RIDECORE：RISC-V DYNAMIC EXECUTE CORE

（翻译：要你命 3000@EETOP）

RIDECORE 概述

本部分将首先解释 RIDECORE 的整体行为，然后，将逐步展示各种决定 RIDECORE 指标的参数。

1. 整体行为

图 1 给出了 RIDECORE 的流水线概况。RIDECORE 拥有 6 段流水线结构，包括取指（IF）、指令译码（ID）、分发（DP）、选择和唤醒（SW）、执行（EX）、完成（COM）。每一个流水段，除了 EX 段的 Load/Store 单元外，都在一个时钟周期内执行完。

在 IF 段，将使用程序计数器（PC）从指令存储器（IMEM）取出最多 2 条指令，并发送到后续流水线。通过在时钟的负沿读取 IMEM，并在时钟的正沿写入 IF/ID 的锁存器，IF 段可以在一个时钟周期内执行完。PC 使用 Gshare 分支预测器被推测地更新。

ID 段译码从 IF 段取到的指令，并将产生的数据发送到后续流水线段。***推测标签产生器（Speculative Tag Gen）***将推测标签指派到指令上。如果存在一条分支指令，推测标签产生器和***错误预测修正表（Miss Prediction Fix Table）***都会被更新。

DP 段首先从体系结构寄存器文件（ARF）和重命名寄存器文件（RRF）取出两条指令所需要的操作数。然后通过在重排序缓冲区和 RRF（在 RIDECORE 中称为 RRFTag）中分配入口，执行寄存器重命名。最后，分配单元将指令执行所需的数据写入到保留站（RS）。数据定向是通过***源操作数管理器（Source Operand Manager）***来执行的。

在 SW 段，发射单元选择已经具有所有操作数的指令，并将它们发射到 EX 段。当一条指令被发射后，它立刻从保留站中移除。

在 EX 段，指令将被执行。当指令执行完后，结果会被写入 RRF。同时，重排序缓冲区被告知一条指令执行完毕。分支单元负责判断分支预测是否正确，并将预测结果写入某些模块以执行某些动作。当出现分支预测错误时，推测执行的指令将通过***失效预测修正表（Miss Prediction Fix Table）***被置为无效。每条指令的 EX 段，除了 Load/Store，将在一个时钟周期内执行完。Load/Store 单元的操作被流水线分为 2 段执行。

在 COM 段，至多 2 条重排序缓冲区中的指令被完成。一条指令被完成时，指令的数据将从 RRF 移动到 ARF。同时更新 ARF 的重命名表。分支预测器也使用新信息进行更新。

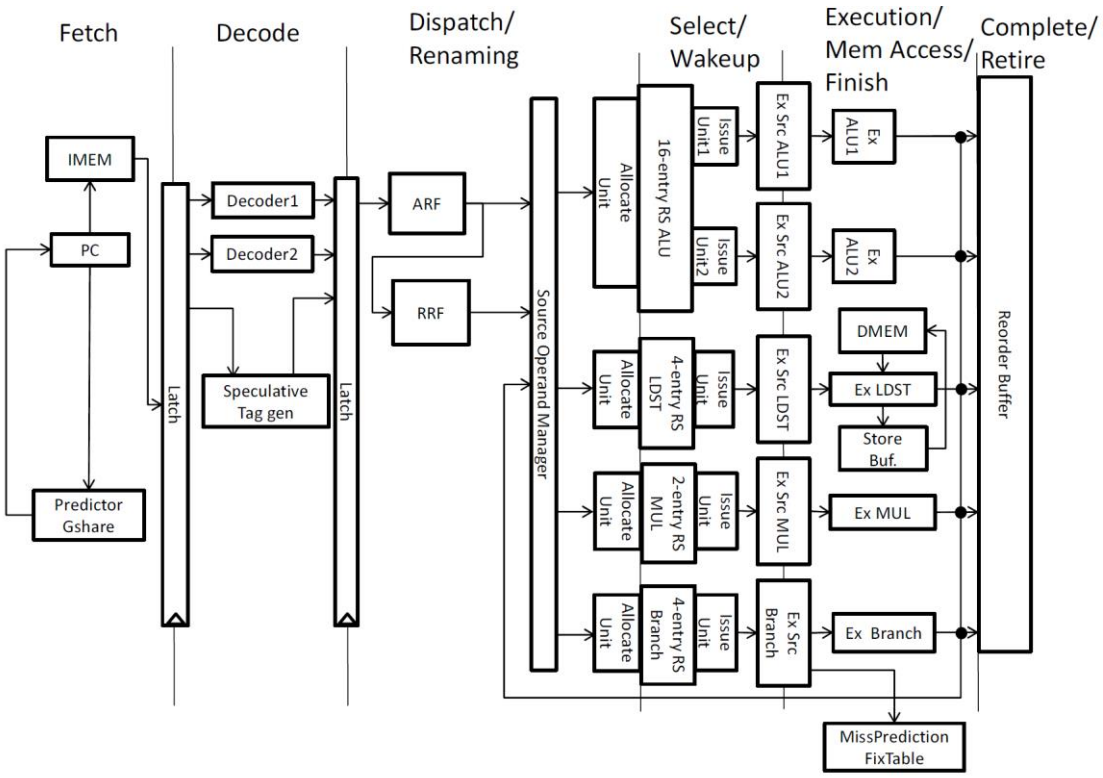


Fig.1 RIDECORE pipeline

2. 规格概况

表 1 给出了 RIDECORE 的规格说明。

Table 1 The specification of RIDECORE

Instruction Set Architecture	RISC-V (A part of RV32IM, see doc/RISC-V-subset.pdf)
Number of ways	2 (Fetch, Decode, Dispatch, Complete)
Width	
Data	32
Address	32
Number of entries	
Architected Register File (ARF)	32
Rename Register File (RRF)	64
Reorder Buffer	64
Number of Checkpoint(Speculative Tag)	5
Number of entries in each Reservation Station	
ALU	16
Load/Store	4
Branch	4
MUL	2
Number of Execution units	
ALU	2
Load/Store	1
Branch	1
MUL	1

硬件模块

本部分介绍 RIDECORE 的各个硬件模块。某些模块更深入的解释，可以在 John Paul Shen 和 Mikko H. Lipasti 的书中找到。此种情况下，我们建议读者首先阅读书中的解释，再来阅读此文档的解释。比如，如果某个模块的解释应当阅读书中从 228 页到 231 页，我们将在解释模块的部分标题上添加（pp. 228-231）。

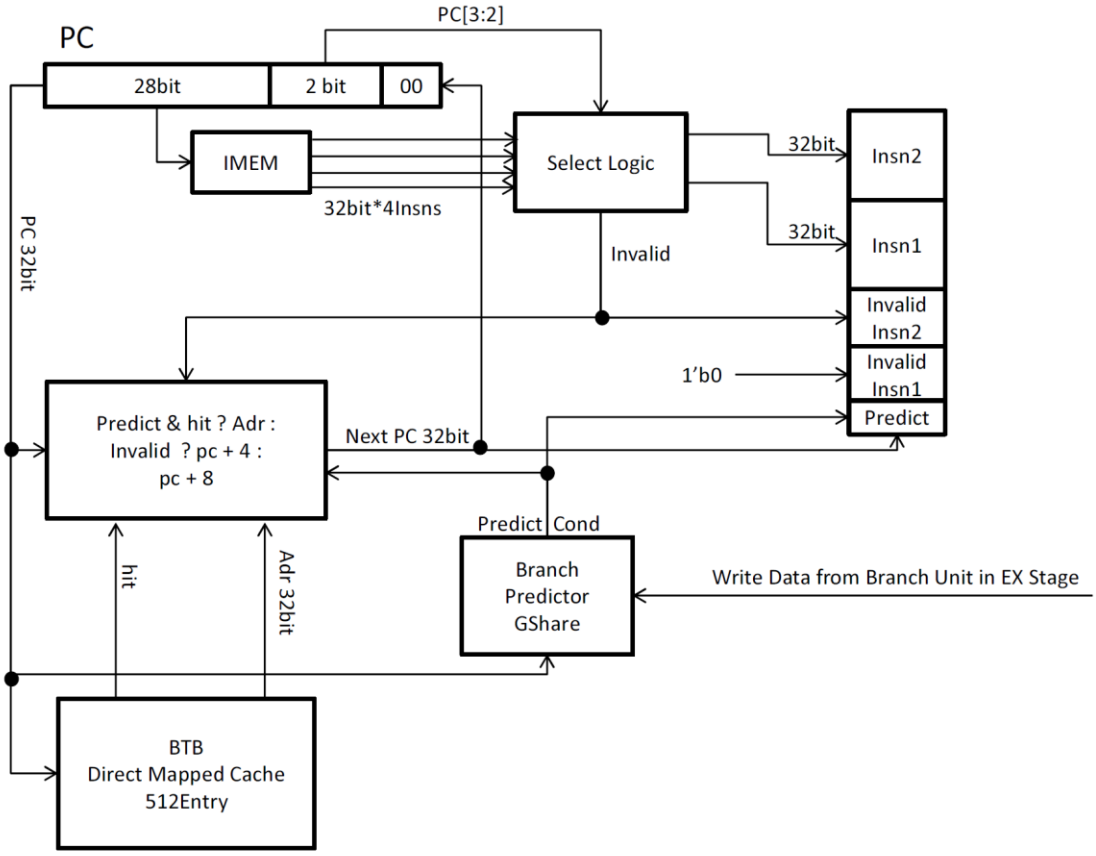


Fig.2 pipe_if

3. pipe_if

在 IF 段，将从 IMEM 中读取 2 条指令。PC 由分支预测器进行更新。图 2 显示了 IF 段的电路。

IMEM 中每一项是 128 位的。因此，由于 RV32IM 是固定 32 位长度的，IMEM 中的每一项可以容纳 4 条指令。

在每个时钟的负沿，pipe_if 使用 PC[31:4]作为 IMEM 的索引，从 IMEM 中读取 4 条指令。然后，***sellog***（图 3）使用 PC[3:2]从 4 条指令中选择 2 条指令，在时钟的正沿写入到 IF/ID 的锁存器中。当 PC 没有被 8 整除时，***sellog*** 将作废第 2 条选中的指令（图 3 中的 Insn2），因为此条指令在前面已经被选择过了。例如，当 PC=0x2C，此时不被 8 整除，4 条从 IMEM 中读出的指令为 0x20、0x24、0x28、0x2C。Insn1 是 0x2C 处的指令，而 Insn2 是 0x20 处的指令，而 0x20 处的指令已经在前面被选择过了（其实我们希望选择的是 0x30 指令）。此种情况下，Insn2 于是要被作废。

为了简化，如图 2 所示，Insn1 作废信号永远是 0。然而，Insn1 作废信号有可能被设置为 1 以便作废掉 IF 段中所有的指令，比如检测到分支预测错误的时

候。

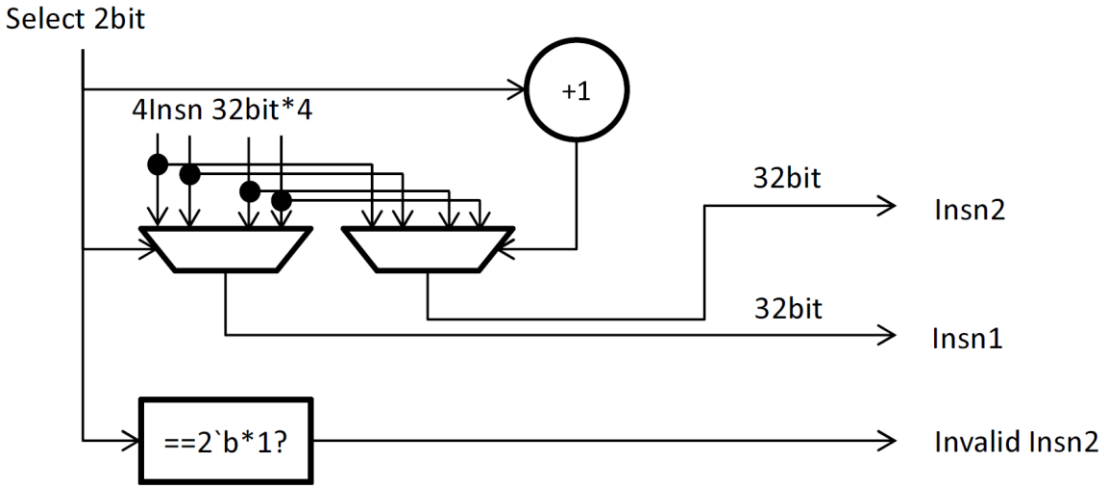


Fig.3 Select logic

4. gshare_predictor（pp. 223-236, 469-472）

图 4 给出了 Gshare 分支预测器电路。它通过使用共享的分支历史寄存器（BHR）、模式历史寄存器（PHT）、PC 和分支目标缓冲区（BTB），在 IF 段预测一条分支指令是否执行。在此，我们仅仅给出一些实现这个预测器的一些设计考虑。

在每个时钟的负沿，我们使用 PC[12:3]⊕BHR 作为读地址，读取 PHT 中的一项。如果读到的数据大于 1，则预测的结果是“分支执行”（Taken），否则为“分支不执行”（Not Taken）。BHR 使用预测的结果来更新。然而，如果预测的结果最后被证实是错误的，BHR 将被恢复为原来的数值。因此，在每次更新前，都需要备份。

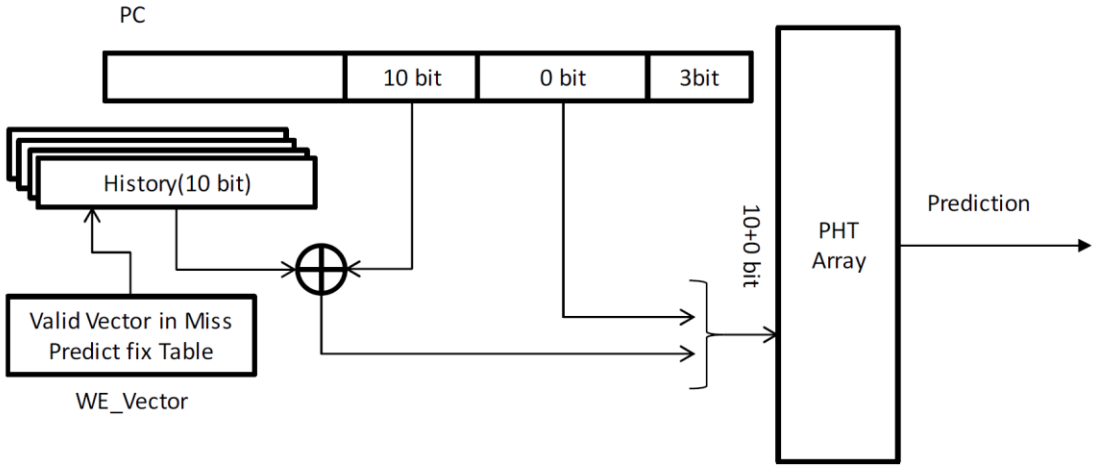


Fig.4 Gshare branch predictor

4.1 pht

PHT 是用于 Gshare 预测器中的一组 2 位饱和计数器。图 5 给出了 PHT 的电路。PHT 将在一条分支指令执行完毕后进行更新。PHT 中的一项将根据分支指令的结果是“分支执行”/“分支不执行”，会“加 1”或者“减 1”。

在 COM 段，PHT 中的一项是这样更新的：首先在时钟负沿读取原来的值，然后在时钟的正沿将更新值写入。PHT 有两个读端口（一个用于 IF 段，另一个用于

COM 段）和一个写端口（用于 COM 段的写入）。在 RIDECORE 中，它使用两个真实的双端口 BRAM 来实现。

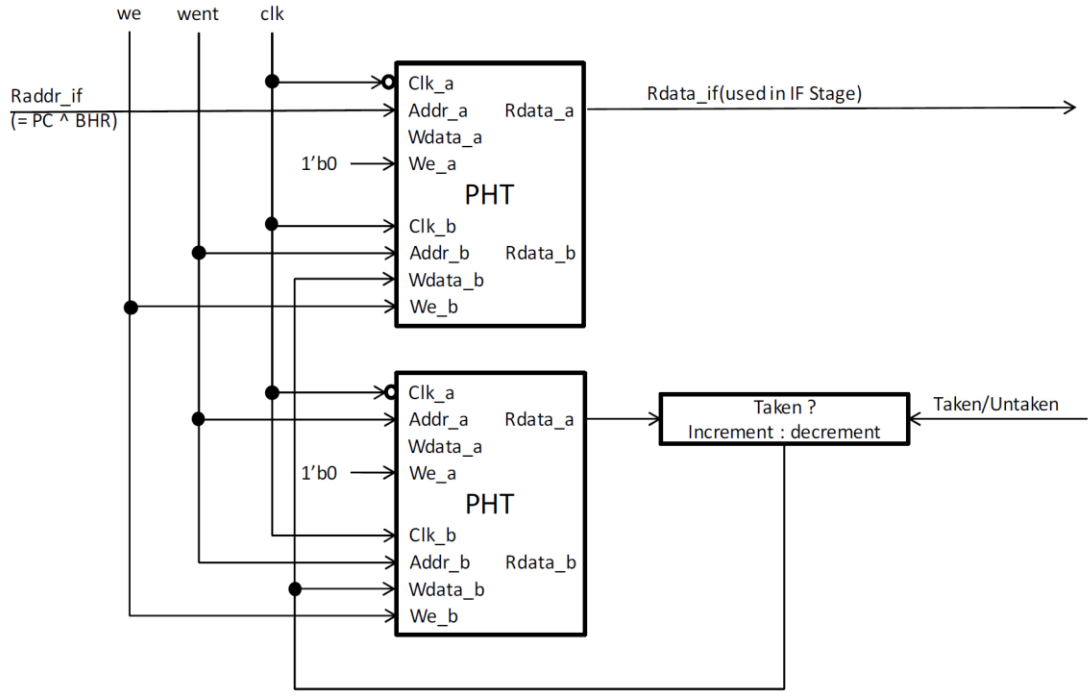


Fig.5 Pattern History Table (PHT)

4.2 btb

分支目标缓冲区（BTB）由成对的分支源和目标地址组成。它被实现为直接映射 Cache 以减少硬件资源使用。正因为如此，预测的效果一般。

5. tag_generator（pp. 228-231）

图 6 给出了标签产生器的电路。这个电路有两个寄存器。一个寄存器保存当前的推测标签（tagreg），另一个寄存器保存当前的分支深度（brdepth）。标签产生器从外界接收 3 个信号：branchvalid1、branchvalid2 和 enable。branchvalid1/branchvalid2 表明 Insn1/Insn2 是否是有效的，而 enable 表明这些指令是否可以发射。使用这些信号，标签产生器产生推测标签（sptag1/2）、推测标记（speculative1/2），并更新 tagreg 和 brdepth。

推测标签被指派到指令上，按照 00001、00010、…、10000（从 00001 左循环）的顺序。如果没有可用的推测标签，标签产生器将把 attachable 标志设置为 0，并暂停标签指派流程。

当分支预测正确时（prsuccess==1），brdepth 将减 1 释放掉推测标签。当分支预测错误时（prmiss==1），tagreg 和 brdepth 将被恢复到预测执行前的值。tagreg 使用 tagregfix（由分支单元产生）来恢复，brdepth 被置为 0，因为分支指令是按序执行的（在一条推测指令被作废后，没有剩余的推测指令）

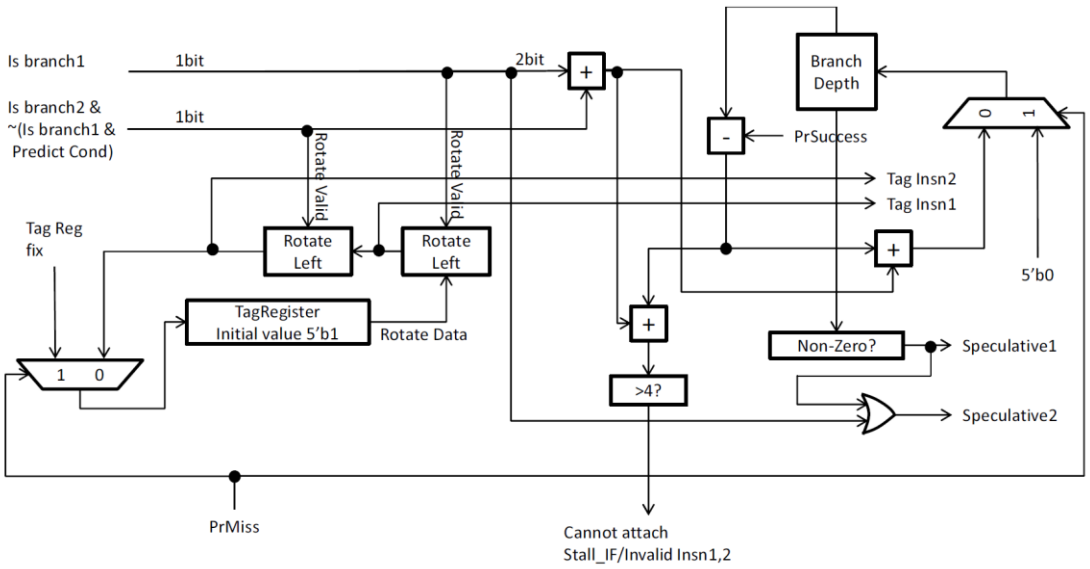


Fig.6 Speculative Tag Generator

6. decoder

译码器是 ID 段使用的一个模块。此模块使用输入的指令信息来产生后续流水线段（DP、EX、…）所需的数据。图 7 给出了译码器的 I/O 定义。

下面是输出信号的解释：

- imm_type：这个信号用于 imm_gen，用于表明指令中立即数的格式。
- rs1、rs2、rd：第 1 源操作数、第 2 源操作数和目标寄存器编号。
- src_a_sel、src_b_sel：用于选择 ALU 的操作数。
- wr_reg：此信号表明指令是否将要把数据写入目标寄存器。
- uses_rs1、uses_rs2：rs1 和 rs2 的有效信号。如果 rs1/rs2 有效，将需要取数据。这些信号用于防止取到不需要的数据（指令无法发射，直到取到所有需要的数据）。
- illegal_instruction：此信号表明指令不是本处理器定义的指令。它将在未来用于异常处理。
- alu_op：ALU 操作类型。
- rs_ent：保留站 ID。每个执行单元包含一个保留站，其 ID 定义如下：
  - ALU：1
  - BRANCH UNIT：2
  - MULTIPLIER：3
  - LOAD/STORE：4
- dmem_size、dmem_type：决定 Load/Store 数据的大小（4 字节/2 字节/1 字节）。这些信号并没有使用，因为 RIDECORE 只支持 4 字节的 Load/Store 指令。
- md_req_op：操作类型（乘法、除法或者取模）。这个信号并没有使用，因为 RIDECORE 不支持除法和取模运算。
- md_req_in_1_signed、md_req_in_2_signed：这些信号表明乘法运算的第 1 个源操作数、第 2 个源操作数是否是有符号的。
- md_req_out_sel：选择高/低。在 RIDECORE 中，乘法器的输出是 64 位的，而 ARF 和 RRF 中的每个寄存器都是 32 位的。这个信号决定乘法器输出的

结果哪一部分被作为最终的结果：是高 32 位，还是低 32 位。

```
module decoder
  (
   input wire [31:0]                    inst,
   output reg [`IMM_TYPE_WIDTH-1:0]     imm_type,
   output wire [`REG_SEL-1:0]           rs1,
   output wire [`REG_SEL-1:0]           rs2,
   output wire [`REG_SEL-1:0]           rd,
   output reg [`SRC_A_SEL_WIDTH-1:0]    src_a_sel,
   output reg [`SRC_B_SEL_WIDTH-1:0]    src_b_sel,
   output reg                           wr_reg,
   output reg                           uses_rs1,
   output reg                           uses_rs2,
   output reg                           illegal_instruction,
   output reg [`ALU_OP_WIDTH-1:0]       alu_op,
   output reg [`RS_ENT_SEL-1:0]         rs_ent,
   output wire [2:0]                    dmem_size,
   output wire [`MEM_TYPE_WIDTH-1:0]    dmem_type,
   output reg [`MD_OP_WIDTH-1:0]        md_req_op,
   output reg                           md_req_in_1_signed,
   output reg                           md_req_in_2_signed,
   output reg [`MD_OUT_SEL_WIDTH-1:0]   md_req_out_sel
   );
```

Fig.7 Decoder module Interface

7. arf（pp. 239-244）

体系结构寄存器（ARF）中的每一项，都由一个被 COM 段更新的寄存器（数据）、其他信息（RRFTag、Busy）构成。图 8、图 9、图 10 给出了 ARF 的电路和行为。重命名表包含了重命名信息。当一个分支预测错误时，它要被恢复到预测前的状态。因此，重命名表的数量等于推测标签的数量。下面我们将解释 ARF 的读写操作，还有当分支预测错误时，是如何恢复重命名表的。

图 8 给出了 ARF 的读行为。最新的重命名表（黑色表）中的数据被用作当前的重命名信息。图 9 给出了 ARF 的写行为。写请求产生于 DP 段和 COM 段。重命名表的写使能信号 we，是失效修正表（mpft）中的 valid 向量的取补。请阅读“miss_prediction_fix_table”章节以获知更多信息。

图 10 给出了重命名表的恢复行为。当一条分支指令处在分支单元，并且分支条件已经计算出来时，将会执行恢复动作。重命名表中的 RRFTag 和 Busy 被实现为 32 位的向量，因此可以在一个周期内重写所有项。当恢复正在进行时，流水线是停顿的，因为此时不能从重命名表中读取到正确的数据。

图 11 给出了一个恢复的例子。如果 branch1（SpeculativeTag=5’b00010）的预测是错误的，我们将把重命名表恢复到 SpeculativeTag=5’b00001 时的状态。于是，Fix Vector 被设置为 6’b111111，并且红色的重命名表被选中，作为 Fix Data。

另一方面，如果 branch1 的预测是正确的，备份表（红色表，SpeculativeTag=5’b00001）的内容将变为无用的了。我们将用黑色表格中的最新数据覆盖此内容，以便备份当前状态。于是，Fix Vector 被设置为 6’b010000，并且黑色的重命名表被选中，作为 Fix Data。

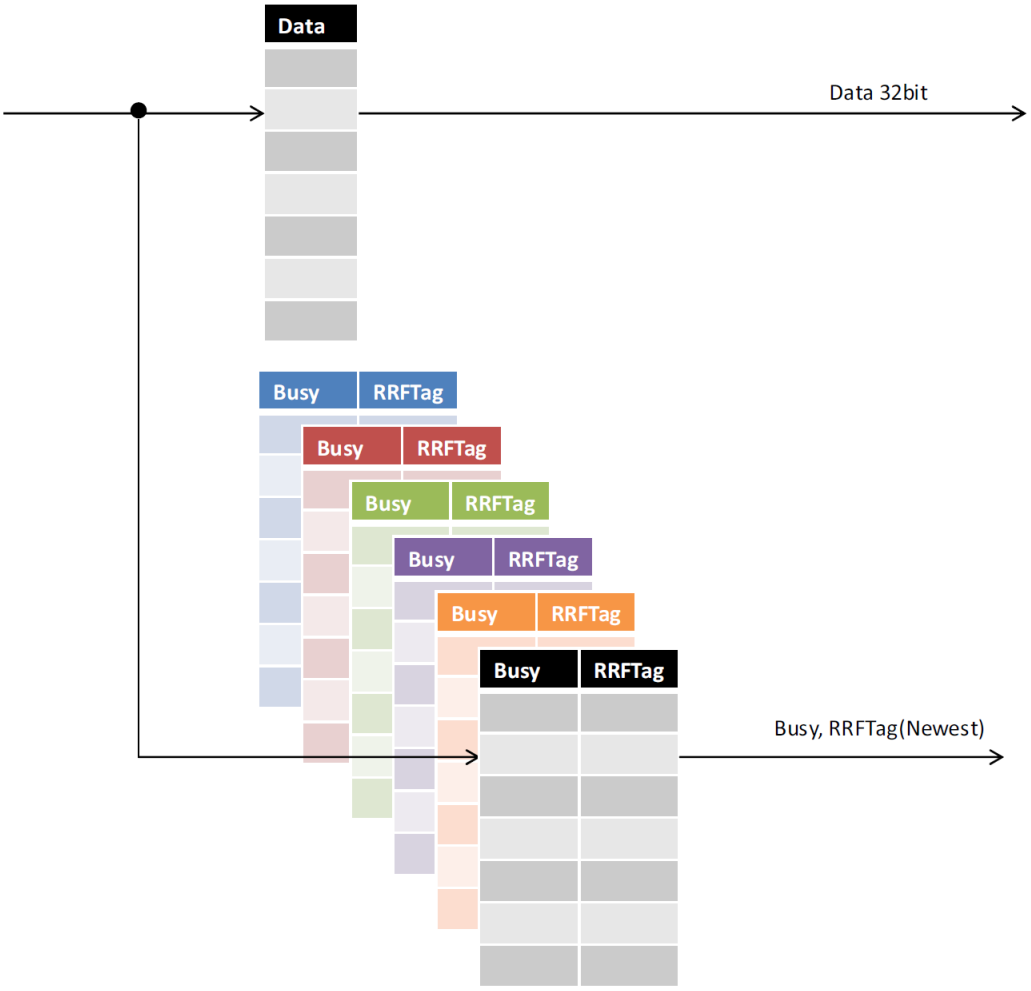


Fig.8 ARF Read

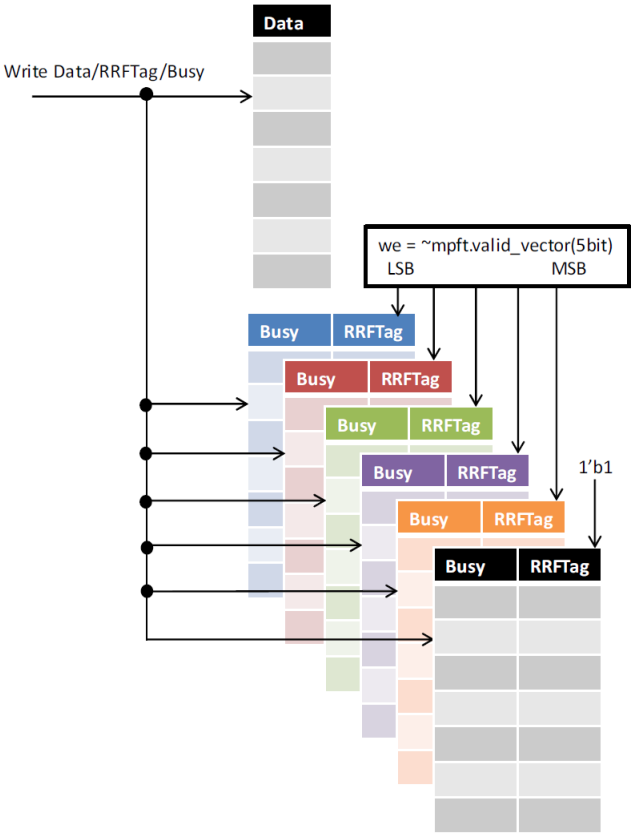


Fig.9 ARF Write

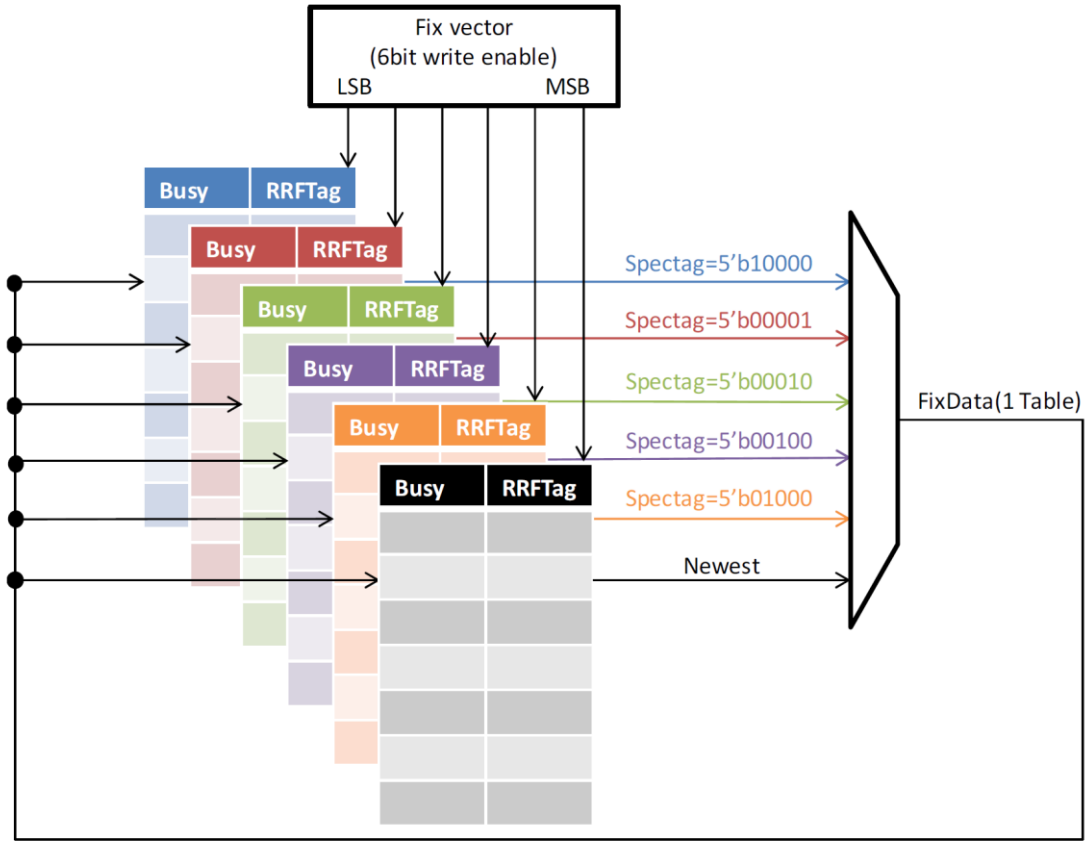


Fig.10 Fix Renaming Table

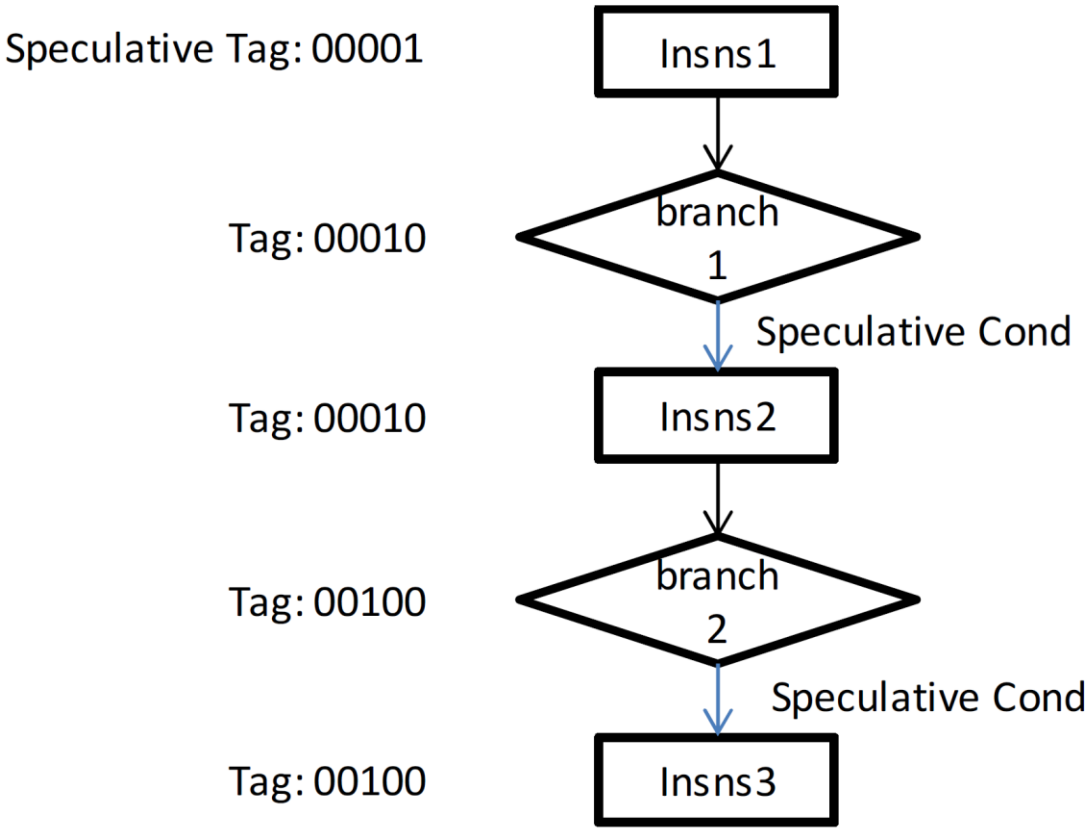


Fig.11 Example of Speculative Exection

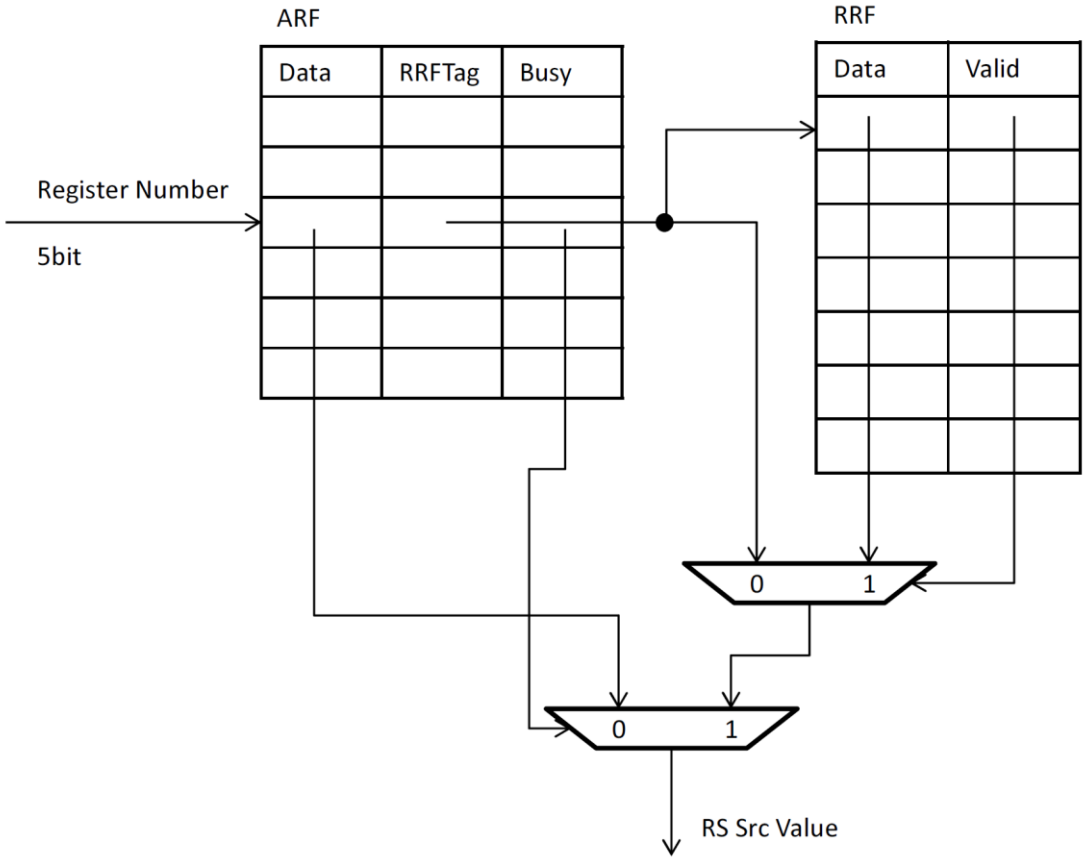


Fig.12 Register Renaming using ARF and RRF

8. rrf（pp. 239-244）

重命名寄存器文件（RRF）中的一项有两个属性：RRFData 和 RRFValid。RRFData 保存着重命名寄存器的值。RRFValid 决定此项是否是有效的。在 RIDECORE 中，RRF 的大小为 64 项，与重排序缓冲区的大小相同。事实上，RRF 和重排序缓冲区是一一对应的。于是用于访问 RRF 的标签，也可用于访问重排序缓冲区。

9. sourceoperand_manager（pp. 239-244）

sourceoperand_manager 从请求的寄存器编号、ARF 和 RRF 的信息、inst1 的 RRFTag，产生写入保留站（RS）的数据。图 12 给出了使用 ARF 和 RRF 实现寄存器重命名的电路。

表 2 给出了一个请求寄存器的 3 种可能状态，以及每种状态下分发到保留站中的数据。假设请求寄存器是 regsrc，它在 ARF 和 RRF 中的状态分别是 ARF.Busy 和 RRF.Busy。当且仅当 regsrc 不是任何未完成指令的目标寄存器时，ARF.Busy 等于 0。此种情况下，regsrc 的最新值在 ARF 中。另一方面，如果 ARF.Busy 等于 1，那么存在一条未完成的指令，它的目标寄存器是 regsrc。RRF.Valid 决定这条指令的 EX 段是否已经完成。当 EX 段完成时，结果被写到 RRF，并且 RRF.Valid 被设置为 1。当 RRF.Valid 被设置为 0 时，即 EX 段的执行还没有完成，RRFTag 被送到 RS，以便在后面（当以 regsrc 作为目的寄存器的这条指令所有流水线段都执行完）将 regsrc 的最新值从 RRF 传送到 RS。

Table 2 Three possible states of a requested register and the data dispatched to Reservation Station in each case.

ARF.Busy	RRF.Valid	State	Write data to RS
0	*	Available in ARF	ARF.Data
1	1	Available in RRF	RRF.Data
1	0	Not available	RRFTag

图 13 给出了 sourceoperand_manager 的源代码。src 是分发给保留站的数据。当 rdy 等于 0 时，src 被设置为 RRFTag。当请求寄存器是 0 号寄存器（如同 MIPS ISA 一样，RISC-V ISA 定义了一个特殊的 0 寄存器）时，src 被设置为 0。当 inst2 的其中一个源寄存器是 inst1 的目的寄存器时（src_eq_dst1 等于 1），src 被设置为 inst1 的 RRFTag 值。除此之外的情形，此模块的操作如表 2 所示。

```
module sourceoperand_manager
  (
   input wire [`DATA_LEN-1:0]  arfdata,
   input wire                  arf_busy,
   input wire                  rrf_valid,
   input wire [`RRF_SEL-1:0]   rrftag,
   input wire [`DATA_LEN-1:0]  rrfdata,
   input wire [`RRF_SEL-1:0]   dst1_renamed,
   input wire                  src_eq_dst1,
   input wire                  src_eq_0,
   output wire [`DATA_LEN-1:0] src,
   output wire                 rdy
   );

   assign src = src_eq_0 ? `DATA_LEN'b0 :
                src_eq_dst1 ? dst1_renamed :
                ~arf_busy ? arfdata :
                rrf_valid ? rrfdata :
                rrftag;
   assign rdy = src_eq_0 | (~src_eq_dst1 & (~arf_busy | rrf_valid));

endmodule // sourceoperand_manager
```

Fig.13 Source Operand Manager

10. rrf_freelistmanager（pp. 239-244）

图 14 给出了 rrf_freelistmanager 的电路，用于管理 RRF 和重排序缓冲区的空闲项。有两个寄存器：FreeNum 和 RRFPtr。FreeNum 保存了空闲的项数目。RRFPtr 是当前 RRF 和重排序缓冲区的当前项编号。

此电路在 DP 段将 RRFTag 指派给指令。当 FreeNum 小于请求写保留站的指令的数目时，IF 段和 ID 段都被暂停，因为没有指令可以被分派。当一条指令在 COM 段完成后，此指令使用的 RRFTag 将被释放，FreeNum 也会加 1。当一条分支指令预测错误时，所有推测执行指令的 RRFTag 也将被释放。为了实现此操作，RRFPtr 被置为 rrfTag fix，rrfTag fix 是导致预测失败的分支指令的下一条指令的 RRFTag。FreeNum 也通过使用 rrfTag fix 和来自重排序缓冲区的 comptr 进行恢复。

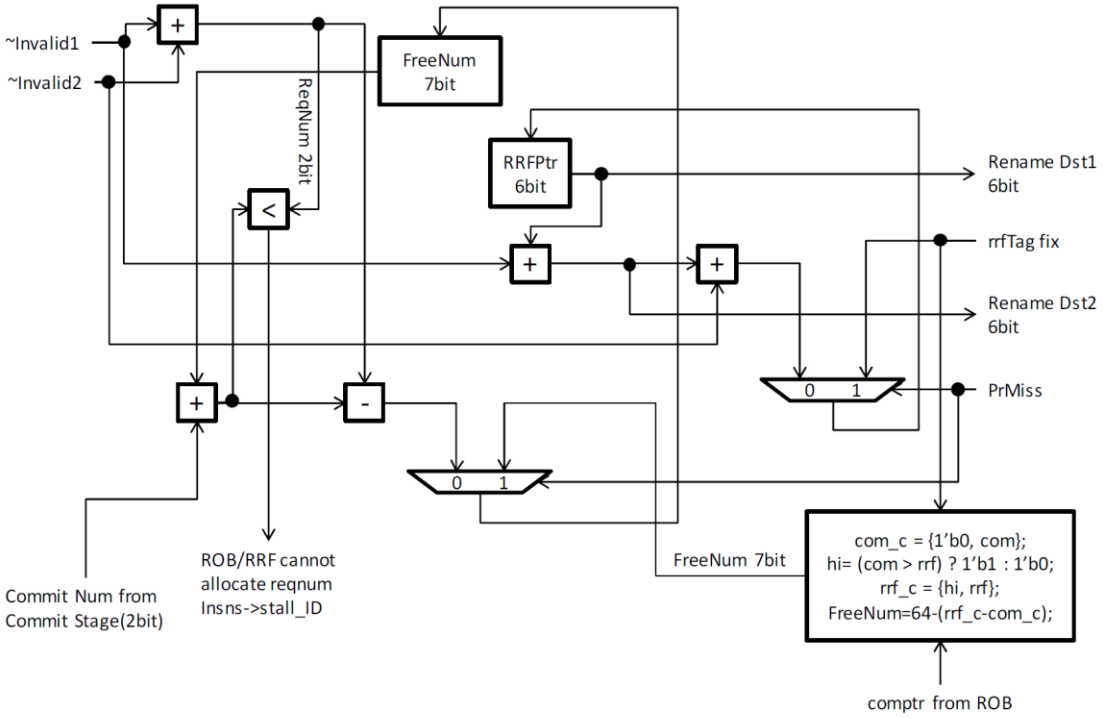


Fig.14 RRF Free List Manager

11. src_manager（pp. 256-259）

图 15 给出了 sourcemanager 的 Verilog HDL 源代码。此模块用于将 DP 段的结果定向到在保留站中等待的请求操作数。此处，我们解释一下它的输入、输出信号：

- opr：DP 段的操作数（从 sourceoperand_manager 到保留站）。如果 opr_rdy 等于 0，opr 包含了产生请求值的 RRFTag，否则，opr 包含了请求值。
- opr_rdy：决定 opr 中是什么值。
- exrslt*、exdst*、kill_spec*：5 个执行单元（ALU1/2、Load/Store 单元、分支单元、乘法器）的输出。当 exdsti 等于 opr，并且 opr_rdy 和~kill_spec* 都是 0 时，src 被设置为 exrslti。
- src、resolved：可能的时候，opr、opr_rdy 将定向到这些信号。

```
module src_manager
  (
   input wire [`DATA_LEN-1:0]  opr,
   input wire                  opr_rdy,
   input wire [`DATA_LEN-1:0]  exrslt1,
   input wire [`RRF_SEL-1:0]   exdst1,
   input wire                  kill_spec1,
   input wire [`DATA_LEN-1:0]  exrslt2,
   input wire [`RRF_SEL-1:0]   exdst2,
   input wire                  kill_spec2,
   input wire [`DATA_LEN-1:0]  exrslt3,
   input wire [`RRF_SEL-1:0]   exdst3,
   input wire                  kill_spec3,
   input wire [`DATA_LEN-1:0]  exrslt4,
   input wire [`RRF_SEL-1:0]   exdst4,
   input wire                  kill_spec4,
   input wire [`DATA_LEN-1:0]  exrslt5,
   input wire [`RRF_SEL-1:0]   exdst5,
   input wire                  kill_spec5,
   output wire [`DATA_LEN-1:0] src,
   output wire                 resolved
   );

   assign src = opr_rdy ? opr :
                ~kill_spec1 & (exdst1 == opr) ? exrslt1 :
                ~kill_spec2 & (exdst2 == opr) ? exrslt2 :
                ~kill_spec3 & (exdst3 == opr) ? exrslt3 :
                ~kill_spec4 & (exdst4 == opr) ? exrslt4 :
                ~kill_spec5 & (exdst5 == opr) ? exrslt5 : opr;

   assign resolved = opr_rdy |
                     (~kill_spec1 & (exdst1 == opr)) |
                     (~kill_spec2 & (exdst2 == opr)) |
                     (~kill_spec3 & (exdst3 == opr)) |
                     (~kill_spec4 & (exdst4 == opr)) |
                     (~kill_spec5 & (exdst5 == opr));

endmodule // src_manager
```

Fig.15 Source Manager

12. imm_gen，brimm_gen

在 RISC-V ISA 中，某些指令在它们的操作数中包含了立即数。imm_gen 和 brimm_gen 从译码后的数据（imm_type）和指令数据中，产生 32 位的有符号立即数值。brimm_gen 用于分支指令，而 imm_gen 用于其他指令。

13. rs_requestgenerator（pp. 254-259）

保留站接收 2 条指令（inst1/2）的数据，以及 2 个写使能信号。写使能信号由 rs_requestgenerator 模块产生。图 16 给出了这个模块的电路。这个电路从译码器产生的 RSNum1/2 中，产生 Req1/2（inst1/2 的写使能信号）。

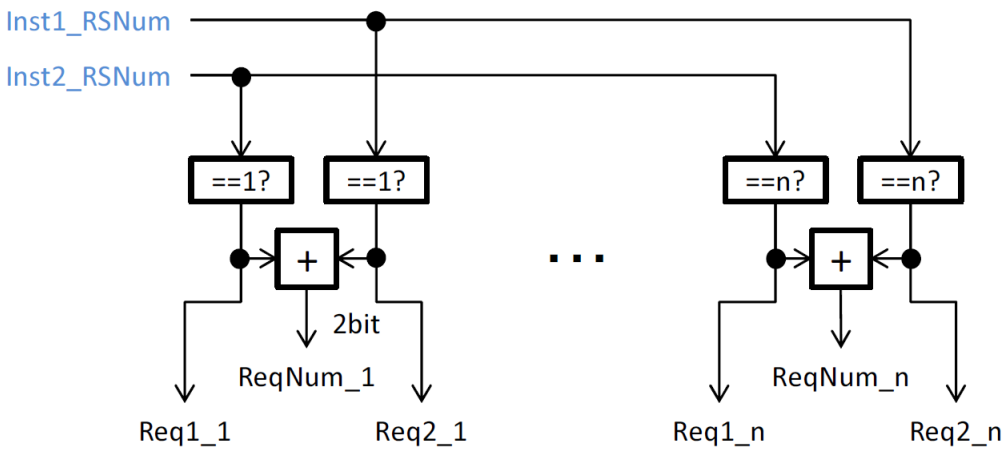


Fig.16 RS Request Generator

14. reservation station(rs_*)（pp. 199-203, 254-261）

保留站（RS）负责寄存器指令操作数的获得和存储。每个执行单元都有一个保留站。分配单元将指令分配给保留站。发射单元将指令从保留站发射到执行单元中。

图 17 给出了保留站的 I/O 和它与分配单元、发射单元的接口关系。当分配单元收到写使能信号（we1/2）时，它在保留站中查找空闲的项，并产生写地址（waddr1/2）。当在保留站中有一条指令可以执行，并且相应的执行单元在下一个周期空闲的话，发射单元将把此指令发射到执行单元。发射单元产生这条指令的项编号（raddr），保留站输出数据（rdata）都执行单元。更详细的分配/发射操作解释，可以在下一部分看到（分配单元和发射单元）。

图 18 给出了保留站中一项的电路。橙色方块表示寄存器。Write data 是执行寄存器指令所需要的数据（操作数、操作码、RRFTag 等）。执行结果通过 src manager 被定向到两个寄存器 Opr1 和 Opr2。下面解释保留站中的寄存器。

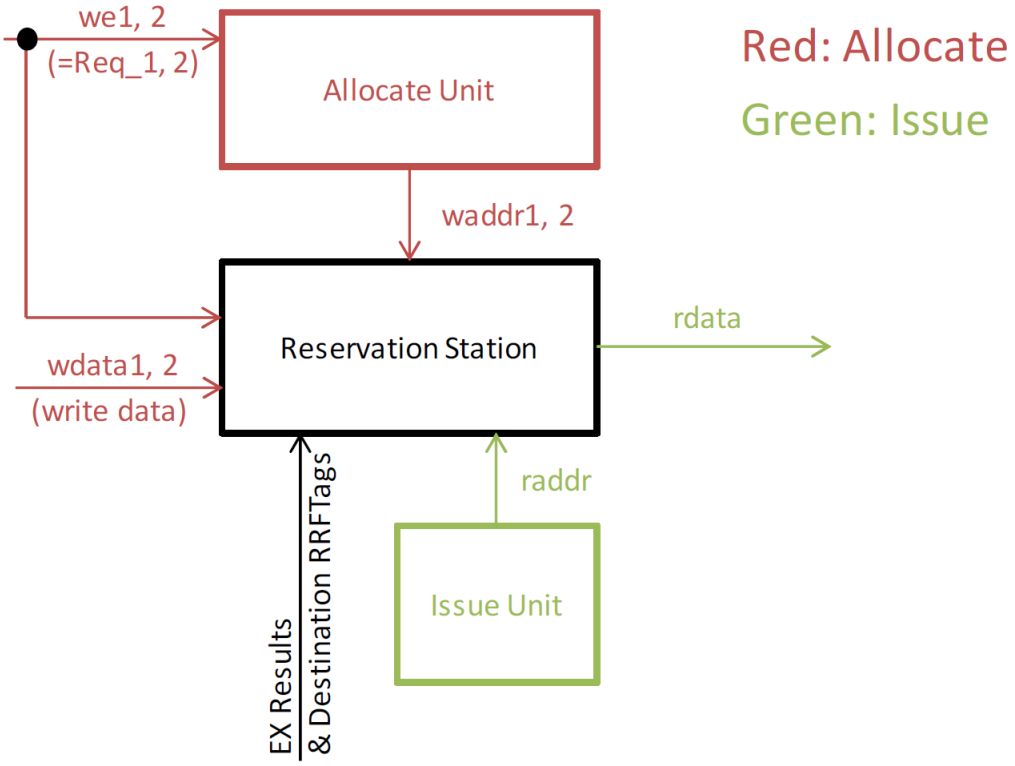


Fig.17 Reservation Station (with Allocate Unit and Issue Unit)

- Opr1、Opr2：第 1 个、第 2 个寄存器操作数。这些数据根据 valid 位的不同，有不同的含义：
  - Valid1/2=1：Opr1/Opr2 就绪。如果指令不使用 Opr1/Opr2，那么 Valid1/2 也等于 1
  - Valid1/2=0：Opr1/Opr2 没有就绪，不能执行指令。此时，Opr1/Opr2 包含了用于生成所需操作数的 RRFTag
- Busy：决定指令是否是寄存器的
- Other Data：执行指令所需的其他数据
  - imm：立即数值
  - rrftag：RRFTag
  - dstval：决定指令是否需要写数据到 ARF
  - specbit、spectag：specbit 决定指令是否是推测执行的，spectag 包含了给该指令分配的推测标签

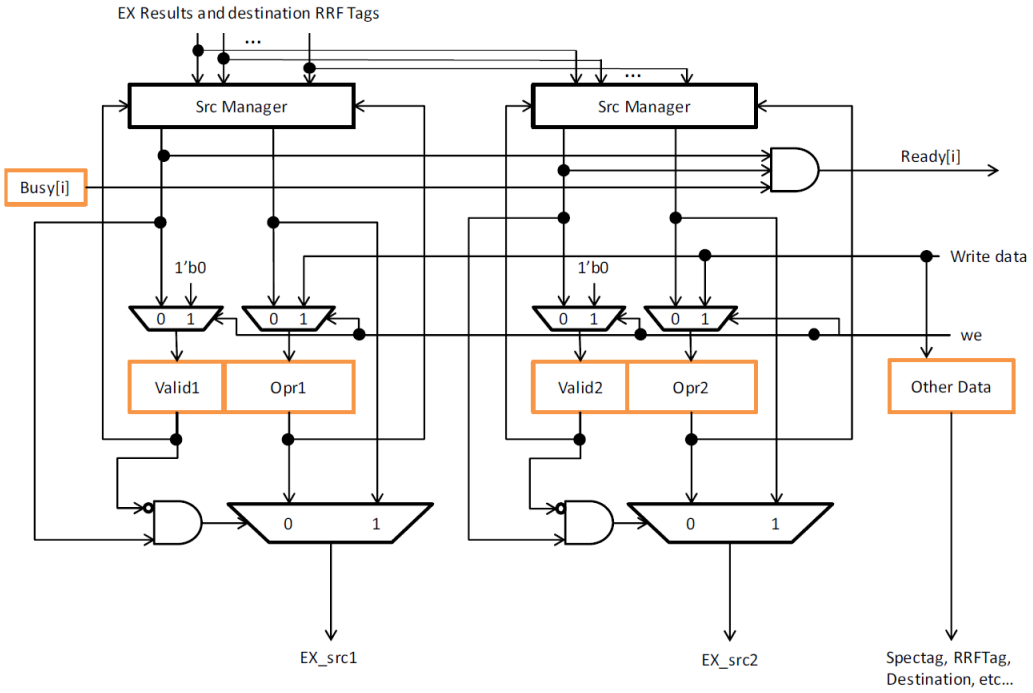


Fig.18 An entry in Reservation Station

Table 3 Allocate Pattern

Req1	Req2	waddr1	waddr2
0	0	*	*
1	0	Free_ent1	*
0	1	*	Free_ent1
1	1	Free_ent1	Free_ent2

15. 分配单元和发射单元（pp. 254-261）

15.1 乱序发射

图 19 给出了分配单元的电路。图 20 给出了分配单元中的 ***RS 空闲项查找器***电路。分配单元像一个保留站的地址解释器一样工作。首先，分配单元接收到保留站的写请求信号 Req1/2。然后，分配单元使用 ***RS 空闲项查找器***在保留站中查找空闲项（Free_ent1/2），将指令数据写入到找到的空闲项中。

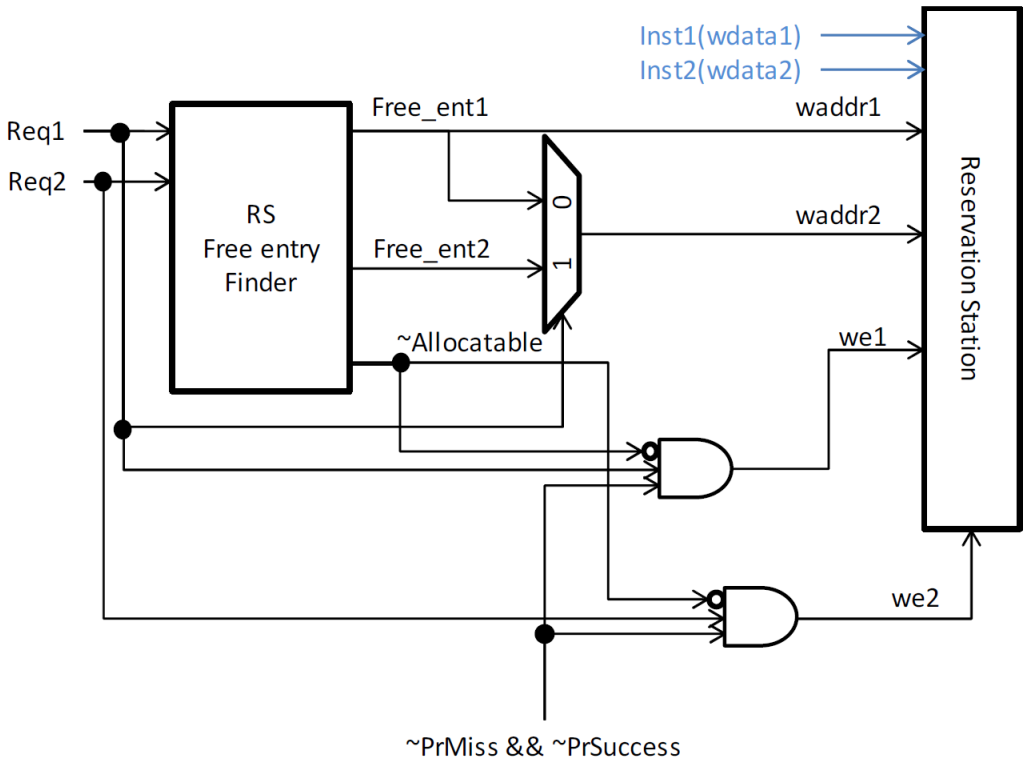


Fig.19 Allocate Unit

RIDECORE 中的保留站接口与 4 端口（2 读 2 写）存储器相同。wdata1/2、waddr1/2、we1/2 分别是写数据、写地址和写使能信号。表 3 给出了 waddr1、waddr2 是如何决定的。

RS 空闲项查找器使用两个优先级编码器来查找两个空闲项。第 1 个优先级编码器的输入是保留站 busy 向量的取反（~Busy0-~BusyN-1）。为了避免选中已经被第 1 个优先级编码器选中的空闲项，第 2 个优先级编码器的输入通过 Mask 单元进行掩码。Entry_en1/2 决定了 Free_ent1/2 是否是有效的。当 Entry_en1+Entry_en2 不小于 Req1+Req2 时，allocatable 变为为 1。

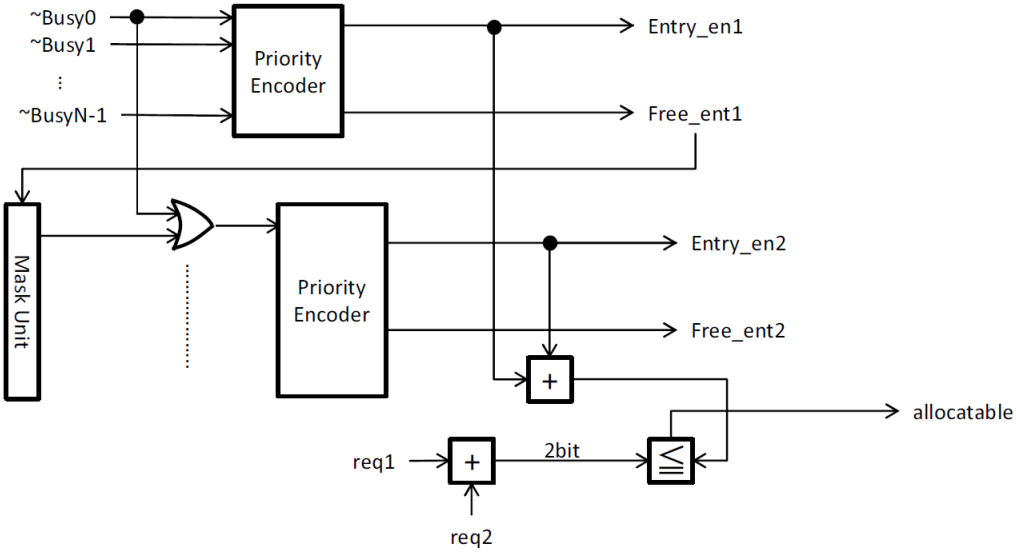


Fig.20 RS Free Entry Finder

发射单元选中一条具备了所有所需操作数的指令，并将其发射到 EX 段。当

一条指令被发射时，它会立即从保留站中删除。我们使用了 Oldest First 算法来选择发射一条指令。图 21 给出了用于实现 Oldest First 算法的最小值选择电路（RIDECORE 中称之为 oldest finder）。

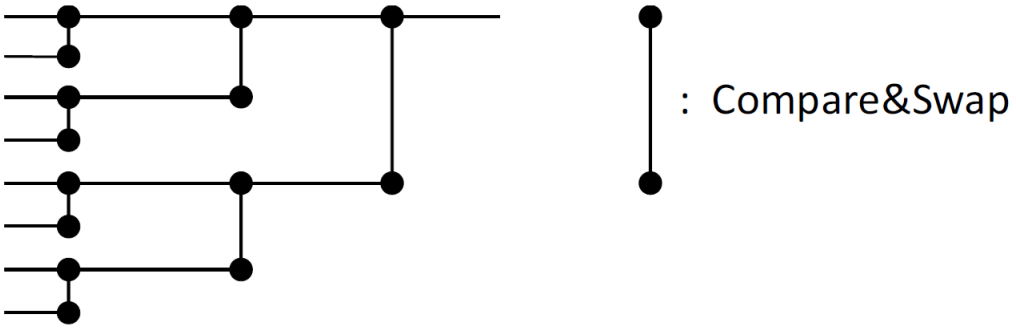


Fig.21 Minimum Selection (Oldest Finder)

图 22 给出了 oldest finder 中一项的格式。一项中的 MSB 是~rdy，因为发射单元必须优先处理已经具有所有操作数的指令。由于 RRFTag 是从 0 开始指派的，并且随着时间推移而不断增加，因此我们可以通过比较 RRFTag 来选择最老的指令。然而，当 RRFTag 溢出时，这就不对了。于是，如图 22 所示，我们在 MSB 和 RRFTag 之间插入了一个排序位。当一条指令被分派到保留站时，它的排序位被置为 1。当 RRFTag 溢出时，保留站中所有的指令的排序位都被置为 0。通过这种方法，发射单元基本上就可以选中具有所有操作数的最老的指令。然而，这有一个限制。在 RIDECORE 中，每个时钟周期，有 2 条指令可以分派到保留站。于是，当 2 条指令的 RRFTag 分别是 63 和 0 时（由于 RRFTag 是 6 位的，它在变成 63 之后将溢出），具有 RRFTag=63 的指令的排序位也被置为 0。这意味着发射单元并不是严格的实现了 Oldest First 算法。

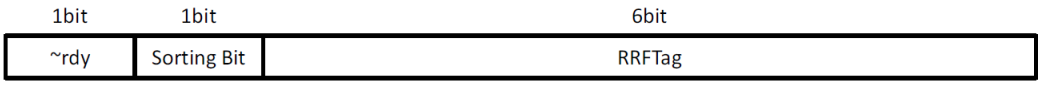


Fig.22 An entry in *oldest finder*

15.2 按序发射

图 23 给出了 alloc_issue_ino 电路，它将保留站作为 FIFO 缓冲区使用，实现了按序执行。在此电路中，只有一个 AllocPtr 寄存器。IssuePtr 指向了将要发射的下一条指令，它是通过保留站的 busy 向量计算得出的，并寄存为 AllocPtr。图 24 给出了 IssuePtr 是如何计算的。图中的 b0/1、e0/1 分别由 search_begin 和 search_end 计算得出（稍后详述）。b0/1 是 busy 向量中第一个 0/1 项的项编号，而 e0/1 是 busy 向量中最后一个 0/1 项的项编号。busy 向量有三种模式：

- 模式 1：busy 向量中的所有项都是 1。此时，IssuePtr 等于 AllocPtr
- 模式 2：b0 和 e0 处在 b1 和 e1 之间。此时，IssuePtr 等于 e0+1
- 模式 3：b1 和 e1 处于 b0 和 e0 之间。此时，IssuePtr 等于 b1

当出现一条分支预测错误时，AllocPtr 必须重新计算，因为此时在保留站中的一些推测执行指令是无效的。AllocPtr 的重新计算，是基于 Prbusyvector_next（作废指令之后的 busy 向量）进行的。这个重新计算，几乎和 IssuePtr 的计算完全一样。

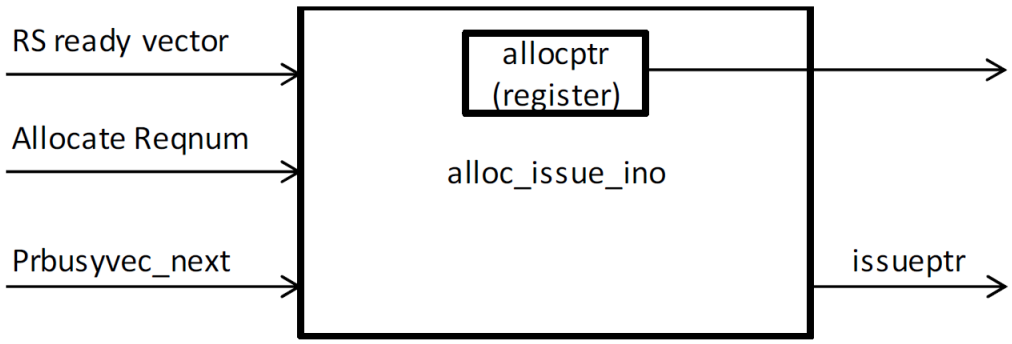


Fig.23 Allocate and Issue In Order

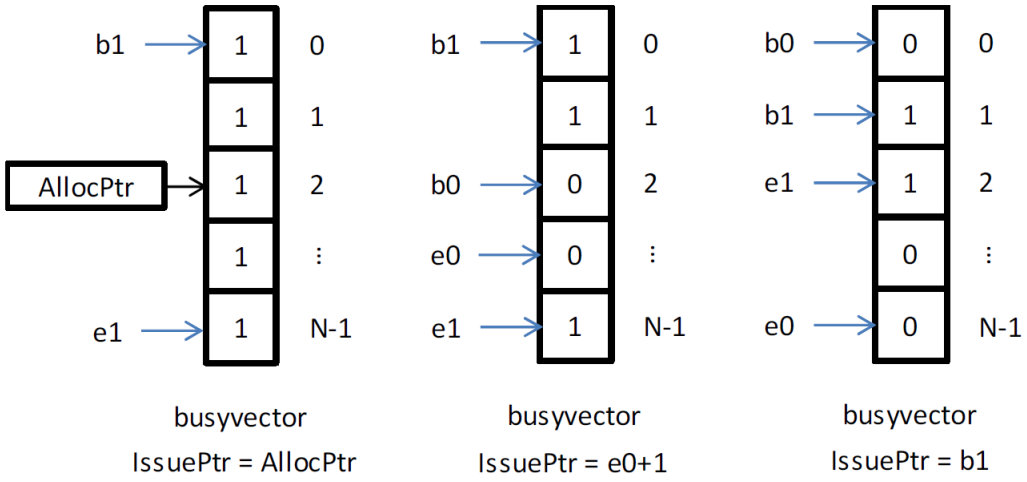


Fig.24 IssuePtr Calculation

15.3 search_begin、search_end

search_begin 和 search_end 都是实现为优先级编码器。然而，search_begin 是低位优先，而 search_end 是高位优先。这两个模块用于 alloc_issue_ino 和 storebuf 模块，以实现按序执行。

16. 执行单元（exunit_*）（pp. 203-206）

RIDECORE 有 5 个执行单元：2 个 ALU、1 个乘法器、1 个 Load/Store 单元和 1 个分支单元。图 26、27、28 给出了这些执行单元的电路。图 29 给出了在图 26、27、28 中使用的“Kill Gen”电路。当出现一条分支预测错误时，Kill Gen 决定是否需要作废掉当前正在执行单元中执行的指令。你可以通过阅读 Miss Prediction Fix Table 部分来理解此电路。在 RIDECORE 中，每一个乘法操作需要花费 1 个时钟周期。因此乘法器的电路几乎和 ALU 一样。

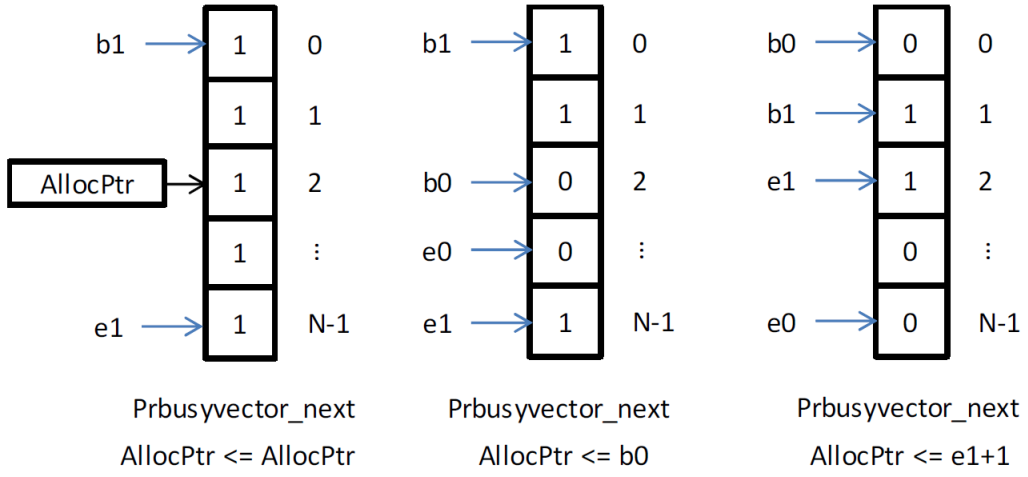


Fig.25 AllocPtr Re-calculation

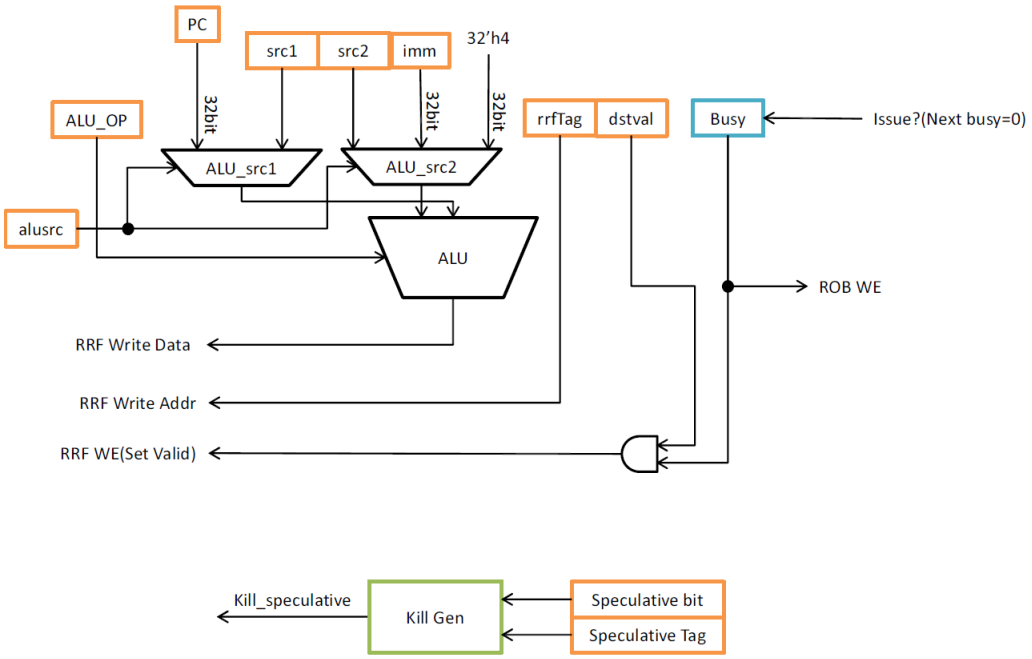


Fig.26 ALU

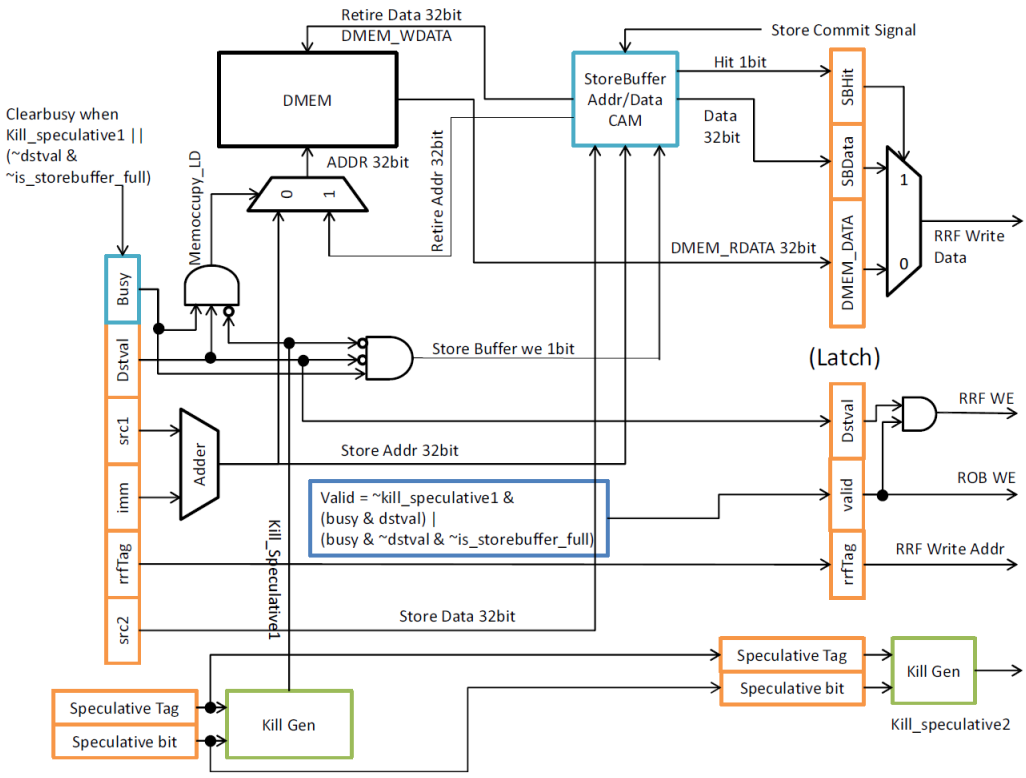


Fig.27 Load/Store Unit

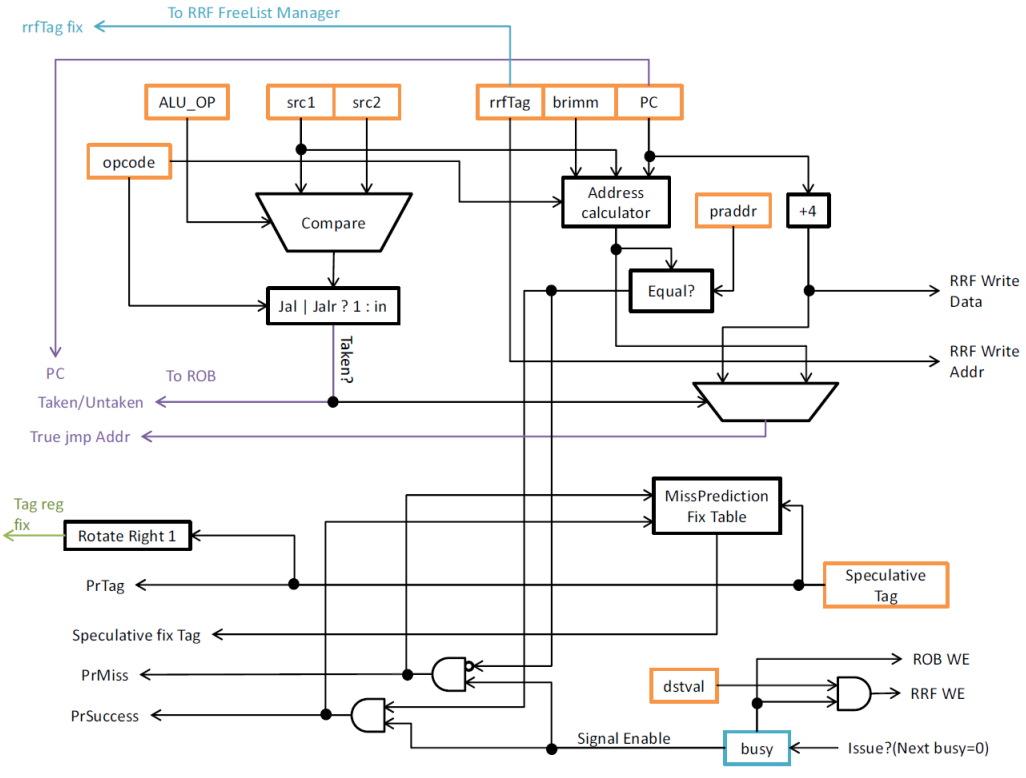


Fig.28 Branch Unit

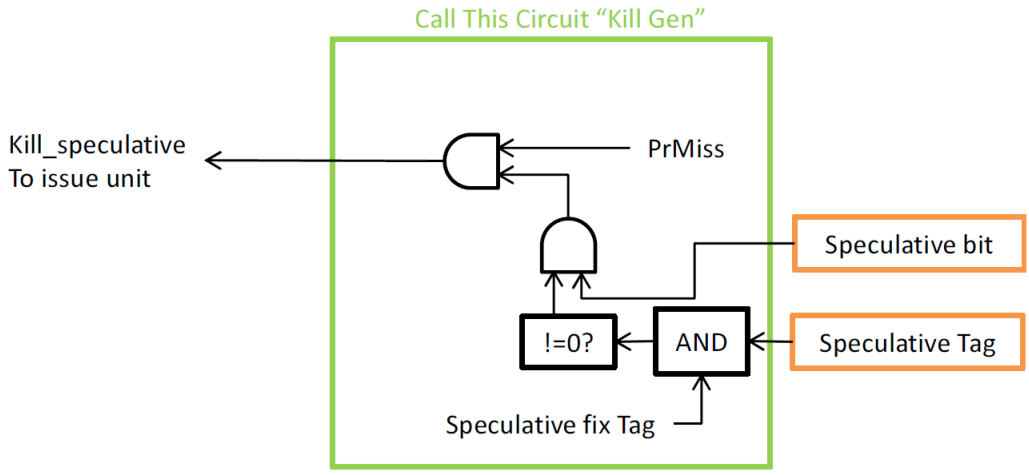


Fig.29 Kill Gen

当一个执行单元执行完一条指令之后，它将完成信息（ROB WE）通知给重排序缓冲区，并将写数据（RRF 写地址/数据）发送给 RRF。除了这些基本的操作之外，Load/Store 单元和分支单元还要执行一些额外的操作，以下详述。

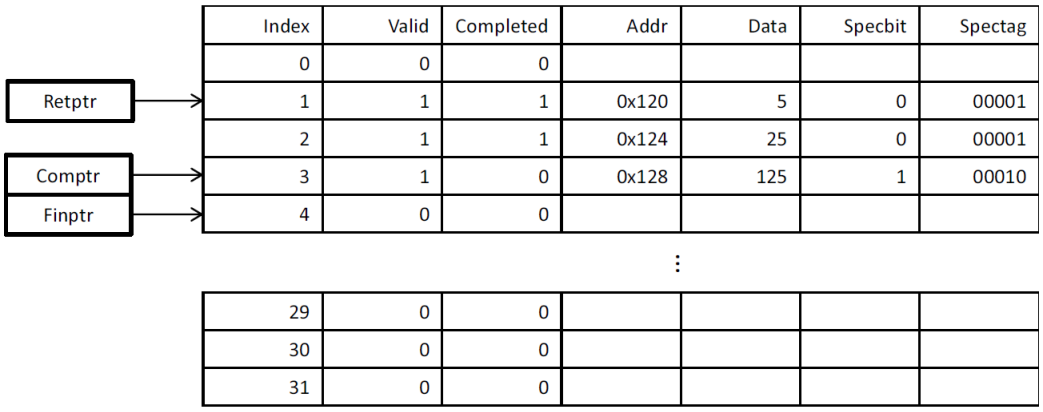
Load/Store 单元的操作被流水为 2 段。当 Load/Store 单元从保留站接收到 1 条 Load 指令时，它要访问 DMEM 和 Store Buffer，并在第 1 段流水接收来自 Store Buffer 的数据。在第 2 段流水，它要接收来自 DMEM 的数据，并写数据（如果是还没有被存到 DMEM 的数据，则是来自 Store Buffer 的数据；否则是来自 DMEM 的数据）到 RRF。

当 Load/Store 单元接收到一条 Store 指令，它在第 1 段流水将 store data 和 store addr 写入到 Store Buffer，并在第 2 段流水将完成信号通知重排序缓冲区。一旦这条 Store 指令完成了它的 COM 段，在 Store Buffer 中的 store 数据就可以写入 DMEM。当没有 Load 指令访问 DMEM 的时候，Store Buffer 中的数据将被写入 DMEM。

分支单元计算分支目标并将它和 IF 段的分支预测目标进行比较。如果分支预测正确，我们将更新 Miss Prediction Fix Table，并清除匹配推测标签的 Speculative bit。另一方面，如果分支预测错误，我们将把处理器的状态恢复到执行分支预测前的状态。这两种情形下，都需要暂停流水线（以便执行备份或者恢复）。在 COM 段，为了通知分支预测器关于一条分支执行完毕的信息，我们需要这条指令的信息。于是，分支单元将分支目标、分支条件写入重排序缓冲区。

17. storebuf（pp. 206-209）

Store Buffer 保存着执行完 EX 段的 store 指令的 store 数据和 store 地址。一旦 store 指令执行完它的 COM 段，该指令的 store 数据就可以从 Store Buffer 传输到 DMEM。Store Buffer 是一个关联存储器。当一条 Load 指令将它的 Load 地址提供给 Store Buffer，它就将最近的一条未排序的数据返回给 Load 指令。最近的未排序数据，可以通过旋转 load address hit 向量来选中。图 30 给出了 Store Buffer 的项和寄存器。有 3 个寄存器：Finptr、Comptr 和 Retptr。Finptr 是指向已结束指令（finished instruction）的指针。Comptr 是指向已完成指令（completed instruction）的指针。Retptr 是指向 store 数据已经被写入 DMEM 的指令的指针。当一条分支指令预测错误时，comptr 和 retptr 仅仅需要加 1，但是 finptr 需要重新计算，如同 alloc_issue_ino 的重新计算一样。



Index	Valid	Completed	Addr	Data	Specbit	Spectag
0	0	0				
1	1	1	0x120	5	0	00001
2	1	1	0x124	25	0	00001
3	1	0	0x128	125	1	00010
4	0	0				
⋮						
29	0	0				
30	0	0				
31	0	0				

Fig.30 Store Buffer

下面解释了 Store Buffer 中每一项的属性：

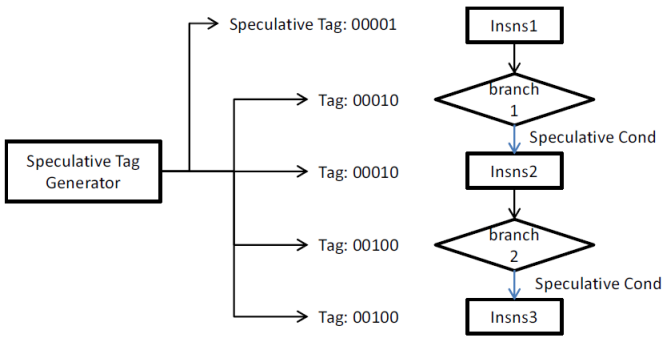
- valid：决定此项是否是有效项
- completed：决定指令是否已经完成。一旦 valid 和 completed 都是 1，此条指令就将 retired
- addr,data：写入到 DMEM 中的数据和地址
- specbit,spectag：specbit 决定了此条指令是否是推测执行的。Spectag 包含了推测标签。这些信息在出现分支预测错误时，作废指令所必须的。

18. miss_prediction_fix_table （pp. 228-231）

Miss Prediction Fix Table 保存了推测标签的状态。推测标签的状态由 2 个值来定义：Valid 和 Value。Valid 决定了拥有这个推测标签的指令是否是推测执行的指令。Value 给出了这个推测标签和其他推测标签之间的依赖关系。

下面，我们用一个例子（图 31）来解释 Miss Prediction Fix Table。图 31 给出了指令流和对 2 条分支指令译码后 Miss Prediction Fix Table 的状态。通过推测标签产生器按照 00001、00010、….、10000 的顺序产生的推测标签，被指派给指令。如果发现 branch1 指令的分支预测是错误的，我们从 Miss Prediction Fix Table 中读取索引为 00010（branch1 指令的推测标签）的值。此值为 00110。然后，需要作废的指令就是推测标签分别是 00010 和 00110 的这 2 条指令。

$$\forall i \in Instuctions(((i.Speculative\ Flag\ \&\ Value)\ !=0) \rightarrow \text{作废指令 i})$$



Miss Prediction Fix Table

Index	Value	Valid
00001	00000	0
00010	00110	1
00100	00100	1
01000	00000	0
10000	00000	0

Fig.31 Speculative Exection

19. reorderbuf （pp. 206-209, 254-259）

重排序缓冲区是一个允许指令按序提交的缓冲区。图 32 给出了重排序缓冲区中的寄存器和项。comptr 是一个指向下一条完成指令的指针。dispatchptr 是一个指向下一条分派指令的指针，它等于 rrf_freelistmanager 中的 RRFPtr。重排序缓冲区每个时钟周期最多可以完成 2 条指令。然而，分支指令和 Store 指令需要一条一条完成，以减少分支预测器和 Store Buffer 所需的存储器写端口数目。

当一条指令完成时，重命名表被更新，并且指令写入 RRF 的结果被复制到 ARF 中。只有当重命名表中的 RRFTag 等于完成指令的时候，重命名表的 Busy 位才被清除。

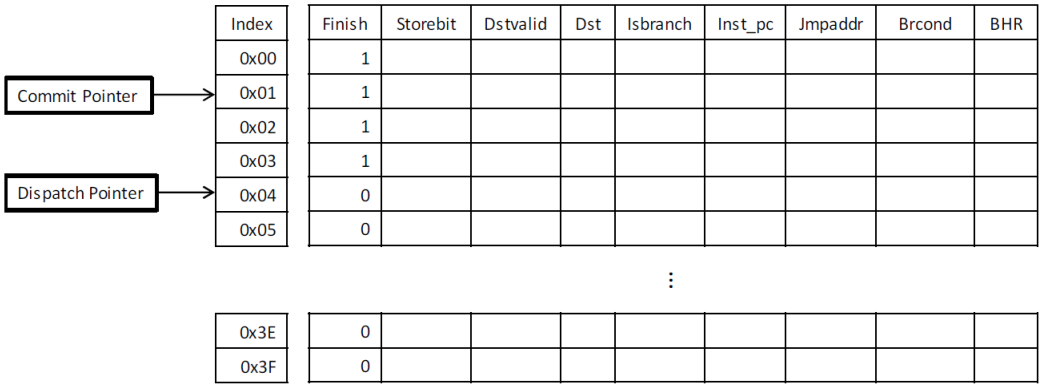


Fig.32 Reorder Buffer

下面解释了重排序缓冲区中每一项的属性：

- finish：指明了指令是否已经结束（finish）。当指令被分配时，finish 置为 0，当指令结束时，finish 被置为 1
- storebit：指明了指令是否是一条 Store 指令。如果 storebit 等于 1（Store 指令），需要将指令完成的信息通知给 Store Buffer
- dstvalid：指明了指令的目标寄存器是否是有效的，即指令是否需要把数据写入 ARF
- dst：目标寄存器编号
- isbranch：决定了指令是否是一条分支指令。如果 isbranch 等于 1（分支指令），我们需要在指令完成时，将数据写入分支预测器（PHT、BTB 等）
- inst_pc、jmpaddr、brcond、bhr：送往分支预测器的数据。inst_pc、jmpaddr、brcond、bhr 分别是指令地址、分支目标、分支条件和 bhr（用于计算 PHT 的写地址）

参考文献：

[1] John Paul Shen, Mikko H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 2013.

[2] McFarling, Scott: Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory, (1993)

[3] RISC-V, http://riscv.org/ (2016/01/20)