Data Hazards in Pipelined Processors

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

Code for the lecture is available on the course website under the code tab
Plan

- General considerations in instruction pipelining
- Revisit L19 to understand rule-splitting and concurrency issues
- Reduce the combinational delay of the Execute stage by separating *decode* and *execute* into two stages
Pipeline speed and throughput

Clock speed is determined by the longest combinational path. Dividing the critical path into multiple steps improves the clock speed but may decrease the throughput. Pipelining steps can improve the throughput.
Pipeline design Issues

Pipelining increases throughput but there is tension because pipelining may require
- speculation and squashing, i.e., killing instructions
- stalls in handling dependent instructions

Each squash or stall decreases the throughput
- the cost of squashing and stalls increases with the depth of pipeline
The role of bypassing

- Bypassing is a technique to reduce the number of stalls by providing extra data paths between the producer of a value and its consumer.
- Bypassing eliminates a stall but may increase the combinational delay (and hence the clock period) and area.
- The effectiveness of a bypass is determined by how often it is used.
  - Compiler can also affect the effectiveness of a bypass by increasing the distance between the instruction that uses a value from the instruction that creates it.
Estimating the combinational delay of a rule

```plaintext
rule ra (p(x)); begin y <= f(x); end endrule

clock period? > max \{t_p, t_f\}

rule rb; begin
  a <- mem.resp; b = rf.rd(5); y <= f(a, b);
end endrule

clock period? > max \{t_{\text{mem.resp}}, t_{\text{rf.rd}}\} + t_f

It is difficult to estimate \(t_{\text{mem.resp}}\) without looking inside the mem module design.
- if the memory module sends a response only after storing it in an internal register the delay would be minimal, otherwise some estimate of the delay has to be provided.

Bypassing creates new combinational paths across rules and may make the combinational delay worse.
Performance is determined by the following factors:

- E & WB steps of instruction is done one-by-one (E & WB rules are mutually exclusive)
- Memory may take more than one cycle to respond
- Propagation delay = \( \max\{F, E, \text{WB}_{\text{normal}}, \text{WB}_{\text{redirect}}\} \)
- No. of bad predictions (each redirection loses at least 1 cycle)
Rule Splitting to enhance concurrency

```
rule doFetch ; ... pc <= pc+4 ... endrule
rule doEx_WB (exState == WriteBack)

let eInst = exInst; let data = eInst.data;
if (eInst.iType == LOAD) data <- dMem.resp();
if (isValid(eInst.rd)) rf.wr(fromMaybe(? , eInst.rd), data);
exState <= Execute;
if (exInst.nextPc != exPpc); // misprediction?
    begin pc <= eInst.nextPc; epoch <= !epoch; end
endrule

rule doEx_WB_Normal (exState == WriteBack &&
    exInst.nextPc == exPpc); // normal path
endrule
rule doEx_WB_Redirect (exState == WriteBack &&
    exInst.nextPc != exPpc); // misprediction
pc <= eInst.nextPc; epoch <= !epoch;
endrule
```

We have speeded up the common case
Today’s plan

Split Execute into two stages

- Separate E step into two stages:
  - Decode and register-file-read
  - Execute – including the initiation of memory instructions
- Will require us to introduce several new concepts in pipelining
- May actually improve the performance if Execute step had the longest propagation delay (decode+execute)
Two stage pipeline (from L19)

rule doFetch;
    iMem.req(MemReq{op: Ld, addr: pc, data: ?});
    let ppc = pc + 4;  pc <= ppc;
    f2d.enq( F2D{pc: pc, ppc: ppc, epoch: epoch} );
endrule

rule doEx_Ex(exState == Execute);  doDecode
    let inst <- iMem.resp;
    ... Filter wrong-path instructions ... 
    ... decode (inst) ... 
    ... read rf ... 
endrule

rule doExecute
    ... execute (dInst, rval1,rval2, pc) ... 
    ... launch memory instructions if needed ... 
    ... save info for the WB step ... 
endrule
Three stage pipeline
First attempt

```plaintext
rule doFetch; ...
rule doDecode;
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq; ... extract pcD, ppc, epochD...
  if (epochD == epoch) begin
    let dInst = decode(inst);
    let rVal1 = rf.rd1(dInst.rs1);
    let rVal2 = rf.rd2(dInst.rs2);
    d2e.enq(D2E {pcD, ppc, epochD, dInst, rVal1, rVal2});
  end
endrule

rule doExecute (exState=Execute);
  let x = d2e.first; d2e.deq; ... extract fields ...
  let eInst = execute(dInst, rVal1, rVal2, pcD);
  ... launch memory instructions if required
  exInst <= eInst; exPpc <= ppc; exState <= WriteBack;
endrule
```

introduce d2e FIFO

Not quite correct. Why?

plus the L19 write-back rules
Three stage pipeline

Problem-data hazard

F → D → Ex → WB

I₀ R₁ ← R₂+R₃
I₁ R₄ ← R₁+R₂
I₂ ...

Fetch Decode Ex WB

Fetch
Decode
Ex
WB

I₀
I₁
I₂

I₀ must be stalled until I₀ updates the register file, i.e., the data hazard disappears

⇒ need a mechanism to stall

The data hazard will disappear as pipeline drains

Could Ex be executing a wrong-path instruction?
Three stage pipeline
wrong-path instruction

I_0, Br ...
I_1
I_2 ...

Fetch
Decode
Ex
Wb_{normal}
Wb_{redirect}

I_2 will be killed in Decode because of epoch but I_1 won’t get killed in the current design
must check if the instruction in E stage is wrong-path and then take appropriate action

May 1, 2018
Dealing with data hazards

- Keep track of instructions in the pipeline and determine if the register values to be fetched are stale, i.e., will be modified by some older instruction still in the pipeline. This condition is referred to as a read-after-write (RAW) hazard.
- Stall the Fetch from dispatching the instruction as long as RAW hazard prevails.
- RAW hazard will disappear as the pipeline drains.

Scoreboard: A data structure to keep track of the instructions in the pipeline beyond the Fetch stage.
Data Hazard

- Data hazard arises when a source register of the fetched instruction matches the destination register of an instruction already in the pipeline.
- Both the source and destination registers must be Valid for a hazard to exist.

```plaintext
function Bool isFound
    (Maybe#(RIndex) x, Maybe#(RIndex) y);
    if(x matches Valid .xv &&& y matches Valid .yv
        &&& yv == xv)
        return True;
    else return False;
endfunction
```

Our current decoder does not record the validity of the source fields, however, the decoder can be changed easily.
Scoreboard: Keeping track of instructions in execution

Scoreboard: a data structure to keep track of the destination registers of the instructions beyond the fetch stage

- **method insert**: inserts the destination (if any) of an instruction in the scoreboard when the instruction is decoded
- **method search1(src)**: searches the scoreboard for a data hazard
- **method search2(src)**: same as search1
- **method remove**: deletes the oldest entry when an instruction commits
Scoreboard in the pipeline

To avoid structural hazards, scoreboard must have two search ports
new Decode + Reg Read rule

rule doDecode;
  let inst <- iMem.resp;
  let x = f2d.first; f2d.deq;
  if (x.epoch == epoch) begin
    let dInst = decode2(inst); // rs1 and rs2 are Maybe types;
    // check for data hazard
    if (!(sb.search1(dInst.rs1) || sb.search2(dInst.rs2))) begin
      let rVal1 = rf.rd1(fromMaybe(?, dInst.rs1));
      let rVal2 = rf.rd2(fromMaybe(?, dInst.rs2));

      sb.insert(dInst.rd); // stall future inst for data hazard
      d2e.enq(D2E {pc: x.pc, ppc: x.ppc, epoch: x.epoch,
                    dInst: dInst, rVal1: rVal1, rVal2: rVal2 });
    end
  end
endrule

Still not quite correct. Why?
We need to keep the fetched instruction while stalling!
Instruction(s) can either wait inside or outside iMem
Three stage pipeline

Change the iMem interface: iMem.firstResp returns the value without deleting it; a new method iMem.deq deletes it.

Don’t dequeue from iMem until the hazard is resolved.
Three Stage

Decode + Reg Read rule

rule doDecode;
let inst = iMem.firstResp; iMem.deq;
let x = f2d.first; f2d.deq;
if (x.epoch == epoch) begin
  let dInst = decode2(inst); // rs1 and rs2 are Maybe types;
  // check for data hazard
  if !(sb.search(dInst.rs1) || sb.search(dInst.rs2)) begin
    let rVal1 = rf.rd1(fromMaybe(?, dInst.rs1));
    let rVal2 = rf.rd2(fromMaybe(?, dInst.rs2));
    f2d.deq; iMem.deq;
    sb.insert(dInst.rd); // stall future inst for data hazard
    d2e.enq(D2E {pc: x.pc, ppc: x.ppc, epoch: x.epoch,
                  dInst: dInst, rVal1: rVal1, rVal2: rVal2 });
  end else begin
  f2d.deq; iMem.deq; end
endrule

Now the instruction will be stalled here until the hazard resolves
Three Stage

Execute rule

```
rule doEx_Execute (state == Execute);
    d2e.deq;
    let x = d2e.first;
    if (x.epoch == epoch) begin
        ... extract dInst, rVal1, rVal2, pcD, ppc ...
        let eInst = execute(dInst, rVal1, rVal2, pcD);
        ... issue memory instructions ...
        exInst <= eInst; exPpc <= ppc;
        exPoisoned <= false;
    end
    else begin
        exPoisoned <= true;
    end
    exState <= WriteBack;
endrule
```

The instruction we are throwing away made an entry in sb. This will mess up bookkeeping!

Instead of throwing away the instruction we can poison it and pass it to WriteBack

An alternative solution would be to remove the wrong-path instruction from sb here
Write-back rule

```verilog
drule doEx_WB (exState == WriteBack);
    sb.remove; exState <= Execute;
    if (!exPoisoned) begin
        let eInst = exInst; let data = eInst.data;
        if (eInst.iType == LOAD) data <= dMem.resp;
        if (isValid(eInst.rd)) rf.wr(fromMaybe(?, eInst.rd), data);
        if (exInst.nextPc != exPpc) begin // redirect
            pc <= eInst.nextPc; epoch <= !epoch;
        end
    end
endrule
```

This rule conflicts with doFetch

split this rule
Three Stage

Write-back rules

```plaintext
rule doEx_WB_Normal ((exState == WriteBack) && !exPoisoned && (exInst.nextPc == exPpc)); // no misprediction
...
```

```plaintext
rule doEx_WB_Redirect ((exState == WriteBack) && !exPoisoned && (exInst.nextPc != exPpc)); // misprediction detected
...
```

```plaintext
rule doEx_WB_Poisoned ((exState == WriteBack) && exPoisoned);
...
```

Now only doEx_WB_Redirect will conflict with doFetch
Suppose we also pipeline Execute and WriteBack rules

```plaintext
rule doExecute;
    let x = d2e.first; d2e.deq;
    if (x.epoch == epoch) begin
        ...
        let eInst = execute(dInst, rVal1, rVal2, pcD);
        ...
        Launch memory instructions ...
        e2w.enq ({eInst: eInst; ppc: ppc; poisoned: false});
    end else
    e2w.enq ({eInst: ?; ppc: ?; poisoned: false});
endrule

rule doWriteBack;
    let x = e2w.first; e2w.deq; sb.remove;
    if (!x.poisoned) begin ...
        if (isValid(eInst.rd)) rf.wr(fromMaybe(?, eInst.rd), data);
        if (eInst.nextPc != x.ppc) begin // redirect
            pc <= eInst.nextPc;
            epoch <= !epoch;
        end
    end
endrule
```

introduce e2w FIFO

Not quite correct
Suppose we also pipeline Execute and WriteBack rules

```
rule doExecute;
  let x = d2e.first; d2e.deq;
  if (x.epoch == epoch) begin
    ...
    let eInst = execute(dInst, rVal1, rVal2, pcD);
    ... Launch memory instructions ...
    e2w.enq ({eInst: eInst; ppc: ppc; poisoned: false});
    if (eInst.nextPc != x.ppc) begin // redirect
      pc <= eInst.nextPc; epoch <= !epoch; end
  end else
    e2w.enq ({eInst: ?; ppc: ?; poisoned: false});
endrule
```

Correct but now doExecute conflicts with Fetch

```
rule doWriteBack;
  let x = e2w.first; e2w.deq; sb.remove;
  if (!x.poisoned) begin ...
    if (isValid(eInst.rd)) rf.wr(fromMaybe(? , eInst.rd), data);
  end
endrule
```

You can split it?
Next Lecture

- Branch prediction
- Bypassing