# ThinRouter 计网联合实验报告

第 4 组

2020 年 1 月

## 目录

# 1 总述及整体设计

## 1.1 预期目标

在搭载 Xilinx Artix-7 FPGA 以及 KSZ8795 交换机芯片的 ThinRouter 实验板（如图 1）上实现如下目标：

1. 硬件实现路由器的转发工作
2. 硬件实现路由器的路由表查询、ARP 表查询
3. （软/硬件）实现 RIP 协议
4. 完成 MIPS/RISC-V 指令架构 CPU 的实现
5. 在 CPU 运行正确监控程序并通过评测网站的测试
6. 实现总线以及路由器和 CPU 的交互

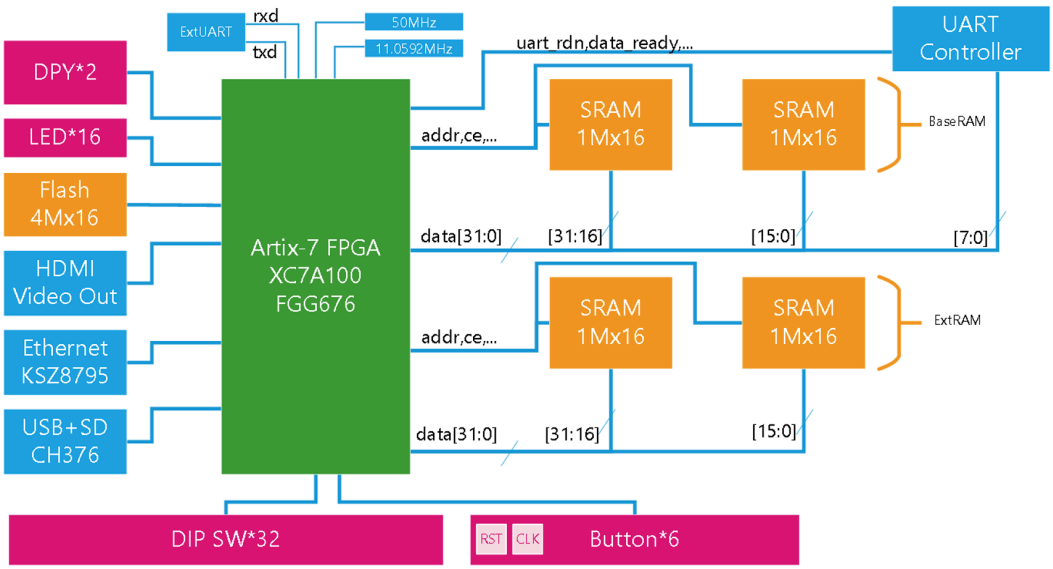


图 1: ThinRouter 硬件资源布局

## 1.2 完成情况

### 1.2.1 路由器

1. 全部硬件实现
2. 除双链双工外，转发速率均接近理论极限值
3. 路由表容量 8000+ 条，正常运行 RIP 协议，支持变长前缀、修改、删除

4. RIP 协议实现水平分割，可以用编译宏控制开启毒性逆转
5. 针对重要模块有 4 个随机化测试，其中路由表在随机的 8000 条路由项进行大量查询均可得到正确结果

#### 1.2.2 CPU

1. 实现了 MIPS32 Release 2 指令集中的 68 条指令
2. 正确运行了监控程序并通过了评测网站的全部测试
3. 实现了中断和异常的逻辑
4. 成功运行起了我们用 C 语言写的交互程序
5. 有比较完善的 3 个测试，可以模拟和监控程序在仿真环境中交互

#### 1.2.3 其他部分

1. 实现了总线
2. 实现了 DVI 显示功能
3. 实现了 BootROM，无需将程序烧入 SRAM 既可运行
4. 实现了把 SRAM、串口、DVI 显示、BootROM 以及路由器挂载到总线
5. 实现了简易的命令行，可以通过串口控制输入
6. 在命令行中实现了输出路由器中路由表项的命令

### 1.3 整体设计

如图 2 为 MIPS32 CPU、总线、BootROM、板载内存、CPLD 串口、DVI 显示以及路由器在 ThinPad 上的模块连接设计。

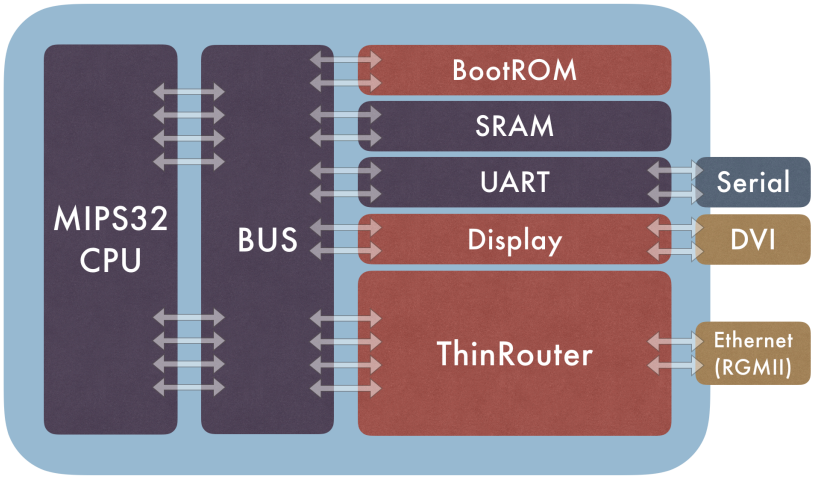


图 2: 各个模块在 ThinPad 上的布局

# 2 路由器设计

## 2.1 整体架构

本组路由器部分完全为硬件实现，不需要 CPU 参与。

使用了 Tri-Mode Ethernet MAC(TEMAC) 的 IP。TEMAC 提供 AXIS 格式的输入和输出接口，可以将路由器发出的以太网帧补充至至少 60 字节，并且在尾部添加 CRC。

路由器的整体架构见图 3。

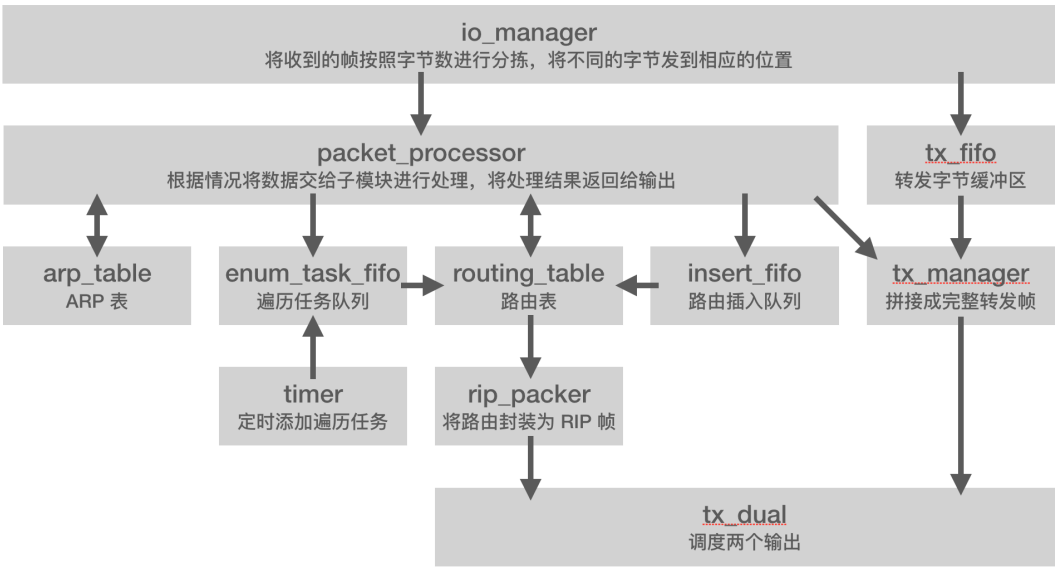


图 3: 路由器的整体架构

注：代码中的模块结构、线路连接可能与本文中叙述有所出入。这是因为项目开发之前，我们对于整体架构没有充足计划，所以大部分功能是在开发过程中增量添加的，可能缺乏合理的调度。文档中所描述的结构则是在开发完毕后对其结构的提炼。

## 2.2 模块功能

### 2.2.1 帧处理模块 io_manager

此模块的输入为 TEMAC 提供的 AXIS 输入，主要功能是鉴别收到的帧的类型，并且相应地做出动作。

在每一次接收以太网帧时，模块会记录当前收到的字节为第几个，相应地做出动作。例如，在收到第 7-12 字节时，会将其存入“来源 MAC”地址的寄存器中；又例如，18 字节时记录此帧协议是 ARP 还是 IPv4，而如果是 IPv4 且目标 MAC 地址为特定组播地址，则记为 RIP（其他情况则丢包）。此模块内包含许多寄存器，会将接收帧的特定字节存下，然后提供给相应的子模块进行处理。

在记录信息的同时，此模块还会将收到的帧中那些“可以直接作为发送的帧转发出去”的部分字节片段写入 **tx_fifo** 模块中。

当需要的信息接收完毕后，此模块会发出信号让子模块开始工作。

### 2.2.2 转发缓冲区 tx_fifo

对于 ARP 请求以及 IPv4 数据包，其大部分数据是可以不加修改直接转发的。因此在我们路由器的设计逻辑中，这两种数据包的转发相当于在原有包的基础上进行一些小的改动：将不需要改动的部分通过这个 fifo 队列传输，而其他部分（例如目标 MAC 地址）则通过其他模块进行处理得到，最终由 **tx_manager** 控制将两部分进行组合即可。

此模块的重要作用是，在 **packet_processor** 模块进行查表操作时，**io_manager** 仍然能够接收数据并且写入缓冲区，提升效率。同时，采用 fifo 结构（而不是直接写寄存器或 RAM）可以极大地节约 FPGA 资源占用。

### 2.2.3 转发帧组装 tx_manager

需要“转发”的帧包括 ARP 请求以及 IPv4（非 RIP）。发送过程则由寄存器记录当前发送了多少字节，再以这个数值决定下一个字节应当从 **tx_fifo** 中读取还是从某个寄存器（例如 MAC 地址）中读取。

对于 ARP 请求，**io_manager** 会通过 **tx_fifo** 将收到的帧去除“目标”部分传入。此模块则在此基础上插入路由器对应的地址即可。

对于 IPv4 请求，此模块将等待 **packet_processor** 传来查表得到的目标 MAC 地址，然后按顺序组装数据包。

如果遇到需要丢弃的包，收到信号后会将 **tx_fifo** 清空。

#### 2.2.4 地址解析模块 packet_processor

此模块的输入为 **io_manager** 从接收帧中提取出的 MAC 地址、IP 地址等需要处理的信息。此模块控制 ARP 表和路由表，而向 **tx_manager** 提供目标 MAC 地址和 VLAN PORT（即转发一个包所需要修改的部分）。

当接收到 **io_manager** 提供的信号时，此模块会根据处理需求进行“查询 ARP 表”“插入 ARP 项”“查询路由表”“插入路由项”四种操作之一或之二（对于转发 IPv4 包需要先查询路由表再查询 ARP 表）。

处理完毕后，则会将结果得到的 MAC 地址传递给 **tx_manager**，作为发送帧的一部分发出去。而如果查询不到结果，则发送一个错误信号，让 **tx_manager** 将当前的包丢弃。

#### 2.2.5 ARP 表 arp_table

由 **packet_processor** 控制进行插入或查询。

由于路由器一般在每个接口只接入一个设备，因此 ARP 表采用简陋的实现：存有 8 个空位，通过组合逻辑进行并行查找。

#### 2.2.6 路由遍历任务队列 enum_task_fifo & timer

当某一个 VLAN 端口发来 RIP 请求时，或 timer 计时触发时，会向队列中写入一项：需要向某个（1-4）端口发送一个 RIP 响应，目标 MAC 和 IP 都什么（定时发送则为组播地址，回复请求则为源地址）。由于路由表模块需要处理的操作很多，因此使用队列来保存未处理的操作。

#### 2.2.7 路由插入任务队列 insert_fifo

当收到 RIP 响应包时，对于每一条路由都会生成一项“插入任务”存在此队列中，由路由表进行读取。

#### 2.2.8 路由表 routing_table

**此模块的逻辑设计很琐碎，可以通过本文大致了解思路，不建议参考代码。**

需要处理“查询”“插入”“遍历”三种任务。因此，任务会以队列的形式缓存，待此模块空闲时按照查询 > 插入 > 遍历的优先级进行处理。

此模块的另一个难点在于，使用的 BRAM 空间很大，访问具有延迟，因此需要巧妙设计访问逻辑。

**数据结构**

路由采用路径压缩 trie 树形式进行存储，节点则分为“路径节点”（用于构成树结构，包含匹配的前缀和两个分支指针）和“叶子节点”（用于存储具体路由项，包含下一跳地址和父节点指针）。节点以存储地址的形式相链接。内存分为两半，低一半存放“路径节点”，高一半顺序存放“叶子节点”（使得遍历时可以顺序遍历）。

**遍历树**

在遍历前，需要确保当前 RAM 位置为根节点（地址为 0）。为了保证在等待 RAM 读取延迟过程中不影响处理逻辑，遍历过程不适用状态机，而应直接根据 RAM 读出的信息计算下一步访问的节点地址——理论上可以通过组合逻辑从 RAM 读取的内容计算下一步访问的地址，从而快速遍历，但是我们并没有做到如此优化。当 RAM 最终读出所需数据后，再将相应数据取出，RAM 地址归零以便下次遍历。

**修改**

主要使用状态机实现：将需要的操作分割成几个分别可以在一个时钟周期完成的操作，然后为每个操作单独设立一个状态，工作时则在这些状态中顺序跳转。

涉及许多琐碎的细节工作。修改 trie 树需要遍历到最后一个匹配的位置，然后对当前节点进行分割，以满足路径压缩的要求。移除路由项时，还需要优化存储使得所有“叶子节点”是连续的，以便于遍历访问（用最后一个来替换被移除的节点）。另外，在转移到新地址时，还需要等待 RAM 读出相应的块之后再开始处理。

**遍历路由表**

需要发送 RIP 响应时，遍历路由表的“叶子节点”，然后将得到的路由信息传递给 **rip_packer**。这里利用了 **enum_task_fifo** 使用 First Word Fall Through 的特性：路由表每次遍历有产生 25 条路由的上限，到达上限后经过一个冷却时间再继续遍历（如果没有冷却时间会导致路由器发送 RIP 响应的速度超过网口传输速率），而当真正将所有路由项遍历完成后，才从 **enum_task_fifo** 中读出这一个任务，表示任务完成。

#### 2.2.9 RIP 封装 rip_packer

遍历路由表时，路由表模块传入一些路由项信息。此模块再根据遍历任务中提供的目标 MAC 和 IP 地址，生成一个完整的以太网帧发送出去。

此模块主要针对的问题是 RIP/UDP 需要对所有路由项的 payload 计算 UDP checksum，而且 checksum 会在所有数据之前发送。因此当收到每一条路由时，需要算入 checksum，而将路由再暂存在另一个内部的 fifo 中。当收到了所有（至多 25 条）路由后，首先发送以太网帧头部、IP 头部和 UDP 头部，再从内部的 fifo 中发送每一条路由。

#### 2.2.10 输出调度 tx_dual

**rip_packer** 以及 **tx_manager** 两个模块都会产生输出，此模块负责在一方进行发送时缓存另一方的数据，让两个输出有序进入最终输出。

### 2.3 处理流程

#### 2.3.1 ARP 请求

如图 4，**io_manager** 将来源 MAC 和 IP 通过 **packet_processor** 交给 **arp_table** 进行学习，而将整个帧需要保留的部分发送给 **tx_manager**，由后者补上路由器自身的地址后发出。

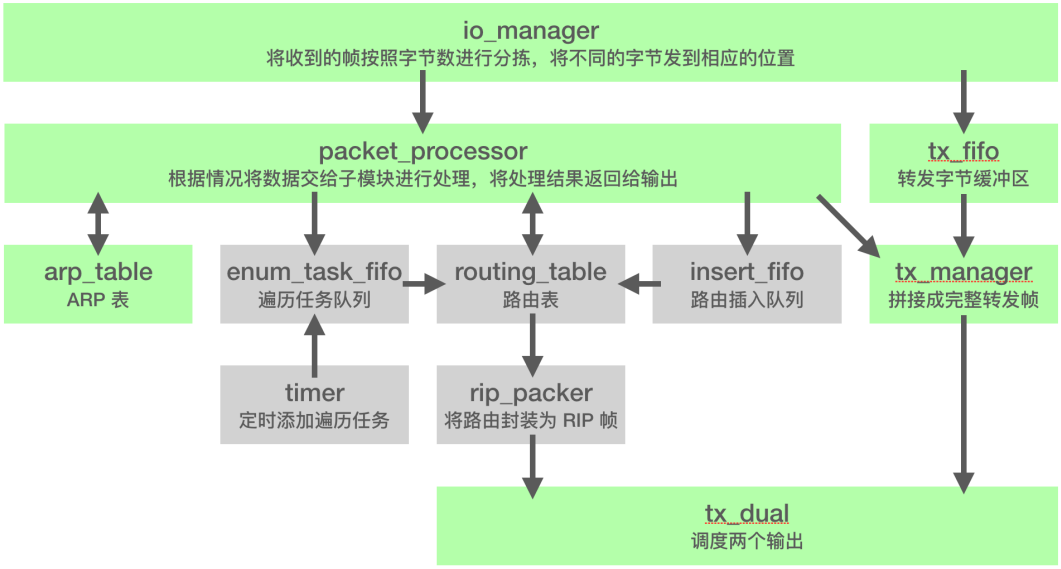


图 4: ARP 请求的处理流程

### 2.3.2 转发 IPv4 数据包

如图 5，**io_manager** 将目标 IP 传给 **packet_processor**，由后者进行查表。

**packet_processor** 首先检查目标 IP 是否属于路由器四个直连子网之一，如果属于则可以跳过查询路由表这一步，否则则从路由表中查到下一条 IP 地址；然后再用 IP 地址查询 MAC 地址，交给 **tx_manager**，由其结合直接转发的数据一并发出。

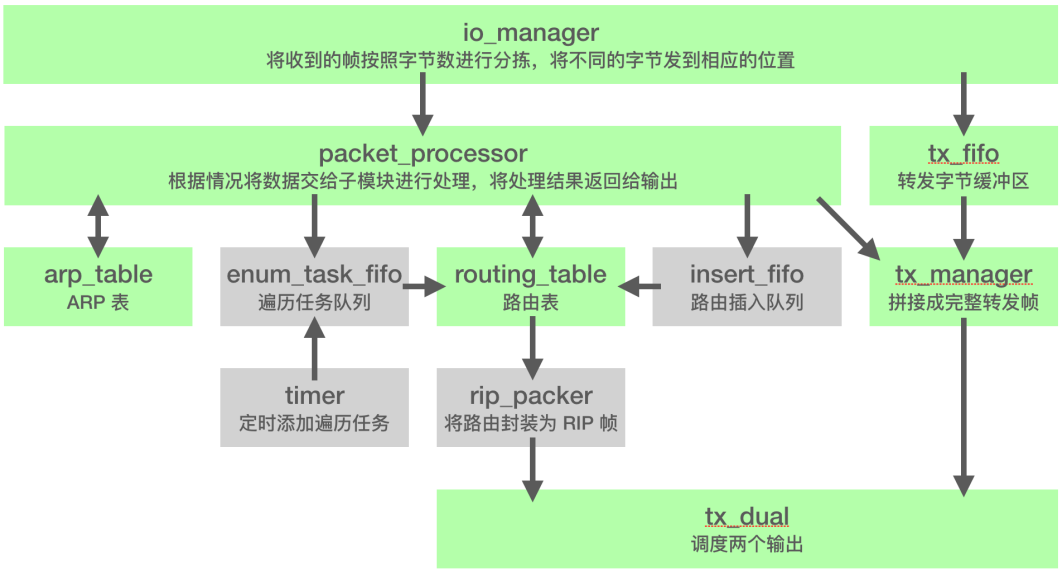


图 5: IPv4 数据包的转发流程

### 2.3.3 RIP 请求

6，一个 RIP 请求只需要被记录在 **enum_task_fifo** 中即可，待路由表空闲时再进行遍历

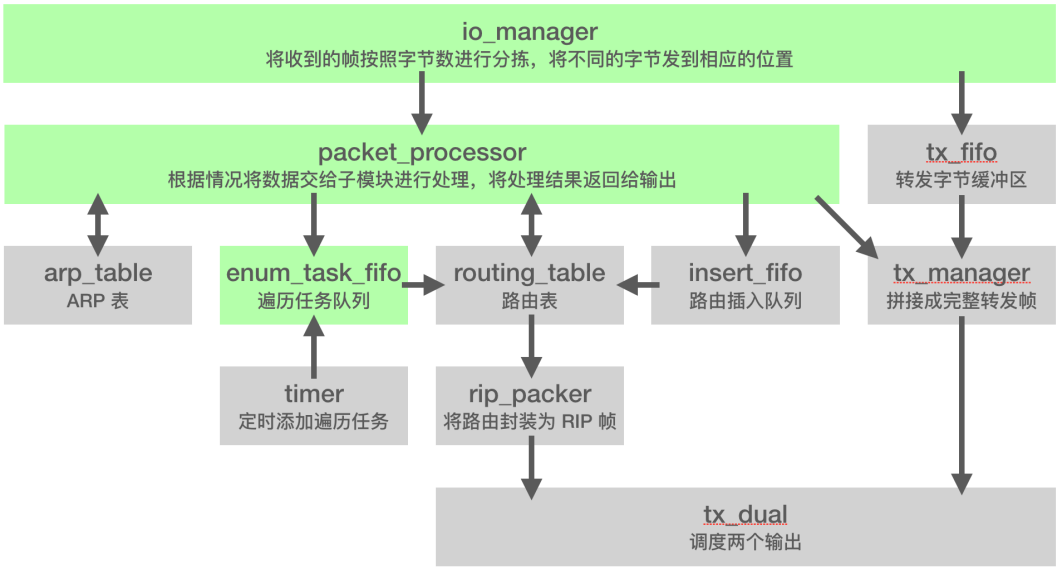


图 6: RIP 请求的处理流程

### 2.3.4 RIP 响应

如图 7，**io_manager** 通过状态机记录接收数据包中的每一条路由项，经过 **packet_processor** 写入 **insert_fifo**，等待路由器在空闲时存入路由表

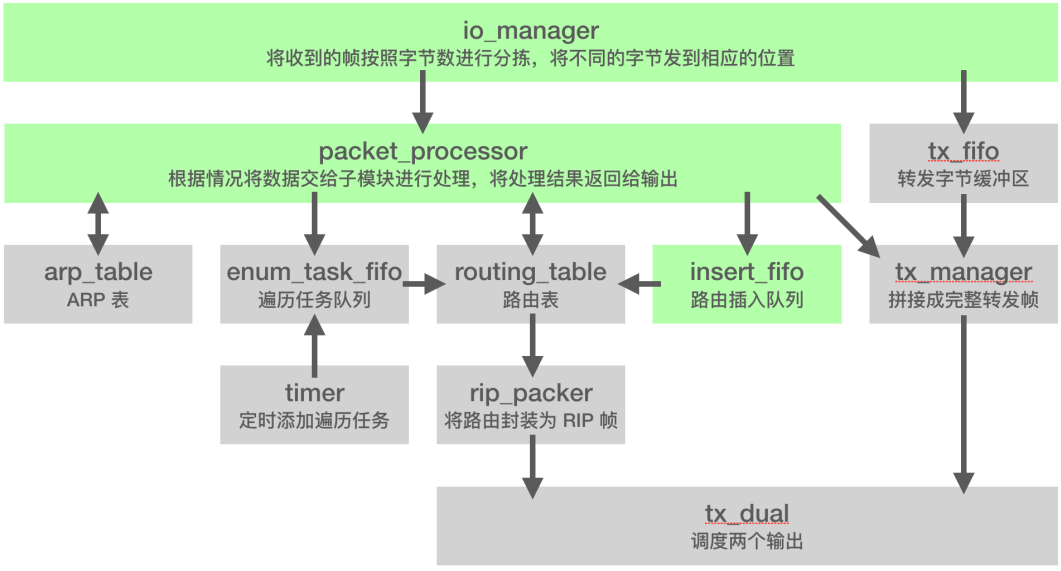


图 7: RIP 响应的处理流程

### 2.3.5 发送 RIP 响应

如图 8，**enum_task_fifo** 中的遍历任务可能是接收到 RIP 请求所记录，也可能是计时器触发写入。而路由表则不在意这些细节，如果存在遍历任务就会对相应的 VLAN PORT 进行遍历路由表，每 25 项发送给 **rip_packer** 打包成一个 RIP 响应后发送出去

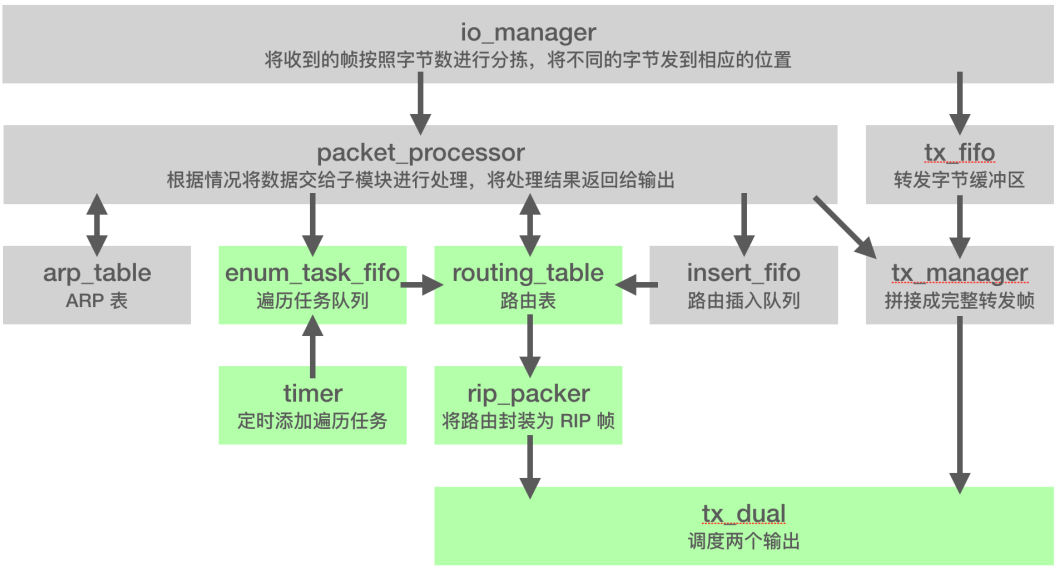


图 8: RIP 响应的处理流程

# 3 CPU 设计

## 3.1 整体布局

本组 CPU 为经典的 5 级流水线设计，划分为了取值、译码、执行、访存和写回这五个阶段，同时有还有通用寄存器、0 号协处理器、Hilo 寄存器、控制模块以及一个地址选择器，具体连接如图 9。

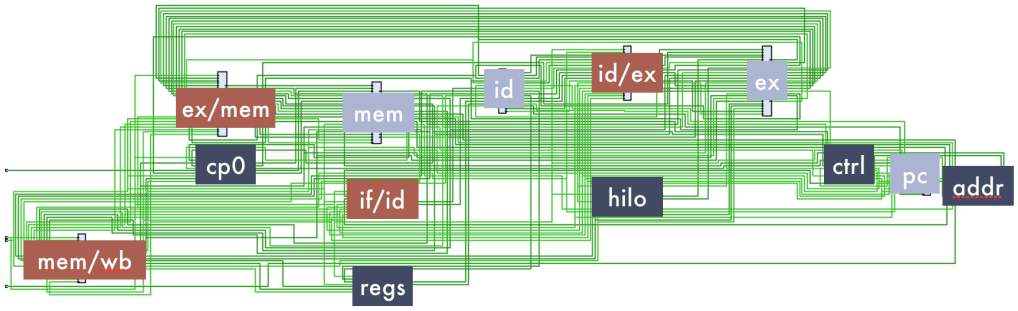


图 9: 各个模块在 CPU 上的布局

其中的 pc、id、ex、mem 和没有例化的 wb（其实就是 regs 中的部分写回逻辑）是 5 级流水线上的每个阶段，而 if/id、id/ex、ex/mem 以及 mem/wb 为两个相邻阶段中的流水线寄存器。

图中的 regs 和 hilo 分别表示通用寄存器以及 MIPS 标准中的 Hilo 寄存器；ctrl 模块用来接收流水线暂停请求和控制流水线暂停；cp0 作为协处理器可以处理中断和异常；最后的 addr 是地址选择器，当 pc 想要取值、mem 同时需要访存时，选择器会控制优先访存，最后 CPU 给总线的地址线也是出自这里。

## 3.2 冲突解决

### 3.2.1 数据冲突

对于所有的寄存器数据冲突（即后面的指令要访问紧接着前面的指令修改过的寄存器），我们的解决方案都是通过旁路做数据前传，如在译码阶段读取寄存器时也会查看流水线上执行阶段和访存阶段要写回的寄存器地址是否一致。对于 Hilo 寄存器的访问也是类似的解决方案。

而对于访问内存的指令，我们采用了插入气泡的方法，暂停了取指，通过访存器给控制器发出暂停请求来插入气泡，避免同时访问带来的冲突。

### 3.2.2 控制冲突

在本 CPU 流水线设计中，唯一可能出现的控制冲突是跳转的指令，如果将跳转的目标地址放入 ex 段，那么在只有一个延迟槽的情况下必须插入一个气泡才可以取值，为了避免这种情况，我们将跳转地址的计算提前到译码阶段。

### 3.2.3 结构冲突

当遇到访存指令时，访存指令与流水线中的另一条指令的取指阶段均需要访问存储设备。如果访问的是同一外设，就会产生结构冲突。我们的设计是遇到访存指令时会直接暂停流水线一个周期，从而解决冲突。

## 3.3 异常与中断

我们能够支持精确异常处理，我们在流水线的各个阶段先收集异常信息，在 mem 阶段统一进行处理。处理异常时，修改 CP0 中的 EPC、Status、Cause 等寄存器，并判断异常指令是否在延迟槽中，随后清除流水线上未完成的指令，执行异常处理的指令。

## 3.4 外设

我们实现的外设有 BootROM、BaseRAM、ExtRAM、UART 串口、DVI 以及 ThinRouter。在总线上，每个外设对应一段固定的地址，CPU 需要访问特定外设时只需读/写对应外设的地址即可。

### 3.4.1 BootROM

BootROM 由板上的 xpm_memory_sprom 实现，利用 mem 文件进行初始化，生成的 bitstream 包含了 BootRom 的初始化内容。BootROM 对应一段地址空间。cpu 初始化时的 PC 指向 BootROM。这样，板子烧入 Bitstream 以后一开机即可看到我们的交互终端界面。

### 3.4.2 SRAM/UART 串口

SRAM 映射到一段物理地址，直接读写即可。对于串口，串口的状态和数据分别对应地址空间中的两个地址。读写时需要先轮询读状态位（查看是否可以读/写），直到可以读/写时再读/写数据位。

### 3.4.3 ThinRouter

直接将路由器路由表对应的内存区域映射到总线一段地址，使得 CPU 可以通过内存地址访问路由表。这段内存使用 XPM 的 True Dual Port RAM，其中路由器对其读写，而 CPU 只读。

### 3.4.4 DVI

DVI 显示可以显示 $78 \times 32$ 的等宽 ASCII 字符矩阵。为了节约 BRAM 空间，内存中不会存储每个像素的值，而仅存储所有字符，而在逐行扫描像素的同时，计算当前像素对应哪个字符的某个像素位置，然后读取字体对应位置的颜色值。

字符矩阵的输入接口仅有 7 位字符编码、1 位使能和 1 位时钟，由显示模块追踪输入字符的位置，并实现换行、退格操作。当遇到总行数超过屏幕行数时，会将第 32 行写至第 0 行，而同时将整体的偏移 +1，最终实现屏幕滚动的效果。

# 4 交互设计

为了使 CPU、路由器和用户有完善的交互体验，我们设计了如图 10 的可接屏幕的命令行体验，成功实现了字体在屏幕的显示、滚动，也成功用 C 语言和汇编编写了命令行的控制程序，支持：输出帮助信息、清空屏幕以及打印路由器中的全部路由表项。同时，为了实现键盘的控制，我们也实现了在 PC 端的键盘控制器，通过串口实现对 ThinRouter 的控制，同时串口也会把要显示的信息发送给 PC 端，PC 端的屏幕会显示和真实的 DVI 屏幕上一样的内容，而且因为是 Qt 实现也支持清屏、删除字符等一系列复杂的操作。

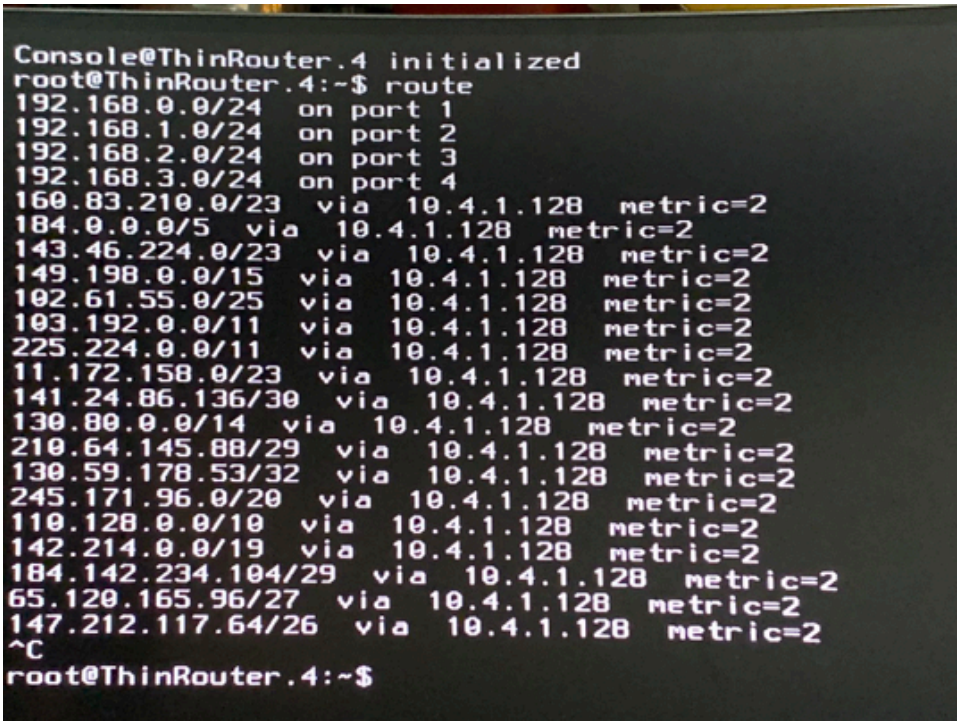


图 10: 交互命令行

### 4.1 软件设计

要使得命令行的控制得以被 CPU 执行，我们用了混合编译的技术将启动代码（设置 C 语言需要用的栈等等寄存器的汇编代码）和 C 语言链接并翻译成二进制流写入了 BootROM 中，使得 Bitstream 在被写入 FPGA 之后就可以直接执行命令行的程序。

为了让命令行的程序更有可读性，我们适配了硬件的 getchar、puts 和 putchar 等函数，都是采用轮询的方式来查询串口的状态并接收或者发送新的数据给 PC 端。另一点需要注意的是，和监控程序的交互是通过终端而非轮询来实现的。

而 PC 端的控制器（如图 11）是一个 Qt 界面程序，包含了一系列读取键盘输入的信号槽以及一个不可编辑的文本显示器，显示器的内容只会从 PC 串口读出的数据中控制。

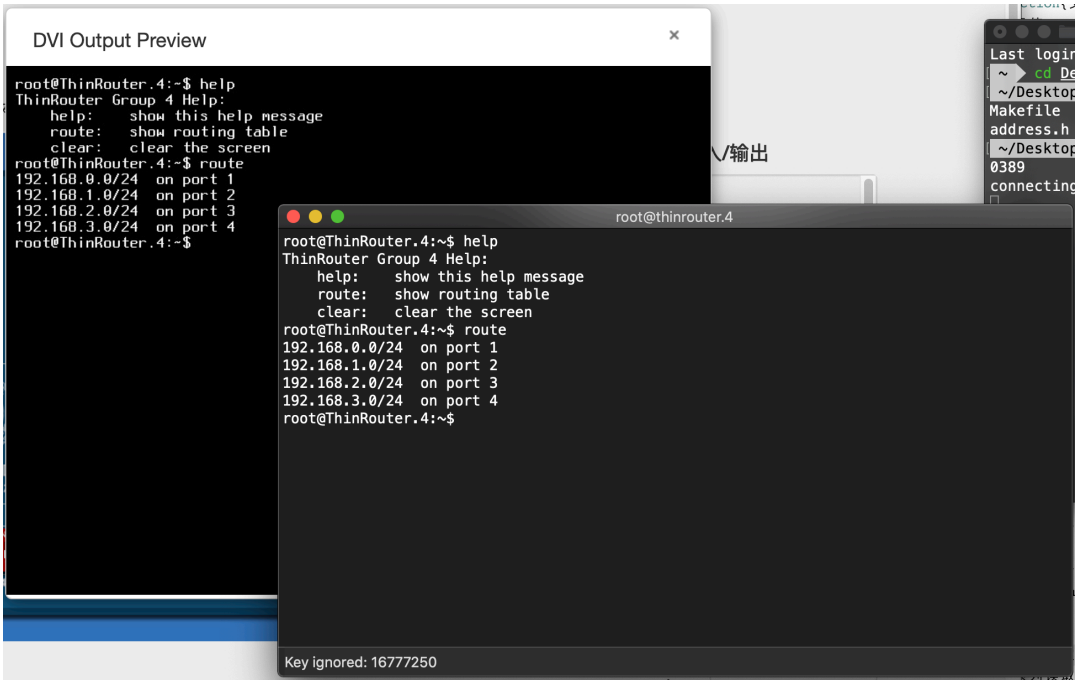


图 11: PC 端控制器

### 4.2 硬件设计

为了实现轮询等方式的访问，我们给每个需要用到的状态或者数据分配了地址，如串口的数据、串口的状态、路由器表项的个数、BootROM 的地址范围、SRAM 的地址范围以及路由器表项的地址范围。

路由器方面，为了能让 CPU 和路由器的查表逻辑可以同时访问路由表，我们使用了双口的 BlockRAM，支持 CPU 和路由器的同时访问以及修改。

## 5 单元测试

为了保证开发的效率，确保每个模块的正常运行，我们进行了大量的单元测试，在 CPU 代码约 2500 行、路由器代码约 3000 行的情况下，我们仍然保持了 3000 行的 Testbench 用于各个功能的测试。

### 5.1 CPU Testbenches

CPU 共有 3 个测试用例，分别是自启动测试、和监控程序交互的测试以及和自制命令行交互的测试。

其中我们为 CPLD 串口发送接收模拟器添加了状态机，实现了命令的状态解析，通过一个 Python 脚本可以事先把想要运行的命令写入一个 mem 文件，然后在仿真中运行对于的 Testbench 会直接在仿真的命令行中打印出来。

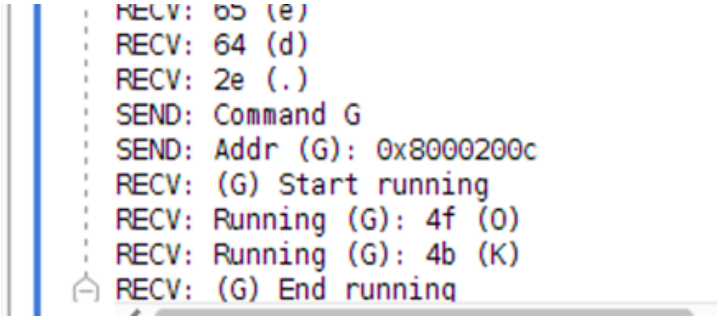


图 12: 和监控程序交互的仿真

如图 12 就是一个和监控程序交互的仿真，其中 Testbench 给 CPU 发送了命令 G 和要运行代码的地址，CPU 发回开始执行的信号、发回执行中打印的信息、发送回结束运行的信号，Testbench 的状态机均能解析出对应的状态并打印，大大提高了调试的效率。其他两个测试与之类似，这里不再赘述。

## 5.2 路由器 Testbenches

路由器全部由硬件实现，在运行环境很难调试，因此充分的仿真测试非常必要。对于路由器关键模块我们写了脚本来进行压力测试。

```
query   196.10.130.62/32 -> 72.48.87.252/32
query   40.130.14.17/32 -> 0.0.0.0/32
query   132.71.221.149/32 -> 213.86.253.166/32
query   9.246.39.36/32 -> 0.0.0.0/32
insert  228.0.0.0/10 -> 185.117.34.1/32 (12, 2)
insert  128.0.0.0/5 -> 149.171.68.68/32 (8, 4)
query   107.96.159.166/32 -> 168.237.72.69/32
insert  239.54.0.0/15 -> 34.216.41.163/32 (7, 4)
```

图 13: 随机生成的路由表测例片段

图 13 是脚本随机生成的一段路由表测例，路由表模块的 Testbench 会将这些条目解析并作为路由表模块的输入，然后将其返回的结果与测例中的“query”的后半部分相比对，如果发现错误则直接终止仿真。

图 14 是脚本随机生成的另一段测例，用于测试整个路由器的功能。Testbench 会将其中的十六进制编码当做网卡的输入提供给路由器，而在仿真运行时则将路由器内部进行的逻辑和路由器发出的网帧打印在控制台，如图 15。

```
eth_frame: 01 00 5E 00 00 09 54 5C C7 F3 9E DC 81 00 00 02 08 00 45
info:      RIP Response from 10.4.1.128:
info:       229.199.169.0/24 -4> 217.20.146.1
info:       162.53.167.0/24 -5> 70.5.243.249
info:       59.185.120.0/24 -15> 232.192.111.125
info:       66.170.131.0/24 -8> 206.108.238.16
info:       39.28.21.0/24 -11> 141.74.212.96
eth_frame: 01 00 5E 00 00 09 CC E4 6E CC 7A F6 81 00 00 01 08 00 45
info:      IP Request: 10.4.3.128 -> 93.80.212.159
eth_frame: A8 88 08 38 88 88 70 37 7E DE C9 F8 81 00 00 03 08 00 45
info:      IP Request: 10.4.4.128 -> 239.50.85.27
eth_frame: A8 88 08 48 88 88 52 CE C6 1F E5 08 81 00 00 04 08 00 45
info:      IP Request: 10.4.1.128 -> 243.16.91.245
eth_frame: A8 88 08 18 88 88 CC E4 6E CC 7A F6 81 00 00 01 08 00 45
info:      RIP Request from 10.4.3.128
eth_frame: 01 00 5E 00 00 09 70 37 7E DE C9 F8 81 00 00 03 08 00 45
info:      ARP Request: 10.4.2.128(54:5c:c7:f3:9e:dc) -> 10.4.2.1
eth_frame: FF FF FF FF FF FF 54 5C C7 F3 9E DC 81 00 00 02 08 06 00
info:      RIP Request from 10.4.3.128
```

图 14: 随机生成的帧片段

```
Query: 85.63.42.220
Not found in routing table

DONE 1
3104000
Info:         IP Request: 192.168.0.128 -> 85.63.42.220
Router IN:   a8 88 08 18 88 88 67 ed b9 7d 58 c7 81 00 00 01

ARP table saving entry:
    IP: 192.168.2.128
    MAC:    a1:6b:2a:d8:13:8f
    VLAN ID:    3

3616000
Info:         ARP Request: 192.168.2.128(a1:6b:2a:d8:13:8f)
Router IN:   ff ff ff ff ff ff a1 6b 2a d8 13 8f 81 00 00 03

Add arp
Router OUT:  a1 6b 2a d8 13 8f a8 88 08 38 88 88 81 00 00 03
```

图 15: 路由器在仿真时的一段输出。由于处理操作在接收过程中进行，而接收完成后才打印接收，所以顺序存在颠倒

# 6 性能测试结果

- 单路单工：93.8 Mbps
- 双路单工：187.6 Mbps
- 双路双工：254.7 Mbps
- 64 字节小包：133.7 Kpps
- 路由表压力测试：5000 条通过

# 7 对实验的建议

## 7.1 针对计网联合实验的建议

- 联合实验中的有些部分（比如单纯硬件转发包）其实更像是单纯写代码和网络、组成原理的知识交集比较少，下次实验建议可以尽量减少这种部分，比如「可以直接把硬件转发也砍掉，硬件那边就做一个带缓冲区的网卡外设，然后要求 CPU 通过中断处理」，这样的话 CPU 的工作要求会更高，组成原理课程评分的部分就会和网络相对均衡，这次的体验其实我们的 CPU 还是比较普通。或者说，精 nèi 进 juǎn 的方向更在于这两门课的范围内，比如「电脑可以从路由器的 80 端口 GET 到路由表」。
- 开会的时候效率没有那么高，有的时候每个组说的问题都是自己遇到的，深入交流效率比较低。
- 最开始上手做的时候了解太少，可以多讲一些基础内容铺垫。
- 提前把免考或者加法在报名前说清楚，对大家更加公平。而且其实更希望鼓励免考，因为很多考试其实大家的普遍情况都是考前突击（像网络原理这样细节的考点需要大量背诵，这样意义也不是很大）。

## 7.2 针对其他专业课的建议

- 关于网络原理的 Router-Lab 文档非常好，希望老师的 PPT/其他课程的作业说明都可以有这么清楚。
- 作为一个 4+3 分的两个课程，联合的工作量可以接受，但也希望其他课程可以根据学分适当调整工作量（比如 2 学分的编译原理大约等于 4 分的课程工作量）。
- 这次计网联合的形式还是相对普通更好的，希望未来也有更多的这样的课程设置，让大家在实践中增进对知识的了解。

# 8 感谢

本次作为历史上第一次参与计算机组成原理和网络原理的联合实验的小白鼠收获非常多，在此感谢几位跟进实验的助教和老师（尤其是替我们踩坑、解决各种疑难杂症的杰哥，还有半夜给我们补习网原知识点的蛤主席），也感谢在实验中遇到的每一个同学（当然还有旁听的学 yuàn 长 shì）。