计算机组成原理、软件工程课程联合项目

“编程是一件很危险的事情”组

TrivialMIPS 项目
设计文档

组员：

计 63 陈晟祺
2016010981
harry-chen@outlook.com

计 64 周聿浩
2016011347
miskcoo@gmail.com

计 54 姚沛然
2015011315
xavieryao@live.cn

2019 年 1 月

目录

第一章　概述

1.1　项目背景

本项目是计算机组成原理与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“编程是一件很危险的事情”（*ProgrammingCanBeVeryDangerous*）小组，组长为计 63 陈晟祺，成员还包括计 64 周聿浩和计 54 姚沛然。

本项目的成果是在新的 32 位 ThinPad 实验板上设计并实现基于 MIPS 32 的 CPU，并使用实验板上的周边硬件，成为一个片上系统（SOC）。其能够支持标准 MIPS 32 Rev 1 指令集的一个较完整子集和 MIPS 32 Rev 2 指令集的部分功能，并能够运行 uCore 操作系统。在此之外，我们还完成了与编译原理的联合实验，能够运行 decaf 语言编写的程序。

1.2　项目概览

本项目计划设计和实现的部分主要包括：CPU、外设、Bootloader、操作系统移植、自动化测试部署。项目使用的硬件语言为 SystemVerilog 2005。下面为各个部分的概览。

1.2.1　CPU

CPU 的设计包含指令集、流水线结构（微架构）、内存管理单元、异常处理机制、协处理器以及其他增强功能。

指令集 本项目的 CPU 实现的指令是 MIPS 32 Rev 1 指令集的一个较完整子集，包括了所有的算术逻辑指令、控制流指令和大部分特权指令（不包括与缓存有关的），覆盖了 uCore 操作系统需要的所有 47 条指令。MIPS 32 Rev 2 中的部分指令（如 CP0 中的 ebase 寄存器）由于被操作系统需要，也包含在实现中。

流水线结构 本项目实现了经典的 MIPS 五级流水线结构，即分为取指、译码、执行、访存、回写阶段，每个阶段在 CPU 内部使用一个时钟周期。为此，需要解决一系列数据和控制流上的冲突、竞争。

内存管理单元 本项目实现了内存管理单元（MMU）以进行从虚拟地址到物理地址的映射，本项目的内存划分遵循 MIPS 32 标准，将使用转换检测缓冲区（TLB）以加速页表的查询，并对所有外设实现内存映射 IO（MMIO）。

异常处理机制 本项目完整支持 MIPS 32 Rev 1 的异常和中断机制，正确处理同步和异步异常，支持硬件和软件中断，并实现精确异常。

协处理器 本项目实现了 MIPS 32 Rev 1 中为 CP0 处理器规定的几乎所有指令和寄存器，以正确运行操作系统。

增强功能 本项目实现了一系列性能与功能上的增强，包括 CP1 浮点协处理器、指令双发射、静态分支预测等。

1.2.2 外设

外设是板上系统中除 CPU 以外的部分，本项目需要根据各芯片给出的参考手册实现与各个设备的通信，将设备的功能统一暴露为一系列可读写的物理地址（即 MMIO）。除此之外，还需要实现简单的指令和数据总线处理和分发读写请求。实现的外设模块有：

SRAM 本模块被用作板上系统的内存，支持读写，大小 8MB。

Bootrom 本模块是每次上电和重置后 CPU 默认的指令加载位置（物理地址 `0x1FC00000`），只读，将预置 Bootloader 加载下一级引导程序或操作系统。

NOR Flash 本模块用作板上系统的非易失性存储，大小 8MB，支持读写（非直接对 NOR 设备的读写，可能被 Flash 片上控制器翻译为各种控制命令）。

UART 串口 本模块将串口通信映射为两个地址（控制与数据寄存器），并在数据到来时向 CPU 发出中断信号。同时收发两侧都有缓冲区以防数据丢失。

Framebuffer 本模块提供了显示功能，提供一块内存区域作为环形的图形缓冲区和一个寄存器用作渲染偏移量指示，并实时地将缓冲区的内容输出为 HDMI 信号。我们还利用特殊的写入方式实现了硬件解码的功能。

GPIO 本模块用作控制 ThinPad 开发板上的 GPIO 设备，如拨码开关、七段数码管和 LED 灯，通过读写对应寄存器能够获取或者改变这些设备的状态。

Timer 本模块提供精确的计时功能，累计自上一次重置以来经过的时间（单位为微妙）和 CPU 周期数。用户也可以在任意时刻改写这些值。

其他 本项目还实现了 USB 模块控制板上的 SL8111USB 控制器，以及以太网模块控制板上 DM9000 控制器，为操作系统提供 USB 与网络通信功能。

1.2.3 Bootloader

Bootloader 用于引导操作系统，本项目中运行的 Bootloader 分两个阶段，分别是自行编写的 TrivialBootloader 和移植的 U-Boot。前者是被固化在 Bootrom 中的程序，需要支持从 Flash、SRAM、串口等多途径启动，提供任意地址转储功能，负责基本的异常处理，并支持内存和外设的检查。而 U-Boot 是被 TrivialBootloader 加载的较复杂的引导程序，支持网络启动、USB 启动、性能测试等高级功能，需要对源代码进行平台相关移植。

1.2.4 操作系统移植

作为联合项目的要求，本组在板上系统上成功运行了 `uCore-thumips` 操作系统，包括进行一些平台相关的改动，以及正确实现 CPU 的各项功能（尤其是 MMU 相关模块）、正确与外设进行通信。为了演示本项目的成果，我们还在操作系统上编写了数个用户态程序，

并移植 USB 键盘驱动到内核。同时，我们也为 decaf 编译器后端编写了对应的链接库，以使其编译出的 MIPS 汇编能够正常在我们的平台上运行，并支持输入、输出等功能。

1.2.5　自动化测试与部署

作为软件工程的要求，也是对计原在线平台的测试与协助开发，本项目实现了基于 GitLab CI 的自动化综合、测试、部署系统，包括以下的功能：

- 项目需求、设计等文档的自动编译
- 基于事先撰写的 testbench 自动对 CPU、外设和整个板上系统运行 RTL 仿真
- 自动调用 Vivado 生成 Bitstream 文件，并缓存可复用的中间结果
- 所有相应软件的自动编译，使用 QEMU 对操作系统进行测试
- 使用在线实验平台 SDK，在真实环境中运行性能、功能测试和操作系统，并提取数据进行分析和报告

1.3　名词解释

表1.1中是本项目中可能用到的一些名词缩写及它们的解释，以后本项目相关的文档中将不加解释地使用这些缩写。

缩写	全称	含义
MIPS	Microprocessor without Interlocked Pipeline Stages	无内部互锁流水级的微处理器
CPU	Central Processing Unit	中央处理器
FPU	Floating Point Unit	浮点处理器
CP0/1	Co-Processor 0/1	协处理器 0/1
ALU	Arithmetic Logic Unit	算术逻辑单元
MMU	Memory Management Unit	内存管理单元
TLB	Translation Lookaside Buffer	旁路快表缓冲
PA/VA	Physical/Virtual Address	物理/虚拟地址
ROM	Read Only Memory	只读存储器
(S)RAM	(Static) Random Access Memory	(静态) 随机访问存储器
UART	Universal Asynchronous Receiver-Transmitter	通用异步接收器-发射器
GPIO	General-Purpose Input/Output	通用目的输入/输出
MMIO	Memory Mapped Input/Output	内存映射输入/输出
SOC	System On a Chip	片上系统

表 1.1: 名词缩写和解释

1.4　开发平台

1.4.1　硬件平台

本项目使用的硬件开发板为 ThinPad-NG，其主要部件为 Xilinx 的 Artix 7 系列 FPGA，型号为 xc7a100tfgg676。此外还有外部器件：

SRAM 模块 ISSI IS61WV102416ALL，每片 16Mbits，共 4 片，读写周期 10 ns

NOR Flash Numonyx JS28F640J3D75，每片 32Mbits，共 2 片，读周期 60 ns

图形控制器 TI TFP410，输出 HDMI，最高支持像素时钟频率 165 MHz

以太网控制器 Davicom DM9000A，带 PHY 与 MAC 支持，支持 10/100 Mbps 自适应

USB 控制器 Cypress SL811HS，最高支持 USB 1.1 Full Speed(12 Mbps)

1.4.2　软件平台

本项目使用 GitLab-CI 进行自动化集成和测试，借助 Docker 保证运行结果可复现。

开发 IDE Xilinx Vivado 2018.1 Web Edition

CI 系统 Ubuntu 18.04.1

文档编译 Tex Live 20180825

编译器套件 cross-mipsel-linux-gnu-binutils 2.29-1, cross-mipsel-linux-gnu-gcc 8.2.0-1 (AUR)

1.5　参考资料

本项目的设计、开发过程需要参考包括且不限于下面列出的书籍、文献和资料：

- 计算机组成与设计*: 硬件/*软件接口. David A.Patterson
- *See MIPS Run Linux.* Dominic Sweetman
- 自己动手写 *CPU.* 雷思磊
- *MIPS® Architecture For Programmers I, II, III.* Imagination Technologies LTD.
- *Vivado* 使用误区与进阶. Ally Zhou
- *32-bits MIPS CPU* 设计文档. 谢磊，李北辰
- 各外设使用手册. 相关厂商

第二章　CPU 部分

2.1　流水线结构

2.1.1　基本单发射结构

流水线可以将一条指令的指令拆分成多个较小的步骤，每个步骤都可以按照更高的频率运行从而能够提高 CPU 的最终运行频率。通常可以将指令的执行划分成为 5 级流水线

- **取指（IF）**：从内存中读取需要执行的指令。
- **译码（ID）**：将指令进行译码。同时读取指令所需要寄存器值，解析指令码中的立即数并进行扩展，对跳转指令给出跳转地址。
- **执行（EX）**：按照译码阶段的指令类型，给出对应结果。
- **访存（MM）**：如果需要访问内存，则在这一阶段进行。
- **回写（WB）**：将运算结果保存到对应寄存器。

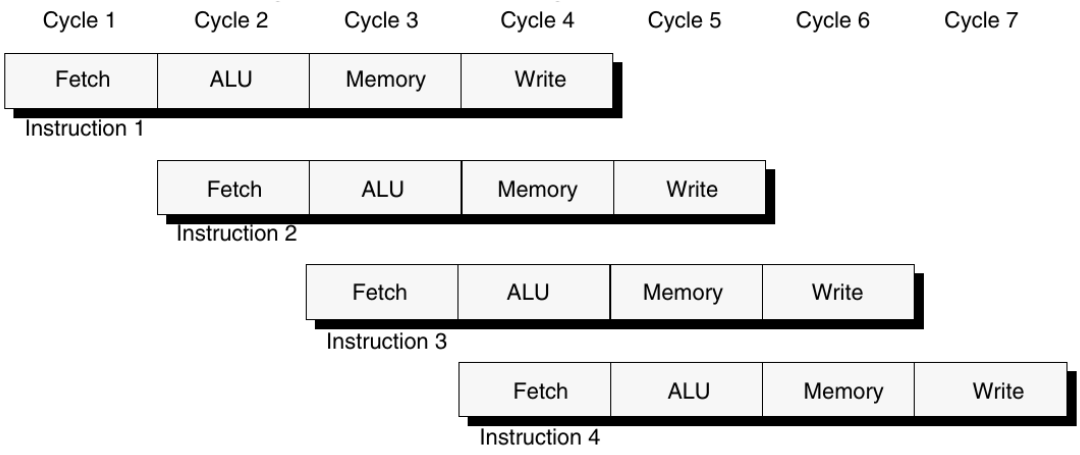


图 2.1: 标准的流水线结构，图示中没有绘制出译码阶段。

流水线结构本身在带来性能的提升的同时还会带来一部分冒险问题，有以下三种

- **结构冒险**：多条指令对同一资源进行访问。例如访存和取指同时对一个地址进行访问。
- **数据冒险**：流水线内部一条指令依赖于上一条指令的执行结果。
- **控制冒险**：在 ID 阶段才能确定跳转地址，但 IF 阶段就需要获取指令。

在 MIPS 架构中，如果按照如上五级流水线结构实现，则不会出现控制冒险。因为对于跳转指令，其下一条指令无论跳转与否均会执行，这样 IF 阶段获取的指令刚好能够继续指令。

对于数据冒险有两个方法进行解决：

- **数据旁路**：将计算结果直接送到需要的地方。比如将 EX 阶段的结果直接送到 ID 阶段。
- **流水线暂停**：插入空指令，暂停流水线的运行。

2.1.2　双发射流水线的设计

双发射是指在一个时钟周期内可以执行两条的 CPU 指令，更一般地，在一个周期同时内执行多条 CPU 指令的技术叫做超标量技术，其流水线结构如图2.2所示。双发射能够有效地提高 CPU 的效率，但是它同时也对流水线的设计带来了一定程度的挑战。

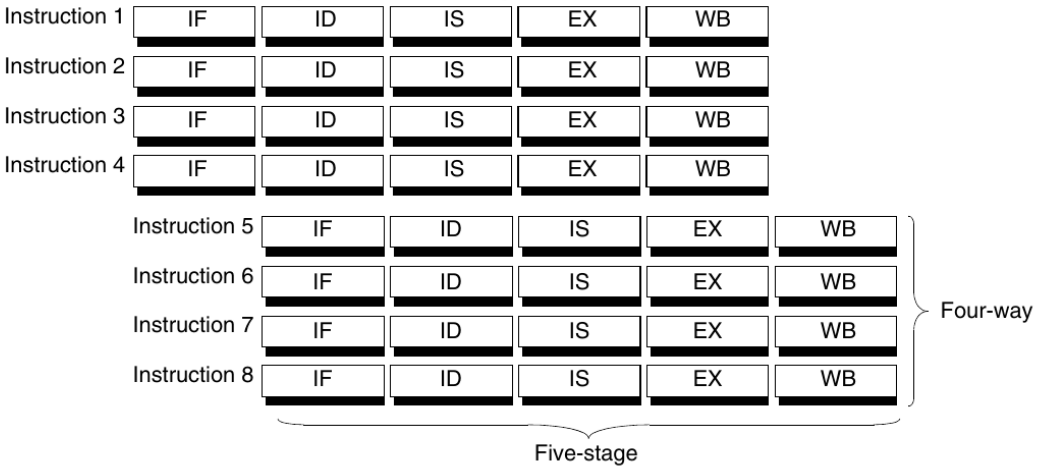


图 2.2: 超标量流水线结构。

在我们的设计中，对于大部分的指令组合来说流水线是可以两条同时执行的。其结构的设计与标准单发射流水线的主要不同在于 IF 和 ID 阶段对指令发射的控制，具体来说有这样两点

1. 在 IF 阶段对 PC 的修改要保证接下来一个阶段有两条新的指令。因为在取指完成后无法判断这两条指令是否能够一起发射，因此对 PC 的修改要考虑的是最好情况，也就是它们能够一起发射，但是到了 ID 阶段，如果这两条指令没有一起发射，那么 PC 就相当于超前了，这时候就需要针对各个情况进行不同处理。
2. 在 ID 阶段译码完成后，需要决定当前的两条指令是否可以同时发射。需要判断的情况较多，例如两条算术指令是否有数据相关，是否是 SSNOP 这样的特殊指令，是否是分支指令等。

根据 ID 阶段对何种指令可以同时发射的控制，之后 EX 和 MEM 阶段的处理也会有不同之处。这部分内容在我们叙述完这两个部分之后再进行讨论。

在此之前，我们假设从总线获取的指令数据宽度是 64 位，也就是一次给出一个物理地址，获取 64 位的数据。

指令发射控制

如上所述，在译码阶段需要控制指令是否能够同时发射。如果可以同时发射，那么就把两条指令送给下一阶段，如果判定第二条指令不能够发射，则将其设置为 NOP，并且保持第一条不动送给下一阶段。之后执行阶段就可以无需考虑当前两条指令是否是同时发射。

下面是我们实现中**不能**够同时发射的指令组合，我们称同时发射的两条指令分别为 A 和 B

- **Superscalar No Operation** 这是一条类似 NOP 的指令，其指令代码是 SSNOP。在 MIPS 规范中该条指令必须单独占用一个时钟周期，不能够和其它指令一起发射。
- **条件移动指令的数据相关** 若指令 B 为条件移动指令，并且 A 和 B 有数据相关，即指令 B 需要读取指令 A 需要写的寄存器。由于我们在 ID 阶段决定寄存器的写信号，需要寄存器的值，而指令 B 需要的寄存器值在 EX 阶段才能够获取，因此不同时发射。
- **多周期指令的数据相关** 对于多周期指令（例如 MADDU 和 FPU 指令）如果指令 A 和 B 有数据相关，那么不能同时发射。注意对于单周期指令，无论是否有数据相关，我们都可以同时发射。
- **分支、特权指令** 出于实现方便，指令 B 不能为分支或者 CP0 指令。
- **延迟槽指令** 若指令 A 为延迟槽，那么下一条指令不能与其一起发射。
- **内存相关指令** 由于数据总线仅一条，我们只支持 A 和 B 中最多有一条访存指令。
- **TLB 边界指令** 如果虚拟地址在 TLB 边界，其对应的物理地址可能是不连续的，因此此类指令不能够同时发射。因为获取的第二条指令可能无效。这一条是较为特殊的，对之后指令获取的控制部分有较大的影响。

指令获取控制

指令获取的控制较为复杂，影响该部分的因素主要有二：总线送来的两条物理地址连续的指令是否在虚拟地址上是连续的；上一个周期的两条指令是否同时发射。

预测的原则是需要寻找一个方法，保证在 IF 阶段对 PC 的增量进行决策，使得无论当前两条指令是否在 ID 阶段同时发射，在下一个周期读取得到的两条指令都能够保证双发射的需求。这部分的实现主要是一个有 3 个不同状态的状态机。

该状态机的状态的意义及其对应的转移如下，在每次转移我们可以得到的信息是 ID 阶段两条指令是否同时发射，以及它们是否在虚拟地址上是连续的（对应跨 TLB 边界的情况）。对应于当前 PC，上一周期的 PC 和下一周期应该设置的 PC 值的 8 种可能如图2.3。

- `HARD_SET` 这表示 PC 是刚刚经过初始化，或者通过一条转移指令或异常刷新设置而来的，而不是通过 PC 自身自增得到。
- `MATCH` 表示 PC 寄存器没有超前于指令的执行，对应于图2.3的 A3、A4、B3 和 B4。
- `AHEAD` 表示 PC 寄存器超前于指令的执行，对应于图2.3的 A1、A2、B1 和 B2。

状态的转移从图2.3基本上可以直接看出。对于情况 A1，由于在 ID 阶段只发射了一条指令，下一阶段如果只发射一条指令，PC 会到达当前中间的位置，如果发射两条指令第三个位置，保守地考虑，我们只能对 PC 增加 4。否则如果增加 8 并且之后只发射一条指令

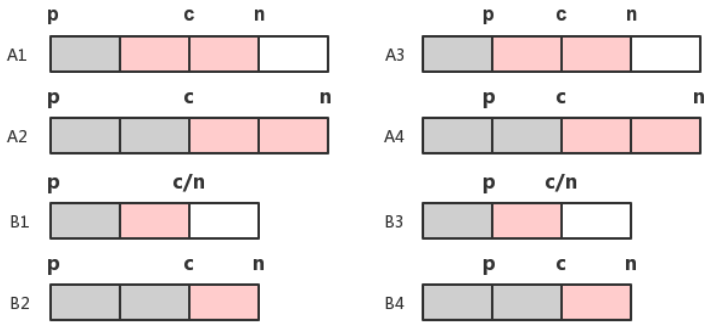


图 2.3: 指令获取控制中可能出现的 8 种情况。p 表示上一周期 PC 的位置，c 表示当前 PC 位置，n 表示下一周期 PC 应该在的位置。灰色部分表示 ID 发射的指令，红色部分是我们认为可能的接下来一个 ID 阶段最大发射指令数。c 指向的位置右侧两格表示当前 IF 阶段读取的两条指令。标号为 A 的四个情况表示当前 IF 阶段读取到的两条指令没有跨 TLB 边界，B 部分的四个情况表示读取的第二条指令可能不可用，即可能跨 TLB 边界使得物理地址不连续。

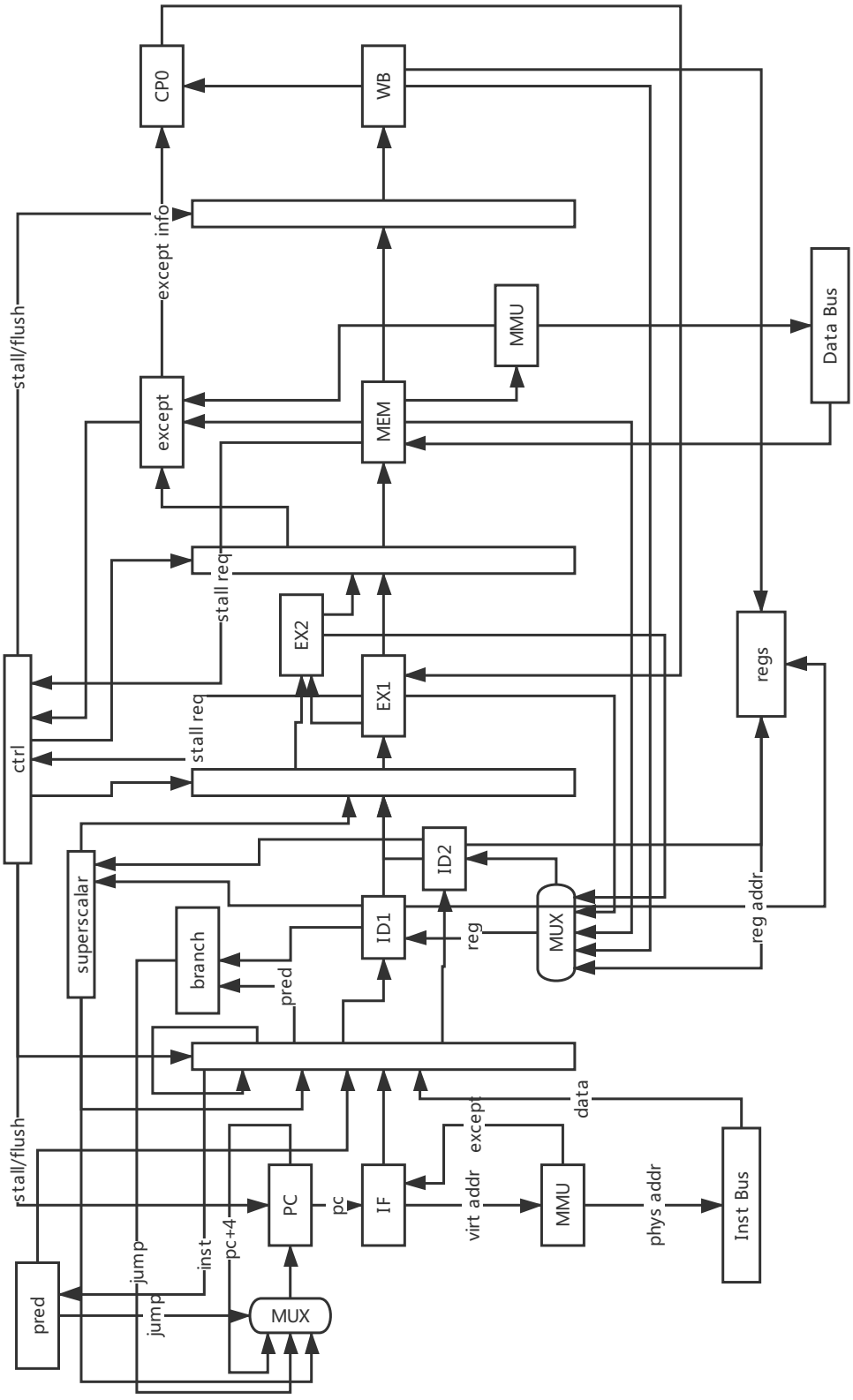
那么下一个阶段的状态 PC 会领先两条真实执行的指令。已经确定 PC 增加 4 后，状态转移自然是当前发射了一条指令那么就转移到 AHEAD，发射了两条就转移到 MATCH。

对于 B2 情况，当前最多只有一条指令可用，当然之后最多只能发射一条指令，PC 也就只能自增 4。再考虑 B1，实际上之后最多可以发射两条指令，但是我们为了方便认为之后也最多只能发射一条。那么情况此时 PC 应该保持不动。

指令的执行和“数据下推”

在不考虑数据相关的情况下，指令的执行只需要简单地生成出两个相同的 ALU 即可。由于我们的设计中，有数据相关的单周期指令也可以同时发射，这样需要进行额外的处理来解决这个相关问题。我们解决的方法类似于“数据前推”，具体的实现是将指令 A 的 ALU 计算结果直接通过组合逻辑送到指令 B 的 ALU，再根据情况进行选择。这种实现我们称为“数据下推”。

另外，在访存阶段，由于双发射控制中已经排除了两条指令同时访存的情况，这阶段只需要判断两条指令是否有其一访存再进行对应的处理即可。


CP0
WB
except info
stall/flush
MMU
Data Bus
except
MEM
stall req
EX2
regs
EX1
ctrl
stall req
ID2
superscalar
branch
ID1
reg
MUX
reg addr
pred
stall/flush
data
except
PC
pc
IF
virt addr
MMU
phys addr
Inst Bus
pred
jump
jump
inst
pc+4
MUX

2.2 指令集

下方按照功能划分列举了 CPU 所支持的 MIPS 指令，各条指令的具体编码以及功能在 MIPS 文档中有详细的描述。

- **自陷指令** TGE, TEGU, TLT, TLTU, TEQ, TNE, TGEI, TGEIU, TLTI, TLTIU, TEQI, TNEI
- **分支指令** BLTZ, BGEZ, BLTZAL, BGEZAL, BEQ, BNE, BLEZ, BGTZ, JR, JALR, J, JAL
- **逻辑指令** AND, OR, XOR, ANDI, ORI, XORI, NOR, SLL, SRL, SRA, SLLV, SRLV, SRAV
- **算术指令** ADD, ADDU, SUB, SUBU, ADDI, ADDIU, MUL, MULT, MULTU, DIV, DIVU, MADD, MADDU, MSUB, MSUBU, CLO, CLZ
- **访存指令** SB, SH, SW, SWL, SWR, LB, LH, LWL, LWR, LW, LBU, LHU, LL, SC
- **特权指令** SYSCALL, BREAK, TLBR, TLBWI, TLBWR, TLBP, ERET, MTC0, MFC0
- **条件移动指令** SLT, SLTU, SLTI, SLTIU, MOVN, MOVZ
- **无条件移动指令** LUI, SLT, SLTU, MFHI, MFLO, MTHI, MTLO

2.3 协处理器 0

CP0 是 MIPS 规范中必要的一个协处理器，它提供了操作系统所必须的功能抽象，例如异常处理、内存管理和资源访问控制等。

在 CP0 中有多个 32 位寄存器，各个寄存器均通过 MTC0 和 MFC0 读写。另外，诸如 TLBWI、TLBWR 和 TLBP 等特权指令还有异常的发生也有可能会影响其值。

表2.1中列出了必须实现的 CP0 寄存器。

编号	名称	功能
8	BadVAddr	最近发生的与地址相关的异常所对应的地址
9	Count	计数器
11	Compare	计时中断控制器
12	Status	处理器状态及控制
13	Cause	上一次异常的原因
14	EPC	上一次异常发生的地址
15	PEId	处理器版本和标识符
16	Config	处理器配置
30	ErrorEPC	上一次异常发生的地址

表 2.1: 必要的 CP0 寄存器

为了实现 TLB MMU 的功能，还需要表2.2中所列出的寄存器。

编号	名称	功能
0	Index	TLB 数组的索引
1	Random	随机数
2	EntryLo0	TLB 项的低位
3	EntryLo1	TLB 项的低位
4	Context	指向内存中页表入口的指针
5	PageMask	控制 TLB 的虚拟页大小
6	Wired	控制 TLB 中固定的页数
10	EntryHi	TLB 项的高位

表 2.2: MMU 所需要的 CP0 寄存器

同时，为了支持自定义异常向量，还需要额外实现一个 MIPS 32 Rev 2 中的 EBase 寄存器。

2.4 中断和异常

2.4.1 中断

MIPS 的中断一共有 8 个，从 0 开始编号。其中 0 号中断和 1 号中断是软件中断，由软件设置 Cause 寄存器中的对应位来触发。其余 6 个中断为硬件中断，由外部硬件触发。在实现中，由 Count/Compare 寄存器组合而成的定时中断的中断号为 7。

在我们的实现里，中断信号产生后会在 IF 阶段捕获，沿着流水线走到 MEM 阶段后，如果该中断满足触发条件则会触发异常并且进入中断处理程序。中断的触发要求如下条件全部满足：

1. `Status` 寄存器中对应的中断被打开；
2. 全局中断使能，即 `Status.IE` 为 1；
3. 当前不在异常处理程序中，即 `Status.EXL` 和 `Status.ERL` 为 0。

2.4.2 异常

MIPS 的异常是“精确异常”，也就是在异常发生前的指令都会执行完毕，异常发生之后的指令不会继续执行。在异常发生时，CPU 会跳转到对应的异常向量处执行异常处理代码并设置 CP0 中对应的寄存器记录异常的原因和一些额外的信息，同时还会进入 Kernel Mode。处理异常代码的异常向量由“基地址 + 偏移”来决定，偏移是根据异常来确定的，基地址是由 CP0 的 EBase 寄存器决定。

在我们的实现中，在流水的各个阶段产生了异常，不会马上触发，而是被记录下来，顺着流水直到 MEM 阶段完成后再进行触发。同时，还会考虑双发射两条流水线，优先触发第一条流水线的异常。特别地，异常返回指令 ERET 被实现为一种特殊的异常，保留到 MEM 阶段触发。

需要支持的异常在表2.3中列出。

异常简称	异常说明
Int	中断
Mod	TLB 修改异常
TLBL	TLB Load 异常
TLBS	TLB Store 异常
AdEL	地址 Load 异常
AdES	地址 Store 异常
Sys	系统调用
Bp	断点
RI	保留指令
CpU	协处理器不可用
Ov	算术溢出
Tr	自陷异常

表 2.3: 主要支持的异常

2.5 内存管理

MIPS 为操作系统的内存管理提供了较为简单的支持，虚拟地址通过 MMU 转换为物理地址。MIPS 标准对虚拟地址和物理地址的映射如表2.4所示。

段	虚拟地址	权限	物理地址
kseg3/ksseg	0xC000000-0xFFFFFFFF	Kernel	由 TLB 转换
kseg1	0xA0000000-0xBFFFFFFF	Kernel	0x00000000-0x1FFFFFFF
kseg0	0x80000000-0x9FFFFFFF	Kernel	0x00000000-0x1FFFFFFF
useg	0x00000000-0x7FFFFFFF	User	由 TLB 转换

表 2.4: 虚拟地址空间

具体的地址转换由 TLB 来完成，TLB 可以认为是在 CPU 内部的地址转换表的高速缓存。具体的内容需要由操作系统来进行填充。如果在 TLB 中没有找到对应虚拟地址则会触发一个 TLB miss 异常，操作系统对该异常处理，并且将对应转换表填入 TLB 中的某一项来完成对地址的转换。我们一共实现了 16 个 TLB 项。

2.6 静态分支预测

静态分支预测采用的方案是

- 当是无条件跳转指令并且跳转目的地址不存在寄存器中时，认为需要跳转；
- 当跳转指令的跳转目标地址小于当前指令地址时，认为需要跳转。这主要是考虑了循环语句；
- 当跳转指令的跳转目标地址大于当前指令地址时，认为不需要跳转。这主要考虑了条件语句等。

具体的实现方法是在 IF 阶段进行简单的判断，如果是分支语句则进行预测控制 PC 的跳转。并且将预测结果往下传入 ID 阶段，ID 阶段再根据预测结果和计算出的该指令的跳转与否进行判断来决定预测是否成功。如果预测失败则需要清空 IF 阶段的流水，并且将 PC 设置到真实的分支地址。

因此，最终 PC 的控制是根据 IF 阶段分支预测单元、ID 阶段分支的反馈信息、双发射控制信息以及上一次的 PC 地址综合起来计算得出。

2.7 浮点运算单元

MIPS 规范中，浮点运算单元（FPU）对应于协处理器 1。其共有 32 个浮点寄存器，由于我们仅支持单精度浮点运算指令，浮点寄存器被实现为 32 位。浮点数按照 IEEE 754 标准进行存储。另外，FPU 还有一部分特殊的控制寄存器，用于存储 FPU 的状态信息。其中较为重要的是 FPU 的比较运算的结果会作为标志位存储在 FCSR 寄存器的 FCC 部分中，浮点分支指令会从这个段读取标识位并且进行分支判断。

FPU 和 CPU 之间的数据交互主要有如下几种：

- LWC1 指令内存中加载数据，SWC1 将数据写入内存中；
- MFC1 指令从 CPU 的通用寄存器加载数据；MTC1 指令将数据写入 CPU 的通用寄存器；
- 浮点分支指令会使得 CPU 读取 `FCSR.FCC` 部分的内容。

在我们的实现里 FPU 直接作为 CPU 的较为独立的一部分，FPU 的流水线就是 CPU 的流水线。FPU 和 CPU 的数据交互中 MFC1 指令需要 2 个周期来完成，这是为了时序的考虑，否则因为数据前推的原因可能会使得组合逻辑过长进而导致 CPU 频率降低。

另外，我们的 FPU 也可以双发射，但是与 CPU 不同的是 FPU 的算术指令的双发射只能在两条指令内有数据相关的情况下进行。

我们实现的浮点指令主要有：

- **分支指令** C.cond、BC1F、BC1T
- **数据指令** LWC1、SWC1、MFC1、MTC1、CFC1、CTC1
- **算术指令** ABS、ADD、SUB、SQRT、NEG、MUL、DIV
- **舍入指令** ROUND、CEIL、FLOOR、TRUNC、CVT.w、CVT.s

这些指令的具体意义参考 MIPS 文档。所有算术指令以及 CVT.w 均按照 ROUND 模式进行舍入。

2.8 代码结构

CPU 部分的代码主要在 src/cpu 目录下，其主要模块见表2.5。

同时，由于在流水线各个阶段之中传递的信号太多，我们利用 SystemVerilog 的 packed 特性，将公共的信号包装成为一个结构以便于能够简化代码的实现。

cpu_defs.svh	CPU 所需常量、结构定义
trivial_mips.sv	CPU 顶层模块
ctrl.sv	流水线暂停和刷新的控制信号
hilo.sv	HILO 寄存器模块
ll_bit.sv	LLBit 寄存器模块
regs.sv	CPU 通用寄存器模块
fpu_regs.sv	FPU 寄存器模块
except.sv	异常请求处理模块
if/cpu_if.sv	IF 阶段处理模块
if/reg_pc.sv	PC 控制模块
if/if_id.sv	IF/ID 数据传输模块
id/cpu_id.sv	ID 阶段处理模块
id/id_type_i/j/r.sv	ID 阶段 I/J/R 型指令译码模块
id/fpu_id.sv	FPU 译码模块
id/branch.sv	分支处理模块
id/superscalar_ctrl.sv	双发射控制模块
id/fcsr_mux.sv	FPU 控制寄存器选择模块
id/id_ex.sv	ID/EX 数据传输模块
ex/cpu_ex.sv	单周期指令执行模块
ex/ex_multi_cycle.sv	多周期指令执行模块
ex/ex_count_bit.sv	CLO/CLZ 指令执行模块
ex/div_uu.v	DIV 指令执行模块
ex/fpu_ex.sv	FPU 指令执行模块
ex/fpu_float2int.sv	浮点转整数指令执行模块
ex/ex_mem.sv	ID/EX 数据传输模块
mem/cpu_mem.sv	MEM 阶段处理模块
mem/mem_wb.sv	MEM/WB 数据传输模块
wb/cpu_wb.sv	WB 阶段处理模块
cp0/cp0.sv	CP0 模块
cp0/lfsr.sv	随机数发生器

表 2.5: CPU 部分代码结构。

第三章　外设部分

3.1　地址分配与总线

表3.1列举了数据和地址总线上各个设备的物理地址空间分配。为了使解码和请求的分发更方便，每一个设备都有 16MB 的地址空间，但其只会用到其中的一部分。下面各节将给出各个设备的详细行为描述。

名称	起始地址	结束地址	有效大小	类型
SRAM	0x00000000	0x007FFFFF	8 MB	存储
Flash	0x01000000	0x017FFFFF	8 MB	存储
Graphics	0x02000000	0x02075300	480004 B	混合
UART	0x03000000	0x03000000	2 地址	寄存器
Timer	0x04000000	0x04000000	2 地址	寄存器
Ethernet	0x05000000	0x05000000	2 地址	寄存器
GPIO	0x06000000	0x06000000	3 地址	寄存器
USB	0x07000000	0x07000000	2 地址	寄存器
Bootrom	0x1FC00000	0x1FC03FFF	16 KB	存储

表 3.1: 各外设的物理地址分配

CPU 向总线传递的所有地址都必须按 4 对齐，也就是说最后两个二进制位必需是 0，否则将会触发一个 “地址错误” 异常。如果尝试访问超出任何设备 “有效大小” 之外的地址，或者向声明为只读的地址写数据，后果将是不可预料的。

本项目计划采用哈佛架构，因此分指令与数据两条总线。其中指令总线主设备为 CPU，从设备为 Bootrom 与 SRAM；数据总线主设备为 CPU，从设备为除 Bootrom 外所有外设。两条总线的工作机制都是一致的，即将 CPU 的读/写指令通过前缀匹配连接到相应设备控制器，并将其回复连接到 CPU。由于此类简单总线不至引起冲突，总线地址分配也互不重合，因此不需要仲裁等进阶功能。

3.2　存储设备

SRAM、Flash 和 Bootrom 都是存储设备，其中 Bootrom 是只读的，SRAM 是易失的，Flash 则是非易失的。出于性能考虑，SRAM 和 Bootrom 都需要能保证在下一个时钟周期上升沿给出当前访存指令的请求结果，而 Flash 则需要较长的时间进行读写（尤其是写）。为了使 CPU 得到正确的结果，相关的设备控制模块需要拉高总线的 `stall` 信号指示 CPU 暂停流水线等待访存结束，直到在下一个上升沿能给出响应为止。而由于 SRAM 被

同时挂接在两条总线上，它需要有能力在一个 CPU 周期内处理两次请求，或者在确实无法完成时正确地给出 `stall` 信号。

SRAM 芯片的每个物理地址都存储一个字（32 bit），所以只需要将地址高 30 位直接传送给它。而 Flash 的每个物理地址存储的是半字（16 bit），所以在进行读时，控制器内部需要事实上进行两次连续的读操作后将结果拼接起来；在进行写时，Flash 芯片只要求半字写，无需进行特殊处理。Bootrom 由一个 Xilinx IP 核生成，每个地址对应一个字。

在物理接口上，Flash 芯片、SRAM 芯片与 Bootrom 所使用的 Xilinx IP 核都遵循了所谓的 “SRAM 接口”，即有片选（CE）、读使能（OE）、写使能（WE）、地址（Address）和数据（Data）信号，其中除数据线为双向（IP 核拆分为出入两条），其余均为 FPGA 需要给出的。三者对读写过程都分别有不同的时序要求，需要根据各自的数据手册正确实现读写时序。此外，Flash 芯片还有一些额外的信号，如写保护、重置等，也需要正确处理。

3.3　UART 串口

UART 是一种无状态协议，实验板上为其预留了 TX 与 RX 两条信号线。本项目将使用开源的串口组件作为底层的收发器，串口控制器共向外暴露两个寄存器地址。第一个寄存器（`0x03000000`）是只读的，最低位指示 CTS (Clear To Send) 信号，表示可以发送数据；次低位是 DR (Data Ready) 信号，表示有未读取的数据。第二个寄存器（`0x03000004`）是可读写的，当 CTS 信号为高，写操作能向 TX 信号线上发送一个 Byte（高位被忽略）；当 DR 信号为高，读操作能够获得一个 RX 信号线上传来的 Byte。

UART 控制器模块使用 115200 的标准波特率，1 个停止位，无校验位。读、写两端都应当有足够大的 FIFO 缓冲区（如 4K）来保证读/写不会因为发送过快或一段时间没有取走而产生非预期结果。只要有数据没有被取走，控制器就应当保持拉高一个外部设备中断。

3.4　显示控制器

表中控制器的类型是 “混合”，是由于它由 240000 字节的图形缓冲区（framebuffer）和一个处于末尾的（0x2075300）配置寄存器组成。在图形缓冲区中，能存储 800*600 像素的图像，每个像素占据 8 比特，格式为 `{RED[2:0], GREEN[2:0], BLUE[1:0]}`。控制寄存器是可读写的，用来指示图形缓冲区中第一个像素的偏移量，这可以帮助操作系统渲染终端等画面时方便地实现滚动功能。

本项目将把缓冲区中的内容，以 `800*600@75Hz` 的画面格式，借助 TFP 410 芯片产生 DVI 信号，通过 HDMI 接口输出。为此，需要借助像素时钟（50MHz），正确地产生符合 VGA 标准的行、场同步信号，从缓冲区读取并输出相应颜色数据。除此之外，还要正确借助控制寄存器来实现含偏移的渲染。

3.5　GPIO 控制器

GPIO 控制器包含三个寄存器地址，第一个（`0x06000000`）是只读的拨码开关状态，第二个（`0x06000004`）和第三个（`0x06000008`）是可读写的，分别代表七段数码管和 LED 的显示。对应实验板上这三种设备的数量，这些寄存器都只有低 16 位是有意义的。特别地，当 `0x060004` 地址的最高为被置于 0 时，七段数码管的译码功能被启用，只有最低八位是

有效且会被显示的；如果被置于 1，则译码停用，七段数码管共有 16 个笔画（包括两个小数点），恰好能够显示一个 16 位整数。

3.6　定时器

定时器模块的第一个寄存器（`0x04000000`）包含了一个每 1 微妙自动递增的整数，并可以被修改为任意值。它可以被用来不受实际 CPU 频率影响地计量一些指令执行的时间。第二个寄存器（`0x04000004`）包含了 CPU 主时钟的周期计数。这两个寄存器都会在硬重置时被归零。

3.7　以太网与 USB 控制器

作为额外的需求，以太网与 USB 控制器的硬件逻辑实现并不复杂。它们的硬件同样基于 SRAM 接口，因此读写可以复用部分 Flash 与 SRAM 控制器的代码。但需要注意的是二者对时序也另有各自的要求，以太网控制器一般可以在 30ns 内完成一个请求，而 USB 控制器则需要较长的等待。二者暴露的第一个寄存器都是可写的地址选择寄存器（SL811 使用了最低 8 位，而 DM9000A 甚至只是用了最低位），第二个寄存器是可读写的数据寄存器（SL811 最低 8 位有效，DM9000A 最低 16 位有效）。需要特别注意的书，对于同一个地址的多次，DM9000A 可能给出不同的结果，这也反映在对其数据寄存器的读取结果上。

这两个部件的操作和通信都较复杂，都需要操作系统中驱动程序的配合才能进行工作。USB 控制器与以太网控制器都有硬件的中断信号输入，项目中需要将它们同步后直接连接到 CPU 的硬件中断端口，以使得操作系统正常处理来自外部硬件的数据和请求。

第四章　Bootloader 部分

4.1　第一阶段：TrivialBootloader

作为板上系统的一部分，Bootrom 中包含了系统每次上电或重置时都会首先执行的代码，起始物理地址为 `0x1FC00000`。由于这部分程序是固化在 FPGA 中的，为了节约有限的板载存储，Bootrom 中的代码不能太大（暂定为 16KB）。因此，有必要撰写一个较小的 Bootloader 来进行初步的系统初始化和加载工作，将其命名为 TrivialBootloader。

本项目计划用 C++ 与汇编语言实现 bootloader。其中汇编部分需要有下列功能：

- 系统的全局初始化，设置 `sp`，`gp` 寄存器，跳转到 C++ 代码入口
- C++ 代码退出后的清理与提示
- 启用异常处理并设置异常向量

C++ 部分是 Bootloader 的主体，其应当具有的功能为：

内存检测 通过不同块大小随机读写检测内存硬件与实现是否存在问题

ELF 启动 支持从非易失存储（Flash）中读取合法的 ELF 文件头，正确地将其复制到内存的相应位置并跳转

直接启动 支持直接跳转到内存入口点（`0x8000000`）启动，便于系统移植时的调试工作

串口旁加载 支持从串口直接向内存中加载数据和指令，并跳转到指定的位置启动

内存转储 支持将指定的内存区域的数据转储到串口输出

异常处理 正确处理各种操作异常、非法情况（如没有选择启动模式、内存检测失败、要复制到内存的数据覆盖了 Bootloader 本身的代码），在发生异常时通过串口、LED 等多种途径给出友好可读的提示

4.2　第二阶段：U-Boot

4.2.1　背景

U-Boot 是一个启动引导程序，常见于嵌入式系统中，用于引导 Linux 等操作系统。通过运行 U-Boot 引导程序，可以支持从 Flash、U 盘、网络等来源加载 uCore、Linux 系统镜像到内存并进行引导。由于 U-Boot 本身有较强的命令行功能和交互能力，它也可以作为一个硬件测试与演示的工具。在本系统的设计中，U-Boot 将作为二级引导程序，放置在 Flash 中，被 TrivialBootloader 所加载。

4.2.2 硬件需求

U-Boot 对 CPU 的功能要求较低，它不使用 MIPS 的中断和 TLB 机制，因此硬件可以不需要实现这些机制。对于其它的异常，仅仅在程序运行不正常时才会发生，如果假定程序能正常运行，对异常处理也没有要求。

作为功能丰富的引导程序，其对外设的要求很高，将用到表3.1中列出的除了图形设备之外的所有外设（其中定时器用于运行 Dhrystone 等性能测试时的准确时间评估）。其中，USB 与以太网模块的正常工作是至关重要的，否则片上系统将失去从外部加载系统的功能。

4.2.3 编译方法

U-Boot 可以直接使用1.4.2节中给出的编译器套件进行编译和调试。具体地，只需要运行下列命令：

- `make CROSS_COMPILE=mipsel-linux-gnu- trivialmips_thinpad_defconfig`
- `make CROSS_COMPILE=mipsel-linux-gnu-`

4.2.4 移植目标

U-Boot 与 Linux 内核源码的组织架构类似，也都采用了 `DTS`（设备树源码）来描述设备，因此移植方法也比较类似。主要的移植工作主要分为两部分，一部分是添加 CPU 相关的 SoC 支持，一部分是添加板级支持。

由于之前的类似项目已经有了完成度较高的工作[1]，本项目计划在其基础上进一步适配。主要的工作有：

- 添加串口控制器驱动（包括系统启动早期和后期两个部分）
- 添加定时器驱动，准确反映运行时间
- 修改设备描述以准确反映 SoC 片上资源，修复驱动错误（包括 SL811 USB 模块和 DM9000A 网卡驱动）

[1] `https://github.com/z4yx/u-boot-naivemips/`

第五章　操作系统部分

本组计划在板上系统上运行 uCore 操作系统，基本规划为：对 SOC、操作系统进行调试使 uCore-thumips 系统运行正常，针对平台进行 MMU、外设等方面等相关实现和测试；对 uCore Plus 操作系统进行扩展，完善其对 MIPS 32 平台的支持，同时达到在板上正常运行的目标。

5.1　uCore-thumips 操作系统的移植与运行

5.1.1　系统概述

uCore-thumips[1] 是针对简化后的 MIPS 32 实现：MIPS32S 平台的 uCore 移植版本。该项目针对 MIPS32S 平台实现了对应的 Bootloader、初始化流程、异常处理、内存管理和上下文切换流程。相比标准的 MIPS 32，MIPS32S 缺少部分指令且不支持延迟槽。针对这些不同，uCore-thumips 对 uCore 操作系统的编译选项进行了相应的修改，并提供了额外的库函数实现缺失的指令（如 `divu`）的功能。

5.1.2　编译方法

在非 mipsel 平台编译、调试 uCore-thumips 需要使用面向 mipsel 架构的交叉编译、调试工具链，所需工具主要包括 binutils、gcc 和 gdb。

Debian 系统下，`gcc-mipsel-linux-gnu` 和 `binutils-mipsel-linux-gnu` 软件包分别提供了预编译的目标平台为 mipsel 的 binutils 和 gcc。其它操作系统的工具链可参考 LinuxMIPS 项目文档[2]自行编译。此外 Sourcery CodeBench Lite[3] 提供了预编译的 mipsel 工具链。

交叉编译时，指定 `CROSS_COMPILE` 环境变量或修改 Makefile 中 `CROSS_COMPILE` 变量为所使用的交叉编译器，即可使用 make 进行编译。编译后得到镜像 `ucore-kernel-initrd` 和 `boot/loader.bin` 分别为系统内核 ELF 和 Bootloader。

进行移植时，需针对片上系统对 Makefile 中相应配置进行修改，包括延迟槽、浮点模块等编译选项、为用户 App 预留存储大小等。

5.1.3　启动流程

uCore-thumips 的引导、启动流程主要分为 Bootloader 加载系统、初始化 C 环境、初始化系统三个步骤。

[1] `https://github.com/z4yx/ucore-thumips`

[2] `https://www.linux-mips.org/wiki/Toolchains`

[3] `https://sourcery.mentor.com/GNUToolchain/release2189`

uCore-thumips 提供了简易的 Bootloader `boot/bootasm.S`，该程序从 Flash（默认为地址 0xBE000000）读取合法的 ELF 文件头，将其复制到内存的相应位置并跳转。

Bootloader 加载系统后将跳转至 `kern/init/entry.S` 中的 `kernel_entry` 过程。在此过程中，系统将重置 CP0 中异常相关寄存器、设置 TLB 相关异常向量；同时，正确设置 `sp`，`gp`，清空 `bss` 以满足 C 程序运行要求，之后跳转至 `kern/init/init.c` 中的 `kern_init` 函数。`kern_init` 函数将完成中断控制、控制台、异常、内存管理、进程管理等系统功能的初始化。

进行移植时，需将 Bootloader 替换为针对 TrivialMIPS 片上系统自行实现的 TrivialBootloader 或 U-Boot，针对平台对中断控制、控制台等功能的初始化过程进行相应修改。

5.1.4　内存管理

MIPS32 使用软件进行 TLB 缺失处理，当发生 TLB 缺失时会触发 TLB Refill 异常。uCore-thumips 已经实现了 TLB Refill 异常的处理。发生 TLB 缺失时，系统会首先检查页表判断是否为缺页，若为缺页调用 `do_pgfault` 进行处理，否则检查权限后填充 TLB 表项。

5.1.5　异常处理

异常处理程序通过访问 CP0 中的 Cause 寄存器获取异常信息，同时需要正确设置 Status 寄存器中的某些位。用户态和特权态切换时，uCore 内核使用 `trapframe` 结构存储程序运行状况。uCore-thumips 已实现和 CP0 中寄存器的交互及 `trapframe` 的保存。

5.2　uCore Plus 操作系统的移植与运行

5.2.1　系统概述

uCore Plus[4] 是 uCore 的全功能版本，且提供 i386、AMD64、ARM、MIPS 等多平台的支持。uCore Plus 被划分为不同模块，各模块可独立编译，通过平台相关的 Makefile 进行组合。为提供多平台支持，uCore Plus 的多数模块都分为架构相关部分和架构无关部分，架构相关部分代码位于各模块 `arch/$ARCH` 目录下。每个模块的架构相关代码都至少包含单独的 Makefile 和链接脚本。

5.2.2　移植评估

Bootloader

uCore Plus 的 Bootloader 模块中，架构无关代码只包括 ELF 头和一些通用结构的定义，各架构需实现对应的 Bootloader。uCore Plus 已有的 MIPS Bootloader 沿用了 uCore-thumips 的 Bootloader 的代码。移植时，可以使用 TrivialBootloader 或 U-Boot。

内核

- 驱动程序 uCore Plus 中，不同架构不共用驱动程序代码，需要为每种架构都实现整套的驱动程序。必须要实现驱动的设备包括 console 输出、输入、计时器和外部存储。移

[4] `https://github.com/oscourse-tsinghua/ucore_plus`

植时需实现 TrivialMIPS 平台对应 console、计时器和外部存储的驱动。各类设备驱动所需实现的接口见表 5.1, 5.2, 5.3。

接口	说明
`void cons_init(void)`	初始化 console
`void cons_putc(int c)`	输出字符 `c`
`int cons_getc(void)`	输入字符

表 5.1: console 驱动需实现的接口

接口	说明
`void clock_init(void)`	初始化计时器
`size_t ticks`	uCore 启动后经过的时间

表 5.2: 计时器驱动需实现的接口

接口	说明
`void ide_init(void)`	初始化外部存储
`bool ide_device_valid(unsigned short ideno)`	检查外部存储是否存在且可用
`size_t ide_device_size(unsigned short ideno)`	返回外部存储每个扇区的大小
`int ide_read_secs(...)`	读取给定扇区数的数据
`int ide_write_secs(...)`	写入给定扇区数的数据

表 5.3: 外部存储驱动需实现的接口

- 初始化 uCore Plus 已包含 uCore-thumips 中的初始化代码，初始化过程与 uCore-thumips 相同。
- 内存管理 uCore Plus 中，至少需为每个架构在 `pmm.c` 中实现 MMU 初始化和 TLB 缺失的处理、在 `vmm.c` 中实现 `copy_from_user, copy_to_user` 和 `copy_string` 三个用于在内核态和用户态之间拷贝数据的函数。uCore Plus 已包含来自 uCore-thumips 中的页表实现和 TLB 管理代码。
- 同步、系统调用及进程管理 uCore Plus 中已实现 MIPS 平台同步、系统调用及进程管理的平台相关代码，部分来自 uCore-thumips，可参考5.1节。移植时不需要太大改动。

ulib

ulib 是一个静态链接到所有用户程序的函数库，源码位于 `src/libs-user-ucore/`。其中的部分函数平台相关，移植时需要实现。相关接口及在 MIPS 平台的实现情况见表 5.4。

接口	说明	状态
initcode.S	程序入口	已实现
clone.S	克隆的进程	待实现
atomic.h	用于用户态锁的实现	已实现
syscall.c	进行系统调用	已实现

表 5.4: ulib 需实现的接口

第六章　测试与部署

作为软件工程的要求，本项目将进行自动化的集成、测试与部署，共分为硬件，软件与文档三部分。所有的流程都将通过 Docker 进行，确保是可完整重现的。

6.1　硬件部分

本项目使用的主要硬件设计语言 SystemVerilog 是一门强大的验证语言，我们将使用其编写 testbench 来测试硬件模块。主要的 testbench 也分为三个模块：

CPU 测试 本部分用于测试 CPU 实现指令的正确性。我们对 CPU 的各条指令都编写了相应的测试程序，同时还对各类可能的冲突现象、异常、TLB 的行为编写了对应的测试。测试的过程是通过一个 testbench 虚拟出外部的总线和 RAM 并且接入 CPU，同时对 CPU 中 WB 阶段对寄存器的写请求进行监视。每个测试代码对所有寄存器的写请求都会给出响应期待的结果，同时在运行 testbench 时会将监测到的真实的写请求和期望的结果进行对比进而确认程序执行的正确性。整个 CPU 的测试过程是自动化的，无需人工介入观察信号，如果发现真实的运行和期望不符会进行报告。

外设测试 本部分用于测试外设控制器的实现正确性（主要是时序）。测试程序作为 Master 挂接于总线上，其余外设控制器作为 Slave 正常连接。测试程序向每个外设分别发出不同的读写指令，硬件部分使用对应的仿真模型文件，或者直接观察信号变化，在返回结果不符合预期，或向硬件发出的指令不正确时，终止运行并反馈错误。

完整测试 本部分用于模拟整个片上系统的运行。测试时 CPU、总线与各个外设控制器正常连接，SRAM、Flash 与 Bootrom 的仿真模型中均配置所需的映像文件。启动测试后，可模拟从加载 Bootloader 到加载操作系统的全过程。

对于主分支的每一次提交，都需要进行持续集成，包括进行 Vivado 项目的编译、综合与进行上述的仿真。通常只运行 CPU 和外设测试部分，在需要时运行完整测试。

6.2　软件部分

在软件方面，本项目计划对编写的所有汇编/C/C++ 代码，移植的 Bootloader、操作系统，以及需要运行的功能测试、性能测试，均编写持续集成脚本，保证每个版本都能进行正确的、可重现的编译。进一步地，借助计原在线平台提供的 ThinPad SDK，可以在上一步硬件部分编译完成 bitstream 的基础上，将硬件设计与软件一同上传运行，在检验硬件实现正确性的同时，也可以自动化地测试其性能表现。

6.3　文档部分

最后，作为设计文档的结尾，本文档已经实现了计划中的自动集成的功能。对任意文档的每一次修改都能自动编译成对应的版本发布到 GitLab，方便开发人员与需求方的查阅。