元相当于 RISC-V 程序中的 4 个字节。因而，将 RISC-V 中指令地址转换为 Logisim 中指令存储器的地址时，需要将把指令地址中的最低两位

图 22: 四种指令类型

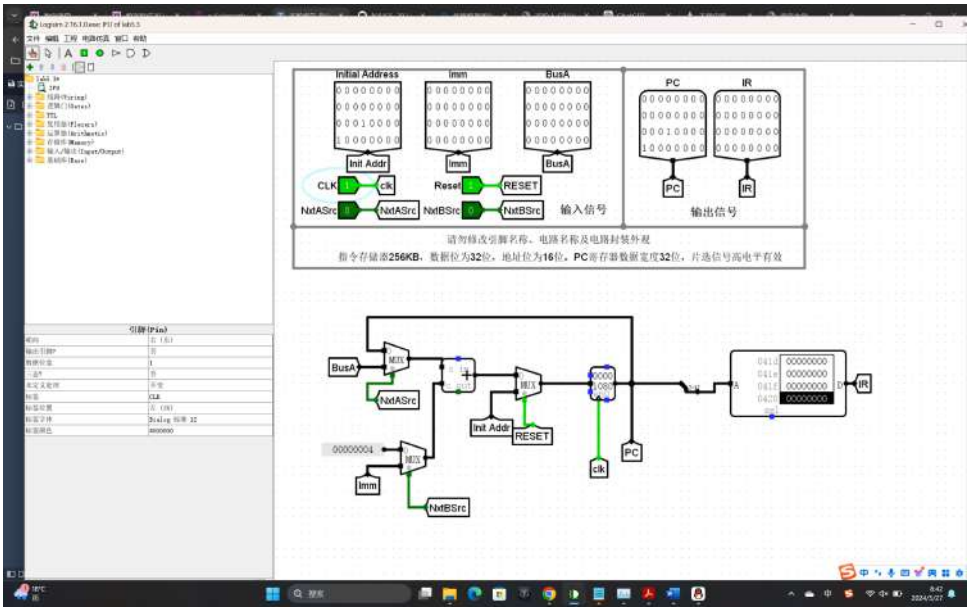


图 23: reset

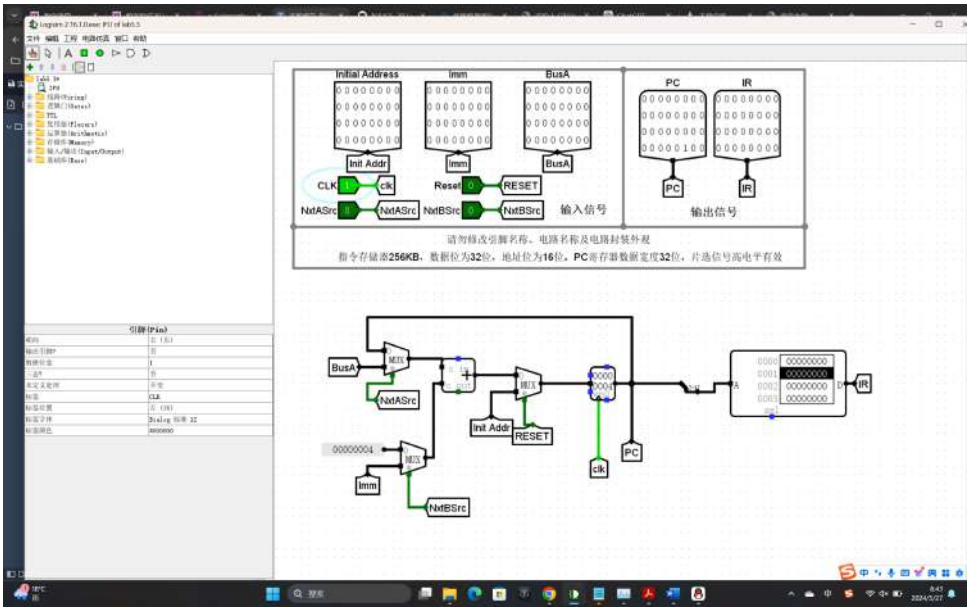


图 24: 00(初始 PC 为 0)

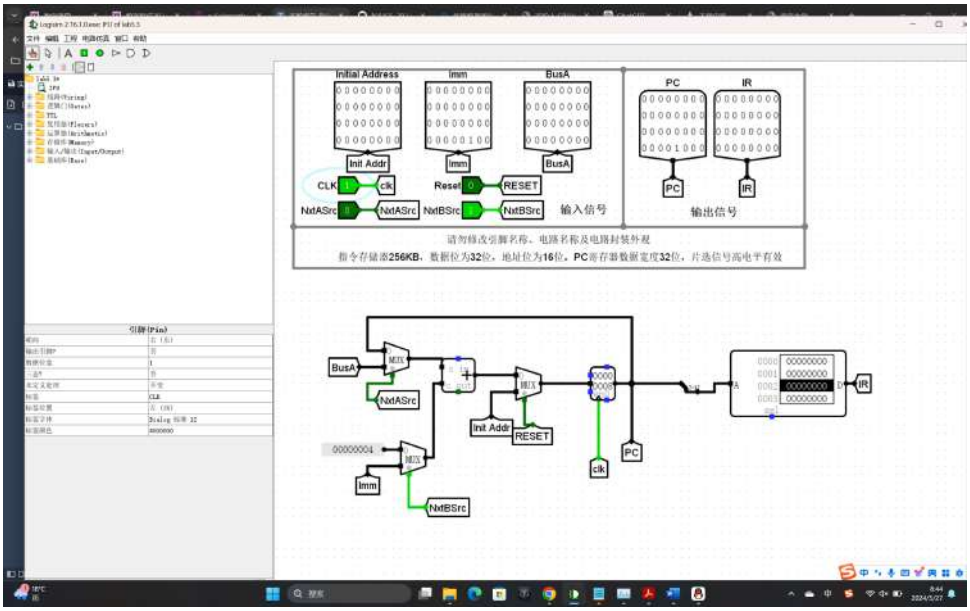


图 25: 01(继上一个图的 PC)

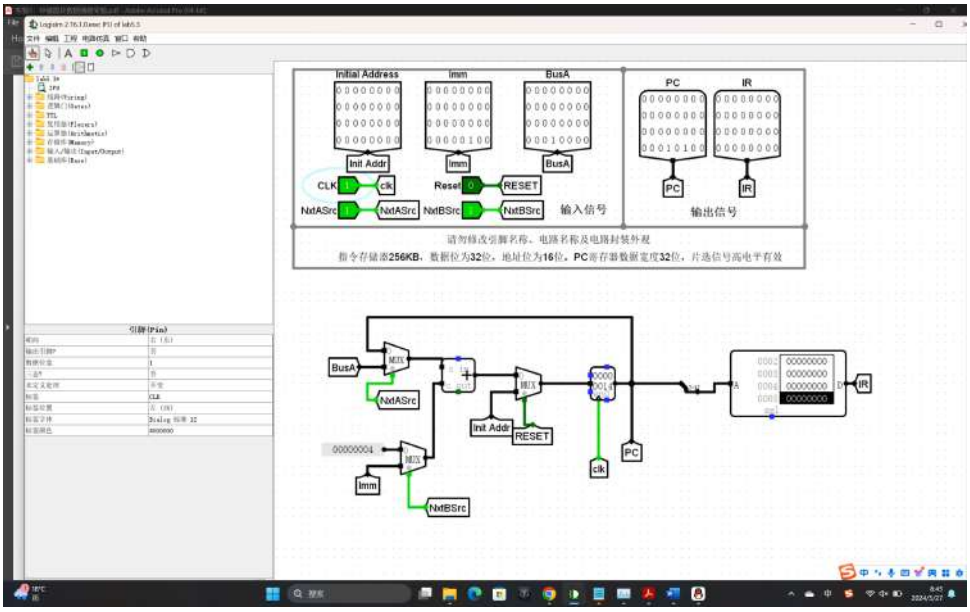


图 26: 11 跳转

功能表在上面。

1.3.5 错误现象及分析

在实验过程中,没有遇到任何错误。

1.4 取操作数部件 IDU 实验

1.4.1 实验内容

设计立即数扩展器子电路。修改寄存器堆子电路。设计“IDU”子电路。

1.4.2 实验整体方案设计

immI = 20Instr[31], Instr[31:20];
immU = Instr[31:12], 12’b0;
immS = 20Instr[31], Instr[31:25], Instr[11:7];
immB = 19Instr[31],Instr[31], Instr[7], Instr[30:25], Instr[11:8], 1’b0;
immJ = 11Instr[31], Instr[31], Instr[19:12], Instr[20], Instr[30:21], 1’b0;
根据这个指令扩展规则,可以连出立即数扩展器。
寄存器堆子电路只需要将 0 号寄存器孤立出来输入端什么也不连就行。
IDU 按照指令字段的长度分别连接寄存器,读取(或写入)完寄存器数据后,ALUAsrc 和 ALUBsrc 用来控制两个多路选择器的输出数据, ALUASsrc 控制 1 个两路选择器, 当 ALUASsrc=0 时选择

BusA 输出到 ALU 的操作数 A 口, 当 ALUASsrc=1 时输出 PC。ALUBSrc 控制 1 个四路选择器, 当 ALUBSsrc=00 时选择 BusB 输出到 ALU 的操作数 B 口当 ALUBSsrc=01 时选择输出常数 4 当 ALUBSsrc=10 时选择输出 32 位立即数 Imm。按照这个要求连线即可。
本实验不需要底层模块设计图。

Imm	立即数
Extop	扩展器控制信号
立即数扩展器	扩展到 Imm
寄存器堆	存/取操作数
ALUA/BSRC	控制最终输出数据

表 4: 功能表

1.4.3 实验原理图和电路图

原理图:寄存器堆不需要原理图。

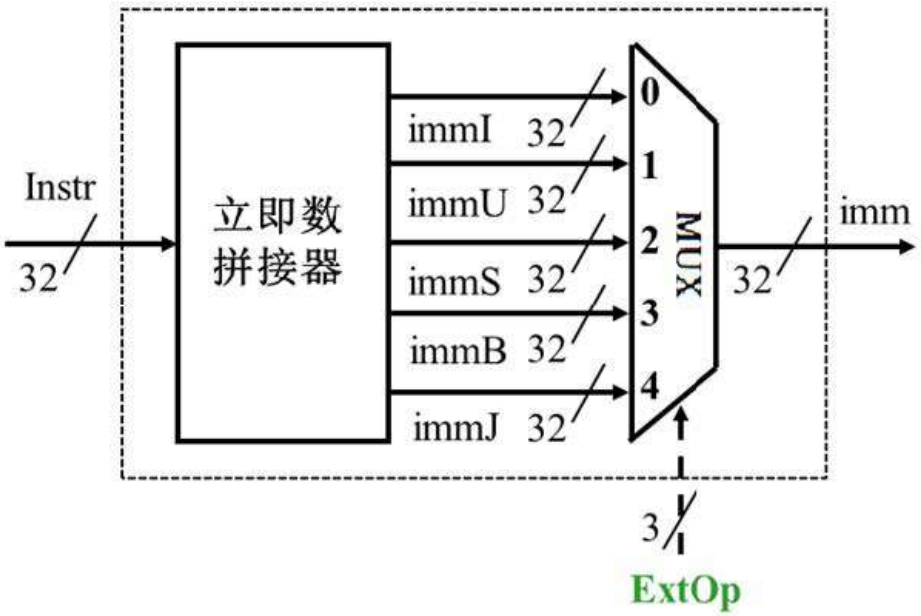


图 27: 立即数扩展器原理图

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
J	imm[20|10:1|11|19:12]					rd	opcode

图 28: 指令的编码

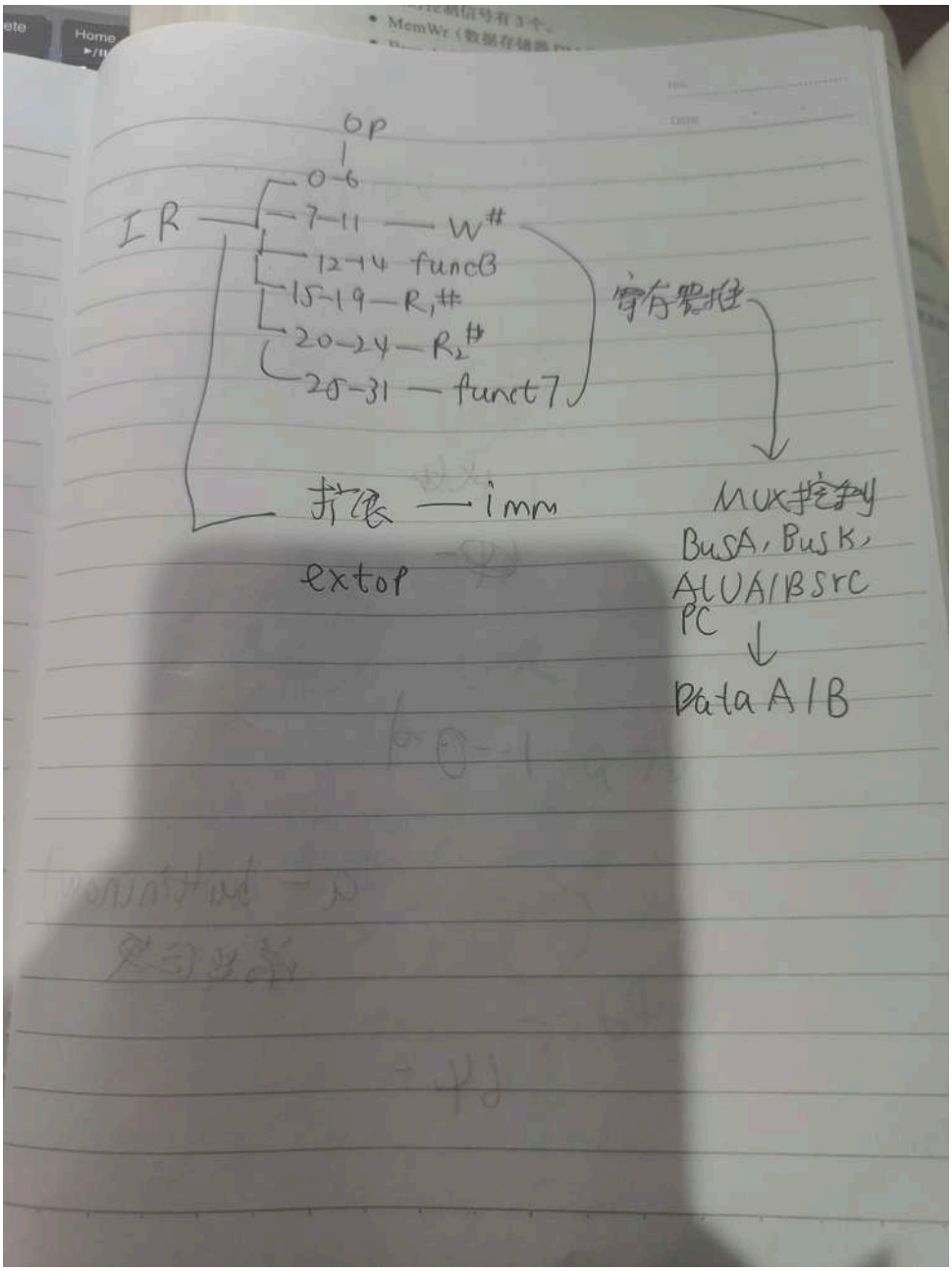


图 29: IDU 原理图

电路图实现：

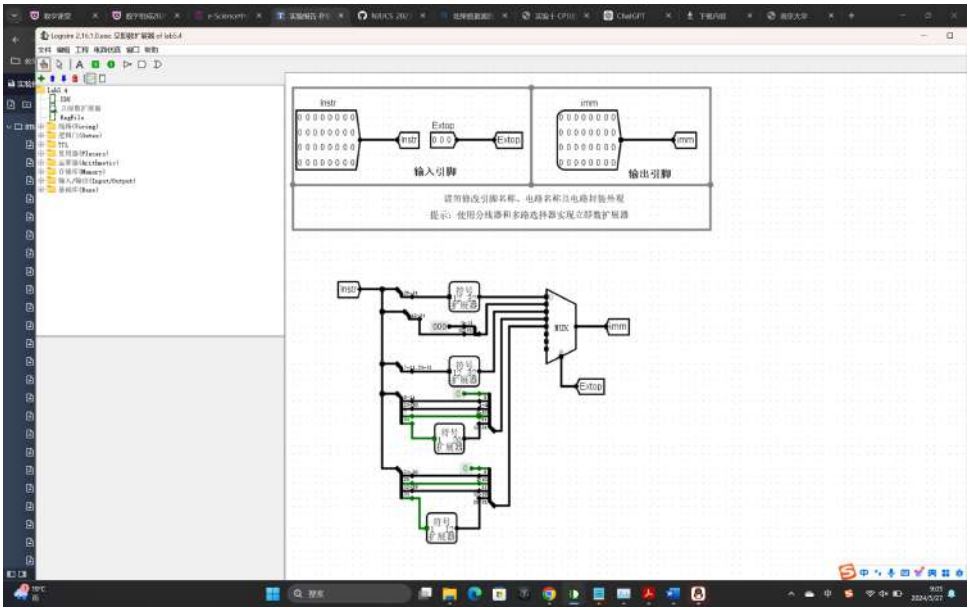


图 30: 立即数扩展器

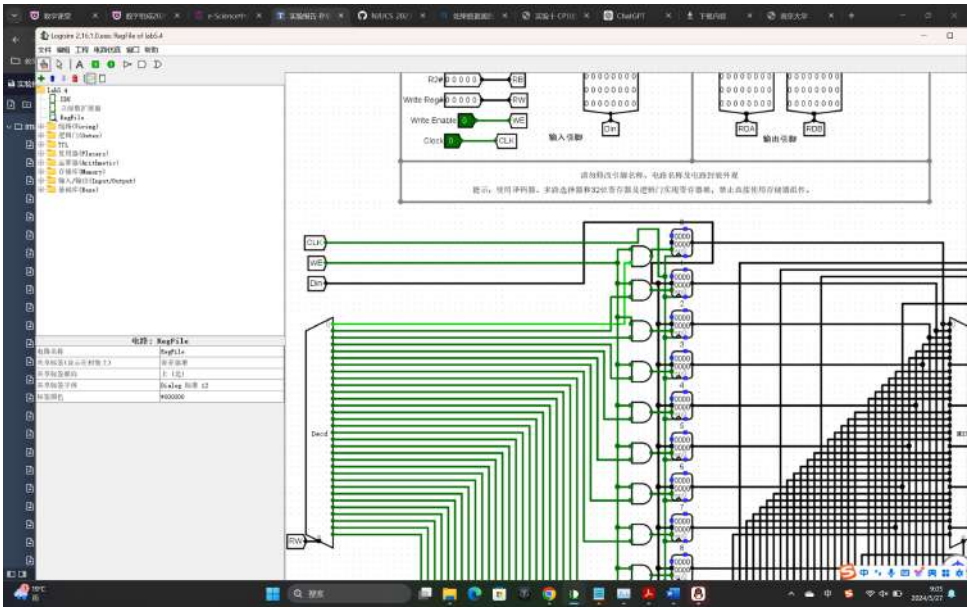


图 31: 寄存器堆(只展示修改部分)

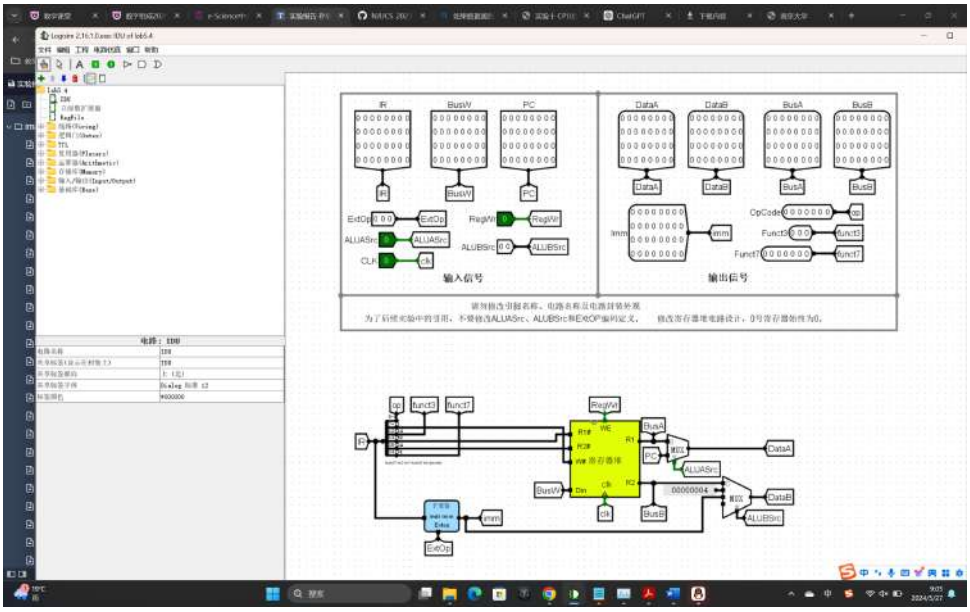


图 32: IDU

1.4.4 实验数据仿真测试图

测试点太多了,这里只给出几个示例。

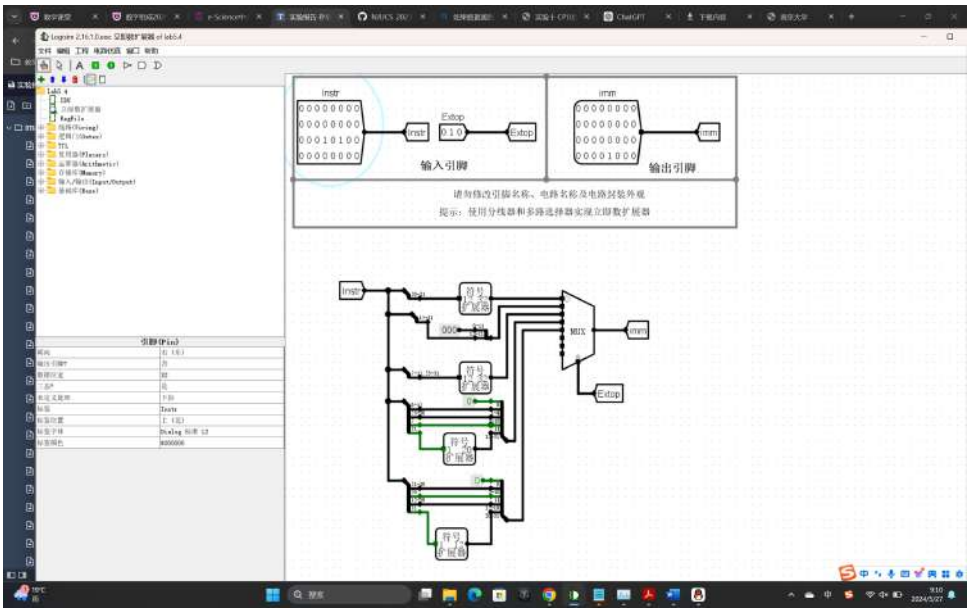


图 33: 立即数扩展器示例

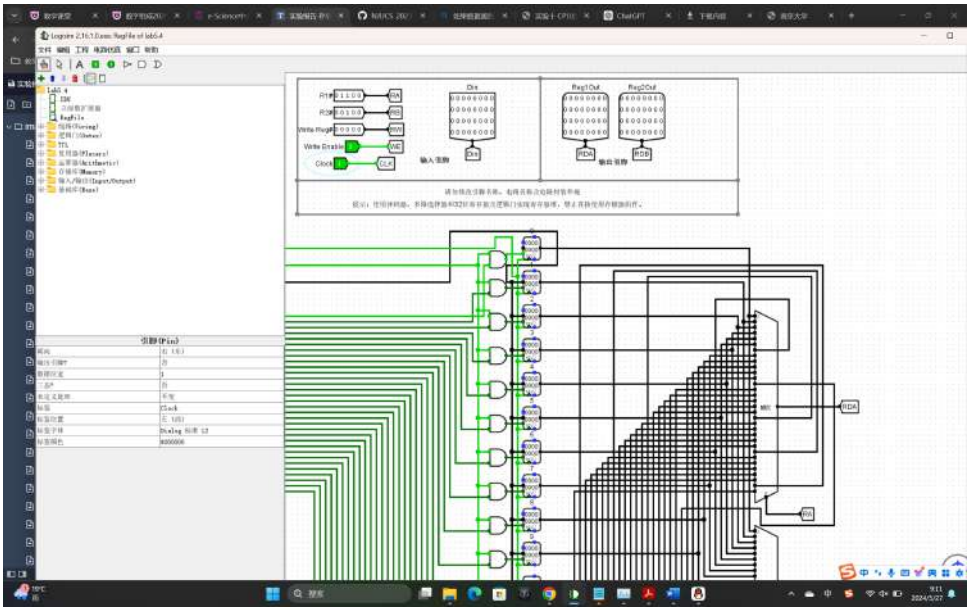


图 34: 零寄存器示意

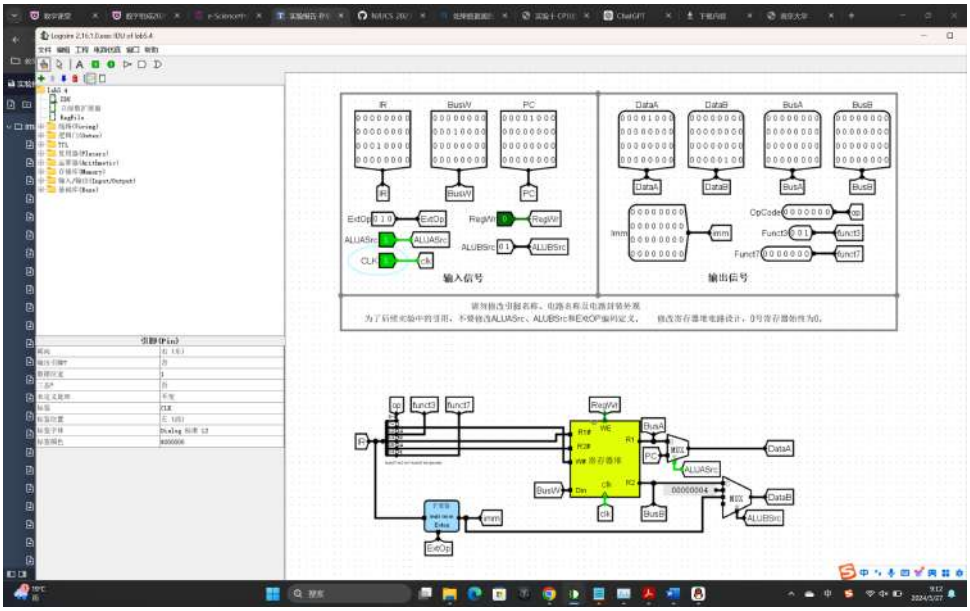


图 35: IDU 示意 1

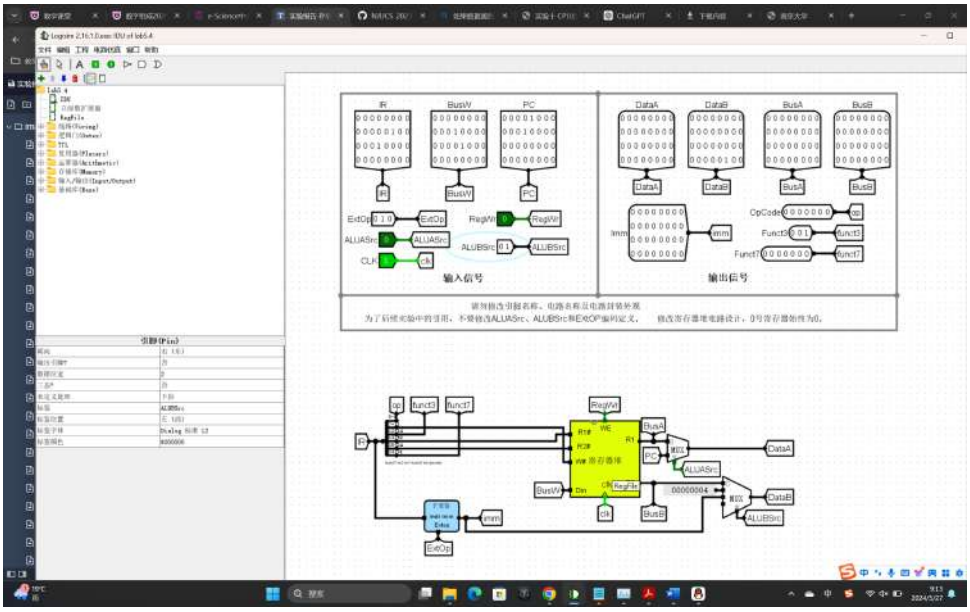


图 36: IDU 示意 2

1.4.5 错误现象及分析

在实验过程中,没有遇到任何错误。

1.5 数据通路实验

1.5.1 实验内容

1)设计跳转控制器子电路。
2) 修改取指令子电路。
3)数据通路实验。

1.5.2 实验整体方案设计

跳转控制器只要按照信号的极小项连线即可。
取指令子电路把 LD 删去就行。
数据通路按照原理图连线即可。
ALU 用上个实验的。


表 5.4 Branch 控制信号含义

Branch	NxtASrc	NxtBSrc	指令跳转类型
000	0	0	非跳转指令
001	0	1	jal：无条件跳转 PC 目标
010	1	1	jalr：无条件跳转寄存器目标
100	0	Zero	beq：条件分支，等于
101	0	! Zero	bne：条件分支，不等于
110	0	Result[0]	blt,bltu：条件分支，小于
111	0	Zero | (! Result[0])	bge,bgeu：条件分支，大于等于

在 Logisim 中添加一个名为“Branch”的子电路，双击该子电路名称，在右侧工作区中构建相应电路。

图 37: branch 信号含义

Branch	控制是否跳转
IFU	下地址逻辑
RAM	指令存储器
IDU	同上面实验
ALU	计算,生成信号
DataRam	数据存储器

表 5: 功能表

各种控制信号之前已讲。

1.5.3 实验原理图和电路图

原理图:BRANCH 不需要原理图。

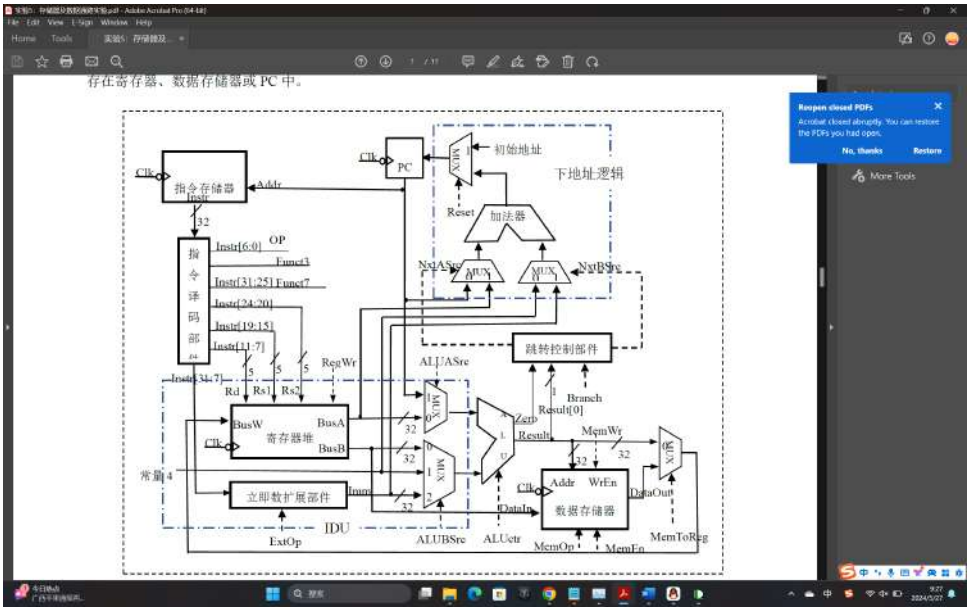


图 38: 数据通路原理图

电路图实现：

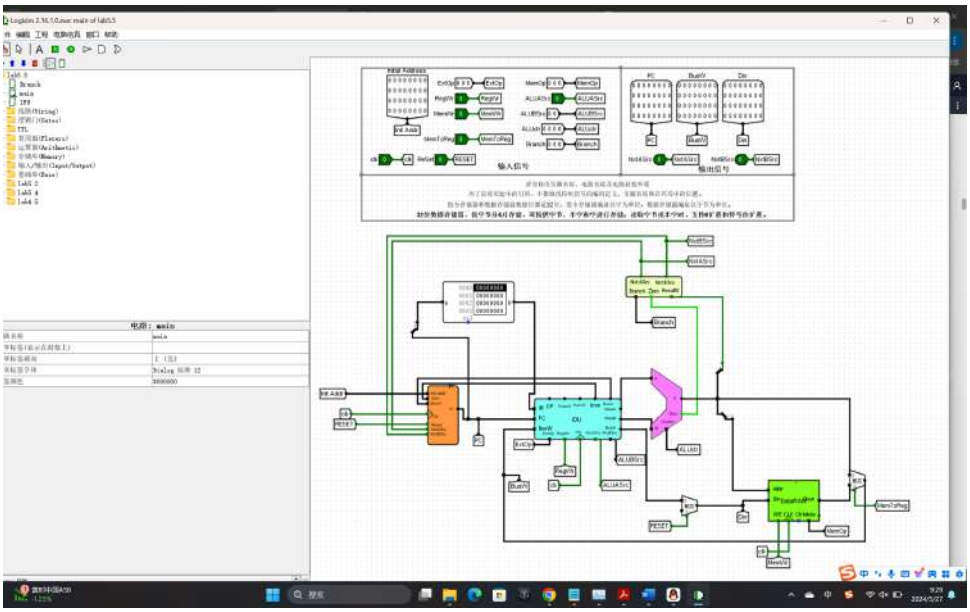


图 39: 主电路电路图

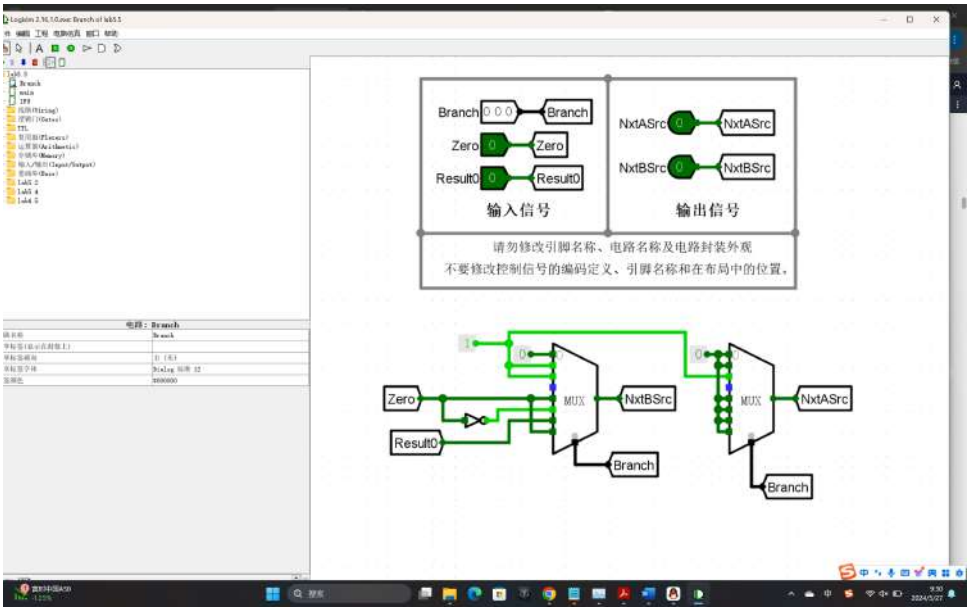


图 40: Branch

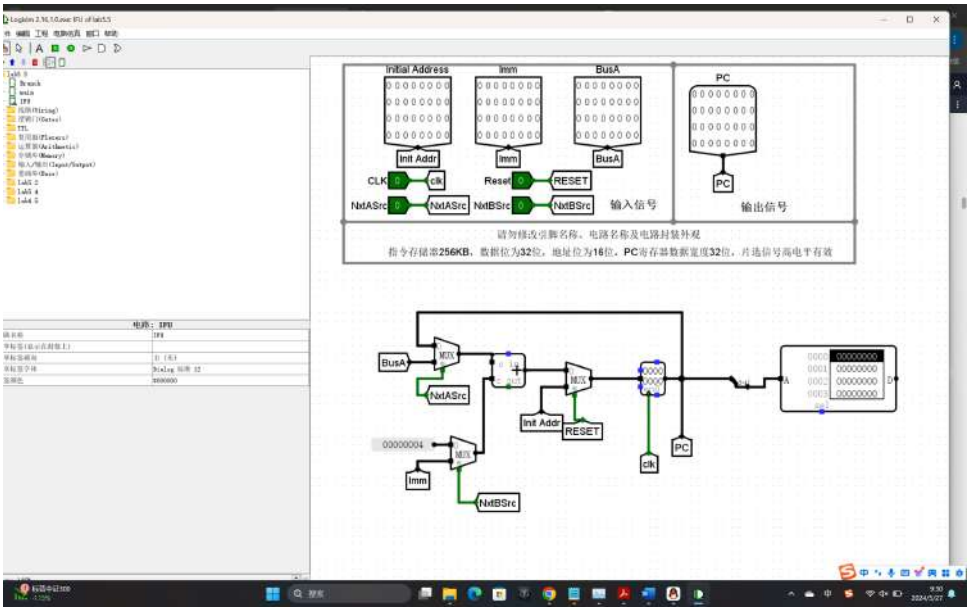


图 41: 修改后的 IFU

1.5.4 实验数据仿真测试图

作为示例,展示前六个时钟周期。

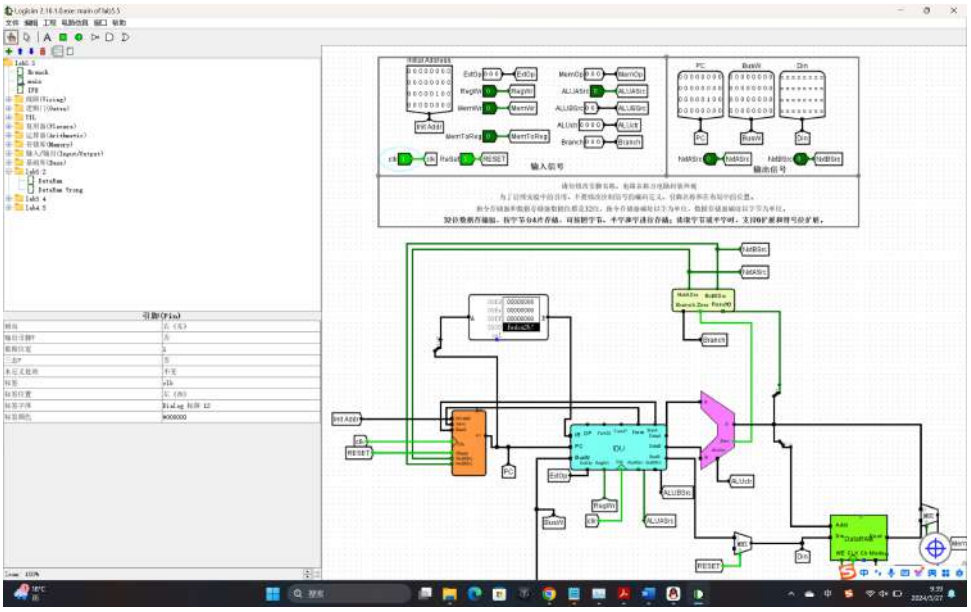


图 42: 1

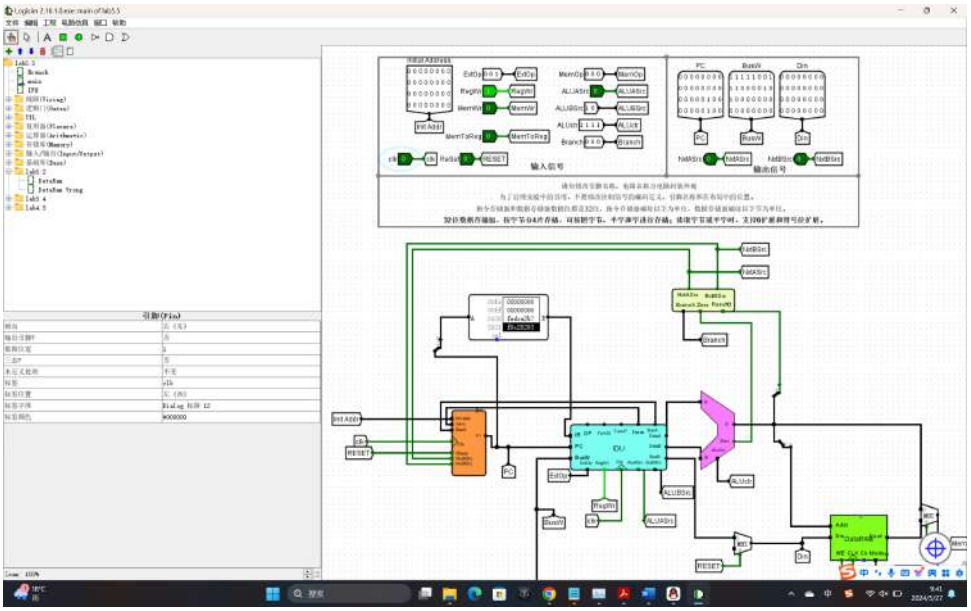


图 43: 2

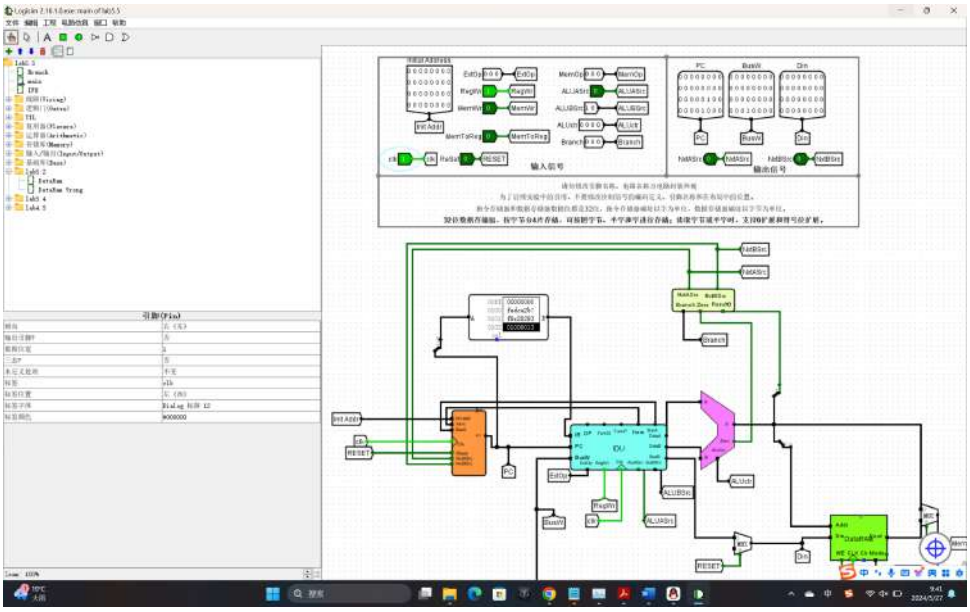


图 44: 3

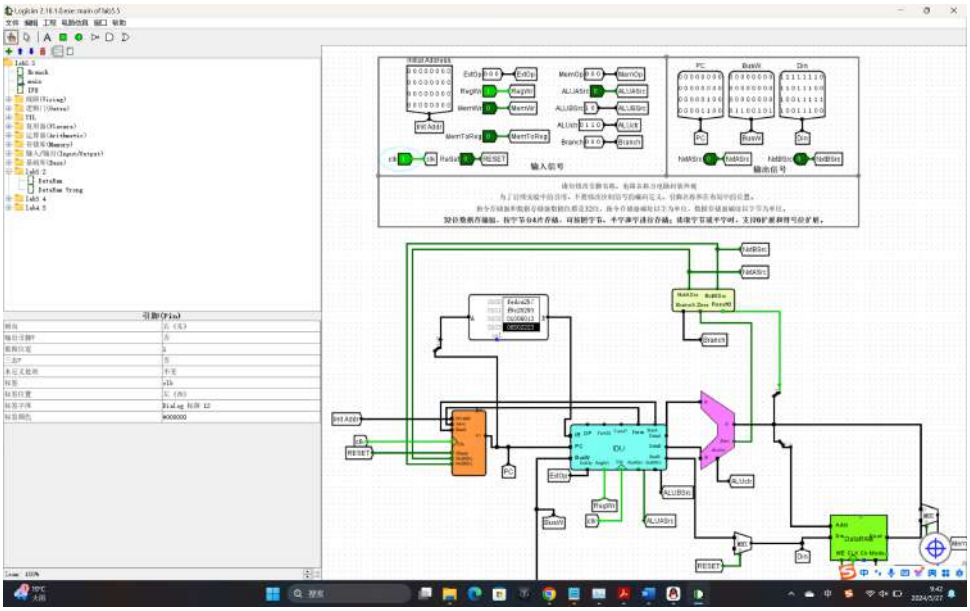


图 45: 4

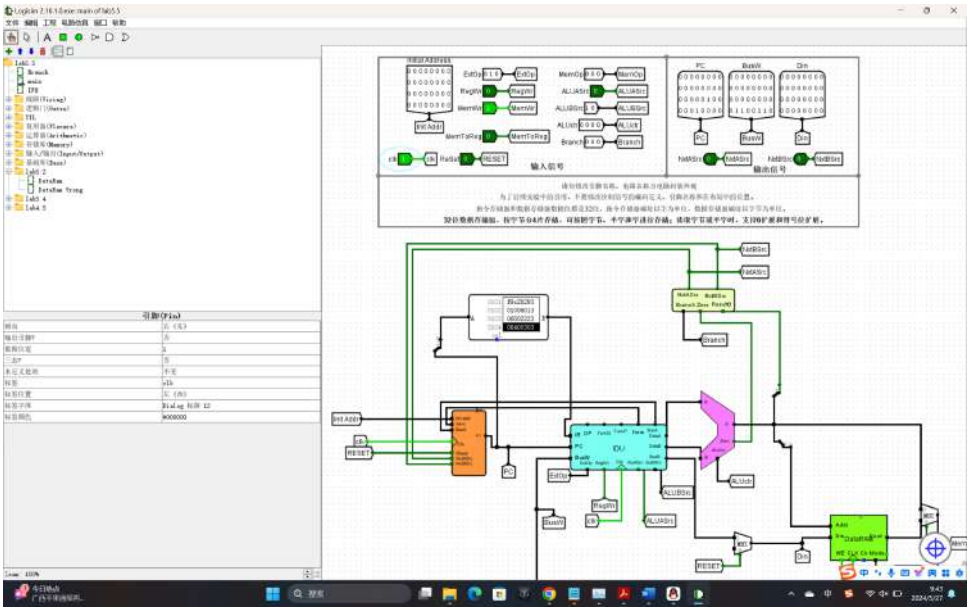


图 46: 5

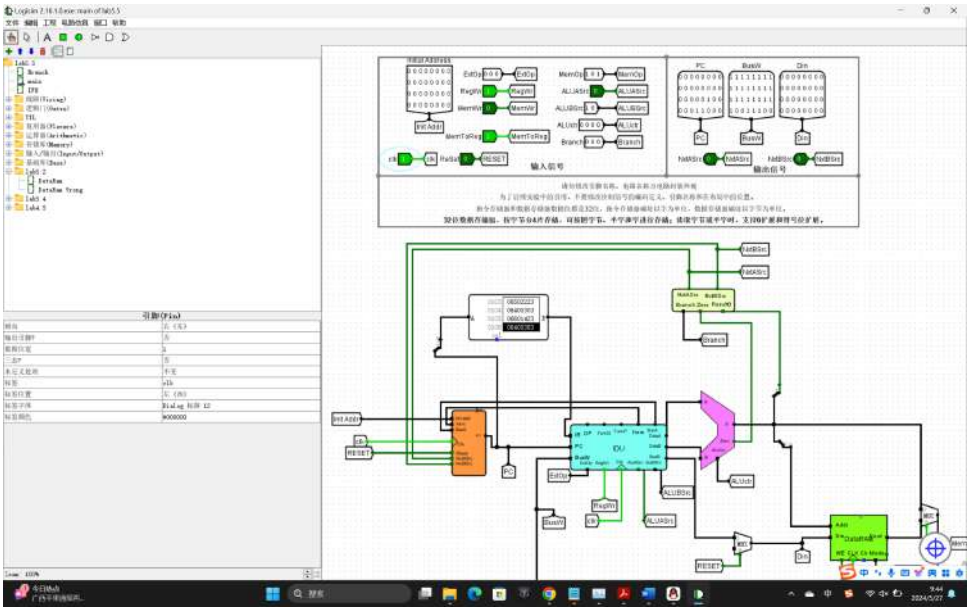


图 47: 6

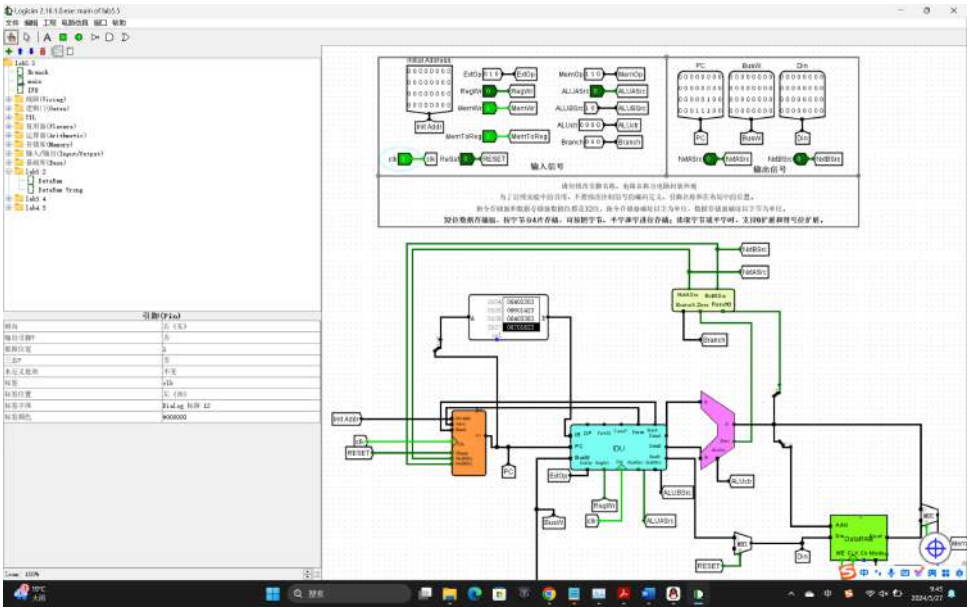


图 48: 7

不需要真值表。

1.5.5 错误现象及分析

1. 在实验过程中，明明原理图正确，但是结果却不对，后来经过分析后，发现是 5.2 的 ADDR 使用了寄存器的问题，使得时钟周期不匹配（详见 5.2 错误分析），后来修改 5.2 后解决
2. 第一个样例 din 为 xxxx，而我输出的为 0000，后来用 reset 和多路选择器控制解决。

2 思考题

2.1 思考题 1

1. 如何利用 ROM 实验实现滚动显示的功能，在 3 个 LED 点阵矩阵中，左右滚动显示 5 个 ASCII 字符，如“NJUCS”。

首先，为了方便，我生成了一个只含有 NJUCS 的镜像文件，存入实验 1 的 ROM 中，为了方便，我去掉了 - 0x20 和比较的地址。

然后，我的实现电路利用了全加器，下面是电路图。助教可以自己测试一下效果。

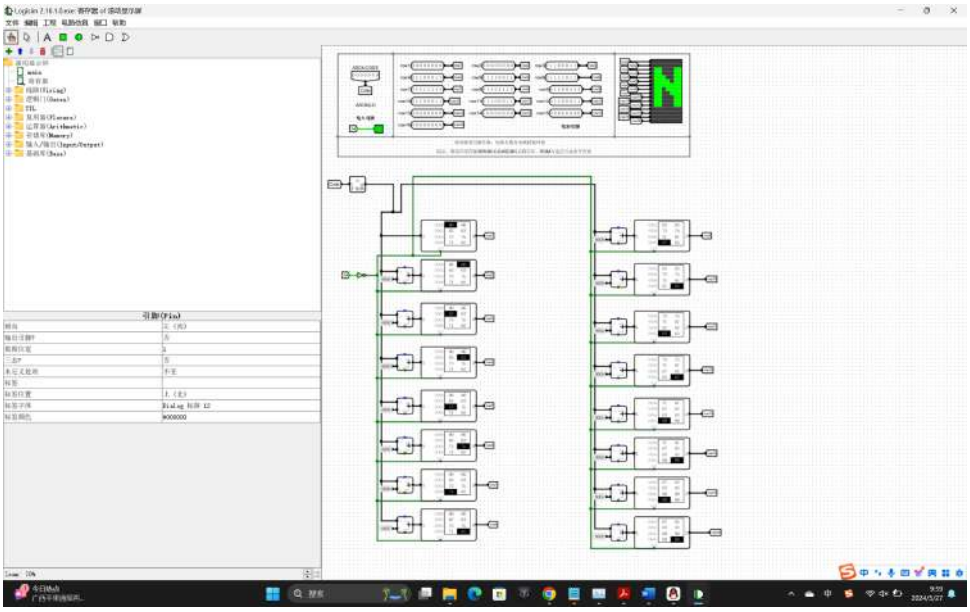


图 49: 1

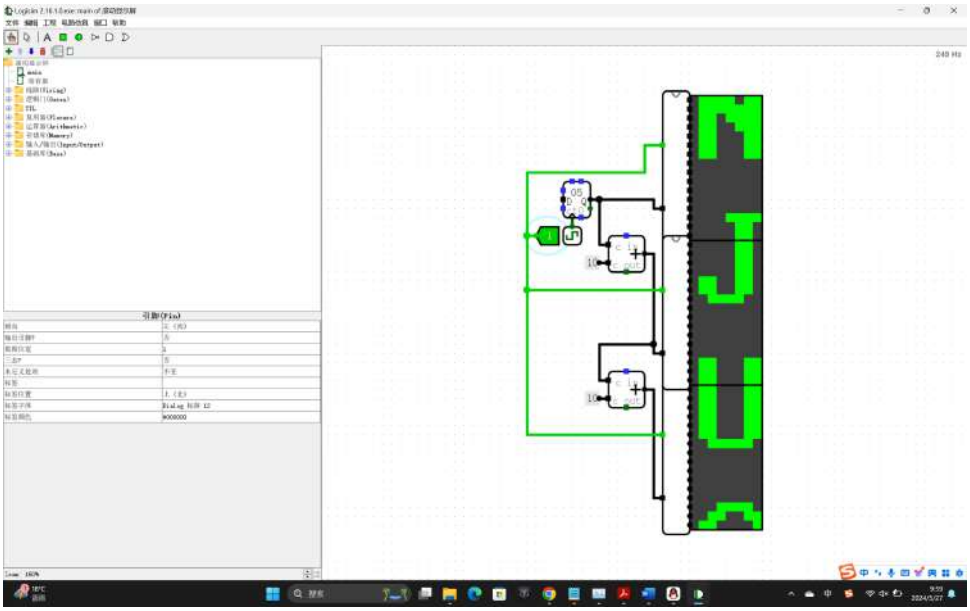


图 50: 2

2.2 思考题 2

2. 分析说明如果寄存器堆写入数据时是下降沿触发有效而 PC 寄存器和数据存储器写入时是上升沿触发有效,则对程序执行结果有什么影响

1. 时序问题: 使寄存器中的数据更新与 PC 和数据存储器的数据更新不同步, 导致执行当前指令时使用的是旧数据。

不同触发沿的更新会导致某些数据在一个周期内没有被正确同步, 写入的数据和寄存器堆读到的数据不一样,导致数据的读取出错。

2.3 思考题 3

3. 在 CPU 启动执行后,如何实现在当前程序结束后,CPU 不再继续执行指令

对于数据通路,将 IFU 的 RESET 改为 1 就能使 PC 停止输出,进而停止执行指令。就程序而言,有很多种返回函数的方法使 CPU 停止执行指令。