Constructive Computer Architecture:

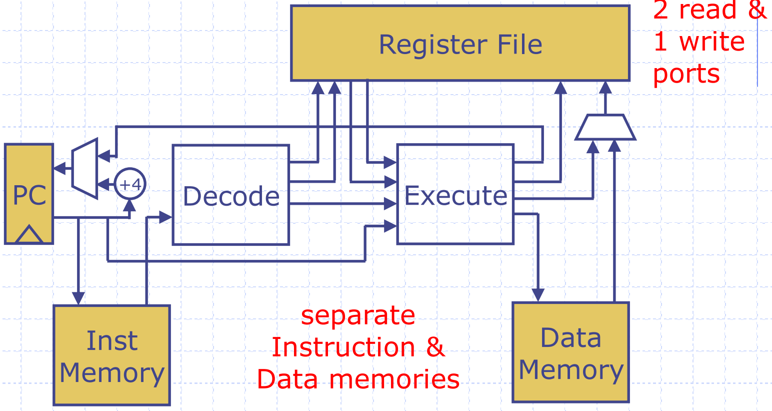
Non-Pipelined and Pipelined Processors

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

Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Single-Cycle RISC Processor



Datapath and control are derived automatically from a high-level rule-based description

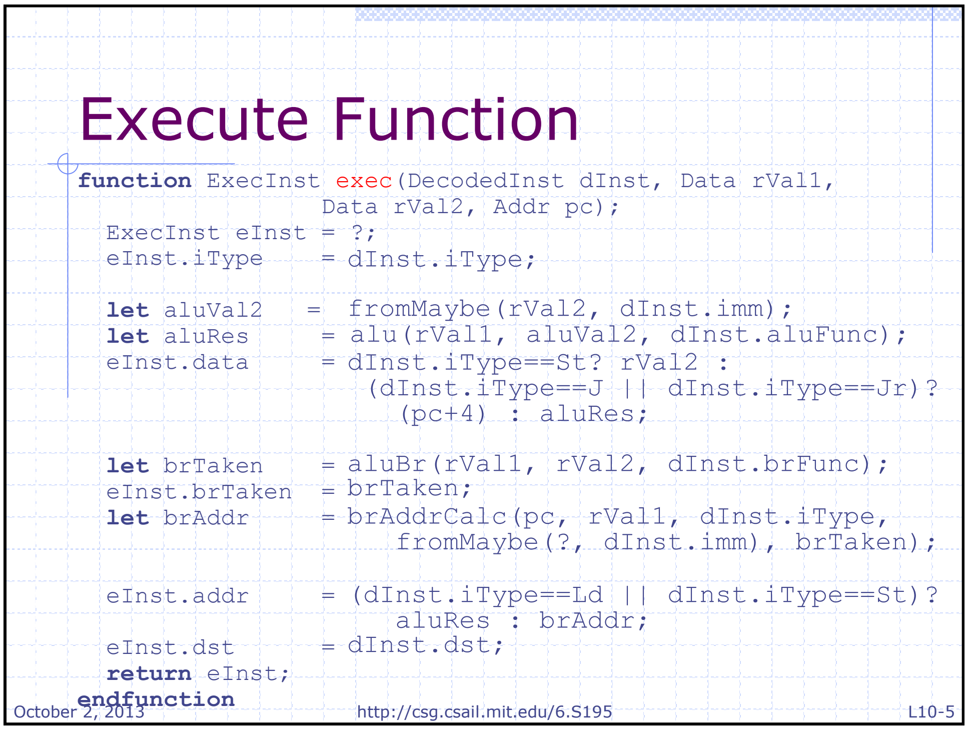
Single-Cycle Implementation

code structure

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

  rule doProc;
    let inst  = iMem.req(pc);
    let dInst = decode(inst);
    let rVal1 = rf.rd1(dInst.rSrc1);
    let rVal2 = rf.rd2(dInst.rSrc2);
    let eInst = exec(dInst, rVal1, rVal2, pc);

    update rf, pc and dMem
```

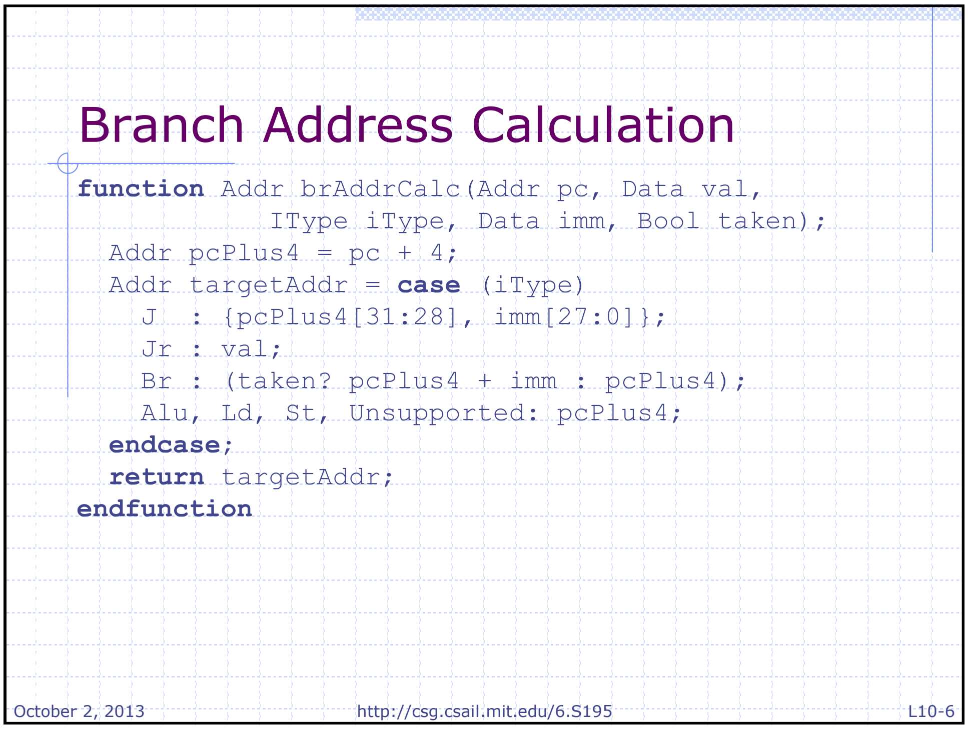
instantiate the state

produces values needed to update the processor state



Execute Function

```
function ExecInst exec(DecodedInst dInst, Data rVal1,
                  Data rVal2, Addr pc);
  ExecInst eInst = ?;
  eInst.iType    = dInst.iType;

  let aluVal2   =  fromMaybe(rVal2, dInst.imm);
  let aluRes     = alu(rVal1, aluVal2, dInst.aluFunc);
  eInst.data     = dInst.iType==St? rVal2 :
                    (dInst.iType==J || dInst.iType==Jr)?
                      (pc+4) : aluRes;

  let brTaken    = aluBr(rVal1, rVal2, dInst.brFunc);
  eInst.brTaken  = brTaken;
  let brAddr     = brAddrCalc(pc, rVal1, dInst.iType,
                     fromMaybe(?, dInst.imm), brTaken);

  eInst.addr     = (dInst.iType==Ld || dInst.iType==St)?
                     aluRes : brAddr;
  eInst.dst      = dInst.dst;
  return eInst;
endfunction
```



Branch Address Calculation

```
function Addr brAddrCalc(Addr pc, Data val,
               IType iType, Data imm, Bool taken);
  Addr pcPlus4 = pc + 4;
  Addr targetAddr = case (iType)
    J  : {pcPlus4[31:28], imm[27:0]};
    Jr : val;
    Br : (taken? pcPlus4 + imm : pcPlus4);
    Alu, Ld, St, Unsupported: pcPlus4;
  endcase;
  return targetAddr;
endfunction
```

Single-Cycle SMIPS *atomic state updates*

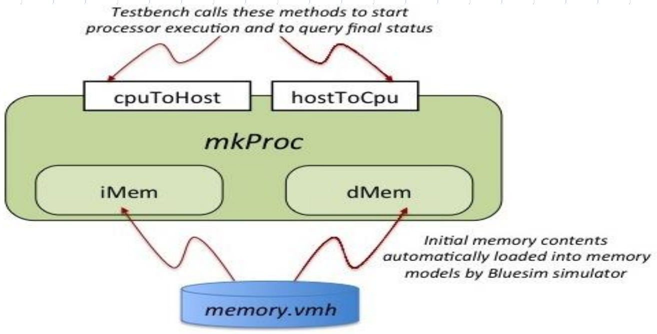
```
if(eInst.iType == Ld)
  eInst.data <- dMem.req(MemReq{op: Ld,
                    addr: eInst.addr, data: ?});
else if (eInst.iType == St)
   let dummy <- dMem.req(MemReq{op: St,
                    addr: eInst.addr, data: data});

if(isValid(eInst.dst))
     rf.wr(validRegValue(eInst.dst), eInst.data);

pc <= eInst.brTaken ? eInst.addr : pc + 4;
```

state updates

```
endrule
endmodule
```

The whole processor is described using one rule; lots of big combinational functions

Processor interface


Testbench calls these methods to start processor execution and to query final status
cpuToHost
hostToCpu
mkProc
iMem
dMem
Initial memory contents automatically loaded into memory models by Bluesim simulator
memory.vmh

```
interface Proc;
   method Action hostToCpu(Addr startpc);
   method ActionValue#(Tuple2#(RIndx, Data)) cpuToHost;
endinterface
```

Stream of register values from the CPU

Coprocessor Registers

- MIPS allows extra sets of 32-registers each to support system calls, floating point, debugging etc. These registers are known as coprocessor registers
  - The registers in the nth set are written and read using instructions MTCn and MFCn, respectively
  - Set 0 is used to get the results of program execution (Pass/Fail), the number of instructions executed and the cycle counts
  - Type `FullIndx` is used to refer to the normal registers plus the coprocessor set 0 registers
  - function `validRegValue(FullIndx r)` returns index of `r`

```
typedef Bit#(5)  RIndx;
typedef enum {Normal, CopReg} RegType deriving (Bits, Eq);
typedef struct {RegType regType; RIndx idx;} FullIndx;
deriving (Bits, Eq);
```

Code with coprocessor calls

```
let copVal = cop.rd(validRegValue(dInst.src1));
let eInst = exec(dInst, rVal1, rVal2, pc, copVal);
```

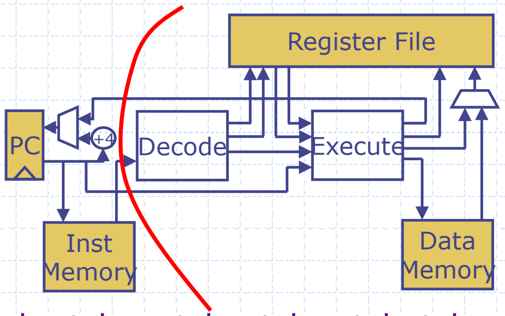
pass coprocessor register values to execute MFC0

```
cop.wr(eInst.dst, eInst.data);
```

write coprocessor registers (MTC0) and indicate the completion of an instruction

We did not show these lines in our processor to avoid cluttering the slides

Single-Cycle SMIPS: *Clock Speed*



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

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

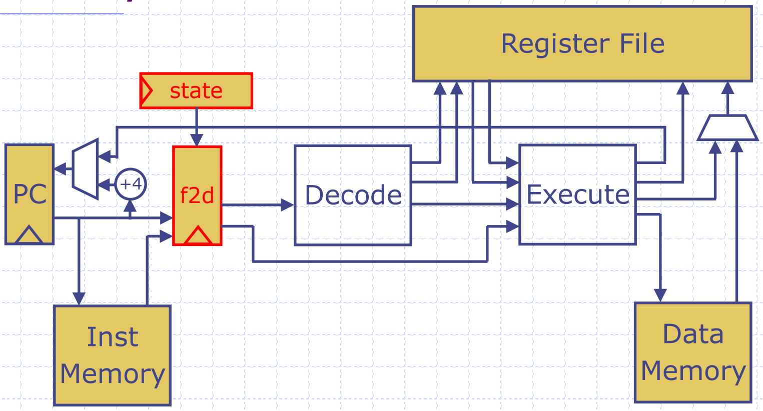
$t_{Clock} > \; \max \{t_M \, , (t_{DEC} + t_{RF} + t_{ALU} + t_M + t_{WB})\}$

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

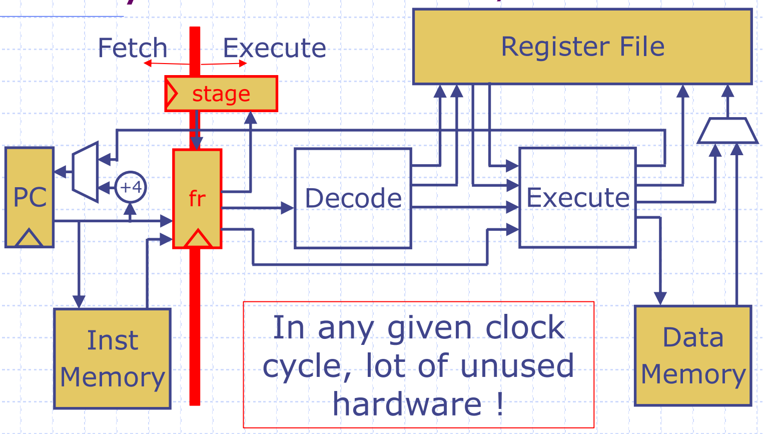
```
module mkProc(Proc);
   Reg#(Addr)   pc <- mkRegU;
   RFile        rf <- mkRFile;
   IMemory      iMem <- mkIMemory;
   DMemory      dMem <- mkDMemory;
   Reg#(Data)   f2d <- mkRegU;
   Reg#(State)  state <- mkReg(Fetch);

   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

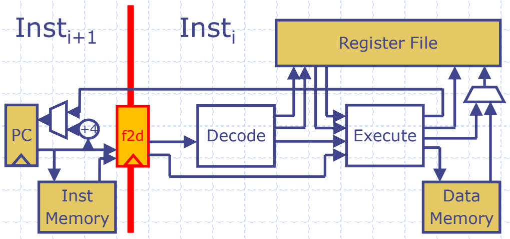
Two-Cycle SMIPS: *Analysis*



In any given clock cycle, lot of unused hardware !

Pipeline execution of instructions to increase the throughput

Problems in Instruction pipelining

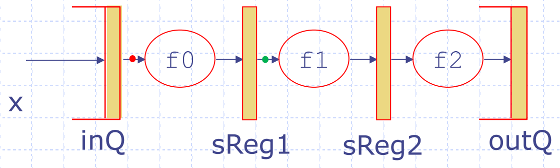


- *Control hazard:* $Inst_{i+1}$ is not known until $Inst_i$ is at least decoded. So which instruction should be fetched?
- *Structural hazard:* Two instructions in the pipeline may require the same resource at the same time, e.g., contention for memory
- *Data hazard:* $Inst_i$ may affect the state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

none of these hazards were present in the FFT pipeline

Arithmetic versus Instruction pipelining

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

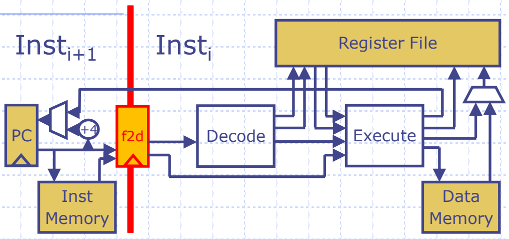


- The entities in an instruction pipeline affect each other
  - This causes pipeline stalls or requires other fancy tricks to avoid stalls
  - Processor pipelines are significantly more complicated than arithmetic pipelines

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

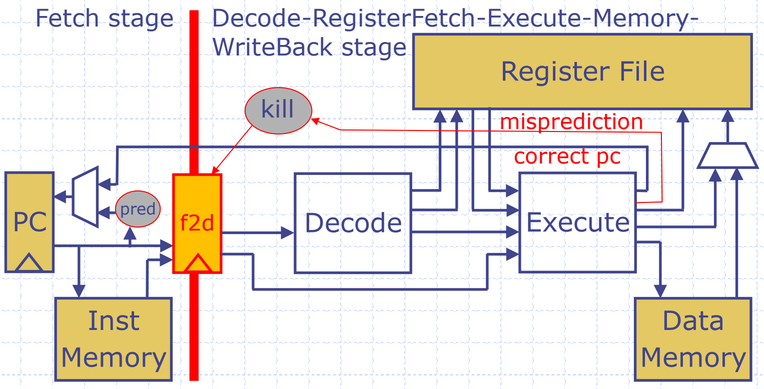
$\Rightarrow$ must deal with hazard

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

Pipelining Two-Cycle SMIPS – singlerule

```
rule doPipeline ;
  let newInst = iMem.req(pc);                              fetch
  let newPpc = nextAddr(pc); let newPc = ppc;
  let newIr=Valid(Fetch2Decode{pc:newPc,ppc:newPpc,
                               inst:newInst});
  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?

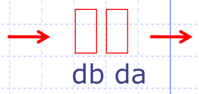
no –
double writes in pc

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)

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

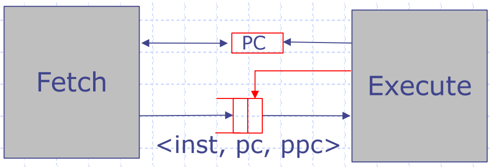
```
rule foo ;
    f.deq; if (p) f.clear
endrule
```

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
```

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