实验六 CPU 综合设计

一、实验目的

1 掌握复杂系统设计方法。

2 深刻理解计算机系统硬件原理。

二、实验内容

1）设计一个基于 MIPS 指令集的 CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；

2）CPU 需要包含寄存器组、RAM 模块、ALU 模块、指令译码模块；

3）该 CPU 能运行基本的汇编指令；（D~C+）

以下为可选内容：

4）实现多周期 CPU（B-~B+）；

5）实现以下高级功能之一（A-~A+）：

(1)实现 5 级流水线 CPU；

(2)实现超标量；

(3)实现 4 路组相联缓存；

可基于 RISC V 、ARM 指令集实现。

如发现代码为抄袭代码，成绩一律按不及格处理。

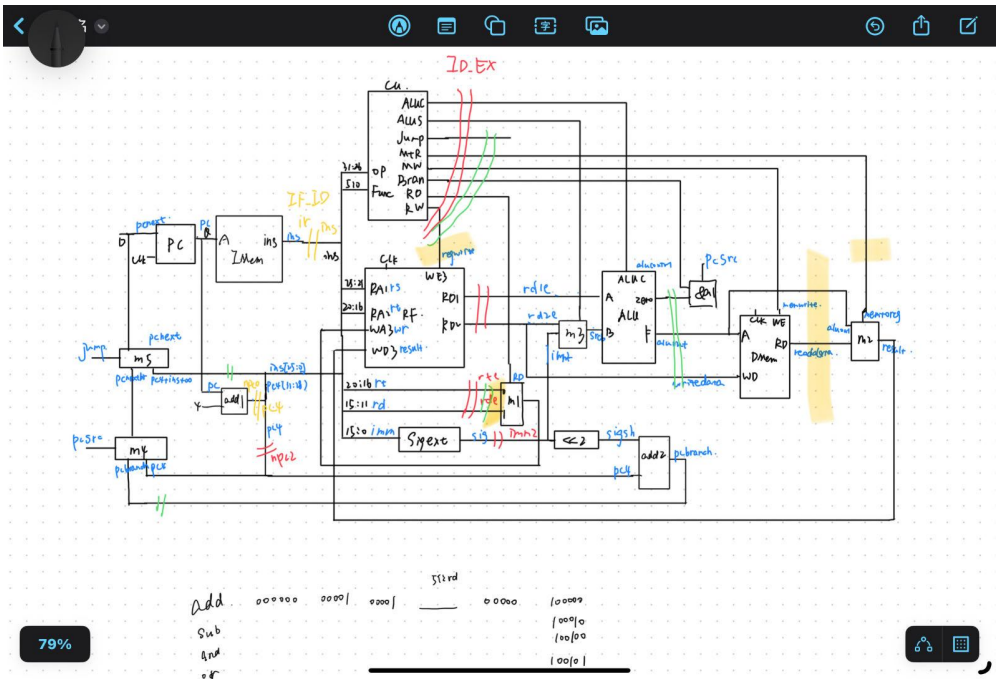
三、实验要求

编写相应测试程序，完成所有指令测试。

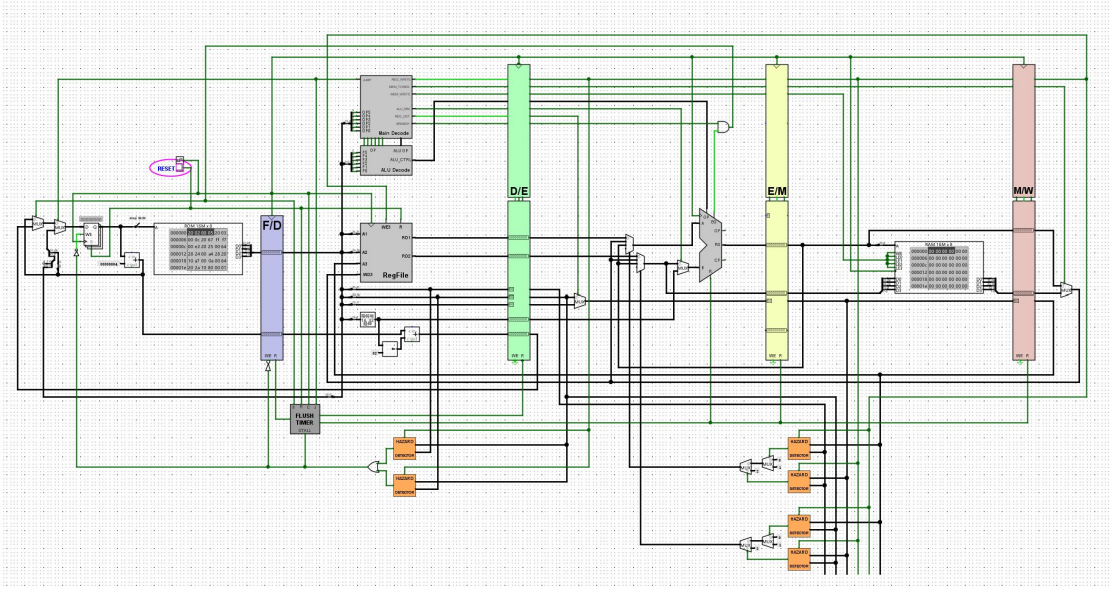
四、实验代码及结果

实现五级流水 CPU 加上冒险功能（logisim 实现）：

五级流水电路图（手写版）：



五级流水 logisim 图片（含冒险）：



以下是 MIPS 指令集的结构：

指令长度： 所有指令都是 32 位长，即 4 个字节。

指令格式： MIPS 指令格式主要分为三种：R（寄存器）、I（立即数）、J（跳转）。具体格式如下：

R-Format： 用于寄存器之间的操作，如 add, sub 等。

opcode | rs | rt | rd | shamt | funct

6 bits | 5 | 5 | 5 | 5 | 6 bits

I-Format： 包括常见的立即数指令，如 addi, lw, sw 等。

opcode | rs | rt | immediate

6 bits | 5 | 5 | 16 bits

J-Format： 主要用于跳转指令，如 j 等。

opcode | target address

6 bits | 26 bits

字段解释：

opcode: 操作码字段，指定指令的基本操作类型。

rs, rt, rd: 寄存器字段，表示源寄存器、目标寄存器等。

shamt: 移位操作时使用的位移量。

funct: 功能字段，与 opcode 一起指定具体的操作。

以上电路实现的 MIPS 指令有 add, sub, and, or, slt, addi, lw, sw, beq, j;

其中 R-Type 有 add, sub, and, or, slt;

I-Type 有 addi, lw, sw;

J-Type 有 beq 和 j;

PC：

下一条指令的 PC 值可以有以下几种可能情况：

顺序执行： 在大多数情况下，PC 的值是当前指令的地址加上指令的长度。这是最常见的情况，表示按顺序执行下一条指令。

beq 指令：

beq 指令用于条件分支，当两个寄存器的内容相等时，会发生分支。其格式如下：

beq $rs, $rt, offset

如果 $rs 和 $rt 的内容相等，将执行分支，offset 是一个相对地址，表示相对于当前指令的偏移量。计算下一条指令的 PC 值的步骤如下：

如果分支条件成立，则 PC = PC + 4 + offset * 4

如果分支条件不成立，则 PC = PC + 4

这里的 offset 需要乘以 4，因为 MIPS 指令的长度是 4 个字节。

j 指令：

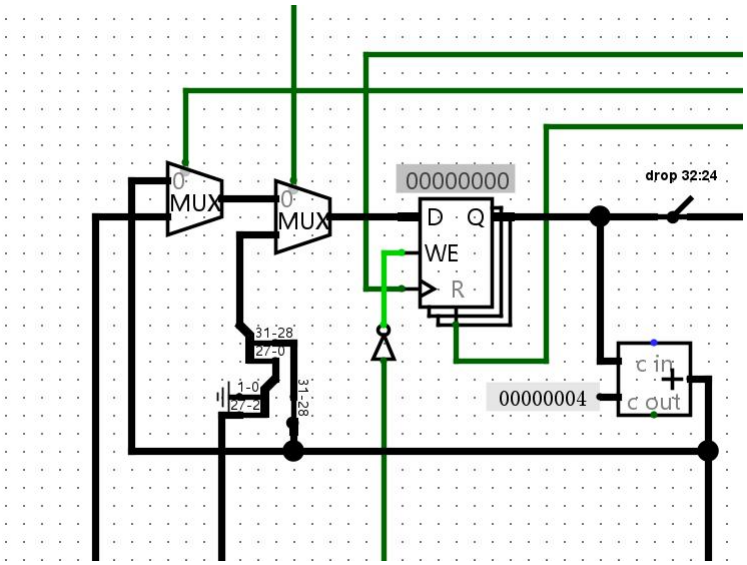
j 指令用于无条件跳转，其格式如下：

j target_address

target_address 是直接给出的跳转目标地址。计算下一条指令的 PC 值的步骤如下：

PC = target_address

在这两种情况下，都需要在指令执行前正确计算出下一条指令的 PC 值，并将其加载到程序计数器（PC）中。综合起来，如果在处理 beq 或 j 指令时有一个相对地址或者立即数，可以先进行符号扩展，然后再乘以 4，确保得到正确的地址。



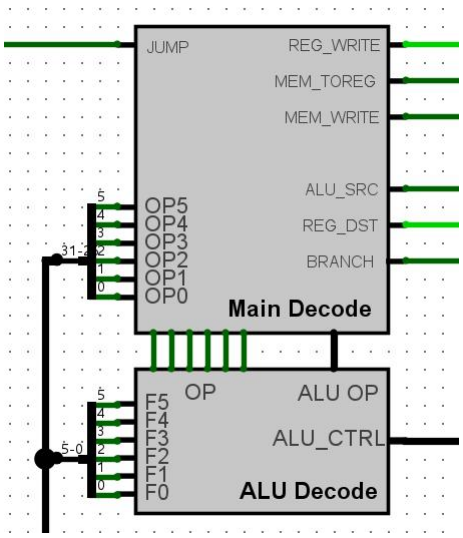
pcsrc 信号：

对于 pcsrc 信号，通常与分支指令相关。这个信号用于决定是选择顺序执行（PC + 4）还是跳转到分支目标地址。在流水线的 EX 阶段（执行阶段），pcsrc 信号根据分支条件的真假进行设置。

jump 信号：

对于 jump 信号，通常与无条件跳转指令（例如 j 指令）相关。这个信号用于决定是否执行跳转，并且跳转到 j 指令指定的地址。在流水线的 ID 阶段（译码阶段），jump 信号根据当前指令是否是 j 指令进行设置。

Controller：



指令类型识别：

ALUCtrl：指定 ALU（算术逻辑单元）的操作类型，例如加法、减法、逻辑与等。

MemWrite：用于指示是否需要向内存写入数据（例如，sw 指令）。

MemtoReg：用于指示数据来源，是来自 ALU 的结果还是来自内存。

寄存器文件控制：

RegDst：指定目标寄存器的选择，即目标寄存器的编号是来自指令中的哪一个字段。

RegWrite：用于指示是否需要写回寄存器。

分支和跳转控制：

Branch：用于指示是否为分支指令。

Jump：用于指示是否为跳转指令。

ALUsrc：选择 ALU 的运算数；

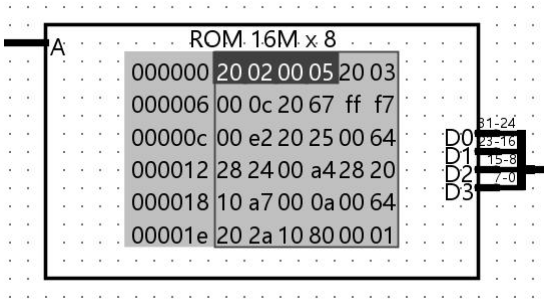
表 7-4　主控制单元真值表

指令	OP_Code	RegDst	RegWr	ALUSrc	MemRd	MemWr	MemtoReg	Branch	JUMP	ALUOp
R-型	000000	1	1	0	0	0	0	0	0	10
LW	100011	0	1	1	1	0	1	0	0	00
SW	101011	X	0	1	0	1	X	0	0	00
BEQ	000100	X	0	0	0	0	X	1	0	01
J	000010	X	0	X	0	0	X	0	1	XX

表 7-5　ALU 控制单元真值表

ALUOp	Funct	ALUCtrl
00	X	100(ADD)
X1	X	110(SUD)
1X	100000	100(ADD)
1X	100001	101(ADDU)
1X	100010	110(SUB)
1X	100100	000(AND)
1X	100101	001(OR)
1X	101010	011(SLT)

指令存储器：



从 pc 处接收指令地址，读取指令输出；

存储指令： 指令存储器负责存储计算机程序的指令集。这些指令用于告诉计算机硬件执行特定的操作，例如加法、减法、加载数据等。指令存储器中的指令按照程序的逻辑顺序排列，计算机在执行程序时按照这个顺序逐条获取和执行指令。

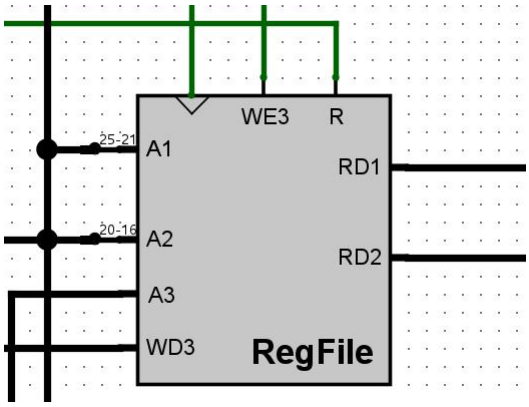
提供指令地址： 指令存储器根据计算机的程序计数器（PC）提供的地址，将相应地址处的指令传送到计算机的指令寄存器，供执行单元执行。PC 存储了当前正在执行的指令的地址，当执行完一条指令后，PC 会自动递增，以指向下一条指令的地址。

支持指令的顺序执行： 指令存储器按照程序的逻辑顺序存储指令，使得计算机能够按照顺序执行程序。这对于流水线处理器等多级流水线结构非常重要，因为流水线的各个阶段需要按照指令的顺序协同工作。

支持跳转和分支： 指令存储器使得计算机能够实现跳转和分支，通过在程序中设置跳转和分支指令，程序可以根据条件选择性地执行不同的指令路径。这对于控制程序的流程非常重要。

实现指令的随机访问： 指令存储器支持随机访问，计算机可以根据指定的地址直接访问存储器中的指令。这与顺序访问的数据存储器有所不同，指令存储器的随机访问性能对计算机指令的快速获取和执行至关重要。

寄存器文件：regfile



用于存储和管理计算机中的寄存器。寄存器是一种存储器件，用于暂时存储计算机中的数据。在给定的寄存器文件中，通常包含多个寄存器，每个寄存器都可以存储一个数据值。

对于输入和输出信号的描述：

输入信号：

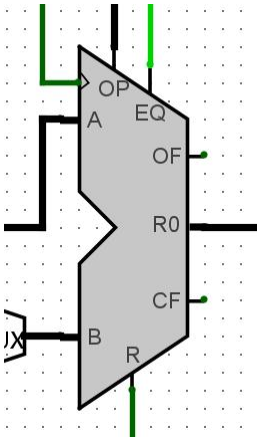
RA1（Read Address 1）和 RA2（Read Address 2）： 用于指定要读取的寄存器的地址。RA1 和 RA2 分别对应于读取的两个操作数的寄存器地址。

WA3（Write Address 3）： 用于指定要写入的寄存器的地址，即要将数据写入的寄存器。

WD3（Write Data 3）： 要写入的数据值。这是要写入指定寄存器的数据。
时钟和控制信号：
CLK（Clock）： 时钟信号，用于同步寄存器文件的读写操作。
CLEAR： 清除信号，用于清空寄存器文件中的所有数据。
WE（Write Enable）： 写使能信号，用于控制写入操作的进行或停止。
输出信号：
RD1（Read Data 1）和 RD2（Read Data 2）： 从指定地址读取的两个操作数的数据值。这是从寄存器文件读取的数据，用于提供给执行单元等其他部件进行运算。

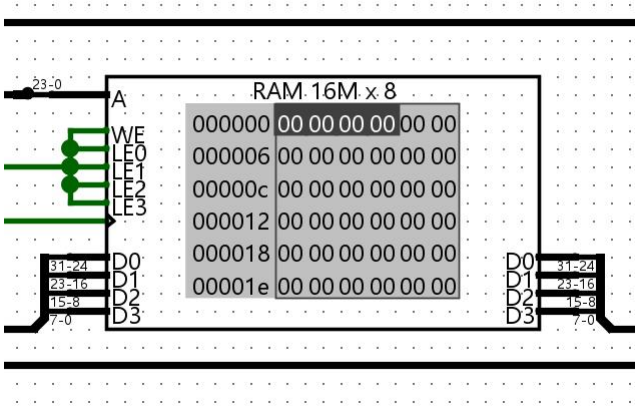
在流水线 CPU 中，寄存器文件通常用于暂时存储和传递指令执行的中间结果。通过合理的设计和控制，可以在多个时钟周期内实现并行的读写操作，以支持流水线的运行。

ALU：



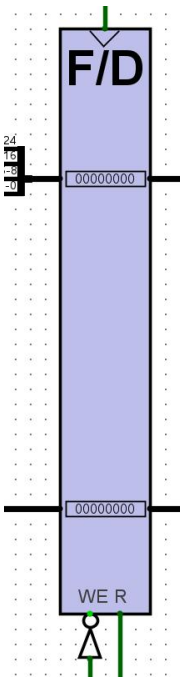
算术运算： ALU 能够执行各种算术运算，如加法、减法、乘法和除法。这些运算是计算机进行数值计算和数据处理的基础。
逻辑运算： ALU 能够执行逻辑运算，如与、或、非、异或等。逻辑运算用于处理二进制数据的位级操作，支持条件判断和位操作。
比较操作： ALU 可以比较两个数的大小，并生成相应的比较结果。比较操作常用于支持条件分支和判断。

数据存储器：



输入信号：
WE（Write Enable）： 写使能信号，用于控制是否允许写入数据到数据存储器。当 WE 为高电平时，表示允许写入。
A（Address）： 地址信号，用于指定要读取或写入的存储单元的地址。
D（Data）： 数据信号，用于写入到数据存储器的数据值。
输出信号：
D（Data）： 读取的数据信号，表示从数据存储器中读取的数据值。
数据存储器的基本作用是存储和提取数据。其具体功能如下：
读取操作： 当 CPU 需要从存储器中读取数据时，通过将地址信号 A 发送给数据存储器，存储器从指定地址读取数据，并通过输出信号 D 返回给 CPU。
写入操作： 当 CPU 需要向存储器写入数据时，通过将地址信号 A、数据信号 D 和写使能信号 WE 发送给数据存储器，存储器在指定地址写入相应的数据。
存储空间的划分： 数据存储器通常被划分为多个存储单元，每个存储单元有唯一的地址。地址通过地址信号 A 传递，确定了要读取或写入的具体存储单元。
读写控制： 写使能信号 WE 控制了写入操作的进行，确保数据在需要时被正确写入存储器。
数据存储器在计算机系统中的角色是存储程序执行过程中产生的中间结果、变量值和其他需要在运行时进行读写的数据。**在流水线 CPU 中，数据存储器通常与寄存器文件一起工作，支持指令的执行和数据的传递。**

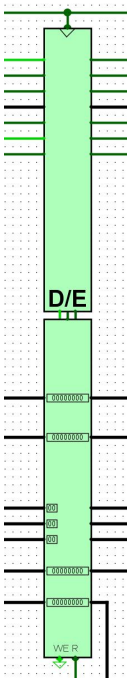
第一级流水：IF_ID



IF_ID 寄存器的完整输入为：（冒险的输入之后再补充）
指令（Instruction）： 来自指令存储器的输出，表示当前时钟周期从存储器中取出的指令。
PC（程序计数器）： 当前时钟周期的程序计数器的值，用于指示下一条要取的指令的地址。
CLK（时钟信号）： 用于同步流水线各个阶段的时钟信号。
而输出仍然包括：
指令（Instruction）： 传递给下一级流水线阶段（Instruction Decode）的指令。
PC（程序计数器）： 传递给下一级流水线阶段的程序计数器的值，用于指示下一条要取的指

令的地址。

ID_EX：



输入：

regwrite： 写寄存器的控制信号。

memtoreg： 决定写回寄存器的数据来源，是来自内存（存储器）还是 ALU 的输出。

memwrite： 写内存的控制信号。

alusrc： ALU 的第二个操作数选择信号，决定是选择立即数还是寄存器的数据作为 ALU 的第二个操作数。

regdst： 写寄存器的目标寄存器选择信号。

branch： 分支指令的控制信号。

aluctrl： ALU 控制信号，用于指示 ALU 执行何种操作。

rd1，rd2： 来自寄存器堆的读取数据信号，分别对应源寄存器 rs 和 rt 的值。

rs，rt： 源寄存器的编号。

rd： 目标寄存器的编号。

扩展后的立即数： 立即数经过符号扩展后的值。

pc： 当前时钟周期的程序计数器的值。

clk： 时钟信号。

输出：

regwrite： 传递给下一级流水线阶段的写寄存器的控制信号。

memtoreg： 传递给下一级流水线阶段的写回寄存器的数据来源信号。

memwrite： 传递给下一级流水线阶段的写内存的控制信号。

alusrc： 传递给下一级流水线阶段的 ALU 的第二个操作数选择信号。

regdst： 传递给下一级流水线阶段的写寄存器的目标寄存器选择信号。

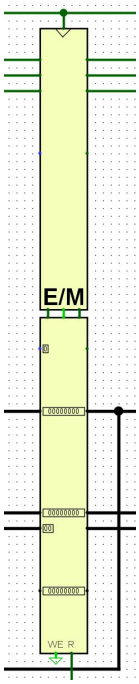
branch： 传递给下一级流水线阶段的分支指令的控制信号。

aluctrl： 传递给下一级流水线阶段的 ALU 的控制信号。

rd1，rd2： 传递给下一级流水线阶段的来自寄存器堆的读取数据信号。

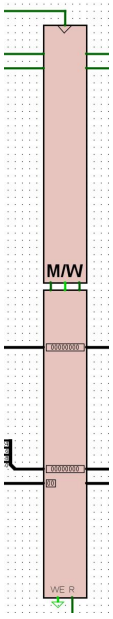
rs，rt： 传递给下一级流水线阶段的源寄存器的编号。
rd： 传递给下一级流水线阶段的目标寄存器的编号。
扩展后的立即数： 传递给下一级流水线阶段的立即数经过符号扩展后的值。
pc： 传递给下一级流水线阶段的程序计数器的值。
这个寄存器的作用是传递来自 ID 阶段的控制信号和数据，以供 EX 阶段使用。

EX_MA：



输入
regwrite： 写寄存器的控制信号。
memtoreg： 决定写回寄存器的数据来源，是来自内存（存储器）还是 ALU 的输出。
memwrite： 写内存的控制信号。
aluout： ALU 的输出结果。
写回的地址： 要写入的寄存器的地址。
写入的数据： 要写入 DM 的数据。
clk： 时钟信号。
输出：
regwrite： 传递给下一级流水线阶段的写寄存器的控制信号。
memtoreg： 传递给下一级流水线阶段的写回寄存器的数据来源信号。
memwrite： 传递给下一级流水线阶段的写内存的控制信号。
aluout： 传递给下一级流水线阶段的 ALU 的输出结果。
写回的地址： 传递给下一级流水线阶段的要写入的寄存器的地址。
写入的数据： 传递给下一级流水线阶段的要写入 DM 的数据。

MA_WB:



输入：

regwrite： 写寄存器的控制信号。

memtoreg： 决定写回寄存器的数据来源，是来自内存（存储器）还是 ALU 的输出。

aluout： ALU 的输出结果。

readdata： 从内存中读取的数据。

写回的地址： 要写入的寄存器的地址。

clk： 时钟信号。

输出：

regwrite： 传递给下一级流水线阶段（写回阶段）的写寄存器的控制信号。

memtoreg： 传递给下一级流水线阶段的写回寄存器的数据来源信号。

aluout： 传递给下一级流水线阶段的 ALU 的输出结果。

readdata： 传递给下一级流水线阶段的从内存中读取的数据。

写回的地址： 传递给下一级流水线阶段的要写入的寄存器的地址。

MA_WB 寄存器的作用是在访存阶段和写回阶段之间传递数据和控制信号，以便于在写回阶段完成对寄存器的写入操作。

冒险解决：

可能出现的冒险：

常见的冒险有数据冒险、控制冒险和结构冒险。下面是各种冒险的简要说明和例子：

数据冒险： 数据冒险指的是在读取或写入寄存器时出现的冲突。有三种类型：

读后写（RAW）冒险： 一个指令在执行阶段写回一个寄存器，而另一个指令在取数阶段要读取同一个寄存器的值。

写后写（WAW）冒险： 两个指令都试图在执行阶段写回同一个寄存器。

写后读（WAR）冒险： 一个指令在执行阶段写回一个寄存器，而另一个指令在取数阶段要读取同一个寄存器的值。

控制冒险： 控制冒险发生在分支指令导致流水线流向错误的情况。例如，当一个分支指令判断为真时，流水线应该转向分支目标地址，否则应该继续按顺序执行。由于分支判断需要在执行阶段完成，因此在这之前的指令已经在流水线中执行，导致控制冒险。

结构冒险： 结构冒险是由于硬件资源的限制导致的，例如多个指令需要访问相同的硬件单

元，但每个硬件单元在同一时刻只能服务一个指令。常见的结构冒险包括数据存储器（Data Memory）和乘法器。

下面是一些具体的例子：
数据冒险：

```
add $t0, $t1, $t2     # 指令 1
sub $t3, $t0, $t4     # 指令 2（RAW 冒险）
```

控制冒险：

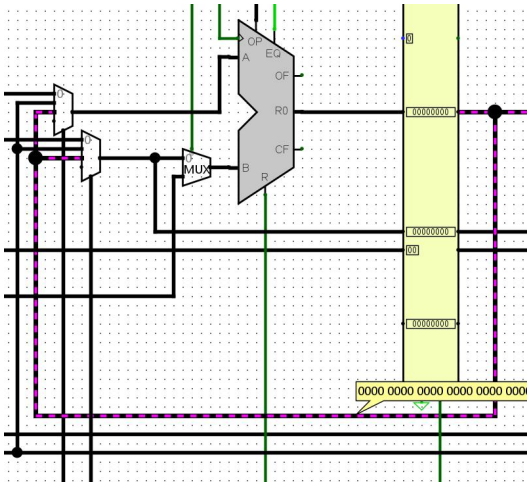
```
beq $t0, $t1, Label    # 指令 1
add $t2, $t3, $t4      # 指令 2（控制冒险）
```

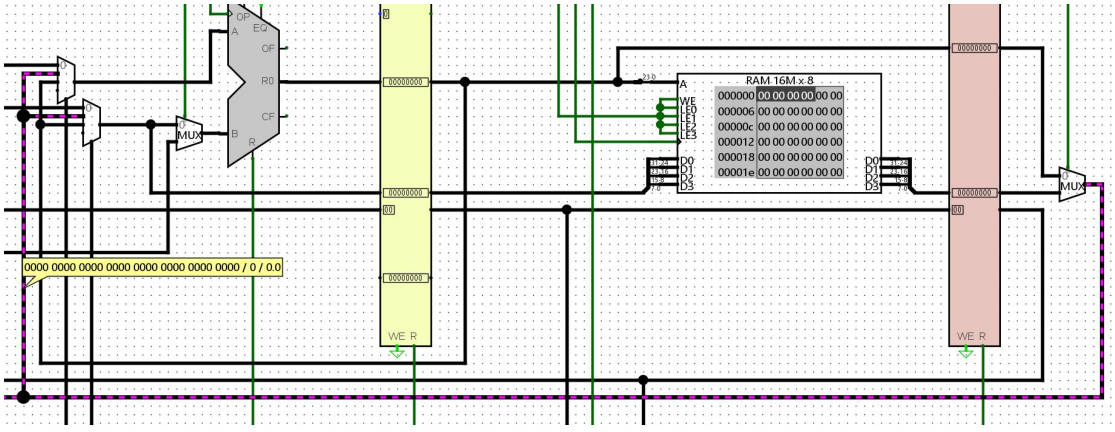
结构冒险：

```
lw $t0, 0($t1)         # 指令 1
sw $t2, 4($t1)         # 指令 2（数据存储器结构冒险）
```

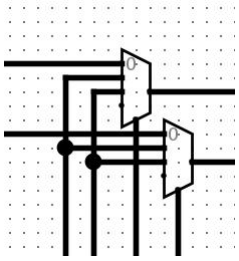
冒险的解决：

数据冒险：
在数据前传中，执行阶段计算的结果直接传递给后续阶段，以供需要的指令使用。这可以通过硬件上的旁路单元来实现。数据前传可以分为三种类型：EX-EX 前传、MEM-EX 前传和 WB-EX 前传，分别对应执行阶段到执行阶段、内存阶段到执行阶段和写回阶段到执行阶段的数据前传。

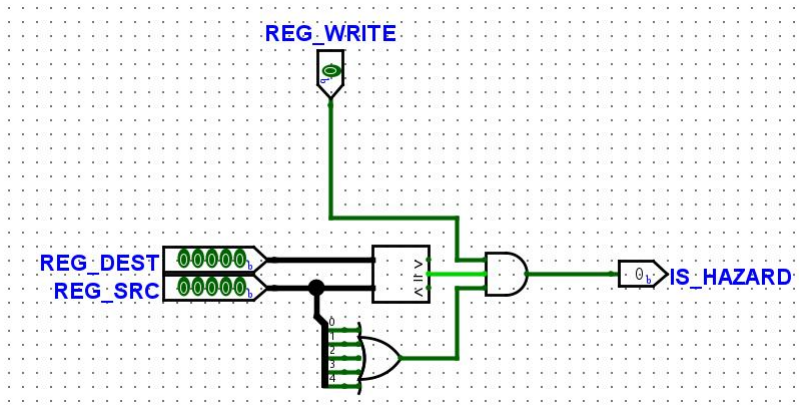


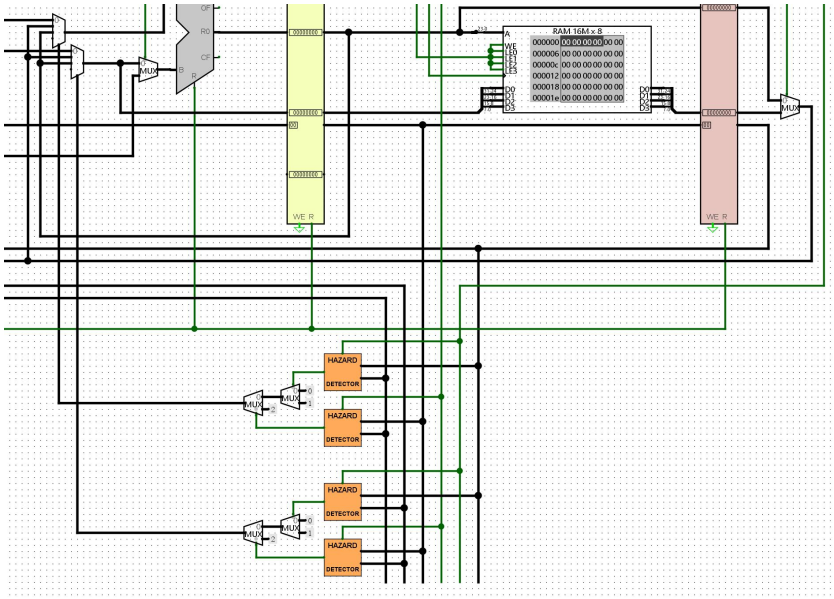


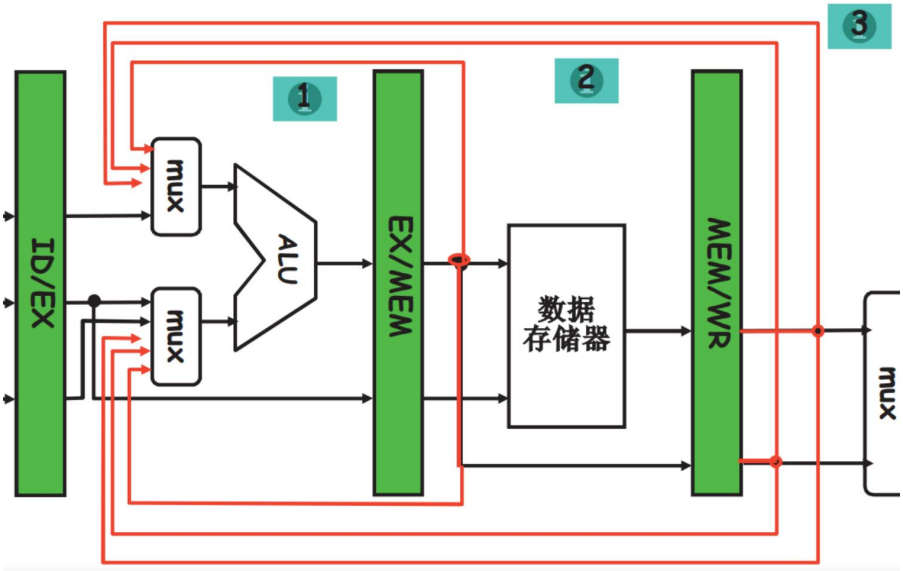
以上是两种数据前传的电路图；



在使用数据前传时，需要在 ALU 前添加多路选择器，选择的数据包括：
寄存器堆写回的数据（WB-EX 前传）： 当执行阶段需要读取写回阶段的寄存器数据时，选择寄存器堆写回的数据。
内存阶段的 ALU 计算结果（MEM-EX 前传）： 当执行阶段需要读取内存阶段的 ALU 计算结果时，选择内存阶段的 ALU 计算结果。
这些数据通过多路选择器进行选择，具体的控制信号由数据冒险检测的硬件逻辑产生。这样，可以避免数据冒险带来的停顿，提高流水线的效率。
冒险检测模块：







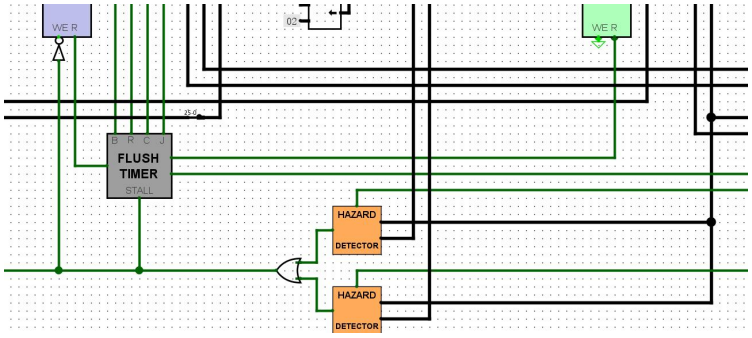
一个冒险检测器判断了在输入的两个 5 位数据相同并且输入信号 regwrite 为 1 时，输出为 1；在第四阶段，第五阶段分别对写回的地址，第二阶段 rs，rt 进行检测，来检测数据冒险；

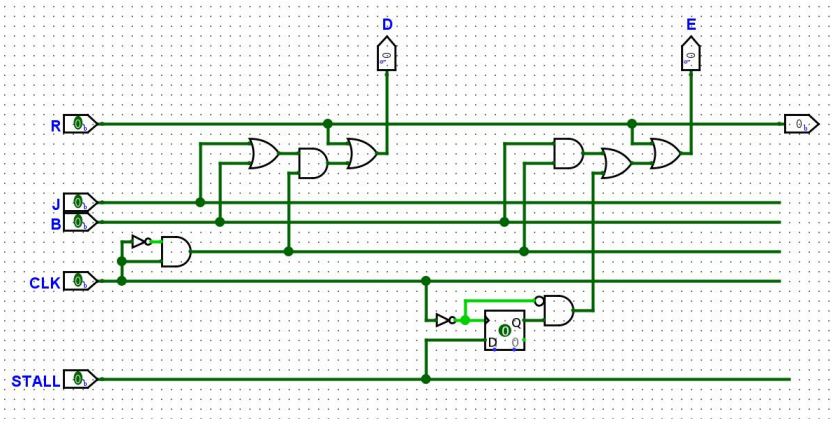
对写回地址的检测
在第四阶段，你可能在写回阶段检测当前指令是否对内存或寄存器进行写操作，如果是，就需要在第五阶段解决可能的数据冒险。
原理：
当前指令的写回阶段产生的写回地址与后续指令的读操作的寄存器地址相同，就发生了 RAW 冒险。
Regwrite=1 的情况下：
EX 的 rs 和 WB 的 WA 相同，但是与 MA 的 wa 不同时，多路选择直接选择 WB 阶段的结果返回 ALU；与 MA 的 wa 已经相同的话，多路选择直接将 MA 阶段的 aluout 返回 ALU 中。
利用数据前推解决了数据冒险。

控制冒险：
在流水线中，通过在分支发生前暂停一个周期（即插入一个气泡）是一种处理控制冒险的方法，也称为流水线停顿。

```
beq $t0, $t1, Label    # 指令 1（分支指令）
add $t2, $t3, $t4      # 指令 2（被暂停的周期，无论分支是否发生都会执行）
```

这种方法的优点是简单直观，但也有明显的缺点，即会浪费一个时钟周期。当分支发生时，插入气泡可以确保不执行错误的指令，但同时会导致性能下降。





利用冒险检测单元检测到冒险之后发出了 stall 信号。在 jump 或者 branch 信号为真时，clk 上升沿中发出 D，E 信号。当然，若是外界直接发出了 reset 信号，那么 DE 信号直接为 1；Stall 信号使得 pc 和 IF_ID 寄存器的读写使能失效，而传出的 DE 信号使得几个流水寄存器中的数据被清空。

在流水线中，通过在分支发生前暂停一个周期（即插入一个气泡）是一种处理控制冒险的方法，也称为流水线停顿。

例子：

```
beq $t0, $t1, Label    # 指令 1（分支指令）
add $t2, $t3, $t4      # 指令 2（被暂停的周期，无论分支是否发生都会执行）
```

这种方法的优点是简单直观，但也有明显的缺点，即会浪费一个时钟周期。当分支发生时，插入气泡可以确保不执行错误的指令，但同时会导致性能下降。

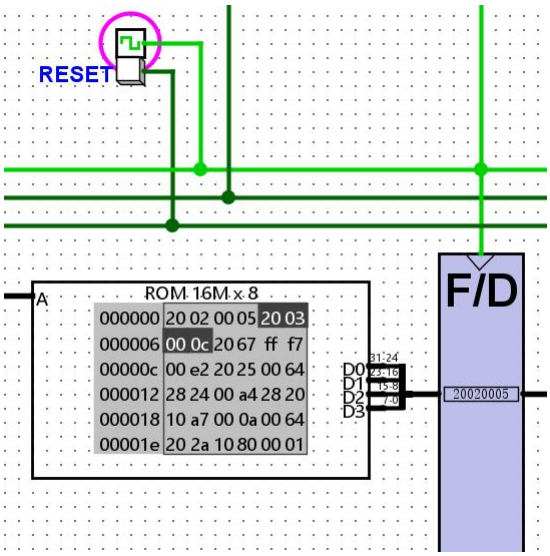
Test Instructions（测试指令）：

#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	c	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	10	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a
	beq $4, $0, around	# should be taken	20	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050
	j end	# should be taken	3c	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	44	ac020054

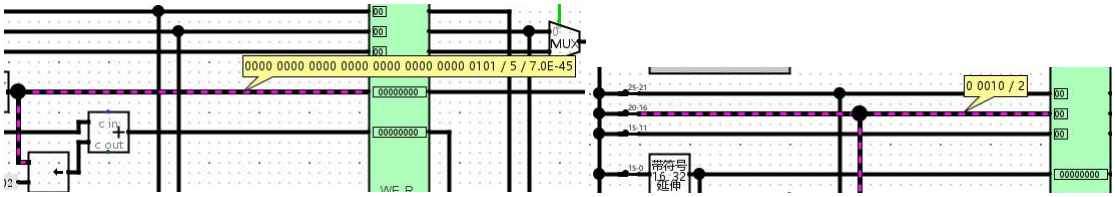
```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
```

10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
将以上指令写入 IM 中；点击 clk 进行模拟。

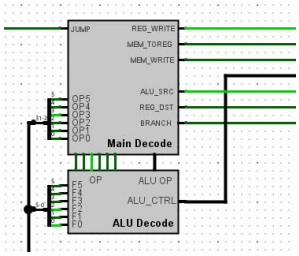
以下简要截取部分执行过程：



第一个上升沿，第一条指令传入 IF_ID 寄存器。并准备下一条指令的读取。



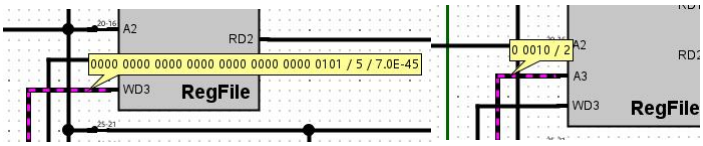
第一条指令的 addi。地址为 2，立即数为 5；



此时控制单元也发出了控制信号。
下一个上升沿时，第一条指令到了第三阶段，而第二条指令到第二阶段；

……

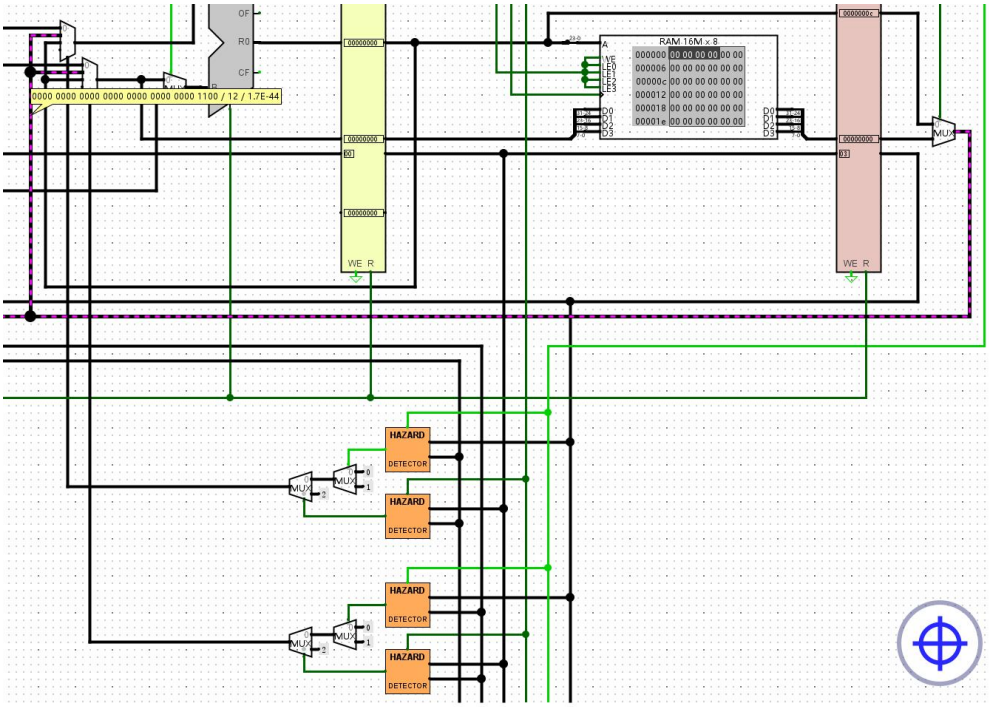
此时第一条指令的 addi，把 5 写回 2 号寄存器；



然后我们发现第二条和第三条指令发生了 RAW 数据冒险。

```
addi $3, $0, 12
addi $7, $3, -9
```

第三条要使用的 3 号寄存器的值要等待第二条指令的结果写回 3 号寄存器。



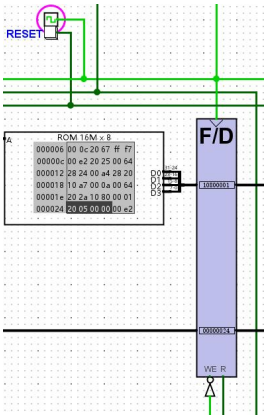
此时冒险检测直接检测到了数据冒险的发生，通过多路选择器将要写回 3 号寄存器的值直接返回到 alu 和第三条指令进行运算。成功解决了数据冒险。

……

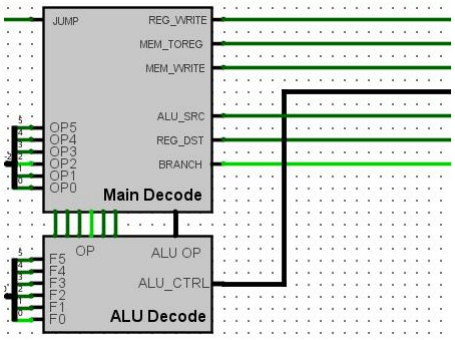
来到 beq 指令；

```
beq  $4, $0, around
```

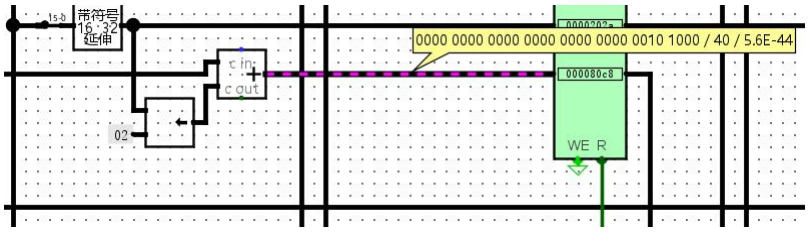
由于 4 号寄存器和 0 号寄存器的值都为 0，所以会发生跳转。



时钟上升沿到来后，beq 指令已经传入译码阶段；

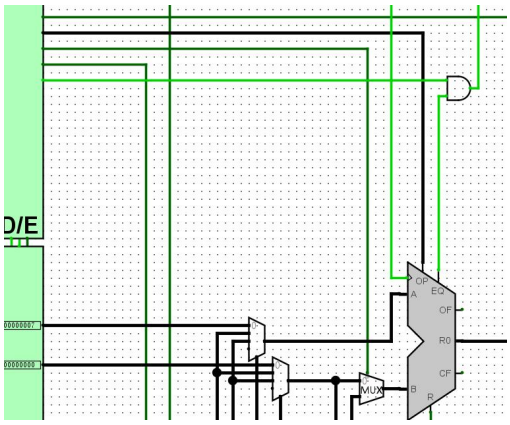


控制器输出了 branch 信号，表明这是一条分支跳转指令；

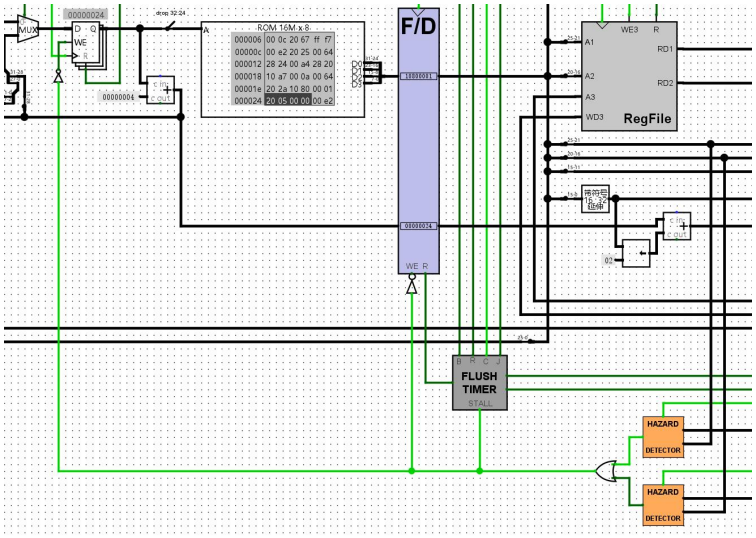


已经计算出本条 beq 指令跳转的 pc 地址；

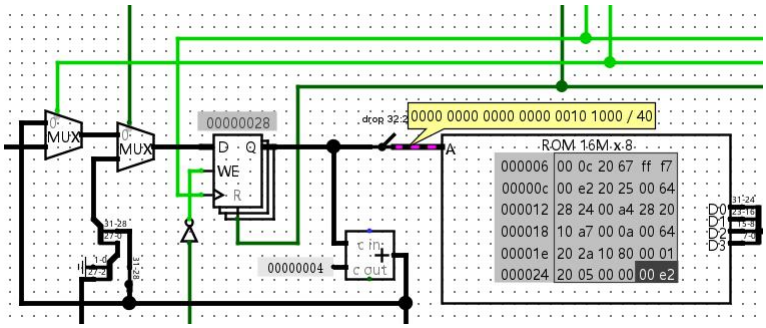
再一个上升沿，ALU 计算得出 beq 需要跳转，和 branch 信号与后输出 pcsrc 选择信号；



检测控制冒险单元检测到了冒险的发生，输出 stall 信号使得流水线停顿。



来到 pc 处，经由 pcsrc 的选择后成功使 pc 的值跳转为 40（28H），读取指令存储器的指令。

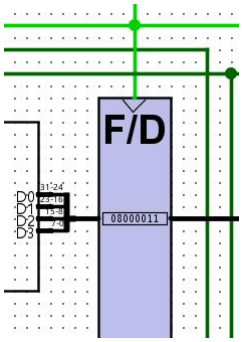


……

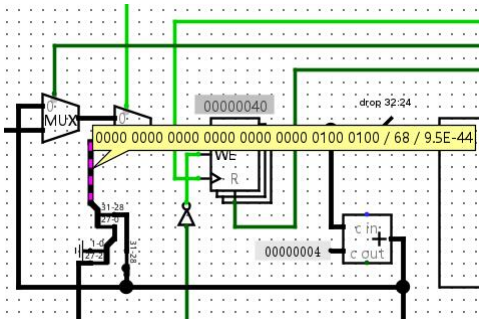
来到无条件跳转语句 jump：

```
j     end                    # should be taken            3c          08000011
```

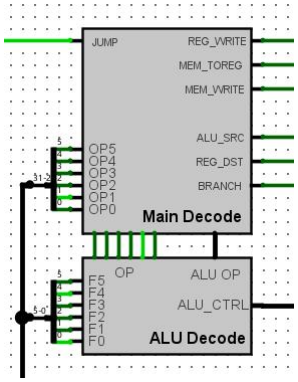
上升沿过后，jump 指令来到第二阶段。



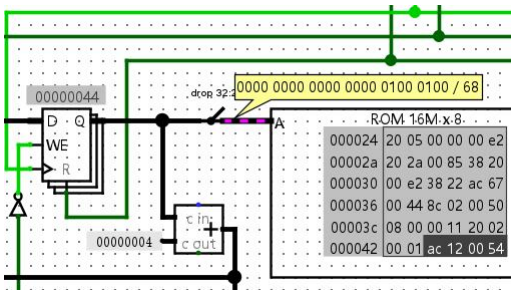
已经计算得出要跳转的 pc 地址；拼接后的 pc 值为 68（44H）



控制器也输出 jump 的选择信号



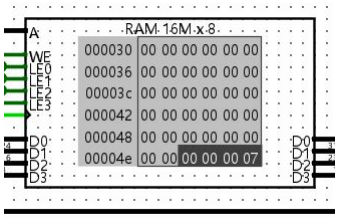
之后成功跳转到 68 地址处；



Sw：

```
sw    $7, 68($3)              # [80] = 7
```

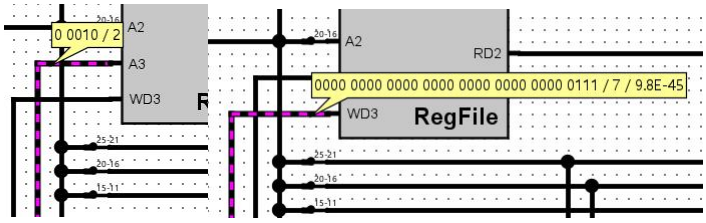
成功将 7 存入地址为 80 的数据存储器中；

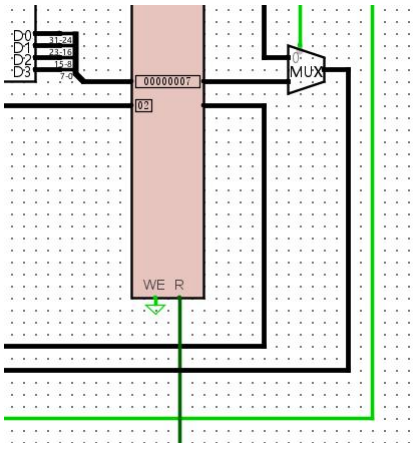


Lw：

```
lw    $2, 80($0)              # $2 = [80] = 7
```

来到取数指令，成功将上一条指令写入数据存储器的 7 读出，并写入到 2 号寄存器中；





五、调试和心得体会

在本次计算机组成原理实验中，我成功地设计并实现了一台 5 级流水 CPU，支持 MIPS 指令集中的 10 条简易指令。在这个过程中，我深入理解了计算机组成原理的核心概念，并应用这些概念进行实际的硬件设计。

最初，我对于设计一个包含 5 级流水线的 CPU 感到陌生，但通过仔细学习 CPU 架构、指令集以及冒险问题的解决方案，我逐渐掌握了相关的知识。最终，我成功地实现了一台可以运行 MIPS 指令的 CPU，并解决了控制冒险、数据冒险和结构冒险等问题。

在实验过程中，我遇到了各种挑战，包括设计流水线、解决冒险问题、调试电路等。面对这些挑战，我采用了积极的态度和团队协作的方式，通过不断的尝试和调整，逐步解决了问题。

通过实际动手设计 CPU，我不仅学到了书本上的知识，更深入理解了计算机组成原理的一些核心概念。对于流水线、指令集、冒险问题等，我有了更为直观的认识。

在实验结束后，我认识到计算机组成原理是一个广阔而深刻的领域。我希望在未来能够进一步深化对计算机硬件体系结构的理解，尝试更复杂的设计和优化，为将来的学习和研究奠定更牢固的基础。

总的来说，这次实验让我从理论到实践，更全面地认识了计算机组成原理，培养了动手解决问题的能力，也增强了团队合作的经验。这对我未来的学习和职业发展都将产生积极的影响。