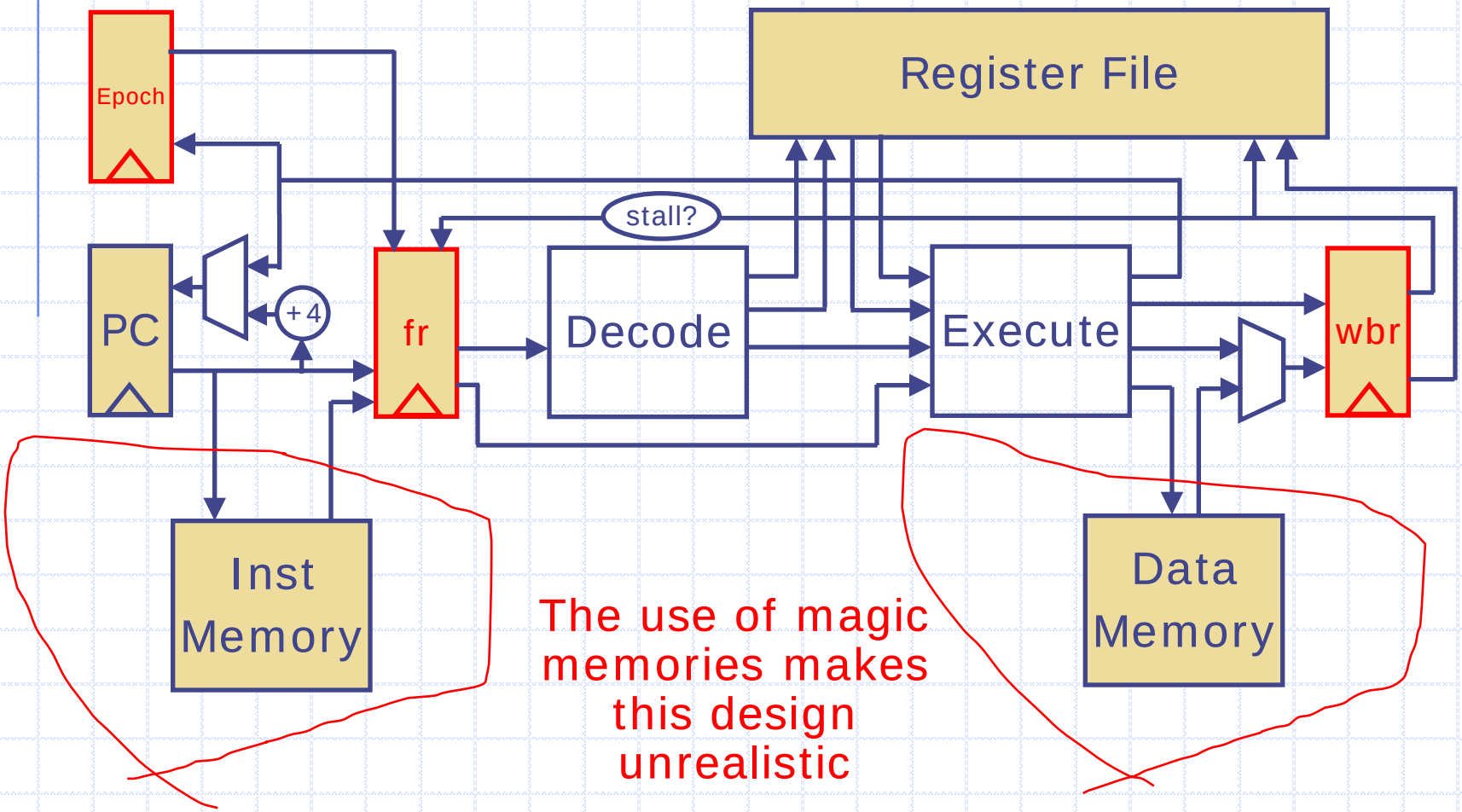
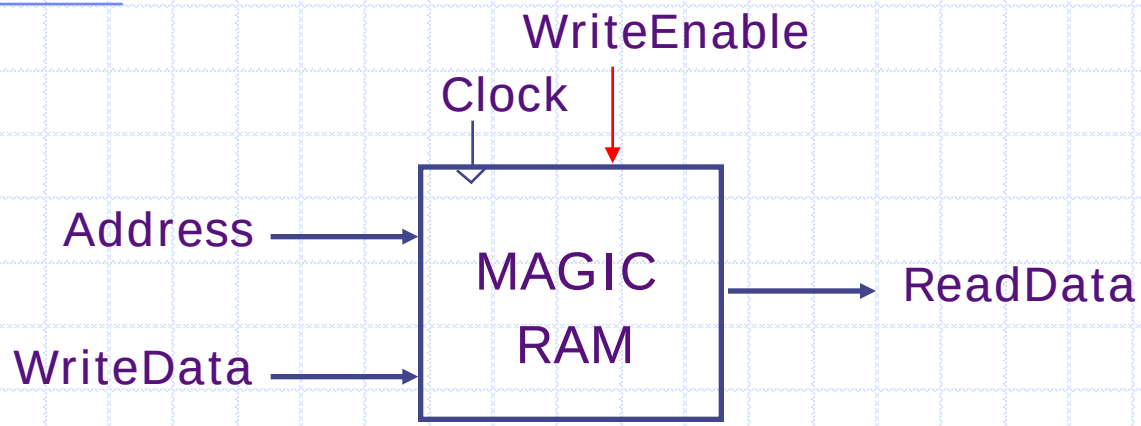
Realistic Memories and Caches

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

Three-Stage SMIPS



A Simple Memory Model

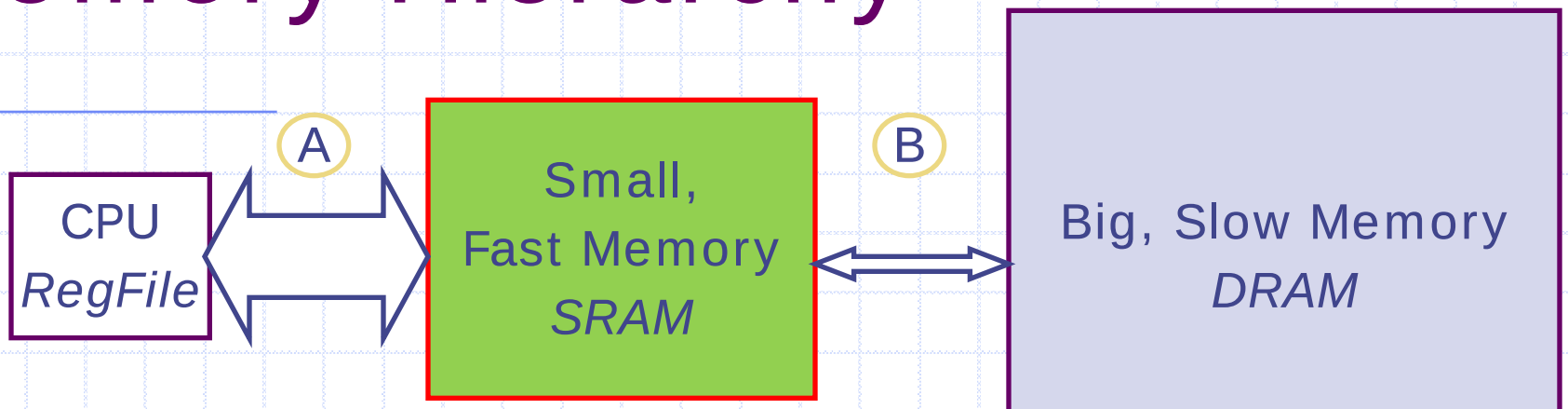


- Reads and writes are always completed in one cycle
  - a Read can be done any time (i.e. combinational)
  - If enabled, a Write is performed at the rising clock edge

    (*the write address and data must be stable at the clock edge)*

In a real DRAM the data will be available several cycles after the address is supplied

Memory Hierarchy



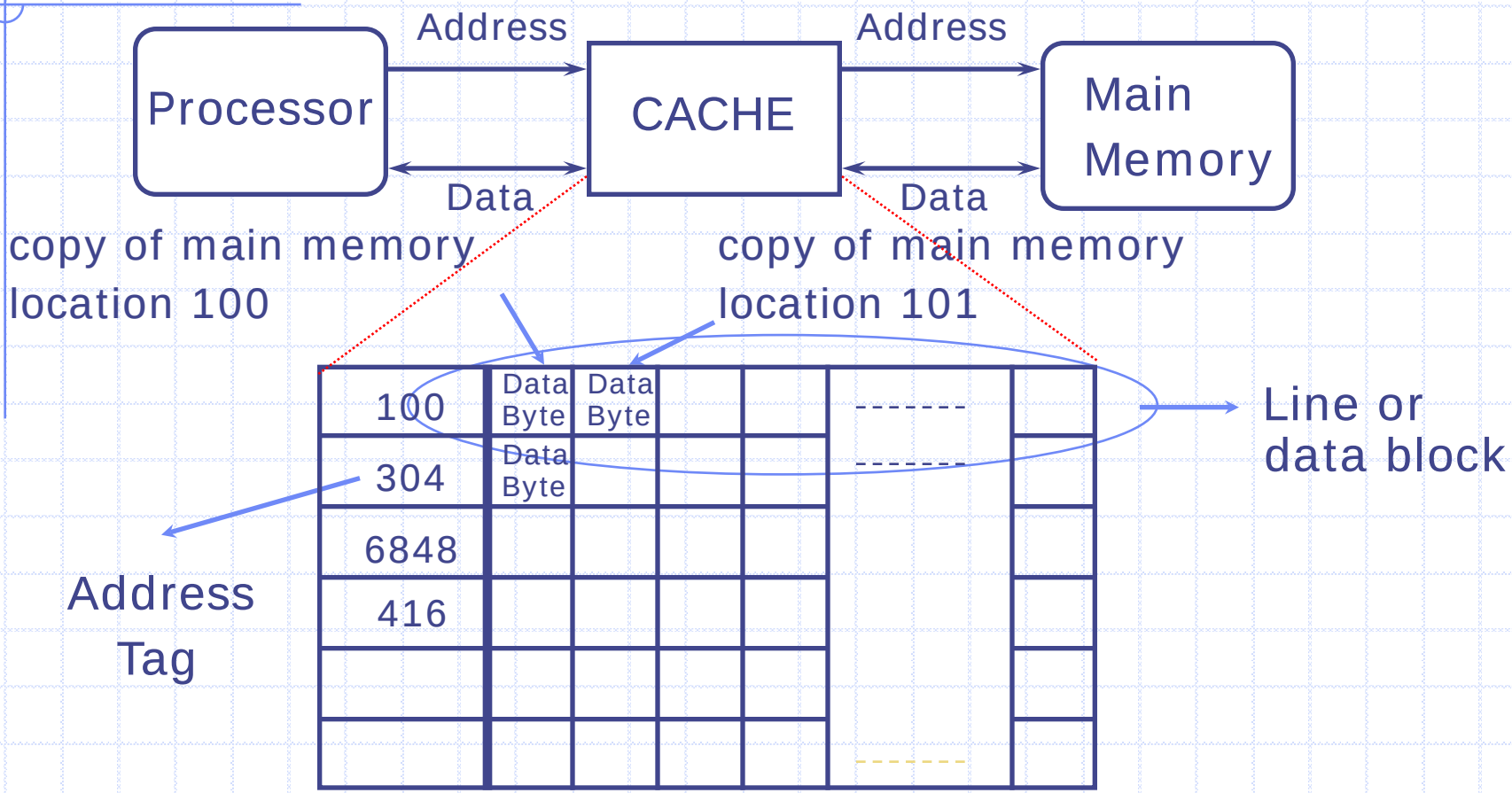
size:	RegFile	<<	SRAM	<<	DRAM *why?*
latency:	RegFile	<<	SRAM	<<	DRAM *why?*
bandwidth:	on-chip	>>	off-chip		*why?*

On a data access:

hit (data $\in$ fast memory) $\Rightarrow$ low latency access

miss (data $\notin$ fast memory) $\Rightarrow$ long latency access *(DRAM)*

Inside a Cache

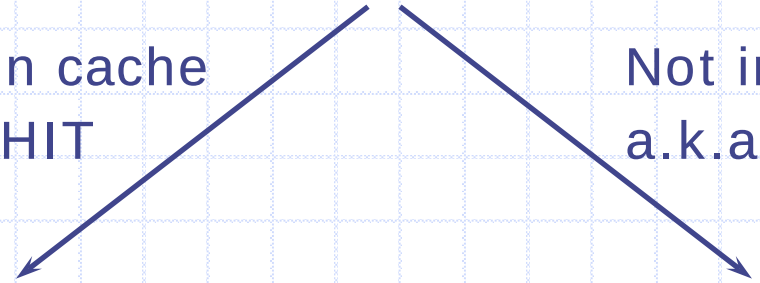


How many bits are needed in tag?

Enough to uniquely identify block

Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either



Found in cache
a.k.a. HIT

Not in cache
a.k.a. MISS

Return copy of data from cache

Read block of data from Main Memory – may require writing back a cache line

Wait …

Return data to processor and update cache

Which line do we replace?

Replacement policy

Write behavior

- On a write hit
  - Write-through: write to both cache and the next level memory
  - Writeback: write only to cache and update the next level memory when line is evacuated
- On a write miss
  - Allocate – because of multi-word lines we first fetch the line, and then update a word in it
  - Not allocate – word modified in memory

We will design a writeback, write-allocate cache

Blocking vs. Non-Blocking cache

- Blocking cache:
  - 1 outstanding miss
  - Cache must wait for memory to respond
  - Blocks in the meantime
- Non-blocking cache:
  - N outstanding misses
  - Cache can continue to process requests while waiting for memory to respond to N misses

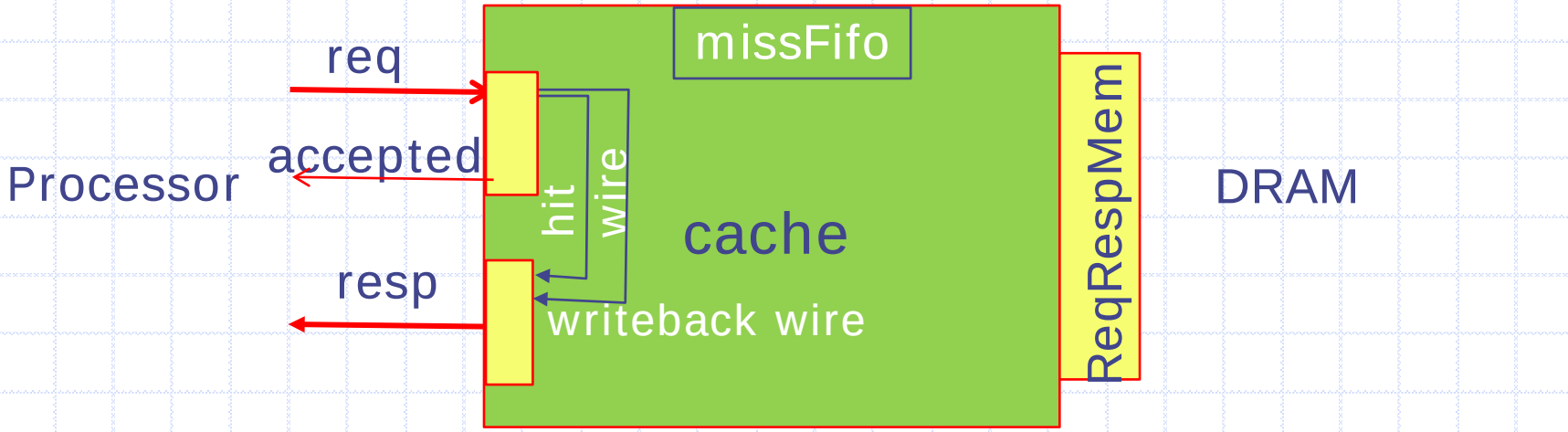
We will design a non-blocking, writeback, write-allocate cache

What do caches look like

- External interface
  - processor side
  - memory (DRAM) side
- Internal organization
  - Direct mapped
  - n-way set-associative
  - Multi-level

next lecture: Incorporating caches in processor pipelines

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

ReqRespMem is an interface passed to DataCache

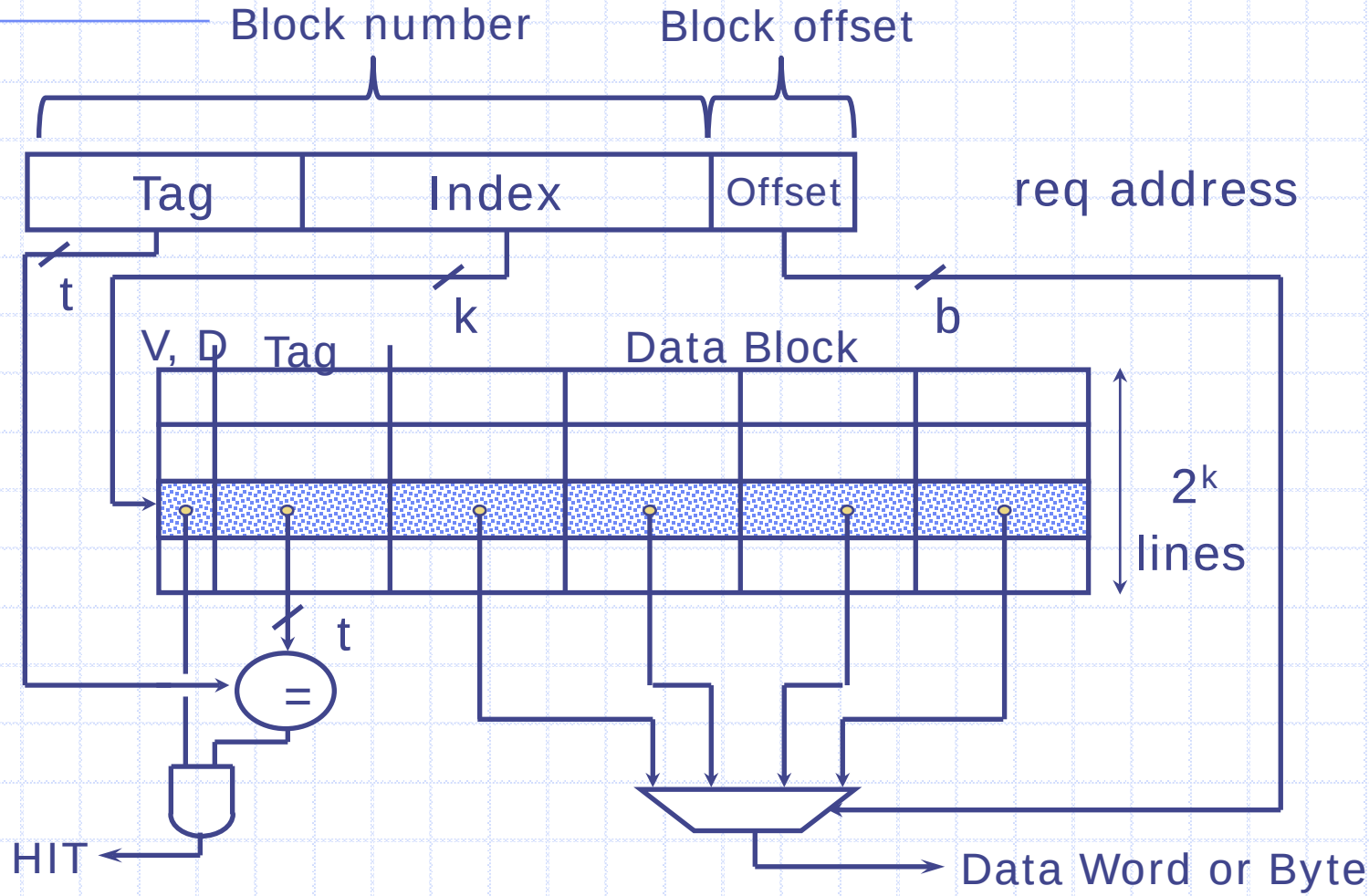
```
interface ReqRespMem;
  method Bool reqNotFull;
  method Action reqEnq(MemReq req);
  method Bool respNotEmpty;
  method Action respDeq;
  method MemResp respFirst;
endinterface
```

Interface dynamics

- Cache hits are combinational
- Cache misses are passed onto next level of memory, and can take arbitrary number of cycles to get processed
- Cache requests (misses) will not be accepted if it triggers memory requests that cannot be accepted by memory
- One request per cycle to memory

Direct mapped caches

Direct-Mapped Cache

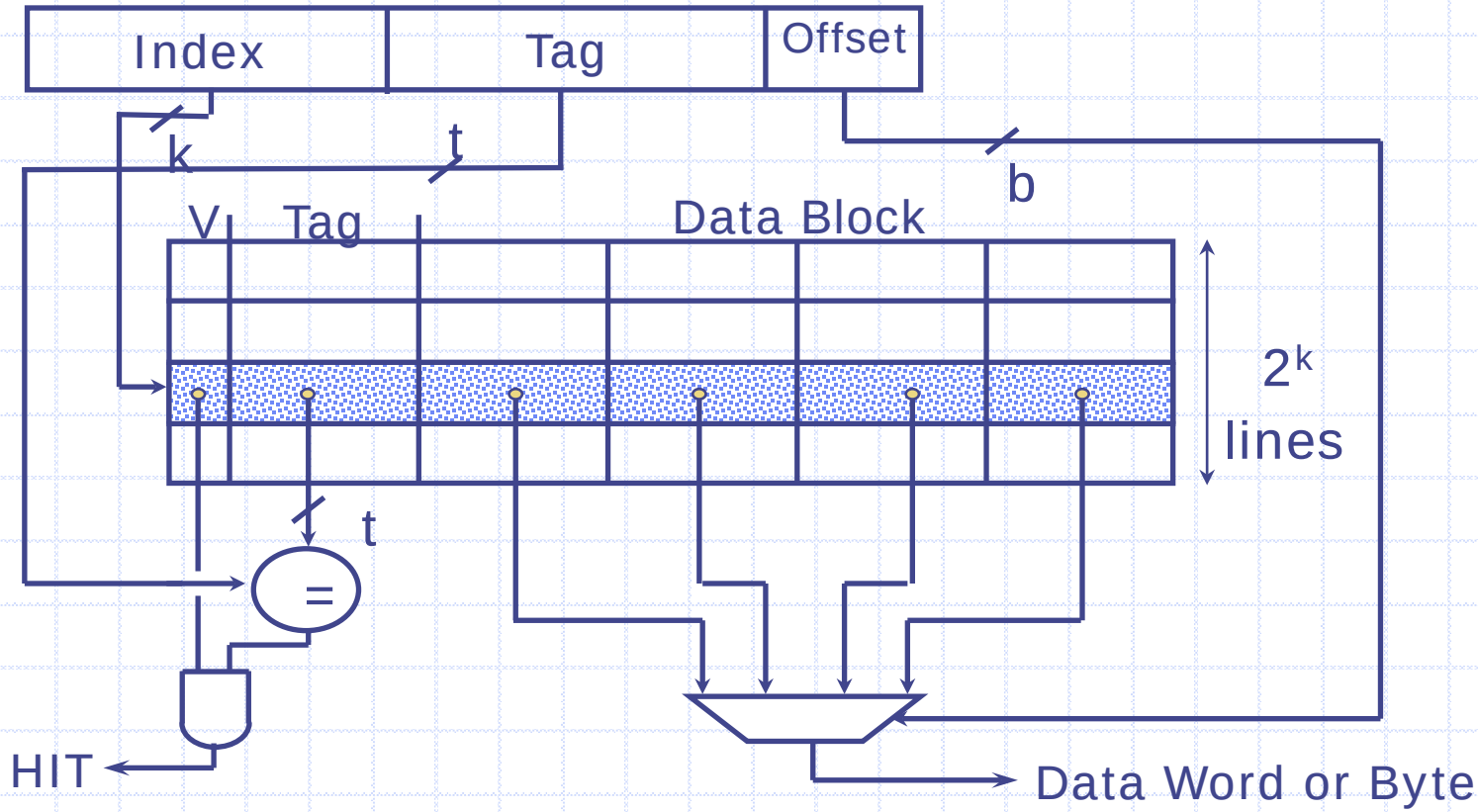


What is a bad reference pattern?

Strided = size of cache

Direct Map Address Selection

higher-order vs. lower-order address bits



Why might this be undesirable?

Spatially local blocks conflict

Data Cache – code structure

```
module mkDataCache#(ReqRespMem mem)(DataCache);
   // state declarations
   RegFile#(Index, Data) dataArray <- mkRegFileFull;
   RegFile#(Index, Tag) tagArray <- mkRegFileFull;

   Vector#(Rows, Reg#(Bool)) tagValid <-
                              replicateM(mkReg(False));
   Vector#(Rows, Reg#(Bool))  dirty <-
                              replicateM(mkReg(False));

   FIFOF#(MemReq) missFifo <- mkSizedFIFOF(reqFifoSz);
   RWire#(MemReq) hitWire <- mkUnsafeRWire;
   Rwire#(Index) replaceIndexWire <- mkUnsafeRWire;

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

Data Cache
typedefs

```
typedef 32 AddrSz;
typedef 256 Rows;
typedef Bit#(AddrSz) Addr;
typedef Bit#(TLog#(Rows)) Index;
typedef Bit#(TSub#(AddrSz, TAdd#(TLog#(Rows), 2))) Tag;

typedef 32 DataSz;
typedef Bit#(DataSz) Data;

Integer reqFifoSz = 6;

function Tag getTag(Addr addr);
function Index getIndex(Addr addr);
function Addr getAddr(Tag tag, Index idx);
```

Data Cache req

```
method ActionValue#(Bool) req(MemReq r);
  Index index = getIndex(r.addr);
  if(tagArray.sub(index) == getTag(r.addr) &&
      tagValid[index])
  begin
    hitWire.wset(r); return True;
  end
  else if(miss && evict) begin
    … return False;
  end
  else if(miss && !evict) begin
    ... return True;
  end
```

Combinational hit case

accepted

The request is

not accepted

accepted

Data Cache req – miss case

```
...
else if(tagValid[index] && dirty[index])
begin
  if(mem.reqNotFull)
  begin
    writebackWire.wset(index);
    mem.reqEnq(MemReq{op: St,
               addr:getAddr(tagArray.sub(index),index),
               data: dataArray.sub(index)});
  end
  return False;
end
…
```

Need to evict?

victim is evicted

Data Cache req

```
    else if(mem.reqNotFull && missFifo.notFull)
    begin
      missFifo.enq(r); r.op = Ld; mem.reqEnq(r);
      return True;
    end
    else
      return False;
endmethod
```

miss&!evict&canhandle

miss&!evict&!canhandle

Data Cache resp – hit case

```
method ActionValue#(Maybe#(Data)) resp;
  if(hitWire.wget matches tagged Valid .r)
  begin
    let index = getIndex(r.addr);
    if(r.op == St)
    begin
      dirty[index] <= True;
      dataArray.upd(index, r.data);
    end
    return tagged Valid (dataArray.sub(index));
  end
  …
```

Data Cache resp

```
else if(writebackWire.wget matches tagged Valid .idx)
begin
   tagValid[idx] <= False;
   dirty[idx] <= False;
   return tagged Invalid;
end
…
```

Writeback case: resp handles this to ensure that state (dirty, tagValid) is updated only in one place

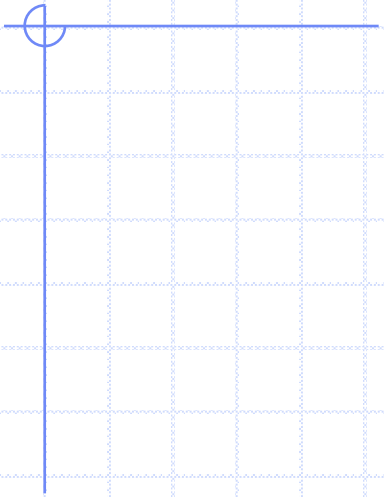
Data Cache resp – miss case

```
else if(mem.respNotEmpty)
begin
  let index = getIndex(reqFifo.first.addr);
  dataArray.upd(index, reqFifo.first.op == St?
                    reqFifo.first.data : mem.respFirst);
  if(reqFifo.first.op == St)
    dirty[index] <= True;
  tagArray.upd(index, getTag(reqFifo.first.addr));
  tagValid[index] <= True;
  mem.respDeq; missFifo.deq;
  return tagged Valid (mem.respFirst);
end
```

Data Cache resp

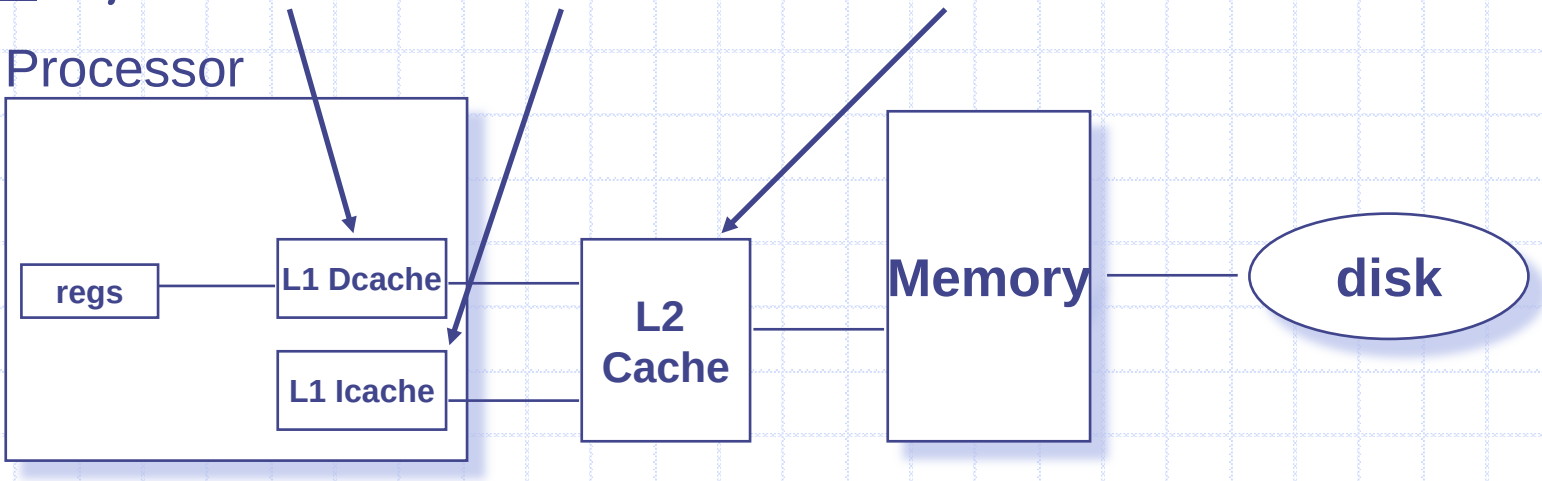
```
    else
    begin
      return tagged Invalid;
    end
endmethod
```

All other cases
(missResp not arrived,
no request given, etc)



Multi-Level Caches

Options: ***separate*** data and instruction caches, or a ***unified*** cache



Inclusive vs. Exclusive

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

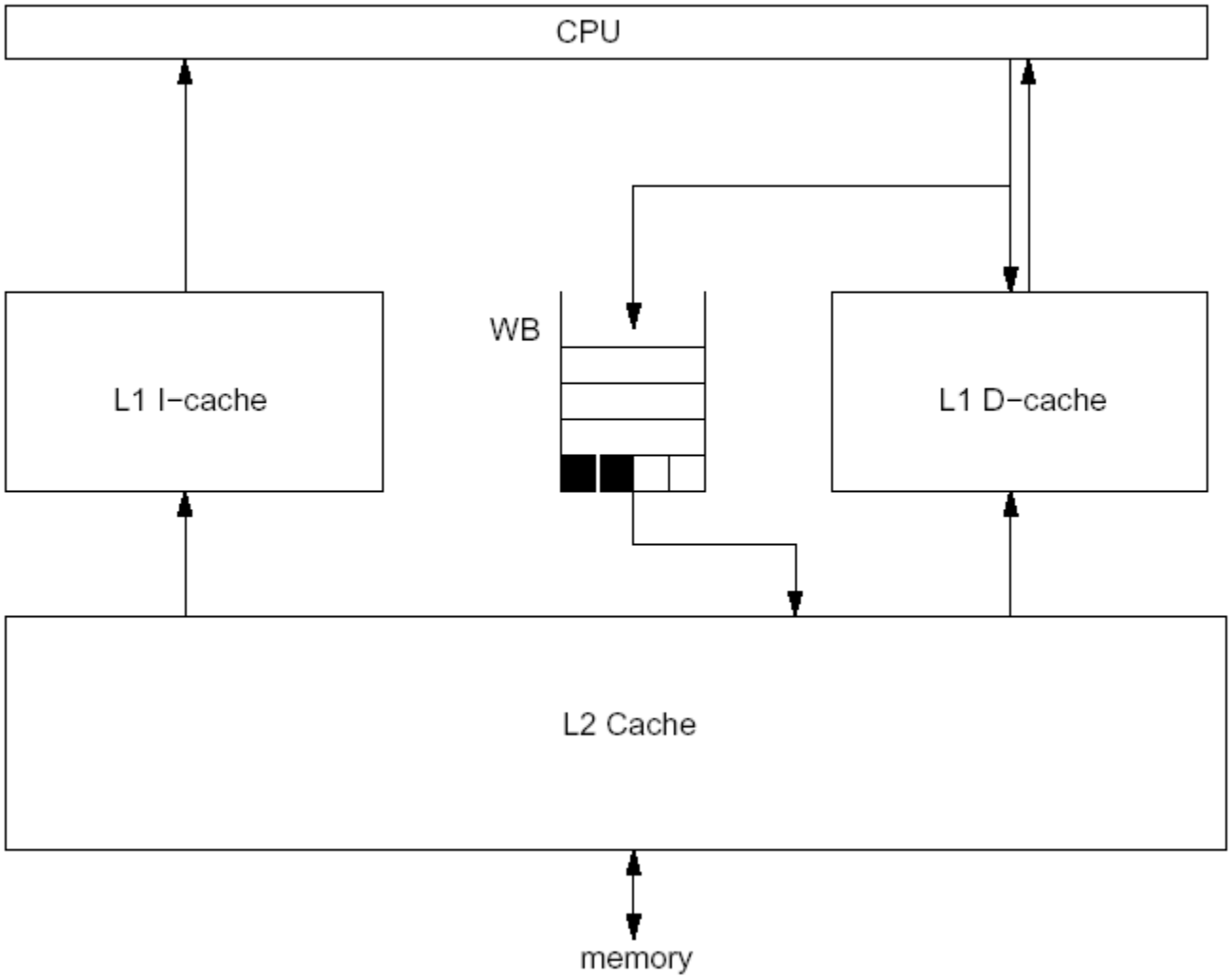
Write back

- 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

Write Buffer



Source: Skadron/Clark

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
  - ...

next integrating into the pipeline