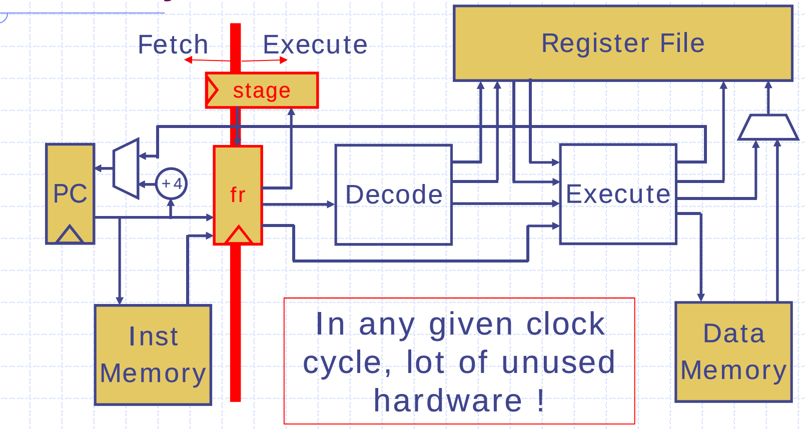
Constructive Computer Architecture:

Control Hazards

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

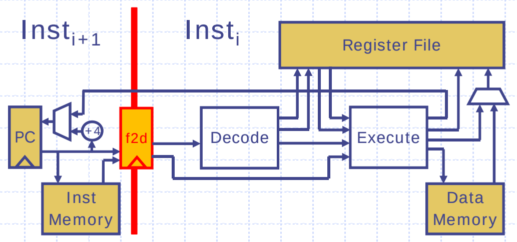
Two-Cycle RISC-V: *Analysis*



In any given clock cycle, lot of unused hardware !

Pipeline execution of instructions to increase the throughput

Problems in Instruction pipelining

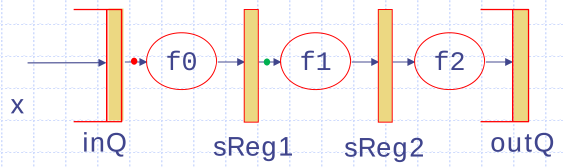


- *Control hazard:* $Inst_{i+1}$ is not known until $Inst_i$ is at least decoded. So which instruction should be fetched?
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory
- *Data hazard:* $Inst_i$ may affect the state of the machine (pc, rf, dMem) – $Inst_{i+1}$ must be fully cognizant of this change

none of these hazards were present in the FFT pipeline

Arithmetic versus Instruction pipelining

- The data items in an arithmetic pipeline, e.g., FFT, are independent of each other

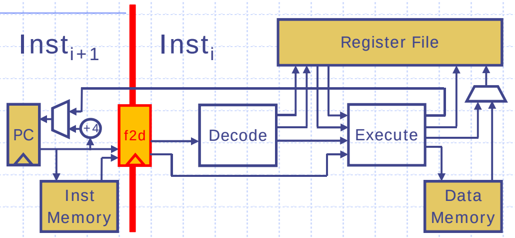


- The entities in an instruction pipeline affect each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls
  - Processor pipelines are significantly more complicated than arithmetic pipelines

The power of computers comes from the fact that the instructions in a program are *not* independent of each other

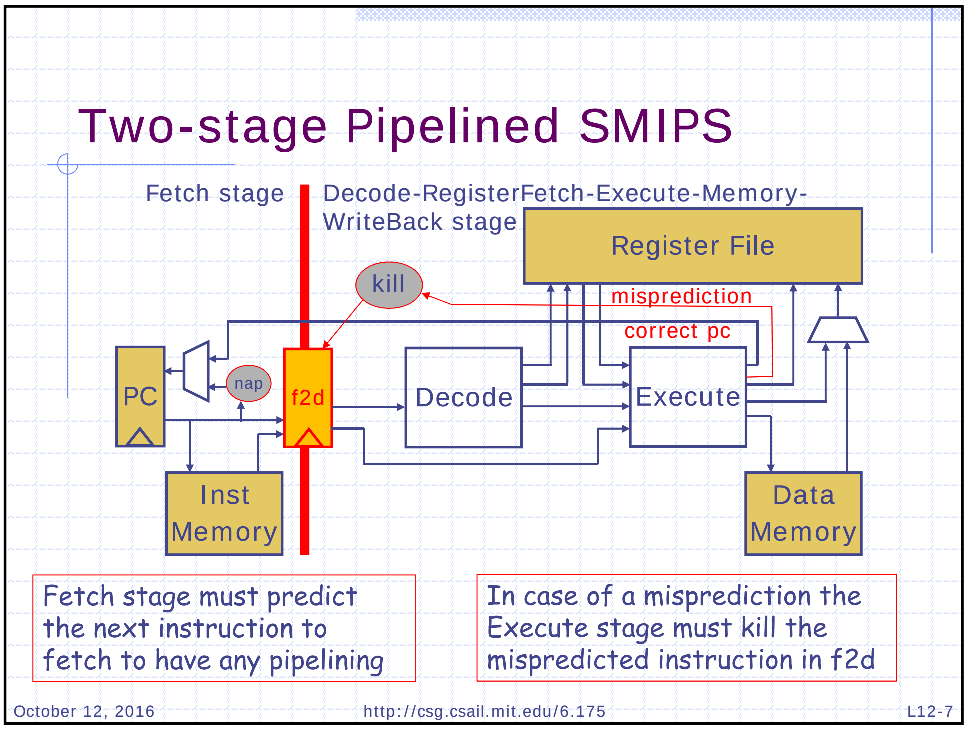
⇒ must deal with hazard

Control Hazards

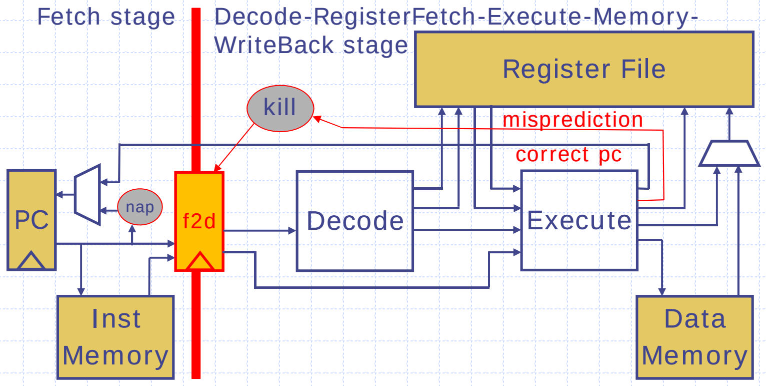


$Inst_{i+1}$ is not known until $Inst_i$ is at least decoded. So which instruction should be fetched?

- General solution – *speculate*, i.e., predict the next instruction address
  - requires the next-instruction-address prediction machinery; can be as simple as pc+4
  - prediction machinery is usually elaborate because it dynamically learns from the past behavior of the program
- What if speculation goes wrong?
  - machinery to kill the wrong-path instructions, restore the correct processor state and restart the execution at the correct pc

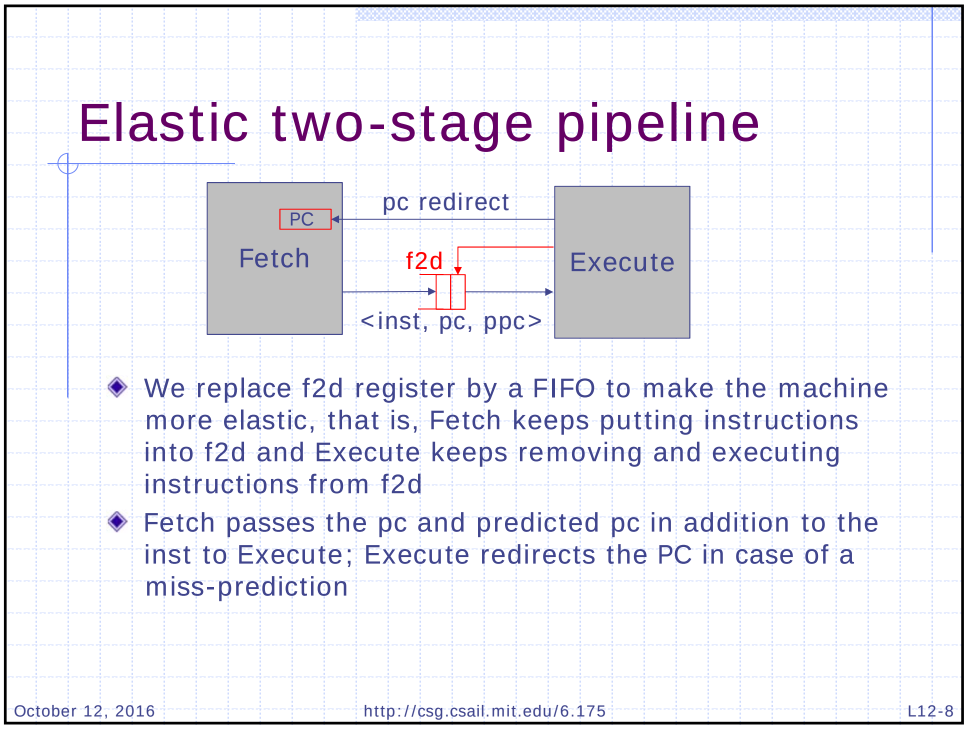


Two-stage Pipelined SMIPS

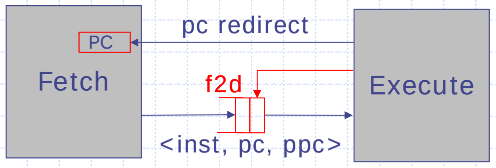


Fetch stage must predict the next instruction to fetch to have any pipelining

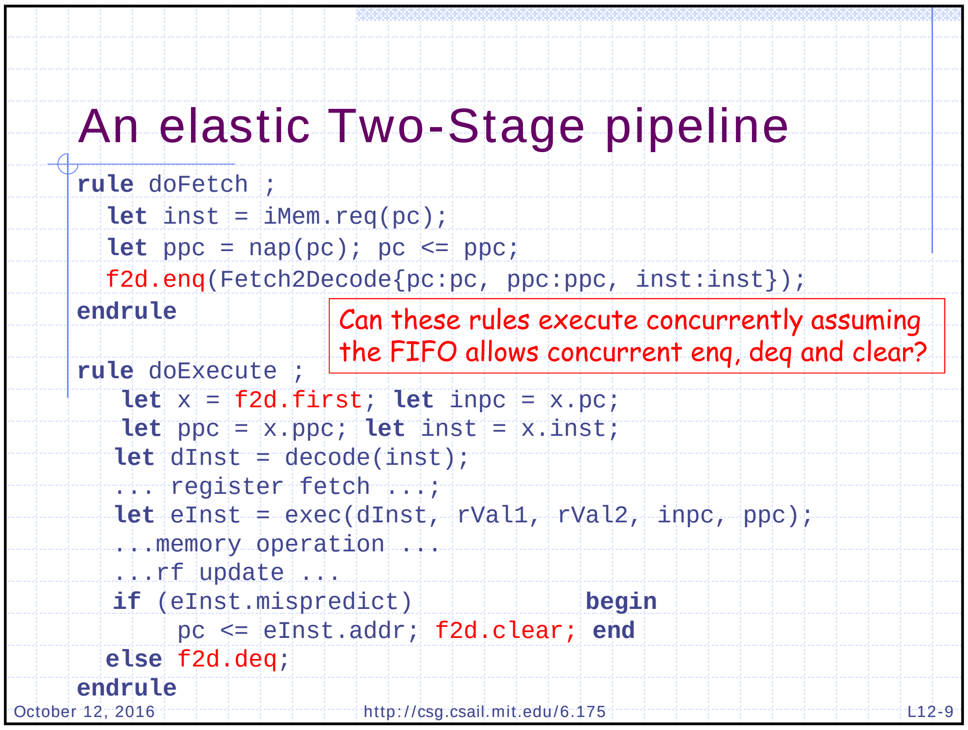
In case of a misprediction the Execute stage must kill the mispredicted instruction in f2d



Elastic two-stage pipeline



- We replace f2d register by a FIFO to make the machine more elastic, that is, Fetch keeps putting instructions into f2d and Execute keeps removing and executing instructions from f2d
- Fetch passes the pc and predicted pc in addition to the inst to Execute; Execute redirects the PC in case of a miss-prediction

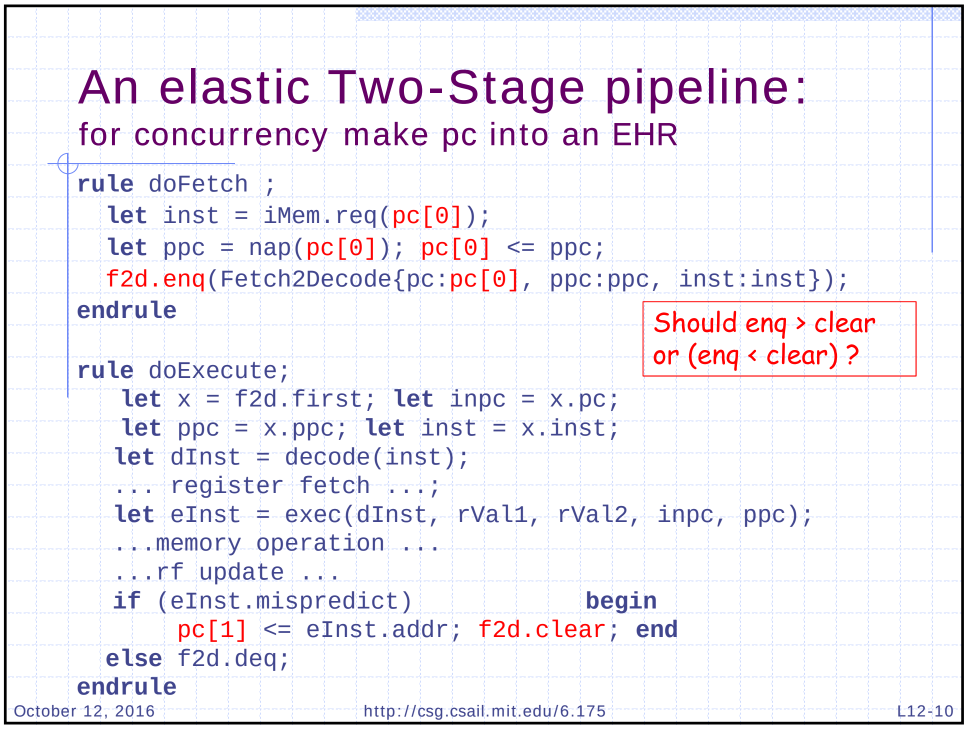


An elastic Two-Stage pipeline

```
rule doFetch ;
  let inst = iMem.req(pc);
  let ppc = nap(pc); pc <= ppc;
  f2d.enq(Fetch2Decode{pc:pc, ppc:ppc, inst:inst});
endrule

rule doExecute ;
   let x = f2d.first; let inpc = x.pc;
   let ppc = x.ppc; let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch ...;
   let eInst = exec(dInst, rVal1, rVal2, inpc, ppc);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict)            begin
        pc <= eInst.addr; f2d.clear; end
  else f2d.deq;
endrule
```

Can these rules execute concurrently assuming the FIFO allows concurrent enq, deq and clear?



An elastic Two-Stage pipeline:

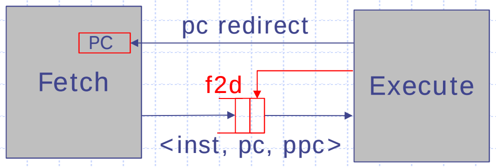
for concurrency make pc into an EHR

```
rule doFetch ;
  let inst = iMem.req(pc[0]);
  let ppc = nap(pc[0]); pc[0] <= ppc;
  f2d.enq(Fetch2Decode{pc:pc[0], ppc:ppc, inst:inst});
endrule

rule doExecute;
   let x = f2d.first; let inpc = x.pc;
   let ppc = x.ppc; let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch ...;
   let eInst = exec(dInst, rVal1, rVal2, inpc, ppc);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict)            begin
        pc[1] <= eInst.addr; f2d.clear; end
  else f2d.deq;
endrule
```

Should enq > clear or (enq < clear) ?

A correctness issue



- Once Execute redirects the PC,
  - no wrong path instruction should be executed
  - the next instruction executed must be the redirected one

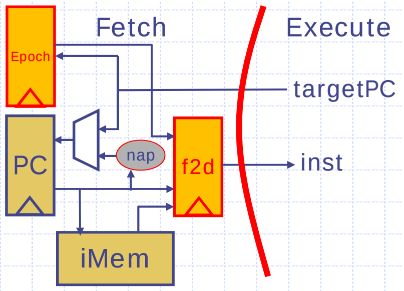
Killing fetched instructions

- In the simple design with combinational memory we have discussed so far, all the mispredicted instructions were present in f2d. So the Execute stage can *atomically:*
  - Clear f2d
  - Set pc to the correct target
- In highly pipelined machines there can be multiple mispredicted and partially executed instructions in the pipeline; it will generally take more than one cycle to kill all such instructions

Need a more general solution then clearing the f2d FIFO

Epoch: a method to manage control hazards

- Add an epoch register in the processor state
- The Execute stage changes the epoch whenever the pc prediction is wrong and sets the pc to the correct value
- The Fetch stage associates the current epoch with every instruction when it is fetched
- The epoch of the instruction is examined when it is ready to execute. If the processor epoch has changed the instruction is thrown away



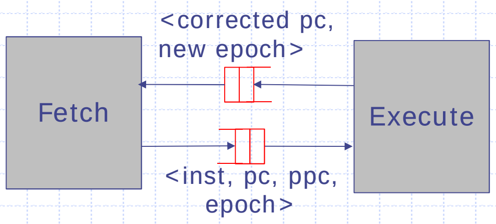
An epoch based solution

```
rule doFetch ;
  let instF=iMem.req(pc[0]);
  let ppcF=nap(pc[0]); pc[0]<=ppcF;
  f2d.enq(Fetch2Decode{pc:pc[0],ppc:ppcF,epoch:epoch,
                       inst:instF});
endrule
rule doExecute;
   let x=f2d.first; let pcD=x.pc; let inEp=x.epoch;
   let ppcD = x.ppc; let instD = x.inst;
   if(inEp == epoch) begin
     let dInst = decode(instD); ... register fetch ...;
     let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
     ...memory operation ...
     ...rf update ...
     if (eInst.mispredict)                         begin
         pc[1] <= eInst.addr; epoch <= next(epoch); end
                    end
  f2d.deq; endrule
```

Discussion

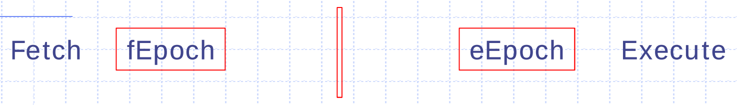
- Epoch based solution kills one wrong-path instruction at a time in the execute stage
- It may be slow, but it is more robust in more complex pipelines, if you have multiple stages between fetch and execute or if you have outstanding instruction requests to the iMem
- It requires the Execute stage to set the pc and epoch registers simultaneously which may result in a long combinational path from Execute to Fetch

Decoupled Fetch and Execute



- In decoupled systems a subsystem reads and modifies only local state atomically
  - In our solution, pc and epoch are read by both rules
- Properly decoupled systems permit greater freedom in independent refinement of subsystems

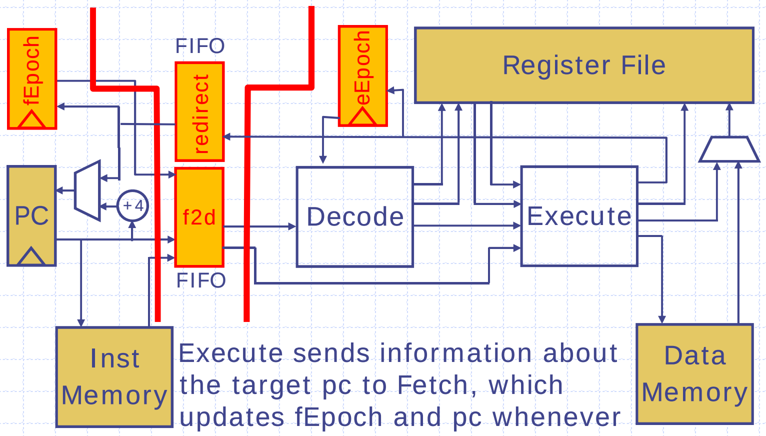
A decoupled solution using epochs



- Add fEpoch and eEpoch registers to the processor state; initialize them to the same value
- The epoch changes whenever Execute detects the pc prediction to be wrong. This change is reflected immediately in eEpoch and eventually in fEpoch via a message from Execute to Fetch
- Associate fEpoch with every instruction when it is fetched
- In the execute stage, reject, i.e., kill, the instruction if its epoch does not match eEpoch

Control Hazard resolution

A robust two-rule solution



Execute sends information about the target pc to Fetch, which updates fEpoch and pc whenever it examines the redirect (PC) fifo

Two-stage pipeline Decoupled *code structure*

```
module mkProc(Proc);
  Fifo#(Fetch2Execute) f2d <- mkFifo;
  Fifo#(Addr) redirect <- mkFifo;
  Reg#(Bool) fEpoch <- mkReg(False);
  Reg#(Bool) eEpoch <- mkReg(False);

  rule doFetch;
    let instF = iMem.req(pc);
    ...
    f2d.enq(... instF ..., fEpoch);
  endrule
  rule doExecute;
    if(inEp == eEpoch) begin
      Decode and execute the instruction; update state;
      In case of misprediction,  redirect.enq(correct pc);
                                    end
    f2d.deq;
  endrule
endmodule
```

The Fetch rule

```
rule doFetch;
 let instF = iMem.req(pc);
 if(!redirect.notEmpty)
    begin
      let ppcF = nap(pc);
      pc <= ppcF;
      f2d.enq(Fetch2Execute{pc: pc, ppc: ppcF,
                            inst: instF, epoch: fEpoch});
    end
 else
    begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;
    end
endrule
```

Notice: In case of PC redirection, nothing is enqueued into f2d

The Execute rule

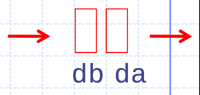
```
rule doExecute;
  let instD  = f2d.first.inst; let pcF    = f2d.first.pc;
  let ppcD   = f2d.first.ppc;  let inEp   = f2d.first.epoch;
  if(inEp == eEpoch) begin
    let dInst = decode(instD);
    let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
    let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
    let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
    if(eInst.iType == Ld) eInst.data <-
      dMem.req(MemReq{op: Ld, addr: eInst.addr, data: ?});
    else if (eInst.iType == St) let d <-
      dMem.req(MemReq{op: St, addr: eInst.addr, data: eInst.data});
    if (isValid(eInst.dst))
      rf.wr(fromMaybe(?, eInst.dst), eInst.data);
    if(eInst.mispredict) begin
      redirect.enq(eInst.addr); eEpoch <= !inEp;
    end
  end
  f2d.deq;
endrule
```

Can these rules execute concurrently?

Epoch mechanism is independent of the branch prediction scheme used. We will study sophisticated branch prediction schemes later

Conflict-free FIFO with a Clear method

To be discussed in the tutorial



```
module mkCFFifo(Fifo#(2, t)) provisos(Bits#(t, tSz));
  Ehr#(3, t) da <- mkEhr(?);
  Ehr#(2, Bool) va <- mkEhr(False);
  Ehr#(2, t) db <- mkEhr(?);
  Ehr#(3, Bool) vb <- mkEhr(False);
  rule canonicalize if(vb[2] && !va[2]);
    da[2] <= db[2]; va[2] <= True; vb[2] <= False; endrule
  method Action enq(t x) if(!vb[0]);
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq if (va[0]);
    va[0] <= False; endmethod
  method t first if(va[0]);
    return da[0]; endmethod
  method Action clear;
    va[1] <= False ; vb[1] <= False endmethod
endmodule
```

If there is only one element in the FIFO it resides in da

```
first CF enq
deq   CF enq
first < deq
enq < clear
```

Canonicalize must be the last rule to fire!

Why canonicalize must be the last rule to fire

```
rule foo ;
    f.deq; if (p) f.clear
endrule
```

Consider rule foo. If p is false then canonicalize must fire after deq for proper concurrency.

If canonicalize uses EHR indices between deq and clear, then canonicalize won't fire when p is false

```
first CF enq
deq   CF enq
first < deq
enq < clear
```