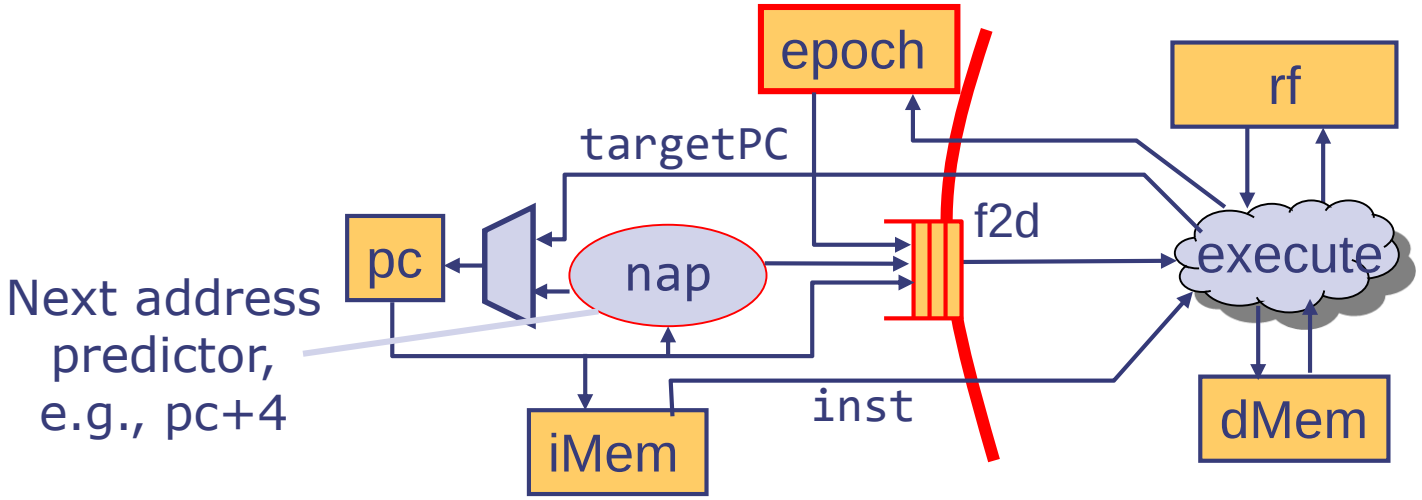
Implementing Processor Pipelines

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

Epoch: a method to manage control hazards



- Add an *epoch* register to the processor state
- The Execute stage changes the *epoch* whenever the pc prediction is wrong and sets the pc to the correct value
- The Fetch stage associates the current *epoch* to every instruction sent to the Execute stage
- The epoch of the instruction is examined when it is ready to execute. If the processor epoch has changed the instruction is thrown away

From multicycle to a Two-Stage Pipeline processor

```
rule doFetch;
    iMem.req(MemReq{op: Ld, addr: pc, data: dwv});
    let ppc = nap(pc);   pc <= ppc;
    f2d.enq(F2D {pc: pc, ppc: ppc, epoch: epoch});
 endrule
```

Can doFetch and doExecute execute concurrently?

```
rule doExecute if (state != LoadWait);
    let inst <- iMem.resp;
    let x = f2d.first; f2d.deq;
    let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
    if (epochD == epoch) begin  // right-path instruction
        Compute eInst from inst
        let mispred = (eInst.nextPC != ppc);
        if (mispred) begin pc <= eInst.nextPC; epoch <= !epoch; end
        Update the state;
        If a memory op, initiate memory req;
        If Ld, go to LoadWait
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...
```

Two-Stage Pipeline processor
Fix1: avoid rule conflict use EHRs

```
rule doFetch;
   iMem.req(MemReq{op: Ld, addr: pc[1], data: dwv});
   let ppc = nap(pc[1]);   pc[1] <= ppc;
   f2d.enq(F2D {pc: pc[1], ppc: ppc, epoch: epoch});
endrule
```

Instantiate pc as an EHR

```
rule doExecute if (state == Execute);
    let inst <- iMem.resp;
    let x = f2d.first; f2d.deq;
    let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
    if (epochD == epoch) begin  // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin pc[0] <= eInst.nextPC;
                           epoch <= !epoch; end
        code to update the state;
        in case of a memory op, initiate memory req and
           in case of Ld go to LoadWait
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...
```

Is this the correct value of epoch?

Two-Stage Pipeline processor
Fix1: avoid rule conflict use EHRs

```
rule doFetch;
   iMem.req(MemReq{op: Ld, addr: pc[1], data: dwv});
   let ppc = nap(pc[1]);   pc[1] <= ppc;
   f2d.enq(F2D {pc: pc[1], ppc: ppc, epoch: epoch[1]});
endrule
```

Instantiate epoch also as an EHR

```
rule doExecute if (state == Execute);
    let inst <- iMem.resp;
    let x = f2d.first; f2d.deq;
    let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
    if (epochD == epoch[0]) begin  // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin pc[0] <= eInst.nextPC;
                           epoch[0] <= !epoch[0]; end
        code to update the state;
        in case of a memory op, initiate memory req and
           in case of Ld go to LoadWait
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...
```

Synthesis results

Processors	Clock (ps)		Benchmarks (Cycles)			
	No Re-timing	Re-timing	gcd	No hazard	Control hazard	Data hazard
Three-cycle	567	457	3508	294	98	288
TwoStage 1st Attempt	701	522	4884	290	113	284
TwoStage EHR	817	615	2022	272	91	271

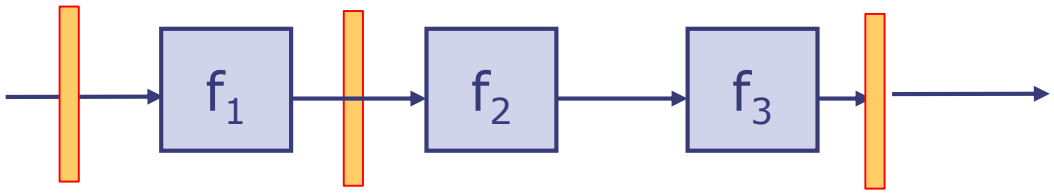
- EHRs reduced the cycle count by eliminating the rule conflict
- But increased the clock period

Message:
1. Exploiting rule concurrency in the common case is essential
2. EHRs are often necessary for concurrency but care is needed because the clock period can get worse

Retiming

Processors	Clock (ps)		Benchmarks (Cycles)			
	No Re-timing	Re-timing	gcd	No hazard	Control hazard	Data hazard
Three-cycle	567	457	3508	294	98	288
TwoStage 1st Attempt	701	522	4884	290	113	284

- *Retiming* moves registers in the datapath to improve timing but preserving the functionality



Suppose $\max\{t_{f1+f2}, t_{f3}\} < \max\{t_{f2+f3}, t_{f1}\}$

Circuit is difficult to analyze after retiming!

Two-Stage Pipeline processor
Fix 2: delay the redirection

```
rule doFetch;
   iMem.req(MemReq{op: Ld, addr: pc[1], data: dwv});
   let ppc = nap(pc[1]);    pc[1] <= ppc;
   f2d.enq(F2D {pc: pc[1], ppc: ppc, epoch: epoch[1]});
endrule
rule doExecute if (state == Execute);
    ...
    if (epochD == epoch[0] ) begin  // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin pc[0] <= eInst.nextPC;
                           epoch[0] <= !epoch[0]; end

        …
endrule
```

- Instead of redirecting the pc and epoch from the execute stage, delay redirection by one clock cycle by moving it into a separate rule
  - This may reduce the critical path delay and increase cycle count in case of redirection

Two-Stage Pipeline processor
Fix 2: move redirection out of Execute - 1

```
rule doExecute if (state == Execute);
    ...
    if (epochD == epoch) begin  // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin pc[0] <= eInst.nextPC;
                           epoch[0] <= !epoch[0]; end
        code to update the state;
        in case of a memory op, initiate memory req and
           in case of Ld go to LoadWait
endrule
rule doLoadWait if (state == LoadWait); ... go to Execute ...
rule doRedirect if (state == Redirect); ... go to Execute ...
```

- In doExecute set the state to Redirect
- Introduce a new doRedirection rule to change epoch and pc

Two-Stage Pipeline processor
Fix 2 – move redirection out of Execute -2

```
rule doExecute if (state == Execute);
    ...
    if (epochD == epoch) begin  // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin state <= Redirect;
                           nextPC <= eInst.nextPC; end
        code to update the state;
        in case of a memory op, initiate memory req and
           in case of Ld go to LoadWait
endrule

rule doLoadWait if (state == LoadWait); ... go to Execute ...

rule doRedirect if (state == Redirect);
     pc[0] <= nextPC; epoch[0] <= !epoch[0];
     state <= Execute;
endrule
                           f2e.deq;
                           let inst <- iMem.resp();
```

Still not correct !

doRedirect must throwaway another wrong path instruction that may have been fetched

- We also need to remember nextPC in a register and pass it to doRedirect

Improved two-stage pipeline

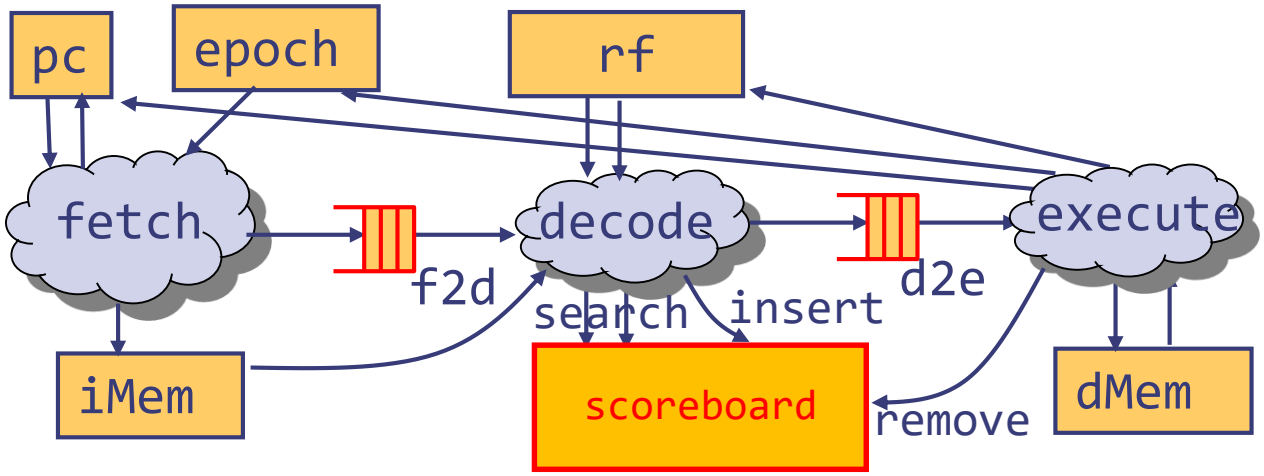
Processors	Clock (ps)		Micro Benchmarks (Cycles)			
	No Re-timing	Re-timing	gcd	No hazard	Control hazard	Data hazard
Three-cycle	567	457	3508	294	98	288
TwoStage 1st Attempt	701	522	4884	290	113	284
TwoStage EHR	817	615	2022	272	91	271
TwoStage DelayRedir	599	524	2711	272	98	271

- Delaying redirection improved the clock period but increased the number of cycles as expected

Processors	Clock (ps)	gcd (cycles)	gcd (ns)
Three-cycle	457	3508	2081
TwoStage 1st Attempt	522	4884	2982
TwoStage EHR	615	2022	1914
TwoStage DelayRedir	524	2711	1625

Three stage pipeline

From L11



- Pipeline the Execute stage by separating decode; should reduce the clock period

Processors	Clock (ps)	gcd	Data hazard
Three-cycle	567	3508	288
TwoStage DelayRedir	599	2711	271
Four-cycle(separate decode and execute)	499	5004	306
ThreeStage Bypass	492	3410	272

how?

Pipeline doExecute
three-stage pipeline

```
rule doFetch; ... endrule
```

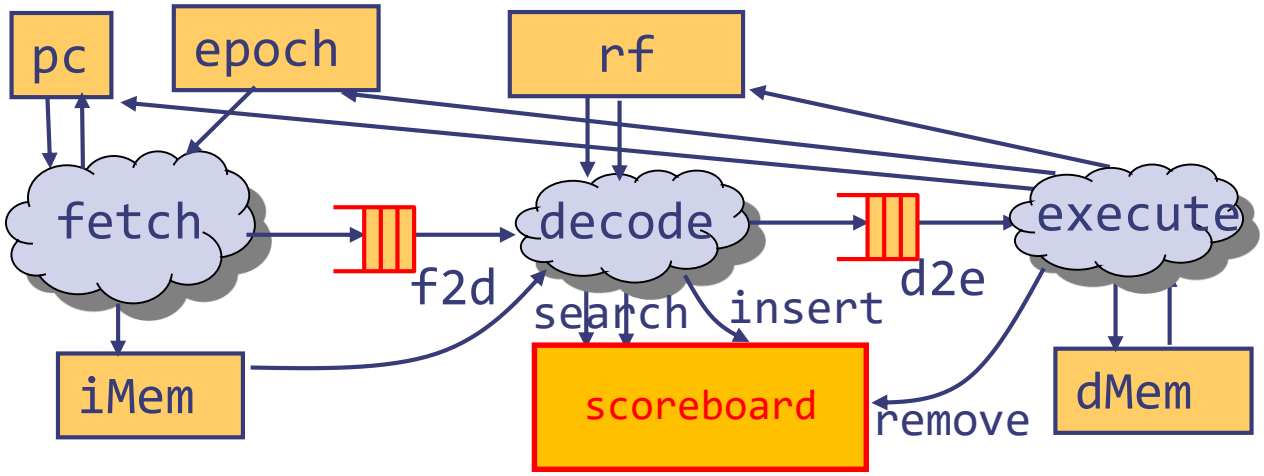
```
rule doDecode;
  let inst <- iMem.resp;
  ... filter wrong-path instructions ...
  ... decode (inst) ...
  ... read rf ...
  d2e.enq(...);
endrule
```

Danger Data Hazards:
doDecode may read stale values

Introduce scoreboard

```
rule doExecute (...);
  let dInst = d2e.first; d2e.deq;
  ... filter wrong-path instructions ...
  ... execute (dInst, rval1,rval2, pc) ...
  ... detect and handle misprediction ...
  ... launch memory instructions if needed ...
  ... save info for the LoadWait step ...
endrule
rule doLoadWait(...) ... endrule
```

Three stage pipeline
a correctness issue



- When an instruction is removed from the scoreboard, its updates to Register File must be visible to the subsequent register reads in Decode
  - `remove` and `wr` should happen simultaneously
  - `search`, and `rd1`, `rd2` should happen simultaneously

doDecode rule

```
rule doDecode;
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq;
  if (x.epoch == epoch) begin
    let dInst = decode(inst); // src1 and src2 are Maybe types;
    // check for data hazard
    if (!(sb.search1(dInst.src1)||sb.search2(dInst.src2))) begin
      read rVal1 and read rVal2 from rf
      sb.insert(dInst.dst); //to stall future inst for data hazard
      enqueue into e2d fifo: pc, ppc, epoch, rVal1, rVal2, dInst
    end
  end
endrule
```

Still not quite correct. Why?

We need to keep the fetched instruction while stalling!

Need a register to hold the fetched instruction while stalling

Fixing the doDecode rule

```
rule doDecode;
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq;
  if (x.epoch == epoch) begin
    let dInst = decode2(inst); // src1 and src2 are Maybe types;
    if (!(sb.search1(dInst.src1)||sb.search2(dInst.src2))) begin
      read rVal1 and read rVal2 from rf
      sb.insert(dInst.dst); //to stall future inst for data hazard
      enqueue into e2d fifo: pc, ppc, epoch, rVal1, rVal2, dInst
      f2d.deq;
    end else begin
      fetchedInst <= inst;  ...
    end
  end else begin  f2d.deq;  ... end
endrule
```

No new instruction should be fetched in the stalled state

stalled instruction must be saved

Execute rule

```
rule doExecute (...);
  ... filter wrong-path instructions ...
  ... execute (dInst, rval1,rval2, pc) ...
  ... detect and handle misprediction ...
  ... launch memory instructions if needed ...
  ... save info for the LoadWait step ...
endrule

rule doLoadWait (...);
  ...
endrule
```

sb.remove has to be inserted whenever an instruction completes execution

Further pipelining

- In the three-stage pipeline (Fetch, Decode, Execute), the Execute stage takes an extra cycle in case of a load
- We can increase the throughput by running the Execute and LoadWait stages concurrently
- Complication: Both Execute and LoadWait stage may want to update the register file and Scoreboard concurrently
- It is a good practice to update state from only one stage in the pipeline, therefore, we can move the RF update from Execute to LoadWait. (In such a case LoadWait state is often referred to as the Write-Back stage)
  - But this will introduce extra bubbles to resolve RAW hazards
  - Bypassing becomes essential to reduce these extra bubbles

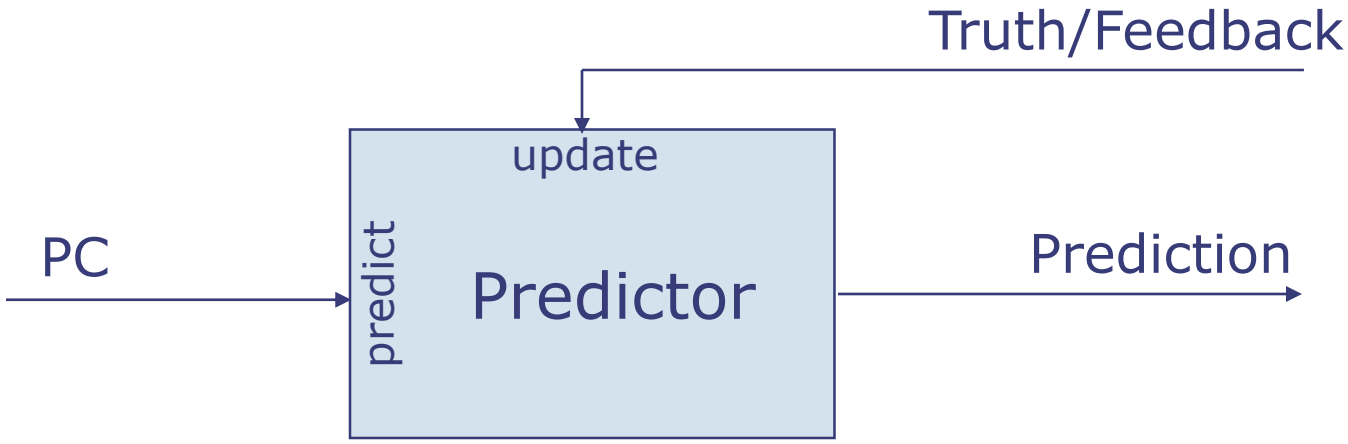
FIFOs

- Normally we use Pipeline FIFOs in pipelines, which results in the following ordering between the stages:
  - WB < EX < DEC < Fetch
- For maximum flexibility is scheduling use Conflict Free FIFOs, which will not force the stages to be ordered from WB to Fetch

Two more techniques to improve processor performance

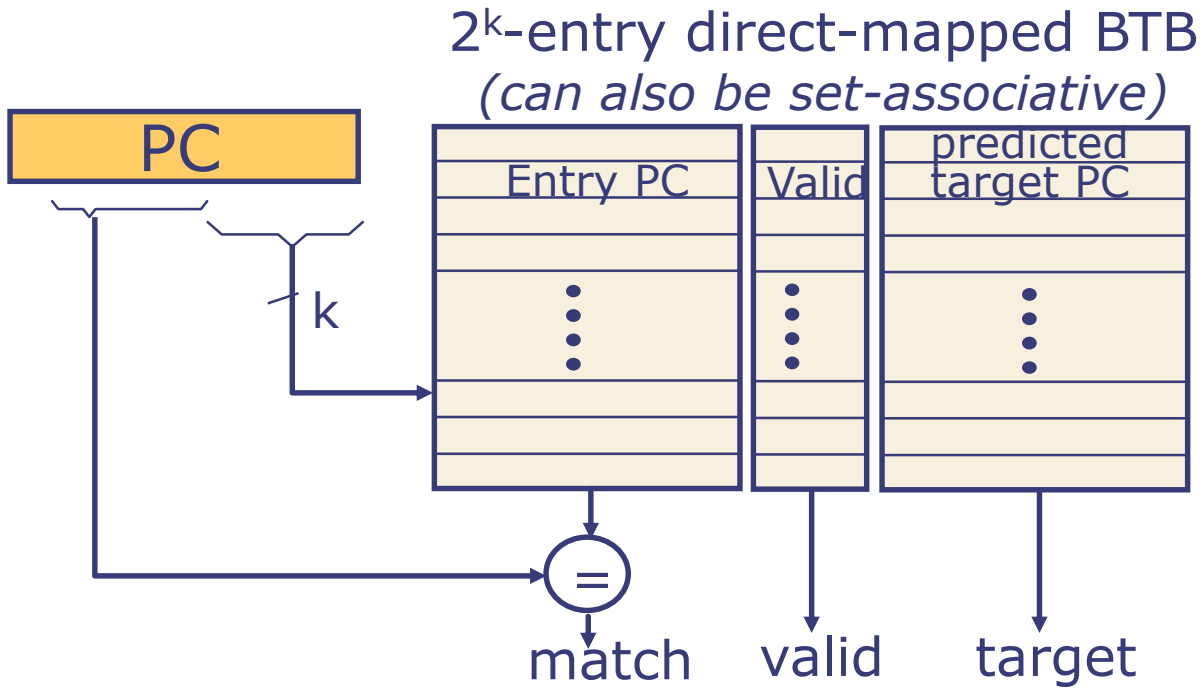
- *Branch prediction* dynamically changes the next address prediction based on the past behavior of the program. Fewer wrong-path instructions reduces the number of pipeline bubbles
- *Bypasses*, i.e., extra data paths between the producer of a value and its consumer can reduce the number of stalls (that is, the number of cycles) by providing

Dynamic Branch Prediction
Learning from past behavior



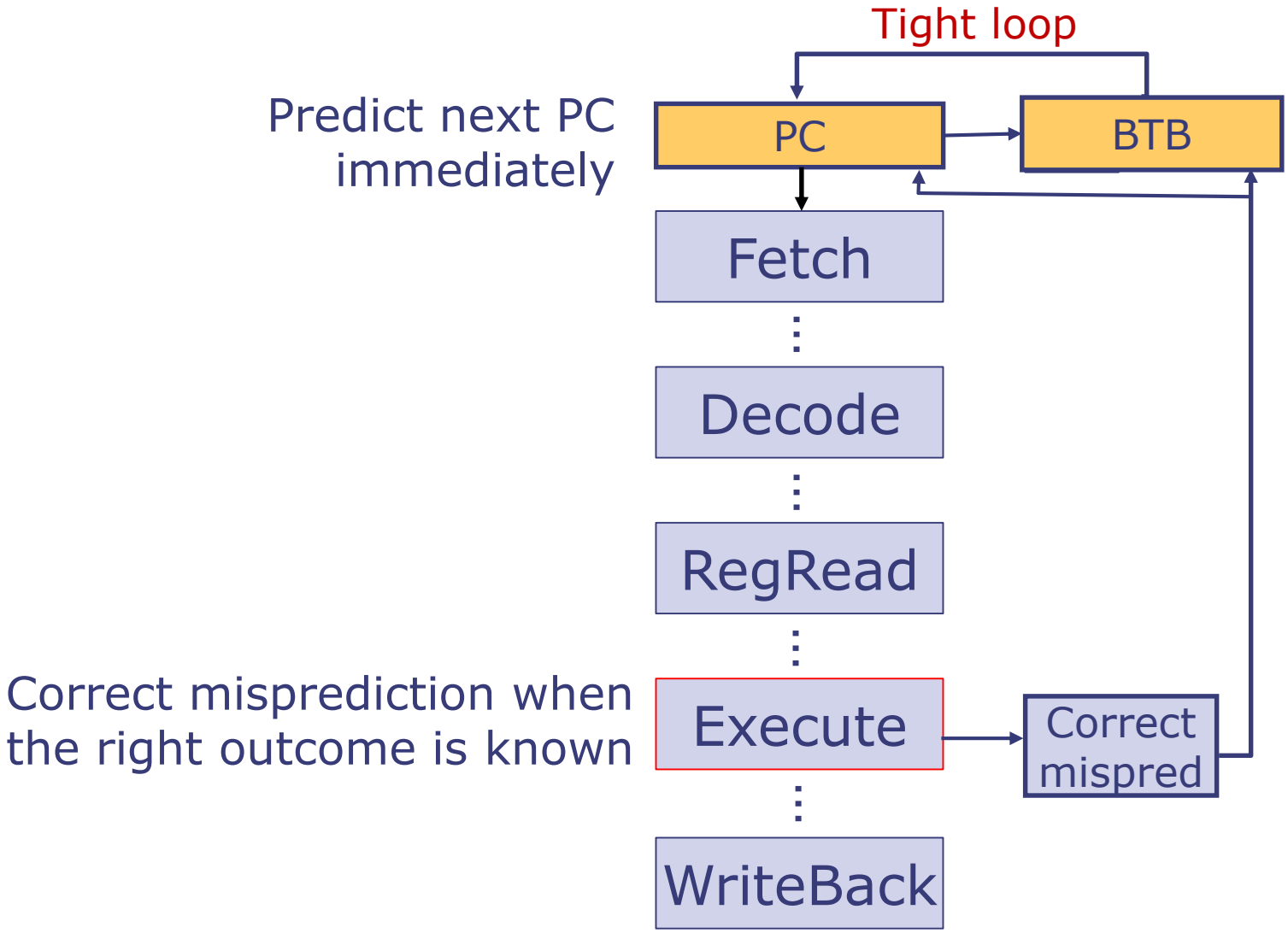
- The way a branch resolves may be a good predictor of the way the branch will resolve when executed next
  - Record every branch resolution in a data structure and consult the data structures at the fetch stage

Next Address prediction: Branch Target Buffer (BTB)

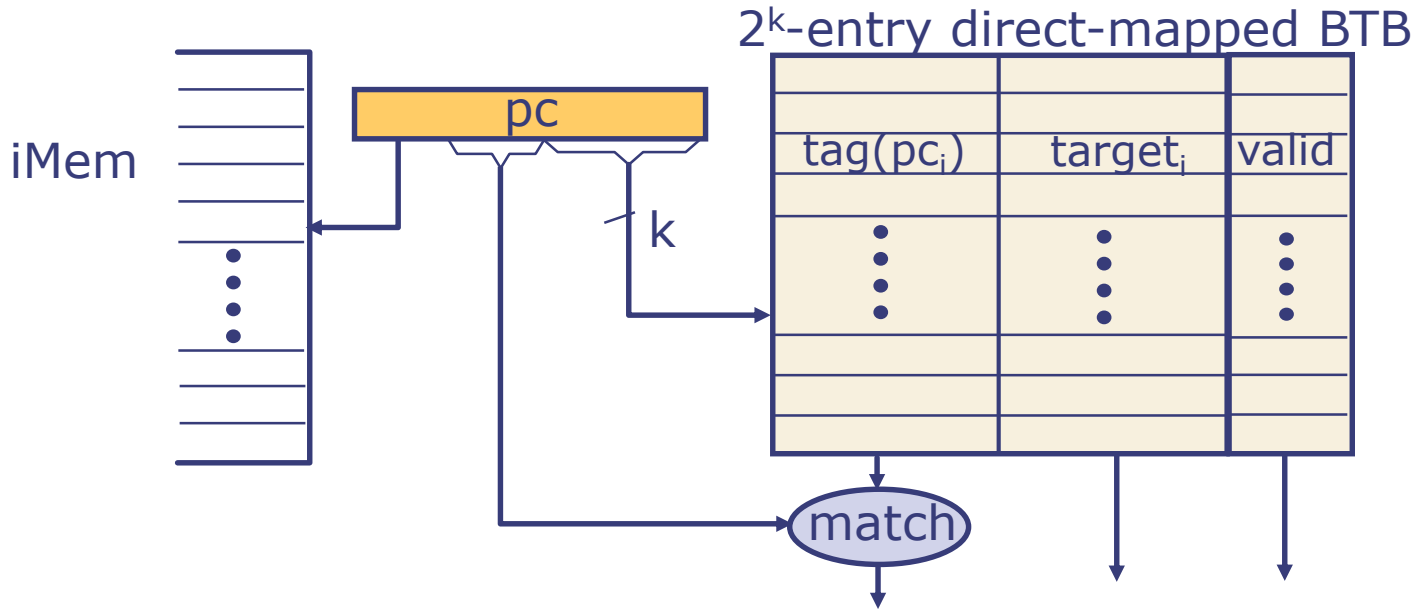


- BTB is a cache for targets: Remembers last target PC *for taken branches and jumps*
  - If hit, use stored target as predicted next PC
  - If miss, use PC+4 as predicted next PC
  - After target is known, update if the prediction was wrong

Integrating the BTB in the Pipeline



BTB Implementation Details



- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits
- Even small BTBs are very effective!

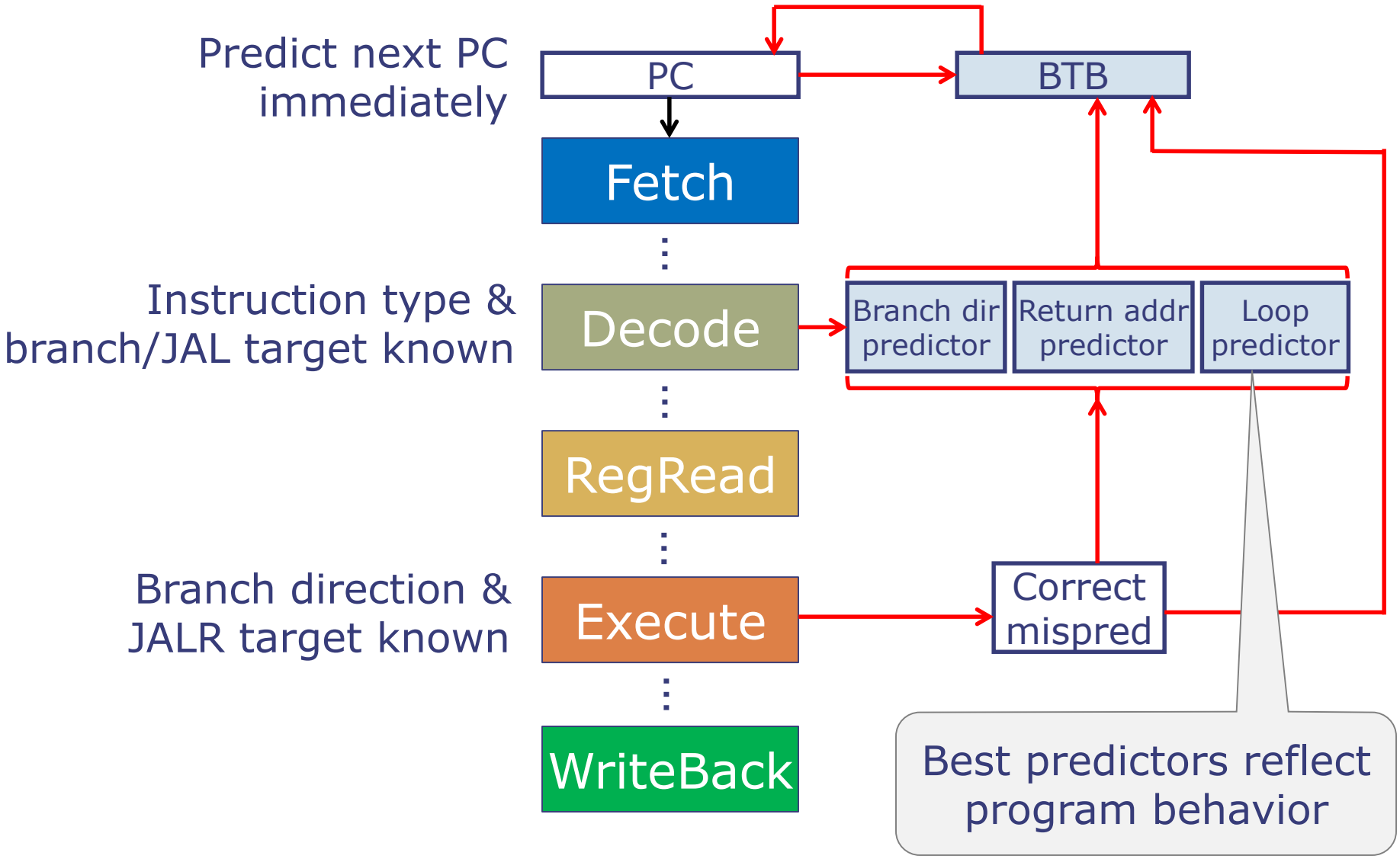
BTB Interface

```
interface BTB;
  method Addr predict(Addr pc);
  method Action update(Addr pc, Addr nextPC,
                       Bool taken);
endinterface
```

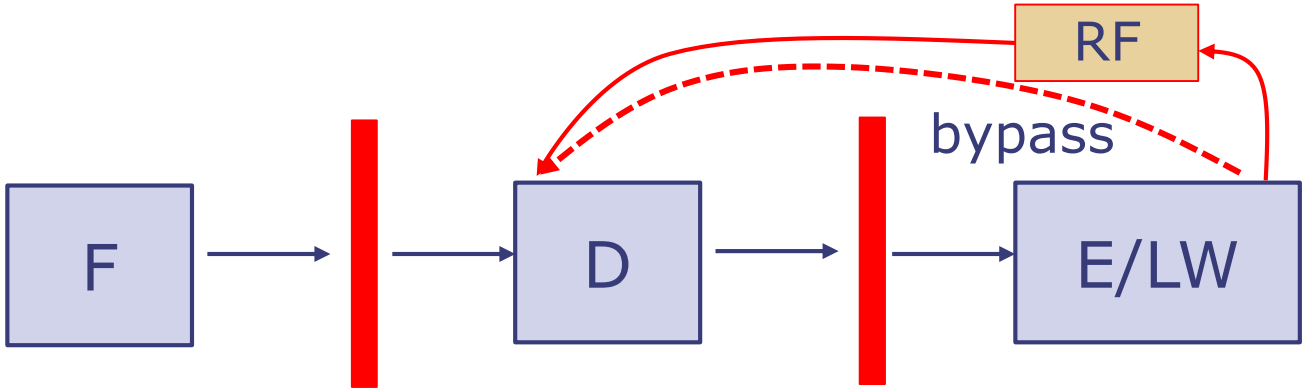
- *predict:* Simple lookup to predict nextPC in Fetch stage
- *update:* On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted

BTB is a good way to improve the performance; if we use small BTB tables there is no danger of increasing the clock period

Modern Processors Combine Multiple Specialized Predictors



Bypassing



- Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer
- Bypassing introduces new combinational paths and this can increase combinational delay (and hence the clock period) and area
- The effectiveness of a bypass is determined by how often it is used
- For correctness, both RF and ScoreBoard must be bypassed.

Normal vs Bypass Register File

```
module mkRFile(RFile);
  Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));

  method Action wr(RIndx rindx, Data data);
    if(rindx!=0) rfile[rindx] <= data;
  endmethod
  method Data rd1(RIndx rindx) = rfile[rindx];
  method Data rd2(RIndx rindx) = rfile[rindx];
endmodule
```

{rd1, rd2} < wr

Can we design a bypass register file so that:

wr < {rd1, rd2}

Bypass Register File using EHR

```
module mkBypassRFile(RFile);
  Vector#(32,Ehr#(2, Data)) rfile <-
                              replicateM(mkEhr(0));

  method Action wr(RIndx rindx, Data data);
    if(rindex!=0) (rfile[rindex])[0] <= data;
  endmethod
  method Data rd1(RIndx rindx) = (rfile[rindx])[1];
  method Data rd2(RIndx rindx) = (rfile[rindx])[1];
endmodule
```

wr < {rd1, rd2}

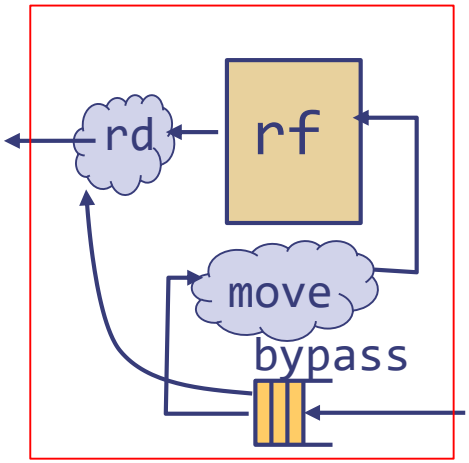
Bypass Register File
with external bypassing

```
module mkBypassRFile(BypassRFile);
  RFile    rf <- mkRFile;
  SFifo#(1, RIdxData#(Bit#(5), Bit#(32)))
           bypass <- mkBypassSFifo;
  rule move;
       …take a entry out of the bypass fifo and
        write it into rf…

  method Action wr(Bit#(5) rindx, Bit#(32) data);
       … bypass.enq…

  method Bit#(32) rd1(RIndx rindx);
       … look for the value in the bypass fifo,
         if not found then read rf…

endmodule
```



Sfifo = Searchable Fifo

```
typedef struct {Bit#(5) index; Bit#(32) data}
RIdxData deriving (Bits);
```

Summary

- Modern processors rely on a handful of techniques to improve performance
  - Deep pipelines → Multi-GHz frequency
  - Wide (superscalar) pipelines → Multiple instructions/cycle
  - Out-of-order execution → Reduce impact of data hazards
  - Branch prediction → Reduce impact of control hazards

- However, one also needs to improve the memory system at the same time to realize full benefits
  - Store buffers
  - Non-blocking memory, i.e., several outstanding misses
  - Fetching multiple words