

RIDECORE ドキュメント

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

2016/03/28

第 I 部

RIDECORE 概要

本章では、RIDECORE 全体の動作を説明する。次に、RIDECORE の仕様を決定づける様々なパラメータを示す。

1 全体の動作

RIDECORE の全体図を Fig. 1 に示す。RIDECORE は 6 段パイプラインとなっており、各ステージを IF(Instruction Fetch) ,ID(Instruction Decode) ,DP(DisPatch) ,SW(Select and Wakeup) ,EX(EXecution) ,COM(COMplete) と呼ぶ。各ステージは EX の一部を除き 1 サイクルで処理される。以下、各ステージの解説を行う。

IF ステージでは、PC(Program Counter) を用いて命令メモリ (IMEM) から命令を最大 2 つまで読み出し、後段のステージに送る。クロックの立ち下がり時に IMEM からデータを読み、クロックの立ち上がりで IF/ID 間のラッチに命令データを送るので、1 サイクルで処理が可能である。PC の更新は Gshare 分岐予測器を用いて投機的に行う。

ID ステージでは、IF ステージで読みだした 2 命令のデコードを行い、後段のステージに必要なデータを生成する。また、Speculative Tag gen が 2 つの命令に投機タグの付与を行う。デコードした命令が分岐命令である場合には、Speculative Tag gen 及び Miss Prediction Fix Table の情報を更新する。

DP ステージでは、2 命令分の ARF のレジスタ値、リネーミング情報及び RRF のデータを用いてオペランドのフェッチを行う。次にリオーダバッファ及び RRF のエントリを割り当てることでリネーミングを行い、Allocate Unit が命令の実行に必要なデータをリザベーションステーション (RS) へ格納する。RS に格納するオペランドは、Source Operand Manager を用いてフォワーディングを行う。

SW ステージでは、RS 内にあるオペランドの揃った命令を、Issue Unit が選択して EX ステージへ発行する。発行する際には、ラッチとして存在する EX Src * に必要なデータを書き込むと同時に、RS から発行する命令を削除する。

EX ステージでは、命令の実行を行う。実行が終了した際に実行結果を RRF に書き込み、リオーダバッファに命令の終了通知を行う。Branch Unit では投機実行の成功失敗が確定するため、これを RIDECORE 全体に通知して然るべき処理を行う。分岐予測失敗時の投機的命令の無効化処理は Miss Prediction Fix Table を用いて行う。また LDST(Load/Store) Unit は例外として内部が 2 段パイプラインとなっているが、他の実行

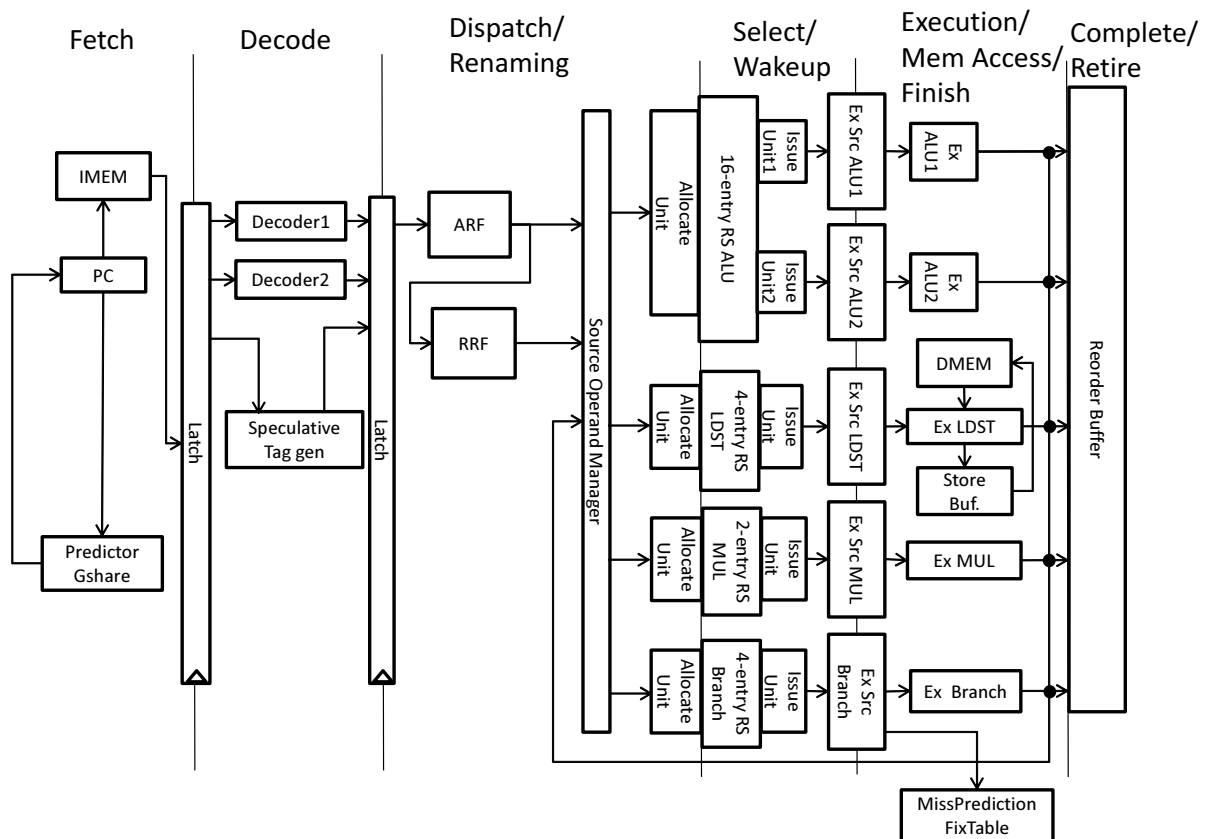


Fig. 1 RIDECORE pipeline

ユニットは 1 サイクルで命令を実行することができる。

COM ステージでは、リオーダーバッファ内にある命令を最大 2 つまで完了する。命令が完了する際には、RRF に書き込まれているデータを ARF に移動し、ARF 内のリネーミングテーブルを更新する。また、分岐予測のための情報を分岐予測器に通知し、データの書き込みを行う。

2 仕様概要

仕様概要を TABLE1 に示す。

Table1 Spec

命令セット	RISC-V(RV32IM の一部)
Way 数	2 (IF, ID, DP, COM)
データ/アドレス幅	
データ幅	32
アドレス幅	32
エントリ数	
物理レジスタ (ARF)	32
論理レジスタ (RRF)	64
リオーダーバッファ	64
投機タグ数	5
リザベーションステーションエントリ数	
ALU	16
Load/Store	4
Branch	4
MUL	2
演算器数	
ALU	2
Load/Store	1
Branch	1
MUL	1

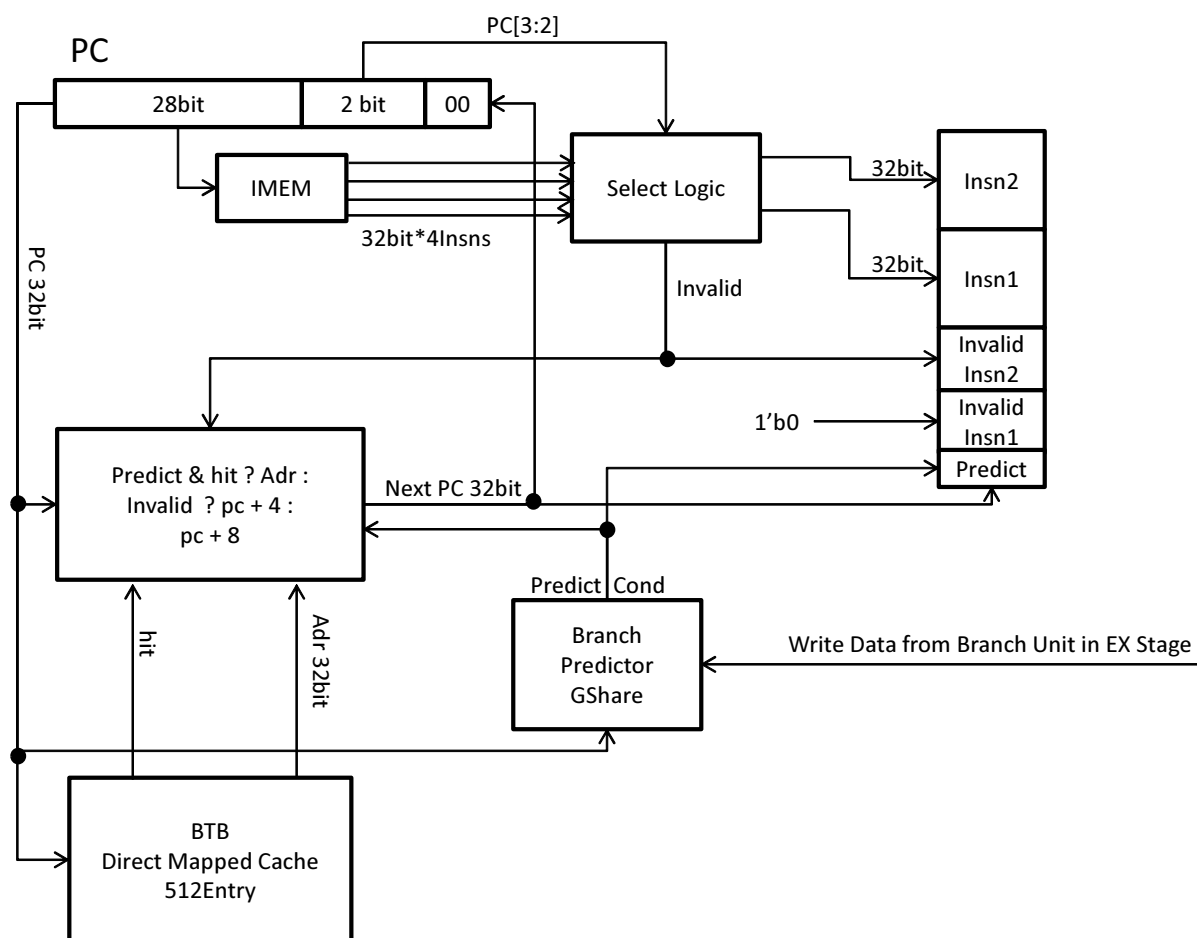


Fig. 2 pipe_if

第II部

各ハードウェアモジュール解説

ここでは各ハードウェアモジュールの解説を行う。幾つかのセクションには [1] に該当する部分のページが書かれている。例えば tag-generator (pp. 228-231) と書かれている場合、そのモジュールは [1] の 228 ページから 231 ページに対応していることを示している。ページ数が記されている場合、該当するページを読んでおくことが推奨される。

3 pipe_if

IF ステージでは、IMEM から 2 つの命令のフェッチを行い、分岐予測器を用いて PC の更新を行う。IF ステージの回路を Fig. 2 に示す。pipe_if は図中のラッチ、PC レジスタ以外のすべての部分に対応する。

IMEM は 1 エントリが 128bit である。RIDECORE で使用している RISC-V のサブセットでは、命令長は 32bit なので、1 エントリで 4 つの命令を持つことになる。クロックの立ち下がり時に PC[31:4] を用いて

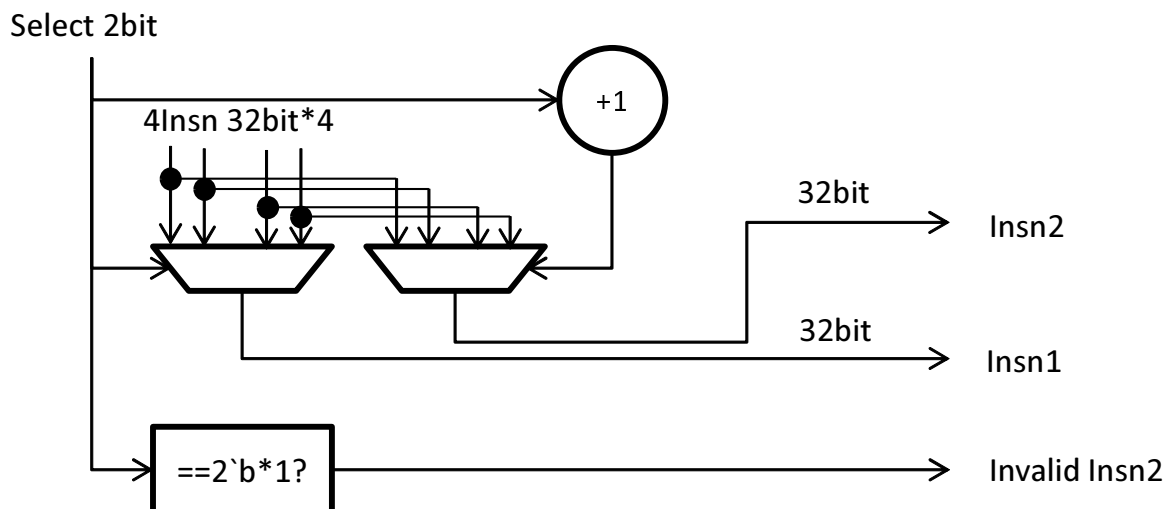


Fig. 3 Select Logic

IMEM から 4 命令分を読みだし，PC[3:2] を用いて 4 命令から 2 命令を選択する．クロックの立ち上がり時に取得した 2 命令を Latch に書き込む．選択する回路を Fig. 3 に示す．基本的には 4 命令から 2 命令を選択するが，PC が 8 の倍数でない場合，2 つ目の命令を無効とするために Invalid 信号を出す．例えば PC=0x2c の場合，IMEM からは 0x20/0x24/0x28/0x2c 番地にある命令が出力され，Fig. 3 の回路における Insn2 の値は，0x20 番地の命令となってしまうからである．

図の複雑化を防ぐため Invalid Insn1 には常に 0 を書き込むよう記載しているが，実際には分岐予測ミスなど，IF ステージの命令を kill する必要がある時に 1 となるような回路となっている．

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

グローバル履歴レジスタ BHR(Branch History Register) と，飽和カウンタの配列 PHT(Pattturn History Table)，PC とその PC の指す分岐命令の分岐先を記憶する BTB(Branch Target Buffer) を使って，現在の PC の指す命令において分岐が起こるか否かを IF ステージで判定する回路である．回路図を Fig. 4 に示す．詳しい説明は書籍 [1] と [3] に譲り，ここでは実装について述べる．

クロックの立ち下がり時に PC[12:3] と BHR の XOR を取り，これを Read Address として PHT のデータを読む．PHT の該当データが 2 以上なら分岐が起こり，2 未満なら分岐が起こらないとする．BHR は分岐予測の結果を用いて，分岐予測が起こるたびに値を更新する．予測が失敗した際に BHR を復元するために，BHR は分岐予測が起こった時点でのバックアップを持つ．

4.1 pht

PHT は Gshare 分岐予測器で用いる飽和カウンタの配列である．回路図を Fig. 5 に示す．PHT の更新は分岐命令の Complete 時に行う．分岐命令の結果が Taken ならば PHT の該当エントリを increment し，Untaken ならば decrement する．クロックの立ち下がり時 PHT の該当エントリを読みだし，increment/decrement を計算する．その後クロックの立ち上がりで PHT の同じ場所書き込みを行う．

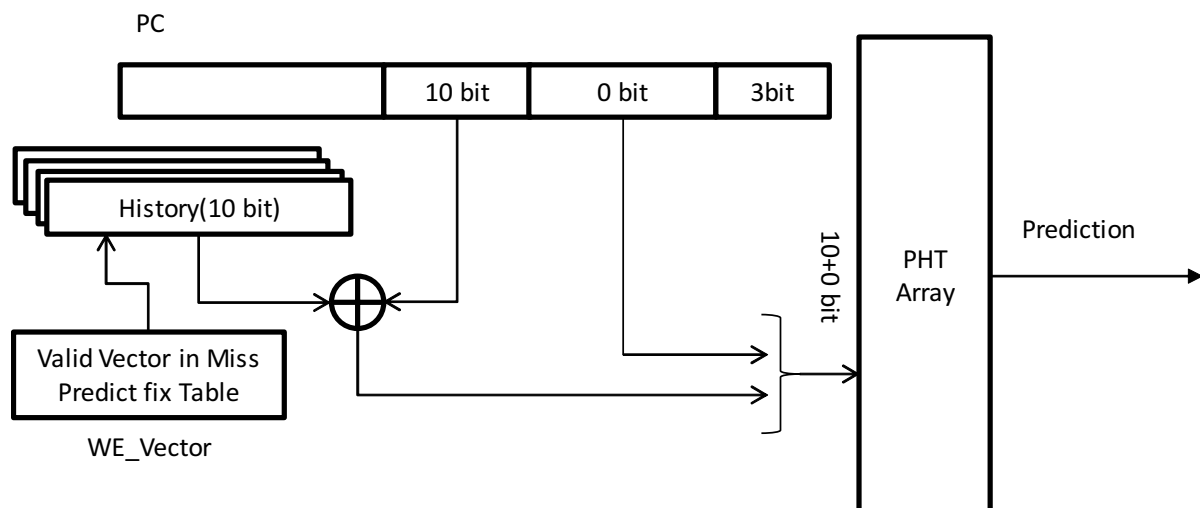


Fig. 4 Gshare Predictor

PHT では上述のように increment/decrement を計算する必要がある．そのため，IF ステージと COM ステージで Read を行い，COM ステージで Write を行う，2Read-1Write のメモリが必要である．このメモリを True Dual-Port RAM を 2 つ用いて実現している．

4.2 btb

btb(Branch Target Buffer) は分岐元のアドレスと分岐先のアドレスをペアで持つ回路である．btb はハードウェア量削減のために Direct Mapped Cache として実装されている．そのため，間違った分岐先を記憶している可能性があり，分岐予測の精度は低下する．

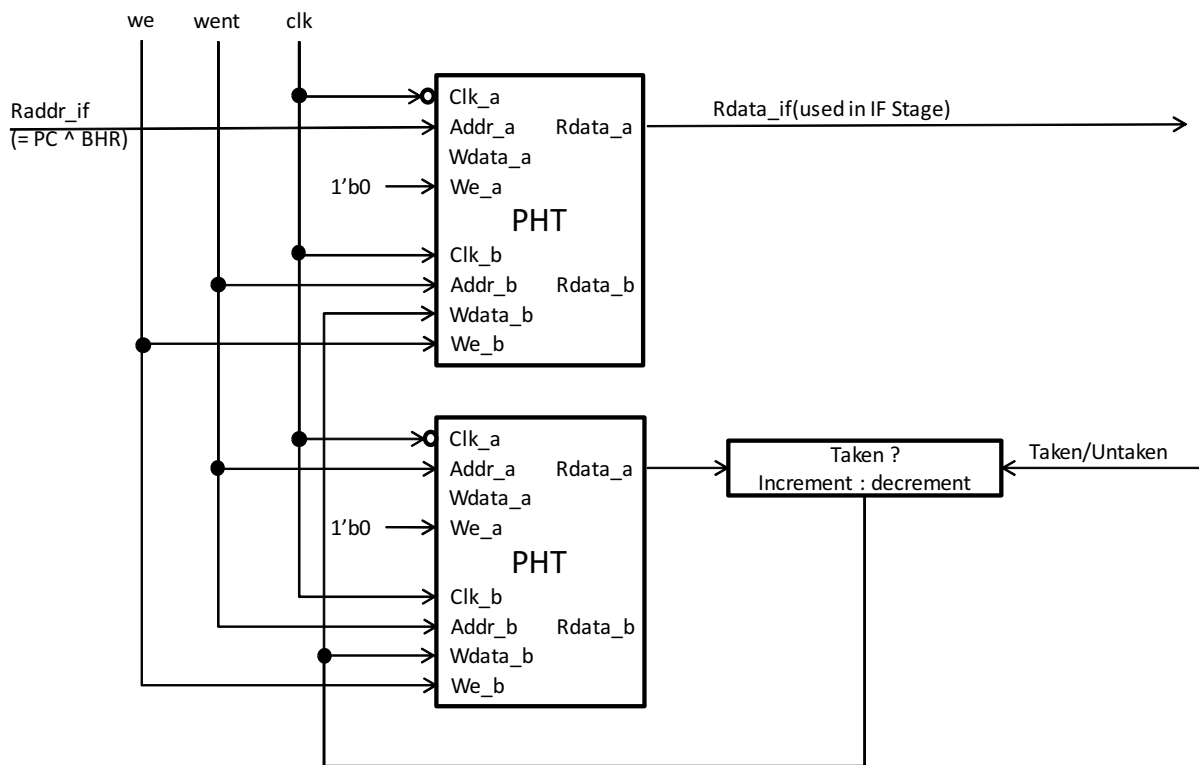


Fig. 5 PHT

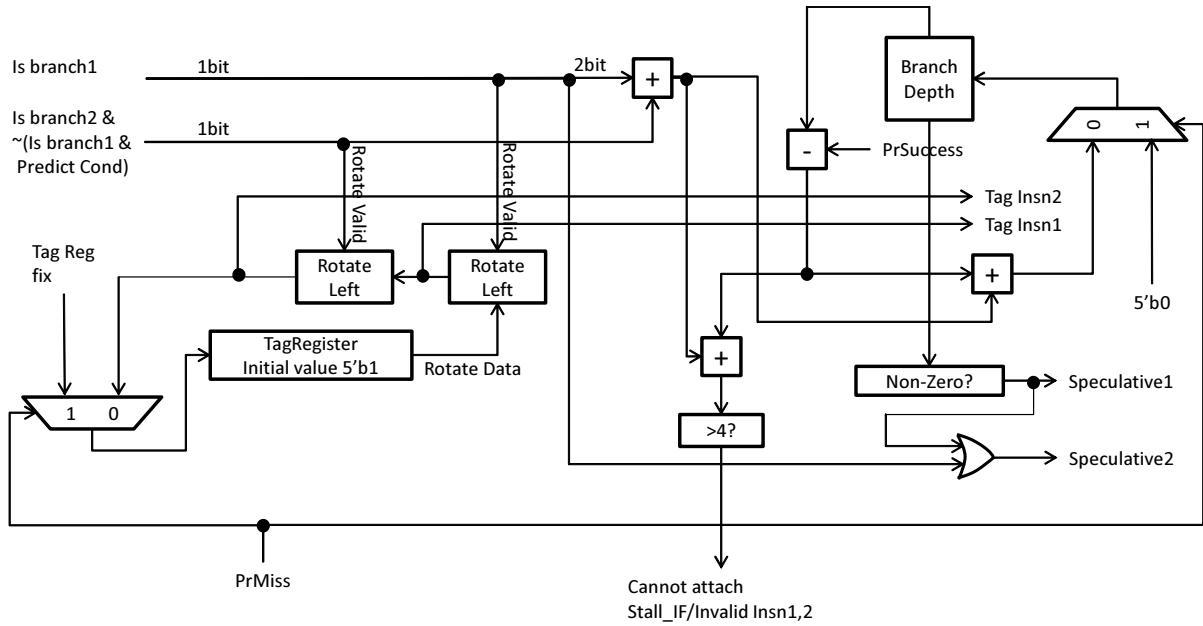


Fig. 6 Speculative Tag Generator

5 tag_generator (pp. 228-231)

Tag Generator の回路図を Fig. 6 に示す．Tag Generator は，内部にレジスタとして，現在割り当てている Speculative Tag(tagreg) と現在の分岐の深さ (brdepth) を持つ．2 つの命令 (Insn1, Insn2) がそれぞれ有効な分岐命令であることを示す信号線，branchvalid1 と branchvalid2，及び命令の発行が可能であることを示す enable を受け取る．それら信号に応じて Speculative Tag(sptag1, sptag2) 及び自身が投機命令であることを示すフラグ (speculative1, speculative2) を生成し，tagreg, brdepth の更新を行う．

Speculative Tag は，00001 から順番に 00010, 00100... というように左ローテートを行いながら割り当てられる．割り当てられるタグがなくなってしまった場合は，タグ割り当て可能フラグ (attachable) に 0 を出力し，ストールを発生させる．

分岐予測に成功した場合 (prsuccess=1)，Speculative Tag の開放を行うため，brdepth をデクリメントする．一方分岐予測に失敗した場合 (prmiss=1)，tagreg 及び brdepth を分岐予測前の状態に戻す必要がある．そのため，tagreg の修復情報，tagregfix を分岐命令実行ユニットから受け取り，修復を行う．brdepth は 0 としているが，分岐命令はインオーダ実行を行うため，分岐予測前の状態に復元した際，その時点での投機命令は必ず 0 個となるためである．

6 decoder

decoder は ID ステージで用いるモジュールである．命令のデータ (32bit) を受け取り，後段のステージに必要なデータを生成する．Fig. 7 に decoder の Verilog 記述の内，入出力を定義している部分を示す．入力 (input) は命令データ (inst) のみで，他はすべて出力 (output) である．図中に output reg と定義している部


```

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

分があるが、これは Verilog HDL にて case 文を使用するための reg 宣言であり、実際は wire となる。

以下、各出力信号線の解説を行う。

- imm_type: imm_gen に用いる。RISC-V では命令データ中の即値のフォーマットが異なり、命令データから即値を抽出するために必要である。
- rs1, rs2, rd: それぞれ第 1 ソースオペランド、第 2 ソースオペランド、デスティネーションのレジスタ番号である。
- src_a_sel, src_b_sel: ALU に入れる 2 つの値を選択するための制御線。RISC-V では計算にレジスタ、即値、Program Counter など様々な値を使用するため、この制御線が必要である。
- wr_reg: デスティネーションレジスタに書き込みを行う命令であることを示す制御線。
- uses_rs1, uses_rs2: rs1, rs2 レジスタがそれぞれ有効なものであり、値のフェッチが必要であることを示す制御線。RS では値が生成されるまで命令の発行ができないため、不要なレジスタのフェッチを防ぐために必要である。
- illegal_instruction: RISC-V で定義されていない命令であることを示す信号線。RIDECORE では用いていないが、主に例外処理で使用する。
- alu_op: ALU で計算する演算の種類を示す制御線。
- rs_ent: 登録すべき RS の番号を示す制御線。以下にそれぞれの実行ユニットにおける RS の番号を示す。
 - ALU: 1
 - BRANCH UNIT: 2
 - MULTIPLIER: 3
 - LOAD/STORE: 4

- `dmem_size`, `dmem_type`: Decode した Load/Store 命令のメモリサイズ (1-byte, 2-byte, 4-byte) を示す。RIDECORE では 4-byte の Load/Store のみ実装しているため、この信号線は用いていない。
- `md_req_op`: 乗除算命令で用いる制御線。Decode した命令が乗算/除算/剰余算のいずれに該当するかを示す。RIDECORE では乗算のみ実装しているため、この制御線は用いていない。
- `md_req_in_1_signed`, `md_req_in_2_signed`: 乗除算器に入る第 1/2 オペランドが符号ありであることを示す。
- `md_req_out_sel`: 乗除算器の出力を選択する制御線。乗算の演算結果は 64bit であり、RISC-V では演算結果の上位 32bit と下位 32bit のどちらを用いるか選択することが可能となっている。本制御線はこれに用いる。

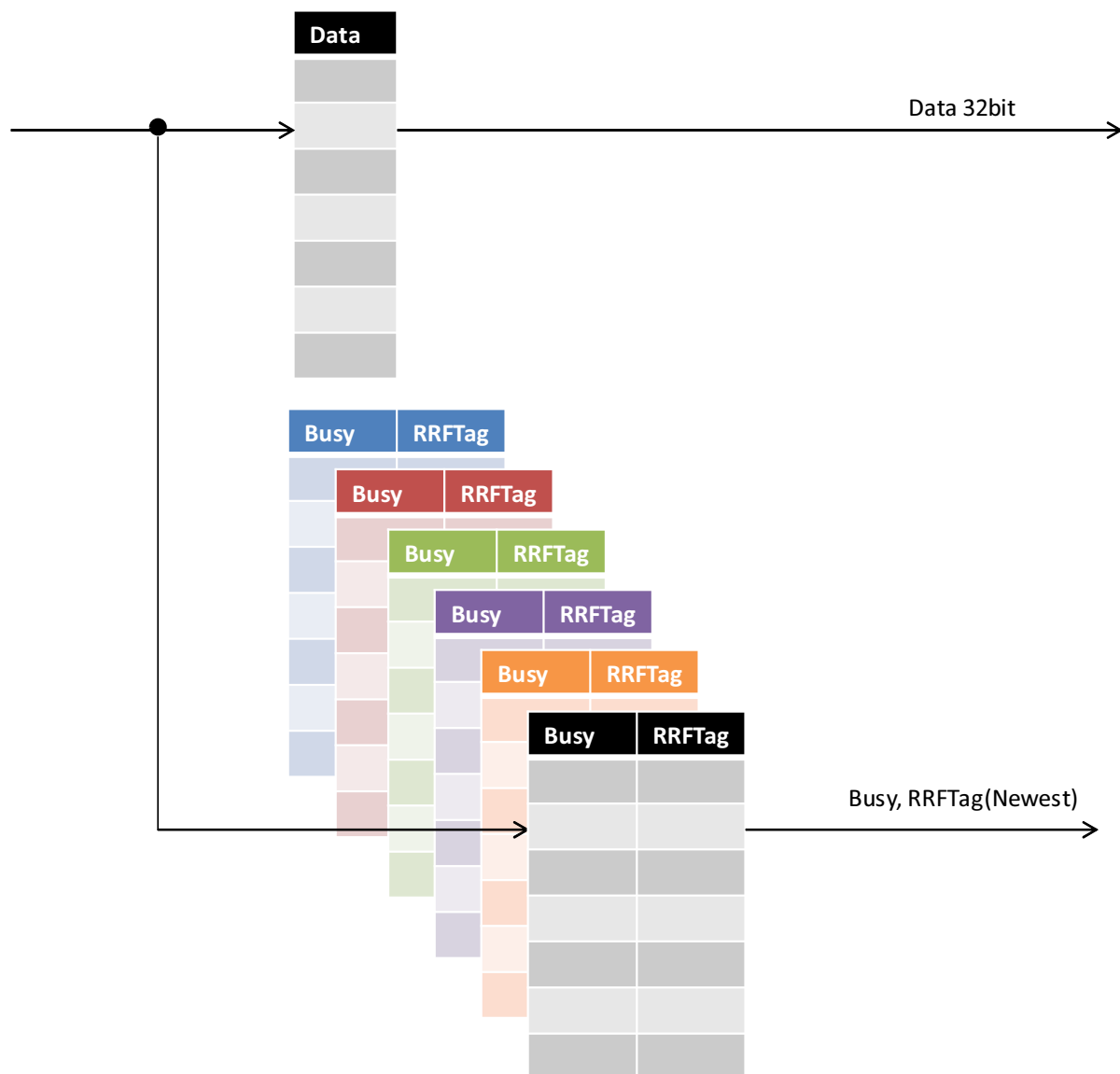


Fig. 8 ARF Read

7 arf (pp. 239-244)

ARF(Architected Register File) の各エントリには、COM ステージを終えて完了状態にあるレジスタの値 (Data) と、Renaming 情報 (RRFTag, Busy) が格納されている。回路図は Fig. 8, 9, 10 のようである。Renaming 情報の格納される Renaming Table は、投機実行を行う際に分岐予測ミスが生じた場合、分岐が起こる前の情報を復元する必要がある。そのため、Renaming Table は Speculative Tag の数だけ用意しておき、常にバックアップを取りながら実行を行う。以下では ARF の Read/Write 及び Renaming Table の復元動作について説明する。

まず、Read について説明する。Read 動作は Fig. 8 のように行う。Renaming Table は最新の値 (黒の Table) を使用する。次に Write について説明する。Write 動作は Fig. 9 のように行う。DP ステージでは Renaming Table のみに書き込みを行い、COM ステージでは Data と Renaming Table の両方に書き込む。Renaming Table は分岐予測ミス時に復元を行うためのバックアップをとるため、mpft(Miss Prediction Fix Table) の valid 列の否定を we(Write Enable) として書き込みを制御する。なお、mpft の valid は、該当する Speculative Tag が使用されている時に 1 となる値である。詳細は miss_prediction_fix_table の項を参照されたい。

最後に Renaming Table の復元動作について説明する。復元は Fig. 10 のように行う。この動作は分岐予測ミス時及び成功時に行う。Renaming Table の RRFTag 及び Busy は、長さ 32bit の Bit Vector を用いて実装しており、1 サイクルで全 Table のエントリを書き換えることが可能である。Renaming Table を正しく復元するため、パイプラインにストールを発生させる。

以下では Fig. 11 の具体例を用いて、branch1(SpeculativeTag = 5'b00010) の予測成功/ミス時の動作を説明する。まず予測ミス時の動作を説明する。ミス時はすべての Renaming Table の情報を branch1 以前のもの、すなわち SpeculativeTag = 5'b00001 の時点での情報に復元する必要がある。そのため、we である Fix Vector をすべて 1(6'b111111) とし、マルチプレクサで選ぶ Fix Data は図中の赤い Renaming Table を選択する。これにより情報の復元が完了する。

次に予測成功時の動作を説明する。成功時には、Renaming Table のバックアップを破棄する必要がある。破棄した Table はすぐにバックアップとして使える状態にする必要があるので、Table を最新の情報、つまり黒の Table の情報に置き換える必要がある。そのため、今回の例では SpeculativeTag = 5'b00001 のバックアップを持つ赤い Renaming Table のみを最新の情報に置き換えるため、Fix Vector = 6'b010000 とし、マルチプレクサで選ぶ Fix Data は図中の黒い Renaming Table を選択する。

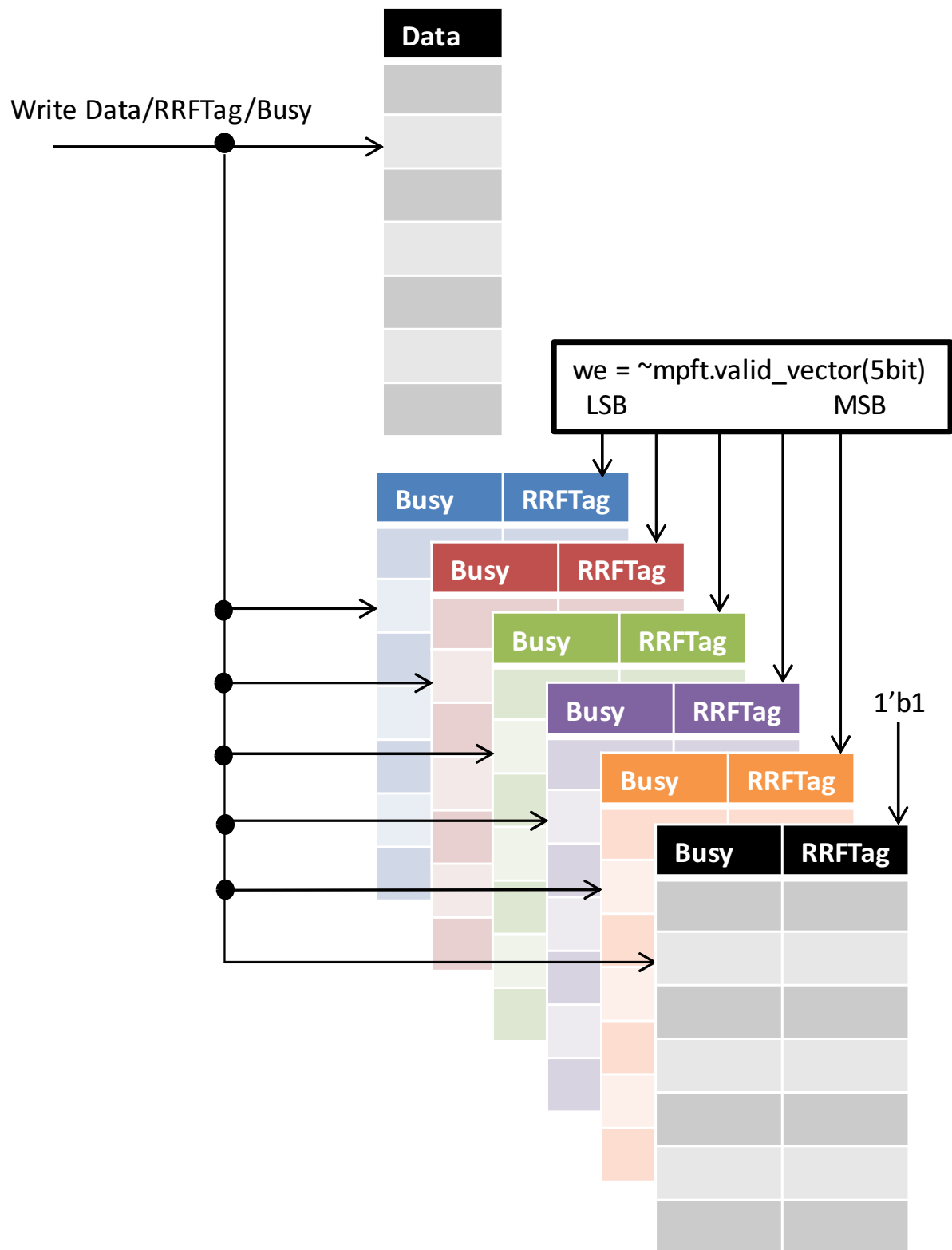


Fig. 9 ARF Write

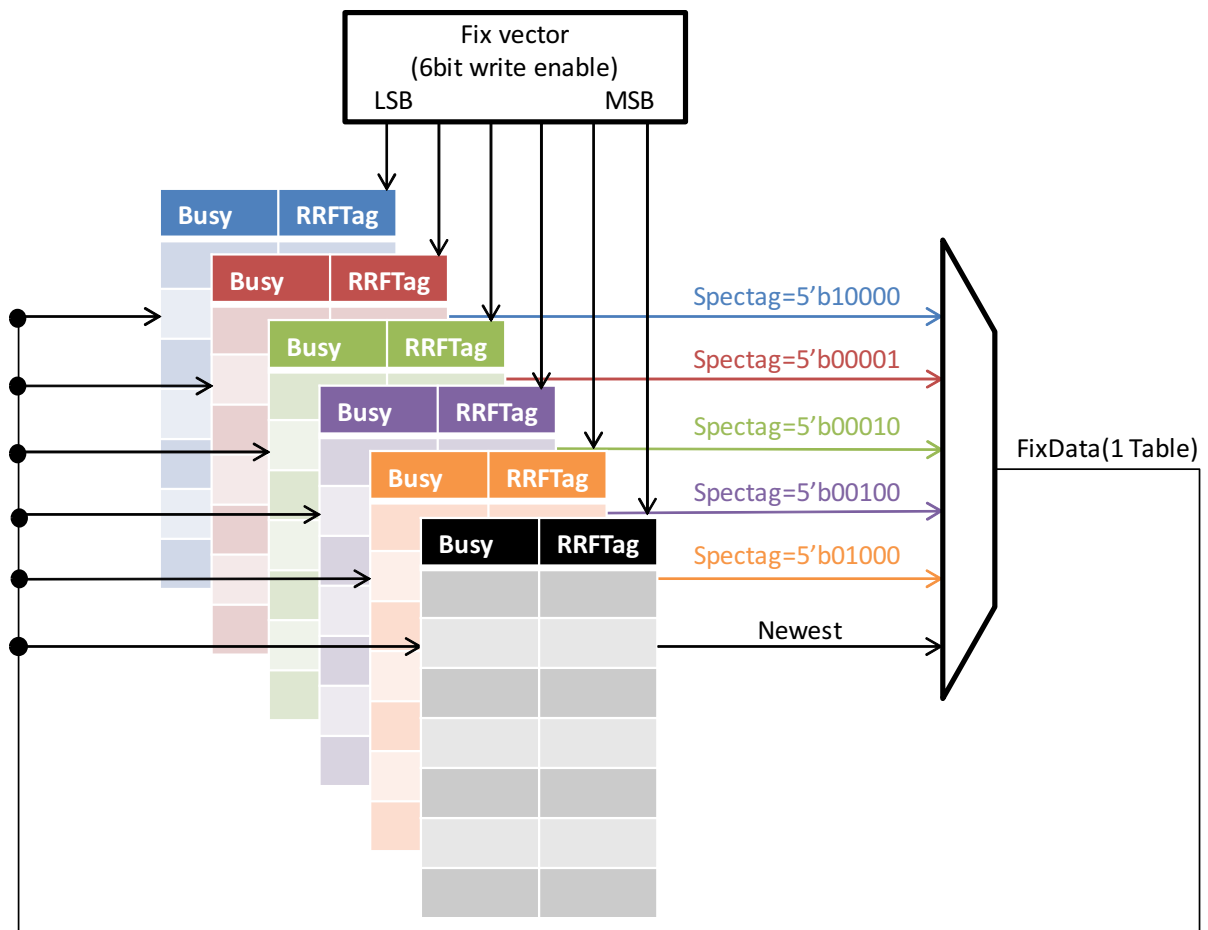


Fig. 10 Fix Renaming Table

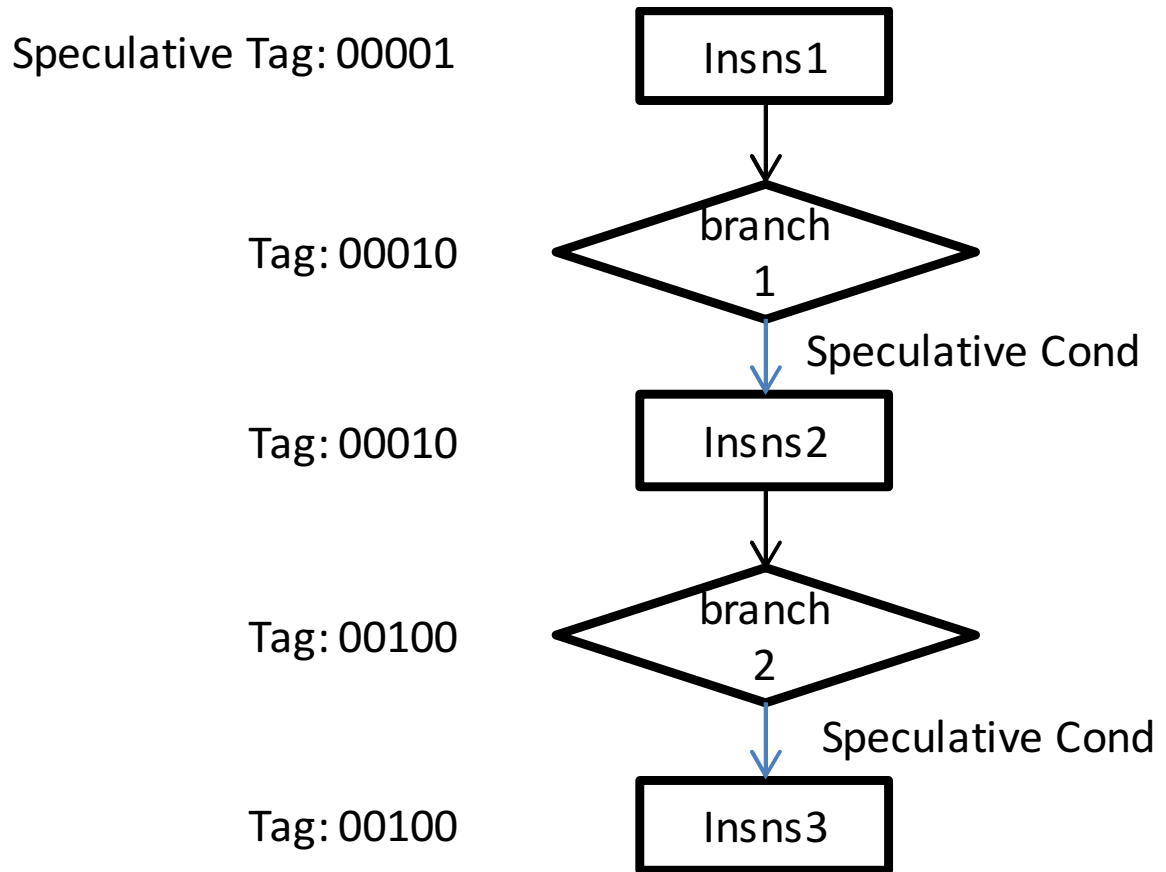


Fig. 11 Example of Speculative Execution

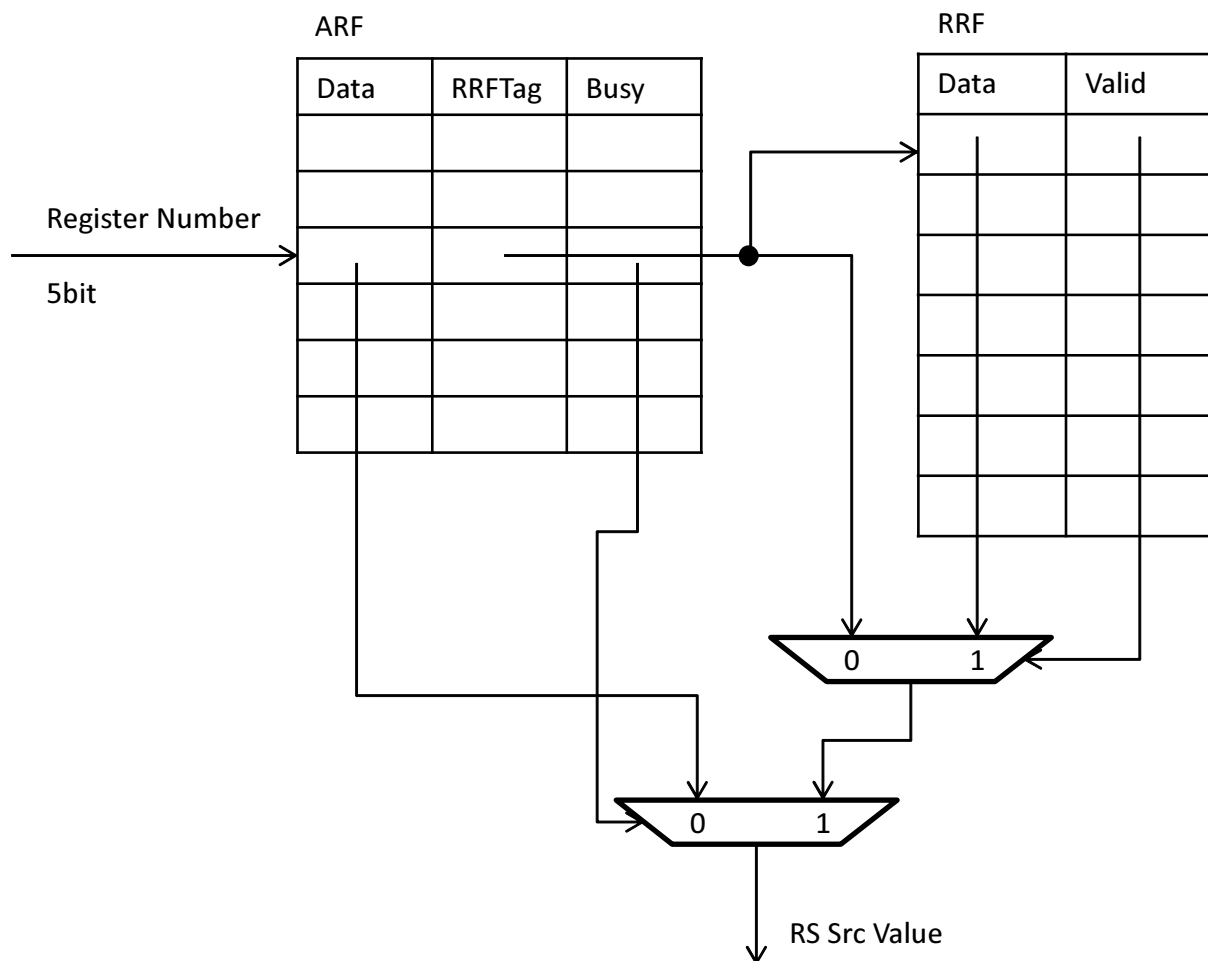


Fig. 12 ARF , RRF を用いた Register Renaming

8 rrf (pp. 239-244)

RRF(Renamed Register File) の各エントリには ,該当する命令の終了状態にあるレジスタの値 (RRFData) と ,実行結果が有効であることを示す情報 (RRFValid) が格納されている . RRF のエントリ数は Reorder Buffer と同じ 64 エントリであり ,RRF と Reorder Buffer は 1 対 1 に対応している . そのため ,RRF の Tag のみを用いて RRF と Reorder Buffer の両方にアクセスすることが可能である .

9 sourceoperand_manager (pp. 239-244)

sourceoperand_manager は ,要求するレジスタ番号の値や ARF と RRF の情報 ,命令 1 の Renaming 情報などを入力として ,RS に登録すべき情報を生成する回路である . この回路の動作を説明するため ,まず ARF, RRF を用いた Renaming 回路の説明を行う .

Table2 計算状況

ARF.Busy	RRF.Valid	計算状況	RS 登録値
0	*	計算完了	ARF.Data
1	1	計算終了	RRF.Data
1	0	計算中	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] rrfddata,
    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 ? rrfddata :
                rrftag;
    assign rdy = src_eq_0 | (~src_eq_dst1 & (~arf_busy | rrf_valid));

endmodule // sourceoperand_manager

```

Fig. 13 Source Operand Manager

ARF, RRF を用いた基本的な Renaming 回路を Fig. 12 に示す．また, ARF.Busy と RRF.Valid から求められる, 該当するレジスタの計算状況及び RS に登録する値を Table2 に示す．

以下簡単のため, 要求するレジスタの番号を regsrc とする．ARF.Busy が 0 の場合は, プロセッサ内に regsrc を Destination とする命令が存在しないことを示しており, ARF に格納された値が regsrc の値となることを示している．

一方, ARF.Busy が 1 の場合, プロセッサ内に regsrc を Destination とする命令が存在することを示している．regsrc の RRF.Valid が 1 の場合, regsrc の値を生成する命令が EX ステージを終了し, RRF に実行結果を保持していることを示している．そのため, RRF の値を regsrc の値とする．RRF.Valid が 0 の場合, regsrc を生成する命令は発行されているが EX ステージを終了しておらず, 値が未生成であることを示している．この際, RS には RRFTag の値を登録しておき, RS は該当命令の終了時に値を取得する．

これらを踏まえて sourceoperand_manager の動作を見るとわかりやすい．ソースコードの一部を Fig. 13 に示す．

出力として, src は RS に登録する値であり, rdy は src の値が RRFTag(rdy=0) か否かを示す値である．src は, 要求レジスタ番号が 0 ならば常に 0 を出力する．それ以外の状況で, この回路が命令 2 の要

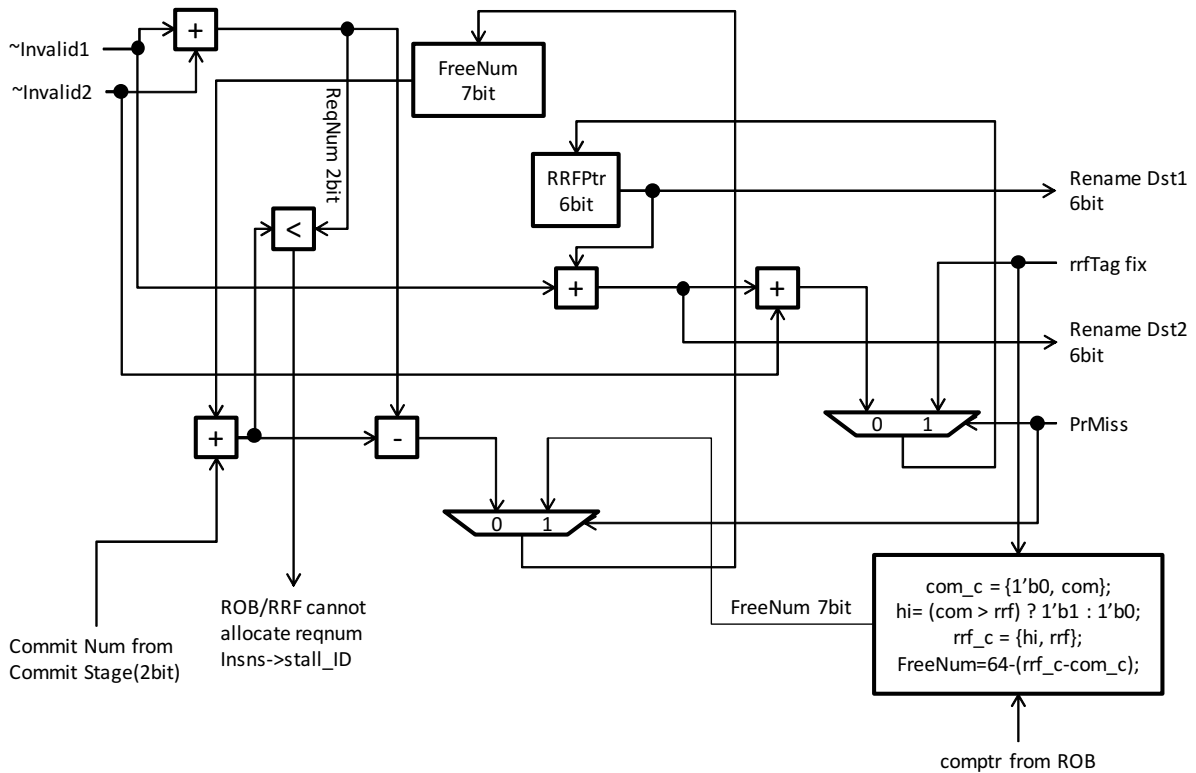


Fig. 14 RRF Free List Manager

求レジスタ番号を処理していて、命令 1 の Destination レジスタの番号と命令 2 の要求レジスタ番号が同じ (src_eq_dst1=1) ならば、命令 2 の該当レジスタは命令 1 の実行結果を用いるため、src の値を命令 1 の RRFTag とする。それ以外の動作は前述の説明のとおりである。

10 rrf_freelistmanager (pp. 239-244)

RRFTag 及び Reorder Buffer の空きエントリの管理を行っている。回路図を Fig. 14 に示す。レジスタとして、空きエントリ数を示す FreeNum と現在のエントリ番号を示す RRFPtr を持つ。

まず、この回路の主な動作について説明する。この回路は主に DP ステージにおいて、命令の Dispatch を行うために RRFTag を割り当てる際に動作する。RRFPtr を要求する命令 (無効でない命令, $\tilde{invalid}$) の数が FreeNum の数を上回る場合、命令の発行ができないため、IF/ID ステージをストールさせる。

次に、割り当てた RRFTag の開放について説明する。命令が Complete した場合、該当する RRFTag の開放を行う。そのために Complete した命令の数を受け取り FreeNum に加算する。RRFPtr は変更しない。また、分岐予測ミスが発生した場合、投機的な命令はキャンセルされるため、その分の RRFTag の開放が必要である。そのために、分岐予測ミスの発生した分岐命令の RRFTag に 1 を加えた値 (rrfTag fix) を受け取り、RRFPtr を rrfTag fix に置き換える。また、受け取った rrfTag fix を用いて FreeNum を復元する。

```

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)

DP ステージの RS 登録前に行うフォワーディングと、RS 内でオペランドが揃うのを待つ回路に用いられる。sourcemanager の Verilog 記述を Fig. 15 に示す。sourceoperand.manager で生成した src と rdy を sourcemanager の opr, opr_rdy として入力する。さらに 5 つの実行ユニット (ALU1/2, LDST, Branch, Mul) の出力である、実行結果を exrslt, 実行している命令の RRF Tag を exdst, 実行結果が無効なものであることを示す kill_spec を入力とする。(kill_spec & (exdst == opr)) となった場合はフォワーディングが可能であるということを踏まえてソースコードを読むとわかりやすい。

RS のソースオペランドとして登録する src は、フォワーディングの必要が無い場合 (opr_rdy==1) とフォ

ワーディングができない場合に値をそのまま (opr) とする．フォワーディングが可能である場合はフォワーディングを行い，resolved を 1 とする．

12 imm_gen, brimm_gen

RISC-V の命令には即値を指定するものがいくつか存在する．addi 命令などの算術論理演算命令や Load/Store 命令，条件分岐命令，ジャンプ命令である．これら命令で用いる即値を，decoder で生成した即値フォーマットの種別にしたがって符号付きで生成する回路が imm_gen, brimm_gen である．ジャンプ命令や条件分岐命令など，PC の更新に関わる即値生成は brimm_gen で，その他の即値は imm_gen で生成を行っている．

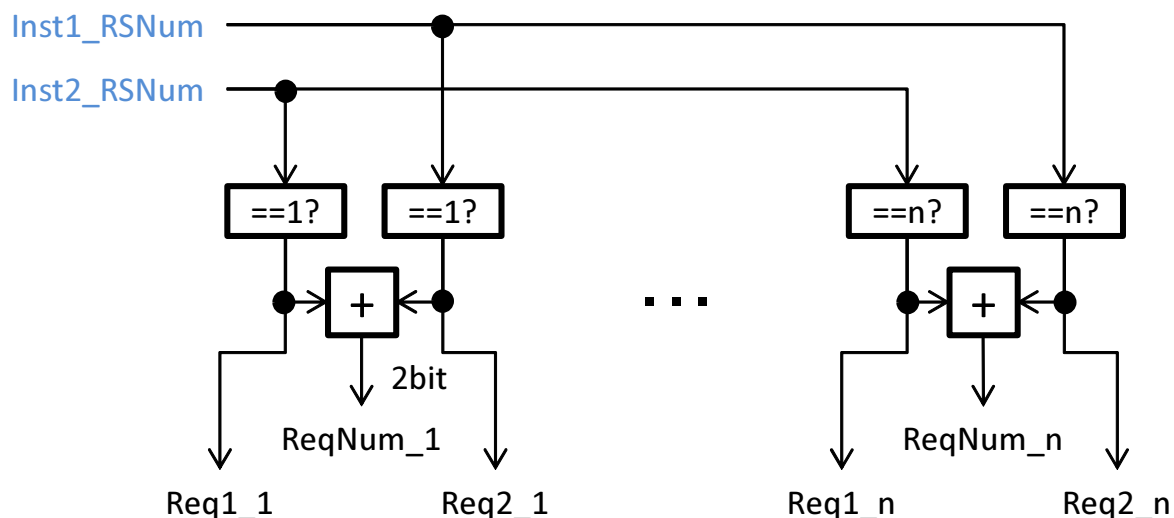


Fig. 16 RS Request Generator

13 rs_requestgenerator (pp. 254-259)

RS は命令を登録する際、命令 1,2 の種々のデータと、2 つの命令が自身の RS に登録すべきかどうかを判定する Write Enable を受け取る。Write Enable は、RS Request Generator が `Req1,2` として RS に出力する。RS Request Generator の図を Fig. 16 に示す。この回路は、decoder で計算した登録する RS の種類を示す `RSNum` を用いて、各 RS への命令 1,2 の Write Enable である `Req1,2` を生成する。

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

RS(Reservation Station) は、命令を登録してオペランドが揃うまで待機する回路であり、演算器ごとに RS を用意する。RS は命令を登録し、オペランドが揃った時に命令を発行するための回路であるが、RS 自身は、各エントリが実行結果のブロードキャストを監視し、必要な実行結果をオペランドとして取得する機能を持ったメモリであり、命令の登録、発行などは Allocate Unit や Issue Unit などが行う。

RS の 1 エントリ (`rs_*_ent`) の回路図を Fig. 17 に示す。橙色で示した部分がデータを保持するレジスタに当たる。1 エントリは `src_manager` を用いて必要な実行結果を取得する。以下、各レジスタの説明をする。

- `opr1`, `opr2`: レジスタの第 1,2 オペランド, `rs1`, `rs2` に関するデータ。後述する `valid` によって入っている値が異なる。
 - `valid = 1`: 該当するレジスタの値を取得済であるか、そもそもレジスタを使用しない命令であることを示す。いずれの場合も該当する `opr` は揃った状態であることを示す。
 - `valid = 0`: 該当するレジスタの値が未取得の状態であることを示す。この際、`opr` には取得すべき命令の `RRFTag` が格納されている。
- `valid1`, `valid2`: `opr1`, `opr2` のデータが、命令の発行が可能な状態であることを示す。`valid1`, `2` の両方

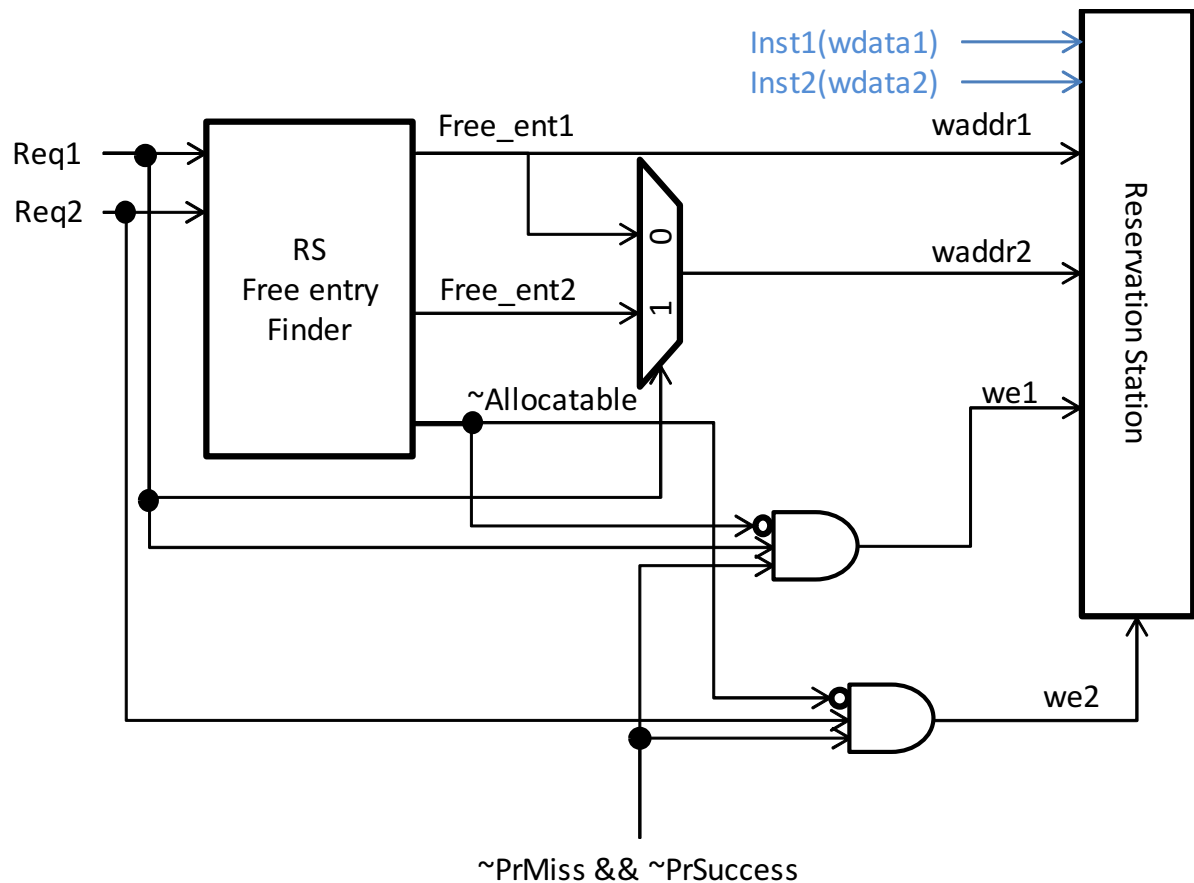


Fig. 18 Allocate Unit

Table3 Allocate Pattarn

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

のメモリとみなして説明する．waddr1, 2 は書き込みアドレス，wdata1, 2 は書き込みデータ，we1, 2 はライトイネーブル信号である．we1, 2 が 1 となることで命令 1, 2 がそれぞれ RS に登録される仕組みである．

Allocate Unit は RS Free Entry Finder を用いて，RS から空いているエントリの番号を最大 2 つまで取得し (Free_ent1, 2)，waddr1, 2 とする．Free_ent の割り当て方を Table3 に示す．waddr2 を見ると Req1=0 の時は Free_ent1，Req1=1 の時は Free_ent2 とすれば良いことがわかる．waddr2 のデータセクタはこれを実現するためのものである．また，we1, 2 はすべての命令が登録できる場合に 1 とするため，書き込み要求があり (Req1, 2=1)，全 Allocate Unit の allocatable 値が 1 となり，RRFTag に空きが存在し，分岐予測処理が起こらない場合に we を 1 とする．

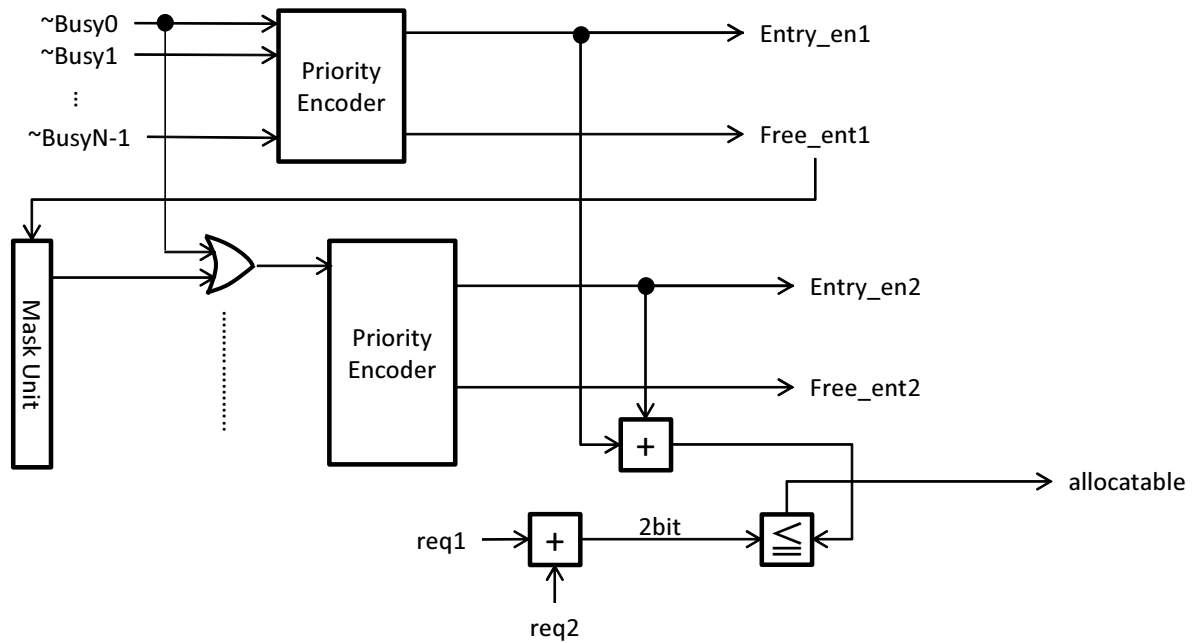


Fig. 19 RS Free Entry Finder

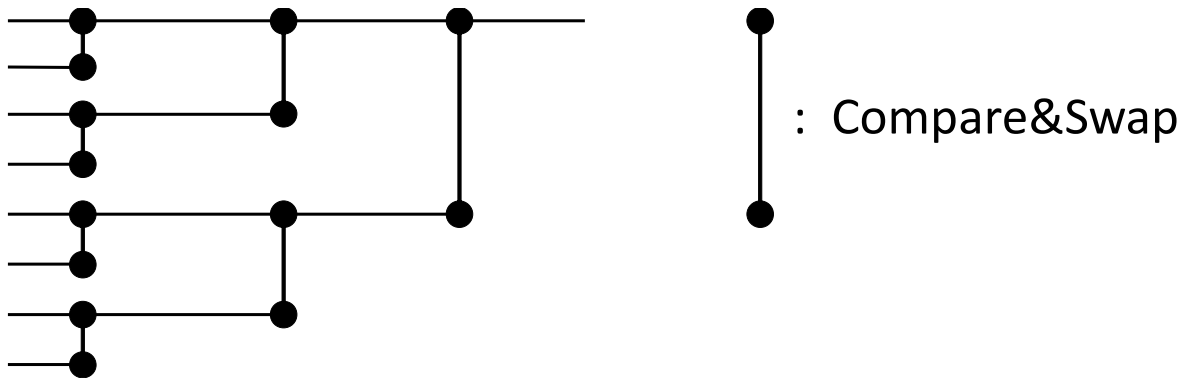


Fig. 20 Minimum Selection(Oldest Finder)

次に RS Free Entry Finder について説明する．RS の Busy のベクターの否定 ($\sim\text{Busy0}$ - $\sim\text{BusyN-1}$) を入力として，RS から空いているエントリの番号を最大 2 つまで取得 (Free_ent1, 2) する回路である．これは 2 段の Priority Encoder を用いて実現される．まず 1 段目の Priority Encoder で Busy が 1 となっているエントリを見つけ，その番号を出力する．次に，再度同じエントリを選択しないために，Mask Unit を用いて 1 段目で選んだエントリの Busy を 1 としてから 2 段目の Priority Encoder に入力する．Priority Encoder では，必ず空きエントリが得られるとは限らないため，得られたエントリが有効であることを示す Entry_en1, 2 を出力する．Entry_en の合計と Req の合計を比較し，Req 分の空きエントリが用意できる場合は allocatable を 1 とする．

Issue Unit では，RS 内のオペランドが揃った命令を 1 つ選択して命令を発行する．発行した命令は RS が

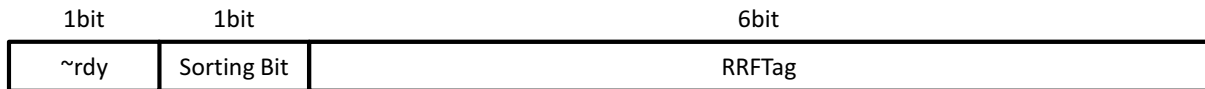


Fig. 21 Sort Entry

ら消える．発行する命令を選択するアルゴリズムとして，発行可能な命令の中で最も古く登録されたものを選ぶ，Oldest First を採用する．それを実現するための最小値選択回路を Fig. 20，また，最小値選択回路の 1 エントリを Fig. 21 に示す．

Issue する命令を選択する上で最も優先されるべき事項は，命令のオペランドが揃っている (rdy=1) ことである．そのため，エントリの最上位ビットを rdy の否定として，オペランドが揃っている命令が最も小さくなるようにする．次に Oldest First で命令を選択するために，命令を古いものから選ぶ必要がある．RRFTag は 0 から順に割り当てられていくため，基本的には RRFTag が小さいほど古い命令であると言える．しかし，RRFTag がオーバーフローして 0 に戻ってきた際に不都合が起こってしまう．

これを解消するために Sorting bit を採用する．Sorting bit は，RS に命令を登録する際には 1 としておくが，RRFTag のオーバーフローが起きた際に RS 内のすべての命令の Sorting bit を 0 とする．こうすることで，新しく登録される RRFTag が 0 の命令よりも，以前から登録されていた命令 (例えば RRFTag=63) のほうが古い命令であることができる．

この動作を具体的な例を挙げて説明する．RRFTag が 63 の命令を A，RRFTag が 0 の命令を B とし，A よりも B のほうが新しい命令であり，両命令のオペランドが揃っているとすると，RRFTag がオーバーフローを起こし，命令 A の Sorting bit が 0 になっていることに注意すると，最小値選択回路で用いるエントリの数値は，命令 A が 63(1'b0, 1'b0, 6'h3f) であるのに対し，命令 B は 64(1'b0, 1'b1, 6'h0) となり，Fig. 20 で命令 A を正しく選択できることがわかる．

ただし，RIDECORE の仕様では RRFTag を 1 サイクルに最大 2 つ割り当てるため，RRFTag 番号 63 と 0 の命令を割り当てる際に，RRFTag=0 の Sorting bit も 0 となってしまう，厳密には Oldest First ではないが，限定的な問題である．

15.2 インオーダー発行

alloc.issue_ino は，RS を FIFO として用いるための回路であり，RS への命令の Allocate 及び Issue を In Order に行う．回路図を Fig. 22 に示す．回路内に持つレジスタは，次に命令の割り当てを行うエントリを示す AllocPtr のみであり，その他はすべて組み合わせ回路で実装されている．alloc.issue_ino は RS の Busy Vector を入力として受け取り，次に Issue するエントリ (IssuePtr) を計算する．計算方法を Fig. 23 に示す．図中に示す b0, b1, e0, e1 は後述する search_begin, search_end で計算される値であり，b0/1 は 0/1 が最初に出てくるエントリの番号，e0/1 は 0/1 が最後に出てくるエントリの番号を示している．

これらを用いて IssuePtr を導出するが，導出には 3 パターンが存在する．まず図の 1 番左のパターンであるが，エントリがすべて 1 の場合，AllocPtr と同じ値が IssuePtr であると言える．次に中央のパターンであるが，エントリの 1 がオーバーフローして途中で切れている場合 ((b1 == 0) && (e1 == ENTRYNUM-1))，空きエントリである 0 は切れていないため，最後の空きエントリ番号は e0 であることがわかる．そのため e0 の次が IssuePtr になると言える．最後に 1 番左のパターンであるが，1 が途中で切れていない場合は単純に

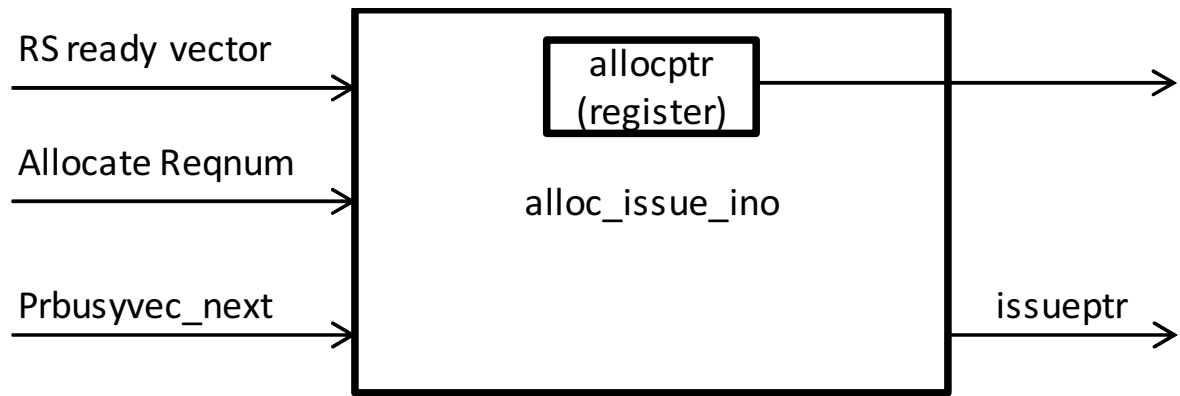


Fig. 22 Allocate and Issue In Order

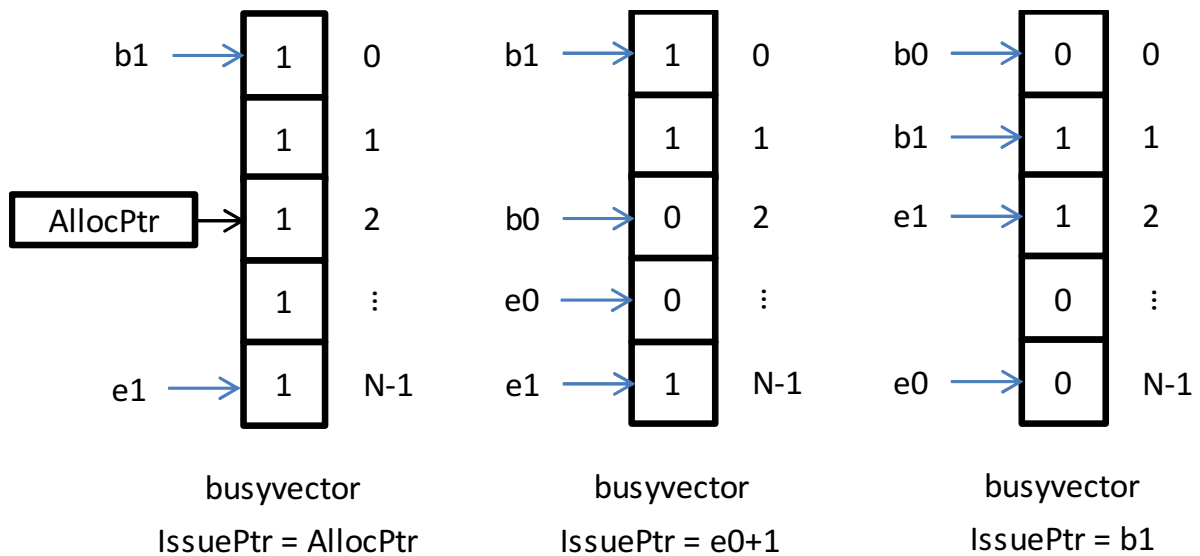


Fig. 23 IssuePtr Calculus

b1 を IssuePtr とすればよい。

また、AllocPtr は基本的に Allocate した命令の数だけ increment すればよいが、分岐予測ミスが発生した場合には RS 中の命令が無効化されるため、無効化後の AllocPtr を再計算しなければならない。そこで alloc_issue_ino は無効化後の RS の Busy Vector である、Prbusyvector_next を受け取り、AllocPtr の再計算を行う。計算方法は IssuePtr とほとんど同じであるため省略する。

15.3 search_begin, search_end

search_begin, search_end は下位 bit/上位 bit 優先の Priority Encoder である。alloc_issue_ino や storebuf などの In Order 実行を行う回路で用いられる。

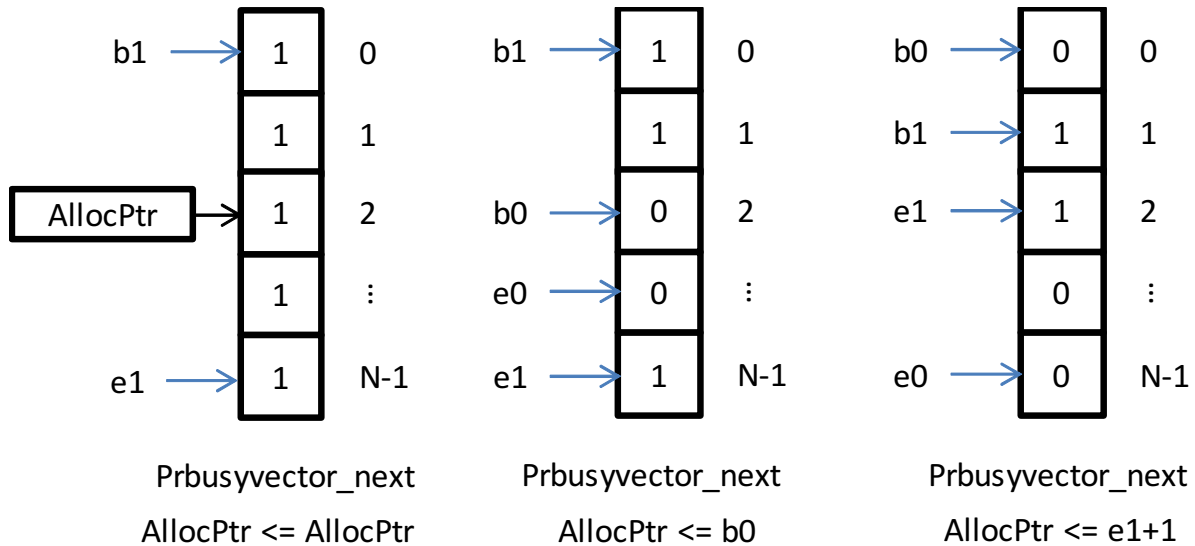


Fig. 24 AllocPtr Re-calculus

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

実行ユニットはALUが2つ、乗算器が1つ、Load/Storeユニットが1つ、分岐ユニットが1つである。ALUの回路図をFig. 25, Load/Storeユニットの回路図をFig. 26, 分岐ユニットの回路図をFig. 27に示す。また、図中のKill Genの回路をFig. 28に示す。この回路は分岐予測に失敗した際に、現在演算中の計算が破棄すべき命令かを判定する。このような回路になる理由は、Miss Prediction Fix Tableのセクションを参照されたし。なお、乗算器については $c=a*b$ のように記述しており、1サイクルで演算を行うことができる。そのため回路はALUとほとんど同じとなる。

いずれの回路も、Reorder Bufferへの終了通知(ROB WE)及びそのエントリ(RRF Write Addr), RRFに書き込む実行結果(RRF Write Data)を最終的な出力としている。これらの他に追加の操作を行うLoad/Storeユニット及び分岐ユニットについて説明する。

まず、Load/Storeユニットについて説明する。Load/Storeユニットは内部が2段のパイプラインとなっている。RSから受け取った命令がLoad命令の場合、まず1段目でDMEMとStore Bufferにアクセスする。Store Bufferのデータは1段目で受け取る。Store Bufferについては該当するセクションを参照されたし。2段目においてDMEMからデータを受け取る。未Storeのデータがある場合はStore Bufferから受け取ったデータ、ない場合はDMEMから受け取ったデータを実行結果としてRRFに書き込む。

RSから受け取った命令がStore命令の場合、1段目でStore Bufferにアドレス、データを書き込み、2段目でReorder Bufferに終了通知を行う。COMステージを経てStore命令が完了した際に、該当する命令はStore Bufferからメモリに書き込みを行うことが許可され、Load命令が実行されていない間にStore BufferがDMEMに書き込みを行う。

次に、分岐ユニットについて説明する。分岐ユニットは条件に従って変わる分岐先アドレスを計算し、IFステージで予測した分岐先との比較を行う。分岐予測が成功した場合、Miss Prediction Fix Tableの更新及び

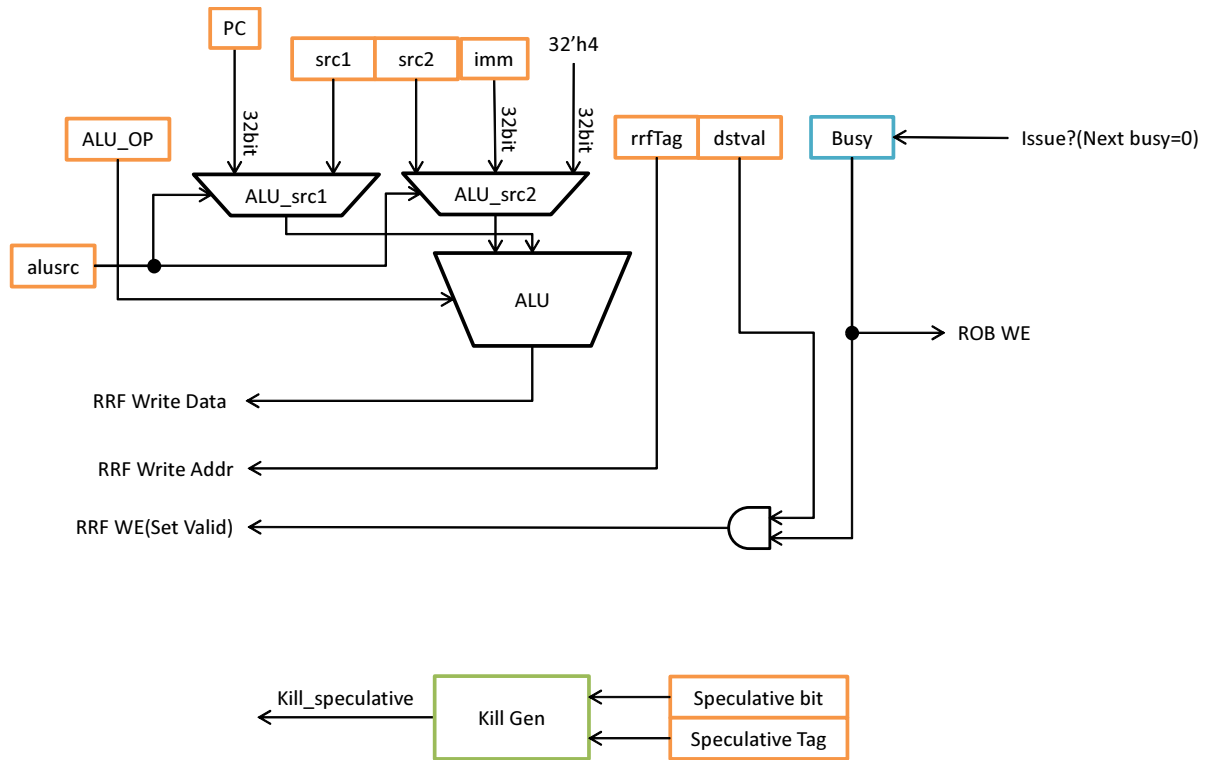


Fig. 25 ALU

投機実行のための情報の破棄を行う．分岐予測が失敗した場合，プロセッサを分岐予測を行った直前の状態まで復元する．いずれの場合も情報を復元するためにストールが発生する．なお，COM ステージで分岐予測器に情報を通知するため，Reorder Buffer には終了通知に加えて分岐先アドレスと分岐条件の情報を書き込む．

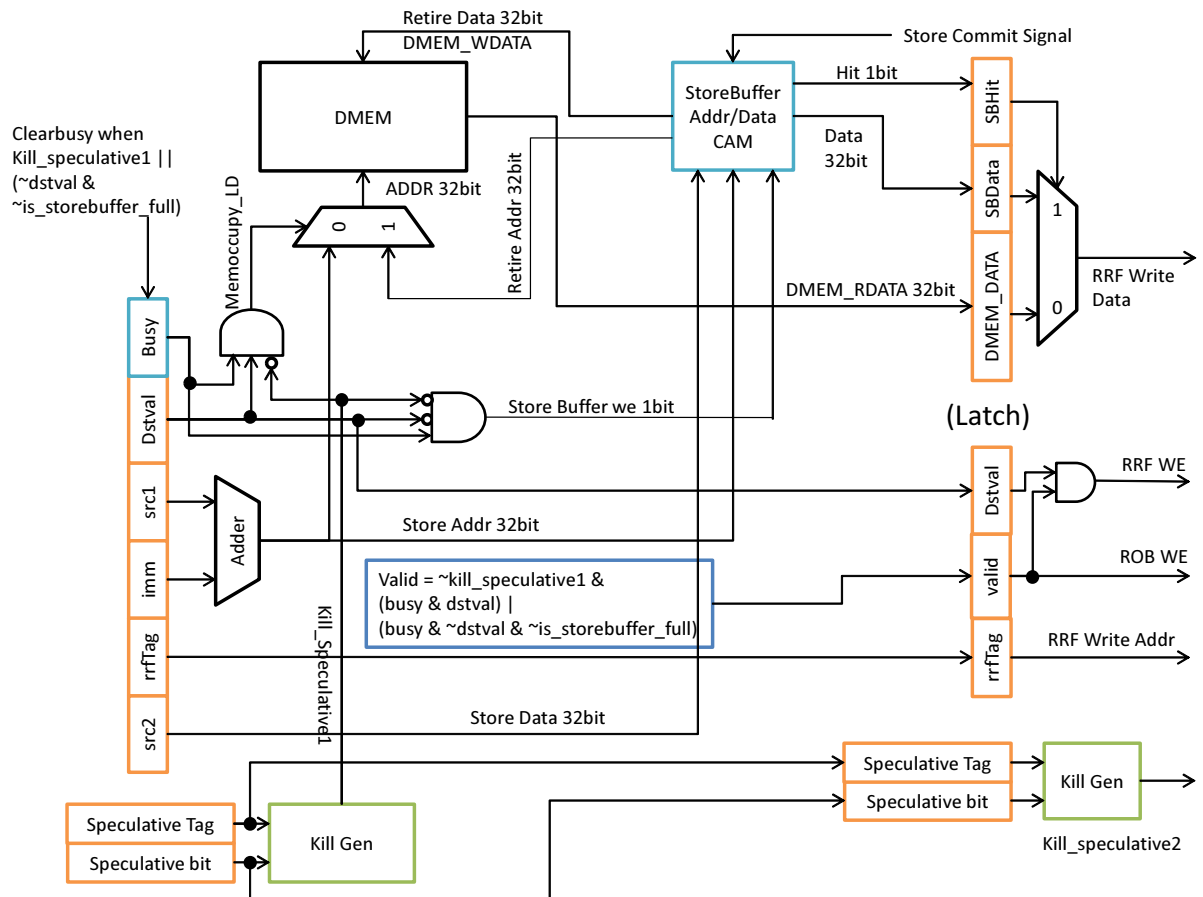


Fig. 26 Load/Store Unit

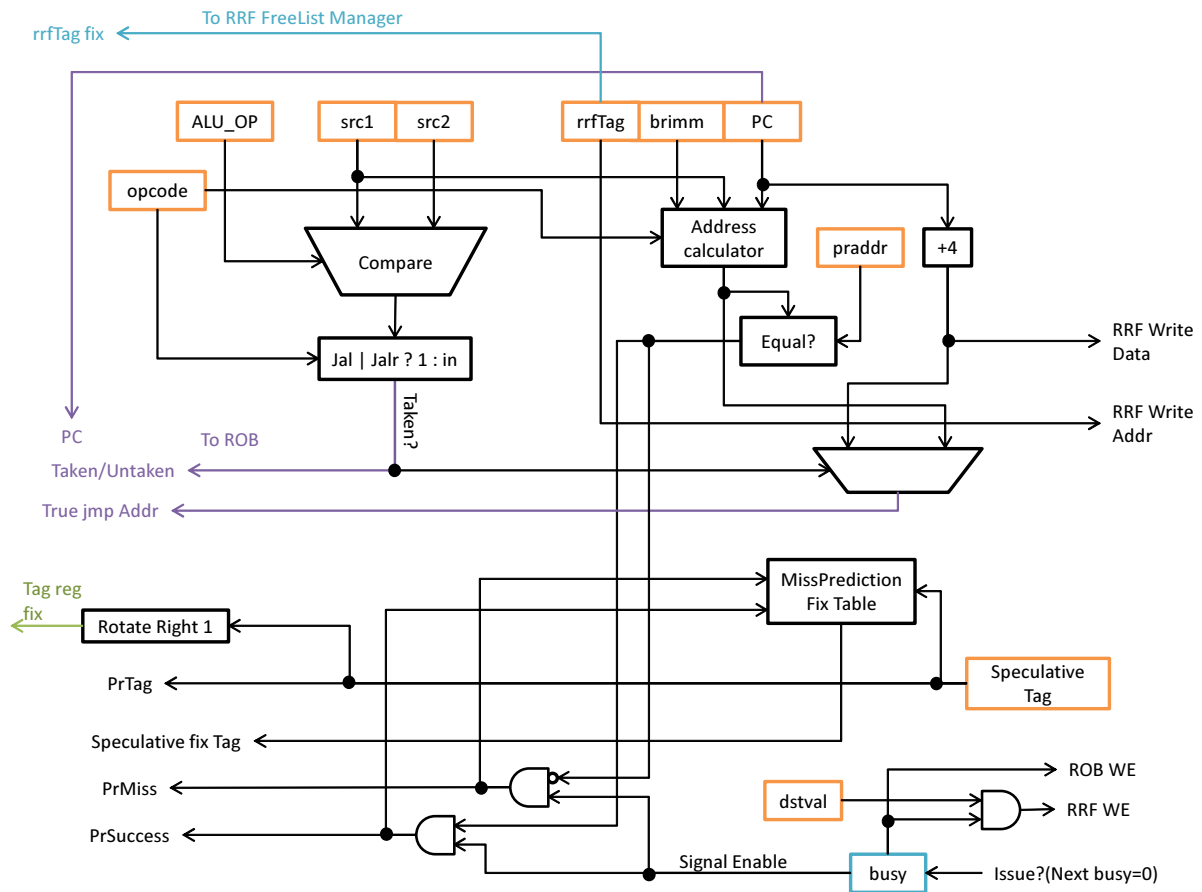


Fig. 27 Branch Unit

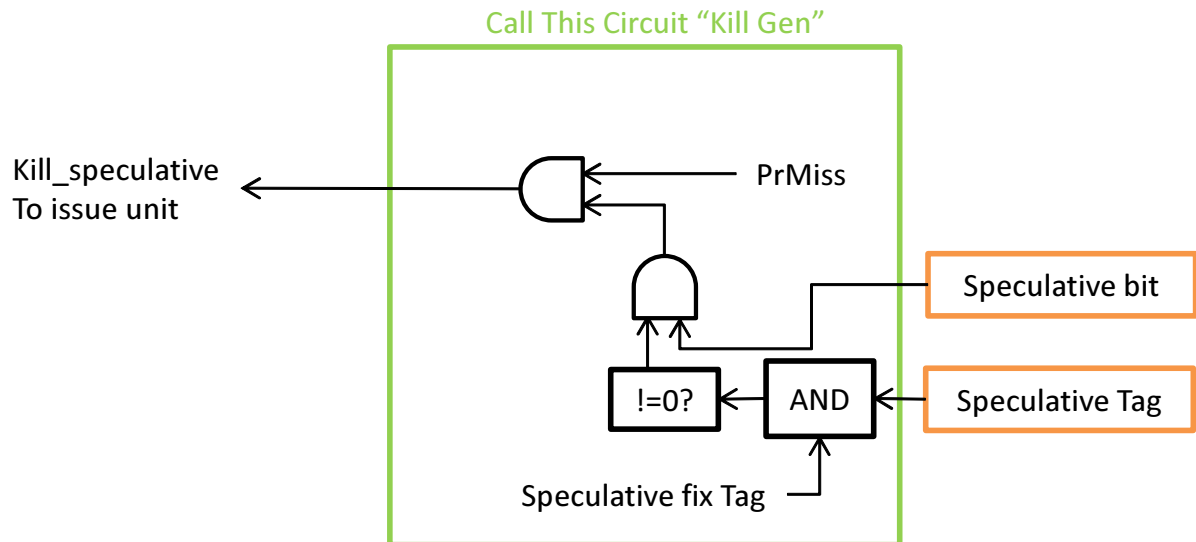


Fig. 28 Kill Gen

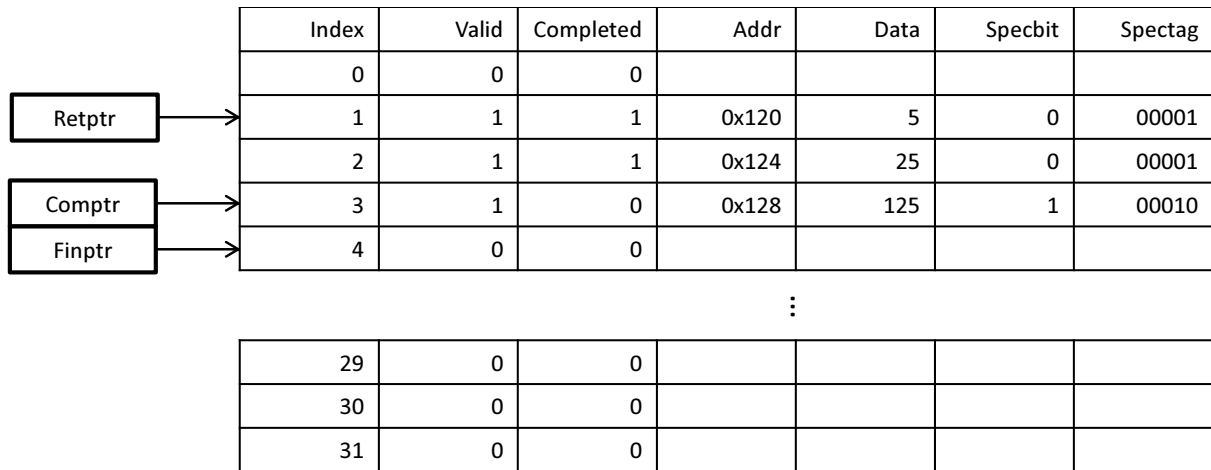


Fig. 29 Store Buffer

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

Store Buffer は、命令は終了したがメモリに書き込んでいない未 Store のデータとそのアドレスのペアを持ち、アドレスを与えると連想検索を行い、最新の未 Store データを返す。最新の未 Store データを持つエントリは、アドレスが一致したことを示す hit Vector を finptr の示すエントリが一番下に来るように Rotate してから search_end を用いることで容易に選択可能となる。Store Buffer のエントリやレジスタを Fig. 29 に示す。finptr は実行が終了した命令、comptr は Reorder Buffer から出て Complete した命令、retptr はメモリへの書き込みが終了した命令をそれぞれ示している。comptr 及び retptr はそれぞれ Complete した命令、Retire 命令の分だけ加算するのみで良いが、finptr は分岐予測ミスが発生した際に再計算を行う必要があり、alloc_issue_ino と同じ要領で再計算している。

以下、各エントリについて説明する。

- valid: 該当する StoreBuffer のエントリが有効であることを示す。
- completed: 該当する StoreBuffer のエントリが Complete 済みであることを示す。valid と completed が両方共 1 の場合、命令を retire することが可能となる。
- addr, data: メモリに書き込むデータとアドレスである。
- spectag, specbit: Speculative Tag と、該当する Store 命令が投機的であることを示す。分岐予測ミスが発生した際、命令を破棄するために必要である。

18 miss_prediction_fix_table (pp. 228-231)

Miss Prediction Fix Table には各 Speculative Tag の状態を示す 2 つの値、Value と Valid が格納されている。Valid は、該当する Speculative Tag のついた命令が投機的であることを示す。Value は該当する Speculative Tag の他の Speculative Tag との依存関係を示している。

Miss Prediction Fix Table について、Fig. 30 を用いて具体的に説明する。図は、分岐命令を 2 つデコー

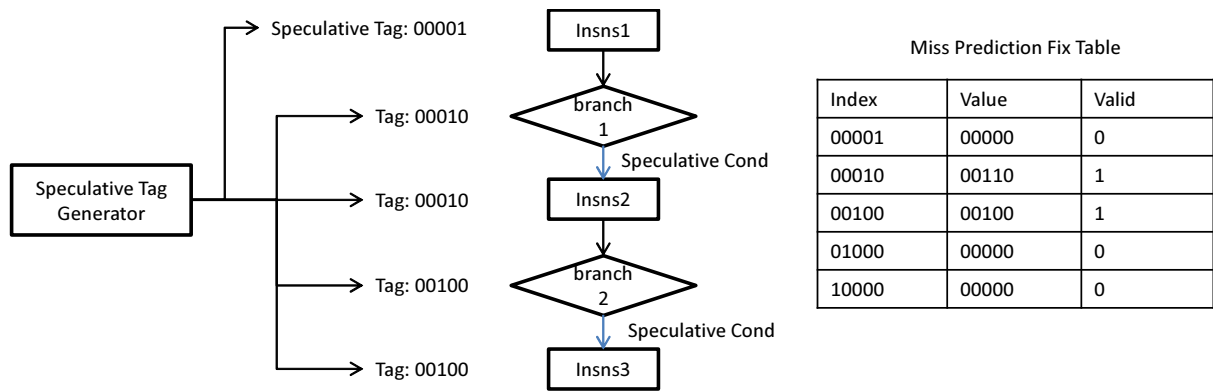


Fig. 30 Speculative Exection

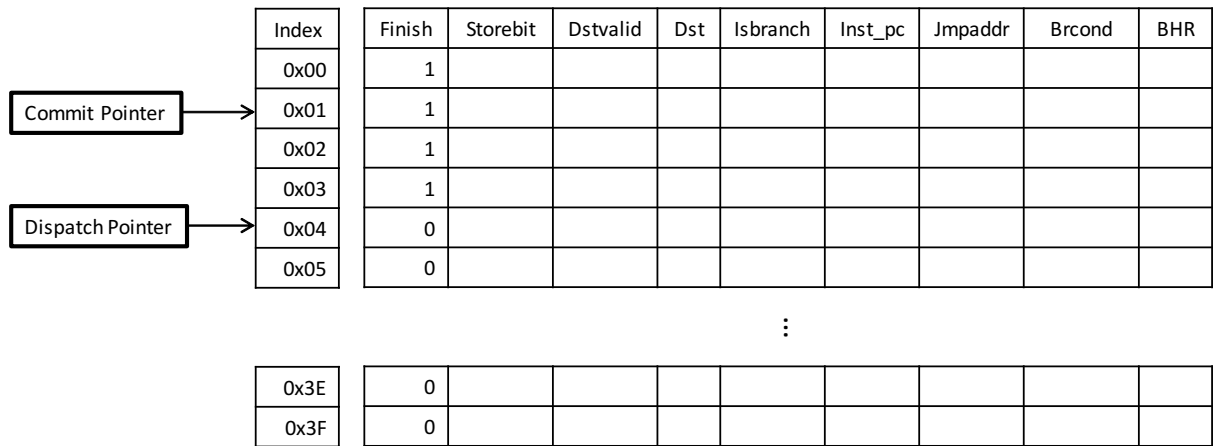


Fig. 31 Reorder Buffer

ドしたあとの各命令群の Speculative Tag と Miss Prediction Fix Table を示している。Speculative Tag は Speculative Tag Generator が、00001 から順に左ローテートしながら割り当てていく。図において、branch1 で分岐予測ミスが発生した場合、該当する Speculative Tag: 00010 の Value: 00110 を読み出す。このとき、分岐予測ミスによって無効化される命令群は、Value: 00110 と AND 演算を行った結果が 00000 とならない Speculative Tag を持つ命令群となる。00110 の場合は、Speculative Tag が 00100 または 00010 となる命令であり、branch1 以降の命令のみが無効化されることを示している。

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

Reorder Buffer は、Out of Order に実行した命令を In Order に完了するためのバッファである。Reorder Buffer の持つエントリやレジスタを Fig. 31 に示す。Reorder Buffer はレジスタとして、次に complete する命令を指す comptr をもち、次に Dispatch するエントリを指す dispatchptr は rrf_freelistmanager の RRFPtr を用いる。完了する命令数は最大で 2 つである。分岐予測器や Store Buffer の Write Port 数を減らすため、分岐命令や Store 命令は 1 つずつ完了する。

完了の際、RRF に書き込まれていた実行結果を ARF.Data に移動する。また、Renaming Table の情報を更新するが、該当するエントリの RRF Tag が、完了した命令の RRF Tag と一致している時のみ Busy をクリアできることに注意する。

以下、各エントリについて説明する。

- finish: 該当する命令が終了していることを示す。Dispatch 時にクリアされ、実行終了時にセットされる。
- storebit: 該当する命令が Store 命令であることを示す。Store 命令が完了した場合は、Store Buffer に完了通知を行う。
- dstvalid: デスティネーションレジスタが有効であり、ARF への書き込みが必要であることを示す。
- dst: デスティネーションレジスタの番号。
- isbranch: 該当する命令が分岐命令であることを示す。分岐命令が完了した場合は、分岐予測器に該当データを書き込む。
- inst_pc, jmpaddr, brcond, bhr: 分岐予測器に書き込むデータ群。左から順に、命令アドレス、分岐先アドレス、条件 (Taken/Untaken)、命令発行時点での BHR(PHT への書き込みアドレス算出に用いる)

References

- [1] John Paul Shen, Mikko H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 2013.
- [2] 奥村晴彦：C 言語による最新アルゴリズム辞典，事典技術評論社，1991
- [3] McFarling, Scott: Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory, (1993)
- [4] RISC-V, <http://riscv.org/> (2016/01/20)