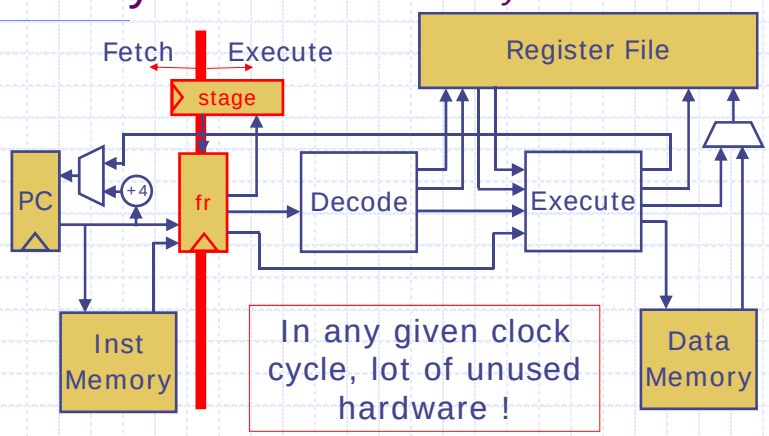
Pipelined Processors

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

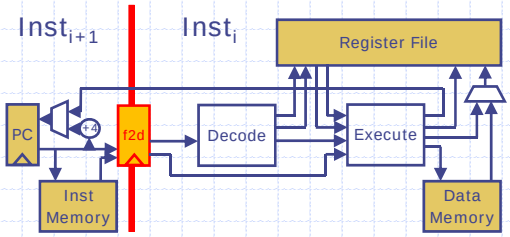
Two-Cycle SMIPS: *Analysis*



In any given clock cycle, lot of unused hardware !

Pipeline execution of instructions to increase the throughput

Problems in Instruction pipelining

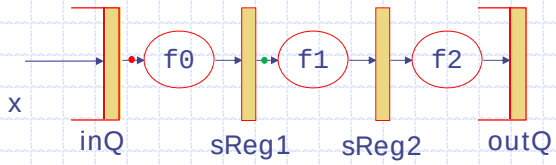


- *Control hazard:* Inst$_{i+1}$ is not known until Inst$_i$ is at least decoded. So which instruction should be fetched?
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory
- *Data hazard:* Inst$_i$ may affect the state of the machine (pc, rf, dMem) – Inst$_{i+1}$must be fully cognizant of this change

none of these hazards were present in the IFFT pipeline

Arithmetic versus Instruction pipelining

- The data items in an arithmetic pipeline, e.g., IFFT, are independent of each other

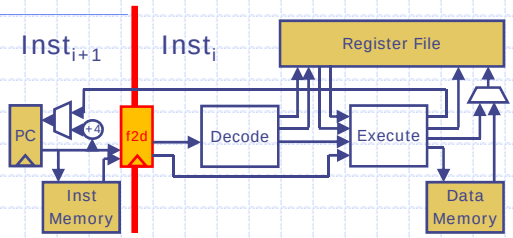


- The entities in an instruction pipeline affect each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls
  - Processor pipelines are significantly more complicated than arithmetic pipelines

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

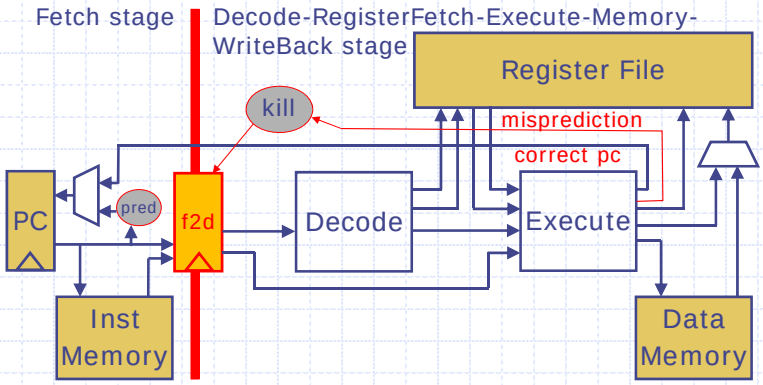
⇒ must deal with hazard

Control Hazards



- 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
- What if speculation goes wrong?
  - machinery to kill the wrong-path instructions, restore the correct processor state and restart the execution at the correct pc

Two-stage Pipelined SMIPS



Fetch stage must predict the next instruction to fetch to have any pipelining

In case of a misprediction the Execute stage must kill the mispredicted instruction in f2d

Pipelining Two-Cycle SMIPS – singlerule

```
rule doPipeline ;
  let inst = iMem.req(pc);                                   fetch
  let ppc = nextAddr(pc); let newPc = ppc;
  let newIr=Valid(Fetch2Decode{pc:pc,ppc:ppc,inst:inst});
  if(isValid(ir)) begin                                      execute
   let x = validValue(ir); let irpc = x.pc;
   let ppc = x.ppc; let inst = x.inst;
   let dInst = decode(inst);
   ... register fetch ...;
   let eInst = exec(dInst, rVal1, rVal2, irpc, ppc);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict)  begin
    newIr = Invalid;
    newPc = eInst.addr;    end
                  end
  pc <= newPc; ir <= newIr;
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 = nextAddr(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?

An elastic Two-Stage pipeline:

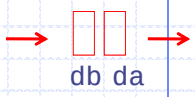
for concurrency make pc into an EHR

```
rule doFetch ;
  let inst = iMem.req(pc[0]);
  let ppc = nextAddr(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 execute concurrently assuming the FIFO has (enq CF deq) and (enq < clear)

Conflict-free FIFO with a Clear method




```
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;

endmodule
```

If there is only one element in the FIFO it resides in da

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

Why canonicalize must be last rule to fire

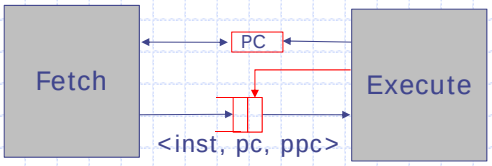
```
rule foo ;
    f.deq; if (p) f.clear
endrule
```

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 false

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


Correctness issue



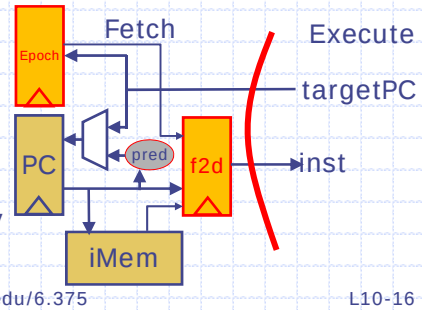
- Once Execute redirects the PC,
  - no wrong path instruction should be executed
  - the next instruction executed must be the redirected one
- This is true for the code shown because
  - Execute changes the pc and clears the FIFO atomically
  - Fetch reads the pc and enqueues the FIFO atomically

Killing fetched instructions

- In the simple design with combinational memory we have discussed so far, the mispredicted instruction was present in the f2d. So the Execute stage can atomically
  - Clear the f2d
  - Set the 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

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



Discussion

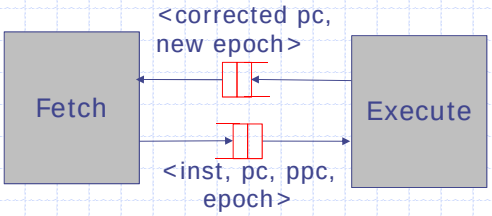
- Epoch based solution kills one wrong-path instruction at a time in the execute stage
- It may be slow, but it is more robust in more complex pipelines, if you have multiple stages between fetch and execute or if you have outstanding instruction requests to the iMem
- It requires the Execute stage to set the pc and epoch registers simultaneously which may result in a long combinational path from Execute to Fetch

An epoch based solution

Can these rules execute concurrently ?

```
rule doFetch ;
  let inst=iMem.req(pc[0]);
  let ppc=nextAddr(pc[0]); pc[0]<=ppc;
  f2d.enq(Fetch2Decode{pc:pc[0],ppc:ppc,epoch:epoch,
                       inst:inst});
endrule
rule doExecute;
   let x=f2d.first; let inpc=x.pc; let inEp=x.epoch;
   let ppc = x.ppc; let inst = x.inst;
   if(inEp == epoch) begin
     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; epoch <= epoch + 1; end
                        end
   f2d.deq; endrule
```

Decoupled Fetch and Execute



- In decoupled systems a subsystem reads and modifies only local state atomically
  - In our solution, pc and epoch are read by both rules
- Properly decoupled systems permit greater freedom in independent refinement of subsystems

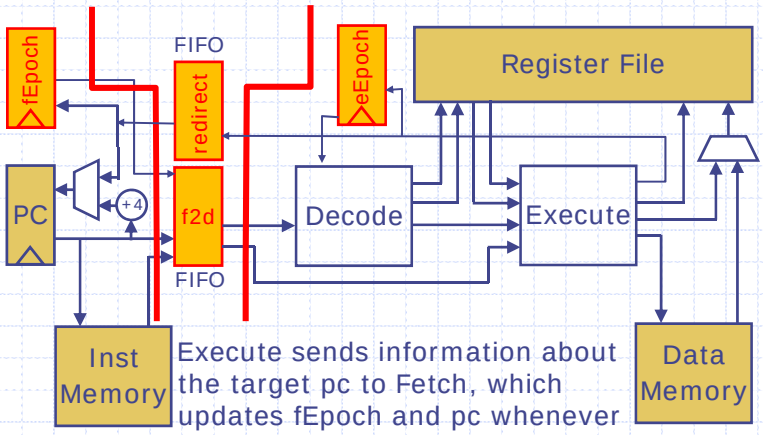
A decoupled solution using epochs

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 the 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

Control Hazard resolution

A robust two-rule solution



Execute sends information about the target pc to Fetch, which updates fEpoch and pc whenever it looks at the redirect PC fifo

Two-stage pipeline Decoupled *code structure*

```
module mkProc(Proc);
  Fifo#(Fetch2Execute) f2d <- mkFifo;
  Fifo#(Addr) execRedirect <- mkFifo;
  Reg#(Bool) fEpoch <- mkReg(False);
  Reg#(Bool) eEpoch <- mkReg(False);

  rule doFetch;
    let inst = iMem.req(pc);
    ...
    f2d.enq(... inst ..., fEpoch);
  endrule
  rule doExecute;
    if(inEp == eEpoch) begin
      Decode and execute the instruction; update state;
      In case of misprediction,  execRedirect.enq(correct pc);
                                          end
    f2d.deq;
  endrule
endmodule
```

The Fetch rule

```
rule doFetch;
 let inst = iMem.req(pc);
 if(!execRedirect.notEmpty)
    begin
      let ppc = nextAddrPredictor(pc);
      pc <= ppc;
      f2d.enq(Fetch2Execute{pc: pc, ppc: ppc,
                            inst: inst, epoch: fEpoch});
    end

  else
    begin
      fEpoch <= !fEpoch;  pc <= execRedirect.first;
      execRedirect.deq;
    end
endrule
```

pass the pc and predicted pc to the execute stage

The Execute rule

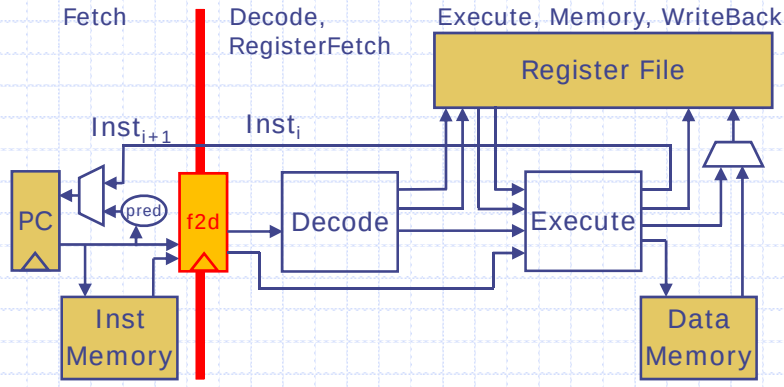
```
rule doExecute;
  let inst  = f2d.first.inst; let pc    = f2d.first.pc;
  let ppc   = f2d.first.ppc;  let inEp = f2d.first.epoch;
  if(inEp == eEpoch) begin
    let dInst = decode(inst);
    let rVal1 = rf.rd1(validRegValue(dInst.src1));
    let rVal2 = rf.rd2(validRegValue(dInst.src2));
    let 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(validRegValue(eInst.dst), eInst.data);
    if(eInst.mispredict) begin
      execRedirect.enq(eInst.addr); eEpoch <= !inEp;
    end
  end
  f2d.deq;
endrule
```

exec returns a flag if there was a fetch misprediction

Can these rules execute concurrently?

Data Hazards

Consider a different two-stage pipeline

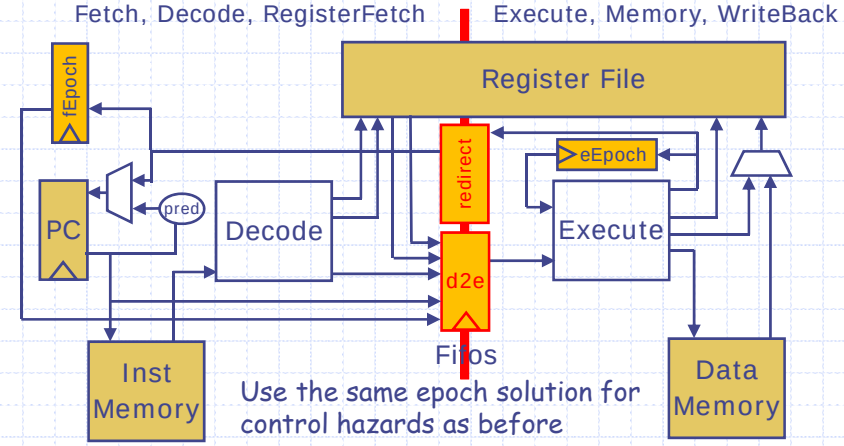


Suppose we move the pipeline stage from Fetch to after Decode and Register fetch

What hazards will the pipeline have? Control?

A different 2-Stage pipeline:

2-Stage-DH pipeline



Use the same epoch solution for control hazards as before

Modify the code for the 2-Stage-CHO pipeline

Type Decode2Execute

The Fetch stage, in addition to fetching the instruction, also decodes the instruction and fetches the operands from the register file. It passes these operands to the Execute stage

```
typedef struct {
    Addr pc; Addr ppc; Bool epoch;
    DecodedInst dInst; Data rVal1; Data rVal2;
} Decode2Execute deriving (Bits, Eq);
```

value instead of register names

2-Stage-DH pipeline

```
module mkProc(Proc);
  Reg#(Addr)        pc <- mkRegU;
  RFile             rf <- mkRFile;
  IMemory         iMem <- mkIMemory;
  DMemory         dMem <- mkDMemory;

  Fifo#(Decode2Execute) d2e <- mkFifo;

  Reg#(Bool)    fEpoch <- mkReg(False);
  Reg#(Bool)    eEpoch <- mkReg(False);
  Fifo#(Addr) execRedirect <- mkFifo;

  rule doFetch …
  rule doExecute …
```


2-Stage-DH pipeline doFetch rule *first attempt*

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

moved from Execute

2-Stage-DH pipeline doExecute rule *first attempt*

```
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) &&
        validValue(eInst.dst).regType == Normal)
        rf.wr(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict) begin
        execRedirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                        end
    d2e.deq;
endrule
```

no change

Data Hazards

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 Add(R1,R2,R3)

I_2 Add(R4,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	

next lecture: Resolving Data Hazards