CPU Router Report

计网联合实验第一组

January 2020

目录

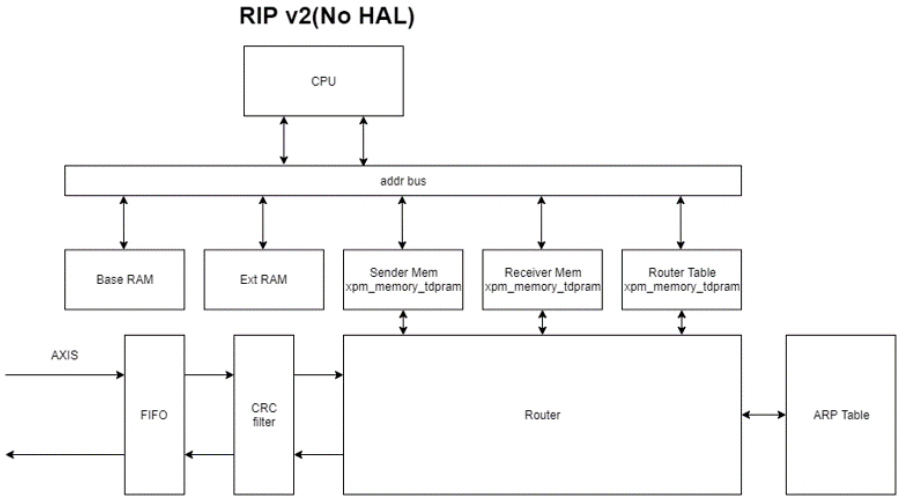

图 1: 路由器结构

1 总述

实现了一个在自己设计的 cpu 上运行的硬件转发（多个包流水线处理）的路由器，实现了包含水平分割、毒性反转等的 rip 协议（软件路由器实验的要求），也实现了软件实验中没有涉及的 ARP 协议（请求与应答）、CRC 校验码处理等功能。

2 设计说明

分为硬件部分和软件部分，软件部分实现了基础 mips32 CPU，还实现了很多要求外的 mips 指令（包括 lwl,lwr 等地址不对齐的访存，异常和中断等）。硬件部分实现了接收发给本路由器的包并缓存给 CPU，发送 CPU 传递过来的包，以及包的转发。硬件与软件之间的交互方式如图 1 所示。

2.1 硬件部分

转发包的数据流如下：

数据流从 eth_rgmii_rd 流入，先通过一个 eth_mac_fifo_block 实现从 rgmii 协议到 axis 协议传输数据，再通过一个 no_crc_filter 去掉尾部的

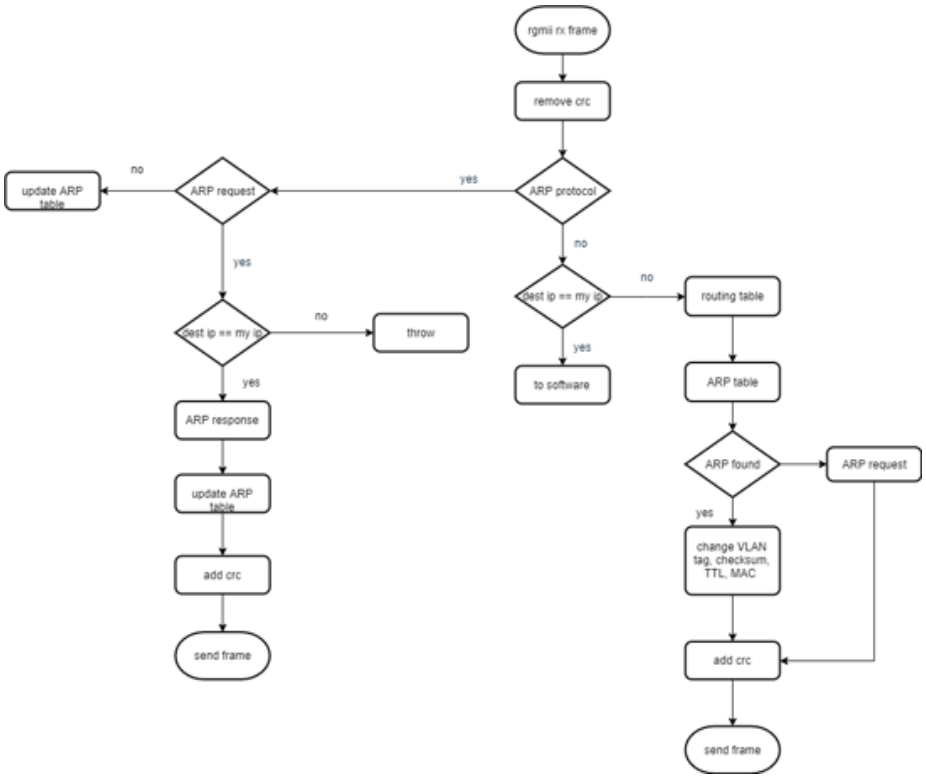

图 2: 主状态机

crc（新包的 crc 会在转发出去的时候被 eth_mac_fifo_block 自动加上），使用 FIFO 来维护当前处理的数据流。实际处理中相当于包位于流水线上，队列一直在不停地读入新包的每一位与不断发出已经处理好的包每一位，与此同时按照先进先处理的原则不断对 FIFO 中的包进行改 mac 地址、根据目标 ip 查路由表、根据 nexthop 查 arp 表、根据最终结果写入目标 ip、目标 mac、vlan tag、ttl、checksum 等操作。

以上为路由器最核心的转发功能，处理逻辑如图 2 所示

路由器的状态机实现逻辑如下：

初始时，状态机处于空闲状态，此时面对从 eth_mac_fifo_block 传入包，上一个包需要发送 ARP request/response，cpu 需要发包三个事件到来时，处理优先级为 ARP 发包 >cpu 发包 > 网口传入包。

如果需要发送 ARP request/response，那么根据设置状态位时保存的信息直接构造包放入 FIFO 中，监听 FIFO 的部分会在之后按照先入先出

的原则将数据发送出去，回到初始状态。

如果 cpu 需要发送包，路由器按位读取 cpu 与路由器交互的缓冲区中的内容，放入 FIFO 中，在将所有数据放入 FIFO 之后将缓冲区标志位置为空表示已经发送完。

如果从网口收到了包，我们的处理逻辑如之前图 2 所示，首先读入数据包的前 60 字节，然后判断该数据包属于 ARP 包还是 IP 包，如果是 ARP 包是否该路由器需要作出应答，如果是 IP 包那么应当传给 CPU 还是转发走。在查表的过程中，路由器的收包并不会暂停，以类似流水线的方式避免了性能的损失。

2.2 软件部分

软件部分由 C 语言实现，结构较为简单，由以下几个模块组成。

串口读写：由于 CPU 上并没有操作系统，因此无法使用预编译得到的 Syscall，而要主动访问内存串口进行读写。本模块长期监听内存上表示串口信号的地址并按需进行读写，在此基础上手动实现了面向串口的 putchar(), putstring(), printbase() 与 printf(), 实现输出监控。这一模块是软硬件联合调试的基础。

HAL: 普通实验的 HAL 是黑盒，无需实验者关注；而在计网联合实验中，我们将必须手动实现软件收发包的逻辑，从软硬件接口留出的内存地址获取某一硬件得到的包并进行解析、判定是否是 RIP，通过硬件发送一段需要发出的包。

RIP 逻辑。这一部分保证的包括了所有普通实验的内容，含：RIP 的拆解、路由表的维护、路由表的定期广播、回应 RIP Request、根据 RIP Response 更新路由表信息。

编译相关逻辑。要将编译到 MIPS 的 C 代码成功运行在 CPU 上，需要编写一段汇编代码对其进行正确引导。

2.3 软硬件交互部分

硬件与软件部分交互的难点在于，硬件与软件处于不同的时钟域上，无法简单地通过中断等方式进行交互，因此我们的想法是通过内存进行交互。当硬件向软件发包时，向指定的一段内存中写入数据，并置标记位，软件部分采用类似于轮询的方式，发现标记位被置位后读出数据并将标记位置零表示已经处理好这个包。发包过程与之相反，cpu 向指定的一段内存地址处

Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (300)
21148	6331	1490	174	88	4	237

图 3: 资源开销

写入包并置标记位，路由器在每个初始状态都会访问一个标志位，一旦发现标志位变动，则说明 cpu 有想要转发出去的包，路由器将数据放入 fifo 中并将标记为置零，通知 cpu，以便其可以从等待中恢复，准备监听收包标志位或是发下一个包。

在 rip 协议中路由表有时要根据发来的包更新，路由表对于 cpu 来说是可读可写，对于路由器来说是只读的，cpu 更新路由表时查表模块会等待以免出现访问不一致的问题，同样使用一段内存进行软硬件交互。

3 结果与分析

单链单工	双链单工	双链双工	单链单工小包
93.8Mbps	187.6Mbps	323.1Mbps	125Kbps

表 1：性能测试

小规模路由表	大规模路由表
正确	错误

表 2：路由表测试

我们的实现能准确地学习小型路由表，并且正确地进行转发。由于修复后的主状态机会保证收包、查表、发包同时进行，已知信息总在第一时刻发出、等待拍数最少，所以我们的转发性能在所有组中都表现极好（几乎是最好）。

美中不足的一个问题是，我们使用的 Trie 树是最经典的 Trie 结构，并未进行路径压缩以减少开销，导致查表效率并不算高，影响了我们的小包速率。同时，Trie 的巨大内存开销使得我们缺乏自由的内存使用。

最大的问题在于大规模路由表。多个 RIP 包同时到达时，由于 CPU 运算较多、主频更低，先到的包会占用软硬件接口的内存导致后续的包无法进入，而停下状态机以等待 CPU 处理又会极大影响效率，因此我们选择了丢弃后来的包，这直接导致大规模路由表计算错误。正确解决这一问题需要一个不太小的 FIFO 在状态机与内存地址间做缓存，然而 Trie 巨大的内存开销使我们无法添加 FIFO。因此，我们最终没有解决这个问题。

与此同时，如图三所示，我们的设计组合逻辑较多、开 reg 太多，过大地消耗了 Slice LUT，整段代码的时序非常紧张，这对后续更多扩展很不利。

4 遇到的问题及解决方案

问题：收到的包有时迟了一拍

解决方案：将 reset 信号连上 PLL 模块的 locked

问题：ping 的时间太长甚至 ping 不通

解决方案：经过排查发现 checksum 模块写错了所以包被丢了，误以为 TTL 减一是在校验和的低位减一，但其实是在高 8 位减一（因为 TTL 的位置是 checksum 计算中的高 8 位），所以不该把 checksum 加一，而是应该把 checksum 的高 8 位加 1，也就是加 256

问题：有时仍然会莫名其妙损失包

解决方案：发现 checksum 算的仍然是错的，经研究发现 checksum 虽然是取模，但是是模全 1，因此全 0 和全 1 是同余的，但加一后若为全 1 则是不可能由正确校验方式算出来的非法状态，应该改为全 0

问题：改完之后有时仍然会莫名其妙损失包

解决方案：忽略了另一种常见得多的“特殊情况”：高 8 位全 1 时加 1 实际上跨过了“待模值”，因此相当于少加了 1（这里的 1 是真的数值上的 1），因此要在低 8 位补回这个 1

低 8 位补 1 并不会进位，因为若发生则原先低 8 位全 1，高 8 位也全 1（补 1 的先决条件），如前所述，是一个 checksum 的非法状态

问题：状态机卡死，发不出包

解决方案：发现是状态机和查表模块的接口约定出了分歧，状态机一直把针对查表模块的 valid 拉高，以至于查表模块误以为一直有任务，无法停止。

之后参考 AXI 的接口方案设置 valid 和 ready 就解决了问题。

问题：从 cpu 发出的包长度少了 1，最后 1 字节也没了

解决方案：发现是因为下标从 0 开始，导致长度传错了，因为发到长度就停止，因此向下游发的时候也不再发最后 1 字节

问题：写给 cpu 的数据 1 个字节中有几位怎么也写不进去

解决方案：仿真发现是位使能设错了，设错是因为用了!，实际上 vivado 按位取反应该用 ~

问题：一有发给 cpu 的包，状态机就卡死

解决方案：发现是因为想要优先给 cpu 更新数据（如路由表）从而对 cpu “可收” 状态进行了 busy waiting，然而 cpu 不可收的原因很多，其中之一甚至是在等着路由器收它要发出的包！这样就形成了死锁，无法脱出

因此最后取消了这个 busy waiting，cpu 不可收则暂时搁置甚至丢弃这个包

问题：时序紊乱，包字节漂移

解决方案：发现是使用的 LUT 资源太多，因此减少了 reg 使用，通过改小数组大小、改用 xpm 等方法

问题：cpu 与串口交互与预期不符

解决方案：因为偷懒没写总线而是特判的，结果就造成了多驱动问题，最后补上了总线就解决了

问题：从 cpu 发出的包总是在最开始多了一个神秘字节

解决方案：软硬件接口有 1 的 read latency（因为要从 cpu 对应位置取得数据），硬件状态机没考虑这一点，结果前面就多读了一个 “脏数据”，因此等一拍再开始存储

问题：从 cpu 发出的包第二个字节总和第一个相同

解决方案：和上个问题类似，虽然最开始多等了一拍，但之后仍按照原来的节奏，所以等于第二拍还是读的第一个字节，就全都延迟了。

所以之后的读也相应延迟，具体在硬件实现中，每拍输入的地址是下下个周期要读的地址而非下个周期要读的地址。

这样会引入一个问题，就是如果周期中断了，比如有个周期不能读或者不能写了，整个系统就紊乱了。

所幸，从 cpu 读时 cpu 准备好数据后就不会中断，可以一直读，而路由器自身可以通过 ready 信号控制上下游读写（何况读数据也并不影响写），所以不会有周期中断，这个方案可以顺利进行。

问题：iperf 效率特别低

解决方案：发现是因为重传，而重传是因为丢包，丢包是因为 fifo 满了，fifo 满是因为处理效率低，处理效率低是因为查表的时候状态机在空跑没有从上游继续读，所以后来改成了流水线。

5 对实验改革的建议

5.1 实验组织方面

请以后务必提前明确通知各组同学本实验的实际工作量，很多人最初是抱着” 做一个大作业交两门课” 这样的心态报名的，之后可能要告诉同学们本实验任务量 » 造机 + 造路由器

请以后务必提前明确通知各组同学这两门课程其他任务的要求，如开学就明确告知是否需要参加考试、完成小作业、完成书面作业等

以后应该对同学的框架设计有更严格的审核, 并且对于往年同学遇到过的坑进行预先提示，比如写错 Checksum, 什么样的状态机不可取等

之后可以考虑调整工作顺序，本学期计网联合实验只有两周时间打通调试，时间相当不充裕。我们在转发模块的时间开销过大，与普通组一起造机太被动，联调的工作量远比纸面看起来更高，以后可以大约五周做转发器，两周做 CPU，留下至少四周调试

5.2 开发工具方面

整个实验过程中最困难的部分是硬件调试，希望之后可以提供较为 “干净” 或标准的环境（如 “干净” 的 linux）方便抓包调试，可以大大加速硬件调试进程，不会因为各种错综复杂的包而陷入混乱

因为交换机的行为不可预测，也给调试带来了一定麻烦，希望交换机能更加透明化，同时最好能让交换机加、删 vlan tag 的过程可视化，方便检查包位置不一致等问题

(填这条的是本组的硬件 QA)，有必要把容易搭建的几个常用调试环境的配置教给大家，并且要求每组至少有一人熟练掌握其配置与使用。比如一台 Windows PC 两虚拟机的回路系统，或者是双 Bird 系统。我虽然不太擅长硬件编程，但是观测问题还是很快的

建议为联合实验的同学准备 RAM 更大的板子和性能更强的 CI(甚至台式机)，这真的是刚需

5.3 软件开发方面

本实验软件部分难度较小，只需要完成普通实验并且手动实现 HAL 即可。不过，可以考虑在一次早期组会上要求同学们归纳清楚 RIP 要做的每

一件事情，这可以减少后续因为误解协议而造成的时间开销

本实验软件更大的难点在于编译，以后可以请杰哥直接将变异内容写进文档

建议软件与造 CPU 同步进行，软件将会成为良好的 CPU 串口调试工具

5.4 硬件开发方面

Verilog 对于同学们来说是一种比较陌生的语言，如果是造 CPU，存在大量可参考的代码，可以快速上手。但是对于计网联合实验来说，从零开始使用 Verilog 搭建硬件路由器在对 Verilog 不熟悉的情况下比较困难，对于我们组来说，最后联调的两周重构了除 CPU 外所有的硬件代码，这都是因为前期对 Verilog 不熟悉，写出的代码风格极差，质量极差。在经历了奋战三星期之后我们组负责硬件的同学对于 Verilog 熟悉之后写出的代码质量显著提高。希望可以将造 cpu 的三周时间前移，造 cpu 所需的知识对于选择联合实验的同学来说自学并不困难。

在硬件开发的过程我们组写了大量的 testbench 用于调试和验证，但是在与其他几组的交流过程中，发现大多数组的模块接口设计相近，如果多个组都有使用 testbench 的需求，那么每个组从零开始搭建各个模块的 testbench 属于重复劳动。希望之后可以提供每个模块的参考 testbench，只要接口制定的相差不多，那么只需要微小改动即可使用。

在路由器开发初期，实现路由表和 ARP 表时，我们组缺乏对于路由器的功能，路由器的工作流程，路由器与交换机之间的关系的理解，导致我们设计路由表和 ARP 表时出现了大问题，路由表没有考虑路由表项中应当保存 VLAN TAG，最后重构代码。因此希望联合实验的开发初期，细致地讲解一下网络和路由器的理论知识，并着重确定一遍工作流程，减少返工。

在奋战三星期后的联调过程中，我们多次发现并修复了之前造 CPU 遗留下来的 bug，在联调阶段发现和修复 cpu 的 bug 代价巨大，以几小时为周期。这是因为造 cpu 的 OJ 上给的测试样例实在是太弱了。希望之后造 CPU 使用的测试样例可以采用龙芯杯的测试样例，这样可以在造 CPU 时解决掉一系列的问题。保证联调阶段 cpu 时完全可信的。

我们认为，造机 CPU 使用的 OJ 非常好，希望计网联合实验可以推出相近的 OJ，支持自己定制包的内容，并从网口打入，我们在调试过程中遇到的一大麻烦是不知道如何向路由器发送指定类型，指定包头，指定 payload

的包。如果从系统方面不好实现这件事，那么希望有一个支持自定义包中每个字节的 OJ

硬件板子的质量直接决定了我们能否做完项目，我们组在联调的过程中不知道板子遇到了什么问题，导致接触不良，并发明了指南针校准式调试法（单手拿着板子在空中绕 8 字可以正常工作，静止就会出问题）。希望之后的硬件板子足够坚固，足够稳定，最好加上足够的保护措施，避免出现我们这样的非技术问题。

在开发的过程中，我们经历过各种各样的资源不足的问题，包括但不限于 LUT 资源不足，Block RAM 不足，slack 超时等等一系列的问题。这些问题主要有三方面的原因：第一，我们对于除了 RAM 之外的资源没有一个清晰的概念，不考虑资源占用的情况下设计出的状态机不得不在后期被重构掉，而且过高的资源占用会带来 implementation 时间急剧上升，导致我们调试周期变长；第二，Vivado 在分析代码并生成对应的 bitstream 的过程中算法奇特无比，修改一行赋值语句在状态机中的位置就可以导致时序错误的 bitstream 变得正常，完全不知道过程中发生了什么；第三，评测使用的路由表规模不具有很好的阶梯性，最高一档规模的路由表基本不可做，我们面向规模设计路由表占用的内存大小时过于束手束脚。希望之后的评测用路由表具有更多的规模梯度，并且在计网联合项目的开始阶段就向大家普及一下 Vivado 的工作方式和资源的概念。

在整个项目中我们为了实现 FIFO，缓存，双工缓冲区等等基础功能，使用了各种各样 IP 核，但是这些 IP 核大多都是在陈嘉杰同学的代码中看到他有在用，然后我们才知道的。因此希望之后可以在计网联合初期向同学们讲解一下可能用到的 IP 核的用法。

在联调阶段我们希望有一间独立的房间进行测试和现场修 bug，评测机总搬来搬去实在是不方便。

根据我们组造 CPU 的经验，以及其他计网联合实验组和熟悉的同学的反馈，我们认为单人造机具有非常高的可行性。

根据我们组造硬件路由器的经验，我们确信计网联合实验可以双人组队甚至单人组队 (单人就够)，有效避免划水的现象。

6 总结致谢

选修计网联合实验是对我们心理承受能力的巨大挑战，更是对我们生理承受能力的巨大挑战。我们熟识了东主楼六点初升的太阳，躺过了 9-320 寒冷的地板。在 8-500 的电流嗡鸣声中地合上血丝遍布的双眼，在 308 的风扇啸叫声中瘫倒在沙发上一整夜。我们对着由于混用有无符号整数而出错的代码放声大笑，抱住“拒绝提速，坚决丢包”的板子欲哭无泪；抓着杰哥通宵评测，烧错 Bitstream 让队友 De 不存在的 BUG 直到天明，费尽心力把队友 buggy 的 Checksum 修成了 BUGGY。我们为了改挂路由器导致转发失效而感到忧心忡忡，我们为了减少逻辑反而时序爆炸而感到心灰意冷。而当我们最终成功之时——却没感到幻想中的欣喜若狂，只感到身心俱疲之后的一丝欣慰罢了。

或许造机反而让我们在日后的工作中对硬件更加忌惮，或许路由器并不足以让我们真正理解互联网发展壮大的内涵，或许做路由器的实际意义并不如把相同的时间投入到一些性价比更高的问题上，或许通宵 Debug 的付出还会影响其他课程的考核。（当然显然这是大三最美好的经历）但或许这就是缘吧？感谢在计网联合实验认识的每一个人。

谢谢大家。