Constructive Computer Architecture
Realistic Memories and Caches

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
The use of magic memories (combinational reads) makes such designs unrealistic.
**Magic Memory Model**

- Reads and writes are always completed in one cycle
  - a Read can be done any time (i.e. combinational)
  - If enabled, a Write is performed at the rising clock edge (*the write address and data must be stable at the clock edge*)

In a real SRAM or DRAM the data will be available several cycles after the address is supplied
Memory System

- View iMem as a request/response system and split the fetch rule into two rules – one to send a request and the other to receive the response.
- Insert a FIFO (f12f2) to hold the pc address of the instructions being fetched.
  - Can be the same as f2d.
- Similar idea applies to dMem.

```
          Epoch
            ↓
              ↓
            PC
            ↓
        nap
        ↓
    f12f2
    ↓
 iMem
    ↓
first,deq
    ↓
 f2d
    ↓
 Decode
```

Assume iMem behaves like a FIFO.
Connecting 2-Stage-pipeline to req/res memory

**doExecute < doFetch**

```verilog
erule doFetch;  
  let instF = iMem.req(pc[1]);  
  let ppcF = nap(pc[1]);  
  let dInst = decode(instF);  
  let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);  
  if(!stall) begin  
    ...fetch register values  
    d2e.enq(Decode2Execute{pc: pc[1], ppc: ppcF,  
      dIinst: dInst, epoch: epoch[1],  
      rVal1: rVal2, rVal2: rVal2});  
    sb.insert(dInst.rDst); pc[1] <= ppcF; end  
endrule

rule doExecute;  
  ...the same as before ...  
  if (x.ppc != nextPC) begin pc[0] <= eInst.addr;  
    epoch[0] <= !epoch[0]; end  
endrule
```
Connecting 2-Stage-pipeline to Req/Res memory  

**doExecute** < **doFetch**

```
rule fetch;
   iMem.enq(pc[1]);
   let ppcF = nap(pc[1]); pc[1] <= ppcF;
   f2d.enq(Fetch2Decode(pc:pc[1], ppc:ppcF, epoch:epoch[1]));
endrule

rule decode;
   let inst = iMem.first; let x = f2d.first;
   let dInst = decode(inst);
   let stall = sb.search1(dInst.src1)|| sb.search2(dInst.src2);
   if(!stall)
      begin
         ...fetch register values
         d2e.enq(Decode2Execute{pc: x.pc, ppc: x.ppc,
                                dIinst: dInst, epoch: x.epoch,
                                rVal1: rVal1, rVal2: rVal2});
         sb.insert(dInst.rDst); iMem.deq; f2d.deq
      end
endrule
```

What is the advantage of nap in fetch1 vs fetch2?

We can also drop the instruction if epoch has changed.

must be done only if not stalling.

October 18, 2017  http://csg.csail.mit.edu/6.175  L14-6
Dropping instructions

```verilog
rule decode;

let inst = iMem.first;  let x = f2d.first;
if (epoch[?] != x.inEp) begin iMem.deq; f2d.deq end

else begin
let dInst = decode(inst);
let stall = sb.search1(dInst.src1) || sb.search2(dInst.src2);
if (!stall) begin
...fetch register values
  d2e.enq(Decode2Execute{pc: x.pc, ppc: x.ppc,
    dIinst: dInst, epoch: x.epoch,
    rVal1: rVal1, rVal2: rVal2});
  sb.insert(dInst.rDst); iMem.deq; f2d.deq end end endrule
```

Are both 0 and 1 correct?
Yes, but 1 is better
Data access in the execute stage

- Execute rule has to be split too in order to deal with multicycle memory system
- How should the functions of execute be split across rules
  - call exec
  - initiate memory ops, wait for load results
  - redirection
  - register update
  - scoreboard updates
Transforming the Execute rule – first attempt

```verilog
rule doExecute;

let x = d2e.first;
let dInstE = x.dInst; let pcE = x.pc; let inEp = x.epoch;
let rVal1E = x.rVal1; let rVal2E = x.rVal2;
if (epoch == inEp) begin
    let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
    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(fromMaybe(?, eInst.dst), eInst.data);
    let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
    if (x.ppc != nextPC) begin pc[0] <= eInst.addr;
        epoch[0] <= !epoch[0]; end
end

d2e.deq; sb.remove;
endrule
```

October 18, 2017

http://csg.csail.mit.edu/6.175
Execute rule: first attempt

rule execute;
  let x = d2e.first;
  let dInstE = x.dInst; let pcE = x.pc; let inEp = x.epoch;
  let rVal1E = x.rVal1; let rVal2E = x.rVal2;
  if (epoch[1] != inEp) begin sb.remove; end
else begin
  let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
  e2w.enq(Exec2WB(eInst:eInst, pc:pcE, epoch:inEp));
  if (eInst.iType == Ld)
    dMem.enq(MemReq{op:Ld, addr:eInst.addr, data:?});
  else if (eInst.iType == St) begin
    dMem.enq(MemReq{op:St, addr:eInst.addr, data:eInst.data}); end
end
  d2e.deq;
endrule
Writeback rule

```
rule writeback;
    let x = e2d.first; let pcE = x.pc;
    let eInst = x.eInst; let inEp = x.epoch;
    if (epoch[0] = inEp) begin
        if (isValid(eInst.dst)) begin
            let data = eInst.iType == Ld ? dMem.first : eInst.data;
            rf.wr(fromMaybe(?, eInst.dst), data);
        end
        if (eInst.iType == Ld) dMem.deq;
        let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
        if (x.ppc != nextPC) begin
            pc[0] <= eInst.addr;
            epoch[0] <= !epoch[0];
        end
    end
    sb.remove;
    e2w.deq
endrule
```

notice, we have assumed that St does not get a response

October 18, 2017 http://csg.csail.mit.edu/6.175 L14-11
Problems with the first attempt

- sb.remove is being called from both execute and writeback
  - out of order removals – correctness
  - simultaneous removals – concurrency

- St that was initiated in execute could be invalidated in writeback (wrong path instruction); consider a branch followed by a store
  - a store, once it is sent to the memory, cannot be recalled

Let us move redirection from writeback to execute and sb.remove from execute to writeback
Dropping vs 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.
rule execute;
    let x = d2e.first; ...
    if (epoch[0] != inEp) begin
        e2w.enq(Invalid); d2e.deq;
    end
else begin
    let eInst = exec(dInstE, rVal1E, rVal2E, pcE);
    if (eInst.iType == Ld)
        dMem.enq(MemReq{op:Ld, addr:eInst.addr, data:?});
    else if (eInst.iType == St) begin
        dMem.enq(MemReq{op:St, addr:eInst.addr,
                        data:eInst.data}); end
    let nextPC = eInst.brTaken ? eInst.addr : pcE + 4;
    if (x.ppc != nextPC) begin
        pc[0] <= eInst.addr;
        epoch[0] <= !epoch[0]; end
    e2w.enq(Valid Exec12Exec2(eInst:eInst, pc:pcE));
    d2e.deq;
end
Writeback rule  second attempt

\textbf{rule writeback;}

\hspace{1em} \textbf{let} \ vx = \text{e2w.first;}

\hspace{1em} \textbf{if} (vx \text{ matches tagged Valid .x}) \textbf{begin}

\hspace{2em} \textbf{let} \ pcE=x.pc; \ \textbf{let} \ eInst=x.eInst;

\hspace{2em} \textbf{if} (\text{isValid(eInst.dst)}) \textbf{begin}

\hspace{3em} \textbf{let} \ data = \text{eInst.iType==Ld ? dMem.first: eInst.data;}

\hspace{3em} \textbf{rf.wr(fromMaybe(?, eInst.dst), data)};

\hspace{3em} \textbf{end}

\hspace{2em} \textbf{if}(\text{eInst.iType == Ld}) \text{dMem.deq;}

\hspace{1em} \textbf{end}

\hspace{1em} \textbf{sb.remove; e2w.deq;}

\textbf{endrule}
Observations

- sb.remove is called only from exec2 ==> no concurrency issues
- Redirection is done from exec1 ==> better for performance
- St was initiated in exec1 and cannot be squashed by any older instruction in exec2 or the exec12exec2 fifo
- stall will work correctly in fetch2 because the scoreboard is not updated until the reg-file is also updated
Memory Hierarchy

size: RegFile << SRAM << DRAM
latex: RegFile << SRAM << DRAM
bandwidth: on-chip >> off-chip

On a data access:

hit (data ∈ fast memory) ⇒ low latency access
miss (data ∉ fast memory) ⇒ long latency access (DRAM)
Managing of fast storage

- **User managed** *Scratchpad* memory:
  - ISA is aware of the storage hierarchy; separate instructions are needed to access different storage levels

- **Automatically managed** *Cache* memory:
  - Programmer has little control over how data moves between fast and slow memory
  - Historically very successful (painless for the programmer)
Why do caches work

- Temporal locality
  - if a memory location is referenced at time t then there is very high probability that it will be referenced again in the near future, say, in the next several thousand instructions (frequently observed behavior)
    - *working set* of locations for an instruction window

- Spatial locality
  - if address x is referenced then addresses x+1, x+2 etc. are very likely to be referenced in the near future
    - consider instruction streams, array and record accesses
Inside a Cache

A cache line usually holds more than one word to:
- exploit spatial locality
- transport large data sets more efficiently
- reduce the number of tag bits needed to identify a cache line

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Byte</td>
</tr>
<tr>
<td>304</td>
<td>Byte</td>
</tr>
<tr>
<td>6848</td>
<td></td>
</tr>
</tbody>
</table>

Data from locations 100, 101, ...