

Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L16: RISC-V: The Fife pipelined CPU



Reminders

Please git clone: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design
(git pull for latest version). Repository structure:

```
./Book_BLang_RISCV.pdf
  Slides/
    Slides_01_Intro.pdf
    Slides_02_ISA.pdf
    ...
  Exercises/
    Ex-03-A-Hello-World/
    Ex-03-B-Top-and-DUT/
    ...
  Code/
    src_Top/
    src_Drum/
    src_Fife/
    src_Common/
    ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the Exercises/ directory.

To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- bsc compiler (see <https://github.com/B-Lang-org/bsc>)
- Verilator compiler (see <https://www.verilator.org/>)

Chapter Roadmap



Flow of information between stages in Drum and Fife

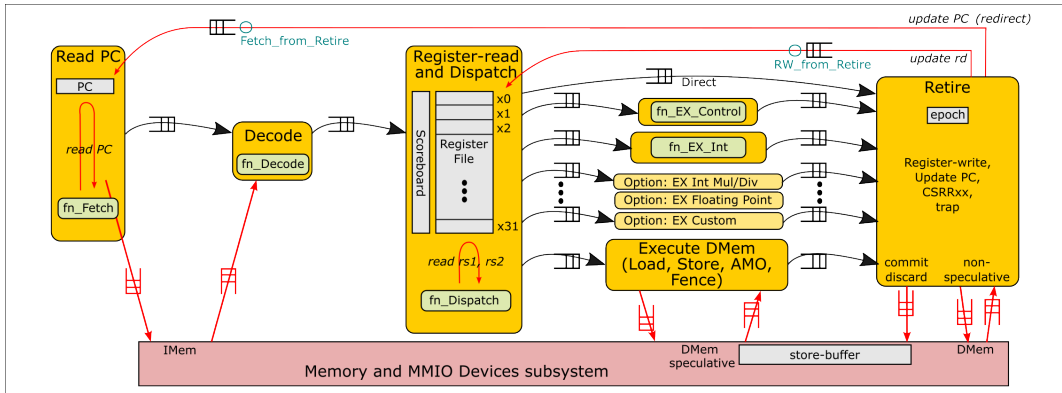


Table of Contents

- 1 Reminders
- 2 Module Interface (same for Drum and Fife)
- 3 Top-level mkCPU module
- 4 Fetch stage
- 5 Decode stage
- 6 Register-Read and Dispatch stage (and Register-Write)
- 7 Execute Control stage
- 8 Execute Int stage
- 9 Retire stage
- 10 Final Comments

Fife CPU: Module Interface (same for Drum and Fife)

Module Interface

```
src_Common/CPU_IFC.bsv: line 27 ...  
interface CPU_IFC;  
  method Action init (Initial_Params initial_params);  
  
  // IMem  
  interface FIFOF_O #(Mem_Req) fo_IMem_req;  
  interface FIFOF_I #(Mem_Rsp) fi_IMem_rsp;  
  
  // DMem, speculative  
  interface FIFOF_O #(Mem_Req) fo_DMem_S_req;  
  interface FIFOF_I #(Mem_Rsp) fi_DMem_S_rsp;  
  interface FIFOF_O #(Retire_to_DMem_Commit) fo_DMem_S_commit;  
  
  // DMem, non-speculative  
  interface FIFOF_O #(Mem_Req) fo_DMem_req;  
  interface FIFOF_I #(Mem_Rsp) fi_DMem_rsp;  
  
  // Set TIME  
  (* always_ready, always_enabled *)  
  method Action set_TIME (Bit #(64) t);  
endinterface
```

The speculative DMem sub-interfaces:

fo_DMem_S_req fi_DMem_S_rsp fo_DMem_S_commit

were unused in Drum, and are used in Fife.

Fife CPU: Top-level mkCPU module

Please simultaneously view file: `Code/src_Fife/CPU.bsv`

Overall Fife CPU module structure

From src_Drum/CPU.bsv

```
(* synthesize *)
module mkCPU (CPU_IFC);
  // *****
  // STATE (sub-modules for pipeline stages)
  ...

  // *****
  // STATE (sub-modules for inter-stage connections)

  ...
  // *****
  // INTERFACE
  ...
endmodule
```

Details in slides that follow

Fife's mkCPU module is actually simpler than Drum's mkCPU because the major functionality is now in the separate stage sub-modules (Drum did not have stage sub-modules).

Fife module state: sub-modules for pipeline stages

```
From src.Fife/CPU.bsv  
  
// Pipeline stages  
Fetch_IFC      stage_F      <- mkFetch;  
Decode_IFC     stage_D      <- mkDecode;  
RR_RW_IFC      stage_RR_RW  <- mkRR_RW;  
EX_Control_IFC stage_EX_Control <- mkEX_Control; // Branch, JAL, JALR  
EX_Int_IFC     stage_EX_Int  <- mkEX_Int;   // Integer ops  
Retire_IFC     stage_Retire  <- mkRetire;
```

Fife instantiates a sub-module for each of the pipeline stages

Fife module state: inter-stage forward-flow connections (sub-modules)

```
_____ From src_Fife/CPU.bsv _____  
  
// Forward flow connections  
  
// Fetch->Decode->RR-Dispatch, and direct path RR-Dispatch->Retire  
mkConnection (stage_F.fo_Fetch_to_Decode,  stage_D.fi_Fetch_to_Decode);  
mkConnection (stage_D.fo_Decode_to_RR,      stage_RR_RW.fi_Decode_to_RR);  
mkConnection (stage_RR_RW.fo_RR_to_Retire,  stage_Retire.fi_RR_to_Retire);  
  
// RR-Dispatch->various EX  
mkConnection (stage_RR_RW.fo_RR_to_EX_Control,  
              stage_EX_Control.fi_RR_to_EX_Control);  
mkConnection (stage_RR_RW.fo_RR_to_EX_Int,  
              stage_EX_Int.fi_RR_to_EX_Int);  
  
// Various EX->Retire  
mkConnection (stage_EX_Control.fo_EX_Control_to_Retire,  
              stage_Retire.fi_EX_Control_to_Retire);  
mkConnection (stage_EX_Int.fo_EX_Int_to_Retire,  
              stage_Retire.fi_EX_Int_to_Retire);
```

Each connection is a FIFO

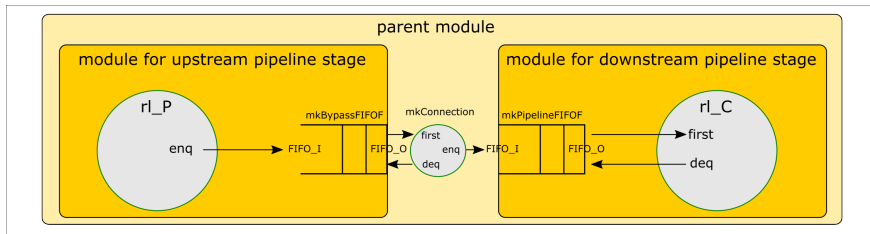
Fife module state: inter-stage backward-flow connections (sub-modules)

```
_____ From src_Fife/CPU.bsv _____  
  
// Backward flow connections  
  
// Fetch<-Retire (redirection)  
mkConnection (stage_Retire.fo_Fetch_from_Retire, stage_F.fi_Fetch_from_Retire);  
// RR-Dispatch<-Retire (register writeback)  
mkConnection (stage_Retire.fo_RW_from_Retire, stage_RR_RW.fi_RW_from_Retire);
```

Each connection is a FIFO

FIFO connections between separately compiled Fife sub-modules

Each inter-stage FIFO `mkConnection` has this form:



- Data can traverse from producer to consumer in 1 tick, as desired, despite there being two FIFOs, because of the semantics of `mkBypassFIFO` and `mkPipelineFIFO` (Chapter 17).
- The structure allows the producer and consumer to be compiled independently by *bsc*, with no “rule-scheduling” constraints leaking across stage boundaries.
- There are no combinational paths crossing the stage boundary (through the two FIFOs).
- The structure allows us to reason about correctness of each stage completely independently of other stages.

Fife module interface definitions

From src_Fife/CPU.bsv

```
method Action init (Initial_Params initial_params);
  rg_flog <= initial_params.flog;

  stage_F.init (initial_params);
  stage_D.init (initial_params);
  stage_RR_RW.init (initial_params);
  stage_EX_Control.init (initial_params);
  stage_EX_Int.init (initial_params);
  stage_Retire.init (initial_params);
endmethod

// IMem
interface fo_IMem_req = stage_F.fo_Fetch_to_IMem;
interface fi_IMem_rsp = stage_D.fi_IMem_to_Decode;

// DMem, speculative
interface fo_DMem_S_req    = stage_RR_RW.fo_DMem_S_req;
interface fi_DMem_S_rsp    = stage_Retire.fi_DMem_S_rsp;
interface fo_DMem_S_commit = stage_Retire.fo_DMem_S_commit;

// DMem, non-speculative
interface fo_DMem_req = stage_Retire.fo_DMem_req;
interface fi_DMem_rsp = stage_Retire.fi_DMem_rsp;

// Set TIME
method Action set_TIME (Bit #(64) t) = stage_Retire.set_TIME (t);
```

The `init` method initializes this module and also each stage sub-module.

The `set_TIME` method is for updating the time CSR.

Fife CPU Fetch stage

Please simultaneously view file: `Code/src_Fife/S1_Fetch.bsv`

Fetch stage: interface

```
_____ src_Fife/S1_Fetch.bsv: line 33 ... _____  
interface Fetch_IFC;  
  method Action init (Initial_Params initial_params);  
  
  // Forward out  
  interface FIFOF_0 #(Fetch_to_Decode)  fo_Fetch_to_Decode;  
  interface FIFOF_0 #(Mem_Req)          fo_Fetch_to_IMem;  
  
  // Backward in  
  interface FIFOF_I #(Fetch_from_Retire) fi_Fetch_from_Retire;  
endinterface
```

Interface

The FIFO_I from Retire carries redirection information (update PC on misprediction/exception/interrupt).

Fetch stage: implementation module

For module `mkFetch` let us examine file: `Code/src_Fife/S1_Fetch.bsv`

- Rule `rl_Fetch_req` is the forward-flow “fetching rule”, repeatedly invoking `fn_Fetch` to produce `IMem` requests and information for `Decode`.

The expression “`rg_pc+4`” is, more generally, “`predict(rg_pc)`”.

In future we can substitute new, improved, predictor functions (see book “Section 18.3.6.4 Better next-PC prediction”).

- Rule `rl_Fetch_from_Retire` is the backward-flow. It receives `Fetch_from_Retire` messages from `Retire` whenever `Retire` needs to change the control flow to something other than the predicted PC (due to `BRANCH`, `JAL`, `JALR`, or traps). This rule updates `rg_pc` to the newly specified PC, and also updates `rg_epoch`.
- The expression `(! f_Fetch_from_Retire.notEmpty)` in `rl_Fetch_req`’s condition gives it lower priority compared to `rl_Fetch_from_Retire` when both are enabled. The sooner we perform redirection, the fewer wrong-path instructions will be fetched.

`rg_oiaat` is used for “one instruction at a time” mode (for debugging), and can be ignored for now.

Fife CPU Decode stage

Please simultaneously view file: `Code/src_Fife/S2_Decode.bsv`

Decode stage: interface

```
_____ src_Fife/S2_Decode.bsv: line 33 ... _____  
interface Decode_IFC;  
  method Action init (Initial_Params initial_params);  
  
  // Forward in  
  interface FIFO_I #(Fetch_to_Decode)  fi_Fetch_to_Decode;  
  interface FIFO_I #(Mem_Rsp)          fi_IMem_to_Decode;  
  
  // Forward out  
  interface FIFO_O #(Decode_to_RR)  fo_Decode_to_RR;  
endinterface
```

Interface

Decode stage: implementation module

For module `mkDecode` code, let us examine file: `Code/src_Fife/S2_Decode.bsv`

- FIFO `f_Fetch_to_Decode` is a `mkSizedFIFO(4)` instance, aiming to balance the Fetch-to-Decode direct path length with the Fetch-to-IMem-to-Decode path length (equalize the number of items that can be “in flight” on each path).
- Rule `r1_Decode` is the forward-flow; it repeatedly invokes `fn_Decode` to produce information for the next stage.

Fife CPU Register-Read and Dispatch stage (and Register-Write)

Please simultaneously view file: `Code/src_Fife/S3_RR_RW.bsv`

Register-Read and Dispatch stage (and Register-Write): Interface

```
src_Fife/S3_RR_RW.bsv: line 40 ...  
interface RR_RW_IFC;  
  method Action init (Initial_Params initial_params);  
  
  // Forward in  
  interface FIFOF_I #(Decode_to_RR)  fi_Decode_to_RR;  
  
  // Forward out  
  interface FIFOF_O #(RR_to_Retire)    fo_RR_to_Retire;  
  interface FIFOF_O #(RR_to_EX_Control) fo_RR_to_EX_Control;  
  interface FIFOF_O #(RR_to_EX)        fo_RR_to_EX_Int;  
  interface FIFOF_O #(Mem_Req)          fo_DMem_S_req;  
  
  // Backward in  
  interface FIFOF_I #(RW_from_Retire)  fi_RW_from_Retire;  
endinterface
```

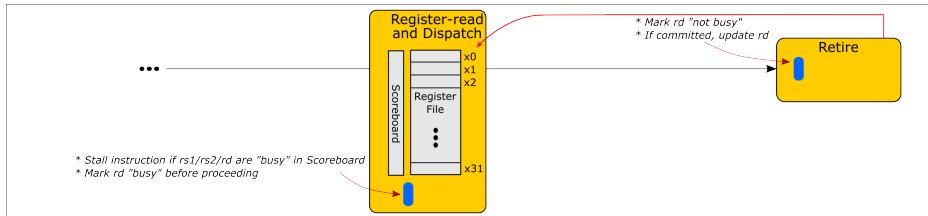
Interface

The four FIFOF_O interfaces feed the four execution paths.

The FIFOF_I from Retire carries information for register-write and release of scoreboard reservations.

Reg-Read and Dispatch (and Reg-Write): GPRs

For module `mkRR_RW` code, let us examine file: `Code/src_Fife/S3_RR_RW.bsv`

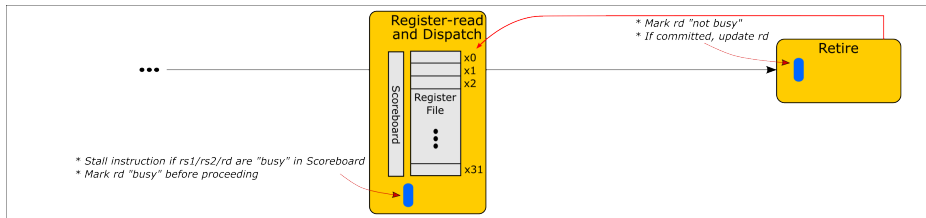


The GPRs (general-purpose registers) are instantiated in this module because we will be reading Rs1 and Rs2 here for each instruction:

```
From src_Fife/CPU.bsv  
GPRs_IFC #(XLEN)  gprs <- mkGPRs_synth;
```

`mkGPRS_synth` is just a version of `mkGPRs` instantiated with a specific `XLEN` width, and with the `(* synthesize *)` attribute so that it becomes a Verilog module.

Reg-Read and Dispatch (and Reg-Write): Scoreboard



The Scoreboard accompanies the GPRs. It contains one bit per GPR, indicating whether it is busy (1) or not-busy (0).

```
From src_Fife/CPU.bsv
typedef Vector #(32, Bit #(1)) Scoreboard;

Reg #(Scoreboard) rg_scoreboard <- mkReg (replicate (0));
```

(Type $\text{Vector}\#(n, t)$ represents a vector of n elements, each of type t .)

Reg-Read and Dispatch (forward flow)

Let us examine `rl_RR_Dispatch` in file: `Code/src/Fife/S3_RR_RW.bsv`

- It stalls (waits) if the instruction has `Rs1`, `Rs2` or `Rd`, and these are busy according to the scoreboard.
- Othersize
 - It reads `Rs1` and `Rs2` registers for the current instruction.
 - It sets the scoreboard for the current instruction's `Rd` to 1, marking it “busy” (if the instruction has an `Rd`);
 - It uses information from `Decode` to dispatch to the four `Execute` pipes. We always (for every instruction) send an execution tag and additional information on the direct pipe. Depending on the instruction, it may also send information into one of the other `Execute` pipes: (`Execute Control`, `Execute Integer`, and `DMem`)

Reg-Write (backward flow)

Let us examine `r1_RW_from_Retire` in file: `Code/src/Fife/S3_RR_RW.bsv`

- We pop the message `x` from the `f_RW_from_Retire` FIFO.
- We perform its specified scoreboard-release for register `Rd`. If the `Rd` value is to be committed, we write it into GPR `[Rd]`.

The two rules `r1_RR_Dispatch` and `r1_RW_from_Retire` run concurrently.

If `r1_RR_Dispatch` was stalled on an instruction whose `Rs1` or `Rs2` was busy, executing `r1_RW_from_Retire` may “un-stall” it, if its `Rd`-write is to the pending register.

The explicit condition on `r1_RR_Dispatch`:

```
(! f_RW_from_Retire.notEmpty)
```

prioritizes the forward rule lower than the backward rule `r1_RW_from_Retire`, so that this “un-stalling” (if it happens) can happen as soon as possible.

Fife CPU Execute Control stage

Please simultaneously view file: `Code/src_Fife/S4_EX_Control.bsv`

This is a very simple, forward-flow-only, one-in, one-out module; it just applies `fn_EX_Control()` (which we have already seen in Drum) to each item as it transits.

Fife CPU Execute Int stage

Please simultaneously view file: `Code/src_Fife/S4_EX_Int.bsv`

This is a very simple, forward-flow-only, one-in, one-out module; it just applies `fn_EX_Int()` (which we have already seen in Drum) to each item as it transits.

Fife CPU Retire stage

Please simultaneously view file: `Code/src_Fife/S5_Retire.bsv`

Retire stage module interface

src_Fife/S5_Retire.bsv: line 33 ...

```
interface Retire_IFC;
  method Action init (Initial_Params initial_params);

  // Forward in
  interface FIFOF_I #(RR_to_Retire)      fi_RR_to_Retire;
  interface FIFOF_I #(EX_Control_to_Retire) fi_EX_Control_to_Retire;
  interface FIFOF_I #(EX_to_Retire)      fi_EX_Int_to_Retire;

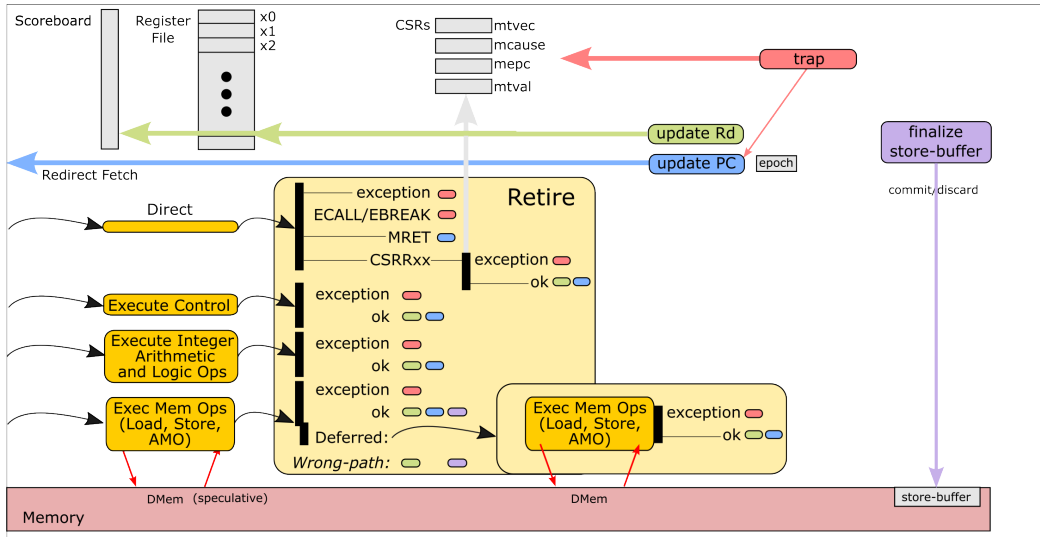
  // DMem, speculative
  interface FIFOF_I #(Mem_Rsp)          fi_DMem_S_rsp;
  interface FIFOF_O #(Retire_to_DMem_Commit) fo_DMem_S_commit;

  // DMem, non-speculative
  interface FIFOF_O #(Mem_Req)  fo_DMem_req;
  interface FIFOF_I #(Mem_Rsp)  fi_DMem_rsp;

  // Backward out
  interface FIFOF_O #(Fetch_from_Retire)  fo_Fetch_from_Retire;
  interface FIFOF_O #(RW_from_Retire)     fo_RW_from_Retire;

  // Set TIME
  (* always_ready, always_enabled *)
  method Action set_TIME (Bit #(64) t);
endinterface
```

Retire stage flows



Retire stage module “mode”

Normally, the Retire stage retires an instruction on every clock (MODE_PIPE).

Except

- For DMem “Deferred” instructions (for MMIO), we send the DMem request, and transition to MODE_DMEM_RSP.
In MODE_DMEM_RSP we collect the DMem response, retire the instruction, and return to MODE_PIPE.
- For exceptions, we transition to MODE_EXCEPTION.
In MODE_EXCEPTION we perform the trap actions, and return to MODE_PIPE.

```
src.Fife/S5.Retire.bsv: line 60 ...  
typedef enum {MODE_PIPE,           // Normal pipeline operation  
              MODE_DMEM_RSP,       // Handle Non-speculative DMem response  
              MODE_EXCEPTION  
} Module_Mode  
deriving (Bits, Eq, FShow);
```


Retire stage module state: highlights

From Code/src/Fife/S5_Retire.bsv

```
module mkRetire (Retire_IFC);
  ...
  // Control-and-Status Registers (CSRs)
  CSRs_IFC csrs <- mkCSRs;

  // For managing speculation, redirection, traps, etc.
  Reg #(Epoch) rg_epoch <- mkReg (0);
  ...
  ... FIFOs for incoming and outgoing flows ...
  ...
  Reg #(Module_Mode) rg_mode <- mkReg (MODE_PIPE);

  ...

endmodule
```

Retire stage module: help-functions and useful predicates

From Code/src_Fife/S5_Retire.bsv

```
module mkRetire (Retire_IFC);
  ...
  function Action fa_redirect_Fetch (...);
  ...
  function Action fa_update_rd (...);
  ...
  function Action fa_retire_store_buf (...);
  ...
  RR_to_Retire x_rr_to_retire = f_RR_to_Retire.first;

  Bool wrong_path = (x_rr_to_retire.epoch != rg_epoch);
  Bool is_Direct  = (x_rr_to_retire.exec_tag == EXEC_TAG_DIRECT);
  Bool is_Control = (x_rr_to_retire.exec_tag == EXEC_TAG_CONTROL);
  Bool is_Int     = (x_rr_to_retire.exec_tag == EXEC_TAG_INT);
  Bool is_DMem    = (x_rr_to_retire.exec_tag == EXEC_TAG_DMED);
  ...
endmodule
```

Retire stage module: rule for wrong-path instructions (mispredicted)

```
----- src_Fife/S5_Retire.bsv: line 190 ... -----
rule rl_Retire_wrong_path ((rg_mode == MODE_PIPE)
                           && wrong_path);
    f_RR_to_Retire.deq;

    // Unreserve/commit rd if needed
    fa_update_rd (x_rr_to_retire, False, ?);

    // Discard related pipe
    if (is_Control) f_EX_Control_to_Retire.deq;
    if (is_Int)     f_EX_Int_to_Retire.deq;
    if (is_DMem) begin
        let mem_rsp <- pop_o (to_FIFOF_0 (f_DMem_S_rsp));
        // Send 'discard' (False) to store-buf, if needed
        fa_retire_store_buf (x_rr_to_retire, mem_rsp, False);
    end
    ...
endrule
```

Retire stage module: rules for Direct path instructions (1/2)

```
----- From Code/src_Fife/S5.Retire.bsv -----  
rule rl_Retire_CSRRxx ((rg_mode == MODE_PIPE)  
    && (! wrong_path)  
    && is_Direct  
    && (! x_rr_to_retire.exception)  
    && is_legal_CSRRxx (x_rr_to_retire.instr));  
  
    ...  
    match { .exc, .rd_val } <- csrs.mav_csrrxx (...);  
    ...  
    if (! exc)  
        ...  
    else begin  
        ...  
        rg_mode <= MODE_EXCEPTION;  
    end  
endrule
```

```
----- From Code/src_Fife/S5.Retire.bsv -----  
rule rl_Retire_MRET ((rg_mode == MODE_PIPE)  
    && (! wrong_path)  
    && is_Direct  
    && (! x_rr_to_retire.exception)  
    && is_legal_MRET (x_rr_to_retire.instr));  
  
    ...  
endrule
```

Retire stage module: rules for Direct path instructions (2/2)

```
_____ From Code/src_Fife/S5_Retire.bsv _____  
rule rl_Retire_ECALL_EBREAK ((rg_mode == MODE_PIPE)  
    && (! wrong_path)  
    && is_Direct  
    && (! x_rr_to_retire.exception)  
    && (is_legal_ECALL (x_rr_to_retire.instr)  
        || is_legal_EBREAK (x_rr_to_retire.instr)));  
    ...  
    rg_mode <= MODE_EXCEPTION;  
    ...  
endrule  
  
rule rl_Retire_Direct_exception ((rg_mode == MODE_PIPE)  
    && (! wrong_path)  
    && is_Direct  
    && x_rr_to_retire.exception);  
    ...  
    rg_mode <= MODE_EXCEPTION;  
    ...  
endrule
```

Retire stage module: rules for Execute Control and Int paths instructions

From Code/src_Fife/S5.Retire.bsv

```
rule rl_Retire_EX_Control ((rg_mode == MODE_PIPE)
                          && (! wrong_path)
                          && is_Control);

  ...
  if (! x2.exception)
    ...
  else begin
    ...
    rg_mode <= MODE_EXCEPTION;
  end
endrule
```

From Code/src_Fife/S5.Retire.bsv

```
rule rl_Retire_EX_Int ((rg_mode == MODE_PIPE)
                      && (! wrong_path)
                      && is_Int);

  ...
  if (! x2.exception)
    ...
  else begin
    ...
    rg_mode <= MODE_EXCEPTION;
  end
endrule
```

Retire stage module: rules for Execute DMem path instructions (1/2)

```
From Code/src_Fife/S5.Retire.bsv
rule rl_Retire_EX_DMem ((rg_mode == MODE_PIPE)
    && (! wrong_path)
    && is_DMem
    && (f_DMem_S_rsp.first.rsp_type != MEM_REQ_DEFERRED));

...
if (! exception)
...
else begin
...
    rg_mode <= MODE_EXCEPTION;
end
endrule
```

```
From Code/src_Fife/S5.Retire.bsv
rule rl_Retire_DMem_deferred ((rg_mode == MODE_PIPE)
    && (! wrong_path)
    && is_DMem
    && (f_DMem_S_rsp.first.rsp_type
        == MEM_REQ_DEFERRED));

...
f_DMem_req.enq (mem_req);
rg_mode <= MODE_DMEM_RSP;    // go to await response
endrule
```

Retire stage module: rules for Execute DMem path instructions (2/2)

```
_____ From Code/src_Fife/S5_Retire.bsv _____  
rule rl_Retire_DMem_rsp (rg_mode == MODE_DMEM_RSP);  
  ...  
  if (exception) begin  
    ...  
    rg_mode <= MODE_EXCEPTION;  
  end  
  else begin  
    ...  
    rg_mode <= MODE_PIPE;  
  end  
endrule
```


Retire stage module: rule for exception-handling

```
src_Fife/S5_Retire.bsv: line 485 ...  
rule rl_exception (rg_mode == MODE_EXCEPTION);  
  Bool is_interrupt = False;  
  Bit #(XLEN) tvec_pc <- csrs.mav_exception (rg_epc,  
                                              is_interrupt,  
                                              rg_cause,  
                                              rg_tval);  
  
  fa_redirect_Fetch (True, x_rr_to_retire, tvec_pc);  
  rg_mode <= MODE_PIPE;  
  log_Retire_exception (rg_flog, x_rr_to_retire, rg_epc, is_interrupt, rg_cause, rg_tval);  
endrule
```

Fife CPU: final comments

- There is very little in this chapter that is RISC-V specific; these pipeline structures are needed for a pipelined CPU for any ISA. All RISC-V-specific issues are the same as in Drum.
- Much scope for performance optimization. *E.g.*,
 - Faster delivery of redirection to Fetch (less wrong-path wastage)
 - Faster delivery of Rd value to Register-Read (less stalling)
 - In Retire, handle exceptions earlier except perhaps in CSRRxx case.

These are discussed further in Chapter 17.

End