Implementing Processor Pipelines

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

This lecture should help you with the design project

Please do the course evaluation
https://registrar.mit.edu/classes-grades-evaluations/subject-evaluation
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
Two-Stage Pipeline processor
first attempt

```pseudocode
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

We want doExecute to execute unless we go into the LoadWait state

```pseudocode
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 <= 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 ...
# Two-Stage pipeline analysis

## Cycle count

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>Benchmarks (Cycles)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Retiming</td>
<td>Re-timing</td>
<td>gcd</td>
<td>No hazard</td>
<td>Control hazard</td>
<td>Data hazard</td>
</tr>
<tr>
<td>Three-cycle</td>
<td>873</td>
<td>593</td>
<td>3508</td>
<td>294</td>
<td>98</td>
<td>288</td>
</tr>
<tr>
<td>TwoStage 1st Attempt</td>
<td>1022</td>
<td>611</td>
<td>4884</td>
<td>290</td>
<td>113</td>
<td>284</td>
</tr>
</tbody>
</table>

- **Why did the cycle count for gcd not decrease?**

```plaintext
rule doFetch;
    ... pc <= ppc; ...
endrule
rule doExecute if (state != LoadWait);
    ...
    let mispred = eInst.nextPC != ppc;
    if (mispred) begin pc <= eInst.nextPC; epoch <= !epoch; end
    code to initiate memory ops and go to LoadWait if necessary
endrule
```

- **Why did the cycle counts for gcd and control hazard increase over Three-cycle?**

Unnecessary (wrong-path) instruction fetches

---

Take home: Explain
Two-Stage pipeline analysis
Clock period

- The clock period increases but not after retiming
- *Retiming* moves registers in the datapath to improve timing but preserving the functionality

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>Benchmarks (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Retiming</td>
<td><em>re-timing</em></td>
</tr>
<tr>
<td></td>
<td>gcd</td>
<td>No hazard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control hazard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data hazard</td>
</tr>
<tr>
<td>Three-cycle</td>
<td>873 593</td>
<td>3508 294 98</td>
</tr>
<tr>
<td>TwoStage 1st Attempt</td>
<td>1022 611</td>
<td>4884 290 113</td>
</tr>
</tbody>
</table>

Suppose \( \max\{t_{f_1+f_2}, t_{f_3}\} < \max\{t_{f_2+f_3}, t_{f_1}\} \)

_not easy to analyze the circuit after retiming!
Two-Stage Pipeline processor
Fix 1: 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

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 ...
```
### Improved two-stage pipeline

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>Benmarks (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Retiming</td>
<td>Retiming</td>
</tr>
<tr>
<td>Three-cycle</td>
<td>873</td>
<td>593</td>
</tr>
<tr>
<td>TwoStage 1st Attempt</td>
<td>1022</td>
<td>611</td>
</tr>
<tr>
<td>TwoStage EHR</td>
<td>1133</td>
<td>947</td>
</tr>
</tbody>
</table>

- 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
Two-Stage Pipeline processor
Fix 2: delay the redirection

- Instead of redirecting the pc and epoch from the execute stage, redirection by one clock cycle by moving 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

```verilog
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; f2e.deq;
  let inst <- iMem.resp();
endrule

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

Still not correct!
doRedirect must throwaway another wrong path instruction that may have been fetched
## Improved two-stage pipeline

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>Benchmarks (Cycles)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Retiming</td>
<td>Re-timing</td>
<td>gcd</td>
<td>No hazard</td>
<td>Control hazard</td>
<td>Data hazard</td>
</tr>
<tr>
<td>Three-cycle</td>
<td>873</td>
<td>593</td>
<td>3508</td>
<td>294</td>
<td>98</td>
<td>288</td>
</tr>
<tr>
<td>TwoStage 1st Attempt</td>
<td>1022</td>
<td>611</td>
<td>4884</td>
<td>290</td>
<td>113</td>
<td>284</td>
</tr>
<tr>
<td>TwoStage EHR</td>
<td>1133</td>
<td>947</td>
<td>2022</td>
<td>272</td>
<td>91</td>
<td>271</td>
</tr>
<tr>
<td>TwoStage DelayRedir</td>
<td>914</td>
<td>599</td>
<td>2711</td>
<td>272</td>
<td>98</td>
<td>271</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>gcd (cycles)</th>
<th>gcd (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three-cycle</td>
<td>593</td>
<td>3508</td>
<td>2081</td>
</tr>
<tr>
<td>TwoStage 1st Attempt</td>
<td>611</td>
<td>4884</td>
<td>2982</td>
</tr>
<tr>
<td>TwoStage EHR</td>
<td>947</td>
<td>2022</td>
<td>1914</td>
</tr>
<tr>
<td>TwoStage DelayRedir</td>
<td>599</td>
<td>2711</td>
<td>1625</td>
</tr>
</tbody>
</table>
Pipeline the Execute stage by separating decode; essential to meet the clock period requirement in the project.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Clock (ps)</th>
<th>gcd</th>
<th>Data hazard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three-cycle</td>
<td>593</td>
<td>3508</td>
<td>288</td>
</tr>
<tr>
<td>TwoStage DelayRedir</td>
<td>599</td>
<td>2711</td>
<td>271</td>
</tr>
<tr>
<td>Four-cycle (separate decode and execute)</td>
<td>454</td>
<td>5004</td>
<td>306</td>
</tr>
<tr>
<td>ThreeStage Bypass</td>
<td>488</td>
<td>3410</td>
<td>272</td>
</tr>
</tbody>
</table>
Pipeline doExecute
three-stage pipeline

```plaintext
rule doFetch; ...
endrule

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

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
```

Danger Data Hazards:
doDecode may read stale values

Introduce scoreboard
Three stage pipeline
a correctness issue

From L20

- 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

```verilog
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;
        // 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
```

decode2 is the modified decoder

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

```plaintext
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

stalled instruction must be saved
No new instruction should be fetched in the stalled state
```
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
Dynamic Branch Prediction
Learning from past behavior

- Branch prediction is another way to reduce the number of pipeline bubbles
- The way a branch resolves may be a good predictor of the way it will resolve at the next execution
Predicting the Target Address: 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

2^k-entry direct-mapped BTB (can also be set-associative)
Integrating the BTB in the Pipeline

Predict next PC immediately

Tight loop

Correct misprediction when the right outcome is known
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 in the design project**
Extras
Modern Processors Combine Multiple Specialized Predictors

Predict next PC immediately

Instruction type & branch/JAL target known

Branch direction & JALR target known

Best predictors reflect program behavior
Summary

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