Constructive Computer Architecture:

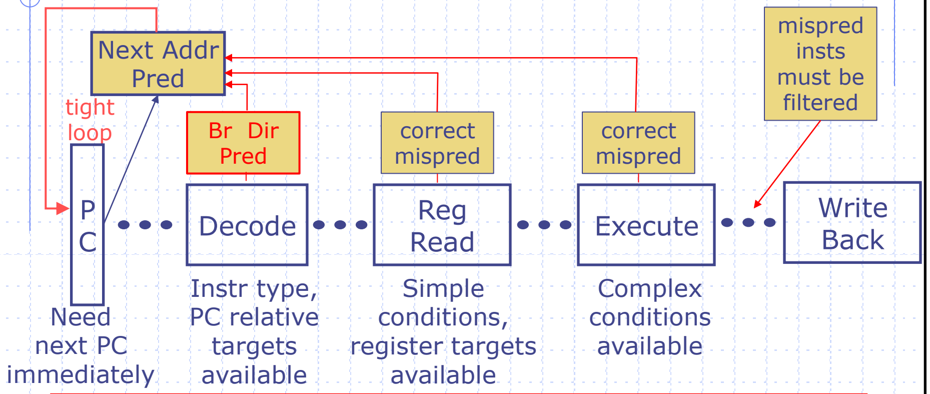
Branch Prediction: Direction Predictors

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

Multiple Predictors: BTB + Branch Direction Predictors


Next Addr Pred
tight loop
P C
Br Dir Pred
correct mispred
correct mispred
mispred insts must be filtered
Decode
Reg Read
Execute
Write Back
Need next PC immediately
Instr type, PC relative targets available
Simple conditions, register targets available
Complex conditions available

- Suppose we maintain a table of how a particular Br has resolved before. At the decode stage we can consult this table to check if the incoming (pc, ppc) pair matches our prediction. If not redirect the pc

Branch Prediction Bits

Remember how the branch was resolved previously

- Assume 2 BP bits per instruction
- Use saturating counter

On ¬taken ↓	↑ On taken	1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly ¬taken
		0	0	Strongly ¬taken

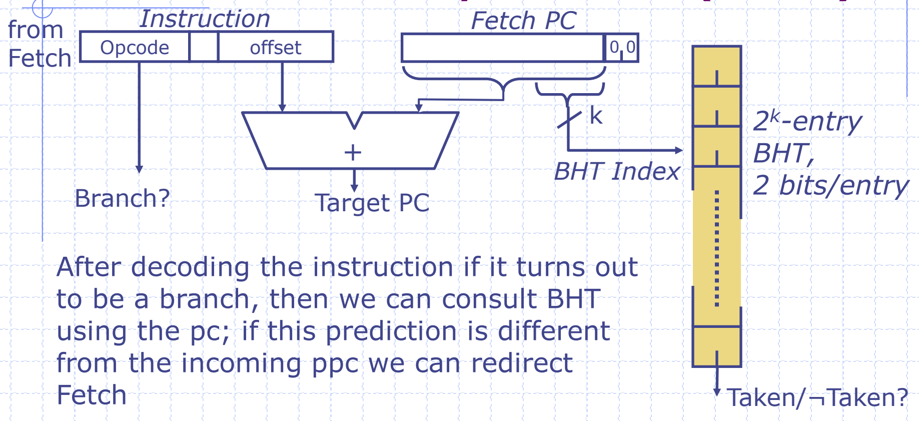
Direction prediction changes only after two successive bad predictions

Two-bit versus one-bit Branch prediction

- Consider the branch instruction needed to implement a loop
  - with one bit, the prediction will always be set incorrectly on loop exit
  - with two bits the prediction will not change on loop exit

A little bit of hysteresis is good in changing predictions

Branch History Table (BHT)



After decoding the instruction if it turns out to be a branch, then we can consult BHT using the pc; if this prediction is different from the incoming ppc we can redirect Fetch

4K-entry BHT, 2 bits/entry, ~80-90% correct direction predictions

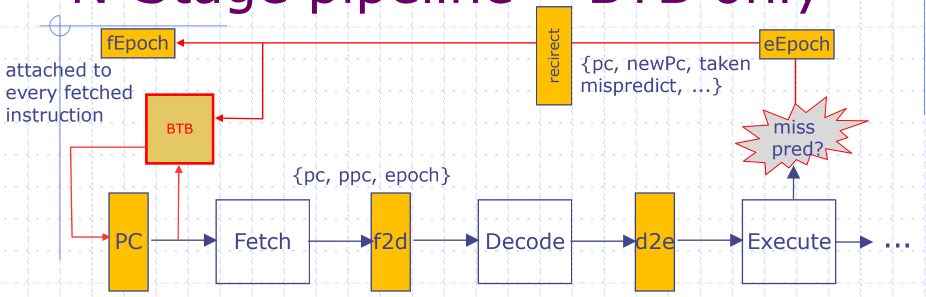
Where does BHT fit in the processor pipeline?

- BHT can only be used after instruction decode
- We still need the next instruction address predictor (e.g., BTB) at the fetch stage
- Need a mechanism to update the BHT
  - where does the update information come from?

Execute

A step-by-step explanation of how pipelines with multiple predictors work

N-Stage pipeline – BTB only

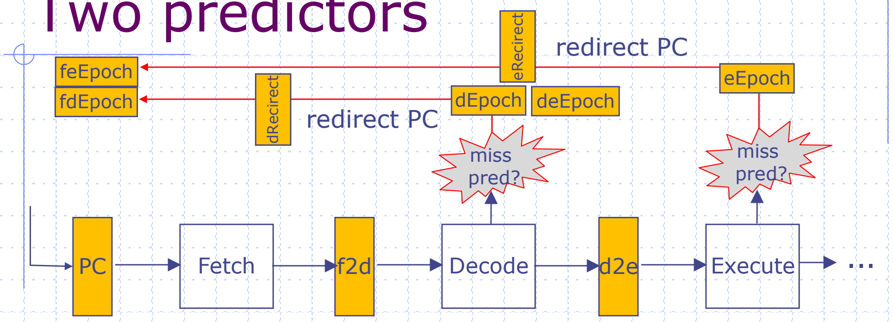


- At Execute:
  - if (epoch!=eEpoch) then mark instruction as poisoned, send it to the latter stages so that scoreboard entry can be removed
  - if no poisoning & mispred then change eEpoch; send <pc, newPc, ...> to Fetch
- At Fetch:
  - msg from execute: train BTB with <pc, newPc, taken, mispredict>
  - if msg from execute indicates misprediction then set pc, change fEpoch

Nomenclature

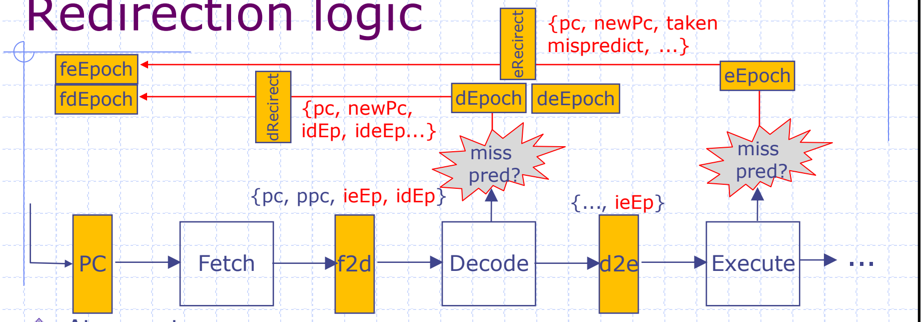
- *Drop an instruction:* What we really mean is poison the instruction so that the subsequent stages know not to update any architectural state. The poisoned instruction has to be passed down for book keeping reasons, i.e., to remove it from the scoreboard.
- *Detecting a misprediction versus training/updating a predictor*. On a pc misprediction, information about redirecting the pc has to be passed to the fetch stage. However for training the BTB and other predictors information has to be passed even when there is no misprediction.
  - we will first focus on pc *redirection* and then on predictor *training*

N-Stage pipeline: Two predictors



- Suppose both Decode and Execute can redirect the PC; Execute redirect should have priority, i.e., Execute redirect should never be overruled
- We will use separate epochs for each redirecting stage
  - feEpoch and deEpoch are estimates of eEpoch at Fetch and Decode, respectively
  - fdEpoch is Fetch’s estimates of dEpoch

N-Stage pipeline: Two predictors Redirection logic



- At execute:
  - if (ieEp!=eEp) then drop the instruction
  - if no-drop & mispred then change eEp; send <correct next pc, new eEp, ...> to fetch
- At fetch:
  - msg from execute: if (mispredict) set pc, change feEp
  - msg from decode: if (ideEp=feEp)then set pc, change fdEp

make sure that the msg from Decode is not from a wrong path instruction

- At decode:
  - if (ieEp!=deEp) then deEp <= ieEp and dEp = idEp
    else if (idEp!=dEp) then drop the instruction
  - for non dropped instructions
    if (ppc != Dpred(pc)) then change dEp, send <Dpred(pc), new dEp, deEp> to Fetch

if incoming eEp does not match deEp then Execute has redirected the pc

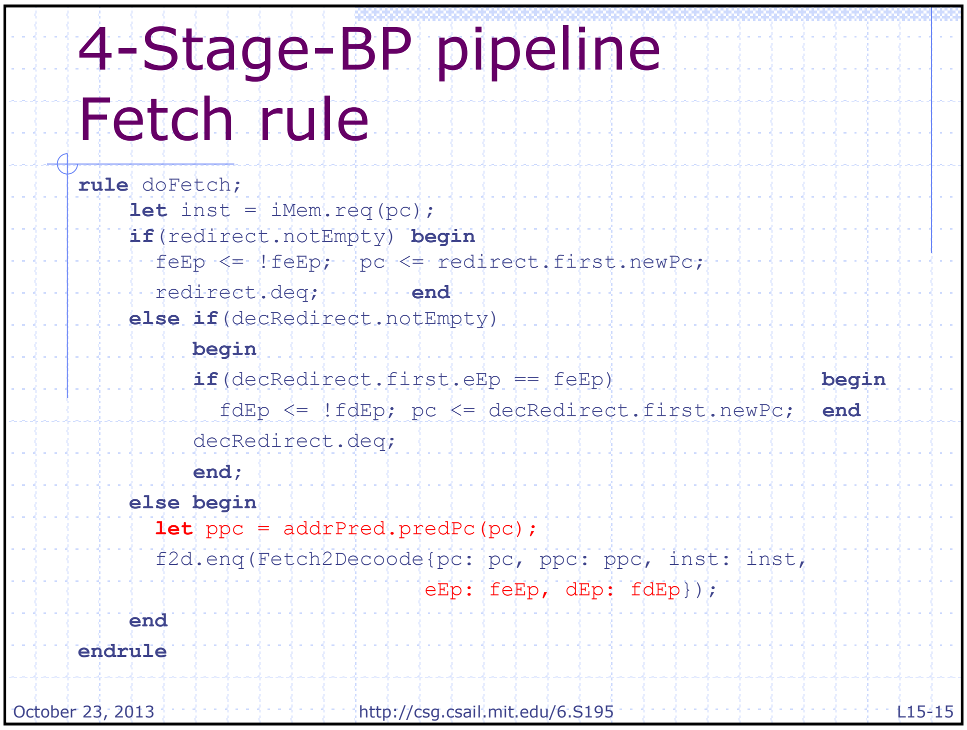
now some coding ...

- 4-stage pipeline (F, D&R, E&M, W)
- No predictor training, so messages are sent only for redirection

You will explore the effect of predictor training in the lab

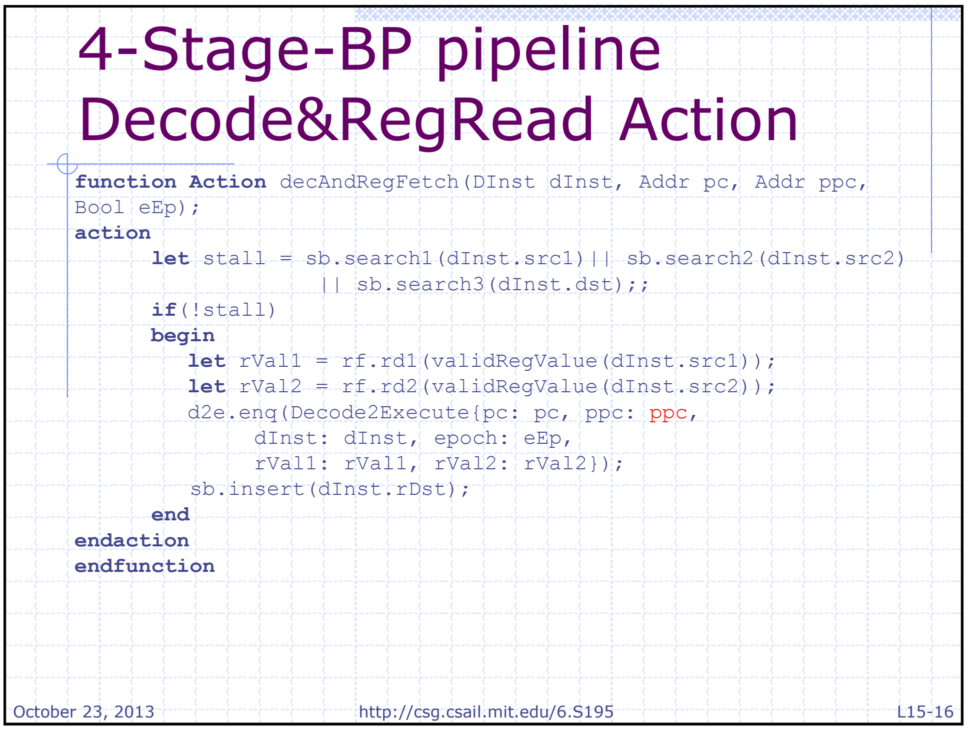
4-Stage pipeline with Branch Prediction

```
module mkProc(Proc);
   Reg#(Addr)         pc <- mkRegU;
   RFile              rf <- mkBypassRFile;
   IMemory          iMem <- mkIMemory;
   DMemory          dMem <- mkDMemory;
   Fifo#(1, Decode2Execute) d2e <- mkPipelineFifo;
   Fifo#(1, Exec2Commit)    e2c <- mkPipelineFifo;
   Scoreboard#(2) sb <- mkPipelineScoreboard;
   Reg#(Bool)     feEp <- mkReg(False);
   Reg#(Bool)     fdEp <- mkReg(False);
   Reg#(Bool)     dEp <- mkReg(False);
   Reg#(Bool)     deEp <- mkReg(False);
   Reg#(Bool)     eEp <- mkReg(False);
   Fifo#(ExecRedirect) redirect <- mkBypassFifo;
   Fifo#(DecRedirect) decRedirect <- mkBypassFifo;
   AddrPred#(16) addrPred <- mkBTB;
   DirPred#(1024) dirPred <- mkBHT;
```



4-Stage-BP pipeline Fetch rule

```
rule doFetch;
    let inst = iMem.req(pc);
    if(redirect.notEmpty) begin
      feEp <= !feEp;  pc <= redirect.first.newPc;
      redirect.deq;       end
    else if(decRedirect.notEmpty)
         begin
         if(decRedirect.first.eEp == feEp)                 begin
           fdEp <= !fdEp; pc <= decRedirect.first.newPc;   end
         decRedirect.deq;
         end;
    else begin
      let ppc = addrPred.predPc(pc);
      f2d.enq(Fetch2Decoode{pc: pc, ppc: ppc, inst: inst,
                           eEp: feEp, dEp: fdEp});
    end
endrule
```



4-Stage-BP pipeline Decode&RegRead Action

```
function Action decAndRegFetch(DInst dInst, Addr pc, Addr ppc,
Bool eEp);
action
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2)
                   || sb.search3(dInst.dst);;
      if(!stall)
      begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, ppc: ppc,
               dInst: dInst, epoch: eEp,
               rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst);
      end
endaction
endfunction
```

4-Stage-BP pipeline Decode&RegRead rule

```
rule doDecode;
  let x = f2d.first; let inst = x.inst; let pc = x.pc;
  let ppc = x.ppc; let idEp = x.dEp; let ieEp = x.eEp;
  let dInst = decode(inst);
  let newPc = dirPrec.predAddr(pc, dInst);
  if(ieEp != deEp) begin // change Decode's epochs and
                         // continue normal instruction execution
      deEp <= ieEp; let newdEp = idEp;
      decAndRegRead(inst, pc, newPc, ieEp);
      if(ppc != newPc)                                  begin
        newDEp = !newdEp; decRedirect.enq(DecRedirect{pc: pc,
                               newPc: newPc, eEp: ieEp}); end
      dEp <= newdEp end
  else if(idEp == dEp) begin
      decAndRegRead(inst, pc, newPc, ieEp);
      if(ppc != newPc)                                  begin
        dEp <= !dEp; decRedirect.enq(DecRedirect{pc: pc,
                              newPc: newPc, eEp: ieEp}); end
                        end
   f2d.deq;
endrule
```

4-Stage-BP pipeline Execute rule

no change

```
rule doExecute;
    let x = d2e.first;
    let dInst = x.dInst; let pc    = x.pc;
    let ppc   = x.ppc;   let epoch = x.epoch;
    let rVal1 = x.rVal1; let rVal2 = x.rVal2;
    if(epoch == eEpoch) begin
      let eInst = exec(dInst, rVal1, rVal2, pc, ppc);
      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});
      e2c.enq(Exec2Commit{dst:eInst.dst, data:eInst.data});
      if(eInst.mispredict) begin
        redirect.enq(eInst.addr); eEpoch <= !eEpoch; end
                        end
    else e2c.enq(Exec2Commit{dst:Invalid, data:?});
    d2e.deq;
endrule
```

4-Stage-BP pipeline Commit rule

```
rule doCommit;
  let dst  = eInst.first.dst;
  let data = eInst.first.data;
  if(isValid(dst))
    rf.wr(tuple2(validValue(dst), data);
  e2c.deq;
  sb.remove;
endrule
```

no change

Exploiting Spatial Correlation

Yeh and Patt, 1992

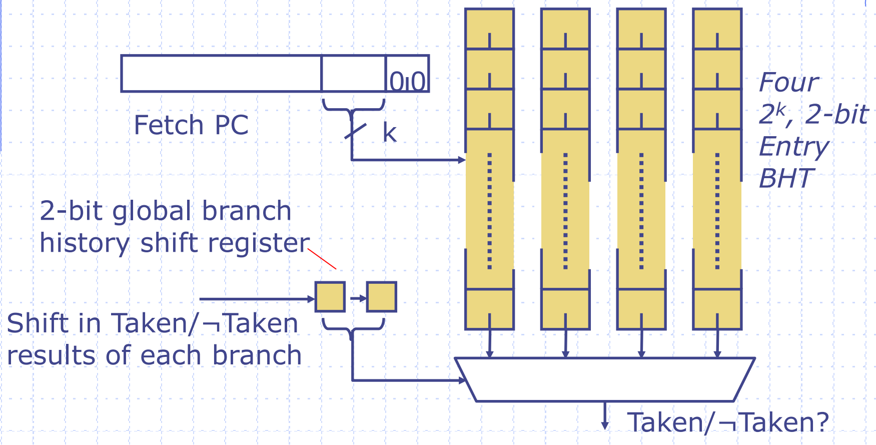
```
if (x[i] < 7) then
        y += 1;
if (x[i] < 5) then
        c -= 4;
```

If first condition is false then so is second condition

History register, H, records the direction of the last N branches executed by the processor and the predictor uses this information to predict the resolution of the next branch

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)


0 0
Fetch PC
k
Four 2^k, 2-bit Entry BHT
2-bit global branch history shift register
Shift in Taken/¬Taken results of each branch
Taken/¬Taken?

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if the same case is used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if the same function is usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

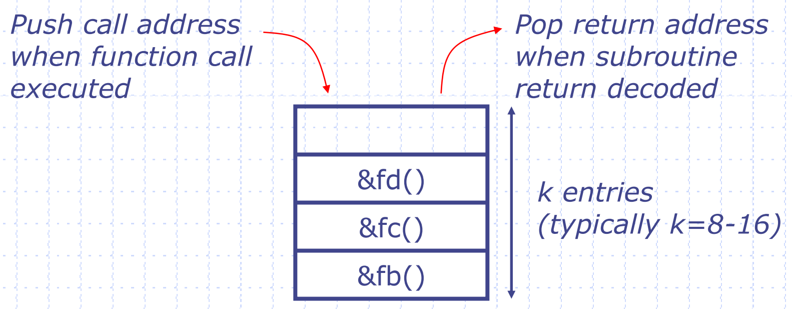
  However, often one function is called from many distinct call sites!

How well does BTB work for each of these cases?

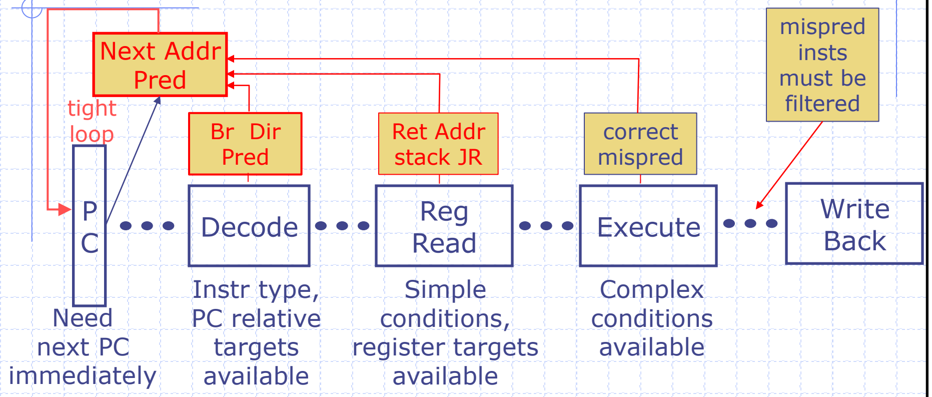
Subroutine Return Stack

- A small structure to accelerate JR for subroutine returns is typically much more accurate than BTBs

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```


Push call address when function call executed
Pop return address when subroutine return decoded
&fd()
&fc()
&fb()
k entries (typically k=8-16)

Multiple Predictors: BTB + BHT + Ret Predictors


Next Addr Pred
tight loop
Br Dir Pred
Ret Addr stack JR
correct mispred
mispred insts must be filtered
PC
Decode
Reg Read
Execute
Write Back
Need next PC immediately
Instr type, PC relative targets available
Simple conditions, register targets available
Complex conditions available

- One of the PowerPCs has all the three predictors
- Performance analysis is quite difficult – depends upon the sizes of various tables and program behavior
- Correctness: The system must work even if every prediction is wrong