设计报告

清华大学 1 队
唐适之、蔡子熙、刘熙航

一、设计简介

我们实现了基于 MIPS32v1 指令集的五级流水 CPU，并通过了初赛所要求的所有测试。性能方面，我们针对 AXI 接口的特性做了针对性的优化。功能方面，我们实现双核心的 CPU，且可以运行双核心版本的 Linux，并对 DDR3 RAM、SPI Flash、以太网网卡、串口外设进行了支持。

二、设计方案

（一）总体设计思路

我们的 CPU 采用了经典的五级流水线的架构：取指、译码、执行、访存、写回。对于功能测试和性能测试，CPU 的总体结构如图 1[1]；而展示包中的完整的 SoC 则含有两个对称

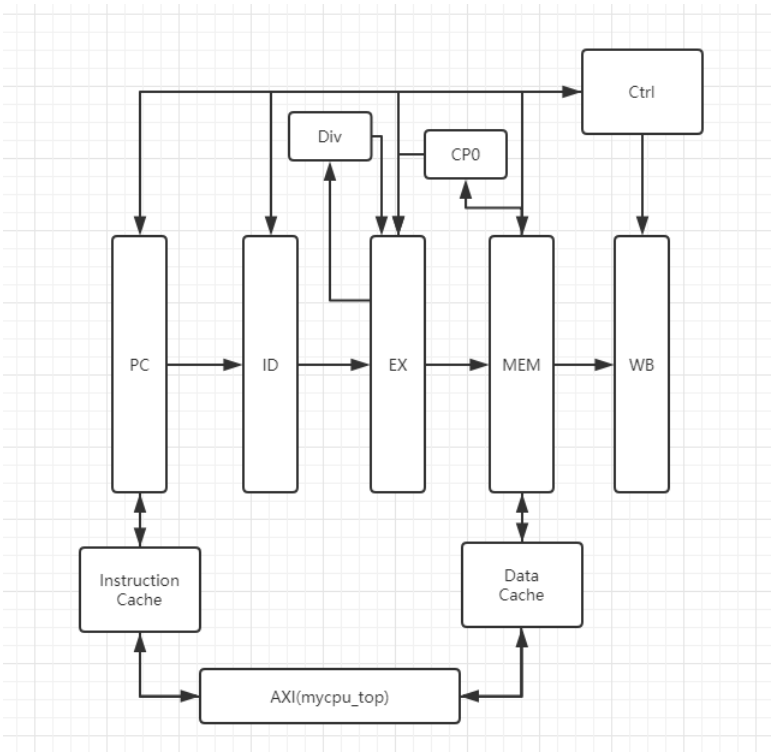

图 1

[1] 流水线中，ID、EX、MEM 每个部分实际上是两个文件，如 id 阶段对应于是 if_id.vhd 和 id.vhd 两个文件，其中一个负责在时钟上升沿传递数据，另一个则负责在周期内计算结果，但需要说明的是，虽然是两个文件，但是在 Vivado 进行综合时会将其合并为一个。而 IF 和 WB 则分别对应 pc.vhd 和 mem_wb.vhd。

的 CPU，还包含了外设控制的相关逻辑，每个 CPU 中还含有处理虚地址翻译的 MMU[2]，总体结构如图 2。

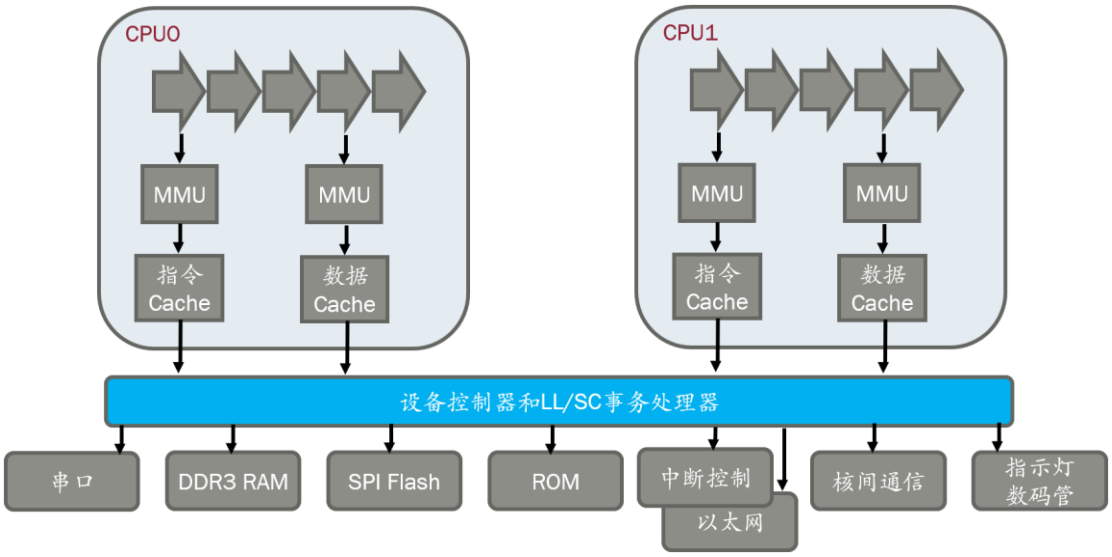

图 2

（二）取指阶段设计

由于指令缓存已经独立于取指阶段，因此取指阶段的设计是所有实体中最简单的，只需在每个周期根据当前 PC 从指令缓存请求指令，并接受译码阶段或 Ctrl 实体给出的跳转地址（分别对应于普通的跳转指令和异常）即可。当取指需要超过一个周期时，取指阶段会请求 Ctrl 实体暂停取值和译码阶段。之所以要暂停译码阶段，是为了简化跳转逻辑，以提升时钟频率。

（三）译码阶段设计

译码阶段的主要目的是对取指阶段所获取到的指令进行译码，并将结果传递给执行阶段。译码过程不仅包括判断指令码，还包括提取操作数。在我们的实现中，会先根据指令码标注各个操作数的来源，然后在寄存器、无符号立即数、有符号立即数等来源之间进行选择。一个例子如图 3：这是针对 CLO 指令的译码过程，其通过 oprSrc1 变量标注了 CLO 指令的第一个操作数来自寄存器，而通过 oprSrc2 变量标注了第二个操作数不存在。

此外，如果该指令是跳转指令，则需要在这一阶段给出跳转地址，与跳转地址相关的异常也需要在此阶段进行处理。我们不检查以立即数为目标的跳转的地址是否对齐，以缩短关键路径。

[2] 性能包和展示包中的 myCPU 是一样的，但性能包中不编译外设控制器等部分。

```
when OP_SPECIAL2 =>
    case (instFunc) is
        when FUNC_CLO =>
            oprSrc1 := REG;
            oprSrc2 := INVALID;
            alut_o <= ALU_CLO;
            toWriteReg_o <= YES;
            writeRegAddr_o <= instRd;
            isInvalid := NO;
```

图 3

我们还对译码阶段接受的数据旁路做了优化，包括：1. 仅非访存指令提供访存阶段至译码阶段的数据旁路；2. JR 和 JALR 指令不接收数据旁路，此举缩短了译码阶段的关键路径，从而提高了时钟频率。

（四）执行阶段设计

在我们的设计中，执行阶段实际上分成了两个独立的实体：EX 和 DIV，其中 EX 实体用来处理普通的指令，而 DIV 实体则用来处理除法指令。当执行某个操作需要超过一个周期时，执行阶段会请求 Ctrl 实体暂停流水线的执行及其之前的阶段。

对于访存指令，执行阶段则负责计算其访存地址，并判断是否发生了地址非对齐的异常。

（五）访存阶段设计

访存阶段首先要对非整字访存的地址，例如 LH、SB 等指令的访存地址，按字对齐。值得注意的是，访存发生异常时，CP0 的 BadVAddr 寄存器应保存指令试图访存的原始地址，而不是对齐后的地址，故原始地址不能丢弃。

访存阶段是 MIPS 五级流水线中最后一个可能产生异常的阶段，所以还担负着汇总此指令在此前所有阶段产生的异常并交予 Ctrl 实体和 CP0 的任务。MIPS 的异常响应优先级与流水线顺序一致，故当有多个异常同时发生时，应优先报告在流水线更早的阶段产生的异常。虽然外设中断不在流水线的任何一级产生，但也在访存阶段统一处理。MEM 实体会向 CP0 查询是否有活跃的外设中断，如有，按类似其他异常的方式处理。

另一点值得注意的是，外设产生异常时，处在访存阶段的可能是流水线暂停导致的空泡。如果此时产生异常，将无法记录异常返回地址。故应等访存阶段中不是空泡时再处理中断。

（六）Ctrl 实体设计

Ctrl 实体用于控制流水线，当被请求的流水线部分暂停时，ctrl 会向各级阶段寄存器发出暂停命令；当发生异常时，Ctrl 会发出清空流水线的命令。

当收到暂停请求时，假设请求暂停的是阶段 X 及其之前的所有阶段，Ctrl 实体则向阶段 X 与阶段 X+1 间的阶段寄存器，及其之前所有阶段寄存器发出暂停命令，其余阶段寄存器不受影响。

当发生异常时，Ctrl 实体向所有的阶段寄存器发出清空命令。此外，还需向取指阶段发送 PC 跳转目标，具体计算分为如下情况：

1. 若此“异常”不是真正的异常，而是当作异常处理的 ERET 指令，则向 CP0 协处理器查询 ERET 寄存器，其值即跳转目标；
2. 若为真正的异常，则向 CP0 协处理器查询当前的配置，根据配置确定选用参数中定义的异常处理向量基地址中的哪一个，以及确定是否选用参数中定义的特殊的偏移量。然后根据异常类型，最终确定异常处理向量的偏移量。异常处理向量的基地址与偏移量的和即跳转目标。

如上所述，异常处理向量的基地址和偏移量可以使用参数配置，这是为了方便地移植到不同环境，例如功能测例、μCore 和 U-Boot, Linux 等。

CP0 ERET 寄存器中记录的异常返回地址可能是无效的。在此情形下，应立即触发一个新的地址无效异常，并向 CP0 BadVAddr 寄存器写入相应的无效地址。

此外，为了通过功能测试，还有两点需要注意：1. 如果跳转指令发生异常，那么需要不清空延迟槽中的指令；2. 如果是 JR 和 JALR 发生异常，那么也不应当清空其往寄存器中写的 PC 值。

（七）CP0 实体设计

CP0 的本质是一个寄存器的集合，其用来存储 CP0 中的不同字段。CP0 应当同时支持一路读和一路写，此外还需要将某些特定的寄存器输出到其他单元，例如将 EBASE 输出到 Ctrl 中。由于对于 CP0 的操作是较为罕见的，为了提升时钟频率，我们不对 CP0 寄存器进行任何的数据旁路。

此外，CP0 实体还支持一些非通用的读写，列举如下：

对于输入而言：

1. $Cause_{IP[7:2]}$用于表示外设中断，每个时钟上升沿，这一数值都会刷新为上层给出的最新值。
2. 当发生异常时，$Cause_{ExcCode}$ 中应当记录异常原因、在 $Cause_{BD}$ 中记录是否在延迟槽中、在 $Status_{EXL}$ 中设置异常级别、在 EPC 中记录异常返回地址。发生某些异常时，

还需设置 BadVAddr、EntryHi 等寄存器。

3. 当异常返回错误地址时，Ctrl 实体可能需要写入 BadVAddr 寄存器。

对于输出而言：

1. Status、Cause、EPC 寄存器需要被 Ctrl 和 MEM 实体不断的查询，因此在我们的处理中，直接输出了这两个实体所对应的值。
2. 上层实体需要获知当前处在用户态还是内核态，故应当直接输出给上层。

（八）指令缓存设计

指令缓存是一个写穿（Write through）、直接映射的缓存。

由于我们希望可以进行针对性的优化，故在具体的实现中，将指令缓存与数据缓存分离。首先，我们对缓存层和 AXI 层之间的通信方式进行了一个规范：将缓存层分成两个部分，一个部分仅用于发送数据，而另一个部分则仅用于接收数据。换言之，发送数据的部分是不会知道数据接收的情况的。这么做的好处在于，可以让设计变得更加简洁，即可以将 cache 实体分成几乎没有相关的两个部分；但是坏处在于，当 AXI Slave 的表现并不够稳定时，会带来一定的问题。

从缓存的视角看，AXI 层与缓存层的通信方式如下：缓存层会发送 arenable_o，araddr_o 这两个信号，一旦 arenable 置为 1，那么这一请求就不可以被撤消。而当 AXI 层成功将 ar 信号发送之后，会在一个周期里将 arrequestack_i 这一信号置为 1，缓存层收到这一信号之后应当立即将 arenable 置为 0，并且在收到数据之前，不能将其重新置为 1。

而 AXI 层每次向缓存中传递数据时，会发送三个信号：enable、data 和 addr。由于我们采用了直接映射的 cache，这么做不会带来任何的问题。缓存在收到数据之后，需要更新对应的数据、tag 和 present 位。

对于指令缓存而言，由于我们希望在一次性读取的一组数据接收完之前就可以返回，故而指令缓存的每个 Cache line 只包含一个指令字。具体而言，每次缓存缺失后，缓存会向 AXI 请求连续的 16 个字，依次填入 16 个 Cache lines 中。在第一个 Cache line 被填入后，即可返回。

（九）数据缓存设计

数据缓存是一个写回（Write back）、直接映射的缓存。

相较于指令缓存而言，数据缓存的逻辑更为复杂，主要体现在以下的几个方面：

1. 指令缓存所接收到的所有地址都在 RAM 中，所以是支持成组读写的，而数据缓存并不满足这一点。事实上，不支持成组读写的地址都位于 Uncached 段，让访存这些地址的请求直接绕过缓存即可。
2. 数据缓存采用的是写回而非写穿机制。

数据缓存和 AXI 的通信协议相对指令缓存而言增加了对五个信号的控制：arenable, araddr, awenable, awaddr, awdata，arrequestack, awrequestack。这么做是为了同时进行读写操

作。

不同于指令缓存，虽然数据缓存也进行 16 个字一组的访存，但其采用了大小为 16 个字的 Cache line。只有当一个 line 完全就绪之后，才会开始对这一个 line 进行操作。

写回时，需要先将 Cache line 里的内容写出，再从 AXI 层读取新的内容放入原来的 Cache line，但这样需要先后进行两个请求。为了提升性能，我们先将要写入 AXI 的内容存进一个 Write buffer 中，使读和写互不干扰，然后同时进行读写，使写回这一过程更加高效。

（十）mycpu_top.vhd 对于 AXI 的操作

此文件中主要实现的是与指令缓存和数据缓存完成前述的通信机制，并将请求正确地发送到 AXI 总线中去。这里涉及到几个问题：

1. 指令缓存和数据缓存的优先级问题。如果指令缓存和数据缓存同时发送了读请求，那么优先发送来自数据缓存的请求。否则，访存阶段暂停后，取值阶段也无法工作，导致系统陷入无穷等待。

2. 缓存写回和 Uncached 段写的优先级问题。在我们的设计中，缓存写回的优先级比 Uncached 段写回的优先级高，这是为了使缓存尽量多地保持在稳定的，而非正在写回的状态中。

此外，为了方便地与缓存层进行通信，我们还为 AXI 维护了一个表格，以请求 ID 为索引，记录当前正在进行的每个 AXI 请求之状态、目标地址、当前请求目标的是指令缓存还是数据缓存、当前请求的数据接收了多少个。此处，请求 ID 是访存地址的哈希函数。若发生了哈希冲突，则冲突一方需要进行等待。

（十一）双核心的相关设计

为了支持双核心，有如下问题需要考虑：

1. 软件为了在不同核间实现同步互斥，需要使用 LL/SC 指令。为此，需要实现一个独立于任何一个 CPU 的 LL/SC 事务处理器，统一处理来自两个 CPU 的 LL/SC 请求。且访存正在进行 LL/SC 事务的地址时，无论发起请求的是不是 LL/SC 指令，均需要绕过缓存。还有一点值得注意：当 SC 失败时，不同 CPU 应有不同的延时，否则当两个 CPU 同时发生 SC 失败后，很快又会同时发起 LL/SC 事务，导致 SC 再次失败，从而陷入无限等待。

2. 不同核心间需要通信协议。为此，我们移植了龙芯 3A 的通信协议，大致分为两个部分：启动时，每个核心监听各自的四个 Mailbox 寄存器，主核向从核寄存器写入入口函数的地址和参数，从核跳转至该函数。启动后，核甲可向核乙的中断控制寄存器写入，以向核乙发出某个编号的核间中断（IPI）。

（十二）启动 Linux 的相关问题

Linux 内核体积很大，无法存放在仅有 1MB 的 SPI Flash 中。为此，我们采用了两级 Bootloader 的方案。启动时，首先执行 ROM 中的初级 Bootloader，从 SPI Flash 中加载一个支持网络的 Bootloader U-Boot，SPI Flash 可以容下一个删除了多余信息的 U-Boot ELF 文件。接下来，U-Boot 通过网络（TFTP 协议）从另一台计算机上下载 Linux 镜像，并启动之。

三、设计结果

可以正常运行功能测试、记忆游戏和性能测试。性能测试得分为 38.668。

展示包可以运行双核版的 Linux 系统，系统中带有 Busybox 用户态环境，支持串口和网络两种交互方式。

（一）设计交付物说明

性能包的格式如大赛要求，展示包的相关说明请见展示包中的 README.md 文件。源码文件夹 myCPU 内的组织格式是：const 目录中包含了若干常量定义、dev 目录中包含了各种外设控制器、datapath 目录中包含了流水线的主体，其他文件位于根目录。

注意：我们的源码采用的是 VHDL2008 标准。若要重新向 Vivado 工程中添加设计文件，需要执行如下命令：“set_property file_type {VHDL 2008} [get_files *.vhd]”。

（二）设计演示结果

功能测试仿真结果如图 4，性能测试硬件运行结果（以最复杂的 coremark 为例）如图 5[3]。

展示包中的 CPU 运行结果如图 6，此处展示的是在 Linux 中运行 htop 指令的效果。可以直观地看出有两个 CPU 在运行。

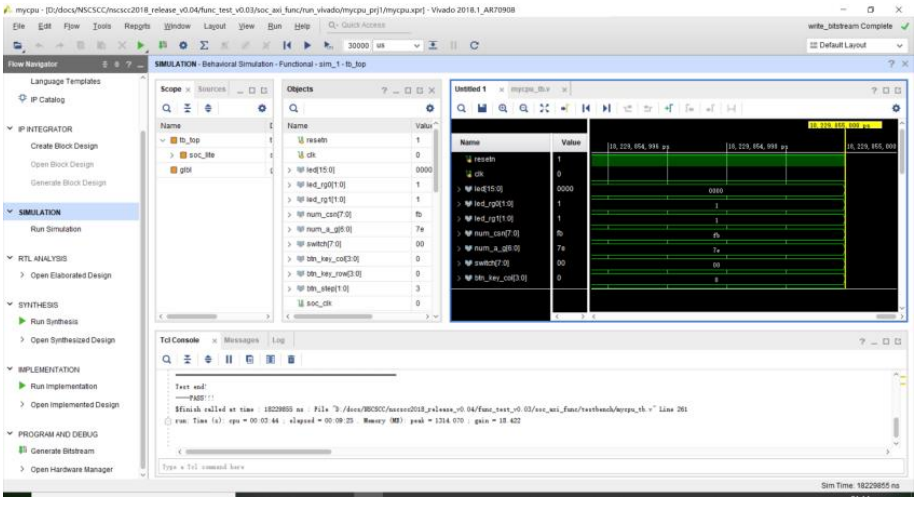

图 4

[3] 我们的板上的 LED 有问题（这已经通过发布包中对开发板进行测试的 bit 文件进行过验证），因此，所显示结果并不完全准确。我们认为结果应当是 0x6d35b8。

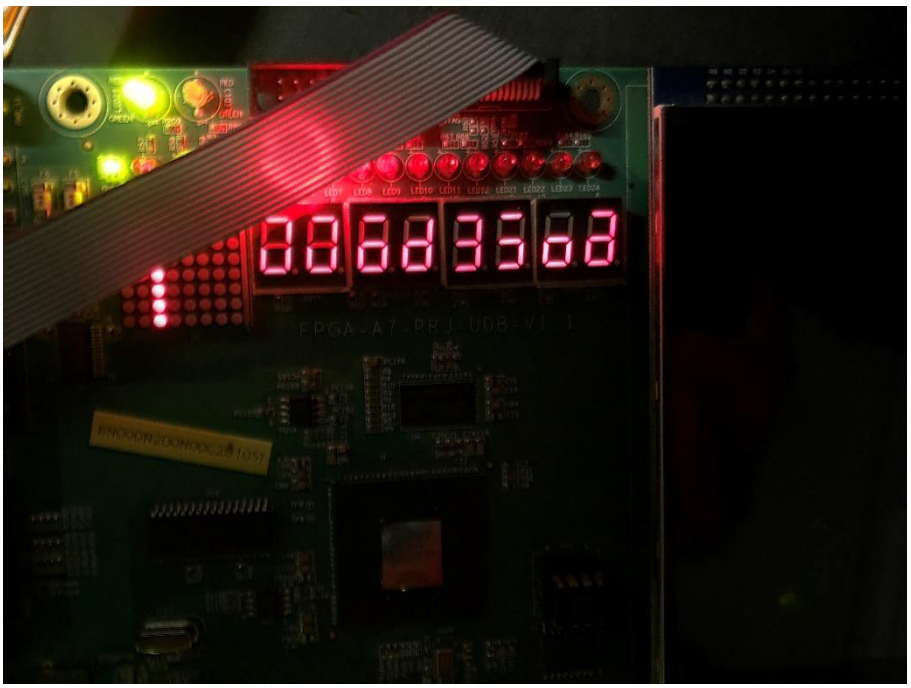

图 5

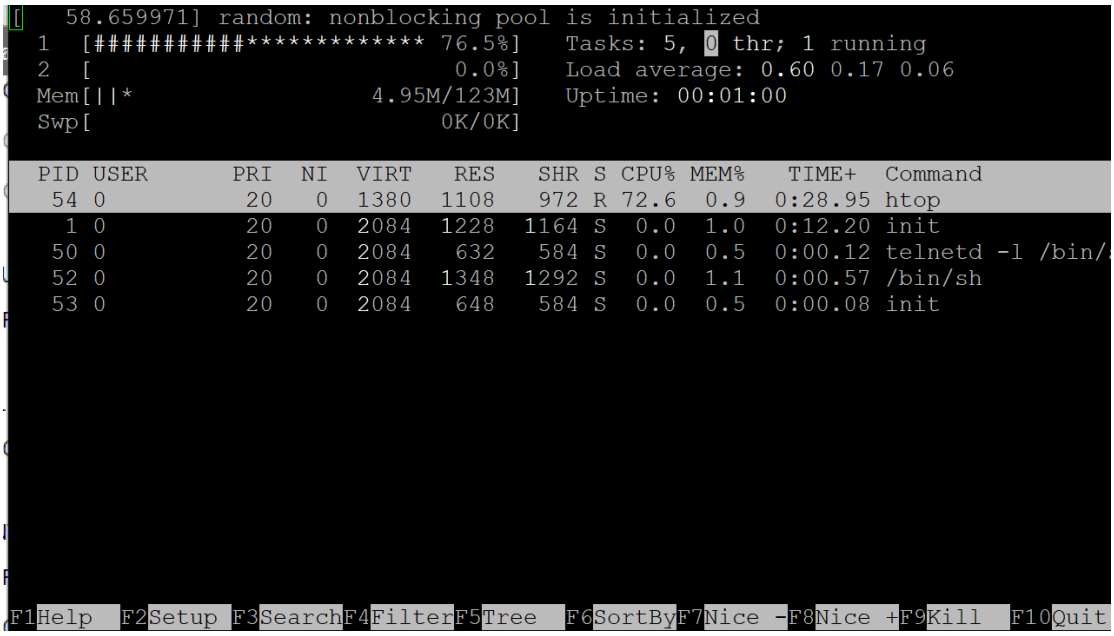

图 6

五、参考文献

[1] Imagination Technologies LTD, MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture. [2014-08-20]. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf

[2] Imagination Technologies LTD, MIPS Architecture For Programmers Vol. III: MIPS 32/

MicroMIPS32 Privileged Resource Architecture. [2015-07-10]. https://s3-eu-west-1.amazonaw-s.com/downloads-mips/documents/MD00090-2B-MIPS32PRA-AFP-06.02.pdf
[3] Imagination Technologies LTD, MIPS Architecture For Programmers Vol. II-A: The MIPS-32 Instruction Set Manual. [2016-12-15]. https://s3-eu-west-1.amazonaws.com/downloads-mips /documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf
[4] 《自己动手写 CPU》雷思磊著，电子工业出版社 2014-09-01 版。