上回看到 riscv_id_stage，这个模块应该是几个模块里面内容最多的了。

由于模块里面经常出现 hwloop 这一个字段，要想更好的理解程序，应当先对硬件循环多些了解。在一个有关 RI5CY 的文章的网站上面，它对硬件循环指令是这么说的：

硬件循环是 RI5CY 引入的一项特殊机制，目的是提高包含有循环操作的代码的执行效率，如下是一段包含有循环操作的 C 代码。</br>

```
for(i=0;i<100;i++){
    d[i]=a[i]+1;
}
```

如果没有使用硬件循环，其对应的汇编代码如下。</br>

```
  mv x4, 0
  mv x5, 100
Lstart: lw x2, 0(x10)
  addi x10, x10, 4
  addi x2, x2, 1
  sw x2, 0(x11)
  addi x11, x11, 4
  addi x4, x4, 1
  bne x4, x5, Lstart
```

寄存器 x4 存放的是已执行的循环次数，寄存器 x5 存放的是需要执行的总循环次数，在每一次循环操作最后，要将寄存器 x4 加 1，然后与寄存器 x5 比较，如果不相等，那么转移到循环起始位置继续执行。这样的一个过程有两个地方影响效率：</br> （1）每次循环，要将 x4 加 1。</br> （2）每次循环，要做一个分支转移，如果分支预测做的不好，那么会浪费至少一个周期。</br> RI5CY 引入硬件循环以改进循环效率，引入硬件循环后的汇编代码如下。</br>

```
  lp.setupi 100, Lend
  lw x2, 0(x10)
  addi x10, x10, 4
  addi x2, x2, 1
  sw x2, 0(x11)
Lend: addi x11, x11, 4
```

上述代码中涉及到 RI5CY 定制的硬件循环相关指令，在后续章节中将有介绍，此处，读者只需要理解第一条指令 lp.setupi 设置了循环次数，设置了循环段的终止地址，随后就是循环段代码，与没有使用硬件循环的汇编代码相比，此处减少了判断是否循环次数达到的代码，同时减少了分支转移指令，效率因此提高。

RI5CY 里面有一套专门的语句是用来描述这种硬件循环语句的，除了这些指令，在 register 里面也设置了几个用来对硬件循环进行处理的寄存器，这些寄存器的内容包括起始地址，结束地址，循环次数等等。

这个时候回到之前看到的 id 部分，很明显能看到，这个 hwloop 部分对应的几个变量的意思是什么 hwloop_start_int 就是起始地址，而 hwloop_target 是结束地址 hwloop_cnt_int 就是循环次数。

这之后的一小块对应的是跳转模块，包括 jal，cond（？），jalr 语句的跳转。

之后的三个语句，分别对应 operand a,b,c，之后包括立即数模块，这些模块在 ri5cy 里面都是非常成熟的模块，能不改就尽量不改。

接着是 register_file_test_wrap，这个模块第一个注释是 0 地址不可写。这个模块看代码，它

支持三个读端口 raddr_a_i,rdata_a_o，raddr_b_i,rdata_b_o，raddr_c_i,rdata_c_o 和两个写端口 waddr_a 和 waddr_b（其他略），还有几个测试相关的接口。
Register_file_test_wrap 里面连接着另一个子模块 riscv_register_file，这个模块只连接了刚才提到的两个写和三个读，首先，这个寄存器不仅支持整型，还支持浮点数（由于这个项目仅考虑 RV32I，浮点部分尽量不作考虑）
对于第一个模块，由于浮点信号量设做 parameter，即为 0，所以 output 值 rdata_a_o,rdata_b_o,rdata_c_o 对应各自地址的 mem 值。这一小个模块就是 register 对应的输出模块。
紧接着的是两个写模块的部分，它使用的是两个 for 语句，遍历一遍，找到地址的时候，将对应地址的内存里面的数值改为要写入的数据，否则置零。并不是，看了之后的代码发现这里并不是直接写入的代码，而是通过一个 32 位的信号将要写入的位置标示出来。
然后是 rst_n 对应的初始化语句。
真正的写入语句在这里，由于不考虑 FPU=1 的情况，它的处理方式就和普通的 register 写没什么区别。
回过头来看 register_file_test_wrap 就和它的名字一样主要作用是一个 test。注意，这个时候回头看 id 模块里面的模块调用发现 bist 系列的线式没有连到外面的。
接着是 decoder 模块，这个模块对应的是译码器，里面的线路是十分的复杂。
首先看输入，有一大波控制信号，这个模块实在太大了，由于它和其他的模块已经完成了比较好的逻辑自恰，在这里为了避免生出额外的 bug，我们尽量只做一些粗略的阅读。
在这里就要看模块调用部分的注解了，这个模块大致信号作用是什么，首先是一部分控制相关信号，紧接着的是从 if 流水线部分来的包括 instr 信号，后面的是 alu 和 mul 信号，紧接着是浮点信号 FPU 系列，而后是寄存器信号和数据总线信号，最后是硬件循环指令信号和跳转信号。
而后的一个模块是控制器系列，包括了信号控制器，整型控制器和硬件循环控制器，最后是连接到下一循环的部分信号。
执行阶段的语句相对来讲简单一些，这个阶段的输入输出信号大致上有以下这些：
首先是 idstage 弄出来的 ALU 控制信号，还有乘法控制信号，浮点数控制信号，APU 信号，寄存器操作信号和数据，这个模块一定和之前的模块一样都是相当自洽的模块，这里不做分析。
还是看看 riscv_core 的输入输出信号（主要还是输入）
Data_memory 和 instr_memory 之前已经很明白了，这个时候来分析一下其他的信号
Rst_n 和 clk 不用讲，test_en_i 这个信号追踪到最后发现这个信号基本上什么都没干，可以不管。
Clock_en_i 参与构成了 clock_en 指令，
clock_en = PULP_CLUSTER ? clock_en_i | core_busy_o : irq_pending | debug_req_i | core_busy_o;
其中 PLUP_CLUSTER 常年置为 1，也就是说当 clock_en_i 或者 core_busy_o 为 1 时 clock_en 为 1，看看 clock_en 干了什么。
Clock_en 主要参与了模块 cv32e40p_sim_clock_gating 中的 en_i，在这个模块里面，它应当和 debug 有关，建议仿真的时候试试 0 还是 1.。
fregfile_disable_i 主要和 id 里面的 fregfile_ena 有关，随便怎么取
接着是 32 位的 boot_addr_i 线，这个线的主要功能是启动地址，它首先和 if_stage 相连，参与了一个 fetch_addr_n 的构建，fetch_addr_n 是 buffer_prefetch 里面的 addr_i，接下来看决定它的 pc_mux_i，总之这个信号可以先置为 0，出错的话到时候再看。

按同样的方法，其他的信号也全部置为 0。
外层的 core_region 调用 riscv_core 的时候有相当多的信号是空的，这些信号尽量不用做考虑。
接下来还是看 pulpino 里面的 core_region。
Core_region 调用指令和数据存储器的位置都在一个叫 ram_mux 的模块里面，两者分别分得一个 ram_mux。
一个 instr_ram 里面有很多东西，包括 instr_ram_wrap，axi_mem_if_sp_ram，ram_mux，由它们延伸出来的模块包括 sp_ram_wrap，boot_rom_wrap，axi_mem_if_sp，xilinx_mem_8192x32，sp_ram_bank_i，sp_ram，boot_code_i。其中，为了简便考虑，我们仅仅追踪需要连到 riscv_core 上的 instr 指令们。

```
output logic              data_req_o,
input    logic            data_gnt_i,
input    logic            data_rvalid_i,
output logic              data_we_o,
output logic [3:0]    data_be_o,
output logic [31:0] data_addr_o,
output logic [31:0] data_wdata_o,
input    logic [31:0] data_rdata_i,
```

```
output logic                                  instr_req_o,
input    logic                                instr_gnt_i,
input    logic                                instr_rvalid_i,
output logic                          [31:0] instr_addr_o,
input    logic [INSTR_RDATA_WIDTH-1:0] instr_rdata_i,//31:0
```

先看几个 input，instr_gnt_i 连在了 core_instr_gnt 上面，而 core_instr_gnt 又连在了 port1_gnt_o 上面，port1_gnt_o 连入 ram_mux 模块，在这个模块里面，设计 port1_gnt_o 的值的模块是一个组合逻辑电路 always_comb，它由线路 port1_req_i 决定，当它为 1 时设置 port1_gnt_o 为 1，否则为 0，其中，port1_req_i 连在了 core_instr_req 上面，类似的可以类比到 data 的 gnt 和 req 上面。
对于 data_gnt_i 和 data_req_o，data_gnt_i 连的是 core_lsu_gnt，data_req_o 链接的是 core_lsu_req，发现这两个的代码有很多相似性，将他们黏贴到这里

```
always_comb
begin
   lsu_resp_NS = lsu_resp_CS;
   core_lsu_gnt = 1'b0;
   if (core_axi_req)
   begin
      core_lsu_gnt = core_axi_gnt;
      lsu_resp_NS = AXI;
   end
   else if (core_data_req)
   begin
      core_lsu_gnt = core_data_gnt;
      lsu_resp_NS = RAM;
```

```
        end
    end
```

还有对应 instr 的 gnt 和 req 的：

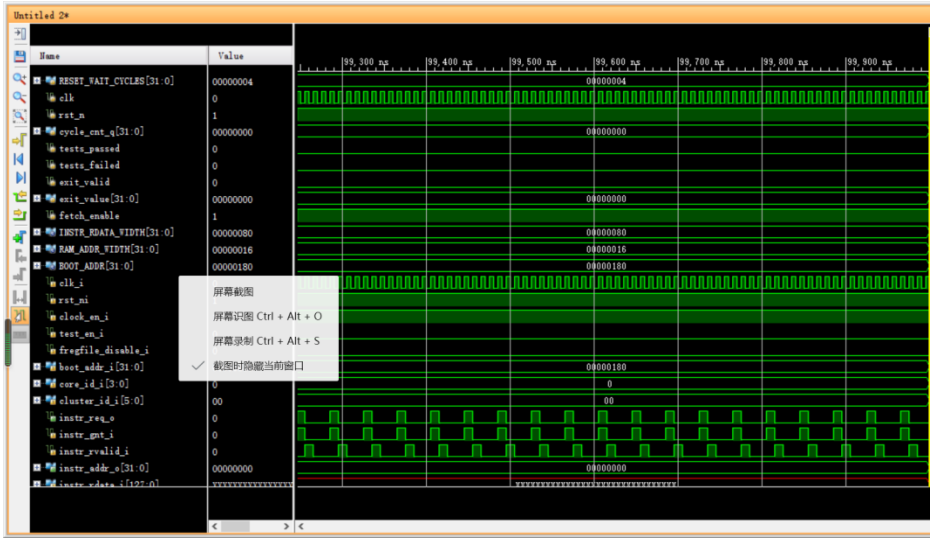
```
  always_comb
  begin
    port0_gnt_o = 1'b0;
    port1_gnt_o = 1'b0;
    if(port0_req_i)
    //port0_req_i 连在了 axi_instr_req 上面，
      port0_gnt_o     = 1'b1;
    else if(port1_req_i)
      port1_gnt_o     = 1'b1;
  end
```

这几天干的事情总结下就是想方设法地进行仿真测试，沿着上面的思路我发现还是由于对 sv 语言不了解，我转变了下思路，想出另外两个方法得到仿真图。

首先是直接使用了源码中的测试文件，但在使用过程中出现了各种各样的 bug，主要原因是目前使用平台版本不支持目标源码里面大量的 sv 语法，首先想到的是平台更新，但更新项目不支持，我为了 vivado 注册的账号连官网都登不上了，对方总是给我寄邮件表示登录失败，请联系一个邮箱。

然后就是尝试下载别的仿真平台，尝试了比较新的 vivado 版本和 quartus 都失败了（不在学校网速太慢），quartus 下载成功了但完全不会用，保留重新回去看 vivado2015.4

到这里我就开始试着改代码了，将无关紧要的 sv 语法代码删去（主要是看前后代码决定它是不是相关），一次次 debug 之后，忠于可以直接用 test 文件直接得到仿真图，但这个仿真图是没有任何内容的，原因如下，这是这条路正要解决的东西：



无内容的仿真图如上

```
initial begin: load_prog
    automatic string firmware;
    automatic int prog_size = 6;

    if($value$plusargs("firmware=%s", firmware)) begin
        if($test$plusargs("verbose"))
            $display("[TESTBENCH] %t: loading firmware %0s ...",
                     $time, firmware);
        $readmemh(firmware, riscv_wrapper_i.ram_i.dp_ram_i.mem);

    end else begin
        $display("No firmware specified");
        //$finish;
    end
end
```

出错的部分代码，$value$plusargs，$test$plusargs 可以改动，关键在于 $readmemh 函数，这个函数的作用就是将对应文件里面的存储器数据读入到对应的 mem 里面，目前看，这个东西已解决问题就能解决一大半。

另一条路线是使用源文件里面的 Makefile 文件，由于这也是我第一次使用这个文件，中间安装了好几个插件，半查半试地进行到了如下步骤：

```
选择管理员: C:\Windows\system32\cmd.exe
E:\2020\cv32e40p-master-8>cd tb

E:\2020\cv32e40p-master-8\tb>cd core

E:\2020\cv32e40p-master-8\tb\core>mingw32-make
verilator --cc --sv --exe \
          \
        --Wno-lint --Wno-UNOPTFLAT \
        --Wno-MODDUP +incdir+../../rtl/include --top-module \
        tb_top_verilator mm_ram.sv tb_top_verilator.sv amo_shim.sv dp_ram.sv riscv_wrapper.sv fpnew/src/fpnew_pkg.sv ../
../rtl/include/apu_core_package.sv ../../rtl/include/riscv_defines.sv ../../rtl/include/riscv_tracer_defines.sv ../../rt
l/include/../../tb/tb_riscv/include/perturbation_defines.sv ../../rtl/riscv_controller.sv ../../rtl/riscv_alu.sv ../../r
tl/riscv_apu_disp.sv ../../rtl/riscv_pmp.sv ../../rtl/riscv_alu_basic.sv ../../rtl/riscv_ex_stage.sv ../../rtl/riscv_int
_controller.sv ../../rtl/riscv_LO_buffer.sv ../../rtl/register_file_test_wrap.sv ../../rtl/cv32e40p_sim_clock_gating.sv
../../rtl/riscv_mult.sv ../../rtl/riscv_hwloop_regs.sv ../../rtl/riscv_prefetch_LO_buffer.sv ../../rtl/riscv_alu_div.sv
../../rtl/riscv_fetch_fifo.sv ../../rtl/riscv_compressed_decoder.sv ../../rtl/riscv_core.sv ../../rtl/riscv_id_stage.sv
../../rtl/riscv_load_store_unit.sv ../../rtl/riscv_register_file.sv ../../rtl/riscv_cs_registers.sv ../../rtl/riscv_deco
der.sv ../../rtl/riscv_prefetch_buffer.sv ../../rtl/riscv_tracer.sv ../../rtl/riscv_hwloop_controller.sv ../../rtl/riscv
_if_stage.sv ../../tb/tb_riscv/riscv_random_stall.sv ../../tb/tb_riscv/riscv_random_interrupt_generator.sv \
        tb_top_verilator.cpp --Mdir cobj_dir \
        -CFLAGS "-std=gnu++11 -O2" \
        -Wno-BLKANDNBLK
Can't locate Pod/Usage.pm in @INC (you may need to install the Pod::Usage module) (@INC contains: /usr/lib/perl5/site_pe
rl /usr/share/perl5/site_perl /usr/lib/perl5/vendor_perl /usr/share/perl5/vendor_perl /usr/lib/perl5/core_perl /usr/shar
e/perl5/core_perl) at /e/2020/verilator-4.032/bin/verilator line 17.
BEGIN failed--compilation aborted at /e/2020/verilator-4.032/bin/verilator line 17.
Makefile:162: recipe for target 'testbench_verilator' failed
mingw32-make: *** [testbench_verilator] Error 2

E:\2020\cv32e40p-master-8\tb\core>
```

目前出现的问题是不支持 verilator，原因应该是我的电脑没有安装 perl，现在 perl 正在下载，还有四十分钟就好了。

Perl 的问题下载后的问题还没有解决，但是第一种方法似乎更加的好，我直接在指令存储器里面加入了指令 00003237　　lui　　x4,0x3

根据下图，4 号寄存器被置为 0x00003000,代码仿真成功