BOOMv3移植UniCore32之
控制流指令 & 读/写PC指令

梁书豪

目录

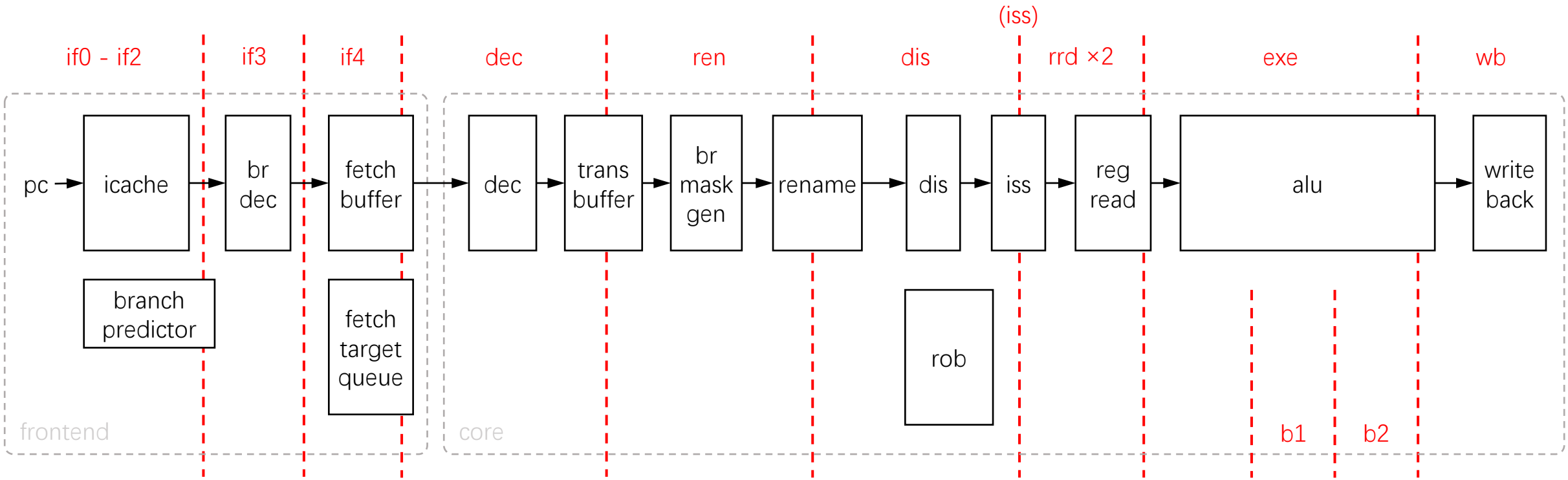
1. **控制流指令**
   - BOOM控制流指令基础
   - 移植UniCore32控制流指令
2. **读PC指令**
   - 增加FTQ端口
   - ALU rs1/rs2重连线
3. **写PC指令**
   - 额外执行JALR
   - 提交时flush

1.
控制流指令

BOOM控制流指令基础

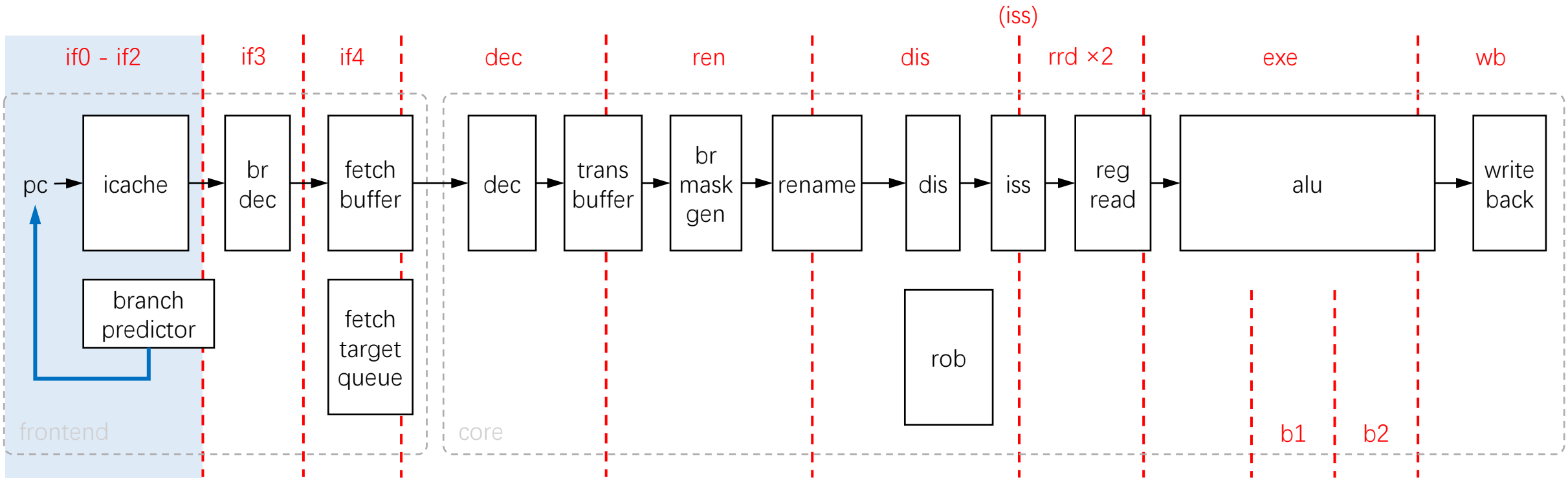
- RISC-V控制流指令
  - JAL：跳转到PC+offset，写回PC+4
  - JALR：跳转到寄存器，写回PC+4
  - BR：根据条件跳转到PC+offset

BOOM控制流指令基础



- 涉及阶段一览
- 执行过程解说

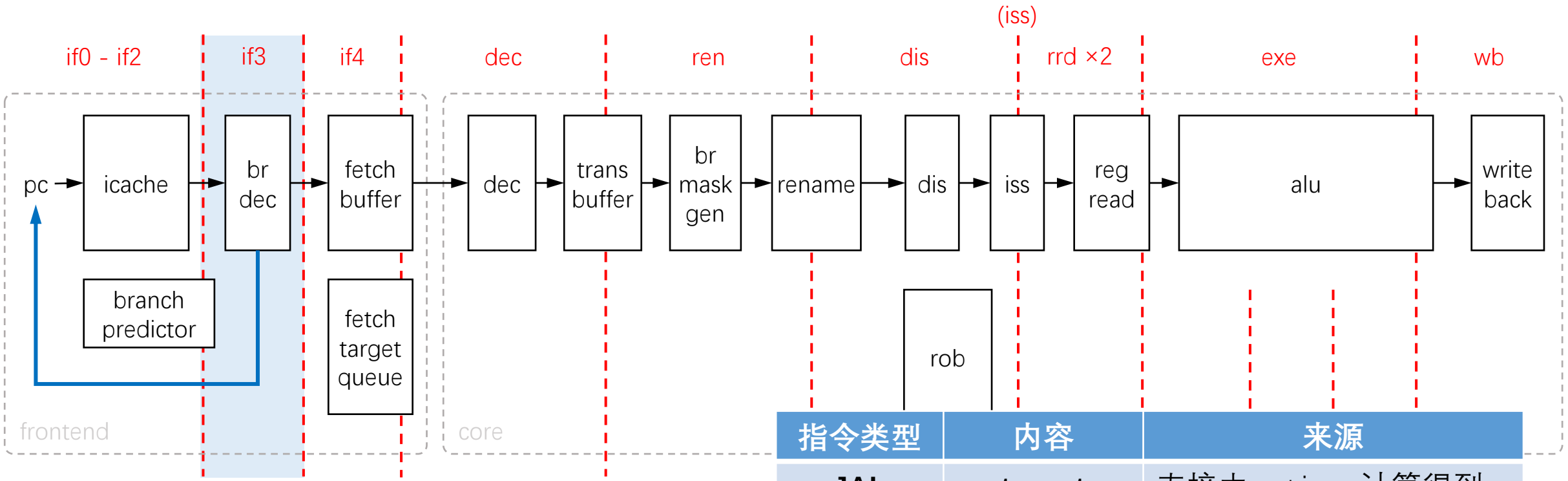
BOOM控制流指令基础



取指0-2

- 用2个周期从icache中取得pc处的指令
- 即使指令尚未取得，也会根据pc和history进行转移预测

BOOM控制流指令基础

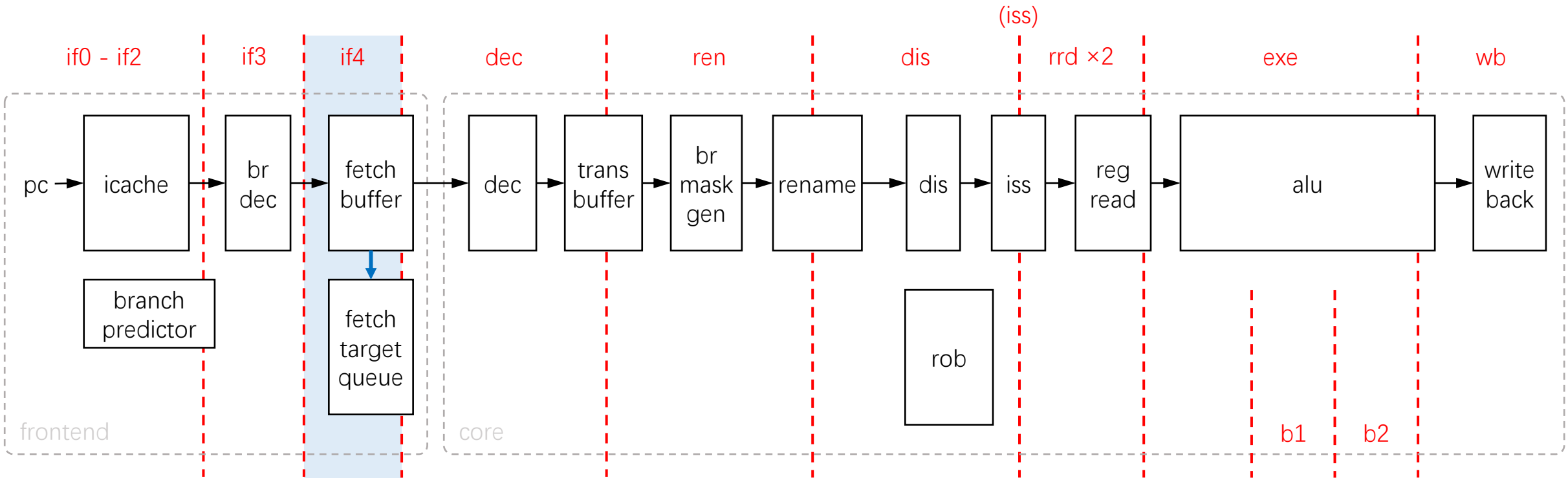


取指3

- 对取得的指令进行branch decode，即只译码其中的控制流指令
- 控制流的更新如表所示
- JAL到此已解决，JALR和BR未解决

指令类型	内容	来源
JAL	target	直接由pc+imm计算得到
JALR (ret)	target	由RAS预测
JALR (!ret)	target	由转移预测器预测
BR	target	直接由pc+imm计算得到
BR	direction	由转移预测器预测

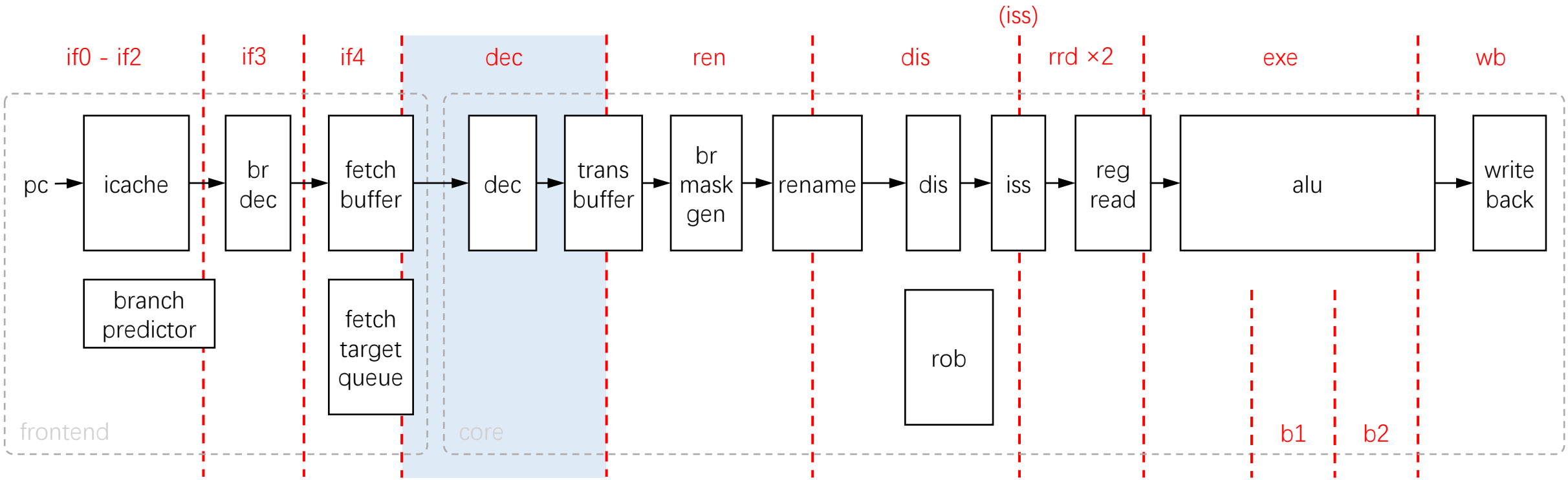
BOOM控制流指令基础



取指4

- 将pc和next_pc存到FTQ中

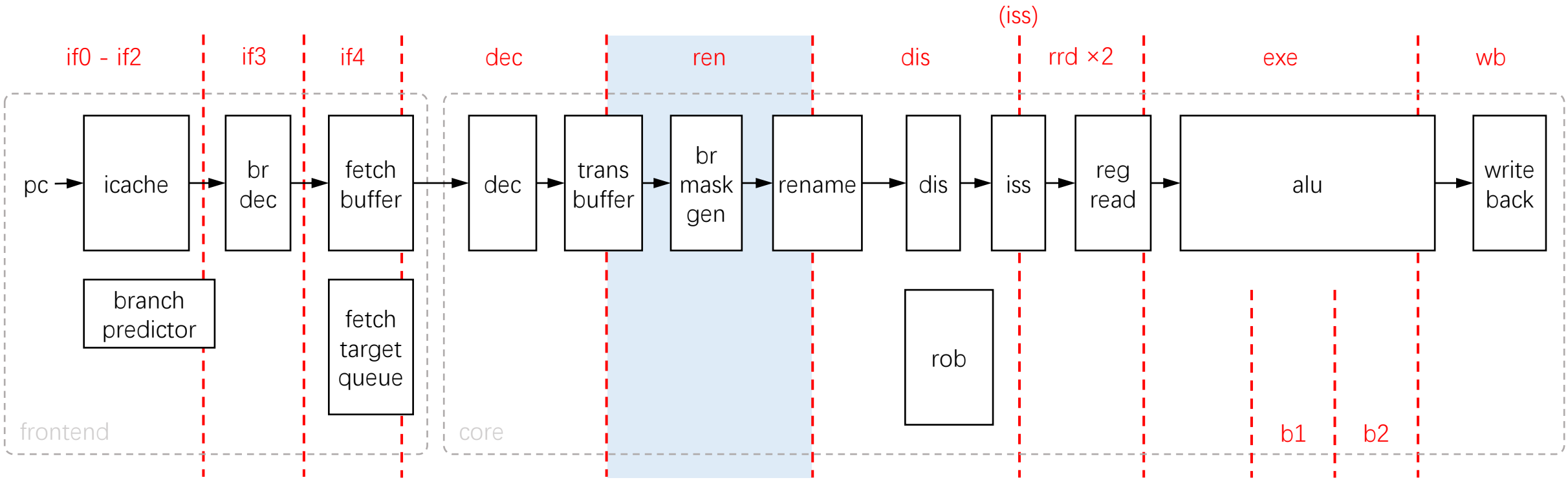
BOOM控制流指令基础



译码

- 将每条控制流指令各译码成一条微码

BOOM控制流指令基础



重命名

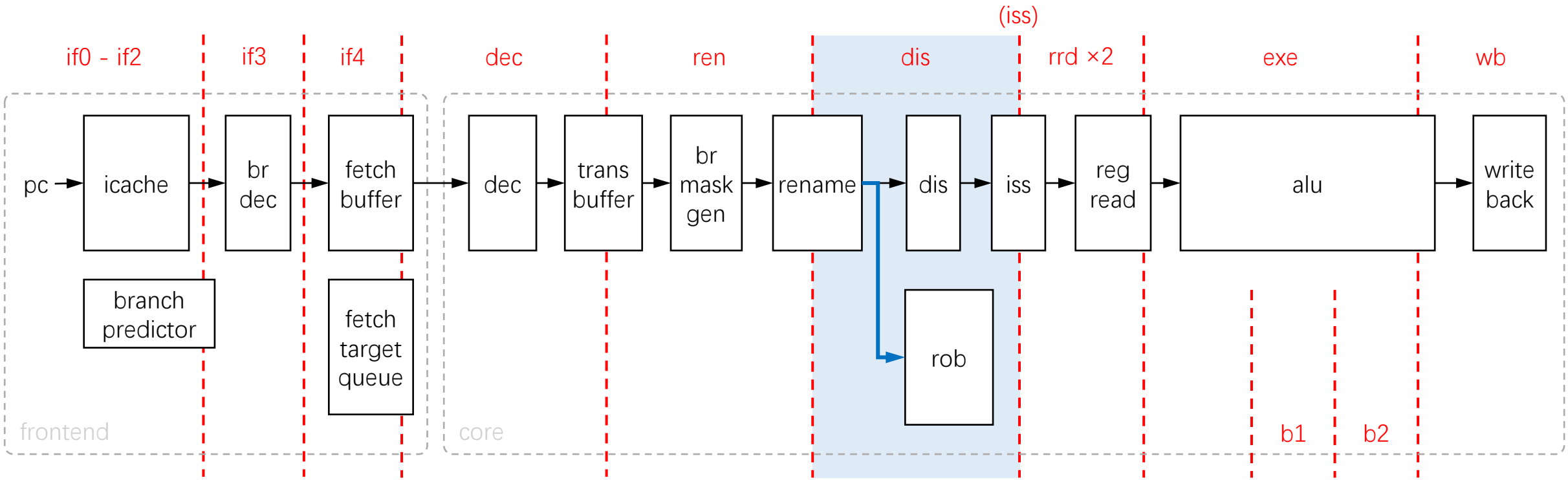
BrMaskGen

- 为每条BR/JALR分配一个br_tag (0≤br_tag<maxBrCount)
- 后续指令的br_mask将包含这条指令的位

Rename

- 为每条BR/JALR复制一份map_table和free_list

BOOM控制流指令基础



指派

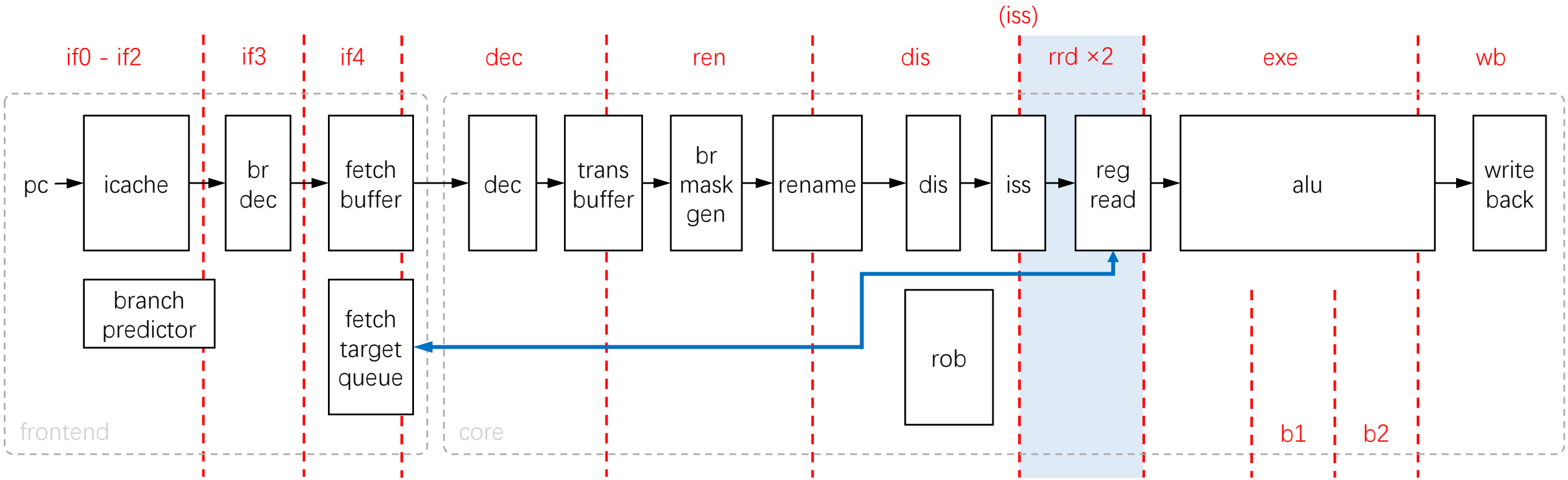
Dispatch

- 将JAL和JALR送往带JmpUnit的ALU
- 将BR送往任意一个ALU

ROB

- 将指令按顺序放入队列

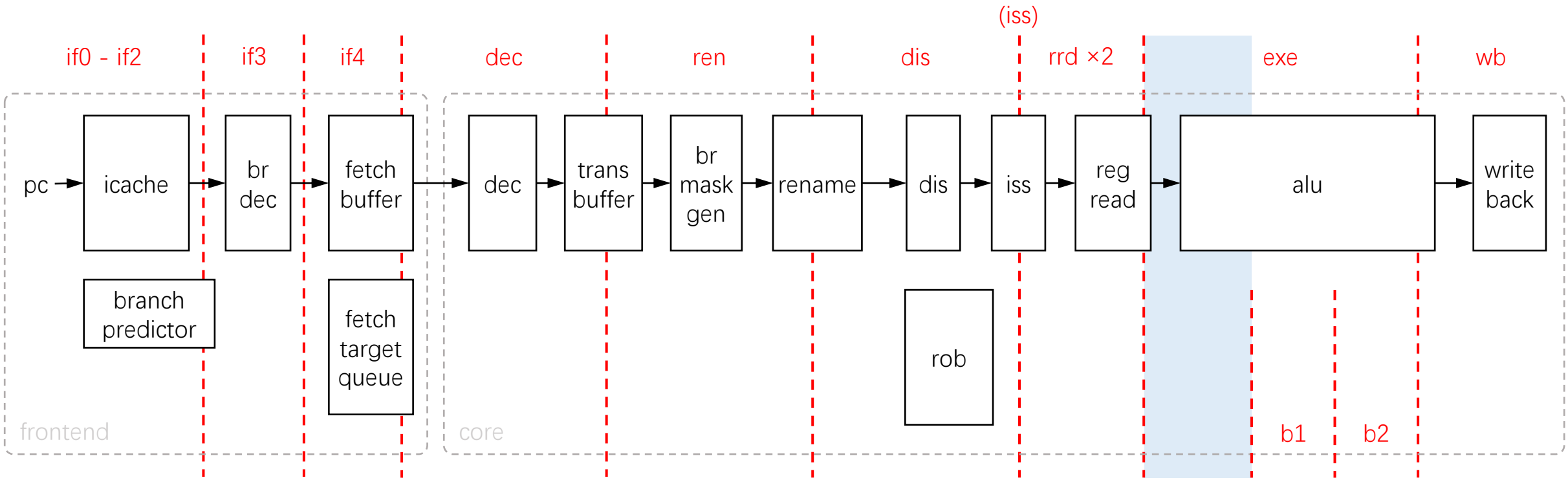
BOOM控制流指令基础



读寄存器

- JmpUnit向FTQ请求pc和next_pc

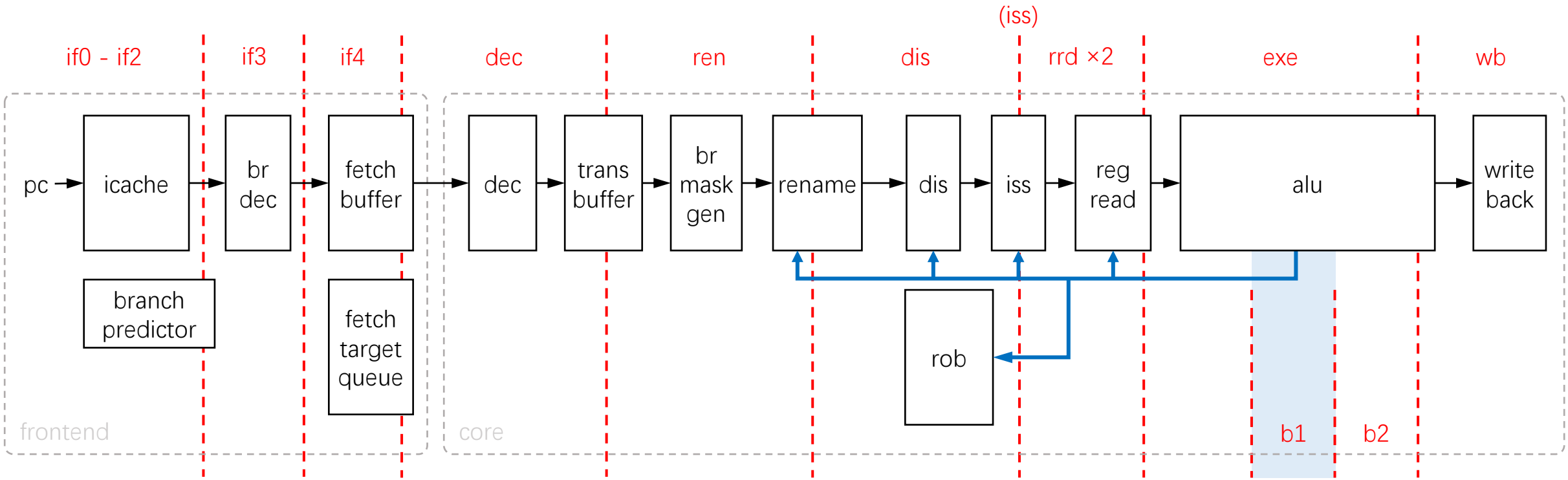
BOOM控制流指令基础



执行

- 计算JAL和JALR的链接地址 (pc+4)
- 计算BR的direction
- 计算JALR的target (rs1+imm)

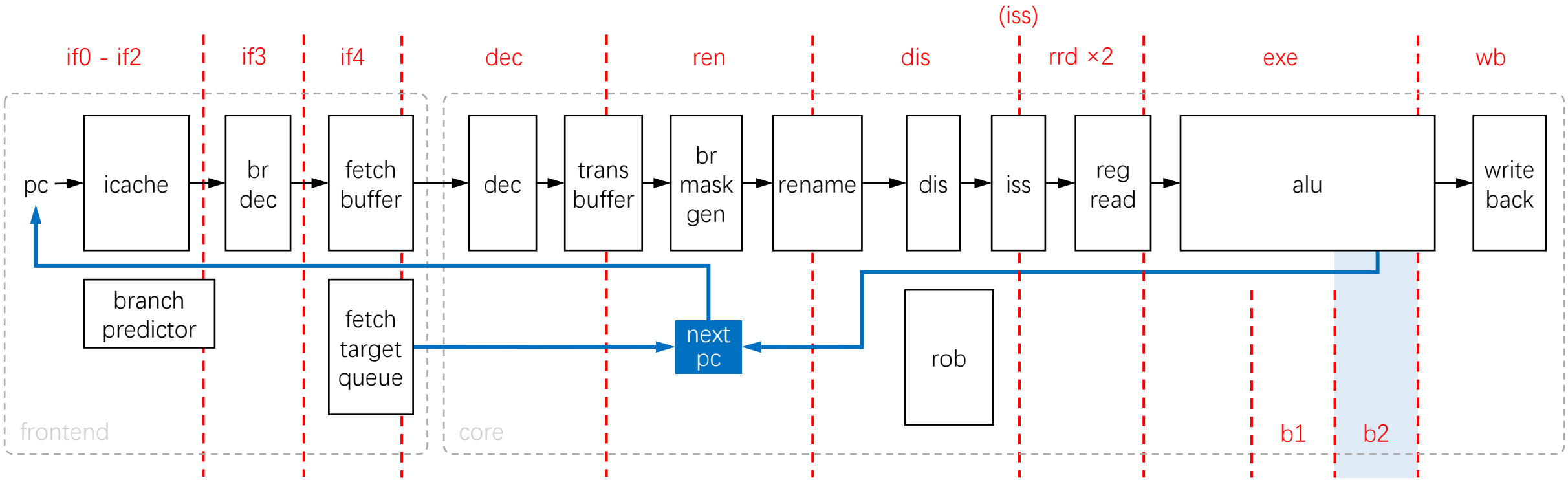
BOOM控制流指令基础



转移更新1

- 若预测正确，后续指令解除br_mask中的对应位
- 若预测错误，br_mask中包含对应位的指令变为气泡

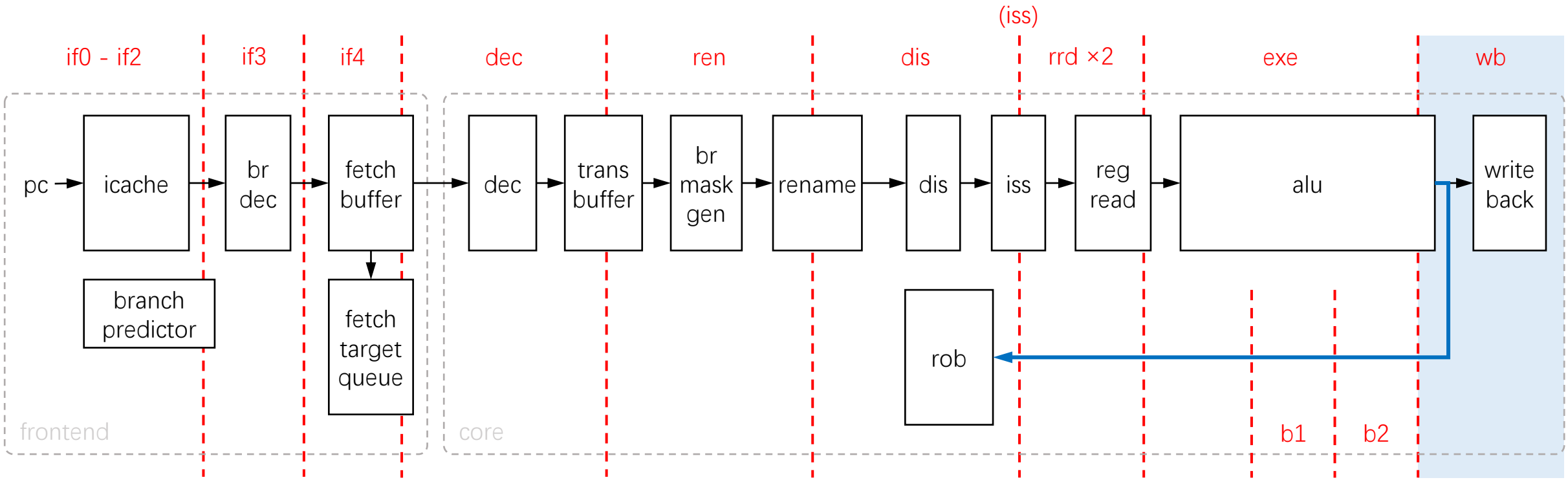
BOOM控制流指令基础



转移更新2

- 更新前端的pc；仅预测错误时执行
- JALR的target已得出
- BR的target需要重新计算

BOOM控制流指令基础



写回

- JAL和JALR写回链接地址
- ROB解除该指令的busy

移植UniCore32控制流指令

UniCore32控制流指令和RISC-V基本可以一一对应

UniCore32	条件	目标地址	RISC-V
B	无	相对	**JAL**
JUMP	无	绝对	**JALR**
Bcc	有	相对	**BR**

移植UniCore32控制流指令

区别

偏移量

- B和Bcc的是24位；运算为PC+4+imm
- JAL是20位，BR是12位；运算为PC+imm

链接

- Bcc.l会根据条件链接
- BR不会链接

条件

- Bcc的条件是通过计算flag而得
- BR的条件是通过比较源操作数而得

移植UniCore32控制流指令

解决方案

偏移量

- 将立即数统一记在uop.inst中
- 在ALU中增加PC+4+imm的运算

链接

- 增加一条微指令uopBCC
- 基本参照BR，但在写回r30上做类似于CMOV的操作
- 先读r30，在ALU中根据条件选择写回r30还是pc+4

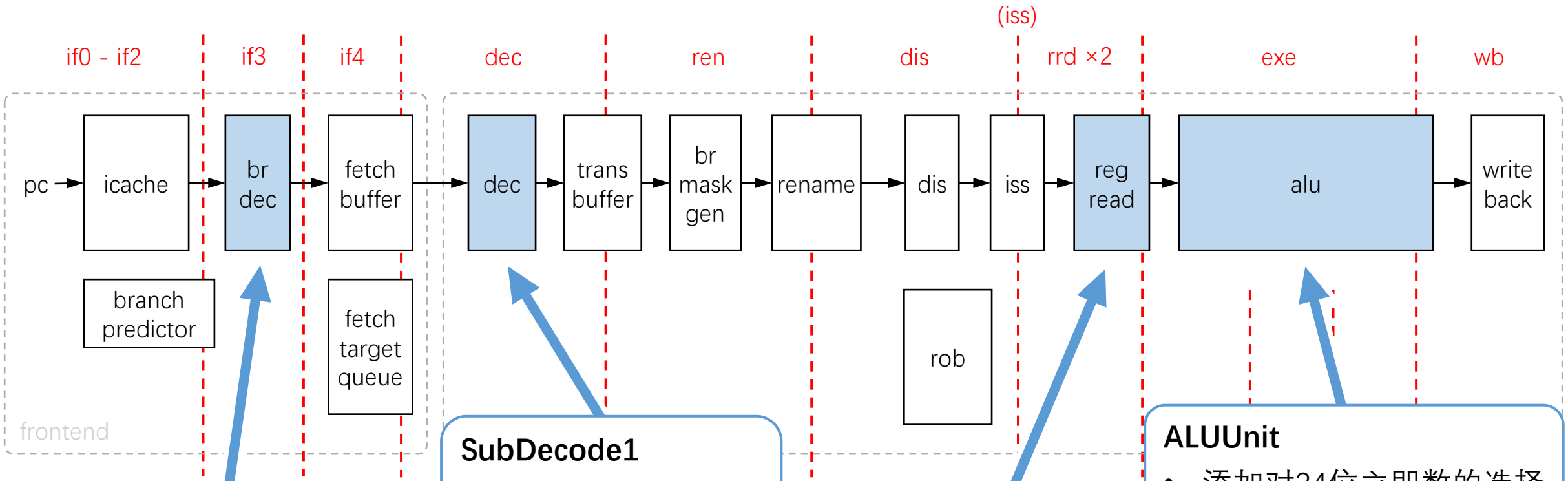
条件

移植UniCore32控制流指令

UniCore32控制流指令微码映射表

UniCore32	语义	uop
B #imm24	跳转到PC+4+sext(imm24)	**uopJAL** RT_NO, imm
B.L #imm24	跳转到PC+4+sext(imm24)，将PC+4写入r30	**uopJAL** r30, imm
Bcc #imm24	根据bc跳转到PC+4+sext(imm24)	**uopBCC** bc, RT_NO, imm
Bcc.L #imm24	根据bc跳转到PC+4+sext(imm24)，将PC+4写入r30	**uopBCC** bc, r30, imm
JUMP rs2	跳转到rs2	**uopJALR** RT_NO, rs2
JUMP.L rs2	跳转到rs2，将PC+4写入r30	**uopJALR** r30, rs2

移植UniCore32控制流指令



SubDecode1

- 添加B/JUMP/Bcc的微指令译码

BranchDecode

- 添加对B/JUMP/Bcc的支持
- 主要是为了解决B

ALURRdDecode

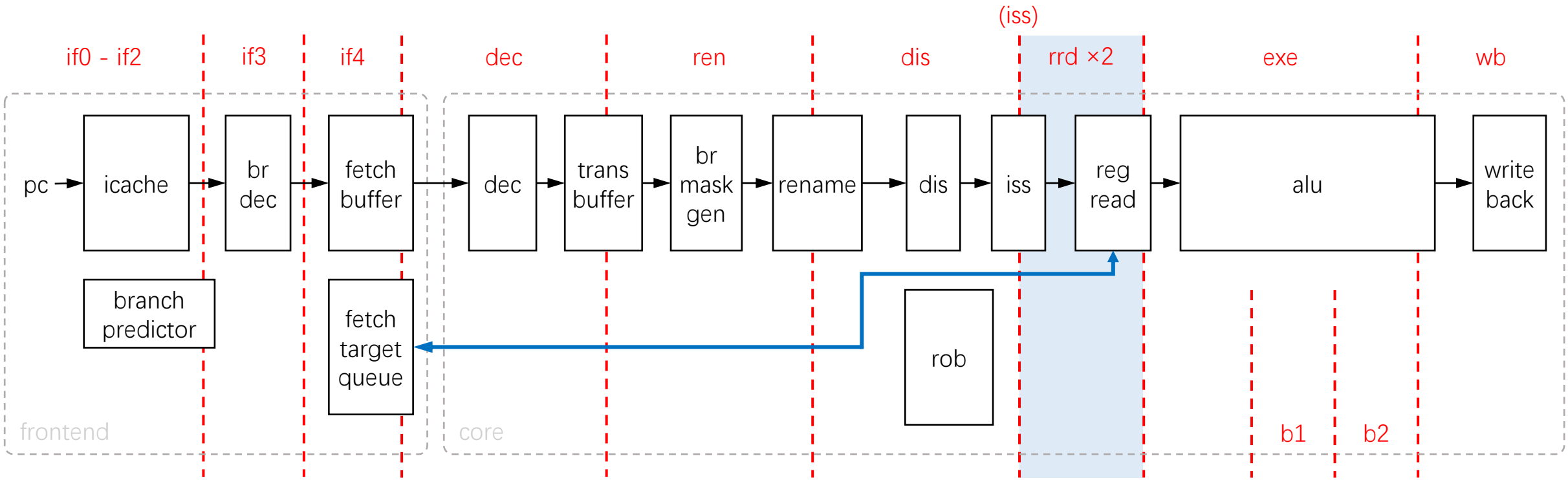
- 添加B/JUMP/Bcc的执行单元译码

ALUUnit

- 添加对24位立即数的选择
- 添加Bcc对flag的运算
- 添加Bcc对r30值的选择
- 添加PC+4+imm的计算

2.
读PC指令

Recap: BOOM读PC的方式



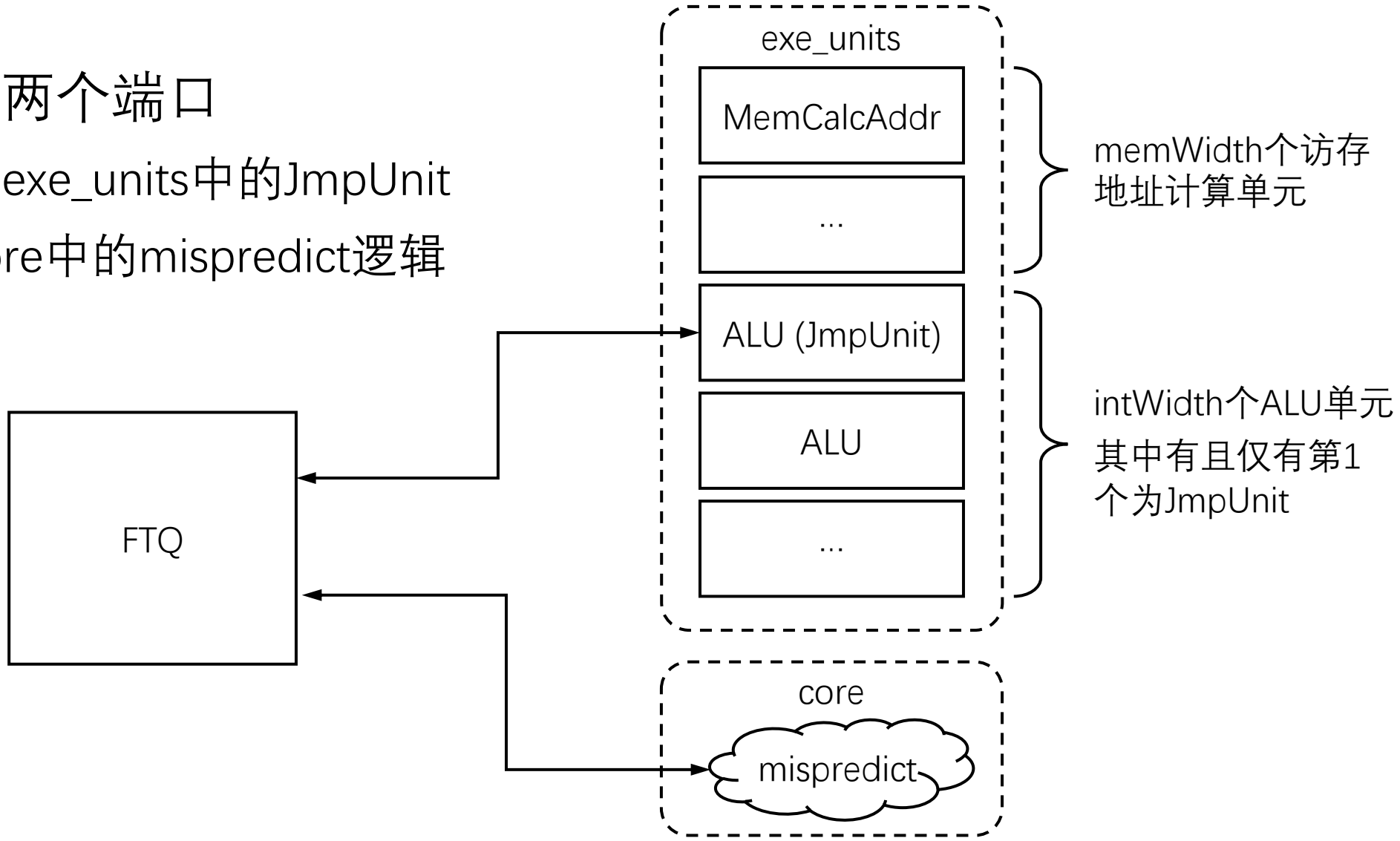
读寄存器

- JmpUnit向FTQ请求pc和next_pc

增加FTQ端口

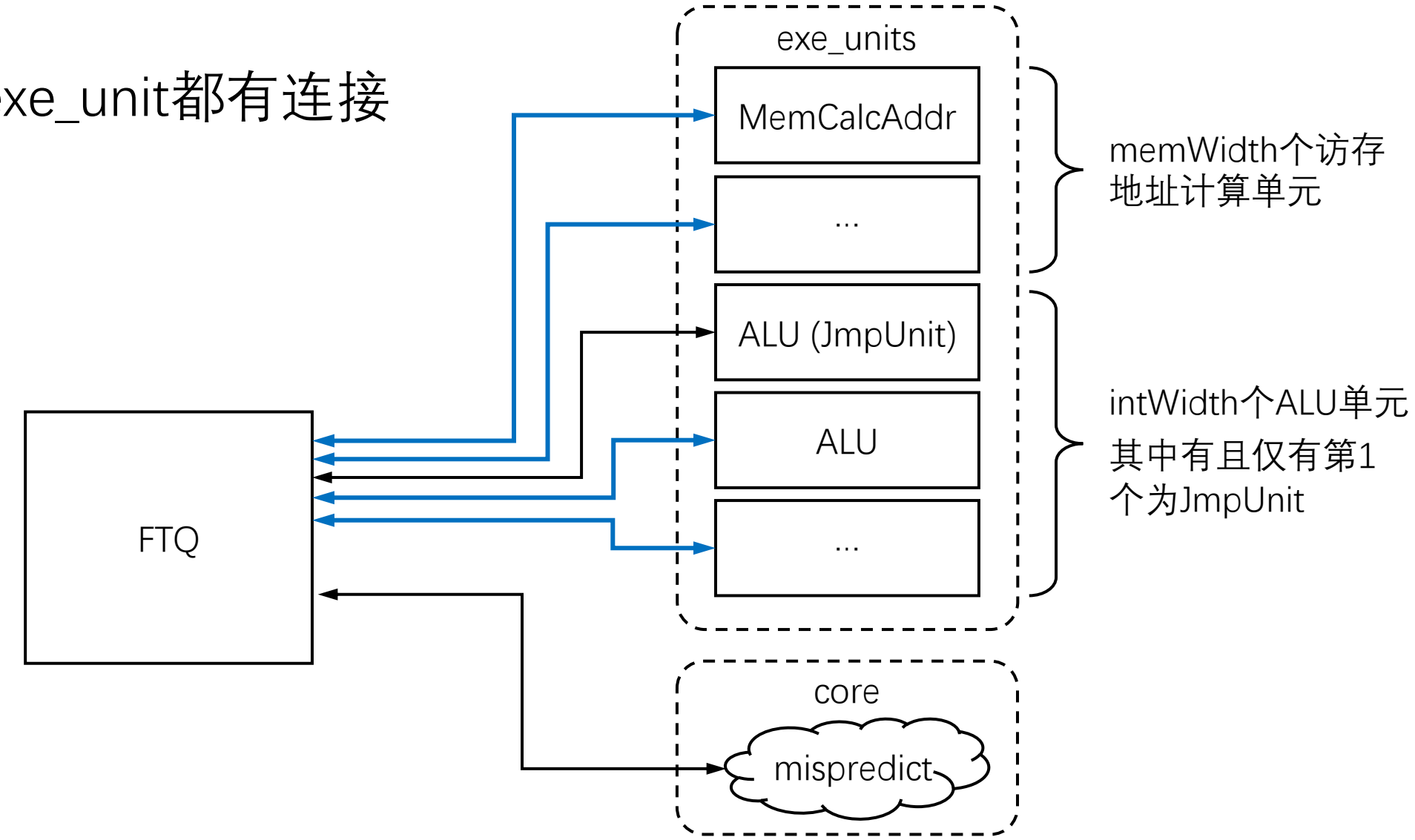
本来FTQ只有两个端口

- 其中一个连接exe_units中的JmpUnit
- 另一个连接core中的mispredict逻辑



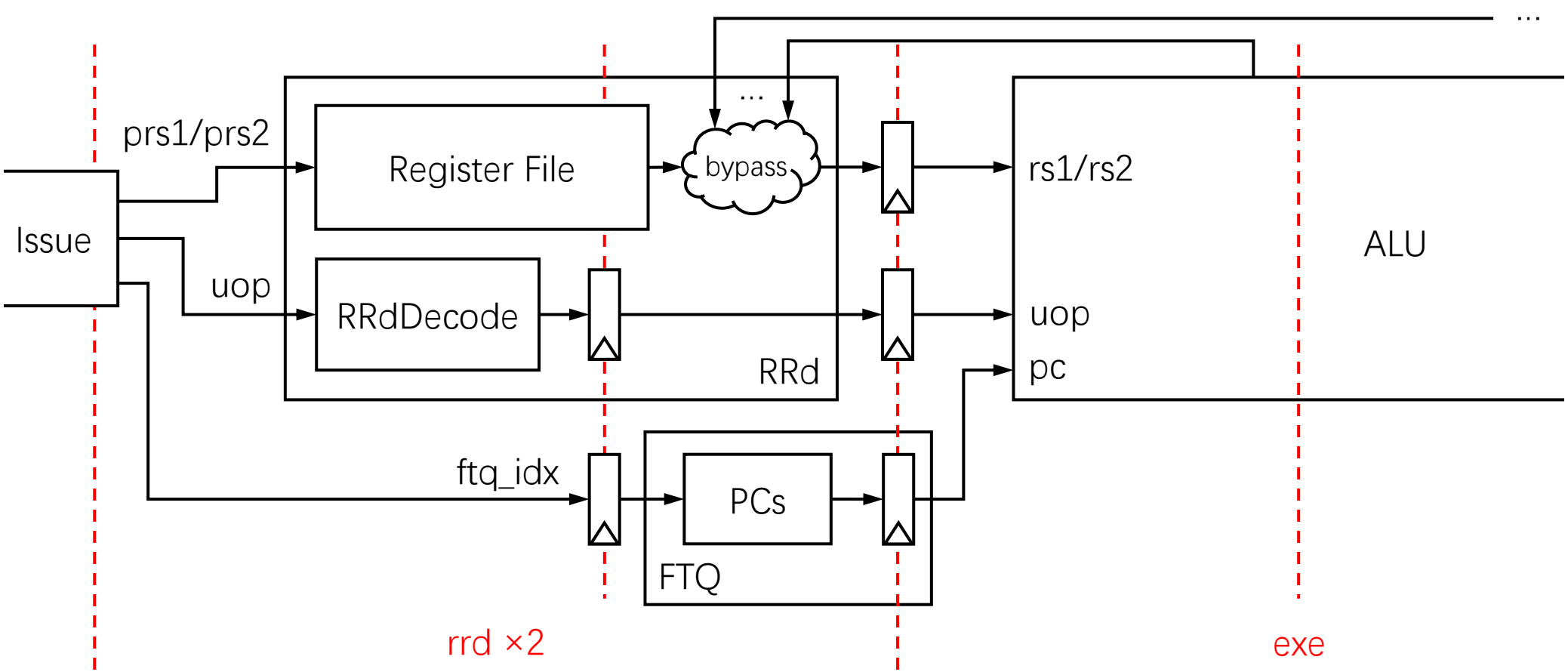
增加FTQ端口

现在和所有exe_unit都有连接



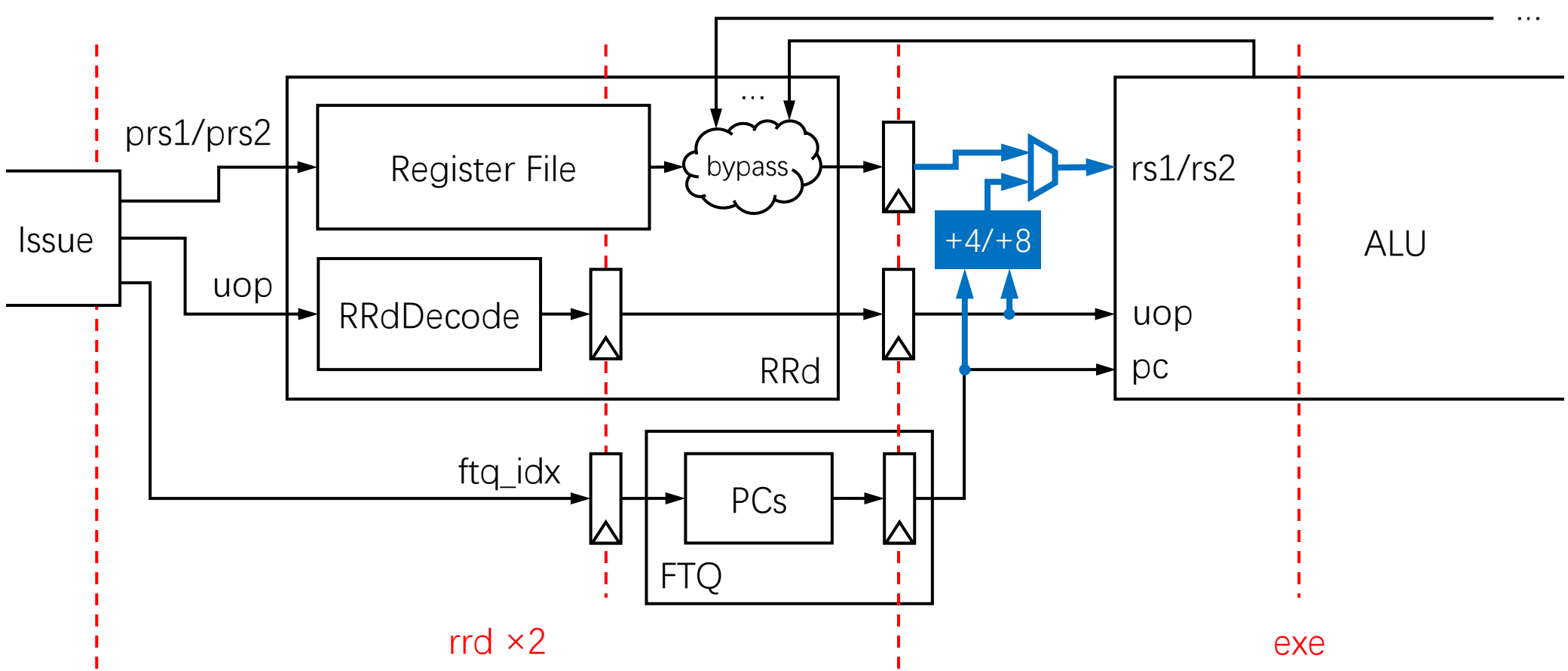
ALU rs1/rs2重连线

ALU的rs1/rs2本来直接来自寄存器堆



ALU rs1/rs2重连线

现在增加选择：若为r31，则重连线到PC



ALU rs1/rs2重连线

注：UniCore32的r31并非PC！

- 对于不同的指令和操作数，r31的含义如下

指令类型	rd/data	rs1	rs2	rss/rs3
算数	-	PC+4	PC+4	禁止
load	-	PC+4	禁止	-
store	PC+8	PC+4	禁止	-
swap	-	禁止	禁止	-

3.
写PC指令

额外执行JALR

在译码阶段检查指令写PC的情况

```
ADD rd, rs1, rs2, imm
```

SubDecode


```
uopSHIFT tmp, rs1, imm
uopADD   rd, rs2, tmp
```

- 检测是否有`rd == r31`的情况

额外执行JALR

在译码阶段检查指令写PC的情况

```
ADD rd, rs1, rs2, imm
```

SubDecode


```
uopSHIFT tmp, rs1, imm
uopADD   rd, rs2, tmp
uopJUMP  r31
```

- 若有，增加一条JUMP r31
- 前面的ADD先将目标地址存入r31，再由JUMP完成跳转

额外执行JALR

在译码阶段检查指令写PC的情况

限制

1. 原指令译码得到的微指令数不能**超过3条**
   - 否则没有位置添加JUMP
2. 微指令序列中不得对rd**先写后读**
   - 否则rd==r31时，读到的仍是PC+4/+8，而不是写进去的值

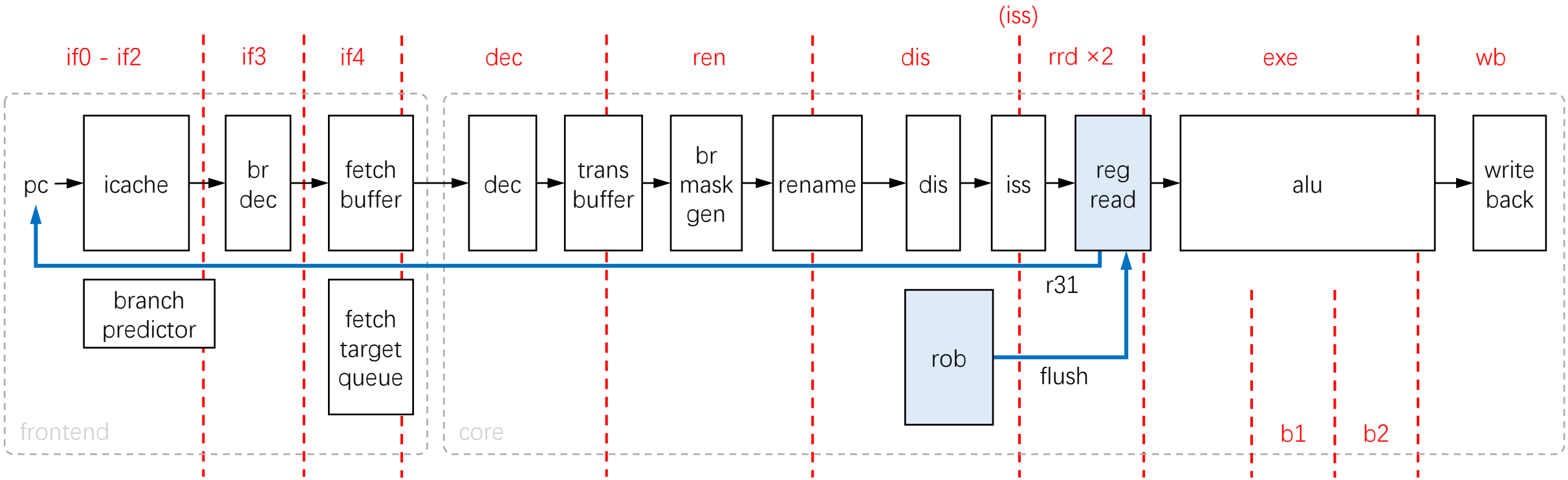
3. is_unique的指令无法拆成多条微指令，无法使用该方法

提交时flush

解决原子指令的写PC问题

- 目前只有原子指令`is_unique`，且原子指令会`flush_on_commit`
- 利用`flush_on_commit`性质
  - 让指令先把目标地址写入`r31`
  - `flush`时再读出的`r31`给前端重定向

提交时flush



在`rr.read_port(0)`上做一个选择器

- 在非`flush`状态时，`rr`给ALU读
- 在`flush`状态时，ALU一定是空的，`rr`可以放心用来读`r31`
- `flush`有2个周期，正好`rr`也有2个周期