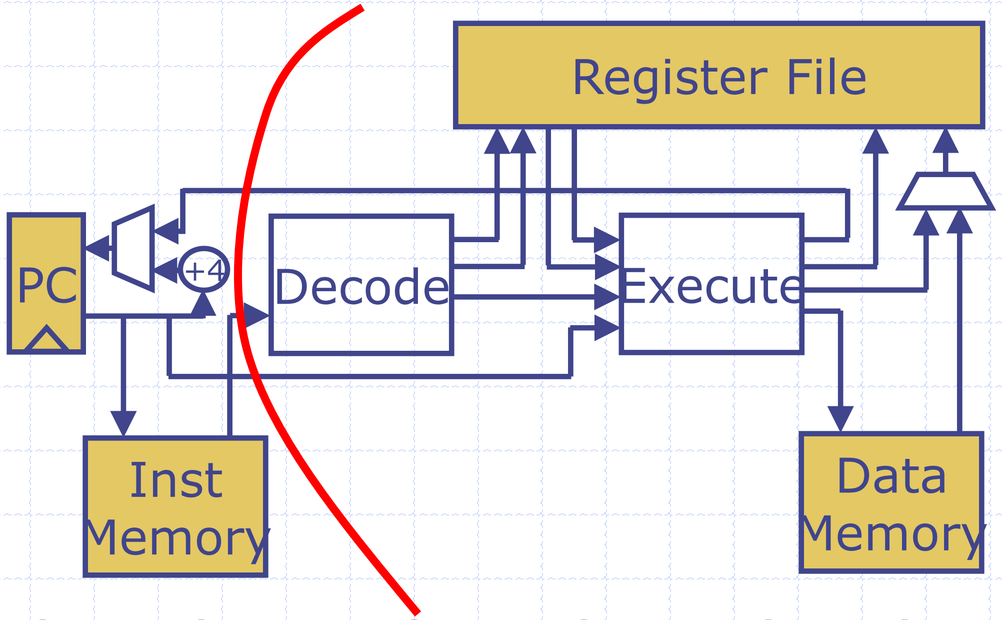
Constructive Computer Architecture:

Pipelined Processors

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

Single-Cycle SMIPS: *Clock Speed*



$t_{Clock} > \quad t_M + t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB}$

We can improve the clock speed if we execute each instruction in two clock cycles

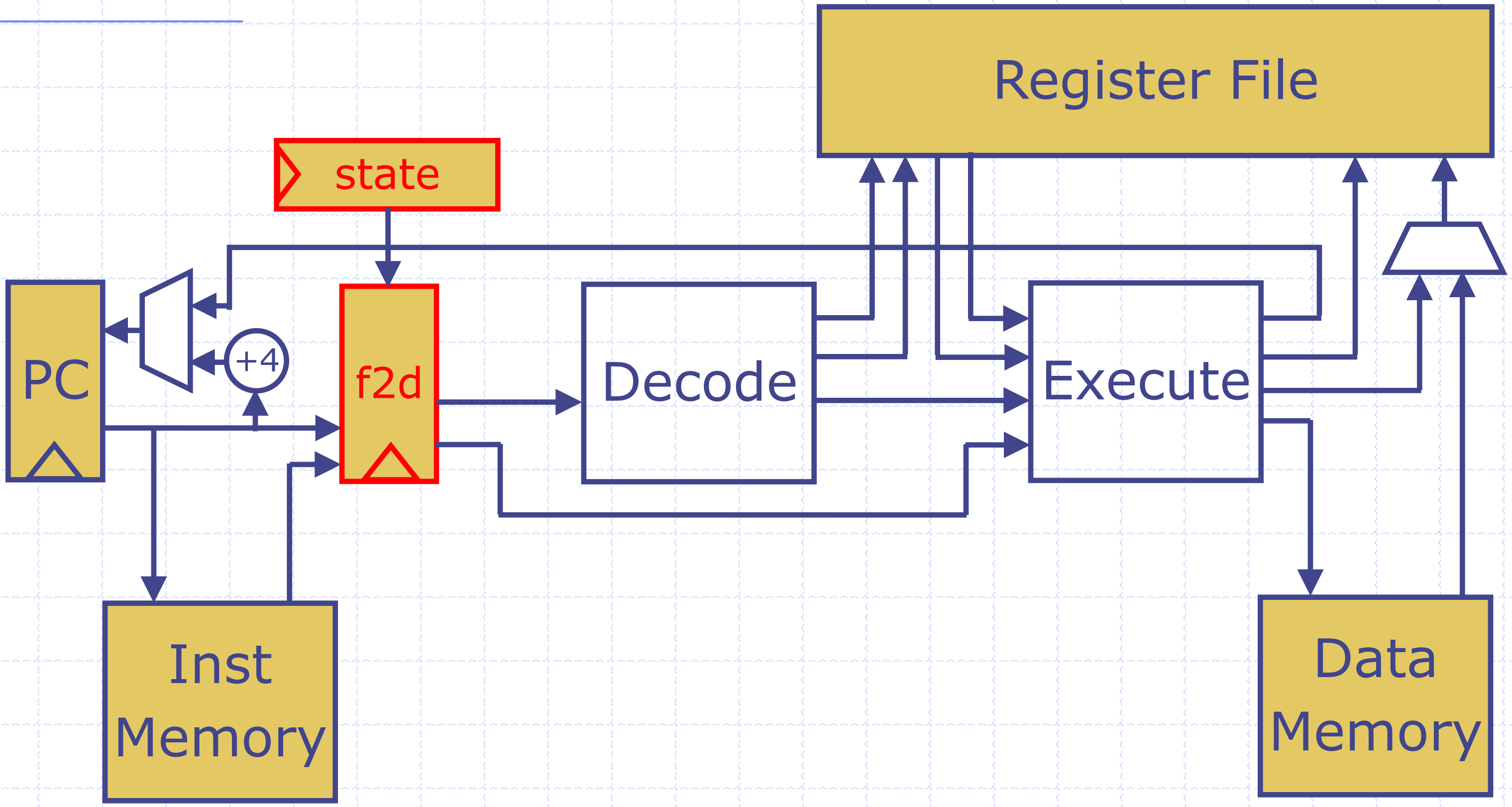
$t_{Clock} > \quad \max\{t_M , (t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB})\}$

However, this may not improve the performance because each instruction will now take two cycles to execute

Structural Hazards

- Sometimes multicycle implementations are necessary because of resource conflicts, aka, *structural hazards*
  - Princeton style architectures use the same memory for instruction and data and consequently, require at least two cycles to execute Load/Store instructions
  - If the register file supported less than 2 reads and one write concurrently then most instructions would take more than one cycle to execute
- Usually extra registers are required to hold values between cycles

Two-Cycle SMIPS



Introduce register "f2d" to hold a fetched instruction and register "state" to remember the state (fetch/execute) of the processor

Two-Cycle SMIPS

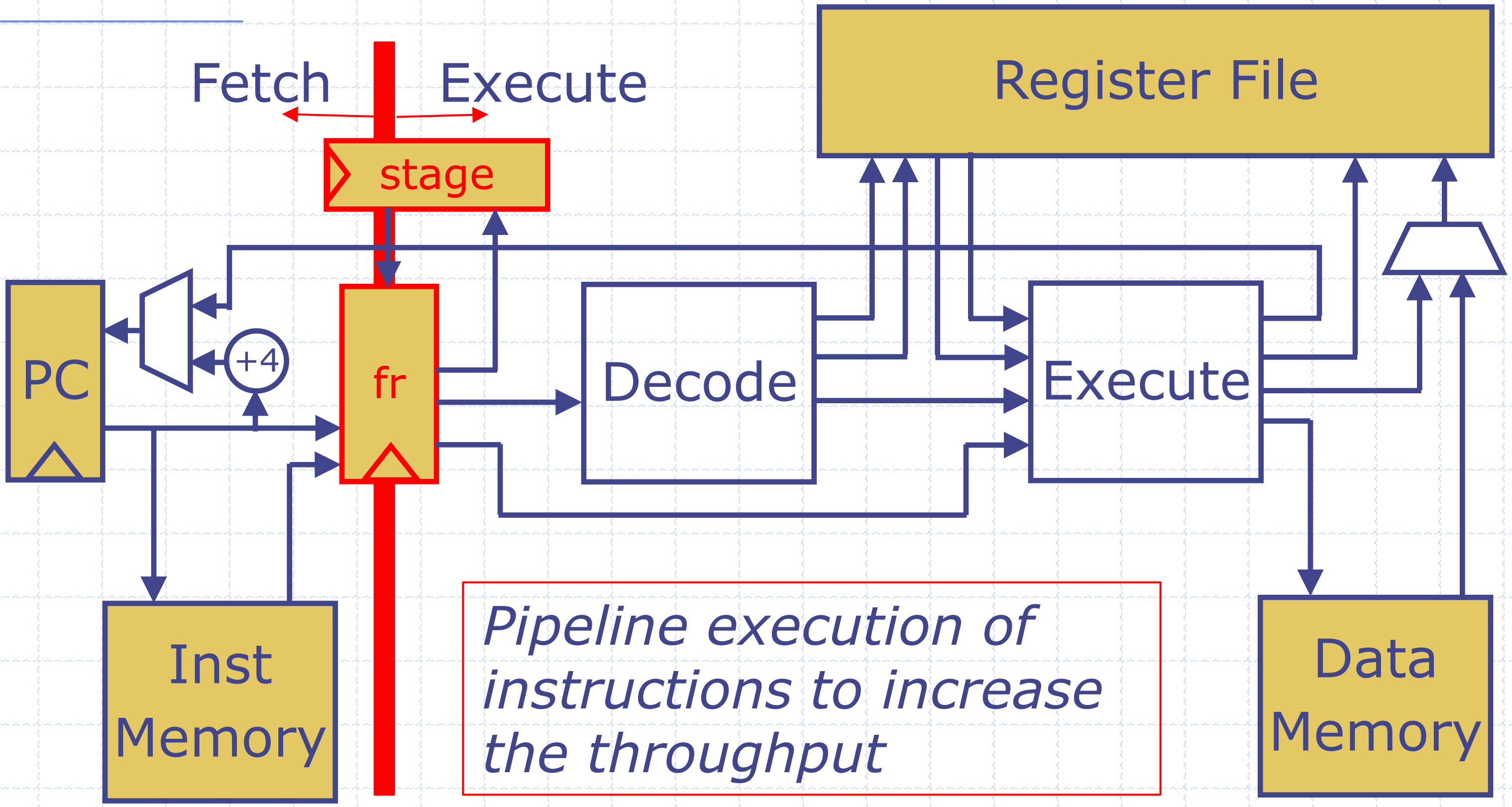
```
module mkProc(Proc);
  Reg#(Addr)   pc <- mkRegU;
  RFile        rf <- mkRFile;
  IMemory      iMem <- mkIMemory;
  DMemory      dMem <- mkDMemory;
  Reg#(Data)   f2d <- mkRegU;
  Reg#(State)  state <- mkReg(Fetch);

  rule doFetch (state == Fetch);
     let inst = iMem.req(pc);
     f2d <= inst;
     state <= Execute;
  endrule
```

Two-Cycle SMIPS

```
rule doExecute(stage==Execute);
  let inst = f2d;
  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);
  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);
  pc <= eInst.brTaken ? eInst.addr : pc + 4;
  state <= Fetch;
endrule endmodule
```

no change from single-cycle

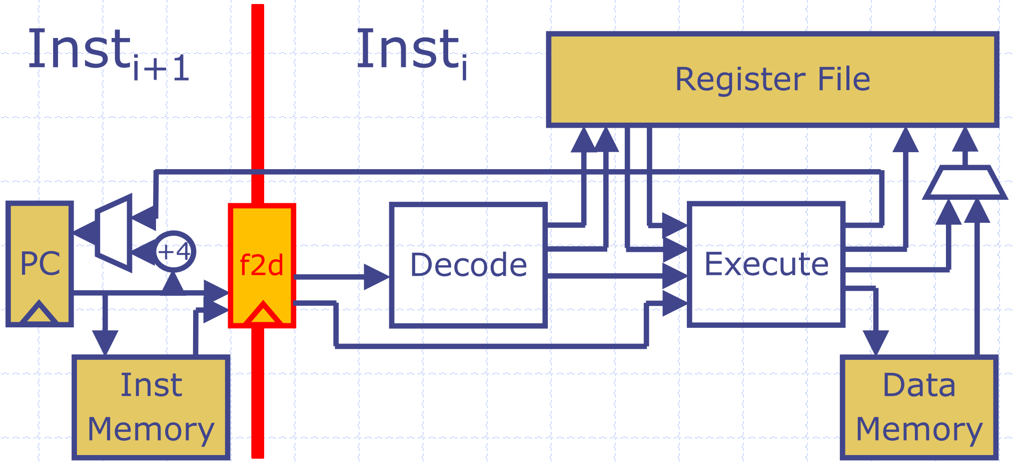
Two-Cycle SMIPS: *Analysis*



Pipeline execution of instructions to increase the throughput

- Cycle time improved but now it takes two cycles to execute each instruction
- In any given clock cycle, lot of unused hardware !

Problems in Instruction pipelining

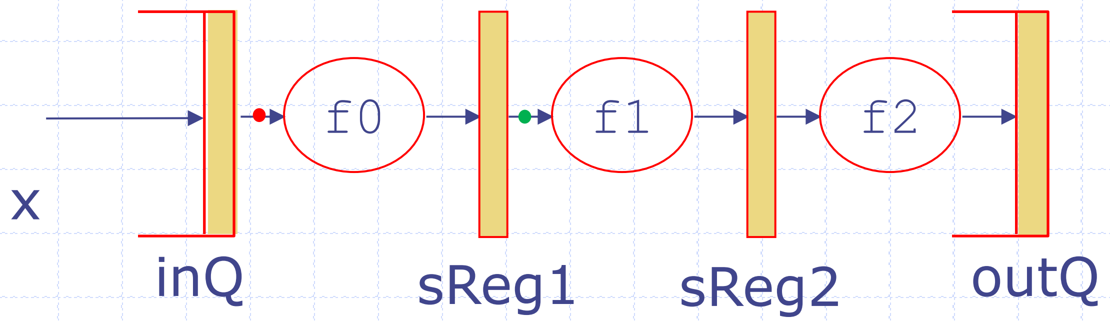


- *Control hazard:* $Inst_{i+1}$ is not known until $Inst_i$ is at least decoded. So which instruction should be fetched?
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory in Princeton-style architecture
- *Data hazard:* $Inst_i$ 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 FFT pipeline

Arithmetic versus Instruction pipelining

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



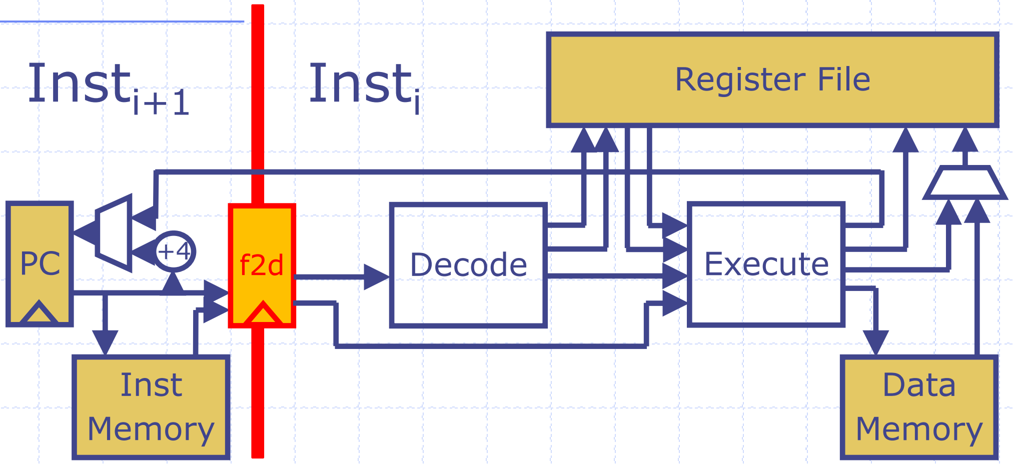
- The entities in an instruction pipeline affect each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls

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

$\Rightarrow$ must deal with hazard

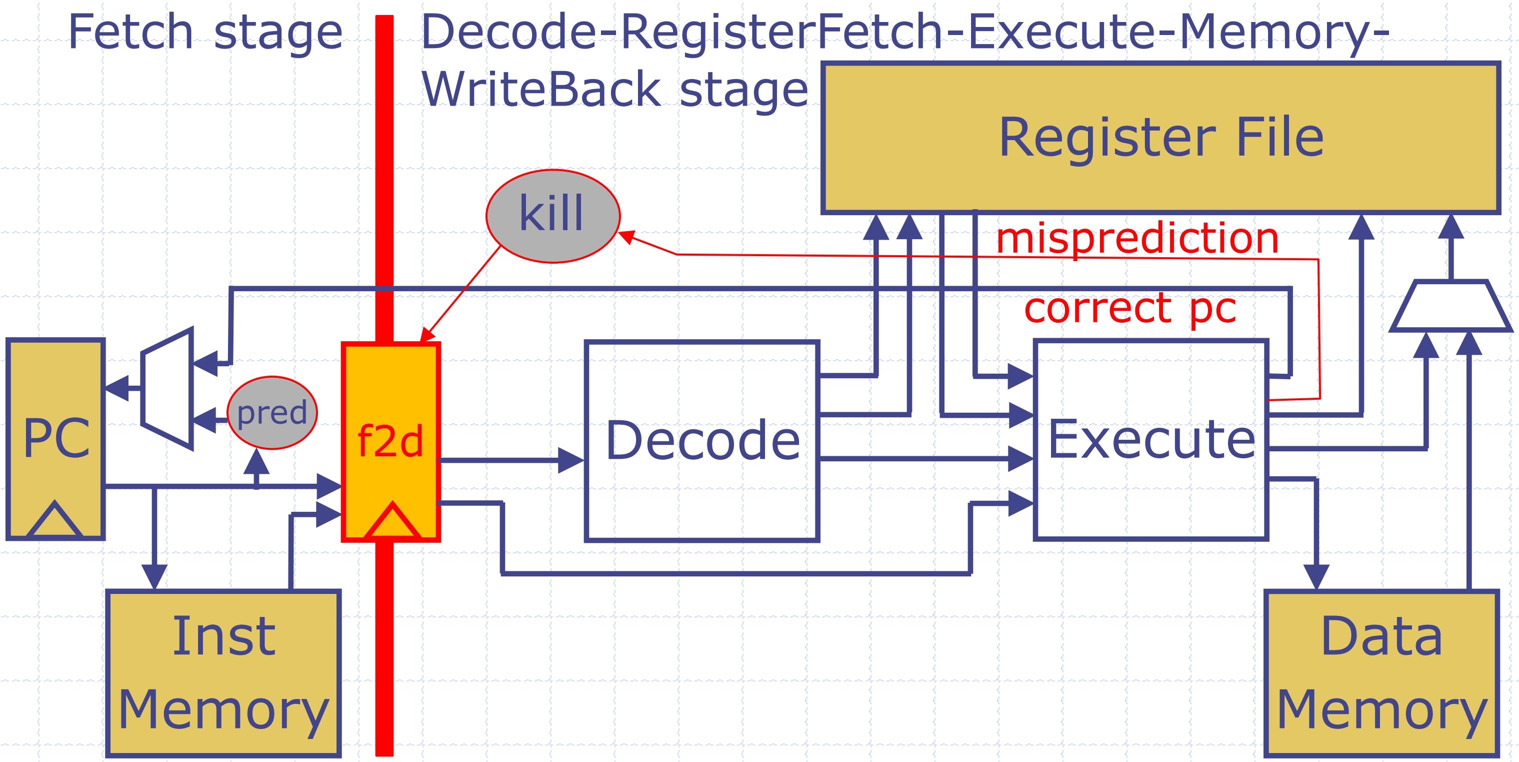
Control Hazards



$Inst_{i+1}$ 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

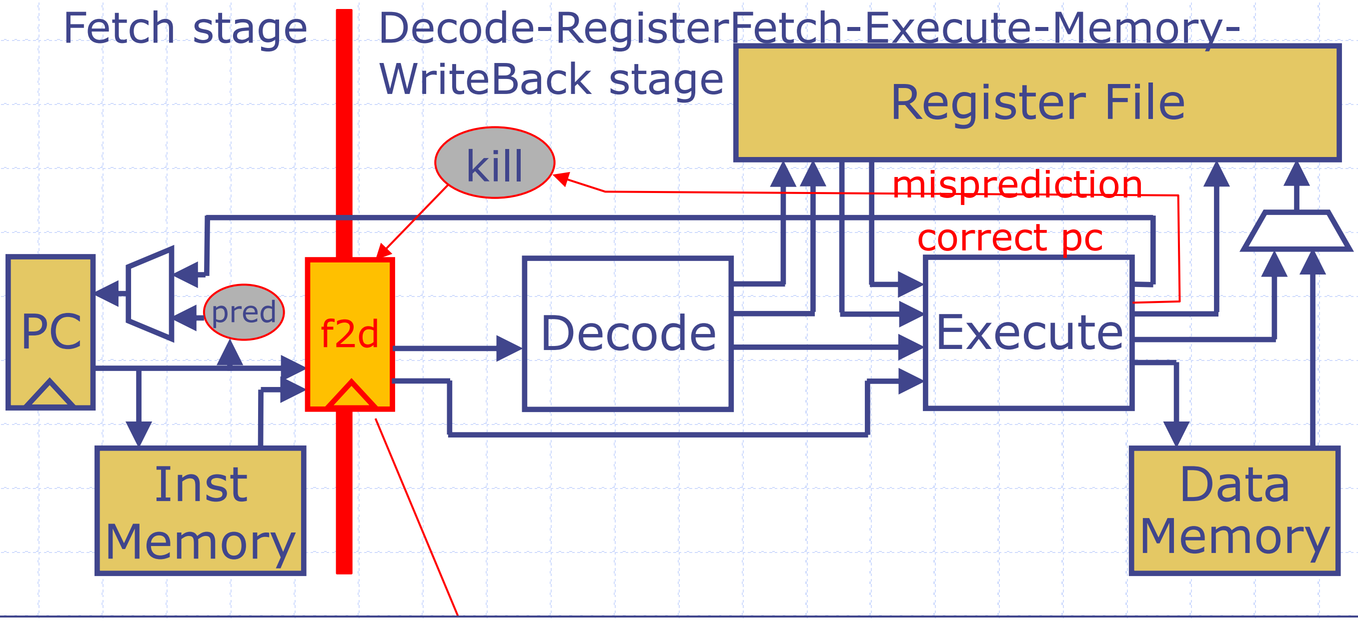
Two-stage Pipelined SMIPS



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

Two-stage Pipelined SMIPS



- f2d must contain a Maybe type value because sometimes the fetched instruction is killed
- Fetch2Decode type captures all the information that needs to be passed from Fetch to Decode, i.e.

  Fetch2Decode {pc:Addr, ppc: Addr, inst:Inst}

Pipelining Two-Cycle SMIPS – single rule

```
rule doPipeline ;
  let instF = iMem.req(pc);                                    fetch
  let ppcF = nextAddr(pc); let nextPc = ppcF;
  let newf2d = Valid (Fetch2Decode{pc:pc,ppc:ppcF,
                                   inst:instF});
  if(isValid(f2d)) begin
                                                             execute
   let x = fromMaybe(?,f2d); let pcD = x.pc;
   let ppcD = x.ppc; let instD = x.inst;
                                                these values are
   let dInst = decode(instD);                   being redefined
   ... register fetch ...;
   let eInst = exec(dInst, rVal1, rVal2, pcD, ppcD);
   ...memory operation ...
   ...rf update ...
   if (eInst.mispredict) begin nextPc = eInst.addr;
                               newf2d = Invalid;   end
                 end
  pc <= nextPc; f2d <= newf2d;
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?

no - double writes in pc

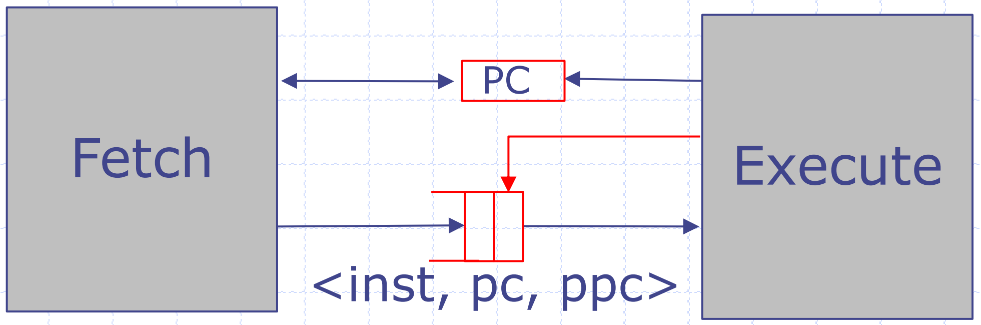
An elastic Two-Stage pipeline:

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
```

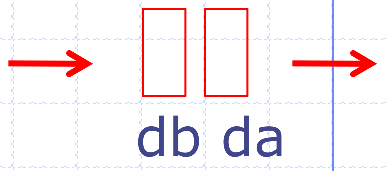
These rules can execute concurrently assuming the FIFO has (enq CF deq) and (enq < clear)

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

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;
    va[1] <= False ; vb[1] <= False endmethod
endmodule
```

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

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

Canonicalize must be the last rule to fire!

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
```