计算机系统 综合实验报告

李晗 191870085 （Github 账号 Undefined01）

高灏 191250034 （Github 账号 UYTFVBNJ）

目录

计算机系统 综合实验报告

- 目录
- 说明
  - 组内分工
- 项目简介
  - 工具准备
  - 硬件设计
  - 硬件测试
  - 持续集成
  - 软件设计
- 硬件实现细节
  - 五级流水线
  - MMIO
  - 测试
- 软件实现细节
  - 最小C运行时
    - crt0
    - libgcc
    - 封装
  - 交互程序
  - 命令
  - 自动化
  - native 环境

说明

本项目全程使用 Git 进行版本控制和开发流程追踪。

主项目地址：https://github.com/Undefined01/riscv

组内分工

李晗

- 设计并实现 CPU 的五级流水线结构。实现 RV32I 指令集。
- 设计并实现基于 Quartus 和 ModelSim 的 CPU 测试激励模块。设计并实现 CPU 自动化测试框架，使用汇编语言编写 CPU 测试用例。
- 设计并实现 MMIO 结构。设计并实现 VGA，键盘，时钟等外设。
- 设计并实现最小 C 运行时，以便展示项目支持使用 C 语言进行编写。
- 编写展示项目中的 md5 哈希算法和 snake 贪吃蛇游戏。

高灏

- 改编riscv-test中的测试数据，使之可以应用在测试框架中。使用gcc编译复杂测试数据。
- 设计并实现交互界面。
- 对最小C运行时进行一些补充。
- 编写展示项目中的求斐波那契数列，hello，help指令。

项目简介

本项目实现了基于 RISC-V 指令集的 CPU （RV32I），并针对 Cyclone 5CSXFC6D6F31C6 开发板实现了相应的外设模块。 CPU 可以通过 MMIO 的方式对外设进行控制并运行具有一定实用性的程序。

实现了一个最小的 C 运行时，展示项目使用 C 语言编写。

工具准备

1. Quartus Prime Lite Edition 18.1： Verilog 综合软件。用于编写 CPU。
2. ModelSim - Intel FPGA Starter Edition 18.1： 仿真软件。用于测试 CPU。
3. RISC-V GNU Compiler Toolchain RISC-V GNU 工具链。用于编译C、汇编等源文件到可执行二进制（目标三元组为riscv32i-unknown-elf）。
4. riscv-tests RISC-V 单元测试。用于测试硬件 CPU。

硬件设计

Quartus 项目位于 fpga 目录。

设计源文件主要位于 fpga/src 目录。

- CPU 频率： 50MHz
- 流水线结构： 五级流水线（IF,ID,EX,MEM,WB）
- 上电启动地址： 0x00000000 （非标准）
- 内存大小： 64KB （定义于 common.v）
- 指令集： RV32I （未实现特权相关指令，如 ecall 和 sret。未实现非四字节内存访问，如 lb, sh 等等。设计程序时应总是使用 int 等四字节变量。）
- 内置固件： firmware/final （定义于 perip.v）
- MMIO： 定义于 common.v ， 在 perip.v 中进行转发。具体映射如下：

Address	Description
0x00000000 - 0x10000000	内存。仅使用开头到实际内存大小的段。指令、数据等均存放在内存中。

Address	Description
0xa0000000 - 0xa1000000	字符显存。仅使用开头到实际显存大小的段。向此处写入字符的 ASCII 码即可被自动显示在屏幕上。
0xa1000010 - 0xa1000020	键盘。仅使用第一个四字节。从此处读取键盘事件。
0xa1000020 - 0xa1000030	启动时间。仅使用前两个四字节。从此处读取以小端序保存、 8 字节的、以微秒计的启动时间。

硬件测试

测试激励模块和简易测试用例位于 fpga/testbench 目录。

可使用 make 自动编译测试用例（需要安装 riscv64-unknown-elf 工具链）。

本项目还移植了 RISC-V 的官方测试集 riscv-tests （位于 riscv-tests 目录下）。 在使用 git submodule update --init 初始化子项目后，可在 riscv-tests/isa 目录下执行 make 命令自动编译测试用例（需要安装 riscv64-unknown-elf 工具链）。

编译完测试用例后，可在 Quartus 中进行 RTL 级仿真。 激励模块中有简易的测试框架，能够对 CPU 进行自动化测试。

持续集成

由于项目的开发基于 git/Github 并配置了相应的测试脚本，因此可以在每次提交 commit 时自动触发持续集成进行测试。持续集成使用的测试用例与上一小节中的测试方法相同，但是测试由 Github Actions 自动执行。因此在小组成员各自独立编写代码的过程中，如果由于代码功能或接口的修改导致另一模块出现错误，会被 Github 立即发现并且定位到 commit 级别。再根据 git diff 重新审查代码的修改，即可确认错误。既减少了修改代码时的心智负担，又能避免日后合并代码时发生隐含的错误。

软件设计

最终展示时使用的软件位于 firmware 目录。使用 C 语言进行开发。

其中， crt 目录为支撑 C 程序运行的最小运行时，包括 putchar 、 sleep 等常用函数）。

final 目录为最终展示程序。可以使用 `make` 指令一键编译。 该程序实现了一个最小终端，并支持运行以下程序。

程序名称	程序描述和参数
help	输出终端支持的指令。
hello	输出 hello world 并退出。
echo	原样输出参数。
fib	需要一个整数`n`，输出斐波那契数列的第`n`项。
uptime	输出 CPU 启动的时间。
md5	需要一个字符串，输出该字符串的 `MD5` 哈希值。
snake	进入贪吃蛇小程序。使用`wasd`进行控制，使用`q`退出。

硬件实现细节

五级流水线

CPU 部分的设计使用经典的五级流水线结构。即 IF,ID,EX,MEM,WB 五个阶段。

在设计时，由于 CPU 的功能较为简单，并没有完全按照五级流水线对应的功能实现模块，而对其进行了一定的简化。本项目中各级流水线的功能如下：

- IF：使用组合电路计算取指的地址，即根据 CPU 当前状态和控制信号选择进入跳转地址或分支预测地址（本项目中分支预测仅为“占位符”，但通过 ID 阶段可以较容易的根据译码信息给出合理的分支预测）。
- IF-ID：此上升沿根据 IF 给出的地址进行取值。实际该信号交由 perip 模块执行内存读取。
- ID：对取得指令进行译码，将指令进一步解释成一系列 RTL 操作。与此同时访问寄存器组或提取立即数，解析出 EX 阶段所需的操作数。由于 X0 寄存器的存在，可以将不需要写入寄存器的指令等效为对 X0 寄存器进行写入，进行简化。 需要注意的是，此指令访问寄存器组的时候上条指令的寄存器写回阶段还并未执行。需要使用“转发”对译码的操作数进行修正。而修正时同样需要注意 X0 寄存器永远为0。例如， add 指令将被译码成 RTLTYPE_ARICH,RTLOP_ADD ， 而src1和src2即为对应寄存器的值。auipc 指令也会被译码成 RTLTYPE_ARICH,RTLOP_ADD ， 此时其操作数为 pc 和指令中的立即数。
- ID-EX：无特殊用意，仅为分割流水线。
- EX：实际执行 ID 阶段中给出的 RTL 指令，如加法、移位等。在进行计算后将计算结果和目的寄存器转发回 ID 模块。 如果遇到 RTLTYPE_JUMP 指令则会输出控制信号，指示 IF 进行跳转。 如果遇到 RTLTYPE_MEMREAD 指令则会输出控制信号，指示流水线暂停。原因将在 EX-MEM中说明。
- EX-WB：如果 EX 执行的不是内存访存指令，则直接进入此阶段。并利用该上升沿对寄存器组进行写回操作。
- EX-MEM：如果 EX 执行的是内存访存指令，则此上升沿将信号传递给 perip 模块执行内存访存。 如果执行的是内存的写入操作，由于不需要写回（即写回寄存器为 X0） ，因此可以安全的和 WB 阶段并行执行。 如果执行的是内存的读取操作，由于需要访问内存，无法在 ID 阶段之前返回需要转发的数据。因此需要 EX 模块立即发出流水线暂停信号，暂停一个周期。在下一个周期时，perip 已经读取到了正确

的数据。于是更新写入目的寄存器的数据和当前状态。由于流水线暂停，其他模块感知不到 MEM 阶段，流水线会自动重新进入 EX-WB 阶段并执行。

MMIO

MMIO 功能由 perip 模块进行处理。其中， perip 模块的功能类似于分段机制。在收到读写地址之后， perip 模块会对地址进行分发。即，设置各个子模块的读写使能标志，根据偏移量计算子模块对应的读写地址，最后通过多路选择将子模块的读取结果返回。

在这样的结构下，内存 RAM 、显存 vga_term 、键盘 keyboard 、时间 time 几个模块是并列关系，都是 perip 的子模块并受其控制。

此外，由于不存在缓存机制（ cache ），如果每次取指都触发流水线暂停（否则可能与 MEM 冲突）显得很不划算。因此此处内存使用了双口RAM代替。

使用双口RAM直接读写内存还有另外一个问题，就是不同字长的读写。目前为了方便，双口RAM的位宽为 32 字节，这样就能一次性读取 4 个字节的指令。然而这种实现难以支持 lb 、 sh 等 1 字节和 2 字节的内存读写。

一种较好的实现是将内存分割成四个独立的 8-bit RAM。此时连续的 4 Byte 被分散到四个独立的RAM中。如果需要进行读取操作，则可以在一个上升沿同时读取四个RAM，并使用多路选择器对结果进行组合，进行符号扩展等，返回需要的读取结果（由于最终需要写入寄存器，内存读取指令所需结果均为 32 位）。

而如果需要进行写入操作，则可以通过进一步译码产生额外的控制信号，单独对每个 Byte 对应的 RAM 进行操作。代码框架如下：

```
reg [7:0] ram0[`RAM_SIZE / 4];
reg [7:0] ram1[`RAM_SIZE / 4];
reg [7:0] ram2[`RAM_SIZE / 4];
reg [7:0] ram3[`RAM_SIZE / 4];

reg [7:0] rbyte0, rbyte1, rbyte2, rbyte3;
reg [7:0] wbyte0, wbyte1, wbyte2, wbyte3;
reg wena0, wena1, wena2, wena3;

wire [29:0] block_addr = addr[31:2];
wire [1:0] sub_addr = addr[1:0];

always @(*) begin
  // 对 sub_addr 译码，设置 wbyte0 - wbyte3 和 wena0 - wena3
```

```
  // 译码也可以在 EX 阶段完成
end

always @(posedge clk)
  if (rw = `MEM_READ) begin
    rbyte0 <= ram0[block_addr];
    rbyte1 <= ram1[block_addr];
    rbyte2 <= ram2[block_addr];
    rbyte3 <= ram3[block_addr];
  end
  else begin
    if (wena0) ram0[block_addr] <= wbyte0;
    if (wena1) ram1[block_addr] <= wbyte1;
    if (wena2) ram2[block_addr] <= wbyte2;
    if (wena3) ram3[block_addr] <= wbyte3;
  end

assign rdata = {rbyte0, rbyte1, rbyte2, rbyte3};
```

由于支持非四字节读写的代码较长，且需要在读取阶段的上升沿前后进行额外的组合操作，时序较为复杂，因此在本项目中并未支持非四字节的存取。其相应的指令应在软件层面进行规避。

测试

由于 CPU 体系较为复杂，本项目在开始设计时就非常重视对 CPU 功能的测试和整个项目基础设施和脚手架的搭建。由于项目的特殊性，一次从编译到上板测试的验证流程至少需要 3 分钟。且很多错误很可能无法一次修正，需要重复很多遍验证流程。且在项目早期的开发过程中，CPU的功能还十分有限，难以支撑需要在开发板上运行的系统测试。因此项目基于的仿真的单元测试非常重要。

在项目的第一个 commit 中，刚刚建立起 IF-ID-EX-WB 流水线的基本架构时便实现了一个最简易的加法功能测试进行验证。此时得益于仿真软件，在执行过程中可以直接对寄存器、CPU控制信号等进行断言。并且通过记录波形可以随时查看 CPU 内部信号的变化过程。 随后随着功能日益复杂，包括转发、跳转等功能的单元测试也被加入了测试集中，测试框架也逐渐完善。 最初由于没有跳转指令，必须手动指定每一个测试用例需要执行的 cycle 数，防止 PC 越界运行意料之外的指令。 后来通过引入“魔数”，在测试框架中设置检测到魔数即停止运行。这样在不修改 CPU 实现的情况下就可以在仿真测试中自动检测当前测试用例的结束并切换至下一个测试用例。 此时，通过基于 testbench 的单元测试可以快速验证某些修改是否会影响 CPU 其他功能的正常运行，或通过测试用例快速判断出错原因并进行

修改。 而如果通过了仿真测试，实际验证却不通过，则可以通过与测试环境的对比，优先考虑没有被自动测试覆盖的条件，节约了大量的时间。

需要注意的是，由于 CPU 使用了流水线结构，在取指到最后一条指令时前几条指令还并未执行完成。因此在早期的测试用例中需要填充几条 nop 指令保证测试用例中的有效指令执行完毕。

此外，项目可以利用 RISC-V 官方的工具链、仿真器等从可读性较高的源代码快速生成测试用例。项目的 Makefile 文件也极大的简化了项目的测试流程。只需要编写测试对应的汇编文件，即可通过 make 指令一键从 .s 文件编译至 $readmemh 所需要的 .hex 文件。在项目早期需要大量更新测试用例的阶段提供了巨大的便利。

另一方面， RISC-V 在 Github 上也给出了一个官方的单指令测试集。该测试集基于分支指令进行结果的验证，并且在代码中大量使用了宏，方便移植。本项目就通过修改宏的定义快速迁移到了本项目的 CPU 上。

软件实现细节

最小C运行时

在 CPU 的测试用例中，由于需要对生成的代码的控制精确到指令级别，因此仍然使用汇编进行编写。 但是如果展示项目仍然使用汇编进行编写将会过于费力，无法享受到现代工具链的快速开发流程和高级语言的强大的表达能力。且如果只使用汇编几乎难以快速完成如 md5 算法、贪吃蛇游戏等的编写。 因此该项目结合了计算机系统基础中的知识，编写了一系列基础库以支持 C 程序的运行。

crt0

一般而言，在 libc 中需要存在一个 _start 入口函数作为程序运行的起点，对运行环境做初始化，包括但不限于设置初始栈指针，设置主函数的执行参数，执行 ELF 中定义的 init_array ，处理主程序的返回值等等。

由于展示程序最终将会运行在裸机环境上，且其内容可控。因此我们并不需要实现一个完整的 C 运行时，只需要提供最基础的功能并避免让 gcc 生成需要使用其他功能的代码即可。 具体而言，我们只需要根据 abi 设置初始的栈指针，用空数组调用主函数，在主函数返回后执行死循环即可。

相关代码位于 crt0.S 中。

libgcc

由于我们实现的 CPU 并不支持硬件乘除法。因此我们需要编写一个基于加减法和位运算的乘除法函数。 在 gcc 中，为了减少硬件间的差异并复用代码，在不支持乘除法的目标机器上使用 a * b 将会自动调用 __mulsi3 等函数进行计算。 由于无法使用 libgcc ，我们同样需要手动编写相关函数以提供支持。

相关代码位于 m.c 中。

封装

此外，为了隐藏项目内部输入输出相关的硬编码，该运行时还为程序提供对键盘，屏幕，时钟以及一些常用函数的接口：

- kbd.h：通过MMIO的方式读取键盘信息，提供API对键盘进行抽象。由于内存只能四位读写的特性，在实现读取字符时，使用内联汇编防止gcc的优化。提供等待事件，读取字符等操作。

- term.h：通过MMIO的方式控制显存，提供API对显存进行抽象。它使用cursor_tot、cursor_row、cursor_col在显存中定位，提供换行，回退，打印字符，清屏等操作。
- time.h：通过MMIO的方式读取时钟信息，提供API对时钟进行抽象。提供获取当前运行时间等操作。
- stdlib.h：实现了一些常用函数，方便程序的编写。提供rand、strcmp等函数。

交互程序

由于没有实现中断相关的指令，无法实现进程调度相关的功能，因此交互程序仅是一个较为简单的 shell 程序。

它会不断地重复读取命令，解析命令，然后调用相应的函数来执行命令的流程。

为了实现的方便，我们把函数来作为程序运行，通过调用函数的方式来运行程序，执行指令；这样，编写一个程序只需编写一个函数，并直接调用运行时提供的接口进行输入输出即可。

命令

交互程序提供了一些可以带参数的命令。为了执行这些命令，交互程序会调用相应的函数，并给定对应的参数，相当于运行了相应的程序。

目前提供的命令已在软件设计节中列出。

这些命令充分利用了硬件提供的键盘，时钟，显示器三个设备。他们的正确执行也充分证明了硬件的可靠性。

例如， md5 哈希程序将会执行较为复杂的位运算。而贪吃蛇程序则会综合使用如 显示相关API 绘制游戏场景，时间相关API 定时让贪吃蛇前进，键盘相关API 控制前进方向，以及较大的数组等储存地图数据。

自动化

编译库文件和可执行文件的过程非常繁琐，我们编写了make脚本实现编写的自动化，节约时间。同时，我们在quartus中设置了增量编译的选项，使得在仅修改软件后，quartus不必重新综合未曾变动的硬件，只需要更新存储相关的模块即可。编译时间也从2分50秒减少到了20秒，大大提高了编写和测试软件的效率。

native 环境

在 firmware-dev 开发环境中，由于 crt 接口设计的抽象性，可以通过更换 crt 的底层令 final 项目直接运行在 linux/native 上。因为不需要重新综合硬件，这样可以在极短的时间内完成展示项目的编译运行。与此同时，在 native 环境下还可以使用熟悉的 gdb 等工具进行调试，也可以通过 gcc 的“栈保护”编译选项检测缓冲区溢出的错误。在先前的代码中，terminal.c 就有一行代码由于笔误无法正确检测字符串的终止（原先为检测 \n，应检测 \0）。而通过 native 环境下的 gcc 和 gdb，即可较为轻松的完成 debug。

由于 C 语言不存在命名空间和 mangling，因此如果沿用之前的命名方式很可能会在链接 libc 时发生命名冲突。因此在 dev 中给绝大多数接口增加了 rt_ 前缀，并且将头文件放在 rt 目录下防止重名。

此外，根据原先设计的 硬件与 crt 的接口，只需要分别将从 MMIO 中获取数据的函数重新使用 linux 的接口实现即可。首先，对于 VGA 模块和 KBD 模块的模拟，使用了 POSIX 中定义的 termios 库对终端进行控制。相比于日常使用的终端，只需要关闭输入回显、关闭读写缓冲、以非阻塞的方式读取输入，即可让终端的输入输出与项目硬件支撑的输入输出高度相似。而 time 模块则可以很轻松的使用 gettimeofday 进行模拟。至于其他在 crt 中实现的功能则无需修改。

使用 gdb 进行调试时，由于展示程序需要独占整个终端以模拟 VGA 输出，如果使用 gdb 直接启动程序，gdb 的输出则会干扰展示程序的正常显示。因此可以通过 gdbserver 和 gdb remote 配合使用达到原终端模拟 VGA 输出而另一个终端进行调试、互不干扰的效果。