|  |  |
| --- | --- |
| 文档编号 |  |
| 文档版本 |  |
| 文档管控 | 内部公开 |
| 存档日期 |  |

core模块设计方案

|  |  |
| --- | --- |
| 编 写： |  |
| 校 对： |  |
| 审 核： |  |
| 批 准： |  |

自主可控项目组

2021年XX月XX日

文档修订记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **序号** | **版本编号** | **变化状态** | **变更说明** | **作者** | **日期** |
| 1 | V0.1 | C |  |  |  |
|  |  |  |  |  |  |
| 3 |  |  |  |  |  |
|  |  |  |  |  |  |

\*变化状态：C—创建，A—增加，M—修改，D—删除

文档审批信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **审核** | **会签** | **批准** | **备注** |
| V1.0 |  |  |  |  |
|  |  |  |  |  |

目 录

[1 简介 1](#_Toc67641771)

[1.1 文档介绍 1](#_Toc67641772)

[1.2 参考文档 1](#_Toc67641773)

[1.3 术语说明 1](#_Toc67641774)

[1.4 技术背景 1](#_Toc67641775)

[2 设计规格 2](#_Toc67641776)

[3 概要设计 2](#_Toc67641777)

[3.1 整体框图 2](#_Toc67641778)

[3.2 接口列表 2](#_Toc67641779)

[3.3 接口时序 3](#_Toc67641780)

[3.4 配置说明 3](#_Toc67641781)

[3.5 XXXX 3](#_Toc67641782)

[4 详细设计 4](#_Toc67641783)

[4.1 寄存器说明 4](#_Toc67641784)

[4.2 时钟切换电路 5](#_Toc67641785)

[4.3 启动流程 6](#_Toc67641786)

[5 编程指南 7](#_Toc67641787)

[5.1 动态时钟切换 7](#_Toc67641788)

[6 验证关注点 8](#_Toc67641789)

[7 遗留问题 8](#_Toc67641790)

1. 简介
   1. 文档介绍
   2. 参考文档
   3. 术语说明

表1.1 术语说明

|  |  |  |
| --- | --- | --- |
| **缩写** | **全称** | **描述** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

* 1. 技术背景

1. 设计规格
2. 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。
3. 添加中断异常处理、csr读写逻辑。
4. 支持k扩展，替换为AXI总线。
5. 概要设计
6. 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。
7. ID解码32位指令，目前包括I、CSR指令。
8. 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。
9. 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。

1. WB结果写回到Scoreboard中，支持结果转发。
2. commit包括regfiles，以及flush逻辑。
   1. 整体框图

**

图3.1 整体框图

* 1. 接口列表

*描述本模块的输入输出接口。接口列表是顶层集成的重要依据，需要准确清晰，并说明注意事项。*

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 |
|  |  |  |  |

* 1. 接口时序

1. 详细设计
   1. 总线接口说明



* 1. 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。*

* 1. 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** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*注：RO——只读寄存器；RW——可读可写寄存器*

*样式2：适用于域段拆分较多的IP*

*参考pulpino datasheet；需要补充各域段的读写属性*

* 1. 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

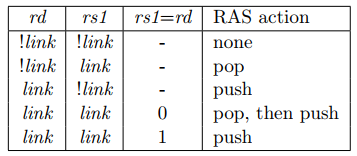
深度可配，超过深度后，最先push的地址溢出，当跳出此条地址对应的函数时，pop出数据的vld为0，需要等待ex计算返回地址。

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。



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;  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。 |

* 1. 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 ;

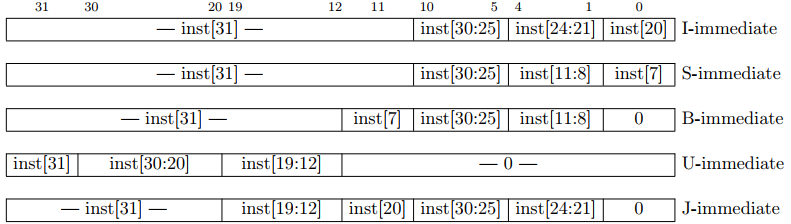
logic instr\_ebreak;

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。



* 1. 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。

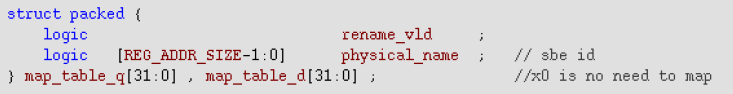
不在map中的rs，数据一定在regfile中，在map中的rs，数据在sbd中。

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} |  |  |  |  |  |  |  |  |  |  |  |

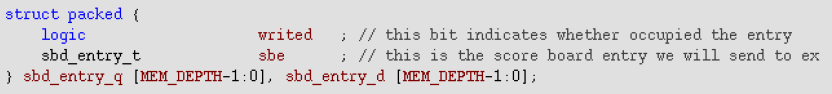
map\_table

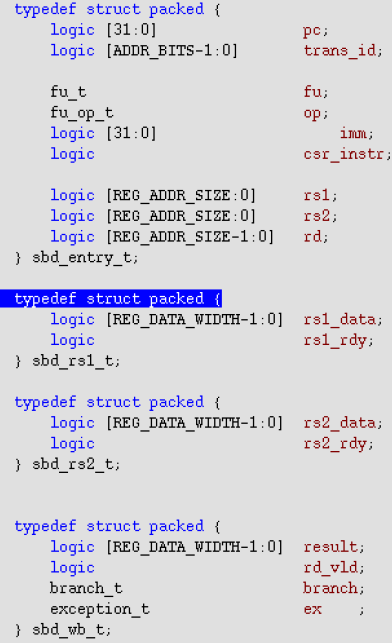
rename方法，写入指令的rd若不是x0，则在map表中被重命名为sbe中目前指针的id（wptr\_q），map表中的索引是寄存器原来的逻辑名，在重命名前加1bit，表示重命名有效。

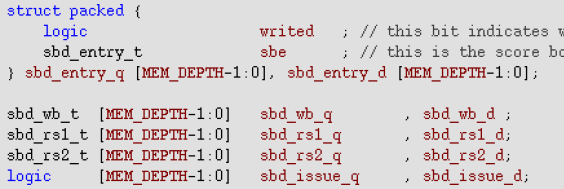


* mem

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。





sbe拆成了5个包。

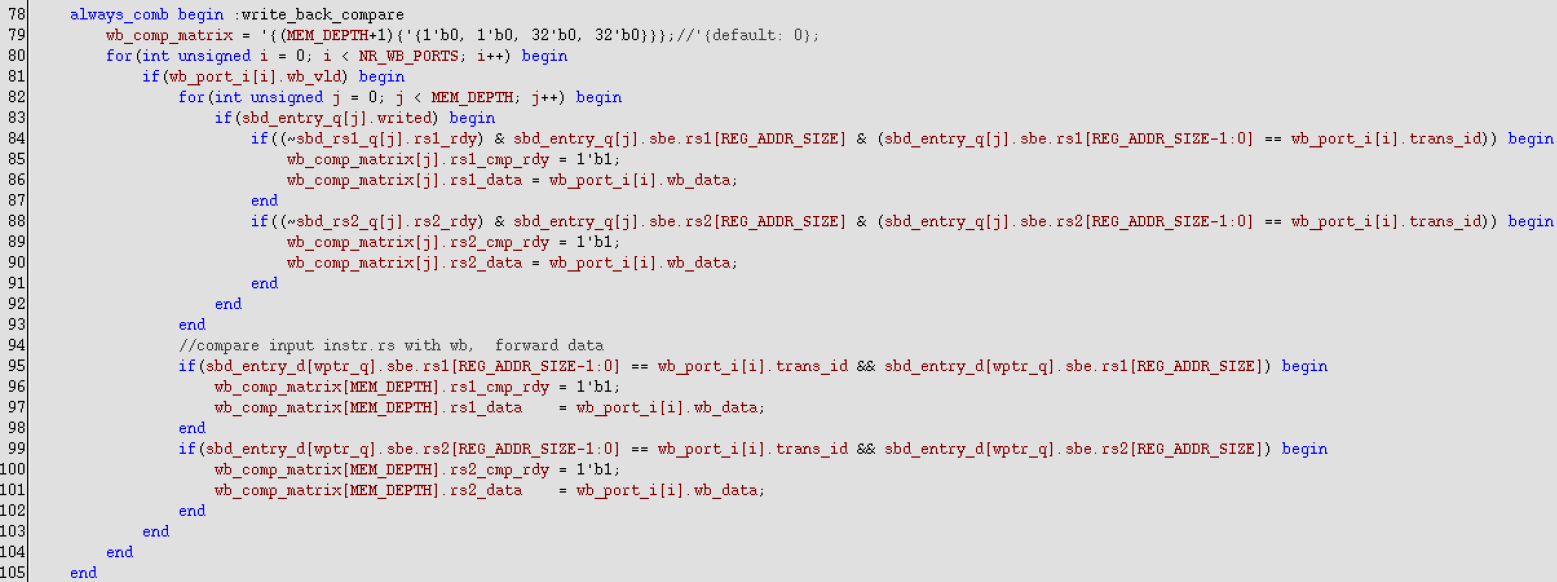
mem主要的4类操作：

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；



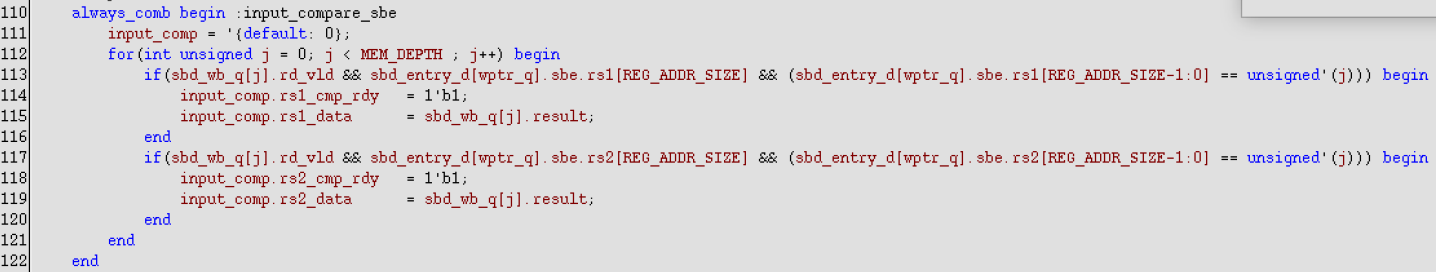
83~93行，比较sbd中已写入的数据，与wb-port

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；

1. 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。



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。

1. 从writeback端口写回数据（wb[0] csr，wb[1] alu，wb[2] lsu，wb[3] md）
2. 使用wb\_cmp\_matrix，比对sbe中所有rs，并写入。
3. 匹配trans\_id项，写回result，将rd\_rdy置1。
4. 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。

1. 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。

1. 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。

1. wptr

写入sbe时指针加一，flush\_issue指针清零。

1. cnt

cnt\_d =cnt\_q + writeen -readen

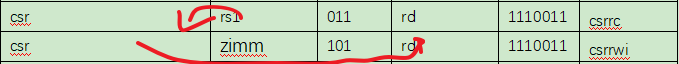
可理解为：

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

commit = read\_en

* 1. EX stage
     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。



|  |  |  |  |
| --- | --- | --- | --- |
| **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 |
|  |  |  |  |

* + 1. 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 |
|  |  |  |  |
|  |  |  |  |

* + 1. 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）

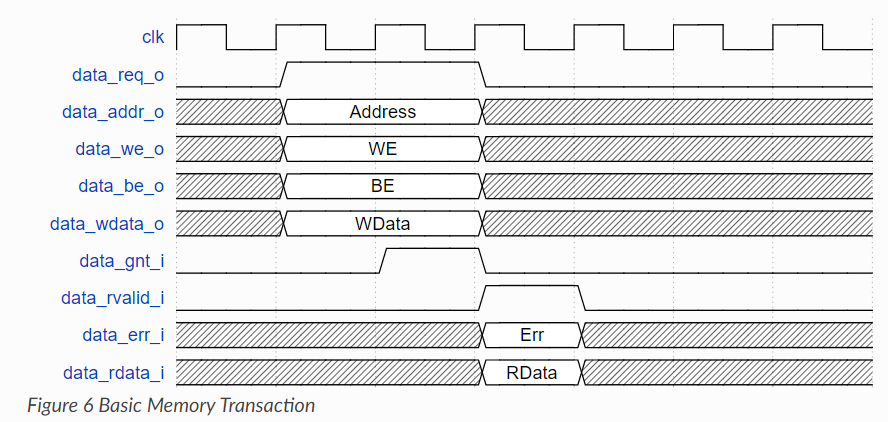
11100（全部取反）->00011（＋1）->00100

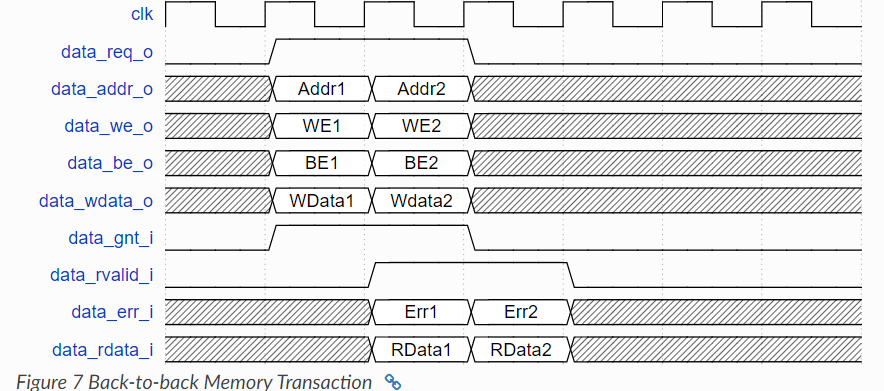
* + 1. load/store

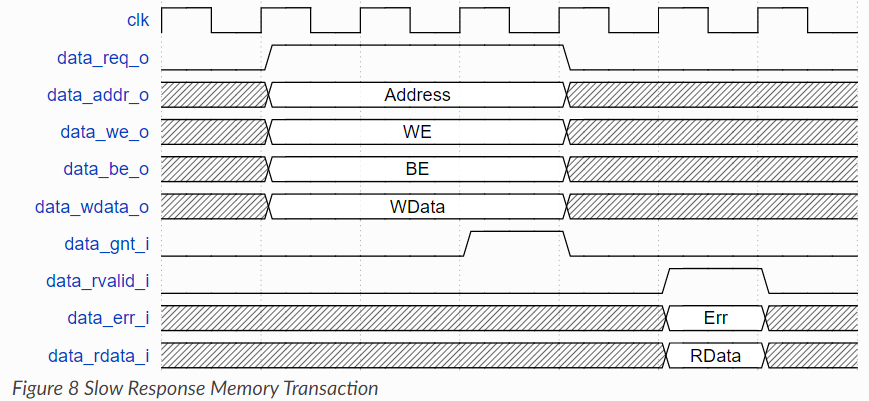
参考ibex的load\_store传输协议。

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.





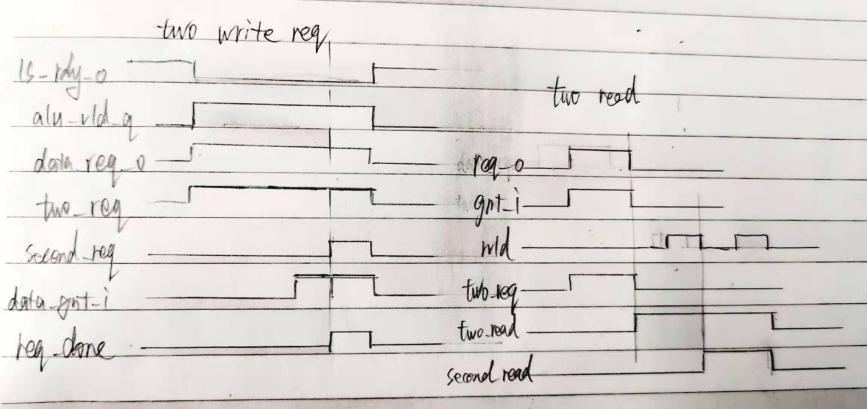


指令先经过alu进行地址计算，然后访问DMEM，至少需要2cycle。

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；

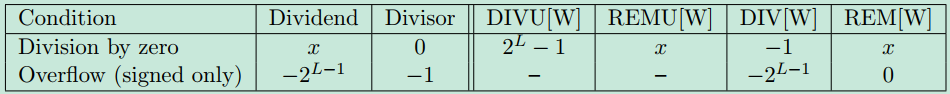
* + 1. mlt/div

|  |  |  |  |
| --- | --- | --- | --- |
| **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-231~231-1，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，232-1) | a |
| -231/-1 | 32’h80000000（-231） | 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。

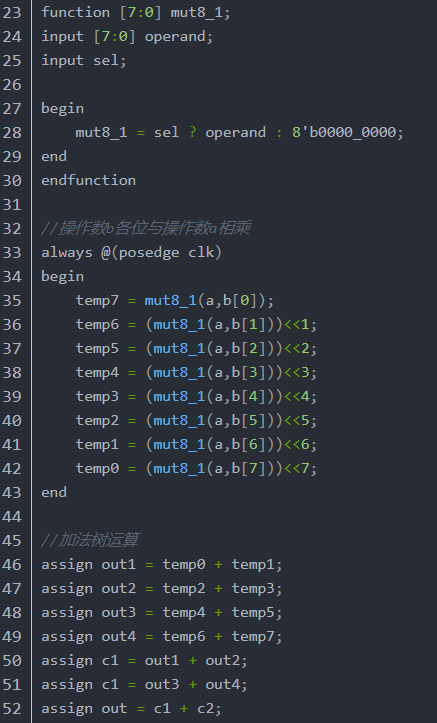
符号位是1的data，需要先用原码计算，计算出值后若为负数，求补码表示计算值；

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：



* 1. COMMIT

包含regfiles和ctrl。

ctrl包含mispredict以及exception导致的flush逻辑。

* 1. 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。 |

1. 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。 |

1. 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; |  |

1. 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  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。 |
| MULT/DIV |  |  |

1. commit

无反压逻辑，可直接写入regfiles。

* 1. ctrl\_flow
     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。

* + 1. 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。
   * 1. 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。

* 1. 异常处理

中断、ebreak、ecall更新mepc=pc+4

异常更新mepc=pc

* + 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。

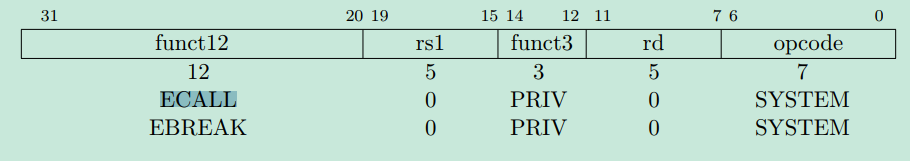
更新csr中mcause为{0,3/11}；

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

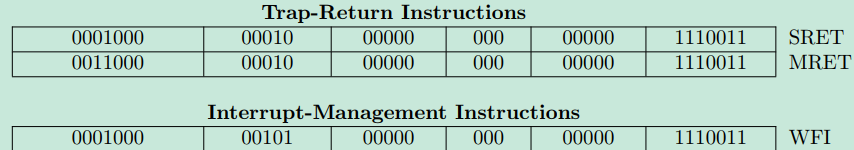
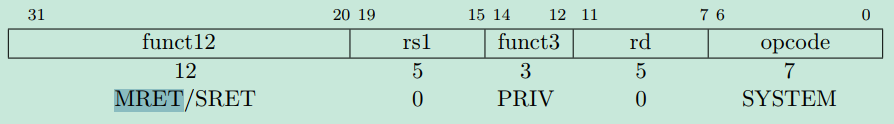
mtval中写入ebreak地址或0；

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；







* + 1. 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。

* + 1. 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；

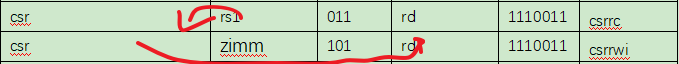
mstatus中，MPIE<=MIE，MIE=0，MPP=privileged；（初始值PRIV\_LVL\_M）

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。

* 1. 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。



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  0xF12  0xF13  0xF14 | MRO  MRO  MRO  MRO | mvendorid  marchid  mimpid  mhartid | Vendor ID. Architecture ID. Implementation ID. Hardware thread ID. |
| Machine Trap Setup | | | |
| 0x300  0x301  0x302  0x303  0x304  0x305  0x306 | MRW  MRW  MRW  MRW  MRW  MRW  MRW | mstatus  misa  medeleg  mideleg  mie  mtvec  mcounteren | Machine status register. ISA and extensions Machine exception delegation register. Machine interrupt delegation register. Machine interrupt-enable register. Machine trap-handler base address. Machine counter enable. |
| Machine Trap Handling | | | |
| 0x340  0x341  0x342  0x343  0x344 | MRW  MRW  MRW  MRW  MRW | mscratch  mepc  mcause  mtval  mip | Scratch register for machine trap handlers. Machine exception program counter. Machine trap cause. Machine bad address or instruction. Machine interrupt pending. |
| Machine Counter/Timers | | | |
| 0xB00  0xB02  0xB03  0xB04  0xB1F  0xB80  0xB82  0xB83  0xB84  0xB9F | MRW  MRW  MRW  MRW  MRW  MRW  MRW  MRW  MRW  MRW | mcycle  minstret  mhpmcounter3  mhpmcounter4  mhpmcounter31  mcycleh  minstreth  mhpmcounter3h  mhpmcounter4h  mhpmcounter31h | Machine cycle counter. Machine instructions-retired counter. Machine performance-monitoring counter. Machine performance-monitoring counter. Machine performance-monitoring counter. Upper 32 bits of mcycle, RV32I only. Upper 32 bits of minstret, RV32I only. Upper 32 bits of mhpmcounter3, RV32I only. Upper 32 bits of mhpmcounter4, RV32I only. Upper 32 bits of mhpmcounter31, RV32I only. |

* 1. 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。

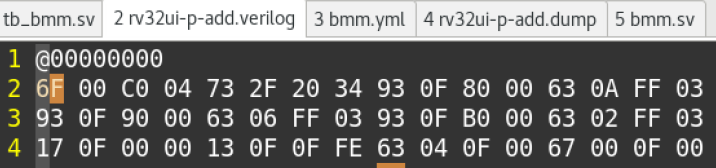
导致exception或mispredict的指令，由于issue内无法做到使wptr=iss\_ptr的同时，清除掉对应sbe中的其他项，以及map\_table没有备份commit时的映射情况，在现有代码量下，只能在commit时响应异常，flush清除掉所有未commit的指令。

1. 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



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；

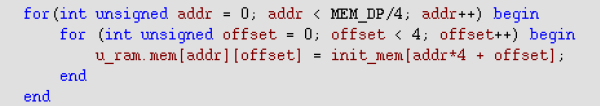




* 数据在mem和ram中的存储顺序

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；

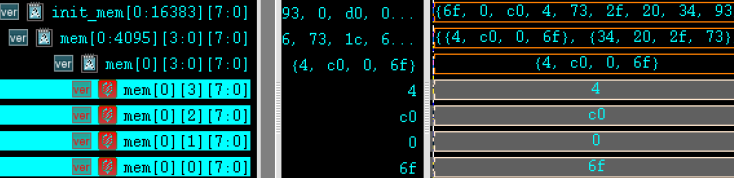
读入到mem，6f存到mem[0],00存mem[1],c0存mem[2]，依次类推；



ram.mem定义4096个行，每行4个byte



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}。

1. 其他

* 每条指令执行的时钟周期，越小越好。
* 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

1. plup
2. 遗留问题

*修改指令取指ram接口，按协议修改*

[RV32I 通用寄存器](https://suda-morris.github.io/blog/cs/riscv.html" \l "rv32i-%E9%80%9A%E7%94%A8%E5%AF%84%E5%AD%98%E5%99%A8)

| **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](https://suda-morris.github.io/blog/cs/riscv.html" \l "%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8%E6%97%B6%E4%BF%9D%E7%95%99%E7%9A%84%E5%AF%84%E5%AD%98%E5%99%A8)

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](https://suda-morris.github.io/blog/cs/riscv.html" \l "%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8%E6%97%B6%E4%B8%8D%E4%BF%9D%E5%AD%98%E7%9A%84%E5%AF%84%E5%AD%98%E5%99%A8)

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”，