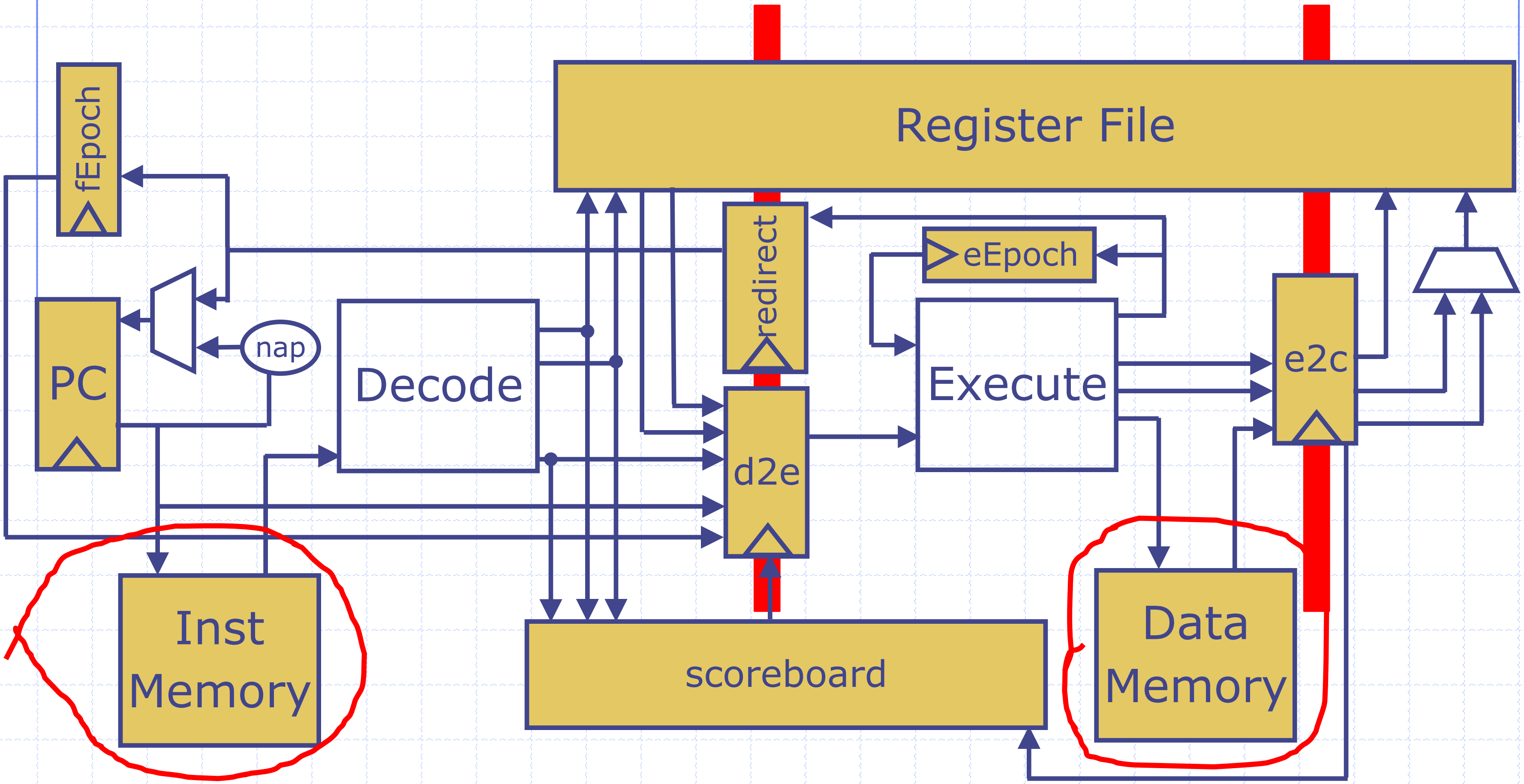
Constructive Computer Architecture

# Realistic Memories and Caches

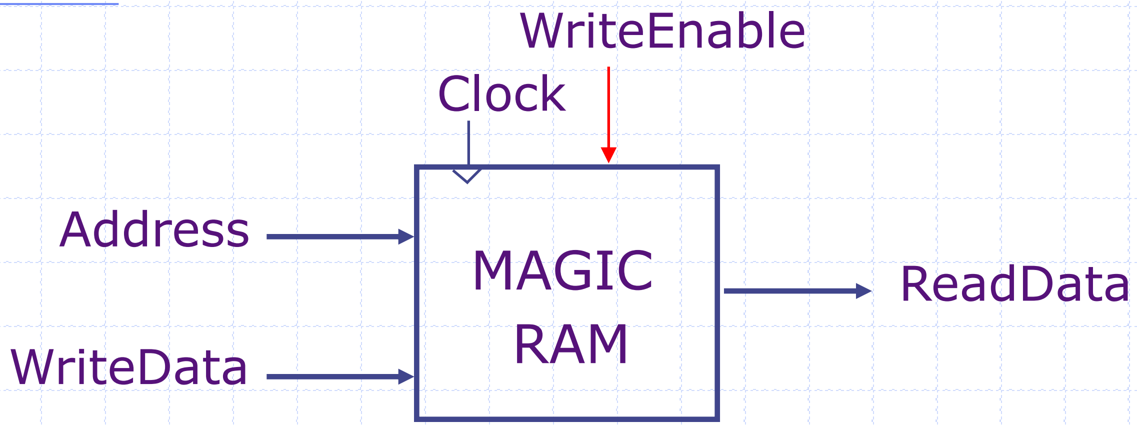
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

# Multistage Pipeline



The use of magic memories (combinational reads) makes such design unrealistic

# Magic Memory Model

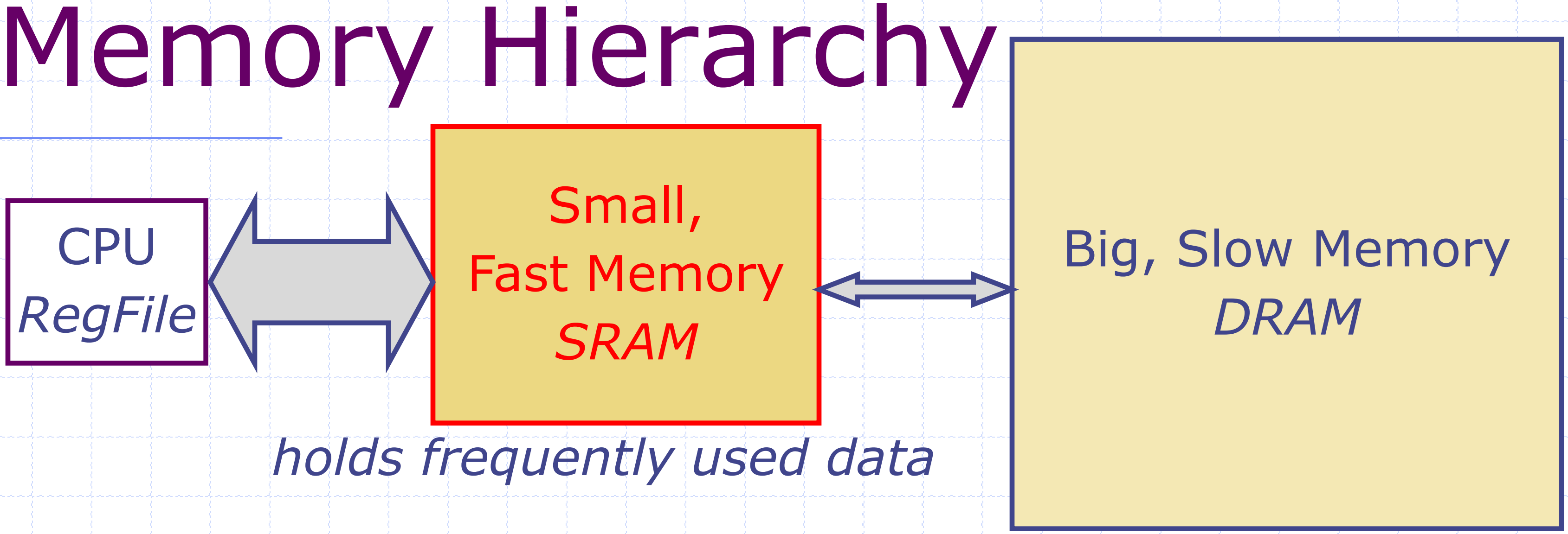


- Reads and writes are always completed in one cycle
  - a Read can be done any time (i.e. combinational)
  - If enabled, a Write is performed at the rising clock edge

    *(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



CPU

*RegFile*

Small,

Fast Memory

*SRAM*

Big, Slow Memory

*DRAM*

*holds frequently used data*

size: RegFile << SRAM << DRAM

latency: RegFile << SRAM << DRAM

bandwidth: on-chip >> off-chip

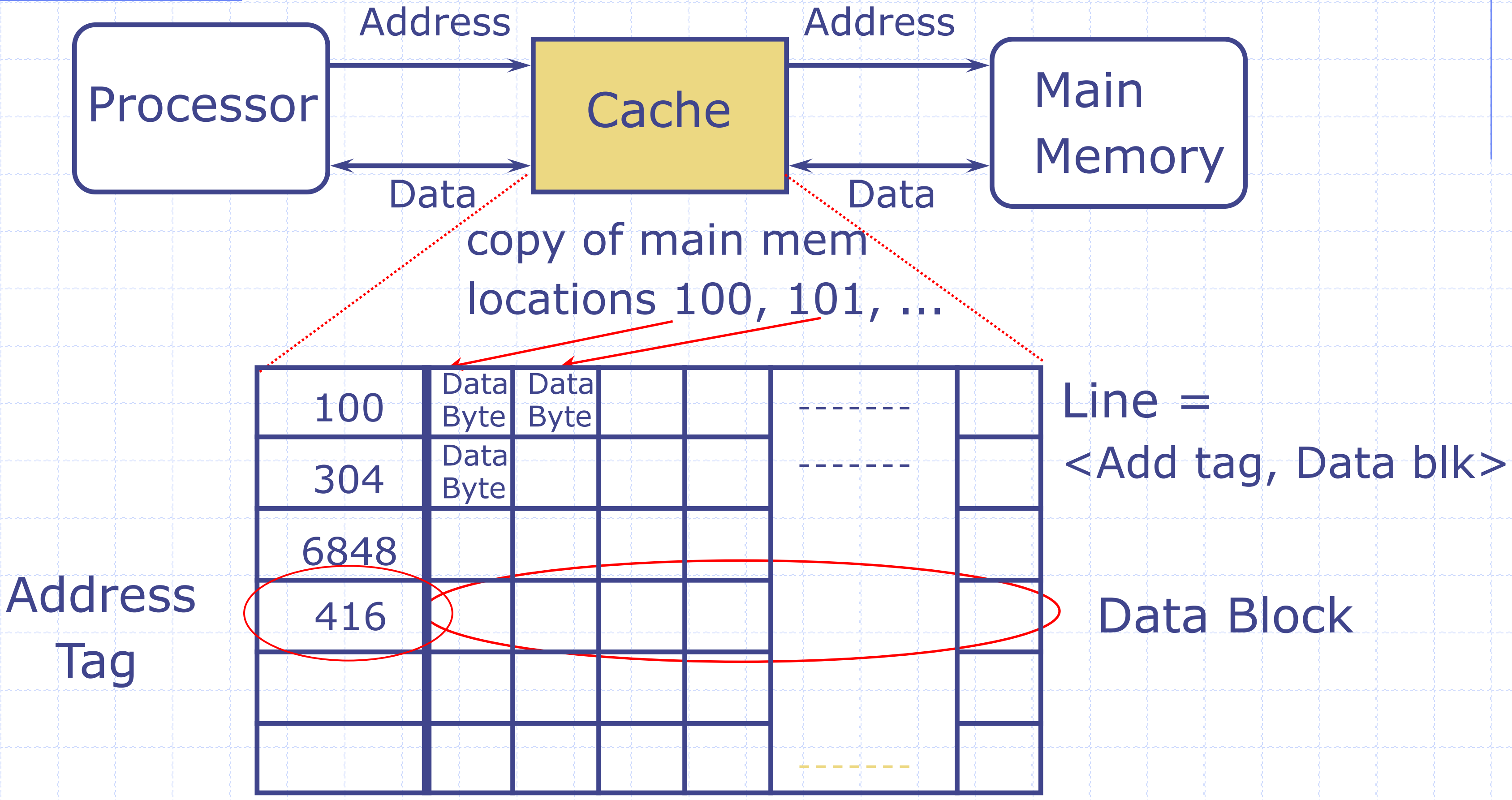
why?

On a data access:

*hit* (data ∈ fast memory) ⇒ low latency access

*miss* (data ∉ fast memory) ⇒ long latency access *(DRAM)*

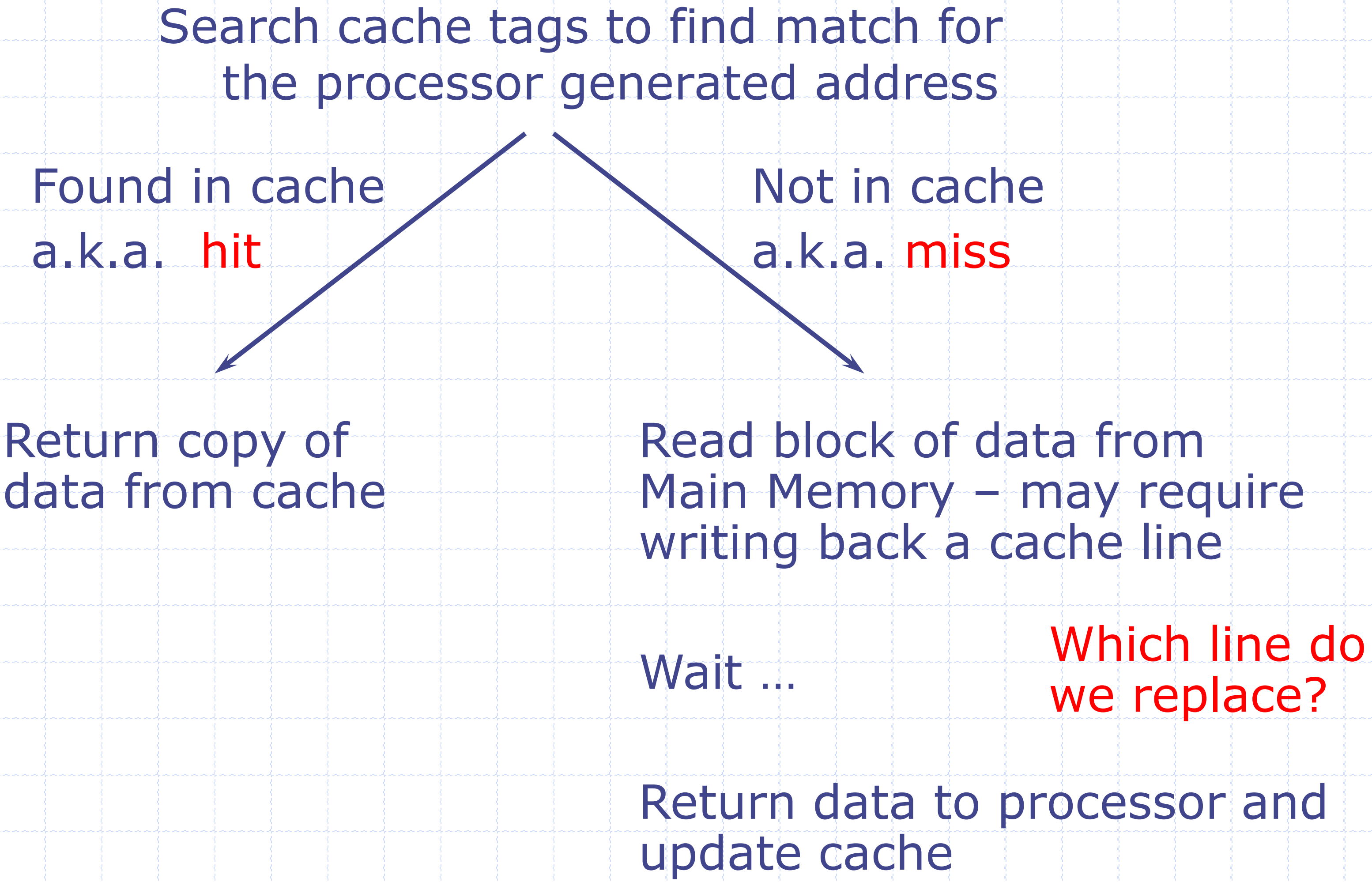
# Inside a Cache



How many bits are needed for the tag?

Enough to uniquely identify the block

# Cache Read



Search cache tags to find match for the processor generated address

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 …

Which line do we replace?

Return data to processor and update cache

# Write behavior

- On a write hit
  - Write-through: write to both cache and the next level memory
  - write-back: 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

# Cache Line Size

- A cache line usually holds more than one word
  - Reduces the number of tags and the tag size needed to identify memory locations
  - Spatial locality: Experience shows that if address x is referenced then addresses x+1, x+2 etc. are very likely to be referenced in a short time window
    - consider instruction streams, array and record accesses
  - Communication systems (e.g., bus) are often more efficient in transporting larger data sets

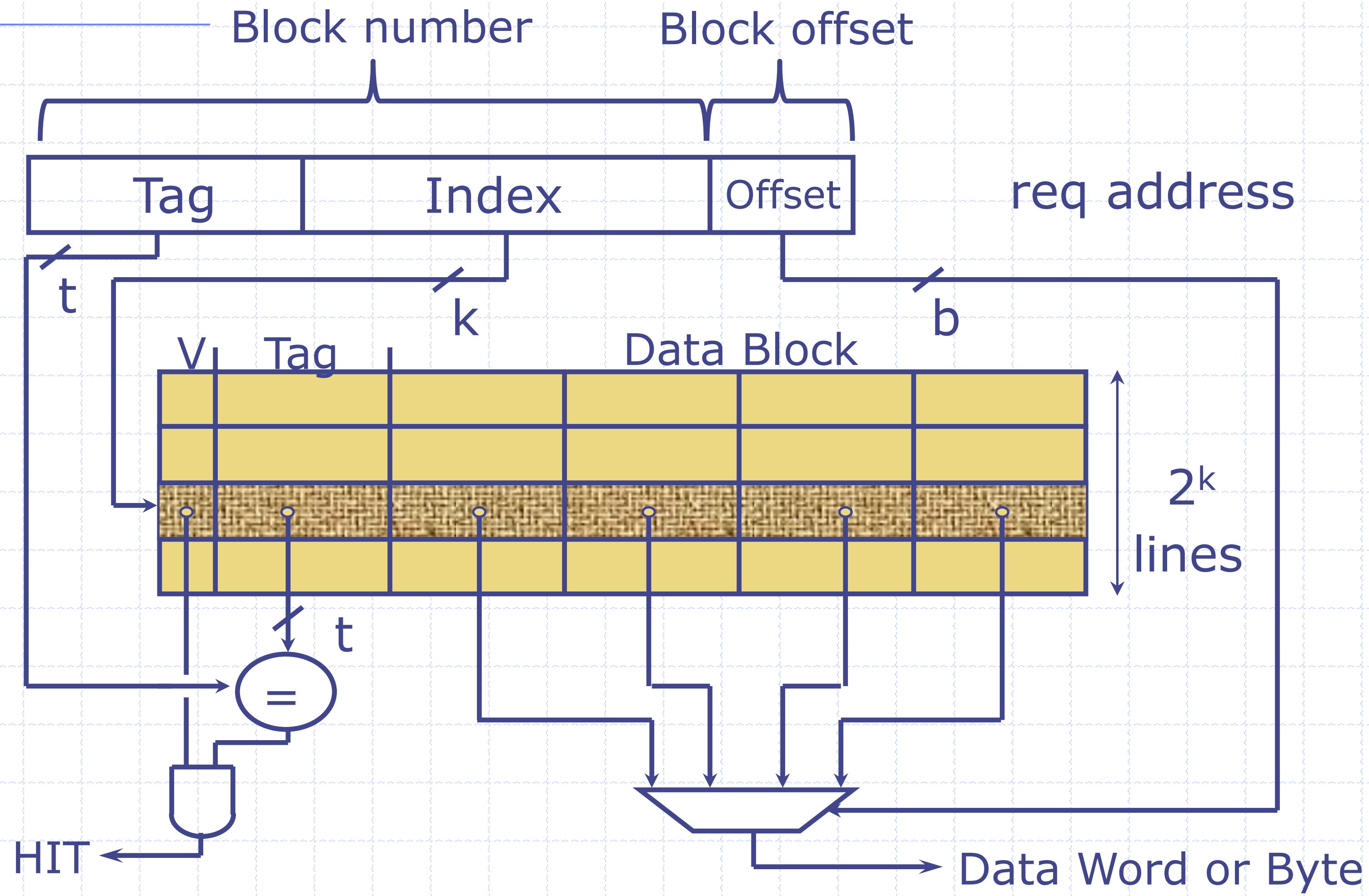
# Types of misses

- Compulsory misses (cold start)
  - 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
  - Usually multiple memory locations are mapped to the same cache location to simplify implementations
  - Thus it is possible that the designated cache location is full while there are empty locations in the cache.
  - Solution: Set-Associative Caches

# Internal Cache Organization

- Cache designs restrict where in cache a particular address can reside
  - *Direct mapped:* An address can reside in exactly one location in the cache. The cache location is typically determined by the lowest order address bits
  - *n-way Set associative:* An address can reside in any of the a set of n locations in the cache. The set is typically determine by the lowest order address bits

# Direct-Mapped Cache

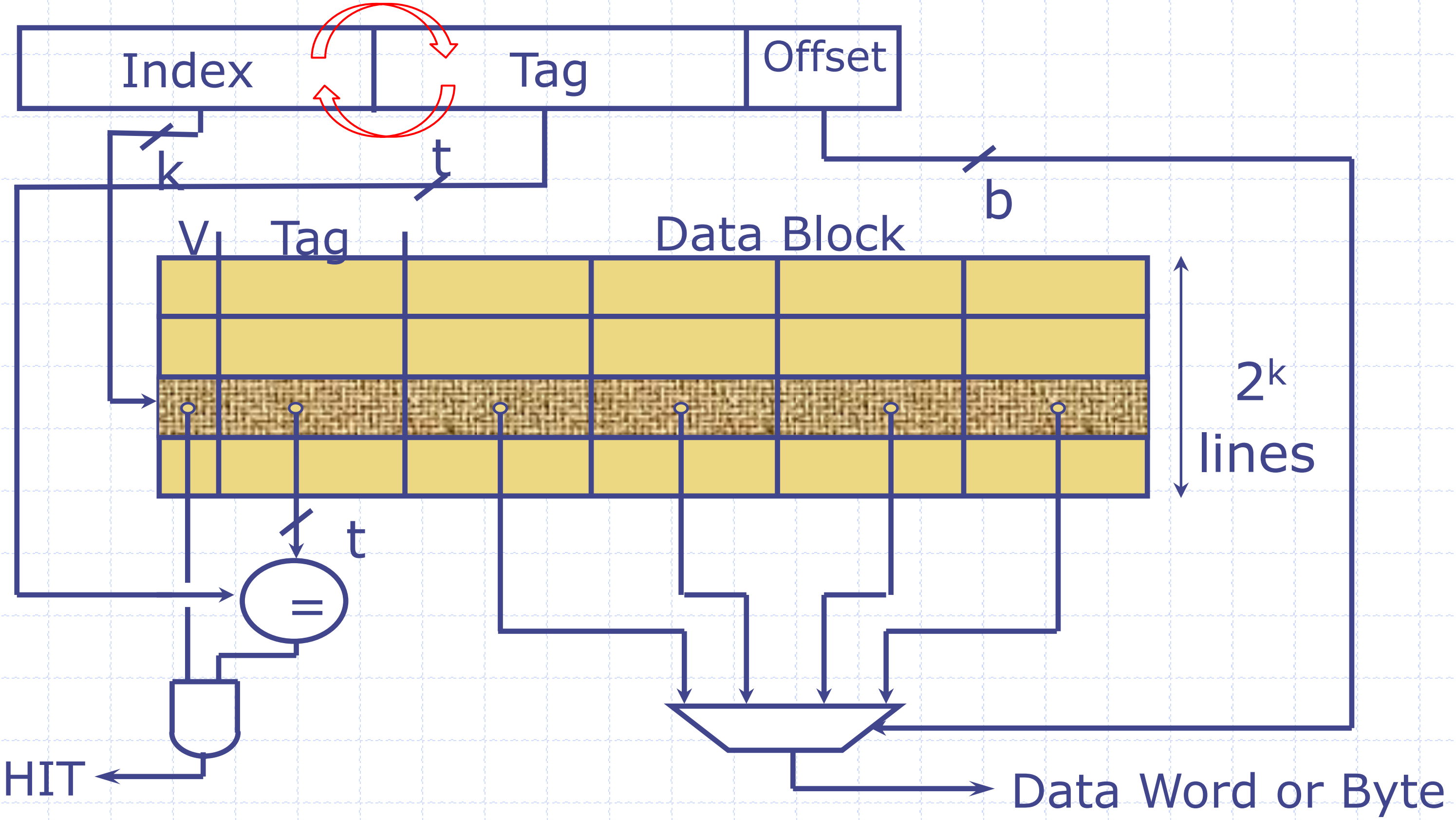


What is a bad reference pattern?

Strided = size of cache

# Direct Map Address Selection

*higher-order vs. lower-order address bits*



Why higher-order bits as tag may be undesirable?

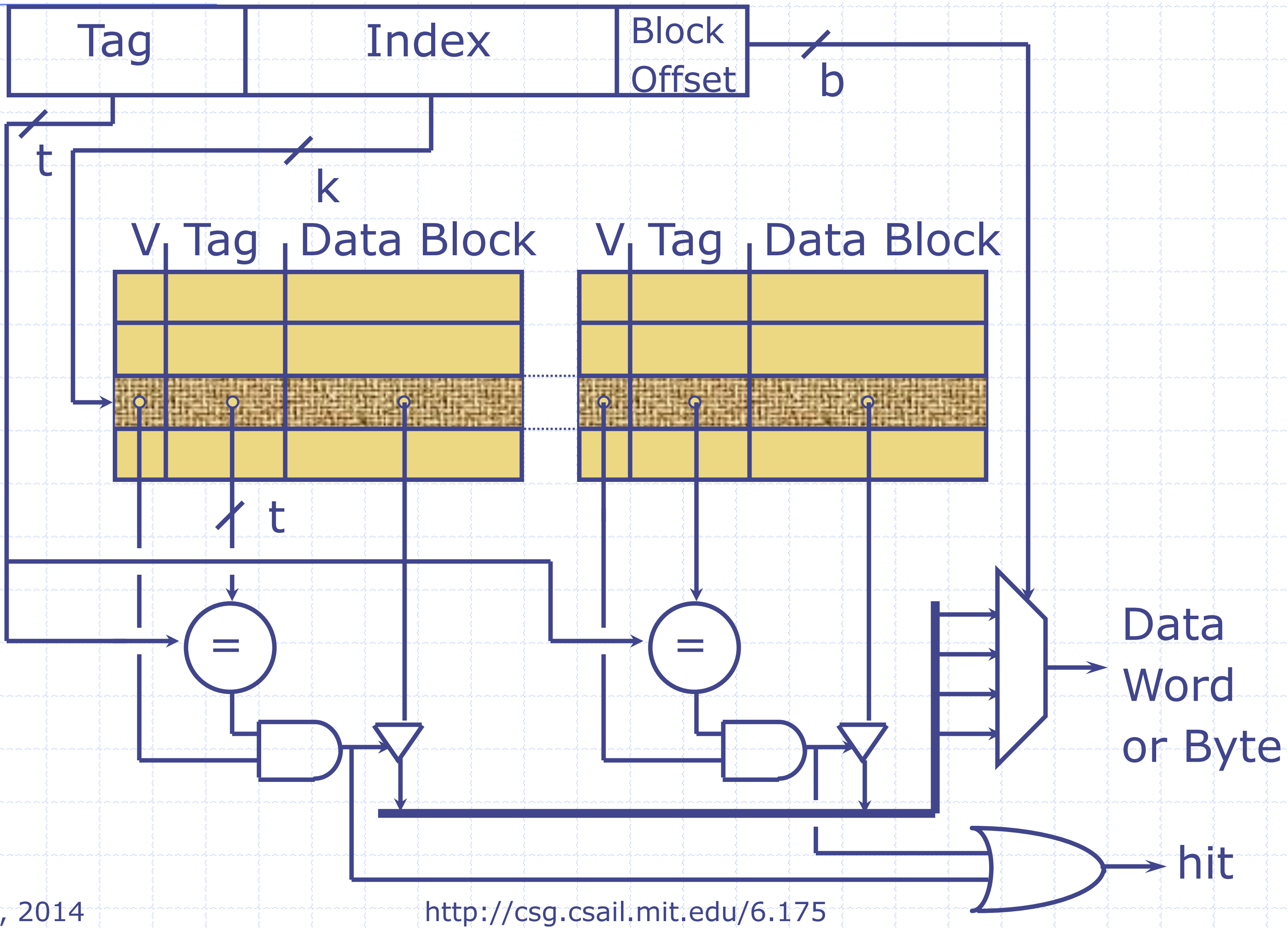
Spatially local blocks conflict

# Reduce Conflict Misses

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

- Associativity: Reduce conflict misses by allowing blocks to go to several sets in cache
  - 2-way set associative: each block can be mapped to one of 2 cache sets
  - Fully associative: each block can be mapped to any cache frame

# 2-Way Set-Associative Cache



# Replacement Policy

- In order to bring in a new cache line, usually another cache line has to be thrown out. Which one?
  - No choice in replacement if the cache is direct mapped
- Replacement policy for set-associative caches
  - One that is not dirty, i.e., has not been modified
    - In I-cache all lines are clean
    - In D-cache if a dirty line has to be thrown out then it must be written back first
  - Least recently used?
  - Most recently used?
  - Random?

How much is performance affected by the choice?

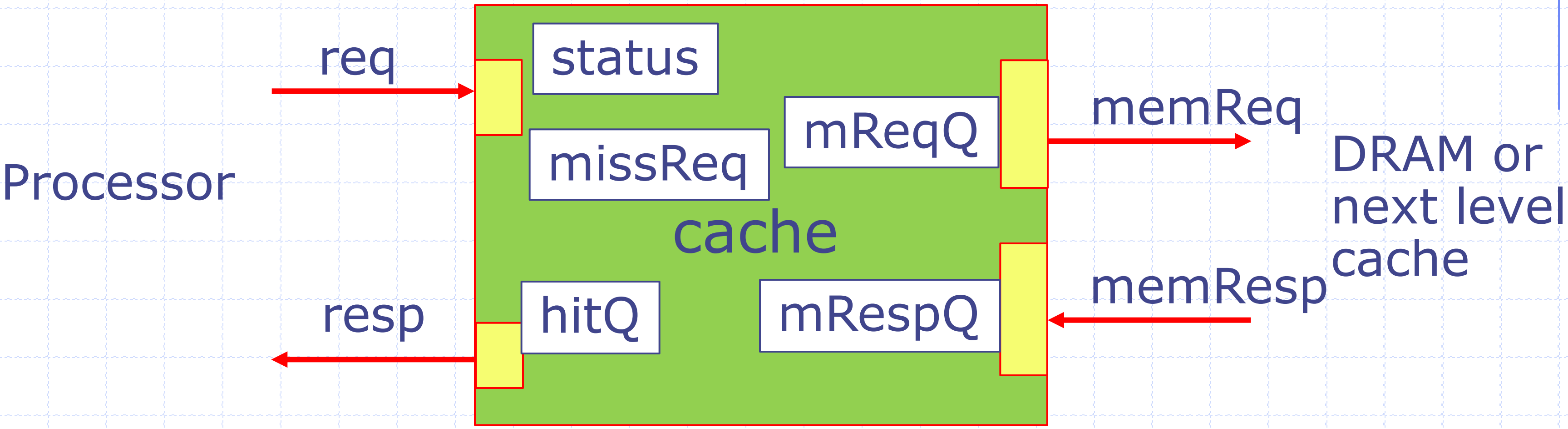
Difficult to know without benchmarks and quantitative measurements

# Blocking vs. Non-Blocking cache

- Blocking cache:
  - At most one outstanding miss
  - Cache must wait for memory to respond
  - Cache does not accept 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 first design a write-back, Write-miss allocate, Direct-mapped, blocking cache

# Blocking Cache Interface



```
interface Cache;
  method Action req(MemReq r);
  method ActionValue#(Data) resp;

  method ActionValue#(MemReq) memReq;
  method Action memResp(Line r);
endinterface
```

# Interface dynamics

- The cache either gets a hit and responds immediately, or it gets a miss, in which case it takes several steps to process the miss
- Reading the response dequeues it
- Requests and responses follow the FIFO order
- Methods are guarded, e.g., the cache may not be ready to accept a request because it is processing a miss
- A status register keeps track of the state of the cache while it is processing a miss

```
typedef enum {Ready, StartMiss, SendFillReq,
   WaitFillResp} CacheStatus deriving (Bits, Eq);
```

# Blocking Cache code structure

```
module mkCache(Cache);
    RegFile#(CacheIndex, Line) dataArray <-
                                       mkRegFileFull; …
    rule startMiss … endrule;

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

    method ActionValue#(MemReq) memReq …     endmethod;
    method Action memResp(Line r) …          endmethod;
endmodule
```

- Internal communications is in line sizes but the processor interface, e.g., the response from the hitQ is word size
- Let us assume the line size is 4 words
- CacheIndexSz + CacheTagSz + 4 = AddrSz

# Blocking cache
## state elements

```
RegFile#(CacheIndex, Line) dataArray <- mkRegFileFull;
RegFile#(CacheIndex, Maybe#(CacheTag))
                               tagArray <- mkRegFileFull;
RegFile#(CacheIndex, Bool) dirtyArray <- mkRegFileFull;

Fifo#(1, Data)       hitQ <- mkBypassFifo;
Reg#(MemReq)      missReq <- mkRegU;
Reg#(CacheStatus)  status <- mkReg(Ready);

Fifo#(2, MemReq) memReqQ <- mkCFFifo;
Fifo#(2, Line)  memRespQ <- mkCFFifo;

function CacheIndex getIdx(Addr addr)  = truncate(addr>>4);
function CacheTag getTag(Addr addr)    = truncateLSB(addr);
```

Tag and valid bits are kept together as a Maybe type

CF Fifos are preferable because they provide better decoupling. An extra cycle here may not affect the performance by much

```
truncate = truncateMSB
```

# Req method
## hit processing

It is straightforward to extend the cache interface to include a cacheline flush command

```
method Action req(MemReq r) if(status == Ready);
  let idx = getIdx(r.addr); let tag = getTag(r.addr);
  Bit#(2) wOffset = truncate(r.addr >> 2);
  let currTag = tagArray.sub(idx);
  let hit = isValid(currTag)?
               fromMaybe(?,currTag)==tag : False;
  if(hit) begin
    let x = dataArray.sub(idx);
    if(r.op == Ld) hitQ.enq(x[wOffset]);
    else begin x[wOffset]=r.data;
               dataArray.upd(idx, x);
               dirtyArray.upd(idx, True); end
  else begin missReq <= r; status <= StartMiss; end
endmethod
```

overwrite the appropriate word of the line

# Rest of the methods

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

method ActionValue#(MemReq) memReq;
  memReqQ.deq;
  return memReqQ.first;
endmethod

method Action memResp(Line r);
  memRespQ.enq(r);
endmethod
```

Memory side methods

# Start-miss and Send-fill rules

```
Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready
```

```
rule startMiss(status == StartMiss);
  let idx = getIdx(missReq.addr);
  let tag=tagArray.sub(idx); let dirty=dirtyArray.sub(idx);
  if(isValid(tag) && dirty) begin // write-back
    let addr = {fromMaybe(?,tag), idx, 4'b0};
    let data = dataArray.sub(idx);
    memReqQ.enq(MemReq{op: St, addr: addr, data: data});
                            end
  status <= SendFillReq;
endrule
```

```
Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready
```

```
rule sendFillReq (status == SendFillReq);
  memReqQ.enq(missReq);   status <= WaitFillResp;
endrule
```

# Wait-fill rule

```
Ready -> StartMiss -> SendFillReq -> WaitFillResp -> Ready
```

```
rule waitFillResp(status == WaitFillResp);
  let idx = getIdx(missReq.addr);
  let tag = getTag(missReq.addr);
  let data = memRespQ.first;
  tagArray.upd(idx, Valid (tag));
  if(missReq.op == Ld) begin
    dirtyArray.upd(idx,False);dataArray.upd(idx, data);
    hitQ.enq(data[wOffset]); end
  else begin data[wOffset] = missReq.data;
    dirtyArray.upd(idx,True); dataArray.upd(idx, data);
        end
  memRespQ.deq; status <= Ready;
endrule
```

# Hit and miss performance

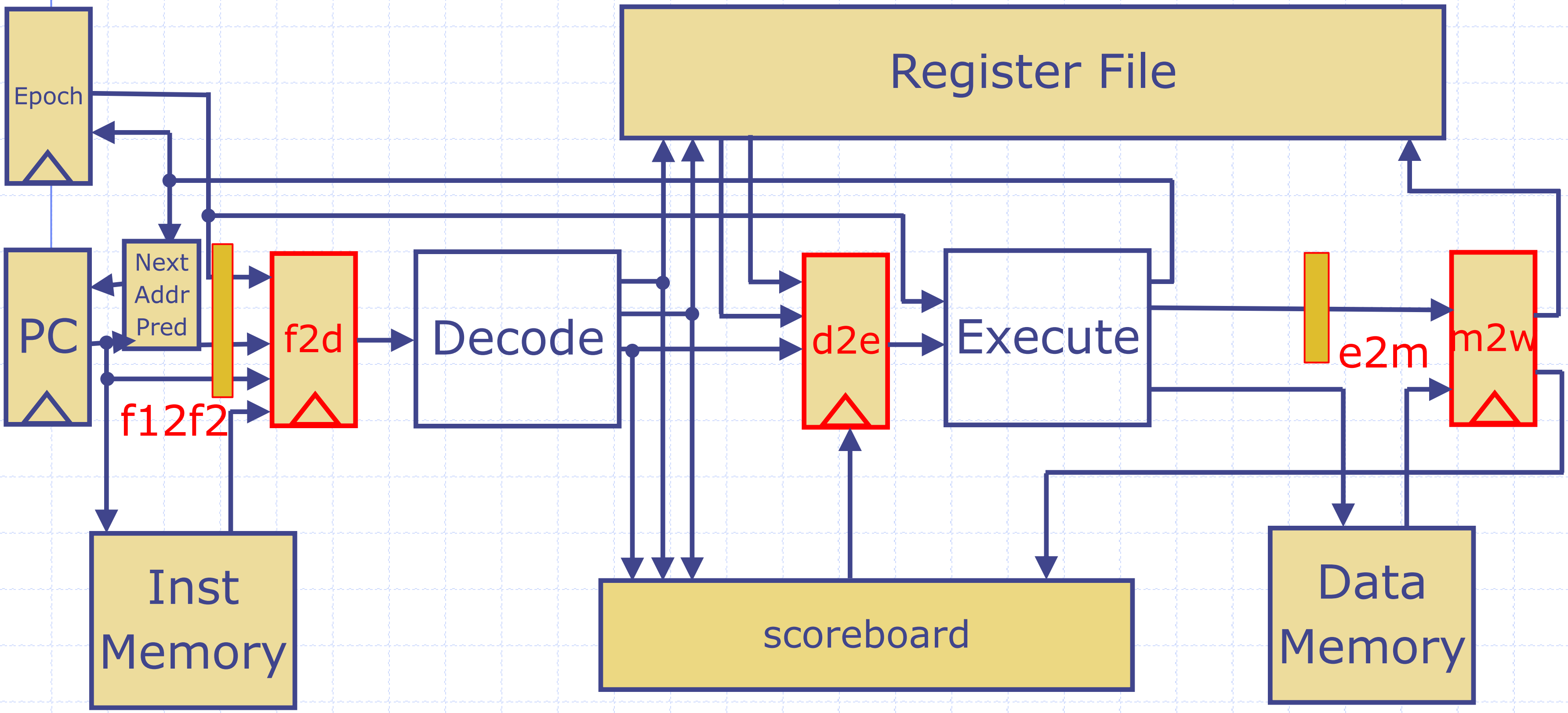
- Hit
  - Combinational read/write, i.e. 0-cycle response
  - Requires `req` and `resp` methods to be concurrently schedulable, which in turn requires

    `hitQ.enq` < {`hitQ.deq`, `hitQ.first`}

  i.e., `hitQ` should be a bypass Fifo
- Miss
  - No evacuation: memory load latency plus combinational read/write
  - Evacuation: memory store followed by memory load latency plus combinational read/write

*Adding an extra cycle here and there in the miss case should not have a big negative performance impact*

# Four-Stage Pipeline



▮ insert bypass FIFO's to deal with (0,n) cycle memory response