Memory Systems

Debiprasanna Sahoo
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

Content

Book

Computer Organization and Design: The Hardware/Software Interface-RISC-V Edition, 5th Edition, 2017

Chapter-5

David A. Patterson and John L. Henessey

Reference Books

Computer Organization and Design: The Hardware/Software Interface-MIPS Edition, 5th Edition, 2017

Chapter-5

Computer Architecture: A Quantitative Approach

Appendix-B

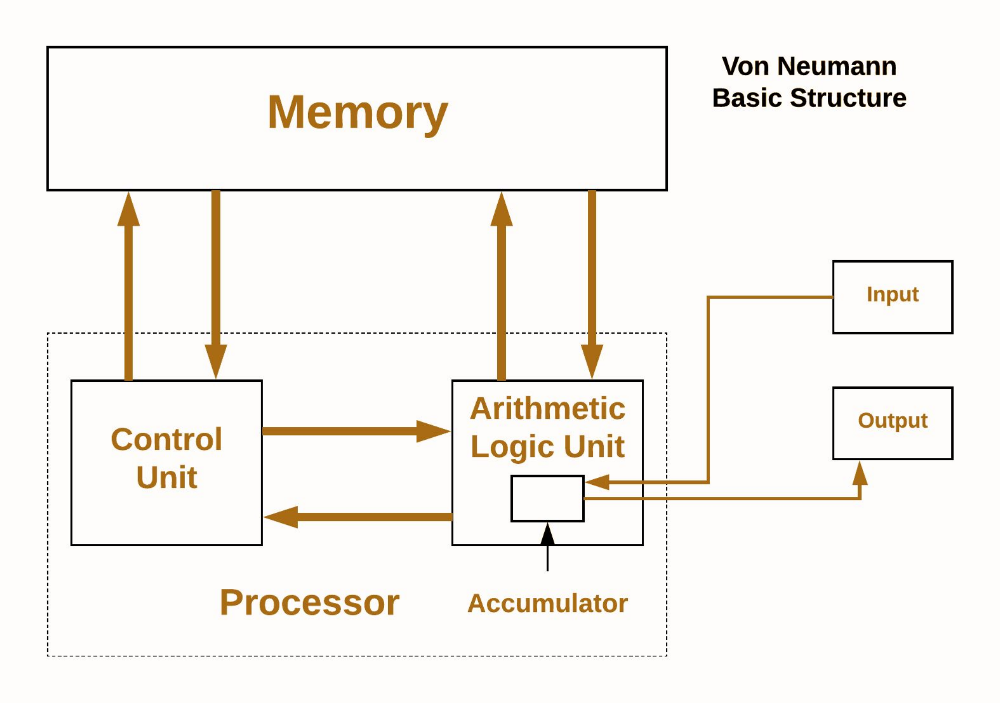
David A. Patterson and John L. Henessey

*Image from the book and manual unless specified

Von-Neumann Architecture (Recap)

Components of a Computer:

- Processor which does the computation with the help of Arithmetic and Logic Unit (ALU) and Control Unit (CU).
- Memory that stores data on which computation can happen.
- Input and Output Devices that supplies or uses data item.



Geeks for Geeks

Different Types of Memory Technologies

Property	SRAMs	DRAMs	Flash	Disk
Access Time	Fast	Medium	Slower than DRAM, faster than Disk	Slow
Power	High	Medium	Medium	High
Non-Volatile	No	No	Yes	Yes
Cost	Really high	Medium	Lower than DRAM costly than disk	Low
Density	Low	High	High	High
Typical Size (Capacity)	2-40 MB	128MB-256/512GB	128 GB to > 1 TB	512 GB to > 1 TB
Usage	Cache	Main Memory (Now Caches also)	Secondary Storage	Secondary Storage

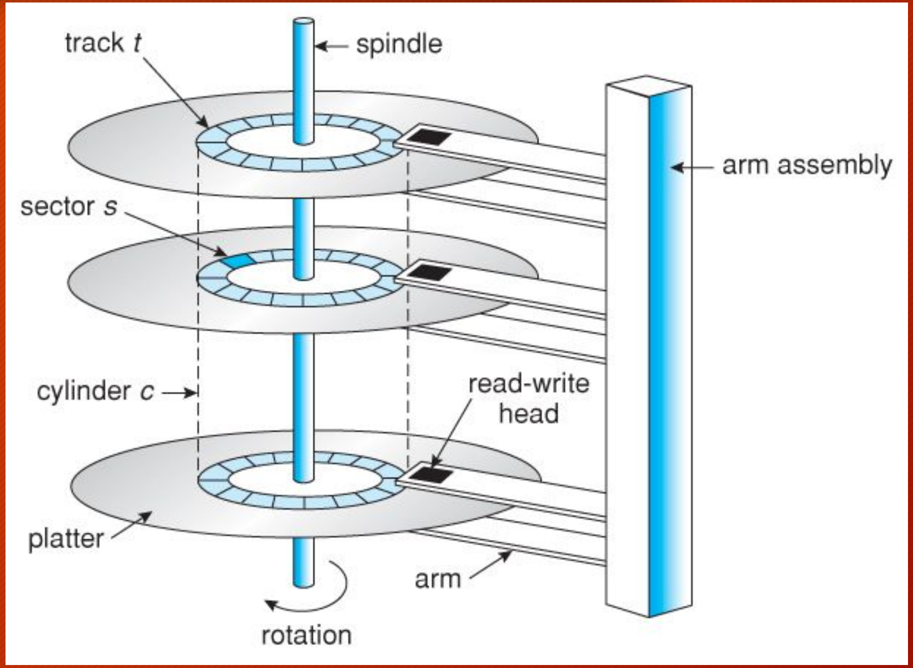
Disk Memory

Set of Platters form a disk

Track: One of thousands of concentric circles that make up the surface of a magnetic disk.

Sector: One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

Cylinder: All the tracks under the heads at a given point on all surfaces.



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/10_MassStorage.html

Latency of Disk Drives

Seek: The process of positioning a read/write head over the proper track on a disk.

Rotational Delay/Latency: The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

Transfer Latency: Time to transfer the data from device to the cores (actually to main memory).

Typical seek time = 3-4 ms

Disk Rotation Speed = v RPM

Rotational Latency = 1/2v RPM = 30/v seconds

For 7200 RPM,

Rotational Latency = 30/7200 sec

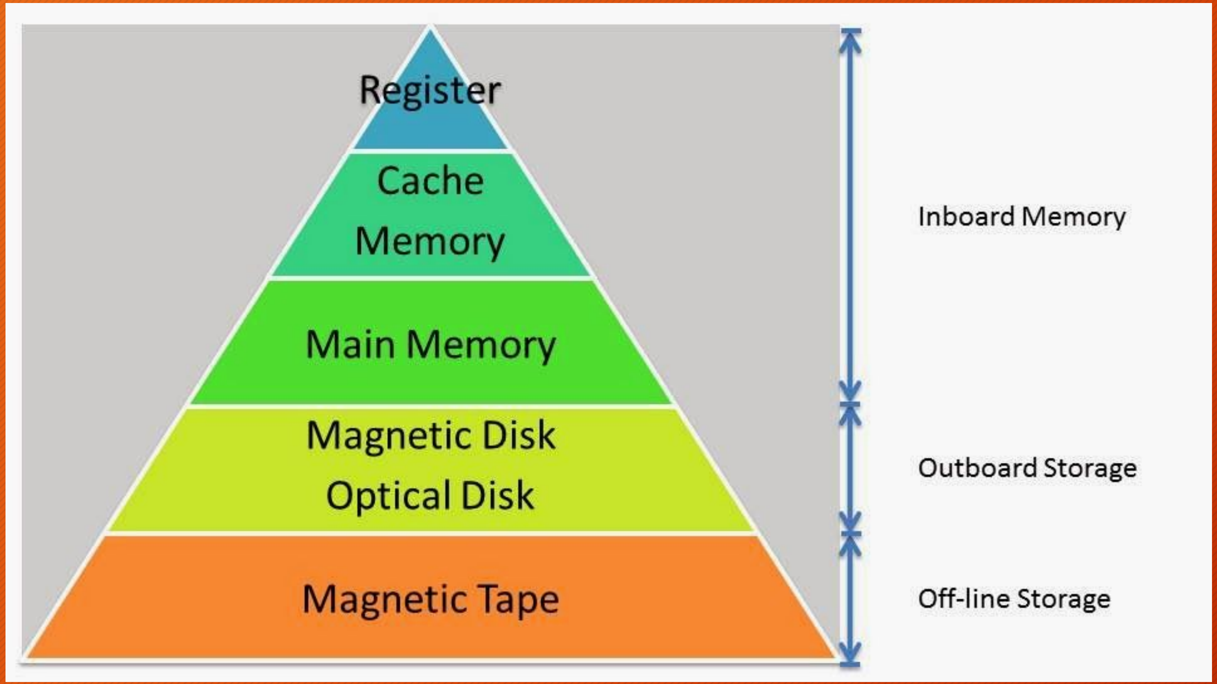
= 1/240 sec = 4.17 ms

Typical transfer rates = Max 90 MBps

Disks are connected to a hub called South Bridge controlled by interfaces like ATA, SATA, USB, SCSI and is advocated by DMA controller.

Memory Hierarchy

Memory Hierarchy: A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

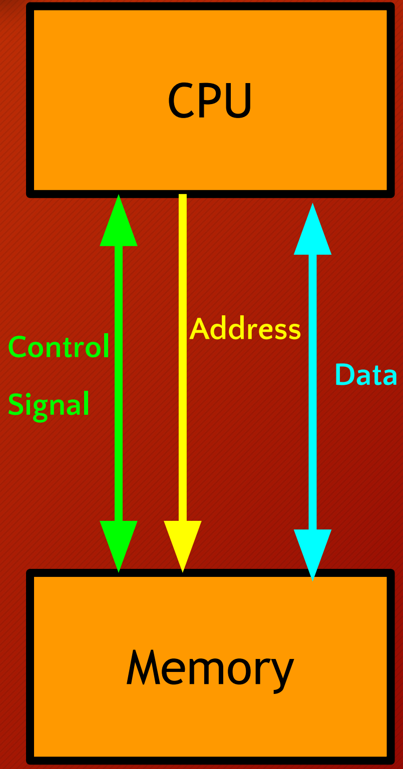


https://abhaycopi.blogspot.com/2014/04/memory-hierarchy.html

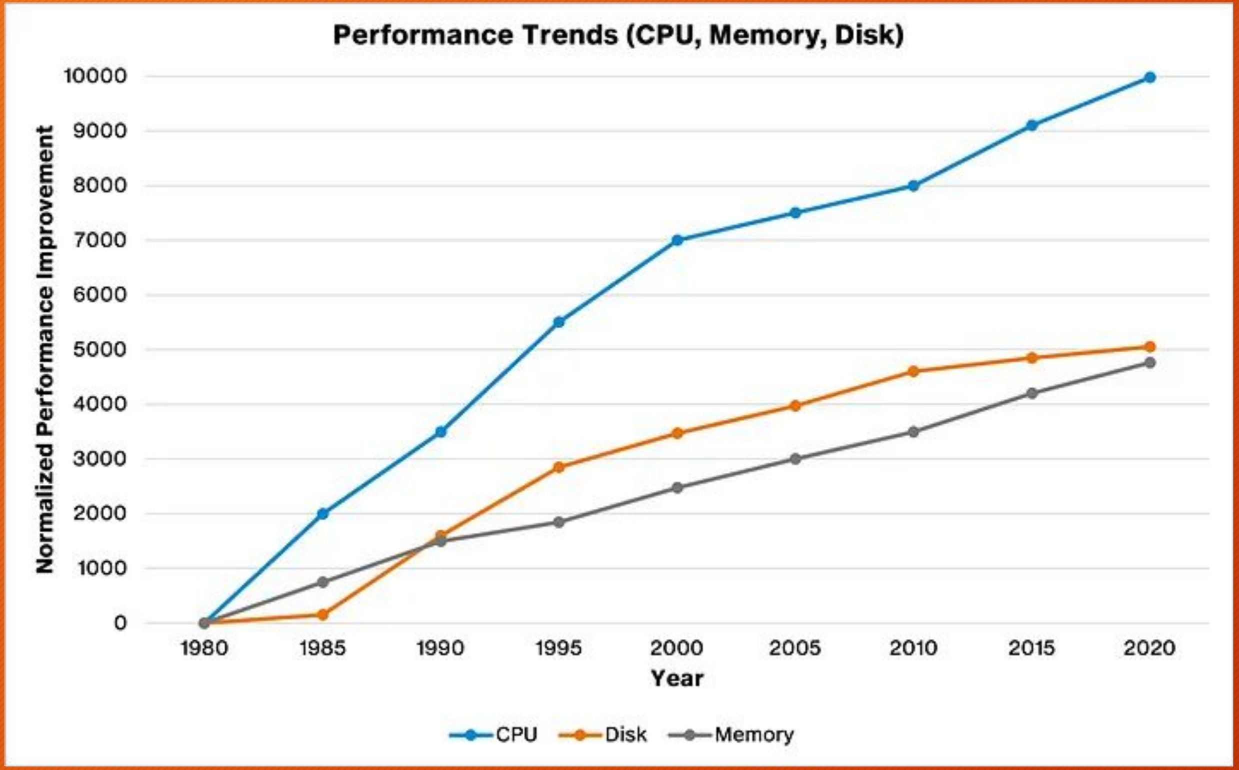
Main Memory

Steps of Memory Access

- ❑ CPU generates load/store operations for addresses to read/write data from the memory operation stage on a **data bus**.
- ❑ CPU generates an address for instruction in the fetch stage on a data bus.
- ❑ The request are transferred over on the **control bus** and the corresponding address over the **address bus**.
- ❑ For loads and instruction access, the data bus is empty while sending the request. However, once the location is read, the bus gets the valid data.
- ❑ The main memory performs a read operation corresponding to a load and a write operation corresponding to a store.



Scaling of memory technology

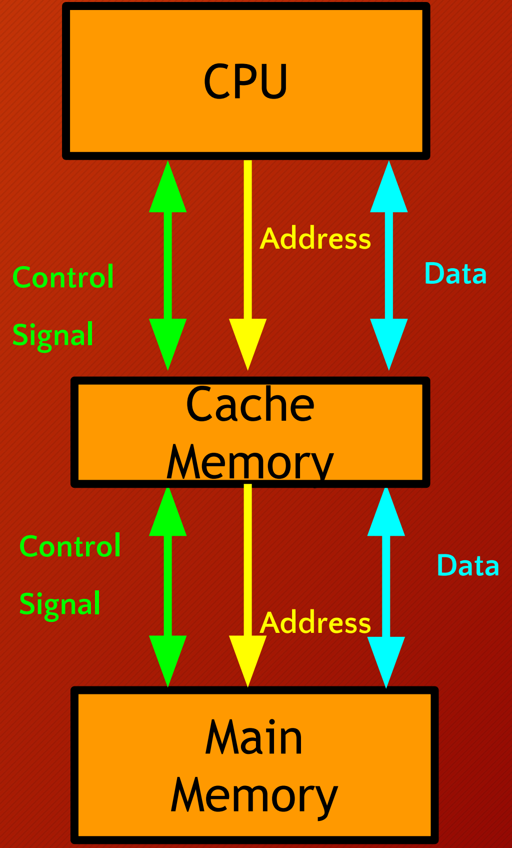


https://verizon5gedgeblog.medium.com/why-cpu-utilization-doesnt-tell-the-whole-story-89239b07a7f6

- ❑ DRAM technology used to design main memory is not scaling when compared to the cores
- ❑ Gap in speed between the core and the main memory is continuously growing
- ❑ **Solution:** Search for new technology to design main memory?

Cache Memory ($ Memory)

- ❑ **Cache memory** is a small and fast memory used to store more recently accessed data; it exploits **data locality**.
- ❑ Its is typically designed using **SRAMs** and it is used between the CPU and the main memory.
- ❑ Addresses in a cache is a subset of the addresses in the main memory. Hence, some requested addresses can be found in the cache and some may not.



Principle of Locality

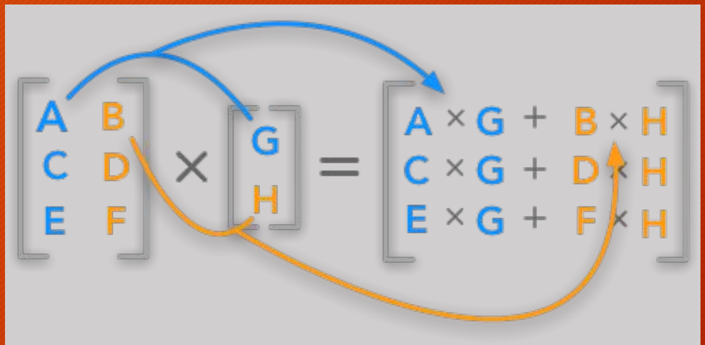
- ❑ **Spatial Locality:** The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.
- ❑ **Temporal Locality:** The locality principle stating that if a data location is referenced then it will tend to be referenced again soon.

```
for(i = 0; i < 100; i++)      {

                C[i] = A[i] + B[i];

}
```

What about Matrix Multiplication?

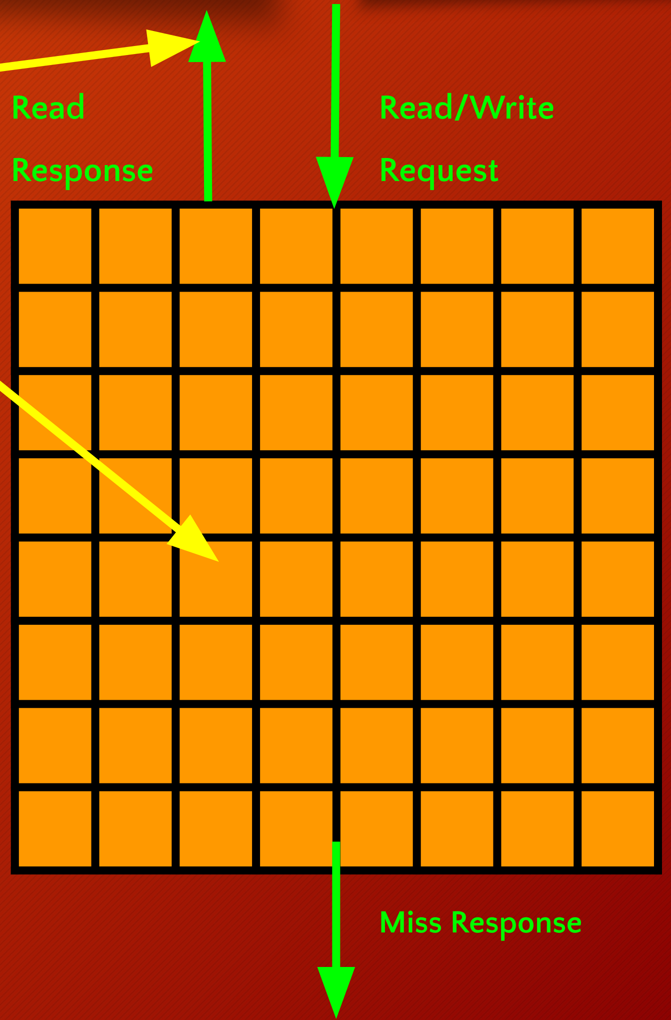


What about Pointer Chasing?

What about Graphs, Graphics, and many more?

Some Terminologies

- ❑ **Word:** Unit of transfer of data between the processor and memory hierarchy. Typically, 4B
- ❑ **Cache Block/Line:** The minimum unit of information that can be either present or not present in a cache. Typically, 64 B.
- ❑ **Cache Hit:** When request is serviced by the cache because the line is already present
- ❑ **Cache Miss:** Request is serviced by main memory because the line is absent in the cache
- ❑ **Hit Rate:** Fraction of memory requests serviced by the cache = No. of cache hits / Total No. of Accesses
- ❑ **Miss Rate:** 1 - Hit Rate
- ❑ **Hit Time:** Time required (typically in cycles) to access the cache including the time needed to determine whether the access is a hit or a miss.
- ❑ **Miss Penalty:** The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.
- ❑ **Parallelism:** Number of requests being processed in parallel/pipelined inside the cache



Performance of a Memory System

Latency: The amount of time in cycles the memory hierarchy takes to service a single request. Latency = End Time - Start Time = ET - ST

Average memory access time (AMAT) = Hit Time + Miss Rate * Miss Penalty

Bandwidth: How much of data per cycle is delivered to the CPU

Bandwidth = Number of Request Serviced * Cache Line Size / Time to service all the requests = $N * S / ET_f - ST_i$

Goal: Reduce Latency and Increase Bandwidth

Four Memory Hierarchy Questions

Block Identification: How to figure out that block is present in the cache?

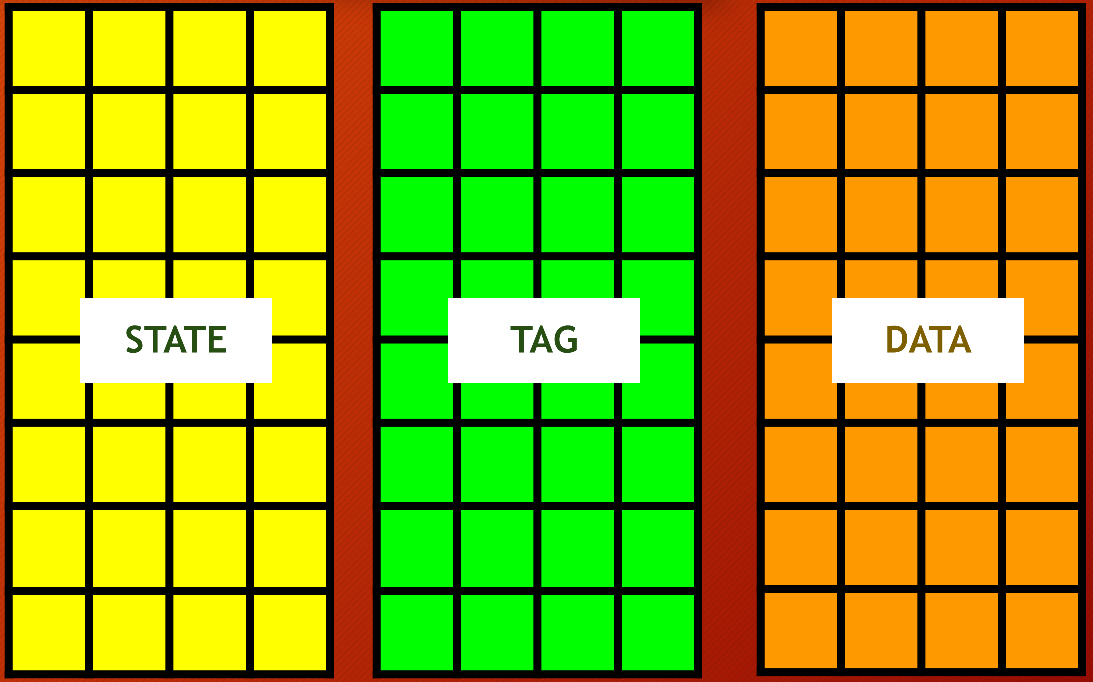
Block Placement: Where can a block be placed in the cache?

Block Replacement: Which block to replace/evict/victimize on a miss?

Write Strategy: What happens on a write?

Block Identification

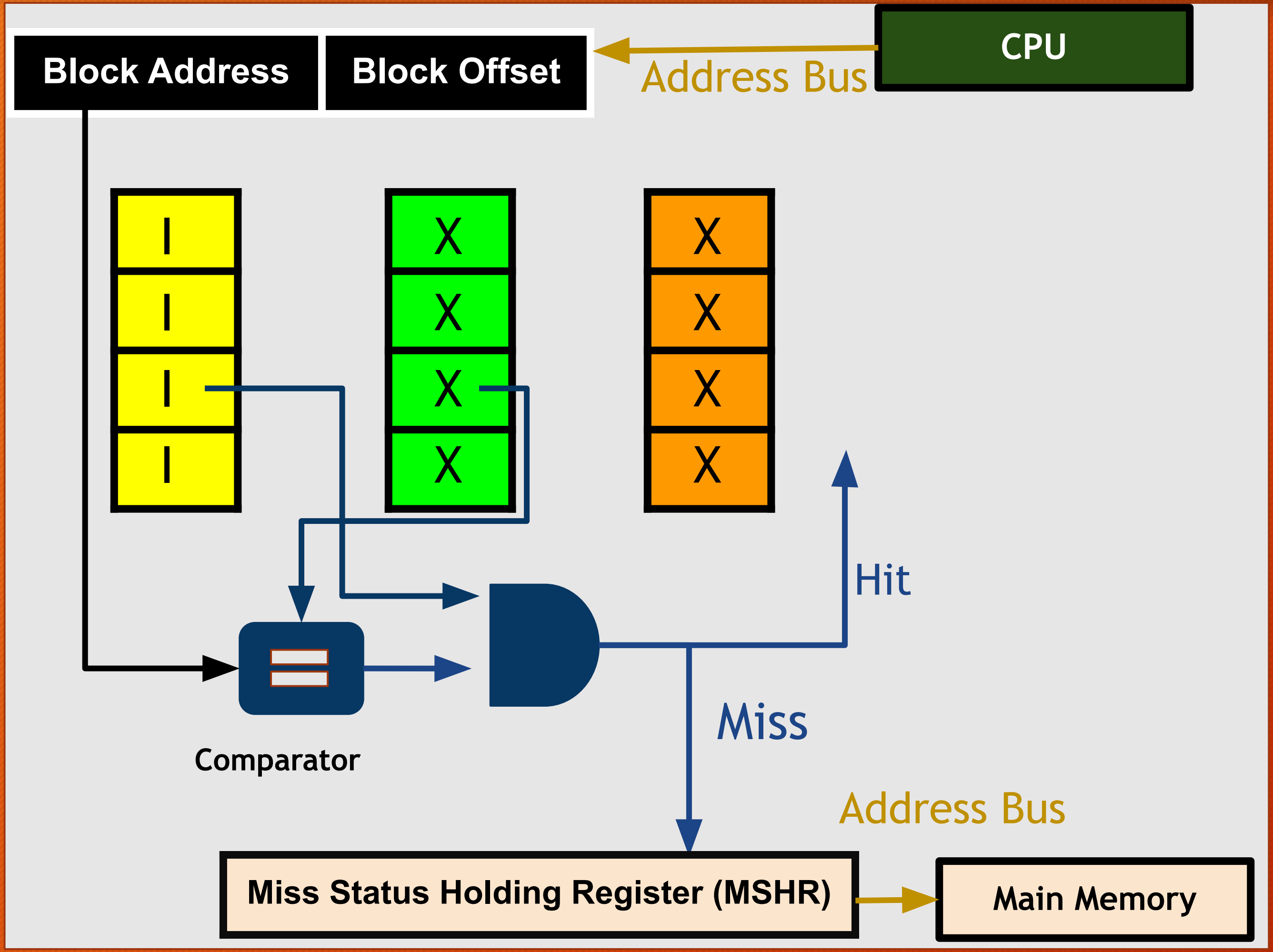
- A block can be present inside the cache (the state shall be Valid or V) or not present inside the cache (the state shall be Invalid or I).
- **Valid bit:** Determines if a given address is present or not.
- CPU can make read or write requests
- On hit, CPU is given a response (CPUResp) for reads (CPURd).
- On miss, memory gets a requests – MemRd for CPURd and MemWr for CPUWr.
- Assume: On write hit data is updated and also sent to memory.
- What if another requests for same block arrives in the cache before response from memory for the previous one come back?



- **State Information:** Valid bit, Miss Pending bit, etc.

- Even an address is present in the cache, how do we know that its is the address that we need.
- Store part of the address inside the cache alongside data called **Tags**.
- Tag part of the incoming address is matched against the **stored tags**.

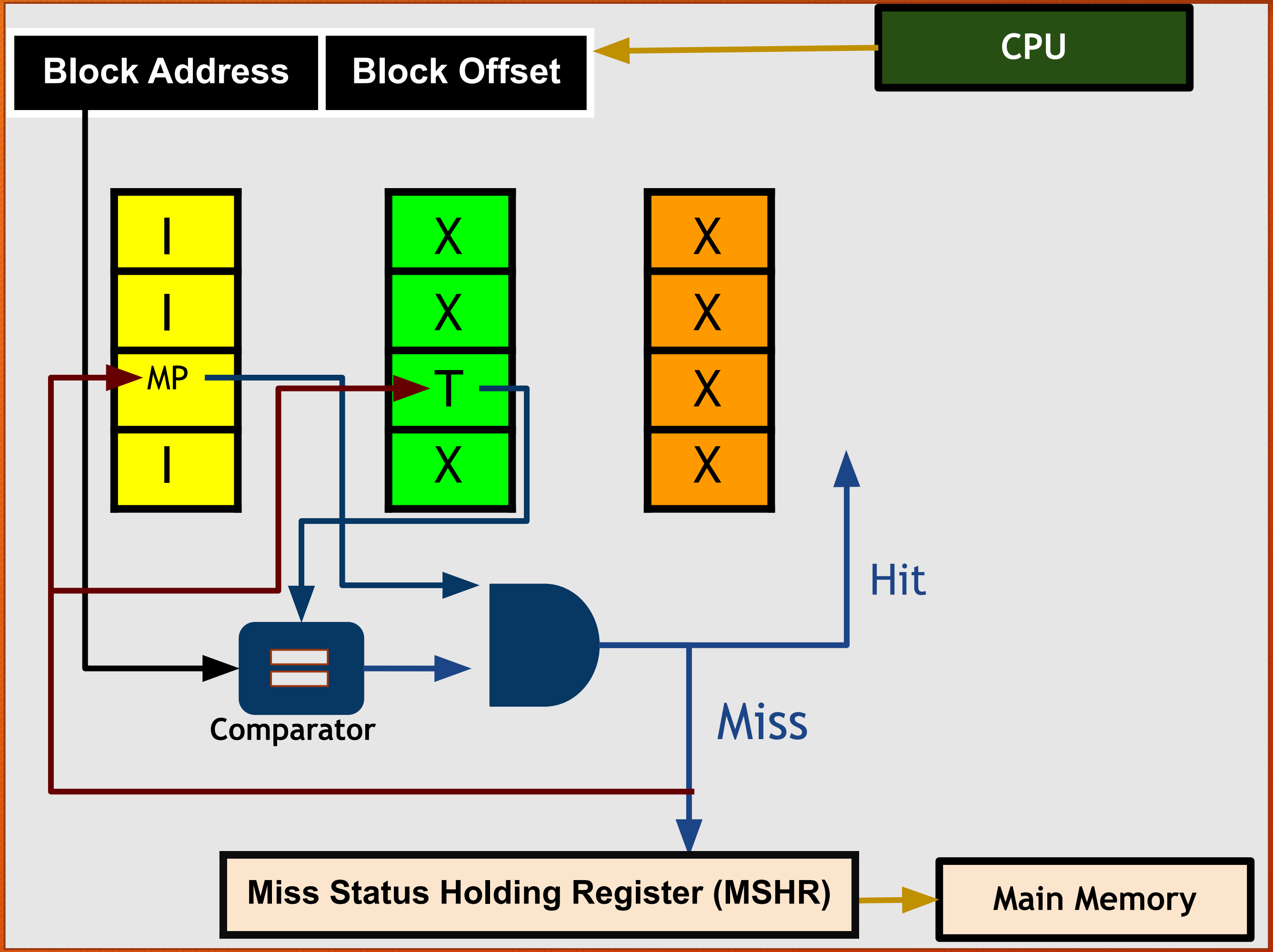
Basic Cache Memory: Design



MISS SCENARIO

- ❑ CPU sends a read/write request corresponding to load or store operation.
- ❑ The address associated with the address is divided into block address and block offset.
- ❑ **Block Address (assume tag)** identifies a block inside the cache and hence, refers to an entry in each tag, data, and state arrays.
- ❑ **Block Offset** refers to the word which has been accessed inside the block.
- ❑ The **comparator** checks if the stored tag matches incoming tag.
- ❑ The **and gate** ensures to provide a hit/miss after checking the state and tag comparison.
- ❑ In this case, all are invalid blocks, so we will see miss for any access.

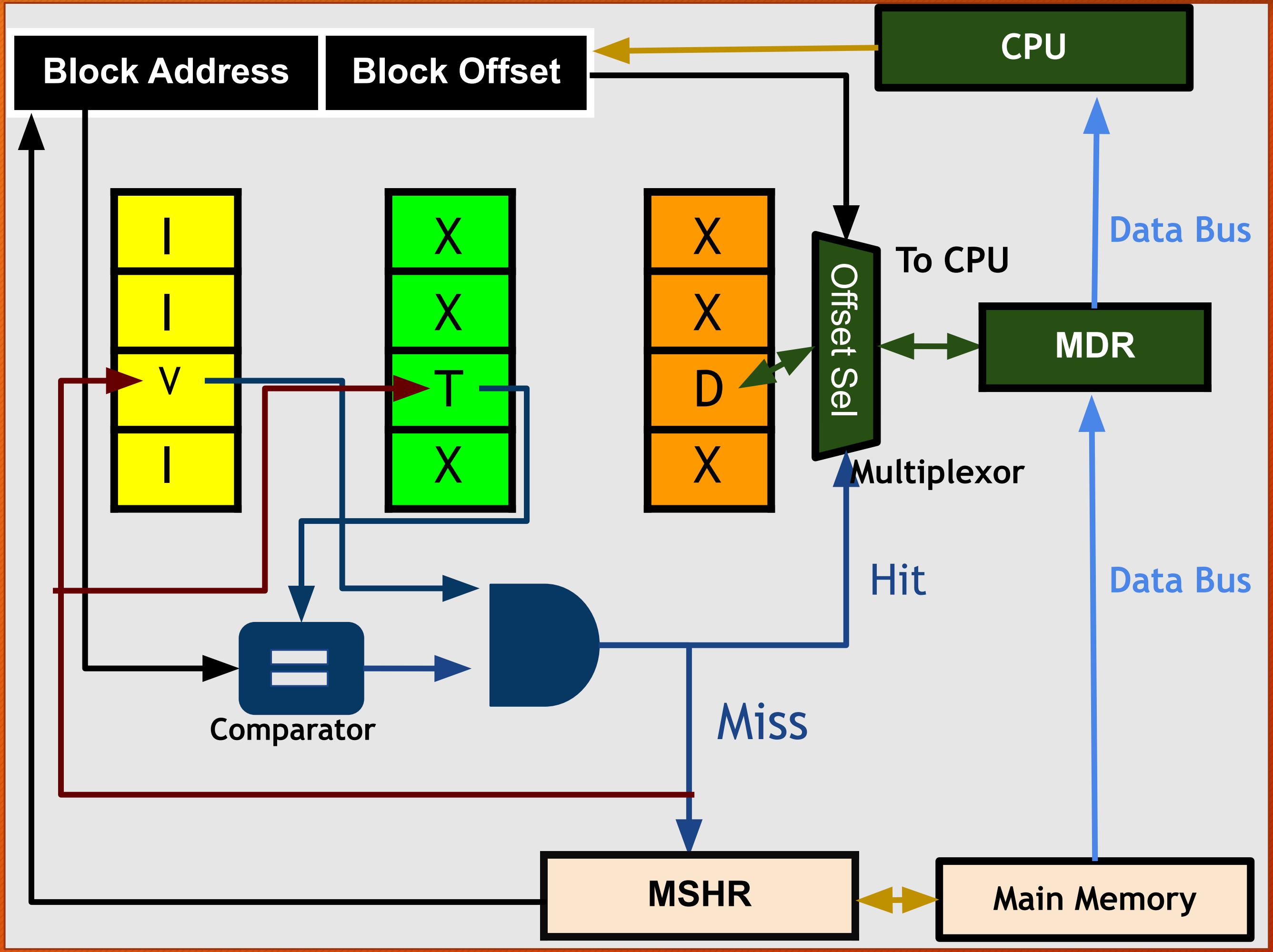
Basic Cache Memory: Design (Cont..)



MISS SCENARIO

- ❑ We determine a place where the block can be placed when the response comes back.
- ❑ The state is marked as pending data or miss pending (MP) and the tag is written.
- ❑ Miss Status Holding Registers (MSHR): MSHR holds the address and status of the memory request upon miss in the cache.
- ❑ It serves the following purposes:
  - Stops another requests to same address to follow into the next level.
  - Quick lookup of the cache when the response comes back; we already have the way -> no tag comparison necessary.
  - Helps in verifying if the memory system is working correctly by implementing liveness property

Basic Cache Memory: Design (Cont..)



FILL SCENARIO

- ❑ If any address, A was accessed which contains D in main memory and D is transferred over Data Bus.
- ❑ Upon servicing a miss for that address, we save the tag of A (T) and data D (placed in Memory Data Register) in the tag and data arrays, respectively.
- ❑ We also mark the block as valid inside the cache. We also forward the data back to the CPU.

HIT SCENARIO

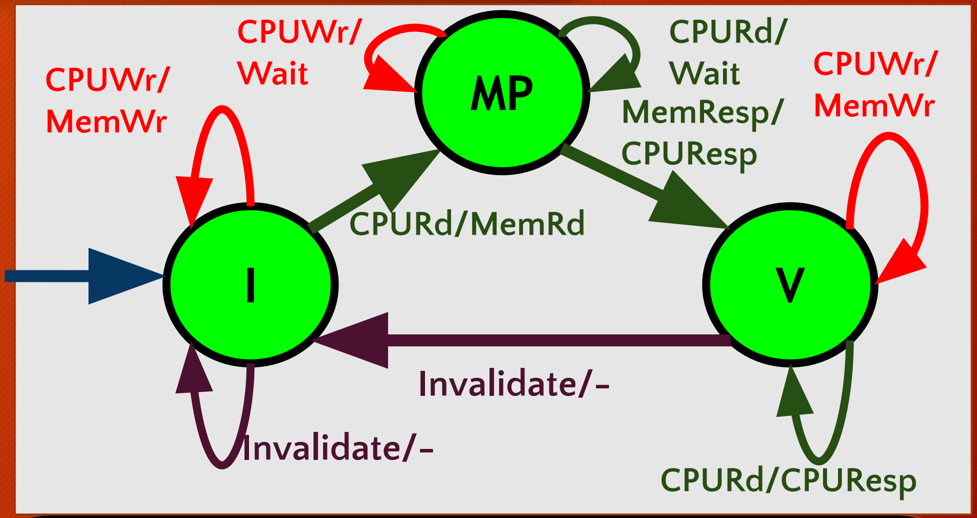
- ❑ If any address, A was accessed which contains D in the cache.
- ❑ The comparator and AND gate specifies hit.
- ❑ The **multiplexor** reads the requested word and delivers it to the CPU.

Basic Cache Memory: Implementation

- **Inputs:**
  - CPURd and CPUWr for load and store instructions.
  - They can be invalidates from replacement/bus.
  - Memory response (MemResp) for read misses.
- **Outputs:**
  - MemRd and MemWr for memory reads and writes upon read miss and writes.
  - CPUResp for reads

State Table

CS	Input	NS	Output
I	CPURd	MP	MemRd
I	CPUWr	I	MemWr
I	Invalidate	I	-
MP	CPURd/ CPUWr	MP	Action is to wait
MP	MemResp	V	CPUResp
V	CPURd	V	CPUResp
V	CPUWr	V	MemWr
V	Invalidate	I	-

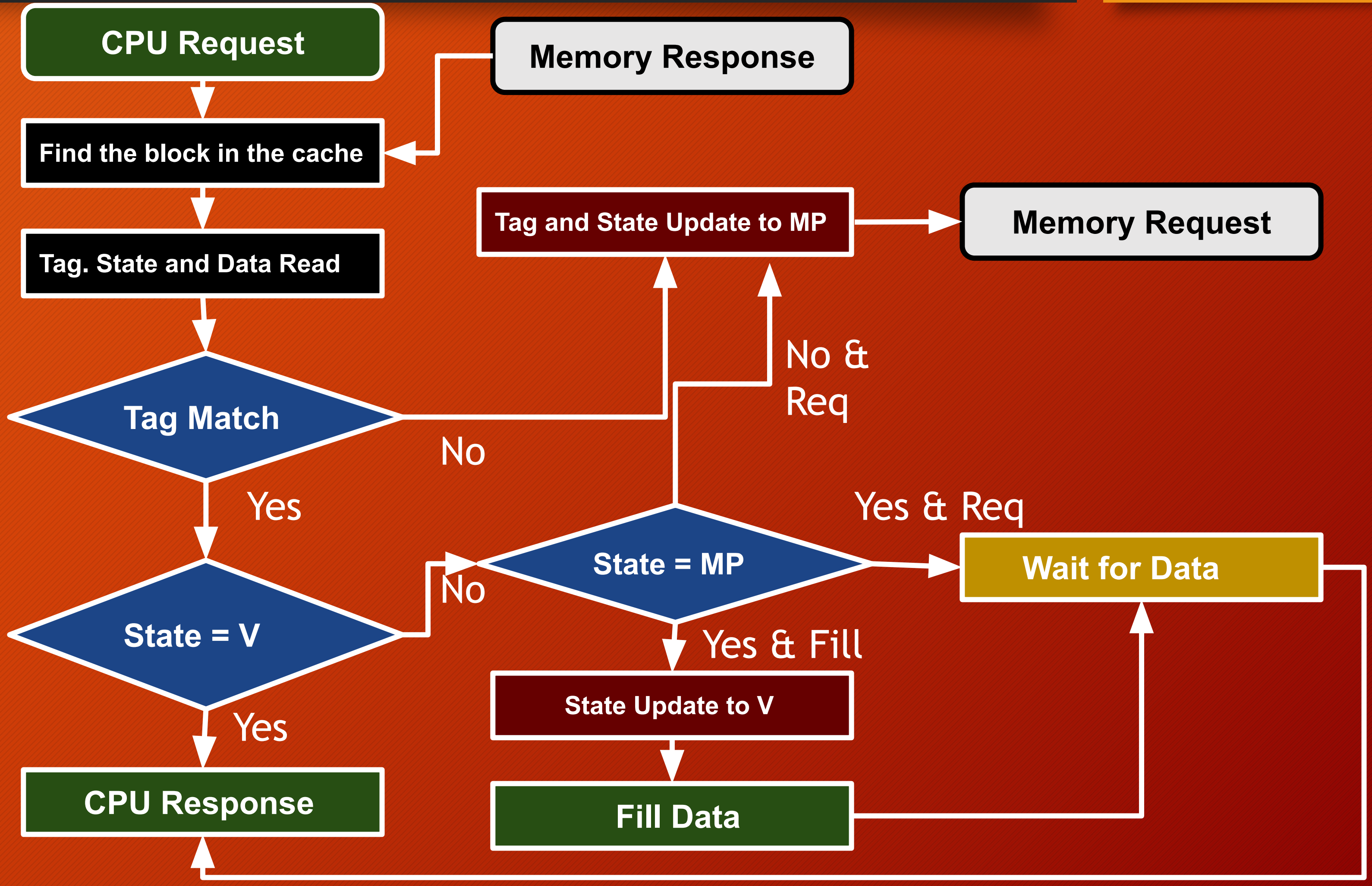


- **States:** Status of all the words part of a block inside the cache.
  - **Invalid (I):** Block is not present inside the cache.
  - **Valid (V):** Block is present inside the cache and all the words in the block have the same content as main memory.
  - **MissPending (MP):** Tags are present but data has not been received from the memory.

Basic Cache Memory: Implementation

```
For each B in Block[]
    If B.Tag = CPUReq.Tag & B.State = V
        Then CPUResp = B.Data
    If B.Tag = CPUReq.Tag & B.State = MP
    Then CPUReq.wait
B = evict(B)
B.State = MP
B.Tag = CPUReq.Tag
MemReq = CPUReq
```

```
For each B in Block[]
    If B.Tag = MemResp.Tag & B.State = MP
    Then B.State = V
        B.Data = MemResp.Data
        CPUResp.Data = B.Data
        Notify All Waiting To Restart
```



Block Placement

- □ **Index:** The location at which a block is stored in the cache. A cache with n-blocks required log n bits to index.
- □ **Block Offset:** The part of the address which determines exactly which data item/word is accessed.
- □ Organization of the Cache:
  - Direct-Mapped Cache
  - Associative Cache
  - Set Associative Cache

Block Address		Block Offset
Tag	Index	

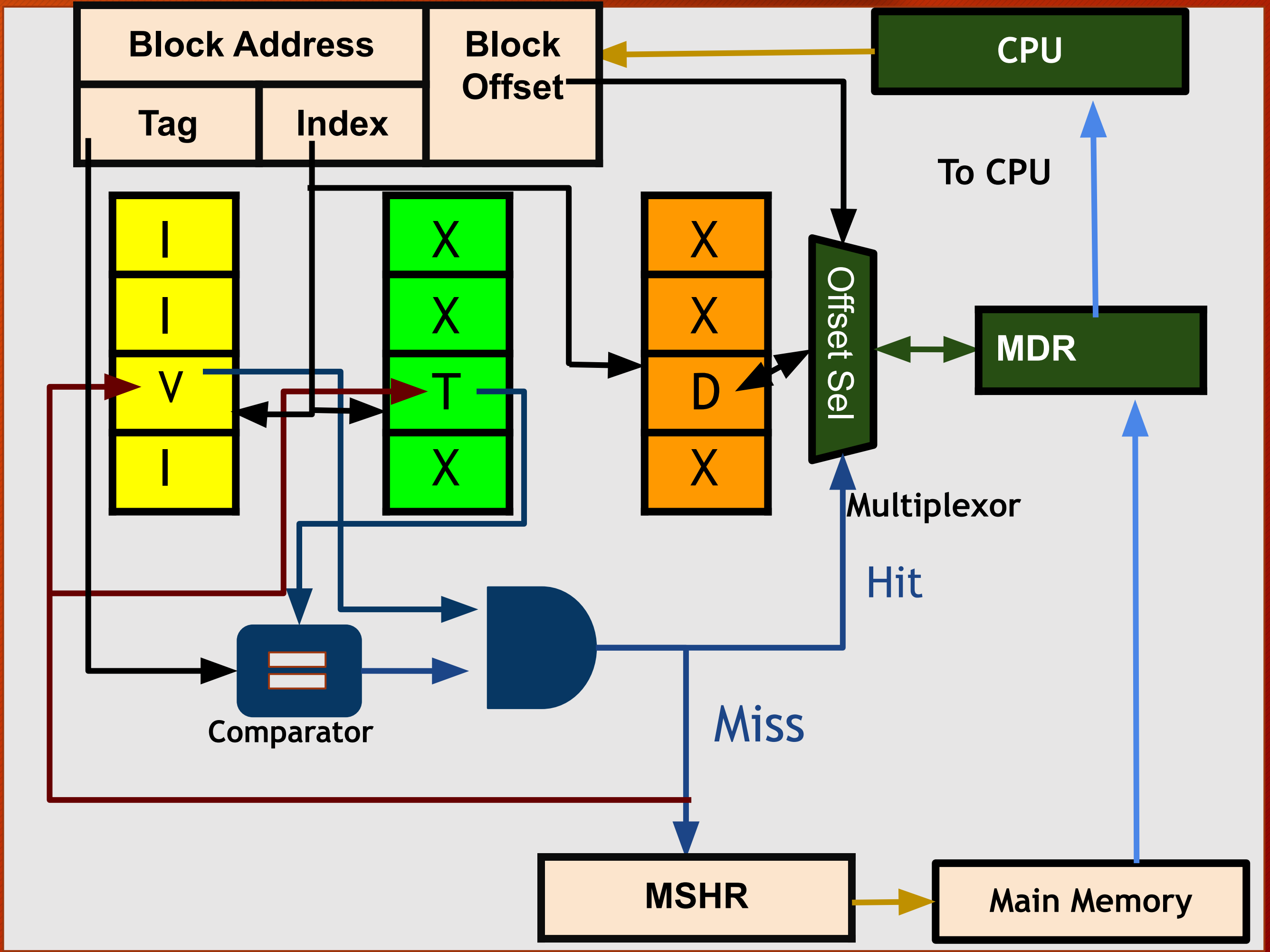
Direct-mapped Cache: Design

- ❑ Every physical address maps to a specific location in the cache
- ❑ **Indexing Function:** The operation used to determine the index of a cache for a given address. Typically, it is

Index = (physical block address) mod (Total #Blocks)

- ❑ **Benefit:** Simple in design, Lower hit time
- ❑ **Demerits:** More conflicts at the same index

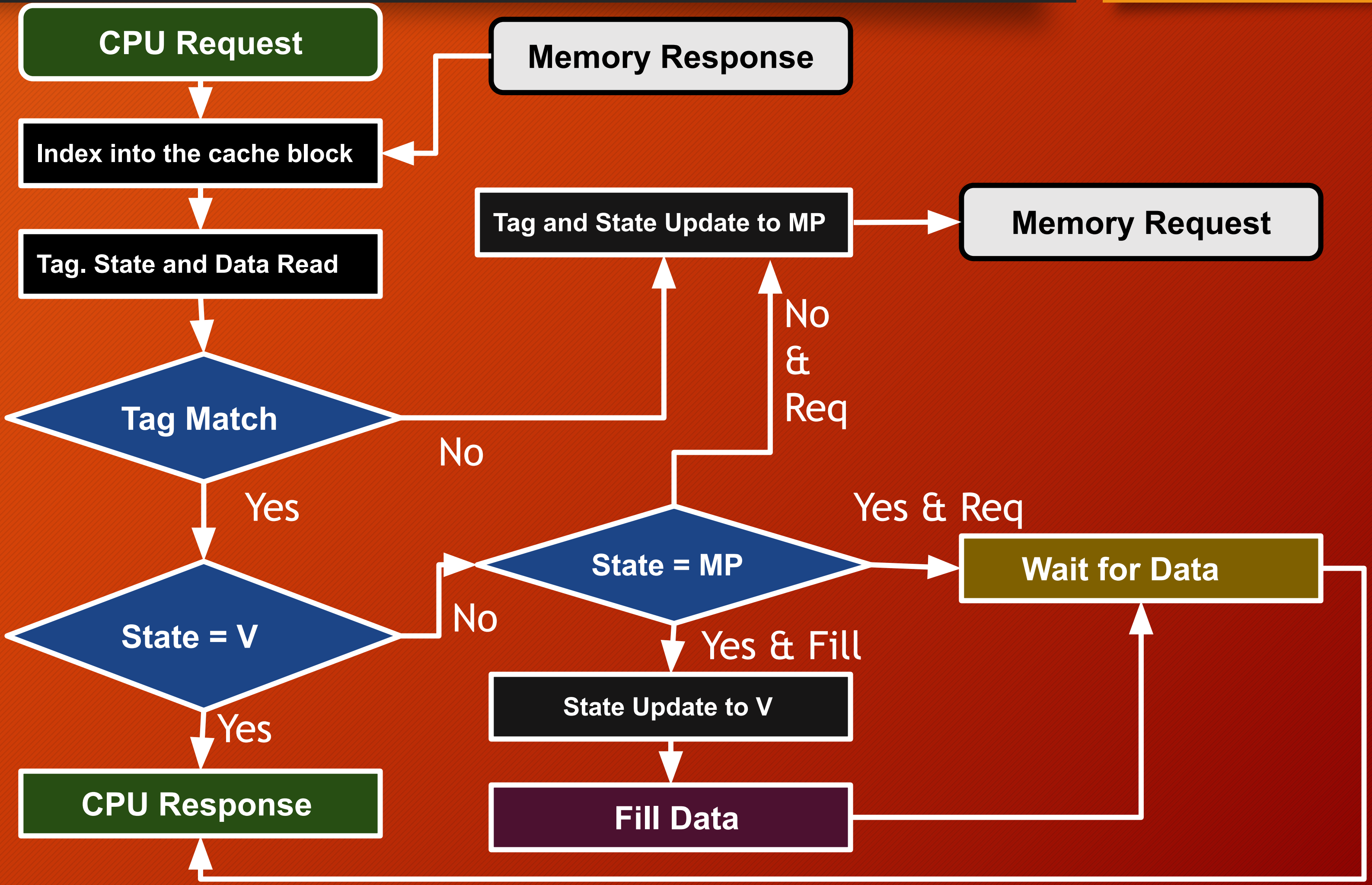
- ❑ Block Size = B, Block Bits = log B, Block Offset = ?
- ❑ Cache Size = C
- ❑ #Blocks = N = C/B. Index Bits = log (C/B), Index = ?
- ❑ Tag Bits = A - log B - log N, Tag = ?
- ❑ State Bits = S
- ❑ Tag Array Size = N * (S + A - log B - log N)



Direct-mapped Cache: Implementation

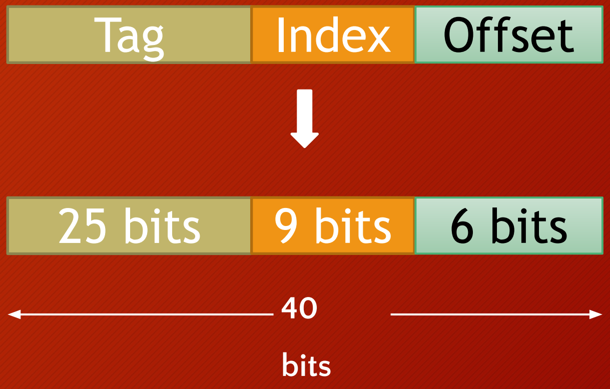
- ***B = Block[CPUReq.Index]***
- **If B.Tag = CPUReq.Tag & B.State = V**
- **Then CPUResp = B.Data**
- **If B.Tag = CPUReq.Tag & B.State = MP**
- **Then CPUReq.wait**
- **B = evict(B)**
- **B.State = MP**
- **B.Tag = CPUReq.Tag**
- **MemReq = CPUReq**

- ***B = Block[CPUReq.Index]***
- **If B.Tag = MemResp.Tag & B.State = MP**
- **Then B.State = V**
- **B.Data = MemResp.Data**
- **CPUResp.Data = B.Data**
- **Notify All Waiting To Restart**



Direct-mapped Cache: Example

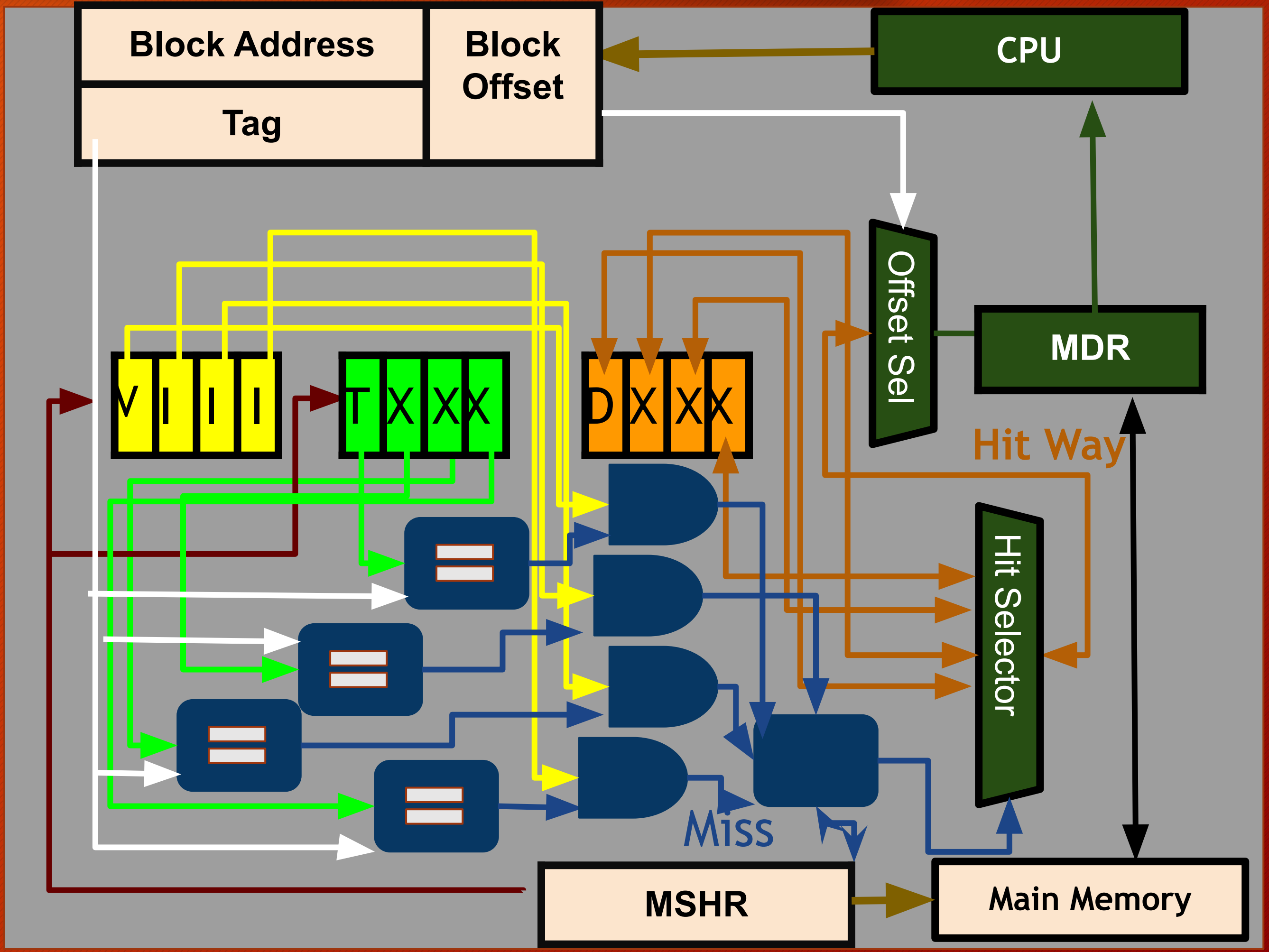
- **Problem:** A direct mapped cache of size 32KB can be addressed using a 40 bit address. Block size of the cache is 64B. Find the total number of bits required for indexing, block offset and tag. Also find the tag array size.
  - Block Size = 64 B => Block Offset = 6 Bits
  - Cache Size = 32 KB => No. of Blocks = 32KB/64B = 512
  - Indexing bits = log 512 = 9 Bits
  - Tag bits = 40 – 9 – 6 = 25
  - Tag array size = ((25 + 1) * 512) bits = 26 * 64 B



Associative Cache: Design

- ❑ A physical address can map to any location in the cache
- ❑ **Index:** No indexing necessary
- ❑ **Parallel Search:** All tags are searched in parallel. Requires comparator for all the block.
- ❑ **Replacement Policy:** A block allocation policy is necessary
- ❑ **Benefit:** Can place blocks anywhere and hence, minimum conflict. (High hit-rate)
- ❑ **Demerits:** Large multiplexor is required that hit time, comparators consume a lot of power.

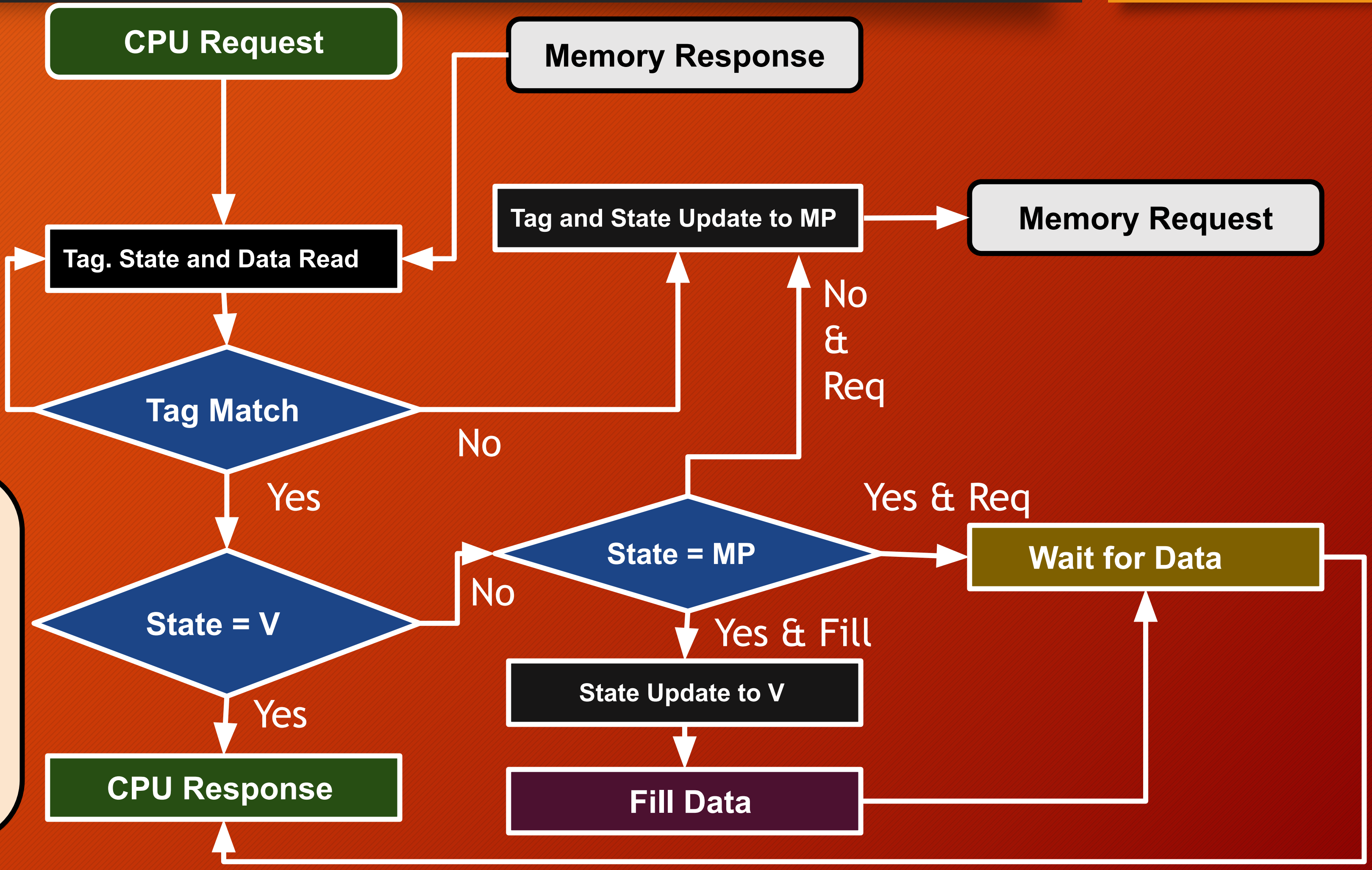
- ❑ Block Size = B, Block Bits = log B, Block Offset = ?
- ❑ Cache Size = C
- ❑ #Blocks = N = C/B. Index Bits = 0, Index = 0
- ❑ Tag Bits = A - log B, Tag = ?
- ❑ State Bits = S
- ❑ Tag Array Size = N * (S + A - log B)



Associative Cache: Implementation

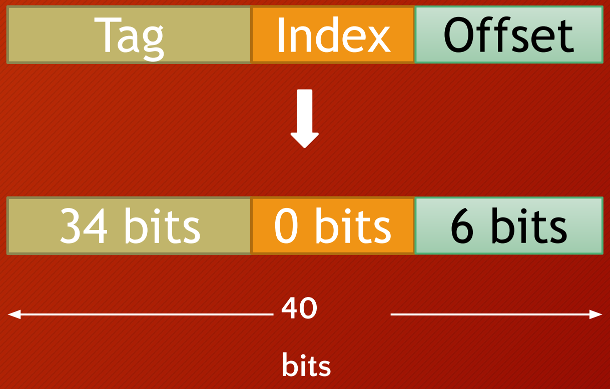
```
•for each way B in the Cache
•         If B.Tag = CPUReq.Tag & B.State = V
•         Then CPUResp = B.Data// Return
•          If B.Tag = CPUReq.Tag & B.State = MP
•          Then CPUReq.wait // Return
•B = evict(Cache)
•B.State = MP
•B.Tag = CPUReq.Tag
•MemReq = CPUReq
```

```
•for each way B in the Cache
•        If B.Tag = MemResp.Tag & B.State = MP
•        Then B.State = V
•             B.Data = MemResp.Data
•             CPUResp.Data = B.Data
•             Notify All Waiting To Restart
```



Associative Cache

- ❑ **Problem:** An associative cache of size 32KB can be addressed using a 40 bit address. Block size of the cache is 64B. Find the total number of bits required for indexing, block offset and tag. Also find the tag array size.
  - ❑ Block Size = 64 B => Block Offset = 6 Bits
  - ❑ Cache Size = 32 KB => No. of Blocks = 32KB/64B = 512
  - ❑ Indexing bits = 0
  - ❑ Tag bits = 40 – 6 = 34
  - ❑ Tag array size = (34 + 1) * 512 bits = 35 * 64 B



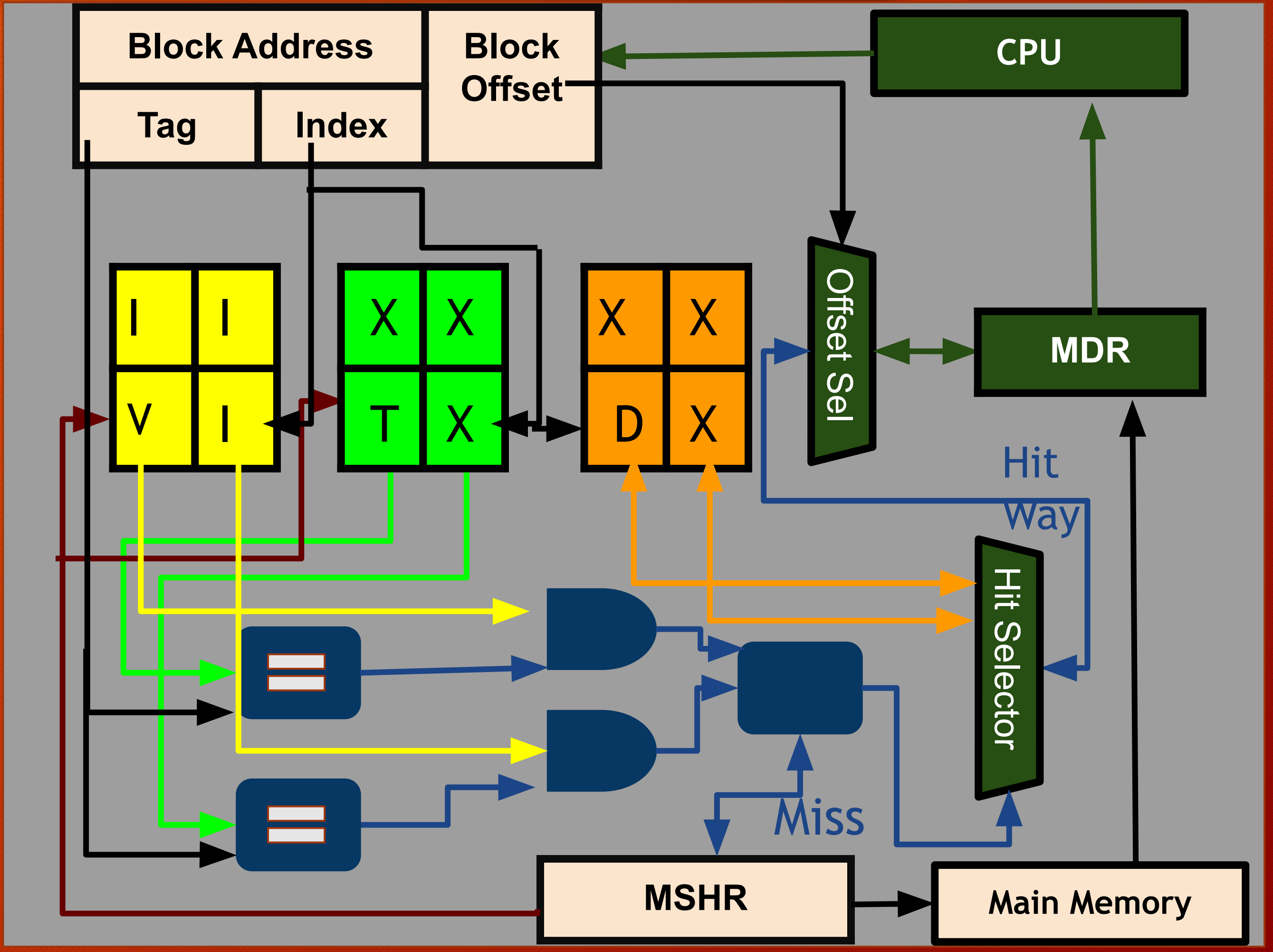
Tag	Index	Offset
34 bits	0 bits	6 bits

40 bits

Set Associative Cache: Design

- ❑ A physical address can map to any subset of location in the cache
- ❑ **Set:** Numbers of Indices
- ❑ **Way:** W-Blocks in a set are searched in parallel. Requires comparator for all the block.
- ❑ **Replacement Policy:** A block allocation policy works inside a set
- ❑ **Benefit:**
  - Can place blocks anywhere inside a set and hence, relatively low conflict. (High hit-rate).
  - Multiplexor size decreases and hence, the hit time.
  - Less comparators and hence, consume a less power.

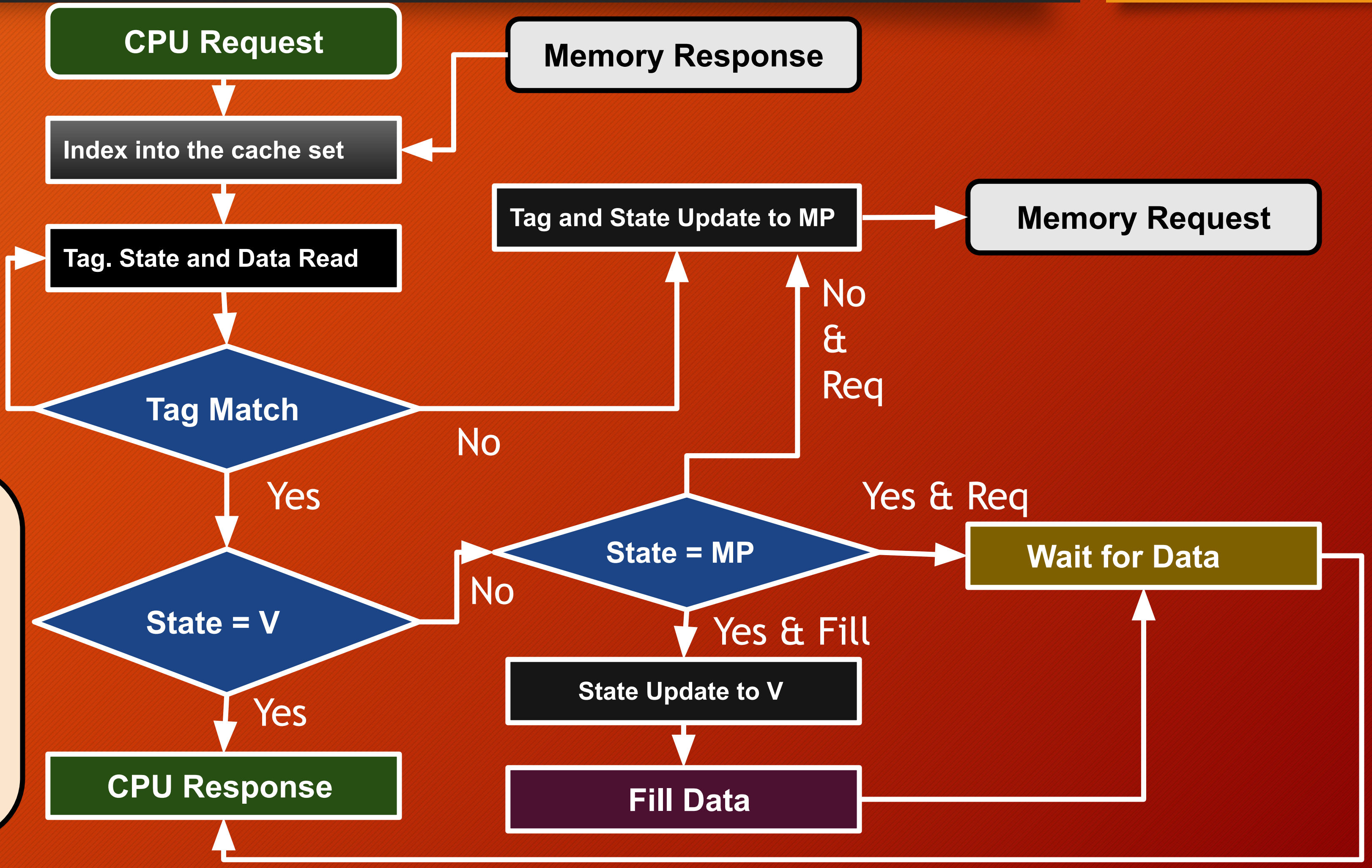
- ❑ Block Size = B, Block Bits = log B, Block Offset = ?
- ❑ Cache Size = C, Associativity = #Way = W
- ❑ #Blocks = N = C/B, #Sets = N/W
- ❑ Index Bits = log (N/W),Index = ?
- ❑ Tag Bits = A - log (N/W), Tag = ?
- ❑ State Bits = S
- ❑ Tag Array Size = N * (S + A - log N/B)



Set Associative Cache: Implementation

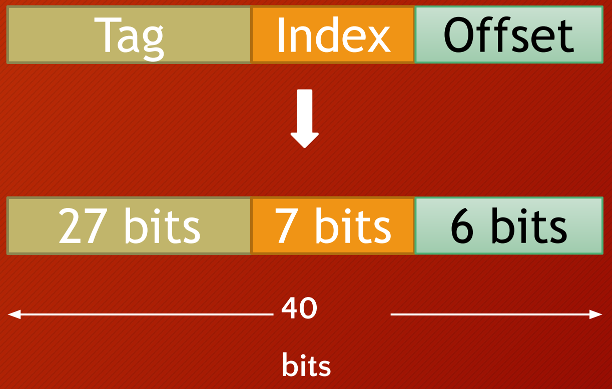
```
•S = Block[CPUReq.Index]
•for each way B in the Set S
•       If B.Tag = CPUReq.Tag & B.State = V
•       Then CPUResp = B.Data// Return
•        If B.Tag = CPUReq.Tag & B.State = MP
•        Then CPUReq.wait // Return
•B = evict(S)
•B.State = MP
•B.Tag = CPUReq.Tag
•MemReq = CPUReq
```

```
•S =  Block[CPUReq.Index]
•for each way B in the Set S
•       If B.Tag = MemResp.Tag & B.State = MP
•       Then B.State = V
•           B.Data = MemResp.Data
•           CPUResp.Data = B.Data
•           Notify All Waiting To Restart
```



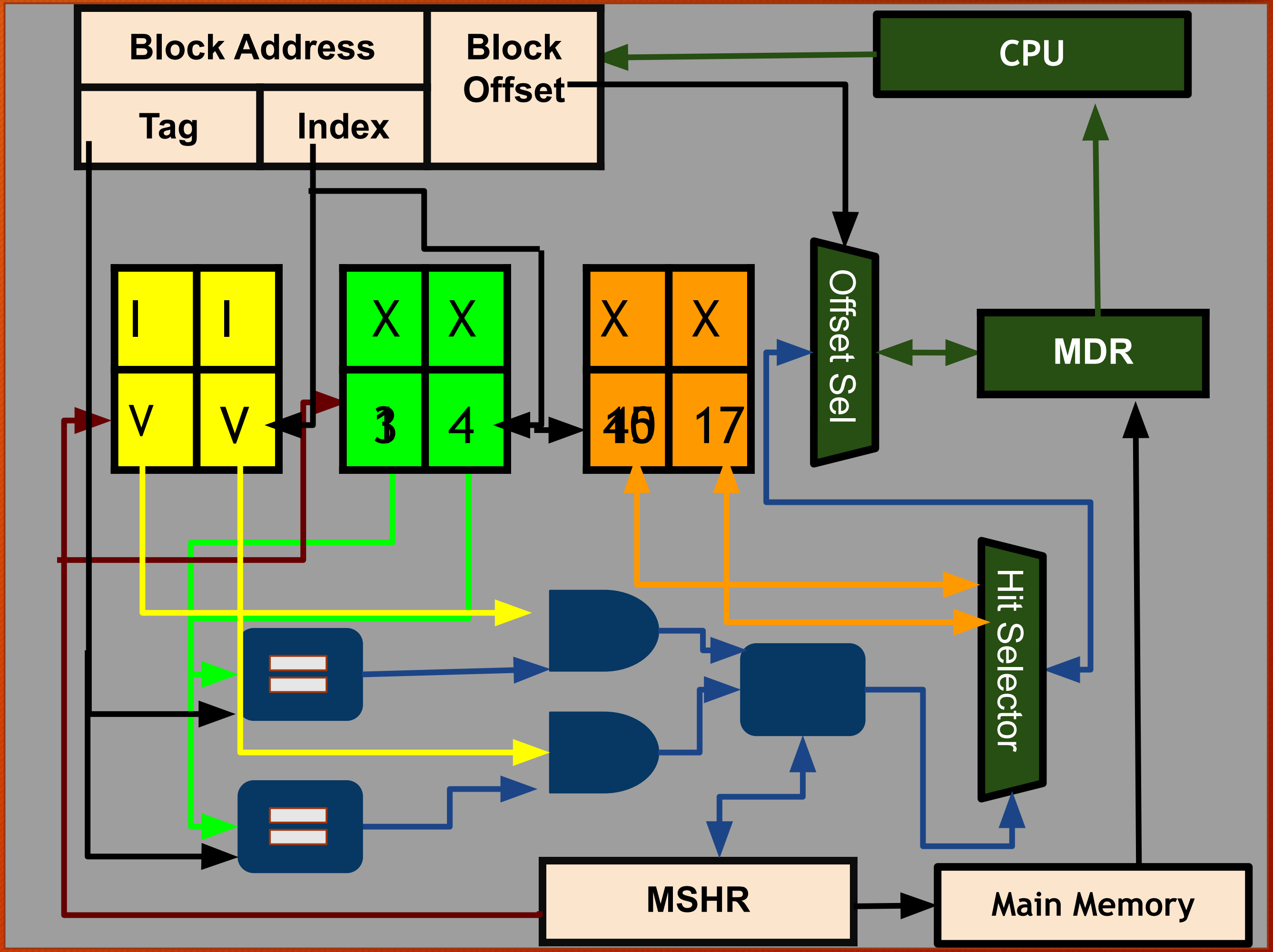
Set-Associative Cache

- **Problem:** A 4-way set-associative cache of size 32KB can be addressed using a 40 bit address. Block size of the cache is 64B. Find the total number of bits required for indexing, block offset and tag. Also find the tag array size.
  - Block Size = 64 B => Block Offset = 6 Bits
  - Cache Size = 32 KB => No. of Blocks = 32KB/64B = 512
  - Ways = 4
  - No. of Sets = 512/4 = 128
  - Indexing or Set bits = log 128 = 7
  - Tag bits = 40 – 7 - 6 = 27
  - Tag array size = (27 + 1) * 512 bits = 28 * 64 B



Block Replacement: Concept and Design

- ❑ **Replacement Policy:** A block allocation policy that works inside a set
- ❑ An existing block is replaced when a new block is allocated inside a set.
- ❑ Predictions are used to replace the block. Why? -> We don't know future.



B A	I	T	D
0	0	0	x
1	1	0	x
2	0	1	x
3	1	1	45
4	0	2	x
5	1	2	x
6	0	3	x
7	1	3	10
8	0	4	x
9	1	4	17

Block Replacement: Concept and Implementation

Random Replacement

- EvictionAlgorithm:
  - S = Set[CPUReq.Index]
  - ReplacedBlock = Random(S)
- Advantages:
  - Easy to Implement
- Disadvantages:
  - Causes performance penalty
  - No certainty

FIFO Replacement

- EvictionAlgorithm:
  - S = Set[CPUReq.Index]
  - ReplacedBlock = Front(S)
- Advantages:
  - Has certainty
- Disadvantages:
  - What about the blocks which are re-used?

LRU Replacement

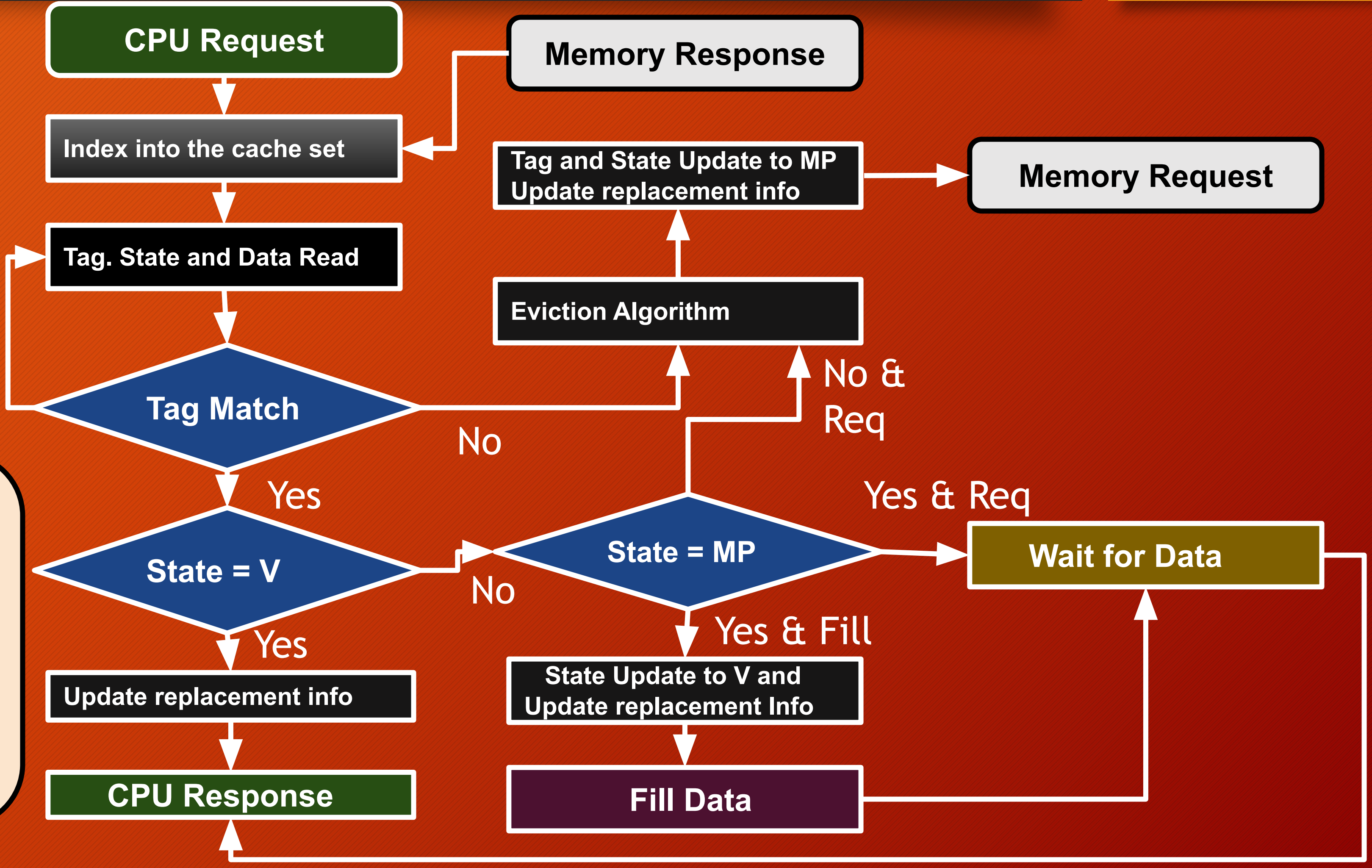
- EvictionAlgorithm:
  - S = Set[CPUReq.Index]
  - ReplacedBlock = maxTime(Accessed[S])
- BlockAccessReplacmentAlgorithm:
  - Accessed[B] = SystemClock
- Advantages:
  - Reuse is taken care-of
- Disadvantages:
  - Complex implementation in Hardware

Study! Pseudo LRU and Re-reference Interval Prediction!

Replacement Policy: Implementation

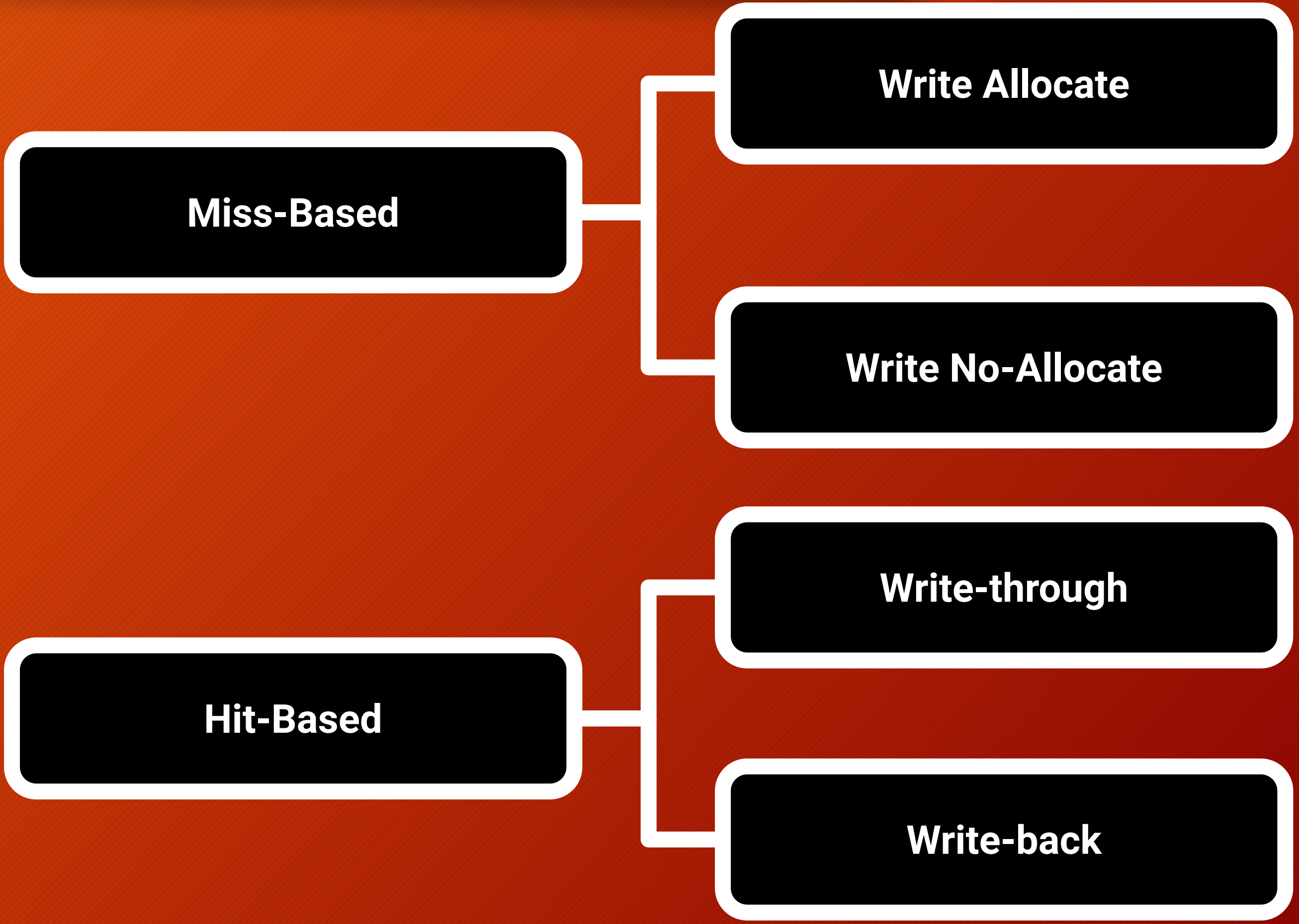
```
•S = Block[CPUReq.Index]
•for each way B in the Set S
•         If B.Tag = CPUReq.Tag & B.State = V
•         Then CPUResp = B.Data// Return
•          If B.Tag = CPUReq.Tag & B.State = MP
•          Then CPUReq.wait // Return
•B = EvictionAlgorithm(S)
•B.State = MP
•B.Tag = CPUReq.Tag
•MemReq = CPUReq
•BlockAccessReplacement()
```

```
•S =  Block[CPUReq.Index]
•for each way B in the Set S
•        If B.Tag = MemResp.Tag & B.State = MP
•        Then B.State = V
•             B.Data = MemResp.Data
•             CPUResp.Data = B.Data
•             BlockAccessReplacement()
•             Notify All Waiting To Restart
```



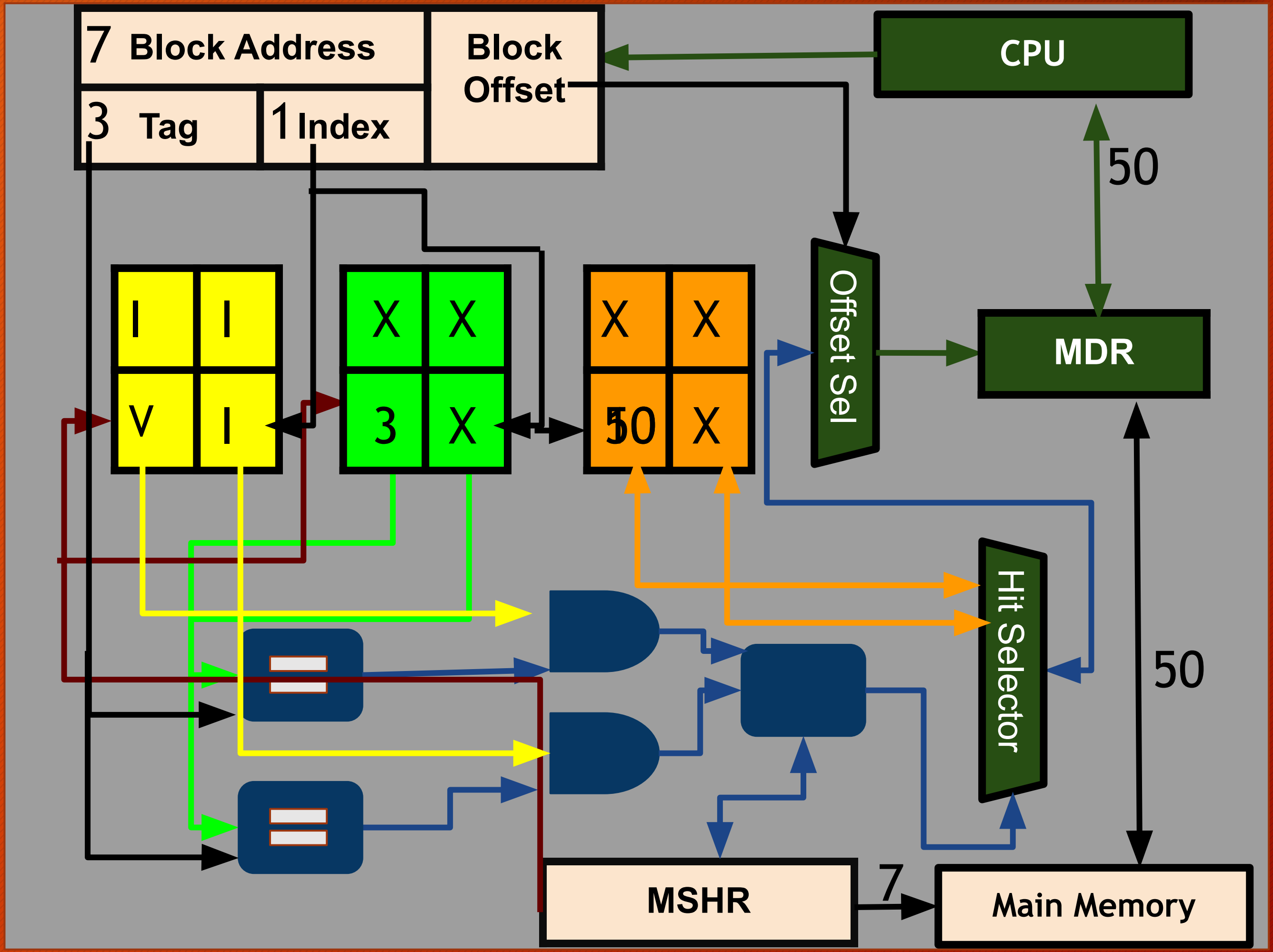
Write Strategy

Writes are important because the order in which they are serviced maintain coherence of the data



Write-through Cache: Overview and Design

- ❑ Simplest way of maintaining memory coherence with the main memory.
- ❑ Data is updated in the cache and is also updated to the main memory immediately.
- ❑ Implementation and hardware complexity is low.
- ❑ Performance bottleneck as the memory starts seeing writes to the same address.
- ❑ Write-through caches are used only with Write-no allocate.



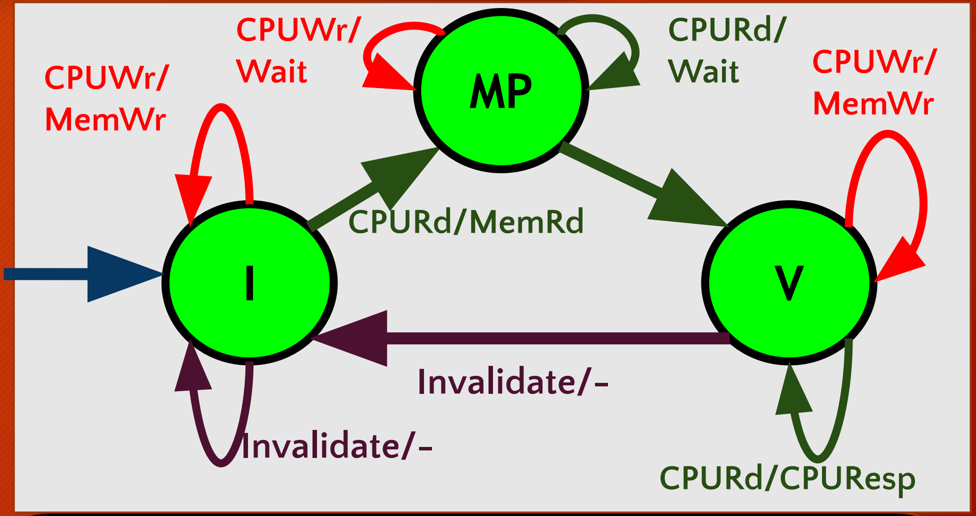
B A	I	T	D
0	0	0	x
1	1	0	x
2	0	1	x
3	1	1	45
4	0	2	x
5	1	2	x
6	0	3	x
7	1	3	50
8	0	4	x
9	1	4	17

Write-through: Mealy Machine

- **Inputs:**
  - CPURd and CPUWr for load and store instructions.
  - They can be invalidates from replacement/bus.
  - Memory response for read misses
- **Outputs:**
  - MemRd and MemWr for memory reads and writes upon read miss and writes.
  - CPUResp for reads

State Table

CS	Input	NS	Output
I	CPURd	MP	MemRd
I	CPUWr	I	MemWr
I	Invalidate	I	-
MP	CPURd/ CPUWr	MP	Action is to wait
MP	MemResp	V	CPUResp
V	CPURd	V	CPUResp
V	CPUWr	V	MemWr
V	Invalidate	I	-

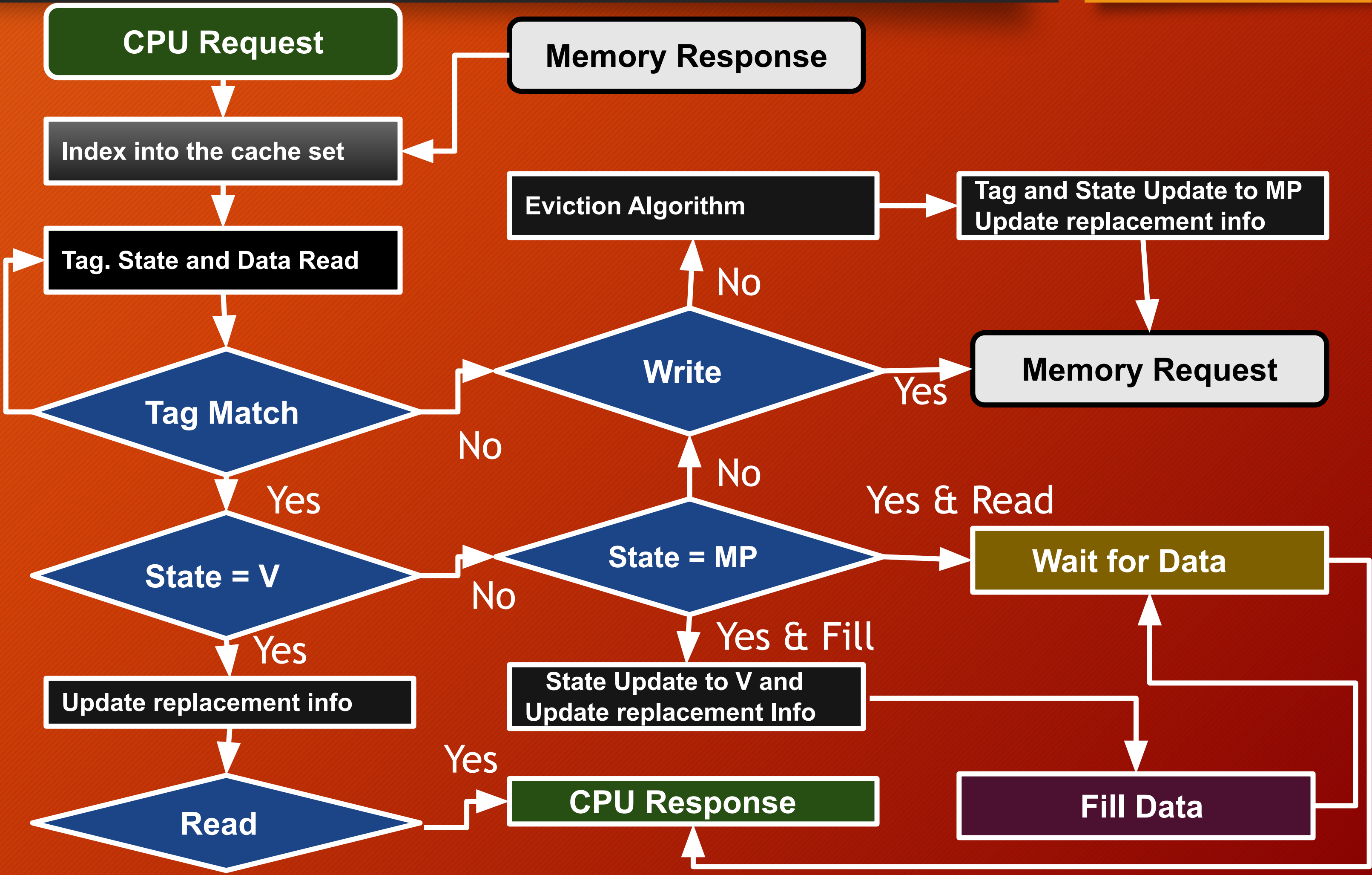


- **States:** Status of all the words part of a block inside the cache.
  - **Invalid (I):** Block is not present inside the cache.
  - **Valid (V):** Block is present inside the cache and all the words in the block have the same content as main memory.
  - **MissPending (MP):** Tags are present but data has not been received from the memory.

Write-through: Implementation

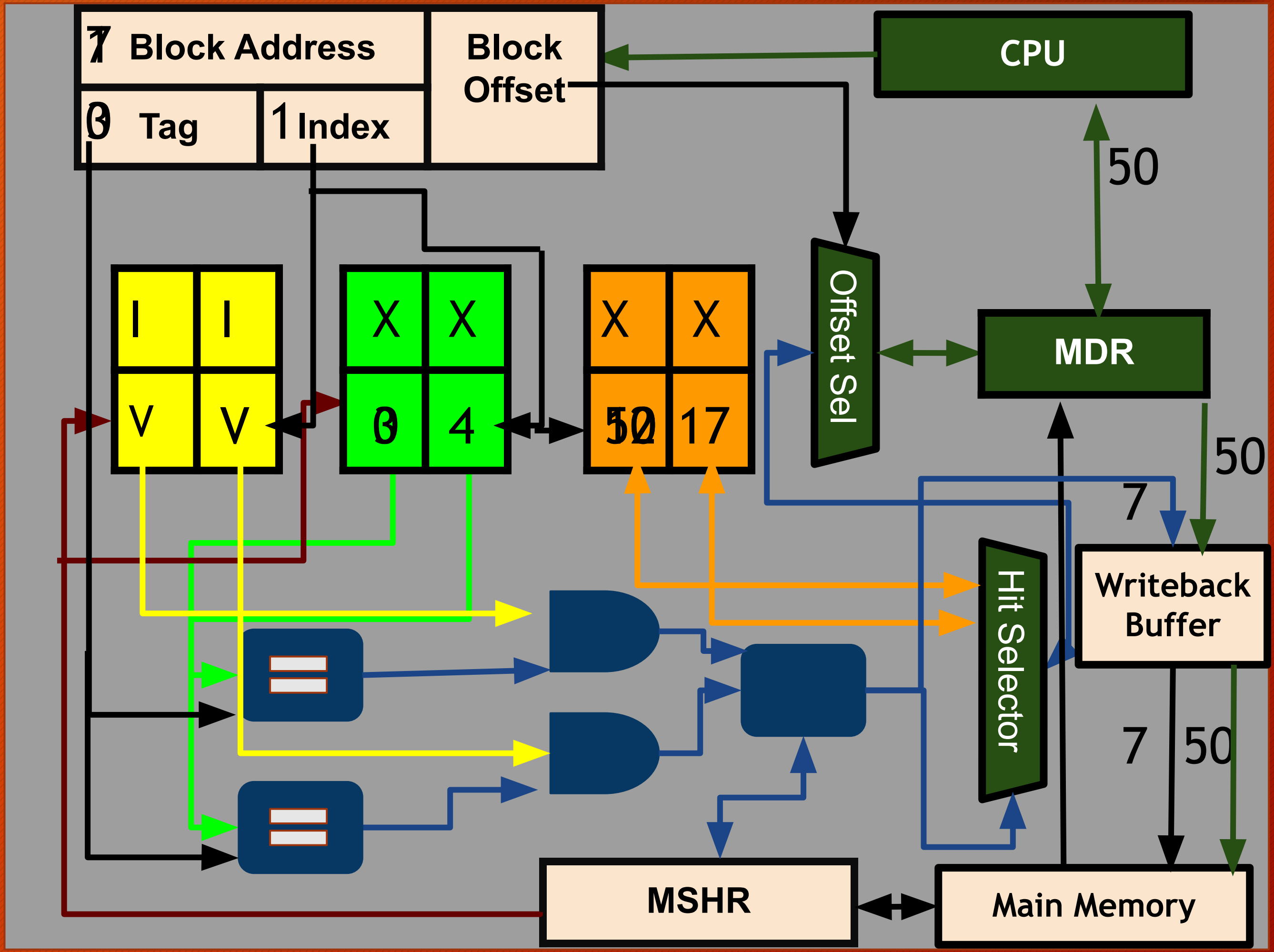
```
•S = Block[CPUReq.Index]
•for each way B in the Set S
•        If B.Tag = CPUReq.Tag & B.State = V
•        Then CPUResp = B.Data// Return
•         If B.Tag = CPUReq.Tag & B.State = MP
•         Then CPUReq.wait // Return
•If CPUReq.Type = READ
•         B = EvictionAlgorithm(S)
•         B.State = MP
•         B.Tag = CPUReq.Tag
•         BlockAccessReplacement()
•MemReq = CPUReq
•
```

```
•S =  Block[CPUReq.Index]
•for each way B in the Set S
•      If B.Tag = MemResp.Tag & B.State = MP
•      Then B.State = V
•            B.Data = MemResp.Data
•            CPUResp.Data = B.Data
•            BlockAccessReplacement()
•            Notify All Waiting To Restart
```



Write-back Cache: Overview and Design

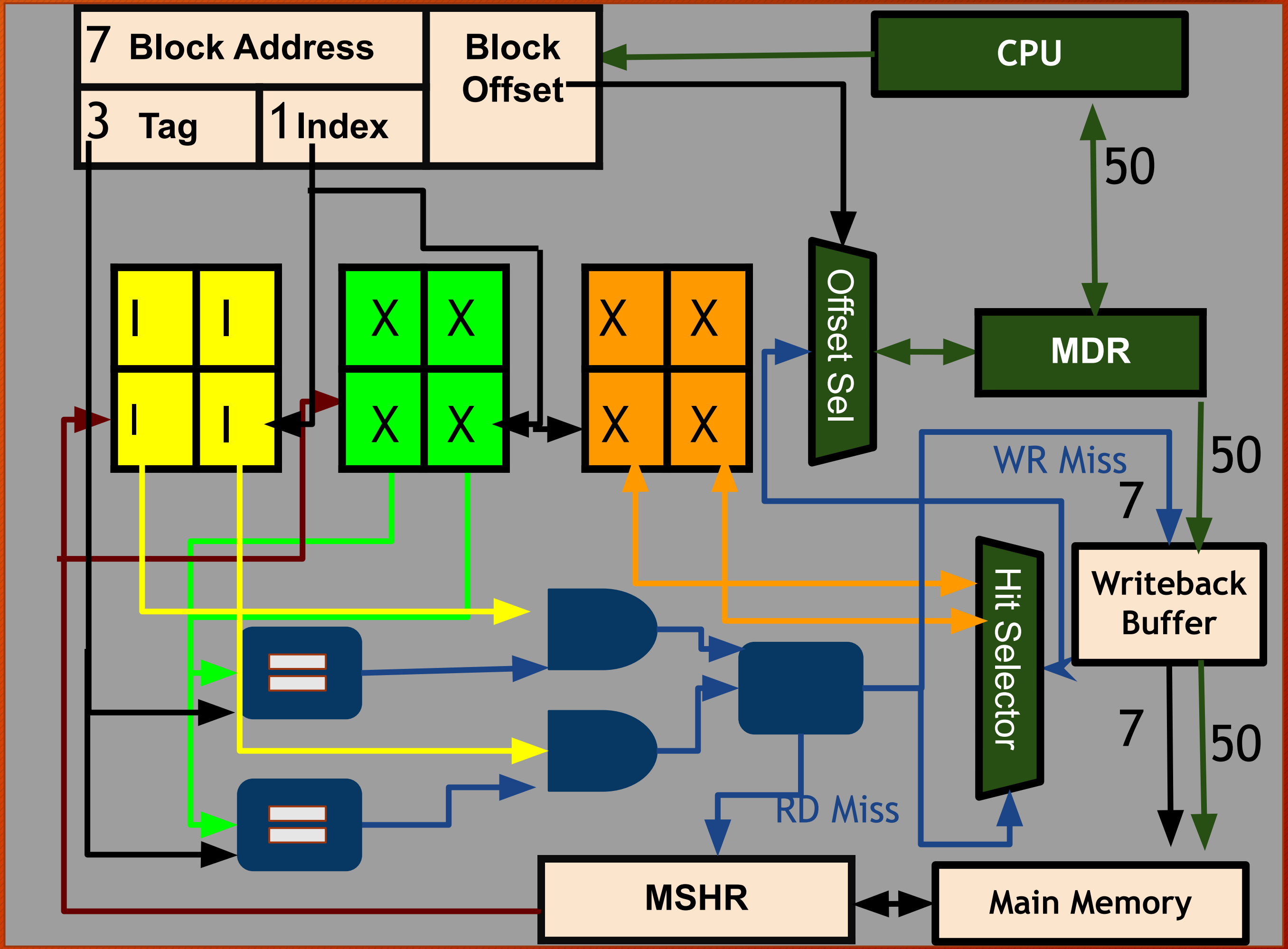
- ❑ Lazy updation of main memory
- ❑ Data is updated in the cache and is only updated to the main memory after eviction of block.
- ❑ Implementation and hardware complexity is high.
- ❑ Performance bottleneck is very low.
- ❑ Write-back caches are used with write-no allocate and write-allocate.



B A	I	T	D
0	0	0	x
1	1	0	12
2	0	1	x
3	1	1	x
4	0	2	x
5	1	2	x
6	0	3	x
7	1	3	50
8	0	4	x
9	1	4	17

Write-no Allocate Cache: Overview and Design

- ❑ Does not allocate the cache line when there is a cache miss for writes.
- ❑ Data is updated in the next level directly.
- ❑ Advantage: Very simple to design, For one touch write operation, this design is performance friendly.
- ❑ Issue: Does not exploit spatial locality for writes



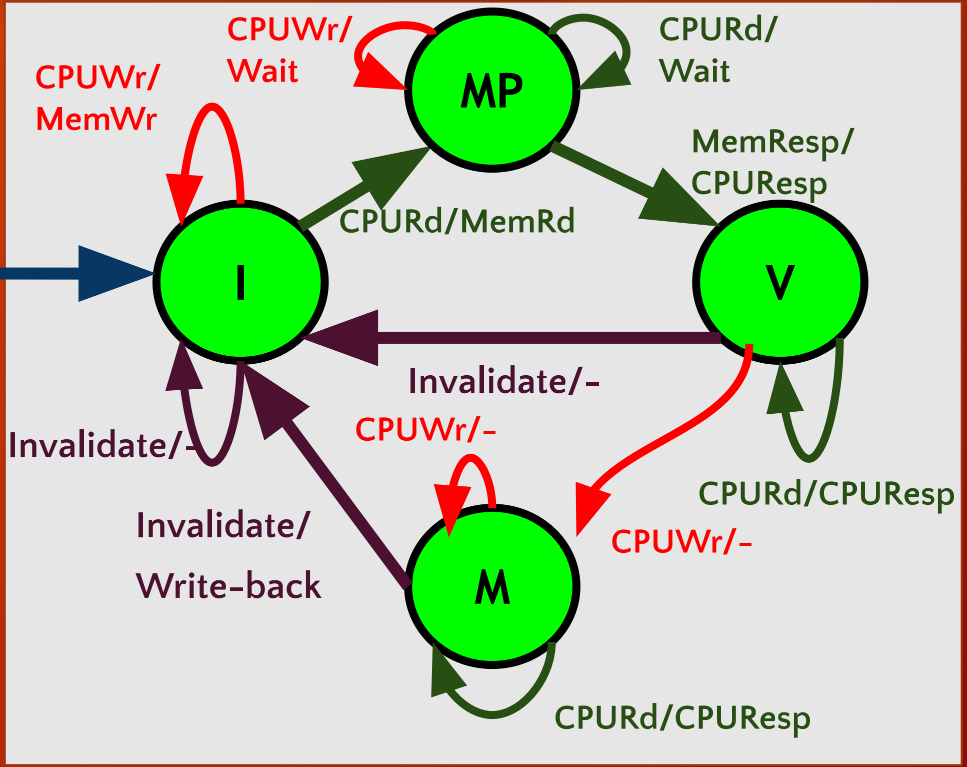
B A	I	T	D
0	0	0	x
1	1	0	x
2	0	1	x
3	1	1	45
4	0	2	x
5	1	2	x
6	0	3	x
7	1	3	50
8	0	4	x
9	1	4	17

Write-back No-Allocate: Mealy Machine

State Table

CS	Input	NS	Output
I	CPURd	MP	MemRd
I	CPUWr	I	MemWr
I	Invalidate	I	-
MP	CPURd/ CPUWr	MP	Action is to wait
MP	MemResp	V	CPUResp

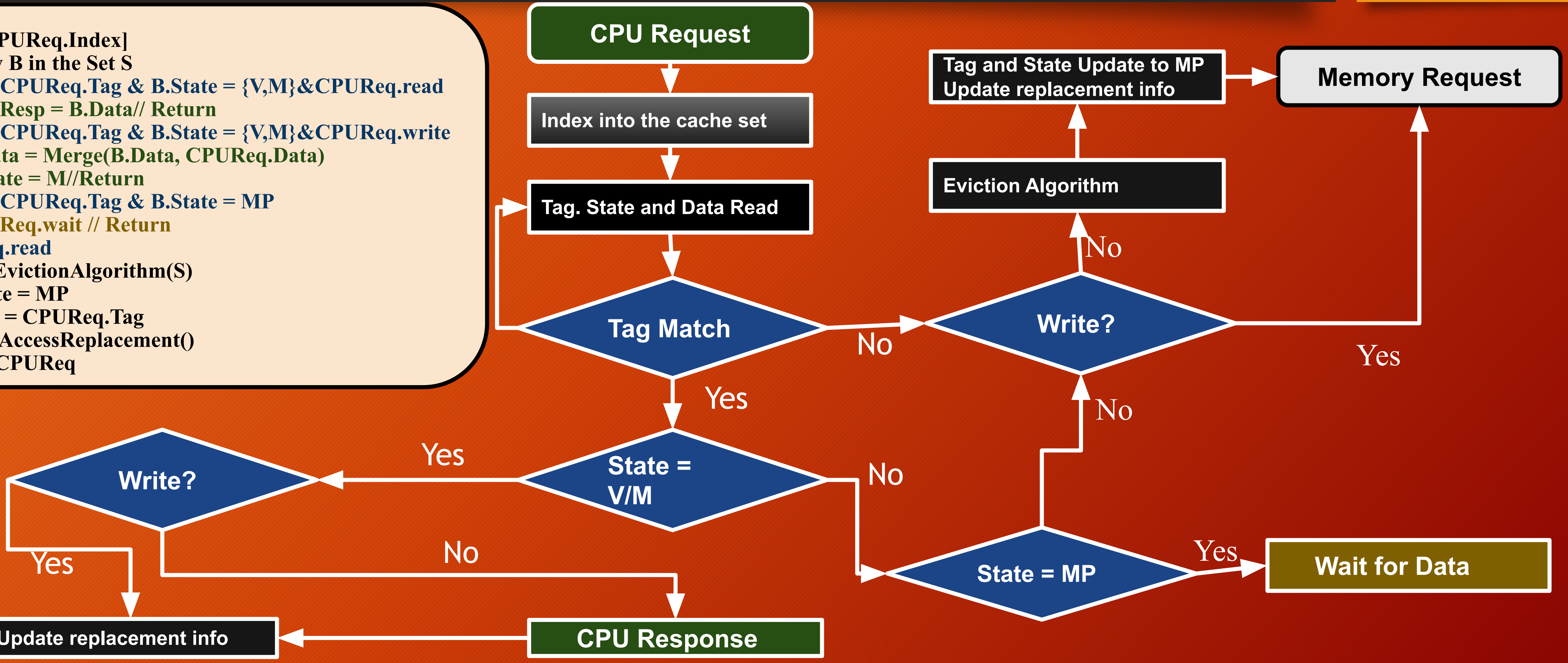
CS	Input	NS	Output
V	CPURd	V	CPUResp
V	CPUWr	M	-
V	Invalidate	I	-
M	CPURd	M	CPUResp
M	CPUWr	M	-
M	Invalidate	I	Writeback



- Modified (M): The data is different from that of the main memory and hence, need to be written back on eviction.

Write-back No-Allocate: Access Algorithm

```
•S = Block[CPUReq.Index]
•for each way B in the Set S
•   If B.Tag = CPUReq.Tag & B.State = {V,M}&CPUReq.read
•   Then CPUResp = B.Data// Return
•   If B.Tag = CPUReq.Tag & B.State = {V,M}&CPUReq.write
•   Then B.Data = Merge(B.Data, CPUReq.Data)
•          B.State = M//Return
•   If B.Tag = CPUReq.Tag & B.State = MP
•   Then CPUReq.wait // Return
•   If CPUReq.read
•   Then B = EvictionAlgorithm(S)
•         B.State = MP
•         B.Tag = CPUReq.Tag
•         BlockAccessReplacement()
•MemReq = CPUReq
```



CPU Request

Index into the cache set

Tag. State and Data Read

Tag Match

Yes

No

State = V/M

Yes

No

Write?

Yes

No

Update replacement info

CPU Response

State = MP

Yes

No

Wait for Data

Write?

No

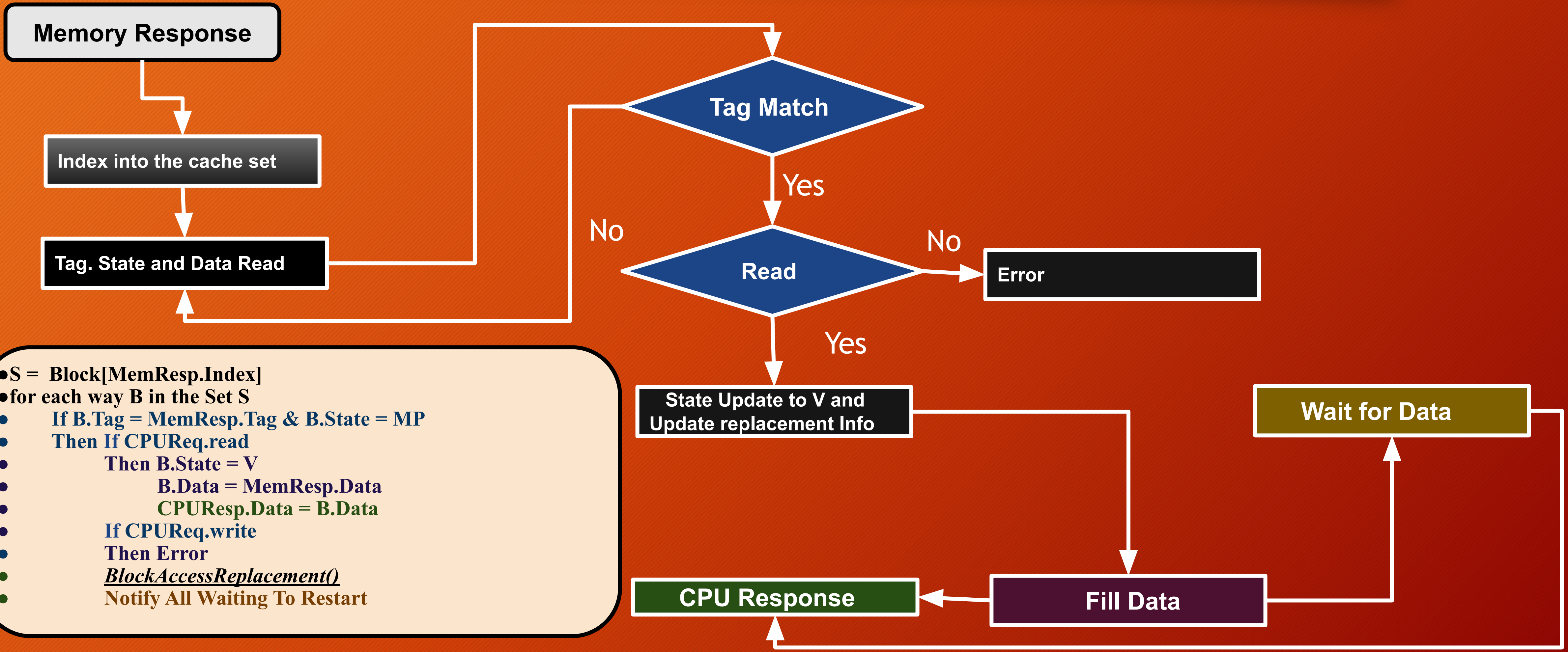
Yes

Eviction Algorithm

Tag and State Update to MP
Update replacement info

Memory Request

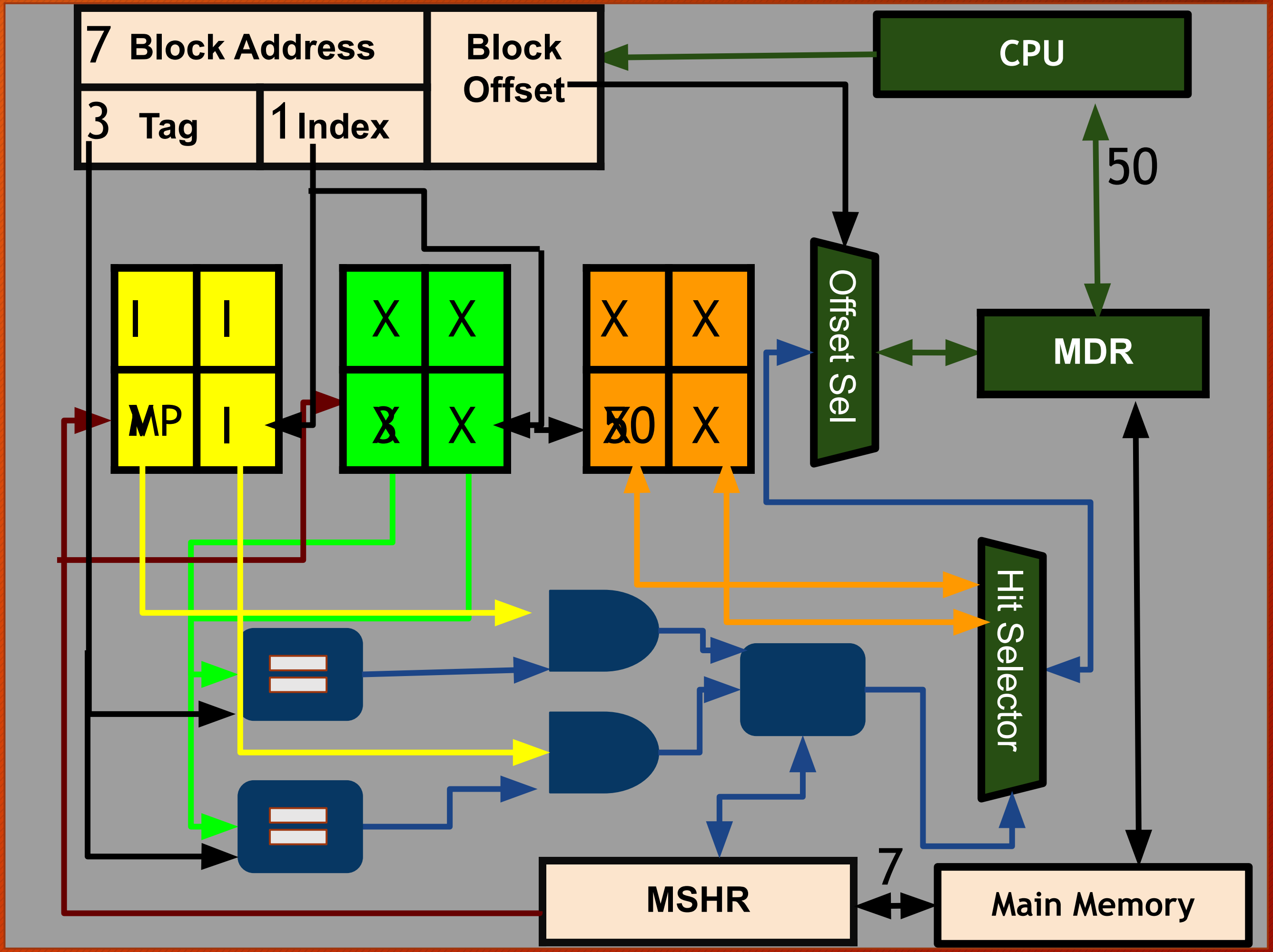
Write-back No Allocate: Fill Algorithm



```
•S =  Block[MemResp.Index]
•for each way B in the Set S
•     If B.Tag = MemResp.Tag & B.State = MP
•     Then If CPUReq.read
•          Then B.State = V
•               B.Data = MemResp.Data
•               CPUResp.Data = B.Data
•          If CPUReq.write
•          Then Error
•          BlockAccessReplacement()
•          Notify All Waiting To Restart
```

Write-allocate Cache: Overview and Design

- ❑ Allocates a cache line when there is a cache miss for writes
- ❑ Upon a write request that is a miss, generate a read request corresponding to this address to read the data from memory.
- ❑ Issue: For one touch write operation, this design is not performance friendly.
- ❑ Advantages: Exploit spatial locality for writes



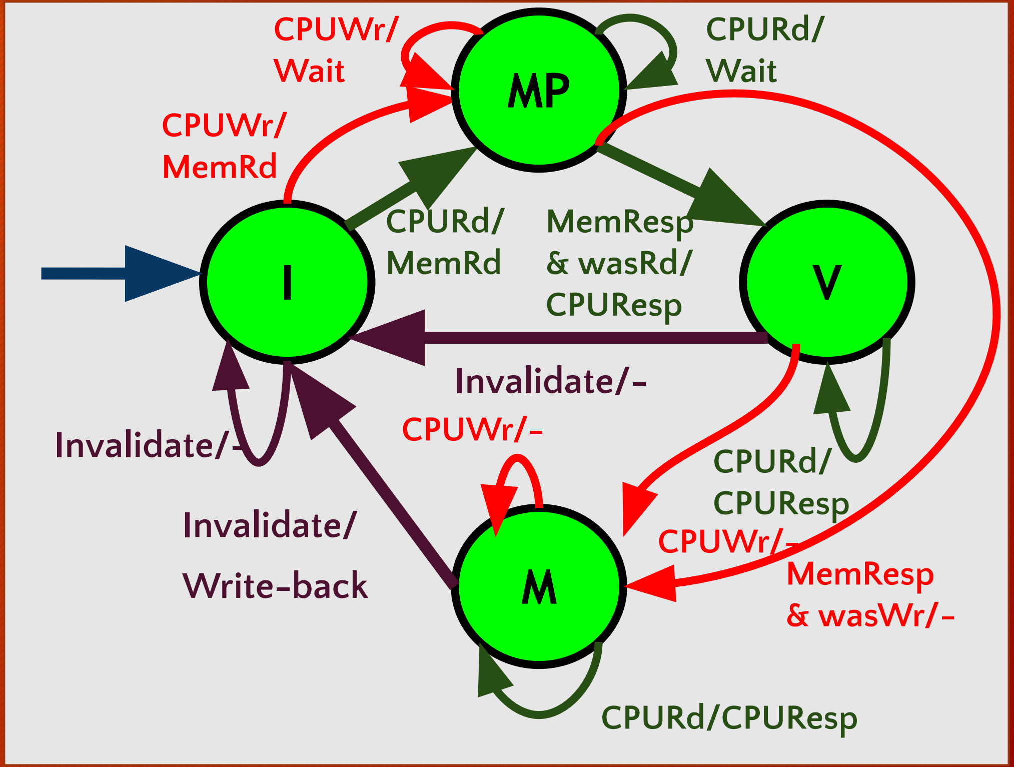
B A	I	T	D
0	0	0	x
1	1	0	x
2	0	1	x
3	1	1	45
4	0	2	x
5	1	2	x
6	0	3	x
7	1	3	10
8	0	4	x
9	1	4	17

Write-back Allocate: Mealy Machine

State Table

CS	Input	NS	Output
I	CPURd	MP	MemRd
I	CPUWr	MP	MemRd
I	Invalidate	I	-
MP	CPURd/ CPUWr	MP	Action is to wait
MP	MemResp & wasRd	V	CacheFill & CPUResp
MP	MemResp & wasWr	M	CacheFill

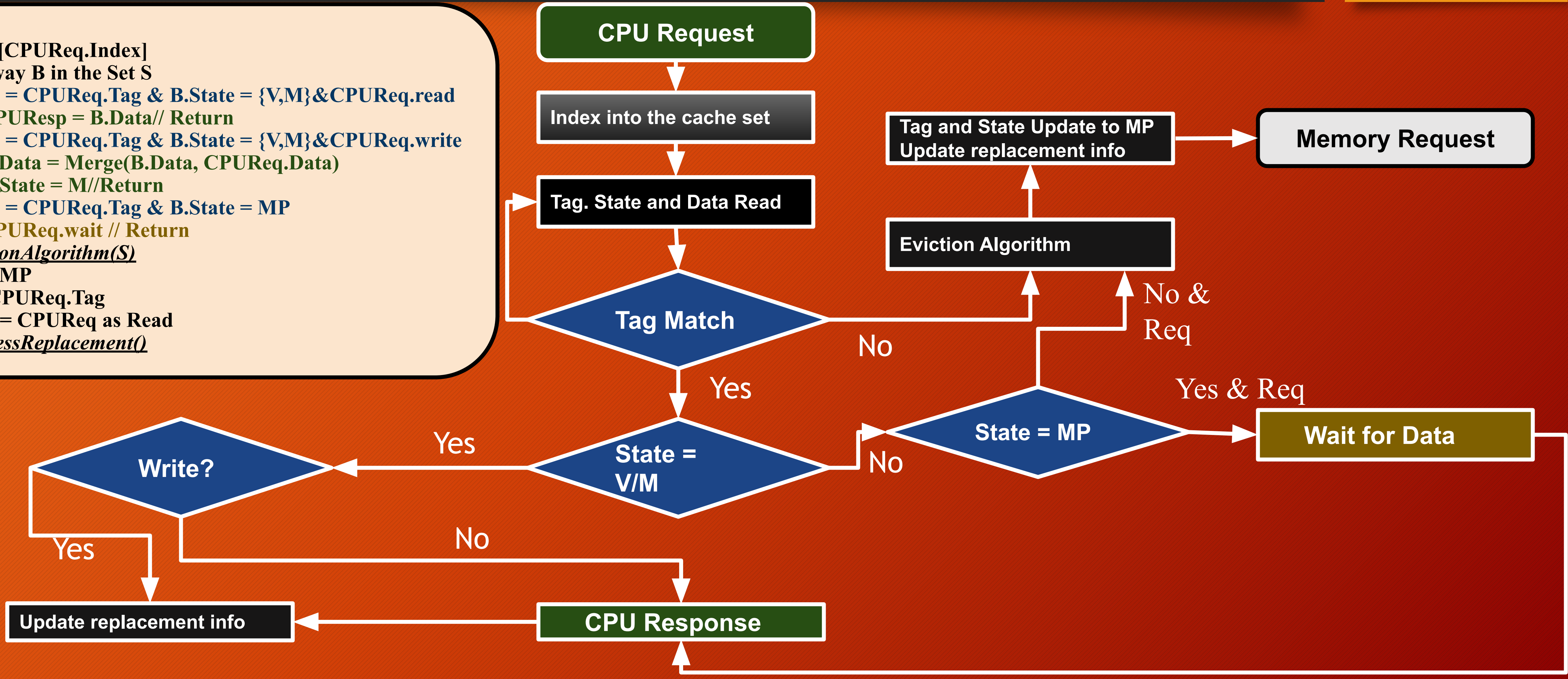
CS	Input	NS	Output
V	CPURd	V	CPUResp
V	CPUWr	M	-
V	Invalidate	I	-
M	CPURd	M	CPUResp
M	CPUWr	M	-
M	Invalidate	I	Writeback



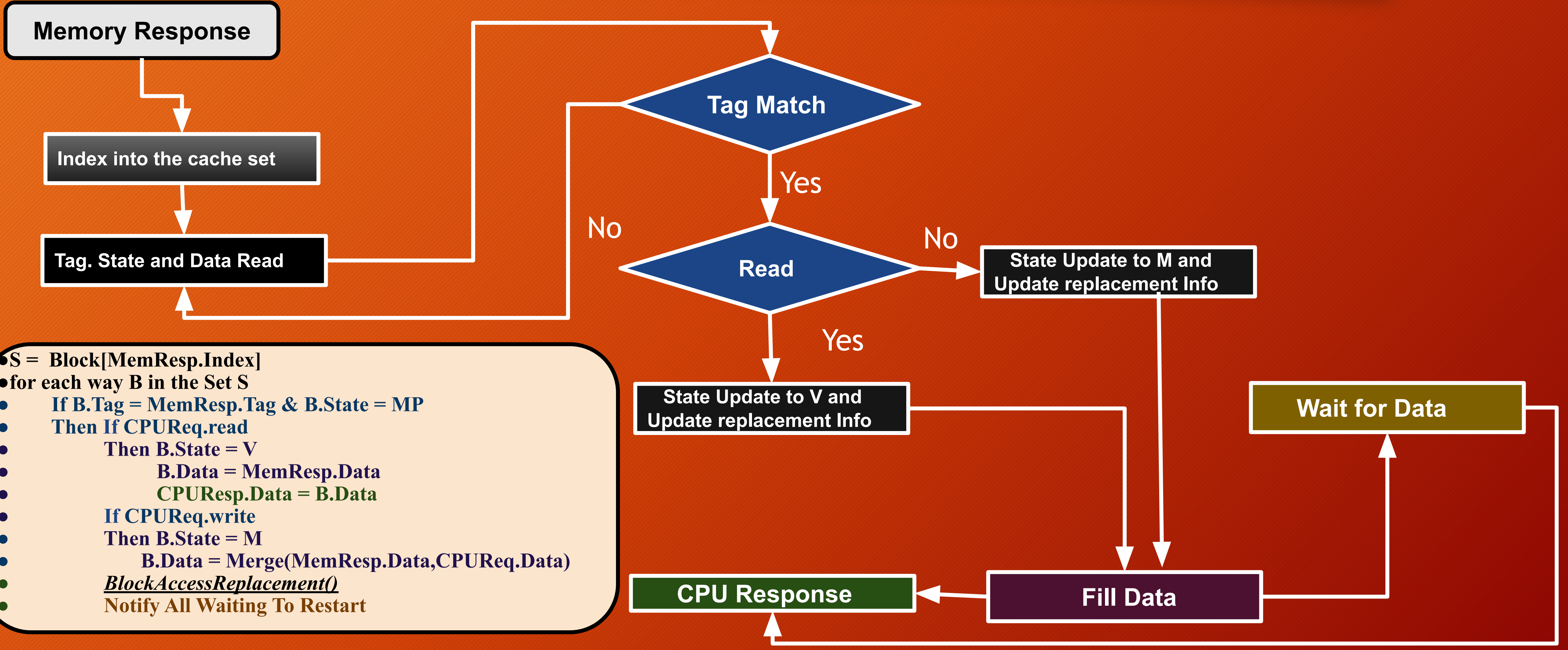
- **Modified (M):** The data is different from that of the main memory and hence, need to be written back on eviction.

Write-back Allocate: Access Algorithm

```
•S = Block[CPUReq.Index]
•for each way B in the Set S
•   If B.Tag = CPUReq.Tag & B.State = {V,M}&CPUReq.read
•   Then CPUResp = B.Data// Return
•   If B.Tag = CPUReq.Tag & B.State = {V,M}&CPUReq.write
•   Then B.Data = Merge(B.Data, CPUReq.Data)
•           B.State = M//Return
•   If B.Tag = CPUReq.Tag & B.State = MP
•   Then CPUReq.wait // Return
•B = EvictionAlgorithm(S)
•B.State = MP
•B.Tag = CPUReq.Tag
•MemReq = CPUReq as Read
•BlockAccessReplacement()
```



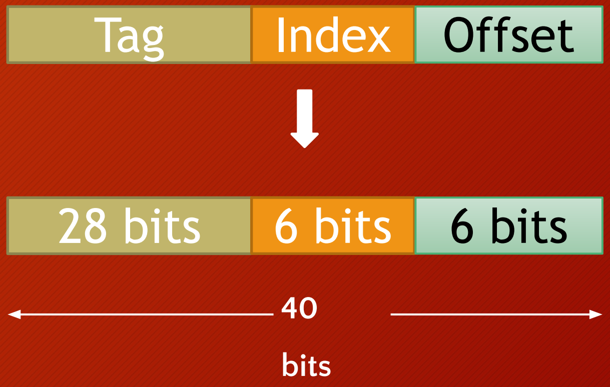
Write-back Allocate: Fill Algorithm



```
S =  Block[MemResp.Index]
for each way B in the Set S
    If B.Tag = MemResp.Tag & B.State = MP
    Then If CPUReq.read
        Then B.State = V
            B.Data = MemResp.Data
            CPUResp.Data = B.Data
        If CPUReq.write
        Then B.State = M
           B.Data = Merge(MemResp.Data,CPUReq.Data)
        BlockAccessReplacement()
        Notify All Waiting To Restart
```

Problem

- **Problem:** A 8-way set-associative cache of size 16KB can be addressed using a 40 bit address. Block size of the cache is 64B. Find the total number of bits required for indexing, block offset and tag. Also find the tag array size. Determine the tag array size if the cache is designed as write-through and write-back protocol.
  - Block Size = 64 B => Block Offset = 6 Bits
  - Cache Size = 16 KB => No. of Blocks = 16KB/64B = 256
  - Ways = 8
  - No. of Sets = 256/8 = 32
  - Indexing or Set bits = log 32 = 5
  - Tag bits = 40 – 6 - 5 = 29
  - Tag array size (write-through) = (29 + 1) * 256 bits = 30 * 32 B
  - Tag array size (write-back) = (29 + 2) * 256 bits = 31 * 32 B



Causes of Cache Miss

- **Compulsory Miss:** First reference to an address
- **Capacity Miss:**
  - Occurs when working set does not fit inside the cache
- **Conflict/Collision Miss:**
  - It is not necessary that cache is full for a conflict miss occurs.
  - Conflict due to block placement in the same index.
  - They don't occur if the cache is fully-associative with LRU replacement policy.
  - Conflict misses are maximum with direct-mapped cache
- **Coherence Miss:** Occurs in multiprocessor/external processor system (Will study later)

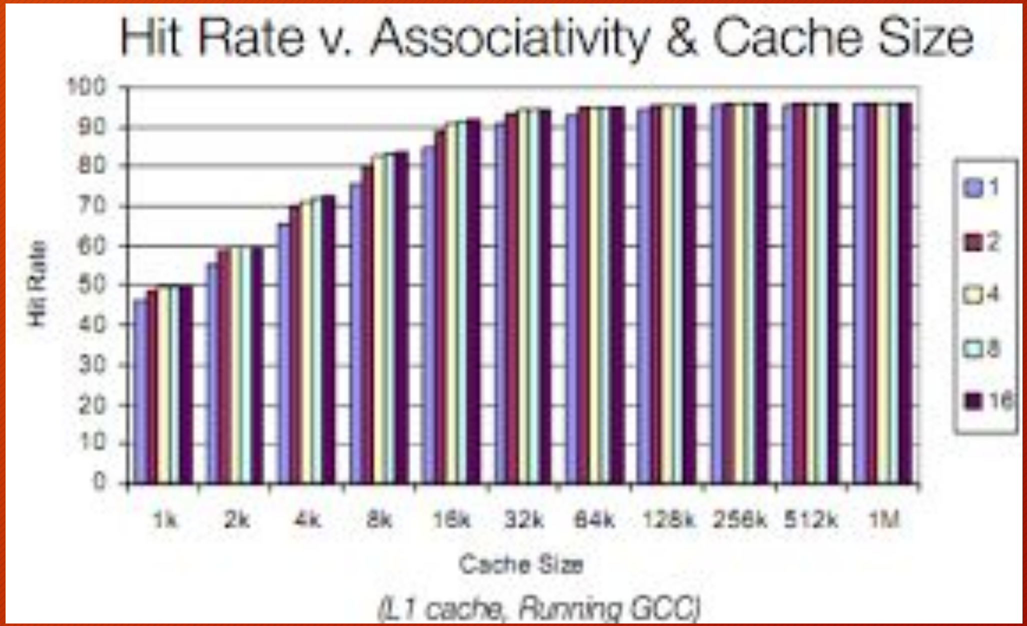
Q: A direct mapped cache of size 4 has the following accesses: 0,1,2,3,4,1,2,3,0,4,0. Categorize the misses

Ans:

0- Compulsory Miss
1- Compulsory Miss
2- Compulsory Miss
3- Compulsory Miss
4- Compulsory Miss
1- Hit
2- Hit
3- Hit
0- Capacity Miss
4- Capacity Miss
0- Conflict Miss

Large Cache Size and Higher Associativity

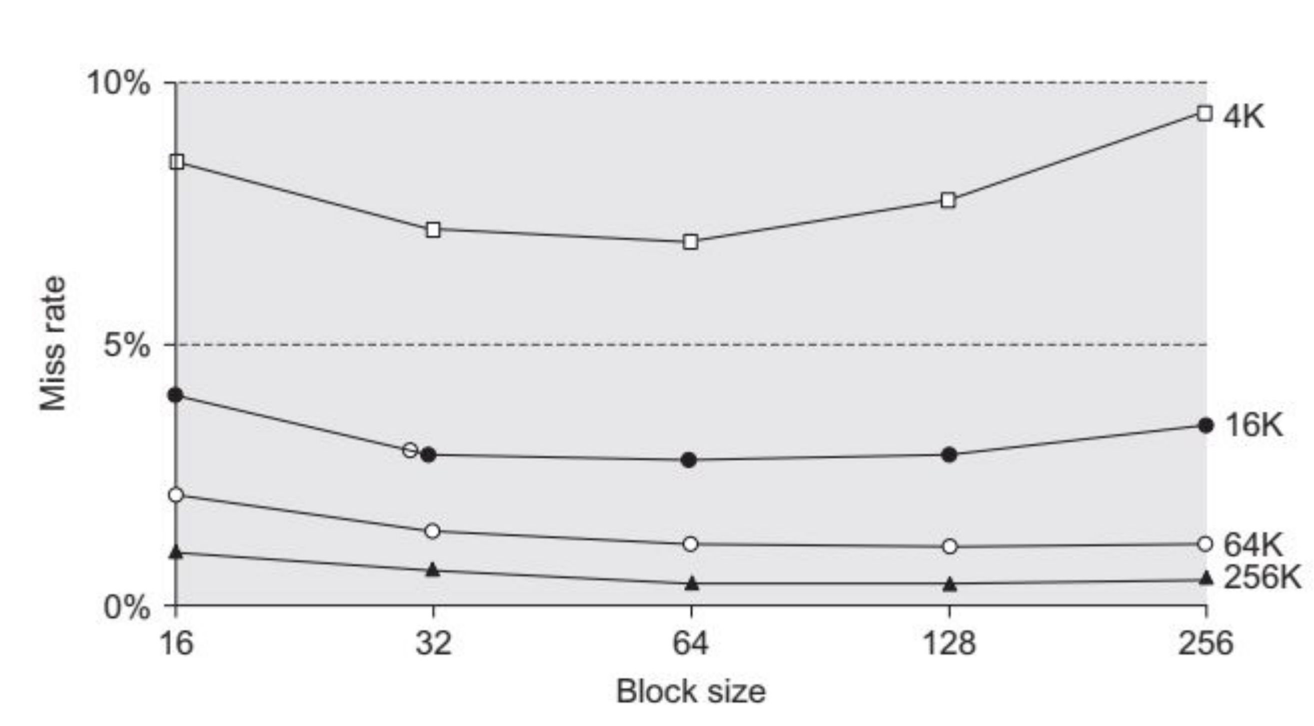
- Large Cache means a large subset of main memory close to the processor
- Large size of cache reduces conflict and capacity misses
- Performance is directly proportional to hit rate; it saturates after some particular configuration.
- Higher associativity reduces conflict misses
- Performance saturates when working set of an application fits inside the cache



Images from slide-11 Prof. Martha Kim

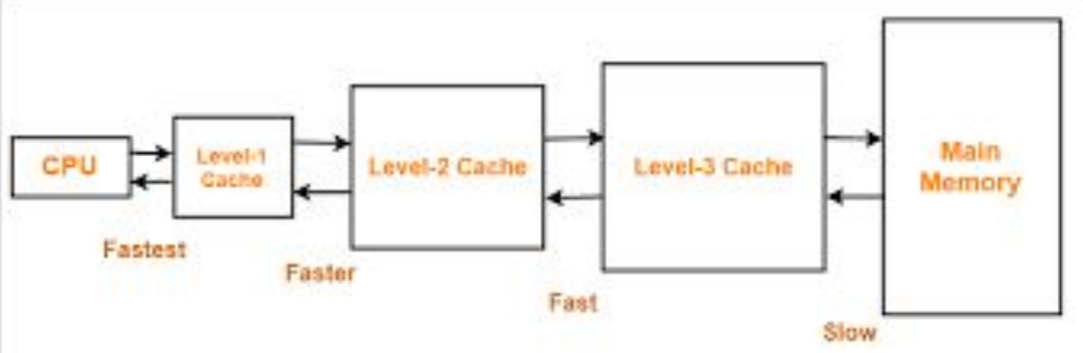
Large Block Size

- Large block size reduces miss rate
- Compulsory misses are the targets
- Exploits spatial locality
- Increases miss penalty
- Could increase conflict/capacity miss for same cache size



Multilevel Cache

- Targets miss penalty
- Large caches have high access time. Why?
- Local Miss Rate
- Global Miss Rate



CPU

Level-1 Cache

Level-2 Cache

Level-3 Cache

Main Memory

Fastest

Faster

Fast

Slow

Three Level Cache Organization

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times \text{Miss penalty}_{\text{L1}}$$

$$\text{Miss penalty}_{\text{L1}} = \text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}$$

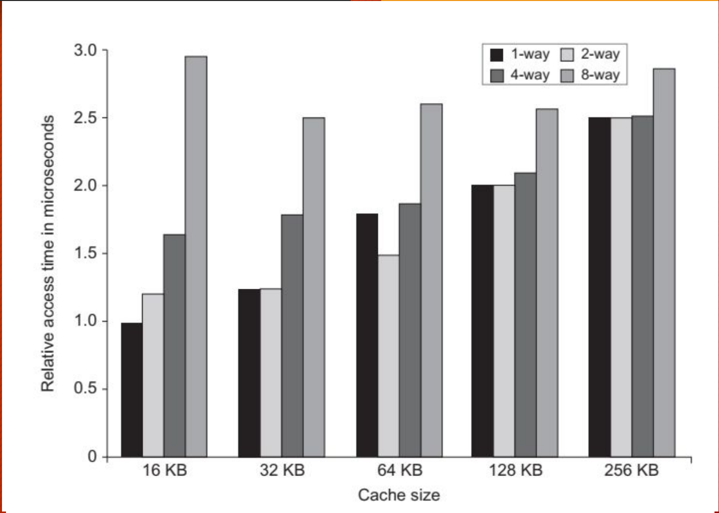
$$\begin{aligned}\text{Average memory access time} = {} & \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \\ & \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})\end{aligned}$$

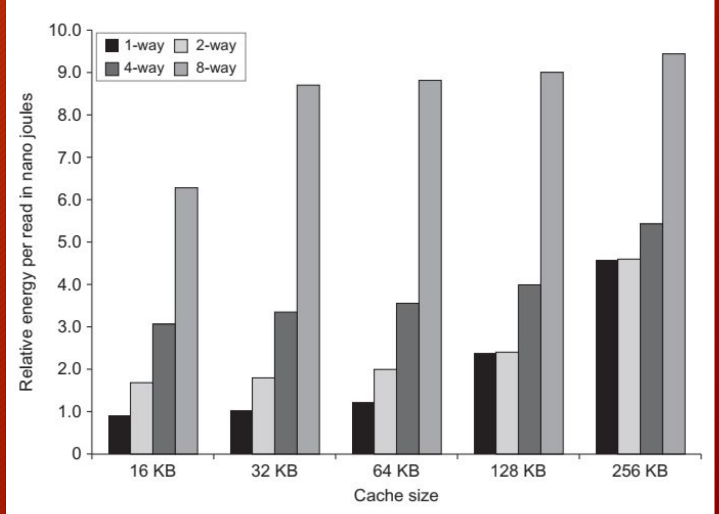
$$\text{Miss rate}_{\text{L1}} \times \text{Miss rate}_{\text{L2}}$$

$$\begin{aligned}\text{Average memory stalls per instruction} = {} & \text{Misses per instruction}_{\text{L1}} \times \text{Hit time}_{\text{L2}} \\ & + \text{Misses per instruction}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}\end{aligned}$$

Small and Simple First Level Cache

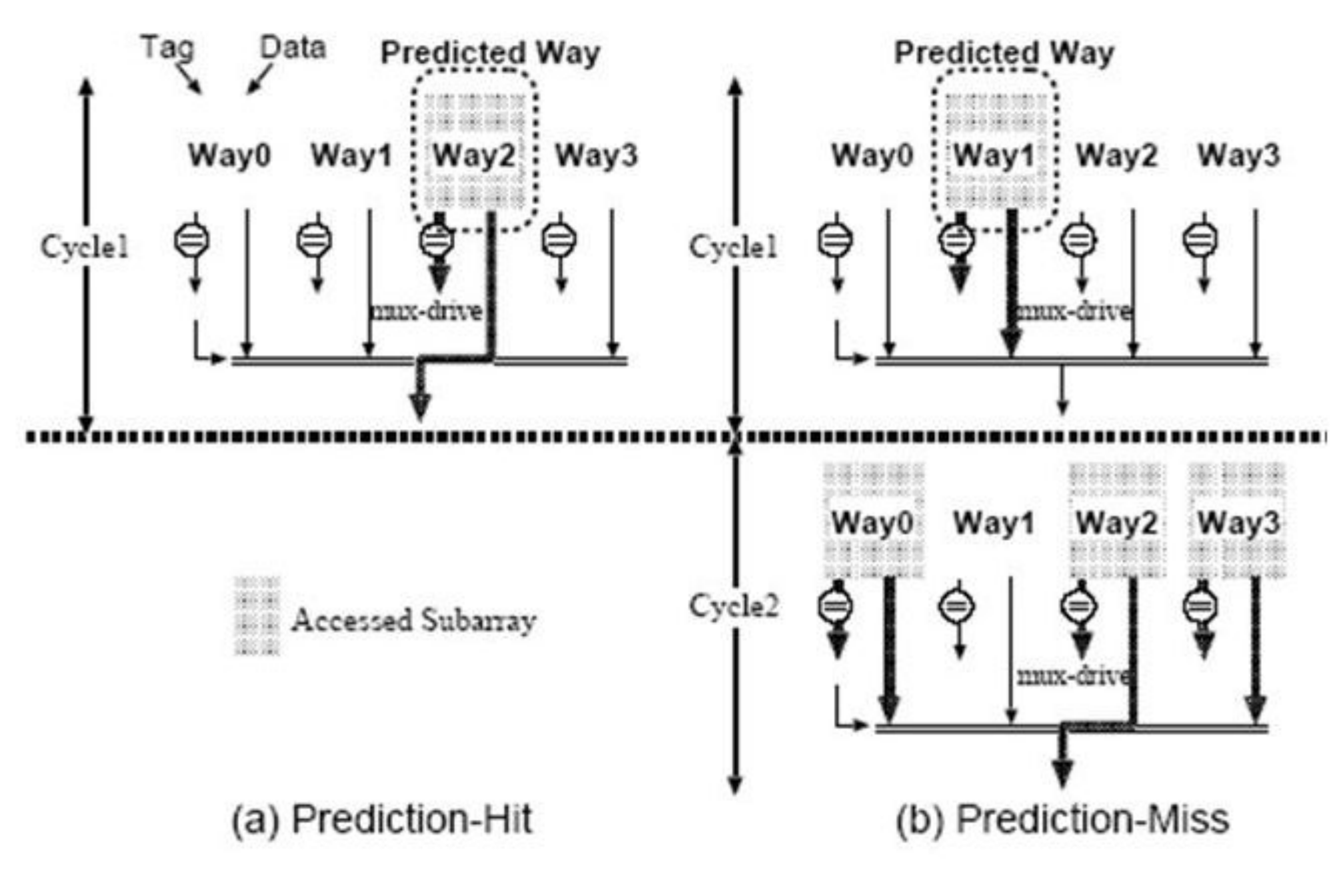
- Cache Access Time on Hit = Indexing time + Tag read/compare time + Multiplexor latency + Data access time + Request queuing delay etc.
- Less number of sets will have low index time
- Large tag size will have more power consumption
- More number of way will have high multiplexor latency
- Large cache size will have long data access latency
- L1 cache is close to the core and hence, must have very low hit latency





Way Prediction

- Target: Power Optimization
- Predict a way that would be hit and access that way.
- On miss, access all the other ways.
- Can use a simple history predictor like n-bit per set



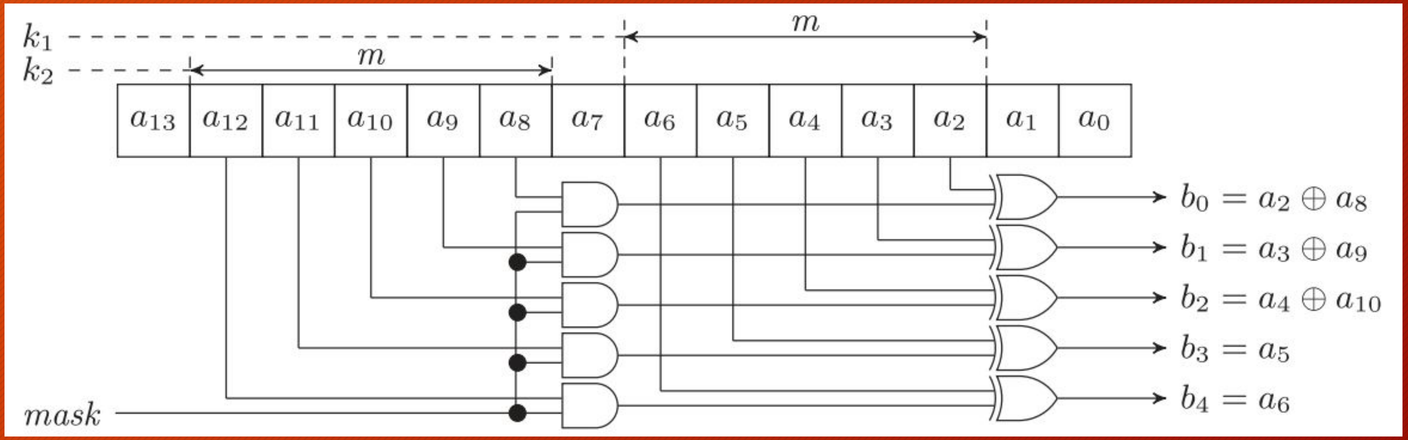
(a) Prediction-Hit

(b) Prediction-Miss

*Way-predicting set-associative cache for high performance and low energy consumption ISLPED 1999

Index Hashing

- Target: Miss Rate
- Attempts to reduce conflict misses due to hot spot
- What is hot-spot?
- Typically, higher-order bits of physical address (tags) are xor-ed with the index bits to get a hashed index
- Large caches may not require index hashing. Why?
- Direct mapped caches require index hashing. Why?



k_1
k_2
m
m
a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
$b_0 = a_2 \oplus a_8$
$b_1 = a_3 \oplus a_9$
$b_2 = a_4 \oplus a_{10}$
$b_3 = a_5$
$b_4 = a_6$
$mask$

Images from paper by Gert-Jan van den Braak et al.,
"Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs",
IEEE Transaction in Computers, Vol. 65, Issue. 7, July 2016