

Tsinghua University
Department of Computer Science and Technology

IntoMips 项目报告

瞿凡 孙维孝 钟闰鑫
2017010636 2017010650 2017010306

Final version

January 2020

目录

1 概览	1
1.1 项目背景	1
1.2 项目摘要	1
1.2.1 CPU	1
1.2.2 外设	1
1.2.3 操作系统移植	1
1.2.4 自动化测试	2
1.2.5 文件结构	2
1.3 开发平台	2
1.3.1 硬件平台	2
1.3.2 软件平台	2
1.4 开发管理	3
1.4.1 分工	3
1.4.2 开发流程	3
1.5 参考资料	3
2 CPU	4
2.1 流水线结构	4
2.1.1 基本五级流水线	4
2.1.2 流水线冒险	4
2.2 CP0	5
2.3 指令集	5
2.4 中断异常	6
2.5 内存管理	7
2.6 代码结构	7
3 外围设备	9
3.1 地址分配	9

3.2	总线	9
3.3	存储设备	9
3.3.1	Sram 控制器	9
3.3.2	Flash 控制器	10
3.3.3	Bootrom 控制器	10
3.4	Uart 控制器	11
3.5	VGA 控制器	12
3.6	USB 控制器	12
3.7	Display 控制器	13
4	操作系统	14
4.1	系统介绍	14
4.2	操作系统移植	14
4.2.1	编译方法	14
4.2.2	移植过程	14
4.3	新增内容	14
4.3.1	修改对退格的支持	14
4.3.2	VGA 显示	15
4.3.3	贪吃蛇程序	15
5	测试与集成	17
5.1	CPU 测试	17
5.2	外设测试	17
5.3	系统测试	18

1. 概览

1.1 项目背景

本项目是计算机组成原理、软件工程两门课程的联合实验。项目需求方为计算机组成原理课程，需求方代表为刘卫东老师；项目承担方是“逐梦软工圈”小组，组员为计 72 瞿凡，计 71 孙维孝，计 71 钟闰鑫。

本项目的成果为在 FPGA 上编程实现一个 32 位 MIPS 指令集的流水线 CPU，支持必要的异常、中断以及采用 TLB 的内存管理器等。在 CPU 完成后，该 CPU 上能够运行 uCore 操作系统，进入用户态，启动控制终端，正常执行测试程序。

本文档描述了片上系统的实现细节，以及项目的开发、测试与集成流程。

1.2 项目摘要

1.2.1 CPU

CPU 的设计包含流水线结构、指令集、协处理器、中断异常、内存管理。

流水线结构: 实现了经典五级流水线，分为取指、译码、执行、访存、写回五级。需要解决冲突问题。

指令集: 实现了能够运行 ucore 的指令集，是 MIPS 指令集的一个子集。

协处理器: 实现了 MIPS 标准中的 CP0 处理器的大部分寄存器和指令，足以运行 ucore 系统。

中断异常: 实现了标准中部分异常和中断，支持硬件软件中断，实现了精确异常。

内存管理: 实现了 MMU 和 TLB，内存划分遵循 MIPS 标准。

1.2.2 外设

外设部分包括存储系列控制器，Uart 控制器，VGA 控制器，USB 控制器和 Display 控制器

存储系列控制器: 实现包括 Sram、Flash 和 Bootrom 的控制

Uart 控制器: 直连串口，处理串口通信

VGA 控制器: 实现对于 VGA 的控制，显示能带颜色，能闪烁的字符，内置显存

USB 控制器: 实现对于 USB 内部寄存器的读写

Display 控制器: 控制拨码开关，按钮开关，LED 和数码管

1.2.3 操作系统移植

我们在实现的计算机上移植了 ucore-thumips 操作系统，该操作系统使用 mips32S 指令集。

我们能够正常运行该操作系统并能够运行用户程序，我们为检验与展示编写了一些展示程序。另外，我们为外设如 vga 增加了部分驱动。

1.2.4 自动化测试

实现了基于 CI 的自动化综合、生成测例、测试的系统。

1.2.5 文件结构

我们在项目早期即定下了文件组织结构，共有如下 5 个文件夹

- src 文件夹，包含 Verilog 源码
- ucore 文件夹，包含 ucore-thumips 操作系统源码
- testbench 文件夹，包含测例与自动化测试代码
- vivado 文件夹，包含 vivado 工程文件
- doc 文件夹，包含文档

1.3 开发平台

1.3.1 硬件平台

1. 开发板:THINKPAD-Cloud
2. FPGA: Xilinx XC7A100T
3. SRAM: IS61WV102416 32 位 8M
4. Flash: 16bit 8M NOR
5. USB: Cypress SL811HS 支持 USB 1.1

1.3.2 软件平台

1. EDA 软件: Xilinx Vivado HL WebPACk 2018.3
2. 综合电路时 OS: windows 10 或 ubuntu 16.04
3. 编译操作系统时 OS: ubuntu 16.04
4. 支持运行 OS: ucore-thumips
5. 操作系统模拟器: qemu

1.4 开发管理

1.4.1 分工

孙维孝: CPU 流水线, CP0, 中断异常

钟闰鑫: Sram, Flash, Bootrom, Uart, VGA, USB

瞿凡: MMU, TLB, 操作系统移植, 用户程序

1.4.2 开发流程

第 6 周: CPU 与 SRAM、CPLD 串口、FLASH 分别基本实现

第 8 周: CPU 与 SRAM、CPLD 串口对接, 可以联合工作, 基本完成 TLB 部分, VGA 可展示图片

第 11 周: 完成了带中断的 CPU, 加入了 bootrom, 编译了操作系统, 采用直连串口

第 13 周: 整合了 TLB, 能够运行 ucore, VGA 改为基于字符, 编写了用户程序

第 15 周: 加入了 USB 键盘, 支持光标闪烁, 贪吃蛇支持多进程

第 16 周: 龙芯杯测试

1.5 参考资料

- See *MIPS Run Linux*. Dominic Sweetman
- 自己动手写 *CPU*. 雷思磊
- *TrivialMIPS* 项目设计文档 & 实验报告. 陈晨祺, 周聿浩, 姚沛然
- *32-bits MIPS CPU* 设计文档. 谢磊, 李北辰

2. CPU

2.1 流水线结构

2.1.1 基本五级流水线

执行一条指令所需的时间很长，性能浪费严重。可以采用流水线技术将一条指令的执行拆分成几个小步骤。通常将指令的执行划分为五级流水线：

1. 从指令存储器中读取指令 (IF)
2. 指令译码并读取寄存器 (ID)
3. 指令执行 (EX)
4. 访问内存 (MEM)
5. 将结果写回寄存器 (WB)

这样的流水线设计，使得处理器在同一时刻最多能够处理 5 条 MIPS 指令，在不缩短单条指令的处理时间的情况下，提高了指令的吞吐率（即单位时间内完成指令数目）。在理想情况下，流水线上的指令执行时间变成非流水线指令执行时间的 $\frac{1}{5}$ 。

2.1.2 流水线冒险

由于流水线中不同指令执行会互相重叠，因此若两条指令之间存在依赖关系，就必须单独考虑，这即是“流水线冒险”。流水线冒险分为结构冒险，数据冒险和控制冒险。

结构冒险

由于硬件不支持多条指令在同一时钟周期执行导致的冒险。例如在单一寄存器多条指令同时访问该寄存器则会发生结构冒险。结构冒险可以直接采用增加硬件资源解决。

数据冒险

由于一条指令必须在另一条指令完成后才能开始而造成的冒险。例如一条指令需要上一条指令写回寄存器文件的结果，即下一条指令必须等到上条指令的 WB 阶段结束后才能开始自己的 ID 阶段，这就浪费了时钟周期。

数据冒险解决一般采用如下两种方式

- 数据旁路

即是从内部寄存器中提前取出数据交给下一步指令使用

- 插入气泡

即在两条指令之间插入空指令 (nop) 来消除相邻指令间的依赖

控制冒险

又称分支冒险，由于取到的指令不一定是所需要的指令（分支跳转），从而指令无法在预定时间周期内执行导致的冒险。常在分支跳转处出现，在一条跳转指令执行时，其下一条指令已经进入流水线，但跳转指令的结果可能不是代码中的下一条指令，因此下一条指令进入流水线的结果可能需要被清除。MIPS 采用延迟槽解决控制冒险，即设定分支之后的指令一定会被执行，可以将一条无关指令放入延迟槽中。

2.2 CP0

协处理器通过拓展指令集或提供配置寄存器的方式来拓展内核处理功能。CP0 是 MIPS 标准中必要的协处理器，提供了异常处理、内存管理等功能。

CP0 中有多个 32 位寄存器，可以使用 MTC0 和 MFC0 进行存取，下表是运行 ucore 必要的寄存器。

寄存器号	寄存器名	功能
0	Index	TLB 阵列的入口索引
1	Random	随机数，最大值为 TLB 索引数
2	EntryLo0	偶数虚拟页入口地址的低 32 位部分
3	EntryLo1	奇数虚拟页入口地址的低 32 位部分
4	Context	指向内存中页表入口的指针
5	Pagemask	表示 TLB 虚拟页大小
6	Wired	Random 寄存器的最小值，表示不被 tlbwr 指令写入的项数
8	BadVAddr	记录最近一次存储发生异常时的虚拟地址
9	Count	与 Compare 寄存器组成片内计时器，两者相等所示发出时钟中断信号
10	EntryHi	TLB 入口地址的高 32 位部分
11	Compare	与 Count 寄存器组成片内计时器，两者相等所示发出时钟中断信号
12	Status	处理器状态和控制寄存器，决定 CPU 特权等级和中断使能等
13	Cause	保存最近一次异常原因
14	EPC	保存最近一次异常的程序计数器
15	Ebase	保存异常处理程序的入口地址
16	Config	CPU 配置

Table 2.1: CP0 寄存器功能表

其中 Ebase 实际上是 MIPS32 Rev 2 中的寄存器，在 MTC0 和 MFC0 中要求 sel 域为 1。因为我们不需要 sel 域为 0 的 15 号寄存器 PRId，为了简便起见我们在程序中没有实现 sel 的识别。

2.3 指令集

我们实现了运行 CP0 必要的指令集，列在下方。

- 逻辑指令: AND, ANDI, OR, ORI, XOR, XORI, NOR, LUI
- 算术指令: ADDU, ADDI ,ADDIU, SUBU, MUL, MULT, MULTU, SLT, SLTU, SLTI, SLTIU
- 移位指令: SLL, SRA, SRL, SLLV, SRAV, SRLV
- 移动指令: MOVN, MOVZ, MFHI, MFLO, MTHI, MTLO
- 分支指令: J, JAL, JR, JALR, BEQ, BGTZ, BLEZ, BNE, BLTZ, BLTZAL, BGEZ, BGEZAL
- 访存指令: LB, LBU, LH, LHU, LW, SB, SH, SW
- 陷入指令: SYSCALL
- 特权指令: MTC0, MFC0, ERET, TLBP, TLBR, TLBWI, TLBWR

2.4 中断异常

MIPS32 中的异常包括中断 (Interrupt)、陷阱 (Trap)、系统调用 (System Call) 以及其他任何可以打断正常执行流程的操作。下表列出了我们实现的异常类型及其优先级。

优先级	异常	描述
1	Reset	硬件复位
7	Interrupt	检测到 8 个中断之一
12	TLB Refill	指令 TLB 缺失
13	TLB Invalid	指令 TLB 无效
16	Sys	执行系统调用指令 SYSCALL
16	RI	无效指令
20	TLB Refill	数据 TLB 缺失
21	TLB Invalid	数据 TLB 无效

Table 2.2: MIPS32 异常类型和优先级表

检测到异常发生后，MIPS32 对异常处理的过程如下：

1. 如果 CP0 中 Status 寄存器的 EXL 字段为 0，将异常原因保存到 CP0 的 Cause 寄存器的 ExcCode 字段。
2. 检查发生异常的指令是否在延迟槽中，如果在则设置 EPC 寄存器的值为受害指令地址减 4，Cause 寄存器 BD 字段为 1；否则设置 EPC 寄存器的值为受害指令地址，Cause 寄存器 BD 字段为 0。
3. 设置 Status 寄存器的 EXL 字段为 1，禁止中断。
4. 清空流水线，并根据异常种类，转移到相应异常处理例程的入口地址。
5. 软件进行异常处理。

- 处理结束，软件调用 ERET 转到异常发生前的状态。ERET 指令会清除 Status 寄存器的 EXL 字段，并且会清空流水线，将 EPC 寄存器中的值作为 PC，继续执行受害指令。

Cause 寄存器中有 8 个中断位，其中 6 个为外部硬件中断，2 个为软件中断。我们的 CPU 中使用了 3 个硬件中断位，3、4、7 号中断位分别用作 USB 中断、串口中断、时钟中断。

2.5 内存管理

MIPS 标准中对虚拟地址和物理地址的映射见下图：

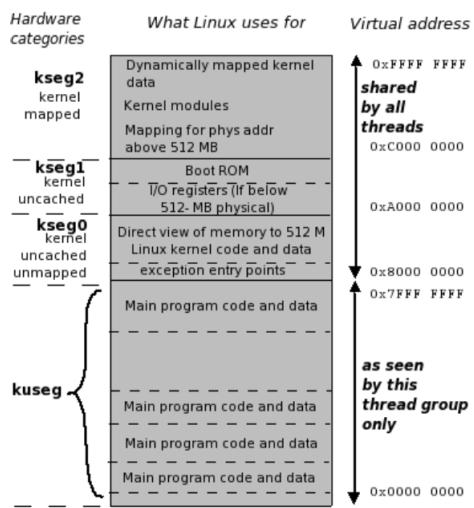


Figure 2.1: MIPS 地址映射 [1]

地址转换由 TLB 完成。如果在 TLB 中没找到相应页表则会触发页缺失异常，软件进行转换并将转换后的项填入 TLB。

2.6 代码结构

CPU 部分代码主要在 src/cpu 下，主要模块见下表。

文件	功能	文件	功能
cpu.sv	顶层文件	id/branch.sv	分支处理模块
cpuDefines.svh	定义 cpu 内部需要的宏	id/id_ex.sv	译码-执行传输模块
register.sv	寄存器模块	ex/ex.sv	执行模块
hilo.sv	hilo 寄存器模块	ex/ex_mem.sv	执行-访存传输模块
cp0.sv	cp0 模块	mem/mem.sv	访存模块
ctrl.sv	控制流水线的清空和暂停	mem/mem_wb.sv	访存-写回传输模块
if_pc_reg.sv	取指模块	mmu/mmu.sv	MMU 模块
if_if_id.sv	取指-译码传输模块	mmu/tlb.sv	TLB 模块
id/id.sv	译码顶层模块	mmu/tlb_lookup.sv	单个 TLB 查找
id/id_type.sv	指令译码模块		

Table 2.3: 文件结构

3. 外围设备

3.1 地址分配

下表列举了对内存空间的分配情况，在 src/defines.svh 中定义

名称	起始地址	结束地址
Sram_Base	0x80000000	0x80400000
Sram_Ext	0x80400000	0x80800000
Uart_data	0xBFD003F8	0xBFD003FC
Uart_Status	0xBFD003FC	0xBFD00400
VGA	0xBA000000	0xBA000E80
Bootrom	0x8FC00000	0x8FC00400
Flash	0xBE000000	0xBF000000
USB	0xBC020000	0xBC020008
Display	0xBFD00400	0xBFD00414

Table 3.1: 地址分配情况

3.2 总线

总线定义在顶层文件 src/intomips_top.sv 中，负责连接 CPU 和各个外设并处理 CPU 与所有外设的交互。

总线全部使用组合逻辑，根据 CPU 给出的地址以及实际地址分配，判断应向哪个外设发送相应的信号，并将得到的结果作为输入返回给 CPU。外设的中断信号/暂停信号也是在此处连接上 CPU 对应的信号。

3.3 存储设备

存储设备包括 Sram(作为易失性的内存)，Flash(作为非易失性的外存) 和 Bootrom(作为启动引导程序的只读存储设备)，这三者有着类似的接口。为了使得 CPU 能够在一个周期内能够往存储设备(主要指 Sram)中进行一次读写，我们存储设备的时钟采用了 CPU 时钟的两倍，并且是反向的，即两个时钟的上升沿不会同时出现。

3.3.1 Sram 控制器

Sram 控制器状态机见图3.1

虽然整体架构采用的是 Harvard 结构，指令与数据分别存放在 BaseRam 和 ExtRam 里，但仍然有可能从 BaseRam 里读数据，从 ExtRam 里读指令。例如，在监控程序 kernel 里，开始时的提示信息“MONITOR for MIPS32 - initialized.”作为全局变量，跟指令放在了一起，因此必然出现从指令 Ram 里读数据的情况，此时需要解决结构冲突。

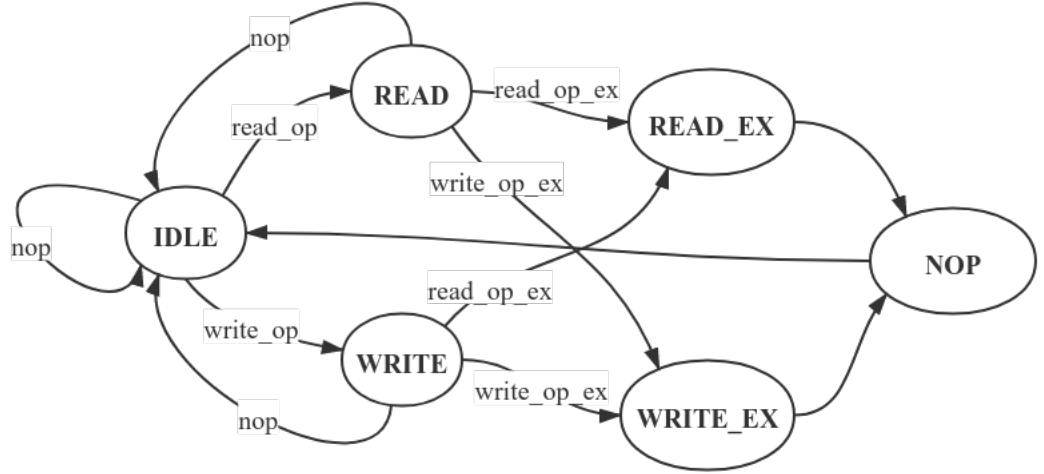


Figure 3.1: Sram 控制器状态机

解决结构冲突采用了增加控制器端口以及暂停流水线的做法。如上图状态机所示，倘若同一时间只有一次读写，则 CPU 部分只会给出 read_op 或者 write_op 信号，Sram 控制器状态经过一次 READ/WRITE 后就会回到初始的 IDLE 状态，为正常的读写；若同时有两次读写，例如同时有读，写请求，则 CPU 会同时给出 read_op 和 write_op_ex，Sram 控制器才结束 READ 状态后判断 write_op_ex，然后进入 WRITE_EX 状态进行写操作，同时对外（CPU）给出暂停信号，然后直到 NOP 状态结束后才关闭暂停信号。

之所以最后还有一个 NOP 状态，是为了对齐时钟周期，使得两次读写用 4 个外设时钟周期（2 个 CPU 时钟周期）完成，而不是 3 个外设时钟周期，这样避免在第 4 个外设时钟周期时收到上次未结束的 CPU 给控制器的信号，从而运行错误。

3.3.2 Flash 控制器

Flash 控制器的状态机见图3.2

由于 Flash 的数据线为 16bit，因此 CPU 的每一个读操作（读取一个 32bit Word）在控制器里需要进行两次读写，将结果拼在一起后才发给 CPU。Flash 的写则只支持 16bit 写（控制器会忽略掉 CPU 发来数据的高 16bit），写的操作则与 Sram 写操作类似。由于 Flash 对于时序的要求，我们在此处的状态机的状态数目更多，并且每次读写后都会留几个不做任何事情的状态来确保连续读写的正确性。

3.3.3 Bootrom 控制器

Bootrom 控制器内部使用 Xilinx 提供的双端口 ROM IP 核，因此没有状态机，只有一个元件例化。ROM IP 核大小选择了 1KB，按字编址。由于是双端口，因此也支持两次同时读（虽然实际

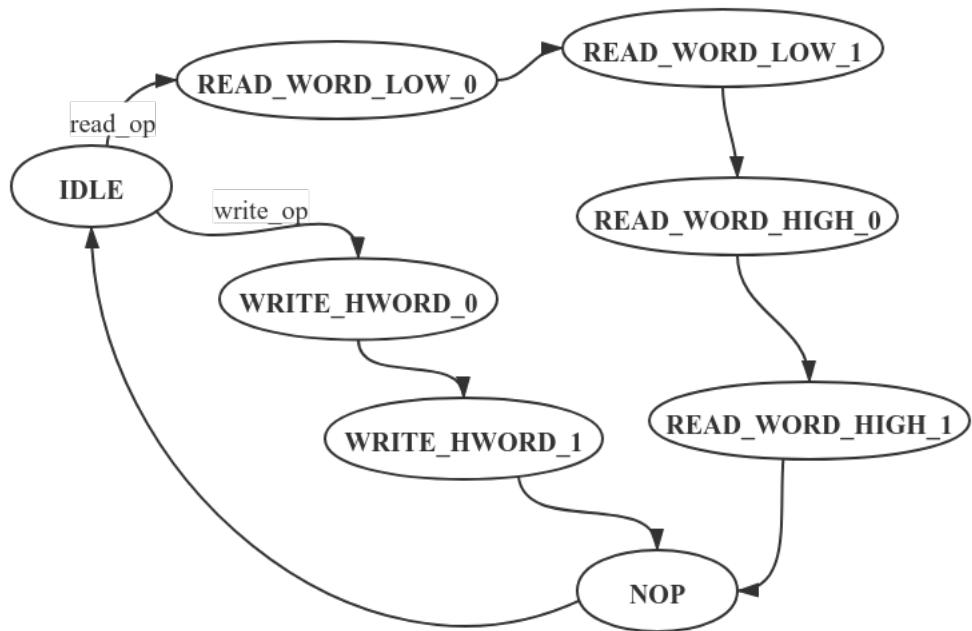


Figure 3.2: Flash 控制器状态机

使用时并不会出现这种情况)。

Bootrom 采用 coe 进行初始化设置，其代码 (位于 vivado/intomips.srcs/sources_1/bootrom.coe) 为简单的从 Flash 复制 4MB 数据到内存起始位置 (0x80000000 开始)。这里写了一个简易的 bash 脚本 (vivado/intomips.srcs/source_1/generate_bootrom_coe.sh) 来方便地从汇编生成对应的 coe 文件

3.4 Uart 控制器

为了避免总线冲突问题，我们最终使用了直连串口 (src/peripheral/ext_serial_controller.sv) 而不是 CPLD 串口 (src/peripheral/serial_controller.sv)，但 CPLD 串口控制器也是能够正常工作的，通过了监控程序第二阶段的测试。

直连串口底层的收发器采用了张宇翔助教所提供的收发器样例 (src/peripheral/async.sv) 来处理协议相关问题，而实际 Uart 控制器内部逻辑就十分简单。当 CPU 给出读信号 (read_op) 并且串口数据已准备好时，则将收到的数据发给 CPU；当 CPU 给出写信号 (write_op) 并且串口此时不忙碌时将 CPU 的数据交给内部收发器。

直连串口的 2 个状态寄存器 (是否忙碌，数据是否准备好) 则直接从 Uart 控制器连出去，在总线部分跟 CPU 进行实际的连接。

3.5 VGA 控制器

整个 VGA 显示器画面格式为 $800 \times 600 @ 75\text{Hz}$, 每个像素为 8bit(3bit 红色, 3bit 绿色, 2bit 蓝色)。为了实现硬件处理字体显示逻辑(使用 8×16 的字体矩阵), 我们将整个屏幕分割成了一个个 block, 每个 block 包含 8×16 个像素, 因此整个屏幕就由 100×37 个 block 组成(忽略掉剩下的几行像素), 每个 block 其实就是一个 ascii 码字(即需要 $8 \times 16 = 128\text{bit}$ 来存储字体矩阵信息)。

VGA 与 CPU 交互的数据总共为 32bit, 使用了其中的 $1 + 8 + 8 + 8 = 25\text{bit}$, 包括 1bit 的闪烁模式(1 表示该 block 应该闪烁), 8bit 的背景色, 8bit 的前景色和 8bit 的 ascii 码信息。闪烁模式, 前景背景色数据被存在了颜色显存中, 而 ascii 信息被存在了字符显存中。

VGA 控制器的存储部分使用 IP 核, 如下表所示, 除此之外还有一个时钟生成器为这些 IP 核提供时钟

名称	IP 核类型	数据宽度
ascii 码字库	单端口 Rom	7bit
字符显存	双端口读写 Ram	128bit
颜色显存	双端口读写 Ram	24bit

Table 3.2: VGA 存储

汇编部分使用 VGA 的方式为首先生成一个包含闪烁模式信息, 前景色, 背景色, ascii 码的 32bit 的 Word, 然后将这个 Word 直接 Save 到对应的位置中(指 block 在屏幕上的位置)。在 VGA 控制器内部, 由于需要先根据 ascii 码信息从字库中得到对应的字体矩阵, 因此我们做了一个小型的二级流水线, 第一级将 ascii 信息交给 ascii 字库(同时将颜色/闪烁放入到颜色显存中), 第二级将从 ascii 字库得到的数据放入到字符显存中, 由此 VGA 就不需要进行 CPU 的暂停操作。

3.6 USB 控制器

USB 控制器的状态机见图3.3

USB 被分配的地址为 0xBC020000 到 0xBC020008, 但实际上对应两个寄存器, 0xBC020000 对应地址寄存器, 0xBC020004 对应数据寄存器。USB 控制器内部将从 CPU 得到的地址右移 2 位后, 其第 0 位被 USB 用于判断下步操作是对地址线还是数据线。实际 USB 的读写逻辑比较类似, 都是先向数据线上发送一个地址数据, 然后若为读, 则从数据线上读取一个字节的数据; 若为写, 则往数据线上写入一个字节的数据。

由于 USB 也有较为严格的时序限制, 这里状态机的状态也相对更多, 并且中途会有暂停信号向外发出。

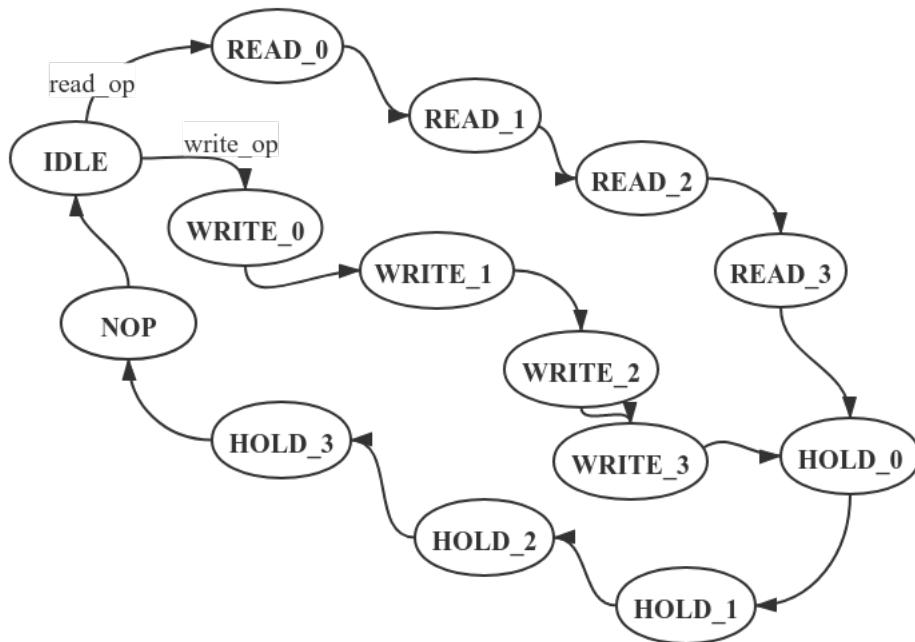


Figure 3.3: USB 控制器状态机

3.7 Display 控制器

Display 控制器组合了硬件板子上的 16 位 LED 灯，32 位拨码开关，4 位按钮开关和两个数码管，通过 CPU 的读写信号来控制各个部件的显示。

地址分配中给 Display 部分分配了 0xBFD00400 到 0xBFD00414 的地址空间，针对每个部件的分配见下表 (位于 src/peripheral/peripheralDefines.svh)

名称	地址
LED	0xBFD00400
数码管	0xBFD00408
32 位拨码开关	0xBFD0040c
4 位按钮开关	0xBFD00410

Table 3.3: Display 地址分配

4. 操作系统

我们需要在计算机系统上运行 ucore 操作系统，为此本组移植了 ucore-thumips 操作系统，对其进行调试、移植，并新增了一些内容来支持我们的硬件系统与运行目标。

4.1 系统介绍

ucore-thumips¹ 操作系统是 ucore 操作系统的 mips32S 平台版本，我们的版本 fork 自张宇翔学长的仓库 <https://github.com/z4yx/ucore-thumips>，ucore-thumips 为 mips32S 设计，并不支持完整的 mips32 指令集，一些指令如 divu 是无法直接使用的。ucore-thumips 可以直接在 qemu 的 qemu-system-mipsel 模拟器上运行，我们的目的是让我们的计算机硬件系统也支持 ucore-thumips。

4.2 操作系统移植

4.2.1 编译方法

由于我们时在非 mips 指令集的非 ucore 系统上编译 ucore，我们必然需要使用交叉编译。我们在 ubuntu 系统中使用 MTI Bare Metal 工具链，对操作系统进行编译。原 ucore-thumips 文档中推荐使用的 GCC-MIPS 工具链也是可以的，操作系统文档中使用的版本为 GCC4.6

编译前，将 Makefile 中的 CROSS_COMPILE 变量修改为工具链前缀如 mips-mti-elf- 即可，另外还可在 Makefile 中修改为用户程序预留空间的编译选项。

4.2.2 移植过程

为了让 ucore-thumips 能够正常运行，需要修改其 CPU 的预设时钟频率，让其能够每 1 毫秒请求一次时钟中断。

为了能够支持我们的外设如 VGA 显示器，我们在 kern/driver 文件夹下新增了一些驱动程序。为了支持 USB 键盘，我们修改了驱动中的设备地址。

另外需要注意的是目前支持的 USB 协议版本为 1.1，但是现存的很多键盘是从 USB2.0 开始支持的，这个问题我们一开始没有注意，导致 Debug 了较长时间，直到我们使用了 logitech 的键盘。

为了让我们的用户程序能够使用除法与取模运算，我们用软件实现了 div 与 mod 运算。

4.3 新增内容

4.3.1 修改对退格的支持

原来的 shell 程序 sh 虽然在程序中处理了 \b 字符，但是键盘如果按下 backspace 键，得到的并不是 \b 而是 del (127) 控制符，因此需要同时对 \b 与 del 控制符进行处理。另外原 shell 程序在退格时没有覆盖原来当前行输出的字符，只是把虚拟光标左移，我们进行了修改，在退格时会用空格覆盖原来输入的字符。

¹<https://github.com/z4yx/ucore-thumips>

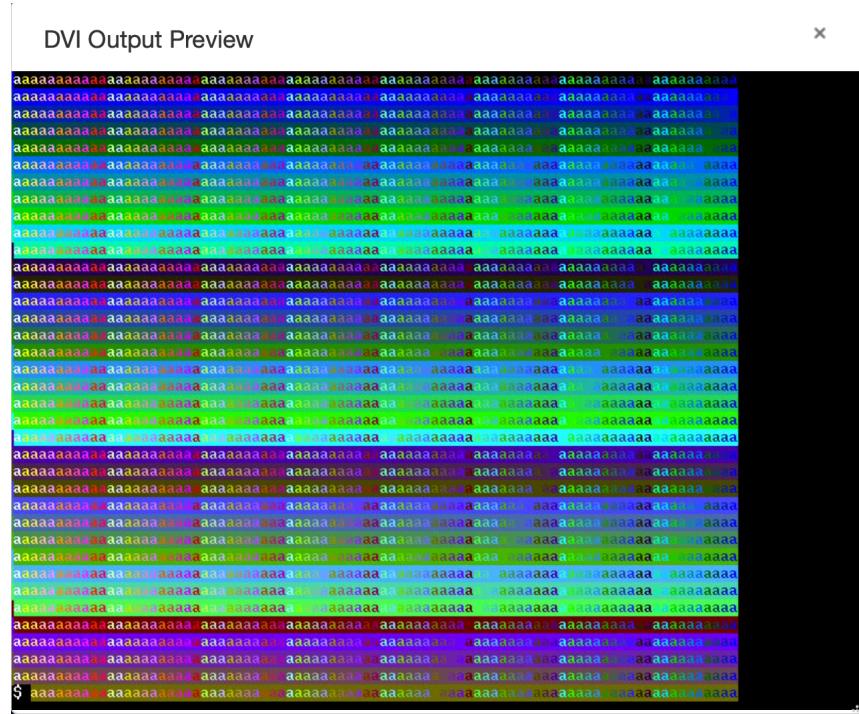


Figure 4.1: 部分字体颜色展示

4.3.2 VGA 显示

在之前的外设部分提到，VGA 控制器接收的信号是基于字符与颜色的，共支持的颜色组合是前景色 256 色，背景色 256 色，共 65536 种颜色组合。Figure 4.1 是我们部分颜色的展示。这个颜色组合的范围大于标准的 VGA 字符模式²规定的 4bit 前景色 3bit 后景色共 128 种颜色组合。

另外，我们的 VGA 驱动中单独处理了光标的位置、每个字符是否闪烁、退格等操作的支持。

4.3.3 贪吃蛇程序

为了验证我们系统的整数运算能力、内存管理能力、usb 键盘支持与 VGA 绘图能力，我们编写了用 VGA 屏幕输出、键盘输入的贪吃蛇程序。Figure 4.2 为用户使用键盘控制贪吃蛇，并从屏幕获取信息的演示。

贪吃蛇需要用户实时从键盘或者串口输入字符进行控制，但是 ucore 本身只有阻塞读取的接口，没有支持带 timeout 的非阻塞输入 syscall，因此需要使用多进程的方法，在进行更新绘制的同时去监听用户的输入。而 ucore 本身没有实现进程间通信的 syscall 如 pipe，我们使用了这样一种方法来替代：

由于我们的时钟是可靠的，我们可以在某一时刻知道自己目前过了多少个时钟周期，那么在我们只要每次刷新时记录此时的周期数，就能算出比起上次刷新过了多少个时钟周期；基于这个方法，我们可以用父进程阻塞地读取用户的输入，子进程自动刷新贪吃蛇及屏幕，当用户输入时，结束子进程并计算距离上次输入的时间，知道时间后就可以推演出这次用户输入前贪吃蛇的位置，在

²<http://www.osdever.net/FreeVGA/VGA/VGAtext.htm>



Figure 4.2: 贪吃蛇程序演示

此位置基础上去根据用户的输入计算游戏状态。这样我们就可以变相地实现一个带进程通信的贪吃蛇。

5. 测试与集成

本项目大部分测试采用了自动化的测试，确保设计的正确性。我们使用 tenchbench 进行硬件测试。

5.1 CPU 测试

本部分用于测试指令实现的正确性。通过编写指定格式的汇编代码，包含指令和指令的期待行为，可以通过 make 生成仿真储存文件和答案文件，再通过仿真期间观察程序行为和答案是否一致来进行自动化测试。此外我们实现了自动化指令测例随机生成，进一步加强了测例。其中特定指令的期望行为在脚本中用 python 实现，实现者与 CPU 指令实现者不同。

文件结构见下表

文件	功能
testbench/cpu/cpu_test.sv	CPU 测试顶层模块
testbench/cpu/cpu_test_tb.sv	测试主要文件
testbench/cpu/fake_ram.sv	一个假的 ram
testbench/cpu/fake_rom.sv	一个假的 rom
testbench/cpu/testcases/gen.sh	最终自动生成全部测例的脚本
testbench/cpu/testcases/generate_cases.py	生成测例的脚本
testbench/cpu/testcases/inst_reg.py	期望指令行为
testbench/cpu/testcases/make_answer.py	生成答案文件
testbench/cpu/testcases/Makefile	编译汇编并生成储存文件和答案文件
testbench/cpu/testcases/instructions/*.s	汇编测试文件，“auto_”开头的文件是自动生成的
testbench/cpu/testcases/instructions/*.mem	仿真储存文件
testbench/cpu/testcases/instructions/*.ans	答案文件

Table 5.1: CPU 测试文件结构

5.2 外设测试

本部分被用单独的外设测试 (不涉及 CPU，测试 Sram 和 Flash)，用于测试外设控制器实现的正确性。由于不涉及 CPU，外设测试采用的方法是直接对外设 (fake_sram, fake_flash) 进行初始化，然后按顺序进行读取和写入，读取测试则与期待结果进行对比，写入测试则直接深入外设内部检查是否正确写入。与 CPU 测试类似，我们实现了自动化样例生成，对于外设即是自动生成随机的内存内容，并同时生成对应的读取/写入期望结果，使用 python 脚本进行生成，以此检测读取/写入的正确性。

文件结构见下表

文件	功能
testbench/peripheral/peripheral_test_tb.sv	外设测试顶层模块
testbench/peripheral/{*_test.sv, *_test_tb}	测试主要文件
testbench/peripheral/fake_flash.sv	一个假的 Flash
testbench/peripheral/fake_sram.sv	一个假的 Sram
testbench/peripheral/make_testcase.py	生成测例的脚本
testbench/peripheral/testcases/*.mem	测例对应的内存初始化内容
testbench/peripheral/testcase/*.ans	测例对应的答案

Table 5.2: 外设测试文件结构

5.3 系统测试

本部分测试被用于完整实现 CPU 各个部件，并与外设进行连接后的测试。具体分为两个部分，云端测试和本地仿真测试。云端测试则是编写各种汇编代码，在云端上进行行为的观察。本地仿真测试则采用了张宇翔助教提供的行为仿真模型来替换之前我们自己编写的存储模型，以达到更高的测试要求。

文件结构见下表

文件	功能
testbench/cloud/*	云端测试
testbench/top/*	本地仿真测试
testbench/model/*	行为仿真模型

Table 5.3: 系统测试文件结构