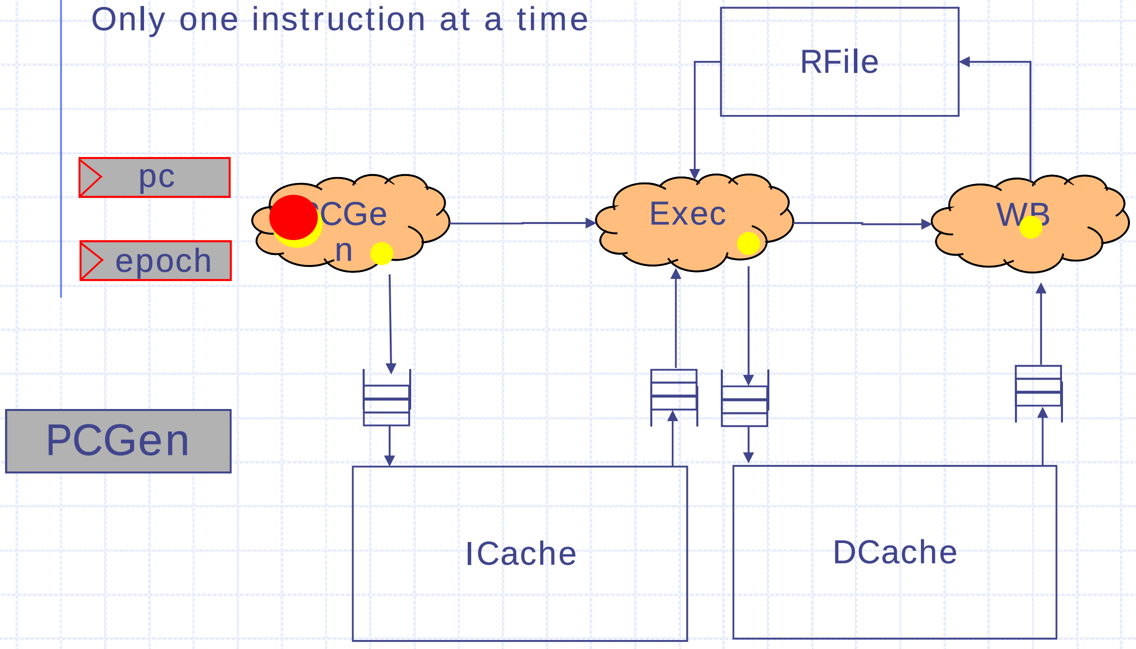
Tutorial: Lab 4 Again

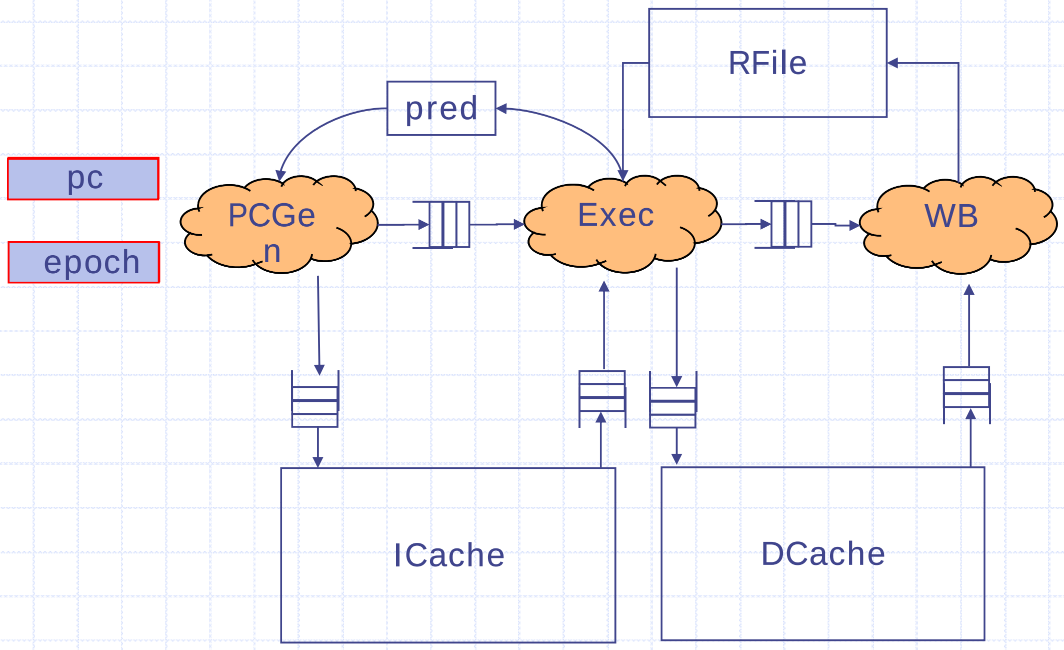
Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

What you've been given

Only one instruction at a time



First Task: Pipelined CPU



Isolate Tasks to stages

- Initially writeback and execute both write to the register file
- This will cause a conflict between the two stages
  - Need to all delay register file writes to the writeback stage

Add Interstage Buffering

- Problem: Multiple stages read/write the same stage

- Only one stage should
  - read each data element
  - Write each data element

Stage stage usage

- PC Gen:
  - PC/epoch: read
  - IMem: request
  - pc2execQ (enq)
- Execute
  - PC/epoch: write
  - pc2execQ: (first/deq)
  - IMem: response
  - DMem: request
  - exec2wbQ(enq)
  - rf (read)
- Writeback
  - exec2wbQ: (first/deq)
  - DMem: response
  - rf:write

Add Stalling Logic:

- We can’t read if there is data in the WB stage waiting to be executed
- Data in exec2wbQ SFIFO:

WWb	{.dst, .val}
WLd	{.dst}
WSt	void
WNop	void

- Stall execute if we need to read one of those registers

Now we can allow exec & WB to happen concurrently

- No longer go to the Writeback stage. Go straight to PCGen stage

Speculation of PC

- We want PCGen and Exec to happen concurrently but:
  - pcGen reads PC/epoch
  - Exec writes PC/epoch
- Have PCGen guess the next PC
  - Pass guess to exec to verify

Original Rules

```
rule pcgen(stage == PCGen);
   imem.req(pc);
   pc2execQ.enq(tuple2(pc, epoch));
   stage <= Execute;
endrule

rule exec(!stall(exec2wbQ) && stage == Execute);
   match {.pc, .epoch} = pc2execQ.first();
   pc2execQ.deq();
   let inst <- imem.response();
   Addr nextPC = epc + 4;
   case (inst) matches ...

   pc <= nextPC;
endrule
```

New Rules

```
rule pcgen(True);
  imem.req(pc);
  let predPC = pc + 4;
  pc <= predPC;
  pc2execQ.enq(tuple2(pc, predPC, epoch));
endrule

match {.epc, .predPC ,.eepoch} = pc2execQ.first();
rule exec(!stall(exec2wbQ) && eepoch == execEpoch);
  pc2execQ.deq();
  let inst <- imem.response();
  Addr nextPC = epc + 4;
  case (inst) matches …
  if (nextPC != predPC)
    begin pc <= nextPC; epoch <= epoch+1;
          execEpoch <= execEpoch+1; end
endrule

rule discard(eepoch != execEpoch);
  pc2execQ.deq(); let inst <- imem.response();
endrule
```

Getting Performance

- Expected Order per cycle
  - WB < Exec < PCGen
- Requires:
  - rf: write < read
  - exec2wbQ: first,deq < enq, read
  - pc2execQ: first,deq < enq
  - epoch: write < read
  - pc: write < read < write

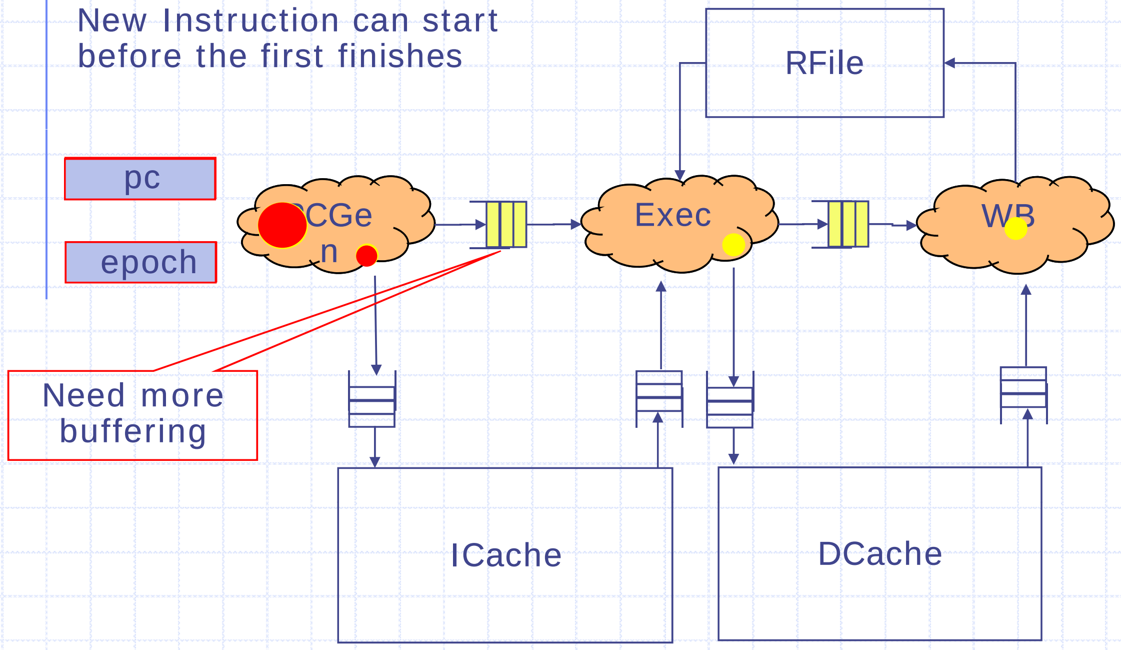
Write from exec, then read & write from pcGen

The new PC module

```
module mkPCReg(ExtendedReg#(a));
  Reg#(a) r <- mkReg(0);
  RWire#(a) rw1 <- mkRWire();
  RWire#(a) rw2 <- mkRWire();
  rule doWrite(isJust(rw1.wget) || isJust(rw2.wget));
    r <= fromMaybe(rw2.wget, fromMaybe(rw1.wget, ?));
  endrule
  method Action write1(x);
    rw.wset(x);
  endmethod
  method a read();
    return fromMaybe(rw1.wget, r);
  endmethod
  method Action write2(x);
    rw2.wset(x);
  endmethod
endmodule
```

First Task: Pipelined CPU

New Instruction can start before the first finishes



Branch Prediction

- Pipeline has simple speculation

```
rule pcGen (True);
   pc <= pc + 4;
   otherActions;
endrule
```

Simplest prediction: Always not-taken

Branch Prediction

```
interface BranchPredictor;
  method Addr getNextPC(Addr pc);
  method Action update (Addr pc,
                                  Addr correct_next_pc);
endinterface
rule pcGen (True);
  pc <= pred.getNextPC(pc);
  otherActions;
endrule
rule execute …
  if (nextPC != correctPC) pred.update(curPc, nextPC);

  case (instr) matches …
    BzTaken: if (mispredicted) …
endrule
```

Make prediction

Update predictions