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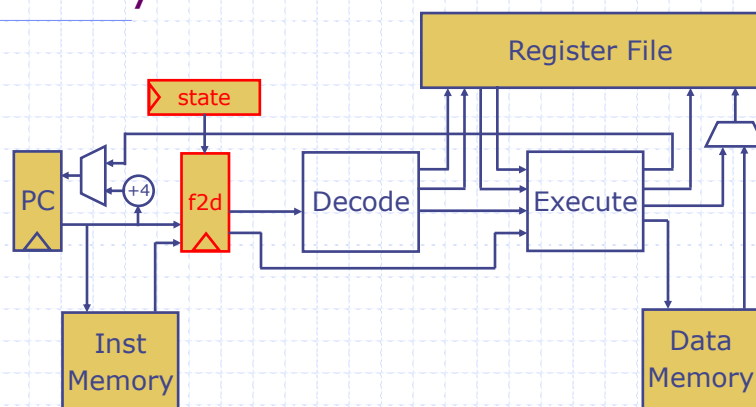
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Two-Cycle RISC-V

Introduce register "f2d" to hold a fetched instruction and register "state" to remember the state (fetch/execute) of the processor

Two-Cycle RISC-V

```
module mkProc(Proc);
  Reg#(Addr)   pc <- mkRegU;
  RFile        rf <- mkRFile;
  IMemory      iMem <- mkIMemory;
  DMemory      dMem <- mkDMemory;
  Reg#(Data)   f2d <- mkRegU;
  Reg#(State)  state <- mkReg(Fetch);

  rule doFetch (state == Fetch);
      let inst = iMem.req(pc);
      f2d <= inst;
      state <= Execute;
  endrule
```

Two-Cycle RISC V
The Execute Cycle

```
rule doExecute(stage==Execute);
  let inst = f2d;
  let dInst = decode(inst);
  let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
  let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
  let eInst = exec(dInst, rVal1, rVal2, pc);
  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);
  pc <= eInst.brTaken ? eInst.addr : pc + 4;
  state <= Fetch;            no change from single-cycle
endrule endmodule
```
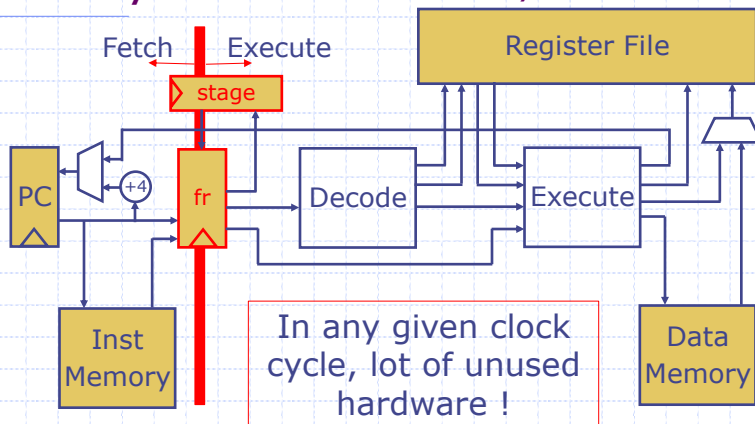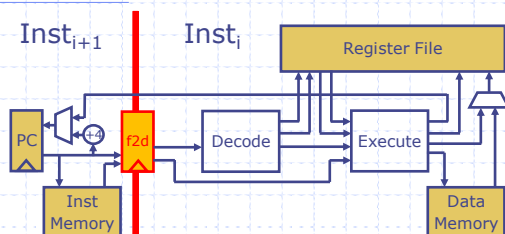
Two-Cycle RISC-V: *Analysis*

Fetch | Execute

stage

PC +4 fr → Decode → Execute

Register File

Inst Memory

In any given clock cycle, lot of unused hardware !

Data Memory

Pipeline execution of instructions to increase the throughput

Problems in Instruction pipelining

$Inst_{i+1}$ $Inst_i$

PC +4 f2d → Decode → Execute

Register File

Inst Memory

Data Memory

◆ *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

◆ Data items in an arithmetic pipeline are independent of each other

x   inQ   sReg1   sReg2   outQ
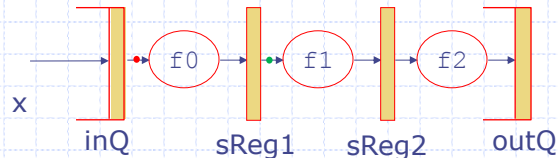
◆ An instruction in the pipeline affects future instruction
  ▪ This causes pipeline stalls or requires other fancy tricks to avoid stalls
  ▪ Processor pipelines are significantly more complicated than arithmetic pipelines

---

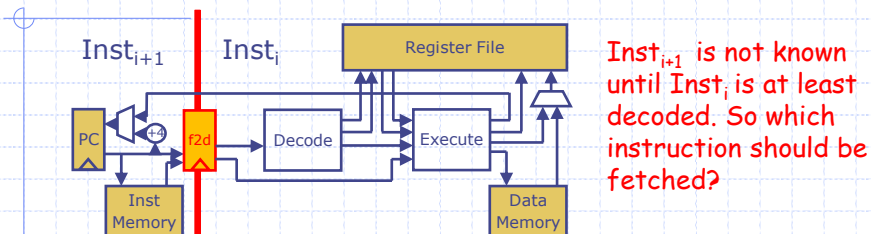The power of computers comes from the fact that the instructions in a program are *not* independent of each other

$\Rightarrow$ must deal with hazard

4

Control Hazards

Inst$_{i+1}$ Inst$_i$

Register File

PC   +4   f2d   Decode   Execute

Inst Memory

Data Memory

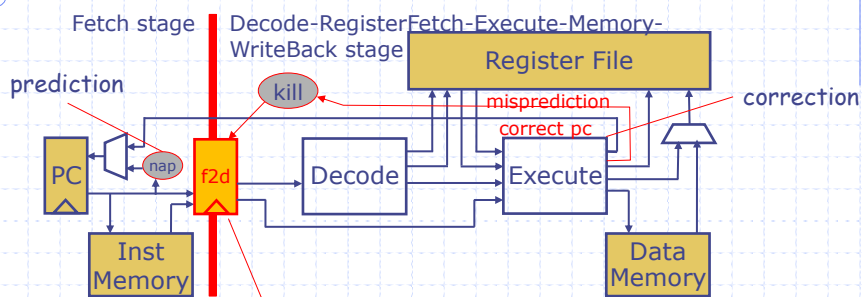Inst$_{i+1}$ is not known until Inst$_i$ is at least decoded. So which instruction should be fetched?

◆ 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
◆ When speculation goes wrong, machinery is needed to kill the wrong-path instructions, restore the correct processor state and restart the execution at the correct pc

---

Two-stage Pipelined RISC-V

Fetch stage   Decode-RegisterFetch-Execute-Memory-WriteBack stage

prediction

kill

misprediction

correct pc

correction

PC   nap   f2d   Decode   Execute

Register File

Inst Memory

Data Memory

◆ f2d must contain a Maybe type value because sometimes the fetched instruction is killed
◆ Fetch2Decode type captures all the information that needs to be passed from Fetch to Decode, i.e.
  Fetch2Decode {pc:Addr, ppc: Addr, inst:Inst}

Pipelining Two-Cycle RISC-V single rule

```
rule doPipeline ;
  let instF = iMem.req(pc);                              fetch
  let ppcF = nap(pc); let nextPc = ppcF;
  let newf2d = Valid (Fetch2Decode{pc:pc,ppc:ppcF,
                                    inst:instF});
  if(isValid(f2d)) begin                                 execute
   let x = fromMaybe(?,f2d); let pcD = x.pc;
   let ppcD = x.ppc; let instD = x.inst;
   let dInst = decode(instD);          these values are
   ... register fetch ...;             being redefined
   let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict) begin nextPc = eInst.addr;
                               newf2d = Invalid;  end
              end
   pc <= nextPc; f2d <= newf2d;
  endrule
```

Inelastic versus Elastic pipeline

◈ The pipeline presented is inelastic, that is, it relies on executing Fetch and Execute together or atomically

◈ In a realistic machine, Fetch and Execute behave more asynchronously; for example memory latency or a functional unit may take variable number of cycles

◈ If we replace ir by a FIFO (f2d) then it is possible to make the machine more elastic, that is, Fetch keeps putting instructions into f2d and Execute keeps removing and executing instructions from f2d

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?

No –
double writes in pc

March 2, 2016 http://csg.csail.mit.edu/6.375 L10-13

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
```

These rules can execute concurrently assuming the FIFO has (enq CF deq) and (enq < clear)
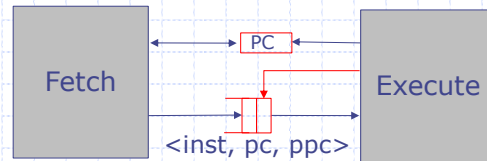
Can you design such a FIFO?

March 2, 2016 http://csg.csail.mit.edu/6.375 L10-14

7

Correctness issue

Fetch — PC — Execute

<inst, pc, ppc>

◆ Once Execute redirects the PC,
  - no wrong path instruction should be executed
  - the next instruction executed must be the redirected one
◆ This is true for the code shown because
  - Execute changes the pc and clears the FIFO atomically (assume the effect of clear is after enq)
  - Fetch reads the pc and enqueues the FIFO atomically

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 for managing 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

Fetch   Execute

Epoch

targetPC

PC   nap   f2d → inst
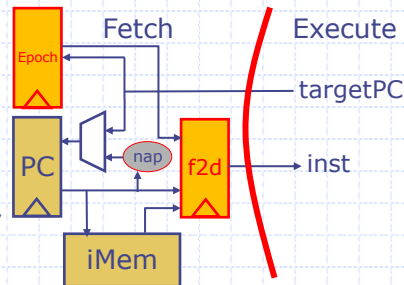
iMem

An epoch based solution

```
rule doFetch ;                    Can these rules execute concurrently ?
  let instF=iMem.req(pc[0]);
  let ppcF=nap(pc[0]); pc[0]<=ppcF;          yes
  f2d.enq(Fetch2Decode{pc:pc[0],ppc:ppcF,epoch:epoch,
                    inst:instF});
endrule                   Two values for epoch are sufficient !
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
```
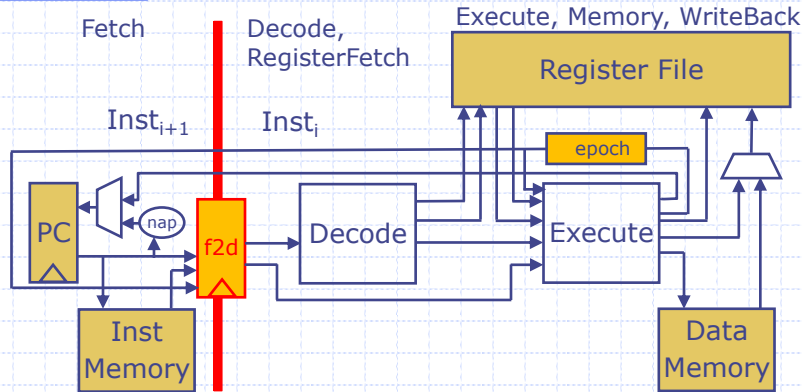
9

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

Data Hazards

Consider a different two-stage pipeline

Fetch | Decode, RegisterFetch | Execute, Memory, WriteBack

Register File

epoch

$Inst_{i+1}$ | $Inst_i$

PC | nap | f2d | Decode | Execute

Inst Memory

Data Memory

Suppose we move the pipeline stage from Fetch to after Decode and Register fetch for a better balance of work in two stages

Pipeline will still have control hazards
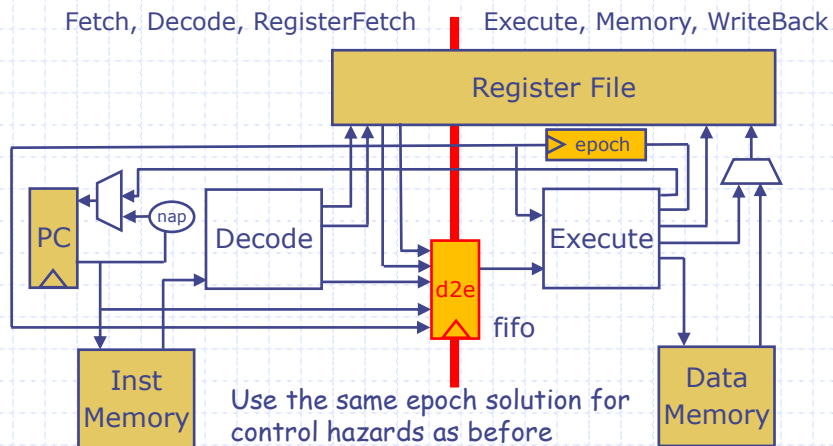

A different 2-Stage pipeline:
2-Stage-DH pipeline

Fetch, Decode, RegisterFetch | Execute, Memory, WriteBack

Register File

epoch

PC | nap | Decode | d2e | Execute

fifo

Inst Memory

Data Memory

Use the same epoch solution for control hazards as before

11

Converting the old pipeline into the new one

```
rule doFetch;
...    let instF = iMem.req(pc);
       f2d.enq(Fetch2Execute{... inst: instF ...}); ...
endrule

rule doExecute;                    instF
...    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);
...
endrule
```

Not quite correct. Why?

Fetch is potentially reading stale values from rf

Data Hazards

fetch & decode → d2e → execute   pc | rf | dMem

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_3	FD_4	FD_5			
EXstage			EX_1	EX_2	EX_3	EX_4	EX_5		

I_1 $R1 \leftarrow R2+R3$
I_2 $R4 \leftarrow R1+R2$

I_2 must be stalled until I_1 updates the register file

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
FDstage		FD_1	FD_2	FD_2	FD_3	FD_4	FD_5		
EXstage			EX_1		EX_2	EX_3	EX_4	EX_5	

12

Dealing with data hazards

◆ Keep track of instructions in the pipeline and determine if the register values to be fetched are stale, i.e., will be modified by some older instruction still in the pipeline. This condition is referred to as a read-after-write (RAW) hazard

◆ Stall the Fetch from dispatching the instruction as long as RAW hazard prevails

◆ RAW hazard will disappear as the pipeline drains

> Scoreboard: A data structure to keep track of the instructions in the pipeline beyond the Fetch stage

---

Data Hazard

◆ Data hazard depends upon the match between the source registers of the fetched instruction and the destination register of an instruction already in the pipeline

◆ Both the source and destination registers must be Valid for a hazard to exist

```
function Bool isFound
    (Maybe#(RIndex) x, Maybe#(RIndex) y);
  if(x matches Valid .xv &&& y matches Valid .yv
                      &&& yv == xv)
      return True;
  else return False;
endfunction
```

13

Scoreboard: Keeping track of instructions in execution

◆ Scoreboard: a data structure to keep track of the destination registers of the instructions beyond the fetch stage

- *method insert:* inserts the destination (if any) of an instruction in the scoreboard when the instruction is decoded
- *method search1(src):* searches the scoreboard for a data hazard
- *method search2(src):* same as *search1*
- *method remove:* deletes the oldest entry when an instruction commits

---

2-Stage-DH pipeline: Scoreboard and Stall logic
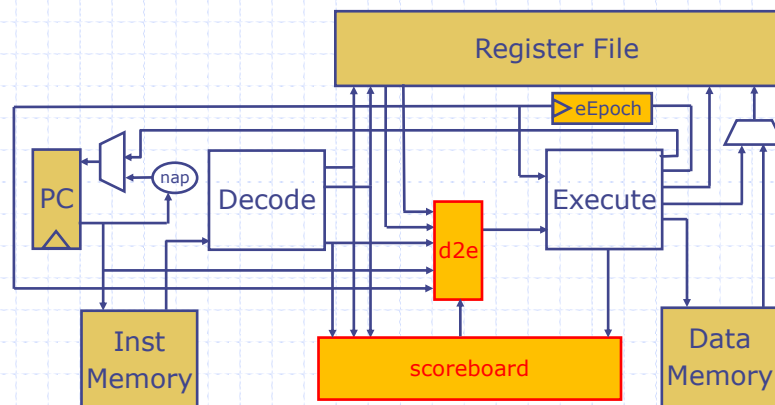
14

2-Stage-DH pipeline doFetch rule

```
rule doFetch;

    let instF = iMem.req(pc[0]);
    let ppcF = nap(pc[0]); pc[0] <= ppcF;
    let dInst = decode(instF);
    let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
    if(!stall)                     begin
       let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
       let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
       d2e.enq(Decode2Execute{pc: pc[0], ppc: ppcF,
              dInst: dInst, epoch: epoch,
              rVal1: rVal1, rVal2: rVal2});
       sb.insert(dInst.rDst); end

endrule
```

What should happen to pc when Fetch stalls?

pc should change only when the instruction is enqueued in d2e

To avoid structural hazards, scoreboard must allow two search ports

2-Stage-DH pipeline doFetch rule

```
rule doFetch;

    let instF = iMem.req(pc[0]);
    let ppcF = nap(pc[0]); pc[0] <= ppcF;
    let dInst = decode(instF);
    let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
    if(!stall)                     begin
       let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));
       let rVal2 = rf.rd2(fromMaybe(?, dInst.src2));
       d2e.enq(Decode2Execute{pc: pc[0], ppc: ppcF,
              dInst: dInst, epoch: epoch,
              rVal1: rVal1, rVal2: rVal2});
       sb.insert(dInst.rDst); pc[0] <= ppcF; end

endrule
```

15

2-Stage-DH pipeline doExecute rule

```
rule doExecute;
    let x = d2e.first;
    let dInstE = x.dInst; let pcE    = x.pc;
    let ppcE   = x.ppc;   let inEpoch = x.epoch;
    let rVal1E = x.rVal1; let rVal2E = x.rVal2;
    if(inEpoch == epoch) begin
      let eInst = exec(dInstE, rVal1E, rVal2E, pcE, ppcE);
      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
        pc[1] <= eInst.addr; epoch <= !epoch;   end
    end
    d2e.deq; sb.remove;
endrule
```

Summary

◆ Instruction pipelining requires dealing with control and data hazards

◆ Speculation is necessary to deal with control hazards

◆ Data hazards are avoided by withholding instructions in the decode stage until the hazard disappears

◆ Performance issues are subtle
  - Data values can be bypassed from later stages to register fetch stage to reduce stalls
  - Bypassing can introduce longer combinational paths which can slow down the clock

Some extra slides follow

16

WAW hazards

◆ If multiple instructions in the scoreboard can update the register which the current instruction wants to read, then the current instruction has to read the update for the youngest of those instructions

◆ This is not a problem in our design because
  ▪ instructions are committed in order
  ▪ the RAW hazard for the instruction at the decode stage will remain as long as the any instruction with the required destination is present in sb

An alternative design for sb

◆ Instead of keeping track of the destination of every instruction in the pipeline, we can associated a counter with every register to indicate the number of instructions in the pipeline for which this register is the destination
  ▪ The appropriate counter is incremented when an instruction enters the execute stage and decremented when the instruction is committed

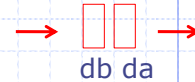   *This design is more efficient (less hardware) because it avoids an associative search*

Conflict-free FIFO with a Clear method

db da

```
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 true

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

18