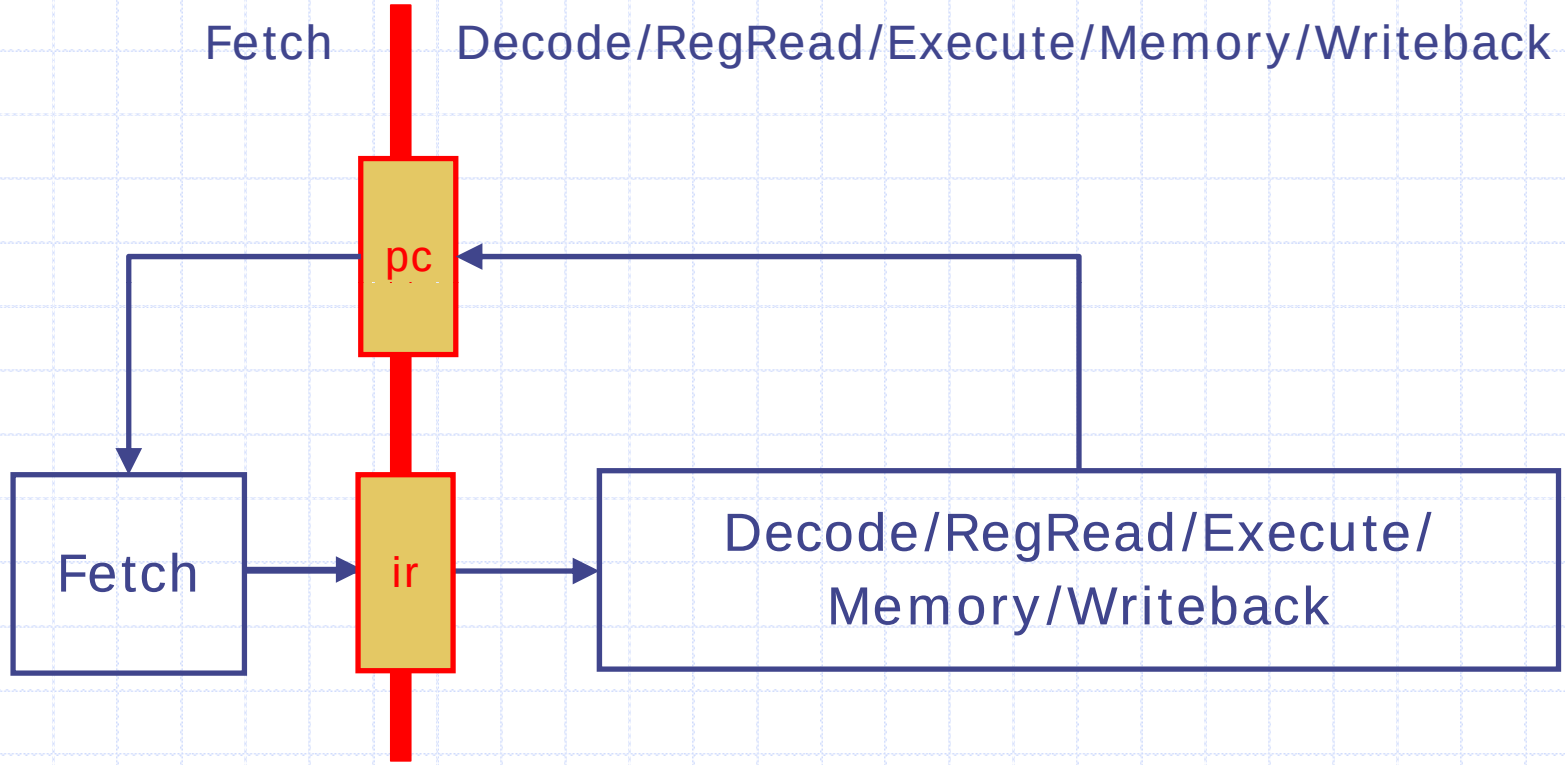
Computer Architecture: A Constructive Approach

Data Dependencies

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

Two-Cycle SMIPS: *Analysis*



Does Fetch depend on the DRXMW stage?

Yes, for next PC

FIFO queues

FIFO queues are useful for communication between distinct parallel activities:

- Methods
  - enq(data) – add data to tail of queue
    - Not ready if queue is full
  - first() – view element at head of queue.
    - Not ready if queue is empty
  - deq() – remove element at head of queue
    - Not ready if queue is empty
  - notEmpty() and notFull() (for FIFOF)
    - Always ready

Representing parallelism

- Bluespec rules represent possible concurrent activity.
  - Any two rules can fire simultaneously if there are no structural “conflicts”. So they can be useful for the individual stages of a pipeline.
  - But Bluespec does not watch for data hazards. It will use whatever data is available. You have to handle data hazards in your code.

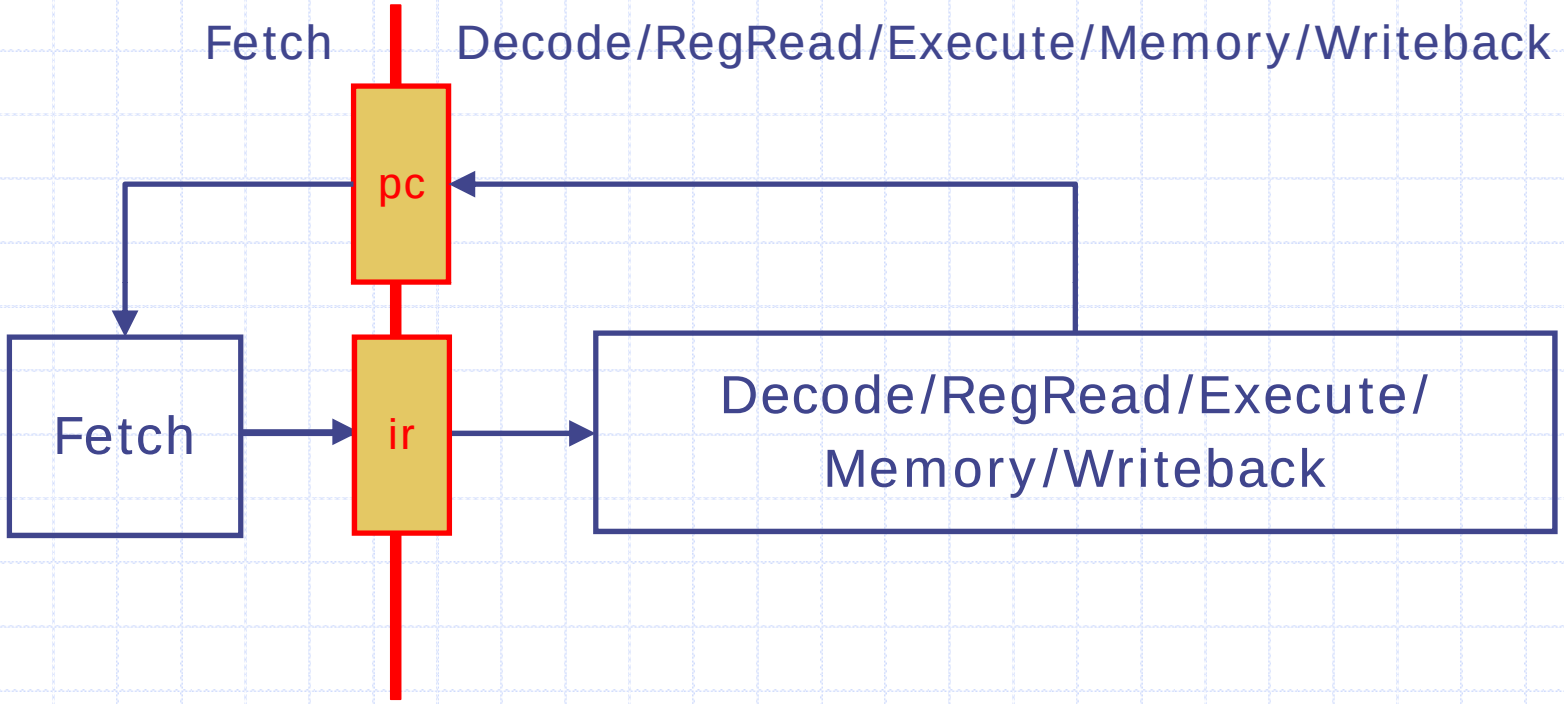
Two-Stage SMIPS (stall) - 1

```
module mkProc(Proc);
    RFile              rf     <- mkRFile;
    Memory             mem    <- mkMemory;
    FIFO#(FBundle)     ir     <- mkFIFO;
    FIFOF#(Addr)       nextPc <- mkFIFOF;

  rule doFetch;
    Addr pc = nextPc.first();
    nextPc.deq();

    let instResp <- mem.side(MemReq{op:Ld,
                                    addr:pc, data:?});
    ir.enq(FBundle{predpc:pc, epoch:epoch,
                   instResp:instResp});
  endrule
```

Two-Stage SMIPS (stall) - 2

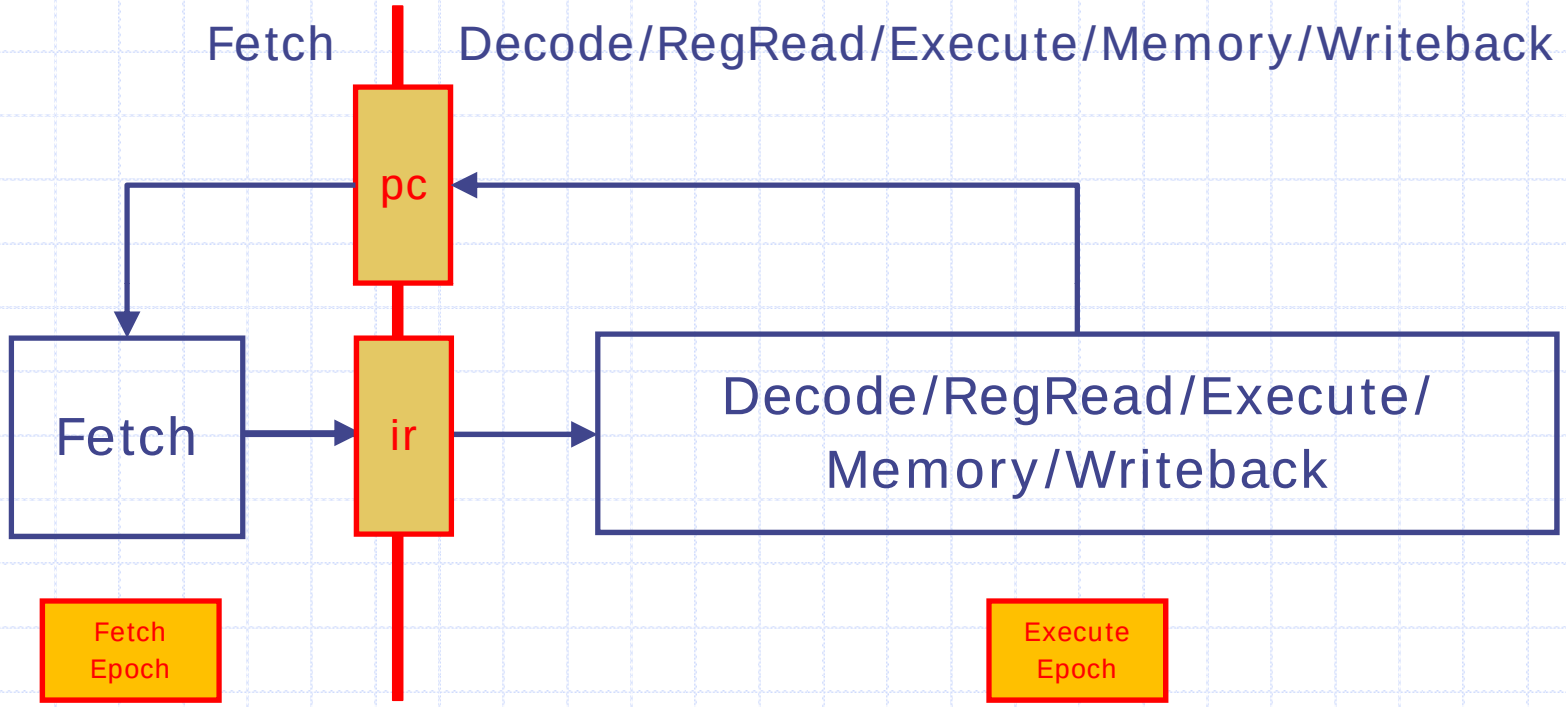
```
rule doDecodeExecuteWb;
     let pcPlus4 = ir.first.predpc + 4;
     let decInst = decode(ir.first.instResp, pcPlus4);
     Data src1 = rf.rd1(decInst.op1);
     Data src2 = rf.rd2(decInst.op2);
     Instr inst = unpack(ir.first.instResp);
     let execInst = exec.exec(decInst, src1, src2);
     nextPc.enq(exec.cond? execInst.addr : pcPlus4);
     if(execInst.itype==Ld || execInst.itype==St) begin
        execInst.data <- mem.side(MemReq{op:execInst.itype==Ld ?
             Ld : St, addr:execInst.addr, data:execInst.data});
     end
     if((execInst.itype == Alu || execInst.itype == Ld) &&
execInst.rdst != 0)
        rf.wr(execInst.rdst, execInst.data);
     ir.deq;
endrule
```

Two-Cycle SMIPS: *Analysis*



Let us assume we keep fetching instructions from pc, pc+4, pc+8, ... and only correct it when wrong.

Two-Cycle SMIPS: *Analysis*



Lets treat a sequence of instructions as an epoch and change epoch on branch - dropping instructions in execute until we see new epoch

Is a single epoch register sufficient?

No

Two-Stage SMIPS (Speculate) - 1

```
module mkProc(Proc);

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

    FIFO#(FBundle)      ir <- mkFIFO;
    FIFOF#(Addr)    nextPc <- mkFIFOF;

    Reg#(Bool)  fetchEpoch <- mkReg(False);
    Reg#(Bool)   execEpoch <- mkReg(True);
```

Two-Stage SMIPS (Speculate) - 2

```
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.side(MemReq{op:Ld, addr:pc, data:?});
   ir.enq(FBundle{predpc:pc, epoch:epoch, instResp:instResp});
endrule
```

Two-Stage SMIPS (Speculate) - 3

```
rule doDecodeExecuteWb;
   if(ir.first.epoch==execEpoch)
   begin
     let pcPlus4 = ir.first.predpc + 4;
     let decInst = decode(ir.first.instResp, pcPlus4);
     Data src1 = rf.rd1(decInst.op1);
     Data src2 = rf.rd2(decInst.op2);
     Instr inst = unpack(ir.first.instResp);
     let execInst = exec.exec(decInst, src1, src2);
      if(execInst.cond)
      begin
         nextPc.enq(execInst.addr);
         execEpoch <= !execEpoch;
      end
```

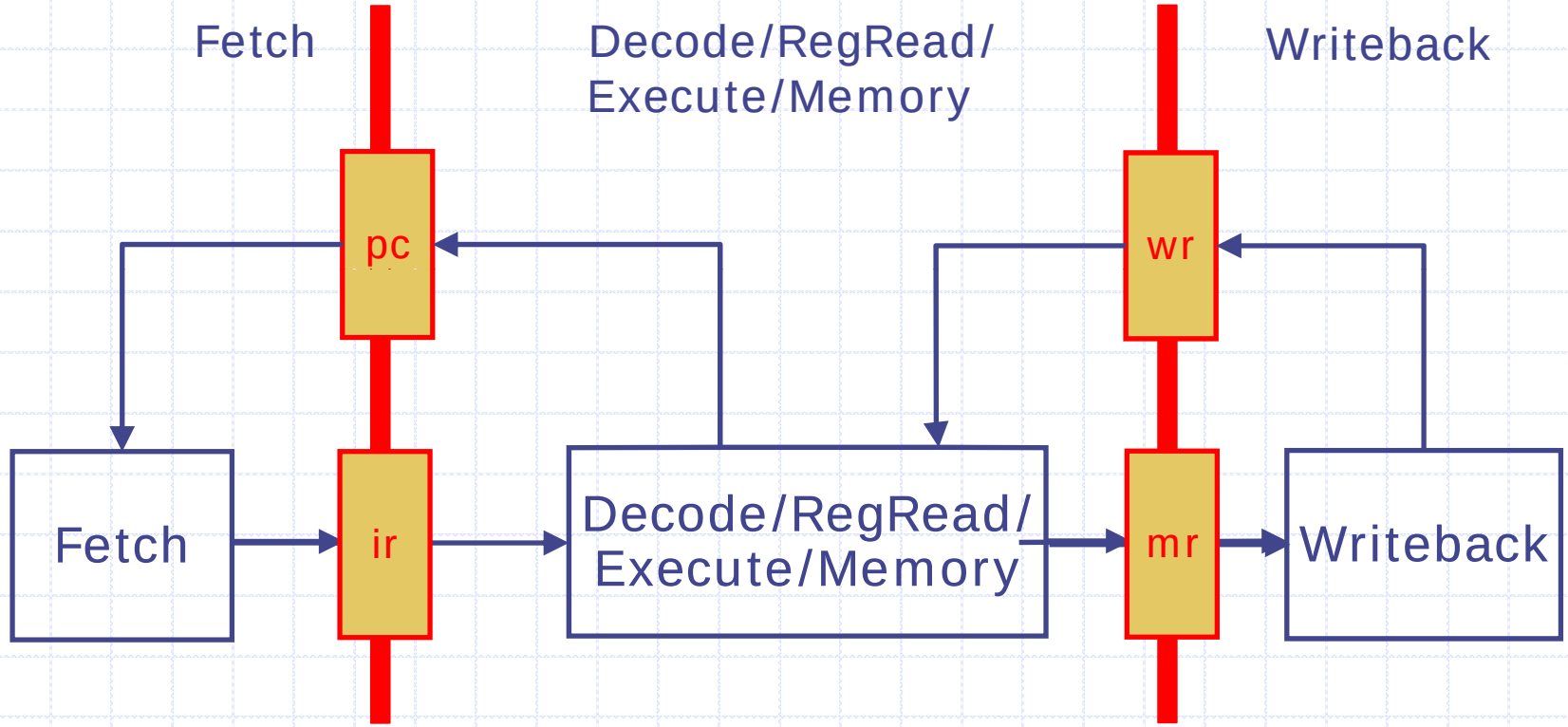
Two-Stage SMIPS (Speculate) - 4

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

        if((execInst.itype == Alu || execInst.itype == Ld) &&
                        execInst.rdst != 0)
            rf.wr(wbr.first.rdst, wbr.first.data);
    end

    ir.deq;
endrule

endmodule
```

SMIPs Pipeline Analysis

	Stage	Tclock >
Two stage		
	Fetch	t_M
	DRXMW	$t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB}$
Three Stage		
	Fetch	t_M
	DRXM	$t_{DEC} + t_{RF} + t_{ALU} + t_M$
	Writeback	t_{WB}

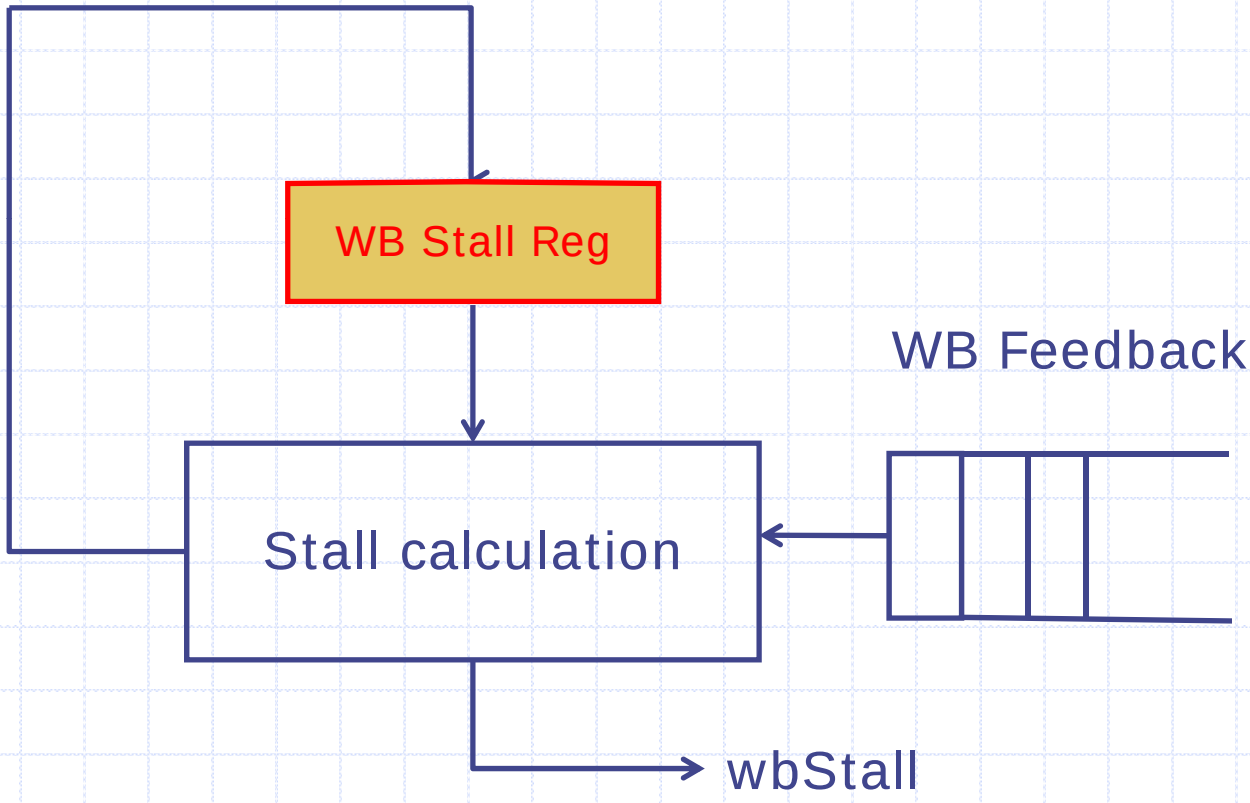
Three-Cycle SMIPS: *Analysis*



Does Fetch or DRXM depend on Writeback?

Yes, for register values

Stall calculation



3-Stage SMIPS - 1

```
module mkProc(Proc);

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

    FIFO#(FBundle)      ir <- mkFIFO;
    FIFOF#(Addr)    nextPc <- mkFIFOF;

    Reg#(Bool)  fetchEpoch <- mkReg(False);
    Reg#(Bool)   execEpoch <- mkReg(True);

    FIFO#(WBRindx) wbRindx <- mkFIFOF;
    Reg#(Bool)  wbStallReg <- mkReg;
```

3-Stage SMIPS - 2

```
rule doDecodeExecute;

    let wbStall = wbStallReg;

    if(wbRind.notEmpty)
    begin
      wbRind.deq;
      wbStall = False;
    end
    if(ir.first.epoch==execEpoch)
    begin
      let pcPlus4 = ir.first.predpc + 4;
      let decInst = decode(fr.first.instResp, pcPlus4);
```

3-Stage SMIPS - 3

```
Bool stall = wbStall;

if(!stall) begin
    Data src1 = rf.rd1(decInst.op1);
    Data src2 = rf.rd2(decInst.op2);
   Instr inst = unpack(ir.first.instResp);
   let execInst = exec.exec(decInst, src1, src2);
    if(execInst.itype==Ld || execInst.itype==St) begin
       execInst.data <- mem.side(MemReq{op:execInst.itype==Ld ?
          Ld : St, addr:execInst.addr, data:execInst.data});
    end
    if(execInst.cond)
    begin
      nextPc.enq(execInst.addr);
      execEpoch <= !execEpoch;
    end
```

3-Stage SMIPS - 4

```
            if((execInst.itype == Alu || execInst.itype == Ld) &&
   execInst.rdst != 0)
            begin
              wbr.enq(WBBundle{itype:execInst.itype, rdst:execInst.rdst
   data:execInst.data});
              wbStall = True;
            end
            ir.deq;
         end
         // ir is not dequeued if we are stalling
      end
      else
         ir.deq;

   wbStallReg <= wbStall;
endrule
```

3-Stage SMIPS - 5

```
rule doWriteBack;

        wbRind.enq(wbr.first.rdst);

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

        wbr.deq;
endrule
```

Types of Data Hazards

Consider executing a sequence of instructions like:

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

Data-dependence

$r_3 \leftarrow (r_1)$ op (r_2)
$r_5 \leftarrow (r_3)$ op (r_4)

Read-after-Write (RAW) hazard

Anti-dependence

$r_3 \leftarrow (r_1)$ op (r_2)
$r_1 \leftarrow (r_4)$ op (r_5)

Write-after-Read (WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1)$ op (r_2)
$r_3 \leftarrow (r_6)$ 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	$R(i) \cap D(j) \neq \emptyset$
WAR hazard if	$D(i) \cap R(j) \neq \emptyset$
WAW hazard if	$R(i) \cap R(j) \neq \emptyset$

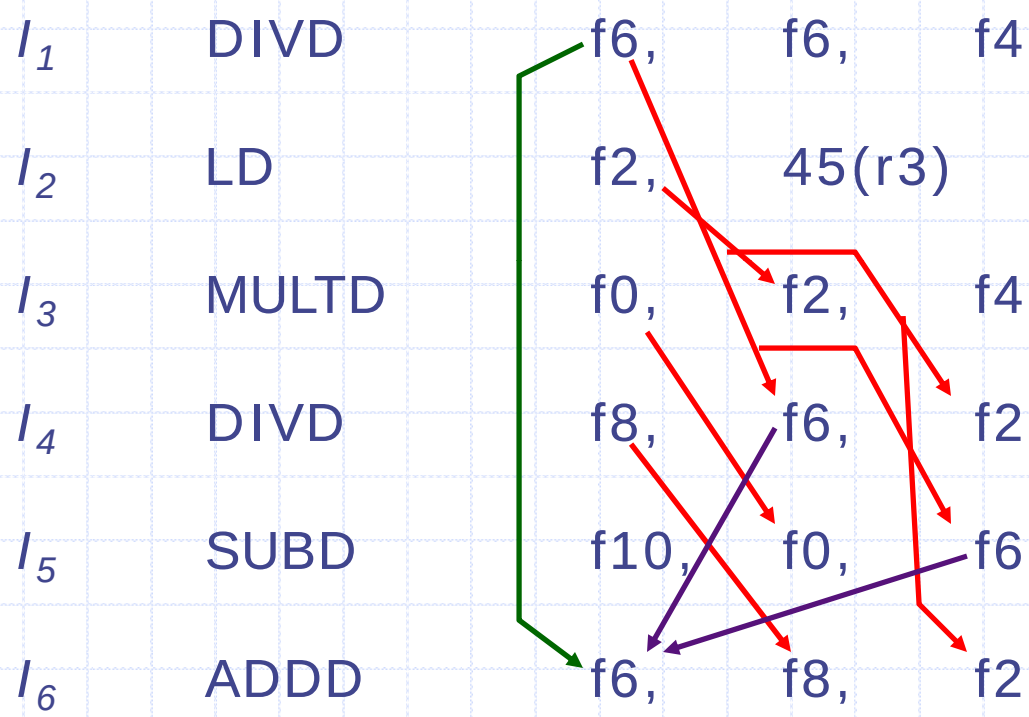
Register vs. Memory Data Dependence

- 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



RAW Hazards

WAR Hazards

WAW Hazards