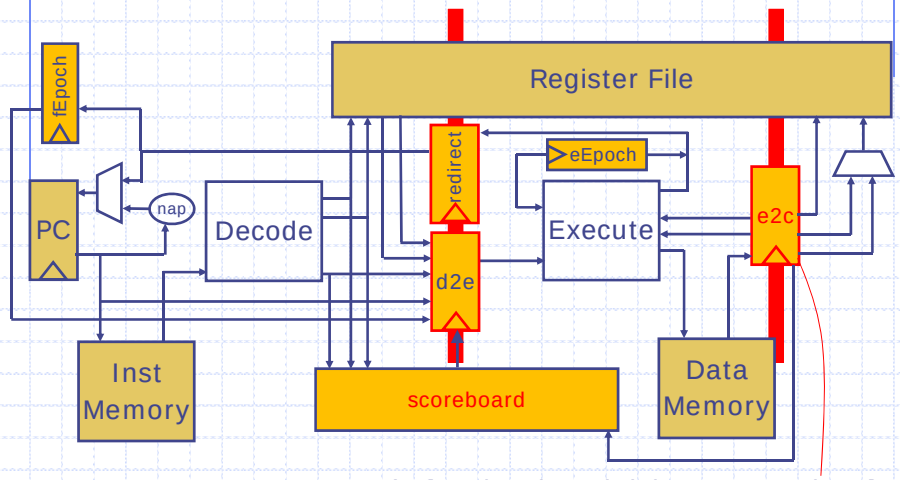
Constructive Computer Architecture:

Multistage Pipelined Processors and modular refinement

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

3-Stage-DH pipeline




```
Exec2Commit{Maybe#(RIndx)dst, Data data};
```

3-Stage-DH pipeline

```
module mkProc(Proc);
  Reg#(Addr)        pc <- mkRegU;
  RFile             rf <- mkBypassRFile;
  IMemory         iMem <- mkIMemory;
  DMemory         dMem <- mkDMemory;
  Fifo#(1, Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(1, Exec2Commit)    e2c <- mkPipelineFifo;

  Scoreboard#(2) sb <- mkPipelineScoreboard;
                 // contains two instructions

  Reg#(Bool)    fEpoch <- mkReg(False);
  Reg#(Bool)    eEpoch <- mkReg(False);
  Fifo#(Addr) redirect <- mkBypassFifo;
```


3-Stage-DH pipeline doFetch rule

Unchanged from 2-stage

```
rule doFetch;
    let inst = iMem.req(pc);
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;       end
    else
    begin
      let ppc = nap(pc); let dInst = decode(inst);
      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: ppc,
              dInst: dInst, epoch: fEpoch,
              rVal1: rVal1, rVal2: rVal2});
        sb.insert(dInst.rDst); pc <= ppc; end
    end
endrule
```

3-Stage-DH pipeline doExecute rule

```
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});
      if (isValid(eInst.dst))
        rf.wr(fromMaybe(?, eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        redirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                          end

    d2e.deq; sb.remove;
endrule
```


3-Stage-DH pipeline doCommit rule

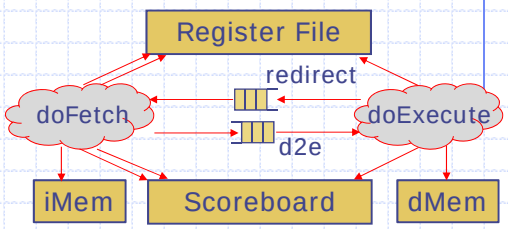
```
rule doCommit;
  let dst = e2c.first.dst;
  let data = e2c.first.data;
  if(isValid(dst))
    rf.wr(tuple2(fromMaybe(?,dst), data);
  e2c.deq;
  sb.remove;
endrule
```

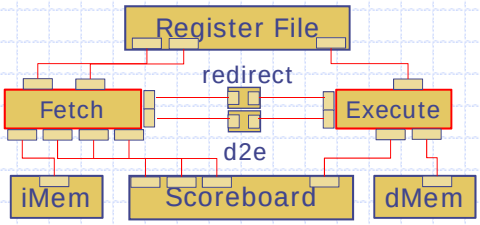
How many pipeline stages?

- Clock speed is determined by the slowest pipeline stage, i.e., the stage with the longest combinational path
  - More stages imply faster clocks but more bubbles
  - Need performance studies using benchmarks to determine the number of bubbles (not in this subject)
- Typical architectural refinements to shorten combinational paths
  - Separating Fetch and Decode
  - Replace magic memory by multicycle memory
  - Multicycle functional units
  - …

Successive refinement & Modular Structure

- Can we derive multi-stage pipelines by successive refinement of a 2-stage pipeline?
- Encapsulate doFetch and doExecute rules in their own modules respectively
  - Requires passing module interface parameters to Fetch and Execute
  - Module methods defs and calls can be directly connected to each other





Modular Processor

```
module mkModularProc(Proc);
  IMemory          iMem <- mkIMemory;
  DMemory          dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(Addr)      redirect <- mkBypassFifo;
  RFile             rf <- mkBypassRFile;
  Scoreboard#(1)    sb <- mkPipelineScoreboard;
  Fetch     fetch <- mkFetch(iMem,d2e,redirect,rf,sb);
  Execute execute <- mkExecute(dMem,d2e,redirect,rf,sb);
endmodule
```

no rules – all communication takes place via method calls to shared modules

Interface Arguments

- Any subset of methods from a module interface can be used to define a new (partial) interface

```
interface FifoEnq#(t);
    method Action enq(t x);
endinterface
```

- A function can be defined to extract the desired methods from an interface

```
function FifoEnq#(t) getFifoEnq(Fifo#(n, t) f);
  return (interface FifoEnq#(t);
            method Action enq(t x) = f.enq(x);
          endinterface);
endfunction
```

Modular Processor

```
module mkModularProc(Proc);
  IMemory          iMem <- mkIMemory;
  DMemory          dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(Addr)      redirect <- mkBypassFifo;
  RFile             rf <- mkBypassRFile;
  Scoreboard#(1)    sb <- mkPipelineScoreboard;
  Fetch     fetch <- mkFetch(iMem, getFifoEnq(d2e),
                 getFifoDeq(redirect)
                 getRfRead(rf),
                 getSbInsSearch(sb);
  Execute execute <- mkExecute(dMem,



endmodule
```


Fetch Module


```
module mkFetch(Imemory iMem,
               FifoEnq#(Decode2Execute) d2e,
               FifoDeq#(Addr) redirect,
               RegisterFileRead rf,
               ScoreboardInsert sb)
    Reg#(Addr)        pc <- mkRegU;
    Reg#(Bool)    fEpoch <- mkReg(False);

rule fetch ;
    if(redirect.notEmpty) begin
....
endrule
endmodule
```

Fetch Module *continued*

```
rule fetch ;
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;       end
    else
    begin
      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,
               dInst: dInst, epoch: fEpoch,
               rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.dst); pc <= ppcF; end
    end
endrule
```

Unchanged from 2-stage

Execute Module

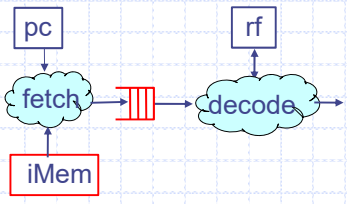

```
module mkExecute(Dmemory dMem,
                 FifoDeq#(Decode2Execute) d2e,
                 FifoEnq#(Addr) redirect,
                 RegisterFileWrite rf,
                 ScoreboardInsert sb)
    Reg#(Bool)    eEpoch <- mkReg(False);

rule doExecute;
...
endrule
endmodule
```

Execute Module *continued*

```
rule doExecute;
    let x = d2e.first; let dInstE = x.dInst;
    let pcE = x.pc; let ppcE = x.ppc;
    let epoch = x.epoch;
    let rVal1E = x.rVal1; let rVal2E = x.rVal2;
    if(epoch == 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.enq(eInst.addr); eEpoch <= !eEpoch; end end
                        end
    d2e.deq; sb.remove;
endrule
```

Unchanged from 2-stage

Modular refinement: Separating Fetch and Decode



Fetch Module refinement

```
module mkFetch(Imemory iMem,
               FifoEnq#(Decode2Execute) d2e,
               FifoDeq#(Addr) redirect,
               RegisterFileRead rf,
               ScoreboardInsert sb)
    Reg#(Addr)        pc <- mkRegU;
    Reg#(Bool)    fEpoch <- mkReg(False);
    Fifo#(Fetch2Decode) f2d <- mkPipelineFifo;

rule fetch ;
    if(redirect.notEmpty) begin
....
endrule
rule decode ; .... endrule
endmodule
```

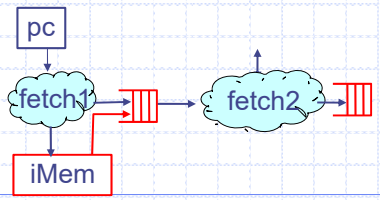

Fetch Module: Fetch rule

```
rule fetch ;
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;       end
    else
    begin
      let instF = iMem.req(pc);
      let ppcF = nap(pc);
      f2d.enq(Fetch2Decode{pc: pc, ppc: ppcF,
                           inst: instF, epoch: fEpoch);
      pc <= ppcF
    end
endrule
```

Fetch Module: Decode rule

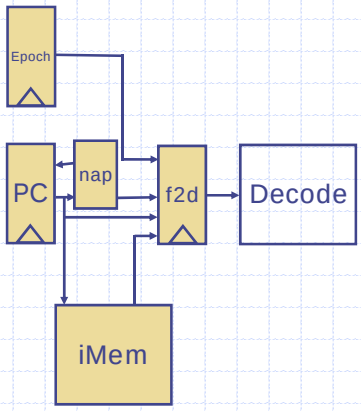
```
rule decode ;
      let x = f2d.first;
      let instD = x.inst; let pcD = x.pc; let ppcD = x.ppc
      let inEp = x.epoch
      let dInst = decode(instD);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)   begin
         let rVal1 = rf.rd1(fromMaybe(?, dInst.src1));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


Modular refinement: Replace magic memory by multicycle memory



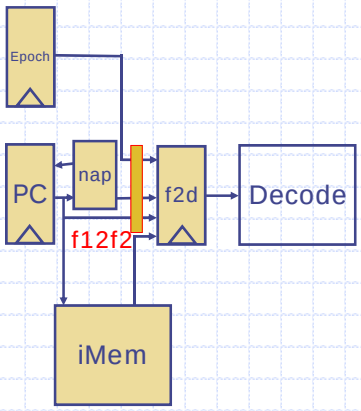
Memory and Caches

- Suppose iMem is replaced by a cache which takes 0 or 1 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

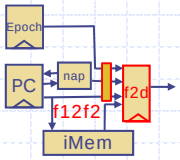


Splitting the fetch stage

- To split the fetch stage into two rules, insert a bypass FIFO's to deal with (0,n) cycle memory response

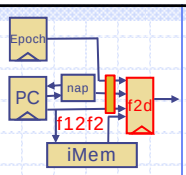


Fetch Module: 2nd refinement




```
module mkFetch(Imemory iMem,
               FifoEnq#(Decode2Execute) d2e,
               FifoDeq#(Addr) redirect,
               RegisterFileRead rf,
               ScoreboardInsert sb)
    Reg#(Addr)        pc <- mkRegU;
    Reg#(Bool)    fEpoch <- mkReg(False);
    Fifo#(Fetch2Decode) f2d <- mkPipelineFifo;
    Fifo#(Fetch2Decode) f12f2 <- mkBypassFifo;

rule fetch1; .... endrule
rule fetch2; .... endrule
rule decode; .... endrule
endmodule
```


Fetch Module




```
rule fetch1;
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first;
      redirect.deq;       end
    else begin
      let ppcF = nap(pc); pc <= ppc;
      iMem.req(MemReq{op: Ld, addr: pc, data:?});
      f12f2.enq(Fetch2Decoode{pc: pc, ppc: ppcF,
                              inst: ?, epoch: fEpoch});
        end
endrule
rule fetch2;
  let inst <- iMem.resp;   let x = f12f2.first;
  x.inst = inst;
  f12f2.deq; f2d.enq(x);
endrule
```

Separate refinement

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

Takeaway

- Multistage pipelines are straightforward extensions of 2-stage pipelines
- Modular refinement is a powerful idea; allows different teams to work on different modules with 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 truly cyclic method calls