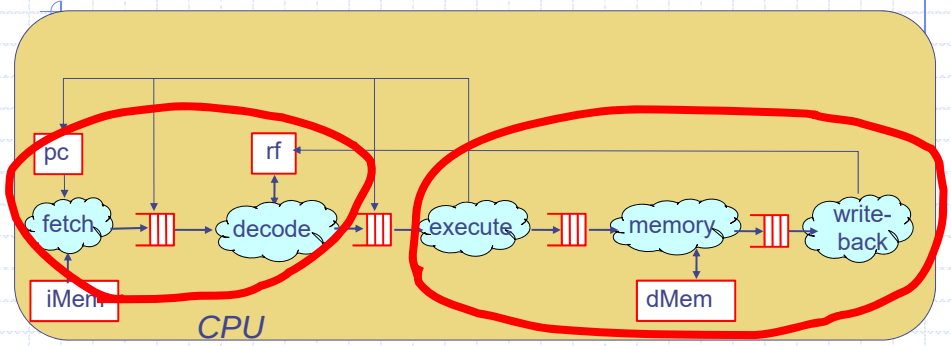
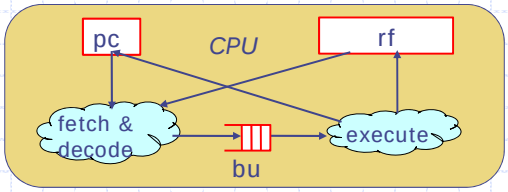
Modular Refinement

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Successive refinement & Modular Structure



Can we derive a multi-stage pipeline by successive refinement of a 2-stage pipeline?

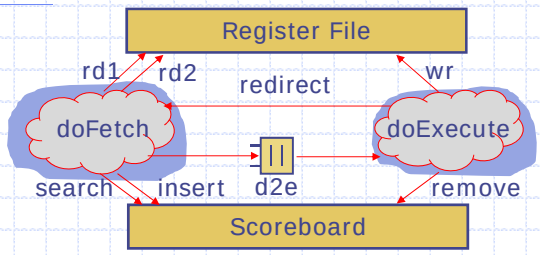


Architectural refinements

- Latency-insensitive communication between modules for better decoupling
- Refine each module independently
  - Separate Fetch and Decode
  - Replace magic memory by multicycle memory
  - Introduce Multicycle functional units (MUL, DIV)
  - ...
- A refined module should be able to work with other unrefined or refined modules

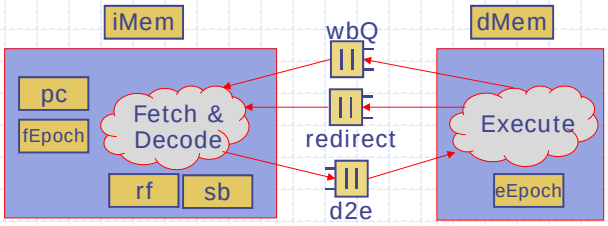
Nirav Dave, M.C. Ng, M. Pellauer, Arvind [Memocode 2010]
A design flow based on modular refinement

A 2-stage Processor Pipeline



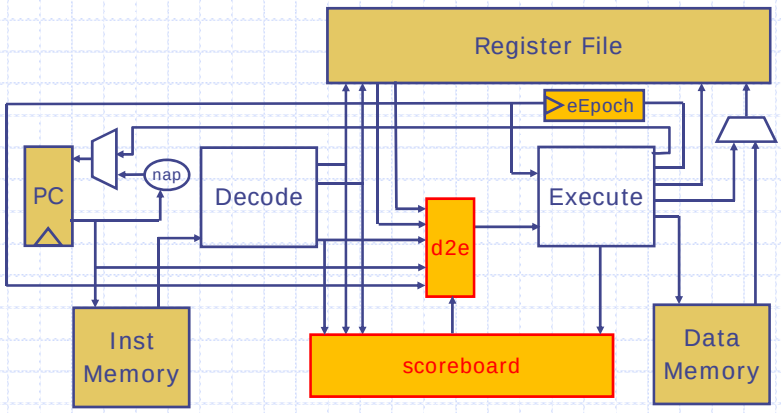
- Suppose we encapsulate each rule into its own module
- Communication from fetch to Execute via d2e is latency insensitive but not the redirection from Execute to Fetch
- Interactions between reads and writes to the register file or the interactions between search and remove to the score board are also not latency insensitive

Making the design latency Insensitive



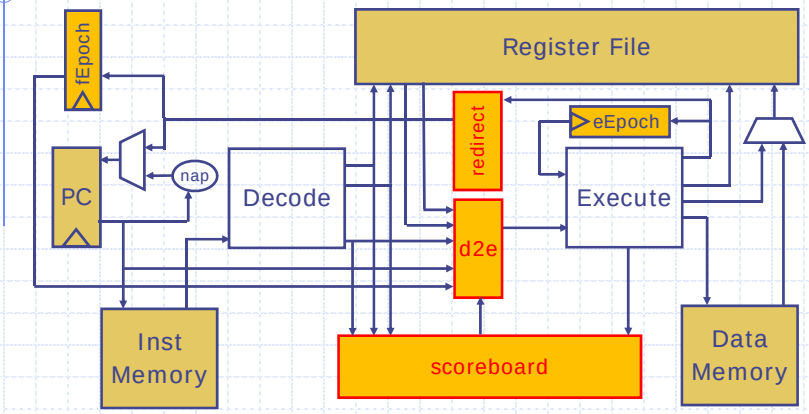
- Instead of directly resetting the pc and epoch in case of a redirection, let the Execute send the relevant information to Fetch via the redirect fifo
- Include rf and sb in the Fetch module and let the Execute send the information to update rf and sb to Fetch via the wbQ

Tightly coupled 2-Stage-pipeline: Scoreboard and Stall logic



From lecture L10

Making redirection latency-insensitive



- No direct access to PC from Execute;
- Fetch accesses its own epoch which is kept consistent with the epoch in the Execute lazily

A decoupled solution using epochs

Fetch | fEpoch | eEpoch | Execute

- 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

Fixing the doExecute rule for latency-independent pc update

```
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
```


Fixing the doFetch rule

```
rule doFetch;

      let instF = iMem.req(pc[0]);
      let ppcF = nap(pc[0]);
      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
```

-pc is no longer an EHR

-redirected pc value much be read from the redirect FIFO

Fetch rules for latency independent pc update,

```
rule redirectPC;
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;
endrule
rule doFetch (!redirect.notEmpty);
      let instF = iMem.req(pc);
      let ppcF = nap(pc); 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, ppc: ppcF,
               dIinst: dInst, epoch: fEpoch,
               rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc <= ppcF; end
endrule
```


Fixing the doExecute rule for latency-independent rf update

```
rule doExecute;
    let x = d2e.first;
    ...
    ...
    ...
    if(inEpoch == eEpoch) 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
        redirect.eng(eInst.addr); eEpoch <= !eEpoch;  end
    end
    d2e.deq; sb.remove;
endrule
```

Latency-insensitive doExecute

```
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 == eEpoch) 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});
      wbQ.enq(eInst.dst, eInst.data);
      if(eInst.mispredict) begin
        redirect.enq(eInst.addr); eEpoch <= !eEpoch;  end
    end
    else wbQ.enq(tuple2(Invalid, ?));
    d2e.deq;
endrule
```


We need to introduce a WB rule in the Fetch stage

```
rule writeback;
   match {.idx, .val} = wbQ.first;
   if(isValid(idx)) rf.write(fromMaybe(?, idx), val);
   wbQ.deq;   sb.remove;
endrule
```

- Are data hazards being handled properly?
- That is, can stale values be read from the rf?

Modular Processor

```
module mkModularProc(iMem, dMem);
  Fifo#(Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(Addr)  redirect <- mkBypassFifo;
  Fifo#(Tuple2#(Maybe#(Indx),Data) wbQ<-mkBypassFifo;
  Fetch     fetch <- mkFetch(iMem, d2e, redirect, wbQ);
  Execute execute <- mkExecute(dMem, d2e, redirect, wbQ);
endmodule
```

We only need to pass the enq interfaces to execute

Similarly we need to pass only deq and first interfaces to Fetch

Fetch Module

```
module mkFetch(iMem, d2e, redirect, wbQ); types not shown
            Make pc, fEpoch, rf, sb;
rule writeback;
   match {.idx, .val} = wbQ.first;
   if(isValid(idx)) rf.write(fromMaybe(?, idx), val);
   wbQ.deq;   sb.remove;
endrule

rule redirectPC ;
    fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;
endrule;

rule doFetch
...
endrule
```

writeback and redirectPC are CF;
doFetch and redirectPC are ME

interaction between writeback and doFetch depends upon what assumptions are made about rf and sb

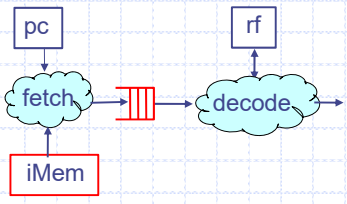
bypass rf and pipeline sb ==> writeback < doFetch

normal rf and CF sb ==> doFetch < writeback

Execute Module

```
module mkExecute(dMem, d2e, redirect, wbQ) types not shown
            Make  eEpoch;
rule doExecute;
    let x=d2e.first; let dInst=x.dInst; let pc=x.pc; let ppc=x.ppc;
    let epoch = x.epoch; let rVal1 = x.rVal1; let rVal2 = x.rVal2;
    if(epoch == eEpoch)                                          begin
      let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
      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});
      wbQ.enq(tuple2(eInst.dst, eInst.data);
      if(eInst.mispredict)                                       begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch;       end end
    else wbQ.enq(tuple2(Invalid, ?));
    d2e.deq; endrule
```


Modular refinement: Separating Fetch and Decode



Fetch Module refinement

```
module mkFetch(iMem,d2e,redirect,wbQ);
    Reg#(Addr)          pc <- mkRegU;
    Reg#(Bool)      fEpoch <- mkReg(False);
    RFile               rf <- mkBypassRFile;
    Scoreboard#(1)      sb <- mkPipelineScoreboard;
    Fifo#(Fetch2Decode) f2d <- mkPipelineFifo;

rule writeback; ... endrule
rule redirectPC; ... endrule

rule fetch  ; .... endrule
rule decode ; .... endrule
```

doFetch rule will be split into fetch and decode rules

Fetch Module: Fetch rule

```
rule fetch ;
      let inst = iMem.req(pc);
      let ppc = nextAddrPredictor(pc);
      f2d.enq(Fetch2Decode{pc: pc, ppc: ppc,
                    inst: inst, epoch: fEpoch);
      pc <= ppc
endrule
```

Fetch Module: Decode rule

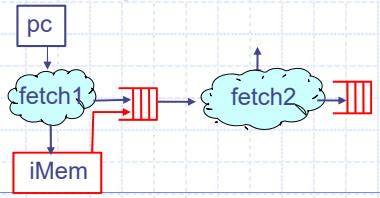
```
rule decode ;
      let x = f2d.first;
      let inst = x.inst; let inPC = x.pc; let ppc = x.ppc
      let inEp = x.epoch
      let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)   begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: inPC, ppc: ppc,
              dInst: dInst, epoch: inEp;
              rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.dst);
         f2d.deq   end
endrule
```

with pipelined f2d decode < fetch

Separate refinement

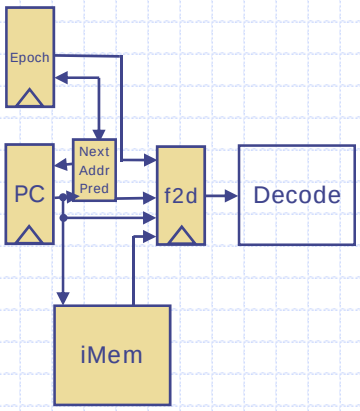
- Notice our refine Fetch&Decode module should work correctly with the old Execute module or its refinements
- This is a very important aspect of modular refinements

Modular refinement: Replace magic memory by multicycle memory



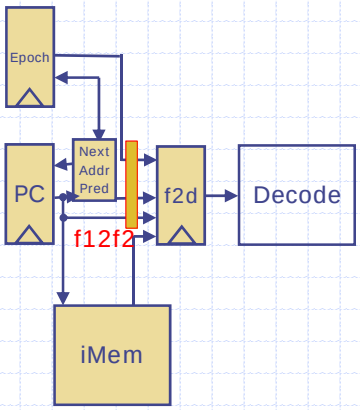
Memory and Caches

- Suppose iMem is replaced by a cache which takes one cycle in case of a hit and unknown number of variable cycles on a cache miss
- View iMem as a request/response system and split the fetch stage into two rules – to send a req and to receive a res

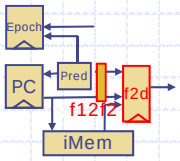


Splitting the fetch stage

- To split the fetch stage into two rules, insert a FIFO to deal with multicycle memory responses



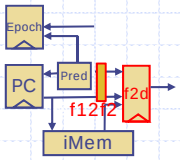
Fetch Module: 2nd refinement




```
module mkFetch(iMem,d2e,redirect,wbQ);
    Reg#(Addr)          pc <- mkRegU;
    Reg#(Bool)      fEpoch <- mkReg(False);
    RFile               rf <- mkBypassRFile;
    Scoreboard#(1)      sb <- mkPipelineScoreboard;
    Fifo#(Fetch2Decode)   f2d <- mkPipelineFifo;
    Fifo#(Fetch2Decode) f12f2 <- mkPipelineFifo;

rule writeback; ... endrule
rule redirectPC; ... endrule
rule fetch1  ; .... endrule
rule fetch2  ; .... endrule
rule decode ; .... endrule
```

Fetch Rules




```
rule fetch1 ;
      let ppc = nap(pc); pc <= ppc;
      iCache.req(MemReq{op: Ld, addr: pc, data:?});
      f12f2.enq(Fetch2Decoode{pc: pc, ppc: ppc,
                              inst: ?, epoch: fEpoch});
endrule

rule doFetch2;
  let inst <- iCache.resp;
  let x = f12f2.first;
  x.inst = inst;
  f12f2.deq;
  f2d.enq(x);
endrule
```


Takeaway

- Modular refinement is a powerful idea; lets different teams work on different modules with only an early implementation of other modules
- BSV compiler currently does not permit separate compilation of modules with interface parameters
- Recursive call structure amongst modules is supported by the compiler in a limited way.
  - The syntax is complicated
  - Compiler detects and rejects recursive call structures

Interface issues

- For better safety only partial interfaces should to be passed to a module, e.g.,
  - Fetch module needs only deq and first methods of execRedirect and wbQ, and enq method of d2e

```
interface FifoEnq#(t); method Action enq(t x); endinterface
interface FifoDeq#(t); method Action deq; method t first;
endinterface
function FifoEnq#(t) getFifoEnq(Fifo#(n, t) f); return
  interface FifoEnq#(t); method Action enq(t x) = f.enq(x);
endinterface endfunction
function FifoDeq#(t) getFifoDeq(Fifo#(n, t) f); return
  interface FifoDeq#(t); method Action deq=f.deq;
  method t first=f.first; endinterface endfunction
module mkModularProc(Proc);
  Fetch fetch <- mkFetch(iMem, getFifoEnq(d2e),
                  getFifoDeq(execRedirect), getFifoDeq(wbQ));
```