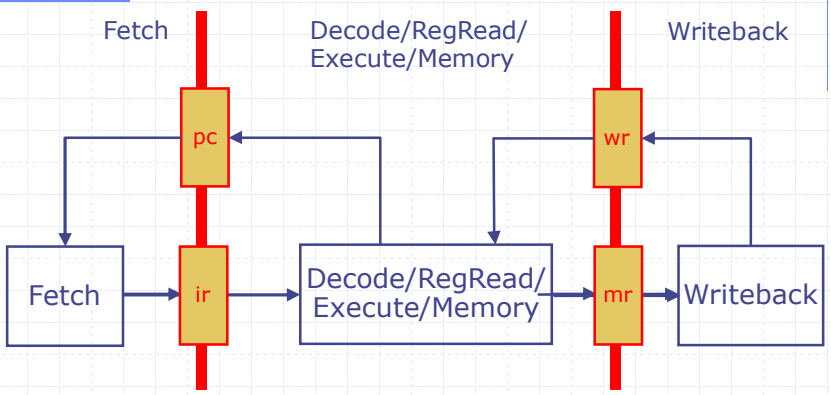
Computer Architecture: A Constructive Approach

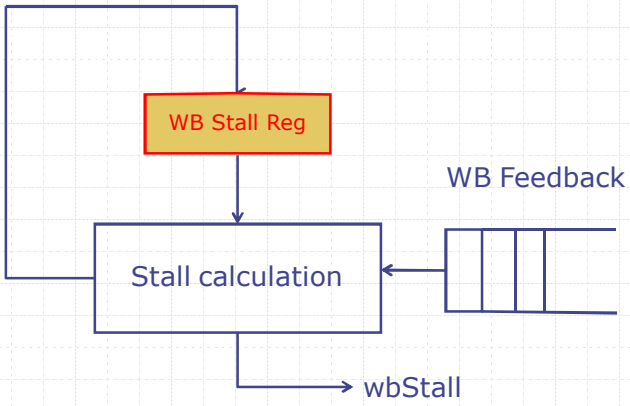
Six Stage Pipeline/Bypassing

Joel Emer
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Three-Cycle SMIPS: *Analysis*



Stall calculation



Data dependence waterfall

From whiteboard

Types of Data Hazards

Consider executing a sequence of instructions like:

$r_k \leftarrow (r_i)\ \text{op}\ (r_j)$

Data-dependence

$r_3 \leftarrow (r_1)\ \text{op}\ (r_2)$
$r_5 \leftarrow (r_3)\ \text{op}\ (r_4)$

Read-after-Write (RAW) hazard

Anti-dependence

$r_3 \leftarrow (r_1)\ \text{op}\ (r_2)$
$r_1 \leftarrow (r_4)\ \text{op}\ (r_5)$

Write-after-Read (WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1)\ \text{op}\ (r_2)$
$r_3 \leftarrow (r_6)\ \text{op}\ (r_7)$

Write-after-Write (WAW) hazard

Detecting Data Hazards

Range and Domain of instruction i

R(i) = Registers (or other storage) modified by instruction i

D(i) = Registers (or other storage) read by instruction i

Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a

RAW hazard if $R(i) \cap D(j) \neq \varnothing$
WAR hazard if $D(i) \cap R(j) \neq \varnothing$
WAW hazard if $R(i) \cap R(j) \neq \varnothing$

Register vs. Memory Data Dependence

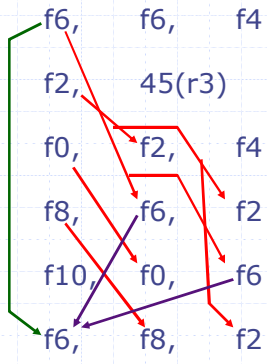
- Data hazards due to register operands can be determined at the decode stage *but*
- Data hazards due to memory operands can be determined only after computing the effective address

store	M[(r1) + disp1] ← (r2)
load	r3 ← M[(r4) + disp2]

Does (r1 + disp1) = (r4 + disp2) ?

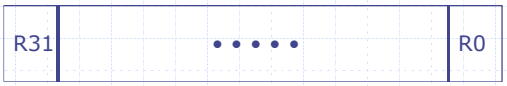
Data Hazards: An Example

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2



RAW Hazards
WAR Hazards
WAW Hazards

Scoreboard



Register#(Bit#(32)) scoreboard <- mkReg(0);

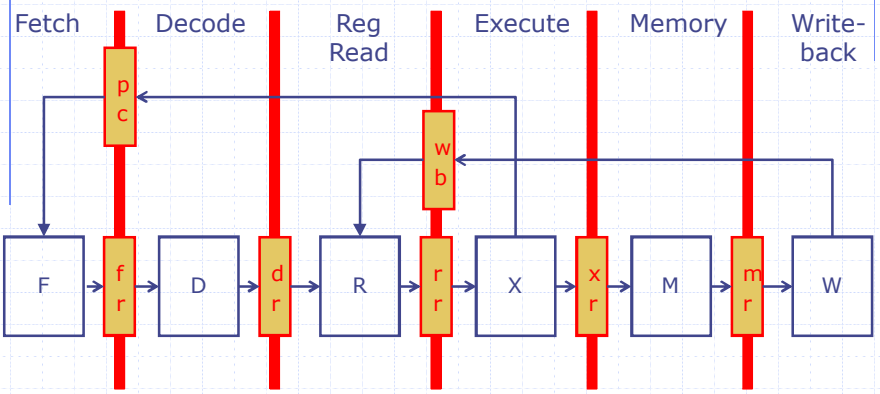
Add a scoreboard of registers in use:

- Set bit for each register to be updated
- Clear bit when a register is updated
- Stall if bit for a register needed is set

SMIPs Pipeline Analysis

	Stage	Tclock >
Six stage		
	Fetch	t_M
	Decode	t_{DEC}
	Register Read	t_{RF}
	Execute	t_{ALU}
	Memory	t_M
	Writeback	t_{WB}

Six Stage Pipeline



Where do we need feedback?

Six-Stage State

```
module mkProc(Proc);
    Reg#(Addr)   fetchPc    <- mkRegU;
    RFile        rf         <- mkRFile;
    Memory       mem        <- mkMemory;

    FIFO#(FBundle)    fr    <- mkFIFO;
    FIFO#(DecBundle)  dr    <- mkFIFO;
    FIFO#(RegBundle)  rr    <- mkFIFO;
    FIFO#(EBundle)    xr    <- mkFIFO;
    FIFO#(WBundle)    mr    <- mkFIFO;

    FIFOF#(Addr)   nextPc   <- mkFIFOF;
    FIFOF#(Rindx)  wbRind   <- mkFIFOF;

    // and internal control state…
```

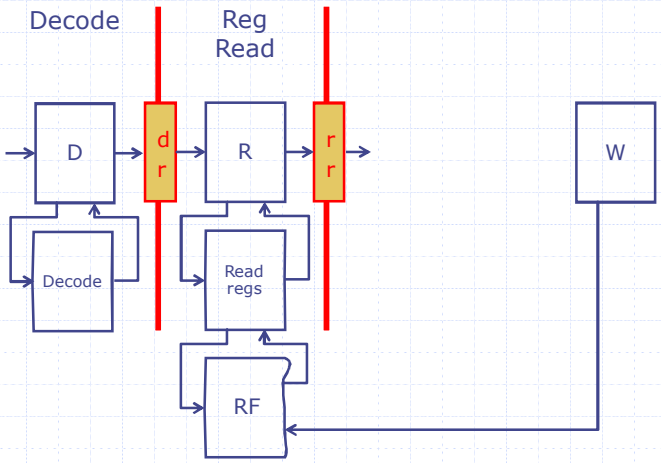
Six-Stage Fetch

```
rule doFetch;
   Addr pc = ?;    Bool epoch = fetchEpoch;

   if (nextPc.notEmpty) begin
      pc = nextPc.first; epoch = !fetchEpoch; nextPc.deq;
   end
   else pc = fetchPc + 4;

   fetchPc <= pc; fetchEpoch <= epoch;

   let instResp <- mem.op(MemReq{op:Ld, addr:pc, data:?});

   fr.enq(FBundle{pc:pc,epoch:epoch,InstResp:instResp});

endrule
```


Six-Stage Decode

```
rule doDecode;
   let fetchInst = fr.first;
   let pcPlus4 = fetchInst.predpc + 4;

   let decInst = decode(fetchInst.instResp, pcPlus4);

   decInst.epoch = fetchInst.epoch;

   dr.enq(decInst);
   fr.deq;

endrule
```

Register Read Stage



RF must be available to ReadSrcs and Writeback!

Register Read Module

```
typedef struct { Data: src1; Data: src2 } Sources;

module mkRegRead#(RFile rf)(RegRead);
   Reg#(Bool)   scoreboardReg <- mkReg(False);
   Dwire#(Bool) scoreboard <-mkDwire(scoreboardReg);

   rule updateScoreboard;
      scoreboardReg <= scoreboard;
   endrule

   method Maybe#(Sources) readRegs(DecodeInst decInst);
      let src1 = rf.rd1(decInst.op1); let src2 = …

      if (scoreboardReg) return tagged Invalid;
      else return tagged Valid Sources{src1:src1,…};
   endmethod
```

Register Read Module

```
typedef struct { Data: src1; Data: src2 } Sources;

module mkRegRead#(RFile rf)(RegRead);
   Reg#(Bool)   scoreboardReg <- mkReg(False);

   method Maybe#(Sources) readRegs(DecodeInst decInst);
      let src1 = rf.rd1(decInst.op1); let src2 = …

      if (scoreboardReg) return tagged Invalid;
      else return tagged Valid Sources{src1:src1,…};
   endmethod
```

To use need to instantiate with:

let readreg <- mkReadReg(rf);

Six-Stage Register Read (cont)

```
   RWire#(Bool) next_available <-mkRWire;
   RWire#(Bool) next_unavailable <- mkRWire;

   method markAvailable();
     next_available.wset(False);
   endmethod
   method markUnavailable();
     next_unavailable.wset(True);
   endmethod

   rule updateScoreboard;
      if (next_unavailable matches tagged Valid .ua)
         scoreboardReg <= True;
       else if (next_available matches tagged Valid .a)
           scoreboardReg <= False;
   endrule
endmodule
```

Six-Stage Register Read

```
rule doRegRead;
   if (wbRind.notEmpty) begin
      readreg.markAvailable(); wbRind.deq; end
   let decInst = dr.first();
   if (execEpoch != decInst.epoch) begin
      readreg.markAvailable(); dr.deq() end;
   else begin
      maybeSources = readregs.readRegs(decInst);
      if (maybeSources match tagged Valid .sources) begin
         if (writesreg(decInst)) rr.markUnavailable();
            rr.enq(RegBundle{decodebundle: decInst,
                   src1: srouces.src1, src2: sources.src2});
         dr.deq();
      end
   end
endrule
```

Better scoreboard will need register index!

Six-Stage Execute

```
rule doExecute;
   let decInst = rr.first.decodebundle;
   let epochChange = (decInst.epoch != execEpoch);
   let src1 = rr.first.src1; let src2 = ...

   if (! epochChange) begin
      let execInst = exec.exec(decInst, src1, src2);
      if (execInst.cond) begin
         nextPc.enq(execInst.addr);
         execEpoch <= !execEpoch;
      end
      xr.enq(execInst);
   end
   rr.deq();
endrule
```

Six-Stage Memory

```
rule doMemory;
  let execInst = xr.first;

  if (execInst.itype==Ld || execInst.itype==St) begin
     execInst.data <- mem(MemReq{
       op:execInst.itype==Ld ? Ld : St,
       addr:execInst.addr,
       data:execInst.data});
  end

  mr.enq(WBBundle{itype:execInst.itype,
                  rdst:execInst.rdst,
                  data:execInst.data});
  xr.deq();
endrule
```

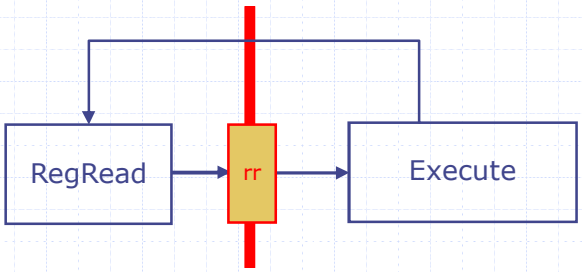

Six-Stage Writeback

```
rule doWriteBack;

      wbRind.enq(mr.first.rdst);

      rf.wr(mr.first.rdst, mr.first.data);

      mr.deq;
endrule
```

Six Stage Waterfall

From whiteboard

Bypass



What does RegRead need to do?

Bypass Network

```
typedef  struct { Rindx regnum; Data  value;} BypassValue;

Module mkBypass(BypassNetwork)
      Rwire#(BypassValue) bypass;

      method produceBypass(Rindx regnum, Data value);
        bypass = BypassValue{regname: regnum, value:value};
      endmethod

      method Maybe#(Data) consumeBypass(Rindx regnum);
        if (bypass matches tagged Valid .b && b.regnum == regnum)
            return tagged Valid b.value;
        else
            return tagged Invalid;
      endmethod
endmodule
```

Real network will have many sources. How are they ordered?

Register Read (with bypass)

```
module mkRegRead#(RFile rf, BypassNetwork bypass)(RegRead);
   method Maybe#(Sources) readRegs(DecodeInst decInst);
      let src1 = rf.rd1(decInst.op1); let src2 = …
      if (!scoreboardReg)
         return tagged Valid Sources{src1:src1,…};
      else
      begin
         let b1 = bypass.consumebypass(decInst.op1);
         let b2 =
         if (b1 matches tagged Valid .v1 && b2 matches…)
          return tagged Valid Sources{src1:v1.value …);
         else
          return tagged Invalid;
       end
    endmethod
```