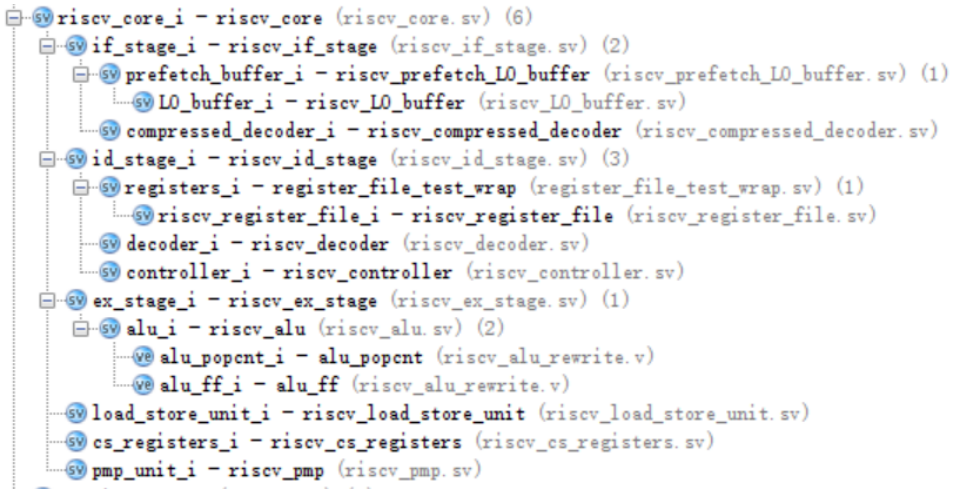
然后是第三处修改：我们的目标是尽量简化代码，所以我把代码里面的当指令长度为 32 的 riscv_prefetch_L0_buffer 给删除了。因为就像之前说的，仿真中所使用的一次预取的长度为 128 位。
一开始以为要删去的是 32 位，还好出错前看了一下。
因为 tp_top 里面的 INSTR_RDATA_WIDTH 才是和 riscv_if_stage 里面的 RDATA_WIDTH 相连的，所以要选用 128 位，测试一下代码。运行成功。
（之一这个地方可以之后回来测试一下当 INSTR_RDATA_WIDTH 取不同的值的时候执行代码的效率怎么样，计算一下 CPI 什么的）
这样的话 riscv_if_stage 就被简化到了只含有两个模块。
RDATA_WIDTH 相关之后还可以改。
更改部分原来在 riscv_if_stage169~169 行

接下来看看 riscv_id_stage，这个模块，首先自然是看到有关 HWLOOP 的部分，看看有什么能删除的。
先别看这个，有点复杂，先看看 riscv_ex_stage，要是它能简化的话就能删去一大堆东西。
第四处修改是在 riscv_ex_stage 里面，首先想得到一个仅仅支持 RV32i 的处理器，就把所有浮点操作全部删了。
删改部分在 riscv_ex_stage 的 336 行。
代码可以运行。试试 test1 也可以正常运行。
第五处修改同样在 riscv_ex_stage 里面，由于只需要支持 rv32i 指令，所以尝试把 mult 模块也一起删去，它的位置是 302 行。
改写这个模块只要注意三个输出就可以了，这三个输出分别是 .result_o (mult_result), .multicycle_o (mult_multicycle_o),.ready_o (mult_ready),
其中 mult_result 不用管，它需要 mult_en 起作用，由于不考虑 mult，所以 mult_en 一定是 0。
multicycle_o 对应的值通过阅读 mult 模块的代码，将其恒设为 0.
同样的阅读代码，mult_ready 也恒设为 0.
出错了，代码跑不了了。
仿真看一下对应部分的三个信号出了什么问题。
Mult_Ready 应当恒设为 1
再跑一次，test1 执行成功，到这里不放心 test2 和 test3，执行一下。
还好都没有出 bug
第六处修改我觉得应该可以在 riscv_alu_div 模块进行。
同样首先删去 FPU 的部分，在 473 行，和 488 行。
接下来试着删除 div 模块。
突然想到，不知道怎么删就跑一下，看谁不动，就给谁删了。
Div 模块删除成功，其他指令跑地很流畅，3 个 test 文件也都可以运行。
第七处删改试着删除 riscv_id_stage 里面的 hwloop_regs_i
删改处为 1377 行的模块，将其改写。
再把 3 个 test 过一遍，防止出错。
删减就差不多完成了，接下来看看还有什么能改写的。

接下来试着针对单条指令进行改写。
首先看第一个模块 cv32e40p_sim_clock_gating,这个模块看波形图输出的两个值一个恒为 1，另一个恒等于 clk，因此完全可以删去这个模块。
为了保险起见，将 3 个 test 文件全跑一遍看看是不是一样的。
第八处删改：把 cv32e40p_sim_clock_gating 删去，换用一条 assign 语句。
接着看回来 riscv_if_stage 文件。看波形图，看到了两个 hwlp 的没有被清理干净的指令，恒为 0，不用管了。
参考波形图，Riscv_prefetch_buffer，不能动了，里面每条指令都有用。
参考波形图，Riscv_l0_buffer 也最好不动，里面是关系程序运行的根本代码。
取指令阶段，能不动就不要动。
第九处看一看 int_control_i 能不能被删掉。
第九处删改，riscv_id_stage 中的 1338 行。
第十处删改：ALU 786 到 938 行。1032 行到 1042 行，1077 行。
第十一处修改：riscv_if_stage 第 312 行。
改写后代码：



改写前代码：

然后就是看看能对仿真过程进行什么样的分析。
（论文的草稿）

首先 RI5CY 是一个四级流水线，它的指令执行流程分别为：取指令，译码，执行指令，送回运算结果。
不同指令在流水线中的执行方向也是不一样的，RV32I 中的指令主要分为六类，分别是 R 型指令，I 型指令，S 型指令，B 型指令，U 型指令和 J 型指令。
我原本只学习过 MIPS 五级流水线的相关知识，五级流水线和 RI5CY 中的四级流水线的最大区别就是四级流水线将 MIPS 五级流水线中的最后两步给融合到一步完成。MIPS 五级流水线的最后两步对应的分别是存入内存和存入寄存器，但四级流水线根据指令的类型，决定最后一步是启动存入寄存器还是存入内存。
除此之外，RI5CY 四级流水线和我曾经做过的 MIPS 五级流水线还有一个很大的不同，就是关于内存的分配：我曾经做过的 MIPS 五级流水线使用的是数据存储器和指令存储器分开来放的哈佛结构，每个存储器的访问和存储都是分开进行的。而 RI5CY 存储器则将两种数据结构放在了同一个 ram 里面，分别通过 riscv_load_store_unit 联系数据部分，riscv_prefetch_buffer 联系指令部分。其中数据和指令是被存放在同一个存储器中的，load/store 指令所操作的地址都是这个内存的绝对地址。
来看指令是如何在内存当中存储的：
内存中数据取每个字节为一个单位，采用小端存储方式，下面是内存在地址 00 到 07 存储着指令 0x0000af13 和指令 0x00000e93 是内存的内容：

04	05	06	07
93	0e	00	00
13	af	00	00
00	01	02	03

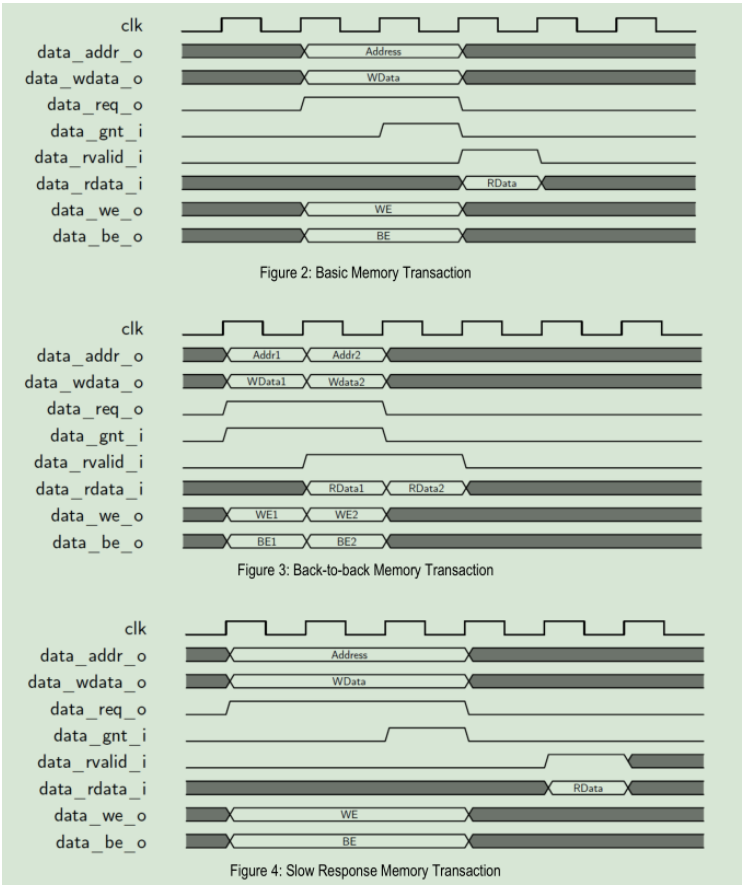
下面执行一段指令，来看看指令具体是怎样执行的：

```
00000013          nop
00000093          li  ra,0
00000113          li  sp,0
00208f33          add t5,ra,sp
```

首先，代码是存放在内存之中的，首先当总的启动线 rst_n 置为 1’b1 之后，riscv_prefetch_buffer 会查看自己有没有已经从内存读取的还没有执行的指令，如果没有，就将 instr_req_o 置为 1’b1，并输出 addr_L0，由于本代码中设置读取指令长度为 128 位，存储器将以 addr_L0 为起始地址读取 4 条指令送到 riscv_prefetch_L0_buffer 里面的 rdata_L0 里面。之后执行每条指令就不用再访问存储器，直接访问 riscv_prefetch_L0_buffer 就可以了。

当代码顺序执行的时候，每当 rdata_L0 中的四条代码执行完成之后，首先将 busy_o 置为 1’b1，意为当前模块正要进行代码更迭，请勿进行其他操作。从内存中读取数据要两个时钟周期，busy_o 在这两个时钟周期内会一直保持在高位，第一个时钟周期将 instr_req_o 设置为高位，向内存说我要请求你的数据了。第二个时钟周期，内存响应，把 instr_gnt_i 设置为高位，意思说你的数据我给你找到了，与此同时更新的 instr_rdata_i 就把更新的四条指令输入进来。通过组合逻辑电路，将 rdata_L0 立即更新为输入的四条指令。这个过程中，由于使用的是组合逻辑电路，流水线的执行不会出现阻塞，会流畅地执行下去。
比较特殊的情况是出现了地址跳转指令的时候，分为两种情况：一个是当对应地址的指令已经被存到了 addr_L0 里面了，还有一种情况是该指令还在内存里面，没有被读取到 addr_L0 中。
当指令已经在 addr_L0 中时，指令执行几乎和没有预取系统的指令系统一样，当出现跳转的时候，输入部分的 branch_i 被设置为高位，于此同时 addr_i 也被设置为目的地址。addr_L0 于一个时钟周期后更新。由于发现对应地址减去之前的起始地址的指令数小于 4，因此将对应指令取出 rdata_L0，其他全部不变。
当指令不在 addr_L0 中时，将会执行内存读取命令，读取过程如之前所述。其中 instr_req_o 的状态主要取决于模块内的 CS 线，它是 enum 数据类型，象征着模块内的状态。

下面看 load_store_unit 部分，RI5CY 作为一款完善的开源中央处理器，它要求不仅仅能够应用于 PULP 这一种嵌入式体系结构，而不同的体系结构所使用的内存的结构大概率是不同的，这就要求 RI5CY 在进行内存数据读写时，不能简简单单地将地址数据输出，类似于 riscv_prefetch_buffer 模块，它需要一个专门的模块来处理内存数据读写的相关信号。
当需要将某个数据从内存读入寄存器的时候，首先要做的是将 busy_o 置为高位，从确定要读入开始，一直到收到内存的确认为止，busy_o 都需要置为高位。在这个过程中，首先，要把 data_req_o 置为高位，当内存收到对应的请求之后，将 data_rvalid_i 置为高位，然后把数据放入 data_rdata_i，两者一同送入 load_store_unit 模块。注意，这个时候，输入的 data_rdata_i 可能并非最终期望得到的数据，data_rdata_i 由 32 位数据构成，但当这条指令是 lb 指令时，它期望的最后数据是 data_rdata_i 中的 8 位数据，加上 24 位的高位符号位拓展。这就要用到另一个四位变量 data_be_o，它决定了最终数据是要取几位。当取 lb 指令时，data_be_o 取 4’b0001。
除了以上的基本访问方式外，在 RI5CY 的用户手册 user_manual.pdf 中，还介绍了 load_store_unit 支持的另外几种数据访问方式：
分别是背对背访问时序和延迟访问时序。



下面看 riscv_id_stage 中的 register_file_test_wrap 和 riscv_register_file 部分，它的读取主要分为三个读端口 a、b、c，和两个写端口 a、b。其中，较常用的读端口是 a 端口和 b 端口，a 端口和 b 端口的读数据部分分别接收指令的不同参数的请求（像是 add r1，r2，r3），a 和 b 端口就负责同时读取 r2 和 r3。写端口 a 和写端口 b 分别负责接收内核执行得出的数据和内存通过 load_store_unit 传入的数据。

Riscv_controller 负责的是整个流水线的控制，它也采用了流水线的常用技术—旁路相关技术，riscv_controller 负责的是输出对应的旁路相关控制器—operand_a_fw_mux_sel_o、operand_b_fw_mux_sel_o、operand_c_fw_mux_sel_o。除此之外还有控制选择 PC 值的控制器—pc_mux_o，以及提供在进行跳转指令和读取内存指令时的阻塞命令—load_stall_o，pref_jump_o，pref_ld_stall_o 等等。

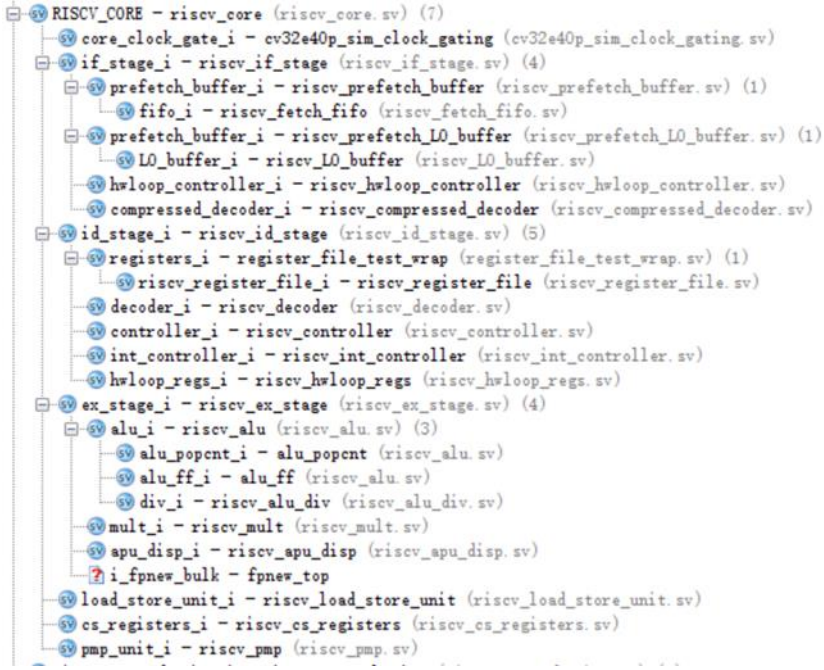
Riscv_compressed_decoder 的功能主要是分辨指令类型，并转化指令格式，它转换指令格式的主要原因是某些指令在执行的时候，它会包括一些立即数操作对象，而这些立即数可能是不连续甚至不按照顺序存放的，这时候就需要这个模块统一化立即数方便之后的操作。某些它在转换指令格式的同时会判断指令是否合法，如果不合法的话会把 illegal_instr_o 置为高位输出。

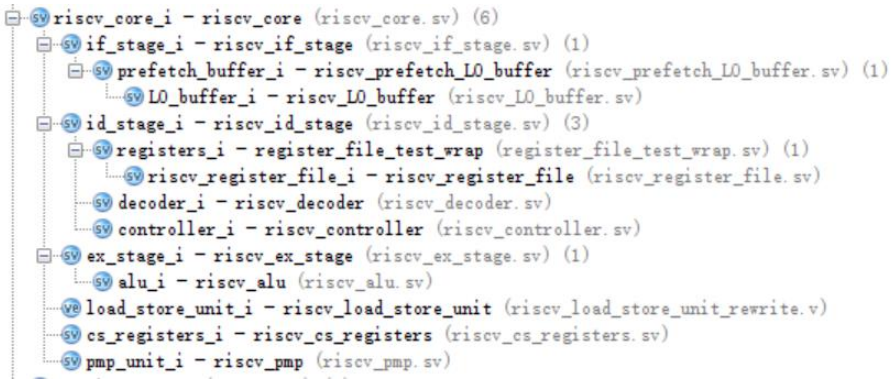
化简后的内核中包括的模块为：

riscv_core//整个处理器的顶层模块
riscv_if_stage//取指令
riscv_prefetch_L0_buffer//预取单元，本实验中一次取 128 位
riscv_id_stage//译码
register_file_test_wrap//register 上层模块，对数据进行一些预处理
riscv_register_file//寄存器
riscv_decoder//译码器
riscv_controller//流水线控制器
riscv_ex_stage//执行阶段
riscv_alu//运算器
riscv_load_store_unit//负责和内存的数据沟通
riscv_cs_register//状态寄存器
riscv_pmp//debug 模块，输出错误

riscv_cs_register 的功能是状态寄存器，它记录了处理器的各种状态，包括有关异常的标记，程序运行的标记，还有性能的指标，举个例子，像是 pc_if_i、pc_id_i、pc_ex_i 就分别保存着对应阶段正在执行的指令的地址。

看看现在的改写后和改写前对比：





（草稿还是放到别的文件来写吧）

5 月 11 号晚

目前论文不包括格式部分的字数，总共写了约五千字，加上自己制作的图片，仿真截图，预计后天能完成初稿。