华东师范大学软件工程学院实践报告

课程名称：计算机组成与实践	**年级：2023 级**	**上机实践成绩：**
指导教师：谷守珍	**姓名：**张建夫	**上机实践日期：**5/21~6/3
实践编号： 实验3	**学号：**10235101477	**上机实践时间：**4 学时

一、 实验名称

MIPS存储设计实验

二、 实验目的

- 理解主存地址基本概念
- 掌握存储系统位扩展基本思想
  - 能构建同时支持字节、半字、字访问的存储子系统
  - 数据位宽可变
- 设计MIPS寄存器文件
  - 32个寄存器，两个读端口，一个写端口
  - 熟悉多路选择器、译码器、解复用器

三、 实验内容

- MIPS RAM设计
- MIPS寄存器文件设计

四、 实验原理

1. **Load/Store实现原理：**

本次实验中支持MIPS Load/Store指令的存储器主要有以下引脚定义：

引脚	输入/输出	位宽	功能描述
Addr	输入	12	字节地址 （字访问半字访问时应硬件强制对齐）
Din	输入	32	写入数据，不同访问模式有效数据均存放在最低位，高位忽略
WE	输入	1	写使能，高电平有效
CLK	输入	1	时钟信号，上跳沿有效
Mode	输入	2	访问模式 00：字访问，01：字节访问，10：2字节访问
Dout	输出	32	输出数据，不同访问模式有效数据均存放在最低位，高位忽略

该定义并不完整，在后面的问题讨论环节中我会将里面的问题指出来。

其中，由于需要支持半字和字节的访问，我们需要对半字和字节的访问位置进行定义：

从上面的图可以看出，我们需要将结果先分成两个大的部分（是load，装入寄存器，还是store， 存入存储器），然后每个部分有7种结果，里面需要注意的是高半字和低半字的部分，要正确选择模式且高半字的两个字节要对的上（例如，不能是Byte2和Byte1）

除去上面的这些，我做的过程中还出现了其他情况，将在实验过程中一一描述。

2. **MIPS寄存器文件设计：**

MIPS寄存器文件的主要功能是将译码的结果进行部分的“实现”，具体是将对应的寄存器的值从寄存器组中取出，同时写入之前指令算出来的结果到寄存器中，具体流程如下图：

（中间黄色部分为需要实现的元件）

而对于具体的接口含义，如下图所示：

引脚	输入/输出	位宽	功能描述
R1#	输入	5	第1个读寄存器的编号
R2#	输入	5	第2个读寄存器的编号
W#	输入	5	写入寄存器编号
Din	输入	32	写入数据
WE	输入	1	写使能信号，为1时在CLK上跳沿将Din数据写入W#寄存器
CLK	输入	1	时钟信号，上跳沿有效
RD1	输出	32	R1#寄存器的值，**0号寄存器值恒零**
RD2	输出	32	R2#寄存器的值，**0号寄存器值恒零**

这里需要注意的点是0号寄存器恒为零（实际上就是选择0号寄存器的话就返回0这个常数），WE为写使能信号，这里因为实现的是单时钟周期的cpu，故没有用到旁路之类更复杂的机制。

五、 实验过程

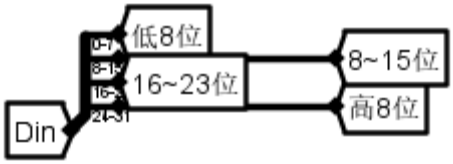
我分了主要两大块完成本实验，第一部分是实现支持MIPS Load/Store指令的存储器，第二部分则是实现MIPS寄存器文件。在第一部分中，我又将过程分为了store指令实现和load指令实现两个小部分，第二部分则是整个一口气做的。

（1） 第一部分（实现支持MIPS Load/Store指令的存储器）

1. Store指令实现：

刚开始做的时候认为应该先尽可能多的把引脚用上，就选择先实现store指令，因为store指令要用上Din和WE，而load只用到Dout，不用Din和WE。

首先进行了指令的分析，由于有存储字节和半字的需求，我们先要将Din“分成”四个字节，这样后面在取用这四个字节就可以很方便了：

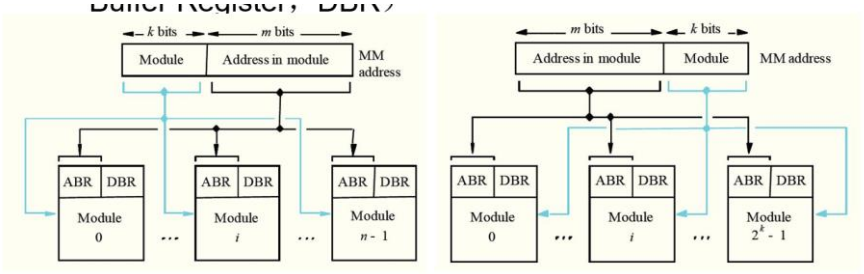


这里要同时标上字节对应的位置，以便后面辨认。

接着查看每个存储器的存储位宽（这样你才能知道你一次可以给一个存储器存多少数据）：

RAM	
Address Bit Width	10
Data Bit Width	8
Label	高8位
Label Font	Dialog Plain 12
Label Color	#000000
Data Interface	Separate load and...
Sel Active On:	High Level

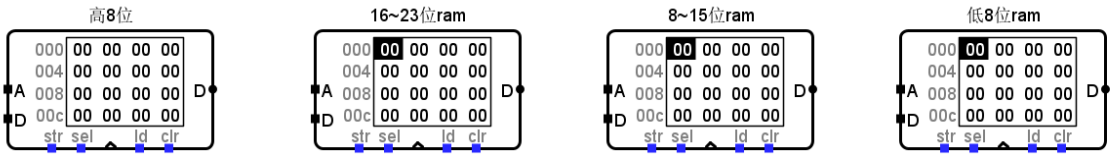
存储位宽为8，意味着每次只能存储8位，而在课上的ppt中我们知道：



（a）一个模块中存储连续的字　　（b）连续的字在多个模块中交叉存储

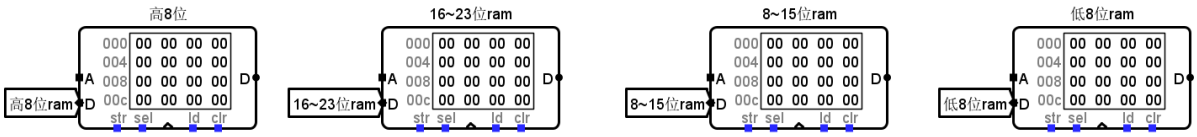
哪种策略更好？后者！因为后者可以并行访问多个字

存储器的实现一般采用交叉存取的方式，也就是说，字的存储是“不连续的”，逻辑上连续的地址在实际存储时是不连续的，这样可以加快字的访问速度，而该页ppt中的字，实际上就是本次实验中字节，这四个提前准备好的ram表示一个字（4个字节），而这个字会被一个一个字节存储到不同的ram中，因此，在实际对ram进行接线前，我们先要决定哪个ram表示哪个位置的字节：



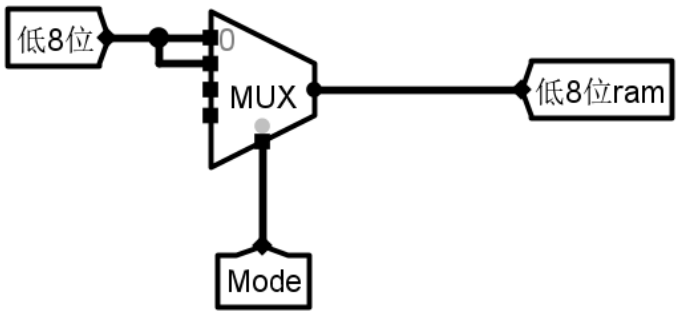
我从左到右按照高位到低位的顺序对ram进行了命名，这样接线的逻辑会清晰很多。

接着，我使用上次实验学到的隧道技术，将各个ram的数据输入用一个隧道表示，实现了模块功能分离：



接下来就是具体计算这些隧道的值。

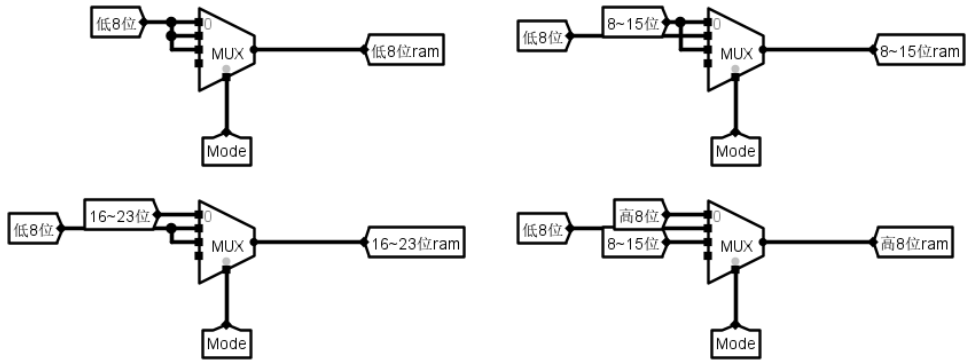
首先考虑低8位的数据输入，由于有半字存储和字节存储，根据之前的实验原理部分的分析可知，mode决定了使用字节还是半字，先拼出一个字和单个字节（00和01）访问的情况：



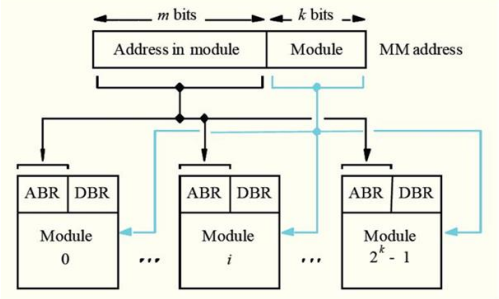
一个字的访问情况：由于需要存储数据到“低8位ram”中，所选择的数据自然是Din的低8位了（见上面指令分析的部分）

一个字节的访问情况：显然只能存储Din的低8位，从实验原理的图中可以看出Din只有低8位保存数据。（这里有一个盲点，对于存储到其他ram中的数据，如“高8位ram”，一个字节的访问情况下仍然只能存储Din的低8位，原因在于虽然每个数据输入都试图向它们对应的存储器输入数据，我们可以通过str（store使能信号）进行控制，这样就达到了存取Byte1或Byte2等特定字节的目的）

对于半字的情况稍微复杂一点，从实验原理中的图可以看出，半字都存储在Din的低16位上，16位显然是超出8位存储器的能力，因此需要两个存储器来实现半字存储，此外，由于半字存储时只能是Byte3+Byte2或Byte1+Byte0的组合，这两个存储器也必须是高16位的两个存储器或低16位的两个存储器，由此，我们可以知道，对于“低8位ram”，半字存储的mode输入也是Din的低8位，最后，根据相同的规律，我们可以把四个ram的数据输入连出来：



接着就是控制str（store使能信号）来执行具体的ram存储，在这里需要补充一些前置知识，在交叉存取中，低位用来表示存储的模块，高位用来表示块内偏移：



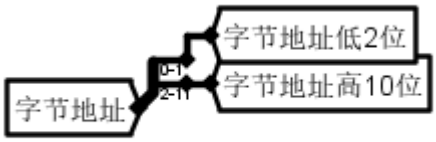
（b）连续的字在多个模块中交叉存储

而该处总共有4个ram，也就是说，字节地址低两位用于表示对应的ram，其他位表示ram内部的偏移，查看ram内部的总地址数也可以验证：

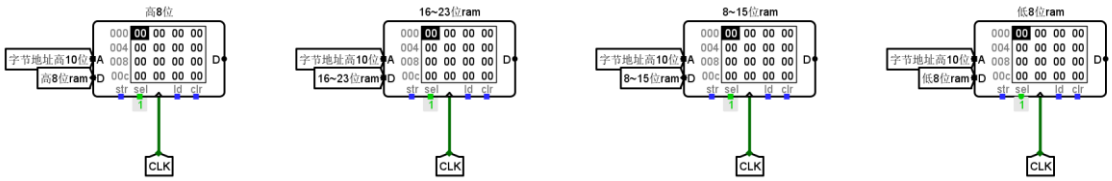
Selection: RAM	
Address Bit Width	10
Data Bit Width	8
Label	高8位
Label Font	Dialog Plain 12
Label Color	#000000
Data Interface	Separate load and...
Sel Active On:	High Level

（一个ram有10位的地址）

而提供的字节地址接口有12位，为了使访问更加便捷，我们仿照Din对字节地址进行预处理：

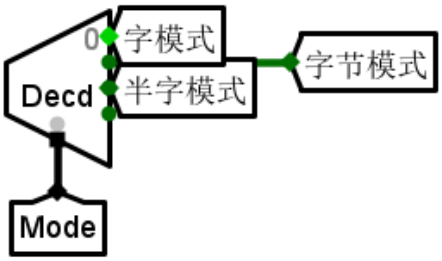


并更新存储器的接口：



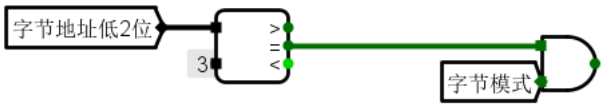
（此处还添加了时钟和片选信号，这两都是必要的，就顺便添上了）

接着是str（store使能信号）的具体门电路设计，由于WE是写使能信号，最后肯定是一个与门来实现WE和另一个（或者多个）变量的输出，这个变量则是由mode和字节地址低两位实现的，对于mode的三种情况，我们要进行分类讨论，因此，我们需要一个译码器来得到不同模式的信号：



首先，对于“字模式”，所有的ram都要使能才能取出一个字，因此，该模式的输入就是“字模式”本身。

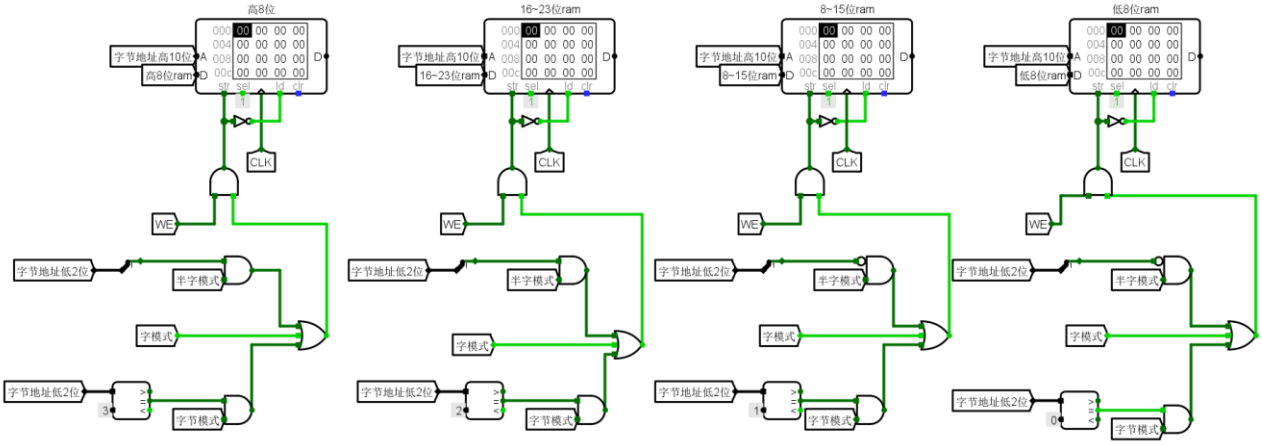
对于“字节模式”，字节地址的低两位会决定数据被写入哪个ram，因此，我们需要一个比较器来对字节低两位的值进行比较，如果对应的值相等，则说明是对应的ram（我原本想的是使用门电路实现值的比较，结果发现很复杂，后面想到上次实验用过比较器，就比较方便），如下图所示：



（此处的3表示匹配的是“高8位的ram”）

最后对于“半字模式”，这里有一个知识点，对于半字模式，由于只有高半字和低半字的区别，因此只要一位就能确定，而这个位是字节地址的第2位，如下图所示：

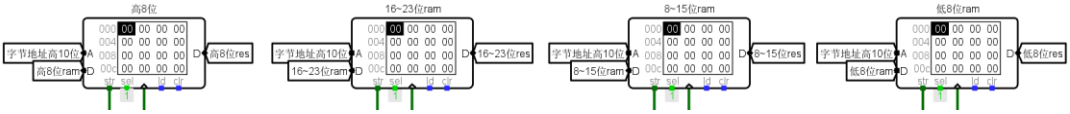
最后，根据上面的分析结果连线，对这三个模式使用或门即可取出结果：



（此处在最开始实现的时候，我认为ld和store肯定是相反的且不能悬空，便连了一个非门到ld使能端，不过事后证明我是错的，这个后面问题讨论时会提到。）

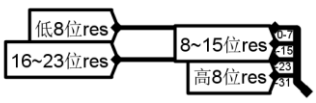
2. Load指令实现：

与store指令的实现过程一样，我们先为每一个ram的输出用一个隧道表示，方便之后使用：

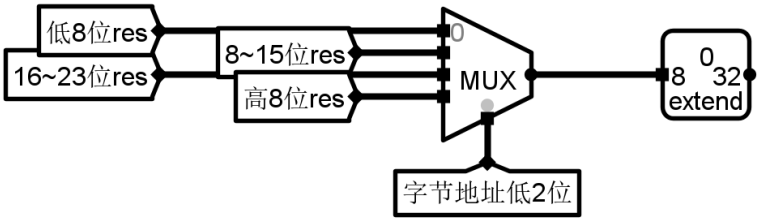


对于输出，load和store一样有三个模式，这三个模式输出长度分别为8位，16位和32位，由于输出结果需要32位，不足32位的需要位扩展器进行位扩展，另外，由于ram的输出都是8位，这8位需要拼接成32位或16位，就要用到splitter器件的反向用法（即拼接功能），这里我是查询了相关资料才知道splitter能够反向使用，之前以为它只能作为位选择器使用。

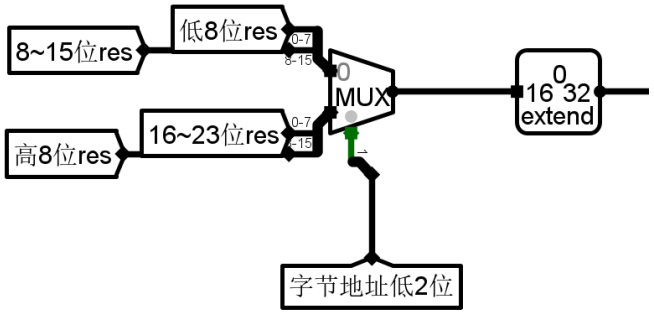
首先拼接一个字的输出：



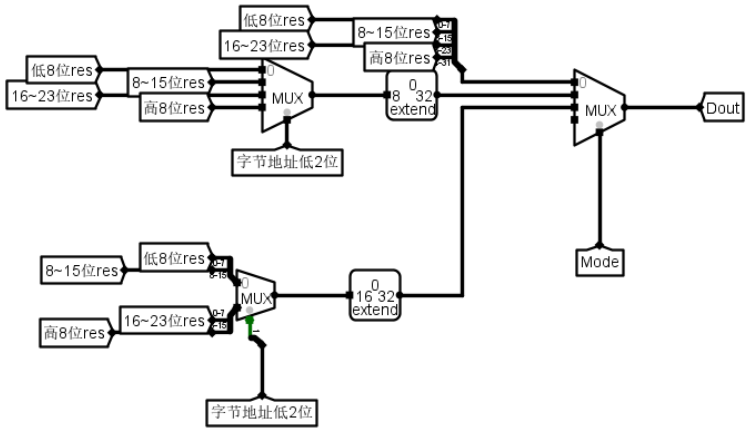
接下来是字节的输出，这里就又要用到先前的字节地址低2位来选择输出的ram，因此要多加一个选择器：



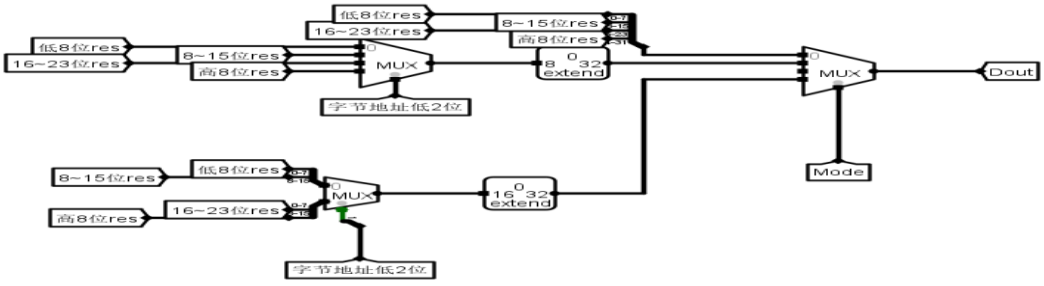
然后是半字的输出，由于只能是Byte3+Byte2或Byte1+Byte0的组合，且字节地址的1号比特位决定了是高16位还是低16位，因此选择器只有两个输入，分别是高16位和低16位的拼接结果：



最后使用一个选择器根据mode选出正确的Dout：



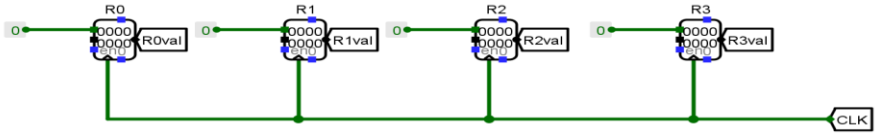
（我第一次接的时候是错误的（半字模式的输出引脚接错了，如下图），结果后面测试的时候一直有问题，但是我找不到为什么，花了比较久的时间才发现问题，遂记录）



（2） 第二部分（实现MIPS寄存器文件）：

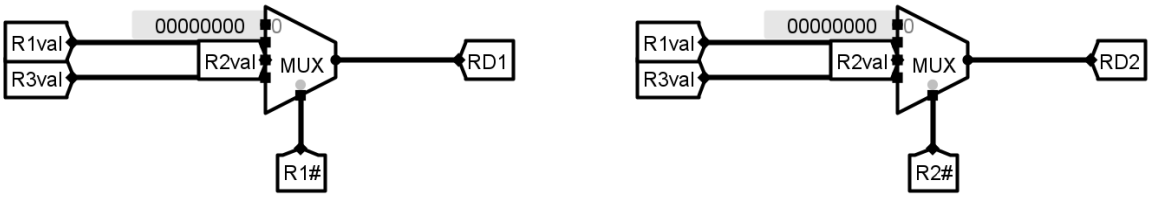
这一部分的实验相比于上一个要简单很多，首先分析其功能，和上一个部分的实验一

样，是分为读写两个部分，这里的读操作相当简单，只要根据R1#和R2#的值返回对应寄存器里的值就行，因此，为了方便读出各个寄存器的值，我为每一个寄存器进行编号并用隧道分别表示对应的值：



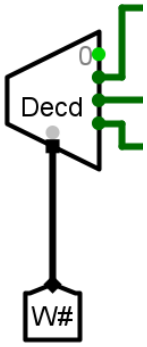
（这里的常数0是为了使器件有效，寄存器的使能信号为低电平）

后面将这些值输入两个选择器即可：



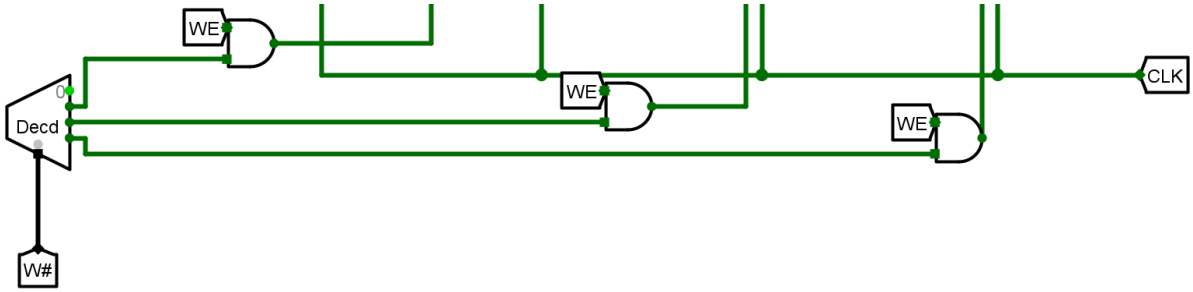
这里需要注意的是R0寄存器里的值恒为0，因此如果选择器选到了R0，返回常数即可（虽然和硬件的实际实现可能不太一样）。

接下来是写操作，实际上和读操作差不多，通过W#选择相应的寄存器，只不过我们需要将W#的信号译码，来选择哪个寄存器是我们要写入的，可以使用一个译码器实现：

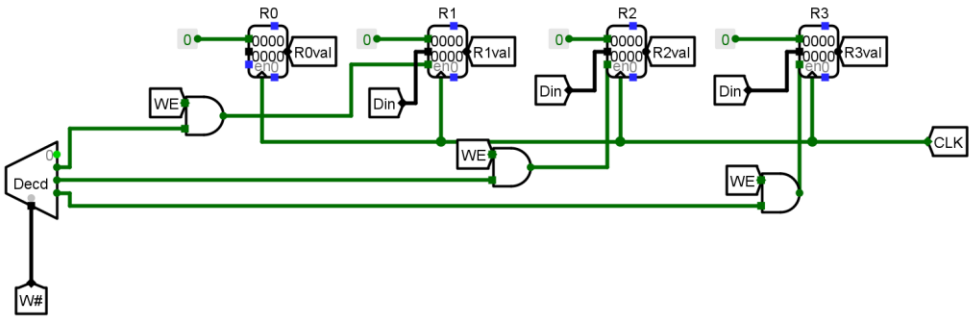


此处0悬空是因为0号寄存器不能被修改。

同时，由于写操作需要WE写使能信号有效，因此我们要将译码的结果分别和WE信号通过一个与门：



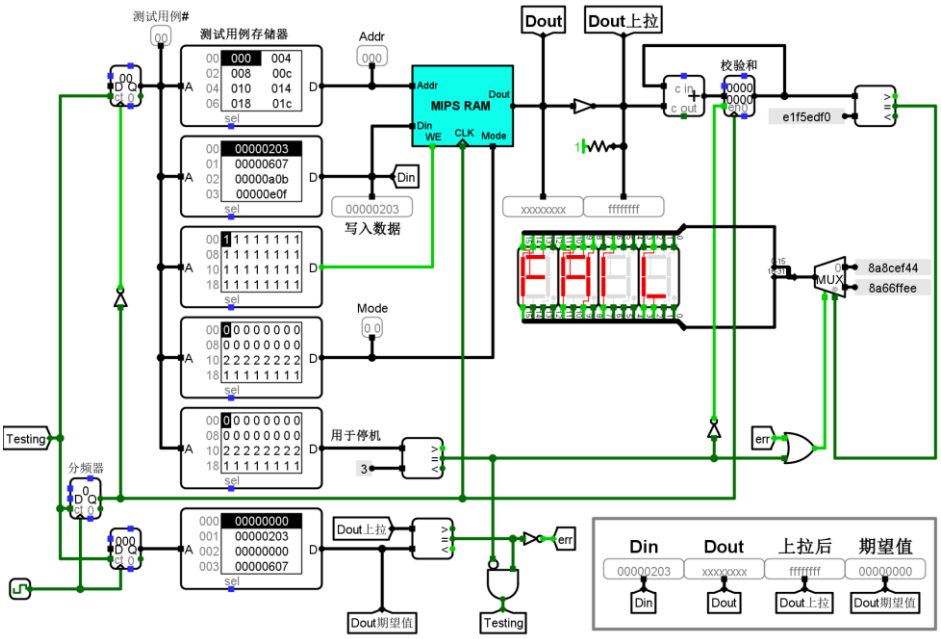
最后将这些结果连到寄存器的写使能信号上，并为寄存器提供Din：



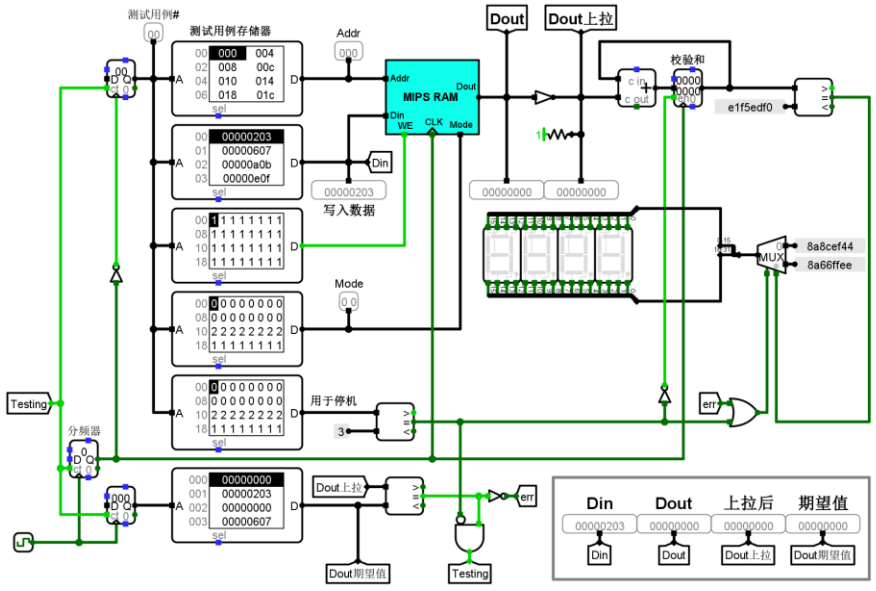
至此，寄存器文件实现。

六、 实验结果及分析

实验测试采用的是模板自带的测试方式，我是先对支持MIPS Load/Store指令的存储器进行测试，结果发现一直无法开始：

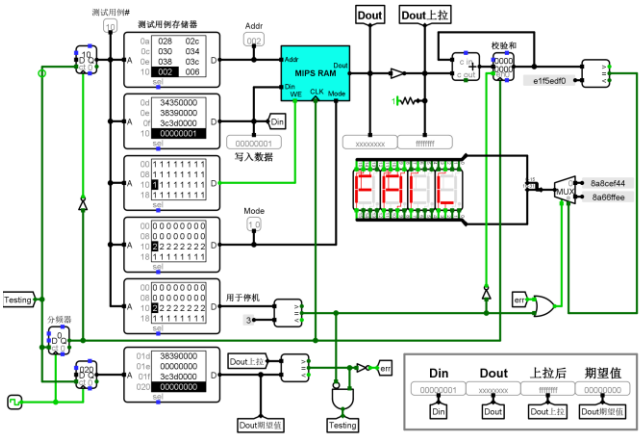


询问了其他同学，他们的初始界面是这样的：

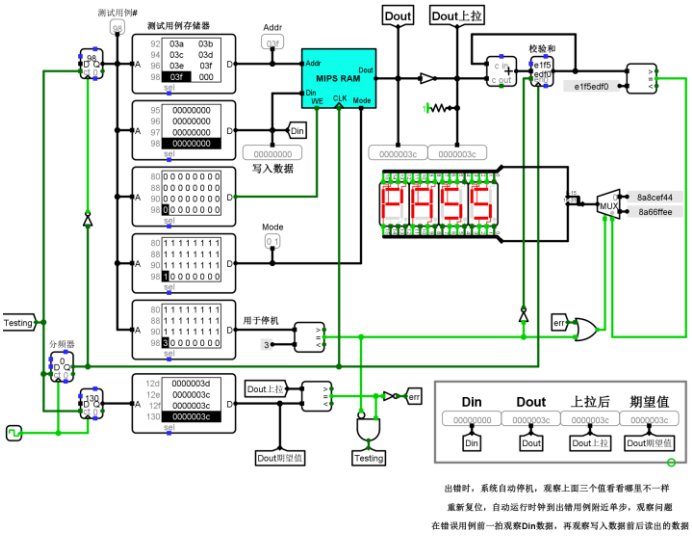


这让我非常疑惑，后面比较了我的实现和其他同学的实现后发现，他们都是将ld端口悬空，但是我想不明白为什么要这样做，后面问了老师才明白原因，解释的部分将会放到问题讨论中。

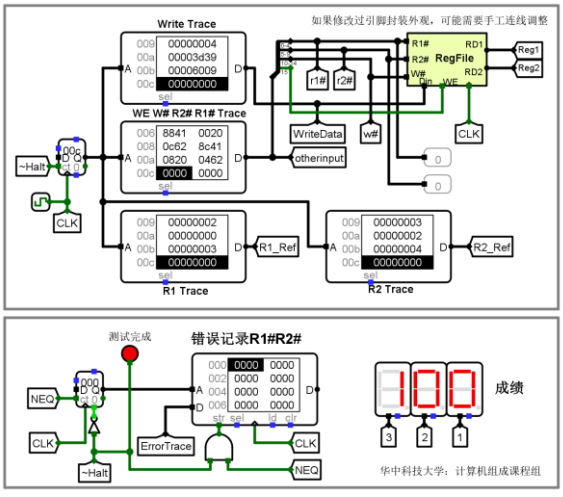
在将ld端口悬空后，测试才正常进行，但是我又遇到如下问题：



测了好多次仍然出现该问题，但是我检查了多遍自己的实现，逻辑上没有问题，到了第二天早上我再看实现的时候，发现是输出里面半字模式的引脚接到了选择器的4号位（实验过程的部分里有详细解释），改过来之后测验顺利通过：



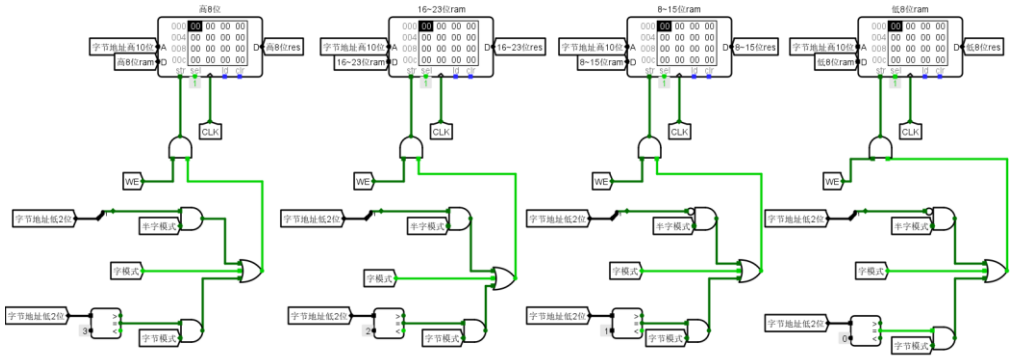
接下来测试寄存器文件的实现，这次相当顺利，一次过：



七、 问题讨论

对于上面多次提到的ld引脚需悬空的原因解释如下：

由于指令并不只是ld或者str，对于非写入的情况，我们可以采用读取ram中的数据但是不写回到寄存器，这样就能够在其他指令执行的时候不影响寄存器的结果，而如果像我之前一样使其不是ld就是str，会导致存储器的值不正确，因此，最终电路修改如下：



八、 心得体会

本次实验体验了实现了load和store指令以及寄存器文件，前者在刚开始实现的时候比较摸不着头脑，但是在做实验的过程中逐渐找到了方向，并一步步完成实验的不同部分。另外，在实验的过程中也巩固了许多课上的知识，也学到了很多课上没有讲到的东西，毕竟具体实现会有具体实现的问题，这与理论课中讲的总体方法会有所偏差。最后一点应该就是logism的使用变得更加熟练了，之前用不惯的元件本次实验都用得很顺手。