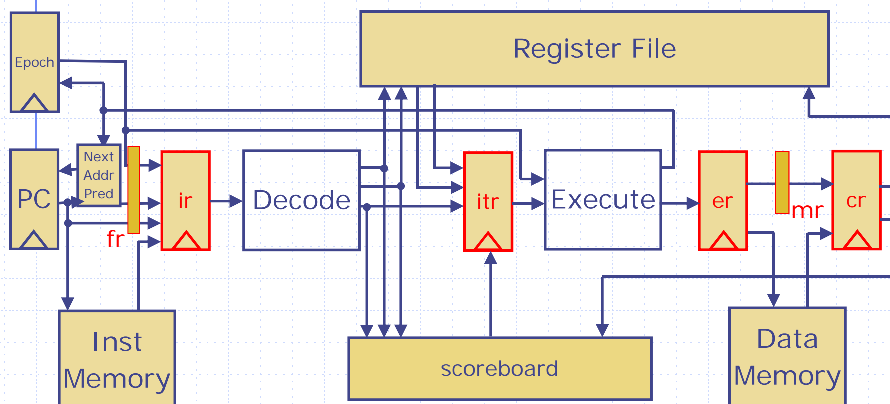
Caches and in-order pipelines

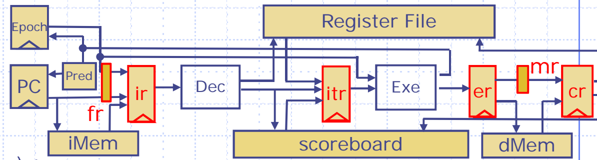
Arvind (with Asif Khan)
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Multi-Stage SMIPS



insert bypass FIFO's to deal with (0,n) cycle memory response

Multi-Stage SMIPS

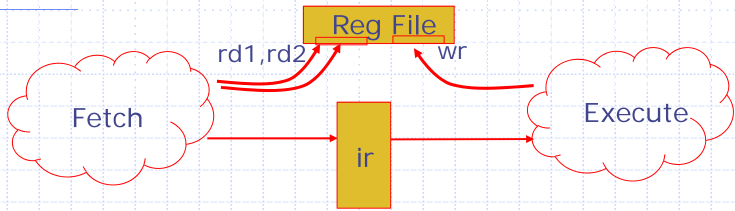


state declarations

```
module mkProc(Proc);
  EHR#(2, Addr)      pc <- mkEHRU;
  RFile              rf <- mkBypassRFile;
  Scoreboard#(4)     sb <- mkScoreboard;
  EHR#(2, Bool)   epoch <- mkEHR(False);
  NextAddrPred    bpred <- mkBTB;
  CombCache      iCache <- mkICache;
  CombCache      dCache <- mkDCache;

  BFIFO#(TypeFetch2Fetch)      fr <- mkBypassFIFO;
  PFIFO#(TypeFetch2Decode)     ir <- mkPipeFIFO;
  PFIFO#(TypeDecode2Execute) itr <- mkPipeFIFO;
  PFIFO#(TypeExecute2Memory)  er <- mkPipeFIFO;
  BFIFO#(TypeMemory2Memory)   mr <- mkBypassFIFO;
  PFIFO#(TypeMemory2Commit)   cr <- mkPipeFIFO;
```

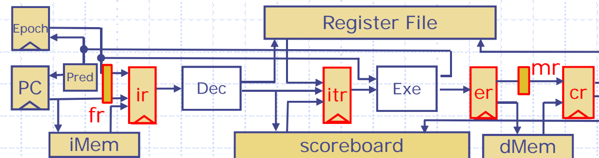
The choice of type of FIFOs, Reg vs EHR depends upon the architecture

Different architectures require different scheduling



- if ir is a pipelined FIFO (deq<enq) then reg file has to be a bypass register file (wr < {rd1,rd2})
  - a design with bypass paths
- if ir is normal FIFO (deq CF enq) then reg file can be either ordinary or bypass register file, resulting in two different architectures

Design 1: Processor Rule ordering



- doCommit <
- (doMem1 < doMem2) <
- doExecute <
- doDecode <
- (doFetch1 < doFetch2)

⇒ cr, er, itr, ir are pipeline FIFOs
(first < deq < enq)

⇒ mr, fr are bypass FIFO
(enq < first < deq)

Cache
req < (resp < respDeq)

Register File
wr < {rd1, rd2}

NextAddrPred
prediction < update

Scoreboard
remove < (search < insert)

epoch and pc
r0 < w0 < r1 < w1

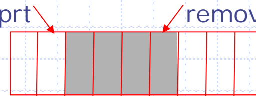
Scoreboard: Keeping track of instructions in execution

- 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 search(src1,src2): searches the scoreboard for data hazards
  - method remove: deletes the oldest entry when an instruction commits

Scoreboard

```
module mkScoreboard(Scoreboard#(size));
  Vector#(size, EHR#(2, Maybe#(Rindx))) sb
                           <- replicateM(mkEHR(Invalid));
  Reg#(Bit#(TAdd#(TLog#(size),1)))   iidx <- mkReg(0);
  Reg#(Bit#(TAdd#(TLog#(size),1)))   ridx <- mkReg(0);
  EHR#(2, Bit#(TAdd#(TLog#(size),1))) cnt <- mkEHR(0);
  Integer vsize = valueOf(size);
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);

  method Action insert(Maybe#(Rindx) r)
                            if(cnt.r1!==sz);
    sb[iidx].w1(r);
    iidx <= iidx==(sz-1) ? 0 : iidx + 1;
    cnt.w1(cnt.r1 + 1);
  endmethod
```

insert prt → iidx; rem ptr → ridx



Counter

Scoreboard *cont*

```
  method Action remove if (cnt.r0!=0);
    sb[ridx].w0(Invalid);
    ridx <= ridx==sz-1 ? 0 : ridx + 1;
    cnt.w0(cnt.r0 - 1);
  endmethod

  method Bool search(Maybe#(Rindx) s1,
                     Maybe#(Rindx) s2);
    Bool j = False;
    for (Integer i=0; i<vsize; i=i+1)
      j = j || dataHazard(s1, s2, sb[i].r1);
    return j;
  endmethod

endmodule
```

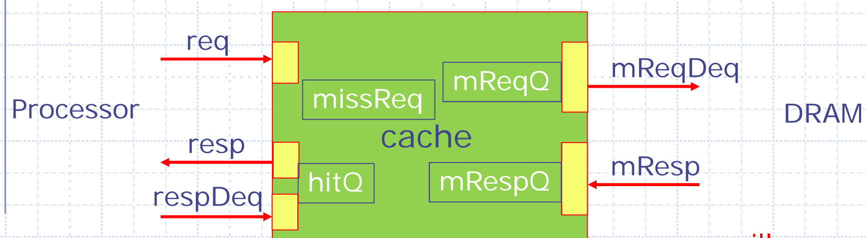
remove < search < insert

Data Hazard

- Given two source registers and a destination register determine if there is a potential for data hazard
- `src1, src2` and `rDst` in `decodedInst` are changed from `Rindx` to `Maybe#(Rindx)`

```
function Bool dataHazard(Maybe#(Rindx) src1,
                         Maybe#(Rindx) src2,
                         Maybe#(Rindx) dst);
 return (isValid(dst) &&
        (   (isValid(src1) &&
             fromMaybe(dst) == fromMaybe(src1))
         || (isValid(src2) &&
             fromMaybe(dst)== fromMaybe(src2))));
endfunction
```


Cache Interface



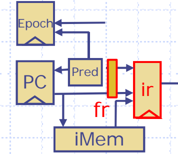

```
interface Cache;
  method Action req(MemReq r);
  method MemResp resp;
  method Action respDeq;

  method ActionValue#(MemReq) mReq;
  method Action mResp(MemResp r);
endinterface
```

processor side

memory side

we will assume caches respond in a FIFO manner and can take 0 to n cycles to respond

Fetch rules




```
rule doFetch1 (fr.notFull);
  iCache.req(TypeMemReq{op:Ld, addr:pc.r1, data:?});
  let ppc = bpred.prediction(pc.r1);
  fr.enq(TypeFecth2Fetch{pc:pc.r1, ppc:ppc,
                         epoch:epoch.r1});
  pc.w1(ppc);
endrule

rule doFetch2 (fr.notEmpty && ir.notFull);
  let frpc    = fr.first.pc;
  let frppc   = fr.first.ppc;
  let frepoch = fr.first.epoch;
  let inst = iCache.resp; iCache.respDeq;
  ir.enq(TypeFetch2Decode{pc:frpc, ppc:frppc,
                          epoch:frepoch, inst:inst});
  fr.deq;
endrule
```


Decode rule

```
rule doDecode (ir.notEmpty && itr.notFull);
  let irpc    = ir.first.pc;
  let irppc   = ir.first.ppc;
  let irepoch = ir.first.epoch;
  let inst    = ir.first.inst;
  let dInst = decode(inst);
  let stall = sb.search(dInst.src1, dInst.src2);
  if(!stall)            begin
    let rVal1 = rf.rd1(fromMaybe(dInst.src1));
    let rVal2 = rf.rd2(fromMaybe(dInst.src2));
    itr.enq(TypeDecode2Execute{pc:irpc, ppc:irppc,
                               epoch:irepoch, dInst:dInst,
                               rVal1:rVal1, rVal2:rVal2});
    sb.insert(dInst.rDst);
    ir.deq;             end
endrule
```

Execute rule

```
rule doExecute (itr.notEmpty && er.notFull);
  let itrpc=itr.first.pc;     let itrppc=itr.first.ppc;
  let dInst=itr.first.dInst;
  let rVal1=itr.first.rVal1; let rVal2=itr.first.rVal2;
  if(itr.first.epoch==epoch.r0)           begin
    let eInst=execute(dInst,rVal1,rVal2,itrpc,itrppc);
    er.enq(TypeExecute2Memory{eInst:eInst, memData:?});
    if(eInst.misprediction) begin
      let npc = eInst.brTaken ? eInst.addr : itrpc+4;
      pc.w0(npc); epoch.w0(!epoch.r0);
      bpred.update(itrpc, npc);
                      end            end
  else begin
    ExecInst eInst = ?; eInst.iType = Nop;
    er.enq(TypeExecute2Memory{eInst:eInst, memData:?});
      end
  itr.deq;
endrule
```


Data Memory rules

```
rule doMemory1 (er.notEmpty && mr.notFull);
  let eInst = er.first.eInst;
  if(memType(eInst.iType))
    dCache.req(MemReq{op:eInst.iType==Ld ? Ld : St,
                 addr:eInst.addr, data:eInst.data});
  mr.enq(TypeMemory2Memory{eInst:eInst, memData:?});
  er.deq;
endrule

rule doMemory2 (mr.notEmpty && cr.notFull);
  let eInst = mr.first.eInst;
  if(eInst.iType==Ld) begin
    let md = dCache.resp;
    dCache.respDeq;   end
  cr.enq(TypeMemory2Commit{eInst:eInst, memData:md});
  mr.deq;
endrule
```

Commit rule

```
rule doCommit (cr.notEmpty);
  let eInst   = cr.first.eInst;
  let memData = cr.first.memData;
  regUpdate(eInst, memData, rf);
  cr.deq;
  sb.remove;
endrule
endmodule
```


Cache implementations

- Blocking vs Non-blocking

  X
- Direct mapped vs Set-associative

  X
- FIFO vs Tags (Out-of-order)

  X
- I-Cache
- D-Cache

Direct-mapped Blocking Cache state declarations

```
module mkCache(Cache);
    Vector#(Rows, Reg#(LineStatus)) sArray <-
                            replicateM(mkReg(False));
    Vector#(Rows, Reg#(Tag))      tagArray <-
                            replicateM(mkRegU);
    Vector#(Rows, Reg#(Data))    dataArray <-
                            replicateM(mkRegU);

    BFIFO#(MemReq)      hitQ <- mkBypassFIFO;
    Reg#(MemReq)     missReq <- mkRegU;
    Reg#(CacheStatus) status <- mkReg(Rdy);

    FIFOF#(MemReq)   mReqQ <- mkFIFOF;
    FIFOF#(MemResp) mRespQ <- mkFIFOF;

 rule doMiss … endrule;     method ....;
endmodule
```


typedefs

```
typedef 32 AddrSz;
typedef 256 Rows;
typedef Bit#(AddrSz) Addr;
typedef Bit#(TLog#(Rows)) Index;
typedef Bit#(TSub#(AddrSz, TAdd#(TLog#(Rows), 2))) Tag;

typedef 32 DataSz;
typedef Bit#(DataSz) Data;
typedef enum {Invalid, Clean, Dirty}
  LineStatus deriving(Bits, Eq);

tyepdef enum {Rdy, WrBack, FillReq, FillResp, FillHit}
  CacheStatus deriving(Bits, Eq);
```

memory-side methods

```
method ActionValue#(MemReq) mReqDeq;
  mReqQ.deq;
 return mReqQ.first;
endmethod

method Action memResp(MemResp r);
  mRespQ.enq(r);
endmethod
```


Blocking D-Cache

processor-side methods

```
method Action req(MemReq r) if (status==Rdy);
  Index idx = truncate(r.addr>>2);
  Tag tag = truncateLSB(r.addr);
  let lnSt = sArray[idx];
  Bool tagMatch = tagArray[idx]==tag;
  if(lnSt!=Invalid && tagMatch)
    hitQ.enq(r);
  else begin
    missReq <= r;
    status <= lnSt==Dirty ? WrBack : FillReq;
       end
endmethod
```

hitQ is a bypass FIFO

It is straightforward to extend the cache interface to include a cacheline flush command

Blocking D-Cache

processor-side methods *cont.*

```
method MemResp resp if (hitQ.first.op==Ld);
  let r = hitQ.first;
  Index idx = truncate(r.addr>>2); return dataArray[idx];
endmethod

method Action respDeq if (hitQ.first.op==Ld);
  hitQ.deq;
endmethod

rule doStore(hitQ.first.op==St);
  let r = hitQ.first;
  Index idx = truncate(r.add>>2);
  dataArray[idx] <= r.data;
  sArray[idx] <= Dirty;
  hitQ.deq;
endrule
```

In case of multiword cache line, we only overwrite the appropriate word of the line

Blocking D-Cache

Rules to process cache miss: cache-to-mem

```
rule doWrBack (status==WrBack);
  Index idx = truncate(missReq.addr>>2);
  mReqQ.enq(MemReq{op:St,
                  addr:{tagArray[idx],idx,2'b00},
                  data:dataArray[idx]});
  status <= FillReq;
endrule

rule doFillReq (status==FillReq);
  mReqQ.enq(MemReq{op:Ld, addr:missReq.addr, data:?});
  status <= FillResp;
endrule
```

Both load miss and store miss generate a memory load request

Blocking D-Cache

Rules to process cache miss: mem-to-cache

```
rule doFillResp (status==FillResp);
  let data = mRespQ.first;  mRespQ.deq;

  Index idx = truncate(missReq.addr>>2);
  Tag tag = truncateLSB(missReq.addr);

  sArray[idx] <= Clean;
  tagArray[idx] <= tag;
  dataArray[idx] <= data;

  hitQ.enq(missReq);
  status <= FillHit;
endrule

rule doFillHit (status==FillHit);
  hitQ.enq(missReq);
  status <= Rdy;
endrule
```


Hit and miss behaviors

- Hit
  - Combinational read/write, i.e. 0-cycle response
  - Cache works correctly even if the processor does not immediately (combinationally) pick up the response
- Miss
  - No evacuation: memory load latency plus combinational read/write
  - Evacuation: memory store followed by memory load latency plus combinational read/write

next lecture - Non-blocking caches