RIDECORE: RIsc-v Dynamic Exection CORE

Masashi Fujinami, Susumu Mashimo, Thiem Van Chu, Kenji Kise

2016/03/28

RIDECORE Overview

This part first explains the overall behavior of RIDECORE. After that, various parameters that determine the specifications of RIDECORE are shown.

1 Overall Behavior

Fig. 1 shows the overview of RIDECORE pipeline. RIDECORE has a six-stage pipeline structure, consisting of Instruction Fetch (IF), Instruction Decode (ID), Dispatch (DP), Select and Wakeup (SW), Execution (EX), and Complete (COM). Every pipeline stage, except the Load/Store units in EX stage, is executed one clock cycle.

In IF stage, at most two instructions are fetched from Instruction Memory (IMEM) by using Program Counter (PC) and sent to later stage. By reading IMEM on the negative edge clock transitions and writing the IF/ID latch on the positive edge clock transitions, IF stage can be executed in one clock cycle. PC is speculatively updated using Gshare branch predictor.

ID stage decodes instructions fetched in IF stage and generates data for later stages. *Speculative Tag gen* assigns Speculative Tag to the instructions. If there exists a branch instruction, both *Speculative Tag gen* and *Miss Prediction Fix Table* will be updated.

DP stage first fetches source operands of two instructions from Architected Register File (ARF) and Rename Register File (RRF). It next performs register renaming by allocating entries in Reorder Buffer and RRF (called RRFTag in RIDECORE) to the instructions. Finally, Allocate Unit writes the data needed for executing the instructions to Reservation Station (RS). Data forwarding is performed using *Source Operand Manager*.

In SW stage, Issue Unit selects instructions that have all the necessary operands and issues them to EX stage. When an instruction is issued, it is removed from Reservation Station immediately.

In EX stage, instructions are executed. When the execution of an instruction is completed, the result is written to RRF. At the same time, Reorder Buffer is notified about the completion of the execution. The Branch unit is responsible for determining whether branch predictions are correct and sending the prediction results to some modules for appropriate actions. When a branch misprediction occurs, speculatively executed instructions are invalidated by using *Miss Prediction Fix Table*. The EX stage of

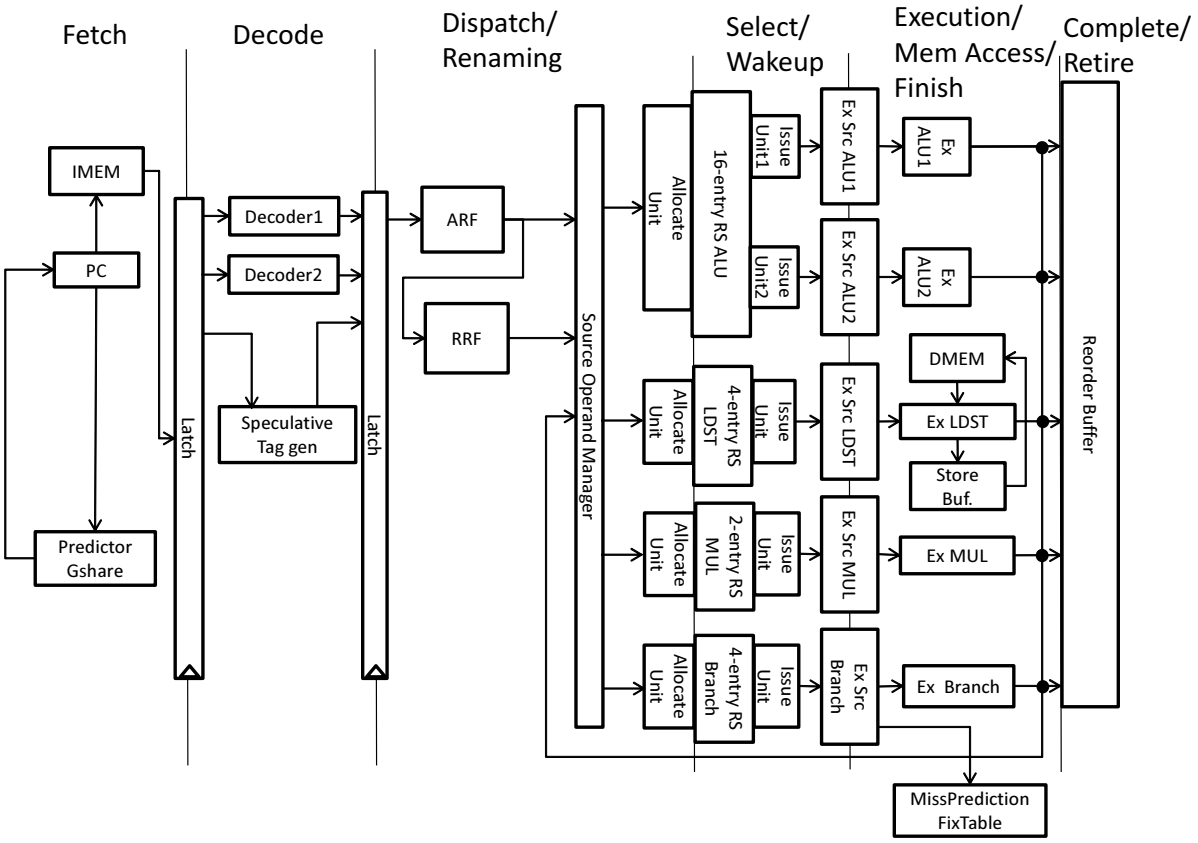


Fig.1 RIDECORE pipeline

every instruction, except Load/Store, is executed in one clock cycle. The operation of Load/Store Unit is pipelined with two stages.

In COM stage, up to 2 instructions in Reorder Buffer are completed. Upon the completion of an instruction, the data of the instruction in RRF are moved to ARF. The Renaming Table in ARF is updated accordingly. The branch predictor is also updated with new information.

2 Specification Overview

Table 1 shows the specification of RIDECORE.

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

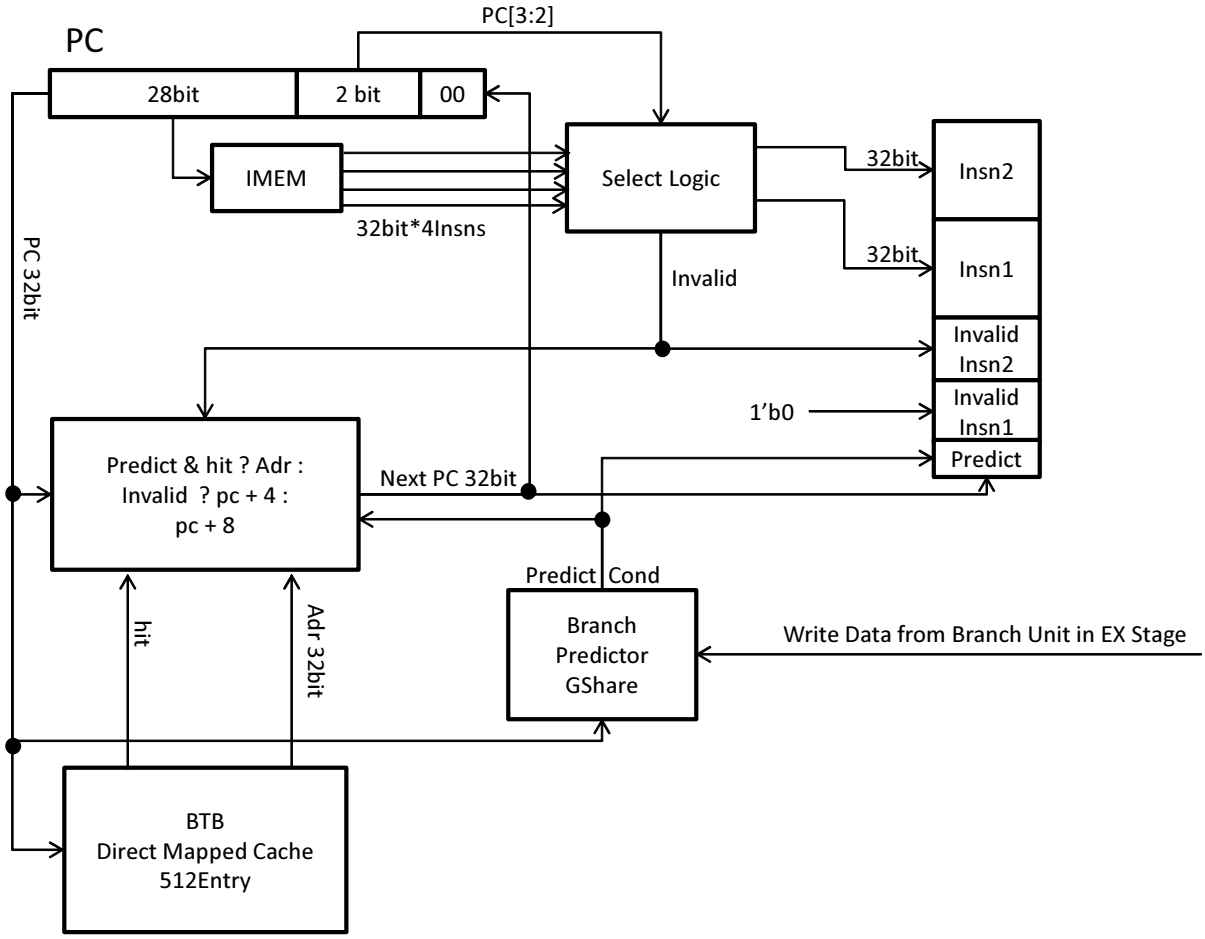


Fig.2 pipe_if

Hardware Modules

This part describes the hardware modules inside RIDECORE. More in-depth explanations of some modules can be found in John Paul Shen and Mikko H. Lipasti [1]. In these cases, we recommend the readers to read the explanations in the book before reading the explanations in this document. For example, if the explanations of a module should be read in the book from page 228 to page 231, we will append *(pp. 228-231)* to the title of the section explaining the module.

3 pipe_if

In IF stage, two instructions are fetched from IMEM. PC is updated using the branch predictor. Fig. 2 shows the circuit of IF stage.

The size of each entry in IMEM is 128-bit. Thus, since the RV32IM has fixed length 32-bit instructions, each entry of IMEM can contain four instructions.

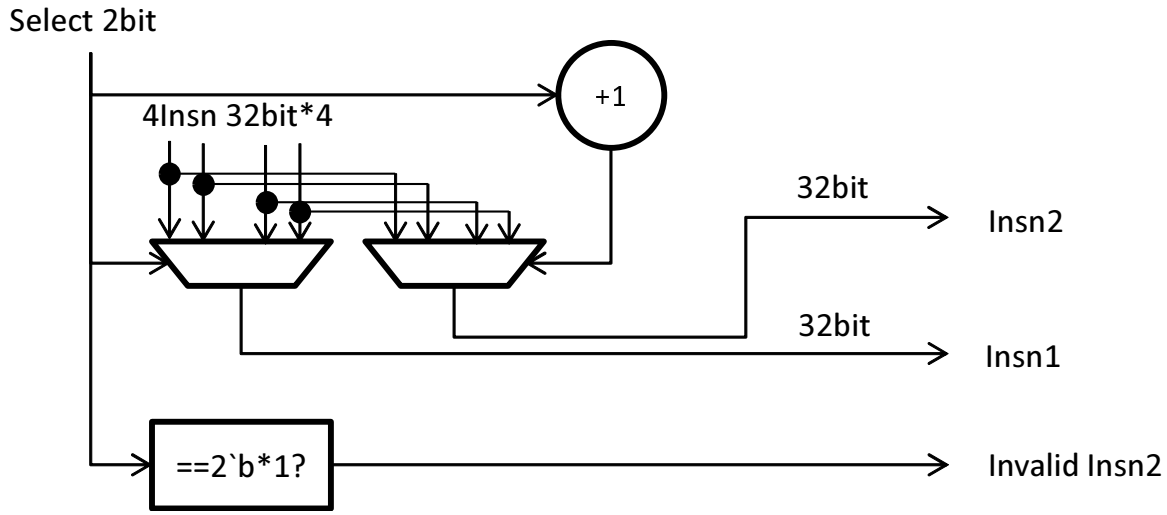


Fig.3 Select logic

At each negative edge clock transition, *pipe_if* fetches four instructions using PC[31:4] as the index of IMEM. Then, *sellog* (Fig. 3) selects two instructions from the four instructions using PC[3:2] and writes them to the IF/ID latch at the positive edge clock transition. When PC is not divisible by 8, *sellog* invalidates the 2nd selected instruction (Insn2 in Fig. 3) because this instruction has been already selected previously. For example, when PC=0x2c, which is not divisible by 8, four instructions from IMEM are at 0x20, 0x24, 0x28, and 0x2c. Insn1 is the instruction at 0x2c while Insn2 is the instruction at 0x20 which has been already selected previously (we expect Insn2 to be 0x30). In this case, Insn2 is thus invalidated.

For simplicity, as shown in Fig. 2, Invalid Insn1 is always set to 0. However, it may be set to 1 to kill all instructions in IF stage when some events such as branch mispredictions are detected.

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

Fig. 4 shows the Gshare branch predictor circuit. It predicts whether a branch is taken in IF stage by using shared Branch History Register (BHR), Pattern History Table (PHT), PC, and Branch Target Buffer (BTB). Here, we present only some design decisions in implementing the predictor.

At the negative edge of every clock pulse, we read an entry of PHT with the read address = $PC[12:3] \bigoplus BHR$. If the read data is greater than 1, the prediction result is set to "Taken". Otherwise, it is set to "Not Taken". BHR is updated using the prediction result. However, if the prediction turns out to be incorrect, BHR is restored to the original value. Thus, it is always backed up before every update.

4.1 pht

PHT is an array of 2-bit saturating counters used in Gshare predictor. Fig. 5 shows the circuit of PHT. PHT is updated when a branch instruction is completed. The entry of PHT is incremented/decremented

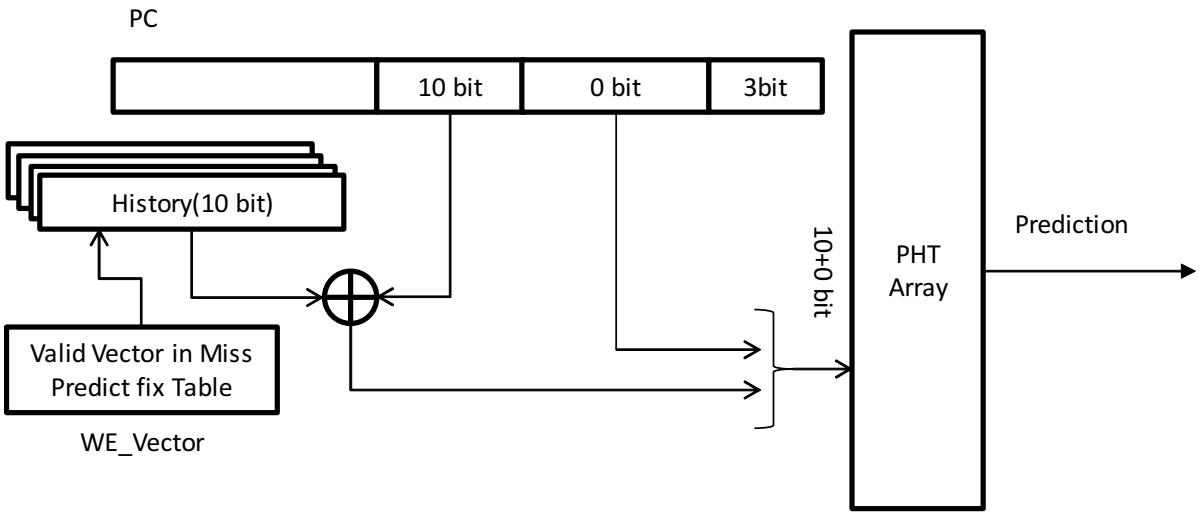


Fig.4 Gshare branch predictor

if the condition of the branch instruction is "Taken"/"Not Taken".

In COM stage, an entry of PHT is updated by first reading its original value on the negative edge clock transition and then writing the updated value on the positive edge clock transition. PHT has two read ports (one for reading at IF stage and the other at COM stage) and one write port (for writing at COM stage). In RIDECORE, it is implemented using two true dual-port BRAMs.

4.2 btb

Branch Target Buffer (BTB) is composed of pairs of branch source and destination addresses. It is implemented as a direct mapped cache to reduce the hardware resource usage. Because of this, the prediction quality is modrate.

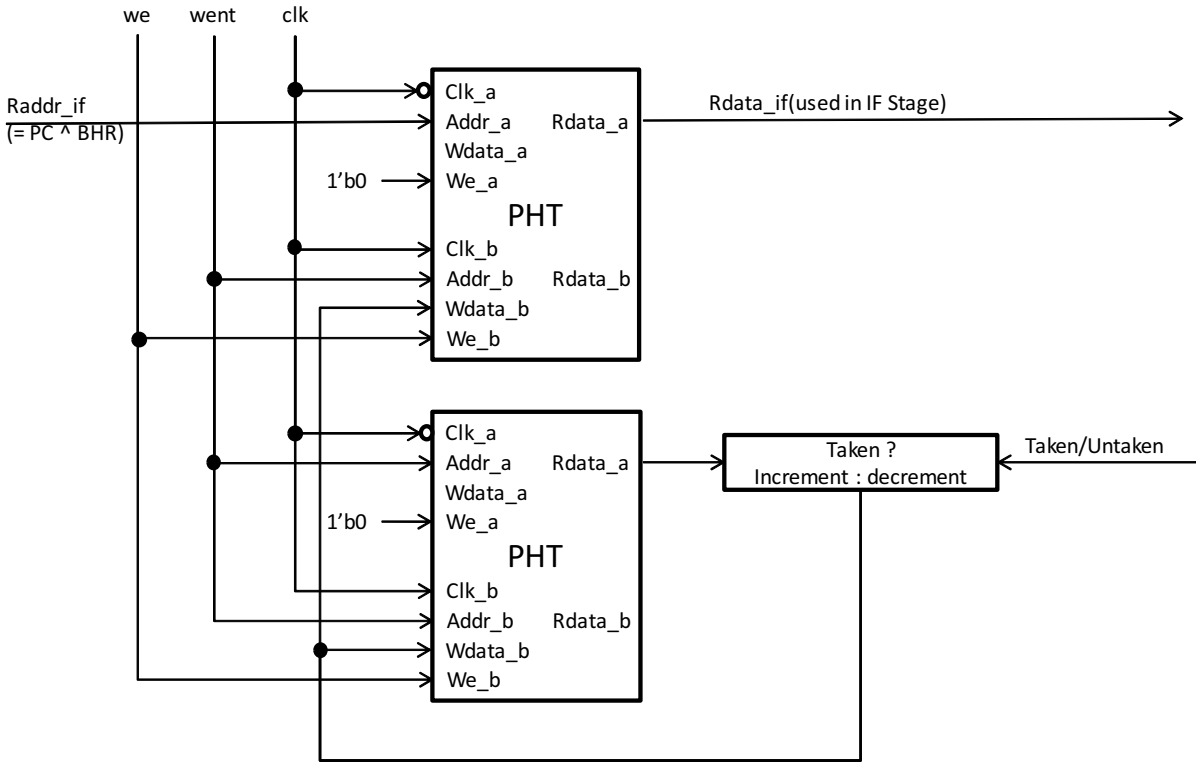


Fig.5 Pattern History Table (PHT)

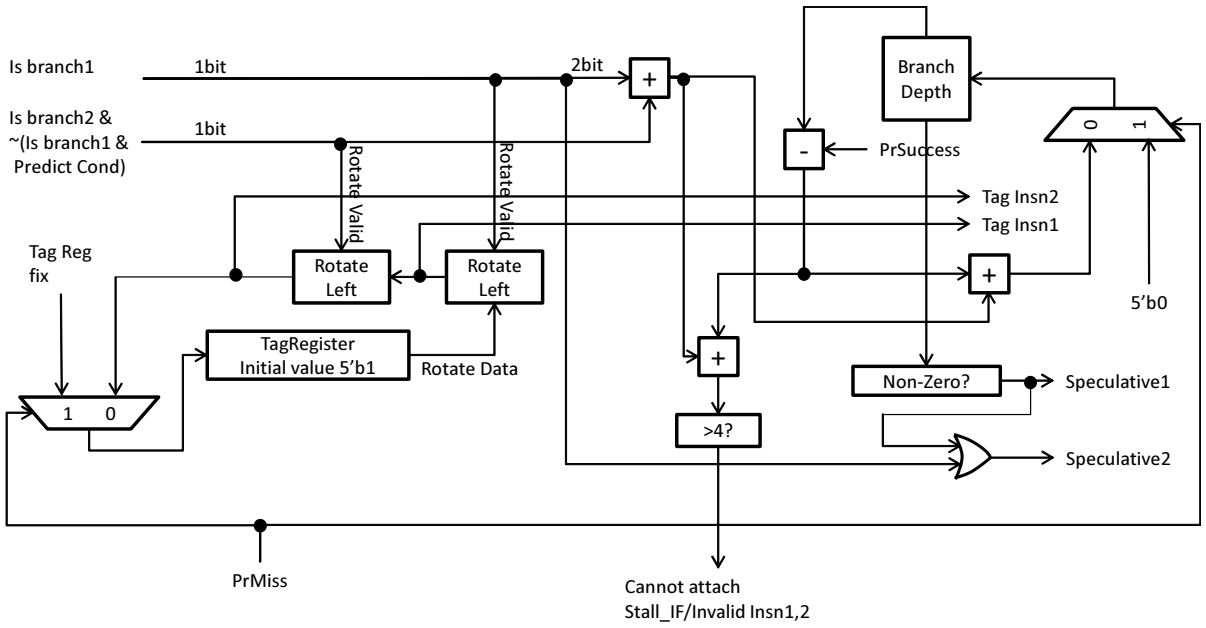


Fig.6 Speculative Tag Generator

5 tag_generator (pp. 228-231)

Fig. 6 shows the circuit of Tag Generator. The circuit has two registers. One contains the current Speculative Tag (*tagreg*). The other contains the current branch depth (*brdepth*). Tag Generator receives three signals from outside: *branchvalid1*, *branchvalid2*, and *enable*. *branchvalid1/2* indicates whether Insn1/2 is valid while *enable* indicates whether the instructions can be issued. Using these signals, Tag Generator generates Speculative Tag (*sptag1/2*), speculative flag (*speculative1/2*), and updates *tagreg* and *brdepth*.

Speculative Tag is assigned to instructions in the order of 00001, 00010, ..., 10000 (left rotation from 00001). If no Speculative Tag is available, Tag Generator will set flag *attachable* to zero and stall the tag assignment process.

When branch prediction succeeds (prsuccess == 1), *brdepth* is decremented to free Speculative Tag. When branch misprediction occurs (prmiss == 1), *tagreg* and *brdepth* are restored to the values before the prediction is performed. *tagreg* is restored using *tagregfix* (generated in Branch unit). *brdepth* is set to zero because branch instructions are executed In Order (there is no remaining speculative instruction right after the time a speculative instruction is invalidated).

6 decoder

Decoder is a module used in ID stage. This module use information from incoming instructions to generates data for later stages (DP, EX, ...). Fig. 7 shows the I/O definition of decoder.

```
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

Below is the explanation of the output singals.

- imm_type: this signal is used in module *imm_gen*. It indicates the format of immediate data in the instruction.
- rs1, rs2, rd: 1st source operand, 2nd source operand, and destination register number.
- src_a_sel, src_b_sel: used to select ALU operands.
- wr_reg: this signal indicates whether the instruction writes data to destination registers.
- uses_rs1, uses_rs2: valid signals of rs1 and rs2. If rs1/2 is valid, data fetch is needed. These signals are used to prevent the fetch of unnecessary data (the instruction cannot be issued until all necessary data are fetched).
- illegal_instruction: this signal indicates that the instruction is not defined in this processor. It will be used in exception handling in the future.
- alu_op: the type ALU operation.
- rs_ent: Reservation Station ID. Each execution unit contains a Reservation Station with an ID defined as follows:
  - ALU: 1
  - BRANCH UNIT: 2
  - MULTIPLIER: 3
  - LOAD/STORE: 4
- dmem_size, dmem_type: determine the size of the Load/Store data (4-byte/2-byte/1-byte). These signals are not used because RIDECORE supports only 4-byte Load/Store instructions.
- md_req_op: the operation type (multiplication or division or modular). This signal is not used

because RIDECORE does not support division and modular operations.

- md_req_in_1_signed, md_req_in_2_signed: these signals indicate whether the 1st and the 2nd source operand of the multiplier are signed.
- md_req_out_sel: selects hi/lo. In RIDECORE, the output of the multiplier is 64-bit while every register in ARF and RRF is 32-bit. This signal determines which portion of the output of the multiplier is selected as the final multiplication result: the upper 32-bit or the lower 32-bit.

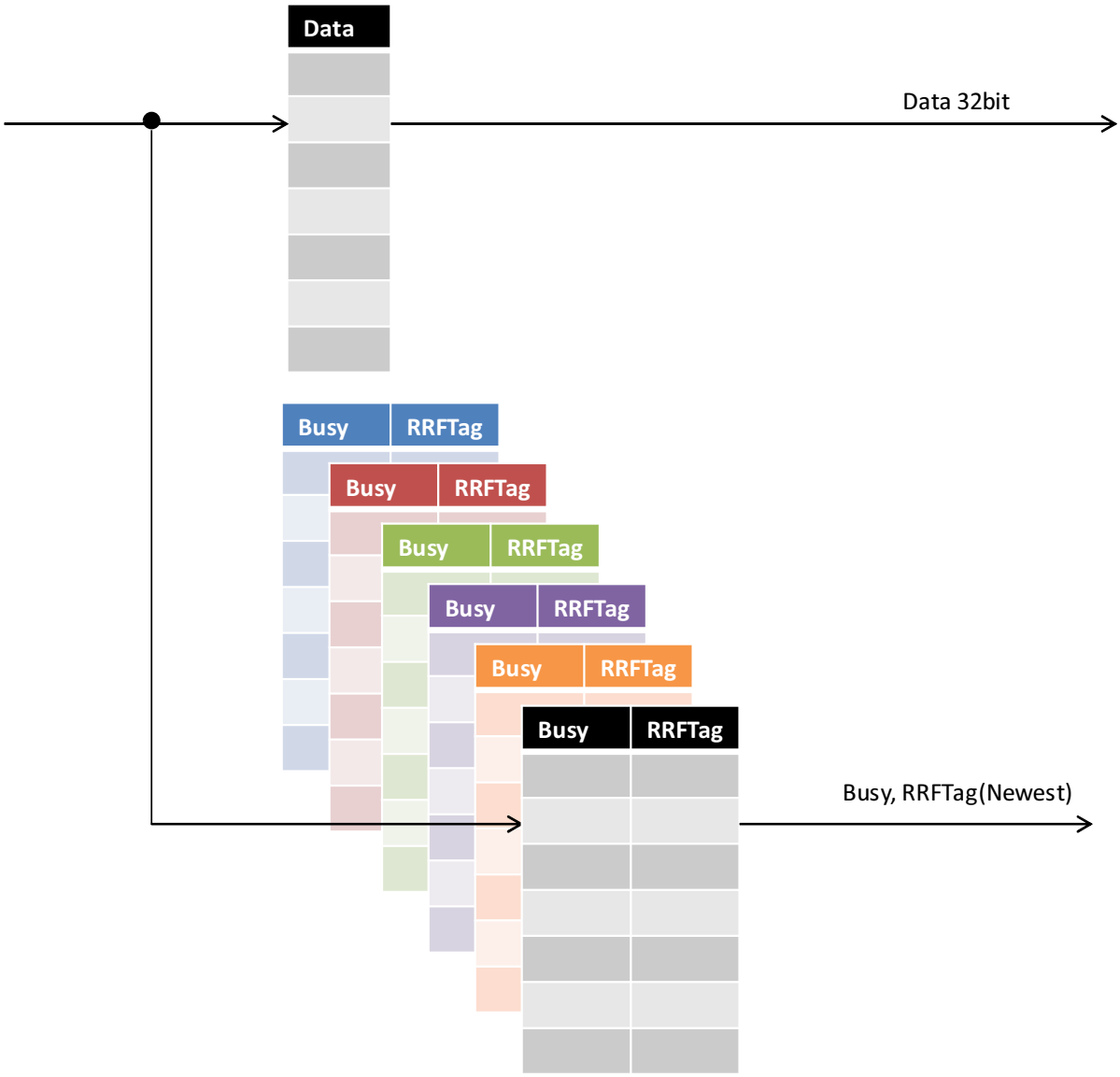


Fig.8 ARF Read

7 arf (pp. 239-244)

Each entry of Architected Register File (ARF) consists of a register (*Data*) updated in COM stage together with some renaming information (*RRFTag, Busy*). Fig. 8, 9, 10 show the circuit and behavior of ARF. Renaming Table contains renaming information. When a branch misprediction occurs, it must be restored to the state before the prediction. Thus, the number renaming tables is equal to the number of speculative tags. Below we explain the read and write operations of ARF as well as how Renaming Table is restored when branch mispredictions occur.

Fig. 8 shows the read behavior of ARF. The data in the newest Renaming Table (Black table) is used as the current renaming information. Fig. 9 shows the write behavior of ARF. Write requests are generated at DP stage and COM stage. The write enable signal *we* for the renaming tables is the complement of the *valid* vector in *Miss Prediction Fix Table* (*mpft*). For more explanations about *mpft*, please read section "miss_prediction_fix_table".

Fig. 10 shows the restoration behavior of Renaming Table. The restoration is performed when a branch instruction is in Branch Unit and the branch condition is calculated. *RRFTag* and *Busy* in Renaming Table are implemented as 32-bit vectors, so it is possible to rewrite all of the entries in one cycle. The pipeline is stalled while the restoration is in progress because Renaming Table cannot be read correctly.

Fig. 11 shows an example of the restoration behavior. If the prediction of branch1 (SpeculativeTag = 5'b00010) is incorrect, we will have to restore the information of all renaming tables to the state when SpeculativeTag = 5'b00001. Thus, *Fix Vector* is set to 6'b111111 and the red renaming table is selected as *Fix Data*.

On the other hand, if the prediction of branch1 is correct, the contents of the backup table (the red table, Speculative Tag = 5'b00001) will become unnecessary. We overwrite these contents with the newest information from the black table for backing up the current state. Thus, *Fix Vector* is set to 6'b010000 and the black renaming table is selected as *Fix Data*.

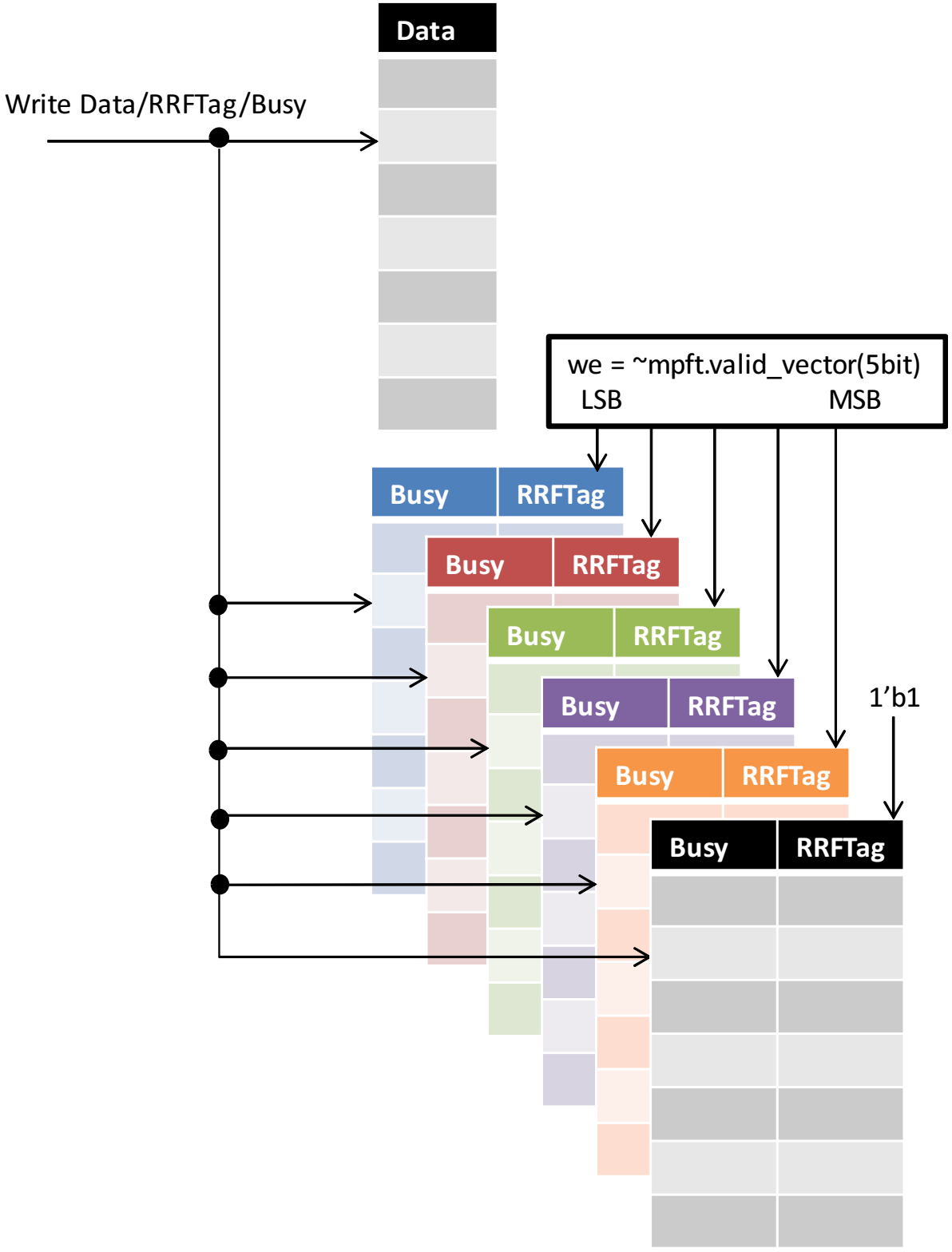


Fig.9 ARF Write

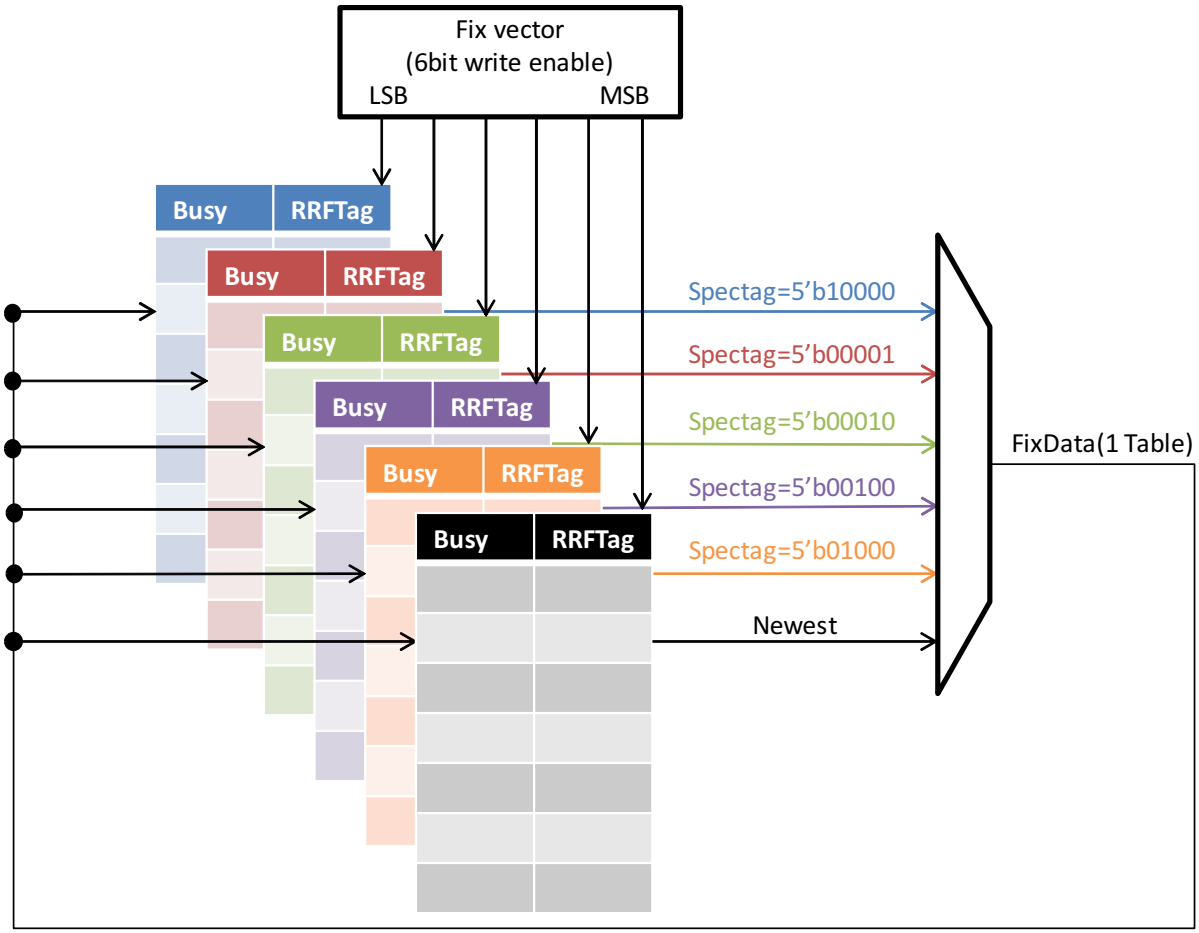


Fig.10 Fix Renaming Table

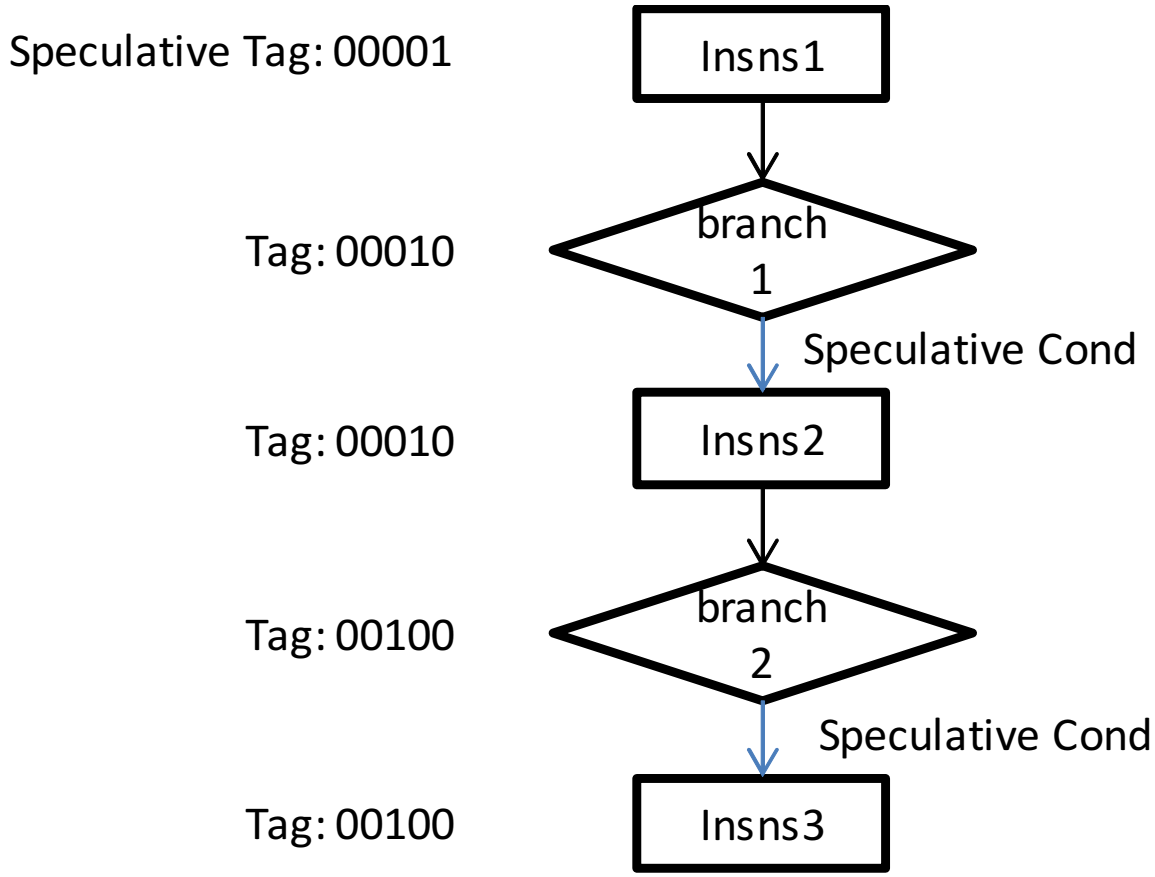


Fig.11 Example of Speculative Exection

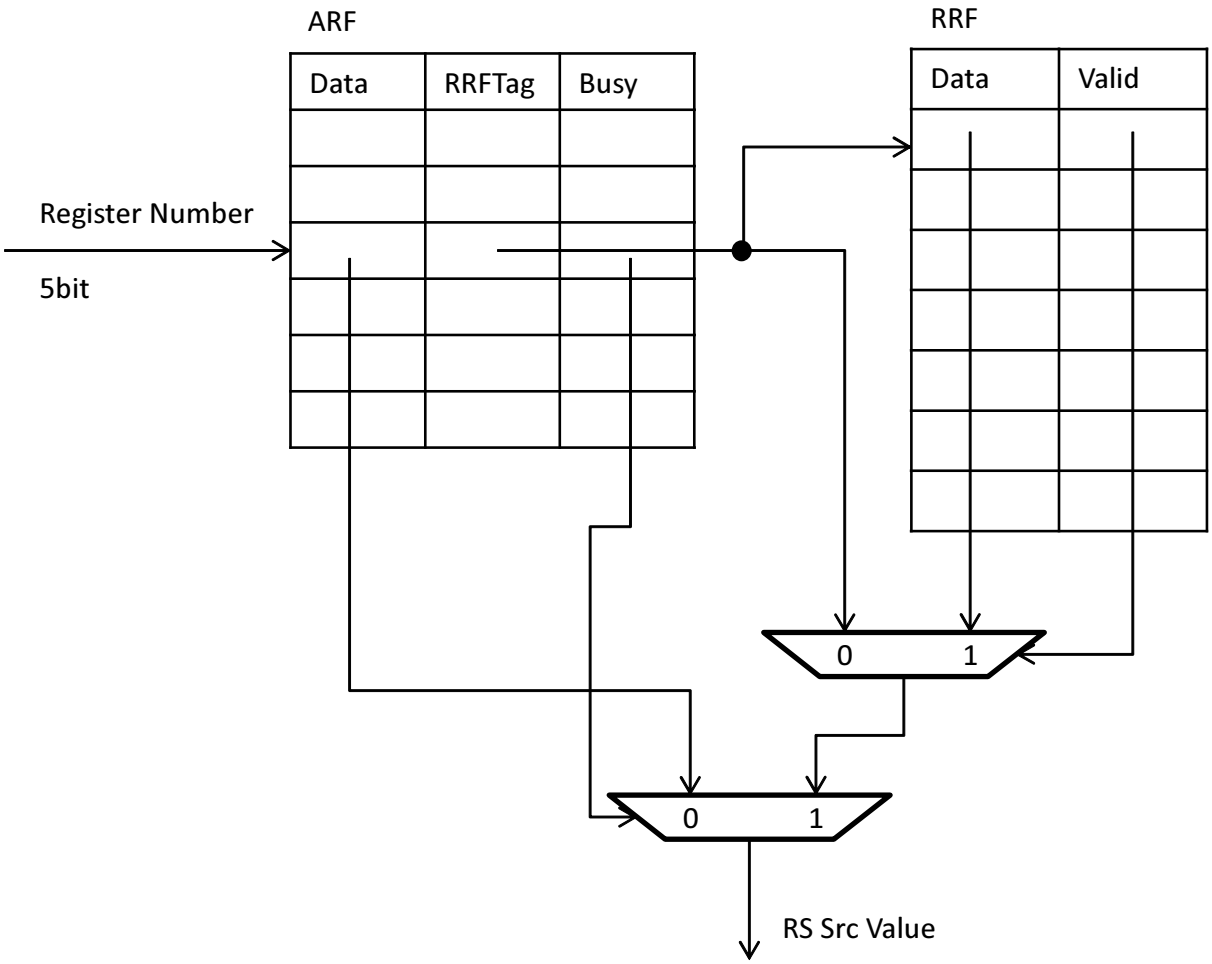


Fig.12 Register Renaming using ARF and RRF

8 rrf (pp. 239-244)

An entry of Renamed Register File (RRF) has two attributes: RRFData and RRFValid. RRFData contains the value of the rename register. RRFValid determines whether the entry is valid. In RIDECORE, the size of RRF is 64-entry which is the same as the size of Reorder Buffer. In fact, there is one-to-one correspondence between RRF and Reorder Buffer. Thus, the tag used to access RRF can be also used to access Reorder Buffer.

9 sourceoperand_manager (pp. 239-244)

sourceoperand_manager generates data to write to Reservation Station (RS) from the requested register number, the information of ARF and RRF, and the RRFTag of *inst1*. Fig. 12 shows the circuit that

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

```
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

implements Register Renaming using ARF and RRF.

Table2 shows three possible states of a requested register and the data dispatched to Reservation Station in each case. Suppose that the requested register is *regsrc*. Its states in ARF and RRF are ARF.Busy and RRF.Busy, respectively. ARF.Busy is equal to zero if and only if regsrc is not the destination register of any outstanding instruction. In this case, the latest value of regsrc is available in ARF. On the other hand, if ARF.Busy is equal to one, there exists an outstanding instruction with the destination register regsrc. RRF.Valid determines whether EX stage of this instruction has been completed. Upon the completion of EX stage, the result is written to RRF and RRF.Valid is set to one. When RRF.Valid is equal to zero, i.e. the execution of EX stage has not been completed yet, RRFTag is sent to RS for transferring the latest value of regsrc from RRF to RS later (when all stages of the instruction with the destination register regsrc are completed).

Fig. 13 shows the source code of *sourceoperand_manager*. *src* is the data dispatched to Reservation Station. *src* is set to RRFTag when *rdy* is equal to zero. *src* is set to zero when the requested register number is 0 (like MIPS ISA, RISC-V ISA defines a special *zero register*). When one of the source registers

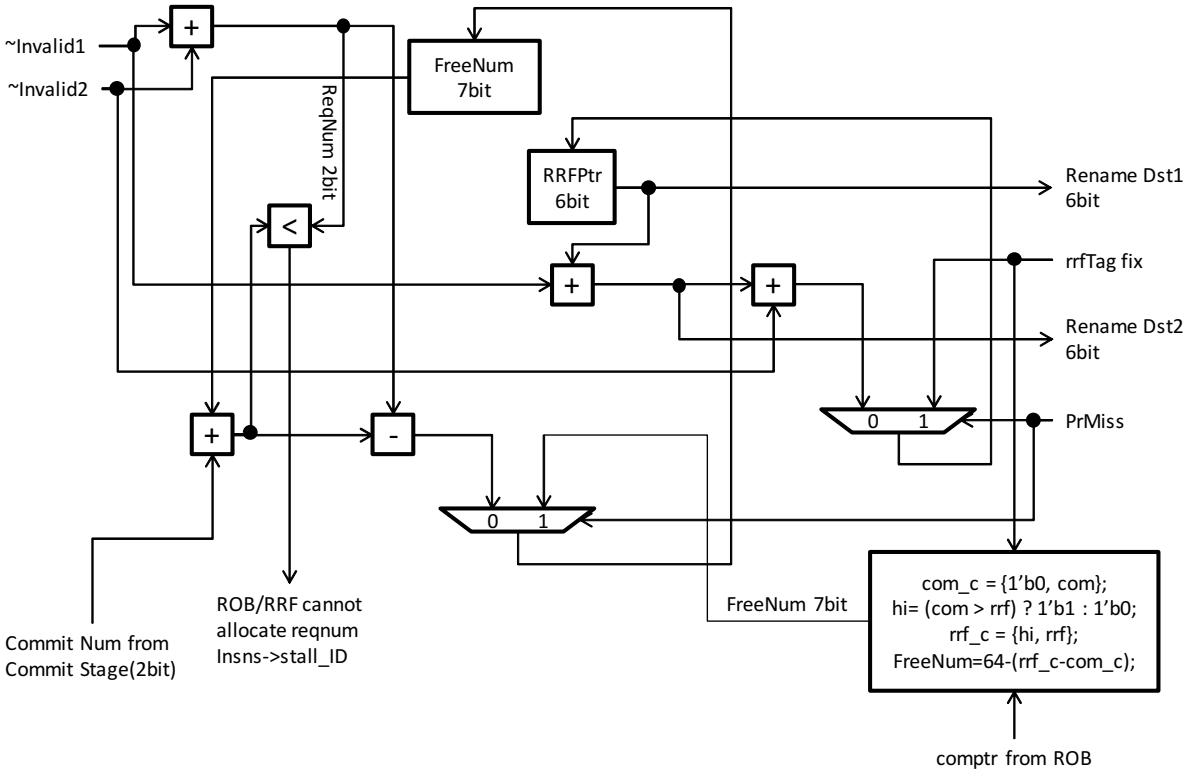


Fig.14 RRF Free List Manager

of *inst2* is the destination register of *inst1* (*src_eq_dst1* is equal to one), *src* is set to the value of *inst1*'s RRFTag. Except for these situations, the module operates as described in Table 2.

10 rrf_freelistmanager (pp. 239-244)

Fig. 14 shows the circuit of *rrf_freelistmanger* which manages free entries of RRF and Reorder Buffer. There are two registers: *FreeNum* and *RRFPtr*. *FreeNum* stores the number of free entries. *RRFPtr* is the current entry number of RRF and Reservation Station.

This circuit assigns RRFTag to instructions in DP stage. When *FreeNum* is smaller than the number of instructions requesting to write to Reservation Station, both IF stage and ID stage are stalled since no instruction can be dispatched. When an instruction is completed in COM stage, the RRFTags used by this instruction are released. Also, *FreeNum* is incremented. When a branch misprediction occurs, the RRFTags of all speculative instructions are also released. To do this, *RRFPtr* is set to *rrfTag fix* which is the RRFTag of the next instruction of the branch instruction which caused the branch misprediction. FreeNum is also restored by using *rrfTag fix* and comptr from Reorder Buffer.

```
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

11 src_manager (pp. 256-259)

Fig. 15 shows the source code of *sourcemanager* in Verilog HDL. This module is used to forward results in DP stage and to wait for required operands in Reservation Station. Here, we explain its input/output.

- opr: operand in DP stage (sent from *sourceoperand_manager* to Reservation Station). If *opr_rdy* is equal to zero, opr contains RRFTag which generates the required value. Otherwise, opr contains the requied value.
- opr_rdy: determines what is stored in opr.
- exrslt*, exdst*, kill_spec*: outputs from five execution units (ALU1/2, Load/Store unit, Branch

unit, multiplier). When exdst*i* is equal to *opr* and both *opr_rdy* and *~kill_spec** are equal to zero, *src* is set to exrslt*i*.

- src, resolved: *opr*, *opr_rdy* are forwarded to these signals whenever possible.

12 imm_gen, brimm_gen

In RISC-V ISA, some instructions contain immediate values within their operands. *imm_gen* and *brimm_gen* generates 32-bit signed immediate values from decoded data (*imm_type*) and instruction data. *brimm_gen* is used for branch instructions while *imm_gen* is used for others.

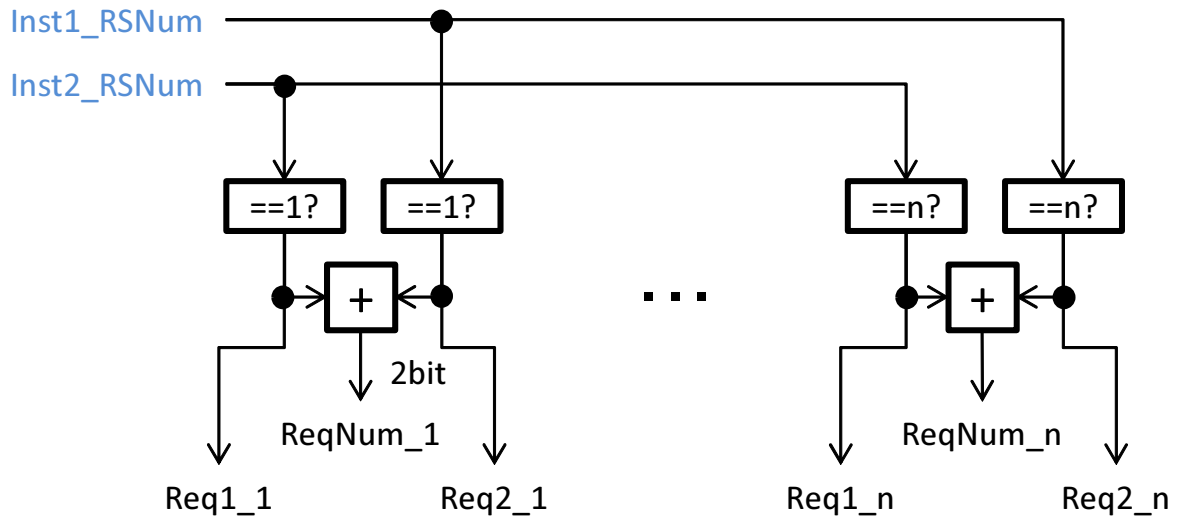


Fig.16 RS Request Generator

13 rs_requestgenerator (pp. 254-259)

Reservation Station receives the data of two instructions (*inst1/2*) together with two write enable signals. The write enable signals are generated by module *rs_requestgenerator*. Fig. 16 shows the circuit of this module. The circuit generates *Req1/2* (write enable signals for *inst1/2*) from *RSNum1/2* generated by the decoder.

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

Reservation Station (RS) is responsible for getting and storing operands of registered instructions. There is a reservation station for each execution unit. Allocate Unit allocates instructions to the reservation stations. Issue Unit issues instructions from the reservation stations to the execution units.

Fig. 17 shows the I/O of Reservation Station and the interfaces between it and Allocate Unit and Issue Unit. When Allocate Unit receives write enable signals (*we1/2*), it looks for free entries in Reservation Station and generates write addresses (*waddr1/2*). When there is an instruction in Reservation Station ready to be executed and the corresponding execution unit is not busy in the next cycle, Issue Unit issues the instruction to the execution unit. Issue Unit generates the entry number of the instruction (*raddr*) and Reservation Station outputs data (*rdata*) to the execution unit. More in-depth explanations of the allocate/issue operations can be found in the next section (Allocate Unit and Issue Unit).

Fig. 18 shows the circuit of an entry in Reservation Station. Orange colored squares indicate registers. *Write data* is the data needed to execute the registered instructions (operands, opcode, RRFTag, etc.). Execution results are forwarded to two registers Opr1 and Opr2 via Src Managers. Below is the explanation of the registers in Reservation Station.

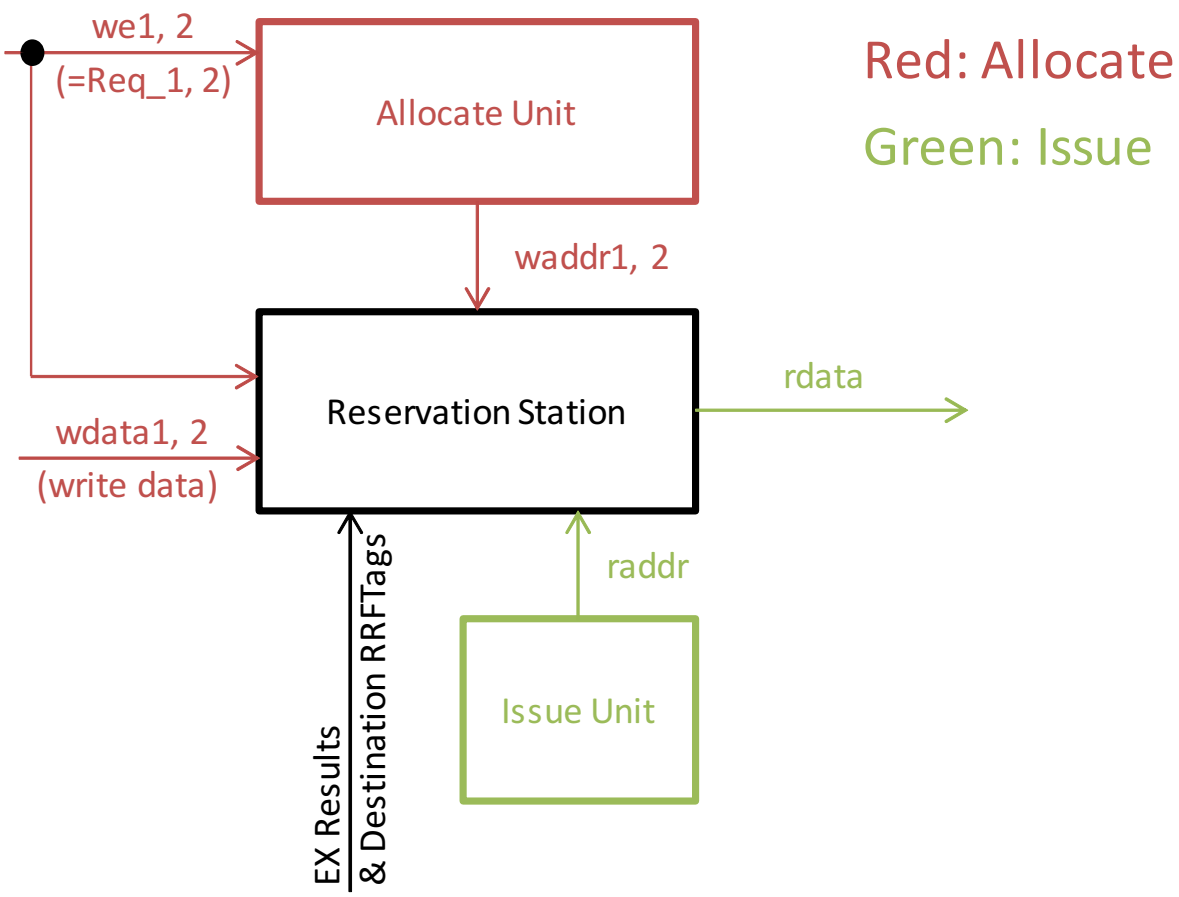


Fig.17 Reservation Station (with Allocate Unit and Issue Unit)

- Opr1, Opr2: the 1st and 2nd register operands. These data have different meanings depending on the valid bits.
  - Valid1/2 = 1: *Opr1/2* is ready. Valid1/2 also becomes 1 when the instruction does not use Opr1/2.
  - Valid1/2 = 0: *Opr1/2* is still not ready for executing the instruction. In this case, Opr1/2 contains the RRFTag that generates the required operand.
- Busy: determines whether the instruction is registered.
- Other Data: other data needed to execute the instruction.
  - imm: immediate value.
  - rrftag: RRFTag.
  - dstval: determines whether the instruction needs to write data to ARF.
  - specbit, spectag: *specbit* determines whether the instrucion is speculative while *spectag* contains the allocated Speculative Tag of the instruction.

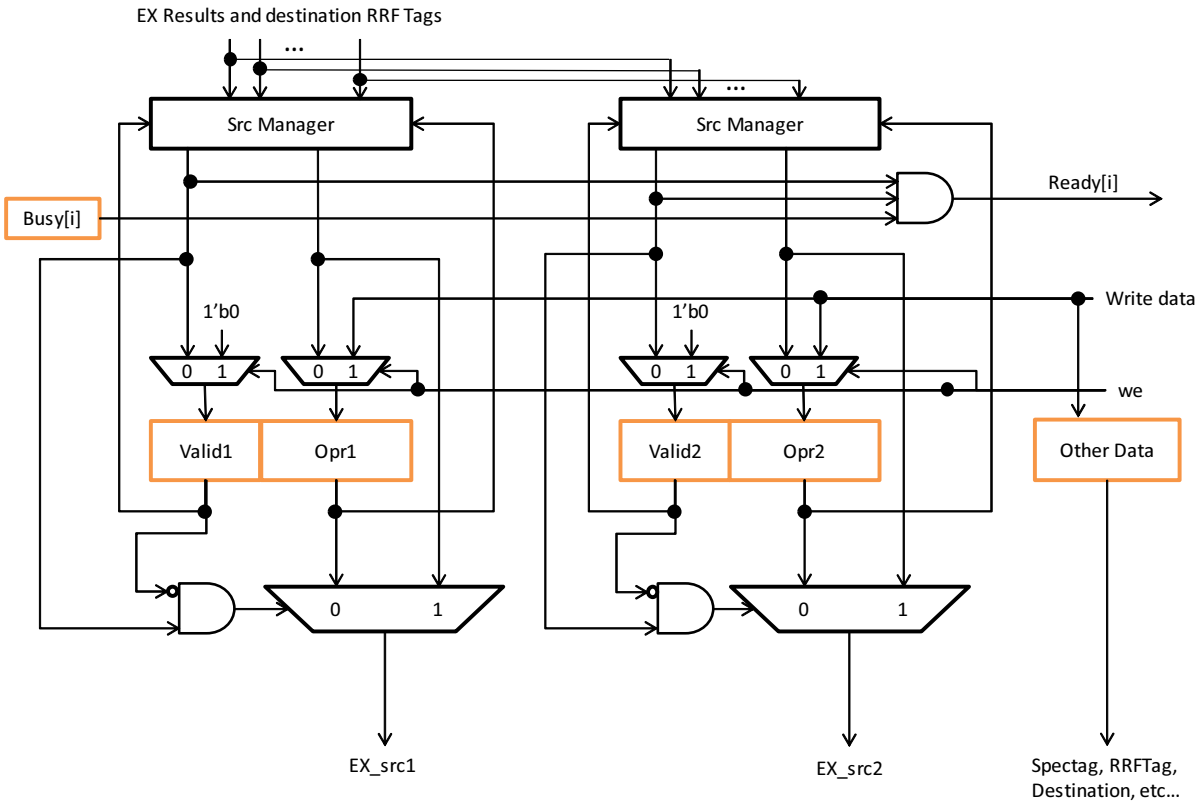


Fig.18 An entry in Reservation Station

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 Allocate Unit and Issue Unit (pp. 254-261)

15.1 Out-of-Order Issue

Fig. 19 shows the circuit of Allocate Unit. Fig. 20 shows the circuit of *RS Free Entry Finder* used in Allocate Unit. Allocate Unit behaves like an address resolver for Reservation Station. First, Allocate Unit receives write requests *Req1/2* to Reservation Station. Then, the unit looks for free entries (*Free_ent1/2*) in Reservation Station by using *RS Free Entry Finder* and writes instruction data to the found free entries.

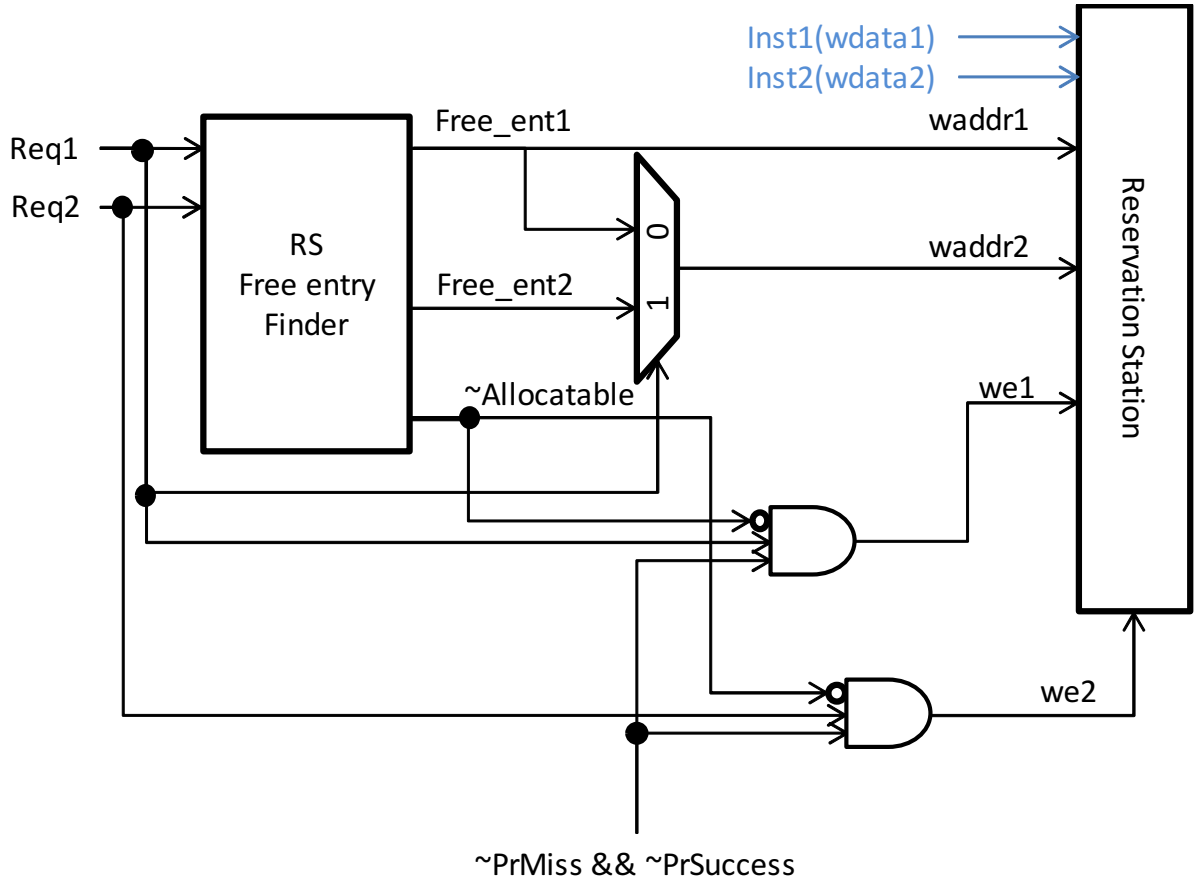


Fig.19 Allocate Unit

The interface of Reservation Station in RIDECORE is the same as a four-port memory (two read ports and two write ports). *wdata1/2*, *waddr1/2*, and *we1/2* are the write data, write addresses, and write enable signals, respectively. Table 3 shows how *waddr1* and *waddr1* are determined.

RS Free Entry Finder looks for up to two free entries by using two priority encoders. The inputs of the 1st priority encoder is the negation of the busy vector from Reservation Station (*˜Busy0*-*˜BusyN-1*). To avoid selecting the free entry already selected by the 1st priority encoder, the inputs of the 2nd priority encoder are masked by *Mask Unit*. *Entry_en1/2* determines whether *Free_ent1/2* is valid. *allocatable* becomes 1 when $Entry\_en1 + Entry\_en2$ is not smaller $Req1 + Req2$.

Issue Unit selects an instruction which has all the necessary operands and issues to EX stage. When an instruction is issued, it is removed from Reservation Station immediately. We use the Oldest First algorithm to select an instruction to issue. Fig. 21 shows the minimum value selection circuit (called *oldest finder* in RIDECORE) to realize the Oldest First algorithm.

Fig. 22 shows the format of an entry in *oldest finder*. The MSB of the entry is *˜rdy* because Issue Unit must prioritize instructions that have all the necessary operands. Since RRFTag is assigned from 0 and increases as time goes on, we can select the oldest instruction by comparing RRFTags. However, this is

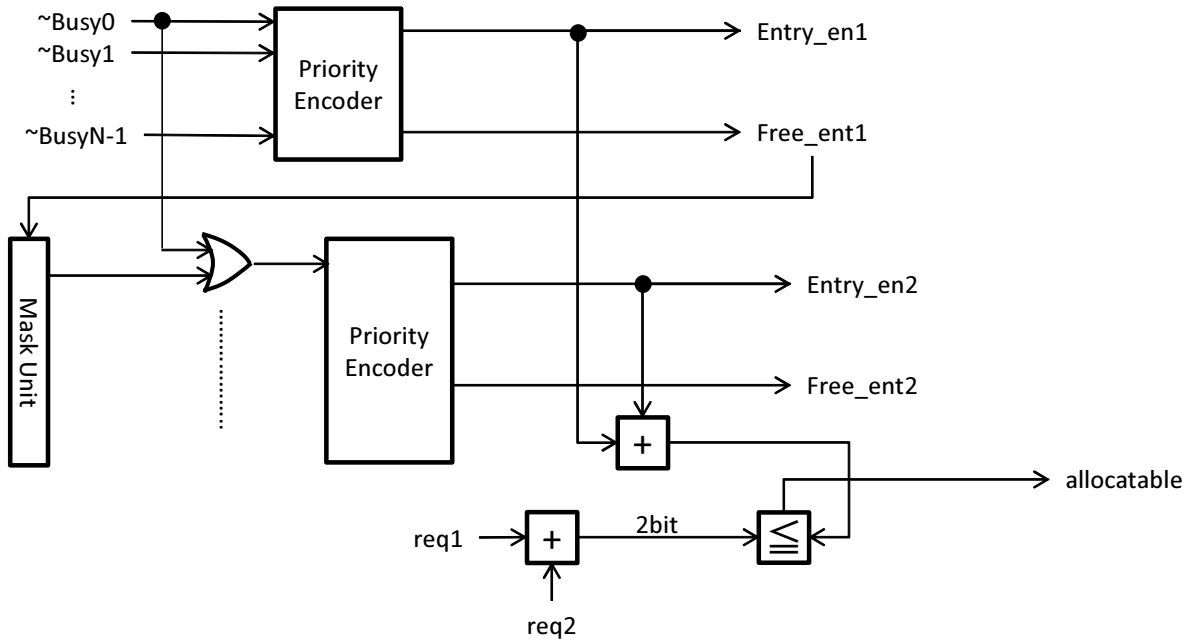


Fig.20 RS Free Entry Finder

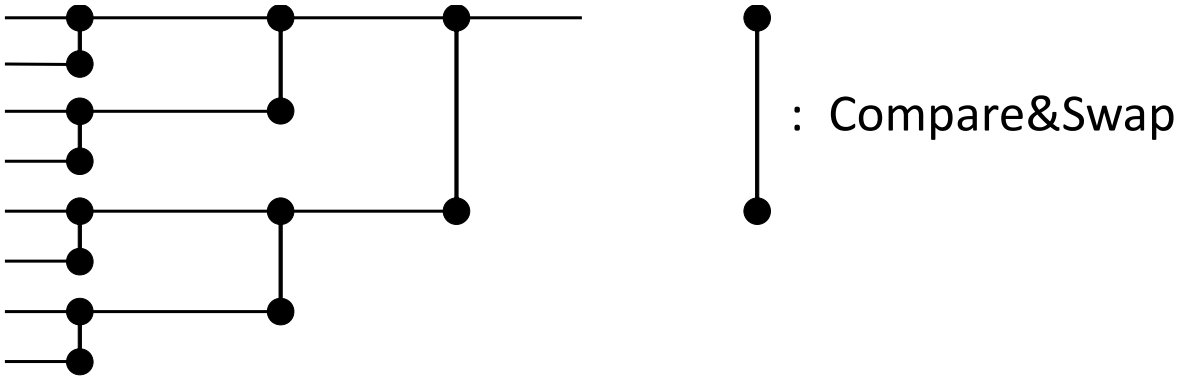


Fig.21 Minimum Selection (Oldest Finder)

not true when RRFTag overflows. Thus, as shown in Fig. 22, we insert a sorting bit between the MSB and RRFTag. When an instruction is dispatched to Reservation Station, its sorting bit is set. When RRFTag overflows, the sorting bits of all instructions in Reservation Station are cleared. By this way, Issue Unit can basically select the oldest instruction which has all the necessary operands. However, there is a limitation. In RIDECORE, up to two instructions can be dispatched to Reservation Station in one clock cycle. Thus, when the RRFTags of two instructions are 63 and 0 (since RRFTag is 6-bit, it overflows after becoming 63), the sorting bit of the instruction with RRFTag=0 is also cleared. This means that Issue Unit does not strictly realize the Oldest First algorithm.

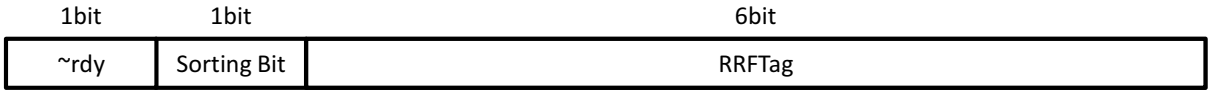


Fig.22 An entry in *oldest finder*

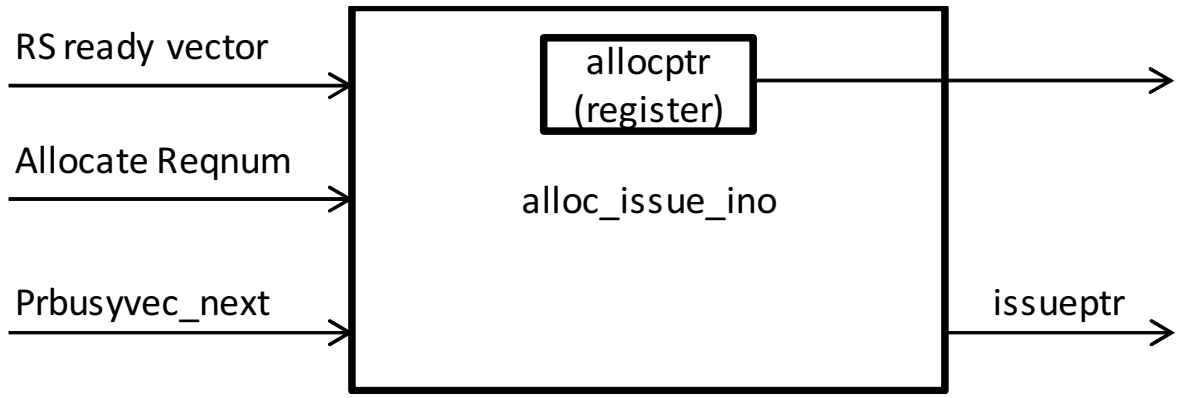


Fig.23 Allocate and Issue In Order

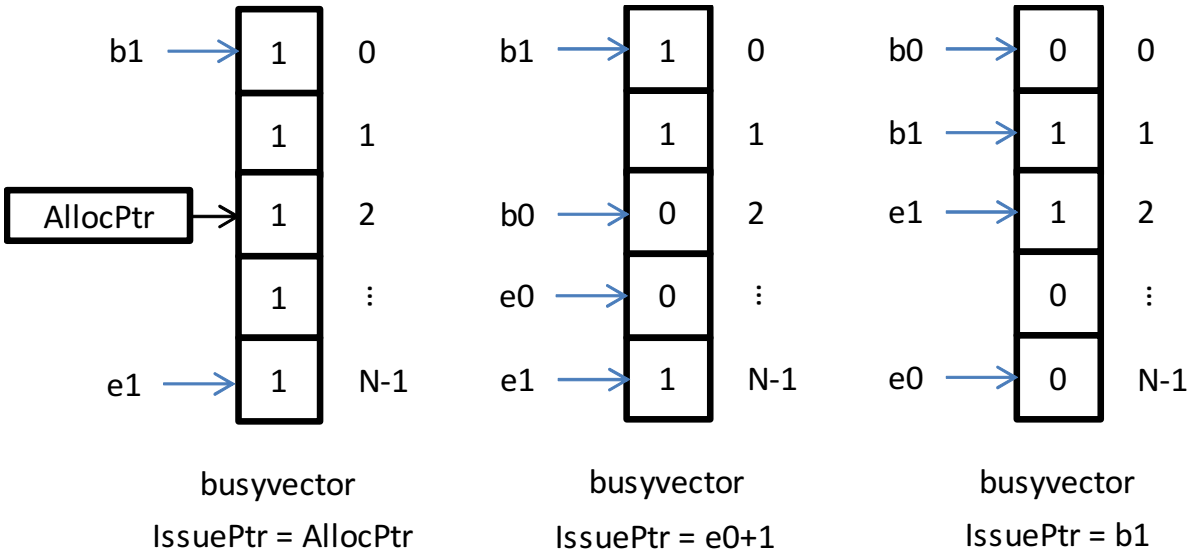


Fig.24 IssuePtr Calculation

15.2 In-Order Issue

Fig. 23 shows the circuit of *alloc_issue_ino* which uses Reservation Station as a FIFO buffer and realizes in-order execution. There is only one register *AllocPtr* in this circuit. *IssuePtr* points to the instruction which will be issued next, and is calculated from the busy vector in Reservation Station and register

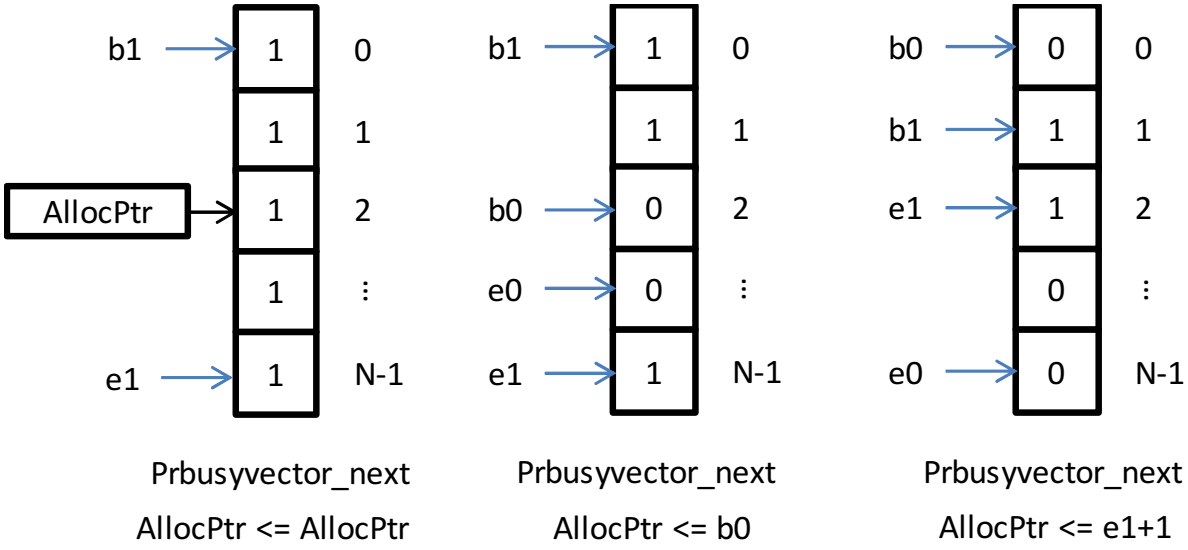


Fig.25 AllocPtr Re-calculation

AllocPtr. Fig. 24 shows how *IssuePtr* is calculated. *b0/1, e0/1* in the figure are calculated by *search_begin* and *search_end* (described later), respectively. *b0/1* is the entry number of the first zero/one-entry, while *e0/1* is the entry number of the last zero/one-entry, in the busy vector. There are three patterns of the busy vector.

- Pattern 1: all entries of the busy vector are equal to one. In this case, IssuePtr is set to AllocPtr.
- Pattern 2: b0 and e0 lie between b1 and e1. In this case, IssuePtr is set to e0 + 1.
- Pattern 3: b1 and e1 lie between b0 and e0. In this case, IssuePtr is set to b1.

When a branch misprediction occurs, *AllocPtr* must be recalculated since some speculative instructions in Reservation Station are invalidated. *AllocPtr* is recalculated based on *Prbusyvector_next* which is the busy vector after the invalidation. This recalculation is almost the same as the calculation of *IssuePtr*.

15.3 search_begin, search_end

Both *search_begin* and *search_end* are implemented as priority encoders. However, *search_begin* prioritizes lower bits while *search_end* prioritizes higher bits. These modules are used in *alloc_issue_ino* and *storebuf* to perform in-order execution.

16 exection unit(exunit_*) (pp. 203-206)

RIDECORE has five execution units: two ALUs, a multiplier, a load/store unit, and a branch unit). Fig. 26, Fig. 27, and Fig. 28 show the circuits of these execution units. Fig. 29 shows the circuit of "Kill Gen" used in Fig. 26, Fig. 27, and Fig. 28. When a branch misprediction occurs, Kill Gen determines whether to invalidate the instructions currently being executed in the execution units. You

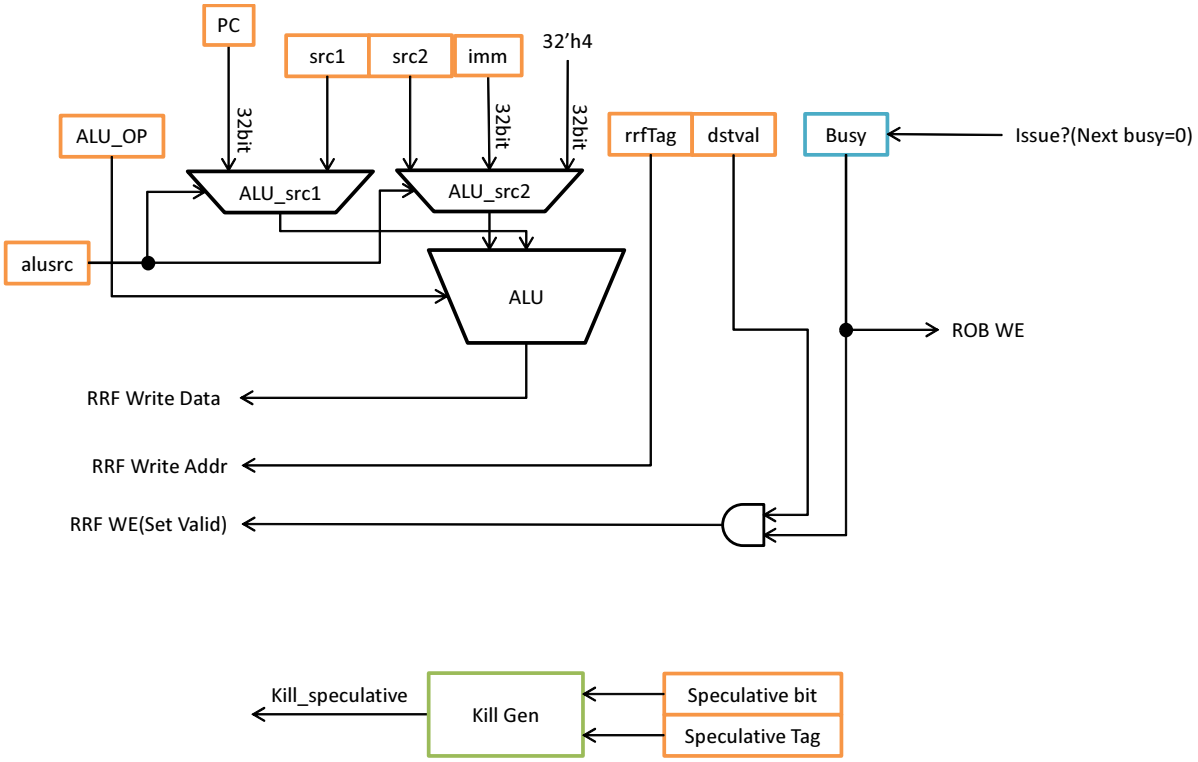


Fig.26 ALU

can understand this circuit by reading the section of Miss Prediction Fix Table. In RIDECORE, each multiplication operation takes one clock cycle. The multiplier circuit is thus almost the same as ALU.

When an execution unit finishes executing an instruction, it notifies the completion (*ROB WE*) to Reorder Buffer and writes data (*RRF Write Addr/Data*) to RRF. In addition to these basic operations, Load/Store Unit and Branch Unit perform some additional operations described below.

The operation of Load/Store Unit is pipelined with two stages. When this unit receives a Load instruction from Reservation Station, it accesses to DMEM and Store Buffer and receives data from Store Buffer in the 1st stage. In the 2nd stage, it receives data from DMEM and write data (data from Store Buffer if there is any data which has not been stored to DMEM yet; otherwise data from DMEM) to RRF.

When Load/Store Unit receives a Store instruction, it writes *store data* and *store address* to Store Buffer in the 1st stage and notifies the completion to Reorder Buffer in the 2nd stage. Once the store instruction finishes its COM stage, the store data in Store Buffer can be written to DMEM. Data in Store Buffer is written to DMEM whenever there is no Load instruction accessing to DMEM.

Branch Unit calculates *branch target* and compares it to the branch target predicted in IF stage. If a branch prediction is correct, we update *Miss Prediction Fix Table* and clear *Speculative bit* that matches Speculative Tag. On the other hand, if a branch misprediction occurs, we restore processor's state to the state before performing the branch prediction. In both cases, the pipeline is stalled (for backing up

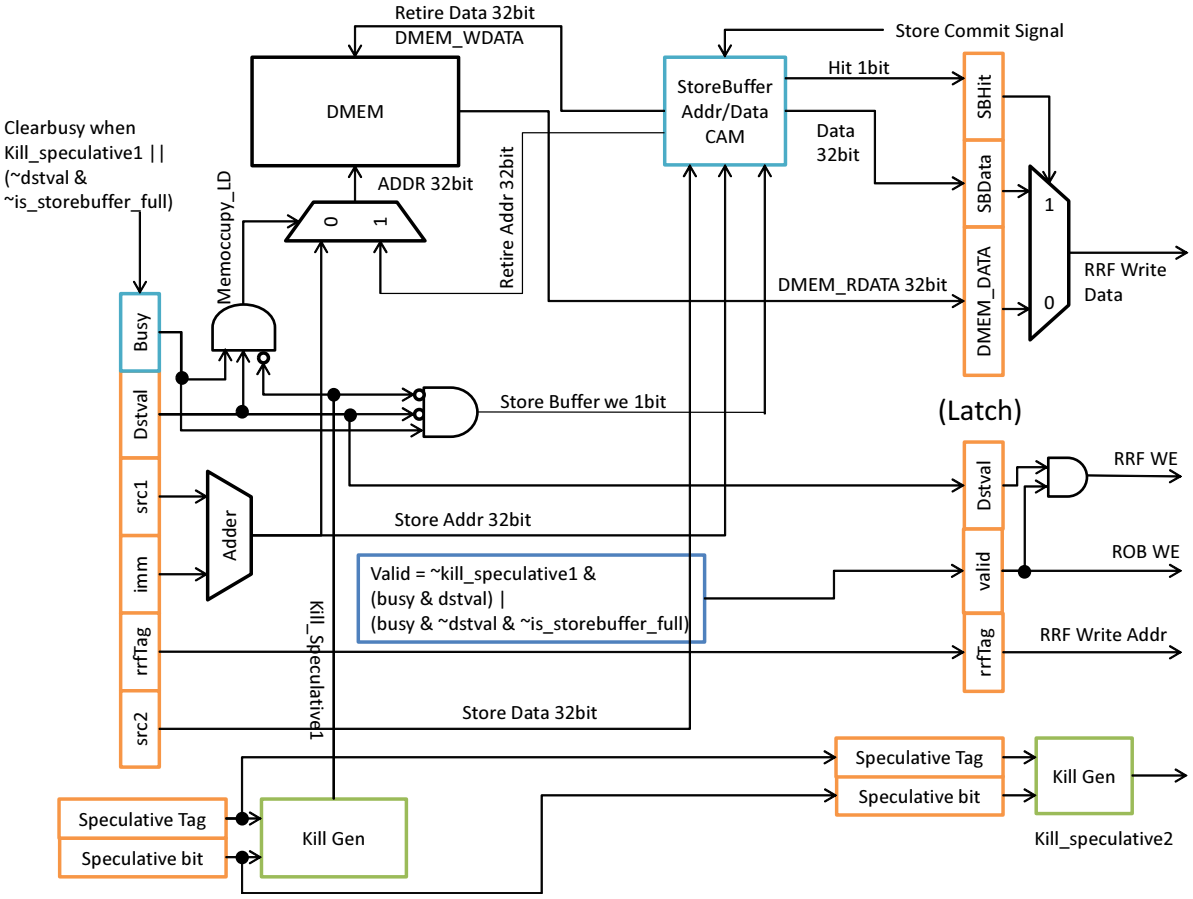


Fig.27 Load/Store Unit

or restoring). In COM stage, to notify the completion of a branch instruction to the branch predictor, we need information of this instruction. Thus, Branch Unit writes *branch target* and *branch condition* to Reorder Buffer.

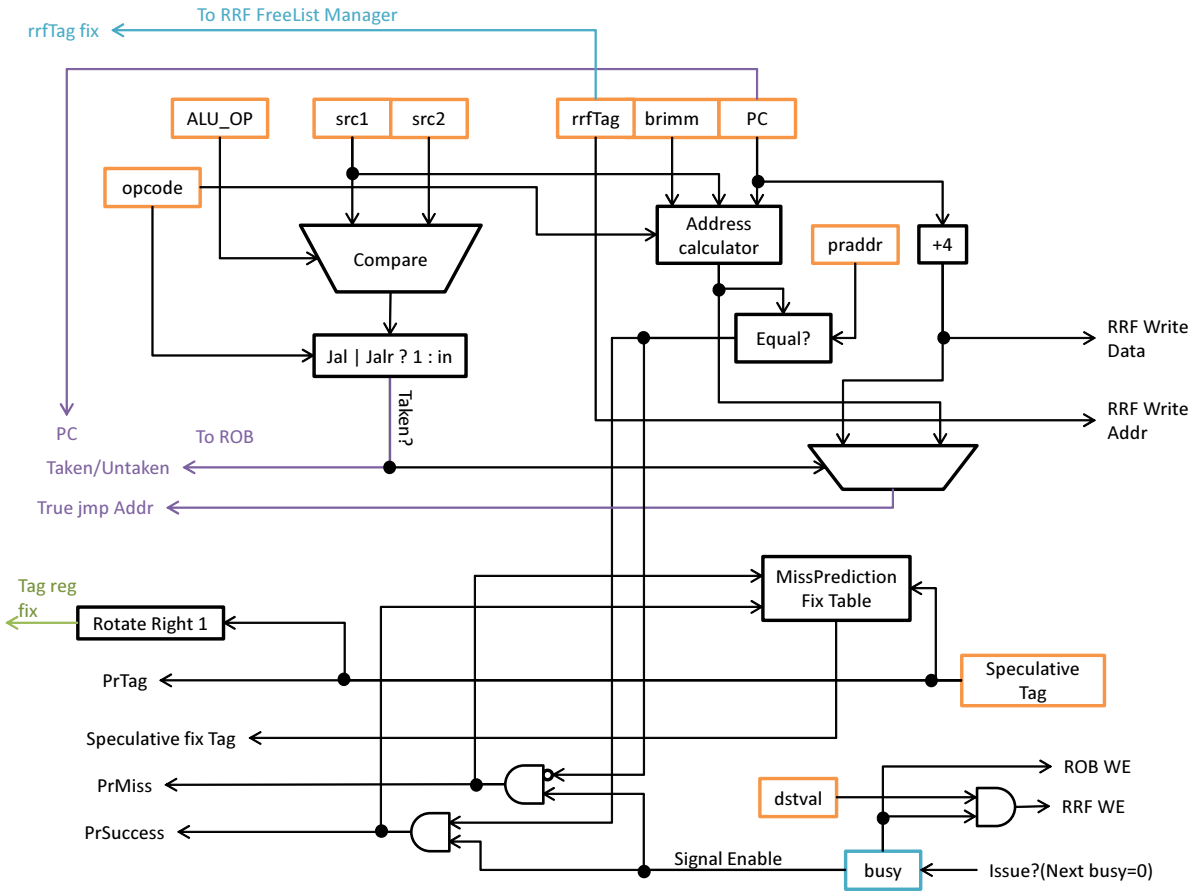


Fig.28 Branch Unit

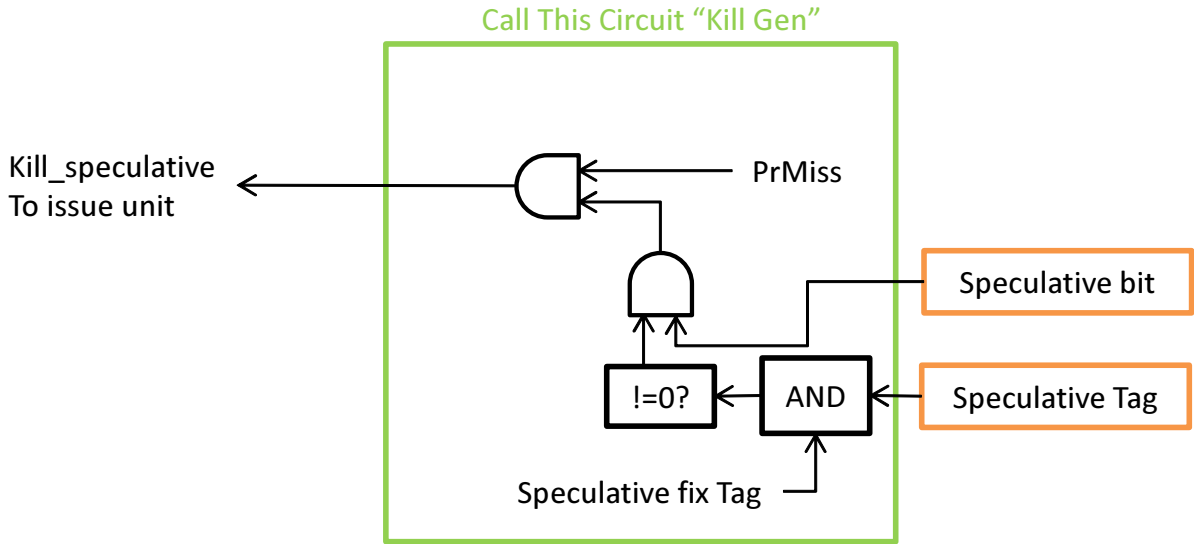
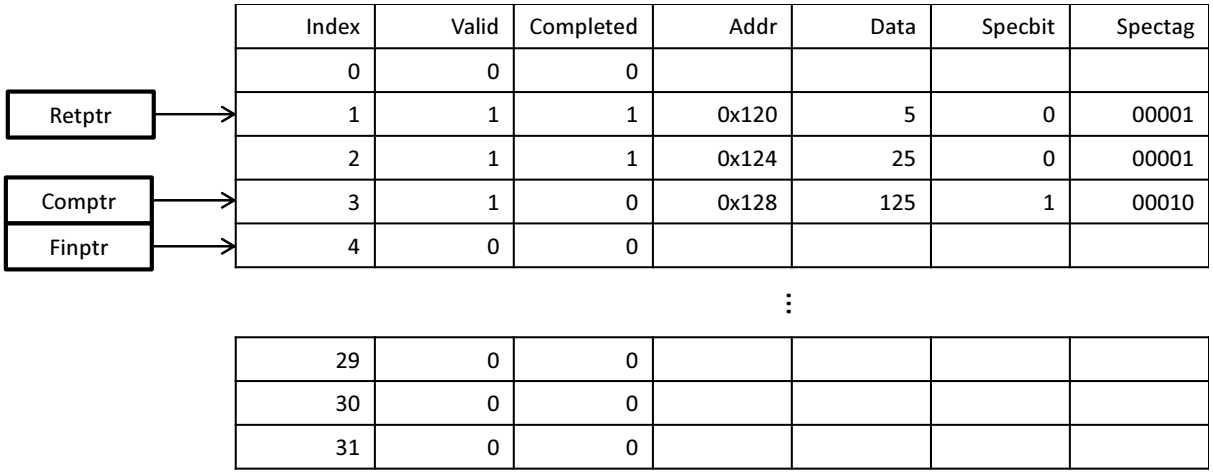


Fig.29 Kill Gen



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

17 storebuf (pp. 206-209, 262-273)

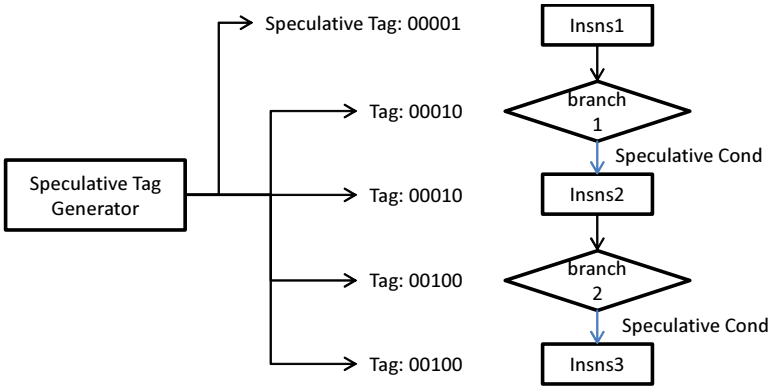
Store Buffer contains store data and store address of store instructions that have completed their EX stages. The store data of a store instruction can be transferred from Store Buffer to DMEM once the instruction finishes its COM stage. Store Buffer is an associative memory. When a load instruction gives a load address to Store Buffer, it returns the latest unstored data to the load instruction. The latest unstored data can be easily selected by rotating *load address hit Vector*. Fig. 30 shows the entries and registers of Store Buffer. There are three registers: *Finptr*, *Comptr*, and *Retptr*. *Finptr* is a pointer to finished instructions. *Comptr* is a pointer to completed instructions. Finally, *Retptr* points to instructions whose store data has been written to DMEM. When a branch misprediction occurs, *comptr* and *retptr* only need to be incremented, but *finptr* needs to be recalculated in the same way that *alloc_issue_ino* is recalculated.

Below is the explanation of the attributes of each entry in Store Buffer.

- valid: determines whether the entry is valid.
- completed: determine whether the instruction is completed. Once both valid and completed are equal to one, the instruction is ready to be retired.
- addr, data: data and address to write to DMEM.
- specbit, spectag: *specbit* determines whether the instrucion is speculative. *spectag* contains Speculative Tag. These information is required to invalidate the instruction when the branch prediction is not correct.

18 miss_prediction_fix_table (pp. 228-231)

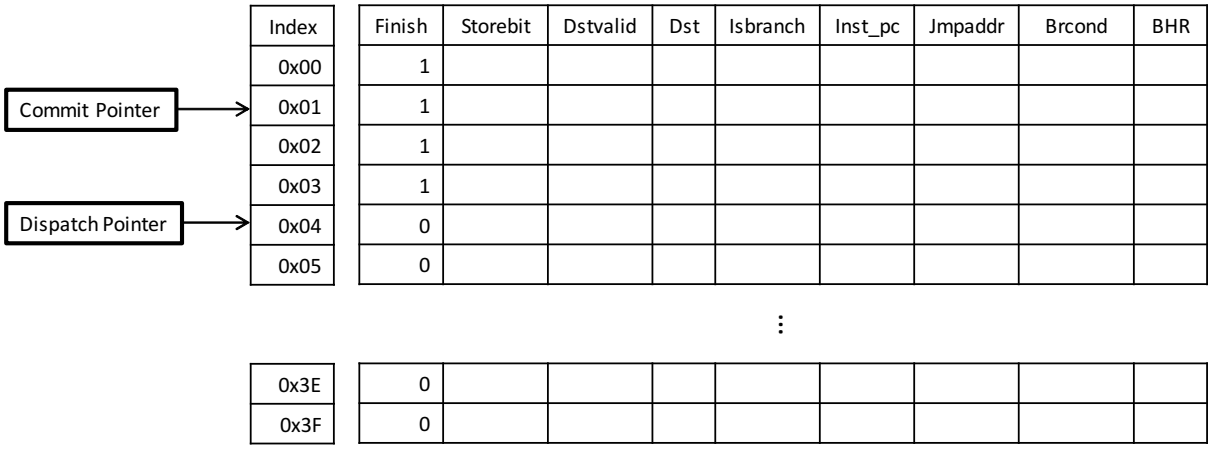
Miss Prediction Fix Table stores states of Speculative Tags. The state of a Speculative Tag is defined



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



Index	Finish	Storebit	Dstvalid	Dst	Isbranch	Inst_pc	Jmpaddr	Brcond	BHR
0x00	1								
0x01	1								
0x02	1								
0x03	1								
0x04	0								
0x05	0								
⋮									
0x3E	0								
0x3F	0								

Fig.32 Reorder Buffer

by two values: *Valid* and *Value*. *Valid* determines whether the instruction having this Speculative Tag is a speculative instruction. *Value* shows the dependency between this Speculative Tag and other Speculative Tags.

Next, we use a specific example (Fig. 31) to explain *Miss Prediction Fix Table*. Fig. 31 shows the instruction flow and the state of *Miss Prediction Fix Table* after decoding two branch instructions. Speculative Tags are assigned to the instructions by Speculative Tag Generator in the order of 00001, 00010, ..., 10000. When the prediction of *branch1* turns out to be incorrect, we read the Value of index 00010 (*branch1*'s Speculative Tag) in *Miss Prediction Fix Table*. The Value is 00110. Then, the instructions to be invalidated are the ones with Speculative Tags 00010 and 00100.

$$\forall i \in Instructions \ (((i.SpeculativeTag \ \& Value) \ != 0) \rightarrow i \ is \ invalidated)$$

19 reorderbuf (pp. 206-209, 254-259)

Reorder Buffer is a buffer which allows instructions to be committed in-order. Fig. 32 shows the entries and registers of Reorder Buffer. *comptr* is a pointer which points to the instruction to be completed next. *dispatchptr*, which is equal to *RRFPtr* in *rrf_freelistmanager*, is a pointer to the entry whose instruction will be dispatched next. Reorder Buffer can complete at most two instructions per clock cycle. However, branch instructions and store instructions are completed one by one to reduce the number of write ports of memories in the Branch Predictor and Store Buffer.

When an instruction is completed, Renaming Table is updated and the execution result of the instruction written in RRF is copied to ARF. *Busy* in Renaming Table is cleared only when RRFTag of Renaming Table is equal to that of the completed instruction.

Below is the explanation of the attributes of each entry in Reorder Buffer.

- finish: indicates whether the instruction has been finished. It is cleared when the instruction is dispatched and set when the instruction is finished.
- storebit: indicates whether the instruction is a store instruction. If storebit is equal to one (i.e. store instruction), it is necessary to notify the completion of the instruction to Store Buffer.
- dstvalid: indicates whether the destination register of the instruction is valid, i.e. whether the instruction needs to write data to ARF.
- dst: destination register number.
- isbranch: determines whether the instruction is a branch instruction. If isbranch is equal to one (i.e. branch instruction), we have to write data to Branch Predictor (PHT, BTB, etc.) upon the completion of the instruction.
- inst_pc, jmpaddr, brcond, bhr: data to Branch Predictor. inst_pc, jmpaddr, brcond, bhr are the instruction address, branch target, branch condition, and *bhr* (for calculating PHT's write address), respectively.

References

[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)