Uranus 设计报告

第二届全国大学生计算机系统能力培养大赛
预赛提交

北京科技大学 1 队
邢其正、王硕、姚广宇、陈搏

一、设计简介

Uranus 是我们此次比赛所设计系统的名称，同时也是系统中 CPU 核心部分的开发代号。依照大赛的要求[1]，我们设计的 CPU 及外围系统使用 Vivado 2018.1 等相关平台进行开发，利用 Verilog HDL 编写相关代码，实现了一个五级流水线、兼容 MIPS 指令集（子集）、精确异常处理并且支持 AXI 总线协议的 CPU 系统，该系统可以在大赛提供的实验平台上正常运行。

整个系统在总体设计上并没有什么特别亮眼的地方，但是在实现上有一些特色之处，例如 CPU 使用了 16 周期除法器来执行除法运算，以及在开发过程中，我们团队使用了自行开发的若干工具软件来辅助整个系统的设计。

二、设计方案

（一）总体设计思路

1. myCPU 工作原理及数据通路

整体上看，myCPU 的设计大体可以分为两个阶段。

在初期阶段，为了减小 CPU 核心部分设计的负担，尽快做出基础功能正常的 MIPS 处理器，CPU 部分对外采用类 SRAM 接口。此阶段中，CPU 会假设外部存在两个 SRAM 存储设备：一个是存储需要执行的指令的 ROM，另一个是在运行时负责存储程序产生的数据的 RAM。这两个设备各自使用一套相似的接口，并且均

能在 CPU 的一个时钟周期内完成读写操作。这样做的好处是 CPU 在执行过程中无需进行额外的等待，即可高效率的完成取指和访存等操作，并且不需要实现 Cache 等部件的设计；但是这种设计过于理想化，实际环境中要想同时实现数据的海量存储与 CPU 的高速运转，解决 CPU 与存储设备之间的速度差异是不可避免的。从另一个方面考虑，只实现类 SRAM 接口而非总线接口，也制约了 CPU 和其他多种外部设备之间的数据交互，大大降低了整个系统的可扩展性。

为了进一步提高系统的实用性，并为其正常执行功能测试、性能测试乃至顺利完成更加复杂的应用打下基础，在后期阶段中，myCPU 实现了 AXI 总线协议的相关接口。为了复用上一阶段的 CPU 核心部分的设计，此阶段的 myCPU 对外提供 AXI 接口，但是在内部依然使用了基于类 SRAM 接口的实现，并且利用 SRAM 到 AXI 接口转换桥来解决发送数据请求的问题，以及使用仲裁器来解决取指和访存延迟与冲突的问题。

在最终的 myCPU 设计中，CPU 核心部分（Uranus）负责解释并执行 MIPS 指令；仲裁器负责接管核心部分的取指和访存请求，并且向外部发出访存信号来获取数据，在数据未准备好时控制核心部分的暂停；SRAM 到 AXI 转换桥负责将仲裁器发出的请求传递到外部的 AXI 总线上去，并且将外部返回的数据和握手信号回传至仲裁器；另外，还有一个 MMU 模块，负责将转换桥发送的虚拟地址转换为实际发出的物理地址。

myCPU 的结构如下图所示：

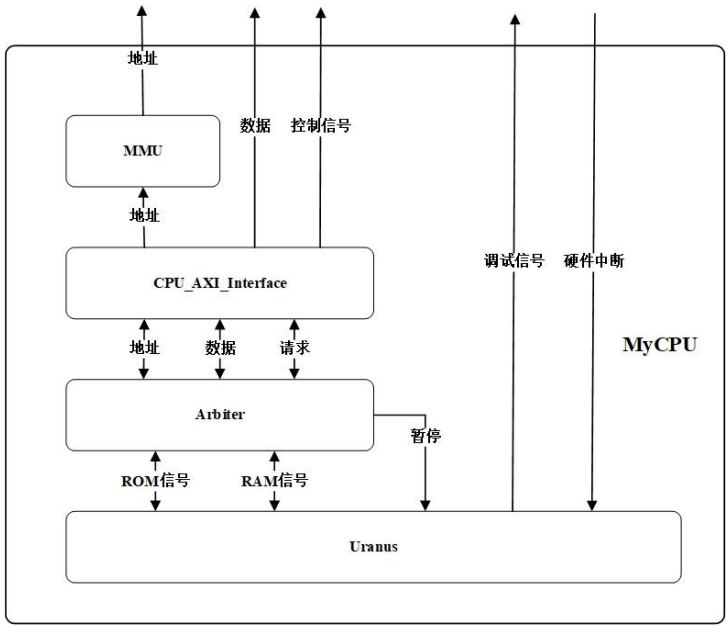

图 1：myCPU 的系统结构

2. CPU 核心部分接口及数据通路

核心部分（Uranus）旨在实现一个支持部分 MIPS I 指令以及中断和异常处理的通用处理器，其工作原理和处理方式与其他基于 MIPS 架构的处理器基本一致。包含 32 个 32 位通用寄存器、HI/LO 寄存器、PC 寄存器以及 CP0 寄存器，并且采用经典的 MIPS 五级流水线结构[2]，支持五个外部中断、定时器中断、自陷指令、各类异常处理等精确异常。

CPU 核心部分由 Verilog HDL 编码，使用 Vivado 进行开发、仿真、调试以及综合实现。在设计过程中，为了节约编码时间，降低出错概率，我们使用 Vivado 提供的 Block Design 功能对 CPU 各个模块之间的接口进行连线。但是考虑到工程代码的可移植性与可读性，使用 Block Design 并不是一个好的解决方案。这种方法还存在生成 HDL 速度慢、生成的文件不能直接作为顶层模块、生成的文件存在潜在的命名冲突等问题。

为了解决 Block Design 存在的问题，并同时利用其优点，我们使用 Python 自行开发了将 Block Design 转换为 Verilog HDL 代码的工具，大大节约了开发与调试的时间，同时提高了工程代码的可读性和可移植性。关于此工具以及其他自行开发工具的详细介绍，请参考设计报告的第二部分第十一小节。

CPU 核心部分整体的数据通路如下图所示。核心部分内部各模块之间具体的连接方式可参考提交包“attachment”目录中的“cpu_datapath.pdf”，由于其过于繁杂，此处不再展示。

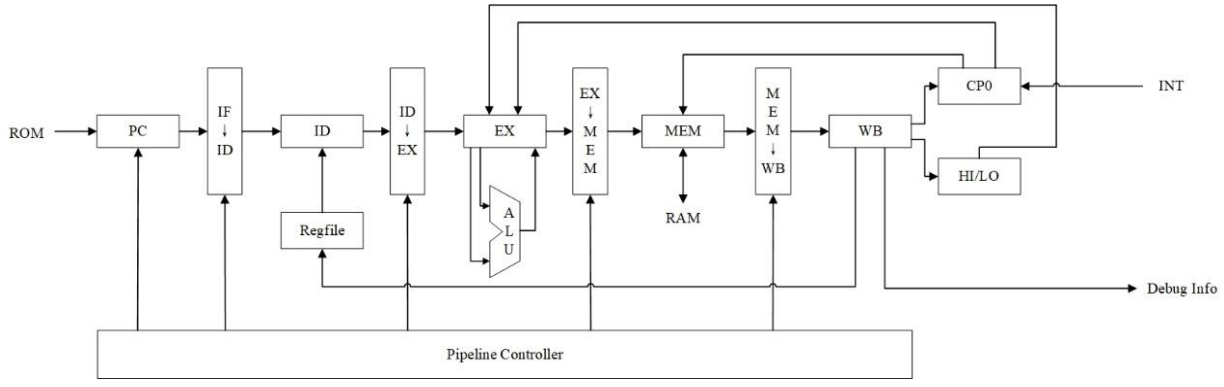

图 2：CPU 核心部分的数据通路

3. 已经实现的指令

处理器支持大赛要求的 57 条 MIPS 指令，具体如下：

- 算术运算指令：ADD、ADDI、ADDU、ADDIU、SUB、SUBU、SLT、SLTI、

SLTU、SLTIU、DIV、DIVU、MULT、MULTU；

- 逻辑运算指令：AND、ANDI、LUI、NOR、OR、ORI、XOR、XORI；
- 移位指令：SLLV、SLL、SRAV、SRA、SRLV、SRL；
- 分支跳转指令：BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ、BGEZAL、BLTZAL、J、JAL、JR、JALR；
- 数据移动指令：MFHI、MFLO、MTHI、MTLO；
- 自陷指令：BREAK、SYSCALL；
- 访存指令：LB、LBU、LH、LHU、LW、SB、SH、SW；
- 特权指令：ERET、MFC0、MTC0。

4. 内存管理

根据 MIPS 规范[3, 4, 5]，虚拟地址划分如下：

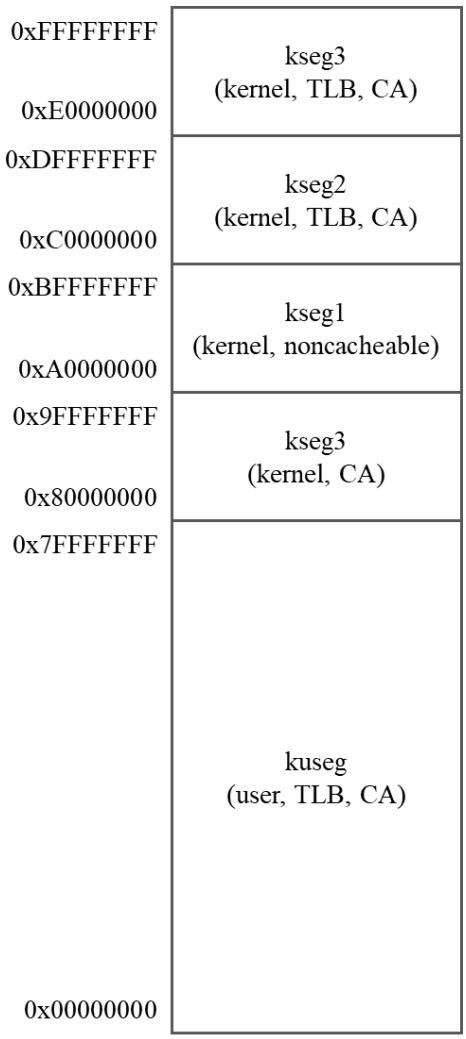

图 3：MIPS 地址空间

由于时间和精力所限，我们实现的处理器系统中并未包含 TLB，虚拟地址到

物理地址的映射均采用固定地址映射（FMT）机制。具体如下：

表 1：Uranus 的地址映射方式

虚拟地址段	映射方法	存储访问类型
kuseg	Physical = Virtual	Cached
kseg0	Physical = Virtual – 0x80000000	Uncached
kseg1	Physical = Virtual – 0xA0000000	Cached
kseg2	Physical = Virtual	Cached
kseg3	Physical = Virtual	Cached

5. 转换桥和仲裁器

由于 CPU 核心部分只实现了基于 SRAM 的数据接口，为了实现与 AXI 总线之间的数据交换，我们必须使用一个类 SRAM 转接 AXI 接口的转换桥来发起请求和进行数据传输。同样，由于访问 AXI 总线上的设备存在延迟，无法保证读写数据能够在一个时钟周期内准备就绪，所以我们还需要实现一个仲裁器，负责将 CPU 发出的数据请求转发给总线，然后等待数据就绪，再将数据回传给 CPU。在此过程中仲裁器还需要控制 CPU 的暂停。

我们实现了基于两种不同思路的仲裁器，其中一种较为简单，使用状态机来完成数据的收发和等待，但是没有实现数据的缓存机制，所以 CPU 几乎在每个取指和访存周期都需要暂停，浪费时间；另一种在此基础上实现了 L1 Cache，Cache 使用 Burst 传输（基于自行实现的 AXI 控制器而非大赛提供的转换桥）读写 AXI 总线上的存储设备，并且使用 LRU 算法进行数据替换。后一种实现在缓存命中时可以确保 CPU 持续运行而无需暂停。

时间所限，我们未能完成后一种实现的调试工作，只是暂时使用了前者作为仲裁器的最终实现，较为遗憾。

6. CP0 和异常处理

一方面为了确保处理器正常、无误地运行，另一方面为了更好的兼容 MIPS 架构，实现更为丰富的控制功能，我们的处理器支持 CP0 和精确异常处理。

处理器所支持的异常类型如下：

- INT：硬件中断、定时器中断以及软件中断；

- RI：无效指令异常；
- OV：整形溢出异常；
- BP：断点异常；
- SYS：系统调用异常；
- AdEL（取指）：取指时的非法地址异常；
- AdEL（访存）：加载时的非法地址异常；
- AdES：存储时的非法地址异常。

为了实现上述精确异常的处理功能，处理器会在流水线的各个阶段收集异常产生的信息，包含指令是否位于分支延迟槽中。当异常发生时，处理器并不会立即处理，而是将异常信息向流水线的下一级依次传递。直至异常信息抵达访存（MEM）级时，处理器会根据异常处理的优先级，向流水线控制器递交最终要处理的异常信息，接着由控制器清空流水线，并且控制取指（IF）级跳转到异常处理程序入口继续执行。

同样的，为了实现完整的异常处理和中断控制功能，处理器还实现了 CP0 相关的寄存器以及控制逻辑。其中包括 BadVAddr、Count、Compare、Status、Cause、EPC 这些寄存器，以及 PRId、Config 这两个额外的寄存器。

（二）取指（IF）模块设计

1. 接口功能描述

表 2：ID 模块的接口

名称	宽度	方向	描述
clk	1	input	时钟信号
rst	1	input	同步复位信号，低电平有效
flush	1	input	流水线清空信号
exc_pc	31:0	input	异常发生时 PC 的转移地址
stall_pc	1	input	流水线控制器发出的 PC 暂停信号
branch_flag	1	input	下一次取指是否要转移
branch_addr	31:0	input	转移的目标地址
rom_en	1	output	ROM 的读使能

rom_write_en	3:0	output	ROM 写使能，恒为 0
rom_addr	31:0	output	ROM 的读地址
rom_write_data	31:0	output	ROM 的写地址，恒为 0

2. 控制逻辑

IF 即取指阶段模块，该模块的功能较为简单，且由于其处在流水线的第一级，又需要控制指令的读取，所以采用时序逻辑实现。通常情况下，IF 模块会从 0xBFC00000 地址开始取指，每次 PC 的值加 4，即顺序取指。当遇到分支/跳转指令时，PC 的变化还会受分支转移信号的控制，这些信号从 ID 阶段发出。这种实现方法同时也体现出了延迟槽的行为。当处理器发生异常时，IF 模块接收 flush 信号，PC 的取值会被重置为异常处理程序的入口地址。需要注意的是，ERET 指令也会被当作异常进行处理，所以处理器在从异常处理中返回时，也需要 exc_pc 传递新的取指地址。

（三）译码（ID）模块设计

1. 接口功能描述

表 3：ID 模块的接口

名称	宽度	方向	描述
rst	1	input	同步复位信号，低电平有效
addr	31:0	input	译码阶段的指令对应的地址
inst	31:0	input	译码阶段的指令
load_related_1	1	input	源操作数 1 存在 load/use hazard
load_related_2	1	input	源操作数 2 存在 load/use hazard
delayslot_flag_in	1	input	当前指令是否位于延迟槽
reg_data_1	31:0	input	从 Regfile 输出的第一个读寄存器端口的输入
reg_data_2	31:0	input	从 Regfile 输出的第二个读寄存器端口的输入
reg_read_en_1	1	output	Regfile 模块的第一个读寄存器端口的读使能信号

reg_read_en_2	1	output	Regfile 模块的第二个读寄存器端口的读使能信号
reg_addr_1	4:0	output	Regfile 模块的第一个读寄存器端口的读地址信号
reg_addr_2	4:0	output	Regfile 模块的第二个读寄存器端口的读地址信号
stall_request	1	output	流水线暂停请求信号
branch_flag	1	output	是否发生转移
branch_addr	31:0	output	转移到的目标地址
next_delayslot_flag_out	1	output	下一指令是否位于延迟槽
delayslot_flag_out	1	output	当前指令是否位于延迟槽
funct	5:0	output	译码阶段的指令要进行的运算的子类型
shamt	4:0	output	译码阶段的指令要进行的运算的类型
operand_1	31:0	output	译码阶段的指令要进行的运算的源操作数 1
operand_2	31:0	output	译码阶段的指令要进行的运算的源操作数 2
mem_read_flag	1	output	访存阶段是否要进行读操作
mem_write_flag	1	output	访存阶段是否要进行写操作
mem_sign_ext_flag	1	output	访存的数据是否要进行符号扩展
mem_sel	3:0	output	访存阶段的字节选通信号
mem_write_data	31:0	output	访存阶段要写入的数据
write_reg_en	1	output	译码阶段的指令是否有要写入的目的寄存器
write_reg_addr	4:0	output	译码阶段的指令要写入的目的寄存器地址
cp0_write_flag	1	output	是否要写 CP0 寄存器

cp0_read_flag	1	output	是否要读 CP0 寄存器
cp0_addr	4:0	output	要写入或读取的 CP0 寄存器的地址
cp0_write_data	31:0	output	要写入 CP0 寄存器的数据
exception_type	7:0	output	收集到的异常信息
current_pc_addr	31:0	output	译码阶段指令要保存的返回地址

2. 控制逻辑

译码阶段产生后续部件需要用到的控制信号和数据，在这个阶段读取寄存器的值并输出保持。另外，对于分支指令，也要在这一阶段设置跳转的地址。ID 阶段是整个流水线中最为重要的部件，它控制了后续几个部件的工作行为。

以 ORI 指令和 OR 指令为例：ORI 指令的 OP 是 2'b001101，OR 指令的 OP 是 2'b000000，查询 MIPS 手册我们得知，这个操作码表示 SPECIAL。以 OP_SPECIAL 为开头的指令有很多，它们通过 5:0 位的 FUNCT 段相互区分。

这里我们做一个归一化的处理。查询 MIPS 手册我们得知，OR 和 ORI 除了操作数 2 的来源不一样（一个来自于立即数的无符号扩展，一个来自于寄存器），要写入的寄存器地址不一样（一个是 rd，一个是 rt），在 ALU 看来它们其实是一种运算。注意到前面 ID 模块添加了给 WB 模块的写入寄存器地址和相关的使能信号，那么写入寄存器地址的区别就可以通过这部分来解决。而操作数来源的不同更好解决：只要控制向 EX 模块输出的 operand_1 和 operand_2 的内容就行了。这样，在 EX 模块看来，它只需要做一个 OR 操作，至于这个 OR 操作是来自于 ORI 指令还是 OR 指令，EX 模块并不需要关注。同样能做这种归一化处理的指令还有很多，例如 addi 与 add，addiu 与 addu，andi 与 and 等等。

（四）执行（EX）模块设计

1. 接口功能描述

表 4：EX 模块的接口

名称	宽度	方向	描述
rst	1	input	同步复位信号，低电平有效
funct	5:0	input	执行阶段要进行的运算的类型
shamt	4:0	input	移位指令的位移量

operand_1	31:0	input	参与运算的源操作数 1
operand_2	31:0	input	参与运算的源操作数 2
mem_read_flag_in	1	input	访存阶段是否要进行读操作
mem_write_flag_in	1	input	访存阶段是否要进行写操作
mem_sign_ext_flag_in	1	input	访存的数据是否要进行符号扩展
mem_sel_in	3:0	input	访存阶段的字节选通信号
mem_write_data_in	31:0	input	访存阶段要写入的数据
write_reg_en_in	1	input	是否有要写入的目的寄存器
write_reg_addr_in	4:0	input	指令执行要写入的目的寄存器地址
exception_type_in	7:0	input	译码阶段收集到的异常信息
delayslot_flag_in	1	input	当前指令是否位于延迟槽
current_pc_addr_in	31:0	input	当前指令的地址
hi_in	31:0	input	HILO 模块给出的 HI 寄存器的值
lo_in	31:0	input	HILO 模块给出的 LO 寄存器的值
co0_write_flag_in	1	input	是否要写 CP0 寄存器
cp0_read_flag_in	1	input	是否要读 CP0 寄存器
cp0_addr_in	4:0	input	要写入或读取的 CP0 寄存器的地址
cp0_write_data_in	31:0	input	要写入 CP0 寄存器的数据
cp0_read_data_in	31:0	input	从 CP0 模块读取的指定寄存器的值
mult_div_done_flag	1	input	是否完成乘除法
mult_div_result	63:0	input	乘除法运算的结果
stall_request	1	output	流水线暂停请求信号
ex_load_flag	1	output	执行阶段存在 load/use hazard
mem_read_flag_out	1	output	访存阶段是否要进行读操作
mem_write_flag_out	1	output	访存阶段是否要进行写操作
mem_sign_ext_flag_out	1	output	访存的数据是否要进行符号扩展
mem_sel_out	3:0	output	访存阶段的字节选通信号
mem_write_data_out	31:0	output	访存阶段要写入的数据
result_out	31:0	output	执行阶段的指令最终要写入目的寄

			存器的值
write_reg_en_out	1	output	执行阶段的指令最终是否有要写入的目的寄存器
write_reg_addr_out	4:0	output	执行阶段的指令最终要写入的目的寄存器地址
hilo_write_en	1	output	执行阶段的指令是否要写入 HILO 寄存器
hi_out	31:0	output	执行阶段的指令要写入 HI 寄存器的值
lo_out	31:0	output	执行阶段的指令要写入 LO 寄存器的值
cp0_write_en	1	output	是否要写 CP0 寄存器
cp0_addr_out	4:0	output	要写入的 CP0 寄存器的地址
cp0_write_data_out	31:0	output	要写入的 CP0 寄存器的数据
exception_type_out	7:0	output	执行阶段收集到的异常信息
delayslot_flag_out	1	output	当前指令是否位于延迟槽
current_pc_addr_out	31:0	output	当前指令的地址

2. 控制逻辑

EX 阶段的核心部件是 ALU。它根据 ID 阶段的信号执行对应的运算，把结果输出。大部分的运算都可以使用组合逻辑瞬间完成，少部分的运算如除法等需要用到多个周期，多周期的运算需要用到流水线暂停机制。

EX 阶段的主要工作是执行运算并传出地址。在 ID 阶段我们已经给出了操作码、操作数以及写入寄存器的信号和地址。也就是说，结果放到哪里我们已经在 ID 阶段指定了，EX 要做的就是计算出结果。

（五）访存（MEM）模块设计

1. 接口功能描述

表 5：MEM 模块的接口

名称	宽度	方向	描述

rst	1	input	同步复位信号，低电平有效
mem_read_flag_in	1	input	访存读标志
mem_write_flag_in	1	input	访存写标志
mem_sign_ext_flag_in	1	input	访存符号扩展标志
mem_sel_in	3:0	input	访存字节使能
mem_write_data	31:0	input	访存写数据
result_in	31:0	input	EX 级写回到寄存器堆的数据
write_reg_en_in	4:0	input	寄存器堆写使能信号
write_reg_addr_in	31:0	input	寄存器堆写地址
hilo_write_en_in	1	input	HILO 寄存器写使能
hi_in	4:0	input	HI 寄存器输入
lo_in	31:0	input	LO 寄存器输入
cp0_write_en_in	1	input	CP0 写使能
cp0_addr_in	4:0	input	CP0 地址输入
cp0_write_data_in	31:0	input	CP0 写数据
cp0_status_in	31:0	input	CP0.Status 寄存器的值
cp0_epc_in	31:0	input	CP0.EPC 寄存器的值
exception_type_in	7:0	input	异常类型输入
delayslot_flag_in	1	input	指令位于延迟槽的标志
current_pc_addr_in	31:0	input	当前 PC 地址输入
mem_load_flag	1	output	访存阶段存在 load/use hazard
mem_read_flag_out	1	output	访存读标志
mem_write_flag_out	1	output	访存写标志
mem_sign_ext_flag_out	1	output	访存符号扩展标志
mem_sel_out	3:0	output	访存字节使能
result_out	31:0	output	MEM 级写回到寄存器堆的数据
write_reg_en_out	1	output	寄存器堆写使能
write_reg_addr_out	4:0	output	寄存器堆写地址
hilo_write_en_out	1	output	HILO 寄存器写使能

hi_out	31:0	output	HI 寄存器写数据输出
lo_out	31:0	output	LO 寄存器写数据输出
cp0_write_en_out	1	output	CP0 写使能
cp0_addr_out	4:0	output	CP0 写地址
cp0_write_data_out	31:0	output	CP0 写数据
cp0_badvaddr_write_data	31:0	output	CP0.BadVAddr 寄存器写数据
cp0_epc_out	31:0	output	CP0.EPC 寄存器数据
exception_type_out	7:0	output	异常类型输出
delayslot_flag_out	1	output	延迟槽标志输出
current_pc_addr_out	31:0	output	当前 PC 地址输出
ram_en	1	output	RAM 使能
ram_write_en	3:0	output	RAM 写使能
ram_addr	31:0	output	RAM 地址
ram_write_data	31:0	output	RAM 写数据

2. 控制逻辑

(1) RAM 使能信号的生成：当内存写标志或者内存读标志有效时，若前级未产生异常则 RAM 选通；

(2) RAM 写使能信号的生成：当内存写入标志有效时，选通 RAM 中 ram_write_sel 信号所选择的部分，否则保持写使能无效；

(3) RAM 被写入的地址信号的生成：当内存写标志或者内存读标志有效时，被写入的地址由 address 信号决定；

(4) RAM 写入选择信号的生成：address 后两位决定 ram_sel_in 的值。另外 mem_sel_in 决定对 address 最后两位的译码方式；

(5) RAM 中写入的数据信号的生成：address 决定向内存中的数据如何写入 RAM，mem_sel_in 决定对 address 最后两位的译码方式；

(6) 异常处理：为了实现精确异常，流水线的 MEM 阶段会汇总之前所有流水线阶段的异常信息，然后根据异常处理的优先级顺序，决定最重要处理的异常类型，并且将异常相关的信息直接传递给 CP0 的相关寄存器写入。CP0 会进一步控制流水线控制器实现流水线的清空和异常转移。

（六）写回（WB）模块设计

1. 接口功能描述

表 6：WB 模块的接口

名称	宽度	方向	描述
rst	1	input	同步复位信号，低电平有效
ram_read_data	31:0	input	从 RAM 中读取的数据
mem_read_flag	1	input	访存读数据标志
mem_sign_ext_flag	1	input	访存数据扩展标志
mem_sel	3:0	input	访存字节使能
result_in	31:0	input	寄存器堆写回数据
write_reg_en_in	1	input	寄存器堆写使能
write_reg_addr_in	4:0	input	寄存器堆写地址
hilo_write_en_in	1	input	HILO 寄存器写使能
hi_in	31:0	input	HI 寄存器数据输入
lo_in	31:0	input	LO 寄存器数据输入
cp0_write_en_in	1	input	CP0 写使能
cp0_addr_in	4:0	input	CP0 写地址
cp0_write_data_in	31:0	input	CP0 写数据
debug_pc_addr_in	31:0	input	调试信号，当前指令的地址
result_out	31:0	output	寄存器堆写回数据输出
write_reg_en_out	1	output	寄存器堆写使能
write_reg_addr_out	4:0	output	寄存器堆写地址
hilo_write_en_out	1	output	HILO 寄存器写使能
hi_out	31:0	output	HI 寄存器数据输出
lo_out	31:0	output	LO 寄存器数据输出
cp0_write_en_out	1	output	CP0 写使能
cp0_addr_out	4:0	output	CP0 写地址
cp0_write_data_out	31:0	output	CP0 写数据

debug_pc_addr_out	31:0	output	调试信号，当前指令的地址
debug_reg_write_en	3:0	output	调试信号，寄存器堆字节写诗能

2. 控制逻辑

(1) result_out 信号的生成：由于 SRAM 为同步读取，对于 RAM 的访问需要在写回级进行。当 mem_read_flag 有效时，address 的后两位决定 reselt_out 信号取 ram_read 中的哪些位，mem_sel 决定对 address 的译码方式；当 mem_read_flag 无效，mem_write_flag 有效时，result_out 恒为 0。没有访存请求时，result_out 维持 result_in 输入的值；

(2) HILO、CP0 相关信号的输出：当 rst 有效时，这些信号的输出等于相应的输入，rst 无效时，这些信号为 0。

（七）流水线控制器设计

1. 模块划分

经典的五级流水线结构设计中，流水线各级的功能均由组合逻辑实现，而流水线各级之间的运行和调度则通过流水线控制器来实现。

流水线控制相关的部件主要分为两类，一类是分布在流水线前后两级之间的类触发器电路，我们称之为流水线中间级；另一类是负责控制各流水线中间级，实现流水线暂停以及清空功能的流水线控制器。

2. 接口功能描述

流水线中间级由多个“PipelineDeliver”这一基础部件构成，此部件实现了类似 D 触发器的功能，除此之外还可以接收流水线暂停信号和清空信号，方便控制流水线各级的动作。其接口如下表所示：

表 7：PipelineDeliver 模块接口

名称	宽度	方向	描述
clk	1	input	时钟信号
rst	1	input	同步复位信号，低电平有效
flush	1	input	流水线清空信号
stall_current_stage	1	input	暂停流水线当前级

stall_next_stage	1	input	暂停流水线下一级
in	width - 1:0	input	触发器输入
out	width - 1:0	output	触发器输出

其中，接口宽度定义中的 width 是模块的例化参数，可以指定模块要传递的数据的宽度。

流水线控制器模块“PipelineController”使用了组合逻辑实现，由于其需要控制和调度流水线各级的运转，所以接口较多。详情如下：

表 8：PipelineController 模块接口

名称	宽度	方向	描述
rst	1	input	同步复位信号，低电平有效
request_from_id	1	input	来自 ID 级的暂停请求
request_from_ex	1	input	来自 EX 级的暂停请求
stall_all	1	input	暂停 CPU 的请求
cp0_epc	31:0	input	CP0.EPC 寄存器的值，用于 ERET 指令从异常处理中返回
exception_type	7:0	input	异常类型
stall_pc	1	output	暂停 PC
stall_if	1	output	暂停 IF 级
stall_id	1	output	暂停 ID 级
stall_ex	1	output	暂停 EX 级
stall_mem	1	output	暂停 MEM 级
stall_wb	1	output	暂停 WB 级
flush	1	output	清空流水线
exc_pc	31:0	output	异常处理时产生的新的 PC 的值，用于控制取指模块转向异常处理入口

3. 流水线暂停（stall）控制

PipelineController 模块在检测到以下情况发生时（按照优先级顺序排列），会控制流水线的暂停：

- stall_all 信号有效：暂停所有流水线中间级的运行，此情况通常发生在外部的仲裁器尝试向 AXI 总线存取数据但是数据还未准备好的时候；
- request_from_id 信号有效：暂停 PC、IF 级和 ID 级，当 ID 级检测到存在 load/use hazard 时，会发出此信号；
- request_from_ex 信号有效：暂停 PC、IF 级、ID 级和 EX 级，当 EX 级正在进行乘除法运算且运算未完成之前，会发出此信号；
- 上述信号均无效：流水线正常运行。

4. 异常处理

当处理器遇到异常时，为了实现精确异常，所有异常类型的信息会被收集并且传递到 MEM 级。MEM 级根据异常的优先级顺序汇总异常信息，并且将其发送到 CP0 寄存器以及流水线控制器。传递到 CP0 的异常信息会决定写入 CP0 相关寄存器的值（例如写入 BadVAddr），而传递到流水线控制器的异常信息会直接控制流水线的清空以及 PC 取指入口的转移。

在异常发生时，流水线控制器会令 flush 信号有效，这时流水线各中间级会清除所有存储的数据；同时，控制器将异常入口处的 PC 值传递到取指级，控制异常处理程序的执行。当异常返回时，ERET 指令同样会被视作一种异常进行处理，此时流水线控制器会将 CP0.EPC 的值传递给取指级，实现异常的返回。

（八）除法器设计

1. 除法器原理

在根据实际情况进行权衡把控之后，我们实现了使用两位试商法的 16 周期 32 位除法器。在使能信号有效时，除法器会进入执行周期，在每个周期中把被除数与除数多次进行比较，决定要补充的两位数商的具体数值。这种方法的优点是计算周期短，但是缺点是占用资源较多，且暂时不能流水化处理。

2. 状态机

除法器的状态机如图 4 所示：

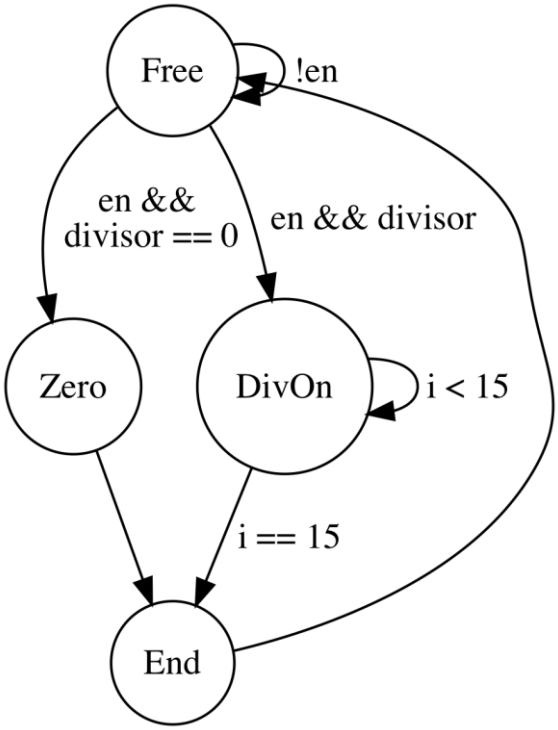

图 4：16 周期除法器的状态机

（九）AXI 总线交互模块设计

该模块的主要功能是为 CPU 提供 AXI 接口支持，方便 CPU 与各从设备连接。该模块原计划自行实现，但由于 cache 模块未实现，无法进行 burst 传输，继续使用原设计占据开销过大，故使用了大赛官方提供的类 SRAM 转 AXI 接口模块以减少开销。

大赛官方提供的类 SRAM 转 AXI 接口模块：

1. 接口功能描述

该模块具有 54 个端口，其中 23 个输入端口，31 个输出端口。这 54 个端口信号分别是：2 个 *GLOBAL* 信号（包括时钟信号和复位信号），8 个 *INST SRAM-LIKE* 信号，8 个 *DATA SRAM-LIKE* 信号，10 个 *READ ADDRESS CHANNEL* 信号，6 个 *READ DATA CHANNEL* 信号，10 个 *WRITE ADDRESS CHANNEL* 信号，6 个 *WRITE DATA CHANNEL* 信号，4 个 *WRITE RESPONSE CHANNEL* 信号。

2. 控制逻辑

对于 arid, arlen, arburst, arlock, arcache, arprot, rready, awid, awlen, awburst, awlock, awcache, awprot, wid, wlast, bready 等信号，由于未实现 burst 传输以及乱序执行，故令这些信号为固定值。

对于 araddr, arsize, awaddr, awsize, wdata 等信号，优先传输 data sram 上的数据，

当 data sram 握手失败时，传输 inst sram 上的数据，当 inst sram 握手失败时，数据保持。

对于 arvalid, awvalid, wvalid 等握手信号，先根据类 SRAM 接口判断主设备是否发起了请求，再判断是不是写，还要判断从设备是否接收到了地址。

原实现的 AXI 模块：

1. 接口功能描述

原设计实现的 AXI 模块具有 46 个端口，其中 26 个输入端口，20 个输出端口。这 46 个端口信号中分别是：2 个 *GLOBAL* 信号，7 个 *WRITE ADDRESS CHANNEL* 信号，6 个 *WRITE DATA CHANNEL* 信号，4 个 *WRITE RESPONSE CHANNEL* 信号，7 个 *READ ADDRESS CHANNEL* 信号，6 个 *READ DATA CHANNEL* 信号，12 个 CPU 输入信号，2 个 *CACHE* 信号。由于大赛提供的 IP 对 *AWLOCK，AWCACHE，AWPORT，ARLOCK，ARCACHE，ARPORT* 这 6 个信号不响应，故在 AXI 接口模块中这 6 个信号没有具体实现。

2. 控制逻辑

由于 AXI 总线协议[6]具有 5 个 CHANNEL，所以在 AXI 接口模块中使用了 5 个独立的状态机分别对信号进行控制，这 5 个状态机分别为 AW 状态机，W 状态机，B 状态机，AR 状态机以及 R 状态机。

对于 *WRITE ADDRESS CHANNEL* 状态机，如图 5 所示，初始状态为 AW_IDLE，所有 *WRITE ADDRESS CHANNEL* 的输出均为 0；紧接着进入 AW_START 状态，状态机接收 CPU 输入的信号，此时对于要写入的地址会执行一个判断，如果要写入的地址为 0，会进入 AW_IDLE 状态，否则就进入 AW_WAIT 状态，等待接收 AWREADY 握手信号，当接收到此信号时，进入 AW_VALID 状态，等待接收 BREADY 握手信号，当接收到此信号时，一次地址传输就已完成，状态机重新进入 AW_IDLE 初始状态。

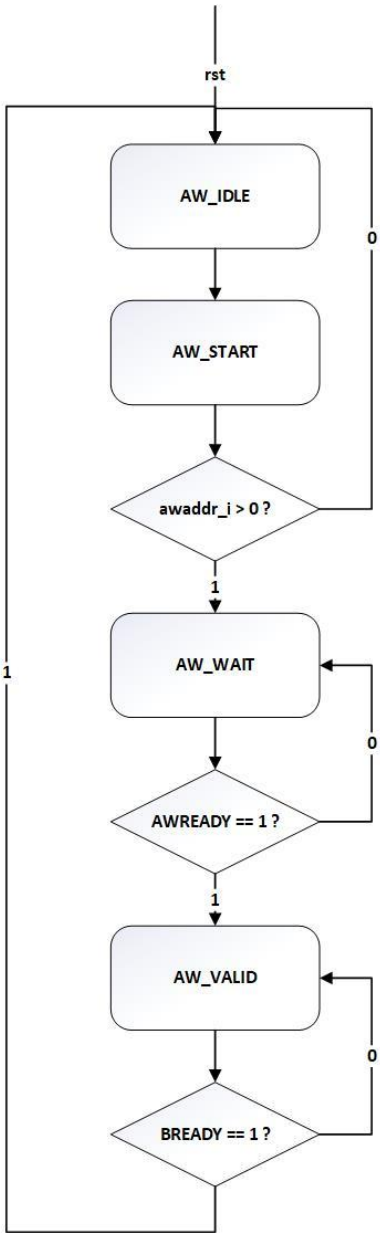

图 5：AW 状态机

对于 *WRITE DATA CHANNEL* 状态机，如图 6 所示，初始状态为 W_INIT，所有 *WRITE DATA CHANNEL* 的输出均为 0，等待接收 AWREADY 握手信号，在接收到此信号时，进入 W_TRANSFER 状态，状态机此时接收 CPU 输入的信号，开始传输数据，对于 burst 传输，在此状态时有一个 count 计数器对写数据计数，对于写入的字节 awsize_i 有一个判断，当写入大于四个字节时会进入 W_ERROR 状态，否则就进入 W_READY 状态，在 W_READY 状态时等待接收 WREADY 握手信号，当接收到此信号时，进入 W_VALID 状态，在此状态判断 count 计数器里的值是否等于输入的 burst 传输大小 awlen_i，当相等时表示一次 burst 传输完成，状

态机重新进入 W_INIT 初始状态，否则进入 W_TRANSFER 状态，接着写入数据；在 W_ERROR 状态的逻辑与 W_VALID 状态大致相同，都是对 count 计数器的值进行判断。

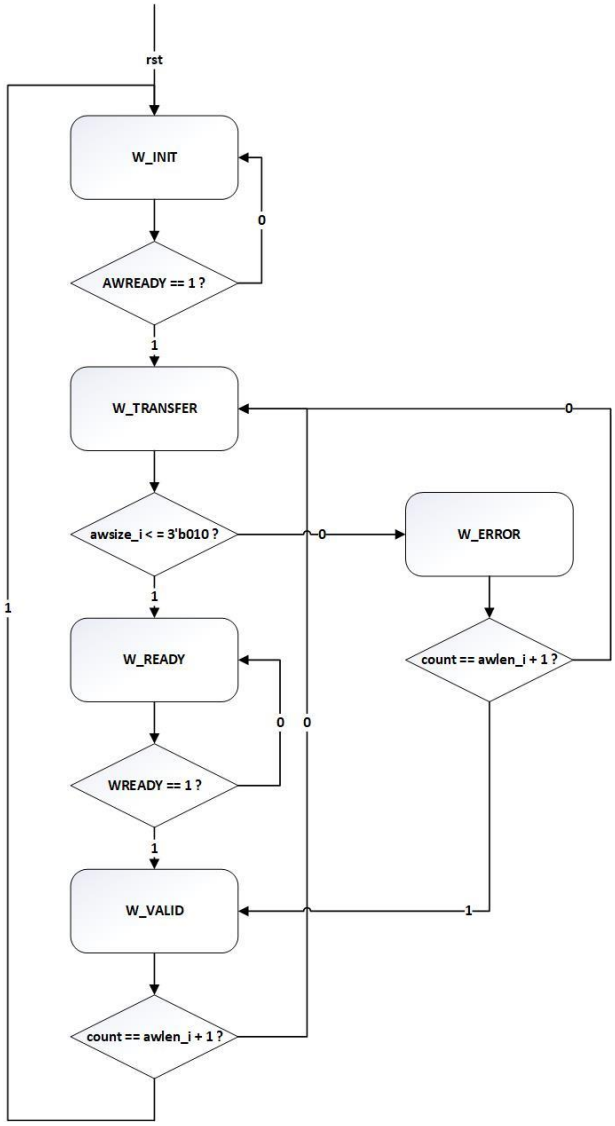

图 6：W 状态机

对于 *WRITE RESPONSE CHANNEL* 状态机，如图 7 所示，此状态机较为简单，初始状态为 B_IDLE，紧接着就进入 B_START 状态，等待接收 BVALID 握手信号，当接收到此信号时，进入 B_READY 状态，在此状态时将 BREADY 握手信号赋为高电平，这样一次 *WRITE RESPONSE* 就传输完成，状态机重新进入 B_IDLE 初始状态。

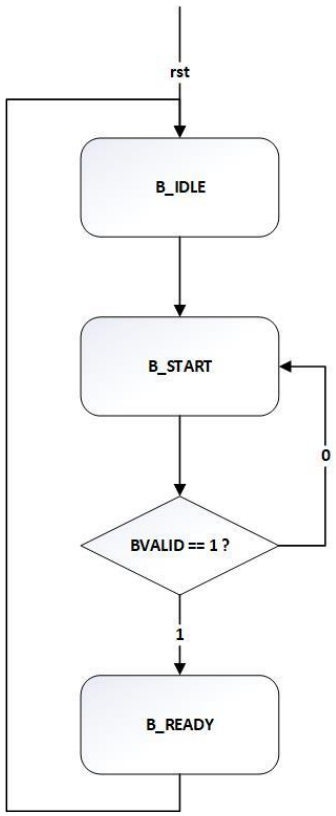

图 7：B 状态机

对于 *READ ADDRESS CHANNEL* 状态机，如图 8 所示，初始状态为 AR_IDLE，所有 *READ ADDRESS CHANNEL* 的输出均为 0；紧接着进入 AR_START 状态，状态机接收 CPU 输入的信号，接着进入 AR_WAIT 状态，等待接收 ARREADY 握手信号，当接收到此信号时，进入 AR_VALID 状态，等待接收 RLAST 信号，当接收到此信号时，说明一次 burst 读取完成，地址传输也就完成，状态机重新进入 AR_IDLE 初始状态。

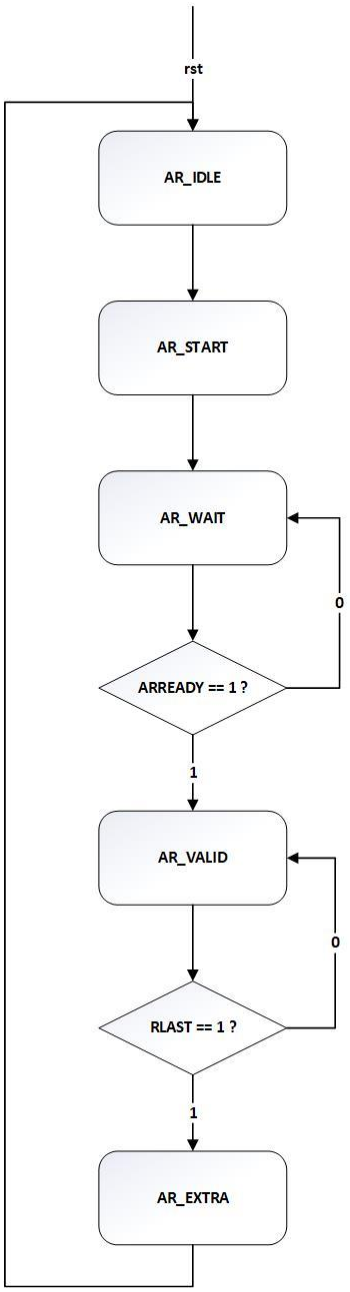

图 8：AR 状态机

对于 *READ DATA CHANNEL* 状态机，如图 9 所示，初始状态为 R_CLEAR，紧接着进入 R_START 状态，等待接收 RVALID 握手信号，当接收到此信号时，进入 R_READ 状态，状态机接收来自总线的数据，将总线上的 RDATA 写入内部寄存器里，对于不同的 burst 读类型，寄存器地址每次的改变量不同，接着进入 R_VALID 状态，等待接收 RLAST 信号，当接收到此信号时，说明一次 burst 传输完成，状态机重新进入初始 R_CLEAR 初始状态。

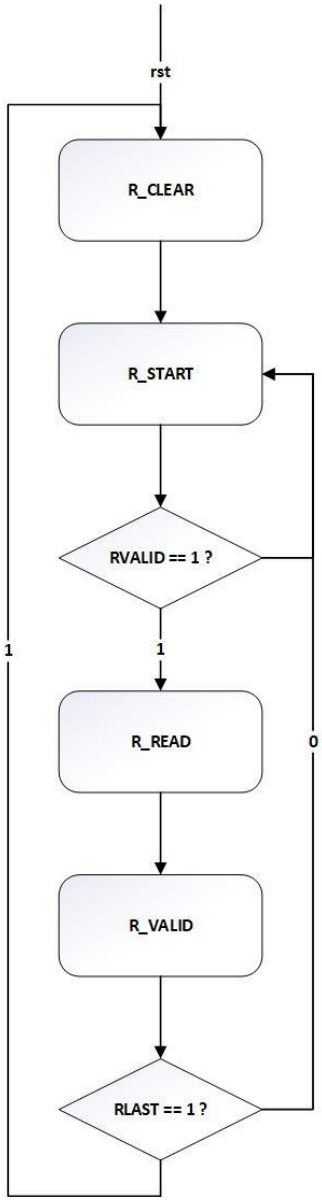

图 9：R 状态机

（十）仲裁器设计

1. 设计思路

为了解决 AXI 访存延迟的问题，CPU 和总线之间需要增加一个仲裁器进行调度。仲裁器需要完成的工作是，接收 CPU 发出的取指和访存的类 SRAM 请求，然后利用转换桥将其转换为 AXI 请求发往 AXI RAM 和 Confreg。在数据有效信号未产生之前，仲裁器必须控制 CPU 暂停，以防 CPU 取到错误的指令或者数据。

由于可能出现连续两次取指或者访存时，请求的地址相同的情况，所以可以针对这种情况进行简单的优化：记录上一次取指和访存的地址以及数据，在读取时

检查地址是否相同，若相同则直接将数据送出而不发出额外的请求。写数据时同理，如果写入的数据长度是四字节，可以直接记录上次写入的地址和数据，下次读取时遇到相同的地址即可直接传递上次写入的数据。需要注意的是，根据 MIPS 地址空间的约定，kseg1 段的虚拟地址（0xA0000000~0xBFFFFFFF）不能被缓存，所以上述优化不应该对这段地址空间生效。

2. 状态机

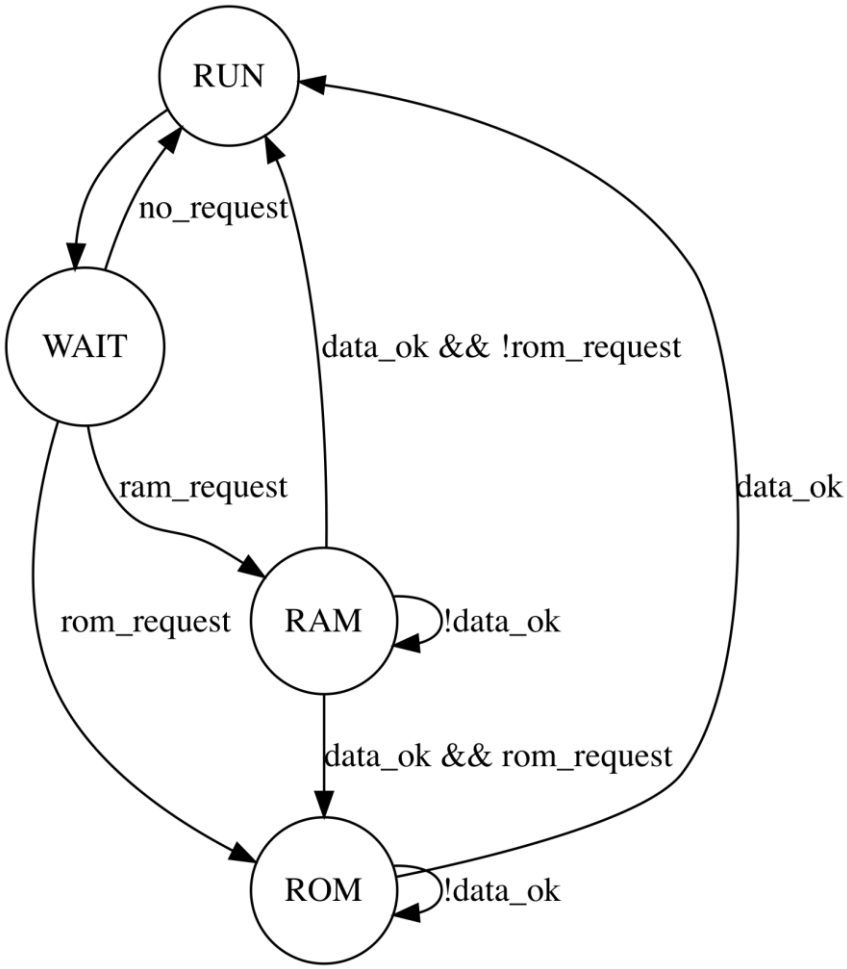

图 10：仲裁器状态机

仲裁器的实现基于上述状态机，状态机中共有四个状态：RUN 状态代表 CPU 运行，此状态时 CPU 暂停信号无效，除此之外所有状态的暂停信号均有效；WAIT 状态代表等待，此状态时会对 CPU 发出的取指和访存请求进行分析，决定下一步的状态转移；RAM 状态代表向数据内存发出数据和地址请求；ROM 状态代表向指令内存发出数据和地址请求。

RUN 状态通常只持续一个时钟周期，然后状态机会无条件转移到 WAIT 状态，

对 CPU 发出的请求进行判断。

WAIT 状态时，仲裁器状态机会判断 CPU 传入的请求，如果存在访存请求，则优先转移到 RAM 状态；否则，如果只存在取指请求，直接转移到 ROM 状态。判断存在访存和取指请求的依据是，此次发出的地址和上次缓存的地址不同，并且访问的地址不属于 kseg1 段。如果不存在任何请求，直接转移回 RUN 状态。

RAM 状态时，仲裁器会向总线请求数据访问。当数据未准备好时状态机将维持在 RAM 状态，直到数据准备就绪——如果存在取指请求，状态机会转移到 ROM 状态，否则转移到 RUN 状态。

ROM 状态时，仲裁器会向总线请求数据访问。ROM 状态和 RAM 类似，区别是当数据准备就绪时直接转移到 RUN 状态，不进行额外的判断。

3. 优缺点

使用基于这种仲裁方式的仲裁器最大的优点就是无需编写复杂的代码，并且调试简单。仲裁器工作时，发送地址请求的规律基本恒定，出错的概率极小，在一定程度上避免了错误的发生。

但是这样做的缺点也十分明显：由于没有 Cache，并且仲裁器不会向 AXI 总线请求进行 Burst 传输，基于这种结构的系统会在等待数据传输上消耗大量的时间。CPU 的速度不能得到完全的发挥，整个系统的性能必定大打折扣。

（十一）辅助工具

1. 概述

在进行系统设计的过程中，为了把控工程的质量，许多问题都需要人工解决。但是有一些问题可以被归约成一种算法，或是一系列流程。这样我们只需要编制解决这些问题的程序，然后给定特定的输入，把剩下的事情交由计算机自动完成即可。有了一些自动化工具的辅助，整个系统设计的过程就能节约不少时间。

在实际的设计过程中，我们根据项目遇到的不同需求，开发了三个工具软件来辅助我们的设计工作。这些软件本身和 CPU 系统的设计没有关系，但是它们的存在保障了我们设计的正确性和高效性，同时也辅助了我们的调试与查错工作。

以下提及的工具均使用 Python 3.6 编写，且均可在提交包中的“attachment/util”目录中找到并运行。详情参考第三部分中的“设计交付物说明”。

2. autowire

在编写CPU核心部分的代码时，为了尽可能地降低问题的出现，同时节省工作量，我们使用 Vivado 内建的 Block Design 功能进行了 CPU 核心的顶层模块设计。使用Block Design的好处是直观，模块与模块之间的连接关系可以直接由工具绘制，一目了然。

但 Block Design 同样存在许多不足。Block Design 采用了类似 IP 核的封装和生成方式，并不能直接产生 Verilog HDL 代码，并且只能由 Vivado 打开，这样大大降低了我们项目的兼容性和可移植性。除此之外，其缺乏必要的警告功能，例如我们可以直接将一个宽度为1的信号输出连接到另一个宽度为32的信号输入上，但是 Vivado 并不会提醒我们这类问题，这样也增大了错误隐患与排查成本。

在 2017.4 版本的 Vivado 中，Block Design 生成的 HDL 文件不会根据模块的名称对例化出的模块命名，而是采用固定的命名。比如其生成了只包含一个模块的 HDL 文件，模块例化的名称会被取名叫做“inst”（应该是“instance”的缩写），但在 CPU 的设计当中，取指模块输出的指令信号也可能叫做“inst”（“instruction”的缩写），这就导致了命名上的冲突。这样生成出来的文件甚至会因为内部信号命名的问题而无法正常执行仿真和综合。

鉴于 Vivado 本身的 Block Design 功能设计上的不足，我们开发了一款软件，其既可以借助 Block Design 的“绘图式”设计功能对大量模块之间的输入输出端口进行连线，又可以绕过自带的生成器，转而由软件接管源代码的生成工作。这款工具被我们命名为“autowire”。大赛提交的 CPU 核心部分的顶层模块（Uranus.v）就是由这款工具生成的。

autowire 能够解析 Vivado Block Design 产生的 *.bd 文件，这种文件以 XML 的形式描述了 Block Design 窗口中模块框图之间的连接方式，以及 Block 本身对外的输入输出端口。autowire 同样会对项目进行扫描，提前获取到项目中各个 Verilog 模块的端口信息。根据上述两种信息，autowire 就可以将 *.bd 文件中的形式化描述转换为单个的 Verilog HDL 源代码（*.v）文件了。

autowire 在功能上有诸多优点。首先，其代码生成速度明显快于 Block Design，针对一个较为复杂的设计，autowire 只需要不到一秒钟的时间就能将其转换为无错的 Verilog HDL 源代码。其次，autowire 产生的源代码结构简单，具有一定的可读性，方便进行后期修改，并且不存在任何命名冲突问题。此外，针对两个不同宽度端口之间存在连线的问题，autowire 会进行检测和警告提示（WARNING），一

定程度上可以避免接错线的情况发生。

可以使用以下命令来执行 autowire：

```
$ python3 autowire.py project_base_path *.bd [output]
```

例如：

```
$ python3 autowire.py ../src ../bd/Uranus.bd ../src/Uranus.v
```

3. assembler

顾名思义，这是一个针对自实现的 CPU 设计的一款汇编器。由于 CPU 在设计过程中需要不断地进行仿真层面的功能测试，尽可能早的排查出潜在的问题，而使用诸如 GCC 一类的编译工具链产生的代码文件又不太可控，且配置相对复杂，我们针对处理器目前实现的 57 条 MIPS 指令，结合 MIPS ISA 规范，自行设计实现了对应的汇编器。

汇编器的功能非常简单，并且较为普通。除了支持 57 条 MIPS 常用指令外，汇编器还支持“nop”、“li”、“la”和“move”这四种伪指令；支持使用标签（label）标记和引用分支/跳转指令的目标；默认输出能够在 Verilog 仿真中被“$readmemh”函数读取的十六进制文本格式文件，每条指令占一行，并且其后附带指令的助记符形式作为注释。

实现汇编器的目的除了一定程度上方便仿真测试之外，还为日后处理器的深度开发打下了基础。汇编器的代码结构简单，复用性强，耦合程度较低，方便扩展或者根据全新的指令集架构重写。

可以使用以下命令来执行汇编器：

```
$ python3 mips_asm.py *.s [output]
```

例如：

```
$ python3 mips_asm.py ../asm/test.s ../asm/test.out
```

4. disassembler

顾名思义，这是一个针对自实现 CPU 设计的一款反汇编器。在大赛的性能测试和功能测试的调试中，很多情况下都需要结合查看反汇编代码来进行问题的排查。尤其是功能测试，虽然发布包中附带了测试程序的汇编源文件，但是文件中还是存在较多的冗余信息，不利于我们的查看。

这款反汇编器可以根据汇编指令的十六进制字符串（可直接从 *.coe 文件中提取），来输出其对应的助记符形式，并且以 0xBFC00000 为基地址，在每条指令的

末尾以注释的形式标注指令的地址，方便结合 Trace 比对快速调试。同样，反汇编器暂时只支持大赛规定的 57 条指令。

可以使用以下命令来执行反汇编器：

```
$ python3 disasm.py text_file [output]
```

例如：

```
$ python3 disasm.py ./abc.txt ./abc.s
```

三、设计结果

（一）设计交付物说明

提交包的文件结构如下：

- design.pdf：文件，此设计报告；
- score.xls：文件，功能测试、性能测试的分数汇总；
- attachment：目录，一些附件；
- soc_axi_func：目录，SoC 的功能测试环境；
- soc_axi_perf：目录，SoC 的性能测试环境；
- soft：目录，功能测试与性能测试中所执行的程序。

其中“attachment”目录并非大赛所要求的提交目录格式中的内容，此目录包含了设计报告中引用的一些附件，以及设计过程中使用到的所有自行开发的辅助工具软件。具体内容如下：

- cpu_datapath.pdf：文件，由 CPU 核心部分的 Block Design 导出的内部模块的具体组成以及连接方式；
- graphviz：目录，包含使用 Graphviz 绘制的状态机源文件和 PNG 图像；
- photo：目录，包含全部上板验证时拍摄的照片；
- util：目录，包含自行开发的辅助工具软件：
  - assembler：目录，内含一个使用 Python 实现的 MIPS 汇编器，仅支持处理器实现的 57 条指令，以及形如“nop”、“li”、“la”一类的伪指令；
  - autowire：目录，内含一个使用 Python 实现的转换程序，可以将 Vivado 的 Block Design 转换为 Verilog HDL 源码（*.v 文件）；

- disassembler：目录，内含一个使用 Python 实现的反汇编器，用于将十六进制格式的 MIPS 汇编转化为助记符形式表示，并且以 0xBFC00000 为基址标注各个指令的地址。仅支持处理器实现的 57 条指令以及“nop”伪指令。

所提交内容的仿真、综合以及上板演示的具体步骤和大赛要求的步骤完全一致，此处不再赘述。需要注意的是，提交包内未包含 gs132 生成的“golden_trace.txt”，大赛评审方应另行提供。

（二）设计演示结果

1. 运行 AXI 功能测试

(1) Tcl Console 输出：

```
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0x9fc007cc
        [  32000 ns] Test is running, debug_wb_pc = 0x9fc04a68
        [  42000 ns] Test is running, debug_wb_pc = 0x9fc04b18
        [  52000 ns] Test is running, debug_wb_pc = 0x9fc04bc0
        [  62000 ns] Test is running, debug_wb_pc = 0x9fc04c48
        [  72000 ns] Test is running, debug_wb_pc = 0x9fc04ce8
        [  82000 ns] Test is running, debug_wb_pc = 0x9fc04d98
run: Time (s): cpu = 00:00:15 ; elapsed = 00:00:24 . Memory (MB): peak = 852.430 ; gain = 0.000
run all
        [  92000 ns] Test is running, debug_wb_pc = 0x9fc04e44
        [ 102000 ns] Test is running, debug_wb_pc = 0x9fc04efc
        [ 112000 ns] Test is running, debug_wb_pc = 0x9fc04f9c
        [ 122000 ns] Test is running, debug_wb_pc = 0x9fc05050
        [ 132000 ns] Test is running, debug_wb_pc = 0x9fc050f4
        [ 142000 ns] Test is running, debug_wb_pc = 0x9fc051b0
run: Time (s): cpu = 00:00:04 ; elapsed = 00:00:22 . Memory (MB): peak = 875.754 ; gain = 0.000
```

图 11：功能测试的 Tcl Console 输出

(2) 仿真波形输出：

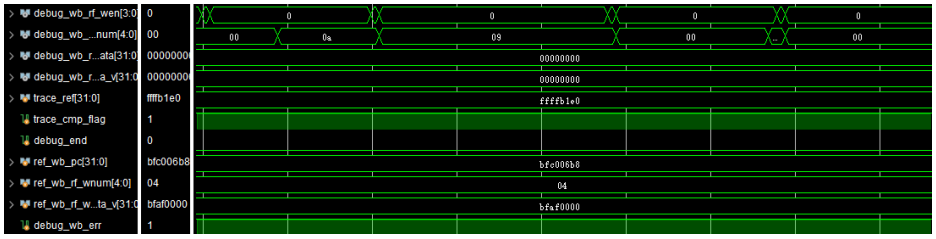

图 12：功能测试的仿真波形输出

(3) 上板验证结果：

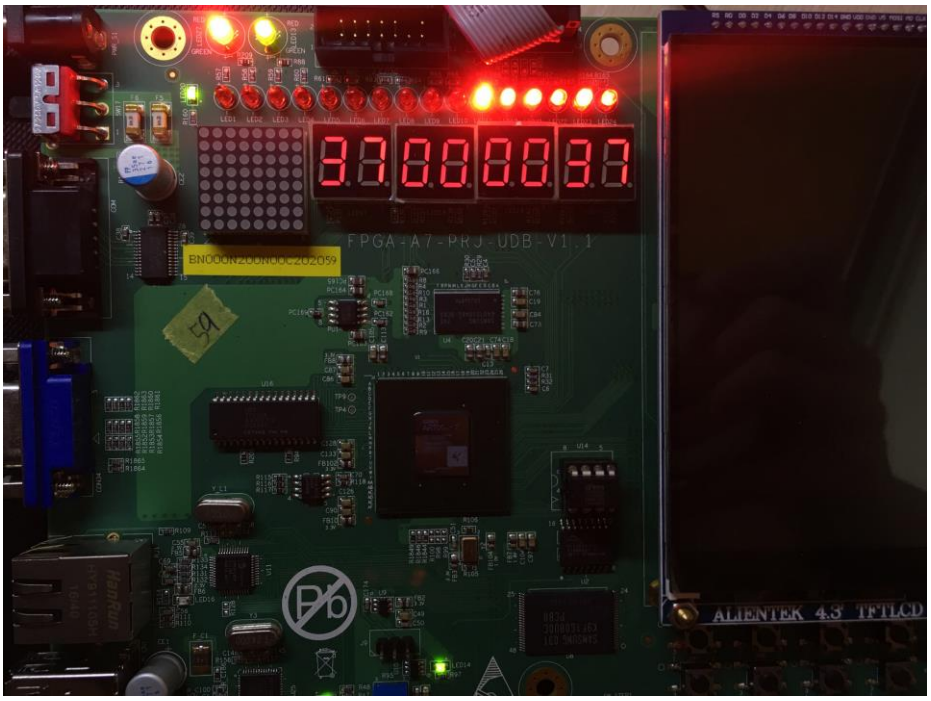

图 13：功能测试上板运行过程

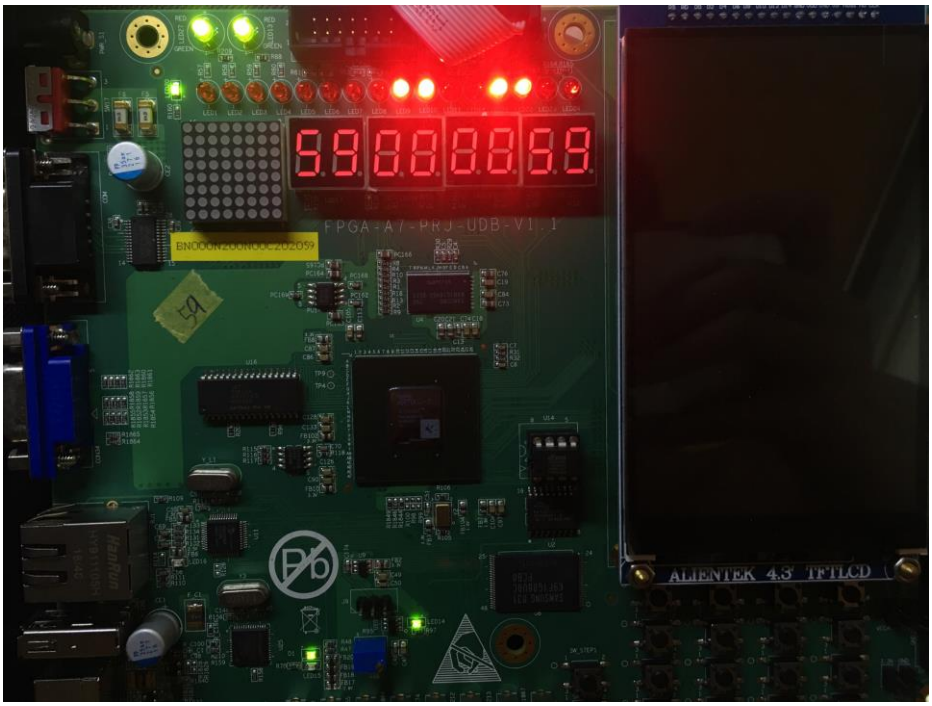

图 14：功能测试上板运行完成

2. 运行记忆游戏

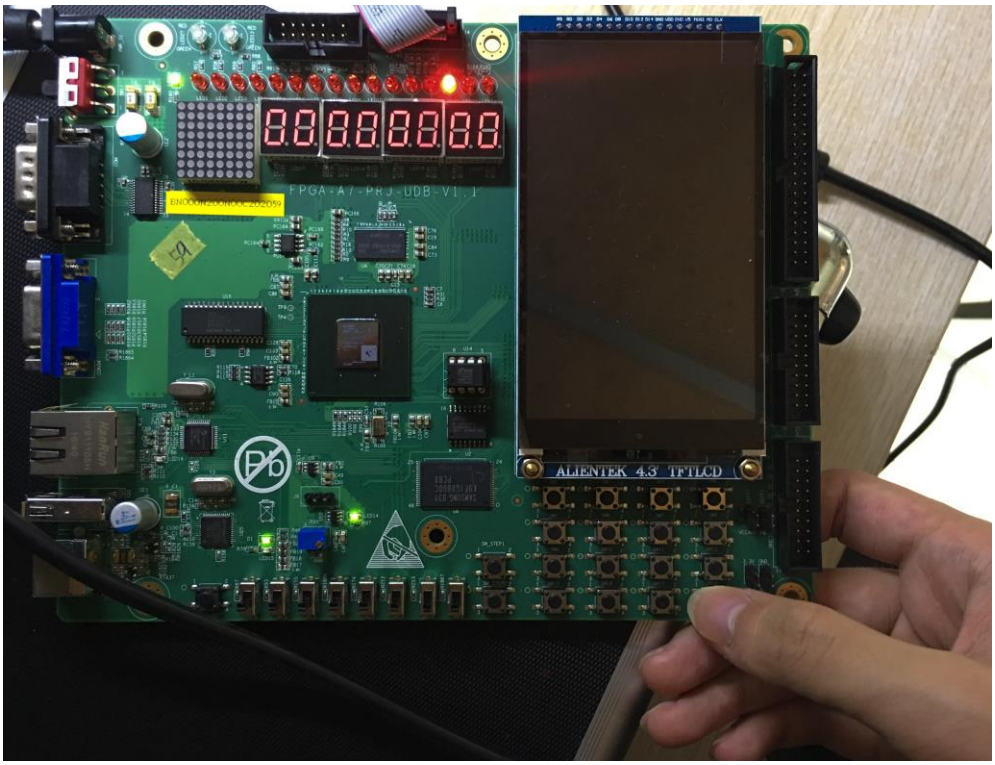

图 15：记忆游戏上板运行

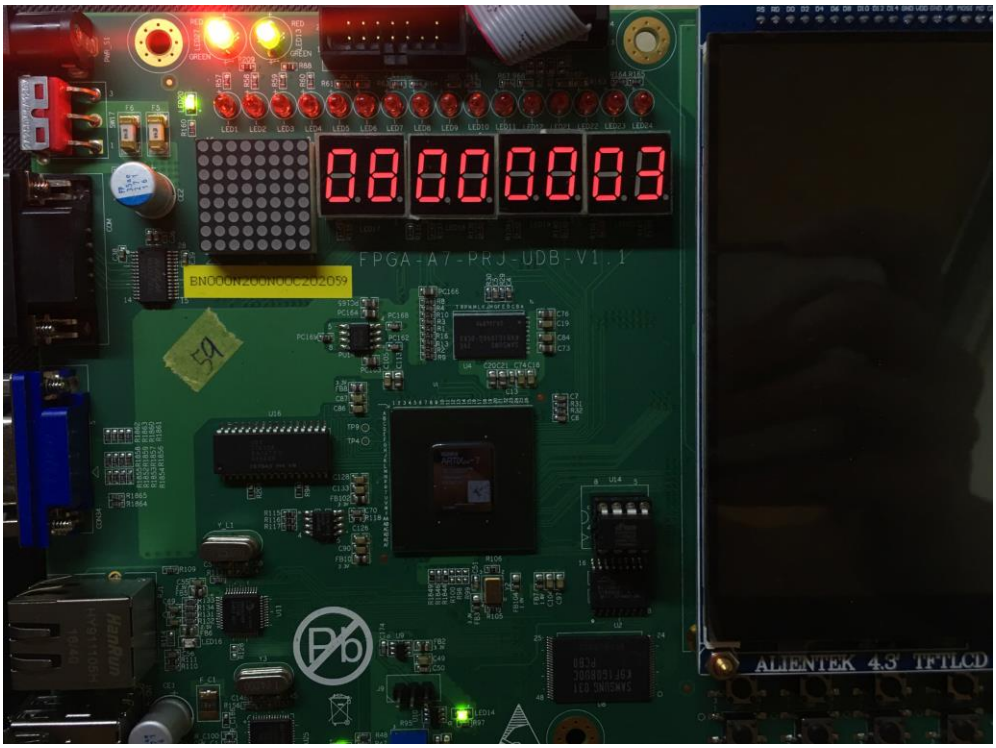

图 16：记忆游戏输入错误

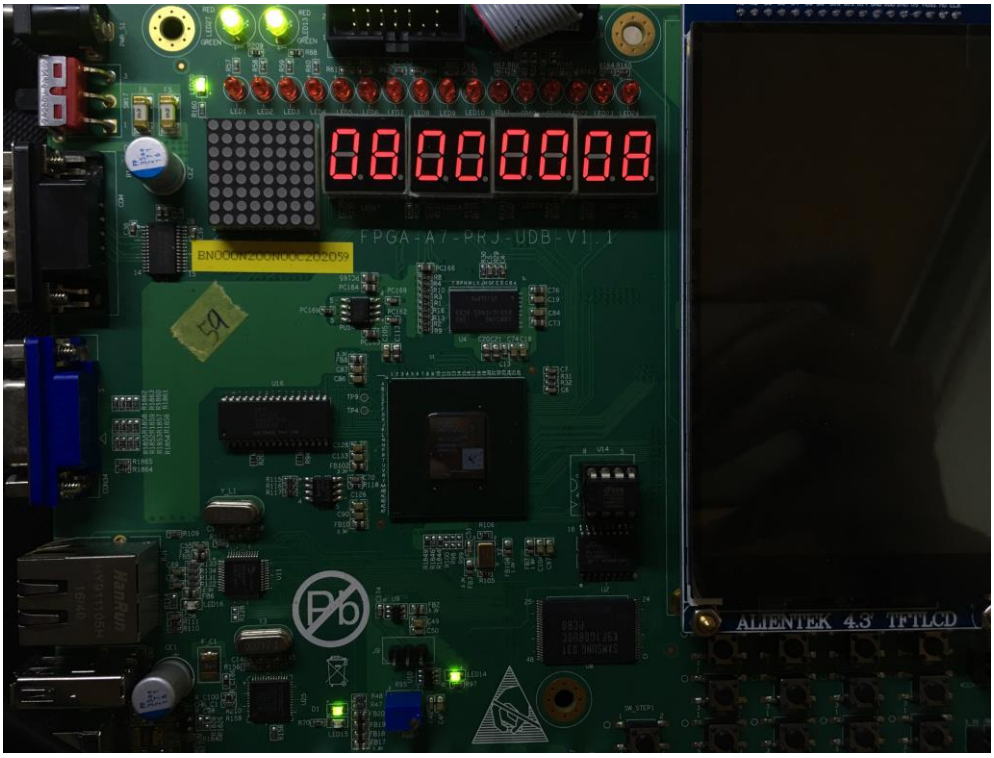

图 17：记忆游戏输入正确

3. 运行性能测试

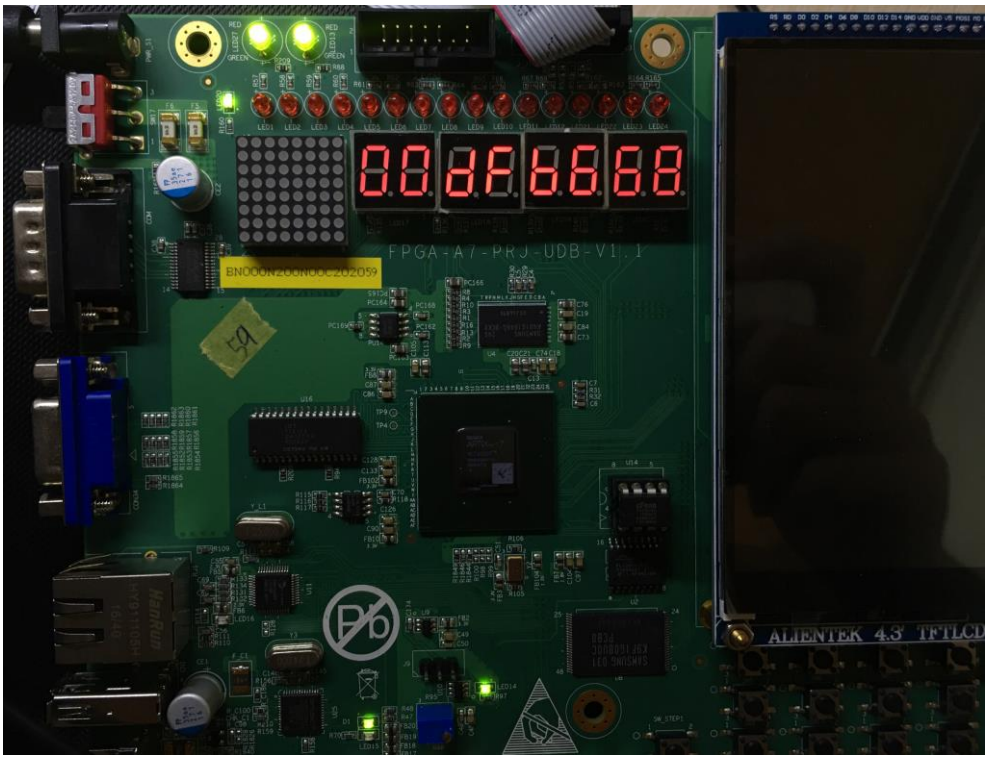

图 18：性能测试运行第一个功能点

四、参考设计说明

由于之前对流水线 CPU 的设计较为陌生，本次大赛提交的 CPU 核心部分的大部分实现都参考和借鉴了《自己动手写 CPU》一书中所述的教学版 OpenMIPS 的设计[7]。包括数据通路的总体结构、流水线各级功能的划分、流水线控制信号的发出、仿真时 RAM/ROM 的实现方式以及异常处理的思路，等等。但是除去总体设计，大赛提交的 CPU 在许多细节上的实现方式均有改动和优化，例如针对同步读取 RAM/ROM 而对 IF、MEM 级做出的修改、处理数据前推的模块、除法器的运算和处理、流水线中间级的实现等。

乘除法运算调度模块（MultDiv.v）的部分思路参考了清华大学的 NaiveMIPS 项目的相关实现（multi_cycle.v），并在此基础上针对 CPU 的意外暂停、可能出现的潜在的乘除法运算导致的 CPU 死锁等特殊情况做出了优化。

总线部分的 SRAM 到 AXI 转换桥模块（cpu_axi_interface.v）为大赛官方提供，我们未对其源代码做出任何修改，只是在此基础上对其工作原理加以理解和吸收，并且完成了仲裁器（SRAMArbiter.v）的设计和开发。

五、参考文献

[1] “系统能力培养大赛” MIPS 指令系统规范 v1.01, 2018, 系统能力培养大赛.

[2] Morgan Kaufmann, Dominic Sweetman, See MIPS Run Linux, 2nd Edition, 2006.

[3] MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture. Revision 3.02.

[4] MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set. Revision 3.02.

[5] MIPS Architecture For Programmers Vol. III: MIPS32/microMIPS32 Privileged Resource Architecture. Revision 3.02.

[6] AMBA® AXI Protocol v1.0 Specification.

[7] 雷思磊, 自己动手写 CPU, 2014, 电子工业出版社.