# 目 录

# 基于 MIPS32R1 标准的 CPU 设计：基本指令

## 一、实验目的

本次实验将结合 2020 年龙芯杯比赛的要求，完成基于 MIPS32R1 标准的 CPU 设计。这里只实现基本指令，即：所有的运算指令、位移指令、分支跳转指令、数据移动指令（协处理器相关指令除外）和访存指令。与协处理器相关的指令及异常、TLB、MMU 等其他 CPU 设计的有关部分将在本次实验之外完成。

## 二、实验原理

### 1. CPU 的流水线结构

本报告中设计的 CPU（下称 HikariMIPS）将使用 MIPS 架构传统的五级流水线结构，各级分别是：取指 IF、译码 ID、执行 EX、访问 MEM 和写回 WB。相比单周期 CPU 的设计，流水线能够更充分的并行利用 CPU 内部不同功能的器件，使得 CPU 执行指令更有效率。同时 HikariMIPS 采取指令与数据分开存储的结构。

#### 取指阶段 IF

取指阶段将从外部只读存储器（下称 ROM）中读取一条指令，并更新程序计数器（下称 PC）指向下一条指令。HikariMIPS 访问外部存储器时按照字节寻址，每次对齐地读入 32 位，即访问[0x00, 0x03]中任意一个地址，都返回地址为 0x00~0x03 的 4 字节数据，这也是为了迎合 MIPS32R1 指令集中要求访存与取指的地址都必须对齐的要求。

此外该阶段还负责响应由译码阶段传来的跳转指令：译码阶段给出跳转信号与新的程序地址，该阶段响应信号并将新的地址写入 PC，随即（一个时钟周期内）便按照新的 PC 进行取指。

#### IF/ID 锁存器

由于一个时钟周期内 IF 分别要完成拿取新的指令及完成 PC 的更新，而后续电路要求前一阶段提供的 PC 值与指令值在一个时钟周期内是保持不变的：如果跳转指令在计算新的 PC 值时 IF 阶段

跳转 PC 到了下一条指令，那么新计算出来的目标 PC 值也就不对了，而在设计 CPU 的时候何时更新 PC 完全是根据硬件寄存器的延迟决定的,因此避免这种窘况的最好方法就是在 IF 和 ID 阶段之间插入一个锁存器，将 ID 阶段所需要的各种信号锁存并保持一个时钟周期。

**译码阶段 ID**

本阶段主要将取到的 32 位指令解析或翻译成后续各阶段的控制信号：例如通过指令的操作码与功能码确定指令要进行的运算，从而给出执行阶段 ALU 的控制信号。本阶段还将获取各指令所需要的操作数（0~2 个，由指令定义），从寄存器堆中读取或按指令要求提取立即数并进行扩展后随控制信号一并交给后续阶段。同时本阶段还负责进行条件的判断：例如分支跳转指令有条件的，将在本阶段测试条件并决定是否跳转；带条件数据转移指令（MOVN 和 MOVZ）也将在本阶段测试条件并根据结果判断是否要进行数据转移。

**ID/EX 锁存器**

与 IF/ID 阶段锁存器类似，但除了锁存并保持必要的信号满一个时钟周期之外，该锁存器还要利用其一拍延迟锁存状态信号为下一条指令提供指示：例如下一条指令是否处在延迟槽中，下一条指令是否被无效化（Nullify，当 Branch Likely 指令条件测试失败时将不执行跳转和延迟遭指令，因此需要无效化其延迟槽位置的指令）。

**执行阶段 EX**

本阶段将按照 ID 产生的信号对数据进行操作，即运算。CPU 支持的全部运算将在这一阶段执行，其中简单运算，如逻辑运算、位移运算、加减乘等都是利用 Verilog 内置运算符完成，综合时将自动产生相应电路。除法则使用试商法构建多周期除法器进行运算。多周期运算执行时需要暂停流水线等待运算结束。

**EX/MEM 锁存器**

与 IF/ID 锁存器类似，除了锁存必要信号与提供下一条指令的部分之外，该锁存器还需要在流水线暂停时为 EX 阶段提供多周期指令执行时状态机所需要的必要信号：例如 MADD 实现为二周期指令，第一个周期计算乘法，第二个周期与 HILO 寄存器累加。当前所在周期是依赖于本锁存器传递的。

### 访存阶段 MEM

MIPS32R1 为精简指令集，其指令设计十分简约：所有指令的操作数来源都只能来自于 CPU 内部的寄存器，只有访存指令能够将数据在内存与寄存器之间移动。因此本阶段将实际执行访存指令，与外部存储器交互。目前 HikariMIPS 假设运行环境是理想环境，即外部存储器能够及时的读写数据，但该模块可以申请流水线暂停，等待存储器完成操作后恢复流水线运作。同时由于目前还没有实现异常等依赖于具体 CPU 具体实现的特性，此时如果访存的地址没有对齐，则不会引发异常，只会使得访存指令变为一个 NOP。

### MEM/WB 锁存器

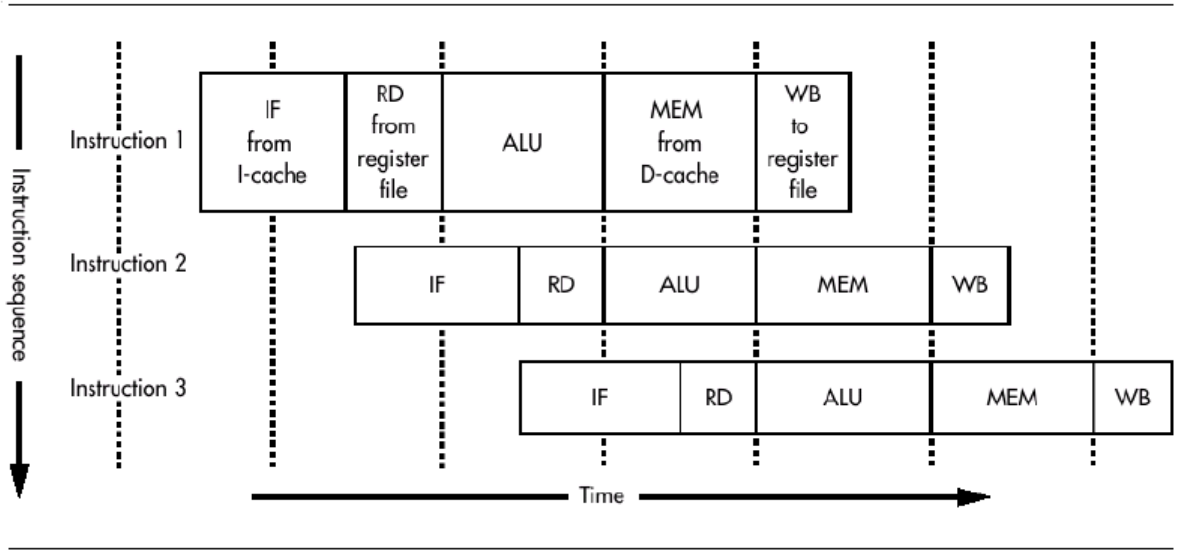
该锁存器传递由 MEM 传递来的信号。

### 写回阶段 WB

所有指令的数据写回将在该阶段完成。目前可写入的目的地只有寄存器堆与 HILO 寄存器。LLbit 我将其认定为是依赖于具体实现的寄存器，不同的 CPU 核心数将影响 LL 与 SC 指令的行为。

在空间上这个阶段并不存在实际的电路，因为实际的写入操作是在寄存器中完成的，只是在时间上单独划出一拍供寄存器写入数据。

## 2. 流水线执行指令



流水线的工作原理可以在上图中形象体现：每个指令依次流过取指、译码、执行、访存、写回五个阶段，每一阶段只完成一个工作，经过五个阶段之后则一条指令被执行完毕。一般情况下每个指令将在 IF 阶段从 ROM 取出，随后在 ID 阶段（图中即 RD）译码产生控制信号，并在该阶段获取指令所需的操作数，按照指令的内容，实际运算将在 EX 阶段指令，访存类指令将在 MEM 阶段执行，最终所有指令的结果将在 WB 阶段实际写入目标寄存器。

### 3. 数据相关问题

在 HikariMIPS 的设计下，众多相关问题只有写后读会影响流水线的正常运行。

考虑如下情况：第一条指令在 EX 阶段，计算结果要写回$1 寄存器；第二条指令在 ID 阶段，需要使用$1 寄存器的内容作为参数，而第一条指令的计算结果要到 WB 阶段才实际写入寄存器，而此时第二条指令已经进入 MEM 阶段，很显然已经晚了。

这种情况解决方法有二：其一是暂停流水线，使第二条指令在 ID 阶段等待，待第一条指令执行完 WB 后再恢复第二条指令的执行，此时第二条指令在 ID 阶段获取到的数据便是正确的；其二是硬件上从 EX 阶段将写入的数据与寄存器编号反馈到 ID 阶段，在 ID 阶段对比是否有读写重叠，若有则优先使用 EX 阶段反馈来的最新数据，否则还是去寄存器堆里读。很显然第二种解决方法虽然增加了电路的复杂性，但是很大程度上节约了时间，减少了流水线空闲的时间，更具有效率。

再考虑另一种情况：第一条指令在 MEM 阶段且不是访存指令，要写入$1 寄存器；第二条指令是 NOP，在 EX 阶段；第三条指令在 ID 阶段，需要使用$1 寄存器的内容作为参数。这种情况与上面的情况有些类似，解决方法也很相似：除了暂停流水线，可以在 MEM 处反馈给 EX 当前指令要写入的数据与寄存器编号，ID 先判断本条指令是否与 EX 阶段有读写重叠，没有则再检查与 MEM 的指令是否有读写重叠（这样的顺序是因为 EX 阶段产生的数据总是比 MEM 阶段的新），以此来解决数据相关的问题。

再考虑第三种情况：第一条指令在 WB 阶段，正要写入$1 寄存器；第二、三条指令是 NOP，分别在 MEM 和 EX 阶段；第四条指令在 ID 阶段，需要使用$1 寄存器的内容作为参数。这样的情况与上面颇有几分相似，但是解决方法却不同。这种情况的重点在于 WB 阶段的行为。HikariMIPS 保证 WB 阶段执行完毕后数据一定会写入到寄存器中，而 WB 阶段中则是写入数据的时间，不能保证下一时钟有效沿来临之前数据被写入，但可以设计寄存器堆为写优先模式：即一个时钟周期内保证写请求优先于读请求被处理。对外的表现则是多个读请求读到的一定是写请求写入的最新数据。由此可以解决这种相关。HikariMIPS 中所有数据寄存器都是写优先模式。

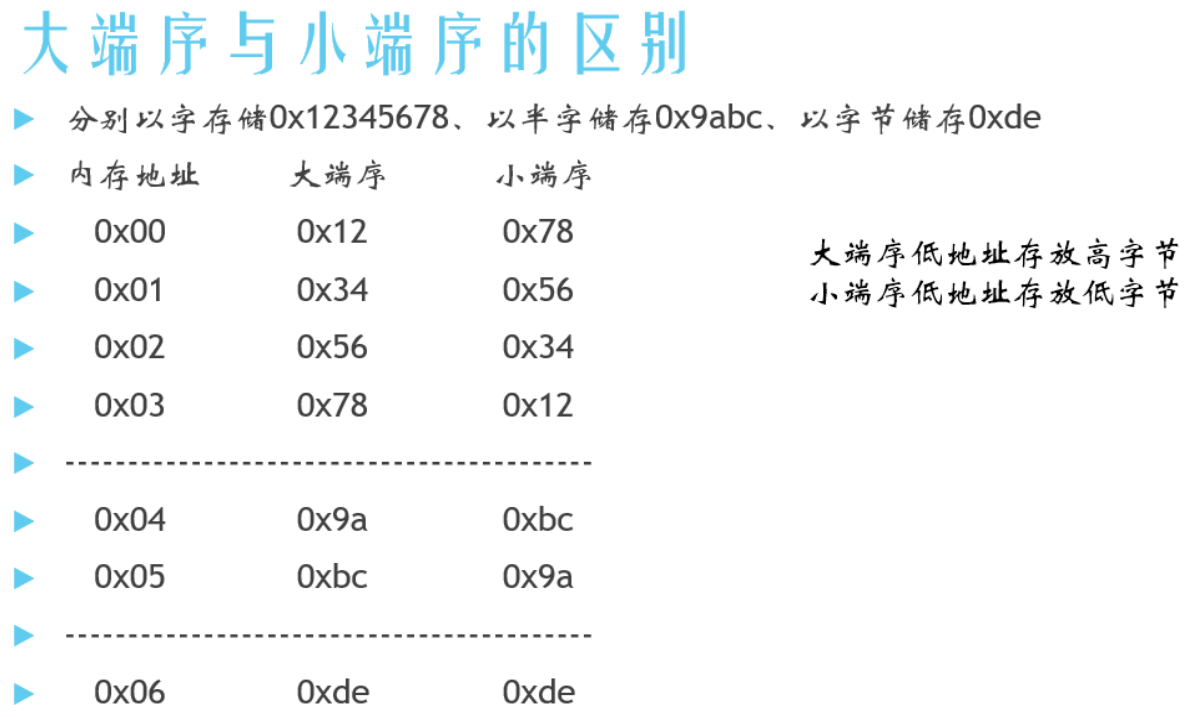
最后再考虑一种访存引起的数据相关：第一条指令是访存指令，要读内存到$1 寄存器，第二条指令在 ID 阶段，需要使用$1 寄存器的内容作为参数。此时无论第一条指令在 EX 或 MEM 中的哪一个阶段，它都只能在 MEM 阶段才能获取到实际要写入$1 的数据。因此若此时第一条指令在 MEM 阶段，则万事大吉，上面的情况已经可以解决了。但是如果第一条指令在 EX 阶段，此时便无法获知$1 的真实内容，因此只能暂停流水线，让第二条指令在 ID 阶段等待，直到第一条指令执行到 MEM 阶段后再通过数据前推来解决。

### 4. 分支跳转指令带来的问题

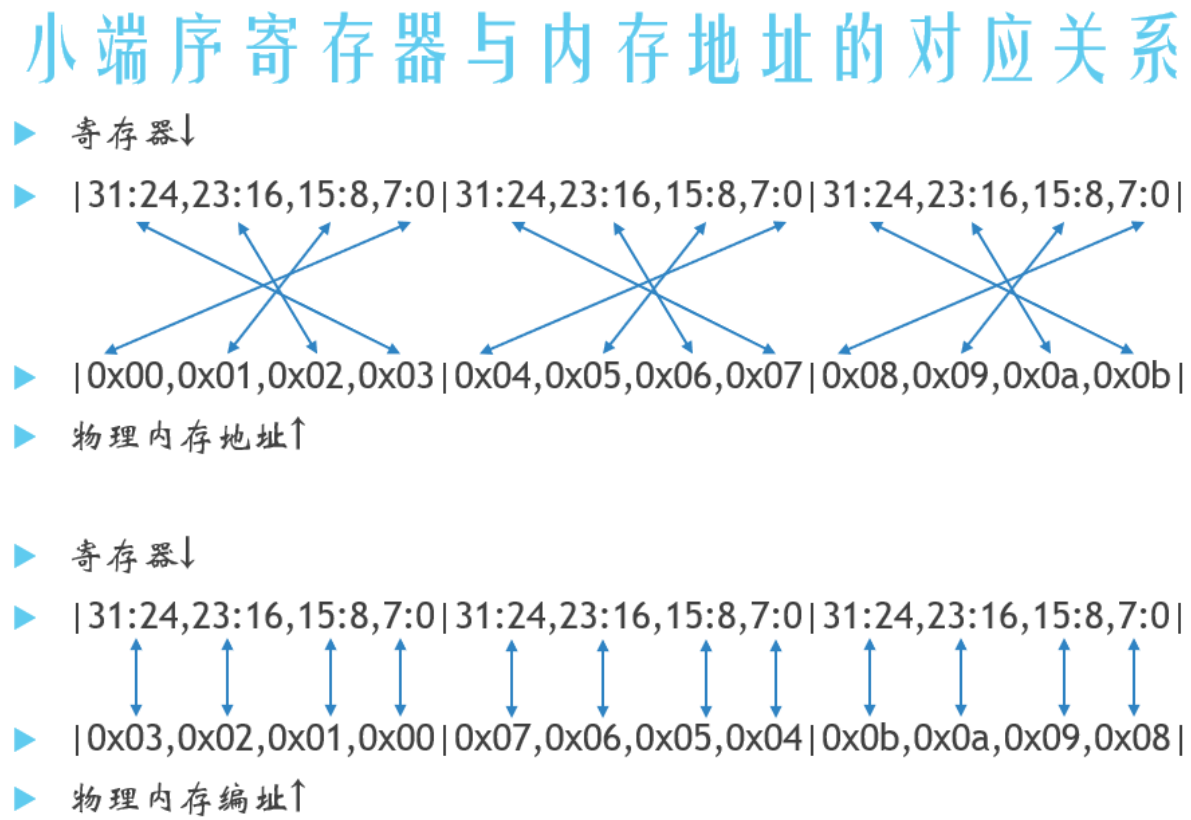
流水线的执行是建立在指令顺序执行的基础上的，而分支跳转指令则打破了这种线性执行：HikariMIPS 设计在 ID 阶段进行跳转判断，此时便能决定处新的 PC 是下一条指令，还是跳转的新指令，但此时后一条指令已经进入 IF，避免浪费一条指令，约定跳转指令后面的一条的位置为延迟槽，无论跳转指令跳转与否，延迟槽内的指令无论如何都会执行（例外是 Branch Likely 系列指令）。这样编译器可以决定在延迟槽内做一些恰当的工作，或者索性什么都不做，就放一条 NOP。

### 5. 访存指令、端序与内存编址

根据龙芯杯的要求，实现的 CPU 需要使用小端序。对于访存指令来说，端序不同只影响半字访问和按字访问，大小端序的区别如下：



而寄存器中的数据总是以 32 位为一块存储的，因此它并没有字节序的概念。这里需要将寄存器中不同的位映射到存储器中不同的字节上，普遍来说有两种方法：




可以看出这两种内存编址方式的区别就在于对齐的 4 字节块内如何分配地址。HikariMIPS 访问外部存储器时按照字节寻址，每次对齐地读入 32 位，即访问[0x00, 0x03]中任意一个地址，都返回地址为 0x00~0x03 的 4 字节数据，而存储器并不关心这 4 个字节到底是 0x03 在先还是 0x00 在先，因此这使得我们可以自由选取两种编址方式。

第一种编址方式保持物理内存地址是连续的，但是在读写数据时需要如同拧麻花一般对数据做一个映射，而这种方式在非对齐访问时（执行 lwl、lwr、swl、swr 指令）也会增加复杂性，因此 HikariMIPS 选取了第二种编址方式。保证寄存器与内存地址之间尽可能简单的映射关系，而打乱内存地址的连续性。这种编址方式也符合指令集的规范，在实现上述四条非对齐访问命令时可以直接按照指令集的解释图进行实现。

### 6. 流水线暂停

由于一些指令并不能在一个时钟周期内完成，因此需要在不同阶段申请流水线暂停。目前允许来自 ID 和 EX 阶段的暂停申请，但是未来也可以轻松地扩展到所有阶段都可以申请。

以 EX 为例，EX 阶段申请流水线暂停，则 IF、ID、EX 阶段暂停执行，MEN 和 WB 继续执行。IF 阶段的 PC 停止计数，IF/ID 锁存器传递的内容保持不变，因此 ID 阶段的状态也保持不变。ID/EX 传递的内容保持不变，EX 阶段执行多周期操作，EX/MEM 阶段则传递出 NOP 指令，使得 MEM 和 WB 执行完当前指令后什么都不做，等到 EX 完成。

流水线暂停的一些原则包括 PC 暂停计数、交界锁存器传递出 NOP 指令保证后续阶段不会产生多余的副作用，同时申请暂停的模块可与其后的锁存器组成一个状态机，辅助申请暂停的模块完成多周期指令。

### 7. 交叉编译

交叉编译是贯穿 HikariMIPS 调试过程中最重要的工具。由于调试过程中需要产生机器码供 CPU 读取并执行，最为传统的方式就是按照指令集编写出每个指令的十六进制数，然后再手动写入文件供仿真时使用。这种方法过于原始且在程序庞大的时候非常麻烦，因此一个普遍的解决方法便是使用交叉编译，在非 MIPS 的 CPU 上编译出 MIPS 程序，类似于中国人用外国话写书的感觉。这样一来就可以使用汇编来编写测试程序（在加载并启动一个可用的内核之前，所有的 C 标准库都不可用，因此只能使用汇编编写程序），然后使用交叉编译工具链直接产生 MIPS 的可执行文件。

## 三、实验过程

### 1. 搭建交叉编译环境

搭建交叉编译环境我使用了 VMware Workstation 15 版本，系统选取 Ubuntu 20.04LTS。安装好龙芯编译器及必要支持库后，配置 VMware 的文件夹共享，使得其能够直接访问到宿主机上的文件进行编译。对龙芯杯提供的测试程序执行 make 命令，确认其中没有报错即是交叉编译环境正确安装。

由于交叉编译环境的配置并非硬件课程设计之重点，这里不多赘述。

### 2. 搭建流水线结构

本节将按照前述原理构建流水线结构。由于源代码十分繁复，其中关键代码都有注释，且变量的定义及使用在同文件中跨度很大，不便于在下面全部展示，因此下面仅截取部分以供说明或示例，具体请移步源代码（链接在文末）。这里将按文件做出简要说明：

### 预定义头文件 defines.v

这个文件内全部是宏定义。在本文件中定义了 HikariMIPS 在全局范围内需要的各种信号量以提供更好的代码可读性。例如将常量 1 定义为 RstEnable、ReadEnable、WriteEnbale 等，虽然都是相同的值，但是在不同用处下可以体现出更好的代码的意图。这个文件还定义了译码阶段所需要的各种指令的操作码、功能码等，分别以 OP_、FUNC_、RT_开头；同时还定义了 ALU 的操作和计算类型，目的是分离 ALU 的计算部分与 EX 阶段产生结果的部分，使得代码更具有复用性。这使得一些指令可以复用相同的 ALU 操作，而在选择结果时可根据指令的意图写入通用寄存器或 HILO 寄存器。

这个文件最后关闭了隐式声明，这将影响所有引入该文件的代码文件。目的是在信号线众多的情况下，如果出现意料之外的拼写错误，综合器默认创建一个新的信号线来对应该拼写错误的名字，导致模块之间的连线出现问题。而这种问题还不好排查，因此直接关闭隐式声明，当综合器找不到同名的已声明信号线时，直接报出错误，有助于更早的发现并改正问题。

### 程序计数器 pc_reg.v

该文件定义了程序计数器模块，其功能有三：其一为产生 ROM 的片选使能；其二为 ROM 产生地址信号；其三为指向下一条要执行的指令。程序计数器没有收到跳转信号和复位信号时，若没有流水线暂停，则将在时钟有效沿自增 4 字节；若收到跳转信号，则将写入新的目标地址并按此寻址获取将要执行的指令。

核心代码如下：

```
always @ (posedge clk) begin
   if (ce == `ChipDisable) begin
      pc <= `ZeroWord;
   end else if (stall[0] == `NoStop) begin
      if(is_branch_i) begin
         pc <= branch_target_address_i;
      end else begin
         pc <= pc + 4'h4;
      end
   end else begin
   end
end
```

### 流水线各阶段

除了 IF 和 WB 阶段没有明确对应的模块，ID、EX 和 MEM 都有对应的模块文件，文件名分别为 id.v、ex.v、mem.v。这三个部分将在后面详细分门别类地介绍。

其中 ID 阶段除了译出控制信号之外还需要给出确定的操作数，各种数据相关的问题也将在这里得到解决。部分代码如下：

```
always @ (*) begin
    stallreq_for_reg1_loadrelated <= `NoStop;
    if(rst == `RstEnable) begin
        reg1_data_o <= `ZeroWord;
    end else if(pre_inst_is_load && ex_waddr_i == raddr1_o && re1_o == `ReadEnable ) begin
        stallreq_for_reg1_loadrelated <= `Stop;
    end else if(re1_o == `ReadEnable && ex_we_i == `WriteEnable && ex_waddr_i == raddr1_o) begin
        reg1_data_o <= ex_wdata_i;
    end else if(re1_o == `ReadEnable && mem_we_i == `WriteEnable && mem_waddr_i == raddr1_o) begin
        reg1_data_o <= mem_wdata_i;
    end else if(re1_o == `ReadEnable) begin
        reg1_data_o <= rdata1_i;
    end else if(re1_o == `ReadDisable) begin
        reg1_data_o <= imm;
    end else begin
        reg1_data_o <= `ZeroWord;
    end
end
```

### 阶段分隔锁存器

上面原理中提到过的每个阶段之间都需要有一个锁存器来保证数据保持一个完成的时钟周期。在代码文件中形如 a_b.v 的文件即表示其为 A 阶段与 B 阶段交界处的锁存器。例如 IF/ID 锁存器的文件名为 if_id.v。各个文件内代码的行为与前述原理一节相对应，关键代码处都有注释。

同时这些锁存器还负责响应流水线暂停信号。行为与原理一节所述一致。

### 寄存器堆文件 regfile.v

这个文件定义了 CPU 的寄存器堆模块。该模块内含 32 个 32 位通用寄存器，编号从 0~31，使用五个二进制位寻址。使用写优先模式，可以同时读两个寄存器并写入一个寄存器。读操作是一个组合逻辑电路，会根据读取的地址和写入的数据实时响应；写入操作则是一个时序逻辑电路，保证

始终有效沿才将数据写入寄存器。

写入部分代码：

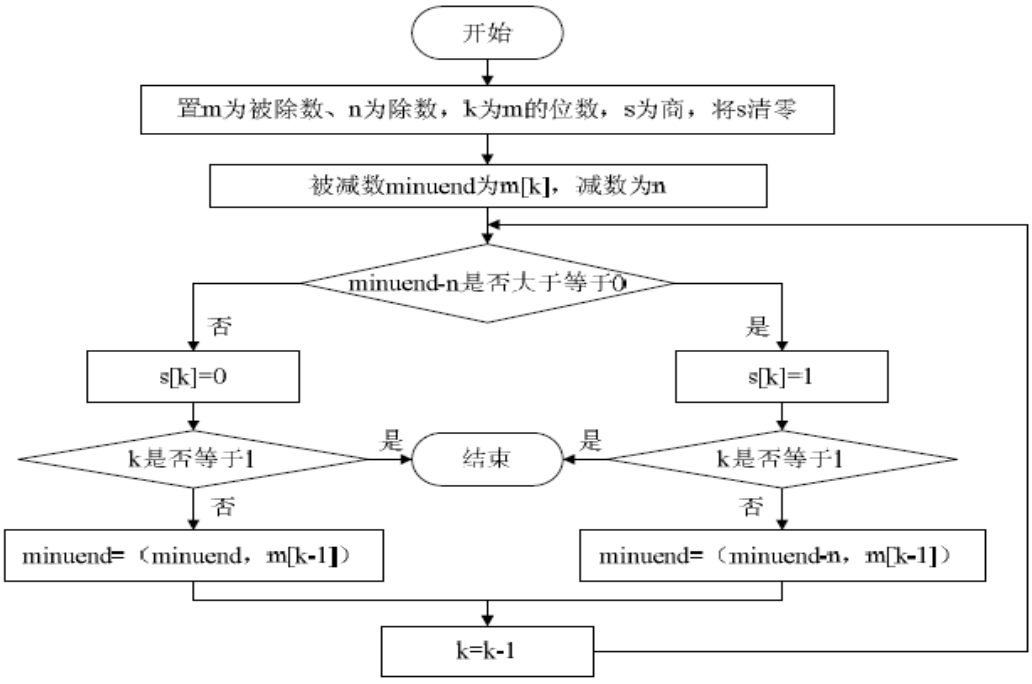
```
always @ (*) begin
    if(rst == `RstEnable) begin
        rdata1 <= `ZeroWord;
    end else if(raddr1 == `RegAddrBusWidth'h0) begin
        rdata1 <= `ZeroWord;
    end else if(raddr1 == waddr && we == `WriteEnable && re1 == `ReadEnable) begin
        rdata1 <= wdata;
    end else if(re1 == `ReadEnable) begin
        rdata1 <= regs[raddr1];
    end else begin
        rdata1 <= `ZeroWord;
    end
end
```

**HILO 寄存器 hilo_reg.v**

这个寄存器是 MIPS 指令集中特有的两个寄存器，本质与通用寄存器无异。他们是用来存储乘除法结果的。每次写入时需要同时写入 HI 和 LO 两个寄存器的值。

**除法器 div.v**

该文件定义了除法器模块。代码行为同原理一节所述。代码行为可用如下流程图总结：



### 流水线控制器 ctrl.v

该文件定义了流水线控制模块。该模块目前的作用是处理流水线暂停请求，未来进行实现相关扩展时还将处理异常等机制。

暂停请求优先级的仲裁是由 if-else-if 逻辑判断的顺序完成的，通常靠后的阶段拥有优先的暂停权限：

```
always @ (*) begin
    if(rst == `RstEnable) begin
        stall <= 6'b000000;
    end else if(stallreq_from_ex == `Stop) begin
        stall <= 6'b001111;
    end else if(stallreq_from_id == `Stop) begin
        stall <= 6'b000111;
    end else begin
        stall <= 6'b000000;
    end
end
```

### HikariMIPS 顶层文件 hikari_mips.v

这个文件将上述各模块组合连线，并对外暴露 SRAM 总线，从而形成一个可用的 CPU 模块。只需要接入时钟和复位信号，以及保证一拍之内取得数据的 ROM 和 RAM 相关信号即可。

### HikariMIPS 测试文件 hikari_mips_sopc.v

该文件定义了 HikariMIPS 的测试文件。该文件内含 HikariMIPS 的顶层文件、时钟源和基于 Xilinx Block Memory Generator 产生的 ROM 和 RAM。后两者使用标准 SRAM 接口，为了解决固定一排延迟，这里将 ROM 和 RAM 的时钟取反，使得 CPU 在上升沿给出地址、使能信号与数据，RAM 和 ROM 分别在下降沿给出响应。

关于 Xilinx Block Memory Generator 的设置，可参考下图：

### 3. 指令译码的扩展

MIPS 的所有指令都可以按照本节所述方法进行译码。所有指令经过译码都需要给出如下信号（赋值为复位情况下的值）：

```
aluop_o <= `ALU_OP_NOP; // ALU 操作
alusel_o <= `ALU_SEL_NOP; // ALU 运算类型
waddr_o <= `NOPRegAddr; // 写入寄存器的编号
we_o <= `WriteDisable; // 写入寄存器的使能信号
inst_valid <= `InstValid; // 指令是否有效
re1_o <= `ReadDisable; // 读寄存器堆端口 1 的使能信号
re2_o <= `ReadDisable; // 读寄存器堆端口 2 的使能信号
raddr1_o <= `NOPRegAddr; // 读寄存器堆端口 1 的寄存器编号
raddr2_o <= `NOPRegAddr; // 读寄存器堆端口 2 的寄存器编号
link_addr_o <= `ZeroWord; // Branch Link 指令的返回地址
branch_target_address_o <= `ZeroWord; // 分支跳转指令的目标地址
is_branch_o <= `False_v; // 分支跳转使能信号
next_inst_in_delayslot_o <= `False_v; // 下一条指令是否在延迟槽
next_inst_is_nullified_o <= `False_v; // 下一条指令是否无效化
```

正常译码开始前将设置写入寄存器的编号为 R 型指令的 rd 部分，同理读端口也分别写入 R 型指令的 rs、rt 部分。译码时根据指令的操作码部分判断做出判断，其中操作码为 SPECIAL、SPECIAL2 和 REGIMM 的指令还需要根据功能码和 rt 位置的值做更进一步判断。当最终能够确定出具体的指令后，需要按照指令集的要求对上述信号做出设置。根据指令实际操作执行位置的不同，译码阶段大体可以分为三类：一类是在 EX 阶段进行计算，产生信号为 ALU 开头；另一类是在 MEM 阶段进行访存获得结果，信号为 MEM 开头；最后一类是跳转指令，需要在 ID 阶段判断是否跳转。

为了便于译码，预先定义一些信号量如下：

```
wire[5:0] opcode = inst_i[31:26];
wire[4:0] rs = inst_i[25:21];
wire[4:0] rt = inst_i[20:16];
wire[4:0] rd = inst_i[15:11];
wire[4:0] sa = inst_i[10:6];
wire[5:0] func = inst_i[5:0];
wire[`RegBus] signed_imm = {{16{inst_i[15]}}, inst_i[15:0]};
wire[`RegBus] unsigned_imm = {16'h0, inst_i[15:0]};
wire[`RegBus] pc_next = pc_i + 4;
wire[`RegBus] pc_next_2 = pc_i + 8;
wire[`RegBus] addr_offset_imm = {{14{inst_i[15]}}, inst_i[15:0], 2'b00};
wire[`RegBus] b_addr_imm = {pc_next[31:28], inst_i[25:0], 2'b00};
wire pre_inst_is_load = ( ex_aluop_i == `MEM_OP_LB || … || ex_aluop_i == `MEM_OP_SC ) ? 1'b1 : 1'b0;
```

上述定义将指令中不同的字段按照指令集定义产生出不同的变量，例如在译码时判断 opcode 变

量即相当于判断指令的[31:26]位。下面给出 XORI 指令译码部分代码示例：

```
// …
case (opcode)
   // …
   `OP_XORI: begin
      waddr_o <= rt;
      we_o <= `WriteEnable;
      aluop_o <= `ALU_OP_XOR;
      alusel_o <= `ALU_SEL_LOGIC;
      re1_o <= `ReadEnable;
      re2_o <= `ReadDisable;
      imm <= unsigned_imm;
      inst_valid <= `InstValid;
   end
   // …
```

### 4. 指令执行的扩展

对于上面译码出来的信号，需要 EX 和 MEM 进行响应。这里只说 EX 阶段，MEM 与其结构类似。EX 阶段使用预先定义的 ALU_OP_XXX 来判断具体执行那种操作。这里的代码实践是将不同类型的运算分开使用 switch 判断。原则上可以通过一个大 switch 对 alu_op 进行判断并运算，但为了增加代码可读性，我在 EX 阶段中使用了多个 switch 进行判断。对于逻辑运算的部分则只在 switch 中判断逻辑运算相关的信号，然后另开一个 switch 块判断算术运算，其余同理。将同一种运算的结果写入独立的寄存器有助于减少代码复杂度。设想所有运算产生的结果都写入一个寄存器，原则上并不冲突，但是当代码出现问题，需要排查时，难以确定到底是哪几个运算修改了结果寄存器，从而为调试带来难度。

计算出具体结果之后，需要根据 ALU_SEL_XXX 选择一种运算类型进行输出。即根据 alu_sel 产生具体要写入的信号。例如 ALU_SEL_LOGIC 将逻辑运算结果写入通用寄存器。关于 HILO 寄存器的写入，由于其特殊性，EX 阶段根据 alu_op 进行判断，除 MUL 外的所有乘除法运算都是要写入 HILO 的，所以不与通用寄存器的写入混为一谈。

对于需要多周期运算的，分为两种情况。一种是多周期除法，除法器本身自成一体，只需要外部提供时钟和操作数，出发开始后等待计算完成即可，这种运算直接等待运算器件的完成运算即可。另一种情况是分阶段的运算，例如 MADD 这一系列的累加累减运算：第一阶段需要先计算出乘法结果，第二阶段在和原来的 HI、LO 相加或相减。这种需要阶段的运算可以依赖 EX/MEM 锁存器来构成一个状态机。

EX 阶段生成逻辑运算结果的部分代码示例：

```
always @ (*) begin
   if(rst == `RstEnable) begin
      logic_result <= `ZeroWord;
   end else begin
   case (aluop_i)
      `ALU_OP_OR: begin
         logic_result <= reg1_i | reg2_i;
      end
      `ALU_OP_AND: begin
         logic_result <= reg1_i & reg2_i;
      end
      // …
      default: begin
         logic_result <= `ZeroWord;
      end
   endcase
   end
end
```

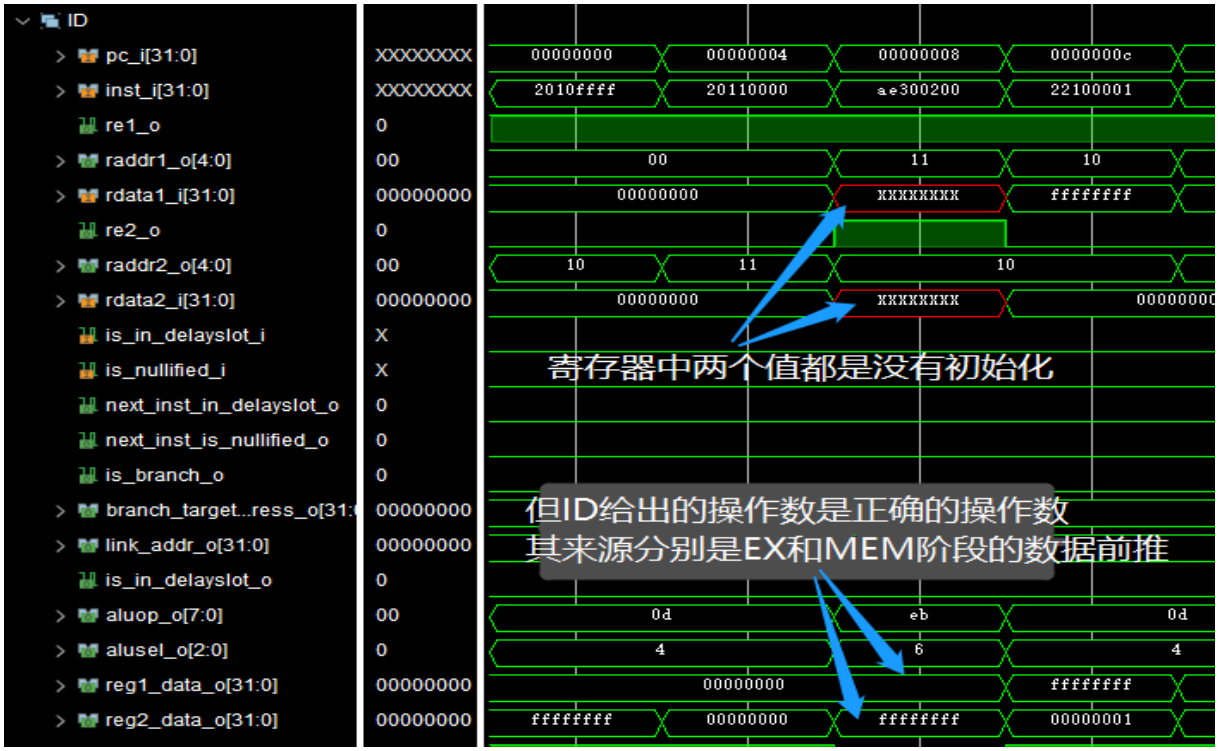
EX 阶段产生通用寄存器写入结果部分代码示例：

```
always @ (*) begin
   waddr_o <= waddr_i;
   if((aluop_i == `ALU_OP_ADD || aluop_i == `ALU_OP_SUB) && sum_overflow) begin
      we_o <= `WriteDisable;
   end else begin
      we_o <= we_i;
   end
   case (alusel_i)
      `ALU_SEL_LOGIC: begin
         wdata_o <= logic_result;
      end
      `ALU_SEL_SHIFT: begin
         wdata_o <= shift_result;
      end
      // …
      default: begin
         wdata_o <= `ZeroWord;
      end
   endcase
end
```

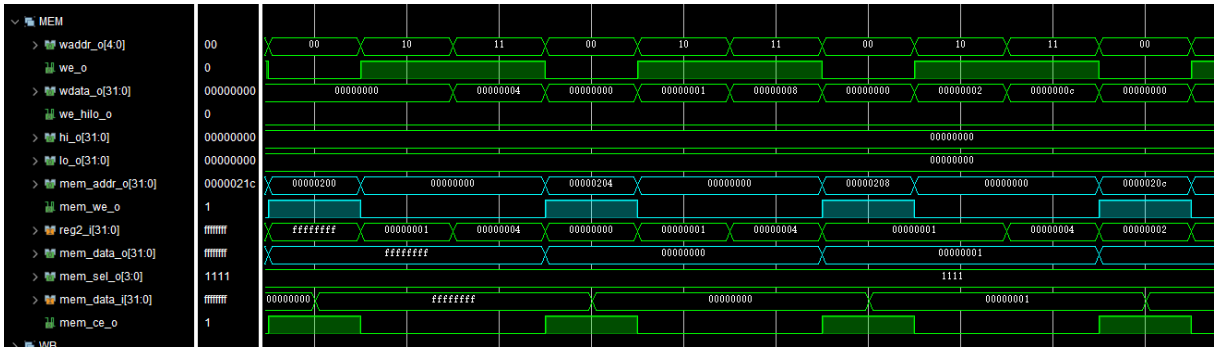
## 四、实验结果

由于指令众多，不能一一测试并体现。并且目前 HikariMIPS 尚处于未完成阶段，没法较为全面的展示其功能。在 Github Repo（地址： https://github.com/hurui200320/HikariMips）的每一次提交都有对应的测试代码，可以在 Vivado 中仿真测试。测试代码全部使用 MIPS 汇编编写，里面也配置好了 makefile，可以直接使用。

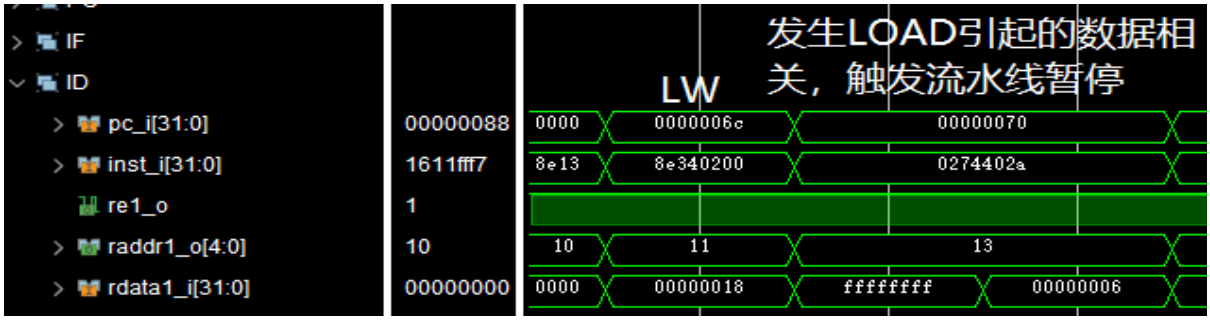
在这里仅展示一个降序排序的程序，用到的指令只有九条，分别是：addi、add、lw、sw、slt、bne、beq、j 和 nop。测试程序源代码可在 version/basic_instructions 中找到。部分仿真波形图如下示，由于波形很长，这里仅挑选关键处做解说。



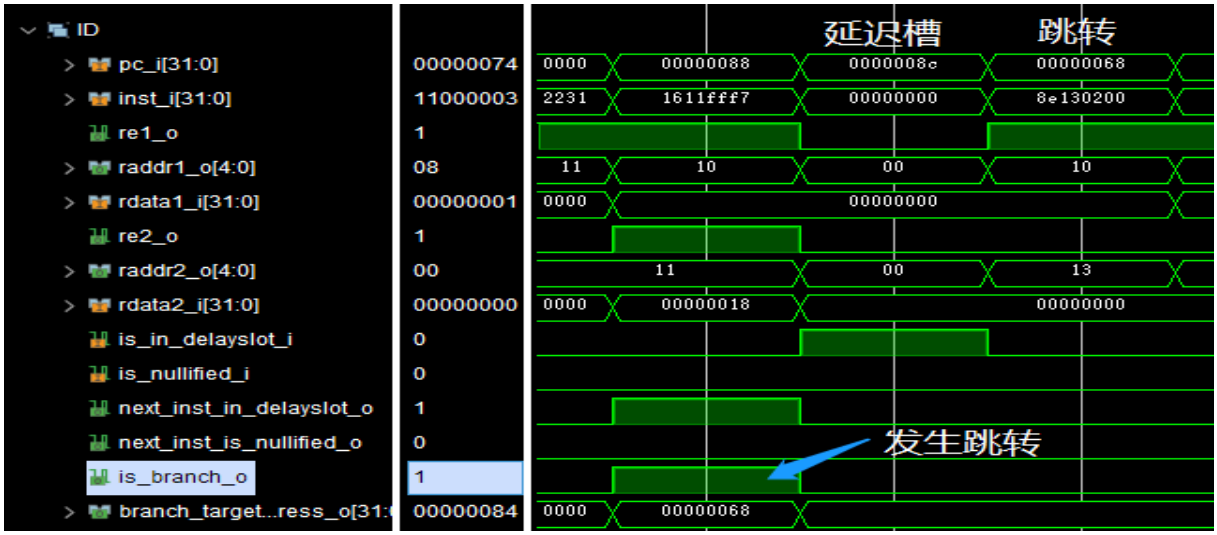

上图中展示了解决数据相关问题的波形图：第三条指令同时用到了 s0 和 s1 寄存器，而二者在寄存器堆中还尚未初始化，说明前面两条指令的运算结果还未写回，而下方 ID 阶段给出的操作数是正确的，说明 ID 模块利用数据前推解决了数据相关问题。



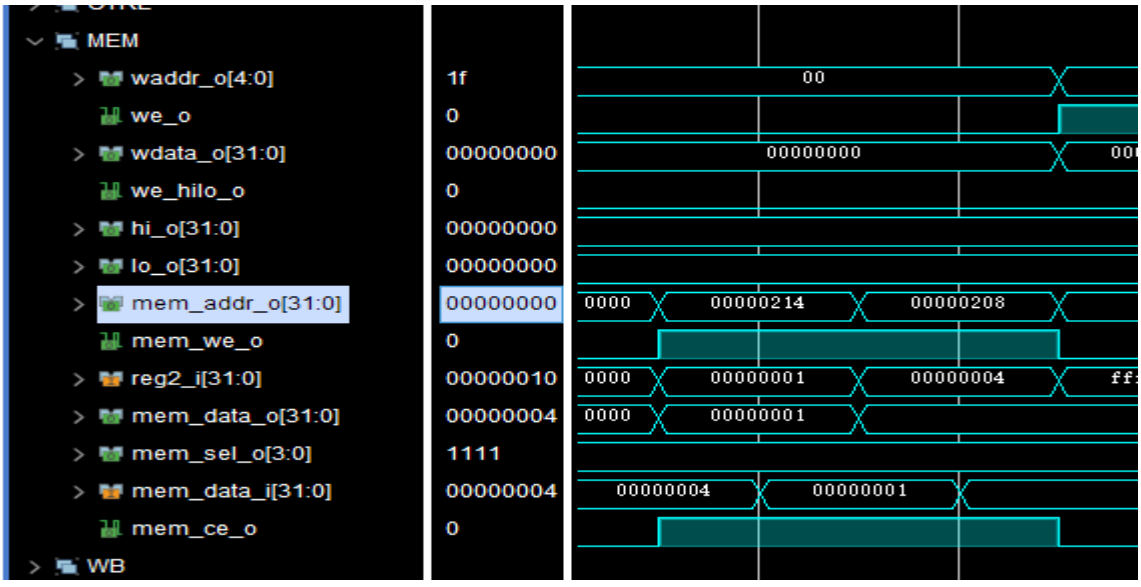
上图两条颜色特殊的线展示了 CPU 写入内存的过程。程序初始化时依次向内存写入了 8 个数字，从 0x200 开始每次写入一个字，从-1 到 6 共计 8 个数字。




上图展示了 CPU 如何解决 LOAD 指令引起的数据相关。在后面发生数据相关时，0x70 处的指令在 ID 阶段发起了流水线暂停，使得在波形图上该指令占了两拍。




上图展示了 CPU 在处理跳转时的延迟槽特性。0x88 处发生跳转，0x8c 成为延迟槽，随后便执行转移后的 0x68 处的指令。



上图展示了降序排序过程中交换两个变量的情景。一开始 0x208 处写入的是 1，而 0x214 处写入的是 4，降序排序中需要交换二者，以使较大者排列在前。因此这里需要将 4 写入 0x208，将 1 写入 0x214。

最终通过追踪波形图可以发现程序依次交换了-1 和 6、0 和 5、1 和 4、2 和 3，最终完成了降序排列。上述测试可以在 https://github.com/hurui200320/HikariMips/tree/version/basic_instructions 处找到。

## 五、个人心得体会

通过本次实验，我第一次接触到了 CPU 设计。本学期学习了计算机组成原理课程，在其中学到了关于 CPU 与计算机系统的相关知识，通过本次实验我将理论知识转化为实际代码。通过 MIPS32R1 指令集体会到了前人们设计指令集与架构的精妙之处，也使我对于 Vivado 工具以及硬件描述语言 Verilog 有了更深刻的认识与更熟练的掌握。

就 HikariMIPS 本身来说，它还并不完美，其中还有许多可以改进之处。例如若要增加主频，可以将流水线更加细分，将其中最耗时的单周期运算拆分成多周期运算，例如乘法。同时还可以给除法器单独一个倍频，因为它进行的操作比较简单，因此可以给出高于 CPU 主频的时钟频率加快运算。而缺乏浮点运算支持也是未来一个可预见的问题，好在 MIPS 指令集通过协处理器的方式可以新增拓展，而 Xilinx 也有对应的浮点数操作 IP 核，只需要包装接口即可。

HikariMIPS 还将继续完成。在本次实验中完成了基本指令的部分，而在接下来的竞赛中还需要实现 CP0、异常、内存管理、缓存等特性，这些将在后续的实验报告中完成。最终希望它可以运行起 Linux 系统，并在竞赛中取得不错的成绩。

# 基于 MIPS32R1 标准的 CPU 设计：CP0 相关部分

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将完成实现相关的 CP0 部分，在尽可能保证灵活性的同时，选择 CP0 的具体实现方式。完成 CP0 部分之后 HikariMIPS 将具有绝大部分 CPU 的功能（LL 和 SC 指令还尚未实现）。

## 二、实验原理

### 1. 协处理器 0 的内部寄存器

根据 MIPS32R1 标准，筛选出如下将要实现的寄存器：

| 编号 | SEL | 名称 | 功能 |
|---|---|---|---|
| 8 | 0 | BadVAddr | Reports the address for the most recent address-related exception |
| 9 | 0 | Count | Processor cycle count |
| 11 | 0 | Compare | Timer interrupt control |
| 12 | 0 | Status | Processor status and control |
| 13 | 0 | Cause | Cause of last general exception |
| 14 | 0 | EPC | Program counter at last exception |
| 16 | 0 | Config | Configuration register |
| 16 | 1 | Config1 | Configuration register 1 |
| 30 | 0 | ErrorEPC | Program counter at last error |

### 2. MFC0、MTC0 指令

MFC0 是 CPU 从 CP0 寄存器中读取数据的唯一指令，类似地，MTC0 是 CPU 向 CP0 寄存器写入指令的唯一寄存器。CP0 内的每一个寄存器都由编号和 SEL 共同决定，我将寄存器编号与 SEL 字

段合并成 8 位的地址，则每一个寄存器由唯一的 8 位地址确定。对于尚未实现的 CP0 寄存器，默认将返回全 0 且忽略写操作。HikariMIPS 的实现决定将 CP0 按照 HILO 寄存器的模式设计。在 EX 阶段访存并拿取 CP0 中的数据，在 WB 阶段写入 CP0。由于是流水线结构，所以对于 CP0 的存取也会出现数据相关的情况。一开始说 HikariMIPS 的所有寄存器都采用写优先模式，因此 WB 阶段不存在数据相关，因为 WB 阶段要写入的数据优先作为读取结果。这样一来只有 MEM 阶段需要处理数据相关，与 HILO 一致。

### 3. CP0 模块设计

CP0 模块同寄存器堆模块类似，可以同时读一个寄存器并写一个寄存器。在 EX 阶段直接输出要读的寄存器地址，便可以将读取的数据回传给 EX 阶段。写入则是在 WB 阶段，输入要写入的地址与数据，将保证在 WB 阶段结束前完成对目标寄存器的写入。

这里与通用寄存器的不同之处在于一些寄存器只有部分可写，或全不可写，这使得在处理写入和写优先逻辑时要额外处理。写入时需要保证只将允许写入的部分写入，而读优先应当保证只有允许写入的部分作为最新数据返回，而其他不允许写入的部分依然来自寄存器。

除了和通用寄存器一样可以读写外，CP0 还负责许多与 CPU 本身的工作方式有关的部分，由于目前 HikariMIPS 还比较简陋，Cache 或 MMU 等还皆未实现，因此目前而言 CP0 的唯一功能就是响应异常信号。HikariMIPS 的设计是在 MEM 阶段响应异常，因此在 MEM 阶段产生异常的 ExcCode，传入 CP0 后将根据这个码来判断具体是哪种异常，以及更新哪些寄存器。异常的设计不再本次实验之列。

总的来说，HikariMIPS 的 CP0 将具有如下端口：读端口、写端口、异常处理相关的端口。读写部分分别需要使能、地址和数据三种信号。异常处理相关的端口将暂时保留，在之后实现异常时实现。

## 三、实验过程

### 1. CP0 模块实现

同上面原理所述，实现写优先的寄存器堆，然后根据指令集定义筛选出哪些是可以写的，那些是部分可写的。并按照指令集定义对所实现的寄存器设定好复位的默认值。有些寄存器默认并非全零，因此需要按照指令集手册一一设定。

### 2. 其他模块的修改

实现 CP0 后还需要修改流水线中的其他模块来接入 CP0。首先需要修改 EX 阶段，增加读 CP0 地址输出和 CP0 读数据的输入。其次要在 EX、MEM 和 WB 三个模块中传递 CP0 的写信号。写信号在 EX 阶段产生，经由 MEM 阶段传递到 WB 模块进行写入。其中要解决 MEM 阶段的数据相关，解决办法即是将 MEM 阶段对于 CP0 的写信号前推到 EX 阶段，在读 CP0 时判断 MEM 前推来的数据是否与要读的地址相关。**但是这种相关的解决方式假定 CP0 中被写入的寄存器是全部可写（没有字段被丢弃），而实际并非总是如此。因此要通过编码的方式排除直接写入 CP0 的情况：一般都是先从 CP0 中读取值，然后在通用寄存器内使用 OR 置 1 或 And 置 0，再写回 CP0，来保证 MEM 数据前推的前提是正确的。**

最后还需要修改 ID 和 EX 阶段使 CPU 能够正确译码并执行 MFC0 和 MTC0 两条指令。两条指令均在 ID 阶段译码，产生通用寄存器的信号：MFC0 产生通用寄存器写信号，MTC0 产生通用寄存器读信号。EX 阶段根据 ALU OP 信号判断，MFC0 产生 CP0 读信号，作为结果写入通用寄存器，MTC0 则产生 CP0 的写信号，根据指令的内容拼接出 8 位的 CP0 内地址，作为 CP0 写地址信号。

## 四、实验结果

两条指令测试通过，经测试符合上述前提的数据相关被完美解决。测试代码及 CPU 代码可参见 https://github.com/hurui200320/HikariMips/tree/56ad111697b64246733983f35f5a73b0541efb77 。

在测试时无需关注 CPU 的每一个波形是否表现正确，没有跳转的情况下只需要追踪 WB 阶段查看写回的信号是否正常即可。对于有跳转的程序，还需要额外查看 PC 的值是否符合预期。

## 五、个人心得体会

通过本次实验，我独立完成了 CP0 的设计。由于参考书《自己动手写 CPU》一书中使用 MIPS1 指令集，大体上与 MIPS32R1 无异，但在 CP0 等实现中差异较大，因此这一节并没有参考书上的做法，而是从头开始设计 CP0 寄存器。为了简化电路，我将 HILO 寄存器和 CP0 寄存器都改成了写优先模式，从而不必考虑 WB 阶段的数据相关，书上则没有这样做。好在从头构想并不难，由于我并不熟练 Verilog，所以预先验证想法的可行性，然后再按照推敲出来的细节进行编程，能够在调试过程中节省很多事情，也能够避免一些复杂的逻辑性错误。

# 基于 MIPS32R1 标准的 CPU 设计：异常相关部分

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将完成实现异常机制。

## 二、实验原理

### 1. 将要支持的异常

根据 MIPS32R1 标准，筛选出如下将要实现的异常：

| ExcCode（16 进制） | 助记符 | 说明 |
|---|---|---|
| 0x00 | Int | Interrupt |
| 0x04 | AdEL | Address error exception<br>(load or instruction fetch) |
| 0x05 | AdES | Address error exception (store) |
| 0x08 | Sys | Syscall exception |
| 0x09 | Bp | Breakpoint exception |
| 0x0a | RI | Reserved instruction exception |
| 0x0c | Ov | Arithmetic Overflow exception |
| 0x0d | Tr | Trap exception |
| 0x10 | ERET | ERET 调用 |

关于协处理器不可用异常和总线异常，将在后面实现 CP1 浮点运算器和 AXI 总线时另行添加。在 MIPS32R1 中，ExcCode 为 0x10 的异常是保留供具体实现使用的，在 HikariMIPS 中将 ERET 也当作一种异常，因为该指令也将影响 CP0 内部寄存器的值。

### 2. 专门用于引发异常的指令

MIPS32R1 指令集中定义了一些专门用于引发异常的指令。这些指令的作用是在特定条件下引

发一个异常，用以和操作系统或其他运行在内核态的代码交互，来完成一些用户态无法完成的操作。这些指令按照异常可以分为三类：系统调用、断点和自陷。

系统调用只有一条 Syscall 指令。该指令无条件的触发 Sys 异常，而该指令与内核交互的方法不同于 x86 体系，该指令的 6~25 位可以自定义写入数据，操作系统处理异常时将实际读取发生异常时被执行的指令，从而读取该指令 6~25 位的内容，再进行不同的操作。

断点也同样只有一条 Break 指令。该指令无条件触发 Bp 异常，与 Syscall 无异。区别在于 Break 指令在启用 EJTAG 调试功能时能够辅助调试。但是出于简化设计的考虑，HikariMIPS 并没有实现 EJTAG 特性，因此 Break 指令可以视作是 Syscall 的翻版。

自陷则是一系列带有条件的指令。这个系列的指令又可以根据操作数分为 R 型和 I 型：R 型即参与条件测试的操作数皆来源于通用寄存器；I 型则是条件测试的操作数中有立即数参与。R 型指令将根据具体指令测试 rs、rt 寄存器的值，在符合条件时触发 Tr 异常，R 型指令的 6~15 位可以自定义写入数据，供操作系统处理异常时查看，而 I 型指令则没有这个字段。

### 3. 可能会引发异常的操作

除了上面说的专门用于引发异常的指令之外，CPU 还可能在运行器件产生意料之外的异常。例如在允许中断的情况下会有外部产生的硬件中断，代码申请的软件中断，以及定时器中断等。另外 MIPS32R1 要求访存指令必须自然对齐，若没有对齐则会引发地址异常。在 EX 阶段一些运算也会产生溢出等异常。另外 ERET 指令也算作异常，但从设计上来说它并非有意引发异常，而是在 HikariMIPS 实现中约定这个指令一定会引发异常。

### 4. 异常机制的实现方式

HikariMIPS 采用精确异常机制。流水线执行过程中任一阶段出现异常都将被传递到 MEM 阶段，MEM 阶段是流水线中最后一个可能会产生异常的阶段，在此阶段，所有传入的异常信号将根据指令集定义的异常优先级来决定到底产生哪一种异常，并生成对应的 ExcCode，ExcCode 和相关信号（例如 PC、延迟槽指示等信号）随后将被传入 CP0 和 Ctrl 模块。前者将根据 ExcCode 更新内部的寄存器状态，而 Ctrl 将控制流水线的清空和转移到异常处理程序的入口。

异常发生后，Ctrl 将立刻发出清空流水线的信号，各阶段间的锁存器收到信号后将传递 NOP 在各阶段的信号，使受害命令及其后的命令无效化。同时该信号还使除法器和乘法器模块停止计算。对于 PC 模块，收到流水线清空信号后会立即载入 Ctrl 传来的新地址，开始异常处理程序的取指。

## 三、实验过程

### 1. 异常信号的传递

异常信号从各部分传递到 MEM 阶段，需要经过各阶段不断传递。考虑到可拓展性及对代码最小程度的修改，HikariMIPS 使用一个 32 位的信号量 exceptions，其每一位代表可能在某个阶段发生的异常，不同阶段的同类异常将分别占用一位表示：0 为未发生，1 为发生。目前对该字段的使用情况如下：

| 位 | 说明 |
|---|---|
| 0 | PC 模块：取指发生地址异常 |
| 1 | ID 阶段：无效指令 |
| 2 | ID 阶段：ERET 调用 |
| 3 | ID 阶段：BREAK 调用 |
| 4 | ID 阶段：SYSCALL 调用 |
| 5 | EX 阶段：溢出 |
| 6 | EX 阶段：自陷 |

在 MEM 阶段内发生的异常将通过模块内定义的网线进行判断，其值不在该字段之内。Exceptions 字段在 PC 模块产生，各模块传递上一模块的信号，并更新本模块内产生的异常，直到 MEM 阶段。

除了上述最关键的表示某个异常是否发生的信号之外，延迟槽信号以及 PC 也要一路传递到 MEM，延迟槽信号将决定 EPC 记录的是 PC 还是 PC-4。

### 2. MEM 阶段产生 ExcCode

MEM 阶段根据两个来源产生 ExcCode：一个是传入的 exceptions 信号，另一个则是 MEM 内部表示是否发生地址异常的字段。最终将产生 3 个信号： exception_occured_o 表示是否发生异常，exc_code_o 表示异常的 ExcCode，bad_addr_o 表示引发地址异常的地址。这三个信号将与其他相关信号一起送往 CP0 和 Ctrl。

由于其中涉及判断中断使能和总中断开关的逻辑，因此 HikariMIPS 使用层叠的 if…else if…else 来完成判断，异常的优先级则体现在哪个最先判断，哪个优先级就更高。由于需要判断中断使能和总中断开关，因此需要来自 CP0 的 status 寄存器，同时中断还受异常级和错误级两个状态位的影响。当任意一个异常发生时，都将产生 exception_occured_o 信号，同时层叠 if 的逻辑将产生具体的

ExcCode，对于地址相关的异常，上述逻辑还将决定 bad_addr_o 来源于 PC 还是 mem_addr_o。

### 3. 修改其他阶段以响应异常

正如原理一节所述，MEM 产生相关信号后将送到 CP0 和 Ctrl，前者需要根据送来的信号更新内部寄存器。由于需要读写 CP0 内的寄存器，为了避免多驱动，应当在一个块内进行写入操作。HikariMIPS 在写数据的块尾处理异常：对于 ERET 之外的所有异常，首先更新 EPC，若在延迟槽内则 EPC 为 PC-4，否则写入 PC。随后更新异常级和 Cause 寄存器的 ExcCode 字段，最后再根据不同的异常执行不同的操作，目前唯一需要特殊行为的就是地址相关异常，需要更新 BadVAddr 寄存器。对于 ERET 指令则简单的重置异常级和错误级即可。

Ctrl 模块则在异常发生时产生 flush 信号，并输出一个 epc 供 PC 模块载入。对于 ERET 之外的异常，epc 的值为预先约定的异常处理程序入口地址，若是 ERET 则输出 CP0 内 EPC 的值，使得流水线从异常处理程序中恢复。最后还需要修改 PC 和各阶段的锁存器，使得 PC 在收到 flush 信号后能够载入新的值，各阶段锁存器则在收到 flush 信号后输出 NOP 指令。

## 四、实验结果

代码完成后并未完整的测试全部实现的异常。这里只测试了三个：Syscall、Trap 和定时器中断。没有测试诸如地址异常、溢出异常、无效指令等异常。前述三个异常均测试通过。

测试的代码可参见 https://github.com/hurui200320/HikariMips/tree/version/exceptions 。

## 五、个人心得体会

通过本次实验，我独立完成了异常机制的设计。经过简单的测试发现没有什么太大的问题。地址异常将会在后续实现 AXI 总线，实际访问外部存储器时测试，因为涉及到协议的控制，单独测试没有意义。至于无效指令，因为目前并没有实现 CP0 之外的协处理器，也不打算跑 Linux 等操作系统，因此对于不可用的协处理器指令也不用模拟，直接按无效指令处理就可以了，但是未来如果需要运行 Linux，我认为可能还需要将协处理器异常单独拆出来，后续再测试这两个异常。

至此 HikariMIPS 在功能上可以算作是完备了，麻雀虽小可也五脏俱全，尽管并没有全部实现 MIPS32R1 指令集中要求的全部内容，但是一些简单的程序已经完全可以仿真运行了。接下来要做的事情就是为 CPU 增加 AXI 总线、高速缓存等特性，使其能够在一个物理的 FPGA 芯片上运行。

# 基于 MIPS32R1 标准的 CPU 设计：AXI 总线部分

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将为 HikariMIPS 增加 AXI 总线接口。

## 二、实验原理

### 1. AXI 总线

AXI 总线全称 Advanced eXtensible Interface，是 ARM 公司研发的 Advanced Microcontroller Bus Architecture 开源标准的一部分，是一种面向高性能、高带宽、低延迟的片内总线。本次实验将实现 AXI4.0 版本。

由于原生 AXI 标准中定义了十分丰富的特性，而初赛并没有要求那么多，而且从实现难度上考虑，我将利用龙芯给的类 SRAM 到 AXI 接口的转换桥，对 AXI 转换桥略作修改以使其支持非对齐读写。

### 2. 转换桥需要的带握手的类 SRAM 接口

原生 SRAM 接口保证当拍给出读写地址，而下一拍就能保证操作完成（成功写入或读出数据），而 AXI 总线则考虑到了读写操作可能失败的情况。对于 HikariMIPS 来说，无论是 IF 阶段还是 MEM 阶段出现访存失败，对于这开发板这样一个简单的系统，我认为这种失败是不可恢复的，因此也就没有实现相关的 AXI 控制信号的判别，也没有实现 CPU 内部的总线异常。同时 AXI 总线也不保证数据时序，请求的响应只依靠握手信号，因此需要给 SRAM 增加握手机制，在等待握手时暂停流水线。同时由于对外只有一个 AXI 接口，因此要求 IF 和 MEM 要共享，由此就需要仲裁。显而易见的，两个阶段同时发出访存请求，暂停流水线：MEM 申请暂停，IF 阶段自然被暂停，此时很显然应当优先响应 MEM 的请求，等 MEM 拿到数据解除流水线暂停后，再响应 IF 的请求。

简单的说，这种类 SRAM 的握手有两次，一种是发起请求时有一个握手，握手成功后应当取消请求信号，并等待请求被响应。请求响应成功握手，表示本次请求的数据已经成功写入，或已经成功取出，本次请求完成。关于类 SRAM 接口的详细信息，请移步大赛文档。

### 3. 状态机

由于硬件没有时序的概念，因此需要一个寄存器来储存当前协议握手进行的阶段，配合时钟及外部信号切换不同的状态，以实现同一套硬件面对不同的信号能够根据时序与阶段产生不同的响应。初步构想需要一个状态机加一个组合逻辑实现类 SRAM。状态机在时钟上升沿激活，根据类 SRAM 的信号判断握手是否成功及是否需要转换到其他状态。组合逻辑电路则根据当前状态和外部信号随时调整流水线暂停信号。具体实现将在实现一节详述。

### 4. 外部存储器的更新

HikariMIPS 为了调试方便，一开始使用了接口为 Native 的 Block Memory Generator 产生的存储器，现在需要修改接口为 AXI 重新生成并连线。

## 三、实验过程

### 1. AXI 换桥的修改

由于 HikariMIPS 在 IF 阶段始终需要读写 32 位总线的全部 4 个字节，而 MEM 阶段能够根据指令和地址自动取出 32 位总线中所需的部分，因此可以设定 AXI 总线总是传输全部四个字节，即 size 字段可以恒设置为 4，而将原本 AXI 换桥所需的 size 成 strb，这样便可以实现 swl 和 swr 所需要的 3 字节读写。修改的部分可以参考 Github 上提交的代码，此处不多赘述。

### 2. IF 阶段的类 SRAM 接口实现

IF 段需要持续对 AXI 进行读请求，原来的逻辑是片选信号 CE 持续为高，而类 SRAM 接口要求地址握手成功后就应当撤销请求使能信号 req，因此额外增加一个寄存器 req_en 来表示当前允许向外发出请求，原本的使能信号 CE 则变为当前有请求。这样一来 req 即等于 ce & req_en，之后便可以在状态机中控制 req_en 来控制对外请求发送与否。

状态机一共分为三个状态：00b 空闲状态，该状态等待 ce 为有效且清空流水线信号 flush 无效时置 req_en 为高，使得 req 为高，向外发出请求，随后转入 01b 状态；01b 状态，该状态将等待地址握手，若 addr_ok 信号为有效，表示地址握手成功，请求已被接收，此时置 req_en 为低，关闭 req，并转入 10b 状态；10b 状态，该状态等待数据握手，若 data_ok 信号为有效，则数据握手成功，转入

00b 阶段等待下一次请求。

流水线暂停信号将通过组合逻辑电路实现：若当前在 00b 状态，则仅当 ce 有效且清空流水线信号 flush 无效时发起流水线暂停请求；若当前在 01b 状态，则在地址握手期间保持流水线暂停；若当前在 10b 状态，则检查 data_ok 信号，一旦数据握手成功，立刻取消流水线暂停信号。

相关代码如下：

```
always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        req_en <= `False_v; status <= 2'b00;
    end else begin
        case (status)
            2'b00: begin // 空闲阶段
                if (ce && !flush) begin
                    req_en <= `True_v; status <= 2'b01;
                end else begin
                    req_en <= `False_v;
                end
            end
            2'b01: begin // 等待地址握手
                if (!addr_ok) begin end else begin
                    req_en <= `False_v; status <= 2'b10;
                end
            end
            2'b10: begin // 等待数据握手
                if (!data_ok) begin end else begin
                    status <= 2'b00;
                end
            end
            default: begin
                req_en <= `True_v; status <= 2'b00;
            end
        endcase
    end
end
```

（接下页）

```
always @ (*) begin
    if (rst == `RstEnable) begin
        stallreq <= `False_v;
    end else begin
        case (status)
            2'b00: begin // 空闲阶段
                if (ce && !flush) begin
                    stallreq <= `True_v;
                end else begin
                    stallreq <= `False_v;
                end
            end
            2'b01: begin // 等待地址握手
                stallreq <= `True_v;
            end
            2'b10: begin // 等待数据握手
                if (!data_ok) begin
                    stallreq <= `True_v;
                end else begin
                    stallreq <= `False_v;
                end
            end
            default: begin
                stallreq <= `False_v;
            end
        endcase
    end
end
```

同时为了方便接线，这里新建一个 IF 模块，内含 PC 模块，同时对 PC 的信号加以透传，这样在一定程度上保证了程序在形式上的一致性。同时也避免了大规模的增加接线。

测试过程中发现上述实现并不能处理程序跳转的问题，原因是 ID 模块产生跳转信号时 PC 正在被暂停，而 ID 模块没有暂停，因此跳转信号只能维持一拍，IF 阶段恢复暂停最少也要四拍，考虑到 IF 被暂停后 ID 实际上是对 NOP 指令解码，因此后续修改 CTRL 模块时 IF 模块申请暂停，也一并暂停 ID 模块。

### 3. MEM 阶段的类 SRAM 接口实现

与 IF 阶段类似，MEM 阶段状态机和组合逻辑的代码与 IF 阶段完全相同。这里只简单提一下如何与原有逻辑对接：

MEM 阶段的使能信号 CE 对于异常的处理已经非常完备了，在上面异常处理部分已经使 CE 信

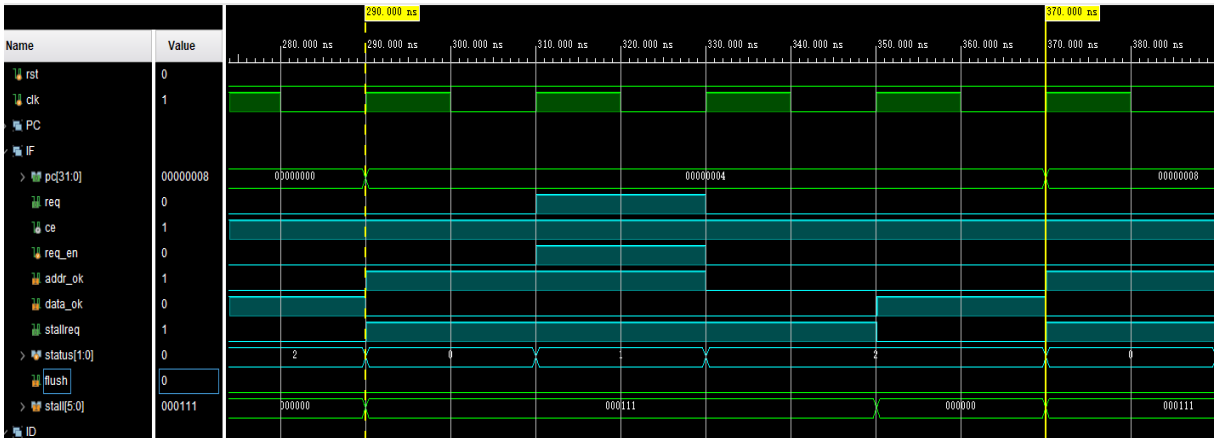
号在发生异常时失效，因此可以保证当地址出现未对齐的情况时，异常在状态机发出请求之前中断访存请求。而一旦总线请求开始，不太可能再发生异常，除非总线返回请求失败，但根据上面原理所说的，对于这样一个简单的开发板片上系统，出现访存错误，就算抛了异常也无济于事，因此这里就不再考虑总线异常的事儿了。由此可以保证新加入的逻辑与原有处理异常的逻辑能够兼容。

### 4. CTRL 模块的修改

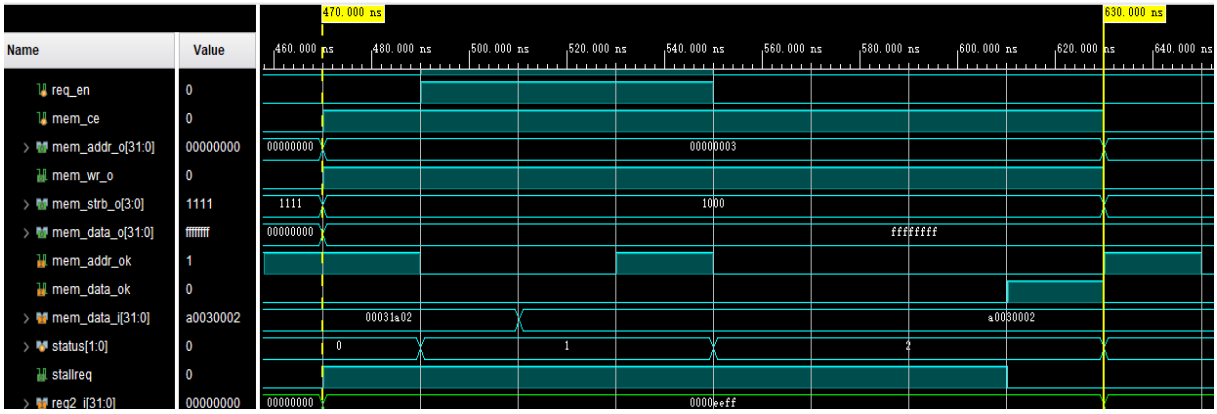
对于 CTRL 模块的修改主要是增加来自 IF 和 MEM 两个阶段的流水线暂停请求。MEM 阶段请求流水线暂停没什么好说的，需要暂停从 IF 到 MEM 的全部阶段。而 IF 阶段申请流水线暂停，应当暂停 IF 和 ID 两个阶段，因为 ID 阶段将会给出跳转信号，而暂停期间的 IF 和 PC 并不处理跳转信号，因此需要将 ID 产生的跳转信号延长到 IF 和 PC 解除暂停。最简单直接的办法就是 IF 暂停时一并连 ID 也暂停。

## 四、实验结果

IF 阶段总线握手波形图如下所示：




MEM 阶段写请求握手波形如下：



## 五、个人心得体会

通过本次实验，我独立完成了带握手的类 SRAM 接口设计与实现，并且修改了 AXI 转换桥，使得 HikariMIPS 能够接入 AXI 总线。我认为这是一个不小的进步，因为 Xilinx 的众多 IP 核都是基于 AXI 接口进行通信的，CPU 支持 AXI 接口之后便可以使用这些 IP 核，诸如浮点数运算器，以及其他复杂运算器（例如平方、开方等）。

至此 HikariMIPS 以初赛的角度看可以算是功能完备了。之后可以考虑跑一下功能测试并上板调试，通过后便可以着手加 Cache 和其他优化机制了。初赛并不要求操作系统，因此一些特性不必全部实现，或根本不需要实现，这为代码编写起到了不小的简化效果。

# 基于 MIPS32R1 标准的 CPU 设计：功能测试

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将对 HikariMIPS 进行功能测试，并加以修改使其通过。

## 二、实验原理

### 1. 功能测试

功能测试是龙芯杯官方给出的一套测试程序，一共 89 个测试点，分别可以通过上板和仿真的方式进行测试。一般来说都是先进行功能仿真确认逻辑上 CPU 的功能没有问题，再通过上板确认硬件设计（诸如时序等）没有问题，全部通过即可在一定程度上表明 CPU 的功能是健全的。

## 三、实验过程

### 1. 功能仿真

功能仿真暴露出的问题普遍在后面异常和跳转处。异常部分由于是我自己实现的，没有按照书上的方法来，因此在理解上与标准 MIPS32R1 标准有所偏差，好在经过仿真之后能够定位到问题并加以修复。跳转处的问题则主要在于延迟槽的判定上，经过仿真后亦可以快速简单的修复。

仿真过程使用 Trace 对比机制，即同样的程序先用 GS132 跑一遍，记录 CPU 的运行过程，后续在与自己实现的 CPU 对比运行过程，如果二者在某一结点上行为不一致，则判定自己实现的 CPU 跑飞了。在定位问题时，跑飞的点不一定是出问题的点，因此要根据编译产物的反汇编文件来判定当前执行的是哪一个测试，并根据出问题的程序计数器值来推断是哪一条指令出了问题，然后再进行调试、修改。步骤虽然繁琐，但不复杂，只需多些耐心与细心即可。

### 2. 上板调试

在通过全部 89 个功能测试点之后便开始综合实现，准备上板。第一次上板的结果发现运行到 4A 之后就立刻跳回到了 01，而功能仿真却没有任何问题。后来按照上板调试的方式，对 PC 等关键

reg 设置调试核，通过抓取信号发现程序运行到跳转 PC 异常这个测试点时，非对齐的地址写入 PC 引发异常，此时 PC 关闭请求使能并收集地址异常信号，关闭使能触发 PC 复位，而此时应当由 MEM 检测到异常之后通知 Ctrl 模块，由后者对 PC 写入新的地址进行异常处理。但这时 PC 已经按照复位地址开始取指，因此引发流水线暂停，导致 PC 暂停时没有响应 Ctrl 的写入请求，最终表现成一个软复位。

修复这个问题之后再次上板，发现上板卡在了 4A，后续抓取 PC 信号发现 MEM 阶段并没有收集到 PC 产生的地址异常，后来几经上板排查，大约花费了一天多的时间，最终确定问题是当时实现异常机制时并未从 IF 阶段开始，而是从 ID 阶段开始的，后面拓展到 IF 时，复制了一行代码，使得 IF 进来的异常收集信号应当为 input，而复制导致其误写为 output。修复问题后全部 89 个测试点上板通过。

## 四、实验结果

功能测试 89 个测试点全部通过。

## 五、个人心得体会

通过本次实验，我可以断言 HikariMIPS 的功能参加初赛没有问题了。虽然还没有做过系统测试，但功能测试和记忆游戏通过，使我信心倍增。接下来的任务就是为 HikariMIPS 增加 Cache，提升性能。截至今日距离比赛提交作品已经不远了，好在一开始我也没有将拿多高的奖作为己任。回想大二伊始，我是抱着弥补我在硬件方面知识的贫瘠，开始的相关学习。至此能够独立完成一个 CPU 并且通过功能测试，我很欣慰，也很满足。

接下来将根据 HikariMIPS 的情况，考虑为其增加 Cache，如有必要，还将修改 AXI 转换桥为其增加 Burst 传输特性。

# 基于 MIPS32R1 标准的 CPU 设计：重写 AXI 总线桥

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将对 HikariMIPS 的 AXI 总线桥进行重写，使其支持 1~16 字任意长度的突发读写。

## 二、实验原理

### 1. 支持 1~16 字任意长度突发读写的总线桥设计

由于后面要增加 Cache，读取和写入的时候都要按行读写，而龙芯的转换桥一次握手只能传输一个字，按行读写会非常的浪费时间。而 AXI 总线可以通过 Burst 方式一次握手并顺序传输多个字。因此本次实验将重写 AXI 总线桥，为其增加 Burst 功能。对于普通的一字读写，可以看作是长度为 1 的 Burst 传输。因此新的 AXI 转换桥将完全拥抱 Burst 传输。

为了避免多驱动带来的问题，实验设计中严格保重每个寄存器仅在一个 Always 块中被写入。为了完成要求，分别需要一个时序逻辑电路、一个组合逻辑电路和两个状态机分别完成 SRAM 握手、W 通道信号生成和 AXI 握手。

时序逻辑电路将负责 SRAM 地址握手并根据握手信号启动 AXI 读写状态机，状态机完成请求后该时序逻辑电路还将按照执行的请求和对应的端口进行数据握手，以完成本次传输。

组合逻辑将判断 wvalid 信号是否有效，如果该信号有效，则按照写状态机的相关信号生成下一个时钟有效沿要用的信号，包括：要写入的数据 wdata、位选择 strb 和最末一次传输 last。

AXI 读写分别由两个状态机组成，实现读写分离，可以在 MEM 阶段写内存的同时，完成 IF 阶段的取指。读状态机将根据启停信号开始工作。按照 AR 通道握手、R 通道握手的顺序执行。AR 通道的其他信号通过 assign 的方式产生，AR 通道握手只需要把 arvalid 拉高并等待握手即可。R 通道握手则需要一个计数器支持，事先确定要要写入的数量（即 Burst 传输长度），然后每次传输根据该计数器的值来决定写入哪个位置。写状态机类似，AW 信号也是通过 assign 的方式生成，握手只需要拉高 awvalid 并等待握手即可。W 通道握手则需要更新一个计数器，前述的组合逻辑电路将根据这个计数器和总共要写的数据来决定当次传输要写入哪一部分，最后写状态机还将额外进行 B 通道的握手，因为 CPU 并未实现总线异常，因此这里也就没有实现对 bresp 信号的检查。

## 三、实验过程

### 1. SRAM 握手时序逻辑电路

该电路分别根据读写两个操作进行握手。写部分握手逻辑如下：

```
if (!write_en) begin
    if (data_wr) begin data_data_ok <= 1'b0; end
    if (data_req && data_wr) begin
        write_addr <= data_addr;
        write_data <= data_wdata;
        write_strb <= data_strb;
        write_burst <= data_burst;
        write_en <= 1'b1;
    end else begin
        write_en <= 1'b0;
    end
end else begin
    if (write_done) begin
        data_data_ok <= 1'b1; write_en <= 1'b0;
    end else begin
        data_data_ok <= 1'b0; write_en <= 1'b1;
    end
end
```

首先检查当前写状态机使能是否有效，有效表示当前正在进行写操作，需要等待 write done 信号来完成数据握手。若没有进行写操作，则等待数据写请求握手，成功后将保存请求的信息供状态机使用。SRAM 握手信号有上述寄存器的状态组合而成：

```
assign data_addr_ok = data_wr ? !write_en : !read_en;
wire data_read_req = data_req && !data_wr;
assign inst_addr_ok = !read_en && !data_read_req;
```

数据端口的地址握手根据 data wr 是否读写来进行，根据对应状态机是否开启决定能否握手。而指令端口的地址握手除了看读状态机是否空闲之外，还需要避让 data 的读请求。对于读部分的握手，逻辑上与写类似，在此不多赘述。

### 2. AXI 读状态机

读状态机将依次进行 AW 握手、R 握手两个步骤。AW 握手前需要的信号应当已经生成完毕。MIPS32R1 指令集中提到的内存映射，以及地址能否被 Cache 的问题在此不多赘述。

```
assign araddr = {3'b000, read_addr[28:0]}; // Fixed map
assign arlen  = {4'b0000, read_burst};
assign arsize = 3'b010; // always transfer 4 bytes
assign arburst = 2'b01; // incr
assign arcache = (read_addr[31:29] == 3'b101) ? 4'b0000 : 4'b1111;
```

读状态机关键代码如下：

```
case (read_status)
   2'b00: begin
      read_status <= 2'b01;
      arvalid <= 1'b1;  // 直接开始握手
   end
   2'b01: begin
      if (arready && arvalid) begin
         // 准备 R 通道握手，重置计数器
         read_counter <= (4'b1111 - arlen[3:0]);  read_result <= 512'd0;
         arvalid <= 1'b0; read_status <= 2'b10; rready <= 1'b1;
      end else begin
         arvalid <= 1'b1; read_status <= 2'b01; rready <= 1'b0;
      end
   end
   2'b10: begin
      if (rready && rvalid) begin
         // 根据 counter 写入接收的数据
         case (read_counter)
            4'b0000: begin read_result[511:480] <= rdata; end
            4'b0001: begin read_result[479:448] <= rdata; end
            // …
            4'b1111: begin read_result[31:0] <= rdata; end
            default: begin end
         endcase
         read_counter <= read_counter + 1;
         if (rlast) begin
            read_if_or_mem[0] <= 1'b1; // read done
            rready <= 1'b0;
         end
      end
   end
   default: begin
      read_if_or_mem[0] <= 1'b0; read_status <= 2'b00; read_counter <= 4'b0000;
   end
endcase
```

代码配合注释简单易懂，在此不多赘述。其中值得说明的便是 counter 的计算。AXI 的 Burst 传输定义为至少一个，由 0000 表示，最多 16 个，由 1111 表示。因此 counter 使用 1111 减去意图进行

的 burst 数量，可以保证最后一次传输始终保存在最低位。因此 IF 和 MEM 可以根据地址判断该次请求是否需要缓存，如果不需要缓存，可以直接将数据存取到最低的 32 位，并发起 burst 为 0000 的请求。而经过缓存则可以根据自己的需要来获取任意数量的字。

写状态机与之类似，只是不在状态机内决定写入哪些，只更新 counter，另由组合逻辑电路根据 counter 决定本次传输写入哪些。因此这里不多赘述。

### 3. 写信号生成电路

写信号生成电路非常简单：

```
if (wvalid) begin
    case (write_counter)
       4'b0000: begin
          wdata <= data_wdata[511:480];
          wstrb <= data_strb[63:60];
          wlast <= 1'b0;
       end
       4'b0001: begin
          wdata <= data_wdata[479:448];
          wstrb <= data_strb[59:56];
          wlast <= 1'b0;
       end
       // …

       4'b1111: begin
          wdata <= data_wdata[31:0];
          wstrb <= data_strb[3:0];
          wlast <= 1'b1;
       end
       default: begin
          wdata <= 32'd0;
          wstrb <= 4'b0000;
          wlast <= 1'b0;
       end
    endcase
end else begin
    wdata <= 32'd0;
    wstrb <= 4'b0000;
    wlast <= 1'b0;
end
```

只需要注意传输到最后一个时，应当拉高 wlast 信号，表明即将进行最后一次传输，其他时候应

当保持该位为低。

## 四、实验结果

令 inst 和 data 的 burst 都为 0000，只取数据的低 32 位进行读写，理论上应当兼容原来龙芯的转换桥。经过测试：功能测试 89 个测试点全部通过。

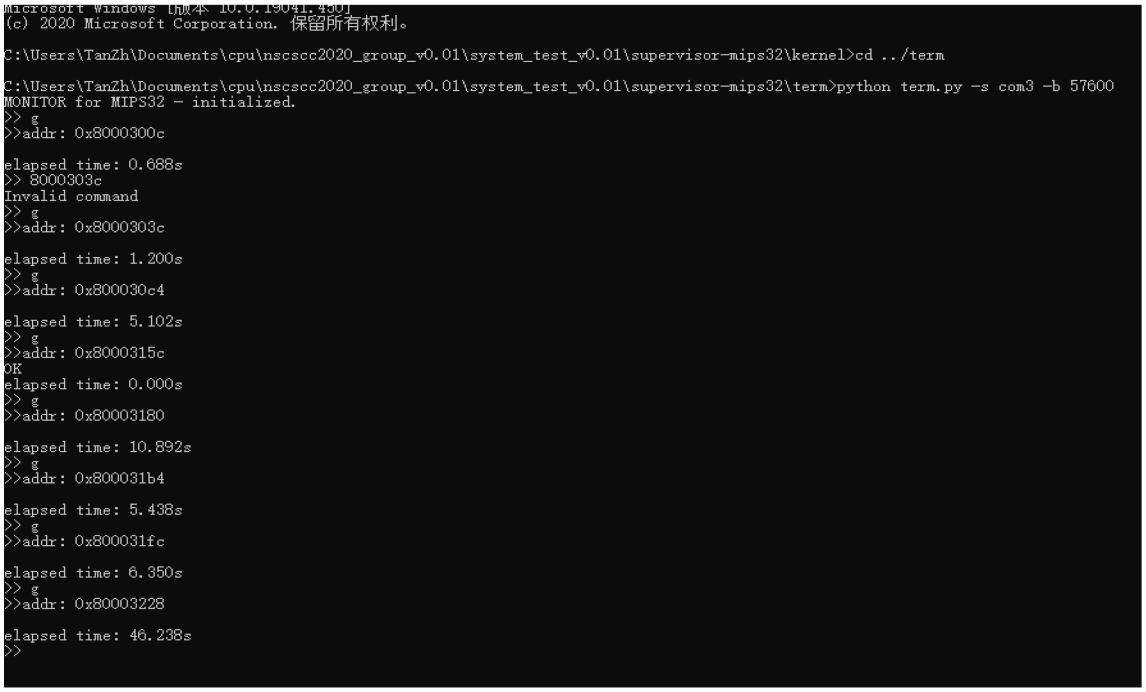
## 五、个人心得体会

通过本次实验，我独立完成了 AXI 总线桥的设计。我很欣慰。接下来将尝试给 CPU 挂上指令 Cache 和二级 Cache。二级 Cache 将使用 Xilinx 的 System Cache IP 核。

## 六、后续修改备注

后续进行系统测试时发现无法正常打印串口，开始以为是 CPU 功能实现有问题，但是后来在决赛官方文档中提到访存部分应当是完全纯粹的单字节和半字节访问。但 HikariMIPS 的实现，访存指令在 AXI 上的表现即是恒为 4 字节访问，而 CPU 内部再进行筛选。而串口 APB 设备是单字节设备，遇到 4 字节访问时会不能正确响应，因此在最开始进行系统测试时才有串口只打印出一个字母 P 而后就没有响应的现象。

进行修改后系统测试完美通过。



# 基于 MIPS32R1 标准的 CPU 设计：挂载 Cache

## 一、实验目的

本实验将结合 2020 年龙芯杯的要求，及上面实验的结果，完成基于 MIPS32R1 标准的 CPU 设计。本实验将对 HikariMIPS 挂载指令 Cache 和基于 System Cache IP 核的二级 Cache。

## 二、实验原理

### 1. Cache

在理想情况下 CPU 取一个字的指令需要至少四拍进行 SRAM 握手，而有时不同的存储器响应时间又各不相同，通常来说 CPU 访存需要等很多拍，存储器才能完成请求的操作。因此为了提高效率，可以利用 AXI 的 Busrt 传输特性，一次握手传输多个字，利用 CPU 执行程序时，在局部时间内访问的地址也仅限于一个局部范围的特性，在第一次访存某一内存地址时一并获取其附近的数据，从而在接下来的访存中直接使用 Cache 缓存的数据，而不用再重新发起握手并等待存储器响应，以此来达到节省时间、提高 CPU 运行效率的目的。

本次挂载的 Cache 分为两个，第一个是与 IF 阶段相融合的指令 Cache，利用 AXI 的 Burst 传输特性，一次读取向下对齐的 16 个字，并以直接相联的方式存储实现了一个行大小为 16 个字，总大小为 1K 的指令 Cache。第二个要挂载的是包在最外层 AXI 上的二级缓存，该缓存使用 Xilinx 的 System Cache IP 核，大小为 32K，行大小 16 个字，采用 4 路组相联的形式，采用 AXI 接口作为 CPU 与外部 AXI 总线的中间件，其控制由 arcache 和 awcache 负责。在上面重写 AXI 总线桥时已经根据 MIPS 的固定内存映射写好了判断，在遇到不需要缓存的地址时将禁止该 IP 核缓存，该 IP 核将原封不动的转发访存指令到外部 AXI 桥上，而其余情况则采用写回式的 Cache 来提高 CPU 的运行效率。

## 三、实验过程

### 1. 挂载指令 Cache

由于指令 Cache 并不是我写的，因此这里不多赘述了。简单地说就是在 IF 阶段将当前 PC 的低 6 为直接抹零，然后令 AXI 总线桥顺序向后取 16 个字节，一次性传输过来后，根据 PC 值生成 Tag，

存入数据存储器并置对应 Valid 位为 1。随后再次访存时先检查有无 tag 被命中，命中则直接寻找对应的数据进行输出，没有命中则需要申请流水线暂停，进行常规的请求来拿取数据。

### 2. 挂载二级 Cache

二级 Cache 需要首先初始化一个 System Cache IP 核，目前使用的版本是 5.0。这里可以使用面向 MicroBlaze 软核优化的 AXI 接口，而不必使用复杂的通用 AXI 接口。设定好 Cache 的工作方式后需要打开全部三个开关，分别是允许处理独占访存请求、允许不安全的访存请求，并启用 AXI 错误处理。关于 Cache 使用的资源，这里手动指定 Tag Ram 使用 LUTRAM，Data Ram 使用 BRAM，LRU Ram 使用 URAM。这里的考虑是这样的：对于 Tag Ram，每次请求时都需要快速查找有无匹配的 Tag，因此使用最快但数量最少的 LUT 资源；而 Data 则需要大量的容量存储被缓存的内容，因此考虑使用 BRAM，而平时读写命中的 Cache 时需要根据 LRU 记录每个块最近的使用数量，以便替换最近使用最少的块，该存储器数量与块数量对应，因此使用比 BRAM 稍快，且资源数量比 LUT 多的 URAM。

完成后只需要将原本直接从 AXI 桥引出到 output 的线路改道至这个 Cache，并将 Cache 的 Master 端引出作为 CPU 顶层文件的 output 即可。这里唯独要注意只输出必要的信号线，其他信号线直接按照数据宽度赋零即可。

## 四、实验结果

最终 HikariMIPS 能最高稳定在 75MHz 运行，无 WNS 警报。裸核搭配不支持 Burst 的转换桥可以上到 120MHz 稳定运行，WNS 无警告；150MHz 稳定运行但 WNS 有警告。更换支持 Burst 传输的桥之后推测桥的电路还是略显庞大，尽管 WNS 没有警告，但是实际测试最高只能上到 75MHz。

HikariMIPS 最终通过了功能测试和记忆游戏的上板测试，但未能通过系统测试。性能测试得分 14.312 分。

## 五、个人心得体会

通过本次实验以及前面的一系列实现，HikariMIPS 已经达到了作为初赛作品提交的标准。而且性能分有 14 分，较去年的不到 4 分相比，我认为进步很大。我很欣慰，不过按照这个分数，决赛可能是悬了，不过我觉得无妨。有道是「道德三皇五帝，功名夏后商周。英雄五霸闹春秋，顷刻兴亡过手。青史几行名姓？北邙无数荒丘。前人田地后人收，说甚龙争虎斗！」功名总会有褪色的那一

天，而知识却从不会褪色。尽管比赛是到今天截止了，但今后我在硬件方面的探索不会就此结束，我盘算着在今后的空闲时间里，利用自己购买的 Zynq7000 系列的 FPGA 开发板，尝试复现 Intel 的 6502CPU，在我看来兴趣使然是最重要的。

至此 HikariMIPS 的设计便到此结束了，尽管没有达到最开始说的那种程度，但过程才是奖励，况且对比去年的分数提高甚多，我很知足。

# 后记

## 一、成果总结

本实验结合 2020 年龙芯杯的要求，实现了一个基于 MIPS32R1 标准的 CPU，在初赛阶段能够运行大赛要求的功能测试、记忆游戏和性能测试，并且性能分达到了 14 分。初赛提交后经过修补，发现了 AXI 转换桥实现上的问题，修复后可以正确运行大赛要求的系统测试，能够运行监控程序，决赛性能分达到了 9，最终荣获三等奖。我认为相比以往，这是不小的进步，就三等奖本身而言，也是出乎我的意料的。毕竟我觉得头一次参加，之前虽然学过相关的原理，可终究还是没有实现过。常言道「知道与做到之间是有着巨大且难以逾越的鸿沟的」，况且在软件开发上，算法理论和最终实现都可能相差甚远，硬件设计大抵也应该是如此吧。因此我就抱着尽己所能而无所期待的心态去参加的比赛，期待太高容易摔着自己，伤害太低又容易没有动力，而最终能够看开这件事：即天外有天人外有人，我凭什么参加比赛就一定会得奖，技不如人我服气，但不得奖也要去试一试，也是对自己一个交代，没有浑水摸鱼。

荣获三等奖高兴是高兴，但三等奖注定不是特等奖，这说明我们的成就虽然得到了认可，但其中还有许多不足，更需要我反思。

## 二、反思不足

### 1. 工欲善其事，必先利其器

虽然在本学期初选修了接口与通讯这门课程，企图通过学习与实践弥补自己在硬件方面的不足。但是说来说去一学期也并未能重新拾起硬件，尤其是 FPGA 编程方面，虽然计算机组成原理的实验通过 Logisim 顺利的完成了，可毕竟 Logisim 并不是工业化的解决办法，如果英特尔 AMD 英伟达之类的公司全都用 Logisim 去画显卡画 CPU，那恐怕人类的发展还是要倒退个几十年的。最终要实用且务实，还是要落到硬件描述语言上。

在软件编程上我就认为编程语言没有好坏之分，而且大多数语言之间能够共同。我想对于硬件描述语言，这个规律犹在。并且实践证明 Verilog 能够很好的胜任编写 CPU 工作，学习实用其他硬件描述语言可以说是锦上添花，让编码工作更加轻松，但是那并不是必须的。因此在编码过程中我发现以下几点不足：

首先是代码规范的问题，Verilog 相比于 Java 这些编程语言来说算是很冷门，因此社区缺乏一个良好的代码规范标准。一般情况下都是沿用 C 风格进行编写，但是由于我 Java 写的比较多，不自觉地就在代码格式上所有混乱，但最重要的还是变量命名上，平时写 Java 可能体会不出来，但是编写硬件时一个模块内部有大量的中间变量是一个很正常的事情，因此将变量命名得言简意赅是一件非常重要的事情，后期维护和修补代码时常被自己写的变量名所迷惑，因此这一点还是有待改进的。

其次是 Vivado 的使用，刚开始编写 CPU 的时候就只会仿真、综合和实现。对于其中的结果，尤其是比较关键的实现后的报告，一开始完全一窍不通，话是这么说，其实到后来也没有完全搞懂，但是多少能够通过报告和结果判断出问题所在，并能够对症下药加以改进。这一点没有什么可多说的，工具不熟唯有多用。

最后是关于代码多人协作的问题，在软件编程上 Git 经常用于版本控制和多人协作。但是在代码的编写过程中这个工具并没有得到充分的利用，版本控制确实是很方便，能够便于我同时展开多种不同模块的编写与改动。但是多人协作方面没有得到充分利用，尤其在后期需要分享代码时，大家还是处于给代码打压缩包然后发在群里的情况。代码托管使用的 Github，在国内访问确实是慢，另一方面关于 Verilog 工程的 gitignore 文件也没有比较成熟的配置方法，导致最开始仓库内的文件不全是代码，还有好多缓存机制的二进制文件，导致克隆代码时增大了下载的负担。这里也是需要改进的一点。

### 2. 众人拾柴火焰高

团队赛虽然只有 4 个人，但是也需要大家相互合作。反观现状，其实本次比赛可以说在 5 月第一次龙芯官方培训之前就开始准备了，但是大家真正动起来却是初赛提交前的两周。在那两周之前，大家在群里就很少交流，作为队长我甚至都不知道有人中途退出，还是后面每周开会的时候才得知的。

使用 Git 促进多人协作只是技术上的手段，技术最终还是为人服务的。最终还是要沟通与交流的，因此初期团队内严重缺乏沟通与交流，是在后续（如有可能）需要额外且重点注意的。另外关于队员的选择，能力均衡是最重要的队员应当保证尽力完成分配给自己的任务，如果有疑问或者确实做不到，至少应该及时与其他队员沟通，解决问题是第一要务，搁置争议应当是我们极力避免的。

关于团队协作这件事儿，我似乎也没什么太多的发言权，平时基本上都是一个人写程序做项目，与人协作的机会不太多。这一点后续还应当继续摸索。

### 3. 脚踏实地，仰望星空

在闭幕式上听闻下一届开始可能要更换指令集了，从 MIPS 指令集更换为龙芯自己的指令集。我觉得不错，而且还听闻龙芯在二进制翻译上取得的成就，感觉消除架构壁垒未来可期。因此这里我也说一说我个人参赛的感受。

关于 MIPS，这个指令集大约是 1981 年出现的，设计之初只是简单的 CPU，类似于现在单片机的设计，还远不如现代 CPU 复杂，功能也没那么全。后来随着操作系统的出现，CPU 越来越复杂，静态结构的 CPU 已经不能胜任当时的软件，需要诸如权限、页表等与 CPU 运行状态相关的内容在运行期间被调整，CPU 也需要根据这些值去实时调整自己的行为，于是在设计之初力图将指令集与具体实现划分开的 MIPS，最终迎来了 CP0 协处理器。我个人认为这一部分是整个 MIPS 指令集最糟糕的部分。在非特权指令部分，他们的实现并不需要考虑特别复杂的事情，即便是跳转也可以利用延迟槽简单的解决，数据相关就更不用说了。但是随着 CP0 的引入，设计人员还要考虑控制遇险的问题，在 MIPS32R1 指令集中，控制遇险是由软件保证的，而在之后的指令集中则有专门用于规避控制遇险的指令，这无异给 CPU 实现带来了更加复杂的问题。有点类似于打补丁的意思，但是最终还是越补越复杂，很不优雅。

而且在开源方面，MIPS 生态尽管不错，但开源资料十分难找。最开始官网加上谷歌，我活找了大半天才找到官网提供的指令集手册和编译工具。相比之下 Risc-V 虽然没有成体系的生态，但我只花了 5 分钟就从官网上找到了指令集和工具链。而从 Risc-V 入手，他的出现比较晚，固然年轻，但设计之初便吸取了前面各种 RSIC 指令集的不足，他的 CSR 寄存器很好的解决的 CP0 不够优雅的问题，而功能上则完全不输 CP0。优胜略汰，或许这就是时代的发展，在我看来 Risc-V 确实是比 MIPS 要友好一些。听闻比赛可能换成龙芯指令集，我很高兴，也期待龙芯指令集在设计的时候能够避开 MIPS 的不足，使得 CPU 更易实现。

大赛是结束了，而今后我对于硬件方面的爱好及探索并不会止步。或许像这种从头写 CPU 的事情，作为爱好不会再做了，但是现在 Xilinx 的 Zynq7 系列的芯片有 PL 和 PS 两部分，PS 部分是一个嵌入式的 ARM 处理器，或许软硬结合能够给我带来更多的乐趣与惊喜。人总是要保持求知若渴的好奇心嘛。

—全文完—