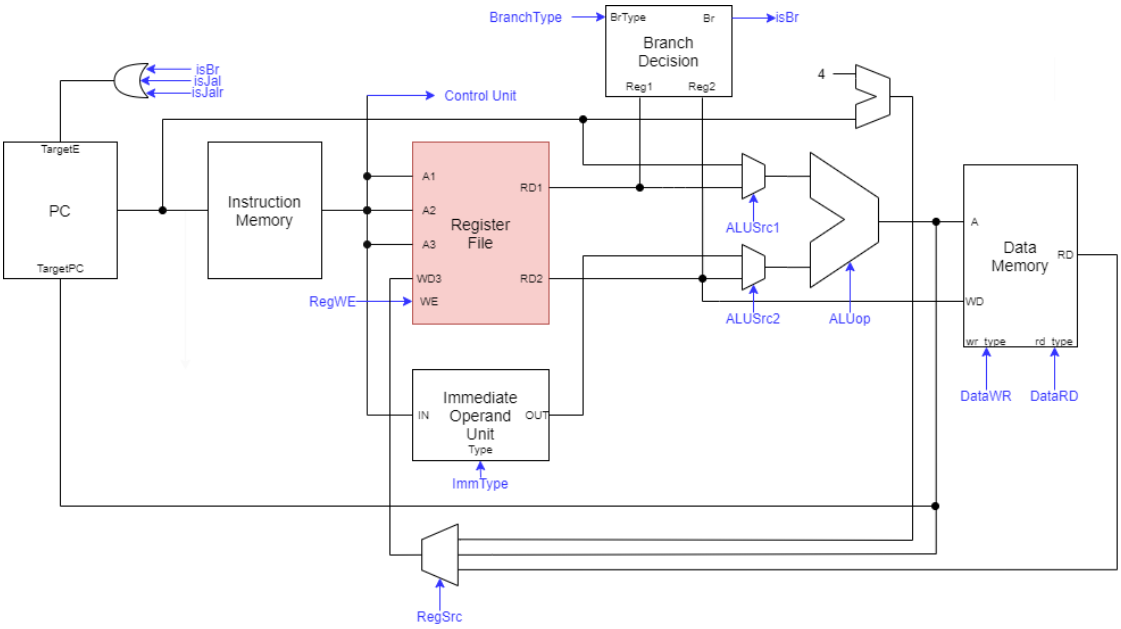
实验1 寄存器文件

一、实验目标

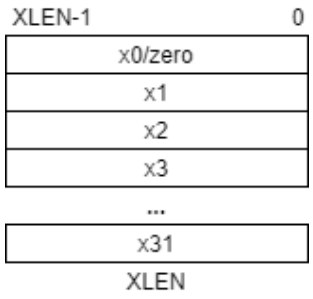
了解隧道、寄存器、多路复用器、信号分离器等元件的功能，完成寄存器模块的设计。



二、寄存器文件介绍

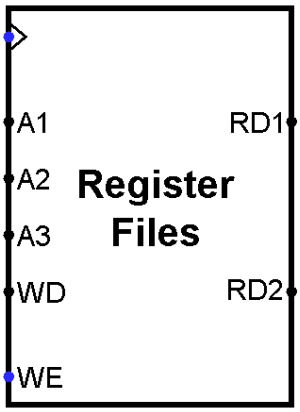
1. 通用寄存器模型

RISC-V32I的通用寄存器模型如下图所示，其中$XLEN = 32$。x_0为硬件连线的常数0，屏蔽所有写入，确保输出32位0；x_1至x_{31}为32位通用寄存器，地址为寄存器编号时，将数据写入寄存器或从寄存器读出数据。寄存器文件即为一系列寄存器的集合，具有专门的读写端口，可并发访问不同寄存器。在本系列实验中，寄存器文件指32个通用寄存器的集合。



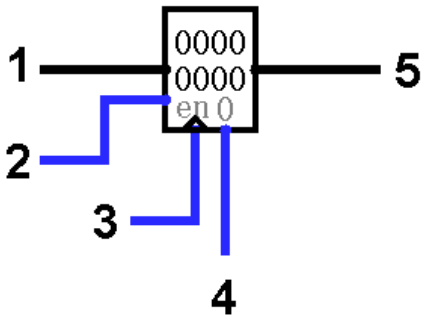
2. 端口说明

寄存器文件（Register File）为时序模块，在每个时钟上升沿写入。寄存器有1个时钟信号端口`CLK`、3个5位地址端口`A1/A2/A3`、32位写数据端口`WD`、1位写使能信号`WE`、2个32位读数据端口`RD1/RD2`。`RD1`、`RD2`分别读出`A1`、`A2`所指寄存器的值。在时钟上升沿到来时，若使能`WE`有效，则将`WD`写入`A3`所指寄存器。



3. 实现方式

寄存器文件的核心为寄存器(Register)。每个寄存器元件有5个端口，1号端口为输入；2号端口为写使能信号；3号端口为时钟信号；4号端口为异步清零信号；5号端口为输出。在每个时钟上升沿，若写使能信号有效，则将输入写入寄存器，否则寄存器内容保持不变；每当异步清零信号有效时，立即将寄存器置零。

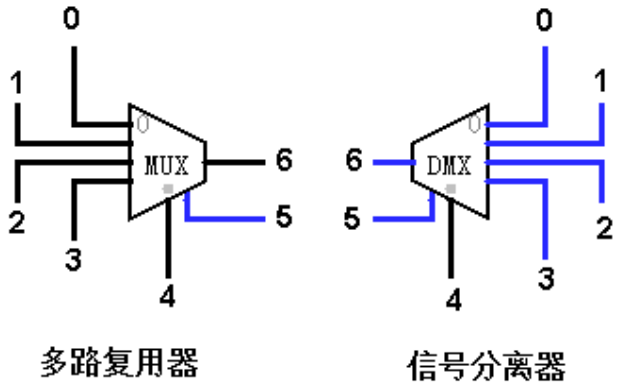


下图为该实验中使用到的两个信号处理元件，多路复用器(Multiplexer)与信号分离器(Demultiplexer)。

多路复用器的0~3号端口为输入，0号端口连接到元件上标0的位置；4号端口为控制信号，连接到元件上标点的位置；5号端口为使能信号；6号端口为输出。输入端口数量与控制信号位数相关，这里控制信号为2位，输入端口为$2^2 = 4$个。当使能信号有效或未连接时，输出端口等于控制信号表示的十进制数对应端口的输入。例如，控制信号为11即十进制数3时，输出等于3号端口的输入。

信号分离器的0~3号端口为输出，0号端口连接到元件上标0的位置；4号端口为控制信号，连接到元件上标点的位置；5号端口为使能信号；6号端口为输入。输出端口数量与控制信号位数相关，这里控制信号为2位，输出端口为$2^2 = 4$个。当使能信号有效或未连接时，控制信号表示的十进制数对应输出端口等于输入端口，其余输出均为0。例如，控制信号为10即十进制数2时，2号端口输出等于6号端口输入，其余输出端口均为0。

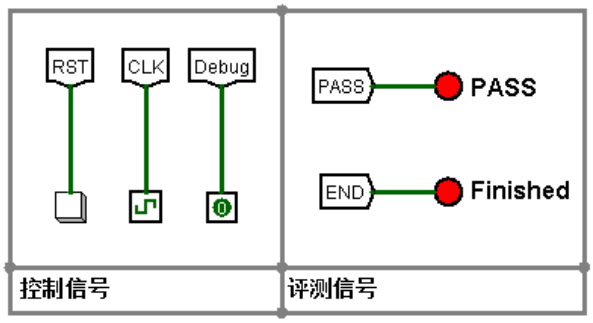
多路复用器用于在多个输入中选择一个值输出，在这里用于选择寄存器输出；信号分离器用于将一个输入映射到多个输出上，在这里用于选择使能信号。



三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹 `testcases` 下提供了测试样例源文件 `Register.testcase`。有关样例的书写方式与相关说明请参考 `/testcases/testcase.md`。

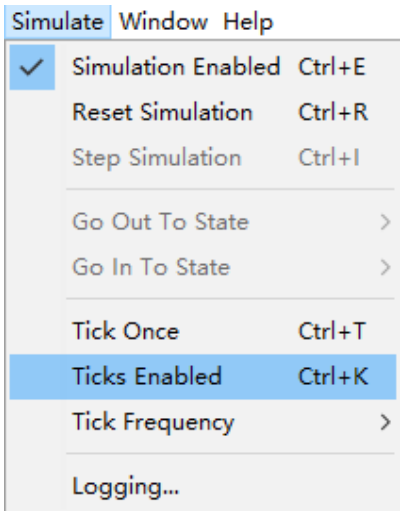
AutoTest电路有3个控制信号、至少1个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST` 信号连接到按钮，按下即可重置测试；`CLK` 信号连接到时钟；`Debug` 信号用于调试，当 `Debug` 信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished` 为完成信号，其余均为评测信号。当完成信号 `Finished` 有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了寄存器的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击 `CLK` 信号连接到的时钟。



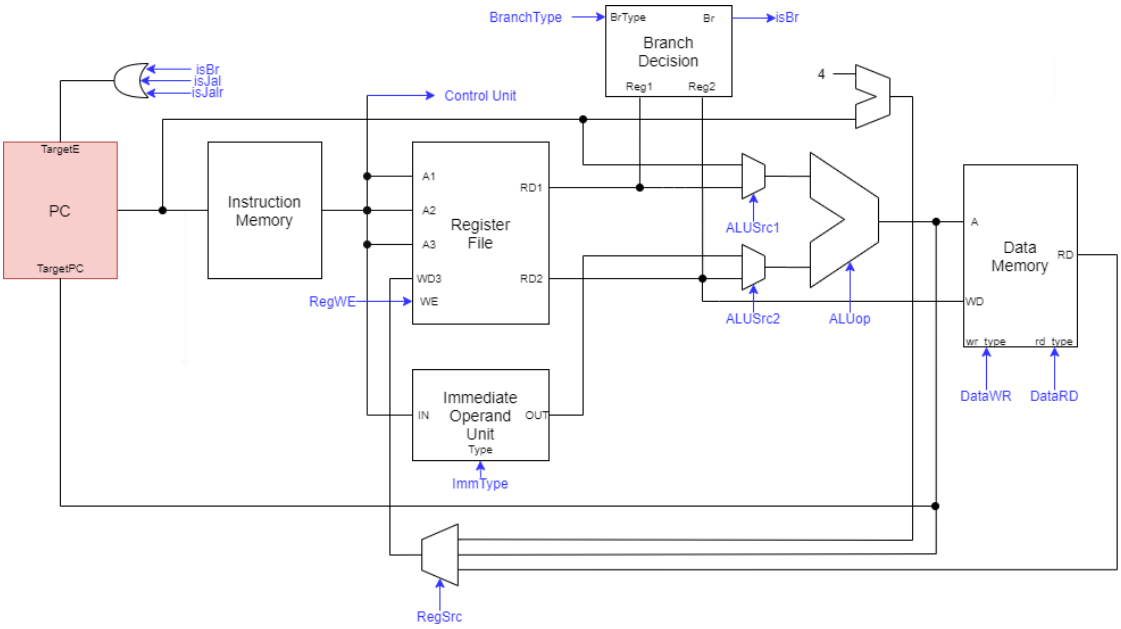
四、实验要求

在本实验中，我们将实现含有4个寄存器的简单寄存器文件。实验电路已给出输出选择电路、写使能信号分离电路以及部分寄存器电路，要求实现通用寄存器的x_0至x_3共4个寄存器。请根据介绍中的信息补全寄存器电路，并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验2 程序计数器

一、实验目标

熟悉寄存器、多路复用器等组件的功能，完成程序计数器模块的设计。



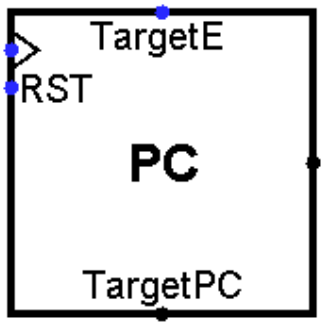
二、程序计数器介绍

1. 程序计数器

程序计数器（Program Counter, **PC**）本质上是一个寄存器，它提供了当前指令的地址。

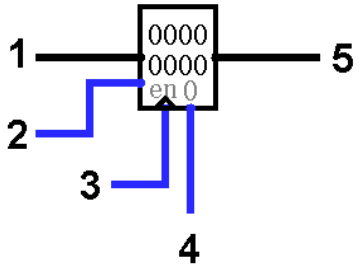
2. 端口说明

PC为时序模块，在每个时钟上升沿写入。PC有时钟信号端口与复位端口各一个、1个目标地址使能信号`TargetE`、1个32位目标地址`TargetPC`、1个32位程序地址端口`PC`。在时钟上升沿到来时，若使能`TargetE`有效，则将`TargetPC`写入寄存器，否则将当前PC增加4后写入寄存器。

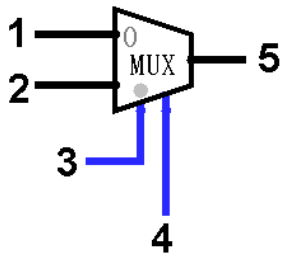


3. 实现方式

PC的主体为一个寄存器(Register)。如下图所示，寄存器有5个端口：1端口为输入；2端口为输入使能，不连接；3端口为时钟信号，连接到`CLK`；4端口为复位信号，连接到`RST`；5端口为输出，连接到`PC`。



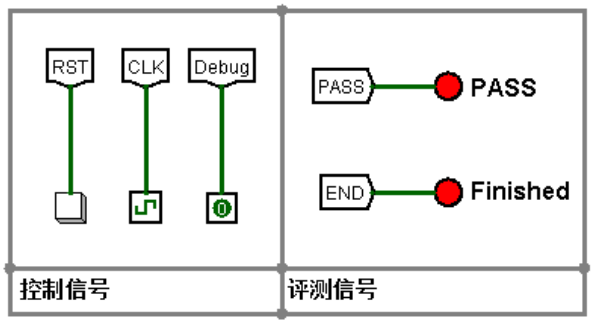
1端口接到一个多路复用器(Multiplier)的输出。如下图所示，多路复用器有5个端口：1端口为输入0，连接到`PC`+4；2端口为输入1，连接到`TargetPC`；3端口为选择信号，连接到`TargetE`；4端口为使能信号，不连接；5端口为输出。当`TargetE`信号有效时，5端口的输出为2端口的输入数据，即`TargetPC`。



三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹`testcases`下提供了测试样例源文件`/PC.testcase`。有关样例的书写方式与相关说明请参考`/testcases/testcase.md`。

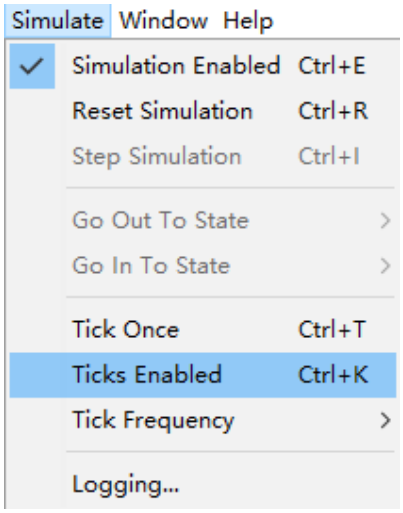
AutoTest电路有3个控制信号、至少1个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST`信号连接到按钮，按下即可重置测试；`CLK`信号连接到时钟；`Debug`信号用于调试，当`Debug`信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished`为完成信号，其余均为评测信号。当完成信号`Finished`有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了**PC**的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击`CLK`信号连接到的时钟。



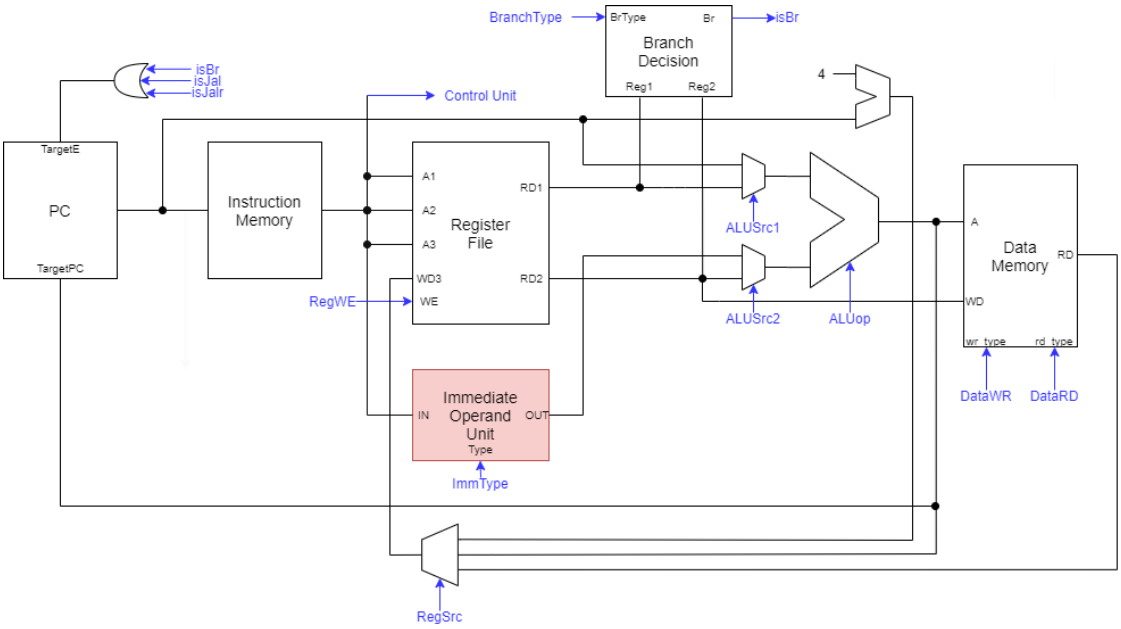
四、实验要求

在本实验中，我们将实现程序计数器。实验电路已给出部分电路，要求参考介绍中的信息补全程序计数器，并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验3 立即数生成器

一、实验目标

了解**RISC-V32I**中立即数的5种形式，熟悉分线器的功能，完成立即数生成器模块的设计。



二、立即数生成器介绍

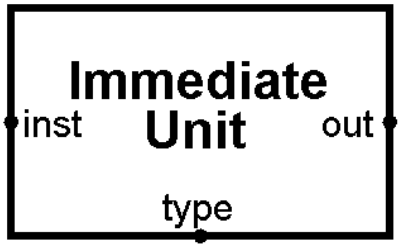
1. 立即数

立即数生成器用于解析指令中给出的立即数。**RISC-V32I**共有5种立即数格式，如下图所示。

31	30	20	19	12	11	10	5	4	1	0	
—inst[31]—						inst[30:25]		inst[24:21]		inst[20]	I立即数
—inst[31]—						inst[30:25]		inst[11:8]		inst[7]	S立即数
—inst[31]—					inst[7]	inst[30:25]		inst[11:8]		0	B立即数
inst[31]	inst[30:20]		inst[19:12]		—0—						U立即数
—inst[31]—			inst[19:12]		inst[20]	inst[30:25]		inst[24:21]		0	J立即数

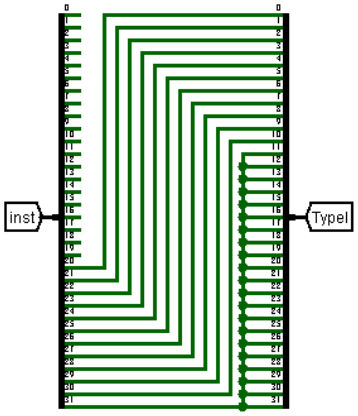
2. 端口说明

立即数生成器有3个端口，分别为32位指令输入`inst`、3位类型输入`type`、32位立即数输出`out`。该模块根据`type`端口指定的立即数类型，提取`inst`中的立即数并输出。



3. 实现方式

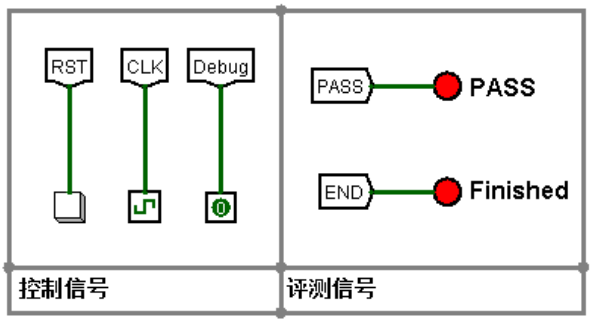
立即数生成器将`inst`按5种立即数格式分别生成一个立即数结果，再以`type`作为选择信号，使用多路复用器(Multiplier)选择出目标立即数。下图为从`inst`生成I型立即数的示例电路，左边为指令通道`inst`，右边为I型立即数通道`TypeI`，中间的连线按照I类立即数格式连接：`inst`的20至30位依次连接到`TypeI`的0至10位，`inst`的31位连接到`TypeI`的11至31位，即进行符号拓展。



三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹`testcases`下提供了测试样例源文件`Immediate.testcase`。有关样例的书写方式与相关说明请参考`/testcases/testcase.md`。

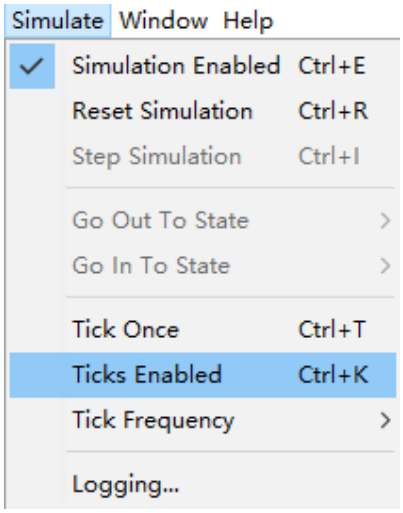
AutoTest电路有3个控制信号、5个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST`信号连接到按钮，按下即可重置测试；`CLK`信号连接到时钟；`Debug`信号用于调试，当`Debug`信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished`为完成信号，其余均为评测信号。当完成信号`Finished`有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了立即数生成器的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击`CLK`信号连接到的时钟。



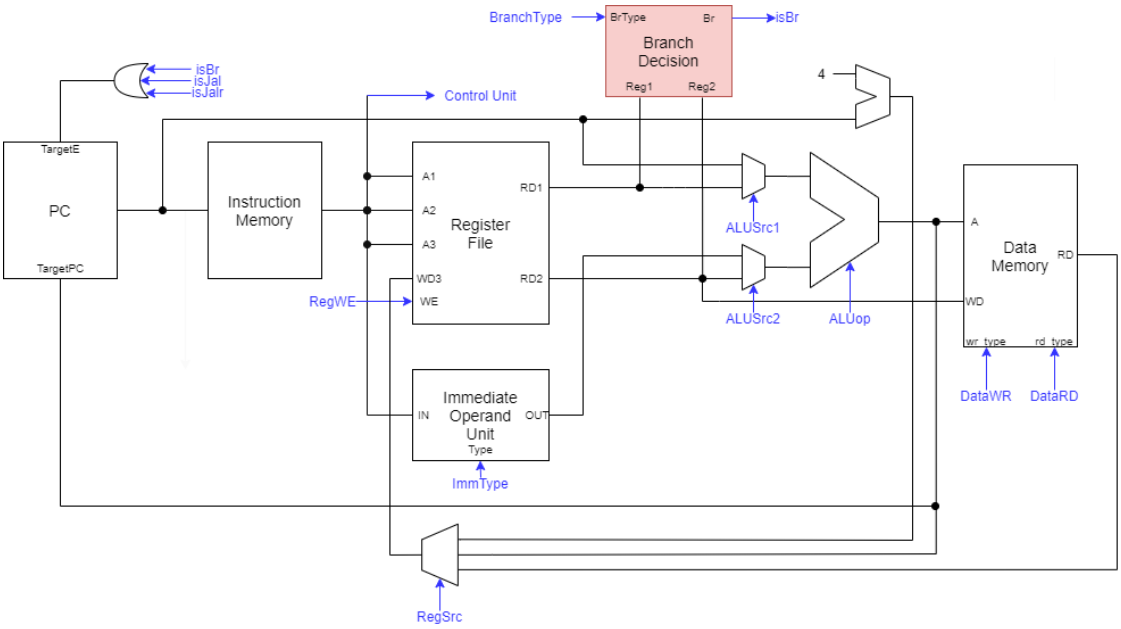
四、实验要求

在本实验中，我们将实现立即数生成器。实验电路已给出了I类立即数的生成电路以及结果选择电路，要求按介绍中的立即数格式补全S、U、B、J类型立即数电路，完成立即数生成器，并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验4 分支判断

一、实验目标

熟悉比较器的使用，设计并完成分支判断模块。



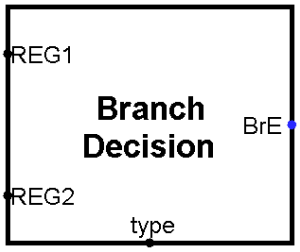
二、分支判断器介绍

1. 分支判断

在执行分支指令的过程中，**ALU**需要用于计算分支目标地址，因此需要该模块以判断是否进行分支。我们将该模块称为分支信号产生器，或分支判断器(Branch Decision)。

2. 端口说明

该模块有4个端口，分别为32位输入`REG1`和`REG2`、3位分支类型输入`type`、1位分支信号`BrE`。该模块根据`type`指示的类型，判断`REG1`和`REG2`的大小关系是否满足条件，若满足则输出`BrE`信号。

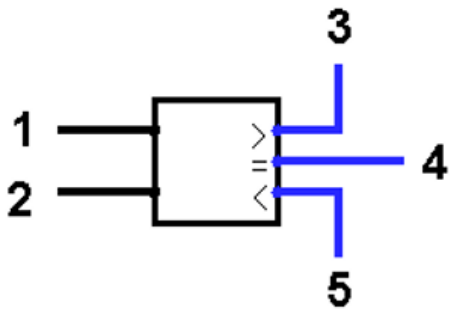


`type`与分支类型对应如下：

Type	操作	输出
010	相等判断	REG1 == REG2 ? 1 : 0
011	不等判断	REG1 != REG2 ? 1 : 0
100	小于判断	REG1 < REG2 ? 1 : 0
101	大于等于判断	REG1 >= REG2 ? 1 : 0
110	无符号小于判断	REG1 < REG2 ? 1 : 0
111	无符号大于等于判断	REG1 >= REG2 ? 1 : 0
其他	无操作	0

3. 实现方式

如下图所示，比较器(Comparator)有5个端口：端口*1*、*2*为两个输入，端口*3*、*4*、*5*为1位比较信号。任何时候，端口*3*、*4*、*5*至多只有一个信号有效。当端口*1*大于/等于/小于端口2时，端口*3*/*4*/*5*有效。比较器有两个属性，分别为输入位数与类型（有符号/无符号）



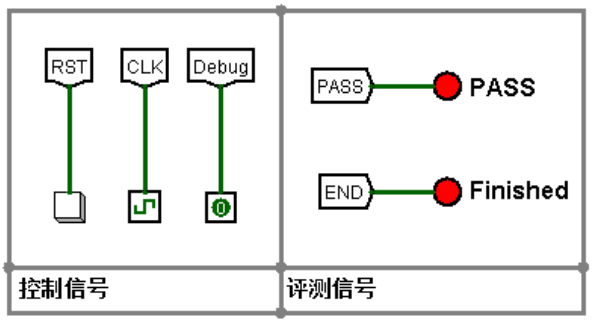
Selection: Comparator	
Data Bits	32
Numeric Type	Unsigned

实验电路已给出一个有符号比较器和一个无符号比较器。按照说明中的信号说明，将比较器输出经过组合逻辑后接到多路选择器输入上即可完成电路。

三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹 `testcases` 下提供了测试样例源文件 `Branch.testcase`。有关样例的书写方式与相关说明请参考 `/testcases/testcase.md`。

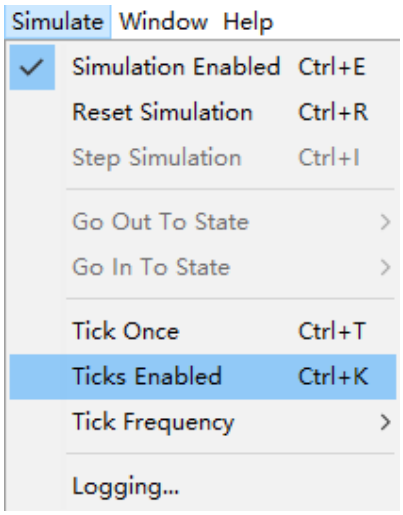
AutoTest电路有3个控制信号、7个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST` 信号连接到按钮，按下即可重置测试；`CLK` 信号连接到时钟；`Debug` 信号用于调试，当 `Debug` 信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished` 为完成信号，其余均为评测信号。当完成信号 `Finished` 有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了分支判断器的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击`CLK`信号连接到的时钟。



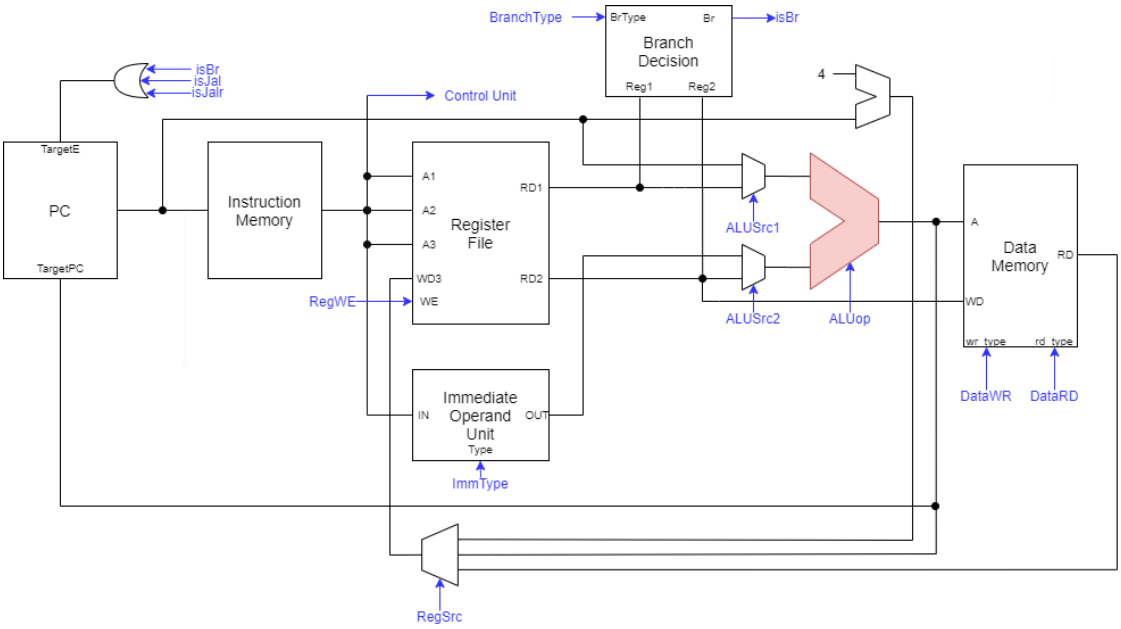
四、实验要求

在本实验中，我们将完成分支判断器，实验电路已给出了部分电路，要求按介绍中的信息补全分支判断电路，并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验5 算数逻辑单元

一、实验目标

了解加法器、移位器等运算组件的功能，完成算数逻辑单元的设计。



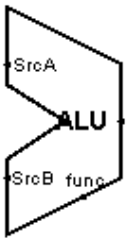
二、ALU介绍

1.ALU

算数逻辑单元（Arithmetic and Logic Unit， **ALU**）是能实现多组算术运算和逻辑运算的组合逻辑电路。

2. 端口说明

ALU有4个端口，分别为32位源操作数`SrcA`和`SrcB`、4位操作类型`func`、32位运算结果`ALUout`。**ALU**对`SrcA`和`SrcB`进行`func`所指明的操作后输出到`ALUout`。



`func`的值对应的运算类型如下表所示：

func	运算	输出
0000	加法	SrcA + SrcB
1000	减法	SrcA - SrcB
0001	逻辑左移	SrcA << SrcB[0:4]
0010	有符号小于	SrcA < SrcB ? 1 : 0
0011	无符号小于	SrcA < SrcB ? 1 : 0
0100	异或	SrcA ⊕ SrcB
0101	逻辑右移	SrcA >> SrcB[0:4]
1101	算术右移	SrcA >>> SrcB[0:4]
0110	或	SrcA || SrcB
0111	与	SrcA & SrcB
1001	无运算，输出0	0
1010	无运算，输出0	0
1011	无运算，输出0	0
1100	无运算，输出0	0
1110	无运算，输出操作数B	SrcB
1111	无运算，输出0	0

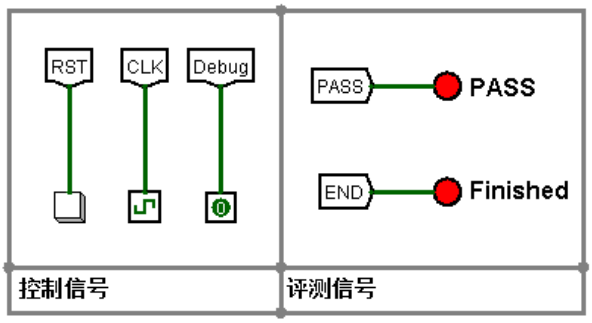
3. 实现方式

由于Logisim已经提供了大量运算元件，我们对**ALU**的设计进行了一定的简化，不需要从一位全加器一步步完成加法器的设计，而是直接使用Logisim提供的加法器(Adder)、减法器(Subtractor)、比较器(Comparator)、移位器(Shifter)元件和与门等门电路。

三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹 `testcases` 下提供了测试样例源文件 `ALU.testcase` 。有关样例的书写方式与相关说明请参考 `/testcases/testcase.md` 。

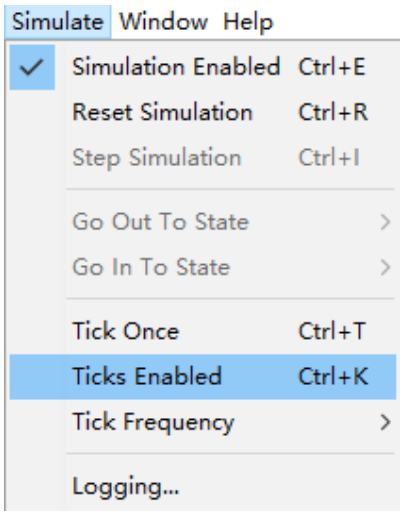
AutoTest电路有3个控制信号、10个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST` 信号连接到按钮，按下即可重置测试；`CLK` 信号连接到时钟；`Debug` 信号用于调试，当 `Debug` 信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished` 为完成信号，其余均为评测信号。当完成信号 `Finished` 有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了**ALU**的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击`CLK`信号连接到的时钟。



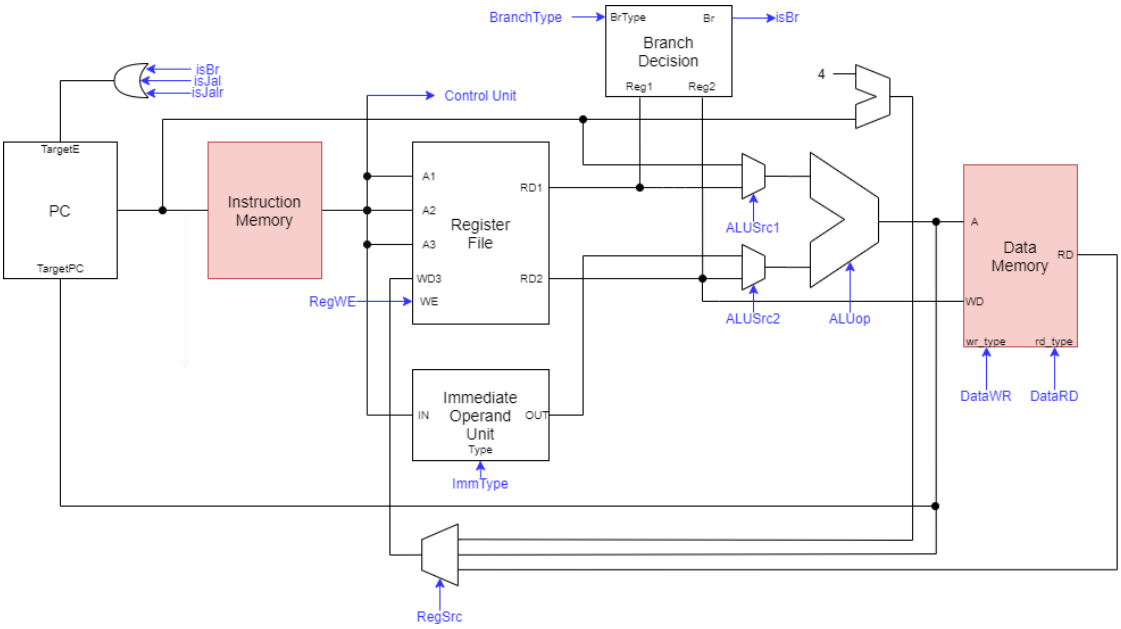
四、实验要求

在本实验中，我们将完成**ALU**。实验电路已给出了部分电路，要求选择合适的运算器补全运算电路，按介绍中的信息连接多路复用器，完成**ALU**并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验6 存储器

一、实验目标

了解ROM与RAM的功能，设计并完成指令存储器与数据存储器。



二、指令存储器介绍

1. 指令存储器

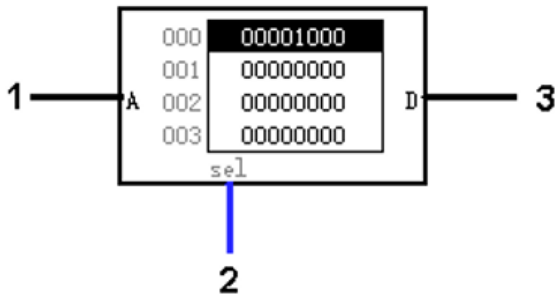
如字面意思，指令存储器用于存放指令。指令存储器按字节编址，按字读取。

2. 端口说明

指令存储器为非时序模块。该模块有一个32位地址输入`Addr`，32位指令输出`Data`。注意指令存储器的容量有限（在本实验中为16KB），因此`Addr`的有效范围为0000 0000~0003 FFFC。当`Addr`有效时，`Data`输出地址对应的指令，否则不输出值。

3. 实现方式

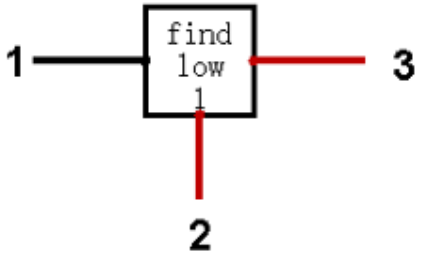
我们在指令存储器中使用ROM元件。ROM有两个参数，分别为地址位宽a与数据位宽d。每个ROM有2^a个存储单元，每个存储单元存储d位数据。下图为$a=12$、$d=32$的ROM，可以存储16KB数据，即4K条指令。ROM有3个端口：1号端口为地址端口，连接到`Addr`的2~13位；2号端口为使能信号；3号端口为数据端口，连接到输出`Data`。当使能信号有效时，ROM将地址所指的存储单元输出。



Selection: ROM	
Address Bit Width	12
Data Bit Width	32
Contents	(click to edit)

为了判断地址是否有效，我们需要判断`Addr`的0~1位与14~32位是否存在1。因此，我们需要使用比特查找器元件(Bit Finder)，该元件如下图所示可以选择数据位数与模式，模式分为最低/最高位的1/0。该元件有3个端口，1端口为2^n位输入，2端口输出1位判断信号，3端口输出n位位序值。以图中所选寻找最低位1的8位比特查找器为例，若输入中存在至少一个比特为1，则端口2输出1，端口3输出最低位1在输入中你那个的位置。

在指令存储器中，我们将`Addr`的0~1位与15~32位连接到比特查找器，将比特查找器的2端口取反后连接到ROM的2端口，即可完成对`Addr`有效性的判断。



Selection: Bit Finder	
Data Bits	8
Type	Lowest-ord...
	Lowest-order 1
	Highest-order 1
	Lowest-order 0
	Highest-order 0

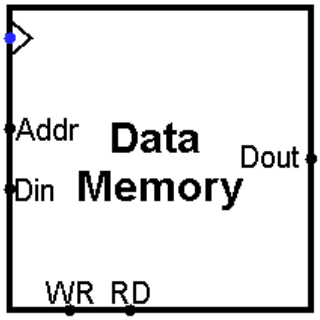
三、数据存储器介绍

1. 数据存储器

如字面意思，数据存储器用于存放数据。数据存储器按字节编址，可以完成对字、半字、字节的对齐存取。

2. 端口说明

数据存储器为时序模块。该模块有1位时钟信号`CLK`、32位地址输入`Addr`、32位数据输入`Din`、32位数据输出`Dout`、2位写信号`WR`、3位读信号`RD`。在时钟上升沿到来时，若`WR`为允许写入信号，则将`Din`根据`WR`指明的方式写入`Addr`所指的存储单元。在任何时候，该模块根据`RD`指明的方式读出`Addr`所指的存储单元到`Dout`。



`WR`与`RD`的值对应的读写方式如下表所述。

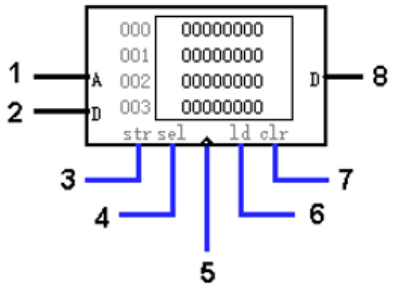
WR	操作
00	不写入
01	写入整字
10	写入半字
11	写入字节

RD	操作
000	读整字
001	读整字
010	读半字，零扩展
011	读字节，零扩展
100	读整字
101	读整字
110	读半字，符号扩展
111	读字节，符号扩展

3. 实现方式

我们在数据存储器中使用RAM元件。RAM与ROM相似，同样有参数地址位宽a与数据位宽d，数据存储方式与ROM相同。此外，RAM有三种模式，分别为同步同端口存取、异步同端口存取与存取端口分离。在本次实验中，我们使用第三种模式的RAM。

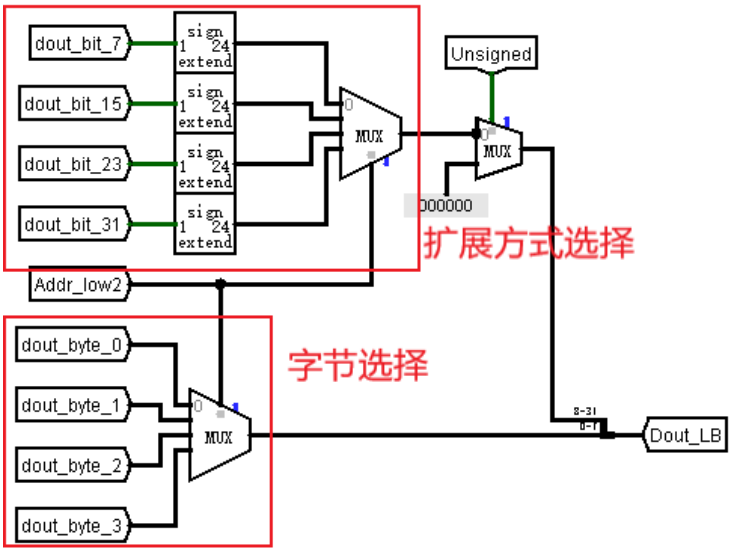
以$a=12$、$d=32$的RAM为例：1号端口为12位地址输入，连接到`Addr`的2~13位；2号端口为32位写数据输入；3号端口为存入使能，当`WR`中存在0比特时该信号为1；4号端口为RAM使能，当`Addr`的14~31位不存在1比特时该信号为1；5号端口为时钟信号，连接到`CLK`；6号端口为读取使能，不连接时默认为1；7号端口为异步清零，不连接时默认为0；8号端口为32位读数据输出。每当时钟信号上升沿到来，若RAM使能与存入使能均有效，则将写数据写入地址对应的存储单元。在任何时候，若读取使能有效，则读出地址所指存储单元的值。



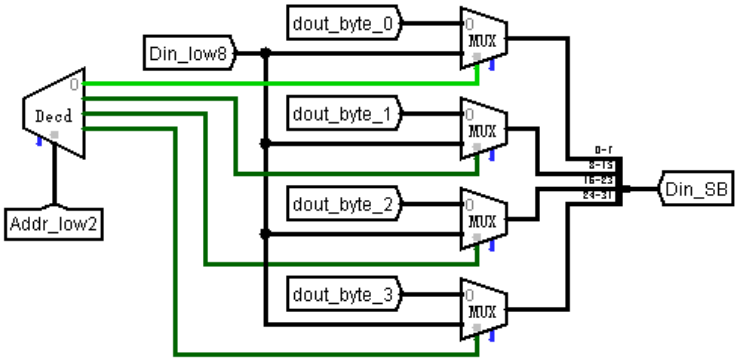
Selection: RAM	
Address Bit Width	12
Data Bit Width	32
Data Interface	Separate load and store...
	One synchronous load/store port
	One asynchronous load/store port
	Separate load and store ports

该RAM是按字存取的，为了可以完成存取半字与字节的功能，我们需要对输入与输出做一些处理。对于存入操作，我们要先读出对应存储单元的内容，将对应位置改为要存入的半字或字节，再将修改后的内容存入；对于读取操作，我们要将读出的内容截取出所需的半字或字节，再做零扩展或符号扩展。

实验电路提供了针对存取字节的输入输出处理。输出处理如下图所示。我们将RAM的输出分为4个字节，分别编号0~3，同时提取出每个字节的最高位进行符号拓展，随后分别使用`Addr`的低2位进行选择，再使用`WR`的最高位进行拓展方式的选择，最后将符号位与字节拼接得到LB输出。



输入处理如下图所示。同样的，将RAM输出分为4个字节并编号，每个字节依据`Addr`的低2位确定是否使用`Din`的最低字节进行替换，最后得到SB输入

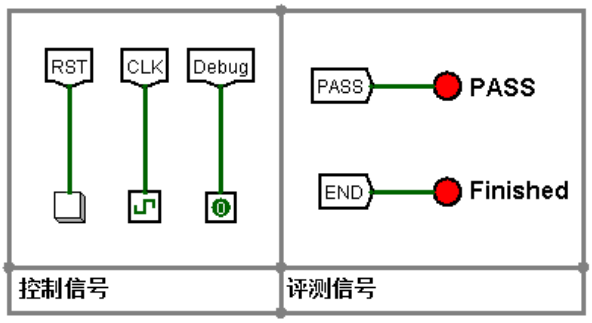


针对存取半字的电路与上述两电路一致。

四、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹`testcases`下提供了测试样例源文件`Memory.testcase`。有关样例的书写方式与相关说明请参考`/testcases/testcase.md`。

AutoTest电路有3个控制信号、1个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST`信号连接到按钮，按下即可重置测试；`CLK`信号连接到时钟；`Debug`信号用于调试，当`Debug`信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished`为完成信号，其余均为评测信号。当完成信号`Finished`有效时，若所有评测信号均有效则通过测试，否则为未通过。

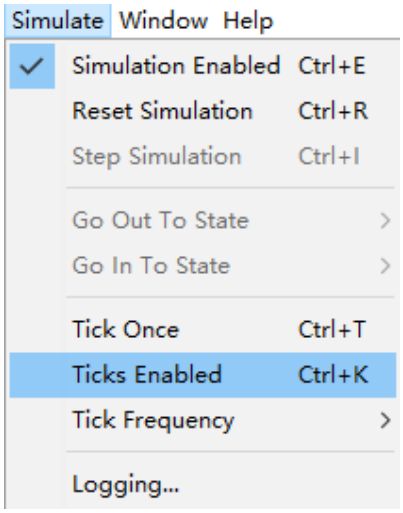


评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了数据存储器的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

由于时序模块的性质，在该时钟周期写入的值需要下一时钟周期上升沿才实际写入，因此写入正确与否需要在下一时钟周期确定。调试信号栏给出了当前样例与上一样例的信息，可以对比相邻两次操作，在上一操作为写入时可以方便的检查错误原因。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击`CLK`信号连接到的时钟。



五、实验要求

在本实验中，我们将完成指令存储器与数据存储器。实验电路已给出了部分电路，要求按介绍1中的信息补全指令存储器电路，按介绍2中的信息补全数据存储器电路，并通过数据存储器的自动测试电路检查。如对部件用途或端口有疑问，可查阅`/references`下的`Logisim部件说明.circ`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

实验7 控制器

一、实验目标

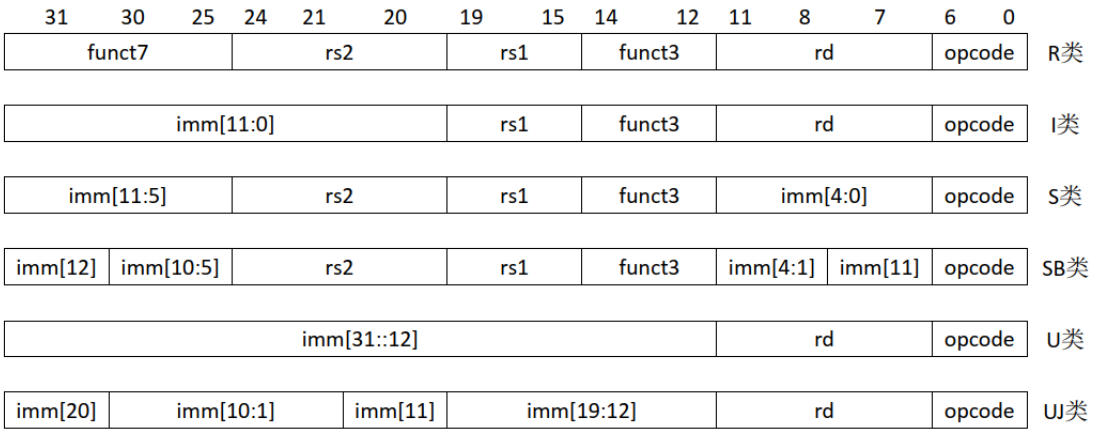
了解优先编码器的使用，学习硬布线方式设计，设计并完成控制器模块。

二、控制器介绍

1. 控制器

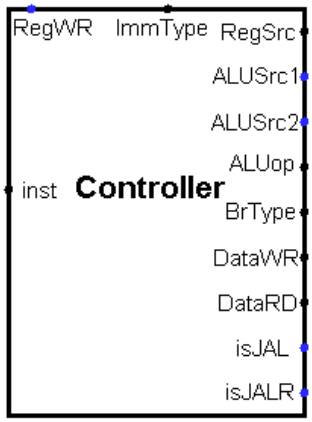
控制器（Controller），用于解析指令并输出控制信号。在本次实验中，我们采用硬布线方式设计控制器，即只使用逻辑电路生成控制信号。

RISC-V32I有6种指令类型。在控制器中需要分析的指令字段是opcode、funct3、funct7三个部分。



在本次实验中，我们依据opcode字段将**RISC-V32I**中37条非特权指令分为9类，分别为*LUI*、*AUIPC*、*JAL*、*JALR*、*Branch*(6条分支指令)、*Store*(3条写内存指令)、*Load*(5条读内存指令)、*ALUI*(9条寄存器-立即数运算指令)、*ALU*(10条寄存器-寄存器运算指令)。同类指令的控制信号相似。

2. 端口说明



控制器共有12个端口，依据其对应的模块分类说明如下。在以下说明中，`这样标记的是端口`，斜体标记的是指令类型，**加粗标记的是模块缩写。**

- 输入

  `inst`：32位指令

- 寄存器文件相关

  `RegWR`：1位使能信号，连接到寄存器文件的写使能端口。

  `RegSrc`：2位选择信号，选择寄存器文件写数据。

RegSrc	**来源**
00	`PC`+4
01	ALU输出
10	数据存储器输出

- **ALU**相关

  `ALUSrc1`：1位选择信号，选择**ALU**源操作数A的来源

  `ALUSrc2`：1位选择信号，选择**ALU**源操作数B的来源

ALUSrc	**ALUSrc1选择**	**ALUSrc2选择**
0	`PC`	立即数输出`Imm`
1	寄存器读数据`RD1`	寄存器读数据`RD2`

  不需要使用到ALU时，`ALUSrc1`和`ALUSrc2`为默认值0

  `ALUop`：4位信号，指明**ALU**的运算方式。

  - *ALU*：`ALUop`为funct7的第5位与funct3拼接而成
  - *ALUI*：当funct3为111时，`ALUop`为funct7的第5位与funct3拼接而成；否则，`ALUop`为1位0与funct3拼接而成
  - *LUI*：`ALUop`=$1110_{(2)}$
  - 其他：`ALUop`=$0000_{(2)}$

- 转移相关

  `BrType`：3位信号，指明分支指令类型。不是*Branch*类时，指定`BrType`=$000_{(2)}$；是*Branch*类时，按下表处理：

Type	操作	指令
010	相等判断	BEQ
011	不等判断	BNE
100	小于判断	BLT
101	大于等于判断	BGE
110	无符号小于判断	BLTU
111	无符号大于等于判断	BGEU
其他	无操作	无

`isJAL`：1位信号，表明指令是否为*JAL*类

`isJALR`：1位信号，表明指令是否为*JALR*类

- 立即数相关

  `ImmType`：3位信号，指明立即数类型。

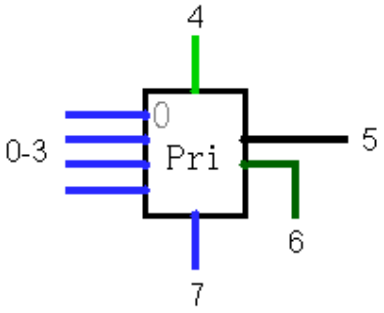
  - *JALR* / *Load* / *ALUI* / *ALU*：`ImmType`=$000_{(2)}$
  - *Store*：`ImmType`=$001_{(2)}$
  - *LUI* / *AUIPC*：`ImmType`=$010_{(2)}$
  - *Branch*：`ImmType`=$011_{(2)}$
  - *JAL*：`ImmType`=$100_{(2)}$

- 存储器相关

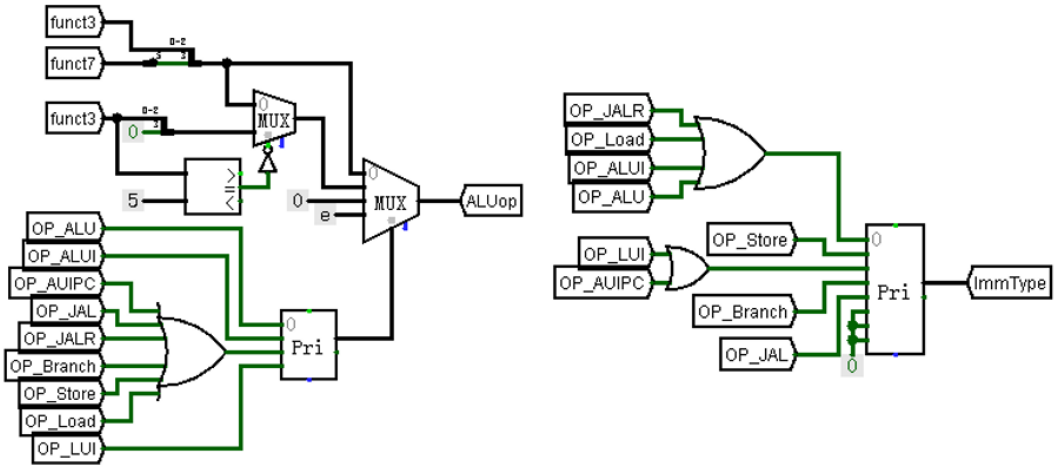
  `DataRD`：3位信号，指明数据存储器读方式。是*Load*类时，`DataRD`为funct3按位取反，否则`DataRD`=$000_{(2)}$
  `DataWR`：2位信号，指明数据存储器写方式。是*Store*类时，`DataWR`为funct3按位取反后取低2位，否则`DataWR`=$00_{(2)}$

3. 实现方式

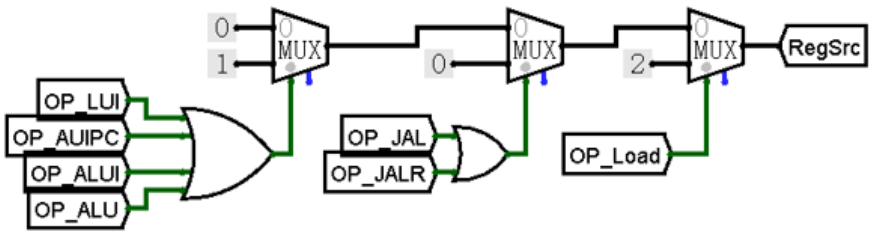
优先编码器(Priority Enoder)是一种能将多个输入压缩成更少数目输出的电路。其输出是序数0到输入最高有效位的二进制表示。下图是一个2位输出的优先编码器，其0-3号端口为1位输入，4、6号端口为一对1位选择信号，5号端口为2位输出，7号端口为使能信号。在使能信号有效时，若四个输入中有信号为1，4号端口输出0，则6号端口输出1，5号端口输出最高位1所在的位置；否则4号端口输出1，则6号端口输出0，5号端口输出0或未知。

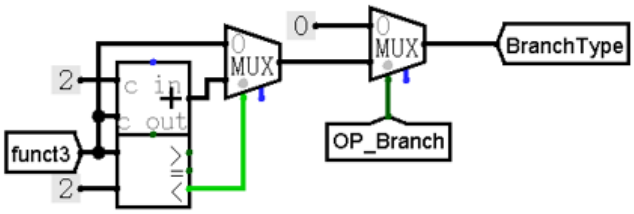


本次实验在`ALUop`与`ImmType`这两个信号上使用了优先编码器。下图左侧为`ALUop`信号，使用优先编码器输出用作多路复用器的选择信号；右侧为`ImmType`信号，优先编码器输出即为`ImmType`。



另外两个已给出的信号为`RegSrc`和`BranchType`。`RegSrc`使用连续的多路选择器完成信号选择。`BranchType`对funct3字段的值进行了处理，当funct3的十进制值小于2时，`BranchType`=$(\text{funct3})_{(2)} + 010_{(2)}$；否则，`BranchType`=funct3。



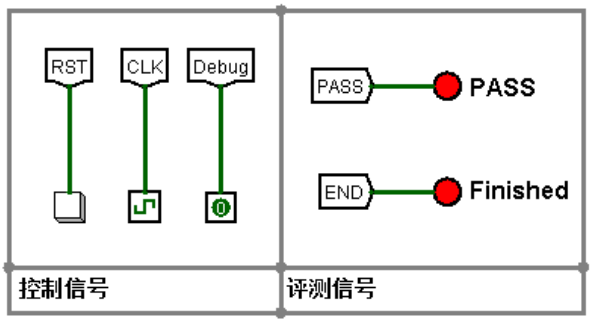


其余需要完成的信号中，`RegWR`、`ALUsrc1`、`ALUsrc2`、`isJAL`、`isJALR`5个信号使用纯组合逻辑电路生成，`DataRD`、`DataWR`使用多路选择器选择。

三、测试

实验文件提供了自动测试电路**AutoTest**与测试样例存储器**TestMemory**。测试样例已事先装载在存储器中。文件夹 `testcases` 下提供了测试样例源文件 `Controller.testcase`。有关样例的书写方式与相关说明请参考 `/testcases/testcase.md`。

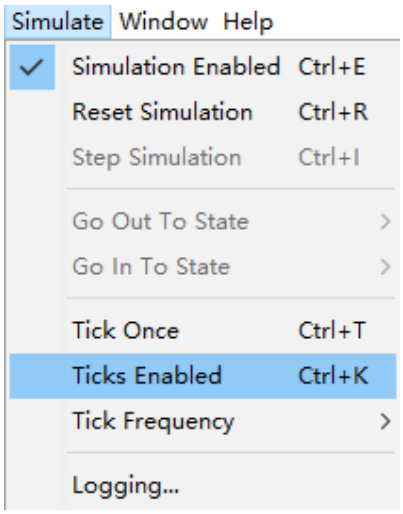
AutoTest电路有3个控制信号、9个评测信号与1个完成信号，下图为示例。控制信号一栏中，`RST` 信号连接到按钮，按下即可重置测试；`CLK` 信号连接到时钟；`Debug` 信号用于调试，当 `Debug` 信号有效时，测试电路会停在第一个错误样例。评测信号一栏中，`Finished` 为完成信号，其余均为评测信号。当完成信号 `Finished` 有效时，若所有评测信号均有效则通过测试，否则为未通过。



评测电路栏为测试的主体，由计数电路、测试电路、评测信号生成电路与测试控制电路组成。

调试信号栏给出了控制器的输入输出端口的数据，可以在这里进行调试。此外，还给出了当前样例在测试样例源文件中的行号，可以在源文件中查看具体样例。

需要测试时，在下拉菜单*Simulate*中勾选*Simulation Enabled*启用模拟，随后选择*Tick Enabled*启用时钟信号，即可开始自动测试。可以通过*Tick Frequency*调整时钟频率以加快测试速度。此外，也可以手动点击 `CLK` 信号连接到的时钟。



四、实验要求

在本实验中，我们将完成控制器。实验已给出了 `RegSrc`、`ALUop`、`ImmType` 和 `BranchType` 4个信号的生成电路，请要求按介绍中的信息完成其余7个信号的生成电路，并通过自动测试电路检查。如对部件用途或端口有疑问，可查阅 `/references` 下的 `Logisim部件说明.circ`；对信号有疑问，可查阅 `/references` 下的 `RISC-V32I单周期部件说明.md`。注意**不要**修改输入输出端口、封装外观和自动测试电路。

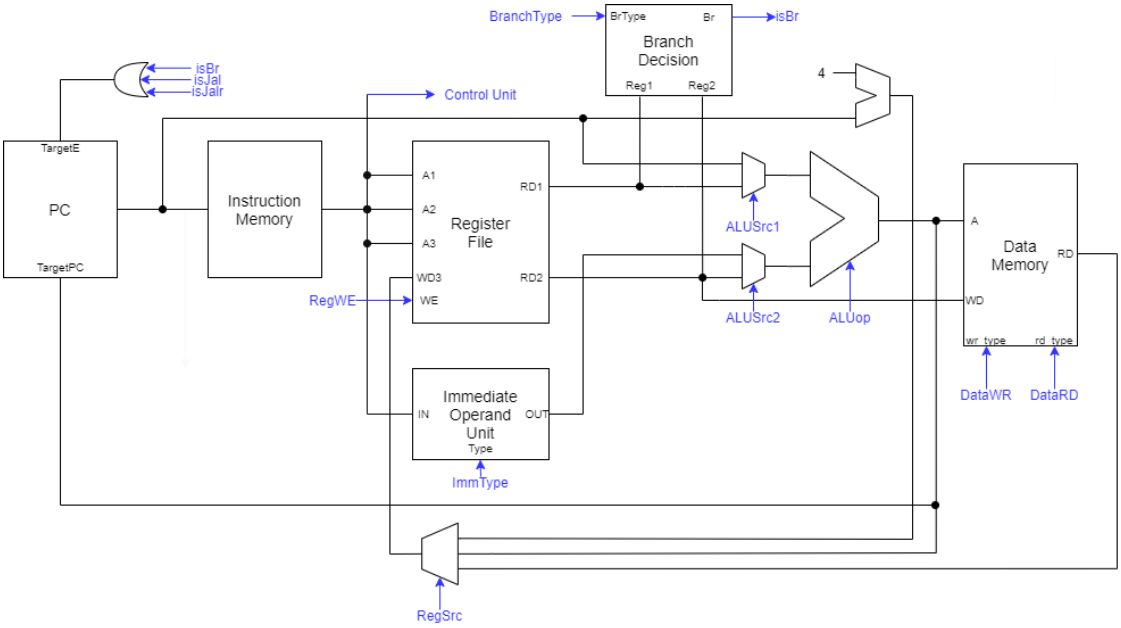
实验8 单周期CPU

一、实验目标

结合前面实现的模块，根据数据通路，完成单周期CPU。

二、CPU介绍

本次实验所设计的CPU为使用专用数据通路的单周期CPU，在每个时钟周期执行一条指令。下图为数据通路。



我们在前面7个实验已经实现了寄存器文件、程序计数器、立即数、分支判断器、ALU、数据及指令存储器、控制器共8个模块。将这8个模块根据数据通路连接即可得到单周期CPU。

在本次实验中，已给出包含全部32个通用寄存器的寄存器模块，其余模块使用你在前面实验中完成的内容，模块通过Logisim库的形式提前加载到文件中，请确保本次实验的电路与前面实验的电路在同一文件夹下。

实验电路已给出部分数据通路，需要你完成的部分包括：

- **ALU**的输入信号
- 分支判断器的输入信号
- 寄存器文件的写回数据`WD`的选择
- 数据存储器的输入信号

三、测试

1. 测试集

本次实验使用汇编文件作为测试源文件。在文件夹`/testcases/Assembly`下提供了原始汇编文件`LogicalTest.S`、该文件编译后整理成的Logisim存储器映像文件`LogicalTest_data`和`LogicalTest_inst`、该文件编译并反汇编生成的代码说明`LogicalTest.txt`。`LogicalTest.S`文件由riscv-tests项目改编而来。

该测试集有322项测试，主要测试各个部件的逻辑功能。每项测试进行一系列操作：

- 将测试序号加载到寄存器`gp`
- 测试，将结果加载到寄存器`t5`，如指令(0001 00c4)~(0001 00cc)
- 加载预期结果到寄存器`t4`，如指令(0001 00d0)
- 比较，与预期相符则进入下一测试，否则进入*fail* (0001 25a8)

```
000100c4 <test_1>:
   100c4:   00000093            li  ra,0
   100c8:   00000113            li  sp,0
   100cc:   00208f33            add t5,ra,sp
   100d0:   00000e93            li  t4,0
   100d4:   00100193            li  gp,1
   100d8:   01df0463            beq t5,t4,100e0 <test_2>
   100dc:   4cc0206f            j   125a8 <fail>
```

若全部测试均成功，PC将转移到*pass* (0001 25ac)，将1加载到寄存器`gp`，随后周期性跳转到*success* (0001 25b0)。

Jal与Jalr指令的测试没有测试序号。

测试文件为位置无关代码。

测试集中寄存器名称对应如下。`x1`到`x31`的所有寄存器均可使用，不需要考虑其功能描述。`f0`到`f31`为浮点运算寄存器，本次实验不会用到。

寄存器	ABI名字	描述	保存者
x0	zero（零）	硬件连线0	—
x1	ra	返回地址	调用者
x2	sp	栈指针	被调用者
x3	gp	全局指针	—
x4	tp	线程指针	—
x5-7	t0-2	临时变量	调用者
x8	s0/fp	保存的寄存器/帧指针	被调用者
x9	s1	保存的寄存器	被调用者
x10-11	a0-1	函数参数/返回值	调用者
x12-17	a2-7	函数参数	调用者
x18-27	s2-11	保存的寄存器	被调用者
x28-31	t3-6	临时变量	调用者
f0-7	ft0-7	FP临时变量	调用者
f8-9	fs0-1	FP保存的寄存器	被调用者
f10-11	fa0-1	FP参数/返回值	调用者
f12-17	fa2-7	FP参数	调用者
f18-27	fs2-11	FP保存的寄存器	被调用者
f28-31	ft8-11	FP临时变量	调用者

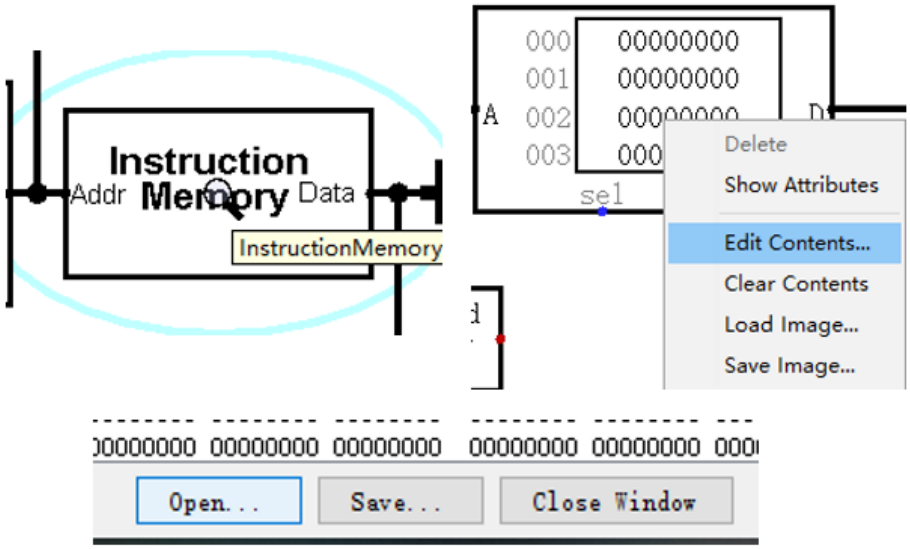
表 20.2：RISC-V 调用约定寄存器使用情况

如果你对**RISC-V**汇编有兴趣，你也可以自己编写测试文件。在文件夹`/scripts`下提供了将汇编文件编译并转换为Logisim存储器映像文件的脚本`get_bin.py`，该脚本需搭配`/scripts/utils`下以`riscv32-unknown-elf-`为前缀的4个文件。运行方法请查阅`/testcases/testcase.md`。

此外，你也可以自行搭建**RISC-V**工具链，自己编写C/C++或汇编测试程序并编译。可以参考文档中的`/references`下的`RISCV-toolchain搭建教程.md`。

2. 测试方法

在连接了所有接线后，在**CPU**电路图下，使用手型工具双击**InstructionMemory**，右键点击ROM，选择Edit Content打开数据面板，在打开的窗口内选择Open，随后选择`LogicalTest_inst`，看到数据改变后即可关闭窗口。



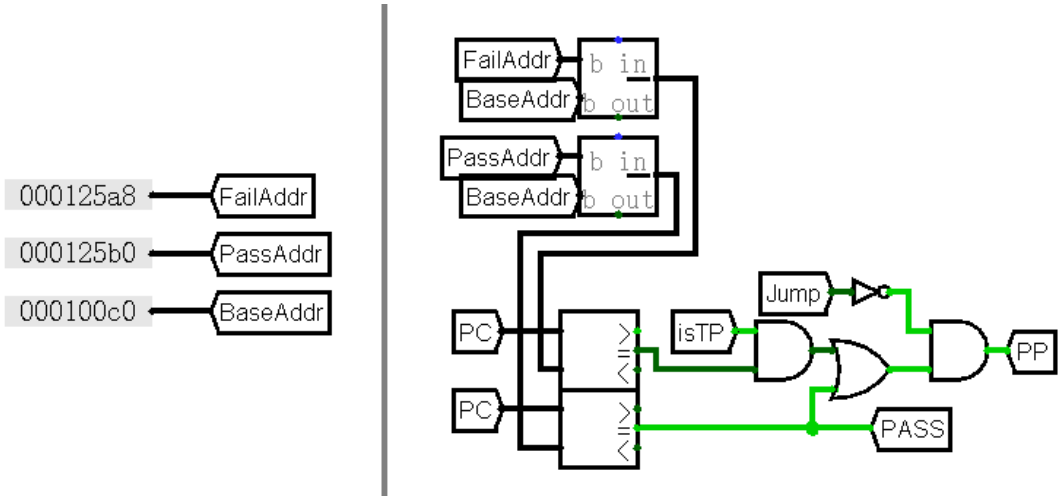
随后，在**CPU**电路图下，使用手型工具双击**DataMemory**，以同样方式将`LogicalTest_data`导入到RAM中。

指令和数据导入完成后即可进行测试。与前面几次实验相同，在下拉菜单栏中选择Tick Enabled开启时钟信号跳转即可。由于测试集较大，测试需要一定时间，建议将Tick Frequency调整至128Hz或更高。

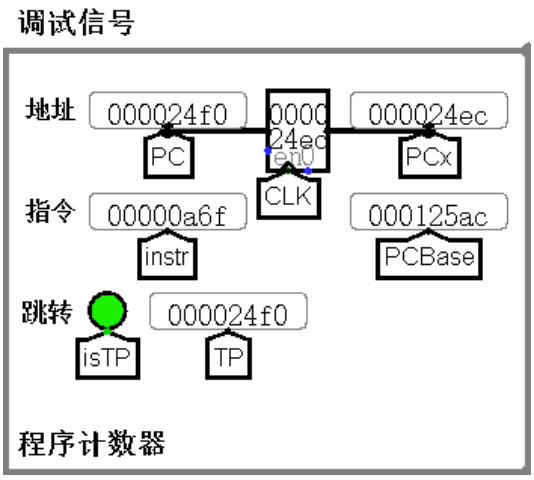
3. 信号与调试方法说明

实验电路中给出了大量调试信号，但是，若前面7个实验正确完成且数据通路正确连接，本次实验将不会使用到这些调试信号。

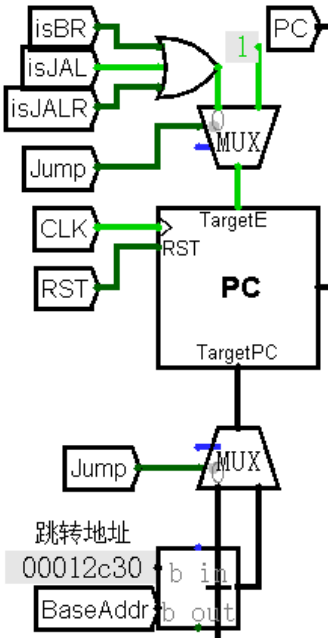
`FailAddr`与`PassAddr`分别指测试集中说明的*fail*与*success*的地址。当PC到达`FailAddr`时，下图右边的电路会给出暂停信号`PP`；当PC到达`PassAddr`时，右侧电路会给出成功信号`PASS`。`BaseAddr`指测试程序的基地址，即`_start`地址，也即第一条nop指令的地址。



调试信号一栏给出了各个模块的主要信号，这里仅对PC部分进行说明。如下图所示，地址一栏左侧的`PC`指当前指令地址，右侧的`PCx`指上一条指令地址，`PCBase`指`PCx`在`LogicalTest.txt`的实际地址。指令一栏为`PC`对应的指令。跳转一栏给出了跳转信号`isTP`与跳转目标地址`TP`。



在调试时可能需要跳转到特定地址，这里给出了一个简单电路用于跳转。在跳转地址位置输入目标地址，在控制信号中将`Jump`信号设为1，双击`CLK`信号，即可将目标地址写入PC。



四、实验要求

本次实验已给出部分电路。请按介绍中的数据通路连接好CPU，并按测试中的方法或自己设计测试电路完成测试。注意**不要**修改寄存器文件模块、引用包。

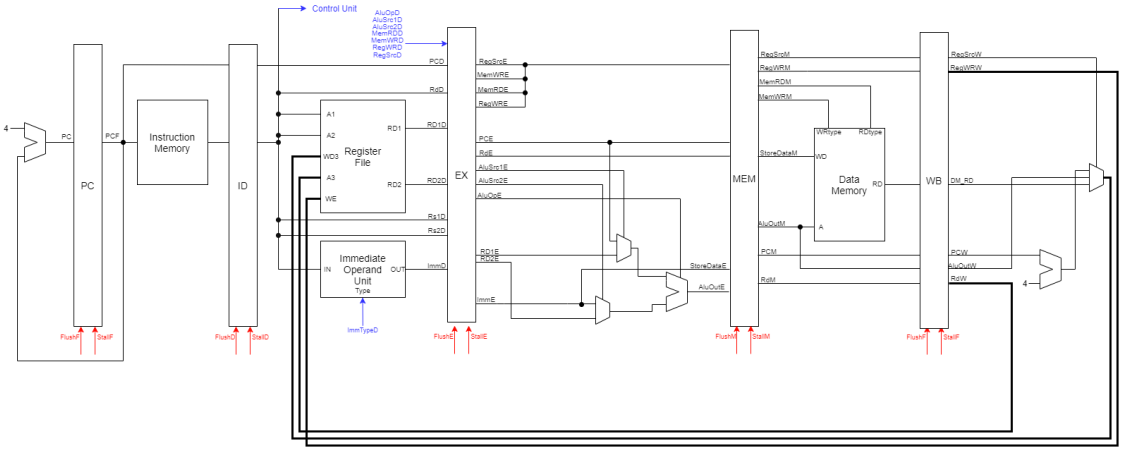
实验九 流水线1

一、实验目标

了解段间寄存器的工作方式，完成程序计数器模块，完成理想流水线的连线。

二、理想流水线

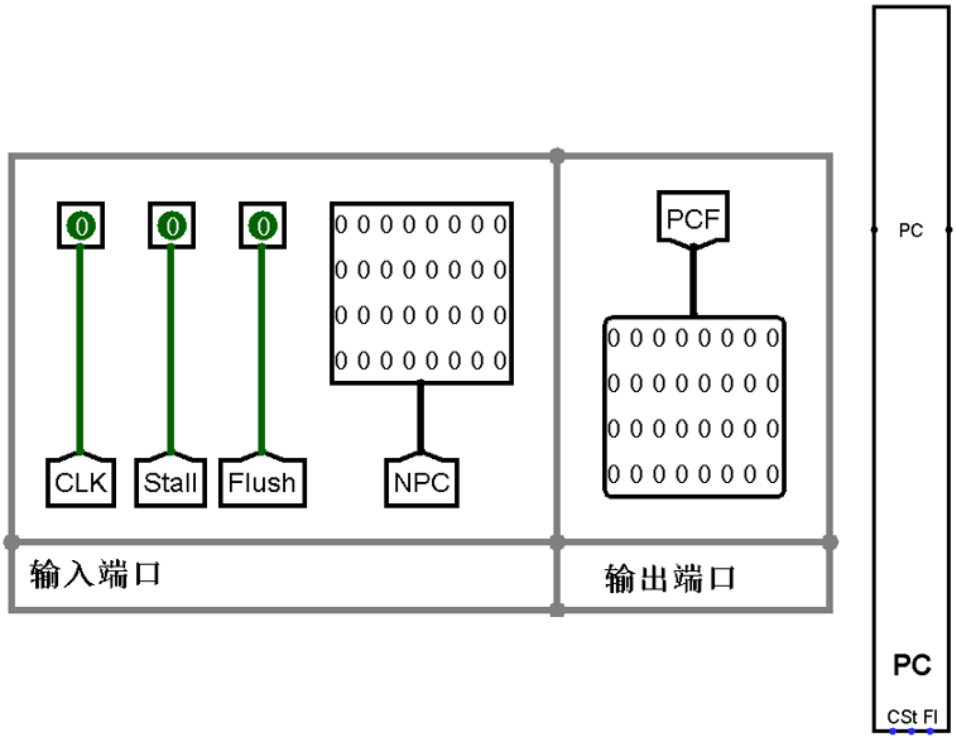
我们将一条指令的执行分为取指IF、译码ID、执行EX、访存MEM、写回WB共5个阶段，每个阶段作为一个流水段构建5段流水线，段间采用寄存器来保存指令执行的数据与信号。本次实验构建的流水线可以运行没有数据冒险、没有分支跳转指令的程序，其数据通路如下图所示。图片位于`/figs/exp9_DataPath.png`。



三、模块介绍

1. 程序计数器

在流水线系列的实验中，我们对**PC**进行了一定的更改以满足实际需要。本次实验所实现的**PC**模块实际上是对寄存器组件的一个封装，由于**PC**和段间寄存器具有相同的工作方式，因此我们进行了与段间寄存器相同的封装。其外观与端口如下图所示：

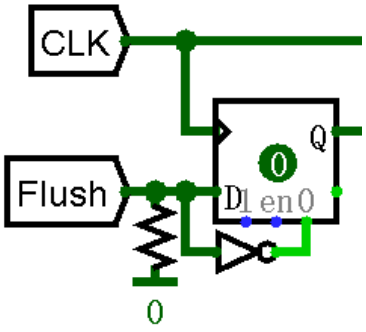


`CLK`为1位时钟信号，`FlushF`为1位同步清空信号，`StallF`为1位停顿信号，`NPC`与`PCF`均为32位，分别为下一指令地址、当前指令地址。在`CLK`到达上升沿时，若`FlushF`有效则将寄存器数据置零；当`StallF`有效时，寄存器数据不会更新。具体实现方式可以参考段间寄存器。

2. 段间寄存器

在本次流水线实验中，我们使用段间寄存器来存储各流水段需要传递的信号与数据。每个段间寄存器除数据与控制信号输入输出外，还接受`CLK`、`Flush`、`Stall`三个信号。`Flush`为同步清空信号，在`CLK`到达上升沿时，若`Flush`有效则将寄存器数据置零；`Stall`为停顿信号，当其有效时，寄存器数据不会更新。

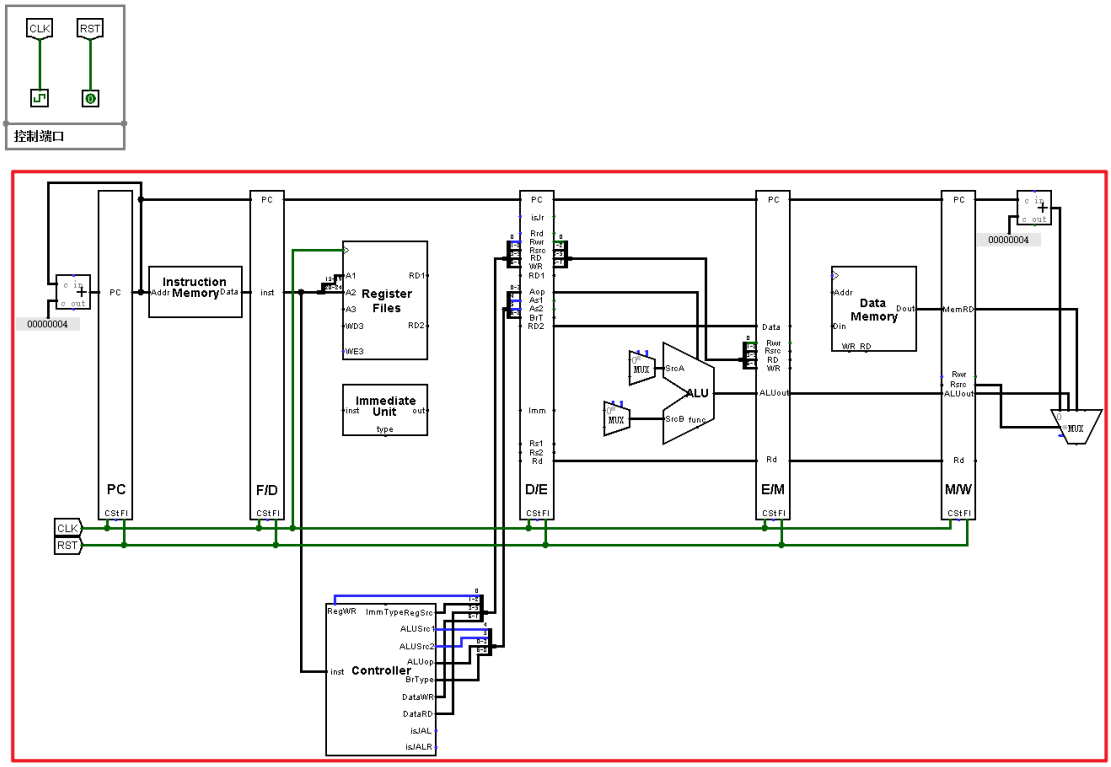
Logisim提供的寄存器组件有异步清空端口，但我们要进行同步清空。因此，这里采用D触发器构建的电路处理Flush信号，该电路在Flush有效时，在时钟上升沿附近提供一个高电平脉冲信号，完成寄存器的清空且不影响下一次写入。



四、测试

本次实验使用的测试集为 `IdealPipeline.S`。该汇编文件与编译后的Logisim存储器映像文件均在文件夹 `testcases/Assembly` 中。**测试集中用注释写出了每条非nop指令执行后，目标寄存器或内存中应有的值。**

需要测试时，使用手型工具双击指令存储器模块，在ROM中导入存储器映像 `IdealPipeline_inst`，随后启用时钟信号即可。在测试完成后，寄存器x1中的值应为$0\mathrm{x}FFFFFE80$。如需要进一步调试，可以将电路**CPU(TODO)**中所有电路（即下图红框部分）直接复制到电路**CPU(Debug)**中，所有重要信号的探针将会直接连接到电路中。



说明：如果需要查看电路中某个模块的内部电路状态，需要使用手型工具双击该模块。这与你在向存储器中加入指令和数据的动作是一样的。

五、实验内容

本次实验已给出的部分：

- **ALU**、立即数生成器、指令存储器、数据存储器、控制器模块的实例。这5个模块采用你在单周期实验中完成的部分，通过Logisim包的形式加载到本次实验的电路中。
- 段间寄存器、寄存器文件的端口、电路与封装形式
- **PC**的端口、封装形式
- 在理想流水线电路中，各个模块的实例与部分连线

需要你完成的部分：

- **PC**的电路
- 理想流水线在ID、EX、MEM、WB段的连线

在完成以上内容后，请自行完成测试。

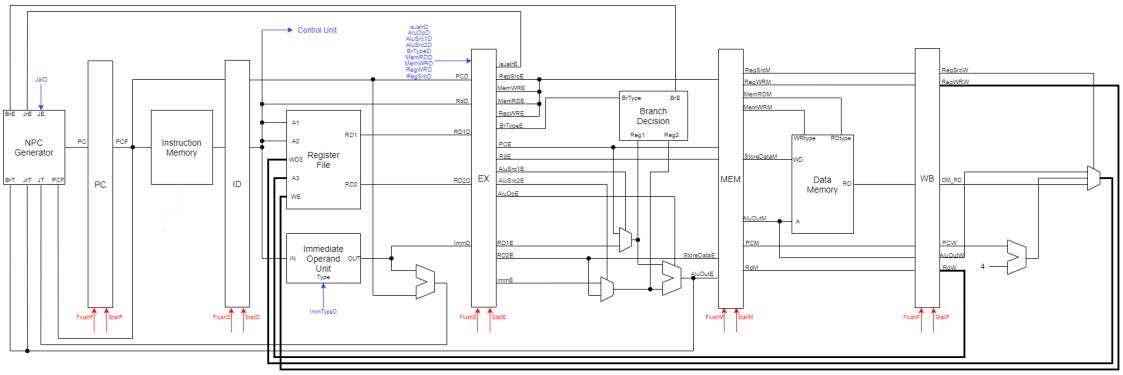
实验十 流水线2

一、实验目标

完成冒险检测模块，完成气泡流水线的连线。

二、气泡流水线

在理想流水线的基础上，我们加入了转移类指令，并加上了数据相关的检测，启用了段间寄存器的`Stall`与`Flush`信号。存在数据相关或分支跳转时，我们提供合适的`Stall`与`Flush`信号使得指令可以正确执行。这样会使得有数据相关的指令执行起来像中间插入了`nop`指令一样。其数据通路如下图所示。图片位于`/figs/exp10_DataPath.png`。

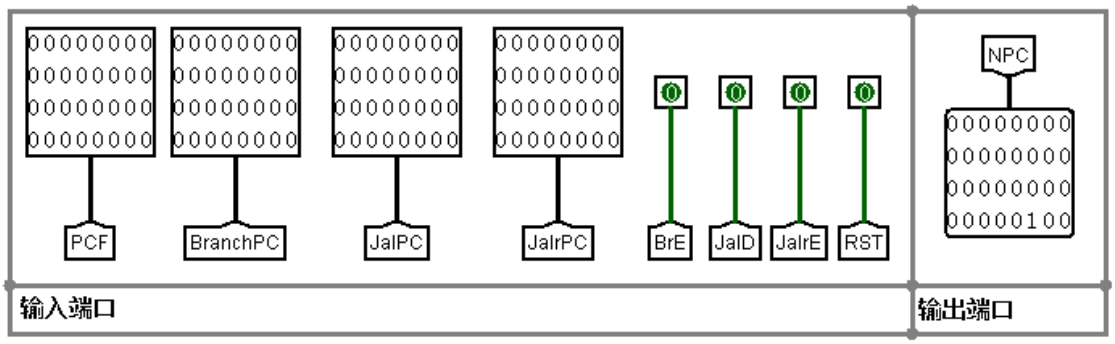


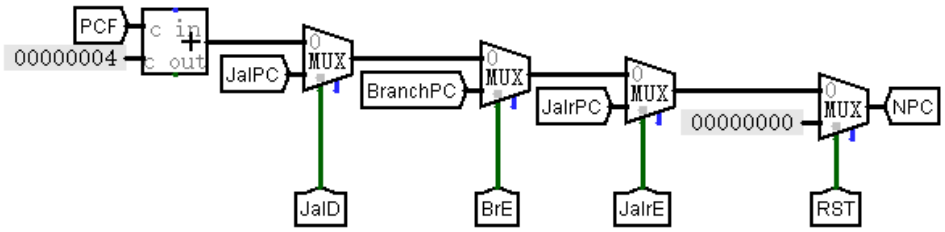
在该流水线中，我们在ID段确认直接跳转指令JAL的目标地址，在EX段确认相对跳转指令JALR、全部分支指令的目标地址。因此，在ID段需要额外使用一个加法器以给出目标地址。

三、模块介绍

1. 地址生成器

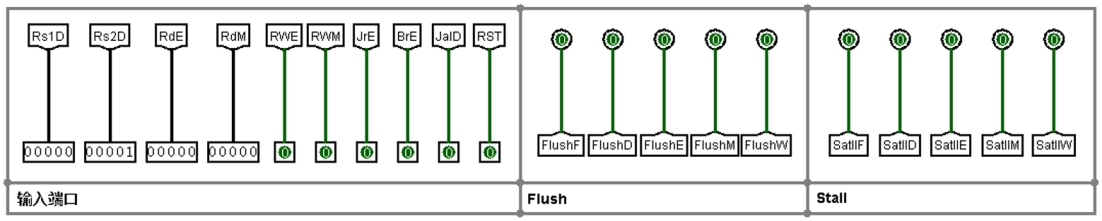
我们将**PC**外部的一系列选择跳转地址的电路封装起来，成为了地址生成器(Next PC Generator)。该模块接受分支、跳转、相对跳转的目标地址与使能信号，输出最终的目标地址。该模块有4个控制信号：`RST`、`BrE`、`JalD`、`JalrE`，在`RST`有效时，输出`PC`=$0x00000000$；`RST`无效且`JalrE`有效时，`PC`=`JalrPC`；`RST`无效且`BrE`有效时，`PC`=`BranchPC`；`RST`、`JalrE`、`BrE`均无效但`JalD`有效时，`PC`=`JalPC`；4个信号均无效时，`PC`=`PCF`$+4$。

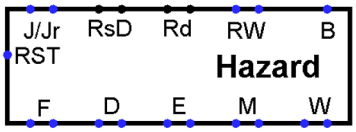




2. 冒险检测模块

该模块接受多个控制信号与寄存器地址信号，分析其中的数据冒险与控制冒险，并输出各段的`Stall`与`Flush`信号。





共有以下4类冒险：

- ID段与EX段的数据冒险：当EX段指令需要写寄存器（`RWE` $= 1$）、写寄存器地址不为0（`RdE` $\neq 0$）、写地址与ID段读地址的其中一个相同（`RdE` = `Rs1` 或 `RdE` = `Rs2`）时，存在数据冒险
- ID段与MEM段的数据冒险：当MEM段指令需要写寄存器（`RWM` $= 1$）、写寄存器地址不为0（`RdM` $\neq 0$）、写地址与ID段读地址的其中一个相同（`RdM` = `Rs1` 或 `RdM` = `Rs2`）时，存在数据冒险
- ID段控制冒险：JAL指令
- EX段控制冒险：JALR、Branch指令

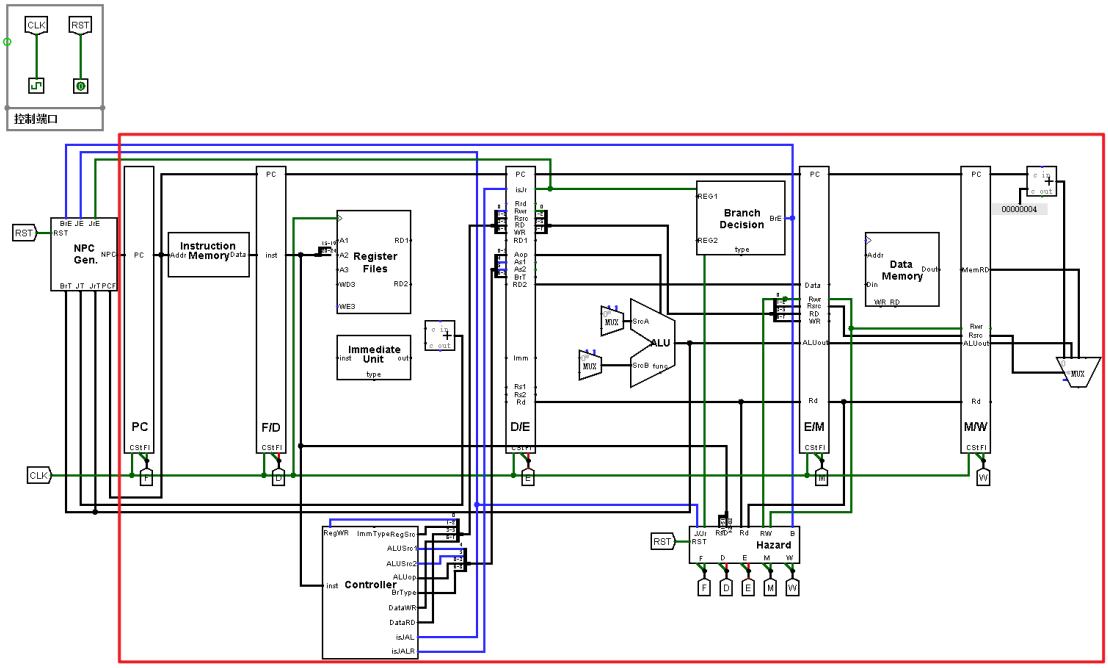
对于数据冒险，我们要阻塞**PC**和IF/ID段间寄存器，清空ID/EX段间寄存器；对于ID段控制冒险，要清空IF/ID段间寄存器；对于EX段控制冒险，要清空IF/ID段间寄存器与ID/EX段间寄存器。

注意：本次实验所使用的寄存器文件在时钟**下降沿**有效。对ID与WB段的数据冒险，由于寄存器在时钟周期前半部分写入，后半部分可以读出，因此该类冒险不存在。

四、测试

本次实验所完成的气泡流水线已经可以正确执行**RISCV32I**中的37条非特权指令，因此本次实验使用的测试集为`LogicalTest.S`。

需要测试时，在指令存储器中导入存储器映像`LogicalTest_inst`、在数据存储器中导入存储器映像`LogicalTest_data`，随后启用时钟信号即可。在测试完成后，寄存器x3中的值应为0x000000001。如需要进一步调试，可以将电路**CPU(TODO)**中**PC**以右的电路（即下图红框部分）直接复制到电路**CPU(Debug)**中，所有重要信号的探针将会直接连接到电路中。



说明：如果需要查看电路中某个模块的内部电路状态，需要使用手型工具双击该模块。这与你在向存储器中加入指令和数据的动作是一样的。

五、实验内容

本次实验已给出的部分：

- **PC**、**ALU**、立即数生成器、指令存储器、数据存储器、控制器模块的实例，这6个模块采用你在单周期、理想流水线实验中完成的部分，通过Logisim包的形式加载到本次实验的电路中。
- 段间寄存器、寄存器文件、地址生成器的端口、电路与封装形式
- 冒险检测模块的端口、部分电路与封装形式
- 在气泡流水线电路中，各个模块的实例与部分连线

需要你完成的部分：

- 冒险检测模块的电路
- 气泡流水线在ID、EX、MEM、WB段的连线

在完成以上内容后，请自行完成测试。

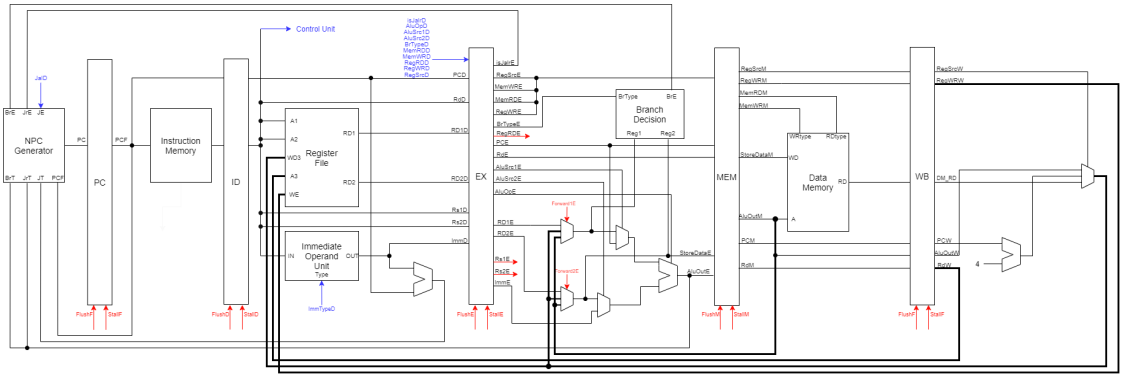
实验十 流水线3

一、实验目标

重写冒险检测模块，修改寄存器文件模块，完成重定向流水线的连线。

二、重定向流水线

在气泡流水线的基础上，我们加入了旁路与Load-use检测，并修改寄存器文件使得其内部拥有转发道路，这使得在存在冒险时流水线要插入的气泡更少，提高了流水线效率。其数据通路如下图所示。图片位于`/figs/exp10_DataPath.png`。

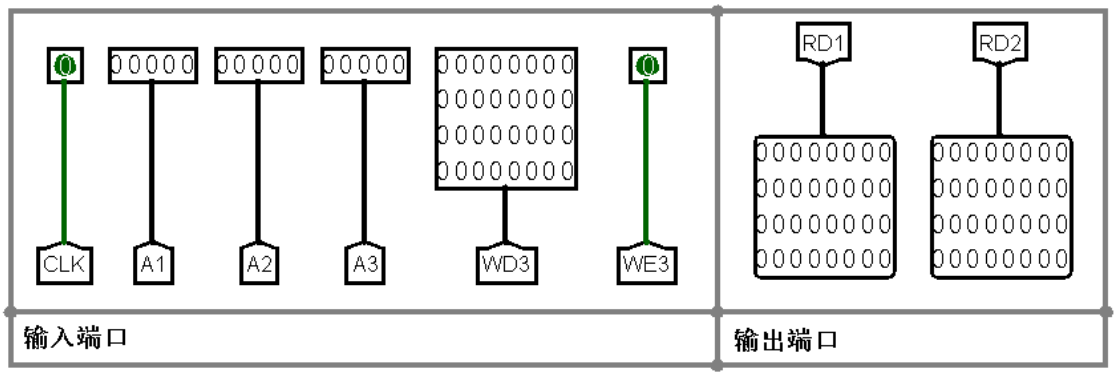


在本次实验中，我们不进行从其余段到ID段的转发，因此我们依然在ID段确认直接跳转指令JAL的目标地址，在EX段确认相对跳转指令JALR、分支指令的目标地址。

三、模块介绍

1. 寄存器文件

本次实验需要增加寄存器文件内的转发。当读地址`A1`或`A2`与写地址`A3`相同，`A3`不为`x0`，且写信号`WE3`有效时，将写数据`WD3`输出到`RD1`或`RD2`。



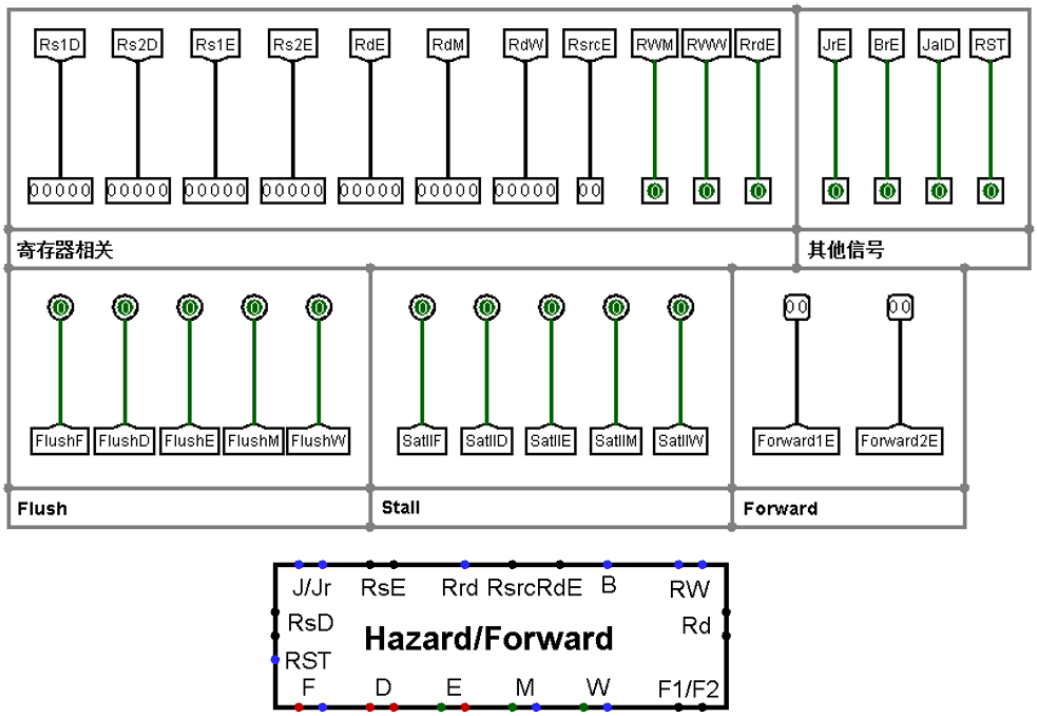
注意：本次实验所使用的寄存器文件在时钟**上升沿**有效。这对实验没有影响，只是提一下。

2. 控制器

由于现有信号无法判断是否存在寄存器读，我们在控制器中增加一个新信号`RegRD`。该信号已在Controller电路中给出，将你在实验7中完成的控制器复制到该电路中即可完成。该信号不一定需要使用在冒险/转发模块中，你在进行拓展实验中可能会使用到这个信号。

3. 冒险/转发模块

该模块接受多个控制信号与寄存器地址信号，分析其中的数据冒险、控制冒险与Load-Use，输出各段的`Stall`与`Flush`信号、EX段旁路选择信号`Forward1`与`Forward2`。



共有以下5类冒险：

- EX段与MEM段的数据冒险：当MEM段指令需要写寄存器（`RWM` $= 1$）、写寄存器地址不为0（`RdM` $\neq 0$）、写地址与ID段读地址的其中一个相同（`RdM` = `Rs1E` 或 `RdM` = `Rs2E`）时，存在数据冒险
- EX段与WB段的数据冒险：当WB段指令需要写寄存器（`RWW` $= 1$）、写寄存器地址不为0（`Rdw` $\neq 0$）、写地址与ID段读地址的其中一个相同（`Rdw` = `Rs1E` 或 `Rdw` = `Rs2E`）时，存在数据冒险
- Load-Use：当EX段为读内存指令(`RsrcE` $= 2$）、写寄存器地址不为0（`RdE` $\neq 0$）、写地址与ID段读地址的其中一个相同（`RdE` = `Rs1` 或 `RdE` = `Rs2`）时，存在Load-Use
- ID段控制冒险：JAL指令
- EX段控制冒险：JALR、Branch指令

对于数据冒险，若冒险的寄存器为`Rs1`，则修改`Forward1`；若为`Rs2`则修改`Forward2`。当EX段与MEM段存在冒险时，将MEM段的ALU数据`ALUoutM`转发到多路选择器的第2个端口；当EX段与WB段存在冒险时，将WB段的写回数据`RegWD`转发到多路选择器的第1个端口；若这两个冒险同时存在，优先转发MEM段数据；若冒险都不存在，选择多路选择器的第0端口，即寄存器读数据。

对于ID段控制冒险，要清空IF/ID段间寄存器；对于EX段控制冒险，要清空IF/ID段间寄存器与ID/EX段间寄存器。

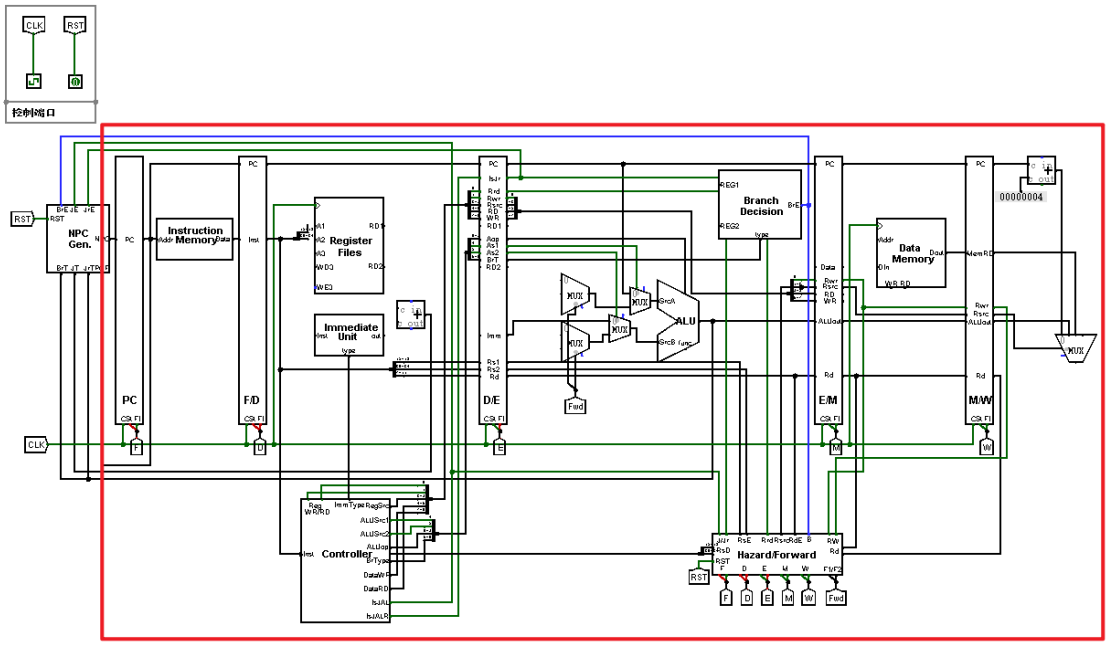
对于Load-Use，要阻塞**PC**和IF/ID段间寄存器、清空ID/EX段间寄存器。

实验电路中给出了一个优先编码器与下拉电阻。优先编码器给出输入中最高位的1的位置，当没有1时输出未知，这时候下拉电阻可以将未知改为0。你可以使用优先编码器实现MEM段冒险与WB段冒险的优先级，也可以自己使用别的组件。

四、测试

本次实验所完成的重定向流水线已经可以正确执行**RISCV32I**中的37条非特权指令，因此本次实验使用的测试集为`LogicalTest.S`。

需要测试时，在指令存储器中导入存储器映像`LogicalTest_inst`、在数据存储器中导入存储器映像`LogicalTest_data`，随后启用时钟信号即可。在测试完成后，寄存器x3中的值应为0x000000001。如需要进一步调试，可以将电路**CPU(TODO)**中**PC**以右的电路（即下图红框部分）直接复制到电路**CPU(Debug)**中，所有重要信号的探针将会直接连接到电路中。



说明：如果需要查看电路中某个模块的内部电路状态，需要使用手型工具双击该模块。这与你在向存储器中加入指令和数据的动作是一样的。

五、实验内容

本次实验已给出的部分：

- **PC**、**ALU**、立即数生成器、指令存储器、数据存储器模块的实例，这5个模块采用你在单周期、理想流水线实验中完成的部分，通过Logisim包的形式加载到本次实验的电路中。
- 段间寄存器、地址生成器的端口、电路与封装形式
- 寄存器文件、控制器、冒险/转发模块的端口、部分电路与封装形式
- 在重定向流水线电路中，各个模块的实例与部分连线

需要你完成的部分：

- 冒险/转发模块的电路
- 寄存器文件的转发电路

- 重定向流水线在ID、EX、MEM、WB段的连线

在完成以上内容后，请自行完成测试。

Logisim 综合实验报告

一、实验内容

Exp1_RegisterFiles

实验截图

Exp2_PC

实验截图

Exp3_Immediate

实验截图

Exp4_Branch

实验截图

Exp5_ALU

实验截图

Exp6_Memory

实验截图

Exp7_Controller

实验截图

Exp8_CPU

实验截图

Exp9_Pipeline Ⅰ

实验截图

Exp10_PipelineⅡ

实验截图

Exp11_PipelineⅢ

实验截图

二、思考题（选择其中 3 道作答）

1. 怎样减少分支延迟槽，减少后 `Hazard Detection` 和 `Forwarding` 有何变化?

2. 在原本流水线结构中，`Branch` 的优先级比 `Jal` 高。现若在流水线中加入分支预测，`Branch` 指令在 `IF` 阶段就可以跳转（假设预测跳转）。假设现在有一条 `Jal` 指令在 `ID` 阶段，有一条 `Branch` 指令在 `IF` 阶段，此时会导致执行顺序位于前面的 `Jal` 指令跳转被忽略，如何解决这个问题?

3. 我们在 `ID` 阶段处理 `Jal` 跳转，在 `EX` 阶段处理 `Jalr` 和 `Br` 跳转以及 `Load-use` 冒险。为什么他们有处理阶段的差别？是否可以将某些处理放在更前的流水段进行？如果可以，应该怎么做，`Forwarding` 模块应该如何修改，效率会不会提升?

4. 在单周期 CPU 中，如果将指令存储器的容量增大到 64KB，有哪些地方需要改动？尽量给出具体的值。

5. 在流水线 CPU 中，当 `ID` 和 `WB` 阶段需要读写同一个寄存器时会存在数据冒险，给出两种解决该数据冒险的方法。

三、扩展实验说明

介绍自己所做的扩展实验，如何设计的，关键结构说明，结果验证截图等

四、实验总结

你的收获，对单周期和流水线的理解，对实验的改进意见等