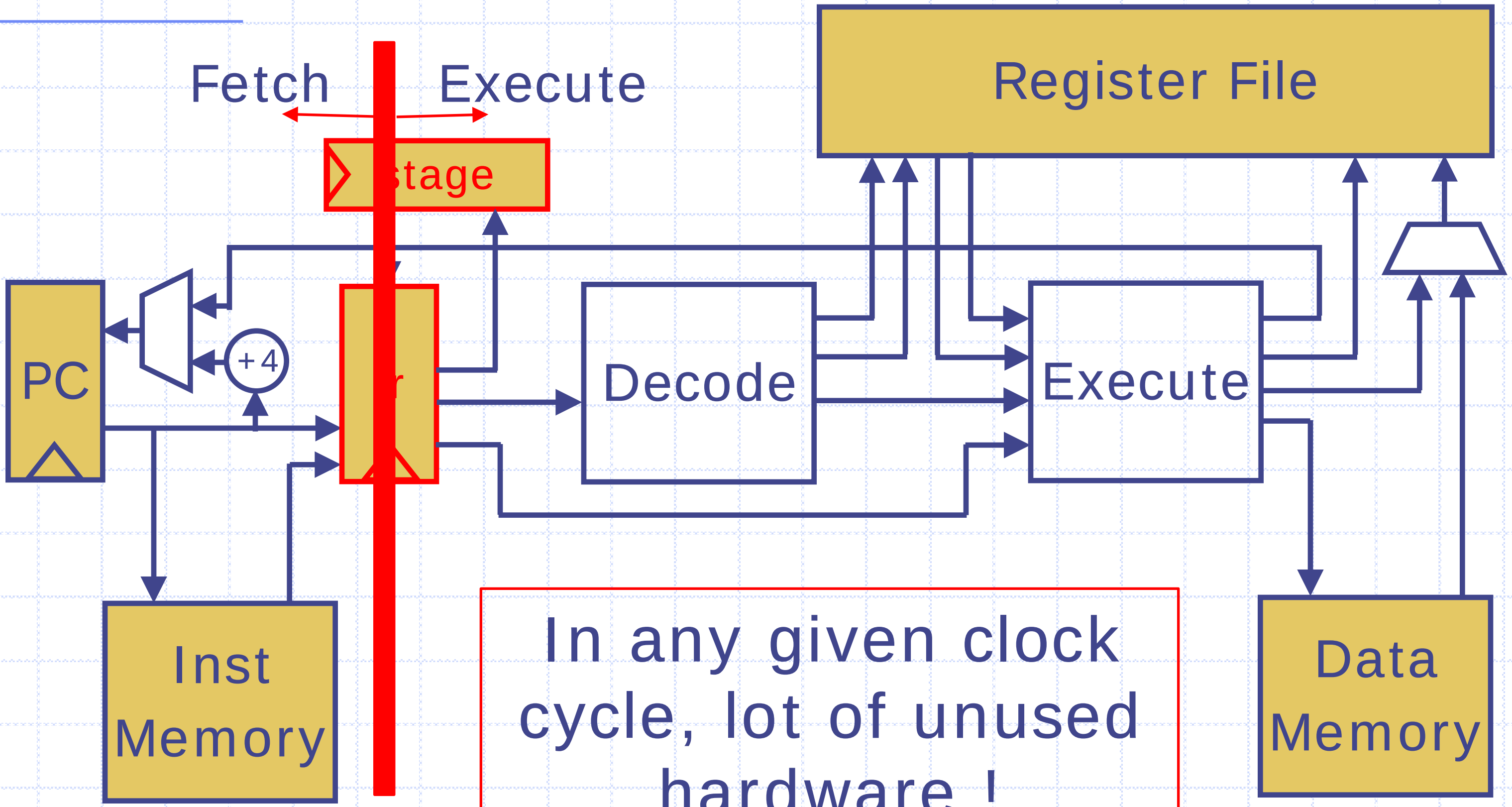
Computer Architecture: A Constructive Approach

Pipelined Implementations of SMIPS

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

Two-Cycle SMIPS: *Analysis*

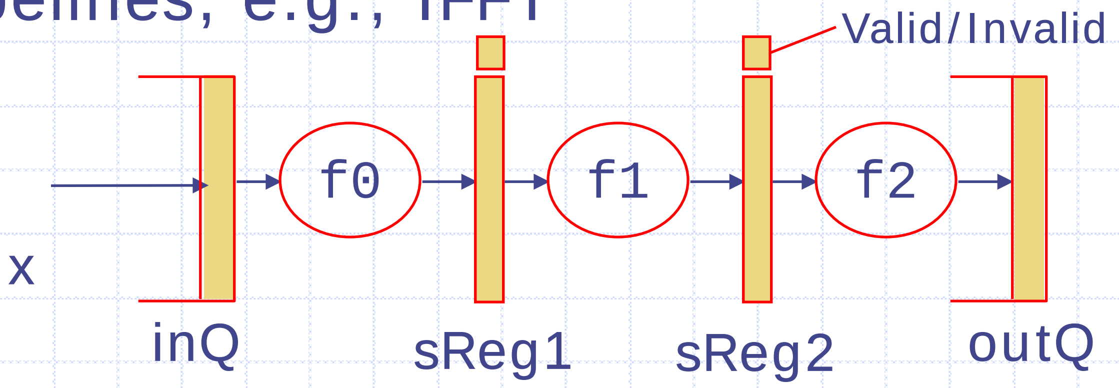


In any given clock cycle, lot of unused hardware !

Pipelined execution of instructions to increase the through put

Instruction pipelining

- Much more complicated than arithmetic pipelines, e.g., IFFT



- The entities in an instruction pipeline are not independent of each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls

Hazards in instruction pipelining

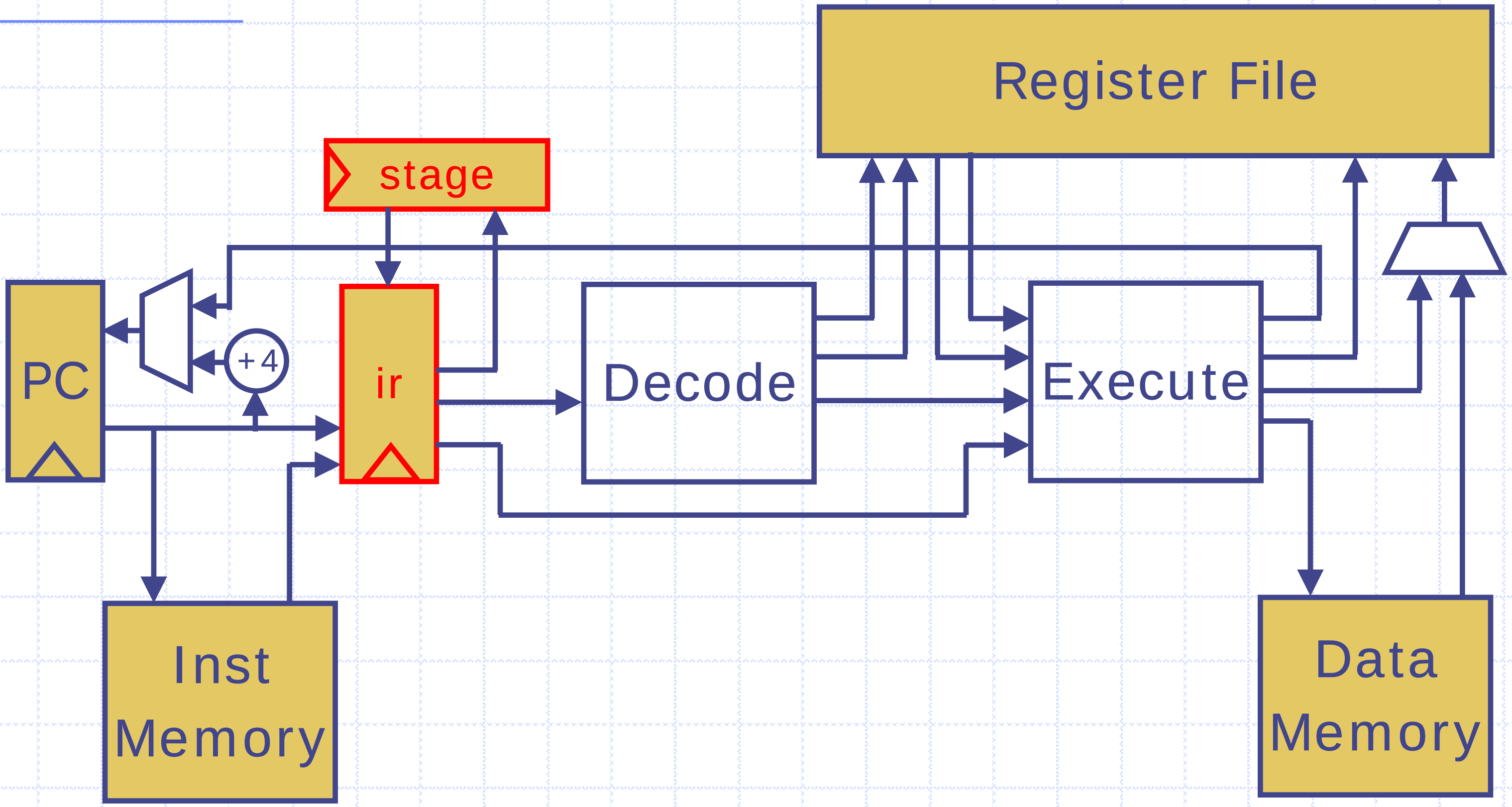
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory
- *Data hazard:* An instruction in the pipeline may affect the state of the machine (rf, dMem or pc) – the next instruction must be fully cognizant of this change
- *Control hazard:* An instruction in the pipeline may determine the next instruction to be executed, e.g., branches

Notice that none of these hazards are present in the IFFT pipeline.

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

$\Rightarrow$ must deal with hazard

Two-Cycle SMIPS

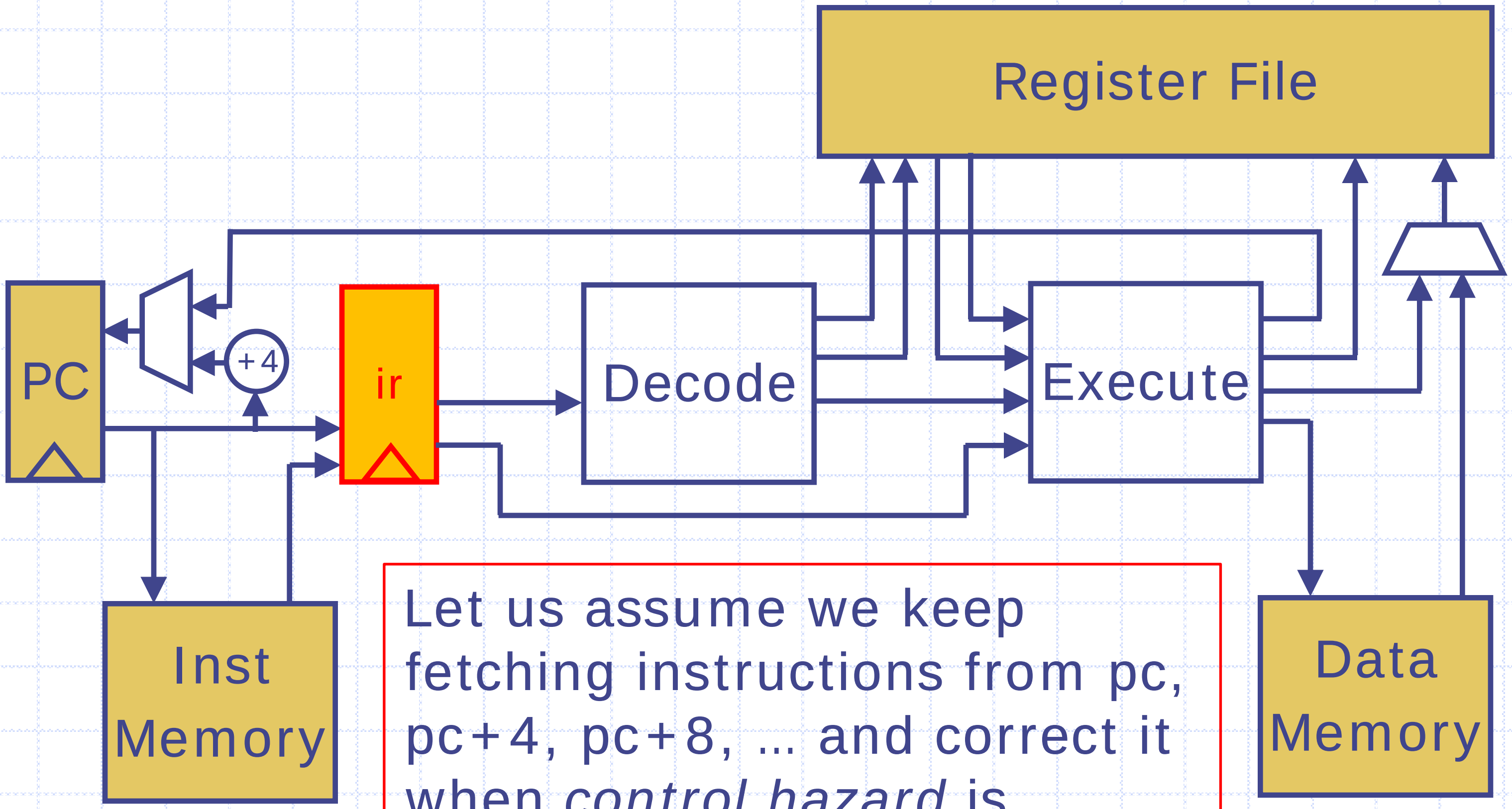


Register ir is to hold a fetched instruction and register stage is to remember which stage (fetch/execute) we are in

ir: The instruction register

- You may recall from our earlier discussion of pipelining (e.g., IFFT) that there is a possibility that the intermediate or pipelined registers do not contain any meaningful data
  - We can associate a (Valid/Invalid) bit to ir
  - Equivalently we can think of a pipeline register as a one-element FIFO

Two-stage SMIPS (Harvard)



Let us assume we keep fetching instructions from pc, pc+4, pc+8, ... and correct it when *control hazard* is detected

Two-stage pipeline SMIPS (Harvard) – *first attempt*

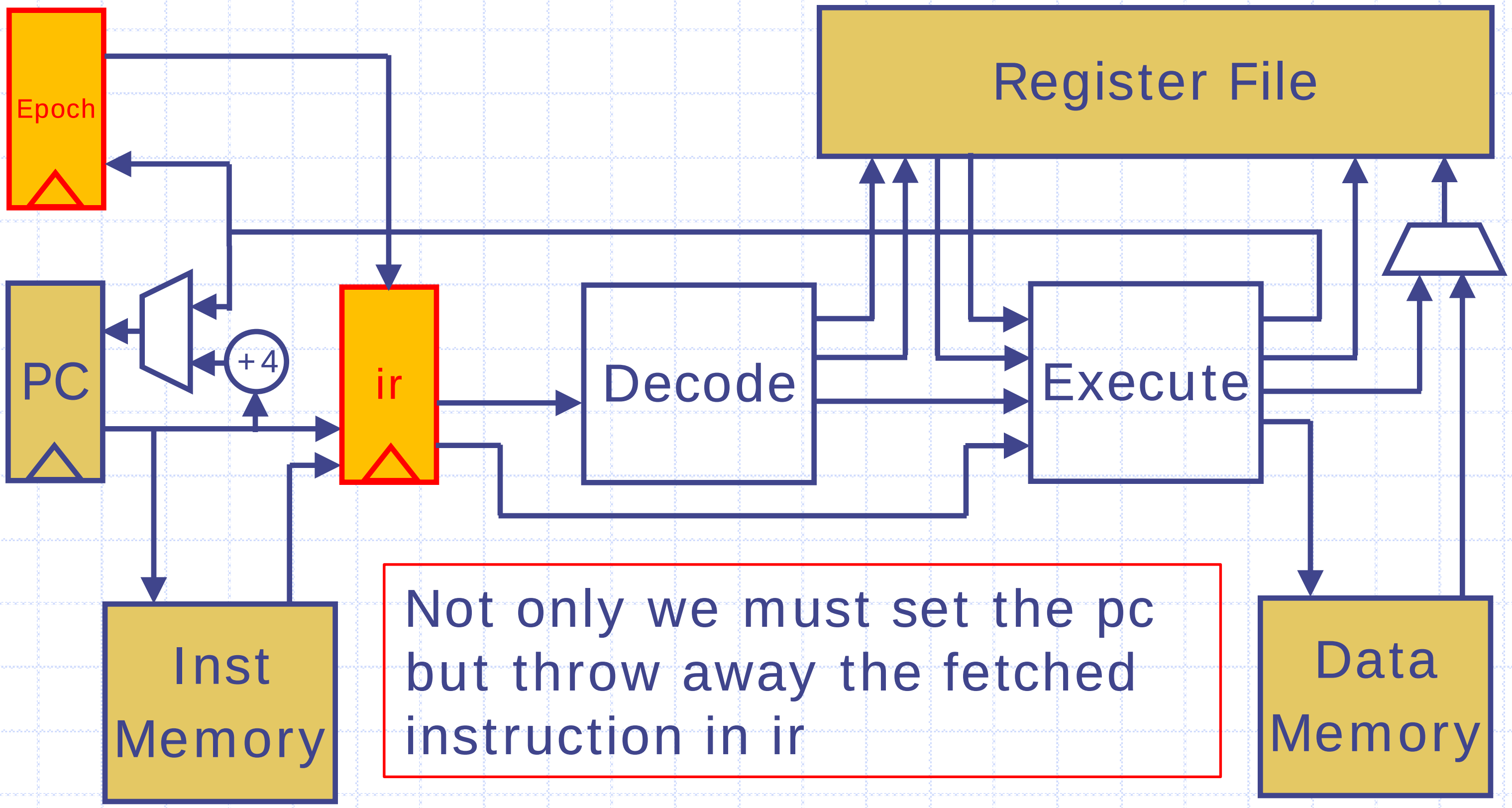
```
module mkProc(Proc);
  Reg#(Addr)              pc <- mkRegU;
  RFile                   rf <- mkRFile;
  Memory                 mem <- mkTwoPortedMemory;
  let iMem = mem.iport;  let dMem = mem.dport;
  PipeReg#(TypeFetch2Execute) ir <- mkPipeReg;

  rule doProc;
    let inst <- iMem(MemReq{op: Ld, addr: pc,
                                           data: ?});
    ir.enq(TypeFetch2Execute{pc: pc, inst: inst});

    let next_pc = pc + 4;
```

Two-stage pipeline SMIPS (Harvard) – *first attempt*

```
if(ir.notEmpty) begin
  let irpc = ir.first.pc;
  let inst = ir.first.inst;
  let dInst = decode(inst);
  Data rVal1 = rf.rd1(dInst.rSrc1);
  Data rVal2 = rf.rd2(dInst.rSrc2);
  let eInst = exec(dInst, rVal1, rVal2, irpc);
  if(memType(eInst.iType))
    eInst.data <- dMem(MemReq{
               op: eInst.iType==Ld ? Ld : St,
               addr: eInst.addr, data: eInst.data});
  if(regWriteType(eInst.iType))
      rf.wr(eInst.rDst, eInst.data);
  if (eInst.brTaken) next_pc = eInst.addr;
  ir.deq; end
  pc <= next_pc;
```

oops!

Control Hazard



Not only we must set the pc but throw away the fetched instruction in ir

A new Epoch register can keep track of whether the instruction in ir is from the current epoch

Two-Stage SMIPS (unpipelined) - 1

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

   rule doFetch;
      Addr pc = nextPc.first;
      nextPc.deq;
      let instResp <- mem.side(MemReq{op:Ld,
                                      addr:pc, data:?});
      fr.enq(FBundle{predpc:pc, epoch:epoch,
                     instResp:instResp});
   endrule
```

Two-Stage SMIPS (Unpipelined) - 2

```
rule doDecodeExecuteWb;
      let pcPlus4 = fr.first.predpc + 4;
      let decInst = decode(fr.first.instResp, pcPlus4);
      Data src1 = rf.rd1(decInst.op1);
      Data src2 = rf.rd2(decInst.op2);
      Instr inst = unpack(fr.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
      nextPc.enq(exec.con? execInst.addr : pcPlus4);
      if((execInst.itype == Alu || execInst.itype == Ld) &&
execInst.rdst != 0)
          rf.wr(execInst.rdst, execInst.data);
        fr.deq;
endrule
```

Two-Stage SMIPS (Pipelined) - 1

```
module mkProc(Proc);
    Reg#(Bool)  fetchEpoch <- mkReg(False);
    Reg#(Bool)   execEpoch <- mkReg(True);
    Reg#(Addr)  fetchPc    <- mkRegU;
    RFile       rf    <- mkRFile;
    Memory     mem    <- mkMemory;
    FIFO#(FBundle)   fr  <- mkFIFO;
    FIFOF#(Addr) nextPc <- mkFIFOF;
    let exec <- mkSMIPSExecutionCombinational;
```

Two-Stage SMIPS (Pipelined) - 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:?});
   fr.enq(FBundle{predpc:pc, epoch:epoch, instResp:instResp});
endrule
```

Two-Stage SMIPS (Pipelined) - 3

```
rule doDecodeExecuteWb;
   if(fr.first.epoch==execEpoch)
   begin
     let pcPlus4 = fr.first.predpc + 4;
     let decInst = decode(fr.first.instResp, pcPlus4);
     Data src1 = rf.rd1(decInst.op1);
     Data src2 = rf.rd2(decInst.op2);
     Instr inst = unpack(fr.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
```

Two-Stage SMIPS (Pipelined) - 4

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

3-Stage SMIPS (Pipelined) - 1

```
rule doDecodeExecuteWb;
     let wbStall = wbStallReg;
     if(wbRind.notEmpty)
     begin
       wbRind.deq;
       wbStall = False;
     end
     if(fr.first.epoch==execEpoch)
     begin
       let pcPlus4 = fr.first.predpc + 4;
       let decInst = decode(fr.first.instResp, pcPlus4, cp0_statsEn,
  cp0_fromhost, cp0_tohost);
       Bool stall = wbStall;
```

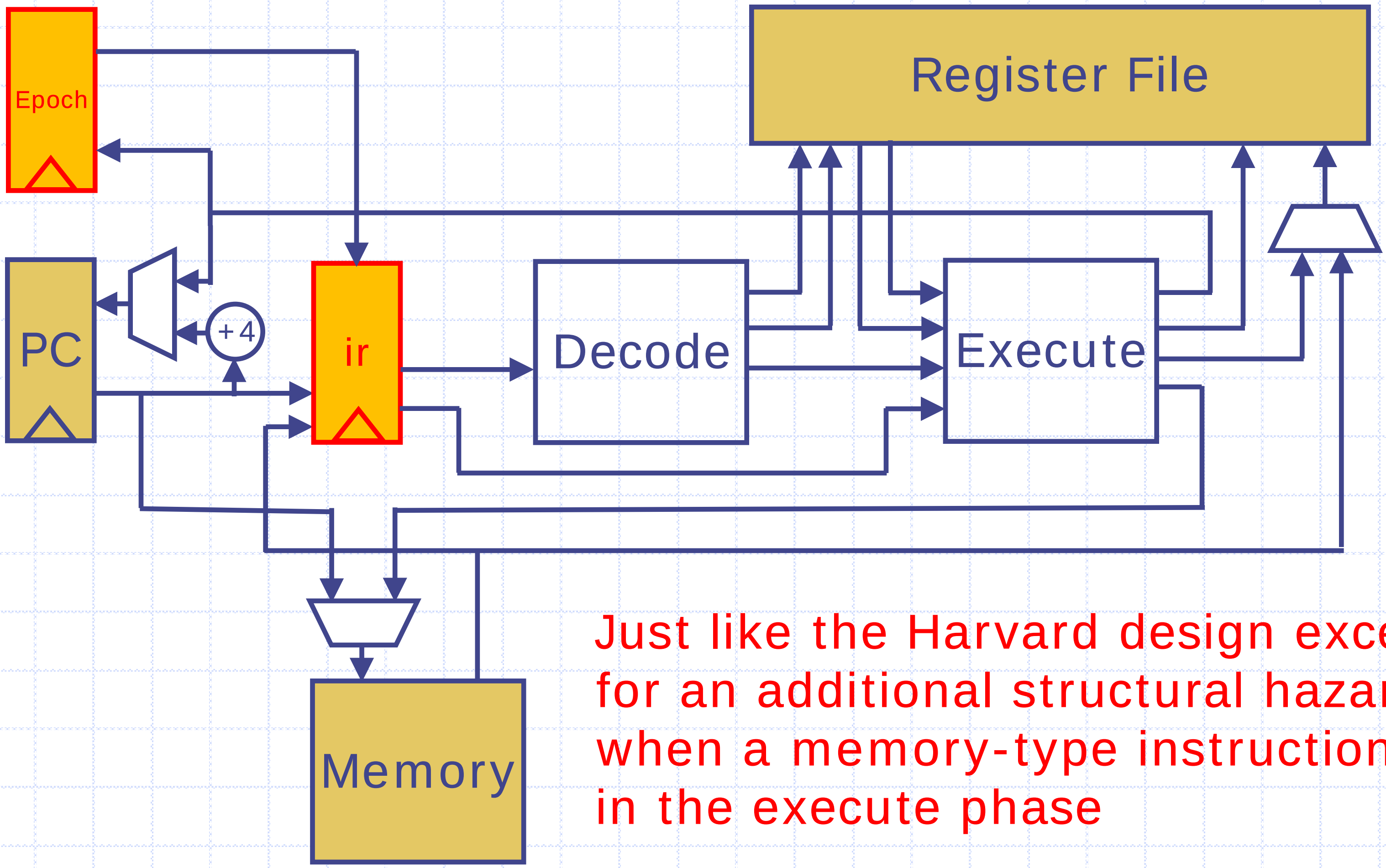
3-Stage SMIPS (Pipelined) - 2

```
if(!stall) begin
   Data src1 = rf.rd1(decInst.op1);
   Data src2 = rf.rd2(decInst.op2);
   Instr inst = unpack(fr.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 (Pipelined) - 3

```
                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
             fr.deq;
          end
       end
       else
          fr.deq;
   wbStallReg <= wbStall;
endrule
```

Two-stage Pipelined SMIPS Princeton Architecture



Just like the Harvard design except for an additional structural hazard when a memory-type instruction is in the execute phase

Pipelined SMIPS (Princeton)

```
module mkProc(Proc);
  Reg#(Addr)          pc <- mkRegU;
  Reg#(Bool)       epoch <- mkReg(True);
  RFile               rf <- mkRFile;
  Memory             mem <- mkOnePortedMemory;
  let uMem = mem.port;
  PipeReg#(FBundle) ir <- mkPipeReg;
  Wire#(Bool)    dAcc <- mkDWire(False);
  rule doProc;
    let next_pc = pc;
    if(!dAcc) begin
      let inst <- uMem(MemReq{op:Ld, addr:pc, data:?});
      ir.enq(TypeFetch2Execute{pc:pc, epoch:epoch,inst:
 inst});
      next_pc = pc + 4;
    end
```

Pipelined SMIPS (Princeton)

```
if(ir.notEmpty) begin
    let irpc  = ir.first.pc;
    let inst  = ir.first.inst;
    if(ir.first.epoch==epoch) begin
      let dInst = decode(inst);
      Data rVal1 = rf.rd1(dInst.rSrc1);
      Data rVal2 = rf.rd2(dInst.rSrc2);
      let eInst = exec(dInst, rVal1, rVal2, irpc);

      if(memType(eInst.iType)) begin
        eInst.data <- uMem(MemReq{
                op: eInst.iType==Ld ? Ld : St,
                addr: eInst.addr, data: eInst.data});
        dAcc = True;
      end
```

Pipelined SMIPS (Princeton)

```
    if(regWriteType(eInst.iType))
            rf.wr(eInst.rDst, eInst.data);
    if(eInst.brTaken) begin
            next_pc = eInst.addr;
            epoch = ! epoch;
          end
end
      ir.deq;
    end

    pc <= next_pc;
  endrule
endmodule
```

next time -- Data Hazards