Best Wishes from Japan
# Internal Cache Organization

## Direct-Mapped Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

req address in bytes

V  Tag  Cache Line

\[ \text{Cache size } 2^k \text{ lines} \]
Internal Cache Organization

Direct-Mapped Cache

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>
```

req address in bytes

Cache size $2^k$ lines
Internal Cache Organization

Direct-Mapped Cache

req address in bytes

Cache size $2^k$ lines
Internal Cache Organization

Direct-Mapped Cache

req address in bytes

Cache size $2^k$ lines
Internal Cache Organization

Direct-Mapped Cache

- Tag
- Index
- Offset

req address in bytes

[b-1:2]

The last two bits of b are dropped because word is 4-byte aligned

Cache size $2^k$ lines

MIT 6.004 Fall 2018

October 25, 2018
Internal Cache Organization

Direct-Mapped Cache

- Tag
- Index
- Offset

Size of data per line:
- $2^b$ bytes
- $2^{b-2}$ words

Tag

Index

Cache Line

... 

[b-1:2]

Data Word

hit

req address in bytes

Cache size $2^k$ lines

The last two bits of $b$ are dropped because word is 4-byte aligned

October 25, 2018
Internal Cache Organization

n-way Set-associative Cache

Tag   Index   Offset

V Tag  Cache Line

hit?

2^k
n-way Set-associative Cache

Typically n is 4 to 8.
High degree of associativity increases hardware complexity and access latency; assume n=2
Internal Cache Organization

n-way Set-associative Cache

Typically n is 4 to 8.
High degree of associativity increases hardware complexity and access latency; assume n=2

A cache line can be stored in either cache-array unit ⇒ fewer conflict misses
Typically $n$ is 4 to 8.
High degree of associativity increases hardware complexity and access latency; assume $n=2$

A cache line can be stored in either cache-array unit ⇒ fewer conflict misses

October 25, 2018
Typically $n$ is 4 to 8.
High degree of associativity increases hardware complexity and access latency; assume $n=2$
Typically n is 4 to 8.  
High degree of associativity increases hardware complexity and access latency; assume n=2

a cache line can be stored in either cache-array unit  
⇒ fewer conflict misses
n-way Set-associative Cache

Typically $n$ is 4 to 8. High degree of associativity increases hardware complexity and access latency; assume $n=2$

A cache line can be stored in either cache-array unit $\Rightarrow$ fewer conflict misses

October 25, 2018
Stores

- On a write hit
  - Write-back policy: write only to cache and update the next level memory when the line is evicted
  - Write-through policy: write to both the cache and the next level memory
Stores

- On a write hit
  - Write-back policy: write only to cache and update the next level memory when the line is evicted
  - Write-through policy: write to both the cache and the next level memory

For Write-back need status to be one of: Invalid, Clean, or Dirty
Stores

- On a write hit
  - **Write-back policy**: write only to cache and update the next level memory when the line is evicted
  - **Write-through policy**: write to both the cache and the next level memory

For *Write-back* need status to be one of: *Invalid*, *Clean*, or *Dirty*

- On a write miss
  - *Allocate* – because of multi-word lines we first fetch the line, and then update a word in it
  - *No allocate* – cache is not affected, the Store is forwarded to memory
Stores

- On a write hit
  - **Write-back policy**: write only to cache and update the next level memory when the line is evicted
  - **Write-through policy**: write to both the cache and the next level memory

For **Write-back** need status to be one of: **Invalid**, **Clean**, or **Dirty**

- On a write miss
  - **Allocate** – because of multi-word lines we first fetch the line, and then update a word in it
  - **No allocate** – cache is not affected, the Store is forwarded to memory

*We will use Write-back and miss-allocate.*
Processing Loads and Stores

Search the cache for the processor generated (byte) address
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache
  - a.k.a. hit
- Not in cache
  - a.k.a. miss
Processing Loads and Stores

Search the cache for the processor generated (byte) address

Found in cache a.k.a. hit

Not in cache a.k.a. miss

Load: Return a copy of the word from the cache-line
Processing Loads and Stores

Search the cache for the processor generated (byte) address

Found in cache
a.k.a. hit

Load: Return a copy of the word from the cache-line

Store: Update the relevant word in the cache-line

Not in cache
a.k.a. miss
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache a.k.a. hit
  - Load: Return a copy of the word from the cache-line
  - Store: Update the relevant word in the cache-line

- Not in cache a.k.a. miss
  - Bring the missing cache-line from Main Memory
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache a.k.a. hit
  - Load: Return a copy of the word from the cache-line
  - Store: Update the relevant word in the cache-line

- Not in cache a.k.a. miss
  - Bring the missing cache-line from Main Memory
  - May require writing back a cache-line to create space
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache
  - a.k.a. hit
  - Load: Return a copy of the word from the cache-line
  - Store: Update the relevant word in the cache-line

- Not in cache
  - a.k.a. miss
  - Bring the missing cache-line from Main Memory
  - May require writing back a cache-line to create space

Which line do we replace?
Processing Loads and Stores

Search the cache for the processor generated (byte) address

- Found in cache a.k.a. **hit**
  - Load: Return a copy of the word from the cache-line
  - Store: Update the relevant word in the cache-line

- Not in cache a.k.a. **miss**
  - Bring the missing cache-line from Main Memory
    - May require writing back a cache-line to create space

... Which line do we replace?

Update cache, and for a Load return word to processor
Replacement Policy

- Direct Mapped Cache:
  - No choice (each address maps to exactly one cache line)
Replacement Policy

- Direct Mapped Cache:
  - No choice (each address maps to exactly one cache line)

- N-way Set-associative cache:
Replacement Policy

- Direct Mapped Cache:
  - No choice (each address maps to exactly one cache line)

- N-way Set-associative cache:
  - N choices for line to replace
  - LRU: evict line from least recently used way
  - Other policies exist but LRU is most common
Replacement Policy

- **Direct Mapped Cache:**
  - No choice (each address maps to exactly one cache line)

- **N-way Set-associative cache:**
  - N choices for line to replace
  - LRU: evict line from least recently used way
  - Other policies exist but LRU is most common

For a 2-way set associative cache, can implement LRU with 1 bit of state per cache line.
Blocking vs. Non-Blocking cache

- Blocking cache

- Non-blocking cache
Blocking vs. Non-Blocking cache

- Blocking cache
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime

- Non-blocking cache
Blocking vs. Non-Blocking cache

- **Blocking cache**
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime

- **Non-blocking cache**
  - Multiple outstanding misses
  - Cache can continue to process requests while waiting for memory to respond to misses
Blocking vs. Non-Blocking cache

- Blocking cache
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept processor requests in the meantime

- Non-blocking cache
  - Multiple outstanding misses
  - Cache can continue to process requests while waiting for memory to respond to misses

We will discuss the implementation of blocking caches
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface

typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...;
Memory, SRAM and DRAM interfaces

```verilog
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface
```

Interfaces have been chosen to minimize type clutter.

```verilog
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq;
```
Memory, SRAM and DRAM interfaces

```verilog
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface
```

Fixed size memory & addresses

```
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq;
```

Interfaces have been chosen to minimize type clutter

no response for Stores
Memory, SRAM and DRAM interfaces

Interfaces have been chosen to minimize type clutter.

Fixed size memory & addresses

typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...;
Memory, SRAM and DRAM interfaces

```verilog
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface

interface DRAM;
    method Action req(LReq req);
    method ActionValue#(Line) resp;
endinterface
```

Interfaces have been chosen to minimize type clutter

```
typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...;
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...;
```

Fixed size memory & addresses

no response for Stores
Memory, SRAM and DRAM interfaces

interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface

no response for Stores

Interfaces have been chosen to minimize type clutter

Fixed size memory & addresses

interface DRAM;
    method Action req(LReq req);
    method ActionValue#(Line) resp;
endinterface

Fixed size DRAM & addresses

typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...;
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...;
Memory, SRAM and DRAM interfaces

```haskell
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface

interface DRAM;
    method Action req(LReq req);
    method ActionValue#(Line) resp;
endinterface

interface SRAM#(numeric type indexSz, type dataT);
    method Action req(Bit#(indexSz) idx, MemOP op, dataT data);
    method ActionValue#(dataT) resp;
endinterface

typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq...
typedef struct {MemOp op; LAddr laddr; Line line;} LReq...
```

Fixed size memory & addresses

Fixed size DRAM & addresses

Interfaces have been chosen to minimize type clutter

no response for Stores
Memory, SRAM and DRAM interfaces

```vhdl
interface Memory;
    method Action req(MemReq req);
    method ActionValue#(Word) resp;
endinterface

interface SRAM#(numeric type indexSz, type dataT);
    method Action req(Bit#(indexSz) idx, MemOP op, dataT data);
    method ActionValue#(dataT) resp;
endinterface

no response for Stores

Interfaces have been chosen to minimize type clutter

Fixed size memory & addresses

Fixed size DRAM & addresses

Size of SRAM = 2^{indexSz} data elements

typedef enum {Ld, St} MemOp deriving(Bits, Eq);
typedef struct {MemOp op; Word addr; Word data;} MemReq;
typedef struct {MemOp op; LAddr laddr; Line line;} LReq;
```
Cached Memory Systems

The memory has a small SRAM cache which is backed by much bigger DRAM memory.

Processor accesses are for words while DRAM accesses are for lines.
Cached Memory Systems

The memory has a small SRAM cache which is backed by much bigger DRAM memory.

Processor accesses are for words while DRAM accesses are for lines.

mkDRAM and mkSRAM primitives are given:

```plaintext
DRAM dram <- mkDRAM;
SRAM#(LogNumEntities, dataT) sram <- mkSRAM;
```
Cached Memory Systems

The memory has a small SRAM cache which is backed by much bigger DRAM memory.

Processor accesses are for words while DRAM accesses are for lines.

mkDRAM and mkSRAM primitives are given:

```plaintext
DRAM dram <- mkDRAM;
SRAM#(LogNumEntities, dataT) sram <- mkSRAM;
```

To avoid type clutter we assume that DRAM has 64Byte (16 word) lines and uses line addresses.
Now we will implement a cache

- One-way, Direct-mapped
- Write-back
- Write-miss allocate
- blocking cache
Now we will implement a cache

- One-way, Direct-mapped
- Write-back
- Write-miss allocate
- blocking cache

Back-end memory (DRAM) is updated only when a line is evicted from the cache

Cache is updated on Store miss

Cache processes one request at a time
Cache Interface

SRAM based cache

- req
- resp

DRAM based memory

- lineReq
- lineResp
Cache Interface

```vhdl
interface Cache#: (numeric type logNLines);
   method Action req(MemReq req);
   method ActionValue#(Word) resp();
   method ActionValue#(LReq) lineReq;
   method Action lineResp(Line r);
endinterface
```
Cache Interface

interface Cache#(numeric type logNLines);
    method Action req(MemReq req);
    method ActionValue#(Word) resp();
    method ActionValue#(LReq) lineReq;
    method Action lineResp(Line r);
endinterface

processor-side methods: fixed size words and (byte) addresses
Cache Interface

```
interface Cache#(numeric type logNLines);
    method Action req(MemReq req);
    method ActionValue#(Word) resp();
    method ActionValue#(LReq) lineReq;
    method Action lineResp(Line r);
endinterface
```

Processor-side methods: fixed size words and (byte) addresses

Back-side methods: fixed size cache lines and line-addresses
interface Cache#(numeric type logNLines);
    method Action req(MemReq req);
    method ActionValue#(Word) resp();
    method ActionValue#(LReq) lineReq;
    method Action lineResp(Line r);
endinterface

Notice, the cache size does not appear in any of its interface methods, i.e., users do not have to know the cache size
Cache Interface

interface Cache#(numeric type logNLines);
  method Action req(MemReq req);
  method ActionValue#(Word) resp();
  method ActionValue#(LReq) lineReq;
  method Action lineResp(Line r);
endinterface

Notice, the cache size does not appear in any of its interface methods, i.e., users do not have to know the cache size.

module mkMemory(Memory);
  DRAM dram <- mkDRAM;
  Cache#(LogNLines) cache <- mkBlockingCache;
  ...

Cache Organization

- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
Cache Organization

- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
- Need queues to communicate with the back-end memory
Cache Organization

- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
- Need queues to communicate with the back-end memory
- hitQ holds the responses until the processor picks them up
Cache Organization

- cache-array unit encapsulates data, tag and status arrays, which are all made from SRAM
- Need queues to communicate with the back-end memory
- hitQ holds the responses until the processor picks them up
- state and current req registers hold the request and its status while it is being processed
Cache-Array Unit Functionality

req → cache-array unit
resp ←
update →

October 25, 2018
Cache-Array Unit Functionality

- Suppose a request gets a hit in the cache-array unit
  - Load hit returns a word
  - Store hits returns nothing (void)
Suppose a request gets a hit in the cache-array unit
  - Load hit returns a word
  - Store hits returns nothing (void)
In case of a miss, the line slot must have been occupied; all the data in the missed slot is returned so that it can be written to the back-end memory if necessary
Suppose a request gets a hit in the cache-array unit
  - Load hit returns a word
  - Store hits returns nothing (void)
- In case of a miss, the line slot must have been occupied; all the data in the missed slot is returned so that it can be written to the back-end memory if necessary
- When the correct data becomes available from the back end memory, the cache-array unit needs to be updated
interface CAU#(numeric type logNLines);
    method Action req(MemReq r);
    method ActionValue#(CAUResp) resp();
    method Action update(CacheIndex index, TaggedLine newline);
endinterface
interface CAU#(numeric type logNLines);
    method Action req(MemReq r);
    method ActionValue#(CAUResp) resp();
    method Action update(CacheIndex index, TaggedLine newline);
endinterface

typedef struct{HitMissType hitMiss; Word ldValue;
            TaggedLine taggedLine;} CAUResp;
typedef enum{LdHit, StHit, Miss} hitMiss;
typedef struct{Line line; CacheStatus status; CacheTag tag;
} TaggedLine;
module mkCAU(CAU#(LogNLines));

  method Action req(MemReq r);

  method ActionValue#(CAUResp) resp;

  method Action update(CacheIndex index, TaggedLine newline);

endmethod
endmodule
module mkCAU(CAU#(LogNLines));
    Instantiate SRAMs for dataArray, tagArray, statusArray;

    method Action req(MemReq r);

    method ActionValue#(CAUResp) resp;

    method Action update(CacheIndex index, TaggedLine newline);
endmethod
endmodule
module mkCAU(CAU#(LogNLines));
    Instantiate SRAMs for dataArray, tagArray, statusArray;
    Reg#(CAUStatus) status <- mkReg(Ready);

    method Action req(MemReq r);

    method ActionValue#(CAUResp) resp;

    method Action update(CacheIndex index, TaggedLine newline);

endmethod
endmodule
module mkCAU(CAU#(LogNLines));
    Instantiate SRAMs for dataArray, tagArray, statusArray;
    Reg#(CAUStatus) status <- mkReg(Ready);
    Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
method Action req(MemReq r);

method ActionValue#(CAUResp) resp;

method Action update(CacheIndex index, TaggedLine newline);
endmethod
endmodule
module mkCAU(CAU#(LogNLines));
  Instantiate SRAMs for dataArray, tagArray, statusArray;
  Reg#(CAUStatus) status <- mkReg(Ready);
  Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
  method Action req(MemReq r);
    initiate reads to tagArray, dataArray, and statusArray;
    store request r in currReq
  endmethod
  method ActionValue#(CAUResp) resp;
  method Action update(CacheIndex index, TaggedLine newline);
endmethod
endmodule
module mkCAU(CAU#(LogNLines));

Instantiate SRAMs for dataArray, tagArray, statusArray;
Reg#(CAUStatus) status <- mkReg(Ready);
Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
method Action req(MemReq r);
    initiate reads to tagArray, dataArray, and statusArray;
    store request r in currReq
endmethod

method ActionValue#(CAUResp) resp;
    Wait for responses for earlier requests
    Get currTag, idx, wOffset from currReq.addr and do tag match
    In case of a Ld hit, return the word; St hit, update the word;
    In case of a miss, return the data, tag and status;
endmethod

method Action update(CacheIndex index, TaggedLine newline);

endmethod
endmodule
module mkCAU(CAU#(LogNLines));
  Instantiate SRAMs for dataArray, tagArray, statusArray;
  Reg#(CAUStatus) status <- mkReg(Ready);
  Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
  method Action req(MemReq r);
    initiate reads to tagArray, dataArray, and statusArray;
    store request r in currReq
  endmethod
  method ActionValue#(CAUResp) resp;
    Wait for responses for earlier requests
    Get currTag, idx, wOffset from currReq.addr and do tag match
    In case of a Ld hit, return the word; St hit, update the word;
    In case of a miss, return the data, tag and status;
  endmethod
  method Action update(CacheIndex index, TaggedLine newline);
    update the SRAM arrays at index
  endmethod
endmodule
module mkCAU(CAU#(LogNLines));
    Instantiate SRAMs for dataArray, tagArray, statusArray;
    Reg#(CAUStatus) status <- mkReg(Ready);
    Reg#(MemReq) currReq <- mkRegU; //shadow of outer currReq
method Action req(MemReq r);
    initiate reads to tagArray, dataArray, and statusArray;
    store request r in currReq
endmethod
method ActionValue#(CAUResp) resp;
    Wait for responses for earlier requests
    Get currTag, idx, wOffset from currReq.addr and do tag match
    In case of a Ld hit, return the word; St hit, update the word;
    In case of a miss, return the data, tag and status;
endmethod
method Action update(CacheIndex index, TaggedLine newline);
    update the SRAM arrays at index
endmethod
endmodule
module mkBlockingCache(Cache#(LogNLines));

method Action req(MemReq req) ...  
method ActionValue#(Word) resp() ...  
method ActionValue#(LReq) lineReq ...  
method Action lineResp(Line r) ...

endmodule
module mkBlockingCache(Cache#(LogNLines));
    CAU#(LogNLines) cau <- mkCAU();

method Action req(MemReq req) ...
method ActionValue#(Word) resp() ...
method ActionValue#(LReq) lineReq ...
method Action lineResp(Line r) ...
endmodule
module mkBlockingCache(Cache#(LogNLines));
  CAU#(LogNLines) cau <- mkCAU();

  FIFO#(Word)       hitQ <- mkFIFO;
  Reg#(MemReq)      currReq <- mkRegU;
  Reg#(ReqStatus)   state <- mkReg(Ready);
  FIFO#(LReq)       lineReqQ <- mkFIFO;
  FIFO#(Line)       lineRespQ <- mkFIFO;

method Action  req(MemReq req) ...
method ActionValue#(Word) resp() ...
method ActionValue#(LReq) lineReq ...
method Action  lineResp(Line r) ...

endmodule
Blocking cache

module mkBlockingCache(Cache#(LogNLines));
  CAU#(LogNLines) cau <- mkCAU();

  FIFO#(Word)    hitQ <- mkFIFO;
  Reg#(MemReq)   currReq <- mkRegU;
  Reg#(ReqStatus) state <- mkReg(Ready);
  FIFO#(LReq)    lineReqQ <- mkFIFO;
  FIFO#(Line)    lineRespQ <- mkFIFO;

method Action req(MemReq req) ...
method ActionValue#(Word) resp() ...
method ActionValue#(LReq) lineReq ...
method Action lineResp(Line r) ...
endmodule

Rules to process a request including cache misses
Ready -> WaitCAUResp ->
  (if miss SendReq -> WaitDramResp) -> Ready

State Elements
Blocking cache methods

```
method Action req(MemReq r) if (state == Ready);
    currReq <= r; cau.req(r);
    state <= WaitCAUResp
endmethod
```
Blocking cache methods

```plaintext
method Action req(MemReq r) if (state == Ready);
    currReq <= r; cau.req(r);
    state <= WaitCAUResp
endmethod

method ActionValue#(Word) resp;
    hitQ.deq(); return hitQ.first;
endmethod
```
Blocking cache methods

method Action req(MemReq r) if (state == Ready);
    currReq <= r; cau.req(r);
    state <= WaitCAUResp
endmethod

method ActionValue #(Word) resp;
    hitQ.deq(); return hitQ.first;
endmethod

method ActionValue #(LReq) lineReq();
    lineReqQ.deq(); return lineReqQ.first();
endmethod
Blocking cache methods

method Action req(MemReq r) if (state == Ready);
    currReq <= r; cau.req(r);
    state <= WaitCAUResp
endmethod

method ActionValue #(Word) resp;
    hitQ.deq(); return hitQ.first;
endmethod

method ActionValue #(LReq) lineReq();
    lineReqQ.deq(); return lineReqQ.first();
endmethod

method Action lineResp (Line r);
    lineRespQ.enq(r);
endmethod
Blocking cache rules

rule waitCAUResponse;

let x <- cau.resp;

case (x.hitMiss)

LdHit:

StHit:

endcase

endrule
Blocking cache rules

rule waitCAUResponse;

let x <- cau.resp;

case (x.hitMiss)
    LdHit : begin Word v = x.ldValue;
        hitQ.enq(v); state <= Ready; end
    StHit :
endcase
endrule
Blocking cache rules

**rule waitCAUResponse**

```plaintext
rule waitCAUResponse;
  let x <= cau.resp;
  case (x.hitMiss)
    LdHit : begin Word v = x.ldValue;
      hitQ.enq(v); state <= Ready; end
    StHit : state <= Ready;
endcase
endrule
```
Blocking cache rules

rule waitCAUResponse;

let x <- cau.resp;

case (x.hitMiss)
    LdHit: begin Word v = x.ldValue;
        hitQ.enq(v); state <= Ready; end
    StHit: state <= Ready;
    Miss: begin let oldTaggedLine = x.taggedLine;
        extract cstatus, evictLaddr, line from oldTaggedLine
        if (cstatus == Dirty) begin // writeback required
        end
    endcase
endcase
endrule
Blocking cache rules

rule waitCAUResponse;
let x <- cau.resp;
case (x.hitMiss)
  LdHit : begin Word v = x.ldValue;
          hitQ.enq(v); state <= Ready; end
  StHit : state <= Ready;
  Miss : begin let oldTaggedLine = x.taggedLine;
          extract cstatus, evictLaddr, line from oldTaggedLine
          if (cstatus == Dirty) begin // writeback required
              lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
              state<= SendReq;
          end
  endcase
endrule
Blocking cache rules

rule waitCAUResponse;

let x <- cau.resp;

case (x.hitMiss)
    LdHit : begin Word v = x.ldValue;
        hitQ.enq(v); state <= Ready; end
    StHit : state <= Ready;
    Miss : begin let oldTaggedLine = x.taggedLine;
        extract cstatus, evictLaddr, line from oldTaggedLine
        if (cstatus == Dirty) begin // writeback required
            lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
            state<= SendReq;
        end else begin // no writeback required
    endcase
endrule
Blocking cache rules

rule waitCAUResponse

let x <- cau.resp;

case (x.hitMiss)
    LdHit : begin
        Word v = x.ldValue;
        hitQ.enq(v); state <= Ready; end
    StHit : state <= Ready;
    Miss  : begin
        let oldTaggedLine = x.taggedLine;
        extract cstatus, evictLaddr, line from oldTaggedLine
        if (cstatus == Dirty) begin // writeback required
            lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
            state<= SendReq;
        end else begin // no writeback required
            extract newLaddr from currReq
        end
endcase
endrule
Blocking cache rules

**rule waitCAUResponse**

```
rule waitCAUResponse;
    let x <- cau.resp;
    case (x.hitMiss)
        LdHit : begin Word v = x.ldValue;
                hitQ.enq(v); state <= Ready; end
        StHit : state <= Ready;
        Miss  : begin let oldTaggedLine = x.taggedLine;
                 extract cstatus, evictLaddr, line from oldTaggedLine
                 if (cstatus == Dirty) begin // writeback required
                     lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
                     state<= SendReq;
                 end else begin // no writeback required
                     extract newLaddr from currReq
                     lineReqQ.enq(LReq{op:Ld, laddr:newLaddr, line: ?});
                     state <= WaitDramResp;
                 end
    endcase
endrule
```

Word v = x.ldValue; hitQ.enq(v); state <= Ready; end
StHit : state <= Ready;
Miss  : begin let oldTaggedLine = x.taggedLine;
        extract cstatus, evictLaddr, line from oldTaggedLine
        if (cstatus == Dirty) begin // writeback required
            lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
            state<= SendReq;
        end else begin // no writeback required
            extract newLaddr from currReq
            lineReqQ.enq(LReq{op:Ld, laddr:newLaddr, line: ?});
            state <= WaitDramResp;
        end
```
Blocking cache rules

rule waitCAUResp

rule waitCAUResp;
let x ← cau.resp;
case (x.hitMiss)
  LdHit : begin
    Word v = x.ldValue;
    hitQ.enq(v); state <= Ready; end
  StHit : state <= Ready;
  Miss : begin
    let oldTaggedLine = x.taggedLine;
    extract cstatus, evictLaddr, line from oldTaggedLine
    if (cstatus == Dirty) begin // writeback required
      lineReqQ.enq(LReq{op:St, laddr:evictLaddr, line:line});
      state <= SendReq;
    end else begin // no writeback required
      extract newLaddr from currReq
      lineReqQ.enq(LReq{op:Ld, laddr:newLaddr, line: ?});
      state <= WaitDramResp;
    end
  endcase
endrule

begin
  Word v = x.ldValue;
  hitQ.enq(v); state <= Ready; end
  state <= Ready;
rule waitDramResponse

rule waitDramResponse(state == WaitDramResp);
let line = lineRespQ.first(); lineRespQ.deq();

endrule
Blocking cache rules

rule waitDramResponse

rule waitDramResponse(state == WaitDramResp);
let line = lineRespQ.first(); lineRespQ.deq();
get idx, tag, wOffset from currReq.addr;
if (currReq.op == Ld) begin
    hitQ.enq(line[wOffset]);
    cau.update(idx,
        TaggedLine {line: line, status: Clean, tag: tag});
endrule
Blocking cache rules

rule waitDramResponse

rule waitDramResponse(state == WaitDramResp);
  let line = lineRespQ.first(); lineRespQ.deq();
  get idx, tag, wOffset from currReq.addr;
  if (currReq.op == Ld) begin
    hitQ.enq(line[wOffset]);
    cau.update(idx,
      TaggedLine {line: line, status: Clean, tag: tag});
  end else begin // St
    line[wOffset] = currReq.data;
    cau.update(idx,
      TaggedLine {line: line, status: Dirty, tag: tag});
  end
endrule
Blocking cache rules

rule waitDramResponse

rule waitDramResponse(state == WaitDramResp);
  let line = lineRespQ.first(); lineRespQ.deq();
  get idx, tag, wOffset from currReq.addr;
  if (currReq.op == Ld) begin
    hitQ.enq(line[wOffset]);
    cau.update(idx,
               TaggedLine {line: line, status: Clean, tag: tag});
  end else begin // St
    line[wOffset] = currReq.data;
    cau.update(idx,
               TaggedLine {line: line, status: Dirty, tag: tag});
  end
  state <= Ready;
endrule
Hit and miss performance

- Hit
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design
Hit and miss performance

- **Hit**
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design

- **Miss**
  - No evacuation: DRAM load latency + 2 X SRAM latency
Hit and miss performance

- **Hit**
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design

- **Miss**
  - No evacuation: DRAM load latency + 2 X SRAM latency
  - Evacuation: DRAM store latency + DRAM load latency + 2 X SRAM latency
Hit and miss performance

- **Hit**
  - Directly related to the latency of L1
  - 1-cycle latency with appropriate hitQ design

- **Miss**
  - No evacuation: DRAM load latency + 2 × SRAM latency
  - Evacuation: DRAM store latency + DRAM load latency + 2 × SRAM latency

Adding a few extra cycles in the miss case does not have a big impact on performance
Lab6: 2-way Set-Associative Cache

- Need to instantiate cache-array unit twice
- Tag matching has to be done for each cache-array unit
- We can get a hit in at most one cache
- Miss detection requires checking both cache-array units; a victim for throwing out has to be selected
- Once the victim has been selected Cache-miss processing is the same
- cache-array unit should be used as is
Take Home problem

- Describe a situation where 1-Way Direct Map cache would get a miss but a 2-Way cache would not.