



- Memory Consistency and Cache Coherence
 - Memory Consistency and Cache Coherence ☐
 - TSO ☐ Total Store Ordering ☐
 - x86 ☐
- einsum
- RAM
- NoC
- Cache ☐ Write-through ☐ Write-back

Memory Consistency and Cache Coherence

Memory Consistency and Cache Coherence

1. Memory Consistency

- -
- -
- -
- -
- - Sequential Consistency**
 - Weak Consistency**
- -

2. Cache Coherence

- -
- -
- -
- -
- - MESI** **Modified** **Exclusive** **Shared** **Invalid**
 - Write Invalidate**
 - Write Update**

TSO Total Store Ordering



- 1. TSO Total Store Ordering
- 2. x86 RISC-V TSO RVTSO x86 TSO
Intel AMD
- 3. x86 SC sequential consistency FIFO write buffer
- 4. TSO SC store write buffer load bypass
 - 1. SC
 - 2. memory consistency
- 5. TSO store-load FENCE FENCE
FENCE write buffer FENCE
load

x86

1.

“”

1. “XCHG” “XADD”

2. LOCK

3. A #LOCK LOCK CMPXCHG

A

A

A
2. fence

1. sfence

2. lfence

3. mfence
3. C++11

1. Acquire-Release Synchronizes-With




2. Release-Consume carry-a-dependency

```
enum memory_order {  
    memory_order_relaxed, // Relaxed  
    memory_order_consume, // Release-Consume  
    memory_order_acquire, // Acquire-Release  
  
    memory_order_release, // Acquire-Release  
  
    memory_order_acq_rel, // Acquire-Release memory_order_acquire memory_order_release  
    memory_order_seq_cst //  
};
```















einsum



☐ ☐ ☐ ☐ ☐ **Free indices** ☐ ☐ ☐ ☐ ☐ ☐ **Summation indices** ☐ ☐

-  i j
 • 
 k



-  equation

 "ik,kj->ij"     
-  equation

-  equation  "ik,kj->ij"  "ik,kj->ji"  einsum 



- equation

"ik,kj"

"ij"
- equation

"..."

```
a = torch.randn(2,3,5,7,9)
# i = 7, j = 9
b = torch.einsum('...ij->...ji', [a])
```

□□□□ bilinear transformation□

```
np_a = a.numpy()
np_b = b.numpy()
```

```

np_c = c.numpy()
np_out = np.empty((2, 5), dtype=np.float32)

np_out = torch.einsum('ik,jkl,il->ij', [a, b, c]).numpy()
# ik broadcast  ikl
# il broadcast  ikl
# 'ik,jkl,il->ij'  'ikl,jkl,ikl->ij'

for i in range(0, 2):
    for j in range(0, 5):
        #  k  l
        sum_result = 0
        for k in range(0, 3):
            for l in range(0, 7):
                sum_result += np_a[i, k] * np_b[j, k, l] * np_c[i, l]
        np_out[i, j] = sum_result

```



```

a = torch.rand(2,3)
b = torch.rand(3,4)
c = torch.einsum("ik,kj->ij", [a, b])
#  torch.mm(a, b)

```

equation

c c[i, j] a[i, k] b[k, j] k

RAM

DRAM

- 1. 11
- 2. 111111
- 3. 11111111

SRAM

- 1. 1111111111
- 2. 11
 - 1. Cpu register Flip Flops 11 bit111111
 - 2. L1/L2 SRAM 6111111111 bank1111
 - 3. L3/L4 eDRAM/GCRAM 4111 /11 1111111111111111

NoC

OpenSMART

<https://github.com/hyoukjun/OpenSMART/tree/master>

connect

<https://users.ece.cmu.edu/~mpapamic/connect/>

<https://github.com/crossroadsfpga/connect/tree/main>

Flexnoc




is a commercial NoC generator by Arteris which generates a customized topology for each SoC,





Cache Write-through Write-back

Cache  write through  write back 

Write-through: Write is done synchronously both to the cache and to the backing store.

Write-back (or Write-behind) : Writing is done only to the cache. A modified cache block is written back to the store, just before it is replaced.

Write-through  Cache 





Write-back  Cache 



Write-misses 














Write allocate (aka Fetch on write) – Datum at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read-misses.

No-write allocate (aka Write-no-allocate, Write around) – Datum at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, actually only system reads are being cached.

Write allocate  write-hit 


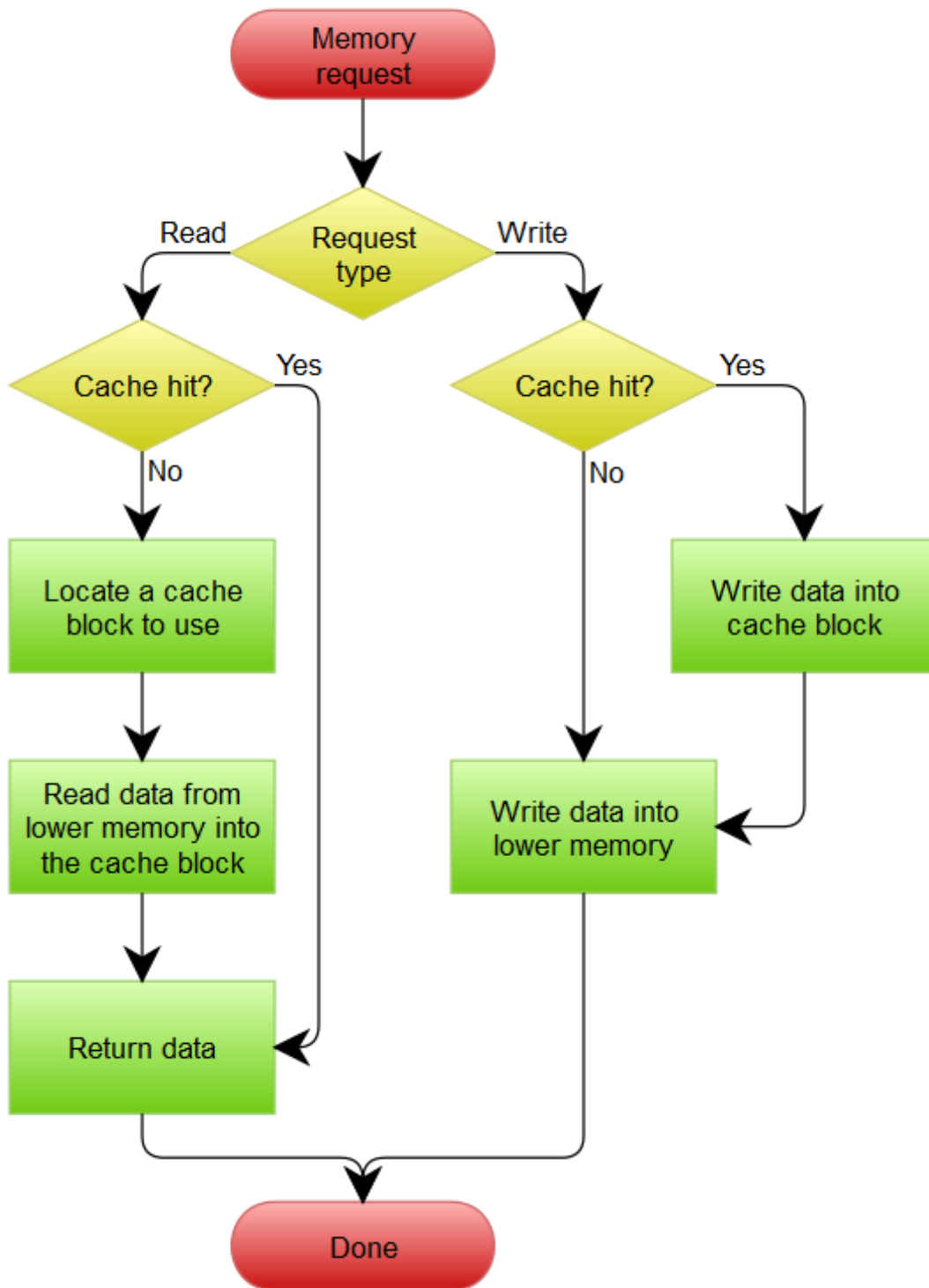
No-write allocate



 **Write-through**  **Write-back**   Write-back 
Write allocate  Write-through  No-write allocate  Write
allocate  Write-back  Write-through 



Write-through  A Write-Through cache with No-Write Allocation



Write-back

A Write-Back cache with Write Allocation

