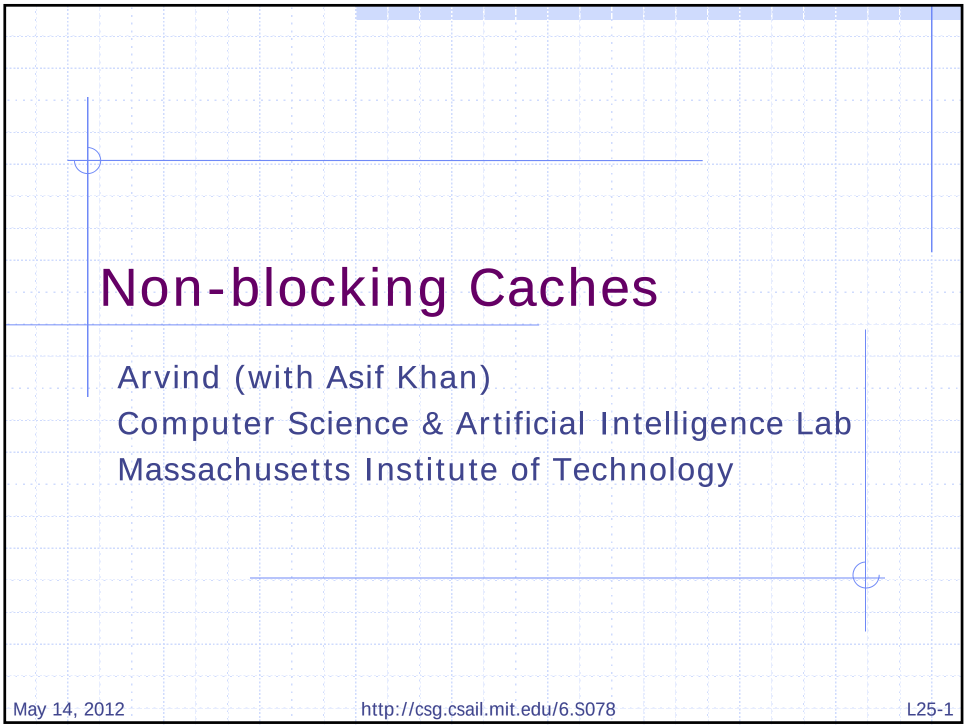
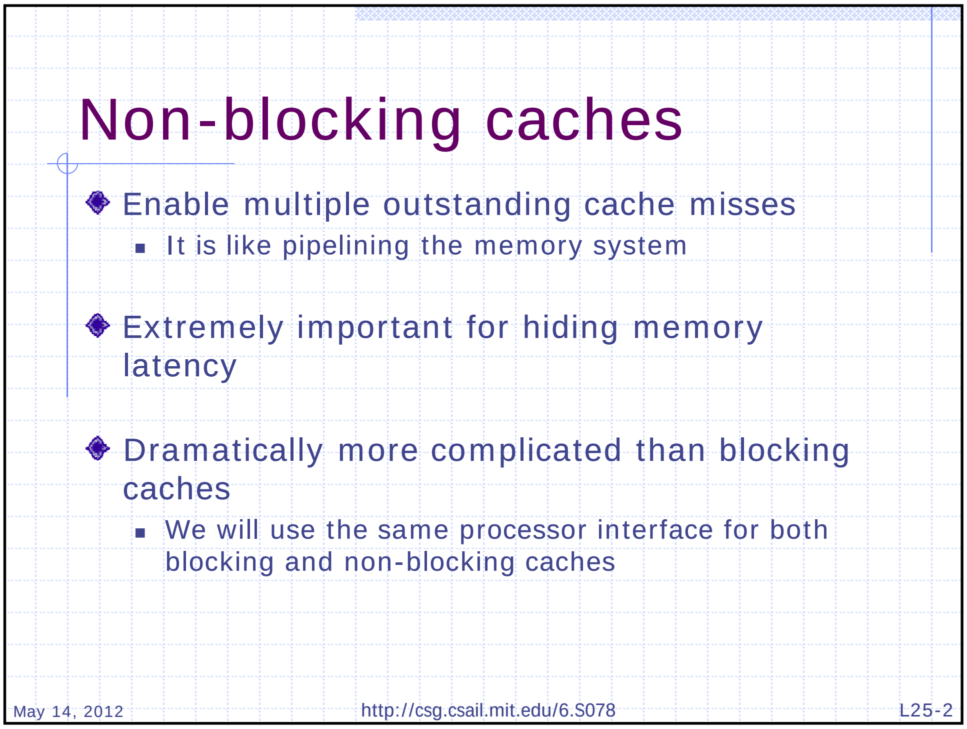


# Non-blocking Caches

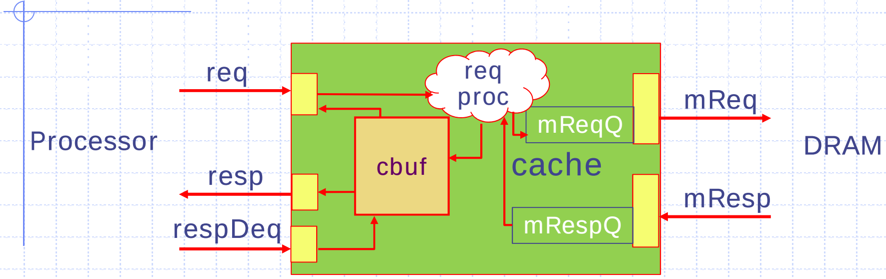
Arvind (with Asif Khan)
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology



## Non-blocking caches

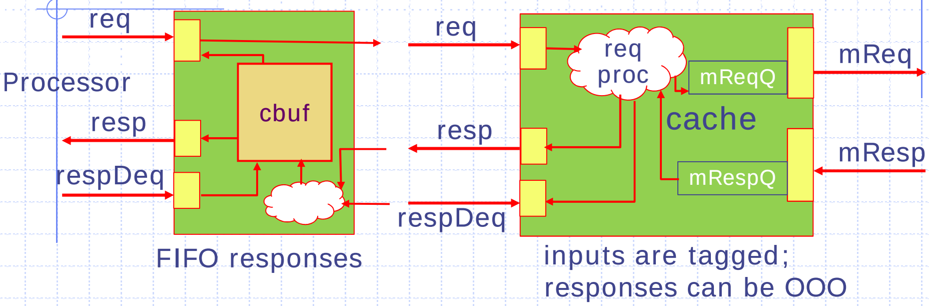
- Enable multiple outstanding cache misses
  - It is like pipelining the memory system
- Extremely important for hiding memory latency
- Dramatically more complicated than blocking caches
  - We will use the same processor interface for both blocking and non-blocking caches

# Non-blocking cache

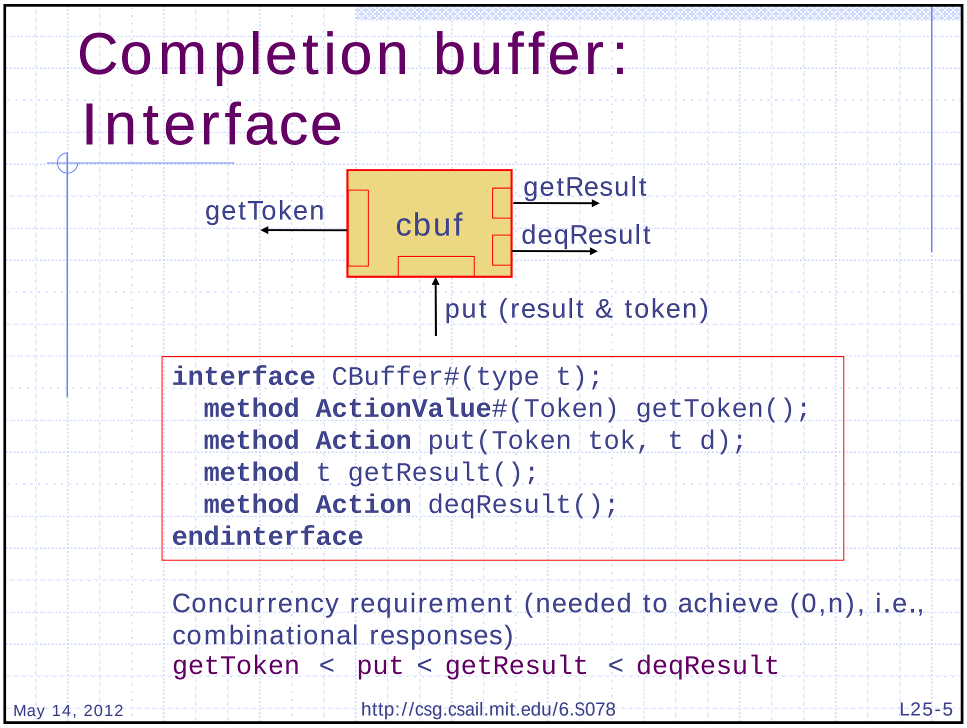


- Completion buffer controls the entries of requests and ensures that departures take place in order even if loads complete out-of-order

# Non-blocking caches



- Split the non-blocking cache in two parts
  - The front end attaches tags to requests and ensures FIFO delivery of responses to the processor
  - The backend is a non-locking cache which can return responses out-of-order
  - One may merge the front end with the processor and directly expose the backend interface



# Completion buffer: Interface

getToken

cbuf

getResult

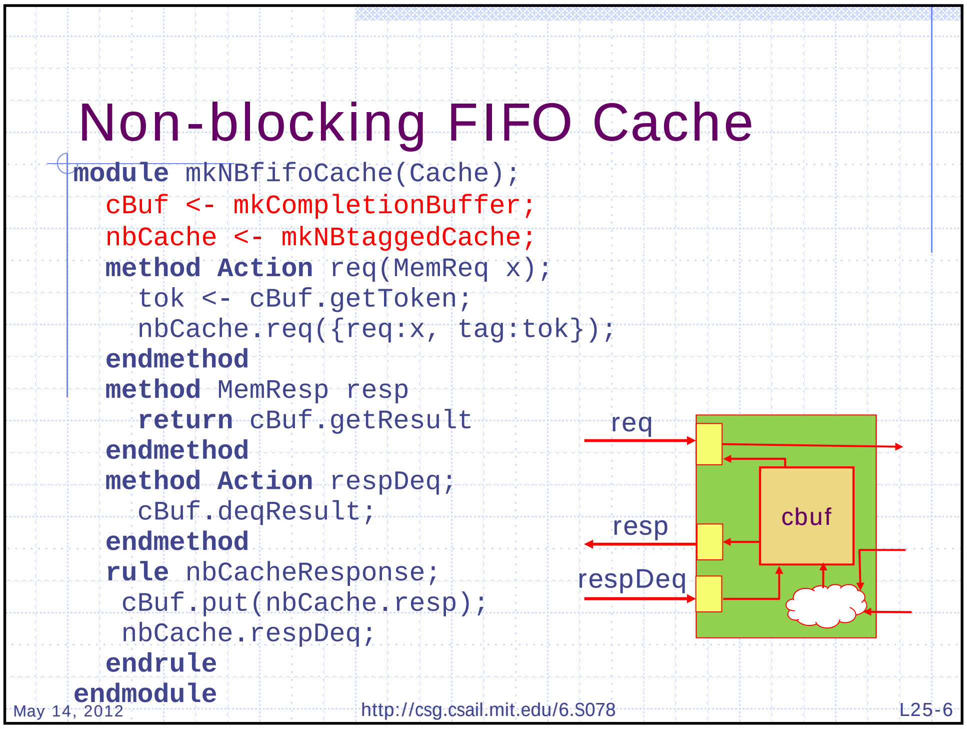
deqResult

put (result & token)

```
interface CBuffer#(type t);
  method ActionValue#(Token) getToken();
  method Action put(Token tok, t d);
  method t getResult();
  method Action deqResult();
endinterface
```

Concurrency requirement (needed to achieve (0,n), i.e., combinational responses)

getToken < put < getResult < deqResult



# Non-blocking FIFO Cache

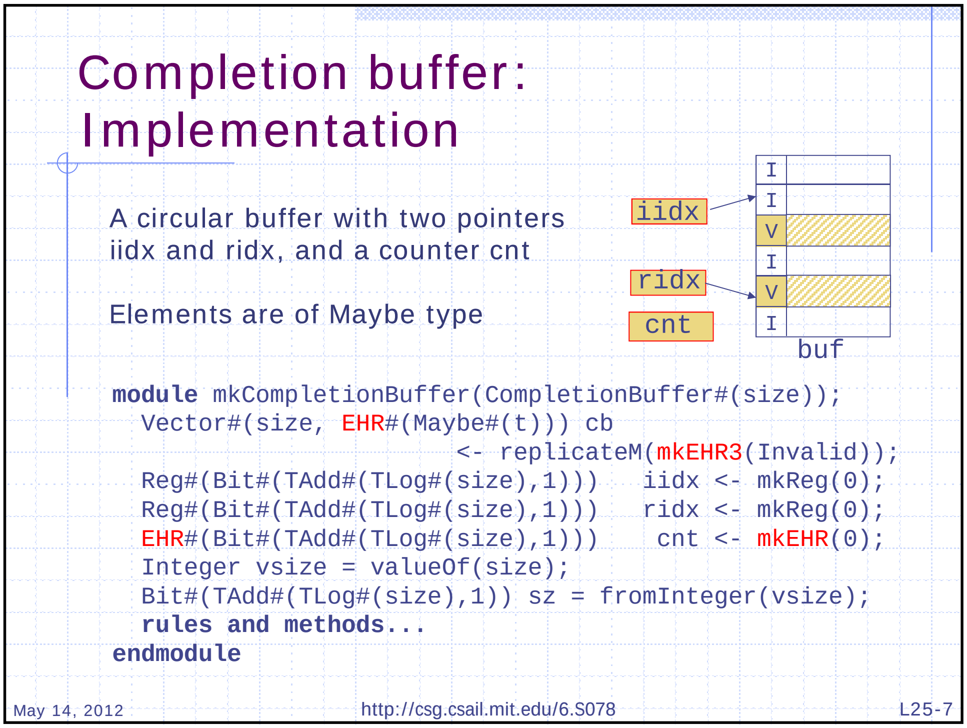
```
module mkNBfifoCache(Cache);
  cBuf <- mkCompletionBuffer;
  nbCache <- mkNBtaggedCache;
  method Action req(MemReq x);
    tok <- cBuf.getToken;
    nbCache.req({req:x, tag:tok});
  endmethod
  method MemResp resp
    return cBuf.getResult
  endmethod
  method Action respDeq;
    cBuf.deqResult;
  endmethod
  rule nbCacheResponse;
   cBuf.put(nbCache.resp);
   nbCache.respDeq;
  endrule
endmodule
```

req

resp

respDeq

cbuf



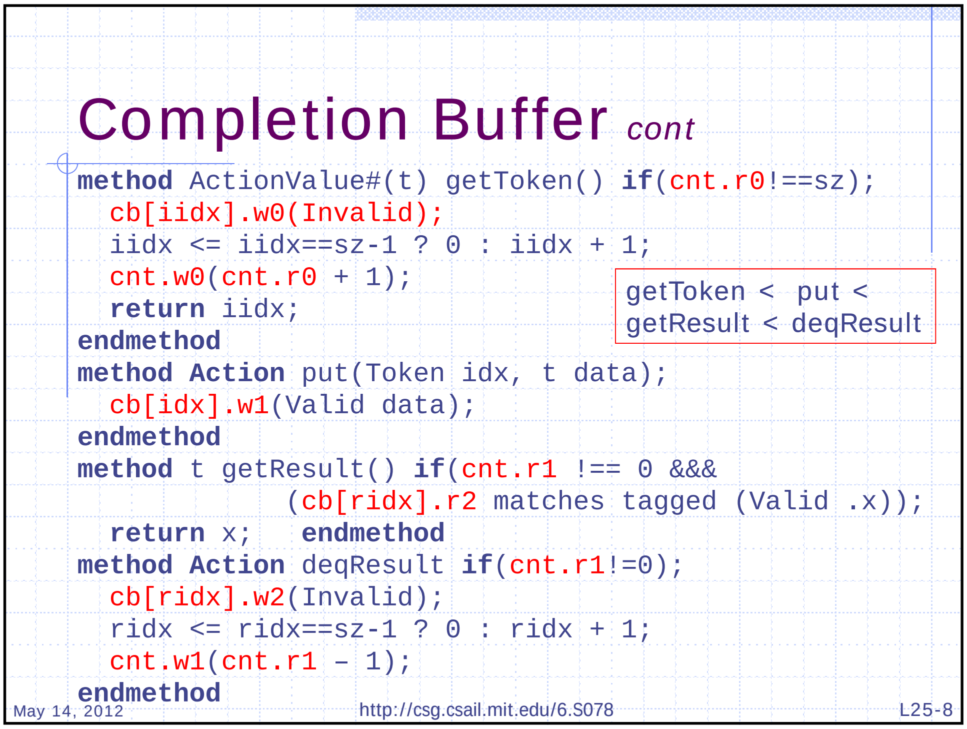
## Completion buffer: Implementation

A circular buffer with two pointers iidx and ridx, and a counter cnt

Elements are of Maybe type

iidx
ridx
cnt
I
I
V
I
V
I
buf

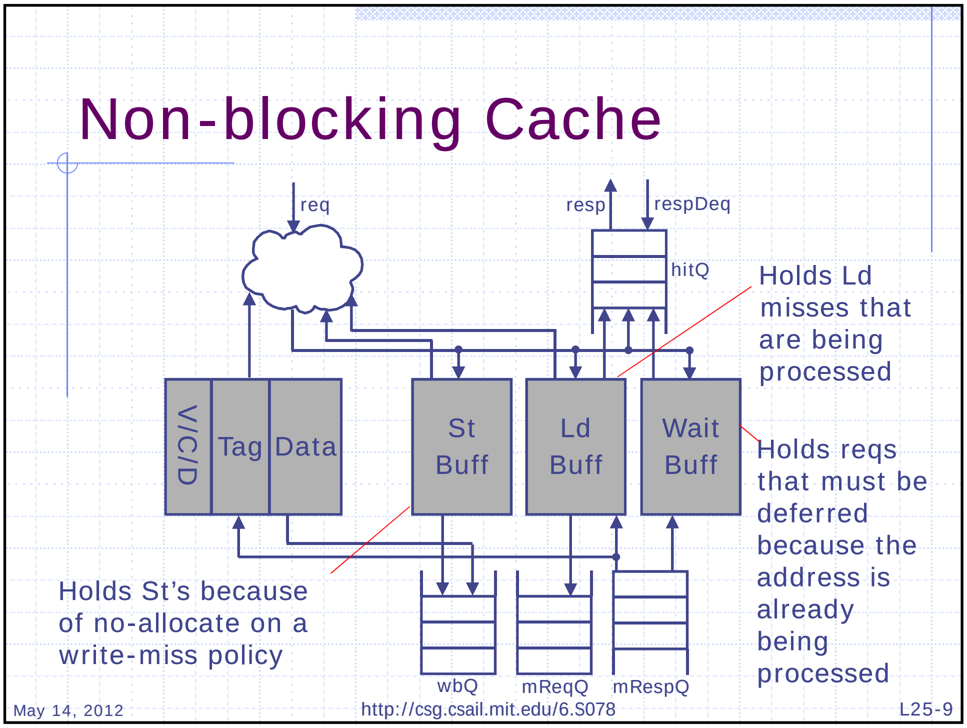
```
module mkCompletionBuffer(CompletionBuffer#(size));
  Vector#(size, EHR#(Maybe#(t))) cb
                              <- replicateM(mkEHR3(Invalid));
  Reg#(Bit#(TAdd#(TLog#(size),1)))   iidx <- mkReg(0);
  Reg#(Bit#(TAdd#(TLog#(size),1)))   ridx <- mkReg(0);
  EHR#(Bit#(TAdd#(TLog#(size),1)))    cnt <- mkEHR(0);
  Integer vsize = valueOf(size);
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);
  rules and methods...
endmodule
```



## Completion Buffer *cont*

```
method ActionValue#(t) getToken() if(cnt.r0!==sz);
  cb[iidx].w0(Invalid);
  iidx <= iidx==sz-1 ? 0 : iidx + 1;
  cnt.w0(cnt.r0 + 1);
  return iidx;
endmethod
method Action put(Token idx, t data);
  cb[idx].w1(Valid data);
endmethod
method t getResult() if(cnt.r1 !== 0 &&&
             (cb[ridx].r2 matches tagged (Valid .x));
  return x;   endmethod
method Action deqResult if(cnt.r1!=0);
  cb[ridx].w2(Invalid);
  ridx <= ridx==sz-1 ? 0 : ridx + 1;
  cnt.w1(cnt.r1 – 1);
endmethod
```

getToken < put < getResult < deqResult

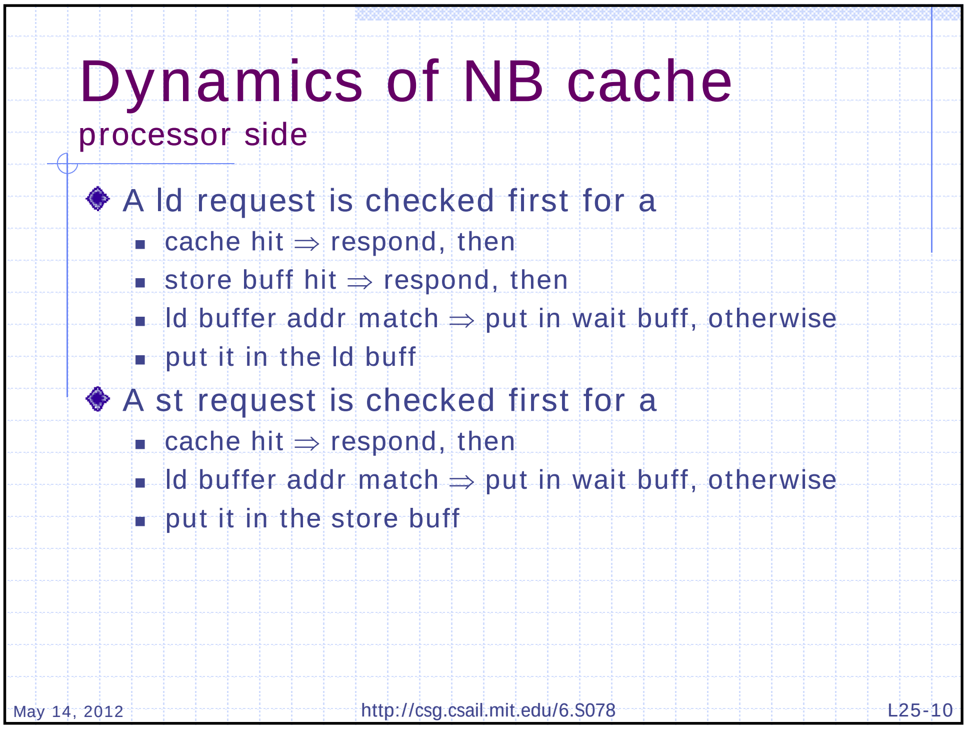


# Non-blocking Cache

Holds Ld misses that are being processed

Holds reqs that must be deferred because the address is already being processed

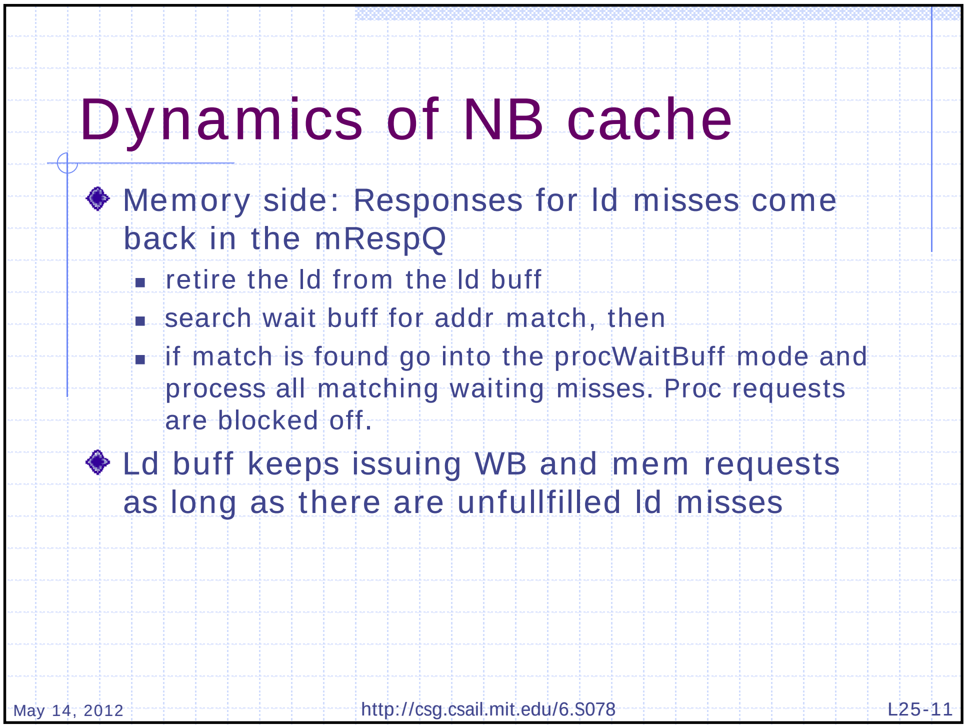
Holds St's because of no-allocate on a write-miss policy



# Dynamics of NB cache

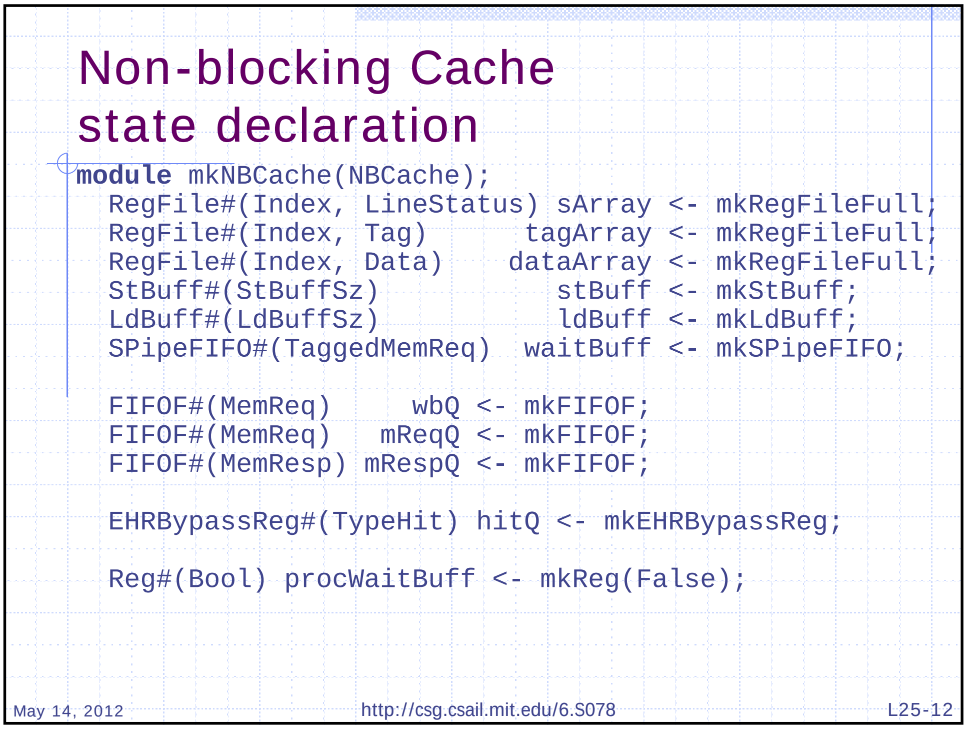
## processor side

- A ld request is checked first for a
  - cache hit ⇒ respond, then
  - store buff hit ⇒ respond, then
  - ld buffer addr match ⇒ put in wait buff, otherwise
  - put it in the ld buff
- A st request is checked first for a
  - cache hit ⇒ respond, then
  - ld buffer addr match ⇒ put in wait buff, otherwise
  - put it in the store buff



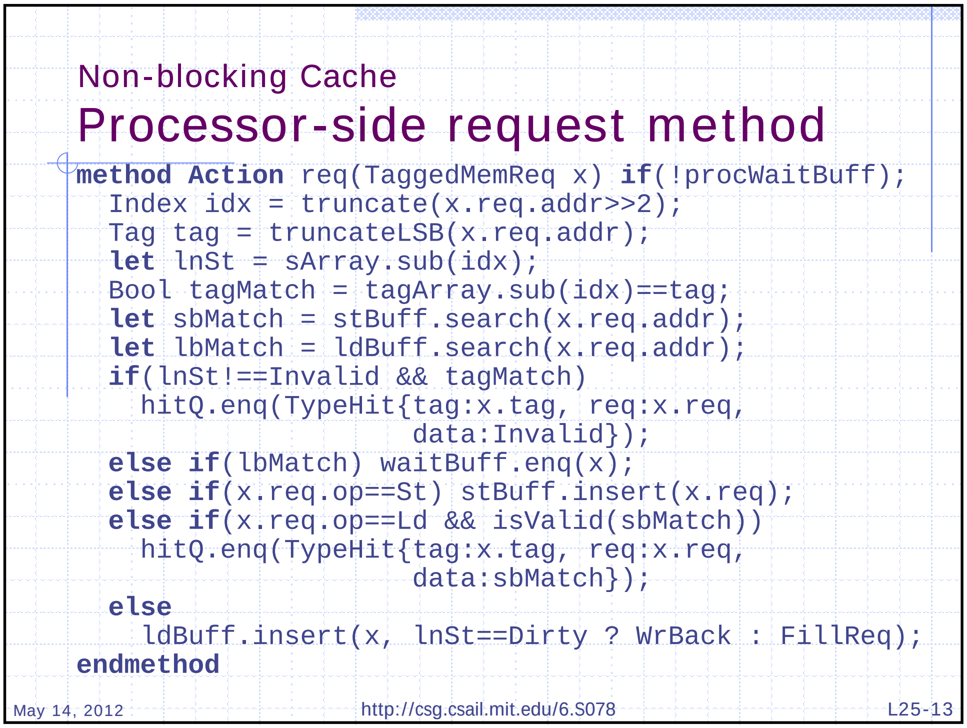
# Dynamics of NB cache

- Memory side: Responses for ld misses come back in the mRespQ
  - retire the ld from the ld buff
  - search wait buff for addr match, then
  - if match is found go into the procWaitBuff mode and process all matching waiting misses. Proc requests are blocked off.
- Ld buff keeps issuing WB and mem requests as long as there are unfullfilled ld misses



# Non-blocking Cache state declaration

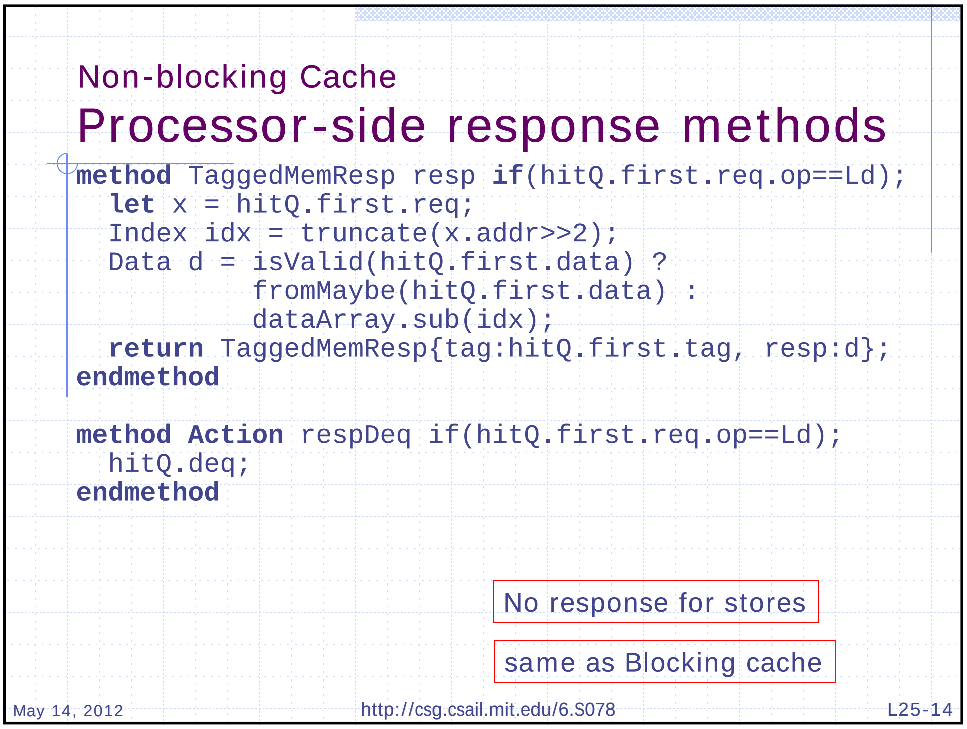
```
module mkNBCache(NBCache);
  RegFile#(Index, LineStatus) sArray <- mkRegFileFull;
  RegFile#(Index, Tag)      tagArray <- mkRegFileFull;
  RegFile#(Index, Data)    dataArray <- mkRegFileFull;
  StBuff#(StBuffSz)           stBuff <- mkStBuff;
  LdBuff#(LdBuffSz)           ldBuff <- mkLdBuff;
  SPipeFIFO#(TaggedMemReq)  waitBuff <- mkSPipeFIFO;

  FIFOF#(MemReq)     wbQ <- mkFIFOF;
  FIFOF#(MemReq)   mReqQ <- mkFIFOF;
  FIFOF#(MemResp) mRespQ <- mkFIFOF;

  EHRBypassReg#(TypeHit) hitQ <- mkEHRBypassReg;

  Reg#(Bool) procWaitBuff <- mkReg(False);
```



Non-blocking Cache

## Processor-side request method

```
method Action req(TaggedMemReq x) if(!procWaitBuff);
  Index idx = truncate(x.req.addr>>2);
  Tag tag = truncateLSB(x.req.addr);
  let lnSt = sArray.sub(idx);
  Bool tagMatch = tagArray.sub(idx)==tag;
  let sbMatch = stBuff.search(x.req.addr);
  let lbMatch = ldBuff.search(x.req.addr);
  if(lnSt!==Invalid && tagMatch)
    hitQ.enq(TypeHit{tag:x.tag, req:x.req,
                     data:Invalid});
  else if(lbMatch) waitBuff.enq(x);
  else if(x.req.op==St) stBuff.insert(x.req);
  else if(x.req.op==Ld && isValid(sbMatch))
    hitQ.enq(TypeHit{tag:x.tag, req:x.req,
                     data:sbMatch});
  else
    ldBuff.insert(x, lnSt==Dirty ? WrBack : FillReq);
endmethod
```



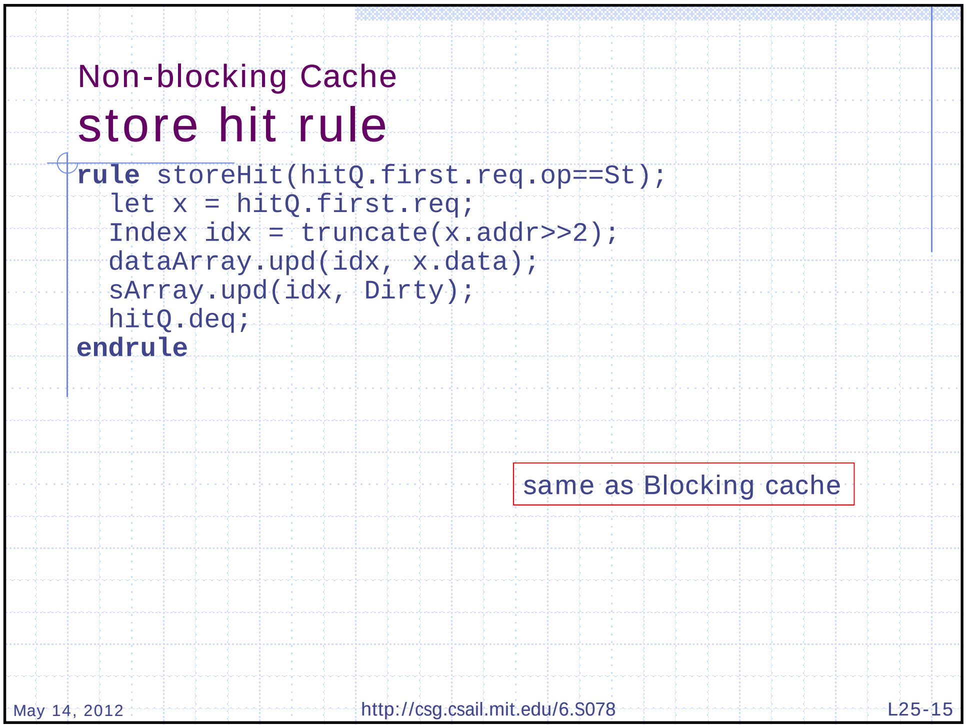
Non-blocking Cache

## Processor-side response methods

```
method TaggedMemResp resp if(hitQ.first.req.op==Ld);
  let x = hitQ.first.req;
  Index idx = truncate(x.addr>>2);
  Data d = isValid(hitQ.first.data) ?
             fromMaybe(hitQ.first.data) :
             dataArray.sub(idx);
  return TaggedMemResp{tag:hitQ.first.tag, resp:d};
endmethod

method Action respDeq if(hitQ.first.req.op==Ld);
  hitQ.deq;
endmethod
```

No response for stores

same as Blocking cache

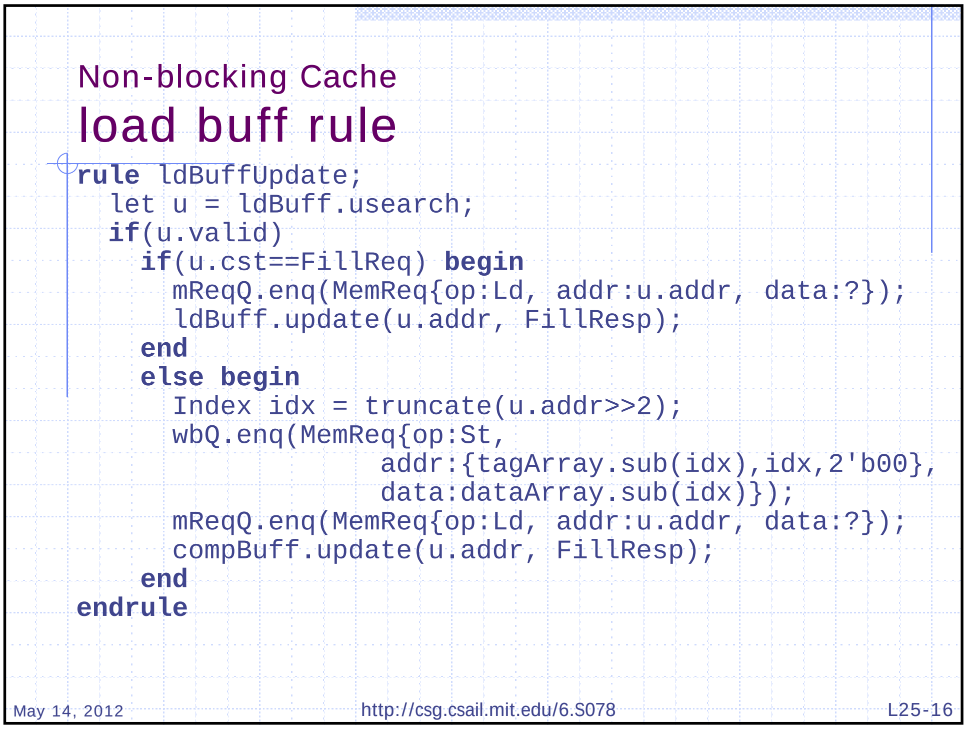


## Non-blocking Cache

# store hit rule

```
rule storeHit(hitQ.first.req.op==St);
  let x = hitQ.first.req;
  Index idx = truncate(x.addr>>2);
  dataArray.upd(idx, x.data);
  sArray.upd(idx, Dirty);
  hitQ.deq;
endrule
```

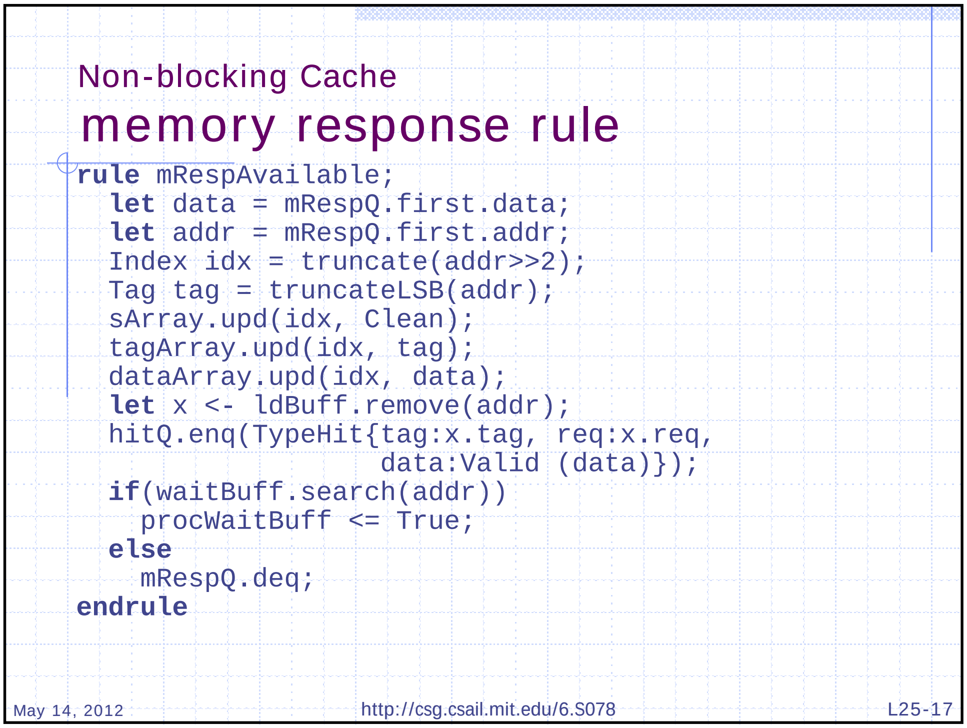
same as Blocking cache



## Non-blocking Cache

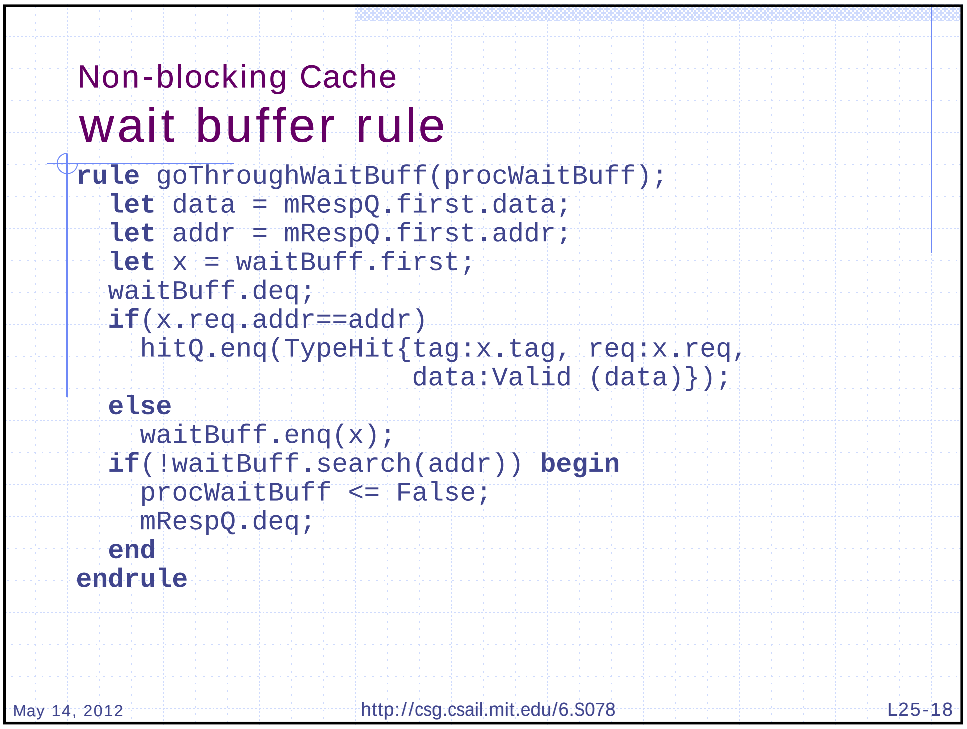
# load buff rule

```
rule ldBuffUpdate;
  let u = ldBuff.usearch;
  if(u.valid)
    if(u.cst==FillReq) begin
      mReqQ.enq(MemReq{op:Ld, addr:u.addr, data:?});
      ldBuff.update(u.addr, FillResp);
    end
    else begin
      Index idx = truncate(u.addr>>2);
      wbQ.enq(MemReq{op:St,
                   addr:{tagArray.sub(idx),idx,2'b00},
                   data:dataArray.sub(idx)});
      mReqQ.enq(MemReq{op:Ld, addr:u.addr, data:?});
      compBuff.update(u.addr, FillResp);
    end
endrule
```



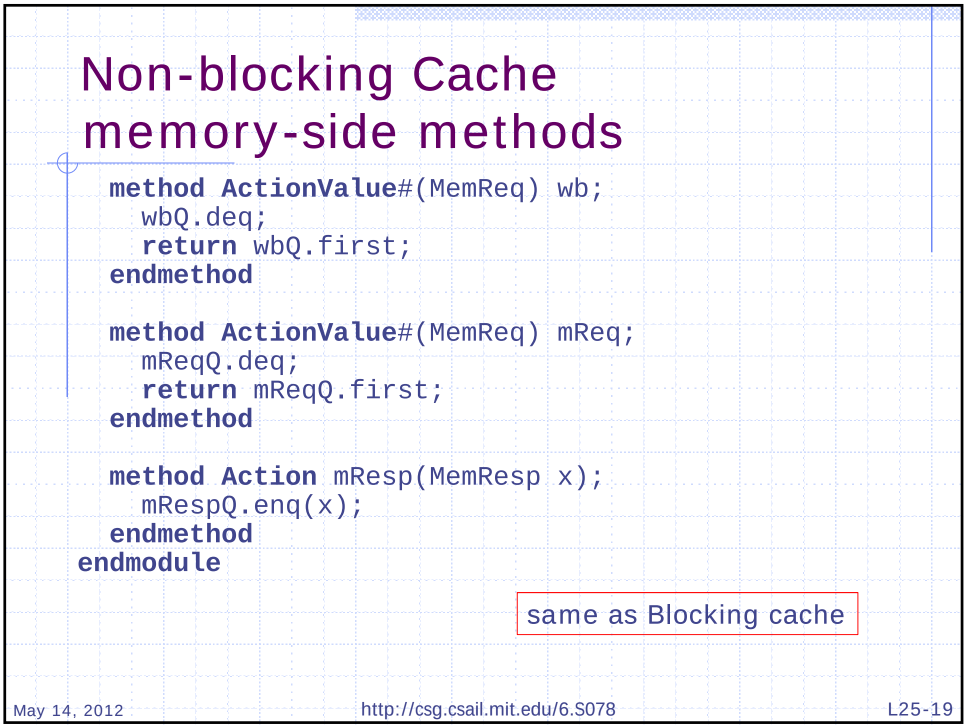
## Non-blocking Cache
# memory response rule

```
rule mRespAvailable;
  let data = mRespQ.first.data;
  let addr = mRespQ.first.addr;
  Index idx = truncate(addr>>2);
  Tag tag = truncateLSB(addr);
  sArray.upd(idx, Clean);
  tagArray.upd(idx, tag);
  dataArray.upd(idx, data);
  let x <- ldBuff.remove(addr);
  hitQ.enq(TypeHit{tag:x.tag, req:x.req,
                   data:Valid (data)});
  if(waitBuff.search(addr))
    procWaitBuff <= True;
  else
    mRespQ.deq;
endrule
```



## Non-blocking Cache
# wait buffer rule

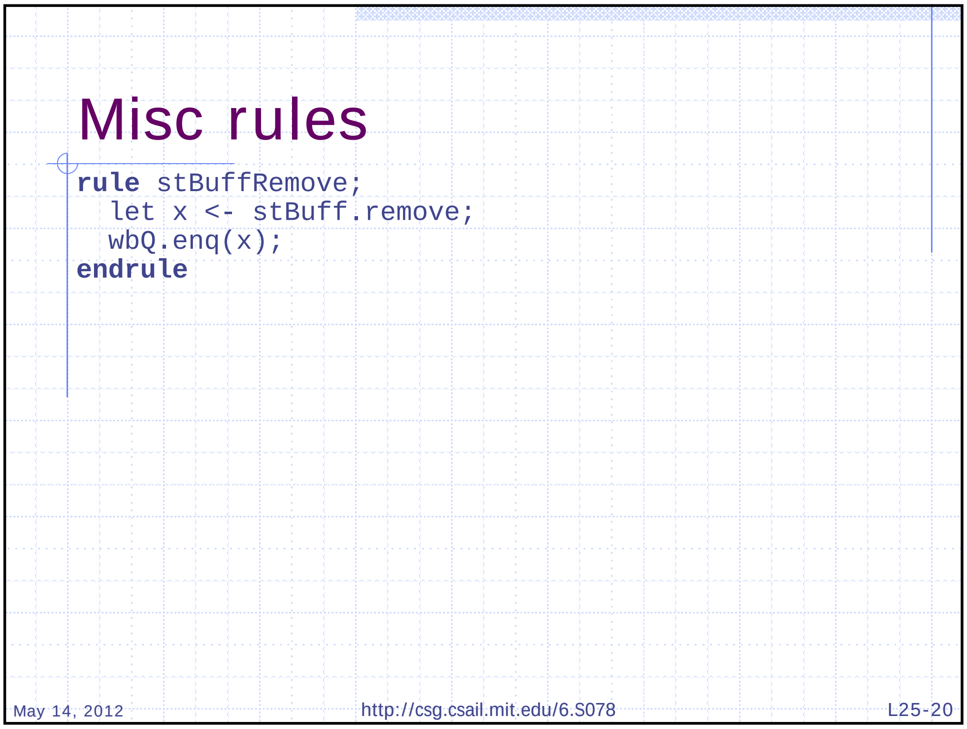
```
rule goThroughWaitBuff(procWaitBuff);
  let data = mRespQ.first.data;
  let addr = mRespQ.first.addr;
  let x = waitBuff.first;
  waitBuff.deq;
  if(x.req.addr==addr)
    hitQ.enq(TypeHit{tag:x.tag, req:x.req,
                     data:Valid (data)});
  else
    waitBuff.enq(x);
  if(!waitBuff.search(addr)) begin
    procWaitBuff <= False;
    mRespQ.deq;
  end
endrule
```



# Non-blocking Cache memory-side methods

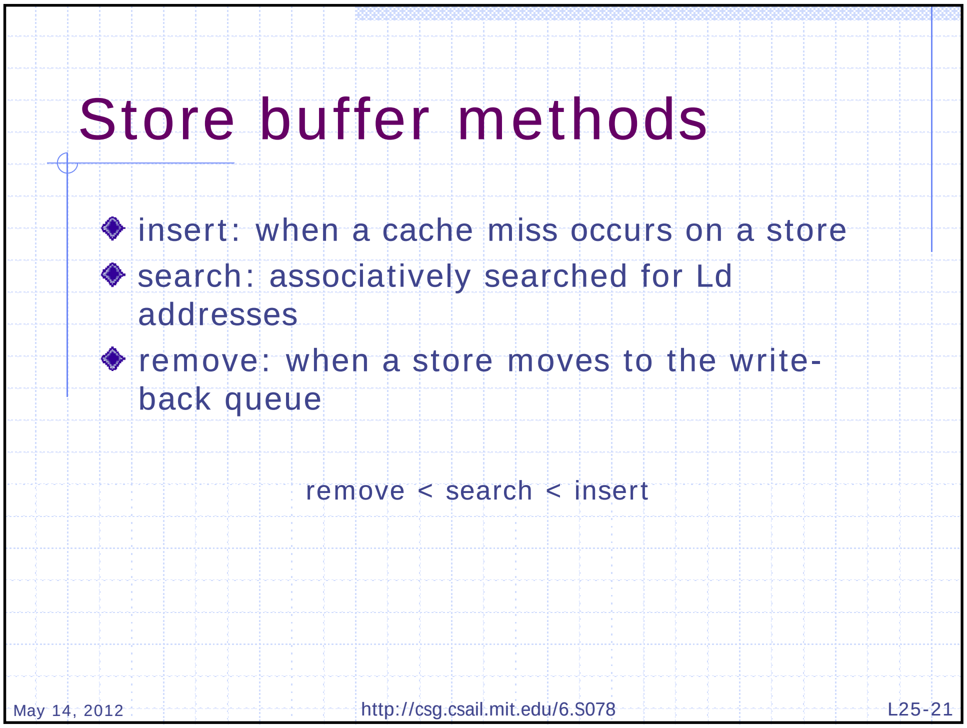
```
  method ActionValue#(MemReq) wb;
    wbQ.deq;
    return wbQ.first;
  endmethod

  method ActionValue#(MemReq) mReq;
    mReqQ.deq;
    return mReqQ.first;
  endmethod

  method Action mResp(MemResp x);
    mRespQ.enq(x);
  endmethod
endmodule
```

same as Blocking cache



# Misc rules

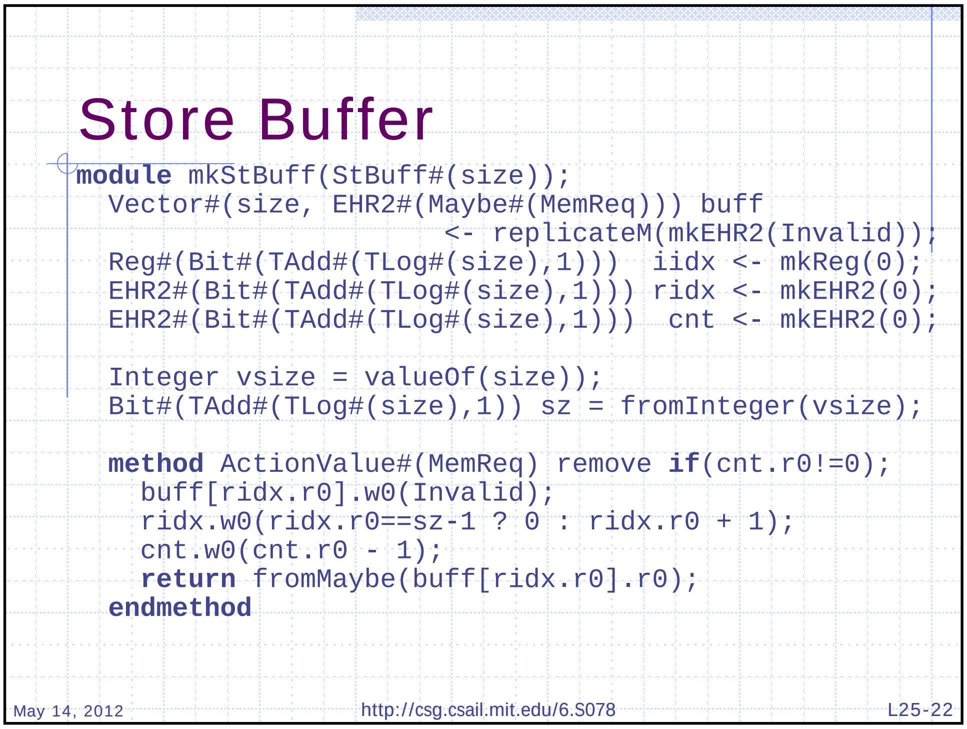
```
rule stBuffRemove;
  let x <- stBuff.remove;
  wbQ.enq(x);
endrule
```



## Store buffer methods

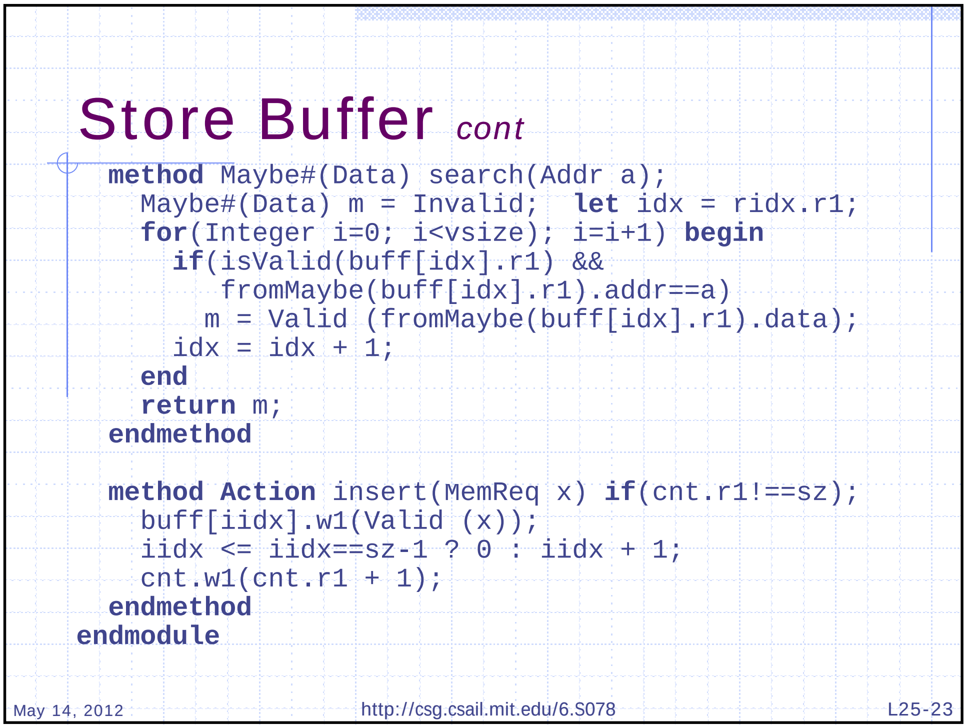
- insert: when a cache miss occurs on a store
- search: associatively searched for Ld addresses
- remove: when a store moves to the write-back queue

remove < search < insert



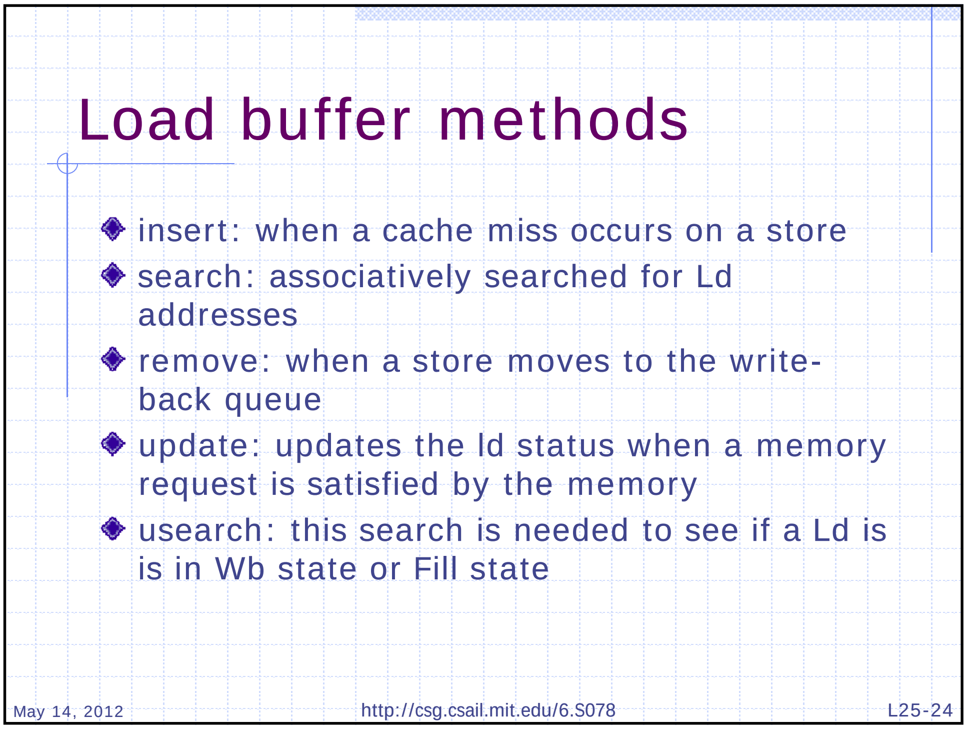
## Store Buffer

```
module mkStBuff(StBuff#(size));
  Vector#(size, EHR2#(Maybe#(MemReq))) buff
                          <- replicateM(mkEHR2(Invalid));
  Reg#(Bit#(TAdd#(TLog#(size),1)))  iidx <- mkReg(0);
  EHR2#(Bit#(TAdd#(TLog#(size),1))) ridx <- mkEHR2(0);
  EHR2#(Bit#(TAdd#(TLog#(size),1)))  cnt <- mkEHR2(0);

  Integer vsize = valueOf(size));
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);

  method ActionValue#(MemReq) remove if(cnt.r0!=0);
    buff[ridx.r0].w0(Invalid);
    ridx.w0(ridx.r0==sz-1 ? 0 : ridx.r0 + 1);
    cnt.w0(cnt.r0 - 1);
    return fromMaybe(buff[ridx.r0].r0);
  endmethod
```



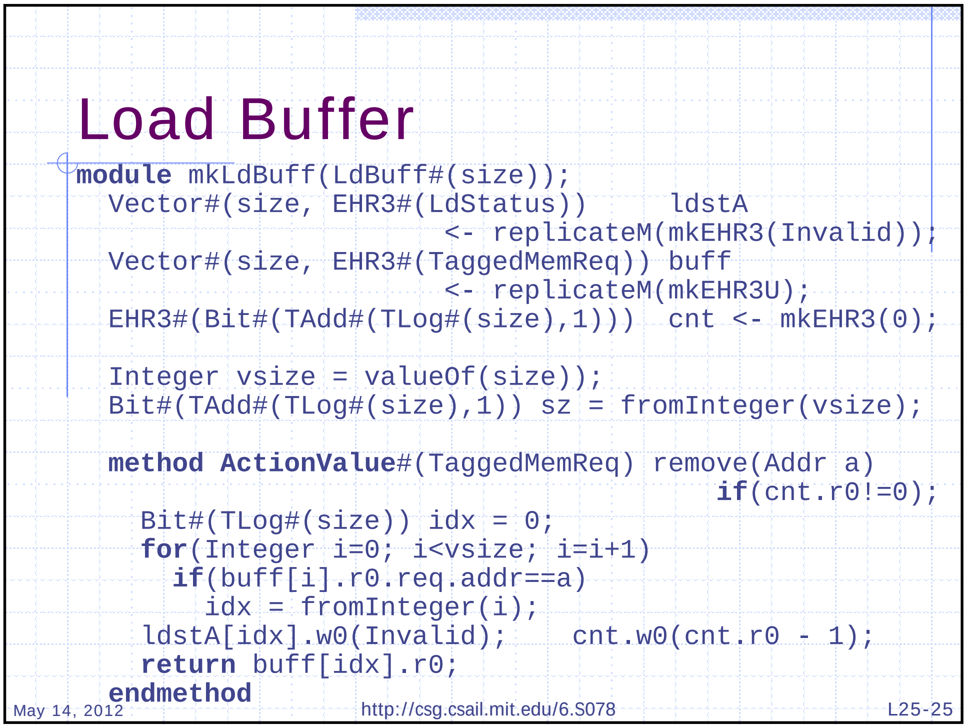
## Store Buffer *cont*

```
  method Maybe#(Data) search(Addr a);
    Maybe#(Data) m = Invalid;  let idx = ridx.r1;
    for(Integer i=0; i<vsize); i=i+1) begin
      if(isValid(buff[idx].r1) &&
         fromMaybe(buff[idx].r1).addr==a)
        m = Valid (fromMaybe(buff[idx].r1).data);
      idx = idx + 1;
    end
    return m;
  endmethod

  method Action insert(MemReq x) if(cnt.r1!==sz);
    buff[iidx].w1(Valid (x));
    iidx <= iidx==sz-1 ? 0 : iidx + 1;
    cnt.w1(cnt.r1 + 1);
  endmethod
endmodule
```



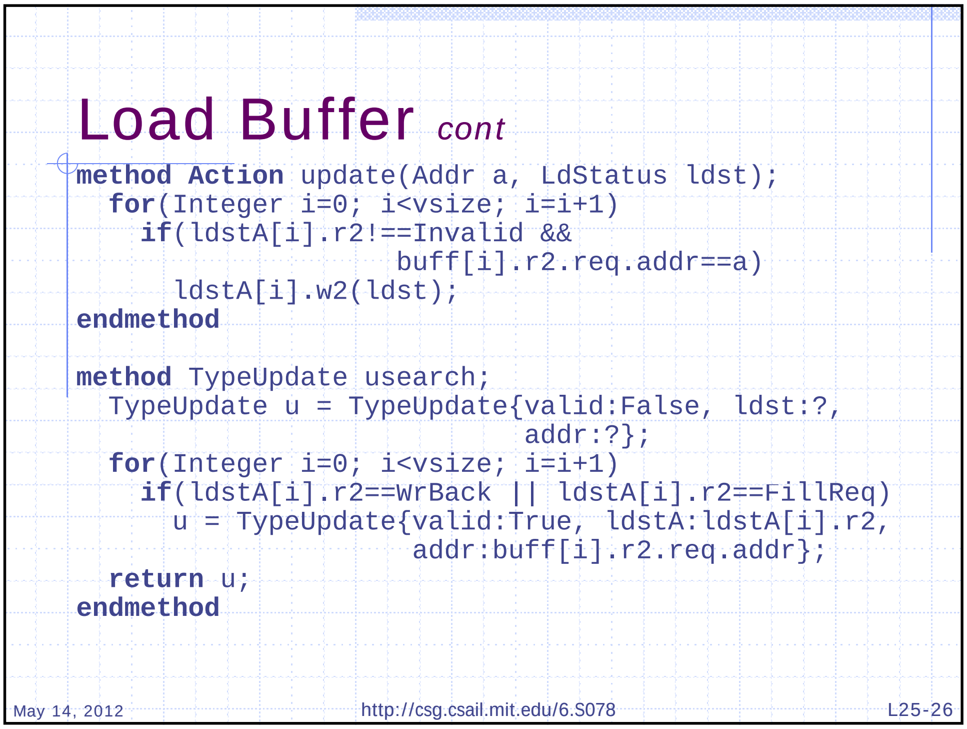
## Load buffer methods

- insert: when a cache miss occurs on a store
- search: associatively searched for Ld addresses
- remove: when a store moves to the write-back queue
- update: updates the ld status when a memory request is satisfied by the memory
- usearch: this search is needed to see if a Ld is is in Wb state or Fill state



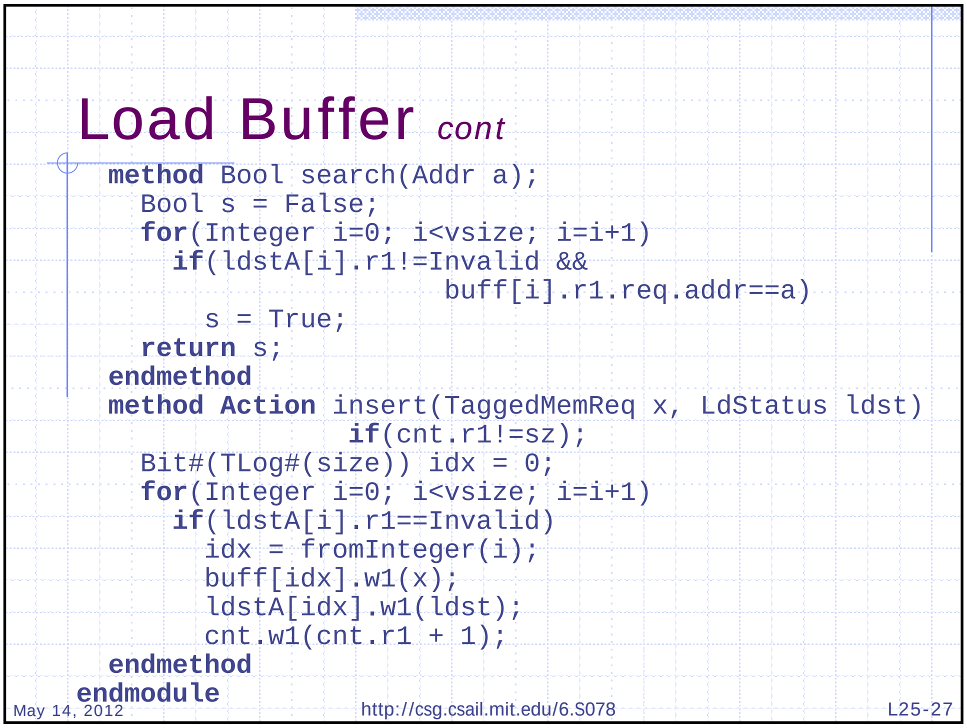
# Load Buffer

```
module mkLdBuff(LdBuff#(size));
  Vector#(size, EHR3#(LdStatus))     ldstA
                     <- replicateM(mkEHR3(Invalid));
  Vector#(size, EHR3#(TaggedMemReq)) buff
                     <- replicateM(mkEHR3U);
  EHR3#(Bit#(TAdd#(TLog#(size),1)))  cnt <- mkEHR3(0);

  Integer vsize = valueOf(size));
  Bit#(TAdd#(TLog#(size),1)) sz = fromInteger(vsize);

  method ActionValue#(TaggedMemReq) remove(Addr a)
                                        if(cnt.r0!=0);
    Bit#(TLog#(size)) idx = 0;
    for(Integer i=0; i<vsize; i=i+1)
      if(buff[i].r0.req.addr==a)
        idx = fromInteger(i);
    ldstA[idx].w0(Invalid);    cnt.w0(cnt.r0 - 1);
    return buff[idx].r0;
  endmethod
```



# Load Buffer *cont*

```
method Action update(Addr a, LdStatus ldst);
  for(Integer i=0; i<vsize; i=i+1)
    if(ldstA[i].r2!==Invalid &&
                    buff[i].r2.req.addr==a)
      ldstA[i].w2(ldst);
endmethod

method TypeUpdate usearch;
  TypeUpdate u = TypeUpdate{valid:False, ldst:?,
                            addr:?};
  for(Integer i=0; i<vsize; i=i+1)
    if(ldstA[i].r2==WrBack || ldstA[i].r2==FillReq)
      u = TypeUpdate{valid:True, ldstA:ldstA[i].r2,
                     addr:buff[i].r2.req.addr};
  return u;
endmethod
```



# Load Buffer *cont*

```
  method Bool search(Addr a);
    Bool s = False;
    for(Integer i=0; i<vsize; i=i+1)
      if(ldstA[i].r1!=Invalid &&
                         buff[i].r1.req.addr==a)
        s = True;
    return s;
  endmethod
  method Action insert(TaggedMemReq x, LdStatus ldst)
                 if(cnt.r1!=sz);
    Bit#(TLog#(size)) idx = 0;
    for(Integer i=0; i<vsize; i=i+1)
      if(ldstA[i].r1==Invalid)
        idx = fromInteger(i);
        buff[idx].w1(x);
        ldstA[idx].w1(ldst);
        cnt.w1(cnt.r1 + 1);
  endmethod
endmodule
```