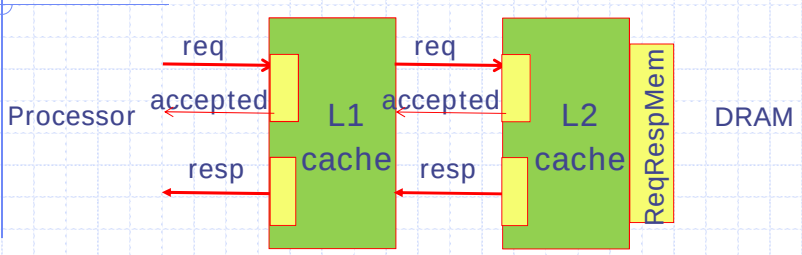
# Realistic Memories and Caches – Part III

Li-Shiuan Peh
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

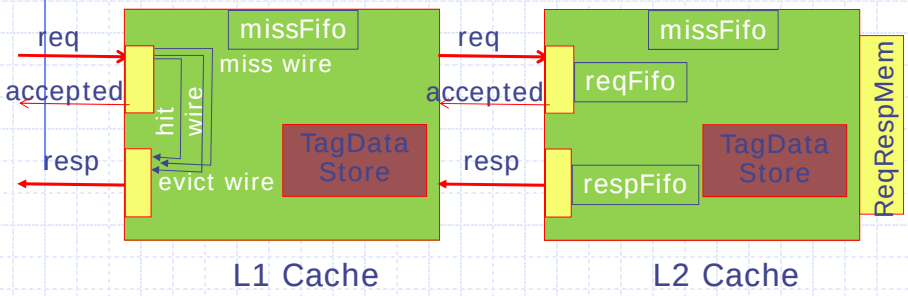


# 2-level Data Cache Interface (0,n)




```
interface mkL2Cache;
    method ActionValue#(Bool) req(MemReq r);
    method ActionValue#(Maybe#(Data)) resp;
endinterface
```

# 2-level Data Cache Internals




Let's build an ***inclusive*** L2 with our earlier direct-mapped L1



# L2 cache: hit case

```
rule procReq(!missFifo.notEmpty);
    let req = reqFifo.first;
    let hitOrMiss <- store.tagLookup(req.addr);
    if(hitOrMiss.hit)
    begin
      let line <- store.dataAccess(
                  getCacheDataReqLine(hitOrMiss.id, req));
      if(req.op == Ld)
        respFifo.enq(unpack(pack(line.Line)));
      reqFifo.deq;
    end
      :
      :
endrule
```

# dirty case

```
    else if (hitOrMiss.dirty)
    begin
      let line <- store.dataAccess(CacheDataReq{id:
hitOrMiss.id, reqType: tagged InvalidateLine});
      mem.reqEnq(MemReq{op: St, addr: hitOrMiss.addr,
                        data: unpack(pack(line.Line))});
    end
    else
```

What about corresponding line in L1?



# miss case

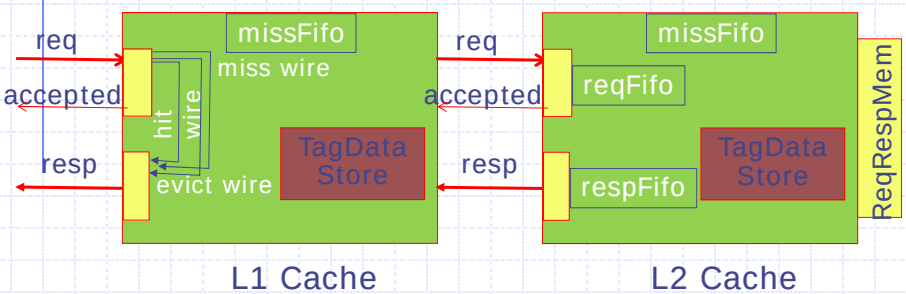
```
else
    begin
      missFifo.enq(tuple2(hitOrMiss.id, req));
      req.op = Ld;
      mem.reqEnq(req);
      reqFifo.deq;
    end
```

# Process miss from memory

```
rule procMiss;
   match {.id, .req} = missFifo.first;
   Line line = req.op == Ld? unpack(pack
     (mem.respFirst)) : unpack(pack(req.data));
   Bool dirty = req.op == Ld? False: True;
   let l <- store.dataAccess(CacheDataReq{id: id,
            reqType: tagged FillLine (tuple3(line,
            req.addr, dirty))});
   if(req.op == Ld)
     respFifo.enq(unpack(pack(line)));
   missFifo.deq;
   mem.respDeq;
 endrule
```



# How will inclusive L2 need to change to interface with a n-way set-associative L1?



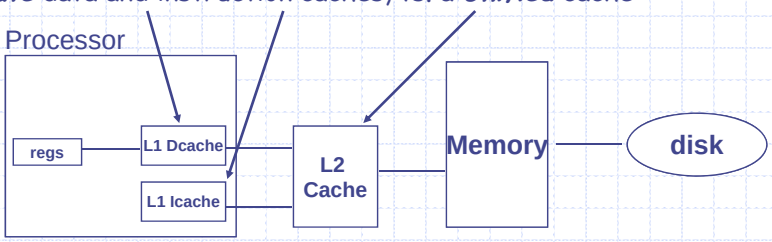

*When a L2 cache line is evicted, the corresponding L1 cache line needs to be invalidated*

*- Interface: + Invalidate information*

*- L1 cache logic: tagLookup, invalidate*

## Multi-Level Cache Organizations

***Separate*** data and instruction caches, vs. a ***Unified*** cache




Inclusive vs. Non-Inclusive vs. Exclusive



## Write-Back vs. Write-Through Caches in Multi-Level Caches

**Write back**

- Writes only go into top level of hierarchy
- Maintain a record of "dirty" lines
- Faster write speed (only has to go to top level to be considered complete)

**Write through**

- All writes go into L1 cache and then also write through into subsequent levels of hierarchy
- Better for "cache coherence" issues
- No dirty/clean bit records required
- Faster evictions

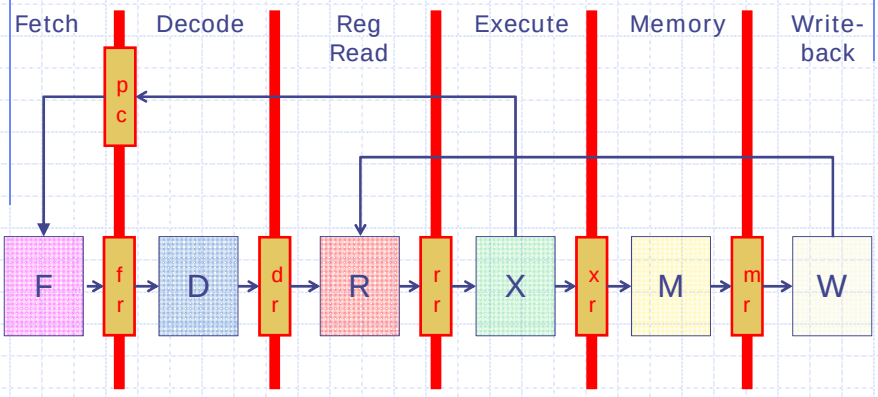
# Summary

- Choice of cache interfaces
  - Combinational vs non-combinational responses
  - Blocking vs non-blocking
  - FIFO vs non-FIFO responses
- Many possible internal organizations
  - Direct Mapped and n-way set associative are the most basic ones
  - Writeback vs Write-through
  - Write allocate or not
- Multi-level caches: Likely organizations?
  - Associativity, Block Size?

*next integrating into the pipeline*

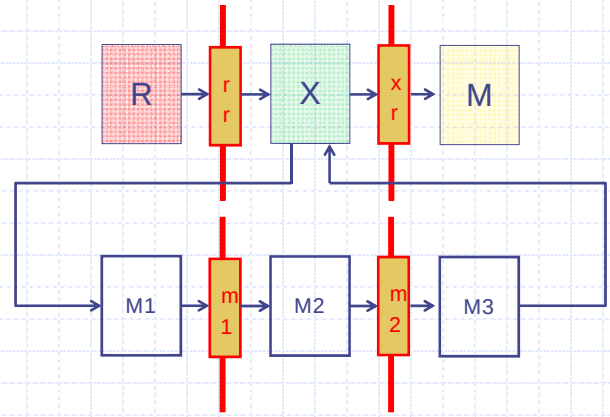


# Recap: Six Stage Pipeline




Writeback feeds back directly to ReadReg rather than through FIFO

# Multi-cycle execute




Incorporating a multi-cycle operation into a stage



# Multi-cycle function unit

```
module mkMultistage(Multistage);
  State1 m1 <- mkReg();  State2 m2 <- mkReg();

  method Action request(Operand operand);
    m1.enq(doM1(operand));
  endmethod

  rule s1;
    m2.enq(doM2(m1.first)); m1.deq();
  endrule

  method ActionValue Result response();
     return doM3(m2.first); m2.deq();
  endmethod
endmodule
```

Perform first stage of pipeline

Perform middle stage(s) of pipeline

Perform last stage of pipeline and return result

# Single/Multi-cycle pipe stage

```
rule doExec;
 ... initialization
  let done = False;
  if (! waiting) begin
    if(isMulticycle(...)) begin
      multicycle.request(...); waiting <= True;
    end else begin
      result = singlecycle.exec(...); done = True;
    end
  end else begin
    result <- multicycle.response();
    done = True; waiting <= False;
  end
  if (done)
    ...finish up
endrule
```

If not busy start multicycle operation, or...

...perform single cycle operation

If busy, wait for response

Finish up if have a result



# Adding a cache (multi-cycle unit!) into the pipeline

- Architectural state:

```
InstCache   iCache    <- mkInstCache(dualMem.iMem);
ReqRespMem  l2Cache   <- mkL2Cache(dualMem.dMem);
DataCache   dCache    <- mkDataCache(l2Cache);
```

- Which stages need to be modified?

# Instruction fetch stage

- Processor sends request to I$

```
if(!inFlight)
      inFlight <- iCache.req(fetchPc);
```

- Processor receives response from I$

```
let instResp <- iCache.resp;
     if(instResp matches tagged Valid .d)
          :
```



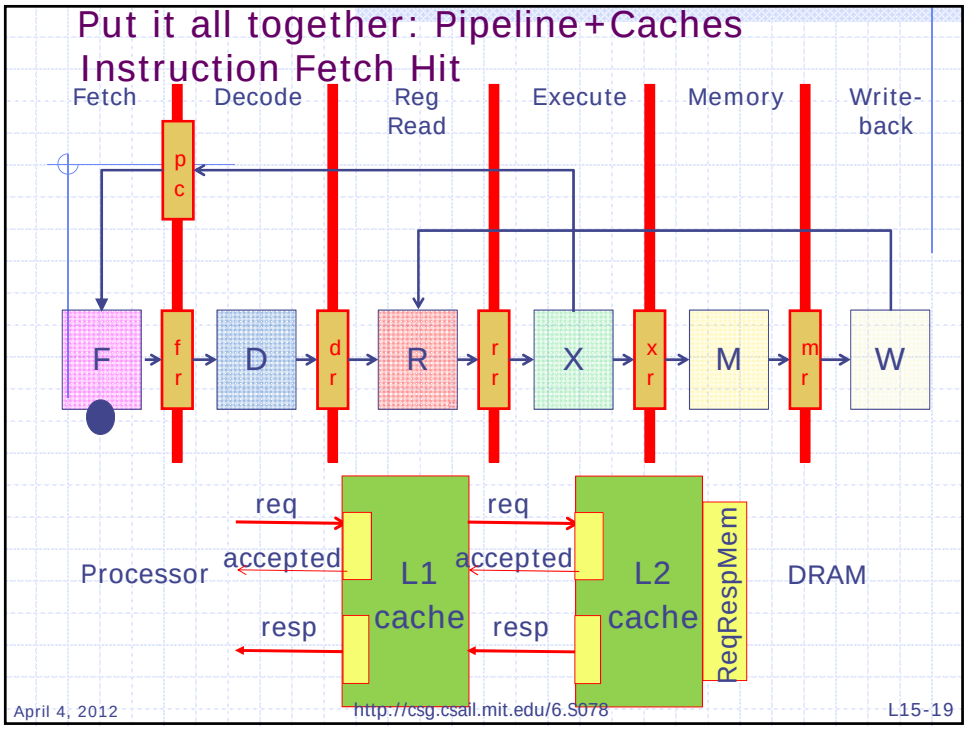
# Memory read stage

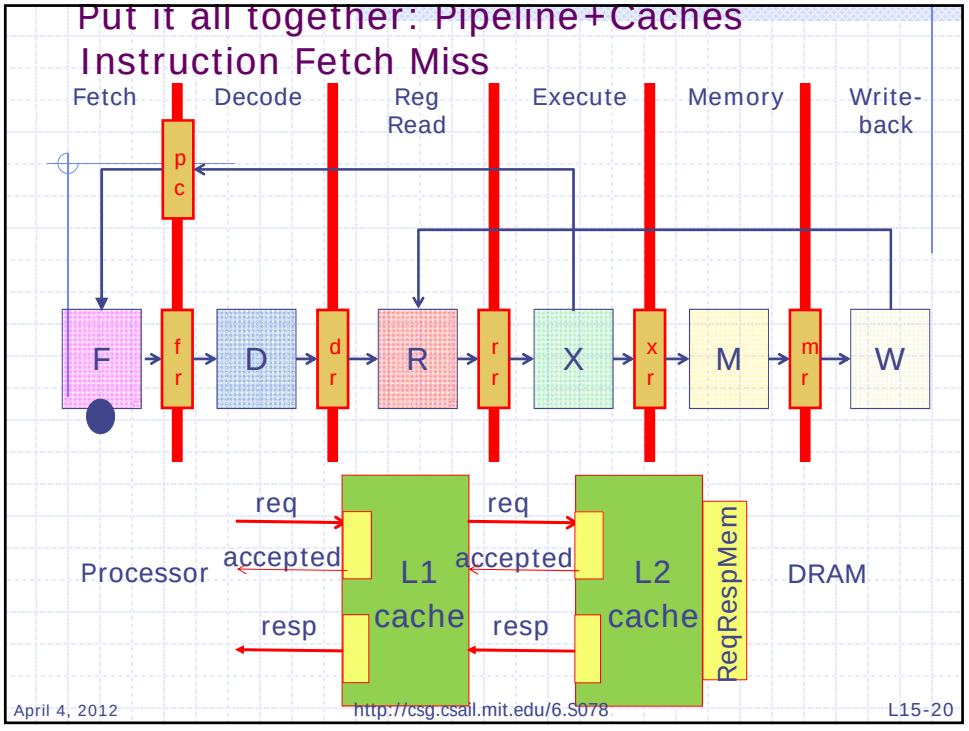
- Processor sends request to D$

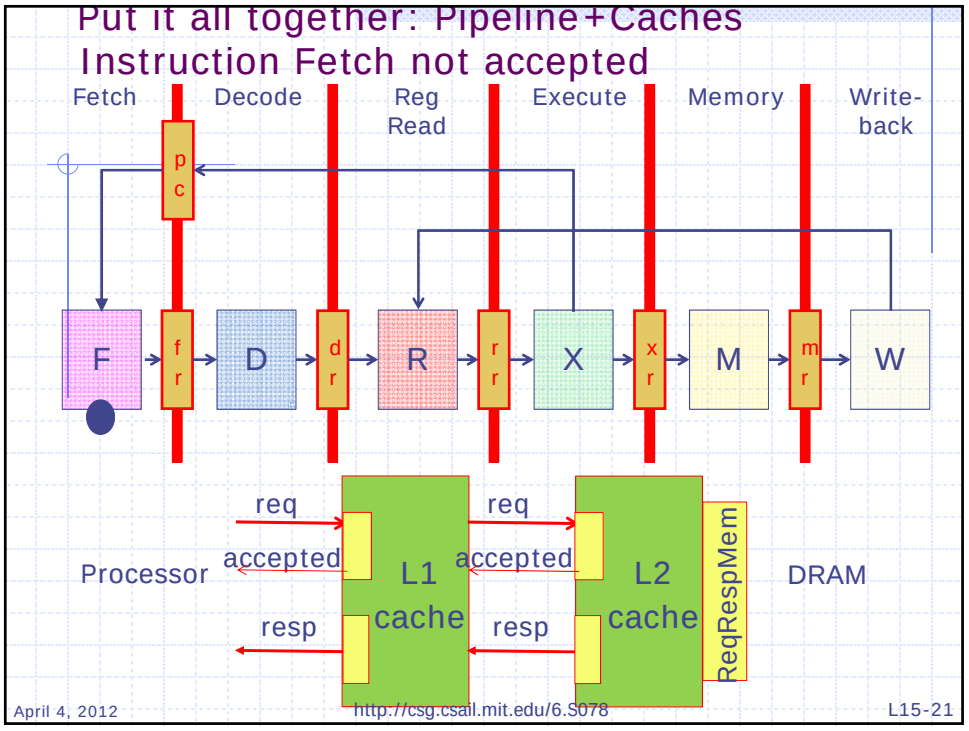
```
if(! poisoned &&
   (decInst.i_type==Ld || decInst.i_type==St) &&
   ! memInFlight && mr.notFull)
      inFlight <-
dCache.req(MemReq{op:execInst.itype==Ld ? Ld :
St, addr:execInst.addr, data:execInst.data});
```

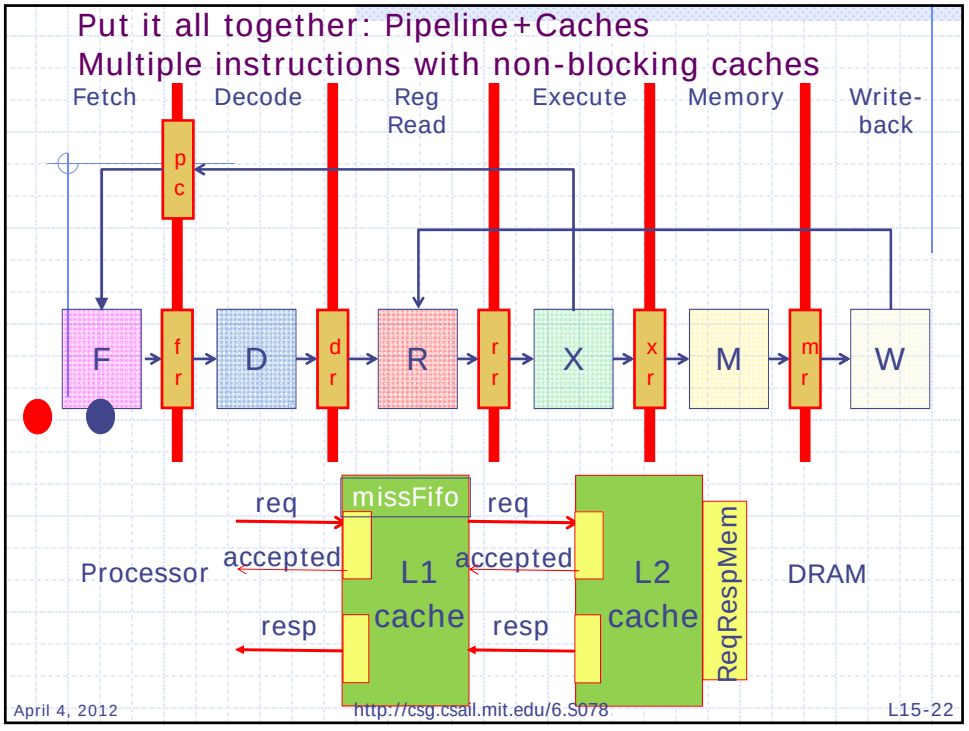
- Processor receives response from D$

```
let data <- dCache.resp;
 if(! poisoned &&
   (decInst.i_type == Ld ||
    decInst.i_type == St    ))
   if(isValid(data))
```


Put it all together: Pipeline+Caches
Instruction Fetch Hit
Fetch
Decode
Reg Read
Execute
Memory
Write-back
pc
F
fr
D
dr
R
rr
X
xr
M
mr
W
req
accepted
resp
Processor
L1 cache
req
accepted
resp
L2 cache
ReqRespMem
DRAM
April 4, 2012
http://csg.csail.mit.edu/6.S078
L15-19


Put it all together: Pipeline+Caches
Instruction Fetch Miss
Fetch
Decode
Reg Read
Execute
Memory
Write-back
pc
F
fr
D
dr
R
rr
X
xr
M
mr
W
req
accepted
resp
Processor
L1 cache
req
accepted
resp
L2 cache
ReqRespMem
DRAM
April 4, 2012
http://csg.csail.mit.edu/6.S078
L15-20


Put it all together: Pipeline+Caches
Instruction Fetch not accepted
Fetch
Decode
Reg Read
Execute
Memory
Write-back
pc
F
fr
D
dr
R
rr
X
xr
M
mr
W
req
accepted
resp
L1 cache
req
accepted
resp
L2 cache
ReqRespMem
Processor
DRAM
April 4, 2012
http://csg.csail.mit.edu/6.S078
L15-21


Put it all together: Pipeline+Caches
Multiple instructions with non-blocking caches
Fetch
Decode
Reg Read
Execute
Memory
Write-back
pc
F
fr
D
dr
R
rr
X
xr
M
mr
W
req
missFifo
accepted
resp
L1 cache
req
accepted
resp
L2 cache
ReqRespMem
Processor
DRAM
April 4, 2012
http://csg.csail.mit.edu/6.S078
L15-22

# Where are the caches?