铜鉴湖

RISCV32IM-Superscalar -CPU

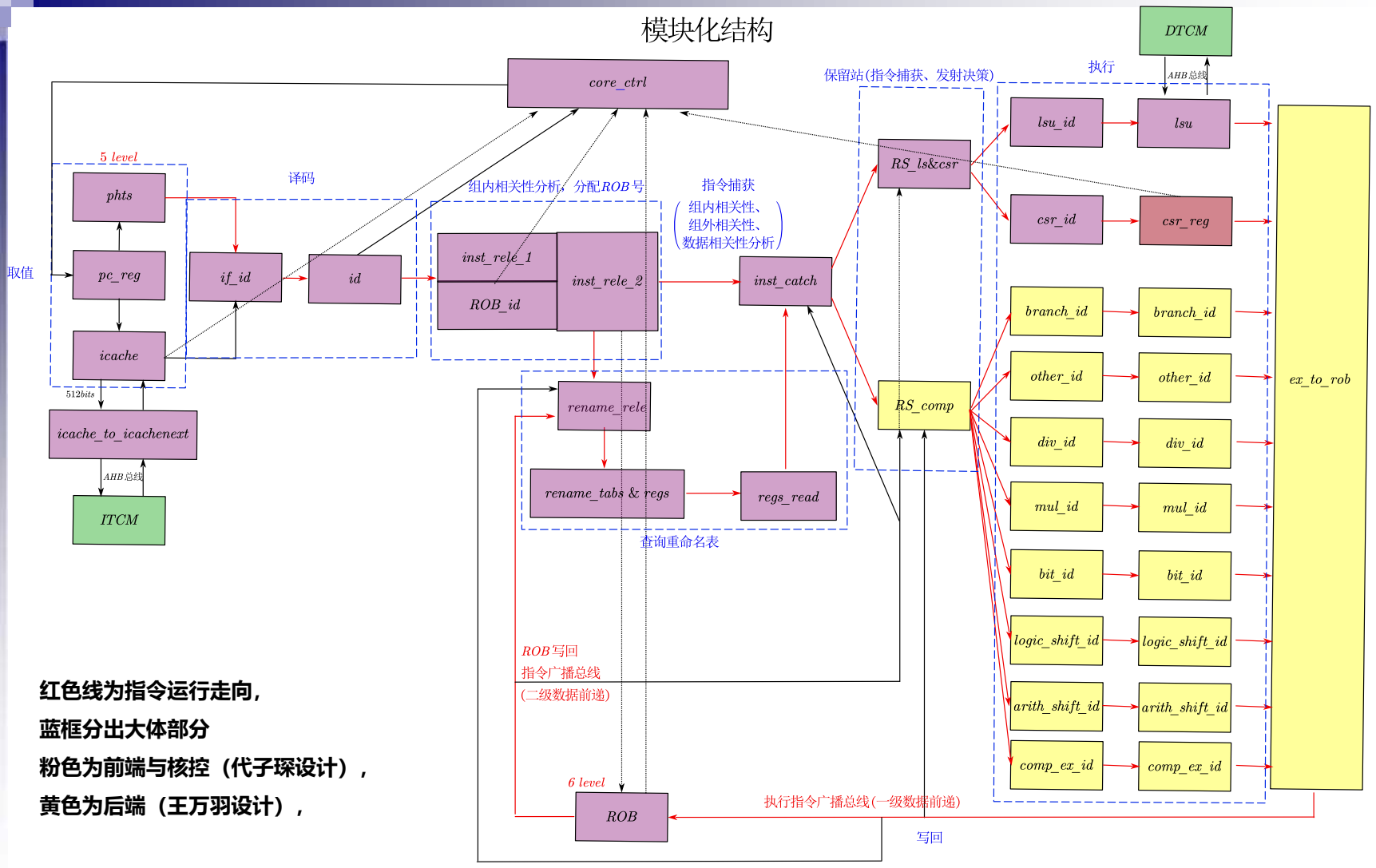
一个可拓展模块化的四发射乱序超标量处理器

主要负责人：代子琛、王万羽
小组成员：时少华、刘正新、陈卓晗、鄢宇豪

2024/3/9

目录

核内总体架构设计



红色线为指令运行走向，

蓝框分出大体部分

粉色为前端与核控（代子琛设计），

黄色为后端（王万羽设计），

核内总体架构设计

- **核心参数：**

每周期最大取值数量：4条

每周期最大发射宽度：5条

每周期最大写回指令数量：5条

icache最大容纳指令数：512条

RS最大指令容纳数：32条

ROB最大指令容纳数：128条

全流水拍数：≈40拍

分支预测表：1024个

使用synopsys进行40nm工艺综合：

面积≈ 1.6*1.7mm^2，静态功耗6W，时钟频率大约100MHz。

- **大体流水划分：**

前端流水：

取值：5cycle

译码：5cycle

组内相关性分析：2cycle

查询重命名表：7cycle

指令捕获：1cycle

保留站：≥3cycle

lsu、csr部件执行：≥3cycle

ROB写回：≥6cycle

后端流水：

更新数据reshape、指令报头译码：1cycle

指令入站：1cycle

保留站发射判断与发射：1cycle

保留站压缩与站内指令rs更新：1cycle

待执行队列选择执行部件：1cycle

待执行队列压缩：1cycle

执行部件控制译码产生控制信号：1cycle

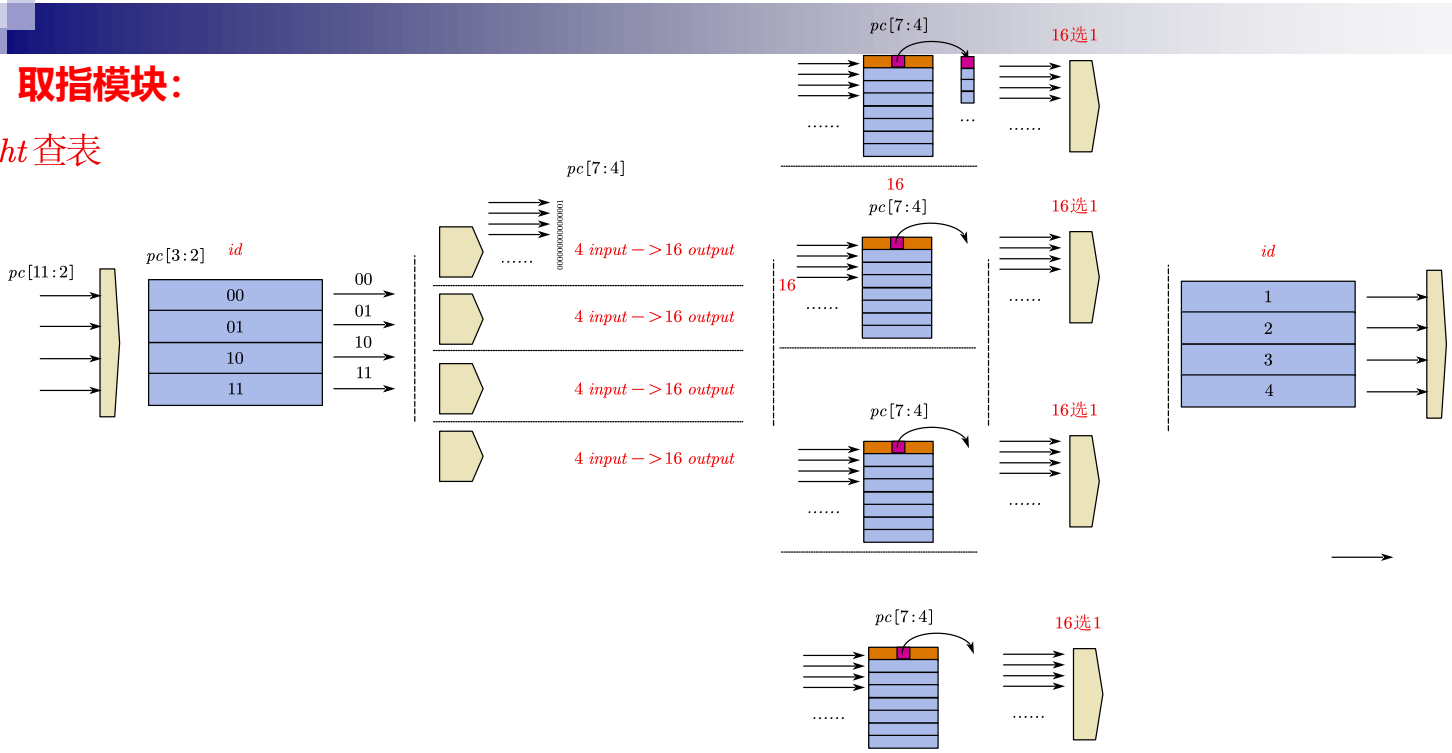
执行部件执行：1cycle

结果仲裁及写回：1cycle

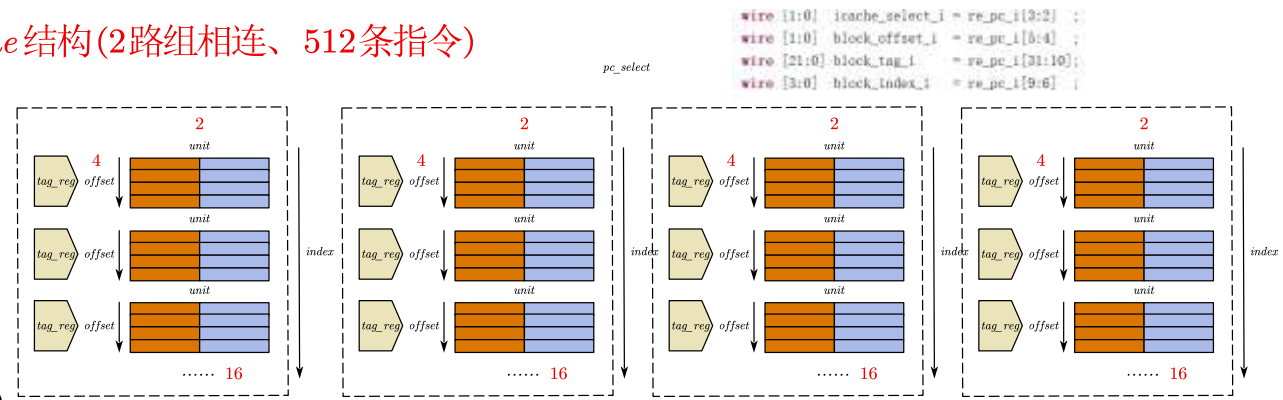
前端设计

取指模块：

pht 查表



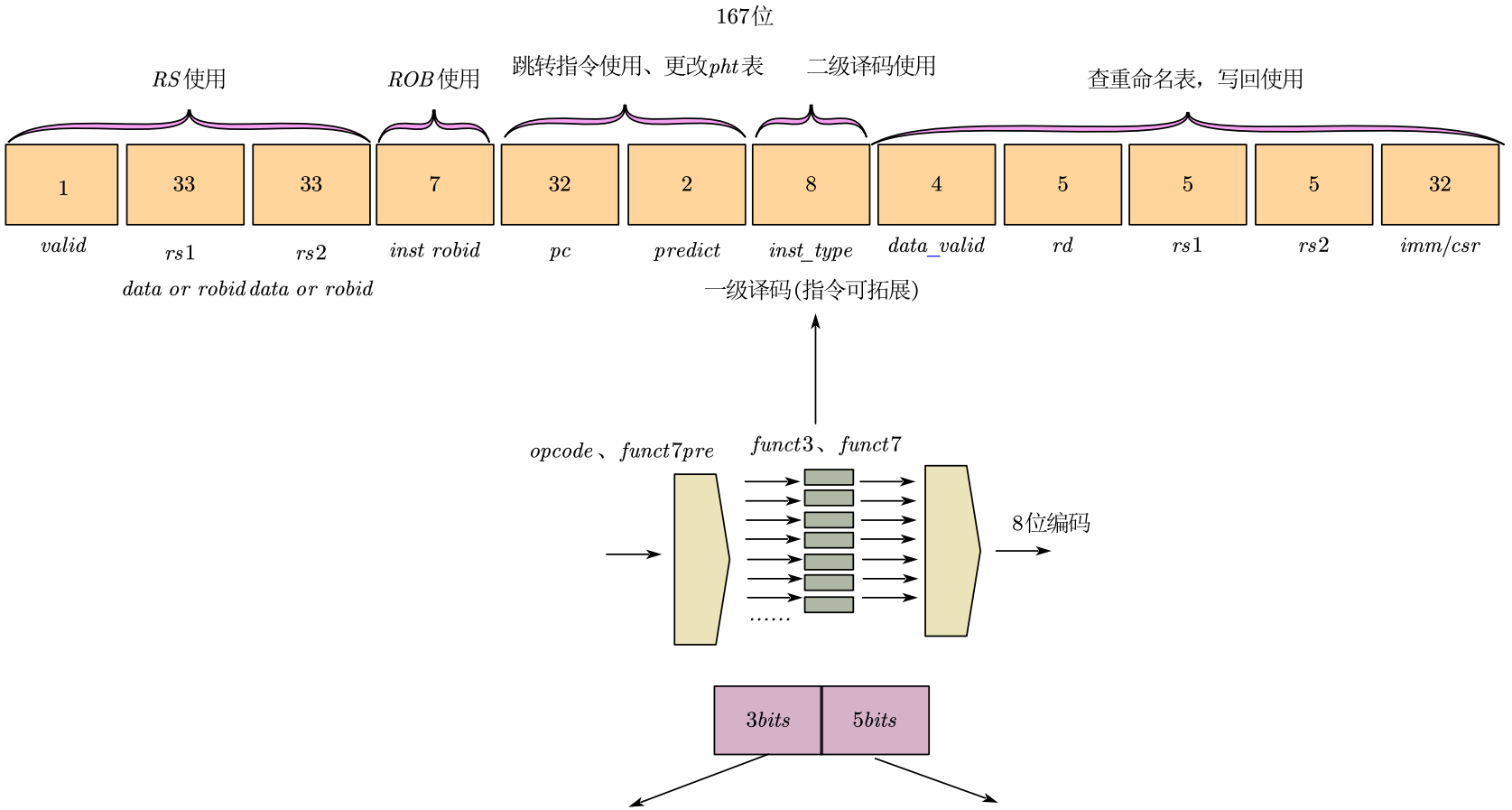
icache 结构(2路组相连、512条指令)



单个*unit*不同单元有不同*tag*，读是单个指令读，写一次写入16条指令(块区域写)。四条不同块两次读写。

前端设计

数据帧格式:(可拓展性、译码后出现、隔绝机器码与前端数据)



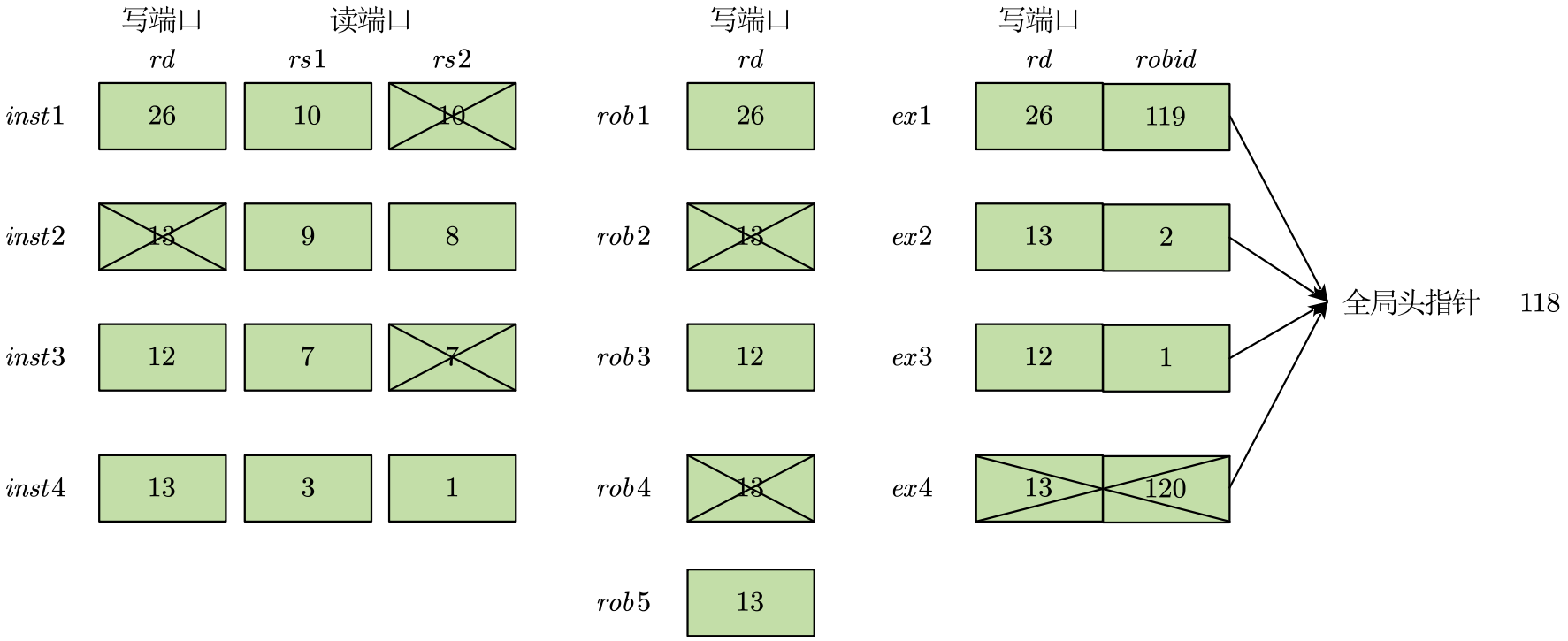
前端设计

重命名表读写：

1、多端口读写，读数保护机制。

2、一周期21端口读写32寄存器

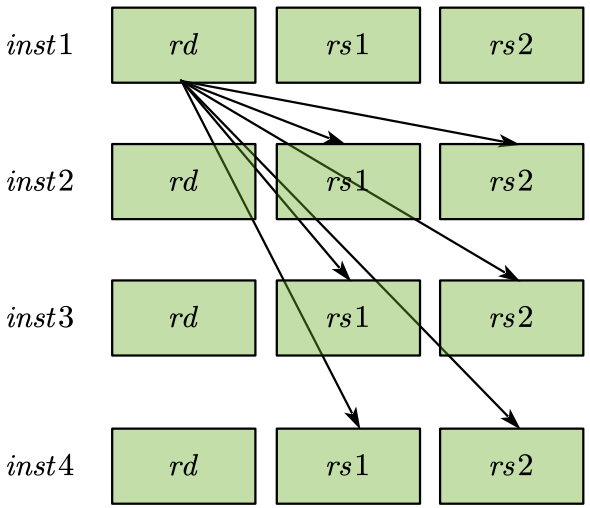
(指令写：4端口；指令读：8端口；ex写回：4端口；rob写回：　5端口；)



前端设计

重命名表处理指令真相关：

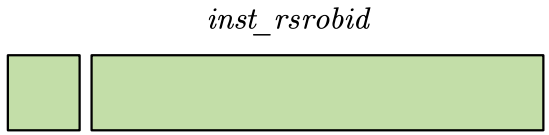
1、组内相关



2、重命名表相关

```
reg [ 1:0]  data_place;      // 最新数据在哪(00:regs; 10:miss 11:robs )
reg [ 6:0]  areg_robid;      // 最新数据相关的pc值
reg [31:0]  areg_data;       // 寄存器数据
reg [31:0]  rob_data;        // rob最新存储数据
```

查询重命名表得出数据



1*bit*	32*bit*
1	*data*[31:0]
0	相关性指令*robid*[6:0]

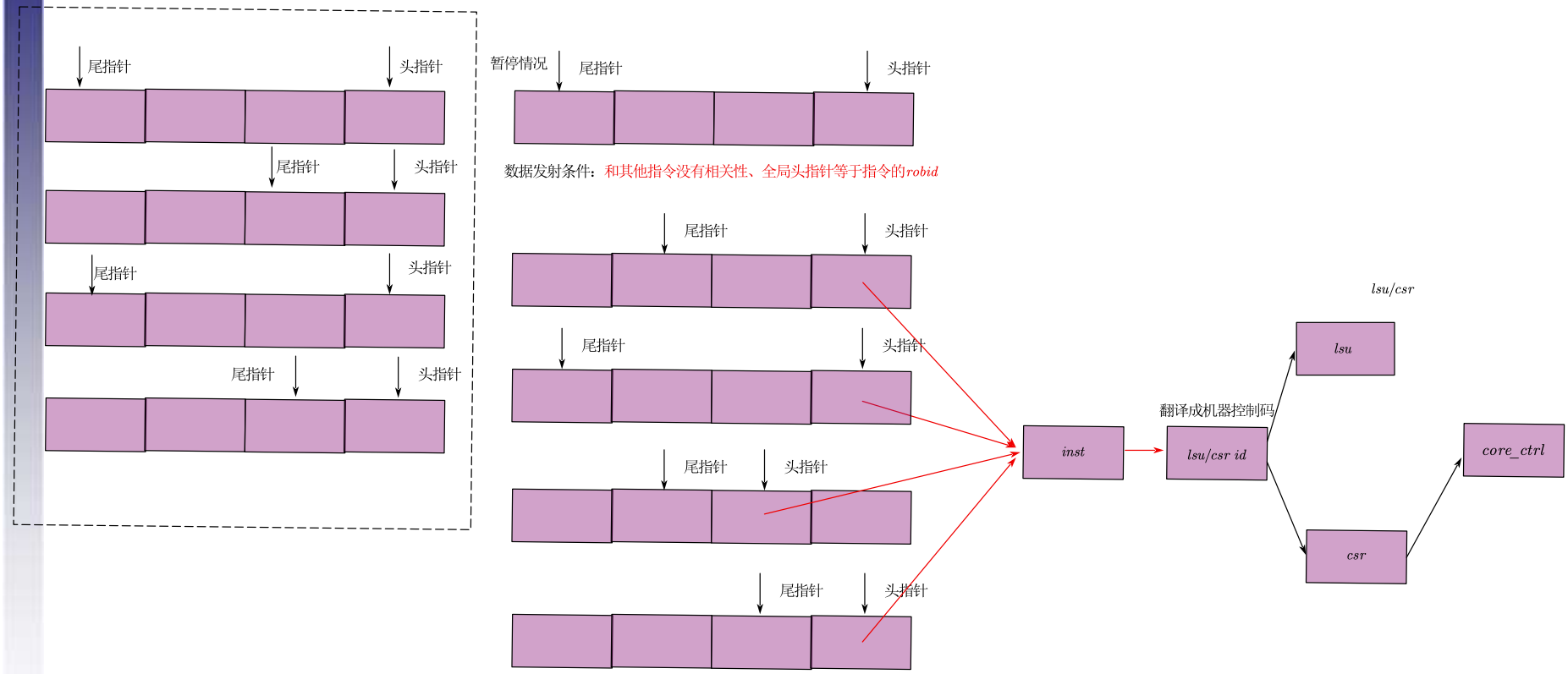
3、inst_catch与RS进行指令捕获

比较指令传过来的*robid*是否与*rsrobid*一致

优先级：组内相关性 > 重命名表相关性

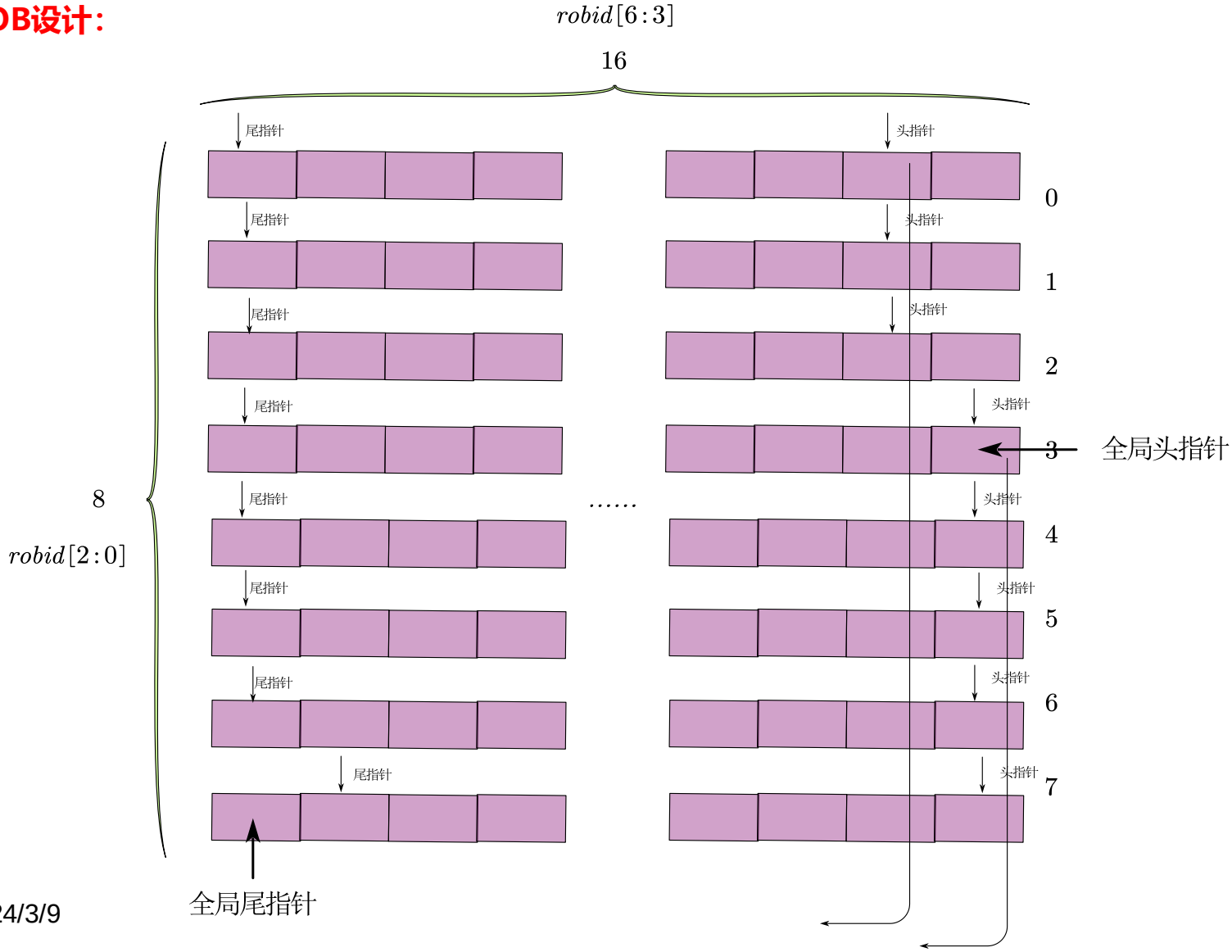
前端设计

RS_ls&csr设计（顺序访存）：



前端设计

ROB设计：



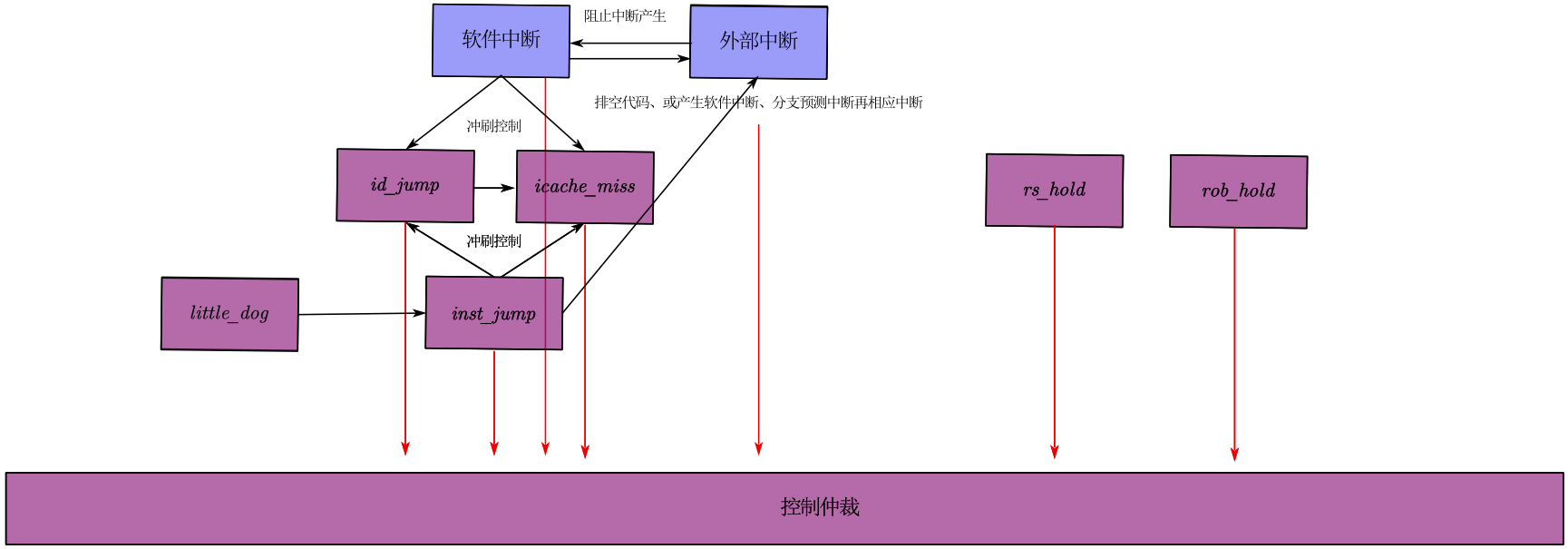
核内控制

核控信号：冲刷、暂停、pc跳转

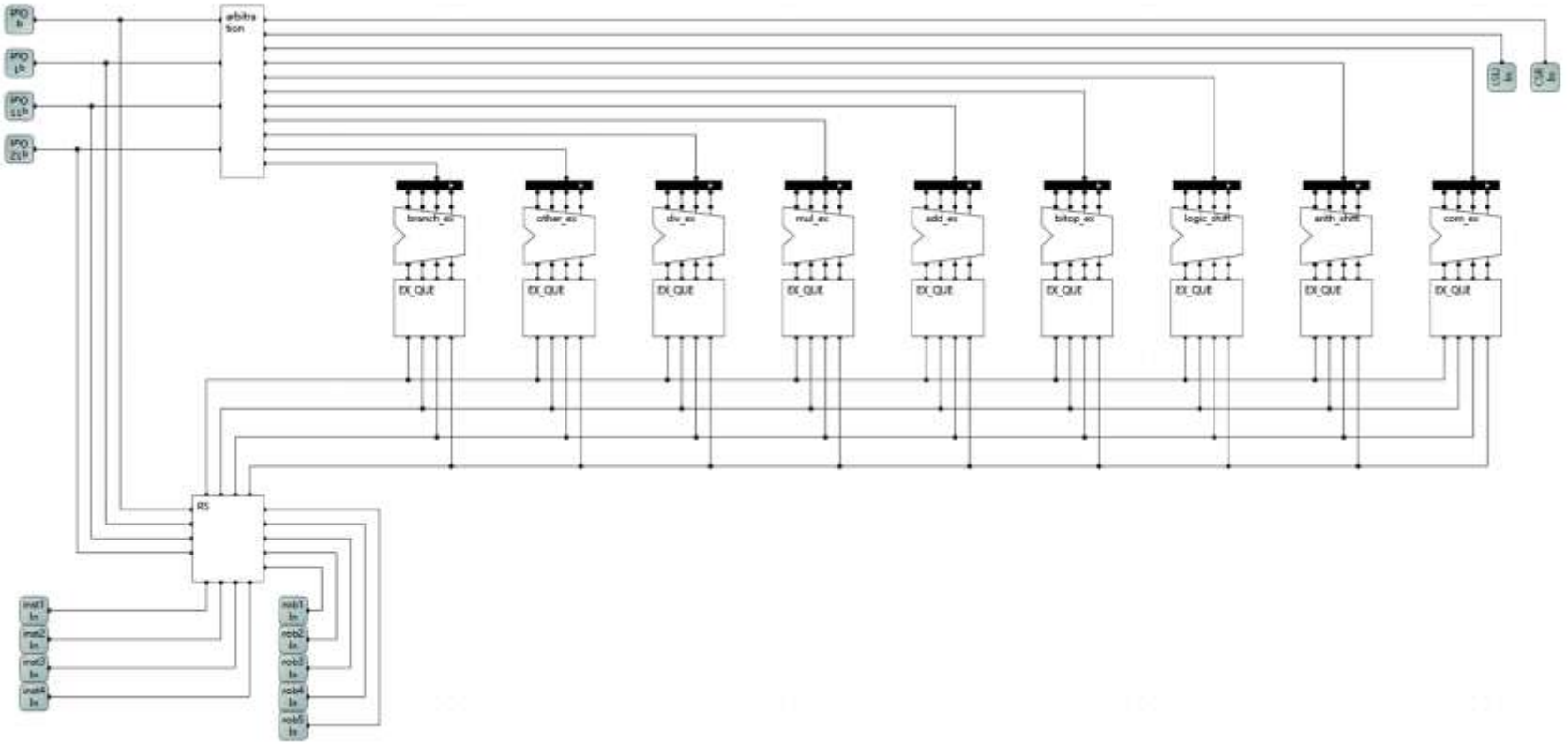
核控原因：外部中断(指令跳转);icache指令缺失，分支预测跳转，指令跳转，外部中断，软件中断（冲刷、指令跳转）；RS、ROB满信号（暂停、指令跳转）；看门狗（冲刷、指令跳转）。

核控优先级仲裁：

外部中断>分支指令跳转=看门狗中断>软件中断>分支预测跳转> icache指令缺失



后端设计


arbitration
RS
branch_ex
other_ex
div_ex
mul_ex
add_ex
bitop_ex
logic_shift
arith_shift
com_ex
EX_OUT

后端设计

- **低位优先编码：在整个工程中大量使用的低位优先编码器，用于各种优先级仲裁判断，并行度和延迟表现都很好。**

```
assign pre_req_0[0] = 1'b0;
assign pre_req_0[7:1] = req[6:0] | pre_req_0[6:0];
assign gnt_0 = req & ~pre_req_0;
```

假设8位信号req为 0011_1100;
pre_req_0即为 0000_0000;

```
pre_req_0[1] = req[0] | pre_req_0[0] = 0 | 0 = 0
pre_req_0[2] = req[1] | pre_req_0[1] = 0 | 0 = 0
pre_req_0[3] = req[2] | pre_req_0[2] = 1 | 0 = 1
pre_req_0[4] = req[3] | pre_req_0[3] = 1 | 1 = 1
pre_req_0[5] = req[4] | pre_req_0[4] = 1 | 1 = 1
pre_req_0[6] = req[5] | pre_req_0[5] = 1 | 1 = 1
pre_req_0[7] = req[6] | pre_req_0[6] = 0 | 0 = 0
```

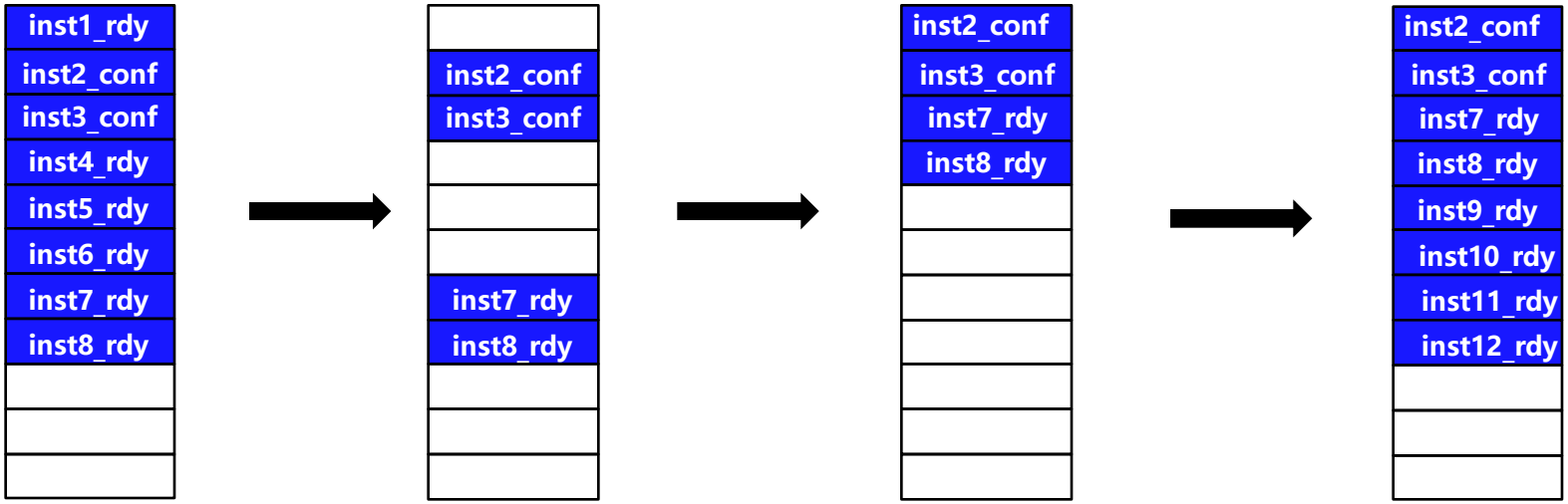
pre_req_0即为0111_1000

gnt_0即为 0011_1100 &
1000_0111

结果为0000_0100

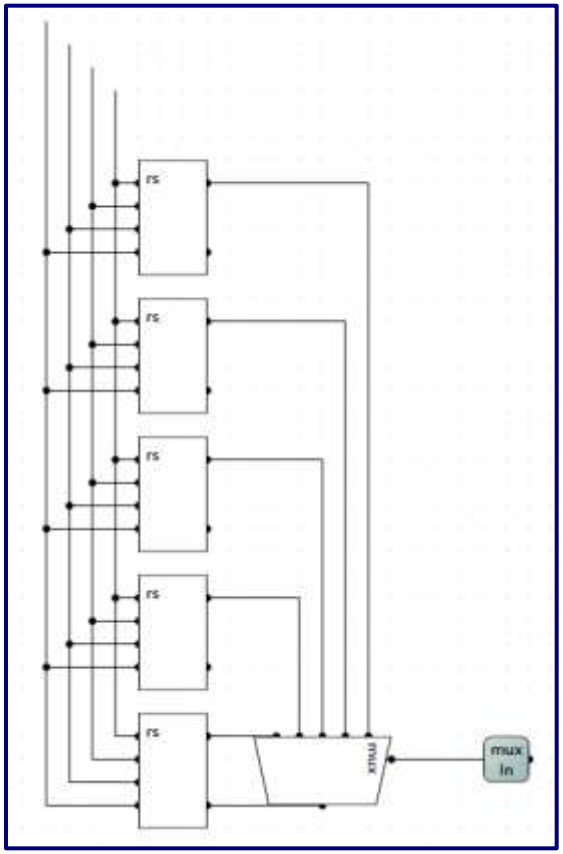
后端设计

- **保留站设计：全局保留站，具有良好的可扩展性；发射后压缩进位，实现Oldest-First机制的同时减少发射指令选择的复杂度；并行性的压缩-发射选择机制设计，减少后续进行功能扩展对时序表现的影响。**



后端设计

保留站内部简略图

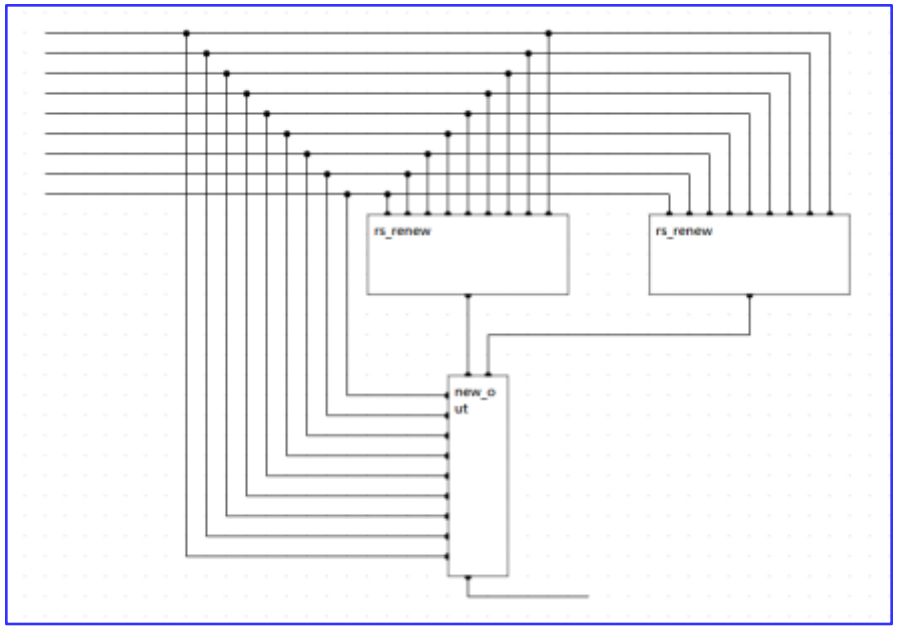


- 指令载入总线全相连到每个站中（自定义寄存器），由外部控制器产生的write_enable信号控制每个站取指or不取指，以及取哪条总线上的指令（类似单主多从的总线通信）；
- mux控制信号由每个站的发射状态（例如本来有指令但下个周期将发射，此站状态设为空泡状态）产生，用来控制MUX进行指令压缩；
- 指令发射使用低位优先仲裁器判断，最靠近栈底的优先级最高（oldest-first）；

MUX 5in 1out，控制信号有5位，控制压缩更新信号的来源，例如00001则自更新，说明不需要压缩，00011则更新上一个站中值，说明上周期产生1空泡，压缩一位；至多为11111，说明上周期发射4指令产生4空泡，压缩4位；
MUX控制信号并行的由一套组合逻辑产生，由12级逻辑运算与5级移位操作实现，延迟很低；

后端设计

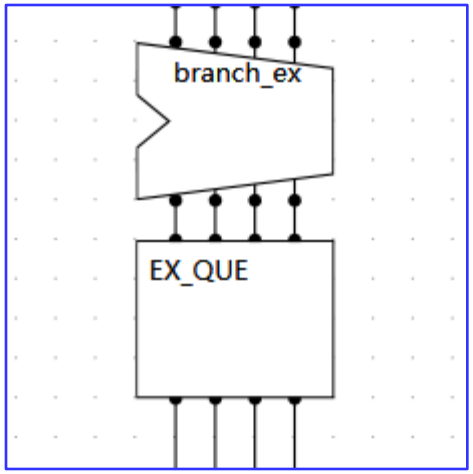
保留站内部数据冒险更新



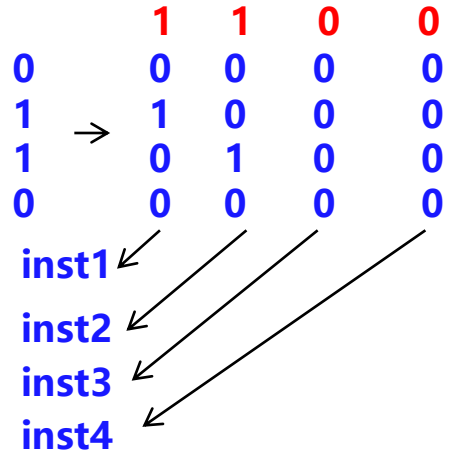
两个更新模块（一个负责rs1一个负责rs2）侦听更新总线的9个回传结果与站内存在相关性的目标ROBID比对，比对成功则output9位独热码，new_out模块结合两个9位独热码在选中的更新总线中取到更新rs_data并产生更新后的完整data_out寄存在站中，同时将站内指令状态位设置为可以发射，由站外控制器进一步选择可发射的指令（还需考虑优先级与执行模块就绪状态）。

后端设计

指令发射后执行部件的取指



- 任何指令在进入保留站前会有进一步译码，筛选该指令执行需要用到的执行部件，并加入9位独热码的指令报头，各个执行部件的待执行队列会实时侦听保留站发射总线上指令的9位报头，有符合本部件的指令就取入队列；
- 每个待执行队列有8位深度，当第四位被占用时向保留站发射执行繁忙信息，保留站便不会再发射该类型的指令，直到繁忙信息消失；
- 指令只要进入待执行队列就一定是没有相关性的，只需要选择空闲的执行部件送入即可（每种执行部件都有4个）；



- 难点在于从保留站发射指令取来的指令不是对齐的，例如有可能取进来后是0101（2和4号位有指令，1和3号位为空），而执行部件空闲状态因为结果仲裁优先级和执行延迟的原因，也不是对齐的，例如乘法器14是繁忙的，23是空闲的；
- 这里有一个控制模块使用4并行的2级优先编码+4并行的1级移位操作为空闲执行部件送入待执行指令。

后端设计

执行部件设计：后端部分设计了9种执行部件，分别为分支跳转、其他、除法、乘法、加法、逻辑运算、逻辑移位、算数移位、比较器；

- 加法器是先行进位加法器；
- 乘法器是dadda tree乘法器；
- 其他执行部件没有做特殊优化；

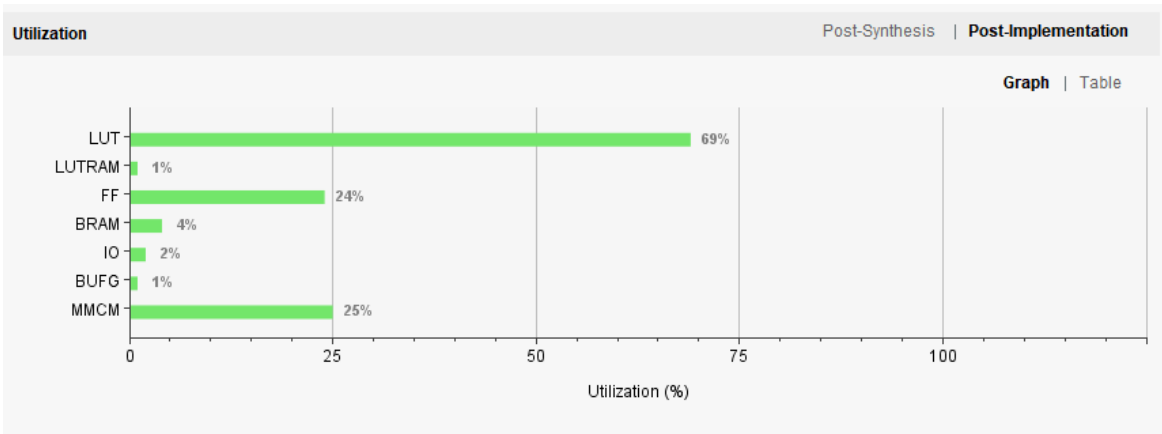
结果优先级仲裁器：所有执行结果的写回是有优先级区分的，因为每种执行部件的执行周期数不同，同样每个周期发送4条待执行指令，可能在第n个周期没有结果执行完毕，可写回结果为0，而在n+1周期，却有大于4条指令执行完毕，可写回的结果也大于4，ROB的项数和执行部件的个数都很多，不可能做到全相连，因此结果总线限制为4条，每周期写回优先级最高的4条指令；

- 结果仲裁器接收LSU和CSR执行结果并设置为最高优先级，这么做有两个考虑，一是这两种指令优先级确实很高，分别影响着访存与中断；二是减少LSU\CSR执行部分的设计复杂度，如果不能保证这两种指令只要执行完毕就能写回，LSU\CSR执行部分也要设计相关的阻塞机制，会让系统变得臃肿；
- 跳转指令的优先级次之，因为分支跳转指令的执行结果影响着分支预测结果是否正确，如果不正确需要及时冲刷；
- 其他指令中也有影响PC值的指令，因此优先级也很高，所有影响PC的指令优先级都必须设置为最高；
- 其他普通的计算指令优先级都差不多，但因为除法和乘法的执行周期较长（为了平衡各个执行部件的门延迟级数，设置为32和3），将他们的优先级设置为最高，因为他们的rd已经等待结果很久了；
- 仲裁通过的指令会一方面发给rob，一方面通过旁路返回给保留站用作解决RAW冲突的RS更新；
- 仲裁通过的执行部件会通过局部CDB总线广播给RS、待执行队列，告知各个模块此部件空闲；
- 仲裁不通过的指令会暂时保存在执行部件中，此时执行部件依然设置为繁忙状态；

核外调试

格外调试部分：对RTL代码的综合、编译；管脚约束；静态时序约束；网络扇入扇出优化；LUTs占用优化；仿真波形与上板ILA对比debug；OS适配（boot部分程序的外设初始化参数设置、测试用例函数编写、OS编译）、串口及SD卡调试；

- **主要测试平台：xilinx ZCU102、xilinx ZCU106;**
- **串口芯片：CP2108、CH340;**
- **EDA工具：vivado;**
- **OS适配环境：Ubuntu 20.04;**
- **impl-WNS:14.581ns;impl-TNS:0ns; syn-WNS:40.104ns;syn-TNS:0ns;**
- **下图附资源使用量：ZCU102平台（600T）；**



上板测试

格外调试部分：基于xilinx ZCU106平台，自定义OS任务上板测试；

- 1次无冲突加法
- 1次RAW加法
- 1次无冲突乘法
- 1次无冲突除法
- 结果打印到串口

```c
#include "stdio.h"
#include "thread.h"
#include "int.h"
#ifdef QEMU
    #include "plic.h"
#endif
#include "user.h"
#include "timer.h"

void main(void)
{
    printf("Hello, OS!\n ");
    //delay(MAIN_D_TIME);
    delay(10);
    printf("- - - - - - - - - - - - - - - - - - - - - - - - -\n ");

    int a = 3, b = 2, c = 3, d = 4;
    int res_add, res2_mul, res3_div;
    int mid_add = a + b;

    // 进行计算
    res_add = mid_add - c; // res_add的结果为mid_add - c
    res2_mul = b * c;      // res2_mul的结果为b*c
    res3_div = d / b;      // res3_div的结果为d除以b

    // 打印结果
    printf("%d ", res_add);
        delay(10);
    printf("%d ", res2_mul);
        delay(10);
    printf("%d ", res3_div);
        delay(10);
    printf("- - - - - - - - - - mission completed - - - - - - - - - - -\n ");
        delay(1000);
    while(1){
        printf("Main Thread!!\n ");
        delay(50);
    }

}
```

组内成员完成情况

- 代子琛：内核总体架构设计，内核前端代码设计与编写，核控代码设计与编写，文档编写。
- 王万羽：内核后端代码设计与编写，核内外接口调试，编译器与OS适配内核，文档编写。
- 时少华：参与csr寄存器模块编写，文档编写。
- 刘正新：参与师兄编译器测试，文档编写。
- 陈卓晗：参与核外调试，文档编写。
- 鄢宇豪：