目 录

1 简介 1
1.1 文档介绍 1
1.2 参考文档 1
1.3 术语说明 1
1.4 技术背景 1
2 设计规格 1
3 概要设计 1
3.1 整体框图 2
3.2 接口列表 2
3.3 接口时序 3
4 详细设计 3
4.1 总线接口说明 3
4.2 配置说明 4
4.3 寄存器说明 4
4.4 IF stage 4
4.5 ID stage 6
4.6 ISSUE stage 7
4.7 EX stage 11
4.7.1 csr_ctrl 11
4.7.2 branch_unit 12
4.7.3 ALU 单元 12
4.7.4 load/store 13
4.7.5 mlt/div 16
4.8 COMMIT 18
4.9 握手处理 18
4.10 ctrl_flow 21
4.10.1 fence 21
4.10.2 branch 21
4.10.3 jump 21
4.11 异常处理 22
4.11.1 指令 22

4. 11. 2 中断 22
4. 11. 3 异常 23
4.12 csr 23
4.13 controller 24
5 tb 说明 25
6 其他 27
7 plup ..... 错误！未定义书签。
8 遗留问题 27

1 简介

1.1 文档介绍

1.2 参考文档

1.3 术语说明

表 1.1 术语说明

缩写	全称	描述

1.4 技术背景

2 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。

3 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。

3.1 整体框图

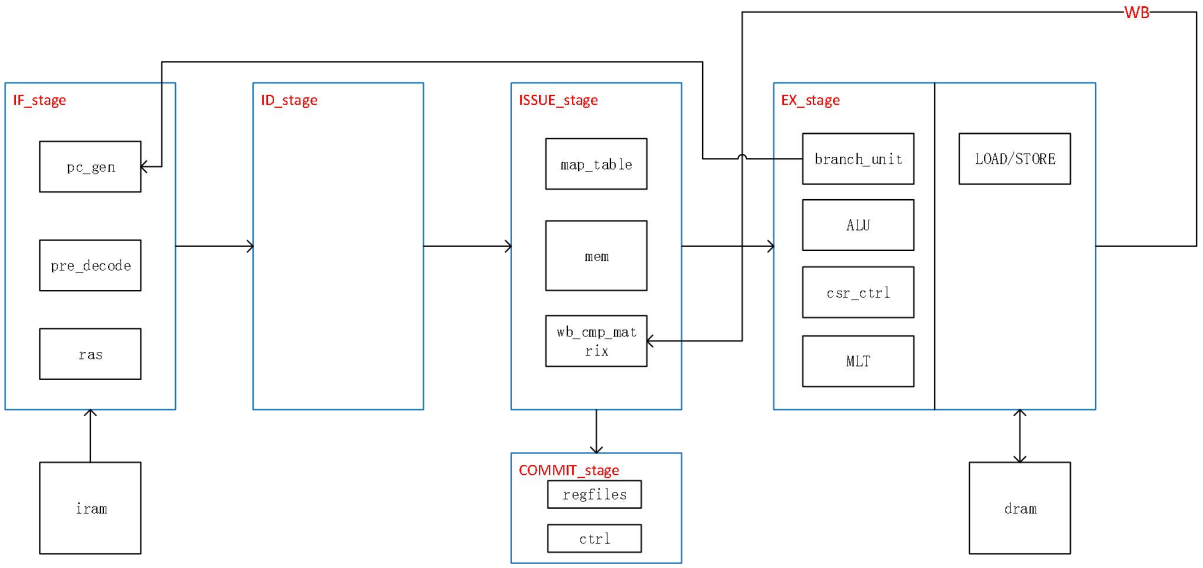


图 3.1　整体框图

3.2 接口列表

表 3.1　XXXX 接口列表

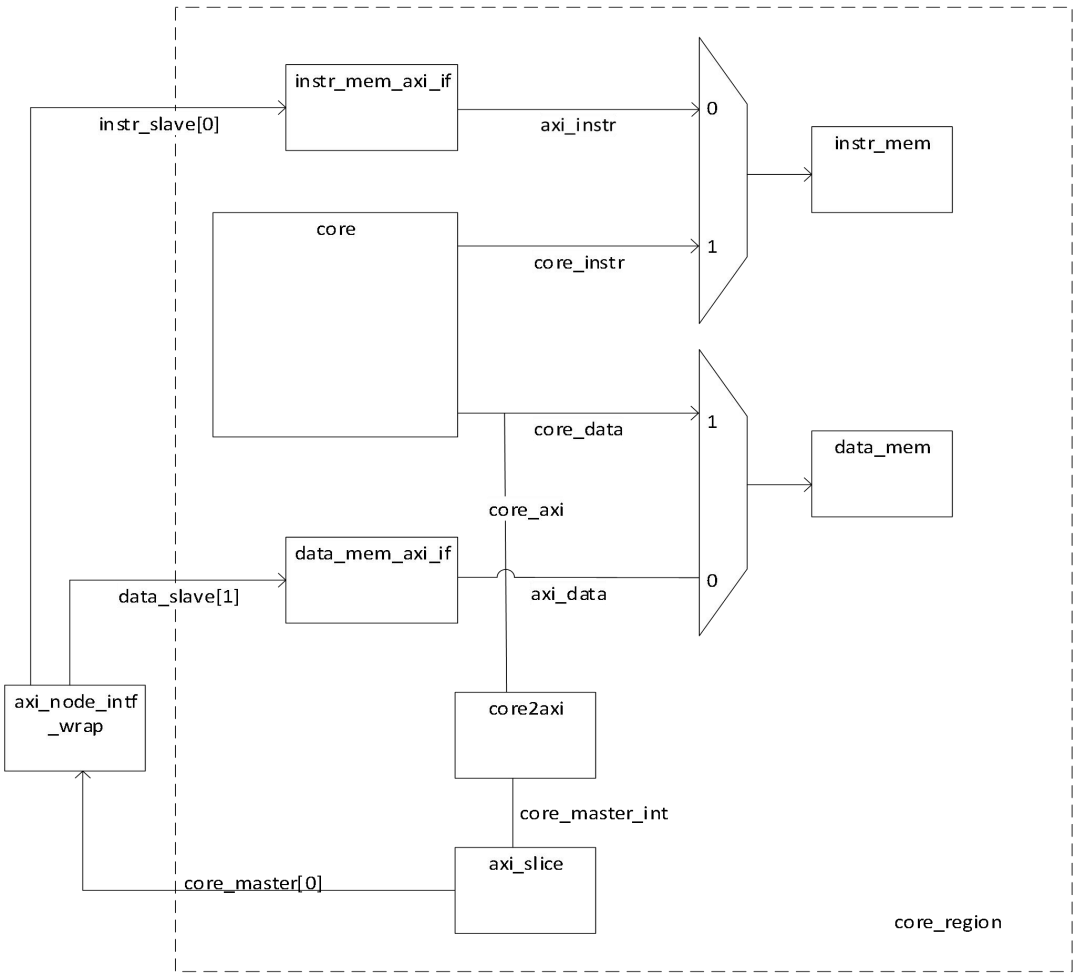
信号	方向	位宽/类型	描述
clk_i			
rst_ni			
instr_boot_addr_i			

instr			iram 接口
data			dram 接口

3.3 接口时序

4 详细设计

4.1 总线接口说明



4.2 配置说明

列明本 IP 支持哪些参数化配置。如不支持则写：不支持参数化配置。

4.3 寄存器说明

表 3.2　XXXX 寄存器说明

寄存器	地址	复位值	属性	描述

4.4 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（ra）、x5（t0）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≠rd，为函数返回 return，需要弹栈；（表 2/4 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
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	pop, then push
link	*link*	1	push

jalr 根据此表选择栈操作，link 表示 rd/rs1=x1/x5

jal	jal rd,imm[20:1]	x(rd)=PC+4;	该指令执行两步操作：1.将下一条指

		PC+=sext({imm[20:1],1'b0})	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。

4.5 ID stage

- 32 位指令的译码

case（opcode）

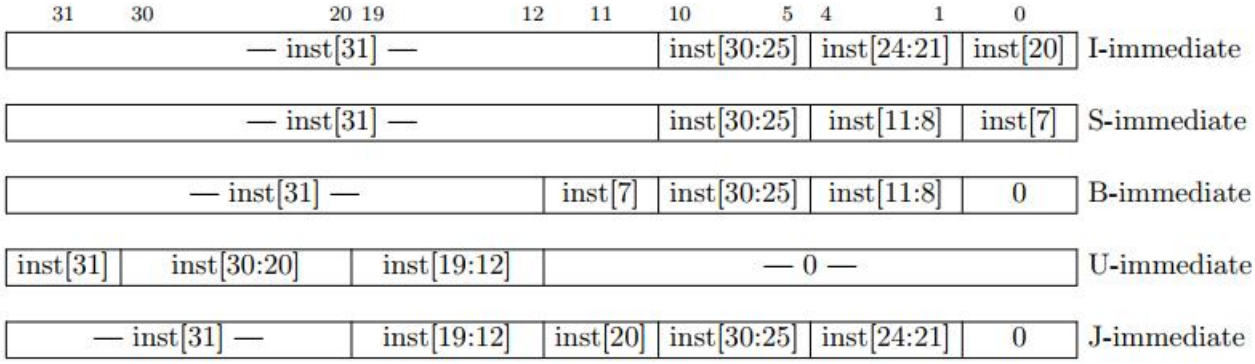
case（func3）

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；
- immsel：

  对 I\S\B\U\J 型指令的 imm 组合输出。



4.6 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。

在写入 sbd 之前，判断 rs_rdy，与 sbd 中已写回的 rd 比对，与 wb_port 比对；

ex 执行完成后，wb_port 写回 sbd，写回对应 trans_id 表项的 rd 值，写回其他表项中 rs 重命名为 trans_id 的 rs 值；

表项中 rs_rdy 且在 sbd 中待 issue 的头部，此表项可 issue；

表项中 rd_rdy 且在 sbd 中待 commit 头部，此表项可 commit；

- 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。WAR 情况，在顺序发射的情况下会自然保证。

在指令写入 sbe 时更新 map 表（每一条指令的 rd 都被 map），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，后续指令使用此 rd 作为 rs，在 rd wb 时，rs 也已写回 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 sbd 中未提交的指令 rd 是否和当前将 commit 的 rd 逻辑名一致（即此 rd 被重命名了多次，未提交的指令中重命名覆盖当前指令的重命名），若有则不清除 map（保留未提交指令 rd 在 map 中的重命名），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，数据在 sbd 中。

注：不是每条指令都有 rd，没有 rd 的指令 rd=0，即不更新 map。csr 指令不可 rename，csr 指令需要等前面的 csr 指令 commit 后才能发射。

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	……	X31
{vld,rename}			{1,wptr_q}											

map_table

rename 方法，写入指令的 rd 若不是 x0，则在 map 表中被重命名为 sbe 中目前指针的 id（wptr_q），map 表中的索引是寄存器原来的逻辑名，在重命名前加 1bit，表示重命名有效。

```
struct packed {
    logic                           rename_vld     ;
    logic   [REG_ADDR_SIZE-1:0]     physical_name  ;   // sbe id
} map_table_q[31:0] , map_table_d[31:0] ;               //x0 is no need to map
```

- 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   ; // this bit indicates whether occupied the entry
    sbd_entry_t             sbe      ; // this is the score board entry we will send to ex
} sbd_entry_q [MEM_DEPTH-1:0], sbd_entry_d [MEM_DEPTH-1:0];
```

```
typedef struct packed {
    logic [31:0]                pc;
    logic [ADDR_BITS-1:0]       trans_id;

    fu_t                        fu;
    fu_op_t                     op;
    logic [31:0]                imm;
    logic                       csr_instr;

    logic [REG_ADDR_SIZE:0]     rs1;
    logic [REG_ADDR_SIZE:0]     rs2;
    logic [REG_ADDR_SIZE-1:0]   rd;
} sbd_entry_t;

typedef struct packed {
    logic [REG_DATA_WIDTH-1:0]  rs1_data;
    logic                       rs1_rdy;
} sbd_rs1_t;

typedef struct packed {
    logic [REG_DATA_WIDTH-1:0]  rs2_data;
    logic                       rs2_rdy;
} sbd_rs2_t;


typedef struct packed {
    logic [REG_DATA_WIDTH-1:0]  result;
    logic                       rd_vld;
    branch_t                    branch;
    exception_t                 ex    ;
} sbd_wb_t;
```

```
struct packed {
    logic                   writed   ; // this bit indicates w
    sbd_entry_t             sbe      ; // this is the score bo
} sbd_entry_q [MEM_DEPTH-1:0], sbd_entry_d [MEM_DEPTH-1:0];

sbd_wb_t  [MEM_DEPTH-1:0]   sbd_wb_q        , sbd_wb_d ;
sbd_rs1_t [MEM_DEPTH-1:0]   sbd_rs1_q       , sbd_rs1_d;
sbd_rs2_t [MEM_DEPTH-1:0]   sbd_rs2_q       , sbd_rs2_d;
logic     [MEM_DEPTH-1:0]   sbd_issue_q     , sbd_issue_d;
```

sbe 拆成了 5 个包。

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.writed 置 1。

在此需要判断 rs_rdy 信号：

① 读 map_table，若没有 rename 则说明不是前序指令的 rd，rs 保存在 regfile 中，置 rs_rdy=1，并从 regfile 读数据写入 rs_data；若有 rename，rs 可通过转发机制获取，即 rs 可从 wb 口获取，或从已写入 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；

```
78    always_comb begin :write_back_compare
79        wb_comp_matrix = '{(MEM_DEPTH+1){'{1'b0, 1'b0, 32'b0, 32'b0}}};//'{default: 0};
80        for(int unsigned i = 0; i < NR_WB_PORTS; i++) begin
81            if(wb_port_i[i].wb_vld) begin
82                for(int unsigned j = 0; j < MEM_DEPTH; j++) begin
83                    if(sbd_entry_q[j].writed) begin
84                        if((~sbd_rs1_q[j].rs1_rdy) & sbd_entry_q[j].sbe.rs1[REG_ADDR_SIZE] & (sbd_entry_q[j].sbe.rs1[REG_ADDR_SIZE-1:0] == wb_port_i[i].trans_id)) begin
85                            wb_comp_matrix[j].rs1_cmp_rdy = 1'b1;
86                            wb_comp_matrix[j].rs1_data = wb_port_i[i].wb_data;
87                        end
88                        if((~sbd_rs2_q[j].rs2_rdy) & sbd_entry_q[j].sbe.rs2[REG_ADDR_SIZE] & (sbd_entry_q[j].sbe.rs2[REG_ADDR_SIZE-1:0] == wb_port_i[i].trans_id)) begin
89                            wb_comp_matrix[j].rs2_cmp_rdy = 1'b1;
90                            wb_comp_matrix[j].rs2_data = wb_port_i[i].wb_data;
91                        end
92                    end
93                end
94                //compare input instr.rs with wb,  forward data
95                if(sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE-1:0] == wb_port_i[i].trans_id && sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE]) begin
96                    wb_comp_matrix[MEM_DEPTH].rs1_cmp_rdy = 1'b1;
97                    wb_comp_matrix[MEM_DEPTH].rs1_data    = wb_port_i[i].wb_data;
98                end
99                if(sbd_entry_d[wptr_q].sbe.rs2[REG_ADDR_SIZE-1:0] == wb_port_i[i].trans_id && sbd_entry_d[wptr_q].sbe.rs2[REG_ADDR_SIZE]) begin
100                   wb_comp_matrix[MEM_DEPTH].rs2_cmp_rdy = 1'b1;
101                   wb_comp_matrix[MEM_DEPTH].rs2_data    = wb_port_i[i].wb_data;
102               end
103           end
104       end
105   end
```

83~93 行，比较 sbd 中已写入的数据，与 wb-port

sbd[j]表项中~rs1_rdy，sbd[j]中 rs1 被重命名，且 wb 端口中 id 就是 sbd[j]表项中重命名的 rs1，那么将 wb 端口上写回的数据赋值给 wb-comp-matrix；

95~102 行，比较正要写入 sbd 的数据与 wb-port

正要写入指令的 rs 与 wb 口 id 相同，且正要写入指令的 rs 被重命名；

③ 从 sbd_wb（与 sbd 的表项一一对应）中已写回的数据获取。比较 sbd_wb 中所有项（乱序写回），若其 rd_vld，且 j 与当前 wptr_q 所指的项中的重命名 rs 相等，则 rs_rdy=1，rs_data = sbd_wb[j].result。

```
110     always_comb begin :input_compare_sbe
111         input_comp = '{default: 0};
112         for(int unsigned j = 0; j < MEM_DEPTH ; j++) begin
113             if(sbd_wb_q[j].rd_vld && sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE] && (sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE-1:0] == unsigned'(j))) begin
114                 input_comp.rs1_cmp_rdy   = 1'b1;
115                 input_comp.rs1_data      = sbd_wb_q[j].result;
116             end
117             if(sbd_wb_q[j].rd_vld && sbd_entry_d[wptr_q].sbe.rs2[REG_ADDR_SIZE] && (sbd_entry_d[wptr_q].sbe.rs2[REG_ADDR_SIZE-1:0] == unsigned'(j))) begin
118                 input_comp.rs2_cmp_rdy   = 1'b1;
119                 input_comp.rs2_data      = sbd_wb_q[j].result;
120             end
121         end
122     end
```

113 行含义：乱序写回，sbd_entry 上每个项都对应 sbd_wb,目前配置 8 个，将 8 个 sbd_wb 与正在写入指令的 rs 做比较。

sbd_wb_q[j].rd_vld 表示第 j 项写回 rd 有效（rd 重命名为 j）；

sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE]表示写入指令的 rs1 有被重命名过，

sbd_entry_d[wptr_q].sbe.rs1[REG_ADDR_SIZE-1:0] == unsigned'(j)表示正在写入指令的 rs1 的重命名等于 j，说明第 j 项写回的 rd 就是正在写入指令的 rs1。

2. 从 writeback 端口写回数据（wb[0] csr，wb[1] alu，wb[2] lsu，wb[3] md）

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；writed=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 时已经写回其他 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_inflignt 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，flush_issue 指针清零。

4. cnt

cnt_d =cnt_q + writeen -readen

可理解为：

mem_d[wptr_q].writed & ~full = write_en

commit = read_en

4.7 EX stage

4.7.1 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

4.7.2 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 sbe 的 rd，rd=pc+4

4.7.3 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'h3ffffffff; //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，只会在高位补 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
fu_data_i		fu_data_t	
wb_alu_o			写回 sbe 的 rd
ls_addr_o			输出给 load/store 的访问地址

减法计算，补码计算，速记法：求补码和原码都是先扩一个符号位，全部取反后+1

15-11

01111-01011（求补码，全部取反（符号位原来是 0，取反后是 1），然后+1）

01111+10101=00100

11-15

01011-01111

01011+10001=11100（求原码，-1，全部取反）（11100-1=11011，取反 00100）

求原码也可全部取反，然后+1（原理补码和原码相加再+1 是模）

11100（全部取反）->00011（＋1）->00100

4.7.4 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.

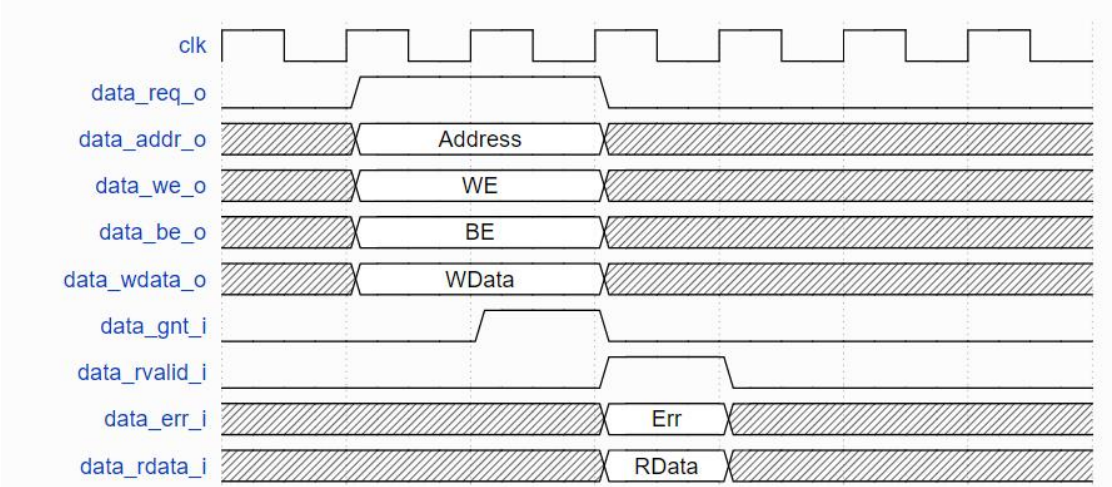


Figure 6 Basic Memory Transaction

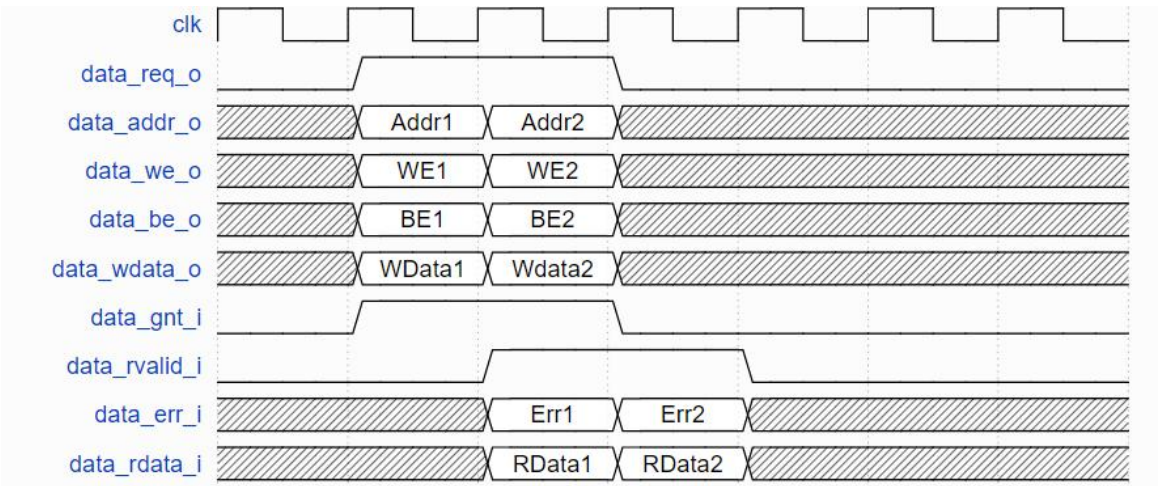


Figure 7 Back-to-back Memory Transaction

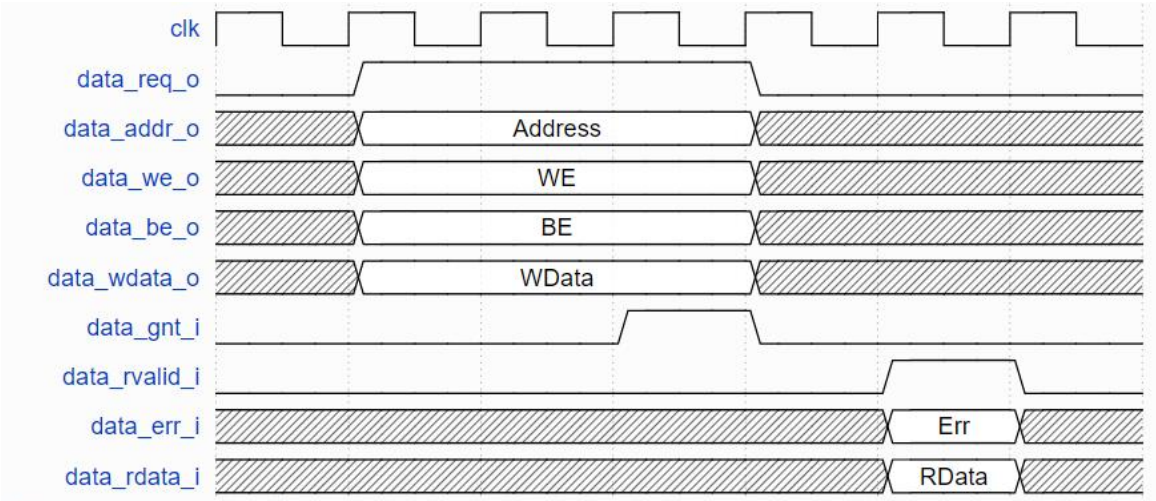


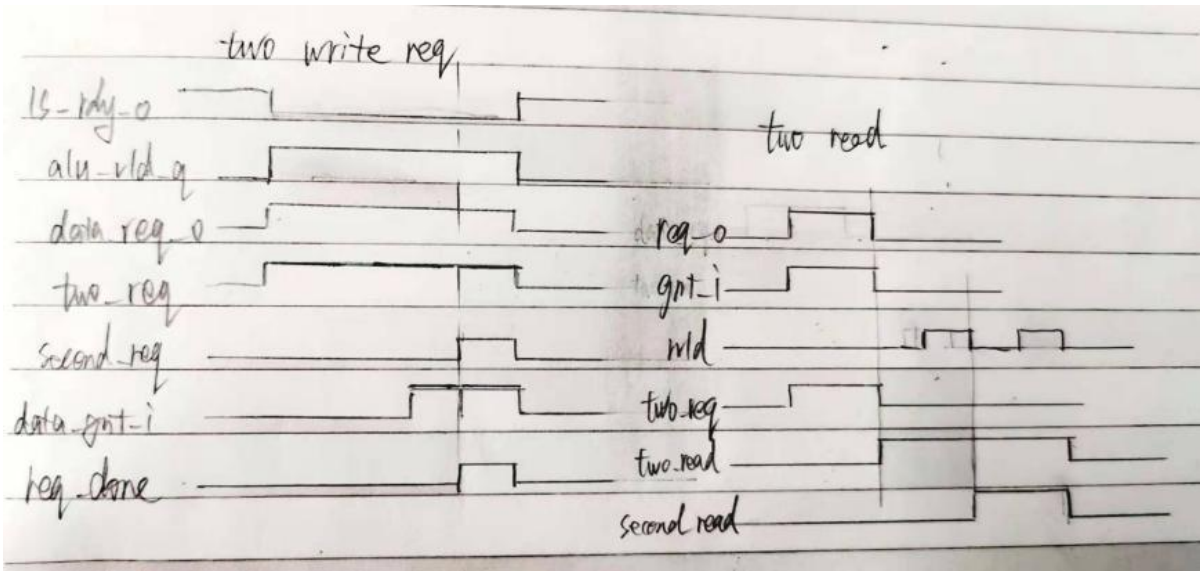
Figure 8 Slow Response Memory Transaction

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，addr[31:13]==19'b0 则地址是在 0x00002000 之前，属于 instr_mem；addr[31:13]!=19'b1 属于 data_mem 范围以外的地址；

4.7.5 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。32 位可表示范围-2^{31}~2^{31}-1，即 0x8000_0000~0x7FFF_FFFF。

考虑 a/b 的情况：

	div	rem	divu	remu
|a| < |b|	0	a	0	a
b = 0	32'hFFFFFFFF (有符号，表示-1)	a	32'hFFFFFFFF (无符号，2^{32}-1)	a
$-2^{31}/-1$	32'h80000000 （-2^{31}）	0	——	——

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

注：例如 L=8，能表示最大范围是-128~127，-128/-1=128 无法表示，算作溢出。1000_0000 表示-128。

符号位是 1 的 data，需要先用原码计算，计算出值后若为负数，求补码表示计算值；

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;
```

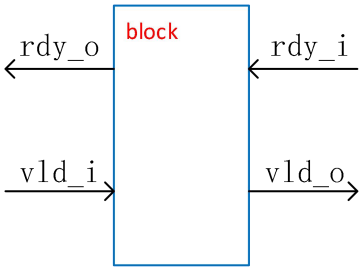
4.8 COMMIT

包含 regfiles 和 ctrl。

ctrl 包含 mispredict 以及 exception 导致的 flush 逻辑。

4.9 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<=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<=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 <= 0;
else if（input_hsk）
vld_q <= 1;
else if(output_hsk)
vld_q <= 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	always_ff @(posedge clk_i or negedge rst_ni) begin
	if (!rst_ni)
	ls_rdy_o <= 1'b1;
	else if(flush_ex_i)
	ls_rdy_o <= 1'b1;
	else if((data_gnt_i && ~two_req) ||
	(data_gnt_i && second_req))
	ls_rdy_o <= 1'b1;
	else if(data_req_o)
	ls_rdy_o <= 1'b0;
	end
		一次请求、两次请求接收到 dram gnt 响应后可解除 load_store 占用。

	always_comb begin
	if((data_rvalid_i && ~data_err_i && ~two_read) ||
	(data_rvalid_i && ~data_err_i && 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 && data_gnt_i && ~two_req) ||
	(data_we_o && data_gnt_i && 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
		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。

	end	
MULT /DIV		

5. commit

无反压逻辑，可直接写入 regfiles。

4.10 ctrl_flow

4.10.1 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），在继续发射。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。

4.10.2 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。（但 map 表需要遇到 branch 指令时备份，恢复 mispredict 指令之前 map 的映射状态）

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。（map 清空）

4.10.3 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。

4.11 异常处理

中断、ebreak、ecall 更新 mepc=pc+4

异常更新 mepc=pc

4.11.1 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。

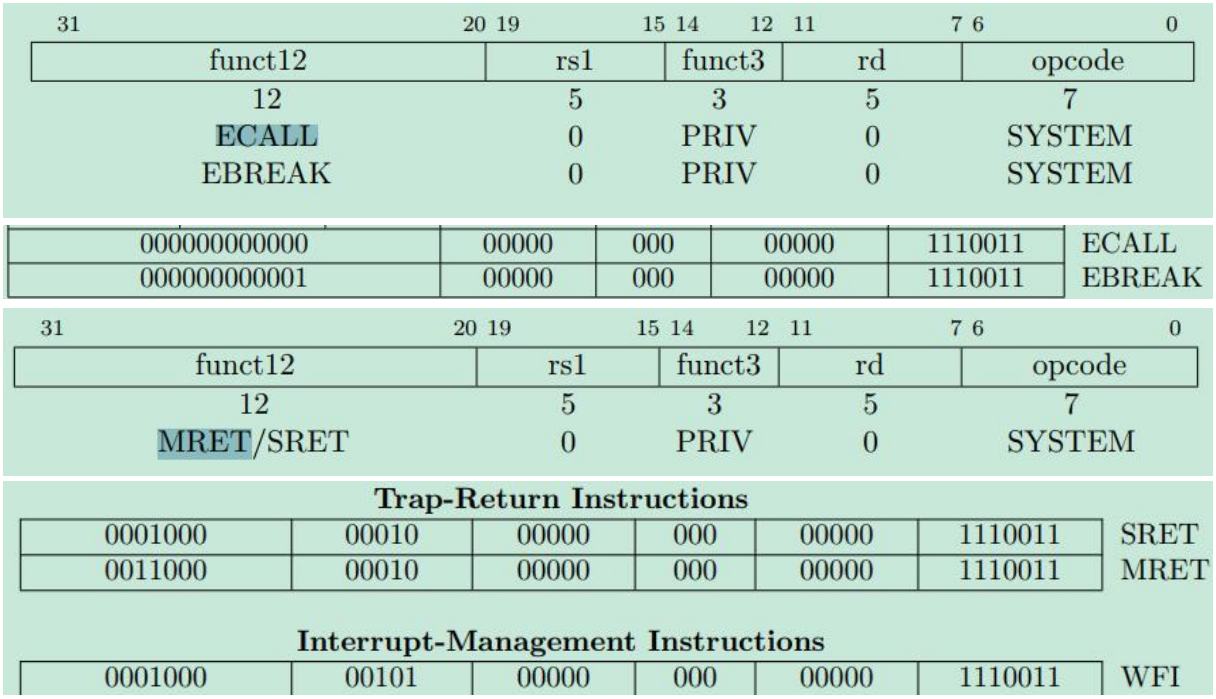
更新 csr 中 mcause 为{0,3/11}；

更新 mepc 为 pc+4；这里不对 mepc 中存入 ebreak 或 ecall 本身的地址

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；



31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRET/SRET	0	PRIV	0	SYSTEM

Trap-Return Instructions

0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET

Interrupt-Management Instructions

0001000	00101	00000	000	00000	1110011	WFI

4.11.2 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 execption，对应的指令已经被 flush，不需要处理。

4.11.3 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。

4.12 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	mcounteren	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.

4.13 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 mispredict 前的映射情况，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。

5 tb 说明

- 定义宏 ISA_TEST_ONE_RAM，将 instr、data 数据存入同一个 selfdef_ram，否则分别存入 instr_ram、data_ram（没实现）。
- readmemx 函数

$readmemx的格式：

这两个系统任务用来从文件中读取数据到存储器中。可以在仿真的任何时刻被执行使用，使用格式共六种：

```
$readmemb("<数据文件名>",<存贮器名>)
$readmemb ("<数据文件名>",<存贮器名>,<起始地址>)
$readmemb ("<数据文件名>",<存贮器名>,<起始地址>,<结束地址>)

$readmemh("<数据文件名>",<存贮器名>)
$readmemh  ("<数据文件名>",<存贮器名>,<起始地址>)
$readmemh  ("<数据文件名>",<存贮器名>,<起始地址>,<结束地址>)
```

在这两个系统任务中，被读取的数据文件的内容只能包含：空白位置（空格、换行、制表格、注释行、二进制或十六进制的数字。数字中不能包含位宽说明和格式说明，对于$readmemb和$readmemh系统任务，每个数字可以是二进制和十六进制数字。另外，数字必须用空白位置或注释行来分隔开。

对于上面6种系统任务格式，需补充说明一下5点：

(1) 如果系统任务声明语句中和数据文件里都没有进行地址说明，则默认存放起始地址为该存储器定义语句中的起始地址。数据文件里的数据被连续存放到该存储器中，直到该存储单元存满为止或数据文件里的数据存完。

(2) 如果系统任务中说明了存放的起始地址，没有说明存放的结束地址，则数据从起始地址开始存放，存到该存储器定义语句中的结束地址为止。

(3) 如果系统任务声明语句中，结束地址和起始地址都进行了说明，则数据文件里的数据按该起始地址开始存放到存储单元中，直到该结束地址，而不考虑该存储器的定义语句的起始地址和结束地址。

(4) 如果地址信息在系统任务和数据文件里面都进行了说明，那么数据文件里的地址必须在系统任务中地址参数声明的范围之内。否则提示错误，并且装载数据到存储器中的操作被中断。

(5) 如果数据文件里的数据个数和系统任务中起始地址及结束地址暗示的数据个数不同的话，也要提示错误信息。

下图文件中地址分段，第一段从 mem[0]开始存，第二段从 mem[4096]，第三段从 mem[8192]开始存；

tb_bmm.sv | 2 rv32ui-p-add.verilog | 3 bmm.yml | 4 rv32ui-p-add.dump | 5 bmm.sv

```
@00000000
6F 00 C0 04 73 2F 20 34 93 0F 80 00 63 0A FF 03
93 0F 90 00 63 06 FF 03 93 0F B0 00 63 02 FF 03
17 0F 00 00 13 0F 0F FE 63 04 0F 00 67 00 0F 00
@00001000
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
@00002000
EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE
EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE
EF BE AD DE EF BE AD DE 00 00 00 00 00 00 00 00
```

- 数据在 mem 和 ram 中的存储顺序

指令流数据在 hex 或 verilog 文件中按 byte 分割，从左到右从 0 地址增加；

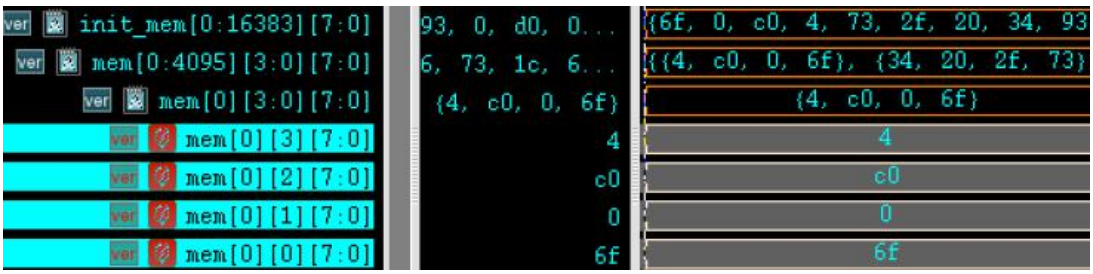
读入到 mem，6f 存到 mem[0],00 存 mem[1],c0 存 mem[2]，依次类推；

```
for(int unsigned addr = 0; addr < MEM_DP/4; addr++) begin
    for (int unsigned offset = 0; offset < 4; offset++) begin
        u_ram.mem[addr][offset] = init_mem[addr*4 + offset];
    end
end
```

ram.mem 定义 4096 个行，每行 4 个 byte

```
init_mem[0:16383][7:0]   d0, 0...   {6f, 0, c0, 4, 73, 2f, 20, 34, 93, f, 80, 0, 63, a, ff, 3, 93, f,
mem[0:4095][3:0][7:0]    1c, 6...   {{4, c0, 0, 6f}, {34, 20, 2f, 73}, {0, 80, f, 93}, {3, ff, a, 63},
```

存入到 ram，从第 0 行依次增加存入，每行从第 0 byte 依次存入



struct packed{

logic a;

logic b;

logic c;

}abc_tmp;

定义 struct 结构 abc_tmp，bit 占位{a，b，c}。

6 其他

- 每条指令执行的时钟周期，越小越好。

$$CPI = \frac{\sum_i 指令i占用的时钟周期数 \times 指令i的数量}{程序中指令总数}$$

- 流水线性能：各流水级实际延迟长度不一致，每个流水级都要选用耗时最长的延时（内部碎片），总流水延时会增加；有些指令不需要经过某一流水级，这级就会空闲（外部碎片）；指令存在时间相关性，后续指令需要前序指令的结果，流水线停顿；
- 冒险：结构冒险，同时访问寄存器或 mem，通过分 iram、dram，寄存器增加一组读写端口来控制；数据冒险，后续指令需要用前序指令结果，转发机制可解决一部分；控制冒险：可能预测跳转错误。无条件直接跳（imm 与 pc 计算跳转地址），无条件间接跳（imm 与 pc 与 reg 计算跳转地址），有条件直接跳（比对 rs，imm 与 pc 计算跳转地址）；
- 分支预测：静态、动态（分支历史表[预测状态][分支指令地址][预测目标地址]）；
- 单发射、多发射；顺序发射、乱序发射（实现方式）；scoreboard 算法 tomasulo 算法

7 遗留问题

修改指令取指 ram 接口，按协议修改

RV32I 通用寄存器

寄存器	ABI 名字	描述	Saver
x0	zero	0 值寄存器，硬编码为 0,写入数据忽略，读取永远为 0	-
x1	ra	返回地址	Caller
x2	sp	栈指针	Callee
x3	gp	全局指针	-
x4	tp	线程指针	-
x5	t0	临时寄存器或者备用的链接寄存器	Caller
x6-x7	t1-t2	临时寄存器	Caller
x8	s0/fp	需要保存的寄存器或者帧指针寄存器	Callee
x9	s1	需要保存的寄存器，保存原进程中的关键数据，避免在函数调用过程中被破坏	Callee
x10-x11	a0-a1	函数参数/返回值	Caller
x12-x17	a2-a7	函数参数	Caller
x18-x27	s2-s11	需要保存的寄存器	Callee
x28-x31	t3-t6	临时寄存器	Caller

1.函数调用时保留的寄存器

被调用函数一般不会使用这些寄存器，即便使用也会提前保存好原值，可以信任。这些寄存器有：sp, gp, tp 和 s0-s11 寄存器。

2.函数调用时不保存的寄存器

有可能被调用函数使用更改，需要 caller 在调用前对自己用到的寄存器进行保存。这些寄存器有：参数与返回地址寄存器 a0-a7，返回地址寄存器 ra，临时寄存器 t0-t6

RISC-V 的架构明确要求其采用默认的静态分支预测机制，即：如果是向后跳转的条件跳转指令，则预测为“跳”；如果是向前跳转的条件跳转指令，则预测为“不跳”，