Constructive Computer Architecture:

Branch Prediction: Direction Predictors

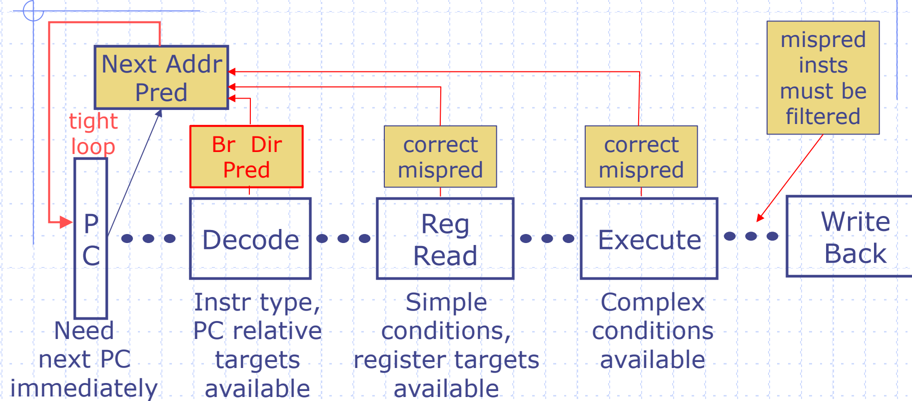
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Slides from L15 are included for completeness sake

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



- 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

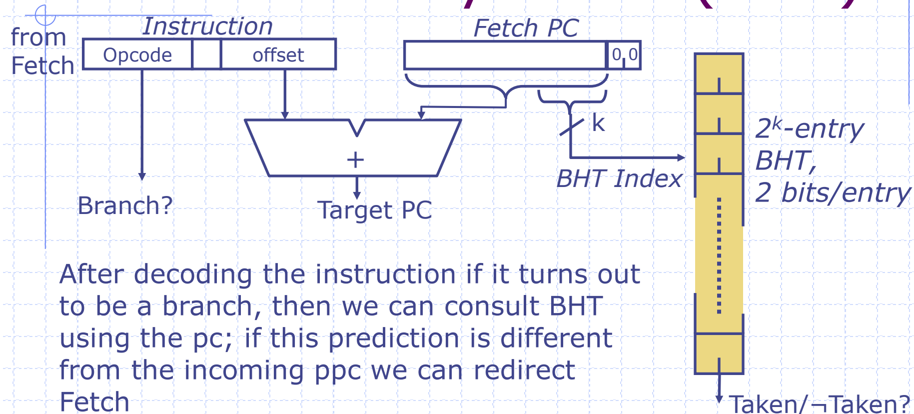
Direction prediction changes only after two successive bad predictions

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
- *Predictor training:* On a pc misprediction, information about redirecting the pc has to be passed to the fetch stage. However for training branch predictors information has to be passed even when there is no misprediction

2-Stage-DH pipeline doExecute rule

```
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});
      if (isValid(eInst.dst))
        rf.wr(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict)  begin
        redirect.enq(RedirectInfo{pc: pc, nextPc: eInst.addr,
                     taken: eInst.brTaken);
        eEpoch <= !eEpoch; end
                        end
    d2e.deq; sb.remove;
endrule
```

Even if there is no misprediction, send information about branch resolution for training predictors

2-Stage-DH pipeline doExecute rule: predictor training

```
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});
      if (isValid(eInst.dst))
        rf.wr(validRegValue(eInst.dst), eInst.data);
      if(eInst.mispredict) eEpoch <= !eEpoch;
      if(eInst.iType == J || eInst.iType == Jr || eInst.iType == Br)
        redirect.enq(Redirect{pc: pc, nextPc: eInst.addr,
            taken: eInst.brTaken, mispredict: eInst.mispredict,
            brType: eInst.iType});
    d2e.deq; sb.remove;
endrule
```

Even if there is no misprediction, send information about branch resolution for training predictors

2-Stage-DH pipeline doFetch rule:

```
rule doFetch;
    let inst = iMem.req(pc);
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first.nextPC;
      redirect.deq; nap.update(redirect.first); end

    else begin
      let ppc = nap.predPC(pc); let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                              begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, nextPC: ppc,
               dIinst: dInst, epoch: fEpoch,
               rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc <= ppc; end
          end
    endrule
```

update nap but change pc only on a mispredict

2-Stage-DH pipeline doFetch rule: predictor training

update nap but change pc only on a mispredict

```
rule doFetch;
    let inst = iMem.req(pc);
    if(redirect.notEmpty) begin
      fEpoch <= !fEpoch;  pc <= redirect.first.nextPc;
      redirect.deq; nap.update(redirect.first); end
    if(redirect.notEmpty && redirect.first.mispredict)
       begin pc <= redirect.first.nextPc; fEpoch <= !fEpoch; end
    else begin
      let ppc = nap.predPC(pc); let dInst = decode(inst);
      let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
      if(!stall)                          begin
         let rVal1 = rf.rd1(validRegValue(dInst.src1));
         let rVal2 = rf.rd2(validRegValue(dInst.src2));
         d2e.enq(Decode2Execute{pc: pc, nextPC: ppc,
                dInst: dInst, epoch: fEpoch,
                rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); pc <= ppc; end
          end
    endrule
```

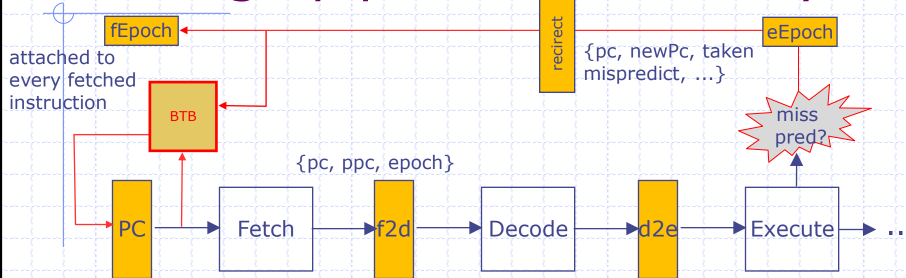

Multiple predictors in a pipeline

- At each stage we need to take two decisions:
  - Whether the current instruction is a *wrong path instruction*. Requires looking at epochs
  - Whether the prediction (ppc) following the current instruction is good or not. Requires consulting the prediction data structure (BTB, BHT, …)
- Fetch stage must correct the pc unless the redirection comes from a known wrong path instruction
- Redirections from Execute stage are always correct, i.e., cannot come from wrong path instructions

Dropping or poisoning an instruction

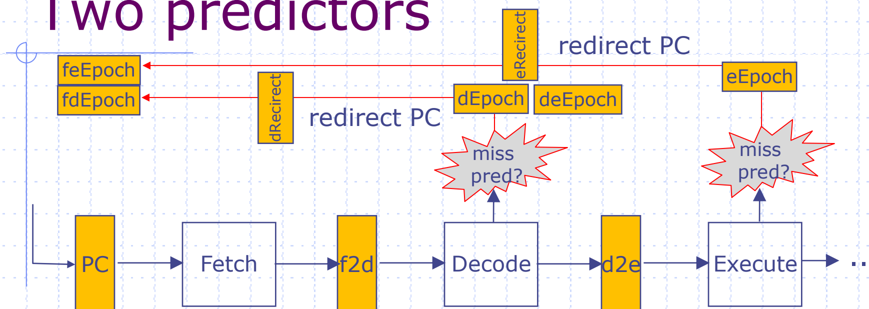
- Once an instruction is determined to be on the wrong path, the instruction is either dropped or poisoned
- Drop: If the wrong path instruction has not modified any book keeping structures (e.g., Scoreboard) then it is simply removed
- Poison: If the wrong path instruction has modified book keeping structures then it is poisoned and passed down for book keeping reasons (say, to remove it from the scoreboard)
- Subsequent stages know not to update any architectural state for a poisoned instruction

N-Stage pipeline – BTB only



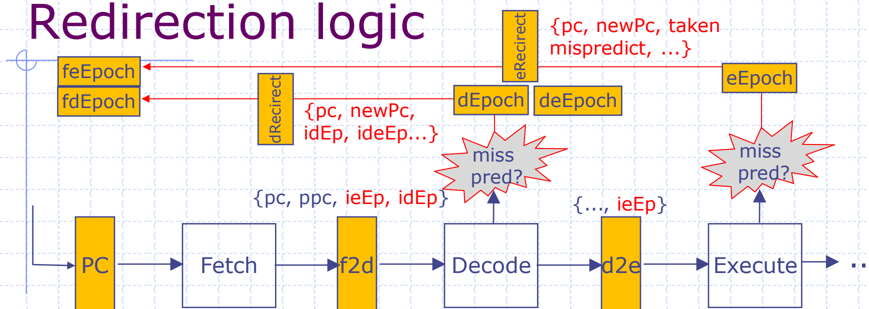
- At Execute:
  - (pc) if (epoch!=eEpoch) then mark instruction as poisoned
  - (ppc) if (no poisoning) & mispred then change eEpoch; send <pc, newPc, ...> to Fetch
- At Fetch:
  - msg from execute: train BTB with <pc, newPc, taken, mispredict>
  - if msg from execute indicates misprediction then set pc, change fEpoch

N-Stage pipeline: Two predictors



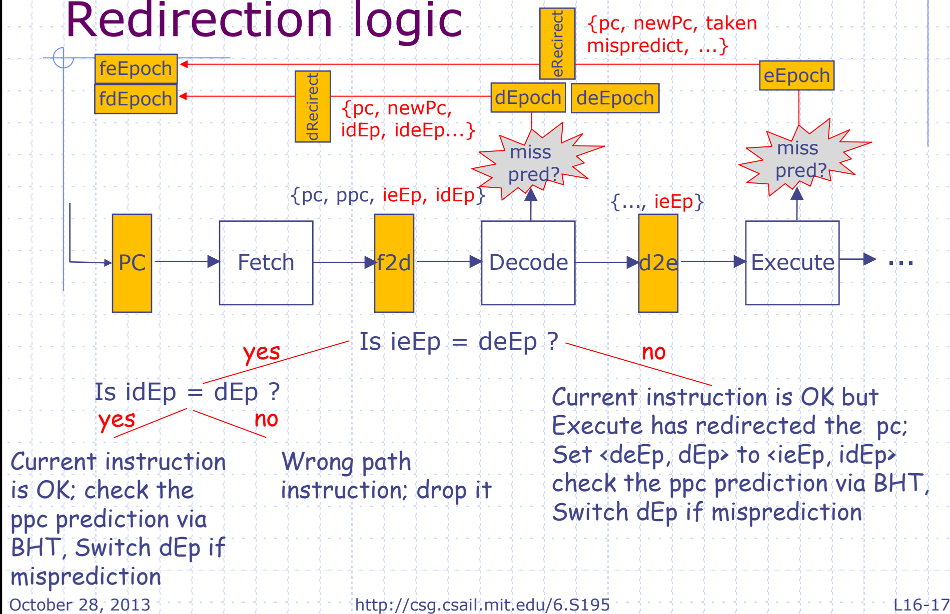
- Suppose both Decode and Execute can redirect the PC; Execute redirect should have priority, i.e., Execute redirect should never be overruled
- We will use separate epochs for each redirecting stage
  - feEpoch and deEpoch are estimates of eEpoch at Fetch and Decode, respectively
  - fdEpoch is Fetch's estimates of dEpoch
  - Initially set all epochs to 0

N-Stage pipeline: Two predictors Redirection logic



- At execute:
  - (pc) if (ieEp!=eEp) then poison the instruction
  - (ppc) if (no poisoning) & mispred then change eEp;
  - (ppc) for every control instruction send <pc, target pc, taken, mispred…> to fetch
- At fetch:
  - msg from execute: if (mispred) set pc, change feEp,
  - msg from decode: If (no redirect message from Execute)
  - if (ideEp=feEp) then set pc, change fdEp to idEp
- At decode: …

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



now some coding …

- 4-stage pipeline (F, D&R, E&M, W)
- Direction predictor training is incompletely specified

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;
  NextAddrPred#(16) nap <- mkBTB;
  DirPred#(1024) dirPred <- mkBHT;
```


4-Stage-BP pipeline Fetch rule: multiple predictors

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

Not enough information is being passed from Fetch to Decode to train BHT – lab problem

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);
     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 nextPc = 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, nextPc, ieEp);
      if(ppc != nextPc)  begin  newdEp = !newdEp;
            decRedirect.enq(DecRedirect{pc: pc,
                             nextPc: nextPc, eEp: ieEp}); end
      dEp <= newdEp end
  else if(idEp == dEp) begin
      decAndRegRead(inst, pc, nextPc, ieEp);
      if(ppc != nextPc)                                 begin
        dEp <= !dEp; decRedirect.enq(DecRedirect{pc: pc,
                           newPc: newPc, eEp: ieEp}); end
                        end // if idEp!=dEp then drop,ie, no action
    f2d.deq;
endrule
```

BHT update is missing- lab problem

4-Stage-BP pipeline Execute rule: predictor training

```
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) eEpoch <= !eEpoch
      if(eInst.iType == J || eInst.iType == Jr || eInst.iType == Br)
        redirect.enq(Redirect{pc: pc, nextPc: eInst.addr,
            taken: eInst.brTaken, mispredict: eInst.mispredict,
            brType: eInst.iType}); 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
```

Exploiting Spatial Correlation

Yeh and Patt, 1992

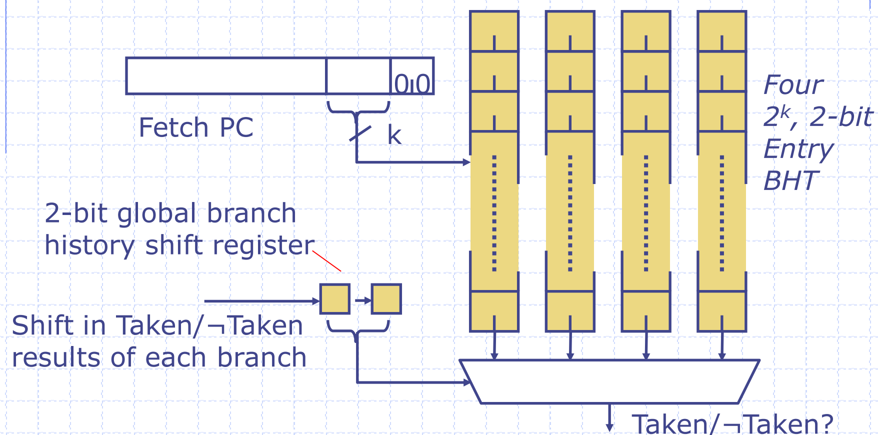
```
if (x[i] < 7) then
        y += 1;
if (x[i] < 5) then
        c -= 4;
```

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)



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 return is usually to the same place
  - However, often one function is called from many distinct call sites!

How well does BTB or BHT work for each of these cases?

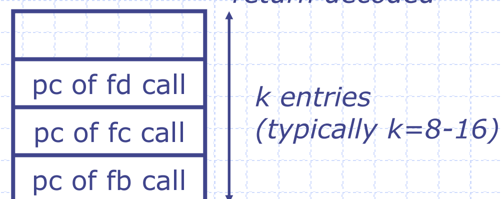
Subroutine Return Stack

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

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

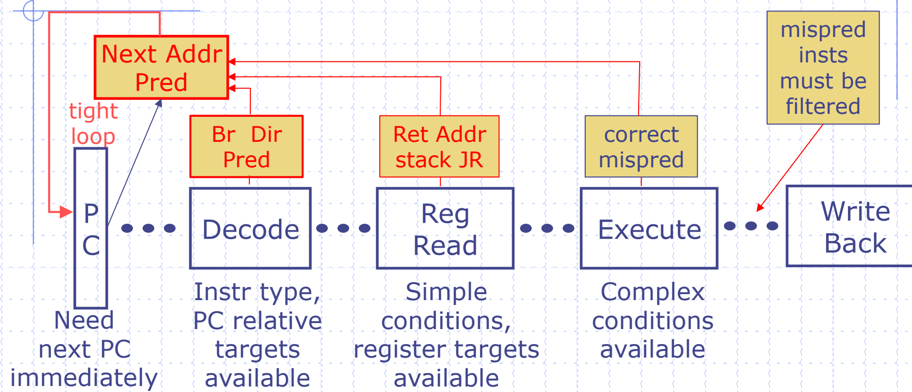
Push call address when function call executed

Pop return address when subroutine return decoded



k entries (typically k=8-16)

Multiple Predictors: BTB + BHT + Ret Predictors



- 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