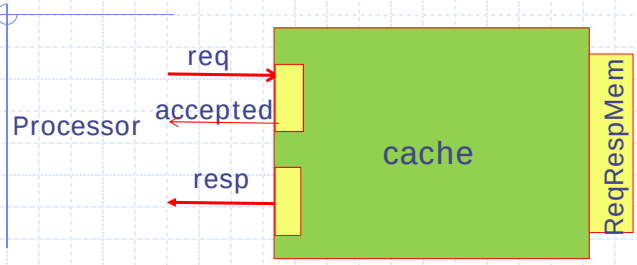
Realistic Memories and Caches – Part II

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

Data Cache – Interface (0,n)

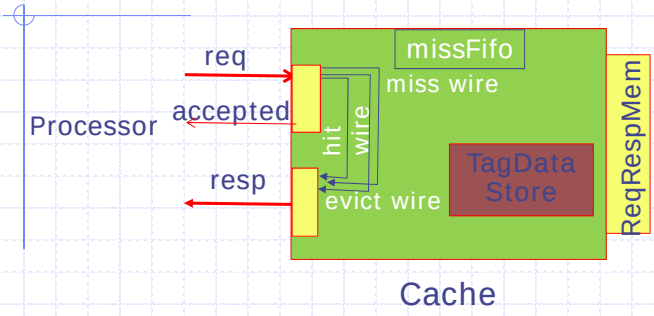


Denotes if the request is accepted this cycle or not

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

Resp method should be invoked every cycle to not drop values

Data Cache – Internals (new)




```
interface TagDataStore;
  method ActionValue#(HitOrMiss) tagLookup(Addr addr);

  method ActionValue#(CacheResp) dataAccess(CacheDataReq req);
endinterface
```

Direct mapped caches

Data Cache – code structure

```
module mkDataCache#(ReqRespMem mem)(DataCache);

  TagDataStore store <- mkTagDataStore;

  FIFOF#(Tuple2#(CacheIdentifier, MemReq)) missFifo <-
                                  mkSizedFIFOF(missFifoSize);
  RWire#(Tuple2#(CacheIdentifier, MemReq))  hitWire <- mkRWire;
  RWire#(Tuple2#(CacheIdentifier, Addr))  evictWire <- mkRWire;
  RWire#(Tuple2#(CacheIdentifier, MemReq)) missWire <- mkRWire;

  function CacheDataReq getCacheDataReqWord(CacheIdentifier id,
                           MemReq r);endfunction

  method Action#(Bool) req(MemReq r) … endmethod;
  method ActionValue#(Data) resp …  endmethod;
endmodule
```

TagDataStore array of a direct-mapped cache

```
module mkTagDataStore(TagDataStore);

   RegFile#(LineIndex, Tag) tagArray <-
                                     mkRegFileFull;
   Vector#(DCacheRows, Reg#(Bool)) valid <-
                          replicateM(mkReg(False));
   Vector#(DCacheRows, Reg#(Bool)) dirty <-
                          replicateM(mkReg(False));
   RegFile#(LineIndex, Line) dataArray <-
                                     mkRegFileFull;
```

TagDataStore: taglookup

```
method ActionValue#(HitOrMiss) tagLookup(Addr addr);
  let lineIndex = getLineIndex(addr);
  let tag = tagArray.sub(lineIndex);

  if(valid[lineIndex] && tag == getTag(addr))
    return HitOrMiss{hit: True, id:CacheIdentifier{
                    lineIndex: lineIndex,
                    wordIndex: getWordIndex(addr)},
                       addr: ?, dirty: ?};
  else
    return HitOrMiss{hit: False, id:CacheIdentifier{
                    lineIndex: lineIndex,
                    wordIndex: ?},
            addr: getAddr(tag, lineIndex, 0),
           dirty: valid[lineIndex] && dirty[lineIndex]};
endmethod
```

TagDataStore: dataAccess

```
method ActionValue#(CacheResp)
                        dataAccess(CacheDataReq req);
   let id = req.id;
   case (req.reqType) matches
     tagged ReadWord:
       :
     tagged WriteWord .w:
       :
     tagged ReadLine:
       :
     tagged FillLine {.l, .addr, .d}:
       :
     tagged InvalidateLine:
       :
   endcase
 endmethod
```

Data Cache requests

```
method ActionValue#(Bool) req(MemReq r);
   $display("mem r %d %h %h", r.op, r.addr, r.data);
   let hitOrMiss <- store.tagLookup(r.addr);
   if(hitOrMiss.hit)
   begin
     hitWire.wset(tuple2(hitOrMiss.id, r));
     $display("Hit %d %h %h", r.op, r.addr, r.data);
     return True;
   end
   else if (hitOrMiss.dirty)
   begin
     $display("Miss");
     if (mem.reqNotFull)
     begin
       $display("eviction");
       evictWire.wset(tuple2(hitOrMiss.id, hitOrMiss.addr));
     end
     return False;
   end
```

Data Cache requests continued

```
   else if (mem.reqNotFull && missFifo.notFull)
      begin
        $display("fill");
        missWire.wset(tuple2(hitOrMiss.id, r));
        return True;
      end
      else
        return False;
    endmethod
```

Data Cache responses (hitwire)

```
method ActionValue#(Maybe#(Data)) resp;
 if(hitWire.wget matches tagged Valid {.id, .req})
   begin
     $display("hit response");
     let word <- store.dataAccess(
                        getCacheDataReqWord(id, req));
     return tagged Valid (word.Word);
   end
```

Data Cache responses (evictwire)

```
else if (evictWire.wget matches tagged Valid {.id, .addr})
   begin
     $display("resp evict");
     let line <- store.dataAccess(CacheDataReq{id: id,
                                 reqType: InvalidateLine});
      $display("resp writeback");
      mem.reqEnq(MemReq{op: St, addr: addr,
                        data: unpack(pack(line.Line))});
     return tagged Invalid;
   end
```

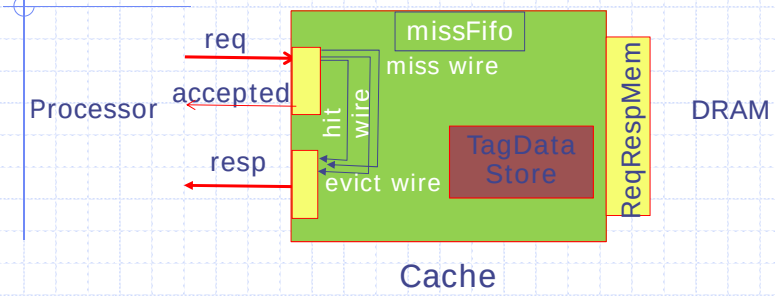
Data Cache responses (missWire)

```
else if (missWire.wget matches tagged Valid {.id, .r})
    begin
      $display("resp miss");
      missFifo.enq(tuple2(id, r));
      let req = r;
      req.op = Ld;
      mem.reqEnq(req);
      return tagged Invalid;
    end
```

Data Cache responses (memory responds)

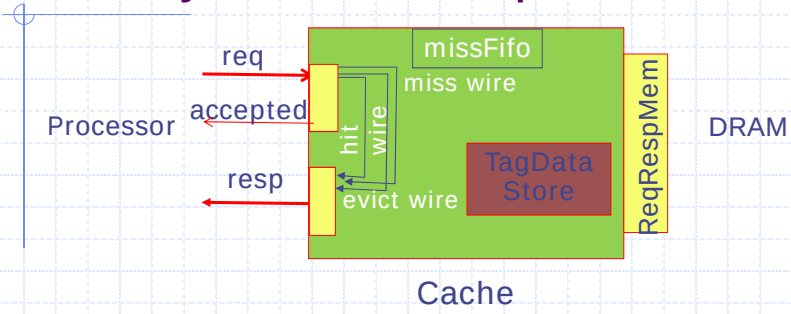
```
else if (mem.respNotEmpty && missFifo.notEmpty)
   begin
     $display("resp process");
     mem.respDeq;
     missFifo.deq;
     match {.id, .req} = missFifo.first;
     let line = req.op == Ld? unpack(pack(mem.respFirst))
                            : unpack(pack(req.data));
     let dirty = req.op == Ld? False: True;
     let l <- store.dataAccess(CacheDataReq{id: id,
      reqType: tagged FillLine (
                    tuple3(line, req.addr, dirty))});

      return tagged Valid line;
   end
```

Non-blocking to blocking: Modify which component?




```
method req:
if (missFifo.notEmpty)
   return False;
else
   // the rest of the
original code
```

Writeback to writethrough: Modify which component?



- TagDataStore's dataAccess
- DataCache's req
- DataCache's resp
- No evict wire, dirty bits

2-way set associative

Types of misses

Compulsory misses (cold start)

- Cold fact of life
- First time data is referenced
- Run billions of instructions, become insignificant

Capacity misses

- Working set is larger than cache size
- Solution: increase cache size

Conflict misses

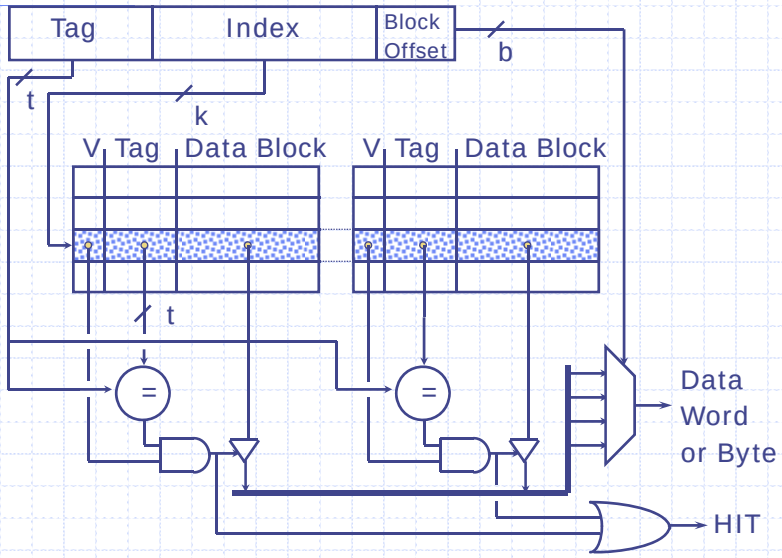
- Multiple memory locations mapped to the same location
- One set fills up, but space in other cache sets
  - Set-Associative Caches

Reduce Conflict Misses

Memory time =
Hit time + Prob(miss) * Miss penalty

- Associativity: Allow blocks to go to several sets in cache
- 2-way set associative: each block maps to either of 2 cache sets
- Fully associative: each block maps to any cache frame

2-Way Set-Associative Cache



Data Cache – code structure: Should this change from direct-mapped to n-way set associative?

```
module mkDataCache#(ReqRespMem mem)(DataCache);

  TagDataStore store <- mkTagDataStore;

  FIFOF#(Tuple2#(CacheIdentifier, MemReq)) missFifo <-
                                    mkSizedFIFOF(missFifoSize);
  RWire#(Tuple2#(CacheIdentifier, MemReq))  hitWire <- mkRWire;
  RWire#(Tuple2#(CacheIdentifier, Addr))  evictWire <- mkRWire;
  RWire#(Tuple2#(CacheIdentifier, MemReq)) missWire <- mkRWire;

  function CacheDataReq getCacheDataReqWord(CacheIdentifier id,
                           MemReq r);endfunction

  method Action#(Bool) req(MemReq r) … endmethod;
  method ActionValue#(Data) resp …  endmethod;
endmodule
```

Data Cache – Building the TagDataStore array: Should this change from a direct-mapped cache to a n-way set-associative?

```
module mkTagDataStore(TagDataStore);

  RegFile#(LineIndex, Tag)
                tagArray <- mkRegFileFull;
  Vector#(DCacheRows, Reg#(Bool))
                  valid <- replicateM(mkReg(False));
  Vector#(DCacheRows, Reg#(Bool))
                  dirty <- replicateM(mkReg(False));
  RegFile#(LineIndex, Line)
              dataArray <- mkRegFileFull;
```

n-way set-associative TagDataStore

```
typedef 2 NumWays;
typedef Bit#(TLog#(NumWays)) WaysIndex;

module mkTagDataStore(TagDataStore);
  Vector#(NumWays, RegFile#(LineIndex, Tag))
            tagArray <- replicateM(mkRegFileFull);
  Vector#(NumWays, Vector#(DCacheRows, Reg#(Bool)))
              valid <- replicateM(replicateM(mkReg(False)));
  Vector#(NumWays, Vector#(DCacheRows, Reg#(Bool)))
              dirty <- replicateM(replicateM(mkReg(False)));
  Vector#(NumWays, RegFile#(LineIndex, Line))
           dataArray <- replicateM(mkRegFileFull);
  Vector#(DCacheRows, Reg#(WaysIndex))
               lru <- replicateM(mkReg(0));
```

TagDataStore's tagLookup: Should this change from direct-mapped to n-way set associative?

```
method ActionValue#(HitOrMiss) tagLookup(Addr addr);
   let lineIndex = getLineIndex(addr);
   let tag = tagArray.sub(lineIndex);

   if(valid[lineIndex] && tag == getTag(addr))
     return HitOrMiss{hit: True, id: CacheIdentifier{
               lineIndex:  lineIndex,
               wordIndex: getWordIndex(addr)},
         addr: ?, dirty: ?};
   else
     return HitOrMiss{hit: False, id: CacheIdentifier{
               lineIndex: lineIndex,
               wordIndex: ?},
        addr: getAddr(tag, lineIndex, 0),
       dirty: valid[lineIndex] && dirty[lineIndex]};
 endmethod
```

N-way set-associative TagDataStore's tag Lookup

- Updating of LRU:

```
if(hit matches tagged Valid .i)
  lru[lineIndex] <= fromInteger(1-i);
else
  lru[lineIndex] <= 1-lru[lineIndex];
```

TagDataStore's dataAccess: Should this change from direct-mapped to n-way set associative?

```
method ActionValue#(CacheResp) dataAccess(CacheDataReq req);
    let id = req.id;
    case (req.reqType) matches
      tagged ReadWord:
        :
      tagged WriteWord .w:
        :
      tagged ReadLine:
        :
      tagged FillLine {.l, .addr, .d}:
        :
      tagged InvalidateLine:
        :
    endcase
  endmethod
```