

---

# 目 录

1 设计规格.....	2
2 概要设计.....	2
2.1 整体框图.....	3
2.2 接口列表.....	3
2.3 接口时序.....	3
3 详细设计.....	4
3.1 总线接口说明.....	4
3.2 IF stage.....	4
3.3 ID stage.....	6
3.4 ISSUE stage.....	7
3.5 EX stage.....	10
3.5.1 csr_ctrl.....	10
3.5.2 branch_unit.....	11
3.5.3 ALU 单元.....	11
3.5.4 load/store.....	12
3.5.5 mlt/div.....	15
3.6 COMMIT.....	17
3.7 握手处理.....	17
3.8 ctrl_flow.....	20
3.8.1 fence.....	20
3.8.2 branch.....	20
3.8.3 jump.....	20
3.9 异常处理.....	21
3.9.1 指令.....	21
3.9.2 中断.....	21
3.9.3 异常.....	21
3.10 csr.....	22
3.11 controller.....	23

---

## 1 设计规格

1. 仅支持 RV32I，暂不包含 exception 处理，csr 读写。静态分支预测，顺序单发射，乱序写回。先完成数据通路构建。
2. 添加中断异常处理、csr 读写逻辑。
3. 支持 k 扩展，替换为 AXI 总线。

## 2 概要设计

1. IF 包括 pc 生成逻辑，预解码判断是否为跳转指令，ras 提供函数返回地址。
2. ID 解码 32 位指令，目前包括 I、CSR 指令。
3. ISSUE 包括寄存器重命名 map、Scoreboard 逻辑，顺序发射，顺序执行，乱序写回 sbe，顺序提交。可解决 WAW 冲突，可实现 ex 计算结果转发。
4. EX 包括第一级 csr\_ctrl、branch\_unit、ALU，第二级 load/store。  
csr\_ctrl: 读 csrfile，取得 csr 索引的寄存器值；rs1 与 csr 运算；将 csr 值与运算结果写回 sbe，commit 时写入 regfile。包括一些异常处理寄存器读写。  
branch\_unit: 运算可在 ALU 完成。判断指令是否跳转，与前端传递的预测跳转方向对比，给出是否 mispredict 信号。给出跳转地址。  
ALU: I 指令运算。  
MLT/DIV: M 指令运算。  
load/store: 先在 ALU 中计算地址，然后访存。
5. WB 结果写回到 Scoreboard 中，支持结果转发。
6. commit 包括 regfiles，以及 flush 逻辑。

## 2.1 整体框图

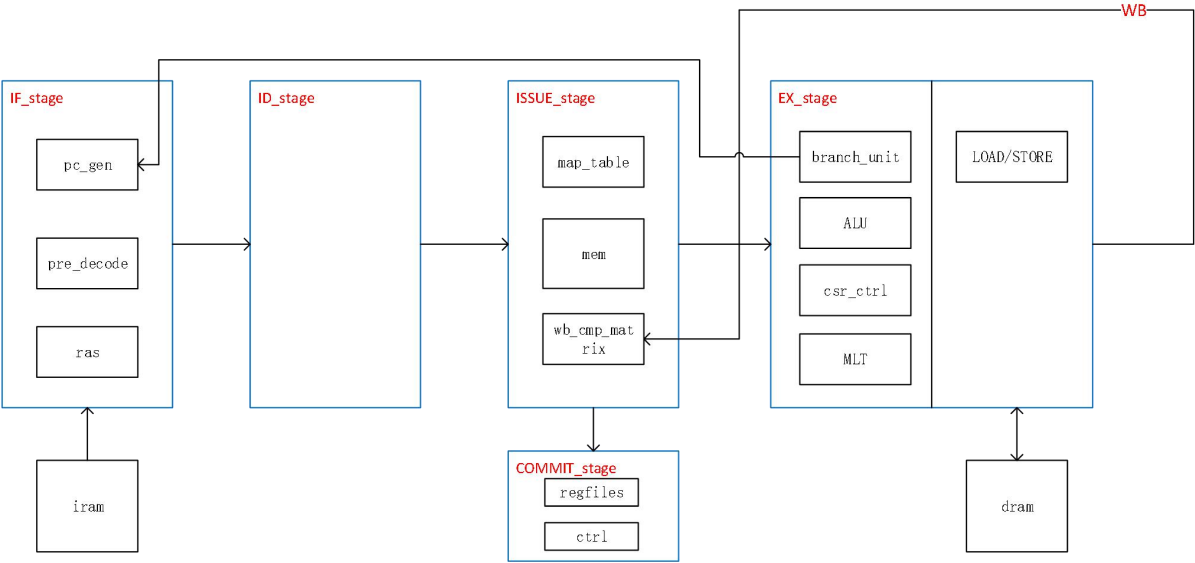


图 3.1 整体框图

## 2.2 接口列表

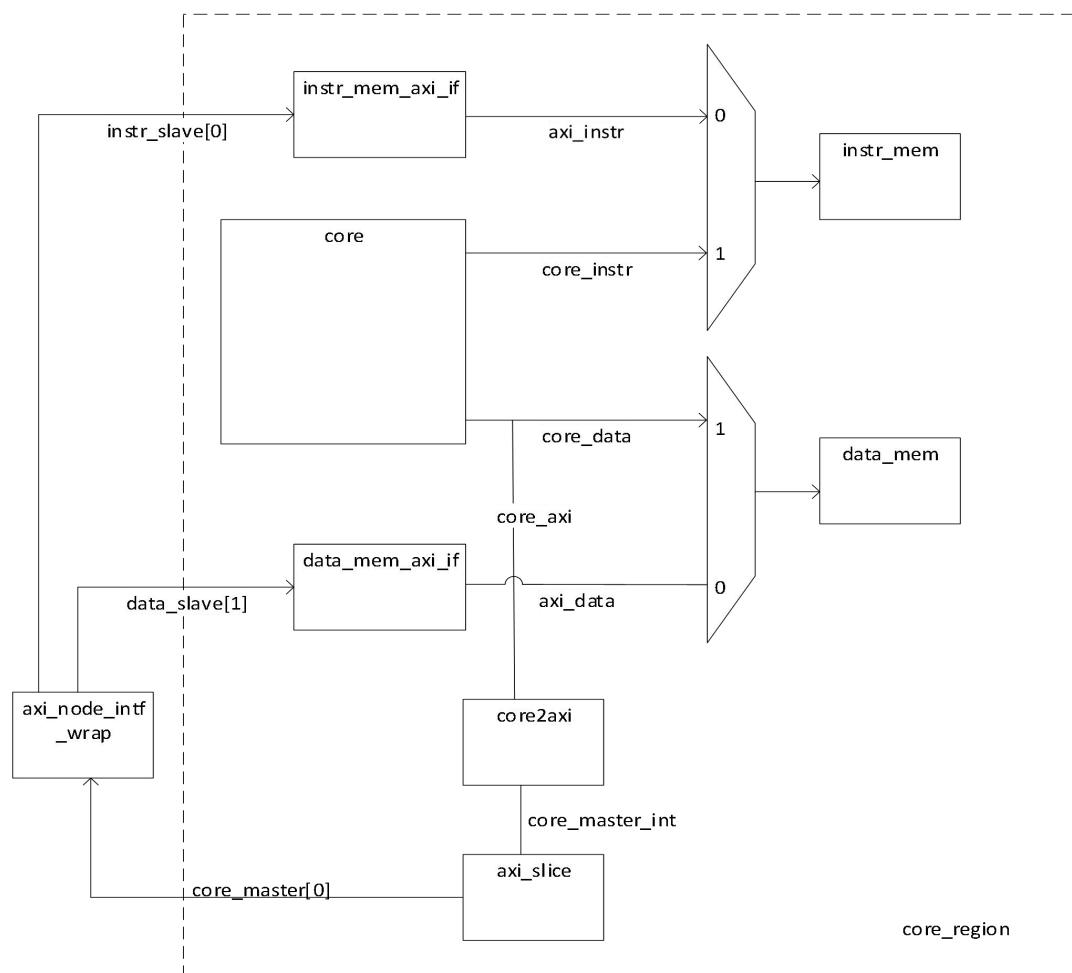
表 3.1 core 接口列表

信号	方向	位宽/类型	描述
clk_i			
rst_ni			
instr_boot_addr_i			
instr			iram 接口
data			dram 接口

## 2.3 接口时序

## 3 详细设计

### 3.1 总线接口说明



### 3.2 IF stage

- PC 产生逻辑

PC 值来源:

复位值: `cpu_reset_addr`

分支不跳转: 顺序取址, `ex` 计算发现错误后刷新。

流水线冲刷:

分支预测错误时, `pc` 为 `alu` 计算的跳转地址;

---

发生中断、异常时，pc 为 controller 给出的 mtvec 地址；

顺序取址：pc+4

stall 情况下：pc<=pc

- 取指令

ram 接口读取指令，ram 接口可参考如下

instr\_ram\_wrap

```
    #(
        .RAM_SIZE    ( INSTR_RAM_SIZE ),
        .DATA_WIDTH  ( AXI_DATA_WIDTH )
    )
    instr_mem
    (
        .clk          ( clk          ),
        .rst_n        ( rst_n        ),
        .en_i         ( instr_mem_en  ),
        .addr_i       ( instr_mem_addr ),
        .wdata_i      ( instr_mem_wdata ),
        .rdata_o      ( instr_mem_rdata ),
        .we_i         ( instr_mem_we  ),
        .be_i         ( instr_mem_be  ),
        //.bypass_en_i ( testmode_i    )
    );
```

- stall

中断导致 flush，清除所有没 commit 的指令，亦可清除 stall 的指令；

mispredict 导致的 flush，清除 mispredict 之后的所有指令，stall 指令在 mispredict 之后，亦可清除 stall 指令；

id exception 导致的 flush，清除 if、id 的指令，也可清除 stall，即产生 id\_exception 的指令；

- RAS 逻辑

深度可配，超过深度后，最先 push 的地址溢出，当跳出此条地址对应的函数时，pop

出数据的 vld 为 0，需要等待 ex 计算返回地址。

jal 的 rd 地址为 x1、x5 时，为函数调用 call，需要压栈 PC+4；

jalr 的 rd 地址为 x1、x5 时，不管 rs1 是不是 x1/x5，不管 rd 是否与 rs1 相等，为函数调用 call，需要压栈 PC+4；（表 3/4/5 行）

jalr 的 rs1 地址为 x1、x5 时，且  $rs1 \neq rd$ ，为函数返回 return，需要弹栈；（表 4 行）

若是 call，需要 stall，等待 ex 计算跳转地址；若是 return 并且 ras\_pop\_addr.vld，不需要 stall，使用 pop 地址；若是 return&~vld，需要 stall，等待 ex 计算跳转地址。

rd	rs1	rs1=rd	RAS action
<i>!link</i>	<i>!link</i>	-	none
<i>!link</i>	<i>link</i>	-	pop
<i>link</i>	<i>!link</i>	-	push
<i>link</i>	<i>link</i>	0	pop, then push
<i>link</i>	<i>link</i>	1	push

jalr 根据此表选择栈操作

jal	jal rd,imm[20:1]	x(rd)=PC+4; PC+=sext({imm[20:1],1'b0})	该指令执行两步操作：1.将下一条指令的 PC 值写到寄存器 rd；2.将立即数乘以 2，与 PC 相加得到最终的跳转目标地址，可以跳转前后 1Mb 的地址区间。
jalr	jal rd,rs1,imm[11:0]	x(rd)=PC+4; PC=(x(rs1)+sext(imm[20:0])) & (~'h1)	该指令执行两步操作：1.将下一条指令的 PC 值写入寄存器 rd；2.寄存器 rs1 值和有符号立即数相加作为最终的跳转目标地址，且地址的 bit0 位需要置零。

### 3.3 ID stage

#### ● 32 位指令的译码

case (opcode)

case (func3)

输出给后级 instruction\_o 为 instruction\_entry\_t 类型

```
typedef struct packed {
```

```
    logic                                valid;
```

```
    logic [31:0]                        pc;
```

```

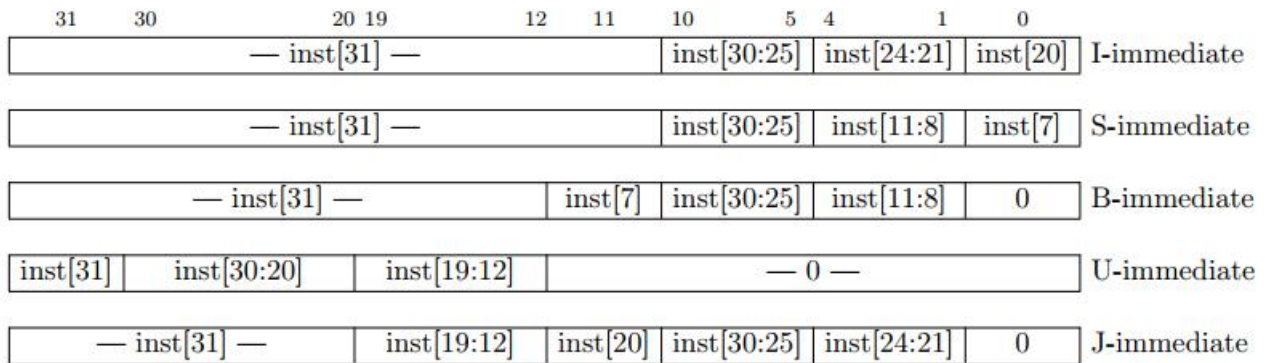
fu_t                                fu;
fu_op_t                             op;
logic [REG_ADDR_SIZE-1:0]          rs1;
logic [REG_ADDR_SIZE-1:0]          rs2;
logic [REG_ADDR_SIZE-1:0]          rd;
logic [31:0]                        imm;
logic                               csr_instr;
logic                               instr_ecall ;
logic                               instr_ebreak;
logic                               instr_mret  ;

} instruction_entry_t;

```

- 解码保留 exception、ecall、ebreak 等端口，先不连接；
- immssel:

对 I\S\B\U\J 型指令的 imm 组合输出。



### 3.4 ISSUE stage

带转发机制的顺序发射，乱序写回，顺序提交，与 ariane 类似。增加 map 表，解决 WAW 冲突。RAW 可使用转发机制或 stall 流水线解决，WAR 在顺序发射下自然保证。

- map\_table

用于跟踪寄存器的重命名。在 WAW 情况下，可不用等待上条指令写回并 commit 后，再进行本条指令的 issue，通过重命名 rd，然后按序 issue 即可。在 RAW 情况下，后续指令的 rs 与 map 表比较，使用重命名后的寄存器作为当前指令的 rs，等待 rs 值写回后再按序发射。

在指令写入 sbe 时更新 map 表，每个寄存器的重命名可被覆盖，不影响后续操作，只跟踪最近的 rename。在 commit 时，若被覆盖的重命名寄存器被提交，后续相关 rs 在

其 wb 时已写回 sbe，没有影响。在 commit 时，若有新的 map 写入，更新 map；若没有，需要判断后续指令是否有使用此寄存器作为 rd 重命名，若有相同则不清除 map，没有相同的 rd 则清除对应寄存的 map。

不在 map 中的 rs，数据一定在 regfile 中，在 map 中的 rs 已写入 sbe 中等待写回 rd。

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	……	X31
rename			E0											
vld			1											

map\_table

## ● mem

作为一个缓存带，从这里判断 WAW、RAW 关系、是否可 issue、是否可 commit。定义结构如下。

```
struct packed {
    logic                                writed    ;
    logic                                issued    ;
    scoreboard_entry_t                  sbe        ;
} mem_q [MEM_DEEP-1:0], mem_d [MEM_DEEP-1:0];
```

```
typedef struct packed {
    logic [riscv::VLEN-1:0]             pc;
    logic [ADDR_BITS-1:0]               trans_id;
    fu_t                                 fu;
    fu_op                                op;
    logic [REG_ADDR_SIZE:0]              rs1;
    logic [REG_DATA_WIDTH-1:0]           rs1_data;
    logic                                rs1_rdy;
    logic [REG_ADDR_SIZE:0]              rs2;
    logic [REG_DATA_WIDTH-1:0]           rs2_data;
    logic                                rs2_rdy;
    logic [REG_ADDR_SIZE-1:0]            rd;
    logic [REG_DATA_WIDTH-1:0]           result;
    logic                                rd_vld;
    logic                                csr_instr;
```



---

```

        logic                                instr_ecall ;
        logic                                instr_ebreak;
        logic                                instr_mret  ;
    } scoreboard_entry_t;

```

mem 主要的 4 类操作:

#### 1. 指令写入 mem

mem 非 full 可写入, 写入 mem 时, mem.sbe.wrtied 置 1。

在此需要判断 rs\_rdy 信号:

① 读 map\_table, 若没有 rename 则 rs\_rdy=1, 并从 regfile 读数据写入 rs\_data; 若有 rename, rs 可从 wb 口获取, 或从以写入 sbe 的项中获取。

② 从 wb 端口获取。wb 端口有写回, 则 rs\_rdy=1, 并从 wb 写入 rs\_data;

③ 从 sbe 中获取。比较 sbe 中未 commit 项, 若其 rd\_vld, 且 trans\_id 与当前 wptr\_q 所指的项中的 rs 相等, 则 rs\_rdy=1, rs\_data = mem\_q[i].sbe.result。

#### 2. 从 writeback 端口写回数据

① 使用 wb\_cmp\_matrix, 比对 sbe 中所有 rs, 并写入。

② 匹配 trans\_id 项, 写回 result, 将 rd\_rdy 置 1。

#### 3. commit 写入 csrfile、regfile

commit=1; wrtied=0; issued=0; sbe=0;

数据必须先写回 sbe, 然后 commit 写入 regfile。不需要刷新 mem 内部与此 rd 相关的 rename 寄存器名, 因为在 wb 时已经写回 rs\_data, 不再关心 rename。

#### 4. issue 到 EX\_stage

一般指令 rs\_rdy 后可发射; csr 指令需要 ~csr\_inflight 后可发射;

mem\_d[issue\_ptr\_q].issued=1 时可发射, 同时也作为指针加 1 的条件;

待 issue 的项需要判断 rs\_rdy, rs\_rdy 在写入 sbe 时与 wb 比对过一次, 在 issue\_ptr\_q 指向时, 需要与 sbe 中已写入的 rd\_vld 再比对一次。

mem\_d[issue\_ptr\_q].issued=1 时分以下两种情况:

① 若此时 wptr\_q 与 issue\_ptr\_q 相同, 表示可直接发射从 decode 来的指令, 并同时写入到 sbe 中。

② wb 口匹配到 issue\_ptr\_q 这项的 rs, 使 mem\_d[issue\_ptr\_q].issued=1, 则可发射 wb\_data\_i+sbe\_data\_q, 同时写入 sbe 中。issue 时数据来源可从 sbe 中或 wb 端口, 由于 wb\_comp\_matrix 逻辑已完成匹配和对 mem\_d 的赋值, 直接使用 mem\_d[issue\_ptr\_q].sbe.rs\_data 作为发射数据。

## ● 指针说明

### 1. issue\_ptr

mem\_d[issue\_ptr\_q].issued=1 时可发射，发射同时 issue\_ptr\_d+1。

### 2. commit\_ptr

commit\_ack\_i && mem\_q[commit\_ptr\_q].sbe.rd\_rdy 时可提交，提交时指针加一。

即 mem\_d[commit\_ptr\_q].commit = 1'b1 时 commit\_ptr\_d+1。

### 3. wptr

写入 sbe 时指针加一。

### 4. cnt

cnt\_d = cnt\_q + writeen - readen

可理解为：

mem\_d[wptr\_q].writed & ~full = write\_en

commit = read\_en

## 3.5 EX stage

### 3.5.1 csr\_ctrl

读 csrfile，取得 csr 索引的寄存器值；rs1 与 csr 运算；将 csr 值与运算结果写回 sbe，commit 时写入 regfile。ebreak/ecall/mret 都需要写入或读取 csrfile，也可在此处处理。

csr	rs1	011	rd	1110011	csrrc
csr	zimm	101	rd	1110011	csrrwi

信号	方向	位宽 / 类型	描述
csr_addr_i			issue 发过来的 csr 地址，即是写入地址也是读出地址，先读后写。
rs1_data_i			issue 发过来的 rs1 地址内的数据
wb_csr_data_o			写回到 issue 的 sbe
csr_addr_o			csrfile 读地址，写地址 csr_addr_o = csr_addr_i
csr_data_i			从 csrfile 读数据

			wb_csr_data_o = csr_data_i
csr_data_o			向 csrfile 写数据, rs1_data_i 与 csr_data_i 运算后的数据
异常处理接口			读取异常原因, 跳转地址等

### 3.5.2 branch\_unit

比对 branch 指令源操作数的大小在 alu 完成。在此模块内判断指令是否跳转, 与前端传递的预测跳转方向对比, 给出是否 mispredict 信号。计算跳转地址。

jar 和 jalr 的地址计算也在此模块进行。PC、imm、rs1 进行地址计算。rd=pc+4 在 alu 内完成。

信号	方向	位宽/类型	描述
rs1_data_i			比较源操作数, 判断是否跳转
rs2_data_i			比较源操作数, 判断是否跳转
predict_jump_i			在 fetch 预测跳转指示, 判断预测与实际是否一致
pc_i			用于计算跳转地址
imm_i			用于计算跳转地址
branch_unit_vld_o			计算地址完成提示, 不表示预测是否正确
branch_unit_addr_o			计算地址输出, 若预测跳转 (向后跳), 实际不跳转, 给出 PC+4 若预测不跳转, 实际跳转, 给出运算地址值 对于 jar/jalr 直接计算跳转地址
wb_j_data_o			写回给 sbc 的 rd, rd=pc+4

### 3.5.3 ALU 单元

定义 11 类运算。

	==	!=	<	>=	+	-	&		^	<<	>>		

前六种运算都可用下面的 alu\_sub 来计算。不论 op\_a、op\_b 是否是有符号数都可以用 sub

来计算。若指令是无符号数计算，在 `op_a` 和 `op_b` 最高位上增加一符号位，由符号位判断两个数值大小。若指令是有符号计算，忽略增加的符号位，使用 `op_a` 和 `op_b` 计算后本身的符号位判断大小。

```
logic [33:0] op_b_neg;
```

```
logic [33:0] op_a_ext;
```

```
logic [32:0] result_t;
```

```
op_b_neg = {1'b0, op_b, 1'b0} ^ 34'h3fffffff; //增加减数符号位为 1，并对 op_b 最低位+1
```

```
op_a_ext = {1'b0, op_a, 1'b1}; //扩展被减数符号位为 0，对 op_b 最低位+1
```

```
result = op_a_ext + op_b_neg;
```

```
is_equal = ~|result[32:1];
```

```
is_less_than = result[33]; //无符号数用增加的符号位判断
```

```
is_less_than = result[32]; //有符号数若两个数同为正数或同为负数，用本身的符号位判断
```

```
is_less_than = op_a[31]; //有符号数若两个数正负相反，用第一个数的符号判断；
```

注意：>>>被操作数默认是无符号数，只会高位补 0，需要转成有符号数才可以高位补 1。

```
function automatic logic [31:0] alu_sra (logic [31:0] op_a, logic [4:0] op_b);
```

```
    alu_sra = $signed(op_a) >>> op_b;
```

```
endfunction
```

根据 `op` 类型指定运算的源操作数，再进行运算。

计算 `rd` 的值。

信号	方向	位宽/类型	描述
<code>fu_data_i</code>		<code>fu_data_t</code>	
<code>wb_alu_o</code>			写回 sbe 的 rd
<code>ls_addr_o</code>			输出给 load/store 的访问地址

### 3.5.4 load/store

参考 `ibex` 的 `load_store` 传输协议。

The protocol that is used by the LSU to communicate with a memory works as follows:

1. The LSU provides a valid address in `data_addr_o` and sets `data_req_o` high. In the case of a store, the LSU also sets `data_we_o` high and configures `data_be_o` and `data_wdata_o`. The memory then answers with a `data_gnt_i` set high as soon as it is ready to serve the request. This may happen in the same cycle as the request was sent or any number of cycles later.
2. After receiving a grant, the address may be changed in the next cycle by the LSU. In addition, the `data_wdata_o`, `data_we_o` and `data_be_o` signals may be changed as it is assumed that the memory has already processed and stored that information.
3. The memory answers with a `data_rvalid_i` set high for exactly one cycle to signal the response from the bus or the memory using `data_err_i` and `data_rdata_i` (during the very same cycle). This may happen one or more cycles after the grant has been received. If `data_err_i` is low, the request could successfully be handled at the destination and in the case of a load, `data_rdata_i` contains valid data. If `data_err_i` is high, an error occurred in the memory system and the core will raise an exception.
4. When multiple granted requests are outstanding, it is assumed that the memory requests will be kept in-order and one `data_rvalid_i` will be signalled for each of them, in the order they were issued.

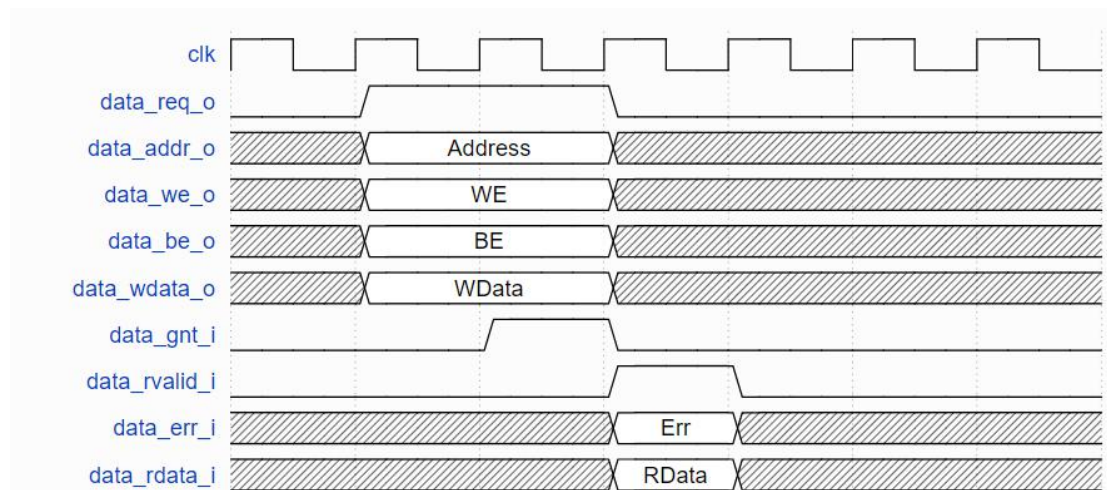


Figure 6 Basic Memory Transaction

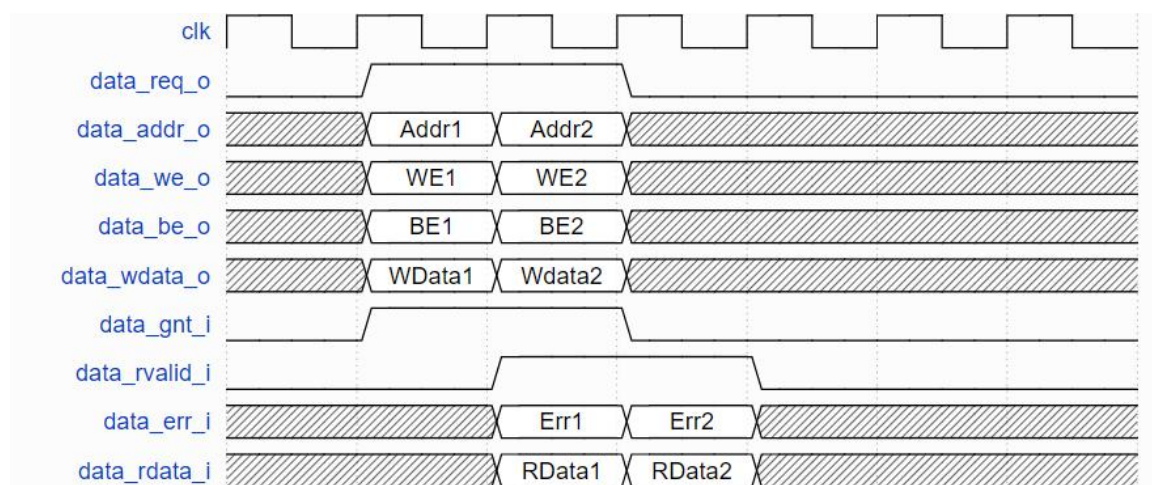
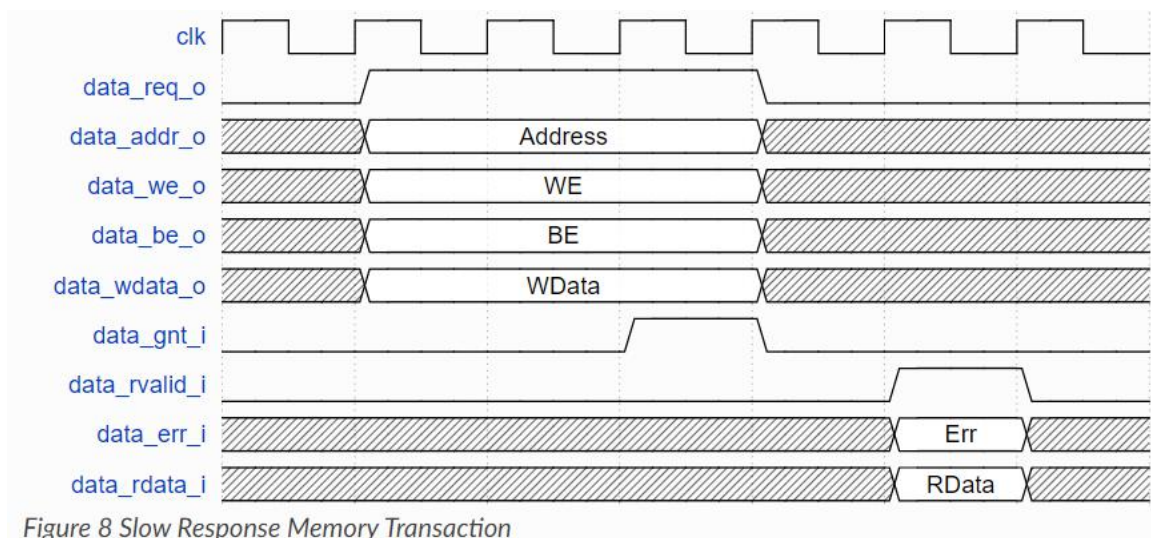


Figure 7 Back-to-back Memory Transaction

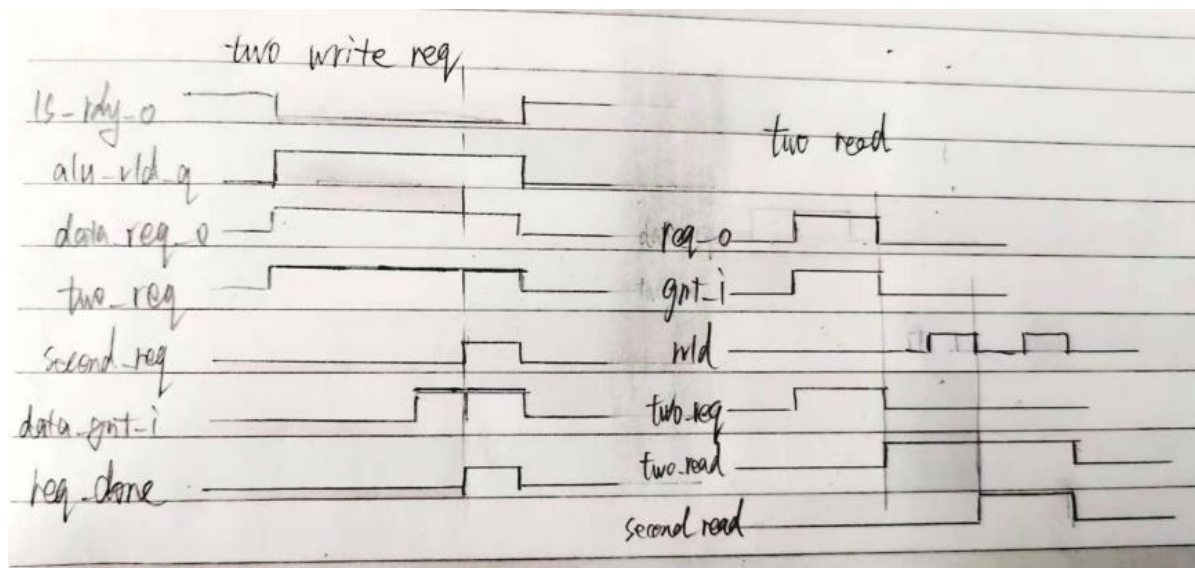


指令先经过 alu 进行地址计算，然后访问 DMEM，至少需要 2cycle。

load\_store.sv 不支持非 4 字节对齐的地址读写，load\_store\_1.sv 支持非对齐读写。根据读写数据长度（operator）和地址偏移（data\_offset）来拆分成 2 次读写，或不用拆分。

由于要支持上述的读写接口协议，req 与 load 分成两个 channel。req channel 向外发送 be\_o、addr\_o，以及组合两次/一次写数据，若为两次发送需要保存 wdata\_2。load channel 控制写回，若 load 读回数据在几个 cycle 后，需要保存 operator 和 data\_offset 信息，以便读回两次/一次数据时，根据这两个信息组合成需要的 wb 数值。若为 2 次读回，读回第一个数据需要保存，在读回第二个数据时组合成 wb\_data 写回 sbe。

下图左侧为拆分两次 store 的波形，two\_req、second\_req 是提示两次写的标志信号，根据这两个信号，分配第一次和第二次的 store 数据。右侧是拆分两次读且几个 cycle 后读回数据的情况，根据 two\_read、second\_read 两个信号确定保存 operator 和 data\_offset 和 trans\_id 的时长，并确定写回 sbe 时间为第二个数据读回时间（second\_read & rvld）。



信号	方向	位宽/类型	描述
ram 接口			

load 指令测试时的错误分析：

lbu 测试包需要预先向 instr\_ram 和 data\_ram 分别载入数据。而之前将所有二进制文件的内容全部载入到了 instr\_ram，导致 lbu 从 data\_ram 中读数据时无数据可用。

临时方案，定义一个 ram，包含两个读端口和一个写端口，即把 instr\_ram 和 data\_ram 合二为一，二进制文件全部载入这个 ram，lbu 从 ram 中也可读到预先载入的数据。另外一些 load 指令还需要从 instr 区域读数据，也可满足需求。

替换为 AXI 接口，应该如何测试？

替换 pulpino 的接口结构，需要分别向 instr\_mem 和 data\_mem 中载入二进制文件，地址 0x00002000 之前的在 instr\_mem，后面在 data\_mem；需要区分 L/S 指令地址对应的是哪个空间， $\text{addr}[31:13] = 19'b0$  则地址是在 0x00002000 之前，属于 instr\_mem； $\text{addr}[31:13] \neq 19'b1$  属于 data\_mem 范围以外的地址；

### 3.5.5 mlt/div

信号	方向	位宽/类型	描述
fu_data_i		fu_data_t	
wb_alu_o			写回 sbe 的 rd
mult_free_o			没占用指示，允许 issue 发送



除法多周期指令。

DIVU 32'bFFFF, 若为有符号数, 此数表示-1

Condition	Dividend	Divisor	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	$x$	0	$2^L - 1$	$x$	-1	$x$
Overflow (signed only)	$-2^{L-1}$	-1	-	-	$-2^{L-1}$	0

考虑 a/b 的情况:

	div	rem	divu	remu
$ a  <  b $	0	a	0	a
$b = 0$	32'bFFFFFFFF	a	32'bFFFFFFFF	a
$-2^{31}/-1$	32'b80000000	0	----	----

串行除法器:

判断 shift\_amount 的依据

1. lzc 数。
2. a-b 是正数时就移动 lzc, 负数时移动 lzc+1。

3. 剩余可移动数目 shift\_cnt, 若上述计算的 shift\_amount > shift\_cnt, 则移动 shift\_cnt, 若小于, 则移动 shift\_cnt。

加法树乘法器:



```

23 function [7:0] mut8_1;
24 input [7:0] operand;
25 input sel;
26
27 begin
28     mut8_1 = sel ? operand : 8'b0000_0000;
29 end
30 endfunction
31
32 //操作数b各位与操作数a相乘
33 always @(posedge clk)
34 begin
35     temp7 = mut8_1(a,b[0]);
36     temp6 = (mut8_1(a,b[1]))<<1;
37     temp5 = (mut8_1(a,b[2]))<<2;
38     temp4 = (mut8_1(a,b[3]))<<3;
39     temp3 = (mut8_1(a,b[4]))<<4;
40     temp2 = (mut8_1(a,b[5]))<<5;
41     temp1 = (mut8_1(a,b[6]))<<6;
42     temp0 = (mut8_1(a,b[7]))<<7;
43 end
44
45 //加法树运算
46 assign out1 = temp0 + temp1;
47 assign out2 = temp2 + temp3;
48 assign out3 = temp4 + temp5;
49 assign out4 = temp6 + temp7;
50 assign c1 = out1 + out2;
51 assign c1 = out3 + out4;
52 assign out = c1 + c2;

```

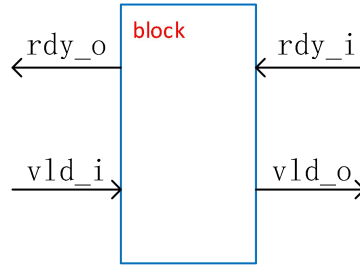
### 3.6 COMMIT

包含 regfiles 和 ctrl。

ctrl 包含 mispredict 以及 exception 导致的 flush 逻辑。

### 3.7 握手处理

issue 内含有缓存，握手可使用 ready 等待 valid 方式，两个信号同时为 1 时，传输数据有效。当下游~ready 时，反压流水线，上游不能够输出 valid，只可在其 stage 保持 1 拍的数据。



### 1. issue

$vld\_i \ \& \ rdy\_o$	可向 mem 写入 data。
$vld\_o = rdy\_i \ \& \ mem\_d[issue\_ptr\_q].issued$	下游 rdy 且 issue_ptr 所指的选项中的所有 rs 都 rdy 可向下游发送有效数据。
$rdy\_o = mem\_not\_full$	缓存非满即可接受上游数据。

### 2. decode

$vld\_i \ \& \ rdy\_o$	可写入 data_d
$vld\_o = rdy\_i \ \& \ vld\_q$	$vld\_q \leq vld\_i \ \& \ rdy\_o$ 下游 rdy 且本级数据 vld，可向下游发送有效数据
$rdy\_o = rdy\_i$	decode 阶段只有 1 拍，目前没有 stall 的逻辑，所以只要下游 rdy 即可接收上游数据。

### 3. fetch

$vld\_i \ \& \ rdy\_o$	可进行预译码、写入 npc_d 等
$vld\_o = rdy\_i \ \& \ vld\_q$	$vld\_q \leq vld\_i \ \& \ rdy\_o$ 下游 rdy 且本级数据 vld，可向下游发送有效数据
$rdy\_o = (stall)? \ 1'b0 : rdy\_i$	rdy_o 表示向 iram 取指令请求，stall 状态下不可取指令。
if(flush) $vld\_q \leq 0;$ else if (input_hsk) $vld\_q \leq 1;$ else if(output_hsk) $vld\_q \leq 0;$	

#### 4. ex

ALU、csr\_ctrl 是一个 cycle 完成计算的模块，没有反压逻辑；load\_store、MULT/DIV 可能会多 cycle 完成，需要握手逻辑。ex\_stage 的 rdy\_o 逻辑，根据 fu 选择输出。

load_store	<pre> always_ff @(posedge clk_i or negedge rst_ni) begin     if (!rst_ni)         ls_rdy_o &lt;= 1'b1;     else if (flush_ex_i)         ls_rdy_o &lt;= 1'b1;     else if ((data_gnt_i &amp;&amp; ~two_req)                 (data_gnt_i &amp;&amp; second_req))         ls_rdy_o &lt;= 1'b1;     else if (data_req_o)         ls_rdy_o &lt;= 1'b0; end </pre>	<p>一次请求、两次请求接收到 dram gnt 响应后可解除 load_store 占用。</p>
	<pre> always_comb begin     if ((data_rvalid_i &amp;&amp; ~data_err_i &amp;&amp; ~two_read)            (data_rvalid_i &amp;&amp; ~data_err_i &amp;&amp; second_read_back)) begin         ls_wb_port_o.wb_data = rdata ;         ls_wb_port_o.wb_vld = 1'b1 ;         ls_wb_port_o.trans_id = (read_trans_id == '0)? trans_id_q : read_trans_id ;     end else if ((data_we_o &amp;&amp; data_gnt_i &amp;&amp; ~two_req)                     (data_we_o &amp;&amp; data_gnt_i &amp;&amp; second_req)) begin         ls_wb_port_o.wb_data = 32'b0 ;         ls_wb_port_o.wb_vld = 1'b1 ;         ls_wb_port_o.trans_id = trans_id_q ;     end else begin         ls_wb_port_o.wb_data = 32'b0;         ls_wb_port_o.wb_vld = 1'b0 ;         ls_wb_port_o.trans_id = 5'b0 ;     end end </pre>	<p>写回时没有反压，一定可以写回到 sbe 中。完成写入或读回后，可写回 sbe。</p>

	end	
	end	
MULT /DIV		

### 5. commit

无反压逻辑，可直接写入 regfiles。

## 3.8 ctrl\_flow

### 3.8.1 fence

fence\_i 指令简单理解是保证 DRAM 或外设地址先写后读的指令。对于乱序执行的结构需要这个指令，在写、读相同地址时，一定先写入后读取，保证读取到最新的数据。由于 dram 读写在 ex\_stage 完成，fence\_i 前发射的指令必须全部写回后，才能发射 fence\_i 后的指令，即当发射 fence\_i 指令时，需要 stall 住 issue，等待前序指令全部 commit 后（即 commit\_ptr\_d==issue\_ptr\_q），再继续发射。由于 issue 中有缓存，可以不用 stall if 和 id，直到 full 再反压。

发射 fence 指令后，在 exstage 什么都不做，按 alu 的拍数和规则写回 sbe，浪费 wb 和 commit 两拍，实际没做任何事。

本设计将 fence、fence\_i 都解读为上述含义。

### 3.8.2 branch

遇到 branch 时不跳转，顺序取址  $PC = PC + 4$ ；计算分支实际是否跳转和跳转地址在 EX\_stage，有两种方式处理预测错误：

1. 在 ex 计算完成后，立即刷新 IF 并从正确跳转地址取指令，刷新 ID，清除 ISSUE 内 issue\_ptr 后的条目（wptr=issue\_ptr）。代价是 branch 后 3 条指令。适合顺序发射。
2. ex 计算完成后写回 Scoreboard，在 commit 时立即刷新 IF 并从正确跳转地址取指令，刷新 ID，清除 ISSUE 内整个 sbe（mispredict 的指令已经 commit，commit\_ptr\_q 后的指令全部不需要，相当于清空整个 sbe），刷新 ex。代价是 branch 后 5 条指令。

### 3.8.3 jump

jal/jalr 情况下，如果是普通的跳转或者函数调用 call，需要 stall 等待 ex 计算跳转地

---

址。如果是函数返回 `return`，不需要 `stall`，从 `ras` 弹出地址。

## 3.9 异常处理

中断、`ebreak`、`ecall` 更新 `mepc=pc+4`

异常更新 `mepc=pc`

### 3.9.1 指令

`ebreak`、`ecall` 在 `decode` 阶段解码到指令，冲刷 `if`、`id`，从 `csr` 中读 `mtvec` 得到 `pc`，给入到 `if` 产生下个 `pc`。

更新 `csr` 中 `mcause` 为 `{0,3/11}`;

更新 `mepc` 为 `pc+4`;

`mtval` 中写入 `ebreak` 地址或 0;

`mstatus` 中，`MPIE<=MIE`，`MIE=0`，`MPP=privileged`;（初始值 `PRIV_LVL_M`）

`mret` 在 `decode` 阶段解码到指令，冲刷 `if`、`id`，从 `csr` 中读 `mepc` 得到 `pc`，给入到 `if` 产生下个 `pc`。`mstatus` 中，`MIE<=MPIE`，`MPIE=1`，当前特权模式转为 `MPP` 中的保存值，只有 `M` 模式 `MPP=11`;

### 3.9.2 中断

当顶层出现 `soft_irq`\`timer_irq`\`ex_irq` 时，冲刷所有流水级，写入对应的 `mip`、`mie` 寄存器，硬件每拍读这两个寄存器，若检测到有中断：

`npc_d = mtvec`;

更新 `csr` 中 `mcause` 为 `{1,3/7/11}`;

更新 `mepc` 为下一条没有执行的指令 `PC`，`instr_raddr_o+4`;

`mtval` 中写入 0;

`mstatus` 中，`MPIE<=MIE`，`MIE=0`，`MPP=privileged`;（初始值 `PRIV_LVL_M`）

优先级最高，发生中断时，应当 `flush` 所有未 `commit` 的逻辑（即清空 `sbe`，所有 `ptr=0`）。若同时发生了 `exception`，对应的指令已经被 `flush`，不需要处理。

### 3.9.3 异常

目前只做一条，非法指令异常。

`id` 阶段解码到非法指令，冲刷 `if`、`id`，从 `csr` 中读 `mtvec` 得到 `pc`，给入到 `if` 产生下

个 pc。

npc\_d = mtvec; :

更新 csr 中 mcause 为 {0,2};

更新 mepc 为当前没被正确执行的指令, id\_pc;

mtval 中写入非法指令码;

mstatus 中, MPIE<=MIE, MIE=0, MPP=privileged; (初始值 PRIV\_LVL\_M)

在 ex\_stage 产生的异常, 由于 sbe 无法保证顺序, 只能在 commit 时响应异常, 并清空 sbe。

### 3.10 csr

csr 指令在 issue 阶段可从 regfile/sbe/wb 读取 rs1。判断指令是否可 issue, 需要先判断前面未 commit 的指令中是否有 csr 指令, 若有则说明 csrfile 还有未完成的读写操作, 不可发射; 若前面没有 csr 指令则此指令可发射。

在 ex 中, 读取 csr 寄存器, 并计算 rs1 与 csr 运算结果, 将 csr 值与运算结果一同写回 sbe, 然后在 commit 时将 csr 值写入 csrfile, 同时将 rd 写入 regfile。

csr	rs1	011	rd	1110011	csrrc
csr	zimm	101	rd	1110011	csrrwi

ecall\ebreak\mret 需要读取或写 csr 寄存器, 并使 PC 跳转。在 id\_stage 当拍产生 flush, 当拍给出跳转地址, 并在下拍写入 csrfile 对应寄存器。

ecall、ebreak 跳转到 mtvec, mret 跳转到 mepc。

目前实现了异常处理和退出相关的寄存器。

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			

0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcouneren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Counter/Timers			
0xB00	MRW	mcycle	Machine cycle counter.
0xB02	MRW	minstret	Machine instructions-retired counter.
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter.
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter.
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB80	MRW	mcycleh	Upper 32 bits of mcycle, RV32I only.
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32I only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32I only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32I only.
0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31, RV32I only.

### 3.11 controller

flush 优先级:

interrupt 外部中断优先级最高。其他指令产生的 exception 按照指令执行的顺序，先进入的指令，优先响应其异常。即流水线前后级若同时发生 exception，优先响应最后级的指令异常。若 mispredict 和 ex\_exception 同时发生时，只可能是产生 ex\_exception 的指令优先于 mispredict 指令执行，即要先响应 ex\_exception 的指令。顺序如下：

interrupt

---

ex\_exception  
mispredict  
id\_exception | mret  
if\_exception

flush\_issue 表示清除整个 sbe, 所有 ptr 指向 0。interrupt 时清除掉所有未 commit 的指令。导致 exception 或 mispredict 的指令, 由于 issue 内无法做到使 wptr=iss\_ptr 的同时, 清除掉对应 sbe 中的其他项, 以及 map\_table 没有备份 commit 时的映射情况, 在现有代码量下, 只能在 commit 时响应异常, flush 清除掉所有未 commit 的指令。