**MIPSSOC设计报告**

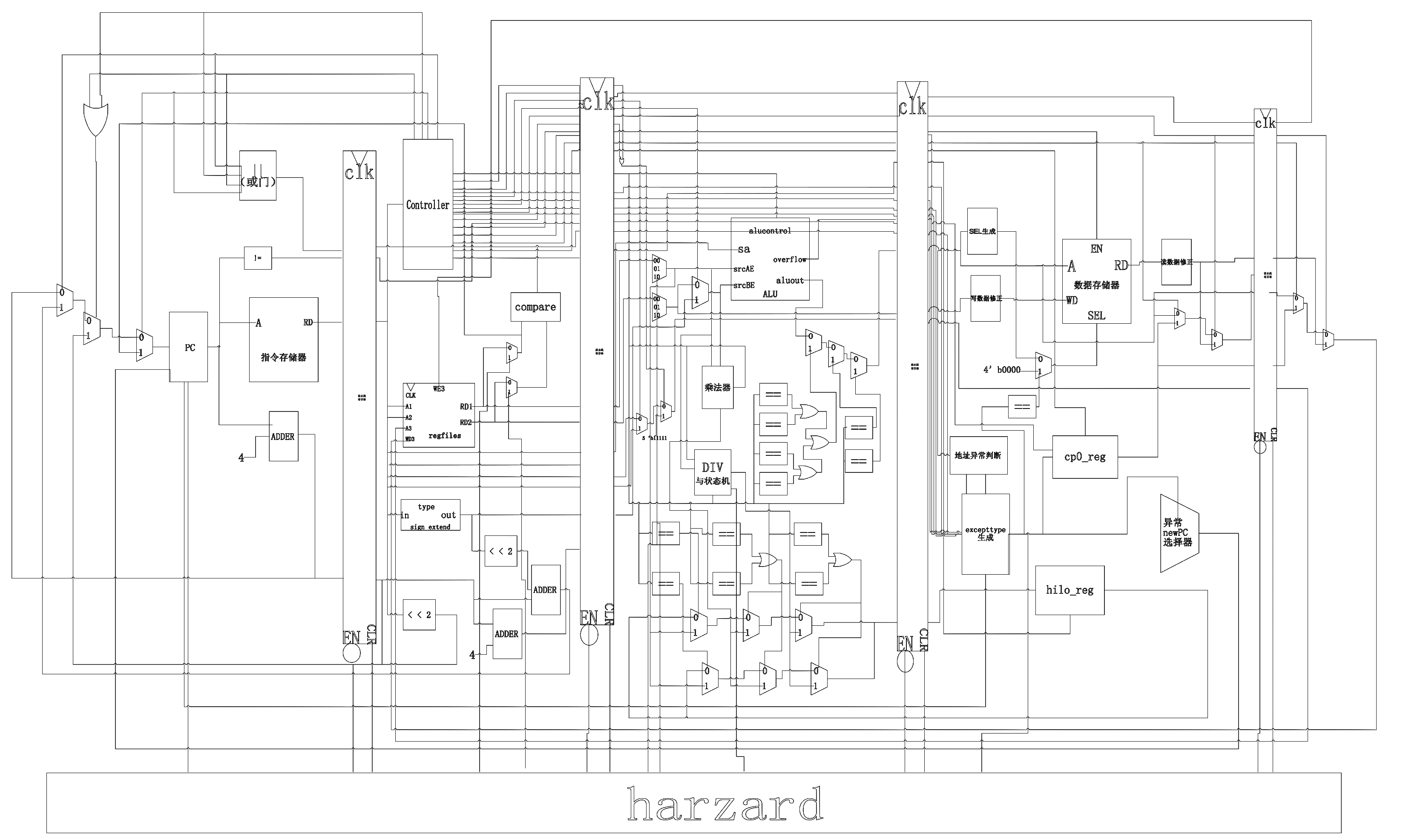
# 1 设计简介

本次小学期我们设计并实现了基于哈佛结构的32位五级流水线处理器。该处理器使用Verilog实现，支持57条指令，连接AXI总线，实现了CP0协处理器，支持中断和系统调用，使用写回四路组相连Cache进行性能优化。

# 2 设计方案

**2.1 总体设计思路**

我们的CPU基于计算机组成与原理的实验四，采用五级流水线形式，整个流水线过程分为取指、译码、执行、访存、回写五个阶段。该CPU主要由AXI模块、数据通路模块、Controller模块、ALU模块、CP0协处理器模块、数据冒险模块、Cache模块、异常处理模块组成。整体数据通路图如下



1. 前52条指令设计：

1、在计算机组成的实验四的基础上，扩充控制器，在controller模块中不断添加新的指令的译码过程。

2、扩充ALU模块，添加针对新的指令功能码的操作。

3、扩充compare模块，使其变为广义比较器，可以根据传入的不同的操作码进行不同的比较。

4、完善数据通路设计，扩展完成各类基础指令，在数据通路中添加了多个多路选择器，如jal、bal信号控制是否要写到31号寄存器，Jr、jal、bal控制写入数据是否是PC+8等。

5、增加Hilo模块，以存储乘除法结果。为避免数据冒险，Hilo寄存器的位置为ALU旁，写信号只可时钟上升沿有效，读为组合逻辑， 随时可读，且接入ALU，在增加了Hilo模块后，需要对数据通路进一步拓展，在执行阶段添加多路选择器，选择执行ALU操作的数据是否来自Hilo模块 。

6、使用项目提供的div除法器模块进行除法运算，在数据通路中添加除法器的状态机。

7、其他一些扩充，如乘法中乘数的修正，会导致溢出指令中的overflow的判断

（2）内陷指令与特权指令设计：

1.增加cp0寄存器存储并处理异常信息。由于取指、译码、执行阶段和访存阶段都有可能出现异常，故方案为每个阶段收集异常信息， 直到M阶段（访存）再处理异常。故cp0寄存器 放在M阶段，写使能在上升沿有效，读为组合逻辑。

2、在M阶段生成异常类型，针对各阶段传下来的异常信息以及cpu0中cpu0status状态进行判断，生成异常类型，将异常类型传入cpu0中。

3、针对生成的异常类型，得到对应的异常处理地址，将其传入cpu0中，在需要处理异常的时候，将pc设为异常处理的地址。

4、由于引入了cpu0，所以对于读写cpu0，依然需要进行数据冒险的处理，对于数据冲突，采用数据前推和暂停两种策略。如果在后续阶段中有尚未被写入寄存器的数据，并且需要将值引入寄存器中，则预先采用所需的后续值。如果在后续阶段中有一些数据需要从数据存储器中获取然后写入，但是当前需要该存储器的值，则流水线将暂停一个周期。由分支跳转指令引起的控制冲突通过延迟槽解决。

（3）AXI接口的处理：

1、在最初，我们采用吕煜峰学长提供的sram转axi转接桥进行处理，并对吕学长的转接桥进一步进行了封装，将仲裁进制等封装到同一个模块内，然后将mips核心模块传出的指令以及数据的sram接口传入到sram转axi转接桥中，即可得到住转换后的axi接口，再将其传入上层。

2、再cache设计时，由于吕学长的桥与体系结构实验的接口相差较多，所以我们改用sram转类sram，再使用龙芯提供的转接桥的方式进行处理，以此更好的复用体系结构实验时已经写好的cache模块。

**2.2 Controller模块设计**

控制器模块端口介绍如下

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 输入/输出 | 作用 |
| clk | 1 | 输入 | 时钟信号 |
| rst | 1 | 输入 | 复位信号 |
| instrD | 32 | 输入 | 输入的指令 |
| equalD | 1 | 输入 | 分支指令的条件判断结果  判断结果 |
| branchD | 1 | 输出 | 判断是否是分支指令 |
| pcsrcD | 1 | 输出 | 分支指令在译码阶段的跳转信号 |
| jumpD | 1 | 输出 | 判断是否是jump指令 |
| jalD | 1 | 输出 | 判断是否是jal指令 |
| jrD | 1 | 输出 | 判断是否是jr指令 |
| balD | 1 | 输出 | 判断是否是bal指令 |
| invalidD | 1 | 输出 | 无效指令的控制信号 |
| syscallD | 1 | 输出 | syscall指令控制信号 |
| breakD | 1 | 输出 | break指令控制信号 |
| eretD | 1 | 输出 | eret指令控制信号 |
| stallD | 1 | 输入 | 触发器暂停信号 |
| memtoregD | 1 | 输出 | 加载指令的输出选择信号 |
| cp0toregD | 1 | 输出 | 是否要从cp0寄存器读数据 |
| alusrcD | 1 | 输出 | 控制传入 alu 的操作数是立即数的扩展 |
| regdstD | 1 | 输出 | 控制寄存器的写入地址 |
| regwriteD | 1 | 输出 | 指令是否要写回寄存器信号 |
| alucontrolD | 8 | 输出 | 控制 alu 要进行的运算类型 |
| hilo\_writeD | 1 | 输出 | 是否要写回hilo寄存器 |
| cp0weD | 1 | 输出 | 是否写cp0寄存器 |
| memenD | 1 | 输出 | 内存使能信号 |
| memwriteD | 1 | 输出 | 数据存储器使能信号 |
| stall | 1 | 输入 | 暂停信号 |

Controller 模块根据传入的指令 inst，解析出指令的类型，给其在数据通路的选择信号赋值，然后将赋值后的信号通过触发器向后传输，当 DataPath 模块中有某阶段用到某信号，Controller 就将该信号传至该阶段，触发器可以根据 DataPath 中的信号选择清空或者暂停。这些控制信号中 PCsrcD 是需要根据 branchD 和 equalD 信号决定，即只有指令是分支指令且分支条件满足才打开。Controller 模块还决定具体的指令需要做何种运算的控制信号 alucontrol。

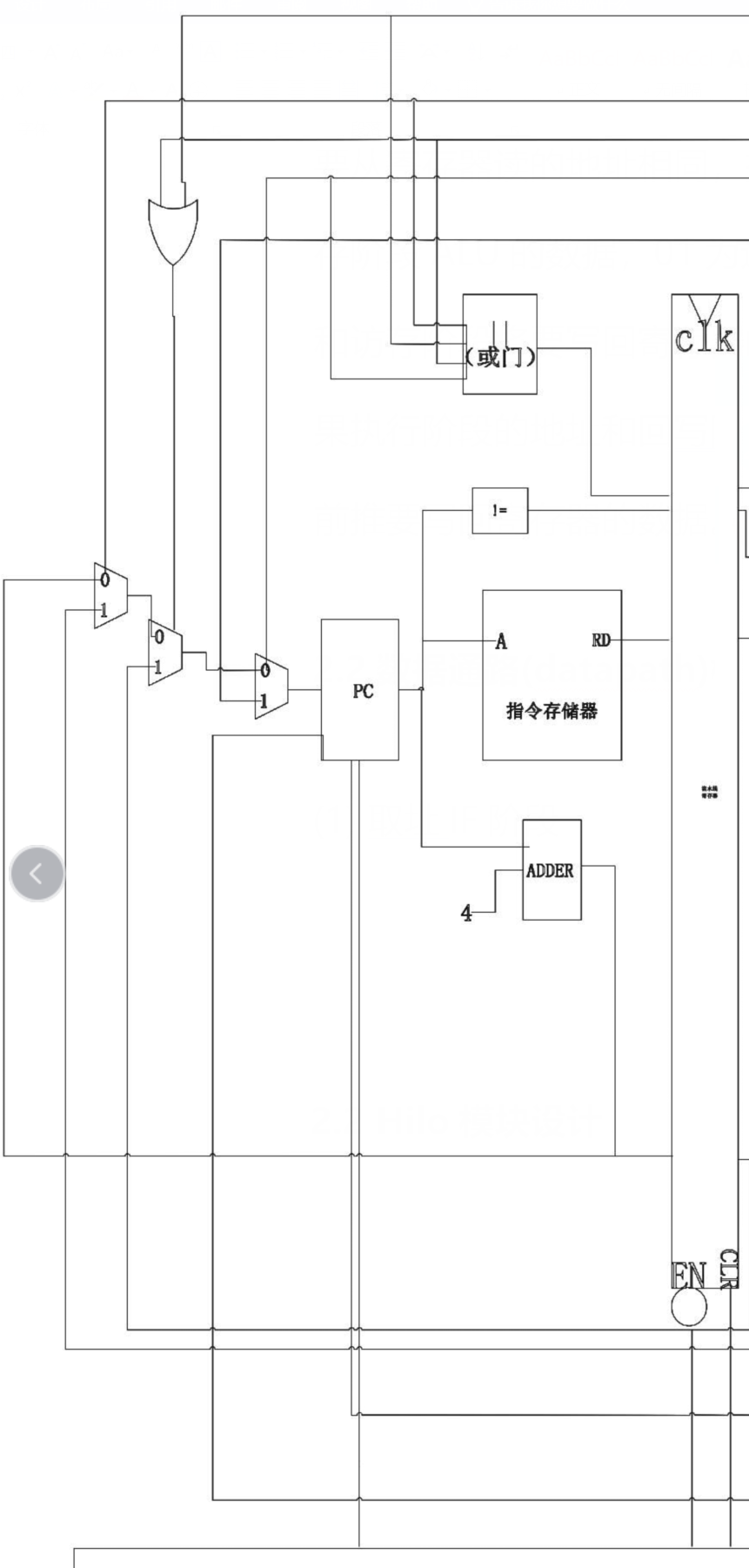
**2.3数据冒险模块设计**

Hazard数据冒险模块主要负责流水线的暂停和清零和数据前推。对于流水线的暂停，考虑除法指令的暂停，lw 类指令的暂停、出现异常与无效指令时的暂停，访存指令导致的暂停以及分支指令前推导致的暂停。对于流水线的清空，除了执行阶段还要考虑是否为 lw 指令外，其他清空的原因都是因为要进行异常处理，收集异常后清空流水线。译码阶段的数据前推，信号regwriteM为访存阶段寄存器堆的使能信号，信号writeregM为写回寄存器的地址，如果访存阶段将要写回寄存器的地址和译码阶段将要从寄存器读的地址相同，把访存阶段的值前推。执行阶段的数据前推，10为选择访存阶段ALU的数据，01为选择写回阶段要写回寄存器的数据，如果执行阶段的地址和访存阶段将要写回寄存器的地址相同而且寄存器使能信号为1，前推ALU数据，如果执行阶段的地址和回写阶段将要写回寄存器的地址相同而且寄存器使能信号为1，前推要写回寄存器的数据。

**2.4数据通路(datapath)模块设计**

(1) 取址IF阶段

取指令阶段即控制单元执行程序的周期内，从存储单元取出所要执行的指令的过程。取址阶段的数据通路图如图一所示



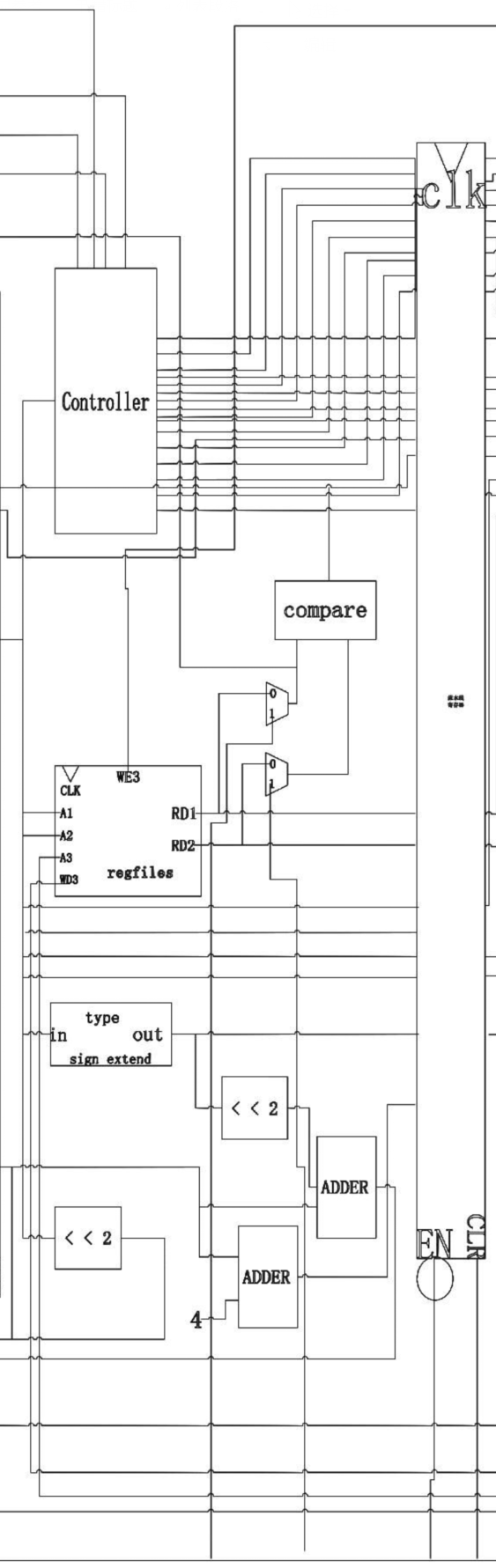
图一

首先使用三个多路选择器选择PC。第一个多路选择器考虑branch的选择条件分支，要么是下一条指令，要么是branch的跳转地址。第二个多路选择器进一步考虑JR的选择条件分支，选择newrd1D为考虑数据前推后的数据。第三个多路选择器再进一步考虑jump的选择条件分支，跳转目标由该分支指令对应的延迟槽指令的 PC 的最高 4 位与立即数 instr\_index 左移2 位后的值拼接得到。然后使用PCmodel模块确定最终的PC地址。该模块中，flush清除流水线，newPC为异常跳转地址，如果清除了流水线，说明遇到了某种异常，就直接跳到异常地址，否则为上述选择器选择之后的PC，最终结果为PCF。pcplus4F为加4得到下一条地址 ，另外要进行异常判断，pc\_not\_duiqiF为PC地址对齐异常，非整字取指。is\_in\_delayslotF确定是否是延迟槽指令。最后通过流水线寄存器，把取址阶段的数据送到译码阶段。

1. 译码ID阶段

取出指令后，计算机立即进入指令译码阶段。在指令译码阶段，指令译码器按照预定的指令格式，对取回的指令进行拆分和解释，识别区分出不同的指令类别以及各种获取操作数的方法。在译码阶段中，译码的工作由Controller完成。Controller负责翻译出指令类型及操作和译码得出需要的具体的算术逻辑运算。根据译码后的地址送入通用寄存器中取出相应寄存器的值，再将这些值送至执行阶段。如果存在数据冲突，即当前需要用到后级还未来得及写回寄存器的值时，则直接采用该值而不是暂停流水线等待值的写回，具体是否前推由数据冒险模块来计算。

译码阶段的数据通路图如图二所示



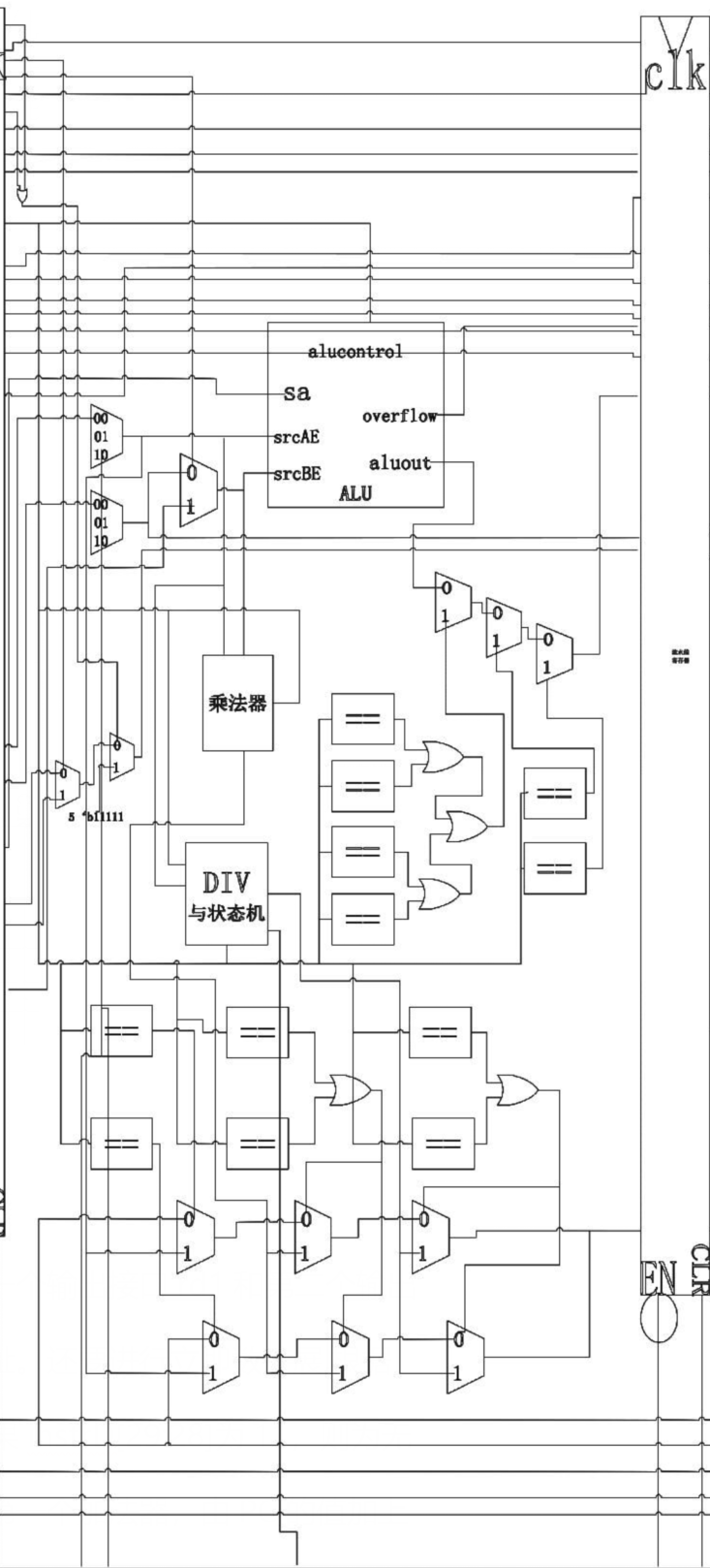
图二

其中寄存器堆根据指令的信号从寄存器中取出第一个输出接口rd1和第二个输出接口rd2的值，resultW为回写阶段写回寄存器的地址。还要进行立即数扩展,根据instrD[29:28]决定是有符号扩展还是无符号扩展，如果instrD[29:28]为11，则为无符号扩展，否则是有符号扩展。有两个加法器，其中第一个加法器，由PC的值加上立即数扩展之后的转移目标得到转移地址，第二个加法器计算延迟槽的下一条指令。

该模块根据数据冒险模块的信号，提前分支预测数据前推,rd1D为寄存器第一个端口的数据，resultM为要前推的数据(从内存中读出来的数据或者从cp0读出来的数据或者访存阶段ALU计算结果)，前推到译码阶段。该模块还包括分支比较模块，对beq,bne,bgez,bgtz,blez,bltz,bgezal,bltzal指令确定是否跳转，结果为equalD 。最后使用流水线寄存器，把译码阶段的数据送到执行阶段。

（3）执行（EX）阶段

执行阶段是我们调用ALU进行真正的计算过程。在该阶段，由alu负责实际的运算。对于普通指令，如 R 型指令、移位指令等，alu 对操作数 a、b 进行运算，将结果输出到 y 端口。对于移位指令，移动位数由输入 sa 给出，移动方向以及是算术移位还是逻辑移位由 alucontrol 控制。对有符号运算，溢出时 overflow 置 1。alu接收从译码阶段传来的 alucontrol确定指令需要的相应的运算，将结果推至访存阶段。同时，如果前级需要运算结果，则立即前推回去以供需要。执行阶段的数据通路图如图三所示

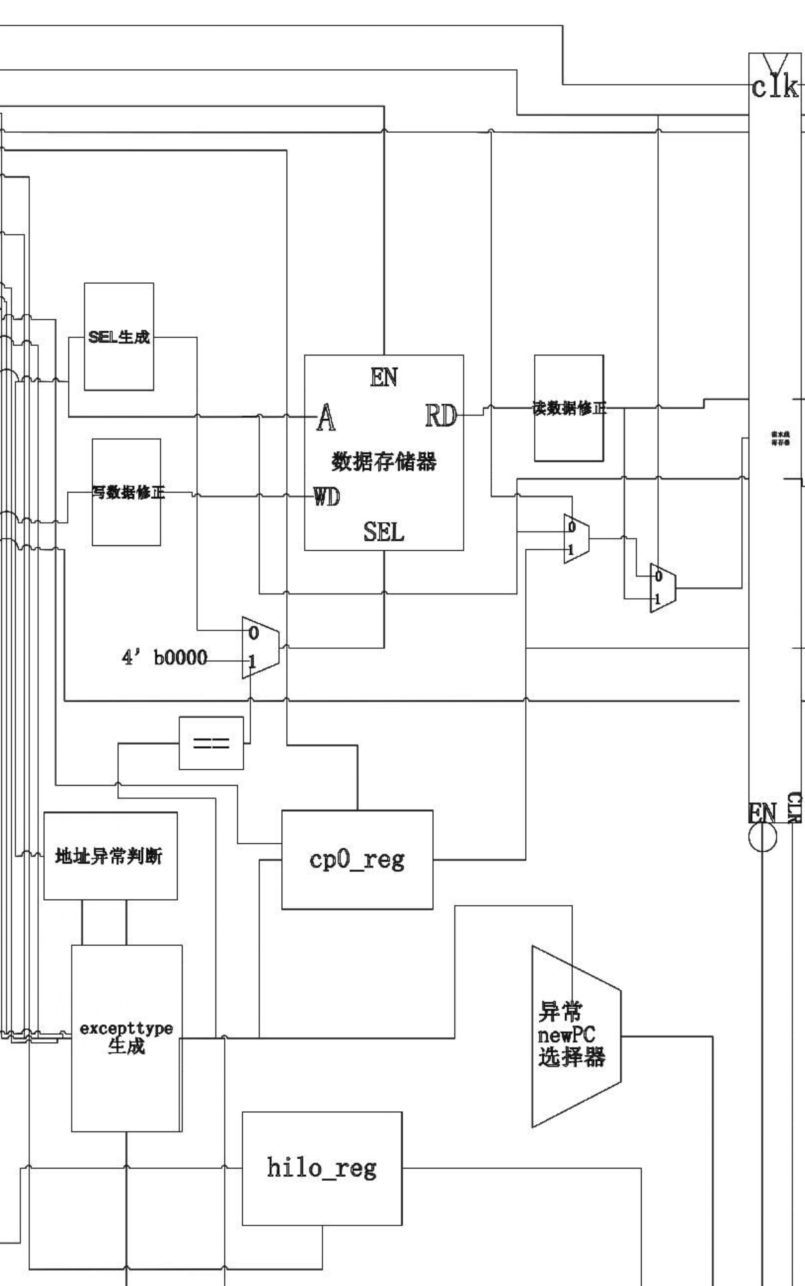


图三

该模块首先使用两个三选一多路选择器处理数据前推的结果，访存阶段ALU数据或者回写阶段要写回寄存器堆的数据前推到执行阶段，算出用于给ALU计算的两个数，再用一个二选一多路选择器选择R-type或J-type指令，如果alusrcE为1，则为跳转指令，要么是立即数，要么是考虑了数据前推执行阶段将要运算的结果B。接下来接入ALU模块，除了乘除法的运算，ALU,mtc0复用输出aluoutE ，srcAE为要运算的第一个数，srcBE为要运算的第二个数，得到运算结果aluoutE\_old和是否溢出结果overflow 。要写入hilo寄存器的内容由六个二路选择器来决定，前两个二路选择器考虑数据移动指令，接下来的两个二路选择器进一步选择乘法器，最后的两个二路选择器再进一步选择除法器，得到最终的要写入hilo寄存器的内容。为了实现容易，本项目的乘法计算直接使用verilog乘号，除法计算使用《自己动手写CPU》的除法器模块。接下来用两个二路选择器判断跳转指令的写回地址，最后使用流水线寄存器，把执行阶段的数据送到访存阶段。

（4）访存（MEM）阶段

在访存阶段，负责存储，加载指令读写内存，异常检验和处理也在这一个阶段进行，即如果存在异常，则刷新流水线并跳至异常处理程序入口，以及CP0中的相关处理。该阶段的数据通路图如图四所示

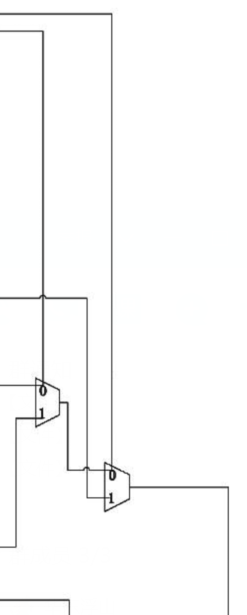


图四

HILO寄存器在访存阶段写回,如果写回信号为1而且没有异常，则写入。访存阶段还对访存指令进行了相关处理。对于加载操作，MIPS32指令集架构中定义的加载指令可以加载字节、半字、字，但是数据总线的宽度是32位，占4个字节。如果执行加载字节指令lb、lbu，那么就要知道通过数据总线输入的4个字节中，哪个字节是要读取的数据；如果执行加载半字指令lh、lhu，那么就要知道哪个半字是要读取的数据，使用一个信号指出哪一部分是有效数据。对于存储操作，MIPS32指令集架构中定义的存储指令可以存储字节、半字、字，但是数据总线的宽度是32位，占4个字节，如果执行字节存储指令sb、半字存储指令sh，那么外部数据存储器就要知道通过数据总线传递过来的4个字节中，哪个字节、哪个半字是要存储的数据，上述信号的作用就是指出哪一部分是要存储的有效数据。部分异常判断和异常处理也在该阶段进行，分别对地址不对齐异常、存数异常 、取数异常进行判断，记录地址相关错误的发生地址，数据存储器访存地址异常时为aluoutM，指令存储器异常为pc，接下来生成异常类型，cp0cause[15:8]的位为1表示产生中断，cp0status[15:8]的位为0表示屏蔽某位对应的中断，cp0status[1]为1为核心态，屏蔽中断，cp0status[0]为0为中断屏蔽态，屏蔽中断，异常跳转地址直接写入PC，送往译码阶段。还要处理数据前推的值，判断要前推的值是从hilo寄存器读出的还是从内存读出的还是由执行阶段ALU传过来的结果。最后使用流水线寄存器，把访存阶段的数据送到回写阶段。

（5）回写（WB）阶段

回写阶段负责将数据写回寄存器中。该模块的数据通路图如图五所示



图五

该模块由两个多路选择器选择写回寄存器的数据，是从内存中读出来的还是从cp0读出来的还是alu的计算结果。

**2.5 Hilo模块设计**

Hilo寄存器位置： E阶段，ALU旁。 模块实现功能：写使能仅在上升沿有效， 读使能始终有效。

**2.6 cp0模块设计**

cp0模块是资料中直接提供给我们了，不需要自己写，不过我们也详细理解了其中的原理。

cp0模块的输入输出端口如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度** | **输入/输出** | **作用** |
| clk | 1 | 输入 | 时钟信号 |
| rst | 1 | 输入 | 复位信号 |
| we\_i | 1 | 输入 | 是否写寄存器 |
| waddr\_i | 1 | 输入 | Rd寄存器为写入 |
| raddr\_i | 1 | 输入 | Rd寄存器为读取 |
| data\_i | 32 | 输入 | Rt的数据 |
| int\_i | 6 | 输入 | 6个外部硬件中断 |
| excepttype\_i | 32 | 输入 | 异常标识 |
| current\_inst\_addr\_i | 32 | 输入 | 指令的PC |
| is\_in\_delayslot\_i | 1 | 输入 | 是否在延迟槽中 |
| bad\_addr\_i | 32 | 输入 | 存储器出错地址 |
| data\_o | 32 | 输出 | mfc0读出的值，存入rt |
| status\_o | 32 | 输出 | 状态寄存器输出 |
| cause\_o | 32 | 输出 | 异常原因 |
| epc\_o | 32 | 输出 | 异常返回地址 |

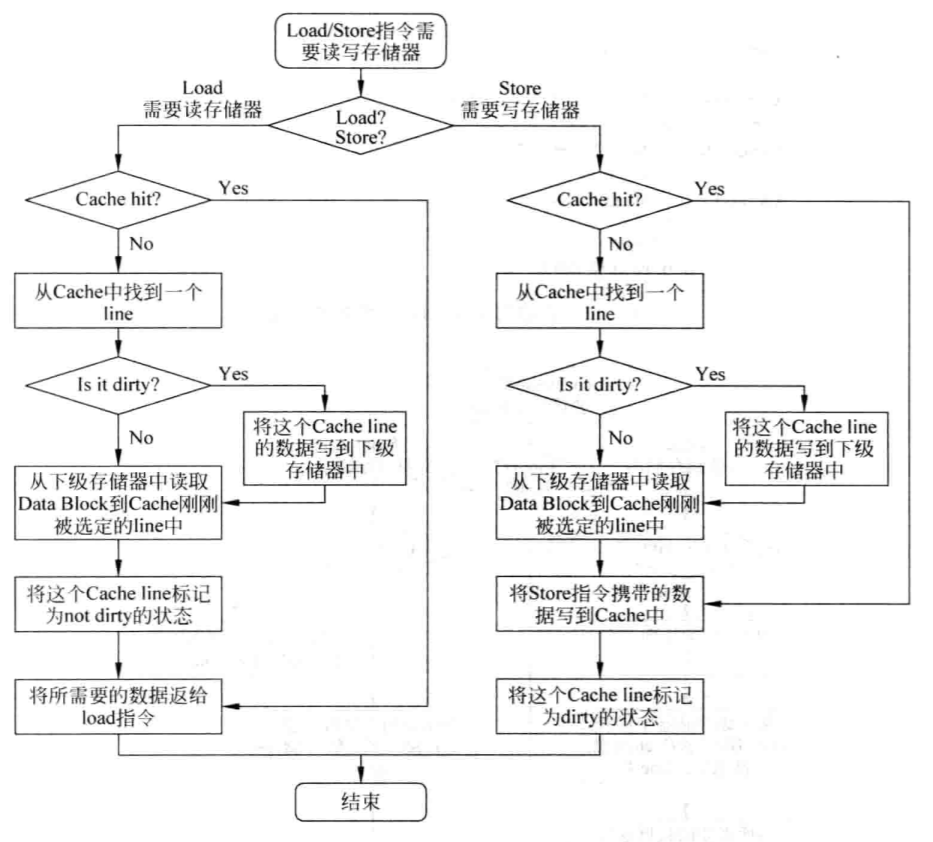
cp0寄存器位置： 访存阶段。

当发生异常或者特权指令eret时，在访存阶段，cp0会将exceptiontype传入冒险处理模块hazard， 并由hazard传给pc新的地址。

其中数据冒险处理与lw指令的处理方式类似，当mfc0指令后遇到一个运算指令并且同时要用到mfc0取出的值时，需要 将cp0取出的数据前推至M阶段。当mfc0指令后遇到分支跳转指令（如bne）并且需要用到mfc0取出的数据时，需要先将F、 D阶段暂停一个时钟周期，再将取出的数据前推至D阶段。

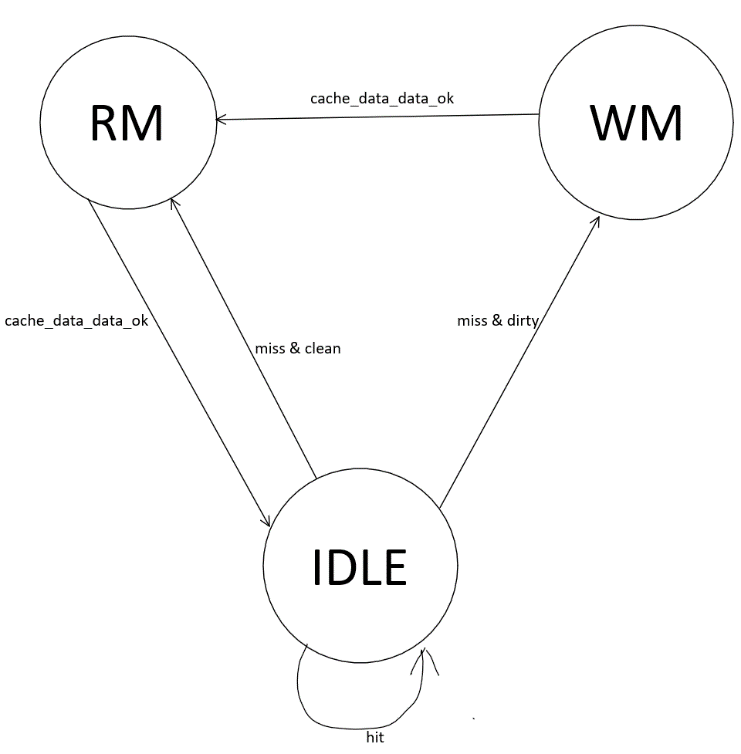
**2.7 Cache模块设计**

为了获得最好的性能，我们实现了4路组相连LRU替换算法的cache。首先设计处理流程图，流程图直接采用了《超标量处理器设计》中的流程图。流程图如图六：



图六

随后根据状态图设计状态机，状态机如图七所示



图七

根据此状态机，完善了状态机部分的代码实现

parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b10; reg [1:0] state; always @(posedge clk) begin if(rst) begin state <= IDLE; end else begin case(state) IDLE: state <= cpu\_data\_req & hit ? IDLE : cpu\_data\_req & miss & dirty ? WM : cpu\_data\_req & miss & clean ? RM : IDLE; RM: state <= cache\_data\_data\_ok ? IDLE : RM; WM: state <= cache\_data\_data\_ok ? RM : WM; endcase end end

cpu传过来的信号要使用寄存器暂存，否则会丢失。代码实现如下：

// 使用寄存器暂存请求 reg [TAG\_WIDTH-1:0] tagsave; reg [INDEX\_WIDTH-1:0] indexsave; reg [31:0] wdatasave; reg wrsave; always @(posedge clk) begin tagsave <= rst ? 0 : cpu\_data\_req ? tag : tagsave; indexsave <= rst ? 0 : cpu\_data\_req ? index : indexsave; wdatasave <= rst ? 0 : cpu\_data\_req ? cpu\_data\_wdata : wdatasave; wrsave <= rst ? 0 : cpu\_data\_req ? cpu\_data\_wr : wrsave; end

参考Cache指导书，实现cache更新部分的代码，流程如下

1 在读命中的情况下，CPU直接读取对应的cacheline的数据；

2 在读缺失的情况，如果索引到的 cacheline 是干净的，那么发送读请求，从内存读取数据，然后返回给 CPU，同时将数据写入到索引到的 cache line 中；如果索引到的 cache line 是脏的，那么首先要发送写 请求，将这个cacheline的脏数据写入到内存中。等待写请求处理完成后，再发送读请求，从内存中读 取对应的数据，然后再把数据返回给CPU，同时将数据写入到索引的cacheline中。

3 在写命中的情况下，如果索引到的 cache line 是干净的，那么直接将数据写入到对应的 cache line 中， 并且将dirty位置为1；如果索引到的cacheline是脏的，直接把数据写入到cache中。

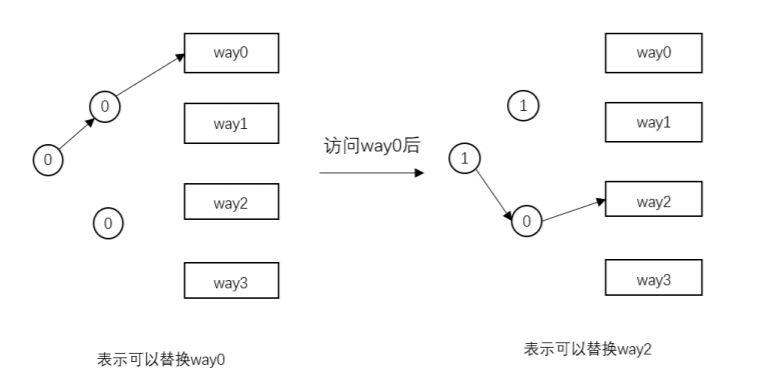
4 在写缺失的情况下，如果索引到的 cacheline 是干净的，那么将数据写入到 cacheline 中，覆盖掉原来 的数据。如果索引到的cacheline是脏的，那么首先发送写请求，将脏的cacheline的数据更新到内存 中；然后等待第一个写请求处理完成后，然后将数据写入到索引到的 cache line 中，并且将脏位标志位置为1；

我们可能会多次将数据写入到某个相同的地址。针对这种情况，我们可以发现，写回策略仅需要在 被替换出去的时候访问内存，而写通策略每次写操作都要访问内存。所以写回--写分配的策略有助于提升 cache性能。

实现伪LRU+4路组相联：

4路组相联：主存和Cache都分组，主存中一个组内的块数与Cache中的分组数相同，组间采用直接映射，组内采用全相联映射。在这里我们将Cache分成4组，主存块存放到哪个组是固定的，至于存到该组哪一块则是灵活的。即主存的某块只能映射到Cache的特定组中的任意一块。

伪 LRU：该部分参考了《超标量处理器设计》中的近期最少使用法和实验指导书中的伪LRU算法的说明。伪 LRU 算法确定替换哪一路的算法类似于一个二分查找，每次通过 1bit 的信息可以将范围缩小到原本的一半，若干次迭代后便可以确定到具体的某一路。具体算法如下：从根节点（1 号节点）开始，如果其值为 0，则表示其右子树（对应的路，即 0 路，1 路）最近没有访问过。反之，则表示其左子树最近没有被访问。假设 1 号节点为 0，则我们通过判断 3 号节点的值来判断替换哪一路：如果为 0，替换 0 路，如果为 1，替换1 路。假设 1 号节点为 1，我们便通过 2 号节点来判断替换哪一路，过程类似。当访问了 cache 的某一路后，需要对 bit 信息进行更新。更新的算法一句话来说就是保持每一 bit 的语义不变，即每一 bit 代表其左右子树最近是否被访问过。具体来说其实只需要更新访问路径（从根节点到某一路的连线）上的节点的状态值即可。（并非直接取反，而是保持语义）。示意图如图八所示



图八

**3.2 错误记录**

**3.2.1 错误 1**

（1）错误现象：仿真时直接秒输出pass，再运行就一直死循环，pc无变化。

（2）分析定位过程：从仿真波形中发现指令无法正常读取，但始终无法定位到错误原因，后经询问同学，解决问题。

（3）错误原因：ram中inst\_coe文件未正确加载。

（4）修正效果：正确加载后可以正确跑动仿真。

**3.2.2错误 2**

（1）错误现象：除法指令无法正确执行得到结果。

（2）分析定位过程：从仿真波形可以看出 ，start\_divE已经正常输出，除法器在接收到start\_divE信号后开始执行，但不能得到正确结果，再将输出信号全部拉出查看，发现div\_result已经正确得到结果，可是流水线被卡住了。

（3）错误原因：除法状态机转移时忘记在div\_readyE后将stall信号拉低，导致流水线一直阻塞。

（4）修正效果：修改后可以正确得到结果。

（5）归纳总结：做硬件设计一定要小心认真，因为不好纠错。

**3.2.3 错误 3**

（1）错误现象：加入cache后pass了第一个测试点，第二个测试点卡住。

（2）分析定位过程：从仿真波形中发现指令无法正常读取，但始终无法定位到错误原因，参考PPT之后修复了地址映射，但是没效果。

（3）错误原因：应该是Cache写错了，但debug一天，一直未找到原因。

（4）修正效果：抛弃原来的Cache代码，换队员重构了Cache后跑通。

（5）归纳总结：debug太难了，还是换人重构吧。

**3.2.4 错误 4**

（1）错误现象：接上sram转axi转接桥后，pc卡住，无法正确取出指令。

（2）分析定位过程：观察仿真波形图，在接入了axi转接桥后卡住，必定是转接桥 的问题，而axi转接桥共有两个输出信号会导致流水卡住，分别是stall\_from\_inst和stall\_from\_ mem，观察波形发现pc卡住后,stall\_from\_inst为低电位，stall\_from\_ mem为高电位，锁定问题为stall\_from\_ mem，再观察可能导致stall\_from\_ mem为1的原因，发现stall\_from\_mem= data\_sram\_en & m\_d\_ready，原来是这个忘了取反，导致数据准备好后无法解除阻塞状态。

（3）错误原因：m\_d\_ready前忘记取反，应改为stall\_from\_mem= data\_sram\_en & ~m\_d\_ready。

（4）修正效果：解决pc阻塞问题 。

**3.2.5 错误 5**

（1）错误现象：跑仿真测试时pc执行到bfc00394时报错

（2）分析定位过程：打开test.S文件搜索该pc，发现其是一条beq指令：beq k1,k0,bfc003f0 ，定位到执行该指令的波形图，发现k1和k0的值是相等的，但是却没有发现跳转，pcSrc为低电位，显然这里出现了问题。定位到代码中的beq部分，译码没有问题，最后发现是Compare模块中对beq指令的比较出现错误，将a==b写成了a!=b，搞反了关系，导致没有跳转

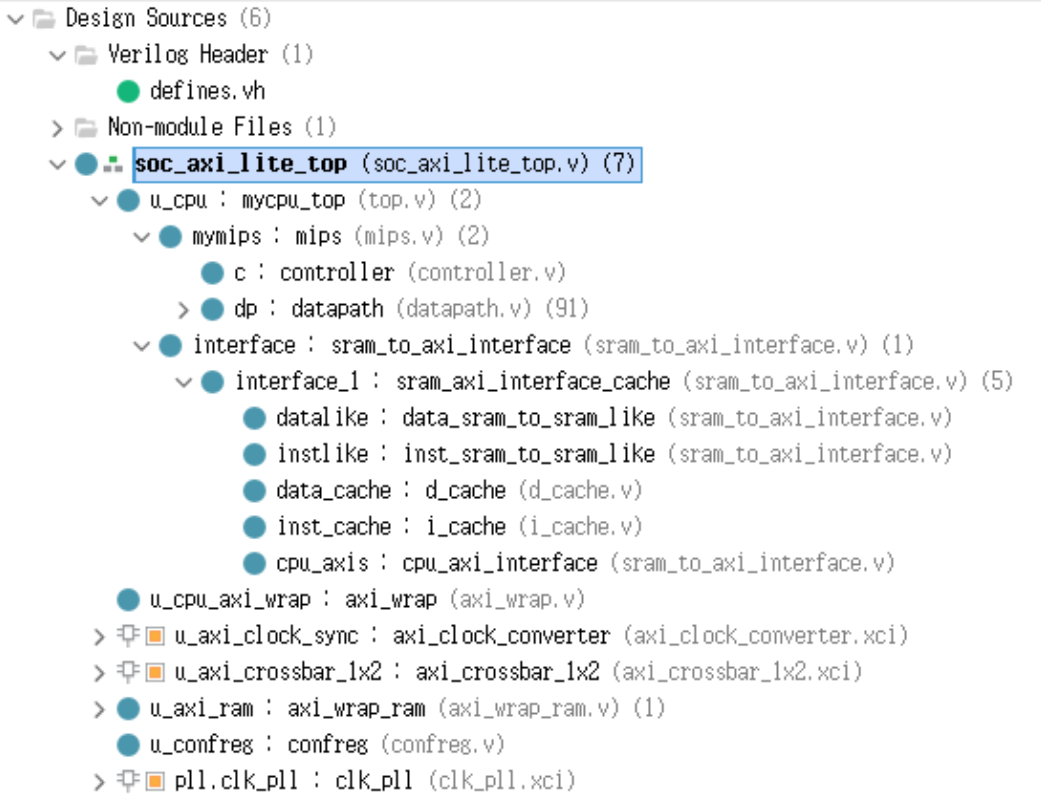
（3）错误原因：Compare模块中对beq指令的比较出现错误，将a==b写成了a!=b

（4）修正效果：修改后解决问题，这里正常跳转

# 4 设计结果

**4.1 设计交付物说明**

总体目录结构如图九所示



图九

defines.vh是宏定义,mycpu\_top是处理器的顶层模块，包含数据通路datapath、控制器controller、AXI模块，Cache模块（i\_cache与d\_cache）。控制器中指令统一译码。指令在译码阶段时需要通过controller的译码，并且算出相应的控制信号。datapath为数据通路， 其中包括取址、译码、执行、访存、写回五个流水阶段、hazard、alu运算器、除法模块、 hilo寄存器、异常处理模块以及选择器等。hilo寄存器存放的是乘除法的64位结果， 将它安排在执行阶段的原因是乘除法都在执行阶段算出结果所以可直接将结果写入无需等待流水线。除法执行完无效的时刻(上升沿)写入，同时在异常处理时不写入，且不存在数据冒险的问题。

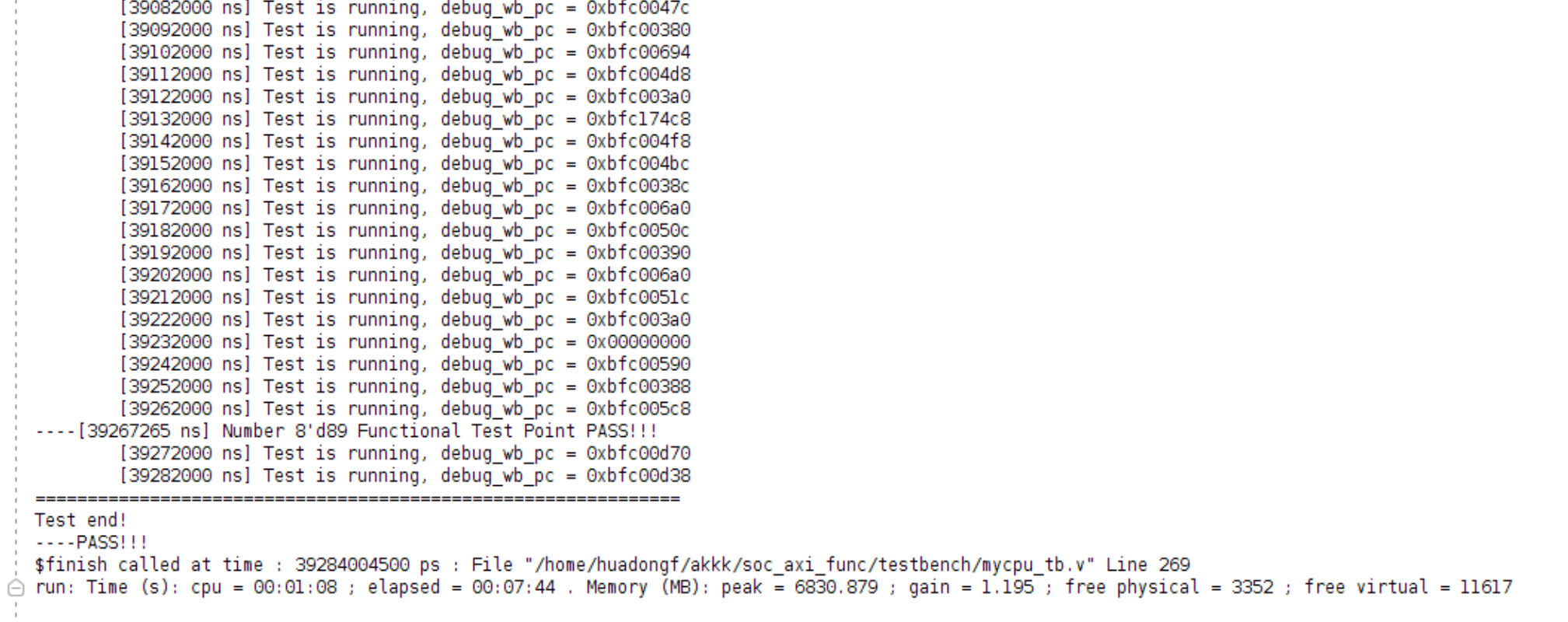
仿真步骤：在仿真文件中更改trace文件路径、u\_axi\_ram中更改相应的coe文件。

上板步骤：连接开发板，在n4ddr文件夹中添加代码，u\_axi\_ram中更改相应的coe文件，生成比特流后上板。

计算性能分步骤: 把十个测试点的结果输入表格中计算性能得分。

**4.2 设计演示结果**

如图十所示，成功跑通89个测试点



图十

成功上板后性能测试结果如图十一，性能分2.552



图十一

# 8参考文献

[1] 雷思磊. 自己动手写*CPU*. 电子工业出版社, 2014.

[2] 姚永斌. 超标量处理器设计. 清华大学出版社,2014.