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

- Separating Fetch and Decode
- Replace magic memory by multicycle memory
- Multicycle functional units
- ...

Suppose we encapsulate each rule into its own module and restrict the communication between modules only via FIFOs. This will require us to include the rf and sb into some module. If rf and sb are included in the Fetch module then Execute must send some extra information to Fetch to update the rf and sb.
A modular structure

- We have encapsulated iMem within the Fetch and dMem within the Execute modules
- Execute instead of updating rf and sb, sends the relevant information to Fetch via wbQ

Atomicity requirements for updating rf and sb?

---

Modular Processor

```verilog
define module mkModularProc(Proc);
  IMemory iMem <- mkIMemory;
  DMemory dMem <- mkDMemory;
  Fifo#(Decode2Execute) d2e <- mkPipelineFifo;
  Fifo#(Addr) execRedirect <- mkBypassFifo;
  Fifo#(Tuple2#(Maybe#(FullIndx),Data) wbQ<-mkBypassFifo;
  Fetch fetch <- mkFetch(iMem, d2e, execRedirect, wbQ);
  Execute execute <- mkExecute(dMem, d2e, execRedirect, wbQ);
endmodule
```
Fetch Module

```haskell
module mkFetch(Imemory iMem, Fifo#(2, Decode2Execute) d2e,
                Fifo#(1, Addr) execRedirect,
                Fifo#(1, Tuple2#(Maybe#(FullIndx), Data)) wbQ);

  Reg#(Addr)    pc <- mkRegU;
  Reg#(Bool)    fEpoch <- mkReg(False);
  RFile         rf <- mkBypassRFile;
  Scoreboard#(1) sb <- mkPipelineScoreboard;

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

rule fetch;
  if(execRedirect.notEmpty)
    begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;  end
  else
    begin
      let inst = iMem.req(pc);
      let dInst = decode(inst);
      let ppc = nextAddrPredictor(pc);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
             || sb.search3(dInst.dst);
      if(!stall)
        begin
          let rVal1 = rf.rd1(validRegValue(dInst.src1));
          let rVal2 = rf.rd2(validRegValue(dInst.src2));
          d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
                                  dInst: dInst, epoch: fEpoch,
                                  rVal1: rVal1, rVal2: rVal2});
          sb.insert(dInst.dst); pc <= ppc; end
    end
endrule
```

Fetch Module continued

```haskell
rule fetch;
  if(execRedirect.notEmpty) begin
    fEpoch <= !fEpoch;  pc <= execRedirect.first;
    execRedirect.deq;  end
  else
    begin
      let inst = iMem.req(pc);
      let dInst = decode(inst);
      let ppc = nextAddrPredictor(pc);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
             || sb.search3(dInst.dst);
      if(!stall)
        begin
          let rVal1 = rf.rd1(validRegValue(dInst.src1));
          let rVal2 = rf.rd2(validRegValue(dInst.src2));
          d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
                                  dInst: dInst, epoch: fEpoch,
                                  rVal1: rVal1, rVal2: rVal2});
          sb.insert(dInst.dst); pc <= ppc; end
    end
endrule
```
Execute Module

```verilog
module mkExecute(Dmemory dMem, Fifo#(2, Decode2Execute) d2e,
                 Fifo#(1, Addr) execRedirect,
                 Fifo#(1, Tuple2#(Maybe#(FullIndx), Data)) wbQ);

    Reg#(Bool)    eEpoch <- mkReg(False);

    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
    endmodule
```

Interface issues

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

```verilog
interface FifoEnq#(t); method Action enq(t x); endinterface
interface FifoDeg#(t); method Action deq; method t first; endinterface
```
Observation

The way we modified the 2-stage code for modularization is very similar to what we did to go from 2-stage to 3-stage code, except that we used pipeline fifo’s instead of bypass fifo’s to communicate from execute to commit stage.

Modular refinement: Separating Fetch and Decode
Fetch Module refinement

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

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

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

Fetch Module: Fetch rule

```haskell
rule fetch ;
    if(execRedirect.notEmpty) begin
        fEpoch <= !fEpoch; pc <= execRedirect.first;
        execRedirect.deq;
    end
else begin
    let inst = iMem.req(pc);
    let ppc = nextAddrPredictor(pc);
    f2d.enq(Fetch2Decode{pc: pc, ppc: ppc,
                           inst: inst, epoch: fEpoch});
    pc <= ppc
end
endrule
```
Fetch Module: Decode rule

```haskell
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, dIinst: dInst, epoch: inEp, rVal1: rVal1, rVal2: rVal2});
            sb.insert(dInst.dst);
        end
endrule
```

Concurrency

- Can writeback, fetch and decode rules execute concurrently?
- 
- 
- 
- Interaction between writeback and decode is subtle. What is the issue?
- Depending upon the choice of d2e, execRedirect and wbQ FIFOs, either
  - Fetch&Decode CF Execute
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 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(...iMem,...d2e,...execRedirect,...wbQ);
    Reg#(Addr) pc <- mkRegU;
    Reg#(Bool) fEpoch <- mkReg(False);
    RFile rf <- mkBypassRFile;
    Scoreboard#(1) sb <- mkPipelineScoreboard;
    Fifo#(Fetch2Decode) f2d <- mkPipelineFifo;
    Fifo#(Fetch2Decode) f12f2 <- mkBypassFifo;
    rule writeback; ... endrule
    rule fetch1 ; .... endrule
    rule fetch2 ; .... endrule
    rule decode ; .... endrule

Fetch Module: Fetch1 rule

rule fetch1 ;
    if(execRedirect.notEmpty) begin
        fEpoch <= !fEpoch; pc <= execRedirect.first;
        execRedirect.deq; end
    else begin
        let ppc = nextAddrPredictor(pc); pc <= ppc;
        iCache.req(MemReq{op: Ld, addr: pc, data:?});
        f12f2.enq(Fetch2Decoode(pc, ppc, inst: ?, epoch: fEpoch));
    end
endrule
**Fetch Module:**

**Fetch2 rule**

```plaintext
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.