

SSRV 全解析

SSRV 志在推出一个高效的处理器架构，欢迎批评指正。加我微信号
rvlite 进群讨论

目录

仿真篇	4
SSRV 仿真运行	4
SCR1 结构	7
SSRV 借用 SCR1	10
SSRV 运行初步	15
SSRV 整体浏览	21
架构篇	26
RISC 指令处理	26
RV32IMC 的指令分类	36
SSRV 的系统架构	37
四个功能模块	40
第一级缓冲器: instrbits	46
第二级缓冲器: schedule	56
第三级缓冲器: mprf	68
第三级缓冲器: membuf	72
四个缓冲器的关系	78
FPGA 篇	81
嵌入 SCR1 的 SSRV	81
FPGA 工程和运行	85

FPGA 综合评估	88
-----------------	----

仿真篇

SSRV 仿真运行

本文是为 RISC-V 指令集的开源处理器的设计爱好者所写。SSRV 会为爱好者们提供一个仿真平台。首先，请下载 Github 上的项目文件，项目的网址为：

<https://github.com/risc-lite/SuperScalar-RISC-V-CPU>。

进入 `scr1/sim/` 这一子文件夹。下面以 Modelsim 仿真软件为例运行仿真。

如果是首次运行，可以敲击命令：

```
vlib work
```

这是建立一个 `work` 的运行目录。

然后是：

```
source compile.do
```

这是一个在 `work` 目录下，编译所有相关文件的命令。在执行完这条指令后，会在 `work` 目录下生成各种 `module` 实体。

接着：

```
source sim.do
```

这是一条进入仿真状态的命令。然后运行下面的命令，即可执行仿真：

```
run -all
```

在这条指令的执行下，仿真软件开始执行，那么在 Transcript 窗口下，会出现运行信息。

如果是其他仿真工具，需要手动转译这两个 `.do` 文件。其中 `compile.do` 文件里面罗列了用到的源文件和引用文件。

```
vlog -work work -mfcu -sv "+nowarnSVCHK" +incdir+../src/includes/ +incdir+../rtl/
\
../src/pipeline/scr1_pipe_hdu.sv ../src/pipeline/scr1_pipe_tdu.sv ../src/pipeline/scr1
_ipic.sv ../src/pipeline/scr1_pipe_csr.sv ../src/pipeline/scr1_pipe_exu.sv ../src/pipe
line/scr1_pipe_ialu.sv ../src/pipeline/scr1_pipe_idu.sv ../src/pipeline/scr1_pipe_ifu.
sv ../src/pipeline/scr1_pipe_lsu.sv ../src/pipeline/scr1_pipe_mprf.sv ../src/pipeline/
scr1_pipe_top.sv ../src/core/primitives/scr1_reset_cells.sv ../src/core/primitives/scr
1_cg.sv ../src/core/scr1_clk_ctrl.sv ../src/core/scr1_tapc_shift_reg.sv ../src/core/sc
r1_tapc.sv ../src/core/scr1_tapc_synchronizer.sv ../src/core/scr1_core_top.sv ../src/c
ore/scr1_dm.sv ../src/core/scr1_dmi.sv ../src/core/scr1_scu.sv ../src/top/scr1_dmem_ro
uter.sv ../src/top/scr1_imem_router.sv ../src/top/scr1_dp_memory.sv ../src/top/scr1_tc
m.sv ../src/top/scr1_timer.sv ../src/top/scr1_dmem_ahb.sv ../src/top/scr1_imem_ahb.sv
../src/top/scr1_top_ahb.sv ../src/top/scr1_mem_axi.sv ../src/top/scr1_top_axi.sv ../sr
c/pipeline/scr1_tracelog.sv ../src/tb/scr1_memory_tb_ahb.sv ../src/tb/scr1_top_tb_ahb.
sv \
../..rtl/ssrv_top.v \
../..rtl/sys_csr.v \
../..rtl/mul.v \
../..rtl/mprf.v \
../..rtl/membuf.v \
../..rtl/alu.v \
../..rtl/schedule.v \
../..rtl/instrman.v \
../..rtl/instrbits.v \
../..rtl/predictor.v \
../..rtl/lsu.v \
../..testbench/tb_ssrv.v
```

需要注意的是两个 include 文件夹：../src/includes/和../rtl/。在这两个目录下存放的是.sv 或.v 引用到的 include 文件所在目录。

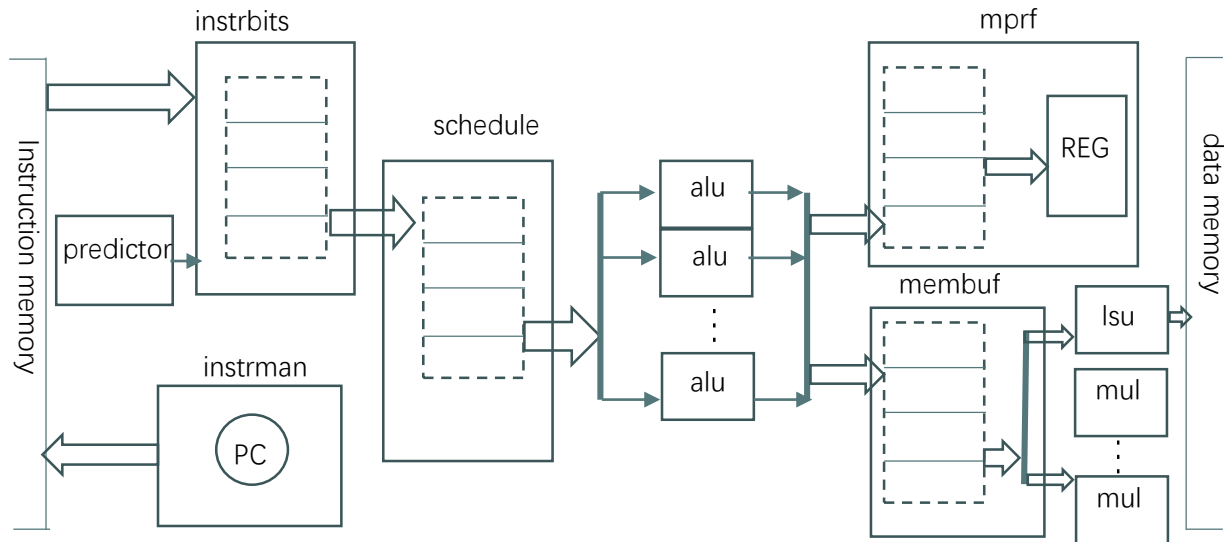
实体文件包括两类，一类是以.sv 结尾，它会引用../src/includes/内的 include 文件；一类是以.v 结尾，它会引用../rtl/内的 include 文件。

sim.do 是一个很简单的命令行：

```
vsim work.scr1_top_tb_ahb work.tb_ssrv -voptargs=+acc
```

它会调用.sv 结尾的文件的顶层模块：scr1_top_tb_ahb 和.v 结尾的文件的顶层 tb_ssrv，这两个顶层同时运行，进行仿真。

SSRV 运行起来非常简单。但是 SSRV 的魅力并不在于运行一次仿真，而在于可以通过修改参数来实现超标量和乱序运行。



上图是 SSRV 的模块框图。它的主体结构是四个缓存器，通过配制这四个缓存器的输入、输出和大小，控制指令并行运行的程度。

打开：rtl/define_para.v 文件，找到下面的语句：

```
//-----
//"instrbits" buffer
//-----
//The bus width of AHB-lite or AXI: 1 --- 32 bits 2 --- 64 bits, 4 ---- 128 bits.
//Only 2^x is allowed. If it is bigger than 1, WIDE_INSTR_BUS should be defined.
`define BUFFER0_IN_LEN 4
//How many words it holds: 1 --- 32 bits, 2 --- 64 bits. Any integer
`define BUFFER0_BUF_LEN (2*`BUFFER0_IN_LEN)
//How many instructions are generated to the next stage. 1 --- 1 instr, 2 -- 2 instr,
//Any integer
`define BUFFER0_OUT_LEN 4

//-----
//"schedule" buffer
//-----
//no IN_LEN, because it equals to BUFFER0_OUT_LEN
//How many instructions are kept. 1 -- 1 instr, 2-- 2 instr, Any integer
`define BUFFER1_BUF_LEN 7
//How many instructions are generated for multiple exec units. 1-- 1 instr, 2 -- 2
//instr, Any integer
`define BUFFER1_OUT_LEN 3

//-----
//"membuf" buffer
//-----
//no IN_LEN, because it equals to BUFFER1_OUT_LEN
//How many MEM instructions are kept. 1 -- 1 instr, 2-- 2 instr, Any integer
`define BUFFER2_BUF_LEN 8/(2*`BUFFER1_OUT_LEN)
```

```
//How many LSU/MUL instructions to be retired at the same clock, it could be 1 or 2
instructions.
`define BUFFER2_OUT_LEN          2

//-----
//"mprf" buffer
//-----
//no IN_LEN, because it equals to BUFFER1_OUT_LEN
//How many ALU instructions are kept. 1 -- 1 instr, 2-- 2 instr, Any integer.
(2*`BUFFER1_OUT_LEN) is recommended
`define BUFFER3_BUF_LEN          8//(2*`BUFFER1_OUT_LEN)
//How many ALU instructions are allowed to write to the register file in the same
cycle, 1 -- 1 instr, 2-- 2 instr, Any integer, BUFFER1_OUT_LEN is recommended
`define BUFFER3_OUT_LEN          2//`BUFFER1_OUT_LEN
```

这是用来配置四个缓存器，需要依次为这四个缓存器配置输入、容量和输出。

在设置完毕后，点击保存，然后运行下面的语句：

```
source compile.do; restart; run -all
```

或者：

```
source compile.do; source sim.do; run -all
```

这相当于根据新的参数重新运行了一次仿真。由于系统自带了性能测试用例，那么在重新运行时，可以检测本次新的配置到底能跑出什么样的性能分数。

SCR1 结构

由于 RISC-V 的开放性，在 Github 上出现了众多 RV32 指令集的处理核。在这其中，SSRV 选择了 Syntacore 公司的 SCR1 处理器内核作为开发平台。之所以选择它，主要是因为它提供了既简单又完备的开发工具。

只需按照 <https://github.com/syntacore/scr1> 上的安装指示，一条条做下来，那么你可以获得一个 C 代码编译工具。这个工具可以对 scr1 自带的测试用例进行编译，获得可以仿真用的二进制指令文件。你只需要选择一个 Systemverilog 的仿真工具，对 scr1 的各个 .sv 文件进行编译，那么就可以架起完整的仿真流程。scr1 的 .sv 文件会自动读取测试用例的二进制指令文件，然后让整个仿真动起来，最后输出 log 信息。

对于 SystemVerilog/Verilog 开发人员来说，scr1 给出的这个开发包太方便了。选择 scr1 的重要原因是 scr1 自带 Dhrystone 和 CoreMark 两大测试用例，无需 Verilog 开发者再对这两个测试用例进行改造和编译。

从 Verilog 开发者的角度来看，它的 SystemVerilog 代码是完备的，它符合我们通常意义上的结构——也就是包含一个顶层 testbench 文件来引导整个仿真流程。

```
../src/pipeline/scr1_pipe_hdu.sv ../src/pipeline/scr1_pipe_tdu.sv ../src/pipeline/scr1_ipic.sv ../src/pipeline/scr1_pipe_csr.sv ../src/pipeline/scr1_pipe_exu.sv ../src/pipeline/scr1_pipe_ialu.sv ../src/pipeline/scr1_pipe_idu.sv ../src/pipeline/scr1_pipe_ifu.sv ../src/pipeline/scr1_pipe_lsu.sv ../src/pipeline/scr1_pipe_mprf.sv ../src/pipeline/scr1_pipe_top.sv ../src/core/primitives/scr1_reset_cells.sv ../src/core/primitives/scr1_cg.sv ../src/core/scr1_clk_ctrl.sv ../src/core/scr1_tapc_shift_reg.sv ../src/core/scr1_tapc.sv ../src/core/scr1_tapc_synchronizer.sv ../src/core/scr1_core_top.sv ../src/core/scr1_dm.sv ../src/core/scr1_dmi.sv ../src/core/scr1_scu.sv ../src/top/scr1_dmem_router.sv ../src/top/scr1_imem_router.sv ../src/top/scr1_dp_memory.sv ../src/top/scr1_tcm.sv ../src/top/scr1_timer.sv ../src/top/scr1_dmem_ahb.sv ../src/top/scr1_imem_ahb.sv ../src/top/scr1_top_ahb.sv ../src/top/scr1_mem_axi.sv ../src/top/scr1_top_axi.sv ../src/pipeline/scr1_tracelog.sv ../src/tb/scr1_memory_tb_ahb.sv ../src/tb/scr1_top_tb_ahb.sv \
```

上面在 compile.do 里面出现的.sv 文件。它们是 scr1 在选择 AHB-lite 接口格式下的仿真和 RTL 文件合集。如果选择 AXI 接口，那么在../src/tb/目录下的../src/tb/scr1_memory_tb_ahb.sv ../src/tb/scr1_top_tb_ahb.sv 替换成对应 axi 为结尾的文件即可。

如果选择 AHB-lite 为总线接口，scr1_top_tb_ahb.sv 是整个仿真的顶层。它负责生成时钟，加载二进制指令文件和检查仿真结果。首先，它会例化 RTL 的顶层：

```
scr1_top_ahb i_top (
    // Reset
    .pwrup_rst_n      (rst_n      ),
    .rst_n            (rst_n      ),
    .cpu_rst_n        (rst_n      ),
    `ifdef SCR1_DBGC_EN
    .ndm_rst_n_out    (),
    `endif // SCR1_DBGC_EN

    // Clock
    .clk              (clk        ),
    .rtc_clk           (rtc_clk    ),

    // Fuses
    .fuse_mhartid     (fuse_mhartid ),
    `ifdef SCR1_DBGC_EN
    .fuse_idcode       (`SCR1_TAP_IDCODE ),
```



```

`endif // SCR1_DBG_C_EN

    // IRQ
`ifdef SCR1_IPIC_EN
    .irq_lines          (irq_lines          ),
`else // SCR1_IPIC_EN
    .ext_irq            (ext_irq            ),
`endif // SCR1_IPIC_EN
    .soft_irq           (soft_irq           ),

    // DFT
    .test_mode          (1'b0              ),
    .test_rst_n         (1'b1              ),

`ifdef SCR1_DBG_C_EN
    // JTAG
    .trst_n             (trst_n            ),
    .tck                (tck               ),
    .tms                (tms               ),
    .tdi                (tdi               ),
    .tdo                (tdo               ),
    .tdo_en             (tdo_en            ),
`endif // SCR1_DBG_C_EN

    // Instruction Memory Interface
    .imem_hprot         (imem_hprot        ),
    .imem_hburst        (imem_hburst       ),
    .imem_hsize         (imem_hsize        ),
    .imem_htrans        (imem_htrans       ),
    .imem_hmastlock     (),
    .imem_haddr         (imem_haddr        ),
    .imem_hready        (imem_hready       ),
    .imem_hrdata        (imem_hrdata       ),
    .imem_hresp         (imem_hresp        ),

    // Data Memory Interface
    .dmem_hprot         (dmem_hprot        ),
    .dmem_hburst        (dmem_hburst       ),
    .dmem_hsize         (dmem_hsize        ),
    .dmem_htrans        (dmem_htrans       ),
    .dmem_hmastlock     (),
    .dmem_haddr         (dmem_haddr        ),
    .dmem_hwrite        (dmem_hwrite       ),
    .dmem_hwdata        (dmem_hwdata       ),
    .dmem_hready        (dmem_hready       ),
    .dmem_hrdata        (dmem_hrdata       ),
    .dmem_hresp         (dmem_hresp        )
);

```

scr1 的 RTL 文件分为三层，分别为：top 层、core 层和 pipeline 层。其中 top 层是顶层，pipeline 层是最底层。在最底层中，实现了内核的基本功能，在其他两层，都有相应的功能添加。

在 top 顶层中，它的指令和数据接口都是采用 AHB-lite 形式。在仿真时，总线访问是没有延迟的，但可以通过下面的语句来添加延迟：

```
imem_req_ack_stall = 32'hffffffff;
dmem_req_ack_stall = 32'hffffffff;
```

如果是全 F，那么表示没有延迟。如果是全 0，那么表示任意延迟。如果是其他数字，它会从低到高循环读取比特，如果是 0 表示这个时钟延迟，如果为 1 表示不延迟，如此循环重复。

可以通过修改这两个变量的配置来对总线的延迟进行修改。

scr1 的处理器内核需要运作，首先需要加载预编译的二进制指令文件。所有的二进制指令文件存放于 scr1/build 目录下，由文件 scr1/build/test_info 列出。打开 scr1/build/test_info 可以看到，每一行代表一个二进制文件名。testbench 文件会逐行读取每一个二进制文件的 hex 格式，送入 scr1_memory_tb_ahb 内的存储器。

为了兼顾单个文件和全部文件的仿真，在 test_info 文件内使用符号#来屏蔽不需要仿真的文件。这是 SSRV 添加的新功能，在 scr1 是没有的。

scr1 自带了单个指令的测试用例：riscv_isa 和 riscv_compliance，以及性能方面的测试用例：dhrystone21 和 coremark。由于单个指令的用例太多，只保留了关于乘除法的测试用例。至于性能方面的测试用例，根据 IMC 代表的指令子集的选择，分为 I、IC 和 IMC 进行编译。而 dhrystone21 又可因为所加编译条件的不同，分为 max 和 noinline 两类。前者代表 best effort，代表最强编译；后者代表不允许 inline 编译选项，代表通常编译。这两种编译结果出来的性能测试分数差别很大，很多处理器会列出这两种编译的测试结果。

SSRV 借用 SCR1

scr1 有完整的开发环境，SSRV 借用它的开发体系，它的实现方法是：替代 scr1 最核心的处理器功能。

和 scr1 的顶层文件：scr1_top_tb_ahb.sv 并列的一个顶层文件 tb_ssr.v，它会例化 SSRV 的 RTL 文件。这个顶层文件 tb_ssr.v 在 testbench/目录下，它会接替 pipeline 层的运作。scr1 的 pipeline 层的处理器功能由 SSRV 的 RTL 文件取代。

下面是 pipeline 层在 core 层的例化：

```

scr1_pipe_top i_pipe_top (
    // Control
    .pipe_rst_n          (core_rst_n          ),
`ifdef SCR1_DBG_C_EN
    .pipe_rst_n_qlfy     (core_rst_n_qlfy     ),
    .dbg_rst_n          (hdu_rst_n          ),
`endif // SCR1_DBG_C_EN
`ifndef SCR1_CLKCTRL_EN
    .clk                 (clk                 ),
`else // SCR1_CLKCTRL_EN
    .clk                 (clk_pipe           ),
    .sleep_pipe          (sleep_pipe         ),
    .wake_pipe           (wake_pipe          ),
    .clk_alw_on          (clk_alw_on         ),
    .clk_dbg_c           (clk_dbg_c          ),
    .clk_pipe_en         (clk_pipe_en        ),
`endif // SCR1_CLKCTRL_EN
    // Instruction memory interface
    .imem_req            (imem_req           ),
    .imem_cmd            (imem_cmd           ),
    .imem_addr           (imem_addr          ),
    .imem_req_ack        (imem_req_ack       ),
    .imem_rdata          (imem_rdata         ),
    .imem_resp           (imem_resp          ),
    // Data memory interface
    .dmem_req            (dmem_req           ),
    .dmem_cmd            (dmem_cmd           ),
    .dmem_width          (dmem_width         ),
    .dmem_addr           (dmem_addr          ),
    .dmem_wdata          (dmem_wdata         ),
    .dmem_req_ack        (dmem_req_ack       ),
    .dmem_rdata          (dmem_rdata         ),
    .dmem_resp           (dmem_resp          ),
`ifdef SCR1_DBG_C_EN
    // Debug interface:
    // DM <-> Pipeline: HART Run Control i/f
    .dm_active           (dm_active          ),
    .dm_cmd_req          (dm_cmd_req         ),
    .dm_cmd              (dm_cmd            ),
    .dm_cmd_resp         (dm_cmd_resp        ),
    .dm_cmd_rcode        (dm_cmd_rcode       ),
    .dm_hart_event       (dm_hart_event      ),
    .dm_hart_status      (dm_hart_status     ),
    // DM <-> Pipeline: Program Buffer - HART instruction execution i/f
    .dm_pbuf_addr        (dm_pbuf_addr       ),
    .dm_pbuf_instr       (dm_pbuf_instr      ),
    // DM <-> Pipeline: HART Abstract Data regs i/f
    .dm_dreg_req         (dm_dreg_req        ),
    .dm_dreg_wr          (dm_dreg_wr         ),
    .dm_dreg_wdata       (dm_dreg_wdata      ),
    .dm_dreg_resp        (dm_dreg_resp       ),
    .dm_dreg_fail        (dm_dreg_fail       ),
    .dm_dreg_rdata       (dm_dreg_rdata      ),
    //
    .dm_pc_sample        (dm_pc_sample       ),
`endif // SCR1_DBG_C_EN

```

```

    // IRQ
`ifdef SCR1_IPIC_EN
    .irq_lines          (irq_lines          ),
`else // SCR1_IPIC_EN
    .ext_irq            (ext_irq            ),
`endif // SCR1_IPIC_EN
    .soft_irq           (soft_irq           ),
    .timer_irq          (timer_irq          ),
    .mtime_ext          (mtime_ext          ),

    // Fuse
    .fuse_mhartid       (fuse_mhartid       )
);

```

里面根据设定有各种接口，除了时钟和复位是必须的，有两类接口是这类处理器内核一定具有的：那就是指令存储器接口和数据存储器接口。其他接口都是为中断和其他功能使用。

下面是指令存储器接口和数据存储器接口：

```

// Instruction memory interface
.imem_req      (imem_req      ),
.imem_cmd      (imem_cmd      ),
.imem_addr     (imem_addr     ),
.imem_req_ack  (imem_req_ack  ),
.imem_rdata    (imem_rdata    ),
.imem_resp     (imem_resp     ),
// Data memory interface
.dmem_req      (dmem_req      ),
.dmem_cmd      (dmem_cmd      ),
.dmem_width    (dmem_width    ),
.dmem_addr     (dmem_addr     ),
.dmem_wdata    (dmem_wdata    ),
.dmem_req_ack  (dmem_req_ack  ),
.dmem_rdata    (dmem_rdata    ),
.dmem_resp     (dmem_resp     ),

```

这两个接口的连接方式类似，这些接口分为两类：

- req 发送通道：包括输出 req、cmd、width、addr、wdata 和输入 req_ack，由于指令存储器的特殊性，包含的输出接口会少一些。在 req 为高时，附属信号 cmd、width、addr、wdata 给出本次数据存储操作的其他信息，如果接收方能接收，那么回应 req_ack 为高，可以再发送下一次 req；如果回应 req_ack 为低，表示 req 没有被接收。

- 应答通道：包括输入 `rdata` 和 `resp`。`resp` 会给出存储器的应答信息，它包括三种：1，存储器没有准备完毕；2，存储器已经准备完毕，结果为正确，此时如果是读操作的应答，读数据位于 `rdata`；3，存储器已经准备完毕，结果为错误。

通过这两个通道，处理器不断发送 `req` 请求，直到 `req_ack` 为低为止；然后存储器对处理器的应答通过应答通道依次送出。

SSRV 在这个接口的基础上进行修正，取消了 `req_ack` 信号。也就是 `req` 请求在发出后，只有在 `resp` 为正确应答时，才能送出第二个 `req` 请求，不需要等待 `req_ack` 信号。简而言之，如果以前没有发出 `req` 请求，那么可以随时发出 `req` 请求；如果以前发送了 `req` 请求，只有在前一个请求给出正确应答后，才能发出第二个 `req` 请求。

经过修正后，对于存储器的访问主动权由处理器掌握，它可以顺序发出请求，根据结果来决定是否送出第二个请求。

存储器接口信号的定义可以参看：`scr1/src/includes/scr1_memif.svh`。这里简略说明，`cmd` 表示读写定义，1 表示写，0 表示读；`width` 表示读写的数据宽度，0 表示数据宽度为字节（8 bits），1 表示数据宽度为半字（16 bits），2 表示数据宽度为字（32 bits）。

打开 `testbench/tb_ssr_v.v`。在这个 SSRV 的 testbench 顶层文件里，会把连接 `scr1` 的 pipeline 层的指令和数据接口到 SSRV 上。下面是 SSRV 的指令和数据存储器接口连接：

```
ssrv_top u_ssr_v(
    .clk          (clk          ),
    .rst          (rst          ),
    // instruction memory interface
    .imem_req     (imem_req     ),
    .imem_addr    (imem_addr    ),
    .imem_rdata   (imem_rdata   ),
    .imem_resp    (imem_resp    ),
    .imem_err     (1'b0        ),
    // Data memory interface
    .dmem_req     (dmem_req     ),
    .dmem_cmd     (dmem_cmd     ),
    .dmem_width   (dmem_width   ),
    .dmem_addr    (dmem_addr    ),
    .dmem_wdata   (dmem_wdata   ),
    .dmem_rdata   (dmem_rdata   ),
    .dmem_resp    (dmem_resp    ),
    .dmem_err     (1'b0        )
);
```

这就是 SSRV 的 RTL 顶层的接口。它只带有指令存储器和数据存储器的接口。实际上，这个仿真包是一次超标量和乱序的实验，是通过控制缓冲器的大小和接口来决定超标量和乱序的深度。为了讲解明白，通过简单的接口来传播这一理念。当然 SSRV 可以深度耦合 scr1，在 ssrv-on-scr1/目录下，就是完全耦合 scr1 的所有资源，是一个准完整的处理器内核。

下面是连接接口：

```

`COMB begin
`ifdef USE_SSRV
    force `CORE_FIELD.i_pipe_top.pipe_rst_n = 1'b0;
    force clk = `CORE_FIELD.clk;
    force rst = ~`CORE_FIELD.core_rst_n;
    force `CORE_FIELD.imem_req = imem_req;
    force `CORE_FIELD.imem_addr = imem_addr;
    force `CORE_FIELD.dmem_req = dmem_req;
    force `CORE_FIELD.dmem_cmd = dmem_cmd ? SCR1_MEM_CMD_WR : SCR1_MEM_CMD_RD;
    force `CORE_FIELD.dmem_width = (dmem_width==2'b10) ? SCR1_MEM_WIDTH_WORD :
( (dmem_width==2'b01) ? SCR1_MEM_WIDTH_HWORD : SCR1_MEM_WIDTH_BYTE );
    force `CORE_FIELD.dmem_addr = dmem_addr;
    force `CORE_FIELD.dmem_wdata = dmem_wdata;
`else
    force clk = 1'b0;
    force rst = 1'b1;
`endif
end

```

在 rtl/define_para.v 里面有一个定义 USE_SSRV，可以用来切换内核处理工作是由 scr1 完成还是 SSRV 来完成。如果由 SSRV 来完成，pipeline 层的复位信号有效，一直处于复位状态，它的指令和数据访问功能导入到 SSRV 上；如果由 scr1 完成，那么 SSRV 处于静默状态。

```

`ifdef WIDE_INSTR_BUS
    reg `N(`XLEN) wide_addr;
    `FFx(wide_addr,0)
    if ( imem_req )
        wide_addr <= imem_addr;
    else;

    reg `N(`BUS_LEN*`XLEN) imem_rdata;
    `COMB begin:comb_imem_rdata
        integer n;
        integer t;
        imem_rdata = 0;
        for(n=0;n<`BUS_LEN*4;n=n+1) begin
            t = scr1_top_tb_ahb.i_memory_tb.memory[wide_addr+n];
            imem_rdata[`IDX(n,8)] = (t===8'hxx) ? 8'h0 : t;
        end
    end

```

```

        end
    end
`else
    wire `N(`XLEN) imem_rdata = `CORE_FIELD.imem_rdata;
`endif

```

另外，在 `scr1` 的设定中，指令存储器访问是 32 bit 位宽的，但为了提高指令处理效率，必须加大指令读取的位宽，让更多的指令能够进入处理器内。因此设定定义：

WIDE_INSTR_BUS。如果启动这个定义，那么 `imem_rdata` 可以直接从指令存储器内读取更多位宽的数据，送入 **SSRV** 处理器。如果没有启动，那么默认是 32 bit 位宽。

在 `tb_ssr.v` 后面，还有 **BENCHMARK_LOG** 这个定义。打开这个定义，那么在性能测试用例，例如 **Dhrystone** 和 **CoreMark** 测试中，对于测试中的各项指标进行统计。它包括：测试中每周期执行的指令个数、每周期执行多条指令的百分比、分支预测的正确和错误统计和数据存储器访问统计。如果不需要这些统计，可以关闭这个定义，也可以添加自己需要的定义。

紧接着注释掉的一段代码是对数据存储器访问的记录信息。**SSRV** 可以使用定义缓冲器的方式来定义超标量和乱序发射的深度，导致处理器的性能不同，但万变不离其宗，它们对于数据存储器的访问顺序是不变的。这段代码会把这些访问顺序打印在文本中，一个是记录时间信息，一个是记录访问过程信息。通过比对访问进程顺序，可以获知发生访问异常出现在哪一行中，然后查看记录时间，即可在波形上查看。如果两个记录信息合在一起，由于性能差别，时间信息必然不同，导致比对时有太多冗余干扰信息。

如果关闭了 **USE_SSRV** 定义，那么它会记录 `scr1` 的数据存储器访问顺序，它可以作为一个标准模型，来比对 **SSRV** 的数据存储器访问过程。

SSRV 运行初步

到这里，再打开 `rtl/define_para.v`，讲解其他定义。

```

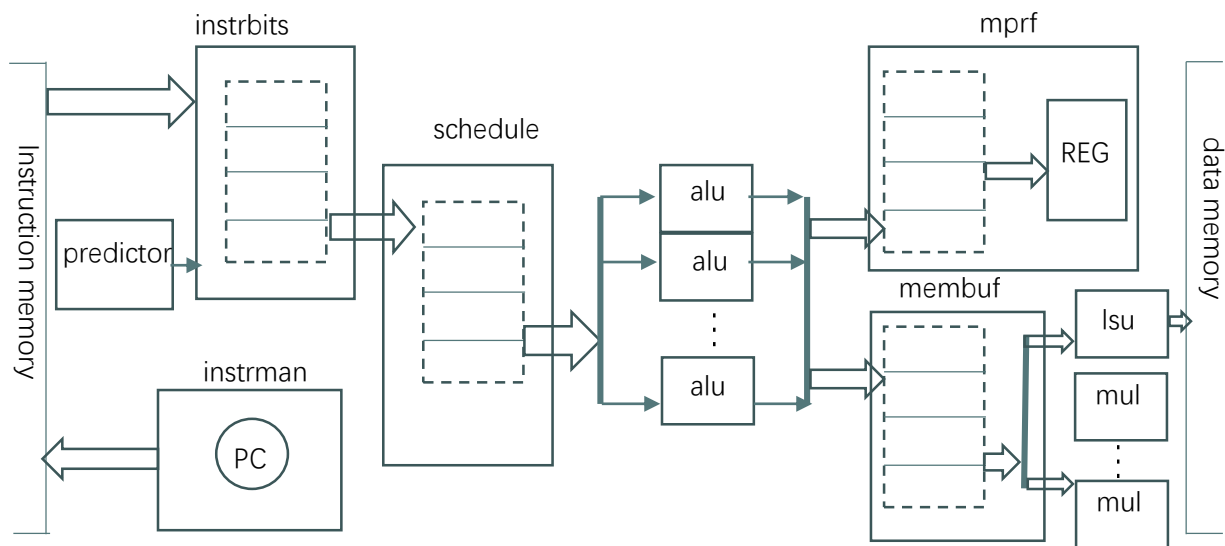
`define USE_SSRV
`define WIDE_INSTR_BUS
`define BENCHMARK_LOG
`define INSTR_MISALLIGNED

```

前面的三个定义，在前一节已经讲过。这里讲 **INSTR_MISALLIGNED** 的作用。

对于指令存储器的指令访问，scr1 是给出一个地址，然后指令存储器给出该地址对应的指令，送来的比特长度为 32 bit。现在，SSRV 做了一个大胆的扩展，可以定义为 32 bit、64bit、128 bit 等，只要是 2 的幂都可以。那么在取指令的时候，就有两种方式，一种是对齐模式，例如指令宽度为 64 bit，跳转到地址 0x204，在对齐模式下，送出地址 0x200，指令存储器送来以 0x200 开始的 64 bit，处理器需自动移除前 4 字节，才是从 0x204 开始的指令内容，因此第一次取指，实际上只取到 32 bit。当然，后续从 0x208 开始，会取到 64 bit。这种对齐模式会在首次时可能取不到足够 64 bit 的指令内容。

第二种是非对齐模式。那就是送出地址 0x204，指令存储器会以 0x204 对应的字节开始，后面跟上 8 个字节，也就是地址 0x204~0x20b 对应的字节，第二次取指会从 0x20c 开始。这种模式下，每次取指，理论上都能取到足额。因为首次取指 bit 数的差别，导致这两种模式下的性能有差别。但第一种模式，还是有它的好处，因为大多数指令存储器都是以对齐模式进行访问。使用定义 INSTR_MISALIGNED 可以来进行选择。在打开这个定义后，选择非对齐模式，同样指标下，CoreMark 分数会高出 0.2~0.3；关闭这个定义，则选择对齐模式。scr1 默认是对齐模式。



下面是定义：

```
`define FETCH_REGISTERED
```

看看系统框图，四个缓冲器是图中虚线框。最右边两个缓冲器 mprf 和 membuf 是“并联”，位于同一级。左边的 instrbits 和 schedule 所带的缓冲器是“串联”，它们中间有一个可选寄存

器来缓存一拍。如果 `FETCH_REGISTERED` 打开，那么这个寄存器存在，如果不打开，`instrbits` 的组合逻辑输出直接送入 `schedule` 缓冲器。

```
`define MULT_NUM 3
```

在这个框图中，`ALU` 模块和 `MUL` 模块的数目是可配的。`ALU` 模块的数目由 `schedule` 的缓冲器的输出而定。`MUL` 模块的数目由本定义给出。实际上乘法器的个数会影响性能，一个乘除法运算都需要多个周期完成，如果没有完成，后续指令必须等待。而这多个周期又无法消除掉，那么可以采用变通的方法，那就是多开几个乘法器，让多个乘法指令同时进行运算。虽然没法消除单个乘法器占用的周期，但多个乘法指令的消耗时间可以重叠，那么分摊在每个乘法指令上的周期消耗就减少了。

接下来是四个缓冲器的大小和接口定义。最后是更加详细琐碎的内部变量定义，并不是很多。

总结起来，`rtl/define_para.v` 的前半部分是非常影响性能的定义，可以由使用者自由选择；后半部分涉及到具体实现，修改选择来调试，也是没问题，但需要谨慎一些。

下面我们来进行初步仿真。`SSRV` 以 `scr1` 自带的 `CoreMark` 测试为例来看配置参数对性能的提升。首先，我们看最小配置。四个缓冲器的配置为：1-2-1、1-1、1-1、1-1。第一个缓冲器的配置大小必须大于输入，是因为需要根据缓冲器的剩余空间来进行加载指令操作，如果两者相等，很可能无法判断剩余空间，而无法进行加载。

以这个配置运行后，会显示如下信息：

```
---Begin testing: ../build/coremark_imc.hex
CoreMark 1.0
ticks =      594551  instructions =      223770  I/T = 0.376368
           0 --      370781 -- 0.623632
           1 --      223770 -- 0.376368
True is      47496  False is      5999  T/(T+F) is 0.887859
MEM number is      71757 --ratio: 0.120691
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 5946
Total time (secs): 0
ERROR! Must execute for at least 10 secs for a valid result!
Iterations       : 1
Compiler version : GCC8.3.0
Compiler flags   : -O2 -funroll-loops -fpeel-loops -fgcse-sm -fgcse-las
```

```

Memory location : STATIC
seedcrc         : 0xe9f5
[0]crclist      : 0xe714
[0]crcmatrix    : 0x1fd7
[0]crcstate     : 0x8e3a
[0]crcfinal     : 0xe714
Errors detected
---../build/coremark_imc.hex Test PASS

```

其中下面的信息是因为打开了 **BENCHMARK_LOG** 而出现的。它会统计在进行测试时，使用的周期数和每个周期执行指令的百分比，供仿真者参考。

```

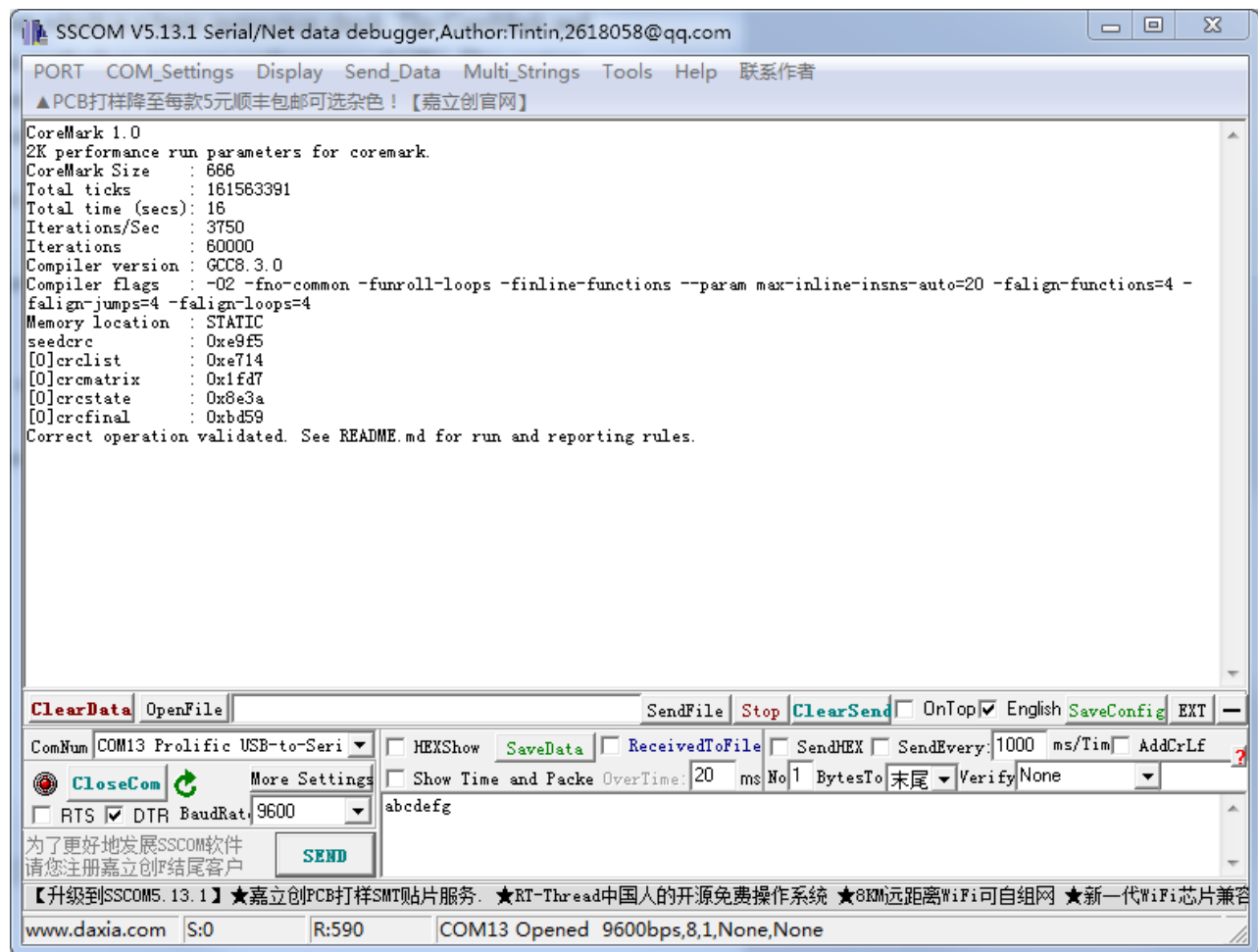
ticks =      594551  instructions =      223770  I/T = 0.376368
      0 --      370781 -- 0.623632
      1 --      223770 -- 0.376368
True is      47496  False is      5999  T/(T+F) is 0.887859
MEM number is      71757 --ratio: 0.120691

```

剩下的是进行 CoreMark 测试时打印的相关信息，其中能够反应测试成绩的是 **Total ticks: 5946**。它表示运行这段测试用去了多少个 tick。实际上，CoreMark 这种对嵌入式 CPU 的测试类似于运动员的百米赛跑测试。运动员在进行百米赛跑前，掐一次秒表，记一下出发时间；等到了终点，再掐一次秒表，记一下到达时间。两者相减，就是本次赛跑的成绩。CoreMark 测试类似，在进行测试前，程序通过 CSR 指令从记录时间的某个 CSR 寄存器上获取出发时间，等到测试完毕，再从这个 CSR 寄存器上获取到达时间，两者相差即为本次测试的时间。这里是 5946 ticks，而我们看到使用 **BENCHMARK_LOG** 统计出来的时间是：594551。差别在于 CoreMark 程序采信的时间系统是每 100 个时钟周期计数一个的 CSR 定时器，**BENCHMARK_LOG** 给出的时间是周期数。

后面还有 **Iterations :1** 的字样，表示重复了一次。在运行 CoreMark 测试时，需要运行很多次，然后使用多少多少 ticks 除去总次数来计算平均用时，最后给出 **Iterations/Sec** 的测试成绩。上面的记录中：**ERROR! Must execute for at least 10 secs for a valid result!** 应该是给出的这个成绩。只不过现在 Iteration 只有 1，程序觉得太少了，不足以给出一个信服的成绩，因此使用 **ERROR** 信息代替。

下图是一个在 FPGA 开发板中运行 60000 次的测试信息。可以看到它的 **Iterations/Sec** 是 3750，按照换算这个 CoreMark 成绩是：3.75 CoreMark/MHz。在仿真中，不可能在有限的时间内运行 60000 次，那会耗费太长的时间。因此，使用 5946 ticks 来估算最终成绩。这两者有一个倒数关系，只需对 5946 ticks 进行倒数，即可估算出它的 CoreMark/MHz。按照这个估算方法，它的 CoreMark 成绩为：1.68 CoreMark/MHz。



这个成绩当然不理想。现在我们看看现有 RISC-V 处理器的顶级成绩。按照阿里平头哥提供的玄铁 910 的成绩是：7.1 CoreMark/MHz。因此如果玄铁 910 跑这个 CoreMark 测试，它会用去 1408 ticks。对比顶级的 1408，现有的 5946 差距是很大的。

但没关系，这是没有发挥 RISC-V 的实力。现在按照下面的方式配置：

- 打开 INSTR_MISALIGNED 定义
- 打开 FETCH_REGISTERED 定义
- 定义乘除法器的个数为 3
- 定义四个缓冲器的尺寸为：4-8-4、7-3、8-2、8-3。

然后执行下面的指令进行仿真：

```
do compile.do;do sim.do;run -all;
```

最后得到仿真结果，可知它用去 1562 ticks。换算成 CoreMark/MHz 为 6.4。对比于玄铁 910，SSRV 这一款配置有它的 90% 性能。

如果试着更改其中的任意参数，那么性能也会随之变化。之所以 SSRV 成为这样灵活配置的处理器内核，是因为两点原因。一是它在产生之初就是基于一定的算法模型，这个算法模型只需要对应虚拟的 N、M 这样的大小生成。二是 Verilog 程序不比其他软件运行。按道理说，为了最高性能，所有配置参数给一个最大值即可。但这是不行的，因为 Verilog 程序最终需要综合成网表，那么面积、时序和性能需要做一个折中。这些配置就可以让开发者做一个折中选择。例如综合后发现某个缓冲器涉及的关键路径太长，能不能减少一下缓冲器的大小，牺牲一下性能，让定时时序通过。SSRV 就可以做到这一点。这就是基于算法模型进行 Verilog 程序编写的好处，它会让配置非常灵活。也正因为 RISC-V 的开放性和它的优秀特性，才让 SSRV 对于性能、时序和面积能够通过参数做出平衡选择。

上面是 CoreMark 测试，另外一个重要的测试是 Dhrystone 性能测试。只需打开：
scr1/build/test_info，把 dhrystone21_imc_max.hex 或 dhrystone21_imc_noinline.hex 前面的 # 去掉，那么 Dhrystone 测试就会加入仿真序列。

同 CoreMark 测试一样，Dhrystone 测试同样是在开始时采集一个时间，然后在结束时采集一个时间。通过两个时间差，可以计算出本次测试的成绩。和 CoreMark 相比，每次测试的内容过于简单，因此 Dhrystone 采集的时间是以周期数基础的，更加复杂的 CoreMark 测试以周期数除以 100 为基础的。也正因为简单，Dhrystone 在循环上 500 次后，软件可以直接给出性能结果，而 CoreMark 只是一种估算。同样因为简单，在编译时 noinline 的选项对于性能差别太大，因此会分成两种 Dhrystone 测试，一种是没有 noinline 选项的，这里以 max 结尾为代表；另外一种是有 noinline 选项的，以 noinline 结尾为标志。

在采用上面 6.4 CoreMark/MHz 的配置进行两项 Dhrystone 测试，得到两个迥异的成绩：8540 和 4923。除上 1757 换算成通常的性能指标是：4.9 DMIPS/MHz 和 2.8 DMIPS/MHz。下面是 scr1 的对应成绩：

SCR1 overview cont

Performance*, per MHz	DMIPS	-O2	1.28
		-best**	1.89
	Coremark	-best**	2.95

* Dhrystone 2.1, Coremark 1.0, GCC 7.1 BM from TCM

** -O3 -funroll-loops -fpeel-loops -fgcse-sm -fgcse-las -fllto

Synthesis data:

Minimal RV32EC config: 11 kGates

Default RV32IMC config: 32 kGates

Range 10..40+ kGates

Dhrystone 的实测成绩是软件计算直接给出的，CoreMark 是通过 ticks 数换算得来的。通过 Dhrystone 的成绩对比，可知对于 CoreMark 的估算是靠谱的。

SSRV 整体浏览

通过上面的性能测试，可以看到 SSRV 把高性能嵌入式 CPU 的性能进行了量化，它把这些性能和缓冲器的大小进行挂钩。正是因为我们给出了多个元件，让缓冲器变大，使得多个指令可以同时执行，最终性能测试的分数也就变得很高了。

正是因为这两项性能测试不涉及到中断和异常，导致在呈现给诸位的仿真中，并没有涉及到这些的代码。并不是因为这部分并不重要，而是因为 SSRV 专注于呈现给诸位可以重复使用的高性能处理器框架。在以 RV32IMC 为底层指令集的处理器内核中，关于中断和异常方面都是各不相同的。在 RISC-V 官方给出的指导中，并没有强行以某种方式，开发者完全可以自定义自己的中断处理方式，可简可繁，完全可以按照自己的方式来开发处理器。因此，建议诸位以 SSRV 为底层代码，加上中断和异常，就可成为一个完整的处理器内核。在 `ssrv-on-scr1/` 目录下，有一个继承 `scr1` 的中断和异常的参考代码，读者可以参考这一设计。

因此，SSRV 现在的接口非常简单：

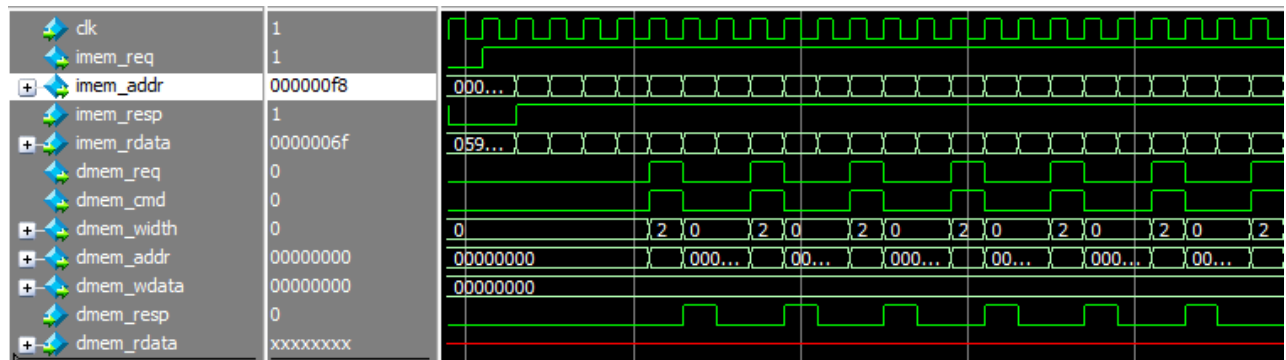
```
ssrv_top u_ssrp(  
    .clk          (clk          ),  
    .rst          (rst          ),  
    // instruction memory interface  
    .imem_req     (imem_req     ),
```

```

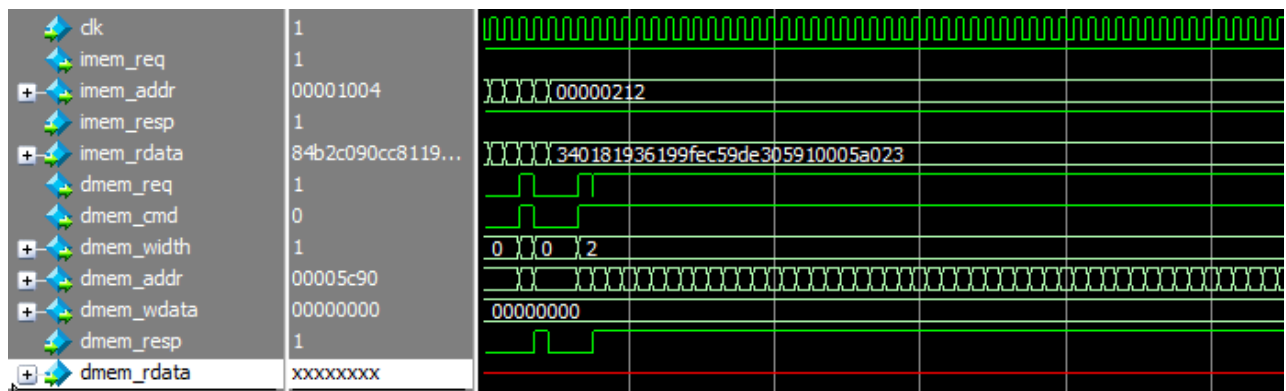
        .imem_addr      (imem_addr      ),
        .imem_rdata     (imem_rdata     ),
        .imem_resp      (imem_resp      ),
        .imem_err        (1'b0          ),
        // Data memory interface
        .dmem_req        (dmem_req       ),
        .dmem_cmd         (dmem_cmd       ),
        .dmem_width      (dmem_width     ),
        .dmem_addr       (dmem_addr      ),
        .dmem_wdata       (dmem_wdata    ),
        .dmem_rdata       (dmem_rdata    ),
        .dmem_resp        (dmem_resp     ),
        .dmem_err         (1'b0          )
    );

```

从整体上看，SSRV 作为主动方，它通过指令存储器接口发起指令读取操作，读取指令后，然后进行译码执行。其中有些指令是对数据存储器访问的，那么相应的，指令存储器接口上会出现 SSRV 对数据存储器进行访问的波形。这些波形有时读数据存储器，有时写数据存储器。因此，我们可以得到下面的波形，这是 SSRV 最低配，也就是缓冲器全配为 1 时的仿真波形：



下图是 SSRV 为 6.4CoreMark/MHz 的配置时的同样时刻的波形：



首先看指令存储器访问信号，两者的 `imem_req` 信号一直居高，表明两者都一直在加载指令。因为 `imem_resp` 信号一直保持为高，表明指令存储器对于访问请求给予了立即应答，因此它需要的指令数据 `imem_rdata` 也是一直有效。差别在于 `imem_rdata` 的位宽，前者是 32 bit，后者是前者的四倍，为 128 bit 位宽。因为后者能够一次取 32 bit 位宽的 4 条指令，因此 `imem_addr` 为 0x212 的地址的指令会一并处理：

212:	0005a023	sw	zero,0(a1)
216:	0591	addi	a1,a1,4
218:	fec59de3	bne	a1,a2,212 <_start+0x12>

0x212 地址对应的是 3 条指令，最后一条指令为条件跳转指令。因为后者配置能够高效处理指令，这三条指令可以并行处理。因此后者会一直读取 0x212 地址，那么 0x212、0x216 和 0x218 这三条指令会在同一个周期处理完毕，导致后者会一直读取 0x212 地址。

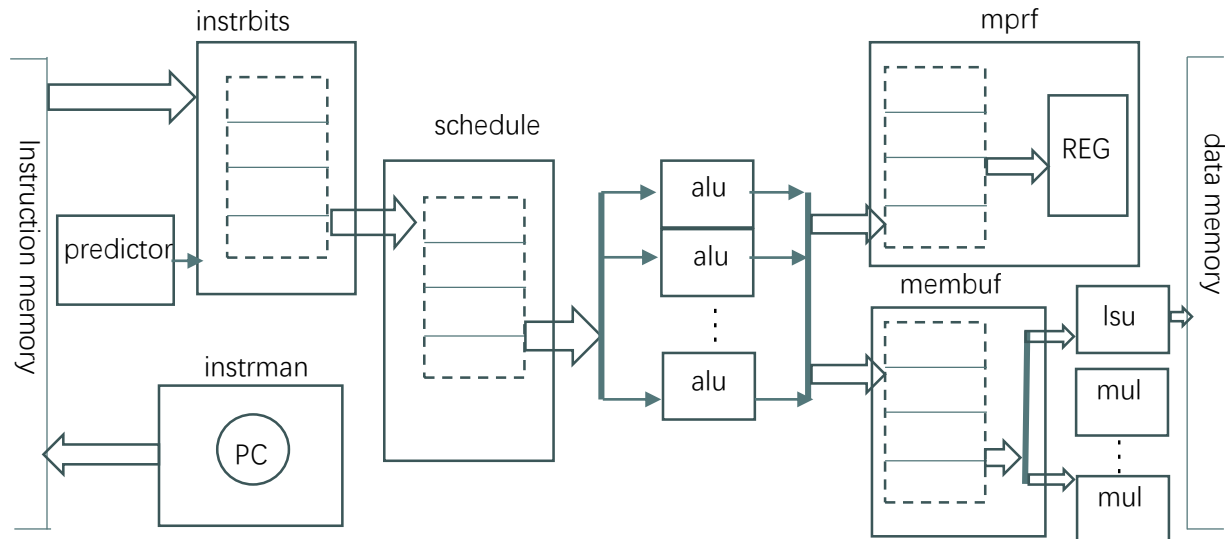
再看数据存储器访问信号。数据存储器访问是受指令驱动的，如前面所说的 0x212 指令，他就是一条写存储器指令，后续是对地址增加，然后返回继续写存储器。在低性能配置中，由于全体参数都选 1，因此相当于逐条指令依次处理。这就是传统的 N 级流水线处理方式，一次一条指令是焦点，占据了所有资源。因此我们看低性能配置波形中的 `dmem_req`，它是每三个周期，发起一次写数据存储器的操作。那么这空余的两个周期是留给 0x216 和 0x218 工作的周期。

高性能配置的波形就不一样了。我们再看后者的波形图，它的 `dmem_req` 是一条直线，表明它一直进行写入存储器操作，好像根本不存在 0x216 和 0x218 这两条指令。它们还是存在的，只是它们的执行是和写存储器操作重叠在一起。

用枪来打个比方。传统的五级流水线，每一条指令挨次执行，那么每一条指令都会一个周期归它所有。那么这样的五级流水线处理器类似于以前的老式步枪，发射完后，装弹、上膛都是要占据时间的。因此，老式步枪是打一枪停顿一会，装弹然后上膛；对于传统流水线处理器，也会是写一条存储器，然后换地址、回跳，一直循环往复。

高性能处理器则类似于连发机枪。你根本看不到装弹和上膛的操作，只看到枪口连绵不绝的发射子弹。同样，高性能并行处理器，也是只看见它不停顿的对存储器操作。不是说没有换地址和回跳的操作，而是这两个操作自动在对存储器操作中伴生完成了，仿佛没有发生一样。

这就引发了 SSRV 的一个缓冲器的重要功能：弹夹。如下面的框图，**membuf** 模块内有一个缓冲器，它存放着待执行的 LSU(load/save Unit)指令。它会一条条的按顺序执行送来的 LSU 指令。最底部的 LSU 指令会发起对数据存储器操作，直到数据存储器给出成功回答。在收到成功应答后，这条 LSU 指令退出，下一条新的最底部 LSU 指令接续它的位置，也就发起对数据存储器操作。



在 **membuf** 内不仅存放着 LSU 指令，还夹杂着乘除法 MUL 指令。乘除法指令需要用到专门的乘除法器，为了加速乘除法运算处理，可以放置多个乘除法器并行执行。然后，每个周期，可以退出一个或多条 LSU/MUL 指令。

还是看下面这三条指令的执行：

212:	0005a023	sw	zero,0(a1)
216:	0591	addi	a1,a1,4
218:	fec59de3	bne	a1,a2,212 <_start+0x12>

其中 0x216 地址的指令属于 OP 指令。它会送入另外一个缓冲器，也就是 **mprf** 模块内的缓冲器。一旦 0x212 指令在 **membuf** 模块内获得成功退出，这也会通知 **mprf** 模块的缓冲器，让 0x216 地址的指令同步退出——也就是寄存器 **a1** 获得更新。

membuf 的缓冲器接纳对于数据存储器操作的 LSU 指令以及乘除法操作的 MUL 指令。这两类指令在上一级缓冲器经过 ALU 模块后只是得到了操作必要的数，而非最终结果。例如 LSU 指令在从寄存器组提取 Rs0 和 Rs1 后，那么经过 ALU 模块计算，会得到对于数据存储器操作的地址和写数据；MUL 指令获取 Rs0 和 Rs1 后，获得两个乘数。它们的这两个必要

参数，会按顺序送入 membuf 模块。membuf 模块则按照它们进入的顺序，如果是 LSU 指令，则送入 LSU 模块；如果是 MUL 指令，则送入某个空闲的 MUL 模块。

membuf 缓冲器的最底部的那条指令，会按照类型检查对应的 LSU 或 MUL 模块，是否完成操作。如果完成，那么退休当前最底部的指令，membuf 缓冲器也就清空了一个位置。

mprf 缓冲器接纳从 ALU 模块出来的 OP 指令。OP 指令在获得必要的 R0 和 R1 后，可以直接给出 Rd 的写入数据。因此，OP 指令送给 mprf 缓冲器的数据形式包含两部分：一是 Rd 是哪个；二是 Rd 需要写入什么。mprf 缓冲器按照顺序保留这两部分内容。

一旦 membuf 缓冲器退休的指令成功完成，那么这条信息送达 mprf 缓冲器后，mprf 缓冲器也就将在这条退休的 LSU/MUL 指令之后的 OP 指令进行同步退休。OP 指令退休的意思是它指定的 Rd 会写入它指定的数据。

总结一下，mprf 和 membuf 缓冲器存放着待退休的两类指令，一等 LSU 指令在获取正确应答后，一条 LSU 指令加多条 OP 指令同步退休，然后后面的 LSU 指令无缝接替。在外面看来，仿佛多条 LSU 指令是自动无缝连缀执行一样。

架构篇

RISC 指令处理

RISC-V 属于 RISC 类型的指令。下面以简化的几条 RISC 指令为例讲解四个缓冲器是如何配合工作的。我们认为下面几类指令是一般的 RISC 类指令都共有的：

1. 数据存储器加载指令 `ld` 和存储指令 `sd`。统称为 **LSU** 指令。
2. 算术或者逻辑指令：`add`、`sub`、`and`、`or`。统称为 **OP** 指令。
3. 直接跳转指令：`jal`——跳转目标地址由 **PC** 与指令自带的操作数构成，可能带有写 **PC** 进入寄存器组的某寄存器的功能。
4. 非直接跳转指令：`jalr`——跳转目标地址由寄存器组的某个寄存器提供，可能带有写 **PC** 进入寄存器组的某寄存器的功能。
5. 乘除法指令：`mul`、`div`。统称为 **MUL** 指令。

这是一个非常简单的设定，大多数 RISC 指令集都包含着这几类指令。我们实际上看到的指令是下面这样的形式：

```
addr0: add0
      ld0
      and0
      sd0
      jal addr1

addr1: add1
      ld1
      and1
      sd1
      mul0
      jalr Rs0
```

算术和逻辑指令 **OP**、存储器操作指令 **LSU**、乘除法指令 **MUL** 是三种常用数据处理指令，其他两类跳转指令 `jal` 和 `jalr` 指令是连接指令，把三种数据处理指令 **OP**、**LSU**、**MUL** 连接在一起。`jal` 和 `jalr` 指令的跳转其实是把两个部分的实体指令合二为一了。那么上面的两段指令就会变成下面的顺序。

```
add0  
ld0  
and0  
sd0  
add1  
ld1  
and1  
sd1  
mul0
```

在执行完 `add0` 及后面的 `sd0` 后，跟上跳转目的地 `add1` 上的指令：`add1` 直到 `mul0`。

有了这样的设定，我们来看看 **SSRV** 是如何来运用四个缓冲器来完成上述指令的处理。

● 第一级缓冲器：instrbits

这一级缓冲器的输入是读取指令存储器的指令数据，输出是多个为下一级准备的 **OP**、**LSU** 或 **MUL** 指令。按照指令处理需要的周期数，**SSRV** 把这三类指令分为两块来处理，一块是单周期执行的 **OP** 指令，也称为 **ALU** 指令；一块是多周期或不定周期执行的 **LSU/MUL** 指令，有时统称为 **MEM** 指令。

缓冲器的输入为指令存储器的读数据。读数据的位宽即为缓冲器的输入大小，可以配置为 32bit、64bit、128bit 等等 2 的幂的宽度。缓冲器的容量是 32 bit 的倍数。这个容量必须大于读数据的位宽，因为在缓冲器的剩余空间大于读数据的位宽时，会自动发起读指令的操作，以求填满缓冲器。如果容量小于读数据的位宽，不可能达到自动续读指令的条件。

缓冲器的输出为已成型的指令。指令长度可能是 16 bit，也可能是 32 bit；指令数目也是不固定的。当指令可以输出到下一级或得到处理后，那么它对应的字节会从缓冲器中移除。因此输出的比特数是 16 的倍数。

由于指令的输入和输出都是未知，那么保存在缓冲器内的比特个数也是不固定的。每次生成指令时，都是聚齐已保存在缓冲器的比特和刚从指令存储器进入的比特。这两者按照前后顺序连接在一起，作为译码的对象。

每次译码多少条指令由下一级的需求决定。下一级可以接受 **N** 条指令，那么它就从缓冲器内的对象比特中尽量生成 **N** 条指令。缓冲器内可供译码的指令比特可能无法支撑译码这么多条指令，那也没办法，能生成几条就是几条。如果只留存有 0 或 16 bit，可能一条指令也无法提供。这里可以说可以生成 **M** 条指令，**M** 小于等于 **N**。

如果这 M 条指令全部是 OP、LSU、MUL 指令，那这最好处理了，直接输出到下一级。下一级的任务就是按照顺序接收这三类指令。现在这 M 条指令直接通过并行连接线，送到下一级去，于是从缓冲器内移除这 M 条指令所对应的比特。这就是缓冲器的最简程序，从它的比特仓库内取出比特，打包成指令的格式送到下一级去。

但是指令类型没有那么简单，并非只有这三类指令，那么这一级缓冲器的战略任务就是处理其他所有非这三类的其他指令。如果是 JAL 指令，处理方式如下。

假定 JAL 指令位于这 M 条指令的中间，它前面有指令，后面也有指令。

1. 前面的指令属于三类指令，这些前面的指令以及 JAL 指令本身，送入下一级。
2. 通过 JAL 指令获取跳转地址，打断取指操作，使用跳转地址作为新取指地址。
3. JAL 指令后续的指令不予处理。
4. 移除 JAL 指令和它前面的指令所对应的比特，同时对缓冲器进行清空。

因为缓冲器的清空，待到新地址的指令比特到来，进行译码后，可以接续 JAL 指令之后，送入下一级。于是下一级就“没感觉”中间发生的跳转，可能只是新地址的指令的 PC 发生了变化。

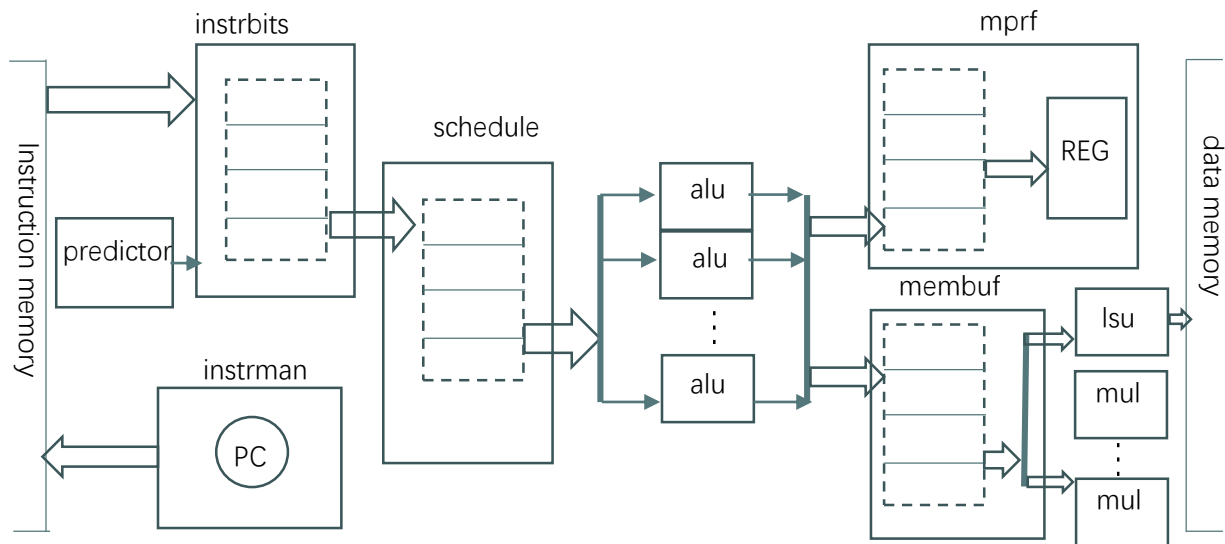
这就是第一级缓冲器清除非三类指令的第一招：现场立即处理，不影响三类指令输送到下一级。但 JALR 指令就和 JAL 指令不同，因为它的跳转地址并非由指令本身“携带”，而是存放于寄存器组的某寄存器。那么 JALR 指令必须从寄存器组去取这个数据，才能实施跳转。它去取数据必须保证能取到正确数据，不能取到的原因是流水线上排满了各种未处理的指令，说不定某条指令的 Rd 正是 JALR 指令需要的那个寄存器。那么在这条指令没有向寄存器组写这个 Rd 时，JALR 指令是取不到正确数据的。

幸好后级流水线会给本级缓冲器一个寄存器列表，它有 32 bit，每一 bit 对应一个寄存器，它等于 1 表示这个寄存器不能读取，因为流水线上至少一条指令的 Rd 等于这个寄存器；它等于 0 表示这个寄存器可以正确读取，流水线上没有一条指令的 Rd 等于这个寄存器。这个寄存器列表可以称为寄存器组的黑名单。于是需要 JALR 指令去查询这个黑名单，如果发现 JALR 指令的 R0 处于黑名单上，那它得等。下一个周期继续查，直到黑名单上没有 R0 了，也就表示流水线上以 R0 为 Rd 的指令都处理完毕后，JALR 指令才能执行。于是，缓冲器清除非三类指令的第二招：等待后处理。

JALR 指令要等的第一条是等到出现在缓冲器的首条指令位置。因为缓冲器在同时译码 M 条指令时，可能是前面一些指令再接上 JALR 指令。那么这时需要做的是，对 JALR 指令的前面指令输送到下一级，再在缓冲器中移除这些前面指令对应的比特。那么在下一个周期，译码时发现 JALR 指令“恰好”出现在首要位置。当然 JALR 指令正好出现在首要位置，那么这个过程也就可以忽略了。

之所以有这个过程是缓冲器只想对首要位置指令的 R0 进行查询，所以首先是让这些指令“移动”到这个首要位置。于是查询得知 R0 处于黑名单当中，那么缓冲器不做任何移除操作，也就没有任何指令输出到下一级。下一个周期，如果 R0 还在黑名单，还是这么操作。一直到 R0 不在黑名单了，那么取出 R0，计算出新的跳转地址，按照 JAL 指令的 4 个步骤进行处理。这个时候，只有 JALR 指令会输出到下一级，因为 JALR 指令还会有对寄存器组的某个寄存器写 PC 的操作，这个操作使得 JALR 指令等同于一 OP 指令，因此它会进入后面的流水线。

总结起来，这一级缓冲器类似于清道夫的工作。它滤除所有非三类指令的其他指令，容易处理则现场处理，不容易处理则让该指令排在首要位置，等待有利时机再进行处理。当这条指令处于首要位置时，它会阻止任何新的三类指令的生成，后面的流水线会感到输送指令的“断流”。这是没办法的事，因为它得按照规程处理异常指令。



● 第二级缓冲器 schedule

本级缓冲器承接上一级缓冲器。它对上一级缓冲器给出它的剩余空间，提示发送给它的指令条数不能超过这个范围。本级缓冲器的输入是多条指令，每条指令是 32 bit 的指令，对于

只有 16 bit 长的指令，只占用低半部分，高半部分无意义，一般填充的是下一指令的低 16 bit。除了指令本身，还包括它的 PC 地址。

本级缓冲器存储的是这样的指令，输出的也是这样的指令。指令按照计算需要的周期数，分为单周期指令，只包含 OP 指令；另外是多周期指令，包括 LSU 和 MUL 两种。缓冲器存储的是这两类指令夹杂在一起，出现什么类型完全是随机的。

本级缓冲器每个周期的主题是：从存储的指令和进入的指令连接在一起的集合，选择最多 ALU 模块数目的指令进入下一级，剩余的指令仍然驻留在缓冲器内。在下一个周期时，进入下一级的选中序列进入 ALU 模块，驻留在缓冲器内的驻留序列则成为存储指令参与下一次的 schedule 模块组织的调度。

不管是单周期指令还是多周期指令，它们都会读取最多两个寄存器 Rs0 和 Rs1，在这两个寄存器的基础上，单周期指令会通过 ALU 模块计算出它的最终结果，也就是 Rd 的写入数据；而多周期指令则会通过 ALU 模块得到两个操作数，这两个操作数在送入计算的实体模块后才能得到最终结果。于是，经过 ALU 模块的单周期指令会进入下一级的 mprf 缓冲器，mprf 缓冲器存储的格式是 Rd 和它的写入数据；经过 ALU 模块的多周期指令会进入下一级的 membuf 缓冲器，membuf 缓冲器存储的格式是两个操作数，会有 Rd 但没有写入数据，写入数据需要在 membuf 缓冲器内依次送入实体计算模块时才能得出。

本级缓冲器的难点在于如何生成选中序列和驻留序列。实际上，这是对缓冲器内的每一条候选指令进行考察，如果达到某条标准，则这条指令进入选中序列，达不到这条标准的则进入驻留序列，一直到选中序列满员为止。

本级缓冲器的候选指令按照顺序排开，每条指令只占用两三个寄存器 Rs0、Rs1 和 Rd。没有可能首条指令使用寄存器 R0、R1 和 R2，就不允许后面的指令使用 R3、R4 和 R5 的和前一条并行执行。这就是这个标准的理论基础，寄存器组有 32 个寄存器，每一条指令只占用两三个，理论上可以选择多条互不干涉的指令并行使用这 32 个寄存器。

指令和指令之间是有依赖和制约关系，前一条指令可能导致后一条指令不能执行，当绝不能让这种制约关系滥用，一定要采用一种科学的方法的来定量。比如某条指令使用 R0 和 R1 作为 Rs，R2 作为 Rd，如果笼统的为后面指令规定 R0、R1 和 R2 不能再使用，理论上是可以作为一个标准的，当这个标准显然滥用了。只要沾边的就禁止后边的使用，是有点宁可错杀，不可放过的意味了。

科学的方法是采用两个黑名单，一个是 **Rs** 黑名单和 **Rd** 黑名单，让对每一条候选指令对照这两个黑名单，一旦发现它的 **Rs** 和 **Rd** 处于黑名单上，那么这条指令就因为黑名单不能进入选中序列，也就代表着它在本周期不适宜执行。不管这条黑名单进没有进入选中序列，它需要对 **Rs** 黑名单和 **Rd** 黑名单进行更新，也就是考察这条指令当前的状态下，它会影响哪一个寄存器不能作为 **Rs**，那么这个寄存器加入 **Rs** 黑名单，哪一个寄存器不能作为 **Rd**，那么这个寄存器加入 **Rd** 黑名单。于是这个更新的黑名单传递到后排紧随的指令上去，跟随指令重复上述过程，这条指令也就可以判断进入哪一个序列，也就会继续更新两个黑名单。

这种更新黑名单的方式是非常民主的方式。也就是每一条指令可以对比前排指令传递的黑名单——这个黑名单是前排指令“意愿”的累计表达，只要这条指令通过了黑名单的考察，那么它就可以宣布它可以越过前排指令来执行，因为它涉及的寄存器和前排的所有指令无关。通过了黑名单的考察，就类似于前一条指令用 **R0**、**R1** 和 **R2**，后一条指令用 **R3**、**R4** 和 **R5** 的关系，这是一目了然的，现在黑名单存放的是 **R0**、**R1** 和 **R2**，后续指令只要除了黑名单以外，使用了其他寄存器，它也就可以越级先执行。

下面首先评估位于缓冲器第一位置的指令，这条指令可能是上一次评估进入驻留序列的第一条，也可能是上一次没有驻留序列，是来自于上一级 **instrbits** 缓冲器新进入的指令。如果是前一种情况，那么在 **mprf** 缓冲器和 **membuf** 缓冲器内存放的是一部分位于它前列的指令和一部分它允许提前超过它执行的指令。现在是想从 **mprf** 缓冲器和 **membuf** 缓冲器得到 **Rs** 和 **Rd** 的最初黑名单，这个黑名单包含了这两部分指令的构成。如果是真正位于它前列的指令，这个黑名单是有效力的；如果是位于第一条指令的后面，越级提前执行的，那么它加入黑名单是对这条指令不会造成任何影响的，原因是它在上一次评估中，越级的指令通过了黑名单的考察。

如果是后一种情况，那么从 **mprf** 缓冲器和 **membuf** 缓冲器内得到的 **Rs** 和 **Rd** 初始黑名单，则是位于它前列的指令的真实“意愿”。那么这第一条指令从这两个初始黑名单进行检查，检查结果可能是通过进入选中序列，或者失败留在驻留序列。如果通过进入选中序列，那么代表该条指令的 **Rs0** 和 **Rs1** 在 **ALU** 模块中会“兑现”，那么这条指令的 **Rs0** 和 **Rs1** 不再有加入黑名单的必要了，此时只有它的 **Rd** 会更新这两个黑名单。如果失败留在驻留序列，作为一条未执行的指令，那么它的 **Rs0**、**Rs1** 和 **Rd** 都需要保护起来，这三者都要加入 **Rd** 黑名单——也就是这三者不允许后面越级的指令去破坏，影响它的执行。同时，它作为待执行指令，后面的 **Rs** 如果等于它的 **Rd**，它并没有执行，那么后面指令的 **Rs** 不可能取得它的 **Rd** 的最新值，因此它的 **Rd** 会加入 **Rs** 黑名单。

第一条指令会添加它的更新到这两个黑名单上，然后传递给第二条指令。第二条指令会重复评估自身和添加黑名单的过程。如果缓冲器内的每一条指令都重复完毕，那么就能到一个选中序列和驻留序列。选中序列会进入寄存器，下一个周期在 ALU 模块取出 Rs0 和 Rs1，这两个寄存器不再是“期货”了。驻留序列会进入 schedule 缓冲器内排在前列。等到下一个周期，后加入缓冲器的指令排在驻留序列的后面，形成一系列新的候选指令。

驻留序列在第二个周期并不会一成不变的再次成为新的驻留序列，原因在于 mprf 和 membuf 缓冲器在发生变化，那么这个初始的 Rs 和 Rd 的黑名单也就发生了变化，于是原有的驻留序列的指令会因为黑名单的不同而有了新的机会。

不管驻留序列变没变化，只要本级缓冲器还有剩余空间，那么上一级缓冲器会不断补充新成员。如此，周而复始，本级缓冲器既能容纳多条指令逗留，又能同时发射多条同时指令。它会完全尊重每一条指令的意愿，民主的选择适宜的指令进入下一级。

● 第三级缓冲器 mprf 和 membuf

本级缓冲器承接上一级缓冲器，接收上一级缓冲器的输出。上一级缓冲器会输出 ALU 模块数目的指令。每一条指令对应一个 ALU 模块。这个 ALU 模块会读出这条指令的 Rs，然后根据指令类型，得到该指令的输出。如果是单周期指令，也就是 OP 指令，会输出 Rd 和它的对应写入数据；如果是多周期指令，也就是 LSU 或 MUL 指令，会得到它的两个操作数，对于 LSU 指令，这两个操作数是地址和写数据；对于 MUL 指令，这两个操作数是两个乘除数。多个 ALU 模块的输出是第三级这两个缓冲器的输入。

缓冲器 mprf 只接收这其中的单周期指令的输出结果。也就是 mprf 的输入分为两个域，一个域是 Rd，5 bit；另外一个域是 Rd 对应的写入数据，32 bit。由于 ALU 模块既可以是单周期指令，也可以多周期指令，因此 mprf 缓冲器的输入并不是连续的形式，而是中间会有空位，这个空位是因为这个位置是多周期指令占用的。因此，在接收输入时，需要排除这些空位，让这两个输入域紧密的连接一起。

之所以定义 mprf 缓冲器，是因为 OP 指令并不能随时写入寄存器组，也就是 Rd 和它的数据域获得写入权限。因为 LSU 指令或者 MUL 指令会存在出现异常的情况，在出现异常时，寄存器组不能让该条异常指令之后的指令写入。举例如下：

```
op0
lsu0
op1
op2
```



```
mul0  
op3  
lsu1  
op4
```

以上是顺序执行的某段代码。其中 op0、op1、op2、op3、op4 送入 mprf 缓冲器，lsu0、mul0 和 lsu1 送入 membuf 缓冲器。op0 当然有权限写入寄存器组，但 op1 和 op2 却不能写入寄存器组，原因在于 lsu0 指令并没有成功执行，一旦 lsu0 指令汇报异常，那么寄存器组必须呈现 lsu0 指令执行之前的状态。一旦 op1 和 op2 指令写入了寄存器组，那么寄存器组无法恢复到 lsu0 指令执行之前的状态。

因此，mprf 缓冲器是逐步放开它内部的 OP 指令的写入权限的。一旦 lsu0 指令成功执行，它会放开 op1 和 op2 指令的写入权限，那么它们就和 op0 指令一样可以有权写入寄存器组，离开缓冲器。以此类推，mul0 指令的成功执行会放开 op3 指令的写入权限，lsu1 会放开 op4 指令的权限。因此，mprf 缓冲器会配合 membuf 缓冲器的执行。mprf 缓冲器会等到 membuf 缓冲器执行一条指令然后放行一批，至于放行多少，由指令结构决定，可能是 0，可能是很多条。

mprf 缓冲器的输出是可配置的，也就是同一周期，可以有自定义的多条 OP 指令离开缓冲器，进入寄存器组。每次 membuf 缓冲器释放一条指令，会导致多条 OP 指令有权限写入，但每周期只有固定的写寄存器组名额。如果来不及写完，这些 OP 指令会在缓冲器内排队等候；当然如果数目太少，那么很快写完，也会等待其他指令解禁获取写寄存器组权限。

membuf 缓冲器接收 LSU/MUL 这样的多周期执行指令。因为这些需要多周期执行，因此通过 ALU 模块时，只是获取了两个重要的操作数，然后进入 membuf 缓冲器进行排队。执行 LSU 或 MUL 指令的实体模块是有限的，只有一个 LSU 模块，可以用来访问数据存储器，执行写入或读取操作；可以有多个乘除法模块。

membuf 缓冲器的成员一定是按照顺序进行排列。对于 scr1 这样的嵌入式 CPU 来说，SSRV 并不想过度复杂化，不愿意涉及到访存乱序带来的问题。membuf 缓冲器会按照先后顺序，分配一个 LSU 模块和多个 MUL 模块对接先来的指令。某指令获取访问这些实体操作模块后，它会提交它的两个操作数和相关操作参数，然后等待操作结果。

membuf 缓冲器总是会关注最底部的那一条 LSU/MUL 指令，它会观察它是否分配到某个实体操作模块；如果分配了，然后查看这一实体操作模块的返回结果。如果是正确应答，那么表示该条指令可以安然退休，离开 membuf 缓冲器。

理论上，**membuf** 缓冲器也可以同时退休多条指令。也就是一个 **LSU** 模块和多个 **MUL** 模块同时汇报最底部的 **N** 条指令已经成功操作，那么 **membuf** 缓冲器可以同时移除多条 **LSU/MUL** 指令。移除的 **LSU/MUL** 指令包含一个写入寄存器组的操作，它会把 **LSU** 指令也就是读操作的数据或者乘除法的结果直接写入寄存器组。这就类似于 **mprf** 缓冲器的退休，以成功写入寄存器组作为退休最后一件事。

至此，在了解了 **mprf** 缓冲器和 **membuf** 缓冲器的运作原理后，可以解答上面 **schedule** 缓冲器如何用这两个缓冲器生成初始的 **Rs** 黑名单和 **Rd** 黑名单。

首先，**mprf** 缓冲器是和寄存器组紧密联系在一起的。在 **schedule** 缓冲器内的指令看来，进入了 **mprf** 缓冲器内的 **OP** 指令，和已经写入没有什么区别。因为在 **schedule** 缓冲器内的指令，如果它们有幸进入了 **ALU** 模块，当他们读 **Rs** 来进行计算时，这个过程分为两部分。一是读真正的寄存器组，这好比查字典；二是从 **mprf** 缓冲器内找到对于寄存器 **Rs** 的补充，这好比查勘误。如果勘误内没有对寄存器 **Rs** 的补充，那么以第一条，也就是字典的数据为准。从获取 **Rs** 的过程来看，**mprf** 缓冲器内存放的指令实际上可以看作已经执行。那么，**mprf** 缓冲器内不会产生任何的 **Rs** 和 **Rd** 的禁忌，因为它们视作执行完毕。

membuf 缓冲器内的指令则不同。这里面的指令都需要多个周期执行。这里面每一条指令都需要把它的两个操作数提交给实体模块，才能按顺序获取执行结果。因此，这里面的每一条指令的 **Rd** 都是未知的。比较特殊的是，待提交结果，也就是准备退休的指令，因为它们的计算结果已经获知。那么可以说，**membuf** 缓冲器内所有未退休指令的 **Rd** 都会造成 **Rs** 的黑名单和 **Rd** 的黑名单。造成 **Rs** 的黑名单是因为这些指令都是将排队获取结果的指令，任何想以这些指令的 **Rd** 作为 **Rs** 的行为，都不可能获取正确的数据，原因在于还没有产生，还是期货。造成 **Rd** 的黑名单同样因为，这些指令还没有完成写入，是不能允许后续指令的 **Rd** 来越过它们。

下面以拟人的方式来谈谈这三级缓冲器的运行原理。我们可以把整个系统看成一个机场。**OP** 指令是国内旅客，**LSU/MUL** 指令是国际旅客。第一级缓冲器相当于检票处。旅客是从指令存储器内到达检票处。检票员根据第二级缓冲器——相当于机场大厅的剩余空间放行旅客。如果是国内或是国际旅客，那么直接放行；如果发现一名内部员工，自称现在从某某序号接纳旅客，后面的旅客不予接待；那么检票员只得对现场清空，等待某某序号来的旅客。如果这名内部员工说这个序号还不知道，但在某某寄存器上，那么检票员就查询这个寄存器是否可用，如果发现某位旅客正要修正该寄存器但没有完成，那么他就让这名内

部员工排在检票处首位候着。一直到他查询到可用，那么他就启动清空检票处，并准备开始接待新序号的旅客。

国内和国际的旅客间杂着进入第二级缓冲器，可以把这一级缓冲器比作机场大厅。机场大厅的工作人员也会尽可能把旅客送入各自对应的候机厅。他也要看他们的候机厅是否有对应空间，还有就是它们的 $Rs0$ 、 $Rs1$ 和 Rd 是否合法。如果合法也有位置，那么国内旅客就可以进入 ALU 模块这一值机口。但是国际旅客必须严格按照次序，也就是如果前面有国际旅客不能值机，还得呆在机场大厅，那么他后面的国际旅客也不能值机。

旅客经过值机口后，发现两个候机厅，他们根据机票进入了对应的候机厅。进入国际旅客候机厅的可以排队登机。但是国际旅客必须做一个护照检查，如果是目的地是国外，那么护照检查只有一个窗口，也就是 LSU 窗口；如果目的地是港澳台，那么因为是内部问题，可以排多个窗口，也就是 MUL 窗口。每一个旅客提交护照后，会排在登机口。管理登机的工作人员，会查看排队的旅客，看他的目的地以及对应窗口返回的信息，如果成功，让这名旅客登机。

进入国内候机厅的旅客发现机场有一个特殊规定，那么就是必须等他前面的国际旅客登机后他们才能登机。但是国内旅客发现他们进来的顺序打乱了，他们根本不知道前面是否有国际旅客或者几个。幸好机场发给了他们一个序号，他们根据机场给出的国际旅客登机的信号，对这个序号减一。一旦这个序号减到 0，那么他们就可以进入登机口排队登机。所以每次国际旅客走了一个后，也有几个国内旅客发现它们的序号为 0，也排队进入登机口，和国际旅客一起登机。

希望 $SSRV$ 能够带来流水线的升级版。我们常知道的都是以单个指令为基础的流水线，一条指令跟随一条指令在流水线上运作。一旦某一条指令卡壳，于是整条流水线停顿。而 $SSRV$ 是以一批指令为基础，总是想传送一批指令到下一级。如果某条指令无法进入下一级，那么其他指令可以越过它，也就是“谁行谁上”。这种一批批进出的模式，非常类似于我们的机场和火车站。

类比于机场或火车站，可以把指令“拟人”化。如果某条指令运行到那一阶段后，可以“拟人化”的设想，如果这条指令处于这样的场景，它会怎么影响其他人，它会怎么保护自己的利益。

机场或火车站是什么？不过是很多人一起想通过一级级的流水线，走向自己的位置。我们旅行时，经过检票、安检、候车，只要有可能，尽可能的走向下一步，否则在某处等候。SSRV 的这一套流程可以让多条指令并行进入流水线，也许每条指令会在某处逗留，但只要条件许可，均可有序的走向自己的座位。容纳更多的指令的流水线，发挥了并行执行的优势。

RV32IMC 的指令分类

RV32IMC 的指令可以分为下面几类：

- **err**: 错误指令，指的是通过加载指令时，存储器汇报 `imem_err` 为高时的指令，这类指令的指令本身都是错的，不能进行解码。
- **illegal**: 非法指令。指令已正确取出，但经过解码，发现不属于任何一条合法指令。
- **sys**: 系统指令。这是合法指令，通常是和系统相关，例如 `break`、`call` 等指令。
- **fencei**: `fencei` 指令。可以理解为停顿指令，通常需要清除指令缓存。
- **fence**: `fence` 指令。可以理解为停顿指令。
- **csr**: `csr` 指令。这类指令通常是对通用寄存器和 CSR 寄存器之间的数据交换而设。

上面的 6 类指令是 **system** 指令，前四条带有跳转，后两条没有跳转。

下面的几类指令称为 **functional** 指令。

- **jalu** 指令：跳转指令，跳转地址由寄存器组的某个寄存器提供，还有可能带有一个对某个通用寄存器写入 PC 的操作。
- **jal** 指令：跳转指令，跳转地址由指令的立即数和 PC 提供，还有可能带有一个对某个通用寄存器写入 PC 的操作。
- **jcond** 指令：条件跳转指令，对寄存器组的 `Rs1` 和 `Rs0` 的数值进行评估，如果条件达到，那么执行跳转，没有，继续执行下面的指令。
- **alu** 指令：ALU 操作指令，这类指令对 `Rs0`、`Rs1` 进行处理，然后对寄存器 `Rd` 写入某个算术或逻辑操作结果。

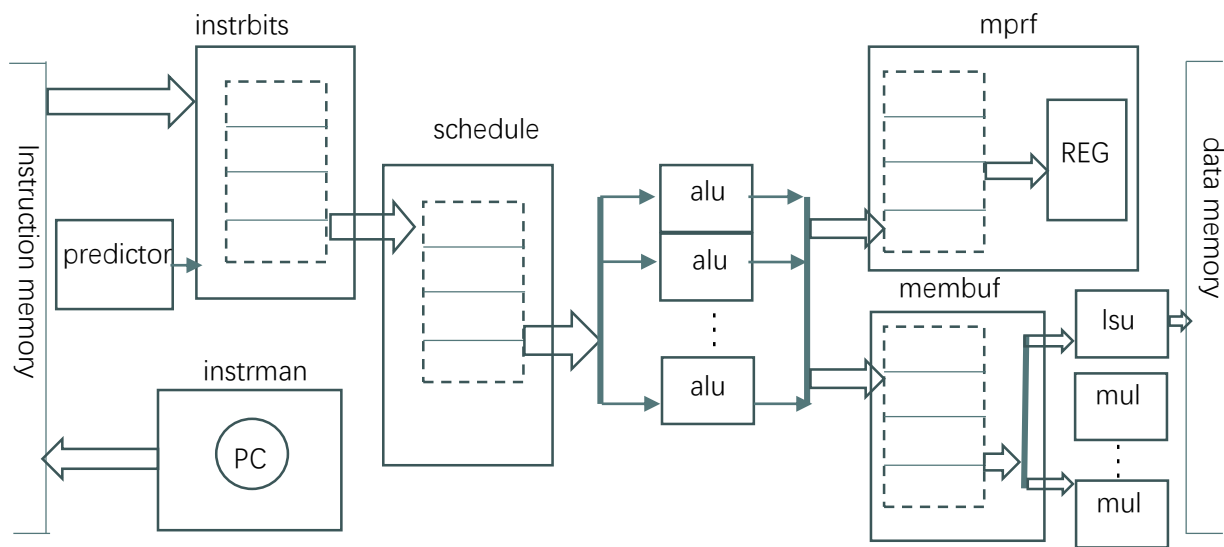
- mem 指令：LSU 指令，它会对数据存储器发起读或写操作。
- mul 指令：乘除运算指令，需要对 Rs0 和 Rs1 进行乘除操作，结果写入 Rd 内。由于乘除运算的特殊性，可能需要多个时间周期来完成。

以上是 SSRV 对指令的分类，大类有两个：系统指令和功能指令。每个大类又包含 6 个小类指令。

在对指令进行解析时，通过对指令的解读，判定现在这条指令是属于哪一类指令。同时给出它的 Rs0、Rs1 和 Rd。有了类别和这三个参数寄存器，就可以判定是否可以执行。

SSRV 的系统架构

SSRV 处理上述的两大类、12 小类指令。它的结构框图如下：



文件结构如下：

```

ssrv_top.v -----Top level
|---- instrman.v
|---- instrbits.v
|---- predictor.v
|---- schedule.v
|---- alu.v
|---- lsu.v
|---- mprf.v
|---- membuf.v
|---- mul.v
|---- sys_csr.v

```

```

define.v      ----- the defination verilog file
define_para.v ----- project parameter verilog file
include_func.v ----- common function verilog file

```

后面三个.v 类似于 C 语言的.h 文件，主要对一些通用的情况进行定义：

- define.v

Verilog 经过这么些年，已经是硬件工程师做 RTL 设计的通用语言。Verilog 类似于我们国家的文言文。最初的文言文是写在竹子上，所以要求言简意赅，但也因而艰涩难懂。但省纸张是最主要的诉求，于是人们必须学会文言文的语法来适用文言文的表达方式。后来纸张已经不是那么紧张了，但文言文已经形成影响力了。于是有人主张采用现代文来书写，这就出现了很多 Verilog 的替代品，比如 Chisel 等等。但我觉得，没有必要采取那么激进的革命去取代 Verilog，可以采用改造的方式，比如尽量消减 Verilog 的繁琐之处，类似于胡服骑射的方式。下面的几个定义是经常用到的：

```
`define N(n)    [(n)-1:0]
```

由于参数化的出现，使用[XLEN*BUS_LEN-1:0]的定义容易引起歧义，而且使用 X-1:0 这种格式并不美观，这里使用`N(XLEN*BUS_LEN)这样的定义来代替，可以直观看到这条变量的宽度是多少。

```
`define IDX(x,y) ((x)*(y))+:(y)
```

同样，在 Verilog 的接口列表中，不能使用多重数组，那么经常是采用多个信号并列在一起，比如 input `N(FETCH_LEN*`XLEN) fetch_instr 这样的定义，它表示

`FETCH_LEN 个指令同时输出。为了直观的引用第 N 个，那么这个定义只需使用：fetch_instr[`IDX(N,`XLEN)]，就可以引用第 N 条指令内容了。它表示对信号按照`XLEN 为粒度来进行分割，取第 N 份的意思。它的范围限制为：0~`FETCH_LEN-1。

```
`define FFx(signal,bits)      always @ ( posedge clk or posedge rst ) if ( rst ) signal <=
bits; else
```

对于时序逻辑描述中，总是定义时钟和复位方式。其实大多数时序逻辑并不需要这么繁琐的定义，只需要给出信号名和复位初值，那么使用`FFx 的定义即可引导时序逻辑描述电路了。这里，缺省时钟名为 clk，复位信号名为 rst。如果需要修改复位方式，例如采用低电平有效复位，只需要修改 define.v 里面的定义即可。

- `define_para.v`

这是 SSRV 用到的参数定义文件。SSRV 里面所有的参数收集在一个文件里。

- `include_func.v`

通用的 function 定义。

唯一不在框图中出现的文件是 `sys_csr.v`。它是用来处理系统指令和 CSR 指令的文件。在这个文件里，会有 CSR 寄存器的实现。它是处理系统指令和 CSR 指令的必要定义。由于仿真需要，这里面只实现了几个和仿真有关的 CSR 寄存器。重要的中断和异常，由于仿真并没有用到这些，因此，这里也没有实现。

系统指令和 CSR 指令是通过下面的接口传递过来的：

<code>input</code>	<code>sys_vld,</code>
<code>input `N(`XLEN)</code>	<code>sys_instr,</code>
<code>input `N(`XLEN)</code>	<code>sys_pc,</code>
<code>input `N(4)</code>	<code>sys_para,</code>
<code>input</code>	<code>csr_vld,</code>
<code>input `N(`XLEN)</code>	<code>csr_instr,</code>
<code>input `N(`XLEN)</code>	<code>csr_rs,</code>
<code>output `N(`XLEN)</code>	<code>csr_data,</code>

通常使用 `*_vld` 代表当前指令的有效性，`*_instr` 代表指令本身。`sys_csr` 模块在接收到这些信息后，则利用当前的 CSR 寄存器状态进行处理。最终，一般会涉及到系统跳转，那么它会启动下面的输出接口：

<code>output</code>	<code>clear_pipeline,</code>
<code>output</code>	<code>jump_vld,</code>
<code>output reg `N(`XLEN)</code>	<code>jump_pc</code>

其中 `clear_pipeline` 等同于 `jump_vld`，`jump_vld` 和 `jump_pc` 是用来启动跳转。一般出现了中断和异常，都会导入一个新的地址，取出新指令来处理这种情况，这两个信号为这种情况下的跳转而用。`clear_pipeline` 是用来清除缓冲器内的指令而用，它一般等同于 `jump_vld`。

四个功能模块

SSRV 包含了四个功能模块。所谓功能模块是指这一个.v 文件完成独立的功能，有简单的输入和输出接口，也就是有明确的功能。功能模块完全可以在其他项目上复用。这四个功能模块是：

- predictor.v
- alu.v
- lsu.v
- mul.v

它们都具有一个共同的特点，即按照接口要求给出输入，那么也会在输出上得到结果。这类功能模块一般是计算模块，这里 alu 和 mul 即是这样的计算型的功能模块。而 lsu 则是完成 lsu 操作的功能模块，predictor 是完成分支预测的功能模块。

- predictor 模块

这一模块的接口简单，功能也很单一。它为取指操作提供分支预测。实现方法是：当向指令存储器发出取指操作时，操作信号 imem_req 和 imem_addr 同时送到本模块，本模块给出这一地址的分支预测信息 imem_predict。

imem_predict 的长度为指令总线的 bit 总数除以 16，也就是对取出的指令的每一个半字给出对应的预测信息。在处理指令时，如果发现该半字对应的是条件跳转指令，那么它就以 imem_predict 的对应半字的预测信息为准，要么跳转，要么继续执行下去。imem_predict 里面的 0 表示不跳转，1 表示进行跳转。

本模块包含了 PDT_LEN 个历史条目。每个条目含有这一地址的最近前 5 次跳转信息。当 imem_addr 给出取指的地址后，每个历史条目就比对 imem_addr 和它存储的地址是否相等，然后给出这一位置的预测信息。它汇聚所有 PDT_LEN 的预测信息后，则得出本次取指的预测。

本模块的重点是如何从前 5 次跳转历史中推算出最近的跳转预测。从多次历史信息得出下一次的预测，这是有相关的实现算法，但 SSRV 给出了另外的独特角度。

在我们学习 FPGA 的构成时，是不是所有的逻辑都综合成多个四输入一输出的 LUT。LUT 是逻辑查找表，也就是根据四个输入或六个输入项，从查找表内找到对应值，输出到输出接口上。不管是什么样的逻辑，加法器、减法器还是与或非操作，都能用这样的 LUT 来代替。现在我们反其道来行之，也就是不管采用什么样的预测算法，都使用一个查找表来代替。不就是用五个历史信息来预测当前跳转嘛，那就转化为五个输入通过查找表得到一个输出。

查找表有五个输入，也就是这个查找表有 32 个预测项。有些预测项是显而易见的，比如前五次为 0，那么一定预测为 0；前五次为 5'b01010，那么绝大概率预测为 1。其他大部分是模棱两可。模棱两可的通过这样的方法来确定。首先，找一定的测试用例，修改模棱两可的选项为 0，跑一遍，记下这种预测项下的用时；然后修改为 1，再跑一遍，记下这种预测项下的用时。比较这两个用时，选用时最短的那个选项。那么用这种方法为每一个选项跑两遍，就可以得到一个查找表的所有选项。

这就类似于人工智能下围棋的思路。比如人考虑下围棋，总是基于某种算法的考虑，觉得对方是什么思路，基于这种算法分析，觉得放在某个位置为好。但如果是人工智能下围棋，它会把所有的可能性都遍历一遍，找到一个最优的解。可能人工智能的最优解暗合人的思路。因此，用一些典型用例多跑几遍，那么得出的这张查找表，一定是这些特定用例的最优解。至于遇到你无法预料的状态，也是依照这张查找表来完成。

查找表解决后，predictor 模块的主要问题是如何保存历史跳转信息。历史跳转信息是在每次遇到 JCOND 指令执行时，都会输出执行 JCOND 指令的状况，这些状况通过接口输送到本模块。本模块保存这些历史信息，通常是保留最近的 5 次信息。

下面是第一个缓冲器送来的每次跳转判决信息：

input	jcond_vld,
input `N(`XLEN)	jcond_pc,
input	jcond_hit,
input	jcond_taken

其中 jcond_vld 表示本次遇到的分支跳转指令有效；jcond_pc 是它对应的 PC 地址；jcond_hit 表示 predictor 模块给出的预测是否猜中，1 表示猜中，0 表示没有猜中；jcond_taken 表示是否发生了跳转，1 表示跳转发生，0 表示没有发生跳转。

如果 jcond_vld 有效时，jcond_pc 正好等于 PDT_LEN 个条目中的一个，表示该地址的跳转指令已经在本模块建档，那么就把本次的 jcond_taken 附在最近的 5 次上，挤掉最旧的那次历史。同时这一地址的条目挪移到最近的位置，防止新建的条目挤掉这个地址的条目。

如果 jcond_vld 有效时，jcond_pc 不是 PDT_LEN 个条目中的一个，这表示是新来的预测。这要看 jcond_hit 是否等于 1，决定如何操作。如果 jcond_hit 等于 1，表示猜中，也就是 predictor 模块给出的 0 的不跳转信息生效了，那么该地址就被丢弃，不占用有限的条目。如果 jcond_hit 为 0，表示真实发生了跳转，那么就会新建一个条目，并挤掉最旧的那条条目。新建的条目并没有历史信息，这里假定为 5'b11111，也就是假设前五次它是跳转。

CoreMark 是一次典型的测试，在这次测试中，predictor 模块一般会有 88% 左右的命中率，有兴趣可以换其他算法，看看是否能够提高预测率。

● ALU 模块

本模块是根据指令的请求，读取寄存器组中的 Rs0 和 Rs1，然后根据指令类型，计算出指令需要的各种结果。首先看看本模块的接口：

input	vld,
input `N(`XLEN)	instr,
input `N(`EXEC_PARA_LEN)	para,
input `N(`XLEN)	pc,
output `N(`RGBIT)	rs0_sel,
output `N(`RGBIT)	rs1_sel,
input `N(`XLEN)	rs0_word,
input `N(`XLEN)	rs1_word,
output `N(`RGBIT)	rd_sel,
output `N(`XLEN)	rd_data,
output	mem_vld,
output `N(`MMBUF_PARA_LEN)	mem_para,
output `N(`XLEN)	mem_addr,
output `N(`XLEN)	mem_wdata

第一部分：vld、instr、para 和 pc 这四个信号是送达的处理指令信息。vld 表示该指令是否有效；instr 是指令本身，主要用来从中提取立即数；para 是指令译码时得到的参数，它的组成时这样的：{ mem,alu,rd,rs1,rs0 }，mem 指的是 LSU/MUL 指令，会送往 membuf 模块，alu 指的是 OP 指令，另外是寄存器指示域；pc 是该指令的 pc 地址。

首先指令会通过第二部分读取 Rs0 和 Rs1，方法是给 rs0_sel 和 rs1_sel 赋予寄存器的序号，那么 rs0_word 和 rs1_word 则会得到这两个寄存器的实际值。

如果是 OP 指令，那么第三部分 rd_sel 和 rd_data 则是它写入寄存器组的最终结果。rd_sel 表示目标寄存器，如果等于 0，表示不写入；rd_data 表示写入 Rd 的实际值。

如果 LSU/MUL 指令，第四部分表示他送给的参数。mem_vld 表示有效性。mem_para 是送达的参数，它的长度是 10 bit，按序号的含义如下：

- [9]: 标志本指令是 LSU 还是 MUL 指令，1 表示 MUL 指令，0 表示 LSU 指令；
- [8:4]: Rd。如果是 MUL 指令，表示乘除法结果送达的寄存器；如果是 LSU 指令，表示加载的数据送达的寄存器，如果是写数据存储器命令，它等于 0。
- [3]: 读写数据存储器标志。MUL 指令等于 0；LSU 指令用来表示读还是写，1 表示写，0 表示读。
- [2:0]: 操作指示。如果是乘除法指令，给出是乘法还是除法，以及乘除法的处理方式；如果是 LSU 指令，给出读写的数据宽度和是否有符号数。

mem_addr 是 LSU 指令的地址，mem_wdata 是 LSU 指令的写数据。如果是 MUL 指令，则它们代表两个乘数或除数：Rs0 和 Rs1。

本模块通过获取 Rs0 和 Rs1 对即将进入两个下级缓冲器的指令进行预计算。mprf 模块获取 rd_sel 和 rd_data 这一对信号；membuf 模块获取 mem_* 等四个信号。

● lsu 模块

lsu 模块连接数据存储器，发起对数据存储器的读写操作，并收集数据存储器的应答。因此在 lsu.v 里面，会有一系列连接数据存储器的接口：

output	dmem_req,
output	dmem_cmd,
output `N(2)	dmem_width,
output `N(`XLEN)	dmem_addr,
output `N(`XLEN)	dmem_wdata,
input `N(`XLEN)	dmem_rdata,
input	dmem_resp,
input	dmem_err,

在另外一边，本模块连接 membuf 缓冲器。membuf 缓冲器积攒了多个 LSU/MUL 指令，其中 LSU 指令需要按照顺序送入本模块来完成对数据存储器的操作。下列信号是接入 LSU 指令对 LSU 操作的信号：

input	lsu_initial,
input	`N(`MMBUF_PARA_LEN) lsu_para,
input	`N(`XLEN) lsu_addr,
input	`N(`XLEN) lsu_wdata,
output	lsu_ready,

其中 lsu_ready 是表示本模块处于空闲状态，可以接收新的 LSU 操作。那么 lsu_initial 为高，表示此次新的 LSU 操作有效。在 lsu_initial 和 lsu_ready 同时为高时，这条指令的 lsu 操作被本模块接纳。

在接纳后，LSU 模块不能接收新的 LSU 操作，它会拉低 lsu_ready，表示不接收新的 LSU 操作。具体的 LSU 操作行为包含在 lsu_para、lsu_addr 和 lsu_wdata 上。在前面讲述 ALU 模块时提到 mem_para，那么 lsu_para 上的信号含义和它一样。而 lsu_addr 和 lsu_wdata 包含了此次读写操作的地址和写数据。

在完成后，会进入本模块自带的缓冲器。缓冲器的大小为 LSUBUF_LEN。在完成操作写入这个缓冲器后，lsu_ready 自动变高。如果缓冲器装满，lsu_ready 也会一直为低，不接收新的 LSU 操作。

下面是本模块输出 LSU 指令应答的接口：

output	lsu_finished,
output	lsu_status,
output	`N(`XLEN) lsu_rdata,
input	lsu_ack,

缓冲器内暂存了数据存储器的回答。一旦缓冲器不为空，那么 lsu_finished 为高，同时 lsu_status 表示缓冲器内最底部的一次操作的状态，如果为 1，表示操作错误。lsu_rdata 是缓冲器内最底部的这次操作的读数据。

如果接收方觉得这次应答可以接收，那么对 lsu_ack 拉高，表示接收此次回答。本模块在 lsu_ack 为高时，弹出最底部的操作应答，如果此时缓冲器为空，lsu_finished 则会自动拉低；如果此时缓冲器不为空，lsu_finished 拉高，lsu_status 和 lsu_rdata 仍是最底部的操作应答。

有了缓冲器，本模块实现了送入和取出的分离。但如果系统需要清除指令，也需要对缓冲器进行清除。这个信号是 `clear_pipeline`，如果这个信号为高，缓冲器则会清空，并取消本次 LSU 操作。

● mul 模块

mul 模块和 lsu 模块一样，同样连接 `membuf` 缓冲器内的指令。因此，mul 模块和 lsu 模块的接口类似：

input	mul_initial,
input `N(3)	mul_para,
input `N(`XLEN)	mul_rs0,
input `N(`XLEN)	mul_rs1,
output	mul_ready,

`mul_para` 和 `mul_rs0`、`mul_rs1` 分别是乘除法的类型以及两个乘除数。在 `mul_ready` 为高时，`mul_initial` 表示本次乘除法操作被本模块接收。

output	mul_finished,
output `N(`XLEN)	mul_data,
input	mul_ack

而上面的三个信号是乘除法结果的输出。如果 `mul_finished` 为高，那么 `mul_data` 是乘除法的结果。通过置高 `mul_ack`，本次操作的 `mul_data` 被取出。在下一个周期，如果本模块自带的缓冲器仍有结果，那么 `mul_finished` 还会保持高。本模块缓冲器的大小也是可以定义的：`MULBUF_LEN`。

如果对乘除法的实现有疑问的，可以参看另外一个 Github 项目：

<https://github.com/risc-lite/rv32m-multiplier-and-divider>。在这里面，会对本模块进行功能测试，同时也可以打开 `test_info` 里面关于乘除法的测试用例。

一次乘除法占用的周期数是不定的。本模块采用的乘除法方法类似于手算乘除法。也就是把乘除数换算成二进制，每次算一个 bit 结果，但是和其他实现不同的时，它会自动避开为 0 的位置。以乘法举例，我们都是以乘数的尾部开始，是 1 就移位相加，是 0 就避开，然后寻找下一个为 1 的地方，因此乘法的运算是和乘数的 1 的个数相关的。除法也是类似，每次都是试商一位，然后自动避开明显不对等的减法，因此除法也就是和商的 1 的个数相关。那么每次计算所占用的周期也是不定的。从这一点来说，MUL 指令和 LSU 指令有相通的地方，不过 LSU 指令是在处理器外部执行，MUL 指令是在处理器内部执行。在处理器外

部不是由处理器能决定的，但在处理器内部，可以通过多放几个乘除法器来同时展开乘除运算，那么乘除法运算的不定时间可以重叠，最终计算时间可以多个乘除法运算分摊。

第一级缓冲器：instrbits

四大功能模块之后，是四个主流缓冲器。四个功能模块是四肢的话，那这四个主流缓冲器则是骨架。缓冲器是对不确定情形的缓冲。好比一个个蓄水池，在丰水期时储满，一旦源头枯竭，可以存储一部分水，在断流时仍有指令处理；也会在下游堵住时，有一部分指令存储在蓄水池，蓄势待发。

第一级缓冲器的作用是滤除除了三类 OP、LSU、MUL 指令以外的其他指令。在前面的简述中，只是举了简单的 JAL 和 JALR 指令的处理，也就总结出来了本级缓冲器的处理理念：那就是要么立即执行，要么等待信号。实际上，其他指令远非这两种，其他难以处理的指令也就会给管理这一“机场”带来了难题。就如同医院，最熟悉的是外科和内科，但并非只有外科内科，什么心理、化验、放射等等，都是因为有了新病症和新的挑战，才让医院有了新的应对机制。这一“检票处”也是如此，最初只是检验放行，但来的指令非常复杂，那么它的应对手法也要升级。

本级缓冲器的来源是从指令存储器取出的指令比特。关于取指的地址，有一个专门的模块来进行管理，这就是 instrman 模块。它的输入输出接口如下：

output		imem_req,
output `N(`XLEN)		imem_addr,
input		imem_resp,
input `N(`BUS_WID)		imem_rdata,
input		imem_err,
input		jump_vld,
input `N(`XLEN)		jump_pc,
input		branch_vld,
input `N(`XLEN)		branch_pc,
input		buffer_free,
output		imem_vld,
output `N(`BUS_WID)		imem_instr,
output		imem_status

第一部分连接指令存储器。它通过 imem_req 的有效信号，来发出以 imem_addr 为取指地址的操作。在接下来的若干周期内，指令存储器如果认为取指完毕，那么它以 imem_resp 为高电平表示取指应答。另外，取指的数据位于 imem_rdata，取指的状态位于 imem_err。

imem_rdata 的比特宽度是可定制的，可以一次取出 32、64、128 等 bit。imem_err 指示当前取指是否有异常，如果有异常，那么在以当前取指比特进行译码时，判定为错误指令，需要引入异常处理程序。

imem_resp、imem_rdata 和 imem_err 会通过最后一部分的 imem_vld、imem_instr 和 imem_status 输送到本级缓冲器。

中间的信号是用来管理取指操作的。jump_vld/jump_pc 是由系统程序发起的跳转，branch_vld/branch_pc 是由 JAL、JALR 和 JCOND 指令引发的跳转。如果两者都发出，以前者为准。如果没有这两个跳转，那么以本级缓冲器发起的 buffer_free 信号为准。如果它为高，表示本级缓冲器可以接收指令比特，可以发起取指操作；如果它为低，那么表示本级缓冲器已满，不能再发起取指操作了。

现在打开 instrbits.v 这个文件，这就是本级缓冲器的 verilog 代码，可以看到下面的接口：

```
input
input  `N(`BUS_WID)
input
input  `N(2*`BUS_LEN)

instr_vld,
instr_data,
instr_err,
instr_predict,
```

这就是指令存储器的指令比特输入接口，它连接 instrman 模块的 imem_vld/instr/status 这三个接口。最后一个接口：instr_predict 来自于另外一个模块 predictor。则是一个内部模块，它观察取指信号：imem_req 和 imem_addr，为取指的地址给出 JCOND 指令的跳转预测信息。

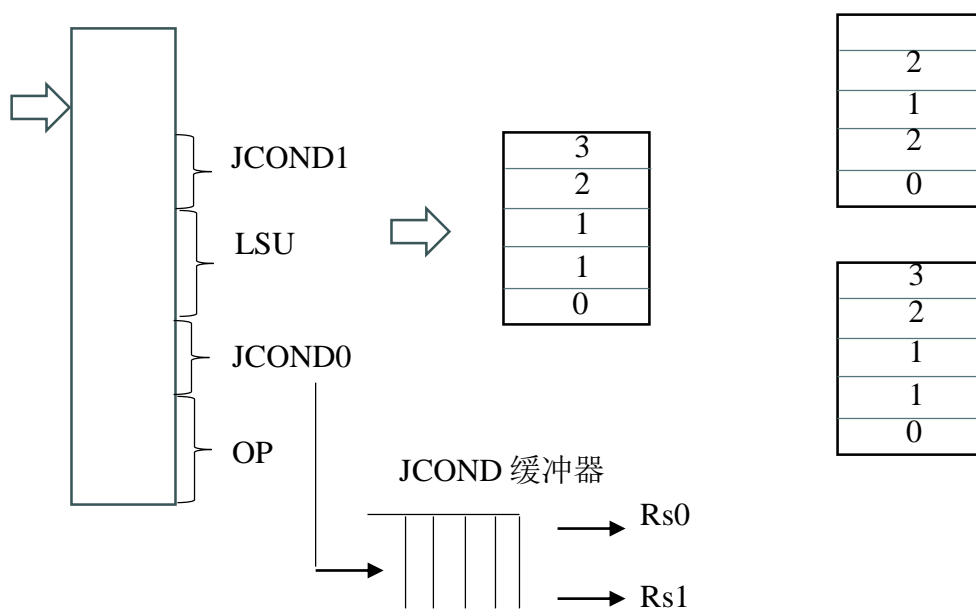
这里要讲到 JCOND 指令。这是一种条件跳转指令。它会从寄存器组取出 Rs0 和 Rs1，按照某种规则比较 Rs0 和 Rs1，如果符合要求，进行跳转；如果不符合要求，继续执行 JCOND 后续的指令。这是一个有两个选择的指令。一般情况，可以按照 JALR 指令，让 JCOND 指令在缓冲器内等待，等到 Rs0 和 Rs1 可用，也就是后续流水线上已经没有以这两个寄存器为 Rd 的指令了，然后取出它们，进行比较，然后进行跳转或继续执行。

但这种效率太慢。举了例子，检票处发现了 JCOND 指令，发现 JCOND 指令需要两个寄存器 Rs0 和 Rs1 可用才能执行，于是让 JCOND 指令等待这两个寄存器可用。可是紧随 JCOND 指令后面的指令会提议，可以让它们提前进入流水线，因为它们后面的指令是 JCOND 指令将要跳转的目标之一。后面的指令进入等候大厅、候机厅去等候，然后让 JCOND 指令留下来继续等待“兑现”它的跳转目标。当然，因为后面的指令的是否合法未定，因此它们绝不能离开这两个地方去登记。一直到 JCOND 指令能够获取 Rs0 和 Rs1，然

后得出真正的跳转方向。如果最终是不跳转，那么让已经预先进入等候的指令转正，可以获准离开；如果最终是跳转，那么清除预先进入等候的指令。

预测不跳转是最便利的预测，因为接下来的指令已经预取在缓冲器内，而另外的跳转选择，则类似于 JAL 指令，需要修改取指的 PC 和清空缓存器。

正因为只有两个选项，因此可以收集 JCOND 指令的历史跳转信息，形成一个跳转预测，在 JCOND 指令出现时，以预测为准，让 JCOND 指令后面预测执行的指令也能进入流水线，也就是后续三个缓冲器。这些因为预测进入的指令只有“等候”权，不能退休或进入 LSU 或 MUL 模块的。但当该 JCOND 指令判决为预测是准确的后，已经进入三个缓冲器的预测指令可以“转正”，也就是获取正常指令的权力。



如上图所示，为了处理多个 JCOND 指令，设置 JCOND 缓冲器，他可以暂存多个 JCOND 指令。而每个 JCOND 指令在进入 JCOND 缓冲器后，它后续的指令也能放行进入后面三个缓冲器。为了管理这些预测执行的指令，为每一条指令配备一个计数器属性 level。

level 属性是指预测执行的层级。它表示这条指令前面有多少 JCOND 指令未做判决。如上图所示，如果说 JCOND 缓冲器内有三条 JCOND 指令，那么 OP 指令的 level 属性为 3，而后面的 LSU 指令因为隔了一个 JCOND0 指令，那么这条 LSU 指令的 level 属性为 4。因此这

些带有不同 level 属性的指令进入后面三级缓冲器，只有 level 属性为 0 的指令是可以自由离开，而 level 属性不为 0 的指令只能“居留”在三个缓冲器内，不能离开。

JCOND 缓冲器最底部也就是最先进入的那条 JCOND 指令，有权去提取 Rs0 和 Rs1 来判决它的跳转情况。判决的情况分为两种，一种是预测成功，那么它发布一个信号：level_decrease。后面三个缓冲器的指令在收到这个信号后，对它们各自的 level 属性做减一归零操作，也就是如果等于 0，不变，如果不等于 0，那么减去一。于是 level 属性等于 1 的指令变成 level 属性为 0 了，也就获得离开的权限了。而其他 level 属性的也就进入了一个更接近“正式”指令的地步了。

另外一种预测错误。那么基于这个错误预测进入三个缓冲器的指令都得清除出去。这条 JCOND 指令后续的 JCOND 指令都是基于这个错误预测而进入 JCOND 缓冲器的，那么在知道预测错误后，JCOND 缓冲器本身就得清除。同样，这种情况也会发布一个信号：level_clear，后面三个缓冲器在收到这个信号后，清除所有 level 属性不等于 0 的指令。

通过 level 属性和两个信号：level_decrease 和 level_clear，可以对 JCOND 指令带来的预测执行做到有收有放，良序管理。

到这里，可以对 instrbits 模块的接口做一个全面描述：

input	jump_vld,
input `N(`XLEN)	jump_pc,
output	branch_vld,
output `N(`XLEN)	branch_pc,
output	buffer_free,

第一部分，是取指操作的控制。有两个途径对 PC 进行修改，分别是系统指令的 jump_* 和跳转指令的 branch_*。如果没有跳转时，以 buffer_free 为准。buffer_free 反应 instrbits 缓冲器的剩余空间状况。如果 instrbits 缓冲器能够接纳一次取指操作的指令比特，那么 buffer_free 为高，可以启动读取指令操作。

input	instr_vld,
input `N(`BUS_WID)	instr_data,
input	instr_err,
input `N(2*`BUS_LEN)	instr_predict,

指令到来后，通过 `instr_*` 这一系列信号送来。同时送来的包括分支预测信号 `instr_predict`。它是对取指操作对应的地址的分支跳转进行预测。

output	jcond_vld,
output `N(`XLEN)	jcond_pc,
output	jcond_hit,
output	jcond_taken,

predictor 模块能够对分支指令进行跳转预测，建立在它能够获得分支指令的历史跳转信息。instrbits 模块通过 `jcond_*` 信号发出它在 JCOND 缓冲器处理的 JCOND 指令的跳转情况。

output	sys_vld,
output `N(`XLEN)	sys_instr,
output `N(`XLEN)	sys_pc,
output `N(4)	sys_para,
output	csr_vld,
output `N(`XLEN)	csr_instr,
output `N(`XLEN)	csr_rs,
output `N(`RGBIT)	csr_rd_sel,

接下来是系统指令的转出途径。instrbits 缓冲器在遇到 SYS 指令或 CSR 指令，本身不能处理，它可以抄送到 `sys_csr` 模块来进行处理，这两个“抄送”路径是对应 SYS 指令或 CSR 指令的。

output `N(`RGBIT)	rs0_sel,
output `N(`RGBIT)	rs1_sel,
input `N(`XLEN)	rs0_word,
input `N(`XLEN)	rs1_word,

这是 JCOND 缓冲器对 JCOND 指令提取 Rs0 和 Rs1 的接口。JCOND 指令通过发出 Rs0 和 Rs1 对应的寄存器序列号，寄存器组会给出对应的数据。

input `N(`SDBUF_OFF)	sdbuf_left_num,
output `N(`FETCH_LEN)	fetch_vld,
output `N(`FETCH_LEN*`XLEN)	fetch_instr,
output `N(`FETCH_LEN*`XLEN)	fetch_pc,
output `N(`FETCH_LEN*`EXEC_PARA_LEN)	fetch_para,
output `N(`FETCH_LEN*`JCBUF_OFF)	fetch_level,

这是送往下一级 schedule 缓冲器的信号接口。操作流程是这样，schedule 缓冲器通过信号 `sdbuf_lef_num` 给出它的缓冲器的剩余空间，由 instrbits 缓冲器确保送入的指令数目不能溢

出。当然，送来的指令最大宽度为:FETCH_LEN，因此实际送来的指令一定小于：FETCH_LEN 和 sdbuf_left_num 的最小值。

使用 vld 表示对应指令的有效性，下一级 schedule 缓冲器通过它确认实际送来的条数，1 bit 对应一条指令的有效性。instr 是指令的内容，统一为 32 bit，如果指令是 16 bit，那么只有低 16 bit 有效。pc 是指令对应的 PC。para 是指令的种类和 Rs0、Rs1 和 Rd 的参数域。level 是指令的 level 属性。

input input	`N(`RGLEN) pipeline_instr_rdlist, pipeline_is_empty,
----------------	--

这两个信号是后面三个缓冲器送来的状态信号。前者指的是流水线，指的是三个缓冲器内所有指令的 Rd 的集合。JCOND 缓冲器使用它来查看它最底部 JCOND 指令的 Rs0 和 Rs1 的有效性，这里的有效性指的是 Rs0 或 Rs1 是否是某条指令的 Rd，如果是，那么因为流水线中的这条指令没有处理，Rd 没有结果，因此要提取 Rs0 或 Rs1 的要求无效。只有等到 pipeline_instr_rdlist 中 Rs0 和 Rs1 的对应 bit 为 0 后，JCOND 指令才能完成评估。

后者指的是流水线内为空，也就是后面的三个缓冲器没有包含任何指令，这是系统指令的处理条件。

output output	level_decrease, level_clear
------------------	--------------------------------

最后是 JCOND 缓冲器发出的有关 level 属性的控制信号，是对后面三个缓冲器发出的。level_decrease 是在预测为正确时发布的信号，是让紧随这条 JCOND 指令之后的指令“转正”所用；level_clear 是在预测为错误时发布的信号，是清除所有预测层级的指令。

在了解它的接口后，后面讨论它是如何进行译码以及如何处理指令的。这一级的输出是分为两种，一种是有寄存器打拍，也就是 FETCH_REGISTERED 的定义打开，此时输送到 schedule 缓冲器的指令必须经过一级寄存器暂存；一种是没有寄存器打拍，直接输送到下一级。

对于后者来说，由于没有寄存器打拍，那么每次能够获准译码的数目是 sdbuf_left_num 和 FETCH_LEN 之间的小者。对于前者来说，需要比较在寄存器打拍的指令数目与 sdbuf_left_num 的关系，如果小于 sdbuf_left_num，也就是在寄存器暂存的指令可以全部输

送到 schedule 缓冲器，那么可以进行译码的数目为 `FETCH_LEN`；如果大于 `sdbuf_left_num`，那么表示一次取不完寄存器暂存的指令，仍有剩余，那么可以进行译码的数目是 `FETCH_LEN` 减去剩余的指令条数。

在代码内部，这个获准译码的条数统一整合成一个信号：`eval_capacity`。这就是本次能够译码的最大条数。之所以统计出译码的最大条数，是希望译码后一定要获得处理，如果全部是送往下一级的三类指令，那么可以直接抄送到下一级缓冲器即可。

译码器译码的对象由两部分组成，一是缓冲器内预先保留的指令比特，二是现从取指操作中输入的指令比特。缓冲器会把内部保存的指令比特和从指令存储器进入的指令比特连接在一起，作为译码器译码的对象。这样做的好处是可以最大限度的获取译码的指令比特。比如我们去景点排队，可能窗口排了几个人，也可能空无一人，如果有人排队，那么你得排在后面，轮的上，本轮就可以进入；如果没人，那你就跟上，直接进去。

SSRV 的译码器是一个 function，它位于 `rtl/include_func.v`。打开这个文件，找到 `rv_para` 这个 function:

```
`define RV_PARA_LEN          (11+`RGBIT*3)

function `N(`RV_PARA_LEN) rv_para(input `N(`XLEN) i,input err);
```

只需要为这个 function 输入指令本身和取指时的错误指示，那么就可以输出这条指令的相关属性：

```
rv_para =
{ err,illegal,sys,fencei,fence,csr,jalr,jal,jcond,(mem|mul),(alu|jal|jalr),rd,rs1,rs0
};
```

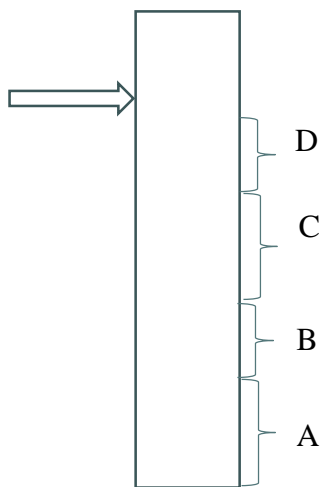
在前面的 RV32 指令分类中，分为两大类，一是系统指令，一是功能指令。每大类又有六个小分类。在前面的 `err`、`illegal`、`sys`、`fencei`、`fence`、`csr` 是系统指令；后面的 `jalr`、`jal`、`jcond`、`mem`(LSU 指令)、`mul` 和 `alu`(OP 指令)是功能指令。后面还会有 `Rd`、`Rs0` 和 `Rs1` 域。

(`mem|mul`)合在一起是因为 LSU 指令和 MUL 指令合并处理；(`alu|jal|jalr`)合并在一起是因为 JAL 和 JALR 指令都带有和 OP 指令类似的写寄存器操作。当 JAL/JALR 指令在本级缓冲器时，它会执行它的跳转功能。当它进入下一级后，它和普通的 OP 指令功能相同，也就是写入 PC 到某一个寄存器内。

在这个分类中，前面 9 个分类：err、illegal、sys、fencei、fence、csr、jalr、jal、jcond 会在本级缓冲器进行处理，那么在输入到一级时，这 9 个分类会去除，只剩下：(mem|mul)、(alu|jal|jalr)这两个指令分类和 rd、rs1、rs0 这三个寄存器指示。

在译码没有出现这 9 个分类时，这是最容易处理的场景，只需要把译码的指令送入本级寄存器打拍，或者直接输送到下一级。如果出现了这 9 个分类中的任何一个，那么这 M 个同时译码的指令变成这样的指令组合：0~N 条普通指令+ 9 个分类中任意一个+无效指令。也就是这 9 个分类指令可以“封杀”后续指令，让这些指令无效。

下图是同时译码多条指令的示意图，其中 A 位置，也就是第一条指令最为关键，后面称呼这条指令为 A 口指令。



这 9 个分类分别进行讨论：

- err、illegal、sys、fencei 指令

这四类指令具有一个共性，也就是一定会引发跳转操作，也就是会让 sys_csr 模块发起 jump_vld/pc。

如果这四类系统指令不是出现在 A 口位置，那么本周期的操作是发送这四类指令之前的普通指令进入下一级，而缓冲器移除这些普通指令的指令比特。那么在下一个周期时，这条系统指令位于 A 口位置。

这四类指令抄送给 sys_csr 模块的条件有两个：1，JCOND 缓冲器为空，表明它不是预测而来的指令；2，pipeline_is_empty 有效，也就是后面三个缓冲器已经清空。

如果这两个条件达不到，那么它会一直“霸占”A 口，让 JCOND 缓冲器或后面三个缓冲器去清空它们包含的指令。因为它“霸占”A 口，导致没有新的指令加入，它们清除各自包含的指令只是时间问题。

如果条件达到，那么它会通过 sys_* 的接口送入 sys_csr 模块。在条件达到时，这条指令仍会占据 A 口，也会在占据时不断送给 sys_csr 模块，需要清除它的条件是触发 jump_vld/pc 来进行清除缓冲器。

● fence、csr 指令

如果这两类系统指令不是出现在 A 口位置，那么本周期的操作是发送这两类指令之前的普通指令进入下一级，而缓冲器移除这些普通指令的指令比特。那么在下一个周期时，这条系统指令位于 A 口位置。

这两类指令需要等待这两个条件：1，JCOND 缓冲器为空，表明它不是预测而来的指令；2，pipeline_is_empty 有效，也就是后面三个缓冲器已经清空。

如果这两个条件达不到，那么它会一直“霸占”A 口，让 JCOND 缓冲器或后面三个缓冲器去清空它们包含的指令。因为它“霸占”A 口，导致没有新的指令加入，它们清除各自包含的指令只是时间问题。

如果达到，那么它会失去它们的特殊性，成为一条普通指令，也就不再“阻碍”后续指令的译码解析。同时，如果是 csr 指令，那么它会激活 csr_* 等信号，抄送给 sys_csr 模块，来对 CSR 寄存器或普通寄存器组进行操作。

● jalr 指令

JALR 指令一定要在 A 口才能获得处理。如果不在，参见前面系统指令的应对方法。

JALR 指令会等待两个条件：一是 JCOND 缓冲器为空，表明它不是预测而来的指令。另外还因为它也需要读 Rs0，而读 Rs0 的通道在 JCOND 缓冲器不为空时被 JCOND 指令占用，因此只能等 JCOND 缓冲器清空时，才能让 JALR 指令读 Rs0。二是 pipeline_instr_rdlist 中 Rs0 对应的 bit 不等于 1。JALR 指令并非系统指令，它不需要等到流水线全部为空才去执

行，它只需要等到它的 Rs0 不再成为流水线上某条指令的 Rd 即可，此时它可以安全读出它的 Rs0 去执行跳转功能。

在这两个条件不满足时，它会一直占着 A 口。如果条件满足，那么它会通过 rs0_sel 和 rs0_word 接口读出 Rs0，计算出跳转地址，置位 branch_vld 和 branch_pc 来进行跳转。同时，这条 JALR 指令会当成普通指令送入寄存器打拍或直接进入下一级，去执行它可能有的写 PC 进入某个寄存器的功能。

branch_vld 会清除 instrbits 缓冲器，这个缓冲器会为新地址的指令准备一个全空的接纳空间。

● jal 指令

JAL 指令的处理最简单。出现了 JAL 指令时，会把这条指令提取出来，然后生成新的跳转地址，对 branch_vld 和 branch_pc 进行置位来引发跳转。它的处理类似于 JALR 指令。

● jcond 指令

JCOND 指令首先会看 JCOND 缓冲器是否为满。

如果 JCOND 缓冲器为满，那么这条 JCOND 指令会和系统指令一样，占据 A 口进行等待。如果 JCOND 缓冲器不腾出空间，那么这条 JCOND 指令会一直在 A 口等待。

如果 JCOND 缓冲器有剩余空间，那么这条指令会参照它的 predict 来进行操作。如果 predict 等于 1，那么表示预测跳转，它就如同 JAL 指令一样处理。如果 predict 等于 0，表示预测不跳转，那么它就视作一条 NOP 类型的普通指令处理，它后面的指令也就能获得译码机会。此时，如果再次遇见了一条 JCOND 指令，那么因为 JCOND 缓冲器一次只能接纳一条 JCOND 指令，那么这个第二条 JCOND 指令将等同于 JCOND 缓冲器已满的情况处理，虽然 JCOND 缓冲器并不一定满。在下一个周期，这条 JCOND 指令将获得进入 JCOND 缓冲器的机会。

这 9 个分类在本级缓冲器得到妥善处理，目的是为了让输出到下一级的指令简单——只有两种形式，因此可以发挥超标量和乱序发射的优势。

第二级缓冲器：schedule

本级缓冲器从上一级缓冲器接收指令，然后给下一级的两个缓冲器分配指令。在送到下一级的两个缓冲器之前，会取出指令的 Rs0 和 Rs1，如果是单周期指令，得到 Rd 的数据；如果是多周期指令，会得到两个操作数。

打开 schedule.v，它的输入和输出接口有下面两个部分：

```
output `N(`SDBUF_OFF)          sdbuf_left_num,
input  `N(`FETCH_LEN)          fetch_vld,
input  `N(`FETCH_LEN*`XLEN)     fetch_instr,
input  `N(`FETCH_LEN*`XLEN)     fetch_pc,
input  `N(`FETCH_LEN*`EXEC_PARA_LEN) fetch_para,
input  `N(`FETCH_LEN*`JCBUF_OFF) fetch_level,

output `N(`EXEC_LEN)           exec_vld,
output reg `N(`EXEC_LEN*`XLEN) exec_instr,
output reg `N(`EXEC_LEN*`XLEN) exec_pc,
output reg `N(`EXEC_LEN*`EXEC_PARA_LEN) exec_para,
output reg `N(`EXEC_LEN*`JCBUF_OFF) exec_level,
output reg `N(`EXEC_LEN*`MMCMB_OFF) exec_order,
```

前半部分是接收上一级的指令，后半部分是输出给 ALU 模块的指令。相比较这两部分，发现在输出时增加了 exec_order 信号。这就是指令的 order 属性。

order 属性是为了单周期指令准备的。假如有下列一串指令进行处理，为这一串指令准备一个 order 属性。order 属性是该指令之前有多少条多周期指令没有执行，严格来说这个定义只对单周期指令有效。两个相邻的指令的 order 属性是：上一条指令+本条指令是否为多周期指令。

```
OP0    → 0
LSU0    → 1
OP1     → 1
OP2     → 1
MUL0    → 2
OP3     → 2
LSU1    → 3
OP4     → 3
```

现在这些指令流动到下一级的两个缓冲器内。上面的是 mprf 缓冲器，下面的是 membuf 缓冲器。mprf 缓冲器内的指令带有 order 属性。如果 order 属性为 0，代表它前面没有多周期指令，它可以安全的写入寄存器组。其他 order 属性不为 0 的指令，代表前面还有多周期指令，也就代表着出现异常的变数，也就不能写入寄存器组。

OP4→3
OP3→2
OP2→1
OP1→1
OP0→0

LSU1
MUL0
LSU0

在图中，OP0 指令是可以写入寄存器组的。如果 LSU0 指令安全执行没有出现异常，那么它会发出一个信号：mem_release，这个信号在 schedule 模块的接口中有，那么 mprf 缓冲器内的指令的 order 属性都减一，于是 OP1 和 OP2 指令新成为 order 属性为 0 的指令，可以获准写入寄存器组。再下一个周期，MUL0 指令安全执行，也会发出 mem_release 信号，于是 OP3 指令的 order 属性也因为两次减一成为新的 order 属性为 0 的指令，它也能写入寄存器组。

注意这里的减一是指归零式减一，也就是如果等于 0 了，也就不用减了。实际上 membuf 缓冲器一个周期内可以退休多条指令。如果 LSU0 和 MUL0 同时退休，那么 mem_release 等于 2，此时 OP1、OP2、OP3 会同时成为新的 order 属性等于 0 的指令。

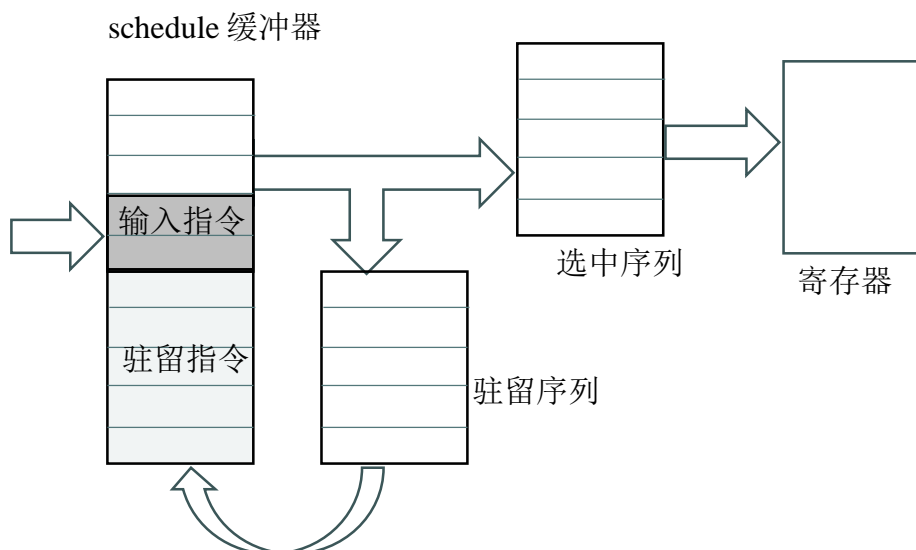
通过 order 属性，可以让 mprf 缓冲器和 membuf 缓冲器内的指令同步退休。order 属性是在进入 schedule 缓冲器时为每一条指令添加上。选择在这里添加是因为在进入 schedule 缓冲器时指令是顺序的，一旦从 schedule 缓冲器内离开，则指令的顺序是乱序的，这个时刻进行添加是理想的。

在进入本级缓冲器的指令之前，多周期指令没有退休时存在两个地方，一个是 membuf 缓冲器，一个是 schedule 缓冲器。只需要把这两个地方的多周期指令的个数统计起来，即为计算 order 属性的起点。

指令就此包含五个域，一个是指令的 32 bit 内容，用 instr 表示；第二个是对应的地址，32bit，用 pc 表示；第三个是参数，包含 mem、alu、rd、rs1、rs0，表示单周期(alu)或多周期(mem)指令，以及指令的寄存器域，有 17 bit，用 para 表示；第四个是预测层级，长度随

JCOND 缓冲器而定，最大为 JCOND 缓冲器的大小，用 level 表示；第五个是前面多周期个数，也就是 order 属性，它最大等于 membuf 缓冲器和 schedule 缓冲器大小之和。

本级缓冲器的作用如下图所示，它把本级缓冲器的驻留指令和输入指令连接在一起，作为候选指令，分裂成两个序列，一个是选中序列，输出到寄存器，然后连接到 ALU 模块；另外一个剩下的驻留序列，会仍保留在 schedule 缓冲器，作为下一个周期的驻留指令。



这个过程每个周期都会进行一次。形成选中序列和驻留序列是本级缓冲器永远的主题。本级缓冲器内的候选指令是进入哪一个序列会有两个因素影响。一个是资源因素，如果是单周期指令，那么 mprf 缓冲器必须有剩余的空间容纳这条指令；多周期指令必须在 membuf 缓冲器有位置。另外一个是指令冲突，也就是它前面的指令因为它处于某种状态，导致这条指令不能进入选中序列。

第一个因素解决的方法是让后面的两级缓冲器给出剩余空间的大小，然后每分配一个减去一，一直到零为止。第二个因素解决的方法是形成一个 Rs 黑名单和 Rd 黑名单，每条位于前列的指令考察它处于当前状态时导致哪些寄存器不能成为 Rs 或 Rd，这些 Rs 和 Rd 的禁地合在一起形成两个黑名单，那么这条指令比对黑名单，可以知道它是否能够进入选中序列。

重点是第二个因素，当选择候选指令中的第一条时，它可能来自于输入指令，那么位于它前列的指令都在 mprf 缓冲器和 membuf 缓冲器；还可能来自于驻留指令，这种情况下，

mprf 缓冲器和 membuf 缓冲器内包含了它前列的指令或越过它的某些指令。注意，如果是驻留指令，它允许后面的指令越过它进入这两个缓冲器，已经代表它的越级不会对这条指令形成影响，否则它就不能越级。因此，对于驻留指令，mprf 缓冲器和 membuf 缓冲器存放的是前列指令和默许无影响的某些后列指令。

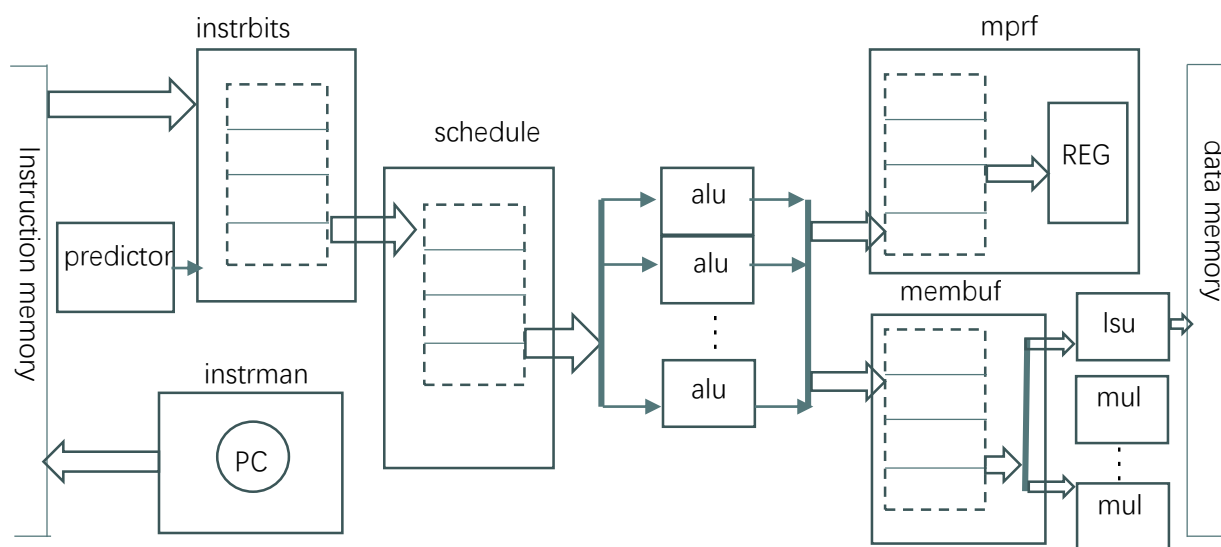
mprf 缓冲器和 membuf 缓冲器会形成第一条候选指令的 Rs 和 Rd 黑名单。

对于单周期指令，一旦它进入 mprf 缓冲器，它就算是准退休状态。原因是我们常理解的狭义的寄存器组是指 32 个寄存器 R0~R31，但 mprf 模块扩大了这个寄存器组的含义，是一种“广义”的寄存器组。只要送到 mprf 缓冲器，那么就算是写进这 32 个寄存器组。当然，这个“广义”不仅包含 mprf 缓冲器的不同 Rd 的写入，还包括 membuf 缓冲器，即将退休的多周期指令对于寄存器组的写入。这些写入只是送达 mprf 模块，但只要送到，即可认为已经写入，因为在取数据的时候，是有三个取数据途径的：

1. 在寄存器组 R0~R31 中取出对应序号的数据；
2. 在 mprf 缓冲器中查找对应序号的写入，如果有一个或多个，以最后一个为准；
3. 在 membuf 缓冲器送来的寄存器组写入中查找对应序号，如果有，找出写入数据。

如果第 3 步能查到对应序号，那么以第 3 步的写入数据为最新数据；如果没有，则以第 2 步为准；如果都没有，才是第 1 步的数据为准。

因此，mprf 模块虽然包含了很多未完成的 Rd 写入，但 ALU 模块在取 Rs0 和 Rs1 是感觉不到这种“未完成”的，它一定可以取出“已完成”状态时的最新的数据。



membuf 缓冲器不能和 **mprf** 缓冲器一样，因为它存放的多周期指令。虽然这些多周期指令的 **Rs0** 和 **Rs1** 经过 **ALU** 模块已经取现，但它们的 **Rd** 却不能获取它们的写入数据。这些写入数据必须在它们把两个操作数送给对应的 **LSU/MUL** 模块后，经过不定周期后，才能获取写入数据。

因此，**membuf** 缓冲器内存放的指令，都有一个不知道写入数据的 **Rd**。但是最底部的那一条除外，因为它作为最先进入缓冲器的指令，是有可能在本周期获得 **LSU/MUL** 模块的输出，从而知道写入数据的。

这种“有可能”是来源于 **LSU** 模块和 **MUL** 模块处理指令的不确定性。**LSU** 模块确定它的输出是依靠外界送来的 **dmem_resp**，这个信号是数据存储器送来的，它有权根据数据存储器的状态来发出 **dmem_resp**。可能这段数据存储器的读写较慢，那么 **dmem_resp** 可能延时若干周期后给出。**MUL** 模块也是类似，在提交两个乘除数后，并不能确定等多少周期后，乘除结果会给出。

在下一个周期时，最底部的那条指令是可能处于两种状态的。一是它在本周期给出了写入数据，那么 **membuf** 缓冲器对寄存器组发起写入 **Rd** 操作，本条多周期指令退休；它对于 **schedule** 缓冲器没有影响，因为它能在 **mprf** 缓冲器获取这个最新结果。二是本周期不能给出写入数据，那么 **membuf** 缓冲器内的 **N** 条指令的 **Rd** 都无法提供写入数据，也就成了 **schedule** 缓冲器的黑名单。

注意，为 schedule 缓冲器的首条候选指令准备黑名单时，一定要注意是下一个周期时的 membuf 缓冲器的状态。因此首先需要排除当前退休的指令，然后在排除外的指令中，第一条指令其实是“灰”名单，然后剩余的才是黑名单。

之所以把第一条指令列为灰名单是因为，在选中序列进入 ALU 模块的那个周期里，这条指令可能退休，也可能不退休，如果不退休，ALU 模块取出这条指令的 Rd；如果退休，是可以取出这条指令的 Rd 的。

这也就有了灰名单指令带来的“两头下注”策略。如果一条指令只和灰名单上的这个 Rd 冲突，那么允许它既出现在选中序列，又可以出现在驻留序列。当然在这两个序列中，只有一个是真的。哪一个是真的由灰名单指令是否退休决定的。如果灰名单的这条指令恰好退休，那么出现在选中序列的“两头下注”指令是真的，它就可以直接读出灰名单指令提交的最新的那个 Rd，而这个 Rd 数据正是“两头下注”指令需要的。如果灰名单的这条指令没退休，那么出现在选中序列的是假的，不会进入这两个缓冲器；而出现在驻留序列的“两头下注”指令是真的，会再次参与 schedule 缓冲器形成两个序列的过程。

这种打“提前量”的方式非常重要，因为在 CoreMark 测试中很多这种上一个多周期指令的结果必须在下一个指令中用到，下面的汇编程序反应了这一点：

```
ec4:    00452803      lw    a6,4(a0)
ec8:    00281883      lh    a7,2(a6)
ecc:    00e88463      beq   a7,a4,ed4 <core_list_find+0x26>
ed0:    4108         lw    a0,0(a0)
ed2:    f96d         bnez  a0,ec4 <core_list_find+0x16>
```

上面的 ec4 指令会读数据存储器，并写入读数据到寄存器 a6。而寄存器 a6 正好是指令 ec8 的地址的 Rs0。于是可以想象不进行“两头下注”的情形：ec4 指令退休，传递到 schedule 缓冲器，ec8 指令进入选中序列；在下一个周期 ec8 指令通过 ALU 模块使用 ec4 的 Rd 来形成 ec8 指令的读地址，进行数据存储器访问。

进行“两头下注”可以在 ec4 指令的 Rd 有了读数据的同时，启动 ec8 指令的读操作，于是会节省一个周期。不要小瞧了一个周期，因为这两条指令会执行相当多的次数。用专业术语说，这叫做数据前推。在 ec4 指令的数据到来，写入寄存器组的那一个周期，立即对下一条指令 ec8 启用这个写入结果。

所以说 mprf 是一个广义的寄存器组，所有到来还没有进入寄存器组的那些操作，都被 mprf 模块算成已经完成，因为 mprf 模块会综合所有正在执行的写寄存器组操作来形成最终输出的 Rs。

mprf 缓冲器是不会对本级缓冲器的首条指令形成任何约束的，mmbuf 缓冲器会形成两个约束，一个是 mmbuf_check_rdnun，是灰名单寄存器，只有一个 Rd，因此大小为 5 bit；另外是 mmbuf_check_rdlst，是后面剩余的多周期指令形成的 Rd 的集合，大小为 32 bit。

这两个条件是本级缓冲器首个候选指令的约束条件。需要根据这个指令与灰名单寄存器和黑名单列表的关系进行判决这条指令的归属，是进入选中序列，还是进入驻留序列，还是两个都进。然后根据这条指令的归属得到一个 Rs 和 Rd 的禁止，并加入到黑名单当中。然后更新的黑名单传递给第二条候选指令，这就是指令冲突判决的方法。

使用代码描述如下：

```
wire      hit_rg = check_blacklist(para,rd_checklist[i],rs_checklist[i]);

.....

wire      exec_idle = exec_active[i];
wire      mem_idle = mem_active[i];
wire      rf_idle = rf_active[i];

wire      hit = hit_rg|(~exec_idle)|(mem & ~mem_idle)|(alu & ~rf_idle);
wire      preload = check_gray(para,check_num);

wire      alu2exec = go_vld[i] & alu & ~hit;
wire      alu2sdbuf = go_vld[i] & alu & (hit|preload);
wire      mem2exec = go_vld[i] & mem & ~hit;
wire      mem2sdbuf = go_vld[i] & mem & (hit|preload);

.....

assign rd_checklist[i+1] =
rd_checklist[i]|(alu2sdbuf<<rd)|(alu2sdbuf<<rs0)|(alu2sdbuf<<rs1)|(mem2exec<<rd)|(mem2
sdbuf<<rd)|(mem2sdbuf<<rs0)|(mem2sdbuf<<rs1);
assign rs_checklist[i+1] =
rs_checklist[i]|(alu2exec<<rd)|(alu2sdbuf<<rd)|(mem2exec<<rd)|(mem2sdbuf<<rd);
```

信号 hit_rg 代表黑名单冲突，exec_idle 代表选中序列是否已满；mem_idle 代表 mmbuf 缓冲器是否已满；rf_idle 代表 mprf 缓冲器是否已满。这四个条件得到一个变量 hit，另外这条指令是否和灰名单寄存器冲突成为另一个变量 preload。有了 hit 和 preload，可以得到本条指令的归属。

alu2exec 表示单周期指令进入选中序列，条件是没有和黑名单冲突；alu2sdbuf 表示单周期指令进入驻留序列，条件是发生黑名单冲突或灰名单冲突；其他两个 mem2exec 和 mem2sdbuf 是多周期指令的进入选中序列和驻留序列的表示。

有了这四种归属，可以对 Rs 和 Rd 黑名单进行更新。

- **alu2sdbuf**: 单周期指令进入了驻留序列。它的执行被挂起，它所有的操作对象需要保留，它对寄存器的写入也不能执行。
 - **Rd 黑名单**: 加入 Rs0、Rs1 和 Rd。
 - **Rs 黑名单**: 加入 Rd。
- **alu2exec**: 单周期指令进入了选中序列。它将取出它的 Rs0 和 Rs1，因此这两个寄存器没有保护的必要。
 - **Rd 黑名单**: 不增加。它允许后续指令写入它的 Rs0、Rs1 和 Rd。写入 Rs0、Rs1，是因为它已启用了，没有保护的必要。写入这同一 Rd 也是可以的。
 - **Rs 黑名单**: 加入 Rd。这条指令的 Rd 不能被后续指令当成 Rs，因为它的 Rd 的数据还在途，不能当成 mprf 模块读 Rs 的一部分。
- **mem2sdbuf**: 多周期指令进入了驻留序列。它的处理方法类似于 alu2sdbuf，两者都是挂起，没有区别。
 - **Rd 黑名单**: 加入 Rs0、Rs1 和 Rd。
 - **Rs 黑名单**: 加入 Rd。
- **mem2exec**: 多周期指令进入了选中序列。多周期指令和单周期指令同处于选中序列时不一样，因为前者的 Rd 是不能立即由 ALU 模块给出结果。
 - **Rd 黑名单**: 加入 Rd。
 - **Rs 黑名单**: 加入 Rd。

逐步递增式的黑名单是在经过指令的过程中，由位于前列的指令给后面的指令增加它对于 Rs 和 Rd 的禁止。因此，在这个指令进行黑名单检查时，是兼顾到它前列指令的所有要求

的，如果它的 **Rs** 和 **Rd** 不在黑名单上，那么说明它的执行是不打扰前列指令的执行的，是能够独立执行的。使用这种方法得到的选中序列是当前一系列候选指令认同的选择，是不会侵扰其他驻留序列的“利益”的。

我们知道有三种依赖：**RAW**（写后读依赖）、**WAR**（读后写依赖）、**WAW**（写后写依赖）。这种黑名单的方式可以解决指令与指令之间的各种依赖关系。任何一条前面的指令对于后面的指令的冲突都可以由前面的指令来申请 **Rs** 的禁止和 **Rd** 的禁止来避免，这就好比 **RAW**、**WAR**、**WAW** 的前面指令对于黑名单进行增加，那么对于有影响的指令会自动筛选下来。剩下来的都是属于无干涉的指令，可以进入选中序列。

RAW 依赖，也就是前列指令的写对于后面指令的读造成的数据依赖，用黑名单的表达方法是，前列指令的 **Rd** 加入 **Rs** 黑名单。

WAR 依赖，则是前列指令的读对于后面指令的写造成的数据依赖。用黑名单的表达方法是前列指令的 **Rs** 加入 **Rd** 黑名单。

WAW 依赖，是前列指令的写对于后面指令的写造成的数据依赖。用黑名单的表达方法是前列指令的 **Rd** 加入 **Rd** 黑名单。

使用黑名单可以让多个指令的 **RAW**、**WAR** 和 **WAW** 收集起来，量化成一个列表。一条指令是否与前面的所有指令发生这三种数据依赖，只要查看这个列表，看看自己对应的寄存器在不在这个列表即可。

注意，经过黑名单检查成功进入选中序列后，它对于处于前列但在驻留序列的其他指令是没有任何数据依赖的，原因就在于这个进入选中序列的后面指令是经过了黑名单检查的。那么可以这么说，只要位于 **mprf** 缓冲器和 **membuf** 缓冲器的指令对于留在 **schedule** 缓冲器的指令来说，都是“合法”允许在它之前执行的。这也是可以使用 **membuf** 缓冲器形成初始黑名单的基础。

使用累进黑名单的方式有一个缺点，那就是关键路径太长了。可以回顾一下累进黑名单的形成方式。首先是 **membuf** 缓冲器提供两个初始黑名单，然后每经过一条位于 **schedule** 缓冲器的候选指令，均要等到该条指令确定进入哪一个序列后，才能决定怎么增加两个黑名单。也就是确定一条指令的状态，增加一次黑名单。那么这条关键路径是候选指令条数乘

上添加黑名单的逻辑延时。如果能够减少添加黑名单的逻辑延时，那么总的关键路径就可以大大减少。

在 schedule 模块中，采用了另外一种方法来降低关键路径。在 schedule.v 代码中找到 TIMING_STYLE 定义，打开这个定义可以显著降低关键路径，那么在同样的时延下，可以增加 schedule 缓冲器的容量。这个方法的实现代码如下：

```
`ifdef TIMING_STYLE
    wire hit_common = check_blacklist(para,check_rdlist,check_rdlist);

    wire hit_hard, hit_soft;
    if (i==0) begin:gen_hit_zero
        assign hit_hard = 0;
        assign hit_soft = 0;
    end else begin:gen_hit_other
        wire `N(i) bits_hard, bits_soft;

        for (j=0;j<i;j=j+1) begin:gen_relation
            assign bits_hard[j] = go_vld[j] &
check_hard(go_para[`IDX(j,`EXEC_PARA_LEN)],go_para[`IDX(i,`EXEC_PARA_LEN)]);
            assign bits_soft[j] = go_stay[j] &
check_soft(go_para[`IDX(j,`EXEC_PARA_LEN)],go_para[`IDX(i,`EXEC_PARA_LEN)]);
        end

        assign hit_hard = |bits_hard;
        assign hit_soft = |bits_soft;
    end

    wire hit_rg = hit_common|hit_hard|hit_soft;
`else
    wire hit_rg = check_blacklist(para,rd_checklist[i],rs_checklist[i]);
`endif
```

在检查黑名单时，候选指令会把这两个黑名单分成三部分：

1. membuf 缓冲器给出的 Rs 和 Rd 初始黑名单造成的冲突；
2. 候选指令和其他 schedule 缓冲器内，位于它前列指令的“硬”冲突。之所以是硬冲突，是指位于前列的指令不管是进入哪一个序列，都将决定候选指令不能进入选中序列。只要前列的指令存在，都会让候选指令不能选中
3. 候选指令和其他 schedule 缓冲器内，位于它前列指令的“软”冲突。之所以是软冲突，是指只有前列的指令进入驻留序列，这个冲突才有意义；如果前列指令进入选中序列，这个冲突就不存在。

和黑名单的方式相比，这种方法让候选指令和提供冲突的前列指令们直接进行“对话”。而黑名单的方式是采取了一个虚拟的媒介：黑名单。任何一条指令表达它的“利益”诉求的方式是提供它的小名单融合到大的黑名单上，然后指令依据黑名单的冲突情况来决定自己的走向。

黑名单方式的延时“生长”是这样的，一条指令根据黑名单决定走向，然后根据走向把它的小名单添加到总的黑名单上。这个循环的重复次数是 schedule 缓冲器的大小。直接“对话”的延时生长是这样的：每一条候选指令的第一和第二部分都可以同时生长，唯一需要进入循环的是第三部分。一条指令知道它的走向后，可以决定后续指令有关它的第三部分，一直到最后一条指令为止。

总的来说，关键路径都是：schedule 缓冲器的大小乘上某个延时。黑名单的延时会包括对话法的第二和第三部分；但对话法只有第三部分。

在理解对话的方式时，可能不太理解何为硬冲突、何为软冲突。下面是硬冲突的实现函数：

```
function check_hard( input `N(`EXEC_PARA_LEN) a_para,b_para );
    reg a_mem,a_alu,b_mem,b_alu;
    reg `N(`RGBIT) a_rs0,a_rs1,a_rd,b_rs0,b_rs1,b_rd;
    begin
        a_mem      = (a_para>>((3*`RGBIT)+1));
        a_alu      = (a_para>>(3*`RGBIT));
        a_rs0      = a_para;
        a_rs1      = a_para>>`RGBIT;
        a_rd       = a_para>>(2*`RGBIT);
        b_mem      = (b_para>>((3*`RGBIT)+1));
        b_alu      = (b_para>>(3*`RGBIT));
        b_rs0      = b_para;
        b_rs1      = b_para>>`RGBIT;
        b_rd       = b_para>>(2*`RGBIT);

        check_hard = (a_rd!=0)&(((a_rd==b_rs0)|(a_rd==b_rs1))|(a_mem&(a_rd==b_rd)));
    end
endfunction
```

硬冲突可以理解为：前面的这条指令进入选中序列对于后续指令造成的冲突。意思也就等于它无论是进入选中序列，还是进入驻留序列，都将决定后面指令不能进入选中序列。说白了，一条指令进入选中序列，它对于后续的冲突只有硬冲突；一条指令进入驻留序列，它对于后续指令的冲突是硬冲突和软冲突之和。

因此，硬冲突的焦点是前列指令的 Rd。只要前列指令的 Rd 不等于 0，也就是只要存在这个 Rd，都会导致后续指令有一个不可越过的“黑”寄存器，它的 Rd。

```
function check_soft( input `N(`EXEC_PARA_LEN) a_para,b_para );
    reg a_mem,a_alu,b_mem,b_alu;
    reg `N(`RGBIT) a_rs0,a_rs1,a_rd,b_rs0,b_rs1,b_rd;
    begin
        a_mem      = (a_para>>((3*`RGBIT)+1));
        a_alu      = (a_para>>(3*`RGBIT));
        a_rs0      = a_para;
        a_rs1      = a_para>>`RGBIT;
        a_rd       = a_para>>(2*`RGBIT);
        b_mem      = (b_para>>((3*`RGBIT)+1));
        b_alu      = (b_para>>(3*`RGBIT));
        b_rs0      = b_para;
        b_rs1      = b_para>>`RGBIT;
        b_rd       = b_para>>(2*`RGBIT);

        check_soft = (b_rd!=0)&((a_rs0==b_rd)|(a_rs1==b_rd)|(a_alu&(a_rd==b_rd)));
    end
endfunction
```

软冲突的焦点是后续指令的 Rd。也就是因为前列指令进入驻留序列，那么前列指令的相关寄存器需要保护。

除了组建选中序列和驻留序列外，schedule 缓冲器还需要对指令进行清除，这段代码是：

```
wire clear = (alu & (rd==0))|( clear_pipeline &
~((orderx==0)&(level==0)) )|( level_clear & (level!=0) );
```

在上一级 instrbits 缓冲器的讲述中，已经提到 level_clear 信号，可以对经过预测进入缓冲器的指令进行清除。也就是指令的 level 参数，如果它等于 0，那么它属于确定执行的执行，如果它不等于 0，那么它是因为预测而提前进入的指令。level_clear 信号表示预测错误，那么因为预测而提前进入的信号需要清除。level_clear 和 level!=0 共同决定了本条命令必须清除出去。它通过置位 clear 信号，clear 信号又会清除这条指令的有效性。那么在下一个周期这条指令属于无效指令，它的 go_vld[i] 会等于 0，alu2sdbuf/alu2exec/mem2sdbuf/mem2exec 这四个关键信号都为 0，也就不会进入选中序列和驻留序列。

(alu & (rd==0))表示清除无用的 NOP 指令。(clear_pipeline & ~((orderx==0)&(level==0)))表示指令发生异常或中断，清除流水线内的信号。

总体来说，只要具有这些操作要素，即可形成两个序列：选中序列和驻留序列。选中序列会进入寄存器，在下一个周期送入 ALU 模块；驻留序列送入 schedule 缓冲器，在下一个周期，继续进行这个“筛选”过程。

这两个序列都包含一个“preload”类型的指令。也就是说这种类型的指令，需要由 mem_release 是否等于 0 来决定哪一个序列的为真。mem_release 不等于 0，表示当时认为的灰名单的指令真实应该是白名单指令，它的 Rd 是可以提供的，于是选中序列的为真。如果 mem_release 等于 0，那么这个灰名单真实应该是黑名单状态，它的 Rd 是不能获取的，于是驻留序列为真。

第三级缓冲器：mprf

schedule 模块在得到选中序列后，统一经过各自的 ALU 模块后，其中单周期指令会直接得出 Rd 和它的对应数据。这些 Rd 和对应数据会送入 mprf 缓冲器，由它暂存，并随着 membuf 缓冲器内的指令退休，逐步写入寄存器组。

寄存器组位于 mprf 模块，因此它也会有其他模块对寄存器组写入的接口。打开 mprf.v，前面的接口正是刚才讨论的：

input	`N(`EXEC_LEN*`RGBIT)	rd_sel,
input	`N(`EXEC_LEN*`MMCMB_OFF)	rd_order,
input	`N(`EXEC_LEN*`JCBUF_OFF)	rd_level,
input	`N(`EXEC_LEN*`XLEN)	rd_data,
input		csr_vld,
input	`N(`RGBIT)	csr_rd_sel,
input	`N(`XLEN)	csr_data,
input	`N(`MEM_LEN*`RGBIT)	mem_sel,
input	`N(`MEM_LEN*`XLEN)	mem_data,
input	`N(`MEM_OFF)	mem_release,

其中 rd_*等信号是由 ALU 模块送入。rd_sel 代表对应写入的 Rd 的寄存器号，由于其中可能有多周期指令，因此 rd_sel 等于 0 代表不是有效的单周期 OP 指令，需要从中滤除，才能写入缓冲器中。

csr_*信号是 CSR 指令写入寄存器组的信号。CSR 指令在执行时，后面的三大缓冲器均已经清空，因此这一系列信号不会和其他操作冲突。

mem_sel 和 mem_data 是 membuf 缓冲器对寄存器组的写入。membuf 缓冲器可以一次退休多条多周期指令。mem_release 代表本次周期退休的指令条数，它是用来对 order 属性进行归零减一的信号。

本模块除了对寄存器组的写入外，一个重要的功能是对其他指令提供 Rs0 和 Rs1。下面的接口是提供这个功能的：

input	`N(`EXEC_LEN*`RGBIT)	rs0_sel,
input	`N(`EXEC_LEN*`RGBIT)	rs1_sel,
output	`N(`EXEC_LEN*`XLEN)	rs0_word,
output	`N(`EXEC_LEN*`XLEN)	rs1_word,
input	`N(`RGBIT)	extra_rs0_sel,
input	`N(`RGBIT)	extra_rs1_sel,
output	`N(`XLEN)	extra_rs0_word,
output	`N(`XLEN)	extra_rs1_word,

其中第一部分是由 ALU 模块发起的读 Rs0 和 Rs1 信号，rs0_sel 和 rs1_sel 代表需要读取的 Rs0 和 Rs1 对应的寄存器号，mprf 模块通过 rs0_word 和 rs1_word 给出对应的寄存器。在输出 Rs0 和 Rs1 时，它不仅从寄存器组，而且还从 mprf 缓冲器和 mem_sel/mem_data 这两个途径给出相应的值。也就是说，只要是对寄存器组的写入，虽然是“在途”，也就是正在发生的写入，也被 mprf 模块视为已经写入，对 Rs 的读取给出最新值。

这是通过一个函数 get_from_array 来给出：

```
function `N(1+`XLEN) get_from_array(input `N(`RGBIT)
target_sel,
                                input `N((`RFBUF_LEN+`MEM_LEN)*`RGBIT)
array_sel,
                                input `N((`RFBUF_LEN+`MEM_LEN)*`XLEN)
array_data);
    integer            i;
    reg `N(`RGBIT)    sel;
    reg `N(`XLEN)     data;
    reg               get;
    reg `N(`XLEN)     out_word;
begin
    get                = 0;
    out_word           = 0;
    for (i=0;i<(`RFBUF_LEN+`MEM_LEN);i=i+1) begin
        sel            = array_sel>>(i*`RGBIT);
        data           = array_data>>(i*`XLEN);
        get            = get|(target_sel==sel);
        out_word       = (target_sel==sel) ? data : out_word;
    end
    get                = (target_sel==0) ? 1'b0 : get;
    get_from_array    = { get,out_word };
```

```
end  
endfunction
```

其中 `array_sel` 和 `array_data` 代表将要写入寄存器组的一系列 `Rd` 和它对应的数据。如果给出寄存器号 `target_sel`，那么在这个写入序列中，查找是否有对应寄存器的写入，如果有，那么这个值代表最新的写入，则提取出来作为读出的最新值。

另外一个读出 `Rs0` 和 `Rs1`，则来自于第一级缓冲器 `instrbits`。这一级缓冲器读出 `Rs0` 和 `Rs1`，主要用作 `JCOND` 缓冲器对 `JCOND` 指令进行跳转判决，另外一个目的是 `CSR` 指令和 `JALR` 指令读出 `Rs0`。如果要用作 `JCOND` 指令，那么它隐含的一个条件是 `level` 属性等于 0，因为 `level` 属性不等于 0 的指令是在 `JCOND` 指令之后发生的，不能用作它的 `Rs0` 和 `Rs1`。因此采用另外一个相似的函数 `get_from_array_level` 来进行读出。它和 `get_from_array` 的区别是会增加 `level` 属性的滤除，只有 `level` 属性等于 0 的写入会作为 `Rs` 的输出值。

`mprf` 缓冲器的输入是来源于 `ALU` 模块的输出。`ALU` 模块输出 `Rd` 和数据并不能直接进入寄存器组，而是需要先进入缓冲器，再由缓冲器输送到寄存器组。这样做的目的是 `ALU` 模块毕竟是对 `Rs0` 和 `Rs1` 进行逻辑或算术运算，相比较于其他实现，它的 `Rs` 来源是由广义的寄存器组取出，时延可能更长，那么让这些写入经由缓冲器缓冲一拍，可以起到降低时延的作用。

本模块首先是对 `ALU` 模块输出序列的整理，它包括下面的操作：

1. 对于 `rd_level` 中的 `level` 属性进行调整，也就是应对 `level_decrease` 对所有指令的 `level` 属性归零式减一的操作；
2. 对于 `rd_order` 中的 `order` 属性进行调整，也就是对 `mem_release` 信号进行响应，对 `order` 属性做出归零式减一的操作；
3. 对它们做出清零操作。主要是响应 `level_clear` 信号或其他中断、异常等清零操作。

经过上述整理的过程，然后再对 `rd_sel` 和 `rd_data` 清理掉 `rd_sel` 等于 0 的个体。这一新的序列会附在 `mprf` 缓冲器的“驻留序列”之后。

mprf 缓冲器也会如同 schedule 缓冲器一样，分裂为两个序列，一个序列也可以称为选中序列，另外一个序列称为驻留序列。

在分裂的过程中，上面对输入序列所做的三件操作，mprf 缓冲器的所有个体也会做一次。分裂的目的是形成一个可以写入寄存器组的序列。这个个数是可以自定义的，也就是在 mprf 缓冲器选取几个 Rd 和数据，可以让这批 Rd 和数据进入寄存器组，完成这几个单周期指令的使命，进行退休。

这个选取的标准是 order 属性等于 0、level 属性等于 0。也就是只要这些 Rd 和数据待到了这两个属性为 0，那么它就获准写入寄存器组。在这时，它们写入寄存器组已经没有任何风险，作为缓冲器的目的是让这批 Rd 和数据待到这个时候。期间会有被清除的风险，也就是 level 属性不等于 0，会有 level_clear 也就是 JCOND 指令预测错误时的风险；order 属性不等于 0，也就是发生异常或中断进行清除的风险。

最后以 order 属性等于 0 和 level 属性等于 0 而留下的 Rd 和数据，是可以安全写入寄存器组，从而为缓冲器腾退空间。

最后，驻留序列会和输入而来的 rd_sel 与 rd_data 形成输入序列汇合，形成 mprf 缓冲器新的成员。而选中序列则会对寄存器组写入，同时对寄存器组写入的还包括 mem_sel、mem_data 这两个信号，它们是 membuf 缓冲器退休它们的指令发出的寄存器写入信号。

在这其中，会形成一个寄存器列表 rdbuf_order_list，它是对 mprf 缓冲器内的所有 Rd 的一个合集。它的作用是防止和 membuf 缓冲器之间的冲突。

一条多周期指令不允许后面再有使用同一 Rd 的指令离开 schedule 缓冲器。如果 schedule 缓冲器内有一条指令的 Rd 等于 membuf 缓冲器的任一成员的 Rd，那么根据 Rd 黑名单，它只能在 schedule 缓冲器内等候这一成员退休为止。mprf 缓冲器内各个成员的 Rd，是可能和 membuf 缓冲器内的 Rd 相同。如果 mprf 缓冲器内的某个 Rd 等于 membuf 缓冲器的 Rd，那么 mprf 缓冲器内的这一条指令一定是发生在 membuf 缓冲器的这一条同一 Rd 指令之前的。原因很简单，如果反之，是不成立的，因为多周期指令有天然排除后续同一 Rd 离开 schedule 缓冲器的权利。

举个例子，在 schedule 缓冲器内，有一条多周期指令和单周期指令是同一 Rd。如果是单周期指令在前，多周期指令在后，那么它们是能进入同一选中序列的。那么就形成了 mprf 缓冲器和 membuf 缓冲器存在着同一 Rd 的情况，此时必须让 membuf 缓冲器等待 mprf 缓冲器

内这一个同一 Rd 的指令先进入寄存器组才行，因为 mprf 缓冲器的那条指令在前。如果是多周期指令在前，单周期指令在后，那么只能是多周期指令进入选中序列，单周期指令是不可能进入的。不仅如此，只要是这一多周期指令还在 membuf 缓冲器内，它都不能进入选中序列，只能在 schedule 缓冲器内驻留。

因此，mprf 缓冲器形成一个 Rd 集合列表给 membuf 缓冲器参考，如果 membuf 缓冲器在退休某一条指令时，发现它的 Rd 恰恰出现在这个列表中，那么表示在它之前的某条同一 Rd 的单周期指令没有完成。它必须等到这一条同一 Rd 的单周期指令完成后，它才能对这一同一 Rd 进行写入。

对于 membuf 缓冲器来说，它要退休它的某一条指令，前提条件就是 mprf 缓冲器提供的 Rd 列表不能包含它对应的 Rd。它只有等到 mprf 缓冲器清除了这一同一 Rd 的指令后，它才能对寄存器组写入。

这是 membuf 缓冲器和 mprf 缓冲器发生冲突的唯一可能：mprf 缓冲器有领先于 membuf 缓冲器内同一 Rd 的操作。这个反之是不成立的。

可能有的会问，mprf 缓冲器这一同一 Rd 的指令会不会不退出，导致 membuf 缓冲器也不能退出，形成死锁？这是不会的，因为根据规则，mprf 缓冲器的这一指令是在前列，是发生在 membuf 缓冲器的临界退休的指令之前的，那么可以说此时 mprf 缓冲器的 order 属性一定等于 0，它一定会退休。只不过因为出口有限，它只是在排队。等到它写入寄存器组时，这个列表会发生变化，membuf 缓冲器也就会随之退休。

在寄存器组 rbank 的描述中可以看到，ch_sel 对于寄存器组的写入是发生在 away_sel 写入之后的。ch_sel 是 membuf 缓冲器对于寄存器组的写入，它附带了 CSR 指令对于寄存器组的写入；away_sel 是 mprf 缓冲器对寄存器组的写入。也就说 membuf 缓冲器的写入权限要高。

第三级缓冲器：membuf

正如 mprf 缓冲器是针对单周期指令，membuf 缓冲器是针对多周期指令。多周期指令以两个实际操作数的形式送达 membuf 缓冲器，它们以绝对顺序的形式先后抵达。membuf 缓冲器也会按照到来的先后顺序，安排实体 LSU 模块和 MUL 模块对接相应的指令。最底部的指令会不断查询安排的实体模块的状态，当实体模块汇报成功结束，那么这条最底部的指令

会退休，离开 membuf 缓冲器。这就是 membuf 缓冲器接纳多周期指令和退休它们的全过程。

下面是 membuf.v 文件里面重要的接口列表：

```
//interface with mul
output `N(`MUL_LEN)                mul_initial,
output `N(`MUL_LEN*3)              mul_para,
output `N(`MUL_LEN*`XLEN)          mul_rs0,
output `N(`MUL_LEN*`XLEN)          mul_rs1,
input  `N(`MUL_LEN)                mul_ready,
input  `N(`MUL_LEN)                mul_finished,
input  `N(`MUL_LEN*`XLEN)          mul_data,
output `N(`MUL_LEN)                mul_ack,

//interface with lsu
output                lsu_initial,
output `N(`MMBUF_PARA_LEN) lsu_para,
output `N(`XLEN)       lsu_addr,
output `N(`XLEN)       lsu_wdata,
input                lsu_ready,
input                lsu_finished,
input                lsu_status,
input `N(`XLEN)       lsu_rdata,
output                lsu_ack,

//interface with mprf
input  `N(`EXEC_LEN)          mem_vld,
input  `N(`EXEC_LEN*`MMBUF_PARA_LEN) mem_para,
input  `N(`EXEC_LEN*`XLEN)     mem_addr,
input  `N(`EXEC_LEN*`XLEN)     mem_wdata,
input  `N(`EXEC_LEN*`XLEN)     mem_pc,
input  `N(`EXEC_LEN*`JCBUF_OFF) mem_level,
output `N(`MEM_LEN*`RGBIT)     mem_sel,
output `N(`MEM_LEN*`XLEN)     mem_data,
output `N(`MEM_OFF)           mem_release,
```

其中前面两项是连接处理多周期指令的 MUL 模块和 LSU 模块的接口。MUL 模块的个数是可定制的，只需要为 MUL 模块输送两个操作数和它的乘除方式，那么在计算完毕后，结果存储在它自带的缓存器内，可以在该指令退休时由 membuf 缓冲器取出，送至寄存器组写入 Rd。

LSU 模块只有一个，它的接口方式和 MUL 模块类似。只需要送达 addr、wdata 和操作类型等参数，那么 LSU 模块就会对数据存储器进行对应的操作。操作结果会存储在自带的缓存器内，在该指令即将退休时，由 membuf 缓冲器取出，如果是读存储器指令，会送入寄存器组进行写入 Rd；如果是写存储器指令，在成功时会自动退休。

多周期指令通过 ALU 模块后，进入 membuf 缓冲器时，有如下域：

- mem_vld: 该指令是否有效，因为通过 ALU 模块是单周期和多周期指令不定，因此 mem_vld 作为多周期指令的指示，也是 0 或 1 位置不定，需要通过移位来去除 0 位的“泡沫”。
- mem_para: 多周期指令的参数域，包括：
 - [9]: 标志本指令是 LSU 还是 MUL 指令，1 表示 MUL 指令，0 表示 LSU 指令；
 - [8:4]: Rd。如果是 MUL 指令，表示乘除法结果送达的寄存器；如果是 LSU 指令，表示加载的数据送达的寄存器，如果是写数据存储器指令，它等于 0。
 - [3]: 读写数据存储器标志。MUL 指令等于 0；LSU 指令用来表示读还是写，1 表示写，0 表示读。
 - [2:0]: 操作指示。如果是乘除法指令，给出是乘法还是除法，以及乘除法的处理方式；如果是 LSU 指令，给出读写的数据宽度和是否有符号数。
- mem_addr: LSU 指令表示地址，MUL 指令表示一个乘除数。
- mem_wdata: LSU 指令表示写数据，MUL 指令表示另外一个乘除数。
- mem_pc: 该多周期指令对应的 PC。
- mem_level: 该多周期指令对应的 level 属性。

多周期指令通过这样的形式送来后，需要挤掉单周期指令带来的泡沫，然后附在 membuf 缓冲器的后面。注意，需要响应对应的清除信号。如果是 level_clear 信号，也就是清除 level 不等于 0 的指令，因为 membuf 缓冲器是严格顺序的指令，这种清除相对于其他缓冲器更为容易。

现在问题的焦点是如何在 MUL 模块和 MUL 指令之间配对，或 LSU 模块与 LSU 指令之间配对。现在假定 membuf 缓冲器容纳指令的条数为 MMBUF_LEN，但实际大小是未知的，每个周期都有新的指令从 ALU 模块进来，同时有旧的指令退休。因此，每一周期都在变化

的，在这变动中，为 N 个 MUL 模块分配不定的 MUL 指令，为一个 LSU 模块也分配相应的 LSU 指令。

首先是 SSRV 关于如何寻找一个 MUL 指令并为它分配一个 MUL 模块。为了寻找并定位一条 MUL 指令，引进了三个变量。一个是组合逻辑变量 `mul_next_order`，另外两个是寄存器变量 `mul_bottom` 和 `mul_order`。

在初始时，`mul_bottom` 等于 0，`mul_order` 等于 `MMBUF_LEN`。`mul_bottom` 等于 0，表示 `mul_next_order` 从 `mul_bottom` 也就是 0 开始寻找下一条 MUL 指令，这里包含 0。`mul_order` 等于 `MMBUF_LEN` 表示并没有寻到真正的 MUL 指令，因为 `MMBUF_LEN` 是容量的最大序号。

一开始 `membuf` 缓冲器并没有进来 MUL 指令，那么 `mul_next_order` 根据 `mul_bottom` 给出的最小边界寻找不到 MUL 指令，那么它等于 `MMBUF_LEN`。在这种情况下 `mul_bottom` 和 `mul_order` 保持不变。

忽然，`membuf` 缓冲器进来了一条 MUL 指令，`mul_next_order` 检查到这条 MUL 指令的序号为 N。因为这个序号小于 `MMBUF_LEN`，是有效的序号，于是 `mul_bottom` 等于 N+1，`mul_order` 等于 N。`mul_bottom` 等于 N+1，表示接收 N 为 MUL 指令，让 `mul_next_order` 去探测比 N 更大的另外一条 MUL 指令；`mul_order` 等于 N，表示当前处理这 N 号的 MUL 指令。

于是根据 `mul_order` 等于 N，对 `membuf` 缓冲器移位 N 个序号，得到了这一条 MUL 指令，和多个 MUL 模块中的一个空闲的去配对。如果没有匹配成功，那么 `mul_order` 和 `mul_bottom` 保持不变，因为这第 N 号 MUL 指令并没有处理，仍需保持这个序号。注意，这里的保持不变是相对的，因为有指令退休，还是会减去相应的位数。

如果匹配成功，也就是 `mul_hit_vld` 为高，那么表示第 N 号已经得到处理，可以处理下一条，于是把代表下一条的 `mul_next_order` 引入。如果 `mul_next_order` 等于 `MMBUF_LEN`，那么表示没有下一条，于是 `mul_order` 变成 `MMBUF_LEN`，也就表示没有新的 MUL 指令，但 `mul_bottom` 保持不变，也就是还是要从第 N 号开始寻找下一条。如果长时间没有 MUL 指令出现，`mul_bottom` 会因为指令的退休，而重新归零。

因此会不断重复这个过程，一旦找到了一条 MUL 指令，即通过 `mul_bottom` 去锚定这条 MUL 指令，让 `mul_next_order` 从此寻找下一条，然后通过 `mul_order` 获得这条 MUL 指令的

序号，去和空闲的 MUL 模块去配对。如果配对成功，那么接收新的 mul_next_order，如果不成功，那么这个序号保持不变，等到有空闲的 MUL 模块去配对。

注意，mul_next_order 找到的 MUL 指令一定是 level 属性为 0 的 MUL 指令。也就是说，因为预测而进来的指令只有“等候”权，它既不能退休，也不能调用实体模块。

这段实现代码如下：

```
wire `N(`MMBUF_OFF)          mul_next_order = chain_mul_next[`MMBUF_LEN];

`FFx(mul_bottom,0)
if ( clear_pipeline )
    mul_bottom <= 0;
else if ( ((mul_order==`MMBUF_LEN)|mul_hit_vld) & (mul_next_order!=`MMBUF_LEN) )
    mul_bottom <= mul_next_order + 1'b1 - mem_release;
else
    mul_bottom <= (mul_bottom < mem_release) ? 0 : (mul_bottom - mem_release);

`FFx(mul_order,`MMBUF_LEN)
if ( clear_pipeline )
    mul_order <= `MMBUF_LEN;
else if ( (mul_order==`MMBUF_LEN)|mul_hit_vld )
    mul_order <= (mul_next_order==`MMBUF_LEN) ? `MMBUF_LEN : (mul_next_order -
mem_release);
else
    mul_order <= mul_order - mem_release;
```

以 mul_bottom 和 mul_order 来共同锚定当前的 MUL 指令，以 mul_next_order 来寻找下一条 MUL 指令。如果当前的 mul_order 序号的 MUL 指令没有处理完毕，是不会接收 mul_next_order 指定的下一个序号的 MUL 指令的，但每次寻找的仍是同一个下一序号的 MUL 指令，这是保持 mul_bottom 不变做到的。

总结一下，这一段代码主动寻找 MUL 指令，一旦找到后，锁定它，然后寻找空闲的实体 MUL 计算模块进行匹配。因此，MUL 指令不管是如何分布在 membuf 缓冲器的什么位置，这段代码总能锁定一个，去寻找匹配。

LSU 指令则不一样。LSU 指令在 membuf 缓冲器内是主流派，大部分都是这一类型的指令。寻找 LSU 指令，去匹配唯一的 LSU 模块的方法也就不同。

```
wire lsu_accept = lsu_initial & lsu_ready;
wire lsu_inc = lsu_accept|( array_mul_flag>>lsu_order );
wire `N(`MMBUF_OFF) lsu_order_in = lsu_order + lsu_inc;
```

```

`FFx(lsu_order,0)
lsu_order <= clear_pipeline ? 0 : ( ( lsu_order_in < mem_release ) ? 0 :
( lsu_order_in - mem_release ) );

```

针对主流派的寻找方法更加简单。上面的 `lsu_order` 从 0 开始，在 0 位置可能有指令或没有指令。如果没有指令，`lsu_order` 不会变化，那么它会一直等待 0 位置出现指令。

一旦 0 位置出现了指令，这条指令要么是 LSU 指令，要么是 MUL 指令。如果是 LSU 指令，那么 `lsu_accept` 表示这条 LSU 指令和实体 LSU 模块已经配对成功，在这种情况下 `lsu_order` 加一，会继续“试验”下一条第 1 位置的指令。如果是 MUL 指令，会直接跳转到下一条第 1 位置的指令。因此，`lsu_inc` 指示是否递进到下一条指令。

这种方法的好处，在没有出现指令或 LSU 指令没有匹配成功，都会等待下去，直到出现指令或匹配成功，转向下一条。如果是 MUL 指令，会直接跳过，因为 MUL 指令不该它处理。

当然，随着指令的退休，`mem_release` 则是指示多少条指令退出，那么这个 `lsu_order` 也会随之不断变化。

MUL 指令和 MUL 模块配对，LSU 指令和 LSU 模块配对，配对的结果都要上报。这个上报是退休的依据。只有上报成功，并且对应的实体模块汇报成功，该指令才能退休。

```

reg `N(`MMBUF_LEN)          mmbuf_done_vld;
reg `N(`MMBUF_LEN)          mmbuf_done_sel;
reg `N(`MMBUF_LEN*`MUL_OFF) mmbuf_done_mul;

`FFx(mmbuf_done_vld,0)
mmbuf_done_vld <= clear_pipeline ? 0 :
( ( mmbuf_done_vld|(mul_hit_vld<<mul_order)|(lsu_accept<<lsu_order) )>>mem_release );

`FFx(mmbuf_done_sel,0)
mmbuf_done_sel <= clear_pipeline ? 0 :
( ( mmbuf_done_sel|(mul_hit_vld<<mul_order) )>>mem_release );

`FFx(mmbuf_done_mul,0)
mmbuf_done_mul <= clear_pipeline ? 0 :
( ( mmbuf_done_mul|(mul_hit_pos<<(mul_order*`MUL_OFF)) )>>(mem_release*`MUL_OFF) );

```

在上面的信号中，mmbuf_done_vld 表示 mmbuf 缓冲器内对应的指令上报成功。

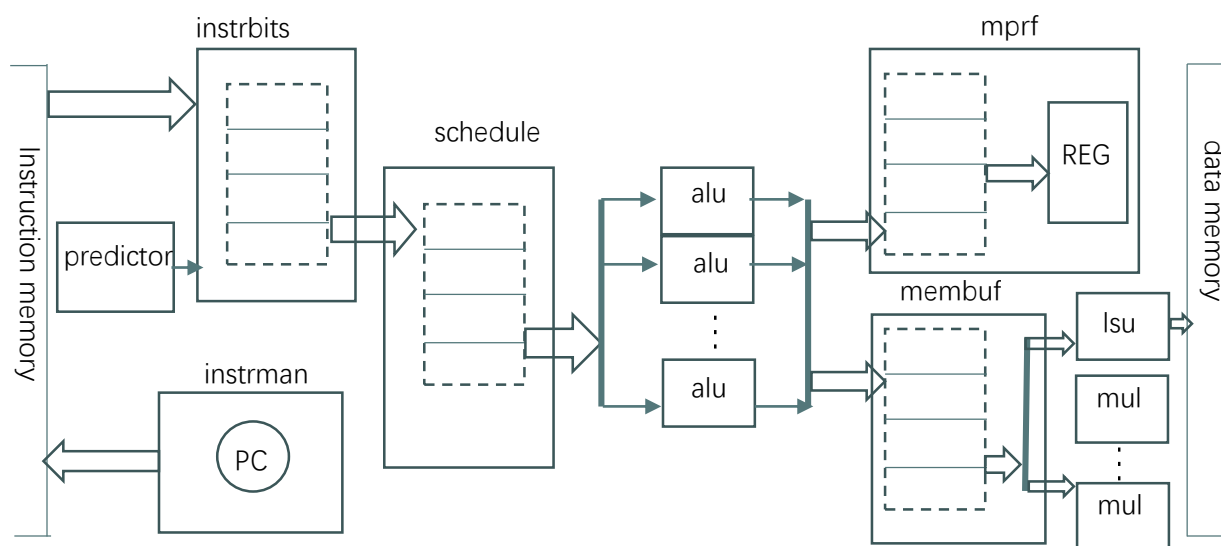
mmbuf_done_sel 表示则是一条 MUL 指令还是 LSU 指令，也就是指示是该查找 MUL 模块还是 LSU 模块。mmbuf_done_mul 表示该条 MUL 指令是分配到哪一个序号的 MUL 模块上。

一旦某条指令和对应的计算模块匹配成功，即通过这三个信号进行统计。后面在退休时，首先查询 mmbuf_done_vld 的最低位是否为 1，如果为 1，那么查询 mmbuf_done_sel，可以直到是 MUL 指令还是 LSU 指令，如果是 MUL 指令，通过 mmbuf_done_mul 可以直到应该到哪一个 MUL 模块上查找它的进程；如果是 LSU 指令，那么直接通过唯一的 LSU 模块来获知这条 LSU 指令的状态。

通过这一主线，可以理解一条多周期指令是如何进入 mmbuf 缓冲器，然后通过匹配算法，找到对应的实体计算模块，并进行登记。在退休部分，会根据登记检查，只要实体计算模块给出正确响应，那么这条指令可以进行退休。同一周期退休的条数可以定制，但受制于一条 LSU 模块和不定的 MUL 模块，这一定制的数目是很受限的。

四个缓冲器的关系

对于 RV32IMC 这一开放的指令集，SSRV 提出了这四个缓冲器来进行流水线传递。完全可以通过指定这四个缓冲器的大小和接口来管控指令进入流水线的深度，真正做到对乱序和多发射的定制。



现在来总结一下这四个缓冲器的关系。

instrbits 缓冲器：后面三个缓冲器是单周期指令和多周期指令的口袋，这一缓冲器的作用是尽可能调拨这两类指令进入这一口袋。它的职责是尽可能处理各种杂类指令，让这两类指令顺利通过。

schedule 缓冲器：对于它来说，它会源源不断的输出一个单周期和多周期指令夹杂的序列。它的问题是，是否可以认为单周期指令进入后一级后，视为已经完成；多周期指令进入下一级，可以视为除了它们的 **Rd** 不能用作后续指令的 **Rs** 和 **Rd** 外，也算已经完成。如果是，在这个限定条件下，它会不断输出这一指令序列，对后一级的两个缓冲器进行尽可能的填满操作。

同样，在这一缓冲器内，它内部的指令只要保证它相关的寄存器不受“干扰”，是完全可以允许后面的指令越过前面的指令进入选中序列的。这种乱序和多发射是一种挑选，简单来说，就是谁行谁上，不行的不用担心它的“利益”受损，因为这种越级是在遵守前列指令“利益”的越级。

mprf 缓冲器：这是寄存器组的一个扩大化。凡是想进入寄存器组的各种写入，都会汇总起来，对于外界来说，可以算是已经写入，但对于 **mprf** 模块内部，这些写入会有序的进入寄存器组。

membuf 缓冲器：多周期指令携带着两个操作数，在这个缓冲器内排队，等候分配对应的实体计算模块。一条多周期指令要退休，一定会经过分配成功，实体模块汇报计算成功这一过程。

如果想实现多个指令同时来执行，总会受到资源和时间的困扰。不论我们去火车站、机场、游乐园、商场，很多人都奔向同一个时间大小不一的资源，都会有资源和时间的干扰。于是人们做的最多的是一件事：排队。

是的，四个缓冲器提供的是排队的场所。排队的场所都会按照某一个规则，执行谁行谁上，不行排队等着的原则。指令也如同人一样，都想着尽可能接近退休的终点，但终归因为资源的限制，它得在某个缓冲器内等待。

缓冲器做的大一点，可以容纳更多的指令在某一个阶段等候，那么更多的指令会参与流水线。和传统的单条流水线相比，是一点一点的移动指令，而 **SSRV** 可以做到一批一批的进行移动指令。因此注意这四个缓冲器的接口不要有瓶颈，也就是理论上两个、三个或四个的一批批的通过。

可能有某些指令因为某种原因滞留在某个缓冲器，没关系，后面的能上则上，不能上也会按照规则进行排队。这就像我们的火车站和机场，并不是很理想的、很机械式的并排走向火车和飞机，总有人因为各种原因滞留，但火车站和机场设计的目的是包容这种滞留。同样，**SSRV** 也会包容指令因为各种依赖关系的“滞留”，因为这是如同火车站和机场一样，容纳人与人之间各种不同冲突的场所。

FPGA 篇

嵌入 SCR1 的 SSRV

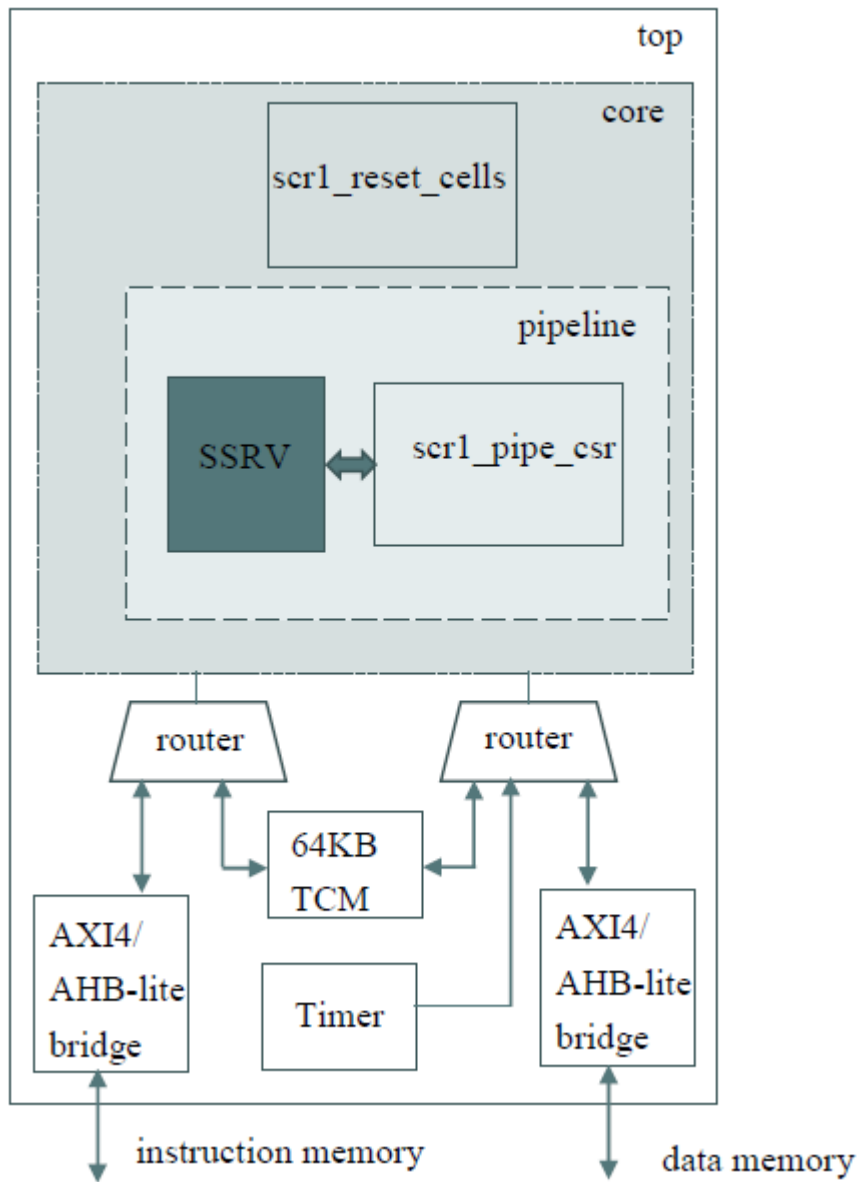
前面进行讲解的并不是一个完整意义的处理器内核，而是一个针对 RV32IMC 指令集的处理框架。SCR1 是一个非常优秀的处理器内核，它的参数众多，可定制化强，看的出来，设计团队是下了很大功夫。作为珠玉在前，SSRV 并没有能力去开发一个和它媲美的完整意义上的内核。

如果 SSRV 不在 FPGA 上运行，那么使用可综合的 Verilog 描述则失去了大半意义。这一篇是讲述如何在 FPGA 上搭建一个可以运行 SSRV 的项目。

最好的方法是借用 SCR1 的所有资源，但这对 SSRV 存在一个问题，那就是取指令的位宽。在 SCR1 的这套内核中，固定的指令宽度为 32 位，但要知道如果要想让 SSRV 跑到 CoreMark 分数到 5 以上，那么至少需要 64 bit 的取指宽度。但要突破这个限制，则必须舍去 SCR1 的架构和它的各种外设。但 SCR1 的架构和外设都是非常宝贵的，对于评估 SSRV 在实战中的性能影响，非常有意义。

经过综合考虑，还是接受 SCR1 对指令宽度的限制，因为结合或者叫做继承 SCR1 的各种资源是很有说服力的。因为这些资源和 SSRV 伴生在一起，更像是一个成熟的处理器内核，那么这次 FPGA 综合就显得更加真实。

下图是 SSRV 嵌入 SCR1 的实现框图。



SCR1 处理器内核分为 TOP、CORE 和 PIPELINE 共三个层级。其中内核处理是在 PIPELINE 这一层级。SSRV 和 SCR1 的 CSR 寄存器的实现模块 scr1_pipe_csr 共同替代 SCR1 的 PIPELINE 层。

关于中断、异常和 CSR 寄存器等处理，SSRV 是放在 sys_csr 模块中处理。这是一个为仿真而写的简单模块。在 FPGA 项目中，它仍然存在，不过只是一个对 scr1_pipe_csr 模块进行服务和转化的内部模块。

真正的各种中断、异常和 CSR 寄存器的实现，都在 scr1_pipe_csr 模块中。scr1_pipe_csr 是 SCR1 内核中实现这一功能的专用模块。这一模块需要的信号都会由 sys_csr 模块提供。

本篇涉及到的文件位于 `ssrv-on-scr1/` 目录下。这一目录包含两个文件夹，一个为 `sim/`，一个是 `fpga/`。前者是以仿真为目的，后者以 **FPGA** 执行为目的。

之所以仍建立一个仿真目录，是为了对 **FPGA** 执行进行调试。打开 `sim/` 这一文件夹，只有 `rtl/` 和 `scr1/` 两个目录。相比较于根目录下的仿真，少了 `testbench` 目录。这是因为 **SSRV** 已经嵌入 **SCR1** 内部，不再需要一个独立的 `testbench` 文件了。

现在打开 `rtl/` 目录，这就是 **SSRV** 改造后的 **PIPELINE** 层的替代品。

首先，顶层文件已经由 `ssrv_top.v` 替换为 `ssrv_pipe_top.sv`，但 `ssrv_top.v` 仍然存在，只是作为 `ssrv_pipe_top.sv` 的下一级顶层。打开 `ssrv_pipe_top.sv`，可以看到它的接口，已经完全替换成 `scr1_pipe_top.sv` 的接口：

```
module ssrv_pipe_top (
    // Common
    input  logic          pipe_rst_n,
    input  logic          clk,

    // Instruction Memory Interface
    output logic          imem_req,
    output type_scr1_mem_cmd_e imem_cmd,
    output logic [`SCR1_IMEM_AWIDTH-1:0] imem_addr,
    input  logic          imem_req_ack,
    input  logic [`SCR1_IMEM_DWIDTH-1:0] imem_rdata,
    input  type_scr1_mem_resp_e imem_resp,

    // Data Memory Interface
    output logic          dmem_req,
    output type_scr1_mem_cmd_e dmem_cmd,
    output type_scr1_mem_width_e dmem_width,
    output logic [`SCR1_DMEM_AWIDTH-1:0] dmem_addr,
    output logic [`SCR1_DMEM_DWIDTH-1:0] dmem_wdata,
    input  logic          dmem_req_ack,
    input  logic [`SCR1_DMEM_DWIDTH-1:0] dmem_rdata,
    input  type_scr1_mem_resp_e dmem_resp,

    // IRQ
    `ifdef SCR1_IPIC_EN
        input  logic [SCR1_IRQ_LINES_NUM-1:0] irq_lines,
    `else // SCR1_IPIC_EN
        input  logic          ext_irq,
    `endif // SCR1_IPIC_EN
        input  logic          soft_irq,

    // Memory-mapped external timer
    input  logic          timer_irq,
    input  logic [63:0] mtime_ext,
```

```

// Fuse
input  logic [`SCR1_XLEN-1:0]          fuse_mhartid

);

```

ssrv_pipe_top 模块包含两个模块，一个是 ssrv_top，一个是 scr1_pipe_csr。前者在前面仿真的基础上增加了很多接口，后者是 SCR1 内核自带的模块。现在打开 ssrv_top.v。

ssrv_top.v 在仿真的基础上增加了很多接口：

```

module ssrv_top(
    input                clk,
    input                rst,

    output               imem_req,
    output `N(`XLEN)    imem_addr,
    input  `N(`BUS_WID) imem_rdata,
    input               imem_resp,
    input               imem_err,

    output               dmem_req,
    output               dmem_cmd,
    output `N(2)         dmem_width,
    output `N(`XLEN)    dmem_addr,
    output `N(`XLEN)    dmem_wdata,
    input  `N(`XLEN)    dmem_rdata,
    input               dmem_resp,
    input               dmem_err,

    //interface between SCR1
    output               exu2csr_r_req,
    output `N(12)        exu2csr_rw_addr,
    input  `N(`XLEN)    csr2exu_r_data,
    output               exu2csr_w_req,
    output `N(2)         exu2csr_w_cmd,
    output `N(`XLEN)    exu2csr_w_data,
    input               csr2exu_rw_exc,

    input               csr2exu_irq,
    output               exu2csr_take_irq,

    output               exu2csr_mret_instr,
    output               exu2csr_mret_update,

    output               exu2csr_take_exc,
    output `N(4)         exu2csr_exc_code,
    output `N(`XLEN)    exu2csr_trap_val,

    input  `N(`XLEN)    csr2exu_new_pc,
    output `N(`XLEN)    curr_pc, //exc PC
    output `N(`XLEN)    next_pc  //IRQ PC

```

```
);
```

这些接口都会连接到它的子模块 `sys_csr`。也就是说除了顶层接口和 `sys_csr` 的例化，其他都和顶层仿真时一样的。

`ssrv-on-scr1` 的仿真和顶层仿真架构是相同的。现在进入：`ssrv-on-scr1/sim/scr1/sim`。敲击命令：`source compile.do; source sim.do; run -all;` 在 `transcript` 窗口，会出现一样的记录信息。

由于嵌入 `SCR1` 内，导致 `define_para.v` 里面的很多定义不能打开，其中包括：

1. ``define WIDE_INSTR_BUS`
2. ``define BENCHMARK_LOG`
3. ``define INSTR_MISALLIGNED`

另外，必须定义 ``define BUFFER0_IN_LEN` 1。注意这一定义必须是 1，因为取指接口固定为 32 位。

除此之外，其他可以进行配置。

FPGA 工程和运行

Altera 的 DE2 系列 FPGA 开发板一直是广受欢迎的教育开发平台，相信很多 FPGA 开发者都有这一系列的某个产品。这次进行 FPGA 下载选择的是 DE2-115，它的核心 FPGA 是 Cyclone IV EP4CE115，是 Cyclone IV FPGA 系列的最大器件。选用的开发工具是官方的 Quartus 软件，具体的版本号为：18.0.0 Build 614 04/24/2018 SJ Standard Edition。

这次 FPGA 执行用到的文件位于 `ssrv-on-scr1/fpga/` 目录下。其中 DE2-115/ 是主目录，是根据 DE2-115 自带的开发工具生成的主目录。打开 `DE2-115.v`，这是顶层文件。它包含下列元件：

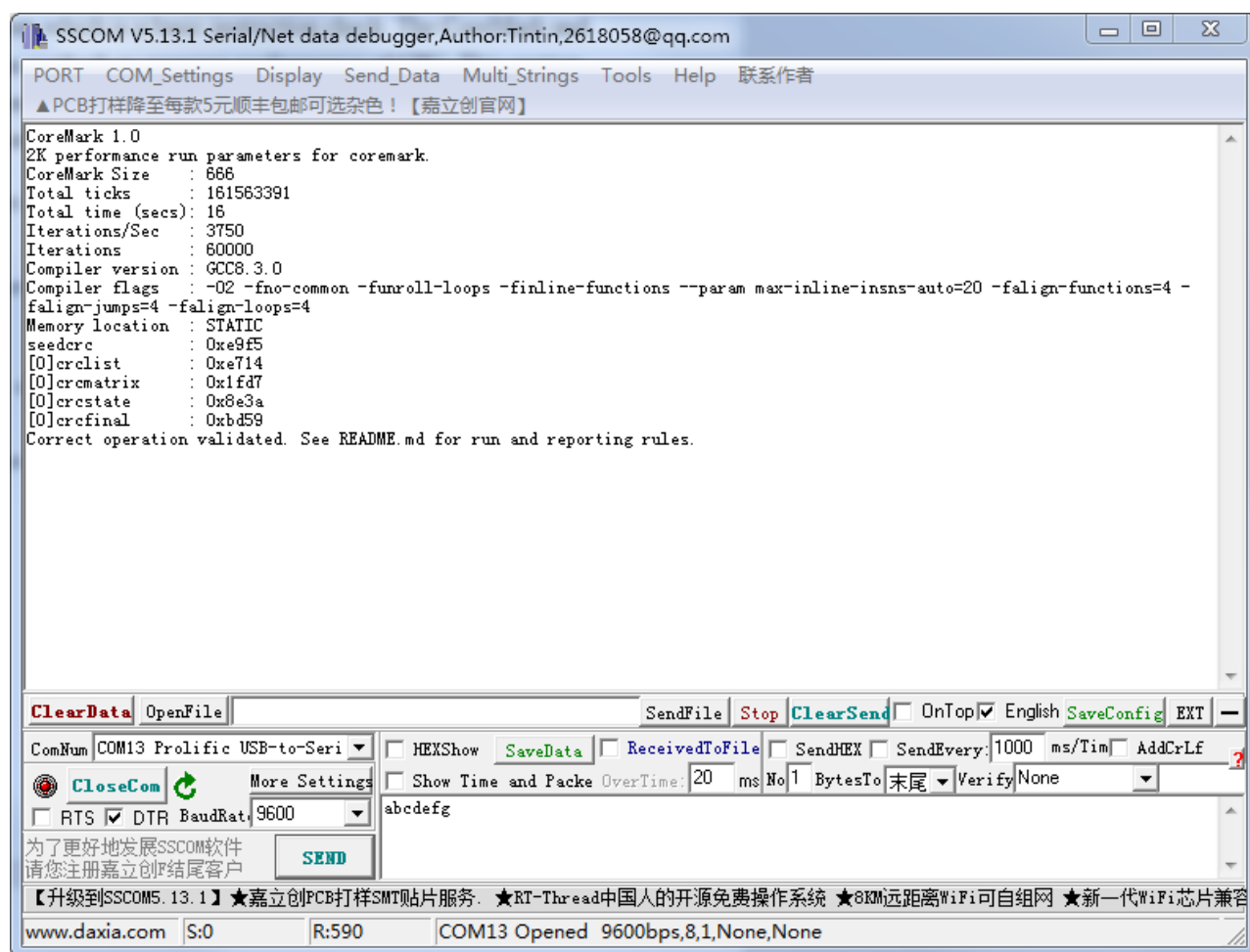
- 复位按键。开发板上自带的 KEY0 按键。
- 使用 MegaWizard Plug-in Manager 生成的 PLL。它的输入端口为板载的 50MHz，输出 c0 连接 30MHz 的 `clk`，c1 连接 0.3MHz 的 `rtc_clk`，均为 SCR1 core 需要的时钟。注意，

在仿真时输送的时钟为 100MHz 和 1MHz，但因为在 DE2-115 中无法运行到 100MHz，因此这里修改为 30MHz 和 0.3MHz。PLL 所用到的文件位于 pll/目录下。

- **scr1_top_ahb**: SCR1 的顶层。可以使用 rtl 目录下的 define_para.v 中的 USE_SSRV 来切换原版的 SCR1 或融合 SSRV 的 ssrv-on-scr1。如果使用了 SSRV，那么注意配置参数。SCR1 需要的文件都位于 scr1/目录下，如果要用到 SSRV，文件位于 rtl/目录下。
- **ssrv_memory**: 这是为 SCR1 core 提供指令存储和数据存储访问的模块。它位于 test/ssrv_memory.v。它承接 SCR1 core 的 AHB-lite 形式的指令存储和数据存储访问接口。在 ssrv_memory 内部，例化了一个 64KB 的双口 RAM，可以让指令接口和数据接口同时访问。这一双口 RAM 是由 MegaWizard Plug-in Manager 生成，位于 ram/目录下。双口 RAM 的初始文件是 ram/code.mif，这里面包含了 CoreMark 测试程序运行 60000 次的测试用例。
- **rxtx**: 这是一个 UART 串口服务模块。SCR1 通过对地址 32'hF0000000 写入某字节来实现通过串口发送字符串。rxtx 在传送字符串时，采用的配置方式是：9600 波特率、偶校验、带结束位。其中波特率可以在例化 rxtx 时进行配置，送入 clk 的时钟也需要指明。这里用的 clk 时钟是 30MHz，因此参数配置为 30。如果更改了 clk 的时钟，这个参数也需要做相应的改变。使用的硬件串口为半载的串口 1。

这是一个类似于仿真的 FPGA 执行案例。在上电后，SCR1 core 会从 ssrv_memory 模块的双口 RAM 中取出指令进行执行。SCR1 core 通过数据接口读写数据时，也会同样连接在这同一双口 RAM 内。

现在对这一 DE2-115 的项目进行综合、布局布线。如果参数选配合适，不超过 30MHz の設定，可以生成 DE2-115 开发板的下载文件。经过下载线进入 FPGA 中执行。大约经过 10 多分钟，在串口服务软件中出现下面的信息：



结果这一项：Iterations/Sec：3750，有些不太合乎逻辑。因为按照仿真算时钟频率 100MHz，计算出来的 CoreMark/MHz 为 $3750/100 = 37.5$ CoreMark/MHz。问题出在 SCR1 编译包里面的配置。打开 CoreMark 配置的 core_portme.c，里面有一个语句：

```
#define CLOCKS_PER_SEC 10000000
```

按照意思理解是每秒有多少个时钟。实际上 CoreMark 测试程序采集的是 100 个时钟周期一次的时钟，按照换算这里 Syntacore 公司定义的时钟频率是 1000MHz。那么如果用 3750 除以 1000，可以得出测试的性能为：3.75 CoreMark/MHz。

不知道 Syntacore 为什么定义时钟为 1000MHz，但既然在他们为 SCR1 的编译包里这样定义，一定有他们的理由，这里不去做过多的猜测。也因为时钟的关系，虽然记录信息里面用的时间是 16s，不过因为时钟参照物的不同，如果换算成 FPGA 运行的时间是：
 $16 \times 1000 / 30 = 533$ 秒，也就在 9 分钟左右。

CoreMark 测试分数的含义是 Iterations/Sec，也就是每秒跑了多少次测试循环。这个分数的倒数是 Sec/Iteration，也就是每次测试循环用了多少秒。在仿真时，是不可能跑出 10s，以便打出 Iterations/Sec 的测试分数的。那么使用 ticks/Iteration，也就是每次循环使用了多少个时钟周期，可以估算出大致的测试分数。因为时钟频率是 100 MHz，那么 tick/Iteration 大致等于 Sec/Iteration。

针对这个例子，可以看出跑了 60000 个 Iteration，用去了 161563391 个 ticks。那么每个 Iteration 用去了 2693 个 ticks。使用这个 ticks 来估算分数是：3.71 CoreMark/MHz，和测试的实际结果大致相等。

FPGA 综合评估

DE2-115 的这个 FPGA 工程可以用来评估 SCR1 和 SSRV 的延时和逻辑面积。这一节是对这两种处理器内核进行评估。

首先，在 rtl/define_para.v 中，使用 USE_SSRV 可以切换处理器内核采用 SCR1 原版还是使用 SSRV 来代替。如果打开这个定义，会引用 SSRV，那么后续的定义生效；如果关闭这个定义，会使用 SCR1 的内核，此时生效的配置文件为：

scr1/src/includes/scr1_arch_description.svh。

● SCR1 内核：使用硬件乘法器

关闭 rtl/define_para.v 中的 USE_SSRV 定义。打开 scr1/src/includes/scr1_arch_description.svh，进行配置。

首先，确保关闭自定义配置：

```
//`define SCR1_CFG_RV32EC_MIN
//`define SCR1_CFG_RV32IC_BASE
//`define SCR1_CFG_RV32IMC_MAX
```

然后打开 RVM 和 RVC 选项。关闭两个 BYPASS 配置，确保为时序最快的实际流水线。然后选择快速乘法器，也就是使用 FPGA 内部的乘法器，一个周期即可计算 32 乘 32 的乘法。

```
//`define SCR1_RVE_EXT           // enables RV32E base integer instruction set
`define SCR1_RVM_EXT             // enables standard extension for integer
mul/div
```



```

`define SCR1_RVC_EXT                // enables standard extension for compressed
instructions

// `define SCR1_IFU_QUEUE_BYPASS    // enables bypass between IFU and IDU stages
// `define SCR1_EXU_STAGE_BYPASS    // enables bypass between IDU and EXU stages

`define SCR1_FAST_MUL                // enables one-cycle multiplication

// `define SCR1_CLKCTRL_EN          // enables global clock gating

//`define SCR1_VECT_IRQ_EN          // enables vectored interrupts
//`define SCR1_CSR_MCOUNTEN_EN     // enables custom MCOUNTEN CSR

```

最后排除其他调试等模块：

```

`define SCR1_CFG_EXCL_UNCORE        // exclude DBG, BRKM, IPIC (also set in
SCR1_CFG_RV32EC_MIN)

```

使用这种配置后，进行仿真，可以得到它的性能测试分数：2.1 CoreMark/MHz。

然后使用这种内核配置进行综合和时序分析。经过 Quartus Prime 综合后，可以知道它的逻辑消耗为：6,001 / 114,480 (5 %)。在 Slow 1200mV 85C Model 下的最高频率为：34.0 MHz。

- SCR1 内核：不使用硬件乘法器

在采用上面的配置，并对`define SCR1_FAST_MUL 进行关闭。然后进行仿真，可得它的性能测试分数：1.3 CoreMark/MHz。

逻辑消耗为：5,983 / 114,480 (5 %)。在 Slow 1200mV 85C Model 下的最高频率为：30.68 MHz。

- SSRV 内核：配置为并行执行 1 条指令

后面都会采用 SSRV 内核。由于 SSRV 配置参数众多，因此是以并行执行多少条指令为核心。也就是说，配置各个缓冲器，保证 N 条指令同时执行。

在 rtl/define_para.v 中，打开 USE_SSRV 定义；关闭 INSTR_MISALLIGNED 定义。

对于内核的配置为：关闭 FETCH_REGISTERED；设置乘法器个数为 1；instrbits 缓冲器：1-2-1；schedule 缓冲器：2-1；membuf 缓冲器：2-1；mprf 缓冲器：2-1。

仿真后的性能评估分数：2.9 CoreMark/MHz。

逻辑消耗为：14,026 / 114,480 (12 %)。在 Slow 1200mV 85C Model 下的最高频率为：32.63 MHz。

- SSRV 内核：配置为并行执行 2 条指令

配置为并行执行 2 条指令的参数定义如下：

在 rtl/define_para.v 中，打开 USE_SSRV 定义；打开 WIDE_INSTR_BUS 和 INSTR_MISALLIGNED 定义。

对于内核的配置为：打开 FETCH_REGISTERED；设置乘法器个数为 1；instrbits 缓冲器：2-4-2；schedule 缓冲器：4-2；membuf 缓冲器：4-1；mprf 缓冲器：4-2。

注意，上面是仿真配置。如果需要 FPGA 执行，必须强制设定 instrbits 缓冲器的输入为 1，并且关闭 WIDE_INSTR_BUS 和 INSTR_MISALLIGNED 定义。因此，这种变化，不会引起最大频率上的差别，综合结果仍有参考意义。

这两种配置的性能分数：4.9 CoreMark/MHz 和 3.7 CoreMark/MHz。

逻辑消耗为：22,068 / 114,480 (19 %)。在 Slow 1200mV 85C Model 下的最高频率为：32.09 MHz。

- SSRV 内核：配置为并行执行 3 条指令

配置为并行执行 3 条指令的参数定义如下：

在 rtl/define_para.v 中，打开 USE_SSRV 定义；打开 WIDE_INSTR_BUS 和 INSTR_MISALLIGNED 定义。

对于内核的配置为：打开 FETCH_REGISTERED；设置乘法器个数为 2；instrbits 缓冲器：4-7-3；schedule 缓冲器：6-3；membuf 缓冲器：6-2；mprf 缓冲器：6-2。

同样 FPGA 执行时 instrbits 缓冲器的输入只能设定为 1。这两种情况下的性能分数分别为：

6.0 CoreMark/MHz 和 3.8 CoreMark/MHz。

逻辑消耗为：35,836 / 114,480 (31 %)。在 Slow 1200mV 85C Model 下的最高频率为：29.21 MHz。