Uranus 设计报告

第二届全国大学生计算机系统能力培养大赛
决赛提交

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。

决赛阶段，我们在预赛设计的基础上进一步实现了以 CPU 为核心的 SoC 硬件系统，包含了诸如 UART 控制器、SPI Flash、VGA 控制器、以太网等必要的外设。

（一）设计变更说明

1. 变更意图

- 为了方便修改中断或者异常处理入口，便于系统程序的开发，需要实现中断或者异常处理的入口点的变更；
- 为了扩展指令集，使 CPU 支持更丰富的操作，需要修改其 ID、EX 模块；
- 为了更好地验证 CPU 的设计，并实现操作系统等软件的运行，需要基于 Uranus CPU 搭建一个 SoC；
- 为了实现程序的快速调试，以及引导操作系统启动，需要实现一个具备基本功能的引导程序。

2. 变更内容

基于以上变更意图，我们做出的变更内容如下：

- 遵循 MIPS 规范，增加了 CP0.EBase 寄存器，实现了异常处理入口的修改；
- 修改了 ID、EX 模块，增加了 9 条指令，分别是：CLO、CLZ、MUL、MADD、MADDU、MSUB、MSUBU、MOVN、MOVZ；
- 基于 Uranus CPU 核心搭建了一个 SoC，支持在大赛实验箱上运行；
- 针对 Uranus SoC 设计实现了引导程序 UBW（Unlimited Boot Works）；
- 移植了嵌入式实时操作系统 μC/OS-II。

3. 达到的效果

最终实现了一个能在大赛实验箱上运行的基于 Uranus 的 SoC，该 SoC 含以下内容：

- Uranus CPU；
- DDR3 SRAM 控制器，支持 DMA 方式；
- SPI Flash 控制器，支持软件读写 SPI Flash；
- 16550 兼容串口控制器，支持串口调试；
- GPIO 控制器，支持软件控制双色 LED 灯、单色 LED 灯和数码管，读取开关和 4*4 矩阵键盘状态；
- VGA 控制器，支持 VGA 图像输出；
- 以太网控制器，支持 100M 以太网通信；
- UBW 引导程序，支持 Flash 启动；
- 嵌入式实时操作系统 μC/OS-II。

关于变更的具体内容，可以参考第三部分（SoC 总体设计方案）以及第四部分（软件及辅助工具）。

二、CPU 核心部分设计方案

（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：

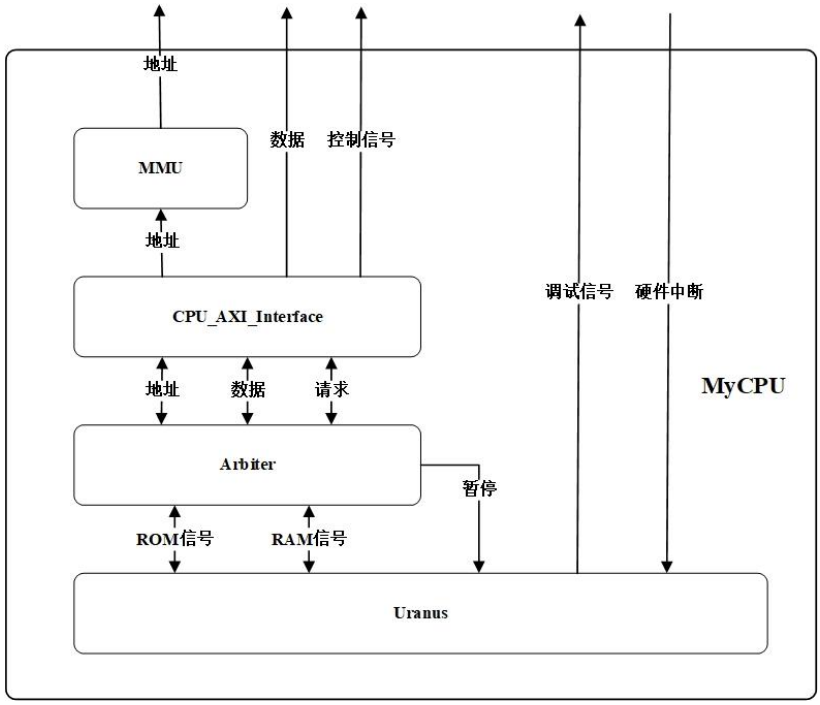

图 1：myCPU 的系统结构

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。

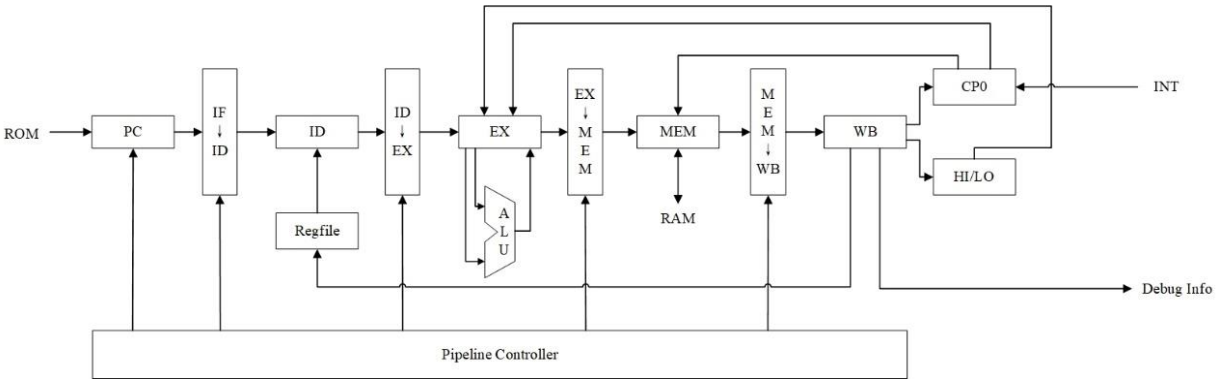

图 2：CPU 核心部分的数据通路

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。

此外，我们还自行扩展了 9 条指令，具体如下：

- 逻辑运算指令：CLO、CLZ、MUL、MADD、MADDU、MSUB、MSUBU。
- 数据移动指令：MOVN、MOVZ。

4. 内存管理

根据 MIPS 规范[3, 4, 5]，虚拟地址划分如下：

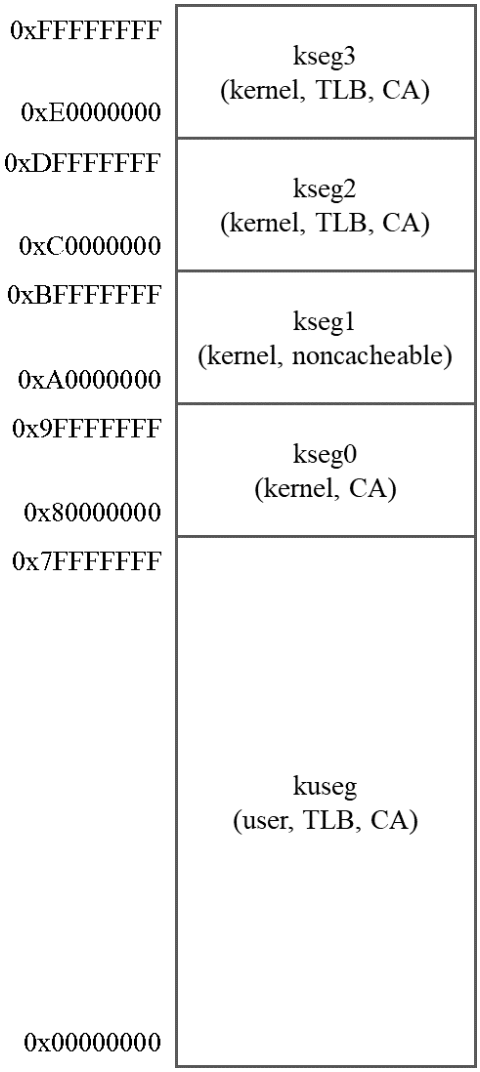

图 3：MIPS 地址空间

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 和 EBase 这三个额外的寄存器，其中 EBase 寄存器可以实现异常处理入口的重定义。

根据 MIPS 特权态资源规范[5]，为了方便系统程序的开发，增强 MIPS 处理器体系的灵活性，我们可以通过修改 CP0.Status 寄存器的 Bev 位来控制和修改异常入口。默认情况下，Bev 位为 1，异常处理程序入口的基地址为 0xBFC00200，如果软件将该位置零，则入口基地址为 CP0.EBase 的值。复位时，EBase 的值为 0x80000000。

MIPS 规范中定义异常入口的计算方法为基地址加偏移量，针对目前处理器已经实现的情况，所有的偏移量均为 0x180。据此可以算出系统复位时，异常处理程序的入口应当为 0xBFC00380，与大赛规定的实现相符。

（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：

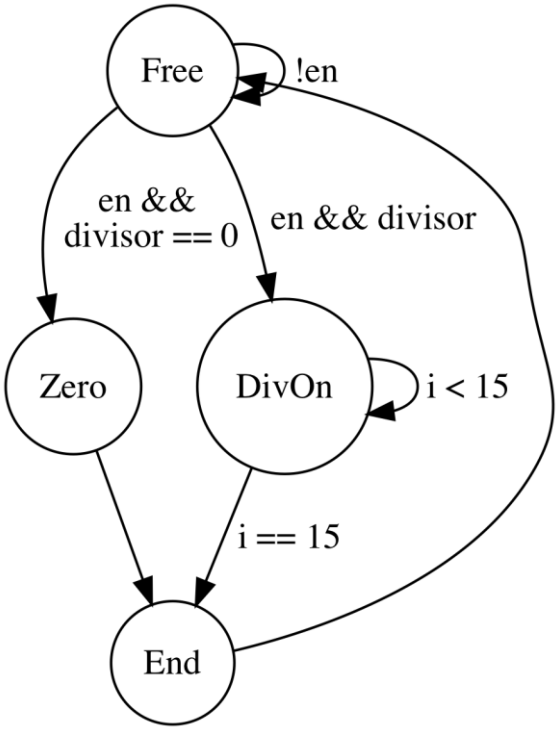

图 4：16 周期除法器的状态机

（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。

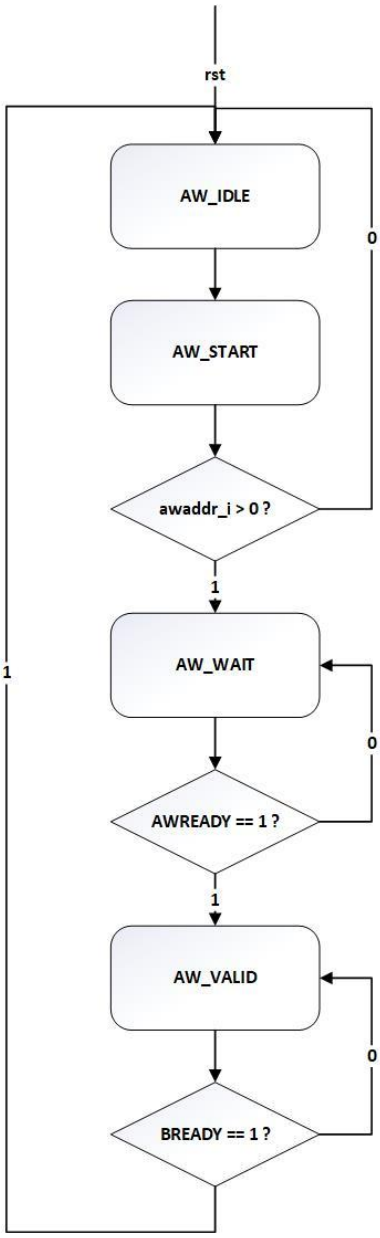

图 5：AW 状态机

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。

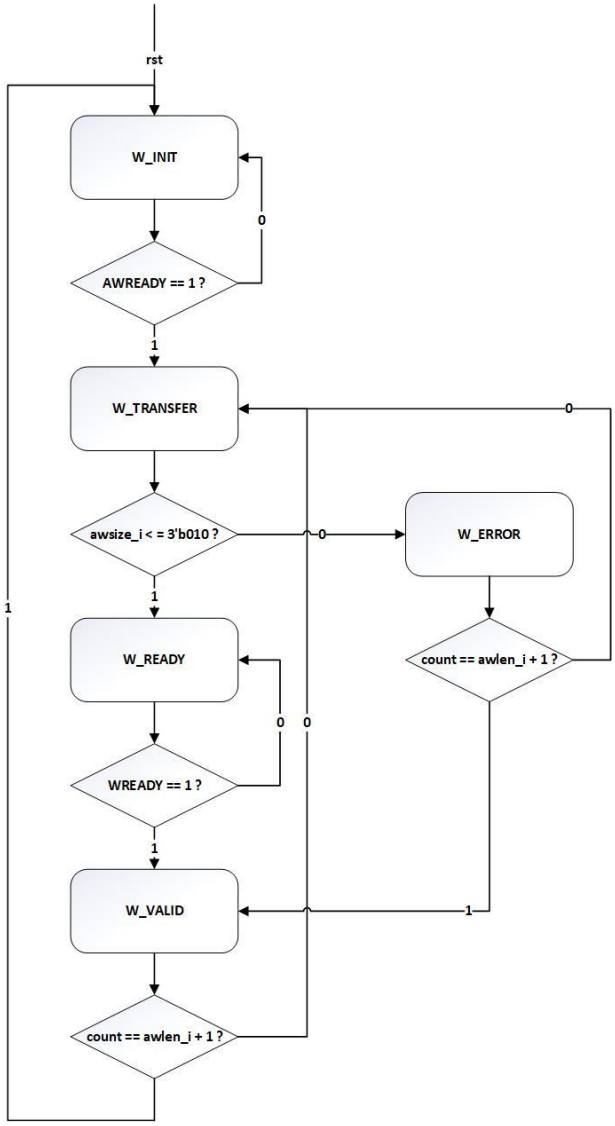

图 6：W 状态机

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。

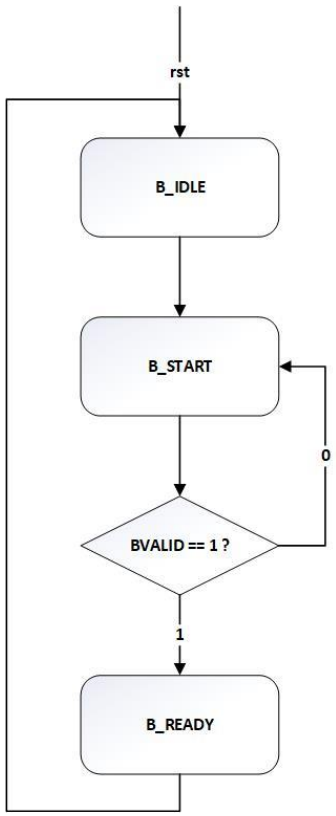

图 7：B 状态机

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。

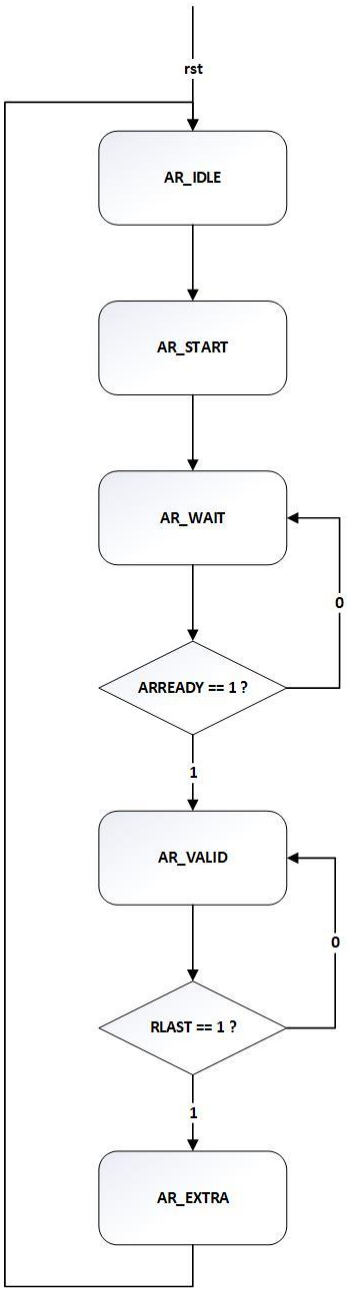

图 8：AR 状态机

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。

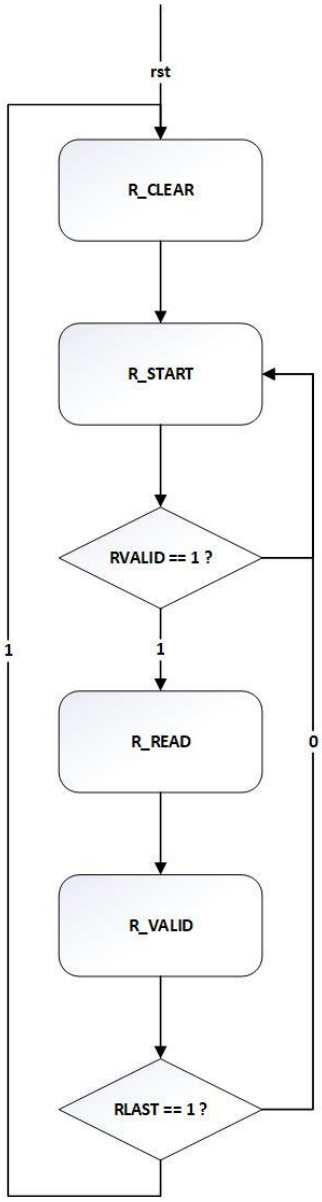

图 9：R 状态机

（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

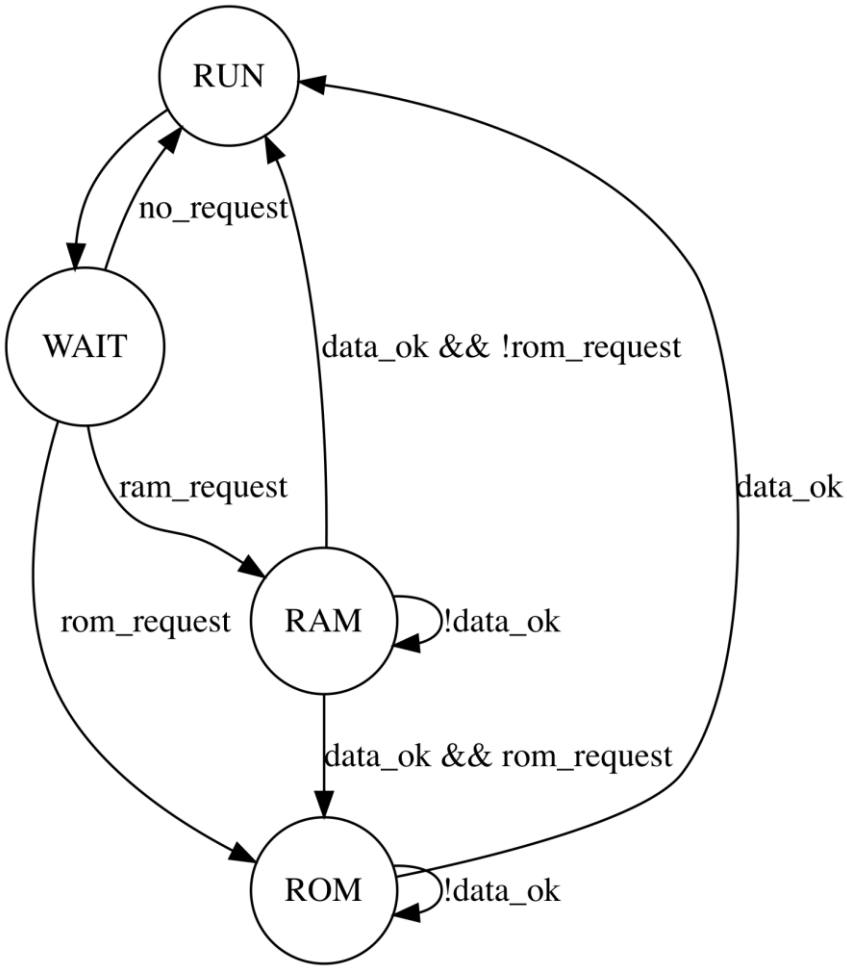

图 10：仲裁器状态机

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。

三、SoC 总体设计方案

（一）总体设计

1. 系统总线设计

SoC 总线结构框图如下所示：

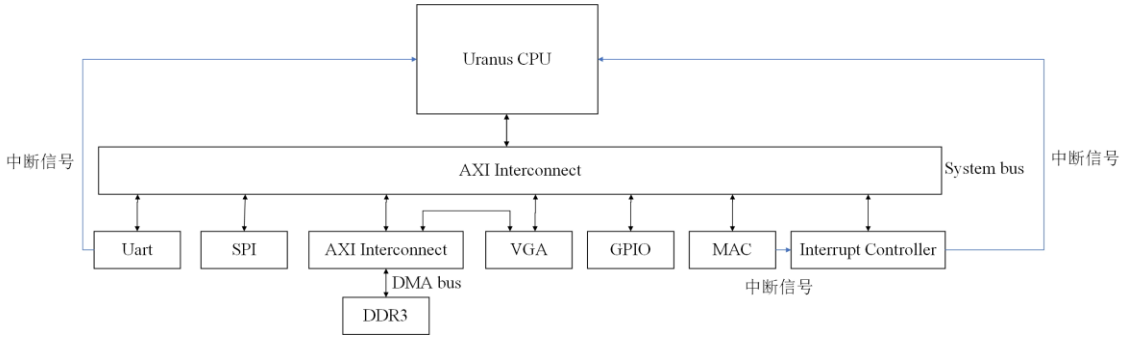

图 11：SoC 总线结构框图

系统总线的设计使用了 Xilinx 提供的 AXI Interconnect（IP），该模块具有一个 AXI 的 slave 端，用来连接 Uranus CPU；具有 7 个 AXI 的 master 端，分别连接 UART 控制器、SPI Flash 控制器、DMA 控制器、VGA 控制器、GPIO 控制器、以太网控制器和中断控制器。

Vivado 除了支持传统的 RTL to Bitfile 的设计流程（即输入 RTL 代码，通过集成后，用 Vivado 来产生 Bitfile），还支持了一种称为系统级集成设计流程（基于 IP 的设计，即可将打包好的 IP 在 Vivado 的 Block Design 中直接进行集成，然后产生 Bitfile 的流程）。SoC 的顶层模块的搭建我们就采用了第二种方法，加速了集成时间，以及降低集成风险。

在 SoC 总体设计阶段时，往往需要使用大量的 Xilinx 或者第三方 IP，这个时候如果采用第一种方法，用手动编码的方式创建顶层模块，一方面是代码量较大，容易出现信号名错误或者信号接线错误等问题；另一方面，这种方式需要预先单独生成所有的 IP，这样的话就无法根据实际情况做出灵活的优化。例如，我们预先生成了固定 32 位地址/数据线宽度的 AXI Interconnect 的 IP 模块，当多个提供了 AXI 总线接口的设备想要连接到这个 AXI Interconnect 时，如果这多个 AXI 接口的地址/数据线的宽度不尽相同，我们就需要自行实现一个对 IP 核的额外封装，使其地址/数据线的宽度符合总线互联模块的要求；并且当我们预先生成 AXI Interconnect 模块时，从设备的地址空间分配已经确定，如果我们想实现地址空间的灵活分配，那我们可能就需要对总线互联模块进行多次重复的定制修改，每次修改都需要对这个 IP 进行单独的综合。

如果我们使用了第二种方法，也既是 Block Design 的方法。这种方法一方面将大量 RTL 代码简化为了图形化操作，减少了集成的时间，降低了集成风险；另一方面，我们就不用预先生成 IP，对应前面提到的例子，Block Design 会自动适应从设备的地址/数据总线宽度，从而改变总线互联模块的地址/数据总线宽度，并且 Block Design 提供了定制化 IP 核的方法，当我们想实现地址空间的灵活分配时，我们只需在该界面输入各个从设备的起始地址和结束地址即可，从而简化了操作。

2. 地址空间分配

SoC 中的地址空间分配由 AXI 总线互联模块实现，分配方案如下表所示。目前的 SoC 设计并未满足预期的设计目标（如实现 NAND Flash 控制器等），考虑到后期可能会进一步实现这些外设，需要预留一部分空间，所以地址空间的分配并不完

全连续。

表 9：外设的地址空间分配

外设名称	类型	起始地址	结束地址
SPI Flash	Memory	0x1FC00000	0x1FCFFFFF
UART	Control Registers	0x1FD00000	0x1FD01FFF
DDR3	Memory	0x00000000	0x07FFFFFF
VGA	Control Registers	0x1FD02000	0x1FD02FFF
GPIO	Control Registers	0x1FD05000	0x1FD05FFF
MAC	Control Registers	0x1FD06000	0x1FD07FFF
Interrupt Controller	Control Registers	0x1FD08000	0x1FD08FFF

3. CPU 中断输入连接

某些外设需要依靠中断机制通知 CPU， 从而实现高效的 I/O 操作。由于 Uranus 只支持电平类型的中断，而以太网控制器的中断类型为边沿类型，所以需要一个 AXI 中断控制器，将边沿类型的中断转换为电平类型的中断发往 CPU。SoC 中各个中断的连接关系如表所示：

表 10：外设的中断连接关系

外设名称	中断类型	中断接收方
以太网控制器	上升沿	AXI 中断控制器
UART	高电平	CPU
SPI Flash	高电平	未使用中断
VGA	高电平	未使用中断

其中，SPI 控制器提供了一个中断信号，我们没有利用该中断；VGA 控制器提供了一个中断信号，该信号会在 VSYNC 脉冲产生之后变为高电平，也就是说 VGA 控制器会在绘制完一帧之后产生一个硬件中断，由于我们并不需要如此实时性强的显示控制，所以我们也没有利用该中断。

（二）外设控制器设计

1. DDR3 内存控制器

内存控制器是内存与 CPU 之间交换数据的重要组成部分，我们使用了 Xilinx 提供的 Memory Interface Generator 这一 IP 来实现该模块。

根据 Memory Interface Generator 的说明文档[7]，由于 DDR3 物理限制，在使用 DDR3 内存之前，需要对 DDR3 的内存颗粒进行校准（Calibration），在此之前是无法直接使用 DDR3 存储数据的。该 IP 核有一个校准完成信号的输出，需要根据此信号做出处理：如果校准完成，则令 DMA 总线 areset 信号有效。

根据这一逻辑关系，我们自行实现了 ResetGenerator 模块来产生这一复位信号，信号列表如表所示：

表 11：ResetGenerator 接口

名称	宽度	方向	描述
ddr_ui_clk	1	input	来自 DDR3 的时钟信号
ddr_ui_clk_rst	1	input	来自 DDR3 的复位信号
ddr_calib_done	1	input	来自 DDR3 的校准完成信号
ddr_interconn_rst	1	output	发往 DMA 的复位信号

2. 串口控制器

串口控制器模块可用于产生串行通信信号时序，我们使用了 Xilinx 提供的 AXI UART 16550 这一 IP 来实现该模块。该 IP 核内部提供了一系列的控制寄存器（Control Registers），可以完成注入设置串口通信的波特率（Baud Rate）、FIFO 大小、中断控制以及中断类型读取等功能。

根据 AXI UART 16550 的说明文档[8]，UART 传输时序大致如下：

发送数据过程：空闲状态，线路处于高电平，当收到发送数据指令后，拉低电平一个数据位的时间，接着数据按低位到高位依次发送，数据发送完毕，接着发送奇偶校验位和停止位，一帧数据结束。

接收数据过程：空闲状态，线路处于高电平，当检测到线路的下降沿，说明线路有数据传输，按照约定的波特率从低位到高位接收数据，数据接收完毕，接着接收并比较奇偶校验位是否正确，如果正确，则通知接收端设备准备接收数据。

根据这一时序，一个简单的 UART 框图如图所示：

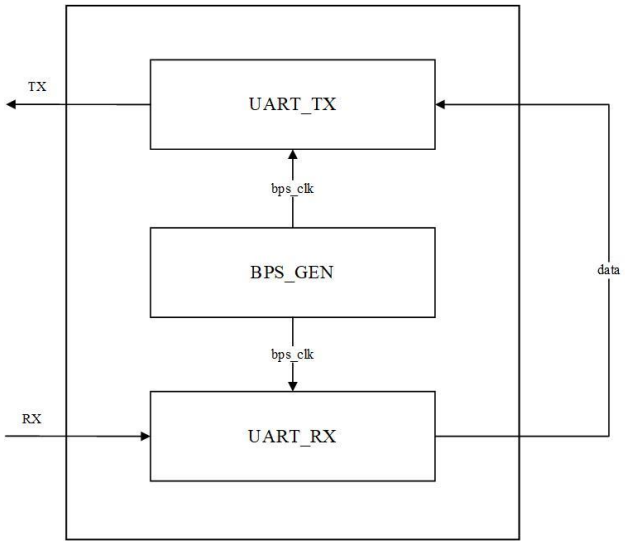

图 12：UART 控制器示意图

串口在初始化时，为了保证传输效率，我们需要给控制器指定一个较高的波特率，BPS_GEN 从而根据此波特率产生一个合适的时钟信号，提供给 UART_RX 和 UART_TX。波特率的设置方法应当参考以下公式：

$$Divisor = \frac{AXI\_CLOCK}{16 \times Baudrate}$$

3. SPI Flash 控制器

实验箱上具有一片 SPI Flash 芯片，为了更好地利用板上资源并存储 UBW 引导程序，所以我们需要一个 SPI Flash 控制器来控制对 NOR Flash 芯片的擦除、读取、写入等操作，我们使用了大赛官方提供的 SoC_up 工程当中的 godson_sbrige_spi.v（spi_flash_ctrl 模块）来实现 SPI Flash 控制器。

该模块提供了一个 AXI 接口，方便接入到系统总线，对 SPI Flash 进行直接读取，这一特性是由 NOR Flash 的物理性质决定的：NOR Flash 支持 XIP（Exectue In Place，原地执行），这意味着存储在 NOR Flash 上的程序不需要复制到 RAM 就可以直接运行。NOR Flash 支持随机读取，读取 NOR Flash 的内容时，如同直接读取 RAM 的内容一样。

CPU 通过该 AXI 接口来访问控制器内部提供的控制寄存器，以便于实现 SPI Flash 的擦出和写入操作。

4. VGA 控制器

为了充分利用实验平台的外设资源，我们使用了 Xilinx 提供的 AXI TFT Controller 这一 IP 来实现 VGA 控制器模块，该模块用于将存储在帧缓冲中的图像数据按照

一定的时序输出到 VGA 接口上，从而实现了基于 VGA 接口的图像显示。

硬件实现 VGA 控制器的主要思想是，利用像素时钟作为系统基准时钟，根据该基准时钟分别生成行同步信号和场同步信号，同时输出水平坐标和垂直坐标，如图所示：

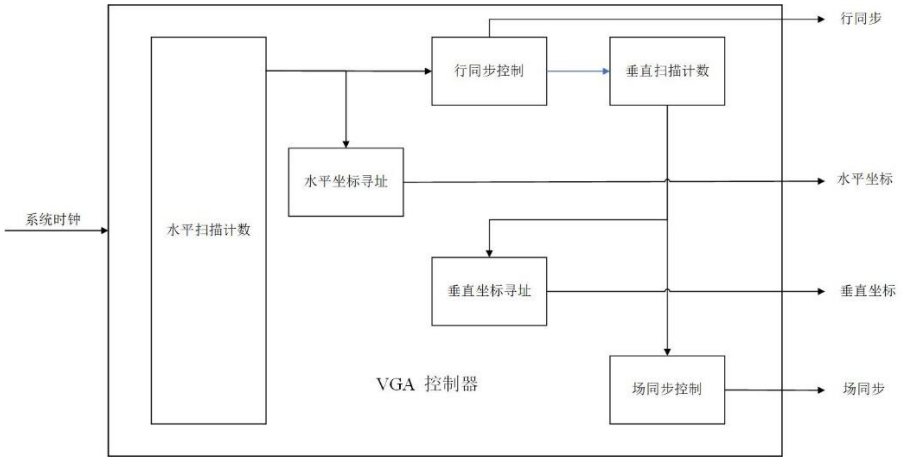

图 13：VGA 控制器示意图

图中的垂直扫描既是场同步信号，负责每帧画面的同步。在场同步的“有效期”（除了同步和消隐期）内，插入 N 行水平扫描（行同步）信号，每路水平扫描信号负责屏幕上 M 个点的显示，这样当扫描完 N 行 M 个点时，一帧画面就显示完成。具体时序参考下图：

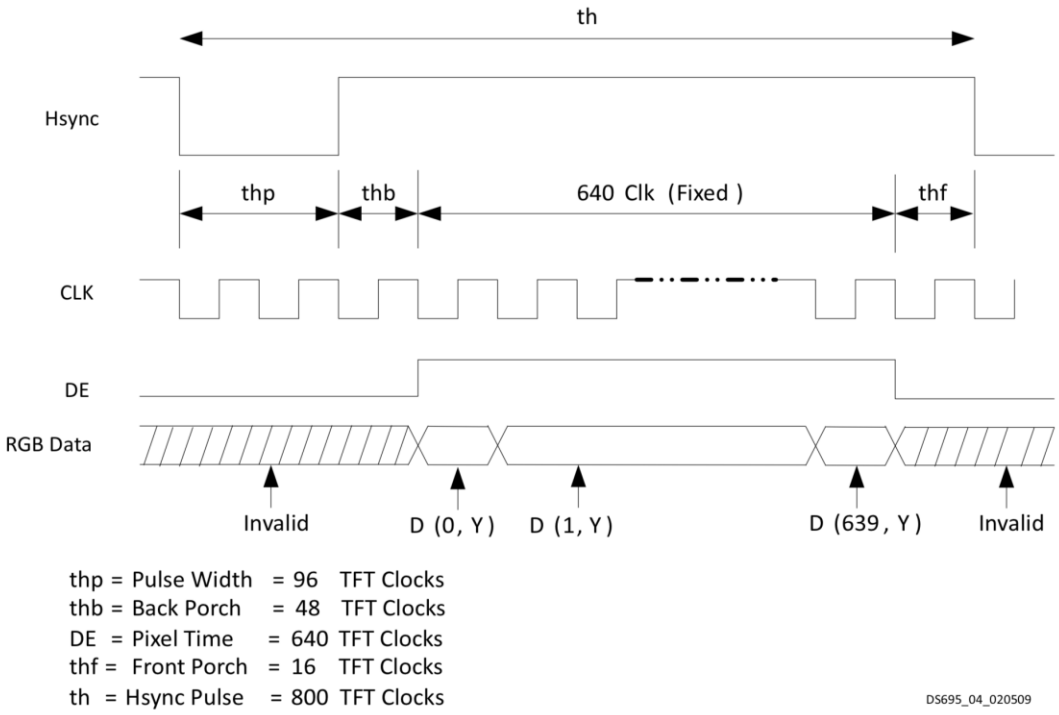

图 14：HSYNC 信号

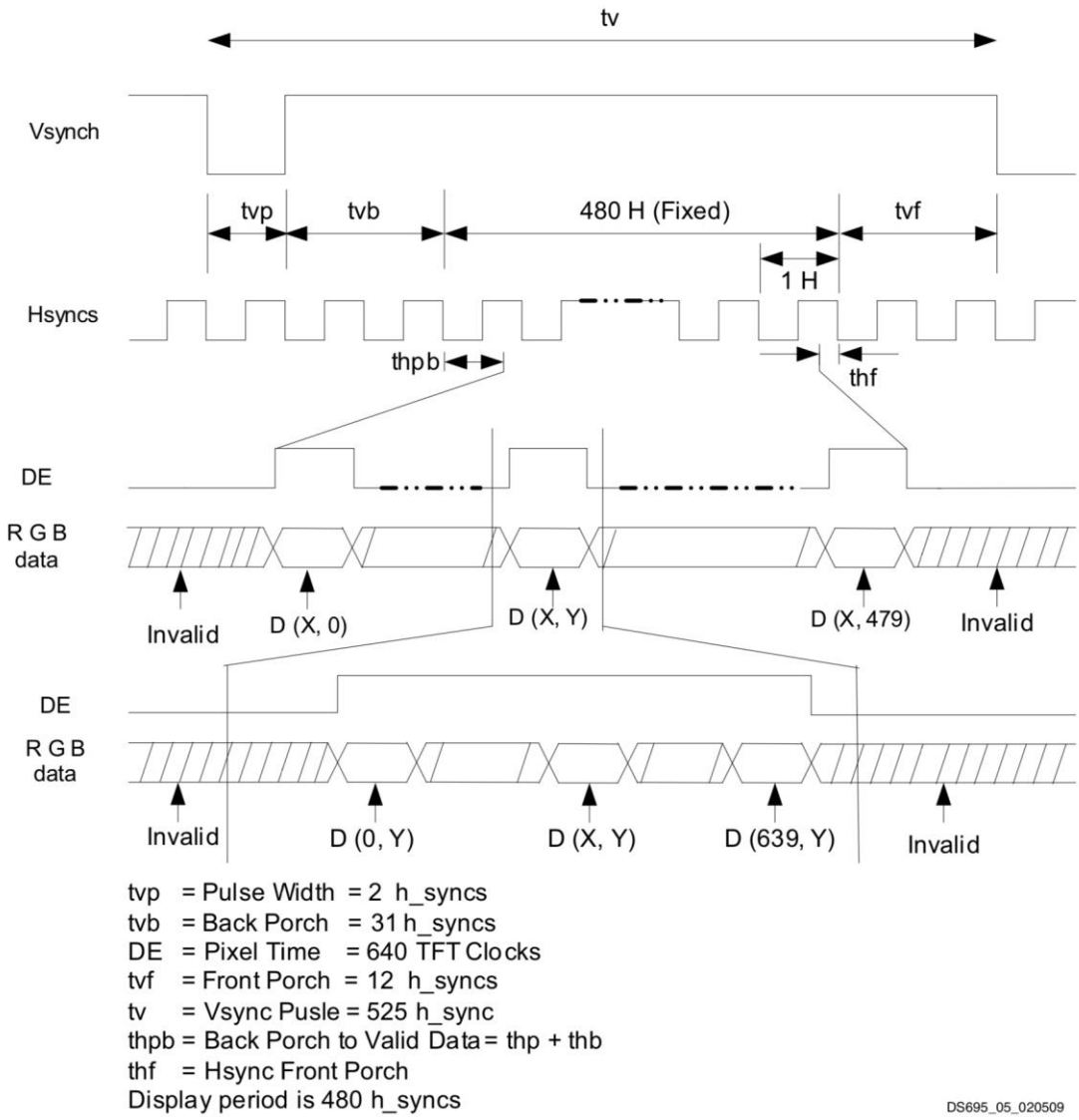

图 15：VSYNC 信号

根据 AXI TFT Controller 的说明文档[9]，该 IP 核提供了两种输出的接口，DVI 接口或 VGA 接口，这里我们仅使用了 VGA 接口。显存方面，VGA 控制器通过一个 AXI 总线接口连接至 DMA 控制器上，从而以 DMA 的方式访问内存的某个区域，该区域的起始地址可以通过控制寄存器来指定。控制器另有一个 AXI 总线接口连接至系统总线上，所有对于 VGA 控制器的寄存器的操作都经由该接口进行。

5. DMA 控制器

DMA(Direct Memory Access)允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载。否则，CPU 需要从来源把每一片段的资料复制到暂存器，然后把它们再次写回到新的地方。在这个时间中，CPU 对于其他的工作来说就无法使用。在本 SoC 中，由于 Uranus 和 VGA 控制器可能需要同时访问内存，所以

DMA 控制器是十分必要的。

DMA 控制器部分使用了和系统总线相同的 AXI Interconnect 这一 IP，但是和系统总线的区别是：DMA 控制器有多个 Slave 端，一个 Master 端，而系统总线仅有一个 Slave 端，多个 Master 端。这就面临一个问题：当多个 Slave 访问同一个 Master 时可能会产生冲突，这就需要总线仲裁。DMA 控制器的仲裁策略使用了 IP 核内建的策略（Round Robbin）。

6. GPIO 控制器

GPIO 控制器由我们自行实现，该模块实现了总线至普通 I/O 接口的转换，提供了读取或操作实验箱上外设的功能，使得程序可以读取拨码开关、4*4 矩阵开关等输入设备的状态，以及控制板载双色 LED 灯、单色 LED 灯、八段数码管输出设备的显示等功能。

除此之外，GPIO 控制器模块还提供了一个定时器，该定时器的是时钟频率恒定为 100MHz，这是为了方便开发系统级应用，或者进行某些性能测试时精确读取时间差而特别设置的。

GPIO 控制器模块的信号列表如表所示（AXI slave 端信号未列出）：

表 12：GPIO 控制器接口

名称	宽度	方向	描述
clk_timer	1	input	定时器时钟，恒定为 100MHz
clk	1	input	CPU 时钟，具体频率取决于 CPU 性能
rst	1	input	复位信号，与 CPU 复位信号一致
switch	7:0	input	板载 8 位拨码开关的状态信号
keypad	15:0	input	板载 4*4 矩阵键盘的状态信号
bicoloc_led_0	1:0	output	板载双色 LED 灯 0 的状态信号
bicolor_led_1	1:0	output	板载双色 LED 灯 1 的状态信号
led	15:0	output	板载 16 位单色 LED 灯的状态信号
num	31:0	output	传递给数码管显示模块的控制信号

GPIO 控制器支持多组输入输出，8 位 switch 输入信号连接至拨码开关，16 位 keypad 信号连接至 Keypad 模块（该模块的作用是检测 4*4 矩阵键盘上按下的键位并将其转换为 16 位的 keypad 信号，由于该模块较为简单所以并未列出），2 位的

bicolor_led_0 信号和 bicolor_led_1 输出信号连接至 2 颗双色 LED 灯，16 位的 led 输出信号连接至实验箱上的 16 颗单色 LED 灯，32 位 num 输出信号经过 SegDisp 模块转换后连接至八段数码管。

GPIO 控制器中将输入引脚和输出引脚的状态存储在内部寄存器里，以便让 CPU 对这些寄存器读取或者写入。其中开关和 4*4 矩阵键盘寄存器是只读的寄存器，双色 LED 灯、单色 LED 灯以及八段数码管的寄存器是可读可写的寄存器，定时器的寄存器是一个只读的寄存器，读寄存器可以获得当前的计数值，该计数器会在每个时钟周期自增 1。

寄存器地址的分配如下表所示：

表 13：GPIO 控制器的控制寄存器地址分配

名称	值	描述
kSwitchAddr	12’h000	开关寄存器地址
kKeypadAddr	12’h004	矩阵键盘寄存器地址
kBicolor0Addr	12’h008	双色 LED0 寄存器地址
kBicolor1Addr	12’h00c	双色 LED1 寄存器地址
kLEDAddr	12’h010	单色 LED 寄存器地址
kNumAddr	12’h014	八段数码管寄存器地址
kTimerAddr	12’h018	定时器寄存器地址

7. 以太网控制器

实验箱上具有一块以太网 Phy 芯片，但是并不具有 MAC，所以需要我们自行实现 MAC，从而支持以太网通信。针对于此，我们使用了 Xilinx 提供的 AXI Ethernet Lite MAC 这一 IP 来实现该模块。

根据 AXI Ethernet Lite MAC 的说明文档[10]，该 IP 核提供了一个 AXI 接口，用来连接至系统总线。CPU 可以通过该接口，向以太网控制器读取或者写入数据。对外接口则为 MII 和 MDIO，其中 MII（Media Independent Interface）接口为同步并行数据接口，它包括一个数据接口，以及一个 MAC 和 PHY 之间的管理接口，用于传输以太网数据和 MAC 与外接的 PHY 互联；MDIO 接口为串行通信总线，是一种简单的双线串行接口，将管理器件与具备管理功能的收发器相连接，从而控制收发器并从收发器收集状态信息。两者均可以直接与 Phy 芯片来连接。

同时该 IP 核提供了一个中断信号，由于该中断信号的类型为边沿类型，而 Uranus 只支持电平类型的中断信号，所以这里需要一个中断控制器来将边沿类型的中断信号转换为电平类型的中断信号，从而发往 Uranus。该控制器我们使用了 Xilinx 提供的 AXI Interrupt Controller 来实现。该模块除了输入端口的以太网控制器的中断信号以及输出端口的 Uranus 的中断信号外，还具有一个 AXI 接口，连接至系统总线，Uranus 可以通过该接口来控制 AXI 中断控制器的内部寄存器。

四、软件及辅助工具

（一）概述

为了更好的体现我们所设计的 SoC 硬件系统的整体性能，进一步挖掘整个系统的综合能力，我们决定针对这样的硬件平台，为其设计和开发更多的软件，乃至移植简单的操作系统。

考虑到参赛经验不足、队员能力有限、时间和精力匮乏等诸多制约因素，我们在决赛阶段没有尝试将现有成熟的系统软件，例如 PMON、U-Boot 或者 Linux 等移植到平台之上——这些程序的体量相对庞大、代码较为繁杂、涉及广度较大，要求 CPU 的指令集实现较为完备、硬件具备 TLB 等功能。目前阶段中，我们的 SoC 实现并不能提供这些硬件条件。如果只基于预赛实现的 CPU 搭建硬件系统，繁复的软件调试必然会成为阻碍我们完成目标的一大因素。

所以，我们决定转变思路，针对现有的硬件平台开发“专用”的系统软件，并且移植简单的嵌入式操作系统，基于底层系统软件再进行进一步的软件开发。这样一来，即可保证整个设计、开发、调试、测试和运行的流程是完全可控的，减少了不必要的试错，并且节约了宝贵的时间和精力。

（二）UBW 引导程序

整个 Uranus SoC 硬件系统在上电初始化完毕之后，系统中的 CPU 核心会直接从 0xBFC00000 地址处取指执行。而对应的硬件实现中，虚拟地址 0xBFC00000 会被转化为物理地址 0x1FC00000，这个物理地址又会直接映射到板载 SPI Flash 的起始存储空间处。我们完全可以通过向 SPI Flash 中烧录我们需要执行的程序的方法，

例如写入测试程序、操作系统等，来实现系统正确性的检验，或者直接启动操作系统。

但是这样做有很多弊端：为 SPI Flash 编程需要插拔 Flash 芯片，亦或是向 FPGA 烧写专用的 Flash 编程程序。如果要调试系统软件，有时候我们只修改了寥寥几行代码，然后就需要执行一次类似这样繁复的烧写、编程的流程。长此以往，SPI Flash 的寿命必然会受到影响，Flash 芯片的芯片座可能损坏，而且最主要的是，我们调试过程中大部分的时间也会浪费在这些机械重复的流程之中。所以，为了实现程序的快速调试，以及日后引导操作系统启动，实现一个具备基本功能的引导程序是十分必要的。

1. 功能说明

UBW 全称为 Unlimited Boot Works，是我们针对 Uranus SoC 设计的一款引导程序。该程序被固化在了 SPI Flash 起始的 16KB 空间当中，以确保系统初始化完毕之后可立即执行该程序。

UBW 会通过拨码开关和串口相应外部的操作。在所有拨码开关均拨下的时候，UBW 会初始化进入串口交互模式。此时我们在 PC 端使用串口软件（例如 Secure CRT 等），设定波特率为 230400，再使用串口连接线连接板子和 PC，即可接入到 UBW 的终端界面中。如下所示：

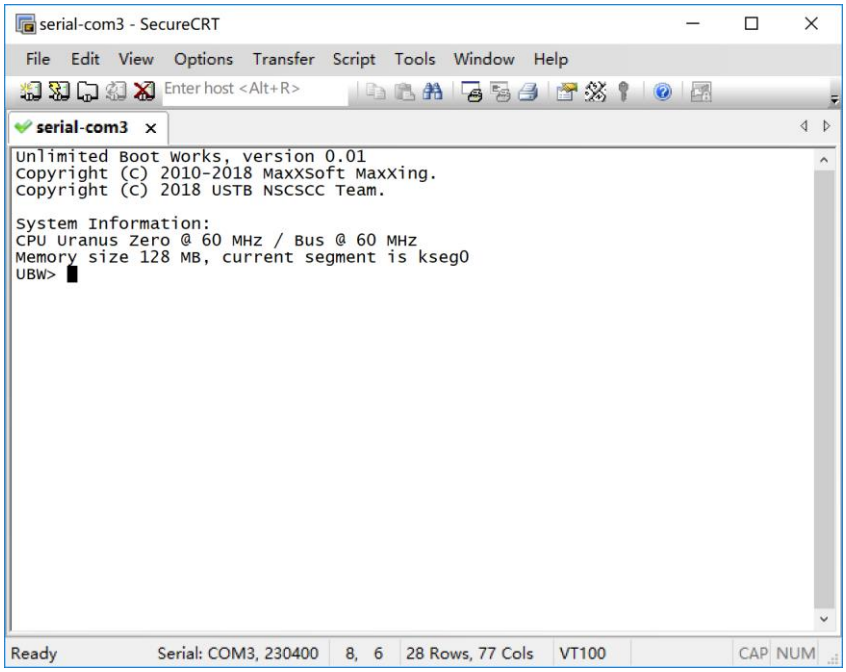

图 16：UBW 的交互式界面

在此模式下，我们可以使用命令交互的方式来操作 UBW。UBW 可以支持如

下命令：

- about：显示“关于”信息；
- echo <string>：回显字符串的内容；
- clear：清空屏幕显示；
- load <mode> <address> [length]：通过串口加载外部程序或操作系统到内存。当 <mode> 被指定为 0 时，UBW 将简单地获取串口传输过来的字节数据，并且将其写入内存地址为 <address> 处，长度为 [length] 的内存空间中。当 <mode> 被指定为 1 时，UBW 会使用 Xmodem 传输协议从串口获取上位机发来的二进制程序数据，此时无需输入 [length] 参数；
- write <address> <length> <offset>：将内存地址 <address>，长度 <length> 的数据写入到 SPI Flash 偏移量为 <offset> 的位置。此处的偏移量并非相对 Flash 起始地址的偏移量。由于 UBW 本身位于 Flash 的起始位置，占用略多于 16KB 的存储空间，在执行“write”命令时指定的偏移量会跳过 UBW 所占用的区域，从 Flash 存储空间的第 18KB 算起；
- init <address>：从内存的指定地址执行程序，或者初始化操作系统；
- override <address> <length>：将内存地址 <address> 处，长度为 <length> 的数据覆盖到 UBW 的存储区域当中。此命令通常用于更新 UBW 本身；
- peep <address> [length]：按字（word）查看内存地址 <address> 处，长度为 <length> 的存储单元的值，参数 [length] 缺省为 4。

通过以上命令可以看出，UBW 已经实现了一个引导程序的基本功能，甚至在此基础之上还做出了进一步的探索（例如实现了类似终端的操作方式，极大地提高了用户界面的友好程度）。

除了使用串口终端进行操作，UBW 还支持直接从 SPI Flash 或者串口来引导操作系统。在 SoC 复位之前，只拨上最右侧的拨码开关，UBW 就会不进入串口交互界面，直接从 SPI Flash 的 18KB 偏移处启动系统。如果只拨上从右往左数第二个拨码开关，UBW 会直接进入串口传输模式。这时我们可以使用上位机程序来向 UBW 传输程序，传输的程序将被存储在内存的 0x00000000 地址处。当传输完毕时，UBW 会自动跳转到虚拟地址 0x80000000 处执行程序的加载。

2. 上位机程序

UBW 的上位机程序（serial_util.py）使用 Python 编写，位于 UBW 的 util 目录中。

该程序可以直接向处于串口传输模式的 UBW 发送二进制程序，从而实现程序或者操作系统的串口引导启动。

我们可以使用如下方式执行上位机程序：

```
$ python3 serial_util.py [<serial_port> <binary_file>]
```

直接运行该程序时，程序会输出目前系统中所有可用串口的列表。我们从中选取连接到 SoC 的串口，复制其名称（如 COM3）并作为命令行参数传入，在之后接着传入要传输的文件名，例如：

```
$ python3 serial_util.py COM3 test.bin
```

确保 UBW 处于串口传输模式，然后按下回车。稍等片刻之后，指定的二进制程序就会被传送到 SoC 的内存当中，并且立即被加载执行。

3. UBW 的编程实现

UBW 的实现总体上可以分为四部分。

第一部分是使用汇编实现的程序入口（init.s）。此部分的程序是最先被执行的程序，所以这一部分必须完成 UBW 的简单初始化，然后快速跳转到内核主函数中完成进一步的内存、静态变量区等的初始化工作。

这部分程序首先开启了 CPU 的硬件中断，方便 UBW 接收 CPU 内建计时器，或者串口等外设的中断请求；接着初始化了栈指针寄存器（$sp），确保接下来在 C 程序内的诸多函数调用不会发生错误；再之后，程序读取了链接脚本文件（linker.ld）中定义的常量，具体是全局指针的位置，和 bss 段的起止地址，这三个常量将作为参数传递到内核主函数中；最后，程序载入了内核主函数的地址，并且完成了从 kseg1 段到 kseg0 段的地址转换，正式跳转到了 C 语言构建的内核程序中。

这段汇编程序还接管了中断和异常的处理。当遇到中断或者异常时，程序将跳转到内核中的 ExceptionHandler 函数，实现异常信息的显示等高级功能，方便 UBW 的调试。

第二部分是使用 C 语言实现的简单的标准库。由于 UBW 并非运行在一个具备操作系统的环境之下，并没有什么其他程序能为其提供诸如标准输入输出、内存分配等的库函数，所以这些函数都需要自行实现。这部分利用 SoC 的各类外设（GPIO、UART、VGA、SPI Flash 等），结合 C 语言的基本功能，实现了涵盖下列功能的“标准库”函数：

- 调试相关的宏定义：DEBUG、DISABLE_DEBUG、

- 各类数据类型上下限的宏定义：limit.h
- 内存分配和内存操作函数：malloc、free、memset、memcpy、memcmp；
- 随机数生成器：srand、rand；
- SoC 所有外设的地址定义：soc.h
- SPI Flash 控制器的操作：擦除、写入和读取指定存储区域；
- 部分标准输入输出：putchar、getchar、puts、gets、printf；
- 字符串操作函数：strcpy、strlen、isalnum、toupper、atoi、strtol 等；
- 获取周期数和延时的函数：GetTick、DelayMicrosecond、DelayMillisecond；
- 类型和常量定义：uint32_t、size_t、NULL 等；
- 串口收发函数：InitUART、GetByteUART、PutByteUART 等；
- VGA 显示控制函数：SetVideoMemAddr、GetVideoMem、SetVGAStatus。

需要注意的是，由于 SoC 硬件实现中的 SPI Flash 控制器使用了龙芯 SoC_up 环境中的对应模块，于是关于 SPIFlash 控制器的相关操作部分的 C 语言实现同样参考了龙芯移植的 PMON 源码当中的相关部分。标准库的其余部分均为自行实现。

第三部分是使用 C 语言实现的内核（kernel）。内核需要完成的任务基本只有两个：一个是在系统上电之后接管内存的初始化工作，另一个是为外壳（shell）程序提供核心功能的调用接口。

内存的初始化是至关重要的，因为 UBW 存储在 Flash 之上而不是内存，而程序中所有使用到的全局变量都存储在 bss 段，程序在访问这些变量之前，必须保证这部分区域的初始值为 0，且可以被写入。所以内核在 UBW 刚刚启动时，需要将内存中的一块区域作为全局变量区，并且将所有 data 段和 bss 段的数据复制到该区域，完成对 bss 段的清零，最后修改全局指针寄存器（$gp）的值使得程序可以正确地访问到这些变量。这样才算完成了内存部分的初始化。

在此之后，内核会相继初始化 UART 控制器，然后根据 GPIO 拨码开关的状态判断，接下来需要加载外壳程序，还是需要加载操作系统，或者需要进入串口传输模式。并且根据这些模式来设置 LED 指示灯的状态，方便用户据此判断 UBW 目前的运行情况。

第四部分是使用 C 语言实现的外壳（shell），这部分程序只会在 UBW 的默认模式下被加载。外壳提供了基于串口的命令行交互界面，其主要部分是一个 REPL（Read Eval Print Loop）。在此部分中，程序会调用先前实现的 gets 函数从串口获

取用户的输入，然后判断用户输入是否可以匹配到现有的命令。若无法完成匹配则输出错误提示，否则，程序会获取用户输入的所有参数，并且将其分割成若干独立的字符串。接下来，外壳会检查命令对应的处理函数（handler），之后将用户输入作为参数来调用这些函数，完成对输入命令的相应。外壳中的处理函数只负责基于外壳的交互，例如输出提示信息或者结果之类，实际的运行逻辑依然由内核负责。

4. 编译 UBW

提交包中附带的 UBW 已经包含了 Makefile，再确认系统中已经安装了交叉编译工具（mipsel-linux-gcc 工具链）的情况下，进入 UBW 根目录直接执行“make ubw”即可完成 UBW 的编译。

除此之外，UBW 的 test 目录中还包含一些使用汇编编写的测试程序，这些程序是在 Uranus SoC 的设计初期用来测试 SoC 的硬件设计是否正常工作的。测试程序包含：

- gpio.s：检查板子上的 GPIO 外设（数码管、LED、拨码开关、矩阵键盘、定时器等）是否可以正常工作，具体操作逻辑请参考汇编程序的内容；
- mem.s：检查 DDR3 内存是否能被正常访问。程序会随机生成 8 个无符号 32 位整数，写入到内存的某个连续区域内。接着程序会尝试从这些区域中读取刚刚写入的数据，并且检查写入的值和读取出来的值是否相同，结果以双色 LED 灯的形式显示；
- vga.s：检查 VGA 控制器是否可以正常地产生图像。程序会向默认的显存基地址（0x05000000）处写入有规则的数据，连接 VGA 显示输出，我们可以从显示设备中观察到方格状的有规律的图像；
- uart.s：检查 UART 控制器是否能够正常的收发数据。程序会初始化 UART 控制器，并且设置其波特率为 230400。我们可以通过串口调试软件连接 SoC，串口会回传我们发送的任何数据。

测试程序可以通过“make test”命令来编译。如欲编译包含 UBW 和测试程序在内的全部程序，可以使用命令“make”或者“make all”；如欲清理编译产生的文件，请执行“make clean”。

（三）嵌入式实时操作系统 μC/OS-II 的移植

前文所述的 UBW 只是一个烧写在 SPI Flash 中的引导程序，其作用是加载其他程序，或者引导操作系统。而为了让整个 SoC 的性能得到更加充分的利用，我们决定为其移植一些简单的操作系统或者系统程序。考虑到嵌入式操作系统的功能较少，体量较小，便于移植和调试，而 μC/OS-II 这一嵌入式实时操作系统资料较多，且应用广泛，我们决定将该系统移植到 Uranus SoC 平台之上。

1. 概述

移植基于 Micrium μC/OS-II，版本 v2.91。由于 Micrium 官方已经提供了一个针对 MIPS M14K 处理器的移植代码，我们可以直接利用该代码对系统行为进行进一步适配，规避 Uranus CPU 暂时不支持的 MIPS 指令，并且尽可能提高系统性能和效率。

μC/OS 的代码较少，我们可以将整个项目组织为四个目录：

- src 目录：用于存放 μC/OS-II 的 C 语言源程序，这部分与平台无关，无需修改；
- port 目录：用于存放移植相关的代码，包含 C 语言源程序和汇编源程序。这部分实现了系统底层调用中平台相关的部分，例如开关中断、处理定时器请求等，所以需要针对 SoC 本身的特点进行修改；
- include 目录：用于存放所有的 C 语言头文件，包含平台相关部分的头文件。平台相关部分主要定义了诸如变量类型、任务切换方法等内容；
- common 目录：用于存放操作系统的入口程序。μC/OS-II 本身只是一个操作系统核心，其支持实时任务调度，但是具体需要执行什么任务，以及何时初始化操作系统，依然需要由外部程序来制定。这部分程序将会负责系统的初始化工作，并且同时执行两个系统任务，以便我们测试系统移植是否正确。

2. 平台相关代码的修改

需要修改的平台相关的代码主要集中在 port 目录中的多个文件当中，其中所有的汇编源程序应当做出大量的修改，以便于适配 CPU 的指令集。

在 cpu_a.s 中，程序使用了 DI 指令来关闭中断，在 Uranus 中并没有这条指令，所以我们将该指令替换为对 CP0.Status 寄存器的等价操作。

在 os_cpu_a.S 文件中，程序缺少初始化代码和中断、异常处理代码，所以我们在程序的最前端添加相关的代码。由于 0xBFC00380 的中断/异常向量入口已经被 UBW 引导程序使用，而操作系统又必须正确处理定时器中断的请求，所以在程序的初始化代码中，需要将 CP0.Status 寄存器的 Bev 位设为 0，这样处理器就会根据 CP0.EBase 寄存器的值来确定中断/异常向量的入口了。EBase 默认为 0x80000000，于是处理程序的入口默认即为 0x80000180，μC/OS 会被 UBW 载入到 0x80000000 地址处，所以我们设置中断和异常处理程序的偏移量为 0x180 即可。

在 os_cpu_a.S 文件中，依然存在许多 EI、DI 指令，我们同样将这些指令替换为对 CP0.Status 寄存器的直接操作。在 TickISR 函数中，默认针对 CP0.Compare 寄存器的处理方法是在其原有值的基础上做加法，加上新的延时值，实现定时器的延时，继而实现任务切换。但是这样做很容易导致系统初始化时 Count 寄存器的值已经较大，而 Compare 的值可能小于 Count 寄存器，在初始化阶段的任务切换可能会等待很长时间，当 Count 寄存器的计数值溢出回绕之后才可触发任务切换。于是我们修改了此处的定时器延时处理方式，将 Count 寄存器清零，接着将 Compare 寄存器的值直接设为延时量，这样可以保证下一个定时器中断发生的时机和延时的预期时间相符。其余部分修改幅度较小，且与前文所述类似，此处不再赘述。

在 os_cpu_c.c 文件中，我们还需要去掉一些无用的外部变量，例如 vec[] 和 endvec[] 等，以及有一些内联的汇编代码也是不需要的，这些变量和代码主要与中断和异常处理有关，我们已经在 os_cpu_a.S 文件中正确处理了这些内容，所以可以直接去掉。

在此文件中，我们还需要修改 OSTaskStkInit 函数的内容。在此函数中，sr_val 变量存储了 CP0.Status 寄存器的内容，为了在初始化任务堆栈时允许定时器中断，以便于任务切换等操作，我们必须修改 sr_val 的对应位使得定时器中断和全局中断均被使能。除此之外，此文件中应当加入两个新函数：BSP_Exception_Handler 和 BSP_Interrupt_Handler，分别负责实际的异常和中断处理。在遇到异常时，简单地切换任务即可；遇到中断时则需要判断是否是定时器中断，如果是的话则需要调用 TickISR 函数进行处理。

以上就是平台相关代码修改方面的具体内容。

3. 编译并启动系统

μC/OS-II 的编译过程较为简单。在 uCOS-UZ 目录中我们已经编写好了编译所需的链接器脚本（Linker Script）以及 Makefile，其中链接器脚本中我们指定了 os_cpu_a.S 文件中定义的若干段（section）在内存当中的存放位置，并且定义了诸如栈指针的初始值等常量，方便系统初始化程序的读取。

Makefile 的内容基本也是常规思路：分散在三个目录内的 Makefile 文件（include 目录并不需要 make）分别负责将目录内的 C 语言或者汇编源文件编译为一个目标文件（*.o），然后位于根目录的 Makefile 再负责将这三个目标文件链接成最终的 elf 文件，接着调用 objcopy 将 elf 文件中二进制的代码和数据内容拷贝到单独的 *.bin 文件中。

在根目录下，我们可以执行“make all”命令。在此之前需要确保系统已经配置了交叉编译环境（mipsel-linux-gcc 工具链）。之后在 build 目录中，我们即可看到 ucos.bin 这个二进制文件，这就是我们需要上传到 SoC 上的最终文件。

启动 Uranus SoC，在系统初始化之前，将拨码开关的最左和右数第二个开关拨上，其余均拨下，此时 UBW 会进入波特率为 115200 的串口传输模式。接着我们即可使用上位机软件将 ucos.bin 传输到 SoC 上执行了。

（四）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
```

五、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；
- perf：性能包，包含以下内容：
  - 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。
- showing：展示包，包含以下内容：
  - rtl：目录，Uranus SoC，也就是大赛展示使用的 SoC 的 Verilog HDL 源代码；
  - run_vivado：目录，Uranus SoC 的 Vivado 工程，以及生成的 bit 文件；
  - soft：目录，大赛展示使用的所有软件，包括 UBW 引导程序以及移植完毕的 μC/OS-II 嵌入式实时操作系统。

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；
- soc_connect.pdf：文件，由 Uranus SoC 设计时使用的 Block Design 导出的内部模块、IP 核的连接方式；

- 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，仅支持大赛预赛要求实现的 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（不含第三方 IP 的情况下）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。仅支持大赛预赛要求实现的 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 17：功能测试的 Tcl Console 输出

(2) 仿真波形输出：

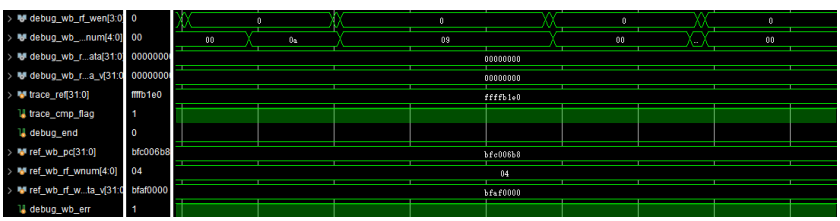

图 18：功能测试的仿真波形输出

(3) 上板验证结果：

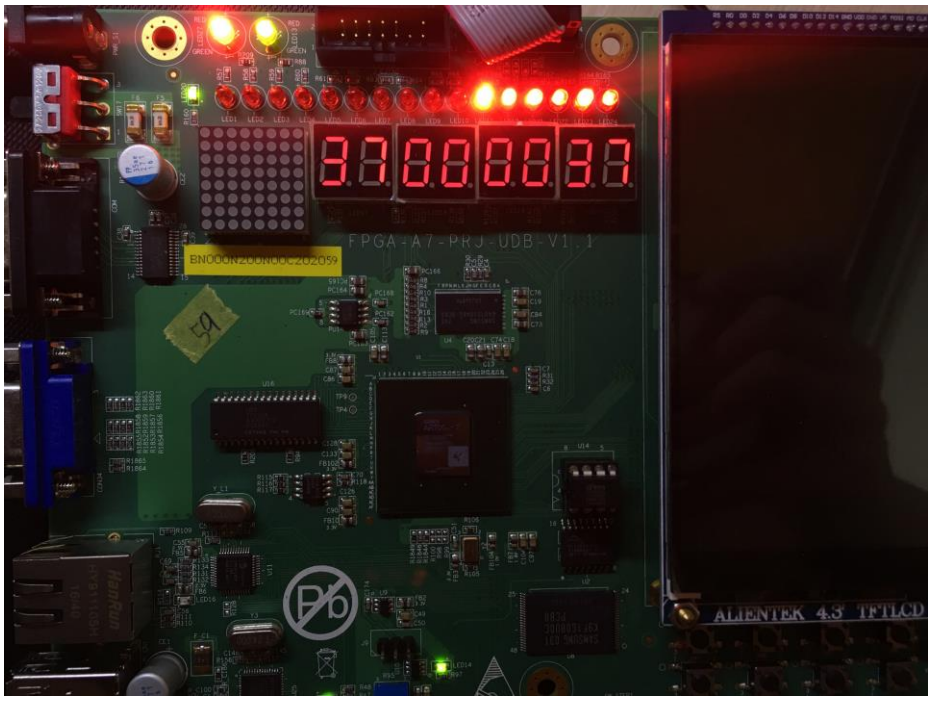

图 19：功能测试上板运行过程

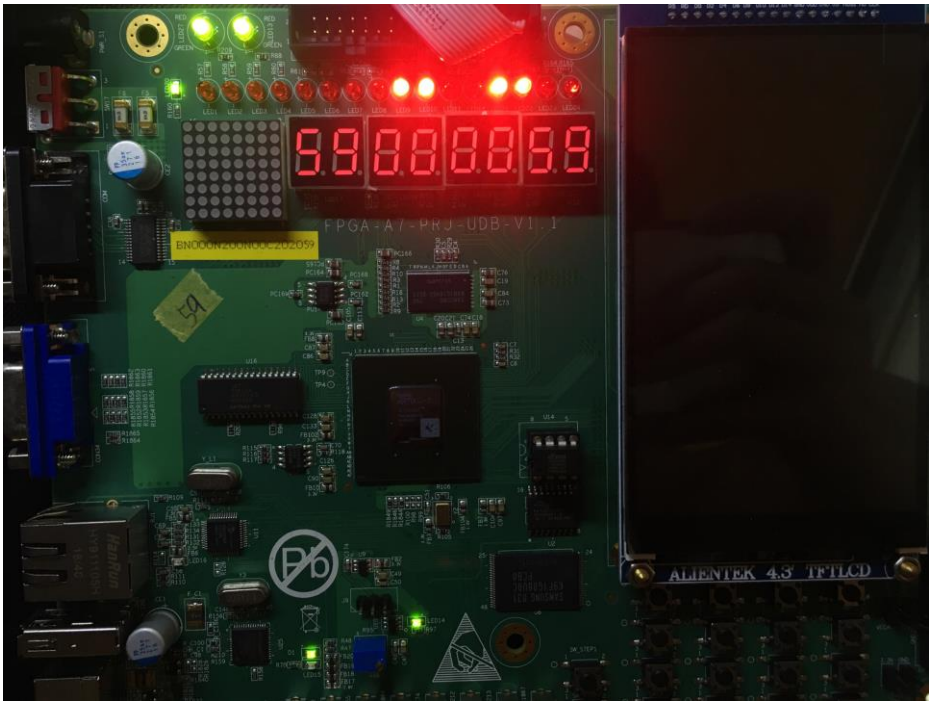

图 20：功能测试上板运行完成

2. 运行记忆游戏

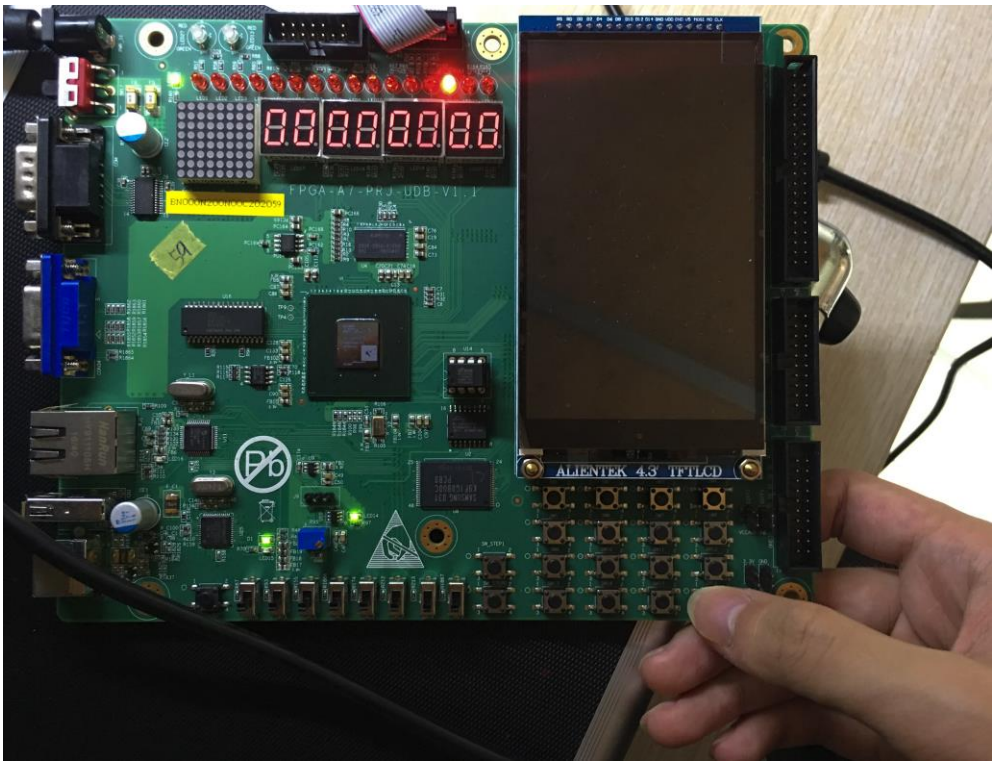

图 21：记忆游戏上板运行

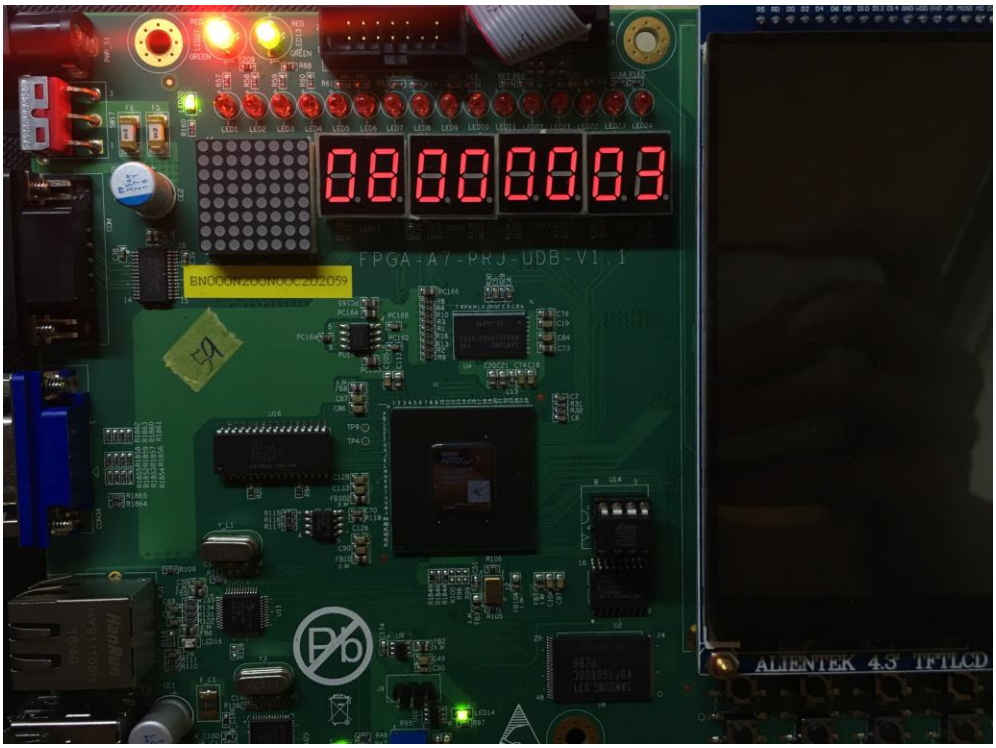

图 22：记忆游戏输入错误

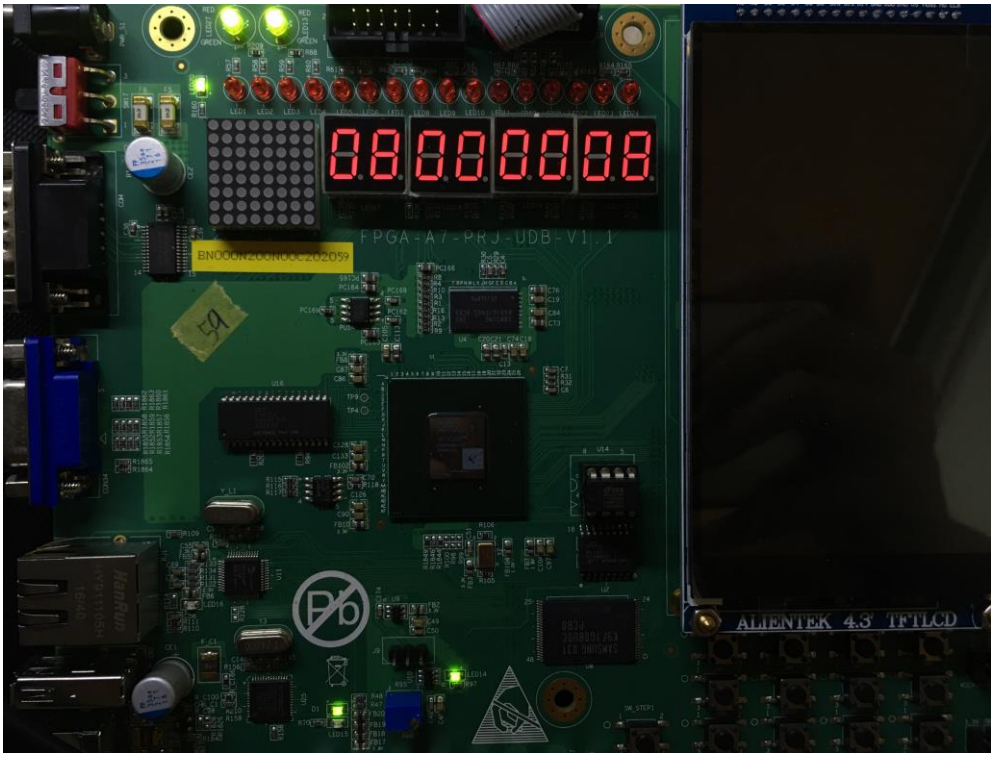

图 23：记忆游戏输入正确

3. 运行性能测试

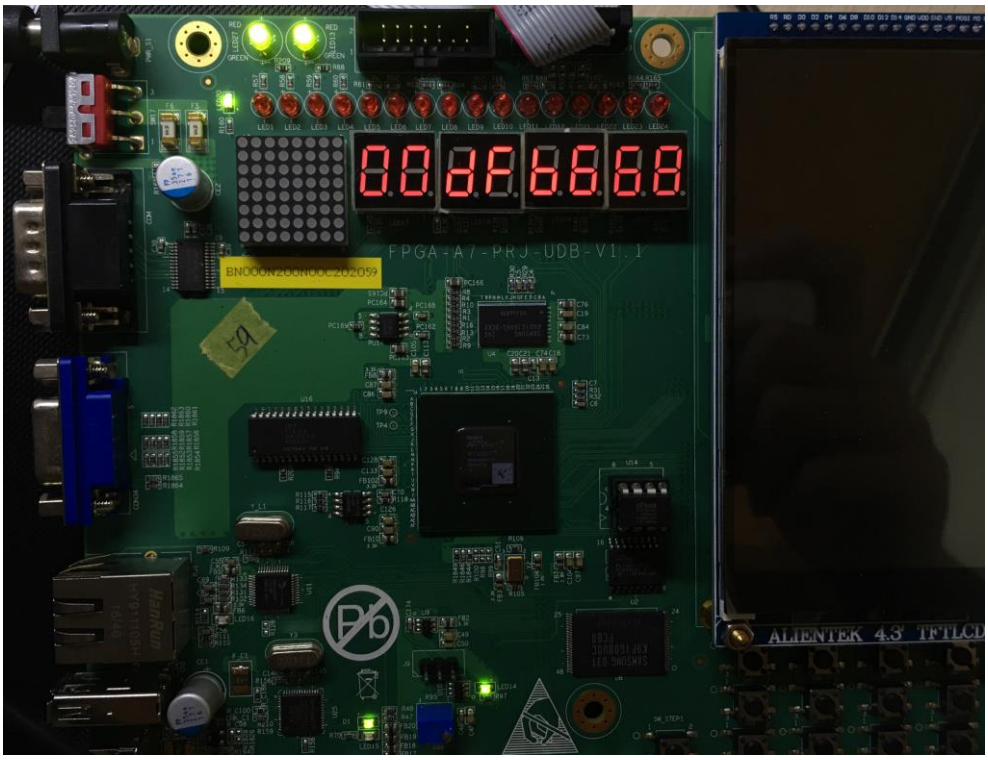

图 24：性能测试运行第一个功能点

六、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[11]。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。

SoC 的大部分外设控制器均使用了 Xilinx 提供的 IP 核，SPI Flash 控制器则使用了龙芯开源的 SoC_up 环境中的 spi_flash_ctrl 模块。相应的，在 UBW 引导程序的代码中，为了基于此控制器实现 Flash 的擦除和编程，我们参考了 SoC_up 平台上 PMON 源代码中的相关部分（pmon_archlab/Targets/LS1B/dev/spi_w.c）。

μC/OS-II 移植部分的大部分内容参考了《自己动手写 CPU》一书中的步骤，但是具体的平台相关的代码均系自行实现，因为 Uranus SoC 和书中所述的 OpenMIPS 所支持的指令集并不完全相同。

七、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] https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_1/ug586_7Series_MIS.pdf

[8] https://www.xilinx.com/support/documentation/ip_documentation/axi_uart16550/v2_0/pg143-axi-uart16550.pdf

[9] https://www.xilinx.com/support/documentation/ip_documentation/axi_tft/v2_0/pg095-axi-tft.pdf

[10] https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernetlite/v3_0/pg135-axi-ethernetlite.pdf

[11] 雷思磊, 自己动手写 CPU, 2014, 电子工业出版社.