

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机体系结构 |
| 姓 名： |  |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： |  |
| 指导教师： |  |

2023年 9月 27 日

**浙江大学实验报告**

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： 实现支持RISC-V的流水线CPU

学生姓名： 专业： 计算机科学与技术 学号：

同组学生姓名： 指导老师：

实验地点： 曹西301 实验日期： 2023 年 9 月 27 日

1. 实验目的和要求

实验目的梗概：

Please do the project step by step according to the project tutorial. Design the basic function components including CtrlUnit, HazardDetectionUnit and RV32core using Xilinx.

实验要求：

1. Understand RISC-V RV32I instructions.
2. Master the design methods of pipelined CPU executing RV32I instructions.
3. Master the method of Pipeline Forwarding Detection and bypass unit design.
4. Master the methods of 1-cycle stall of Predict-not-taken branch design.
5. Master methods of program verification of Pipelined CPU executing RV32I instructions.
6. 实验内容和原理

Design the basic function components including CtrlUnit, HazardDetectionUnit and RV32core.

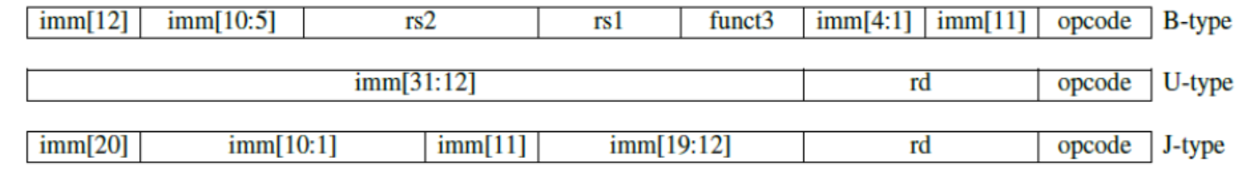
在本次实验中，流水线CPU的大体框架已经给出，我们只需要在其中填写设计部分模块即可。在这一小节，我们先对需要实验填写的部分进行原理的分析与探究。

2.1 RV32core

在这一部分中，我们关注的是流水线CPU硬件中，各模块之间的数据链接。从实验指导的PPT中，我们可以得到以下清晰的模块交互示意图。

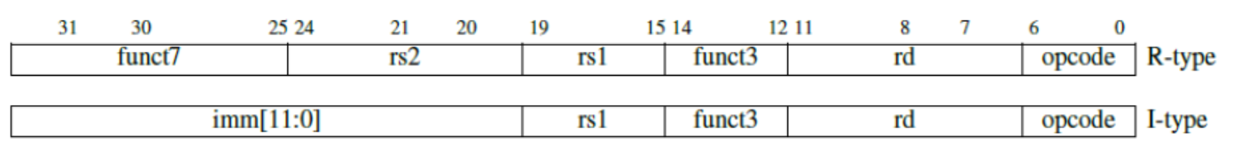
在填写RV32core时，我们需要关注的就是各个mux选择器的选择。

对于IF阶段的mux而言，我们需要通过选择器选择下一CPU时钟周期内进行解析的PC指令来源；对于一般的顺序操作而言，我们只需要选择PC+4这一指令即可；但对于JAL、JALR与SB型而言，我们需要去（有条件的）选择跳转目标地址的PC指令，从而需要这一mux的作用。

可能介导PC跳转的指令类型示意图

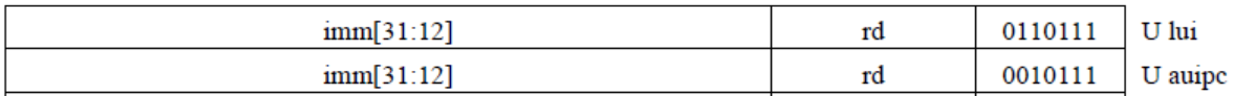
接下来我们分析ID阶段的两个mux选择器。在ID阶段的这两个mux，都是四选一数据来源，选择进入ALU计算的具体数据。

那对于第一个（上侧的选择器而言），我们需要根据指令解析结果的不同，来选择数据来源；可能是从数据寄存器组中得到，也有可能从ALU计算结果，MEM阶段输入或MEM内存数据得到；这对映了一般R/I型指令的计算，或考虑不同阶段data hazard的产生，从而需要对映data forward部署进行一定的克服。

R/I型指令的通用指令格式

那对于第二个mux选择器而言，数据来源上的选择大体与第一个mux一致。可能从数据寄存器组中得到，也有可能从ALU计算结果，MEM阶段输入或MEM内存数据得到；也分别对映着R/I型指令的计算与一些forwarding的情况。

另外，MEM阶段有最多的mux选择器需要我们进行设计。一方面，根据指令的不同，我们需要选择进行ALU计算的第一个数据，是来源于寄存器组等的选择，还是来源于PC，因为在ALUPC指令中，我们需要对PC指令进行数值的加减，并填入寄存器中。

AUIPC与LUI指令的指令格式

第二个mux选择器本质上还是选择进入ALU进行数值计算数据来源。一方面，它可以选择数据来自数据寄存器组等的选择；另一方面，它也可以选择立即数进入ALU进行计算。最后一个mux寄存器用于选择数据来源为数据寄存器组中结果还是内存中读取结果，对映于S型指令必须的forward途径。

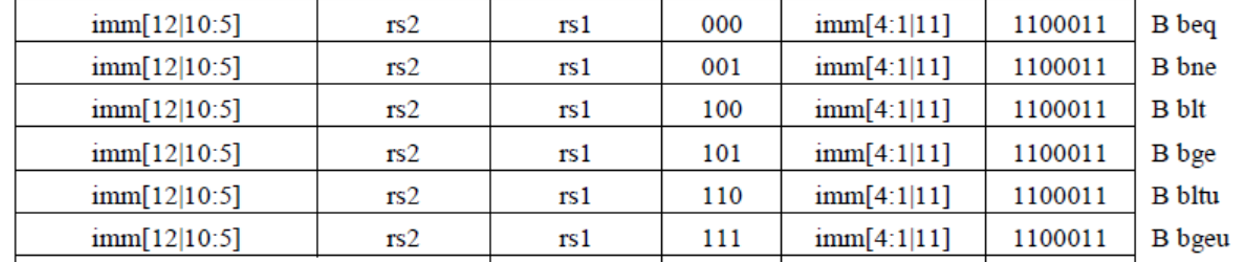
2.2 cmp\_32

在这一部分，我们需要设计填写32bit比较单元cmp\_32，为B型指令介导的PC Branch作出选择指导。

对于equal与not equal信号，我们的设计十分简单，只有当两个32bit的二进制数据每一bit都相等时，我们才认为这两个数相等。但对于大小比较而言，我们既要考虑无符号数，也需要考虑有符号数。

对于无符号数而言，我们可以直接使用 < 或 > 进行两个数的大小比较；但对于符号数而言，我们需要先判断两数的最高位bit，并在除却最高位bit后，进行无符号数类型的比较，其中负数结果还需要相反。

另外，我们的B型指令设计，是基于不同的大小比较条件，进行PC的跳转；因而，我们需要额外设置一个信号，来接受B型指令需要判断的信息，从而根据条件，进行PC是否跳转的指导。

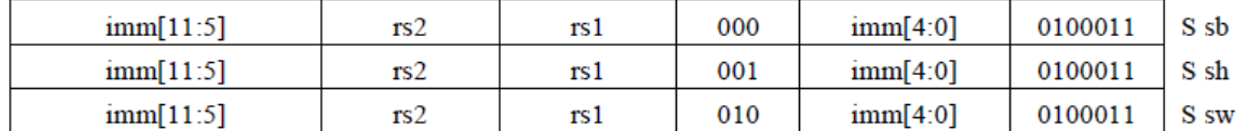
不同条件B型指令的设计

最后，我们需要输出一个控制信号，来表示根据大小比较条件，给定的B型指令是否会介导PC的跳转，从而统一模块间的信息设计。

2.3 CtrlUnit

进而我们需要设计CtrlUnit这一控制单元，对控制信号进行统一的判断、设计与输出。

首先，我们要对输入的指令进行一个拆解，先判断它是什么类型的指令，再将这32bit指令拆开，获得每三位与七位的详细信息，方便我们进行后续的操作。根据已有的框架，我们需要填补的，是B型指令、L型指令、S型指令大类中具体指令使用的判断。

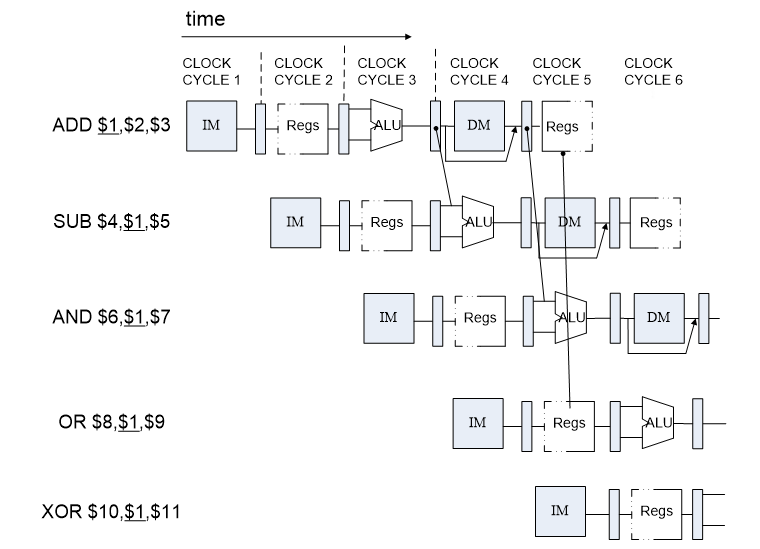
S型指令的具体指令类别

对于这三者的大类判断以及小类甄别，我们可以通过最开始七位来判断指令类型，而使用中间的三bit来判断具体的指令小类。而对于AUIPC与LUI指令而言，在本实验有限可以实现的指令类别中，我们只需要用最先七位来判断即可。

接下来我们需要管理一些控制指令的产生。其中，最直接的是jalr控制指令，这一指令用于选择需要跳转的PC地址，是基于当前PC进行跳转，还是基于寄存器中的存储数值进行add跳转。

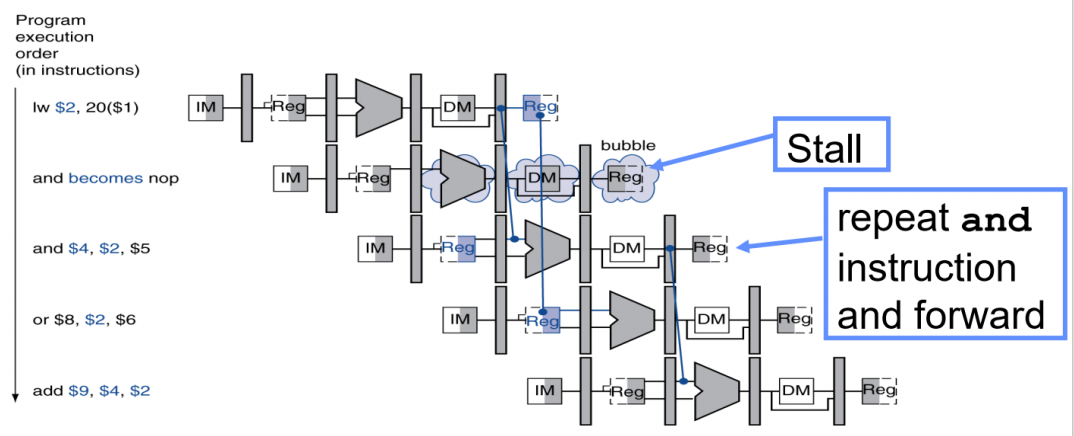
另外，我们还需要在CtrlUnit控制模块中生成合适的比较信号指令，用于指导需要比较模块比较的类型，并进行输出。对于我们在2.1阐述的最上端的两个mux选择器而言，我们需要给出对应的控制信号，来判断我们需要进行ALU数值计算的数据来源；这些控制信号的选择，只需要具体的指令类别即可。

另外，我们还需要额外设置两个信号，用于记录指令中是否提及使用了寄存器组；详细对映于数据来源是否是寄存器组中寄存器，以及数据写入对象是否是寄存器组。这在一般的设计中看似多此一举，但却为后续hazard的解决与forward的实现提供了充足的信号保障。

最后，我们还需要给出hazard\_type的判断。对于本实验的设计而言，我提出了四种hazard的判断类型。第一种对映于简单的数据冲突，即在进行相邻指令的流水线操作时，可能会有上一条指令的目标寄存器，是这一条指令的数据来源的情况；且这种冲突不仅会发生在EXE阶段，也会发生于MEM阶段，因而，我们需要对映的forward通路进行data hazard的解决。

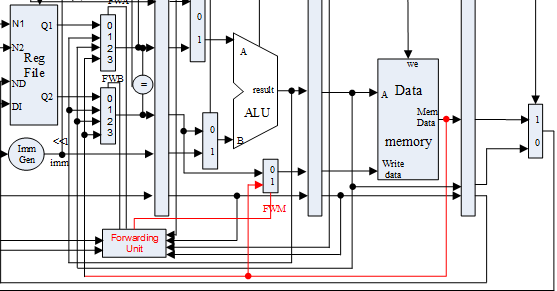
data hazard示意图

第二种hazard来源于L型指令，即我们需要从内存中读出数据，进行后续可能的数据计算。但对于L型指令而言，这里存在一个致命的问题，即L型指令获得数据一定要在MEM阶段，如果流水线CPU下一条指令需要对这个数据进行操作，则必然发生冲突，且任何forward途径都不能形成有效帮助，因而我们需要对此进行特殊的运算。

L型指令介导的hazard示意图

第三种hazard来源于指令的跳转，因为PC的指令需要跳转，所以在正常流水线CPU运作时，可能有部分指令会存在多余解析的问题（因为跳转后不再执行本条指令）；从而，关于跳转的指令，需要使用一种插入空白的方式，使流水线CPU停止一回合，这样可以辅助PC跳转到正确的指令位置，进行后续的指令操作。

最后一种hazard来源于S型指令，因为S型指令需要将一个数据存入MEM中，而这个MEM中的数据，可能会被后续的指令使用，如参与具体的计算；因而在此我们也需要相应的forward途径，能及时的将这一从内存中读取的结果，迅速的通过旁路前递给计算单元。

关于S型指令的前递旁路

2.4 HazardDetectionUnit

最后，我们需要设计冲突解决模块。在这一模块中，我们既要考虑如何甄别每一种可能发生的hazard冲突，也要考虑如何使用forward旁路来解决这些问题，或者说，使用插入nop的方式，来延缓流水线CPU，达到稀释hazard影响的效果。

对于第一种hazard，我们需要判断两（一）个源寄存器是否处于使用状态，并判断当前指令的目标寄存器是否与先前指令的源寄存器使用产生冲突，在此基础之上；我们还需要判断使用的源寄存器是否为0号寄存器，因为0号寄存器的数值始终为0，所以不会产生所谓的数据冲突；而当产生数据冲突时，我们只需要简单的将一些已经计算获得的结果进行前递即可。

同理于第四种hazard，这种hazard由S型指令介导，本质与第一种hazard一致，也是因为数据产生位置靠后（在MEM阶段），从而需要进行一定的前递；判断方法与控制信号的产生要求与第一种hazard基本一致。

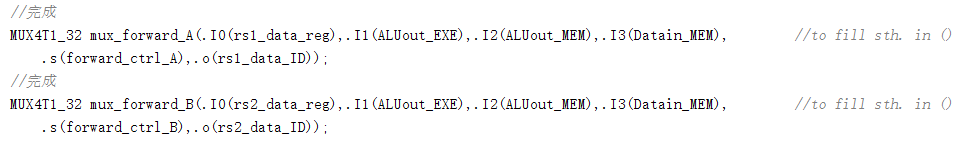
而对于第二种hazard，我们对于这类由L型产生的hazard而言，不能使用简答的forward方法来解决，只有旁路不能解决这个问题。另外，我们在采用类似的判断hazard方法的基础上，还需要记录前两条指令的hazard\_type，这可以用于判断hazard产生的具体位置，更好的介导forward结局方案。因而，对于这一类的hazard，我们需要将对映的寄存器部分进行清零或保留，以模仿插入nop的效果。

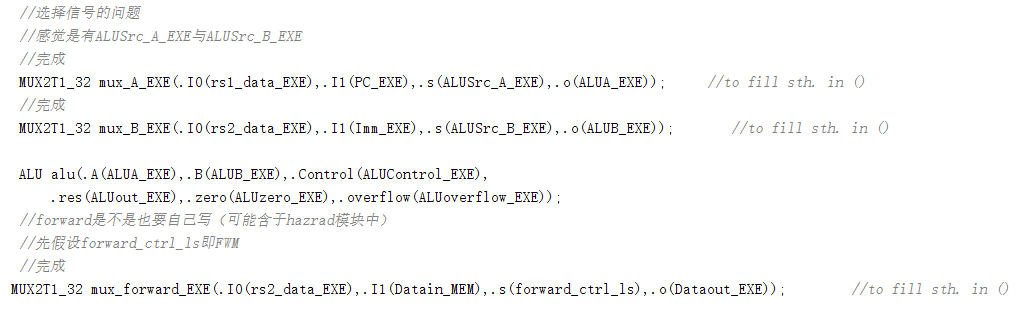
最后第三种hazard是由B型指令、JAL/JALR指令跳转介导产生的。对于这类指令而言，后者介导无条件跳转；前者可能会产生跳转。若条件不满足，不产生PC的跳转时，流水线CPU可以照常运行，不会产生任何问题；而当需要产生PC跳转时，我们也可以像上一类hazard的处理办法一样，通过插入一个nop来正常运行ID阶段及以后的流水线PC指令，而对IF阶段的指令，进行清零与修改，以达到PC跳转的效果。

1. 实验过程和数据记录

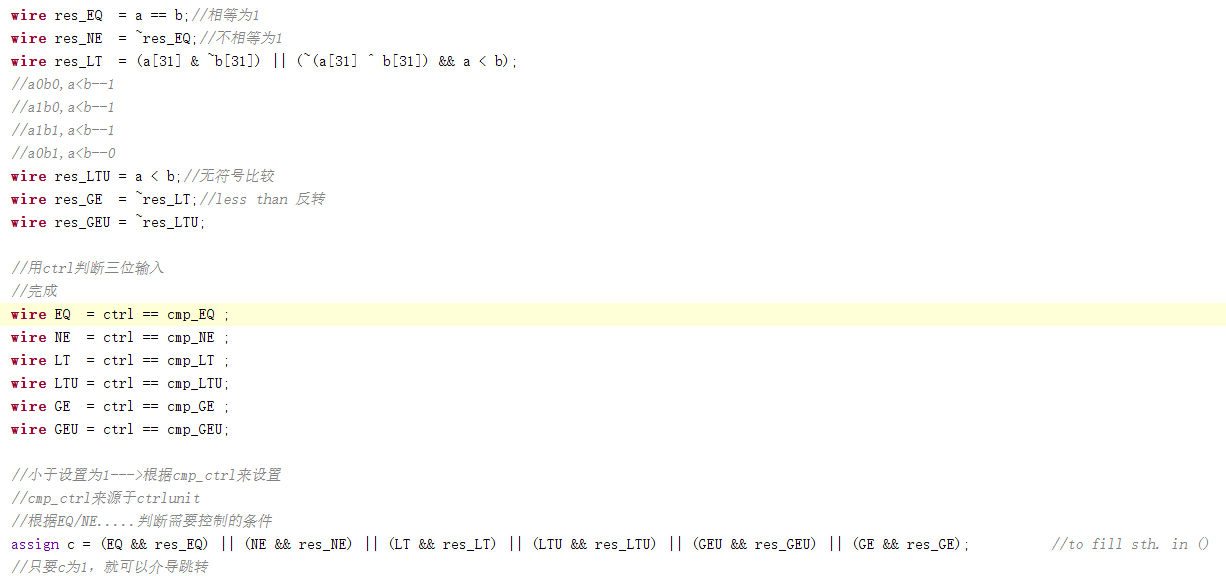
在这一部分，我将主要展现本次实验的设计代码，并对一些设计细节作出详细的解释。

3.1 RV32core模块

在本模块，我们只需要填充一些mux的数据来源与选择线的甄别即可。

由填充的代码可以看出，对于ID阶段的mux而言，我们输入了寄存器的数据，ALU在EXE与MEM阶段的数据以及从内存中获得的数据，并使用在CtrlUnit生成的forward旁路控制信号进行挑选，尽可能解决data hazard的冲突问题。而在指令的EXE阶段，我们则使用了对映的EXE选择信号A与B去选择ALU计算的数据来源，是寄存器还是PC，是寄存器还是立即数，来满足不同指令的多样化需求。

3.2 cmp\_32

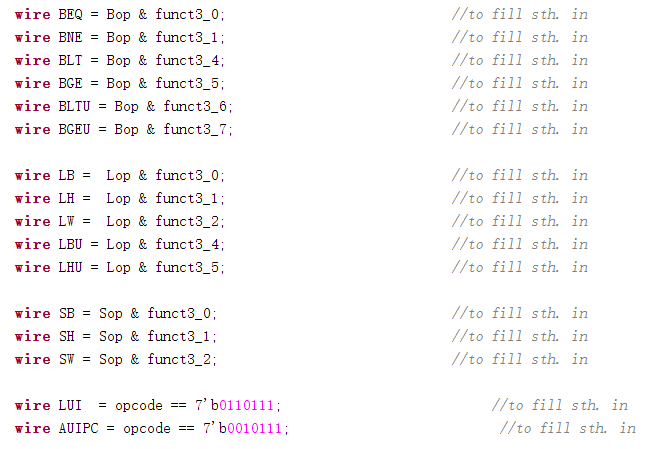
我们直接给出这一模块的设计代码。

从设计中可以看出，我们优先定义了res\_xx这一比较信息结果符号，用于记录相等、不相等、大小于之间的关系比较；我们可以看到，在less than的比较过程中，我们的设置相对复杂，这是因为我们需要考虑无符号数和有符号数的两种比较过程，两者不进相同。

在上述比较模块中，我们也设置了cmp\_xx这一比较信息收集符号，用于与其他模块进行信息连接。cmp信号可以使本模块知道目标指令的预期比较信息，是全等比较、不等比较还是大小于比较等等，从而给出对映的条件比较结果。

3.3 CtrlUnit

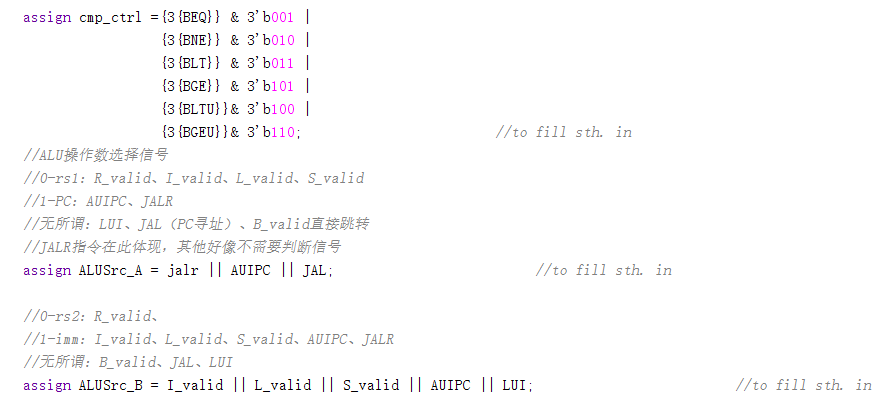
在这一模块中，我们主要需要设计、计算需要的控制信号，用于流水线CPU中不同模块间的通信与控制。

CtrlUnit模块指令判别设计

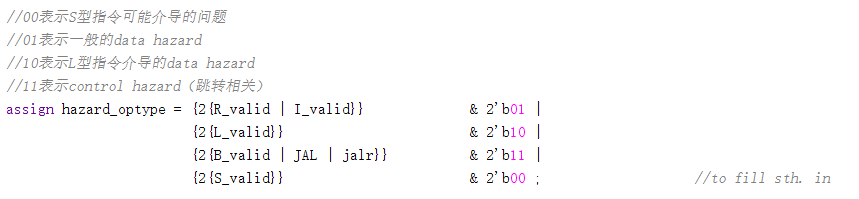
首先，我们可以根据得到的指令信息，将其进行拆解，获得我们需要的对映数据。最直接的，我们可以根据指令的最前7bit以及func3bit与func7bit，来确定指令的具体类型---属于哪一指令大类的哪一指令，从而方便后续的控制信号产生。

另外，对于选择PC的跳转方式而言，我们也需要一个额外的控制信号，来介导PC直接偏移或者基于寄存器的偏移；而对于B型指令的跳转而言，我们则需要考虑比较条件的结果，只有结果符合的，才能正常执行PC跳转的操作。

另外，我们还需要设置一个控制信号，来通信cmp\_32模块我们需要进行比较的条件。在此之上，我们对于EXE阶段进入ALU进行数值计算的数据，也需要经过一定的mux筛选，因此，我们还需要两个选择信号；当然，这两个关于mux的选择信号，只需要和具体指令类型相关即可，同样类型的指令，往往会有相同的数据选择方法。

三个控制信号的具体设计方案

最后，我们需要对hazard进行一个type分类，这个控制信号产生的十分简单，只需要根据不同的指令类别作出不同的2bit信号即可；但是这些信号，可以在hazard处理模块协助我们更好的检测hazard的产生以及使用forward旁路或者插入nop的方式，来解决这些问题。

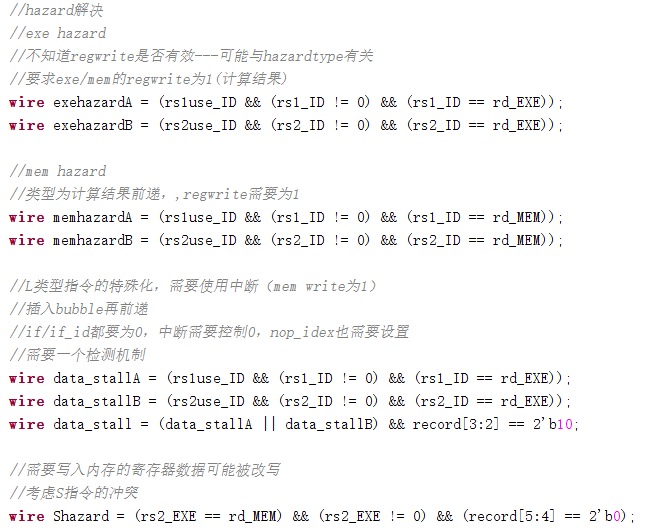
Hazard类型设计

3.4 HazardDetectionUnit

在本模块中，我们有两大任务，在基于已有接口的前提下，我们需要进行hazard产生的判定，以及使用插入nop或进行forward旁路前递数据的方法，来解决这些产生的hazard问题。

因为考虑到S型指令的延迟判定，我选择使用一个reg寄存器存储前两个循环内的指令信息，可以确定是否发生第三类hazard冲突。

其余的hazard冲突判定，都可由2.4部分原理解释得到，通过寄存器的使用，使用寄存器非0，使用寄存器与先前指令目标寄存器一致来判断是否发生hazard。具体的判断条件见下：



另外，对于每一类hazard冲突，我们都要采取不尽相同的处理措施。对于一般的hazard而言，我们只需要使用forward旁路，将数据进行前递即可；但对于L型指令，我们必须刷新前置的几个寄存器，来模拟插入nop的效果，因为只用旁路是无法解决L型指令引起hazard的问题。

同理，B型指令的跳转，若条件符合，也是需要nop的插入的；不然，流水线只会顺序的执行指令，B型指令介导的跳转前会存有一条不应出现的参与指令，会导致错误的产生。

 部分hazard的具体处理细节

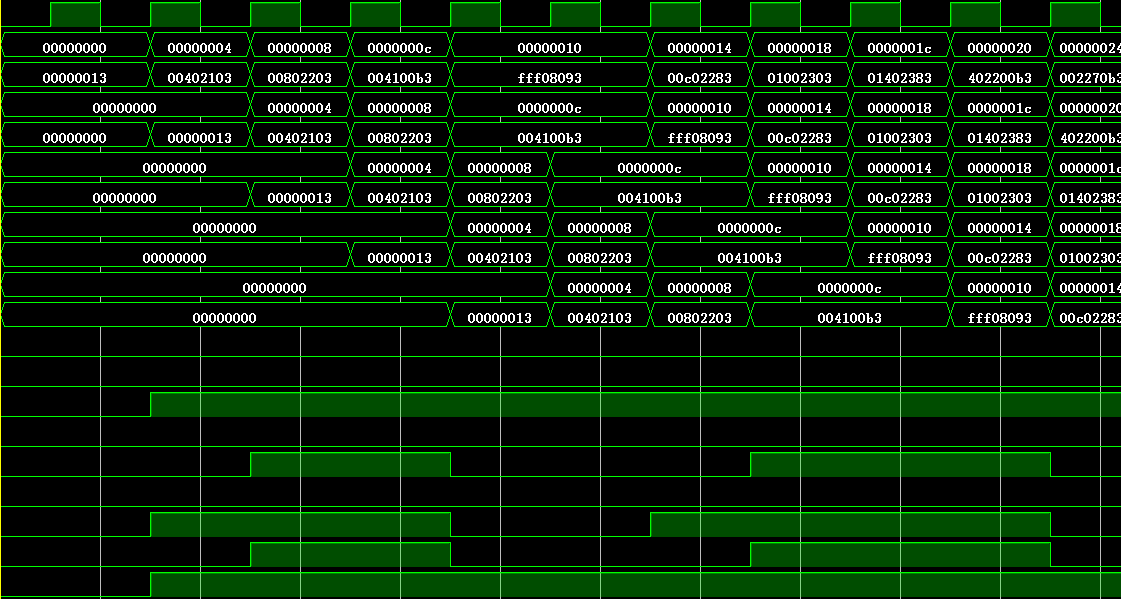
1. 实验结果分析

4.1 仿真结果

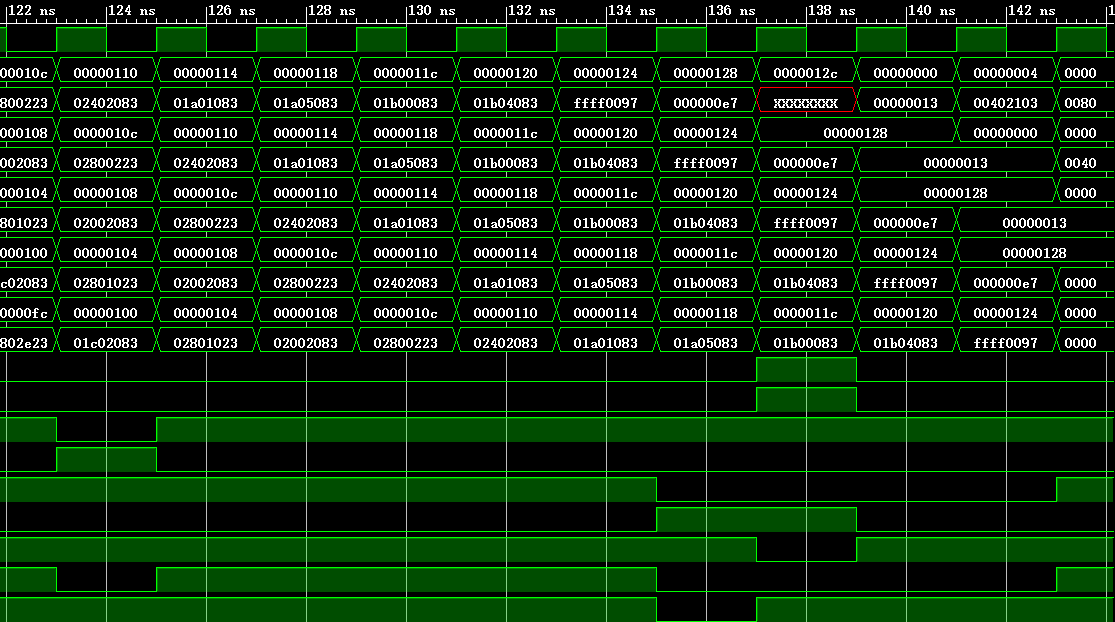
这里给出部分的仿真结果。

我们可以看到，PC指令一直在正常执行，直到PC = x00000010时，处于ID阶段的指令为x004100b3，表示 add x1, x2, x4，但这里x4作为源寄存器，而在这一语句之前，我们进行的指令是lw x4,8(x0)，x4 作为目标寄存器，所以发生了L型指令的hazard。

对于这类hazard，我们必须采用插入nop的方式，才能使PC进行正确且有序的运行；因而我们可以从结果中看到，ID阶段的指令stall了一回合，但是后续的阶段指令依旧稳步后移，等价于一个nop的插入，因而我们的设计具有一定的正确性。

仿真结果示例（1）

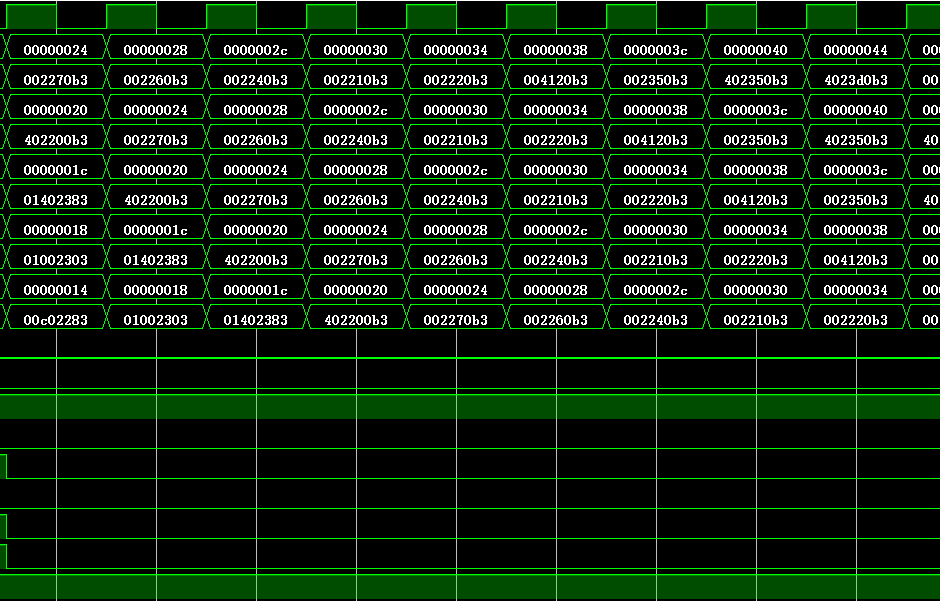
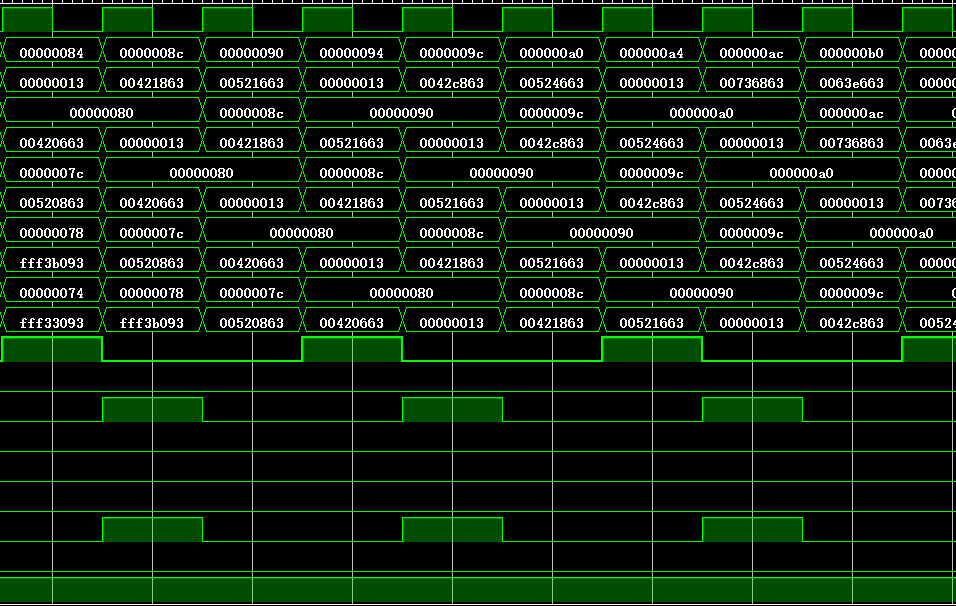
我们再来分析最后的仿真阶段。

如图可见，最后的指令为0x000000e7，对映于jalr x1,0(x0)的指令。根据我们的分析，对于jalr指令而言，我们应该进行无条件的PC指令跳转，并且指令跳转的目标，即为address 0，意味着指令开始的地方。

首先，由结果来看，我们的指令确实从最后一条跳转到了第一条指令，进行了正确的偏移。且在指令执行的过程中，IF阶段因为没有后续指令的读入，确实产生了一回合的错误，但因为hazard模块的检测，我们采取插入nop的方法进行填补；因而在后续对映的阶段，我们在不应出现指令的位置出现了0x00000013的无效指令，从而保证流水线CPU的正常持续运行。

在控制信号方面，因为我们选择了由寄存器偏移进行指令的跳转，因而jalr信号与Branch跳转信号，都被置位为1；因为数据写入寄存器组，从而我们的RegWrite信号也同样被置位为1；从进入ALU计算的单元来看，我们一方面需要选择寄存器来源，一方面需要选择立即数扩展来源；因而这些对映的控制信号，也保持着正确的选择。

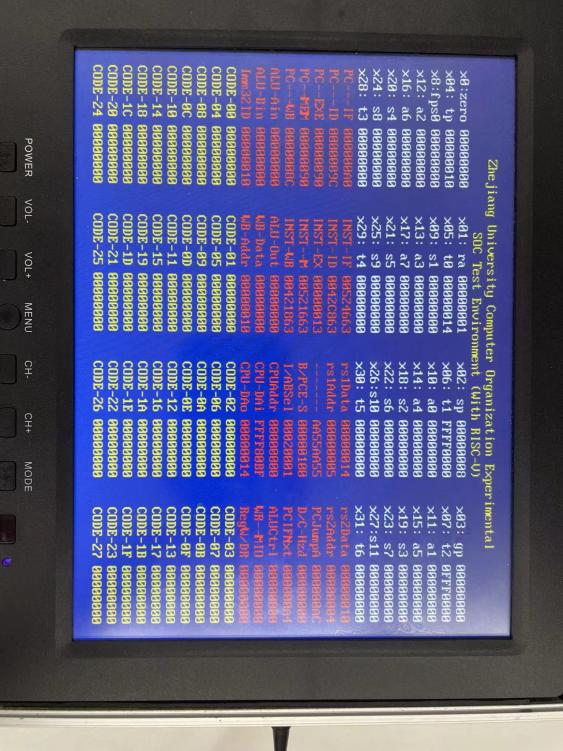
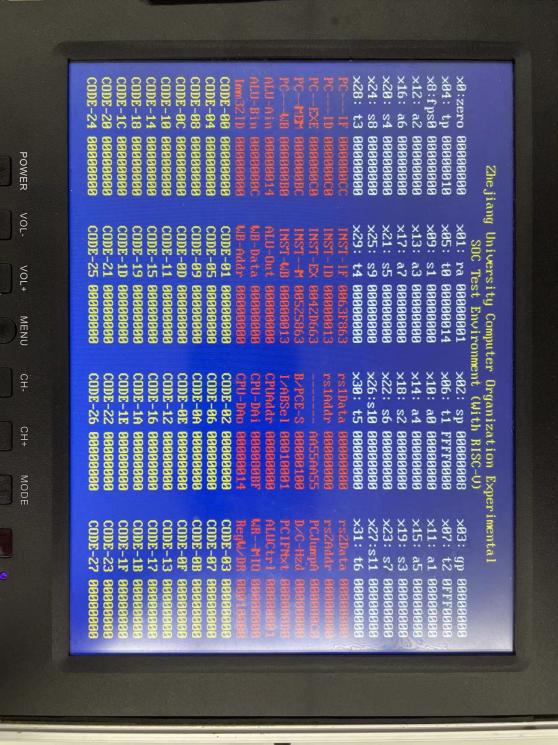
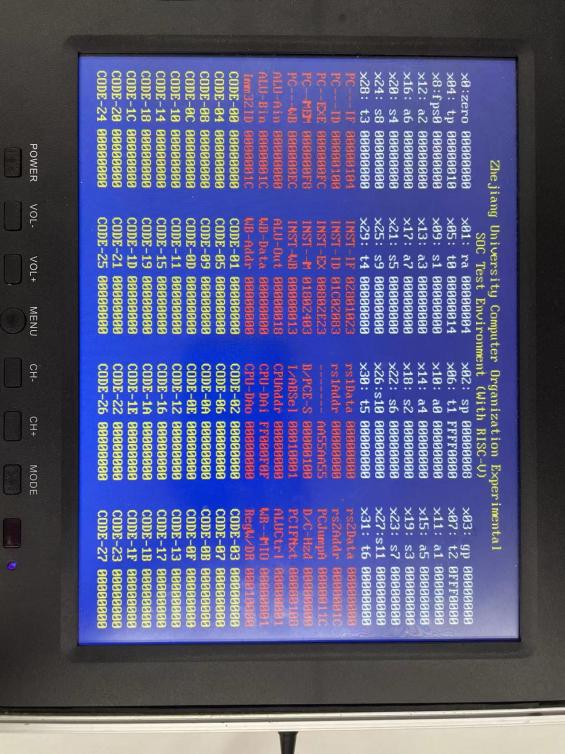
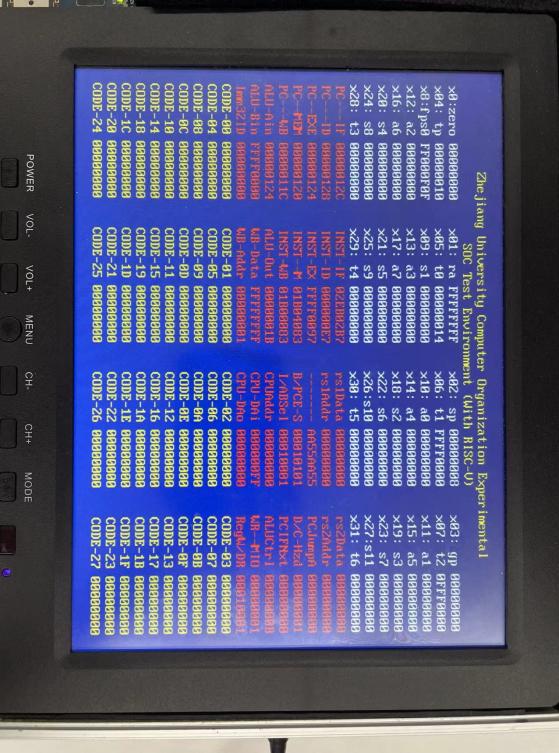
一些其他的仿真过程记录图。



4.2 上板结果

在这里先成列一些实验上板的结果。其实上板屏幕所显示的各类信息，包括寄存器当前数据存储、控制信号当前状态、当前各个阶段所执行的PC指令等等，我们都可以在仿真中得到。

实验上板屏幕显示的正确性，也在一定程度上证明了我们设计的流水线CPU的正确性。因为这里显示的数据与仿真时所记录的数据基本一致，我们就不再对实验结果作详细分析，这里只对结果进行一些陈列。



1. 讨论与心得

本次实验也不是一帆风顺。

首先因为亚运会志愿者的原因，我也没有成段的时间，来很好的解决这个实验。只能通过零碎的午休时间，忙里偷闲，一点一点设计需要的模块，填充大体的框架，最后也是十分激动，完成了含有hazard监测机制与forward旁路前递的流水线CPU设计。

在CPU设计的过程中，我也遇到了不少问题。第一个难题在于控制信号的生成设计上。我们由前分析可得，在介导B型指令跳转时，我们的流水线CPU必须插入一个nop，来防止多余指令的执行，在这里，我为其设置了一个control\_stall来指示是否需要nop的插入；最开始因为判断条件过于严格，导致一些需要跳转的指令，在此也不尽跳转，或发生了其他解决hazard问题的错误操作；经过好久的debug，我才发现了这个问题，将这个控制信号直接与CtrlUnit模块的Branch控制信号对标，只要Branch允许，我们就可以介导跳转，就需要nop的插入，这很好的解决了判定条件过严的问题。

另外一个，则是时序上产生的问题。在HazardDetectionUnit的设计中，我们可以使用CPU clk接口，但在后续的模块设计时，我既使用了always@(\*)的时序出发，也使用了always@(posedge clk)的时序触发；虽然在仿真及implementation时没有出现问题，但在生成bit流的时候，时序混乱却被提醒而禁止。除却一些声明合理的特殊方案，我经过研究，选择使用always@(posedge clk or negedge clk)的时序判断来介导，在保证hazard判断变化足够时，也统一了时序的问题。

完结撒花！