Introduction to Caches

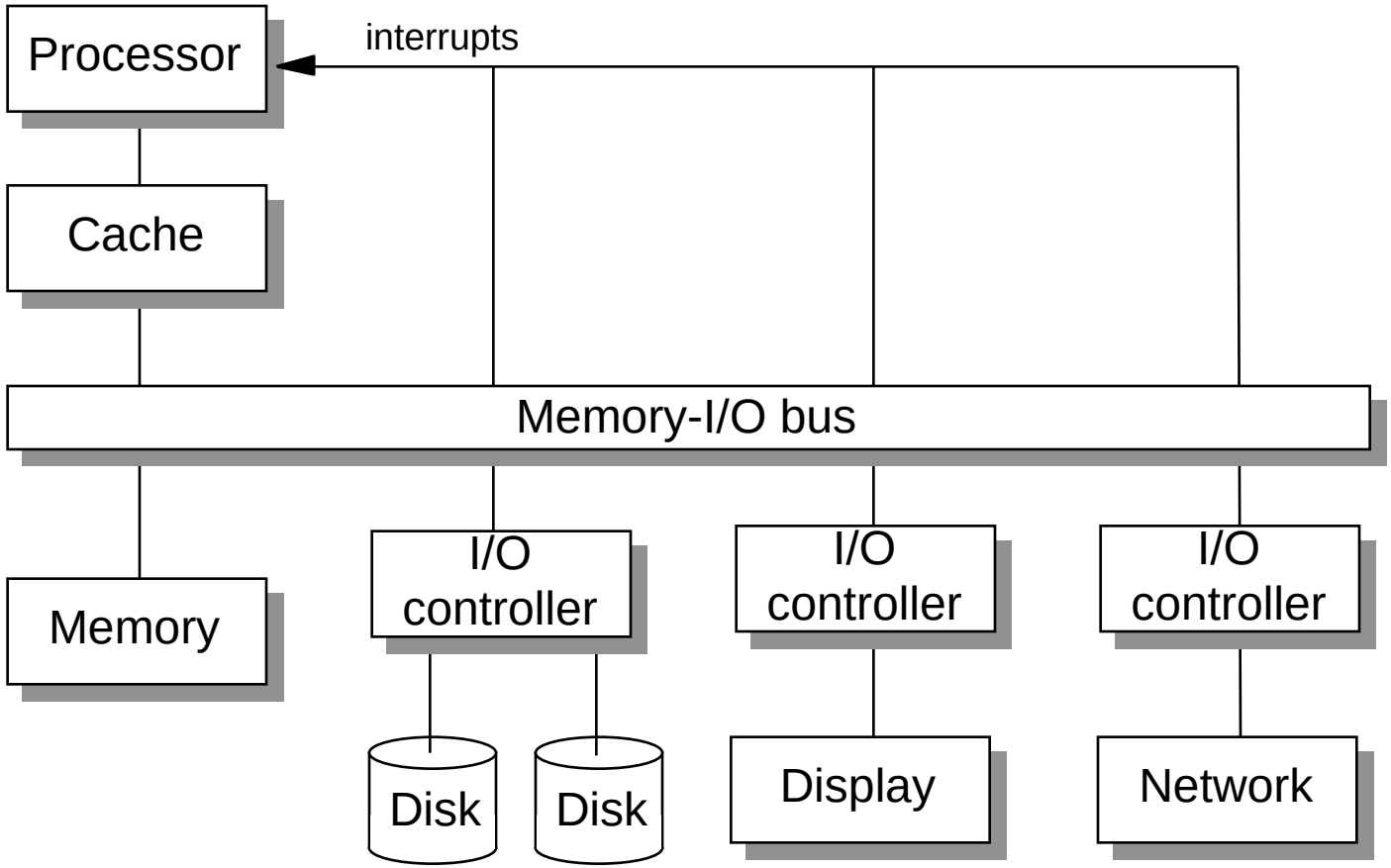
Randal E. Bryant

CS347 Lecture 7
February 3, 1998

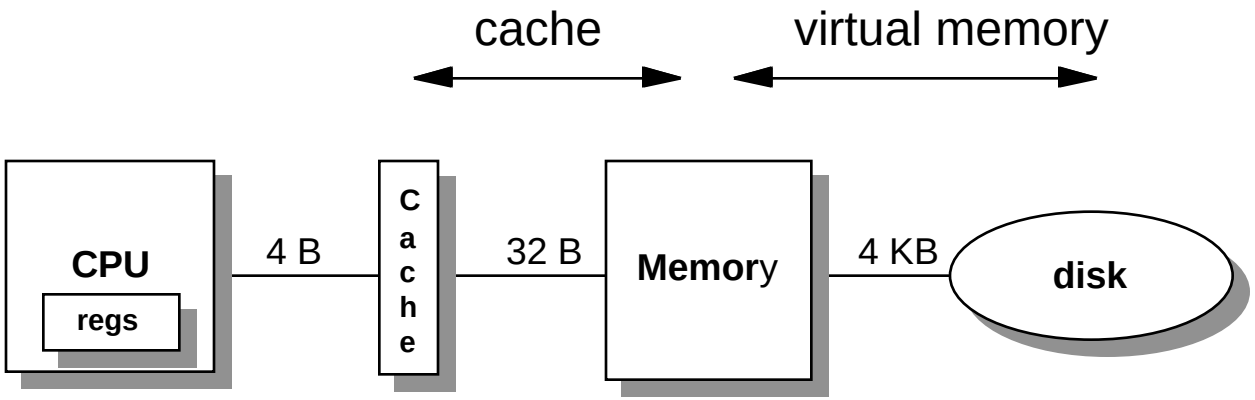
Topics

- memory hierarchy
- locality
- address spaces
- direct mapped caches
- associative caches

Computer System



Levels in a typical memory hierarchy



	register reference	cache reference	memory reference	disk memory reference
size:	200 B	32KB -- 4MB	128 MB	10GB
speed:	5 ns	6 ns	100 ns	10 ms
$/Mbyte:		$256/MB	$2/MB	$0.30/MB
block size:	4 B	16 B	4 KB	

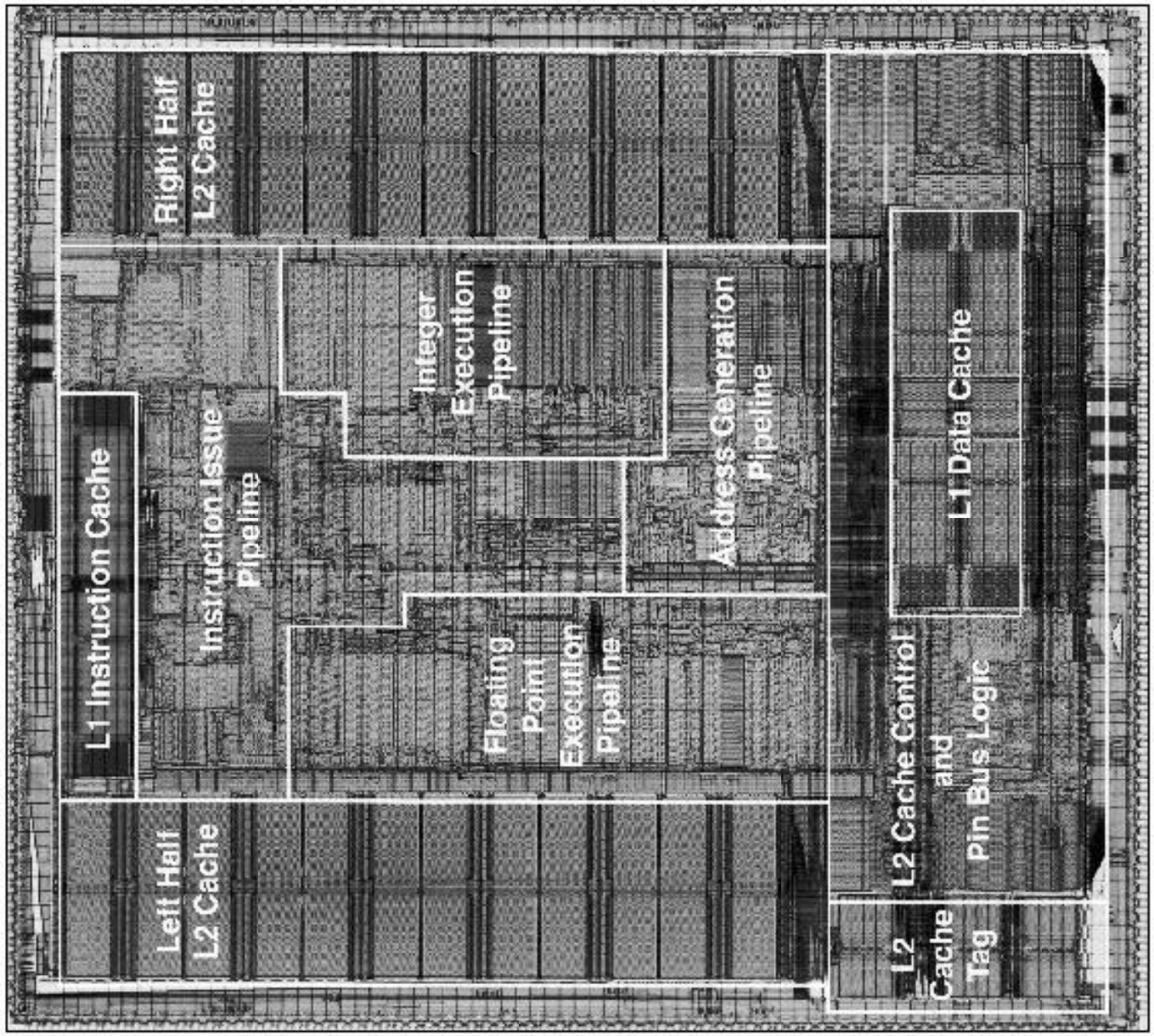
larger, slower, cheaper

Alpha 21164 Chip Photo

- Microprocessor Report 9/12/94

Caches:

- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history

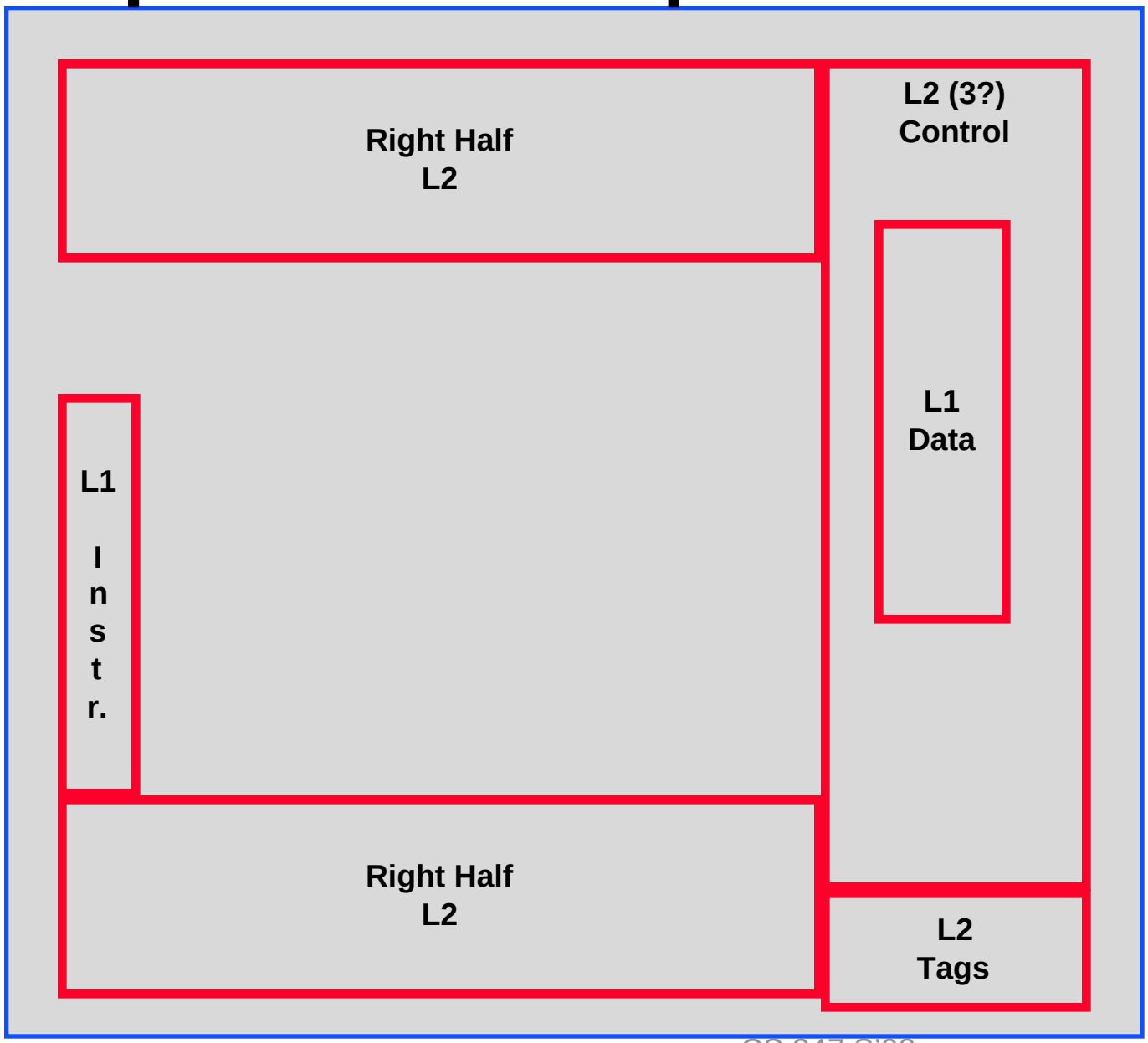


Alpha 21164 Chip Caches

- Microprocessor Report 9/12/94

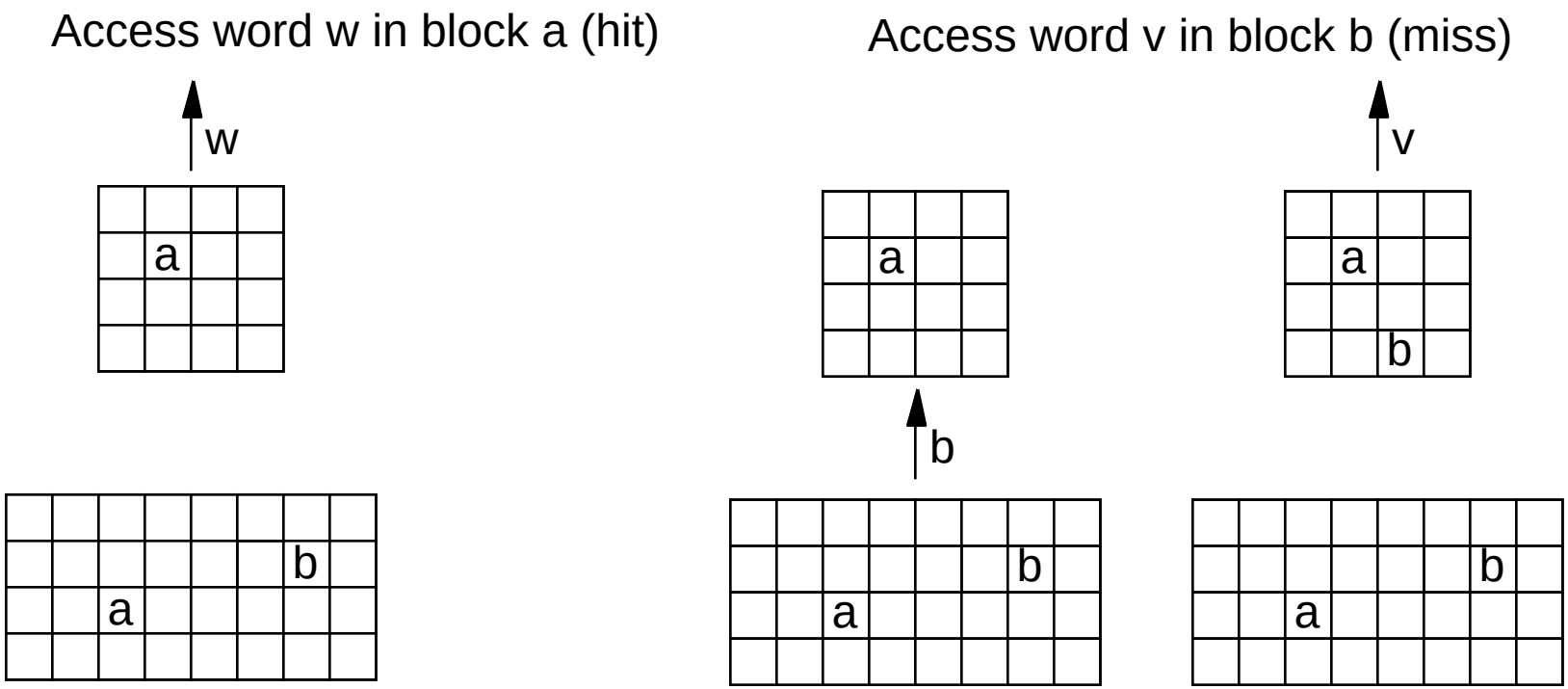
Caches:

- L1 data
- L1 instruction
- L2 unified
- TLB
- Branch history



Accessing data in a memory hierarchy

Between any two levels, memory divided into *blocks.*
Data moves between levels on demand, in block-sized chunks.
Upper-level blocks a subset of lower-level blocks.



Sources of Memory References

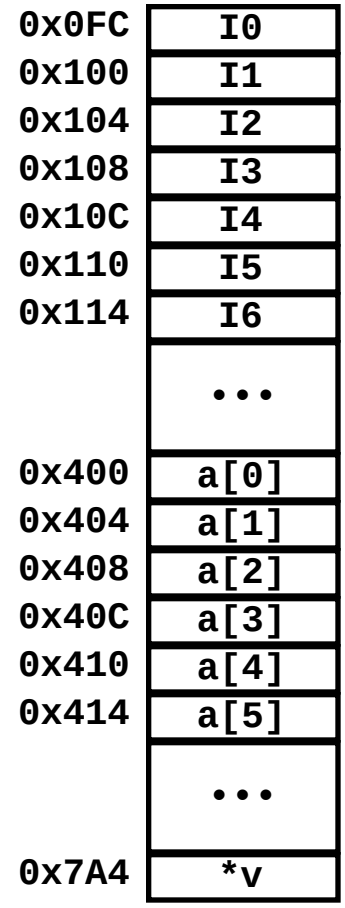
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

Abstract Version of Machine Code

```
I0:        sum  <-- 0
I1:        ap  <-- a
I2:        i   <-- 0
I3:        if (i >= n) goto done
i4: loop:  t   <-- *ap
I5:        sum <-- sum + t
I6:        ap  <-- ap + 4
I7:        i   <-- i + 1
I8:        if (i < n) goto loop
I9: done:  *v  <-- sum
```

- Memory addresses in bytes
- Each instruction & data word 4 bytes
- Instruction sequences & data arrays laid out as contiguous memory blocks

Memory Layout



Locality of reference

Principle of Locality

- **programs tend to reuse data and instructions near those they have used recently.**
- ***temporal locality:*** **recently referenced items are likely to be referenced in the near future.**
- ***spatial locality:*** **items with nearby addresses tend to be referenced close together in time.**

Locality in Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- **Data**
  - Reference array elements in succession (spatial)
- **Instruction**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

Key questions for caches

Q1: Where should a block be placed in the cache? (block placement)

Q2: How is a block found in the cache? (block identification)

Q3: Which block should be replaced on a miss? (block replacement)

Q4: What happens on a write? (write strategy)

Address spaces

An n-bit address defines an address space of 2^n items: $0,...,2^{n-1}$.

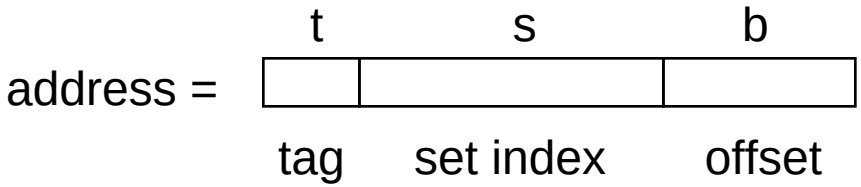
00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111

Address space for n=5

Partitioning address spaces

Key idea: partitioning the address bits partitions the address space.

In general, an address partitioned into sets of t (tag), s (set index), and b (block offset) bits, e.g.,



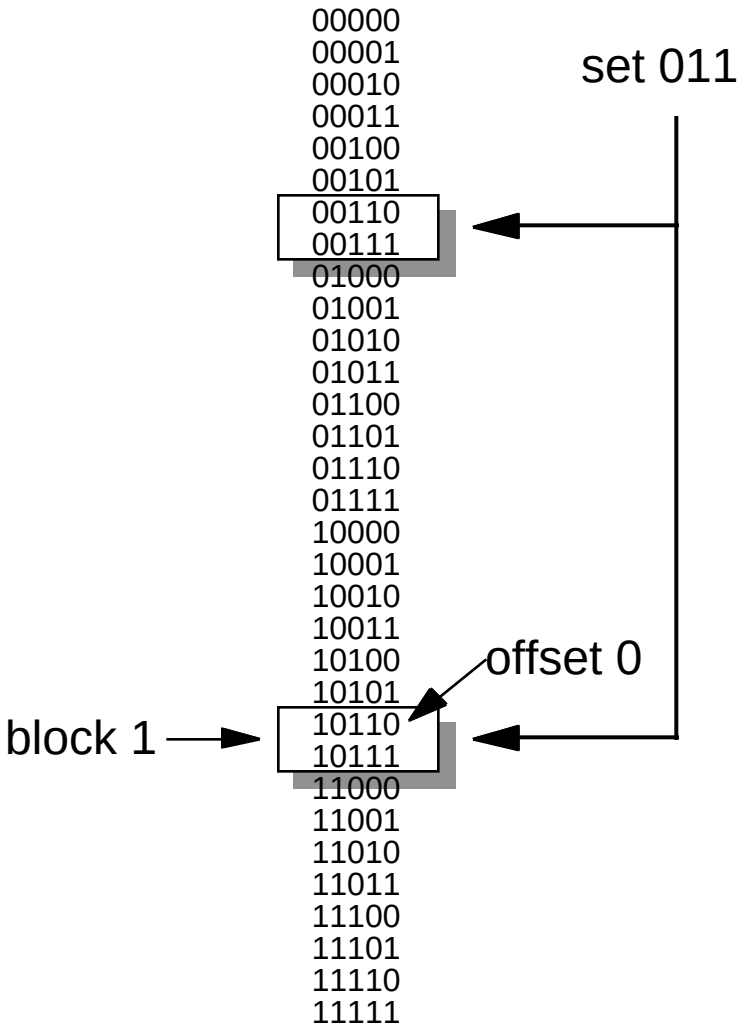
belongs to one of 2^s equivalence classes (sets), where each set consists of 2^t blocks of addresses, and each block consists of 2^b addresses.

The s bits uniquely identify an equivalence class.
The t bits uniquely identify each block in the equivalence class.
The b bits define the offset of an address within a block (block offset).

Partitioning address spaces

t=1	s=3	b=1
1	011	0

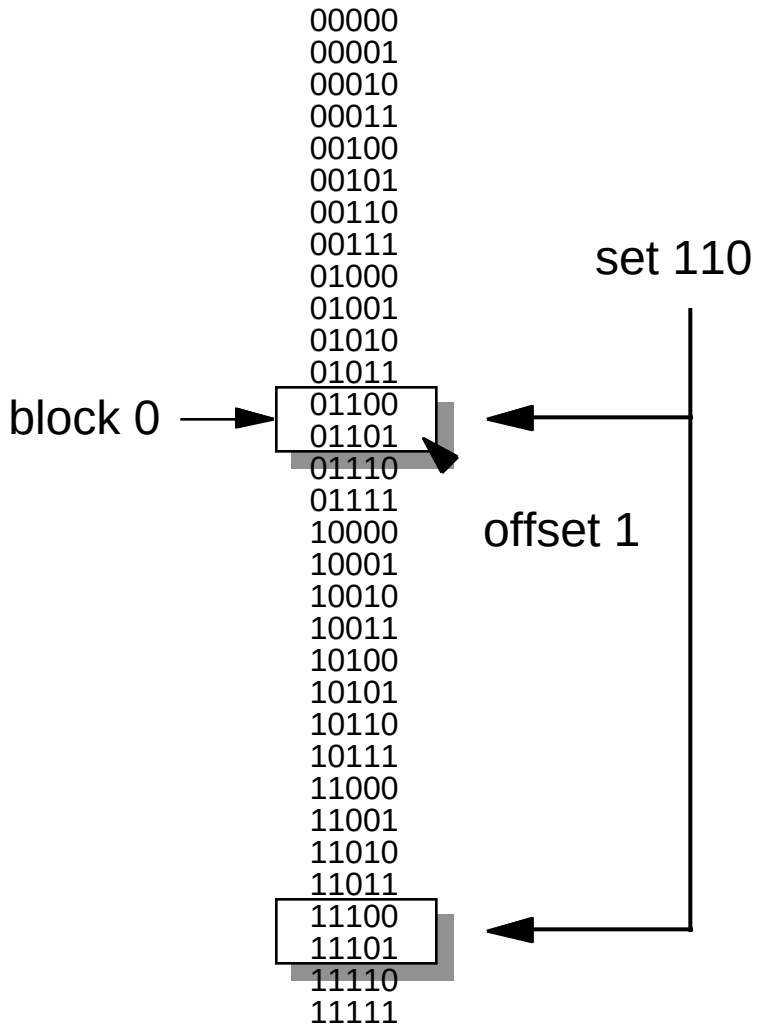
2^s = 8 sets of blocks
2^t = 2 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=1	s=3	b=1
0	110	1

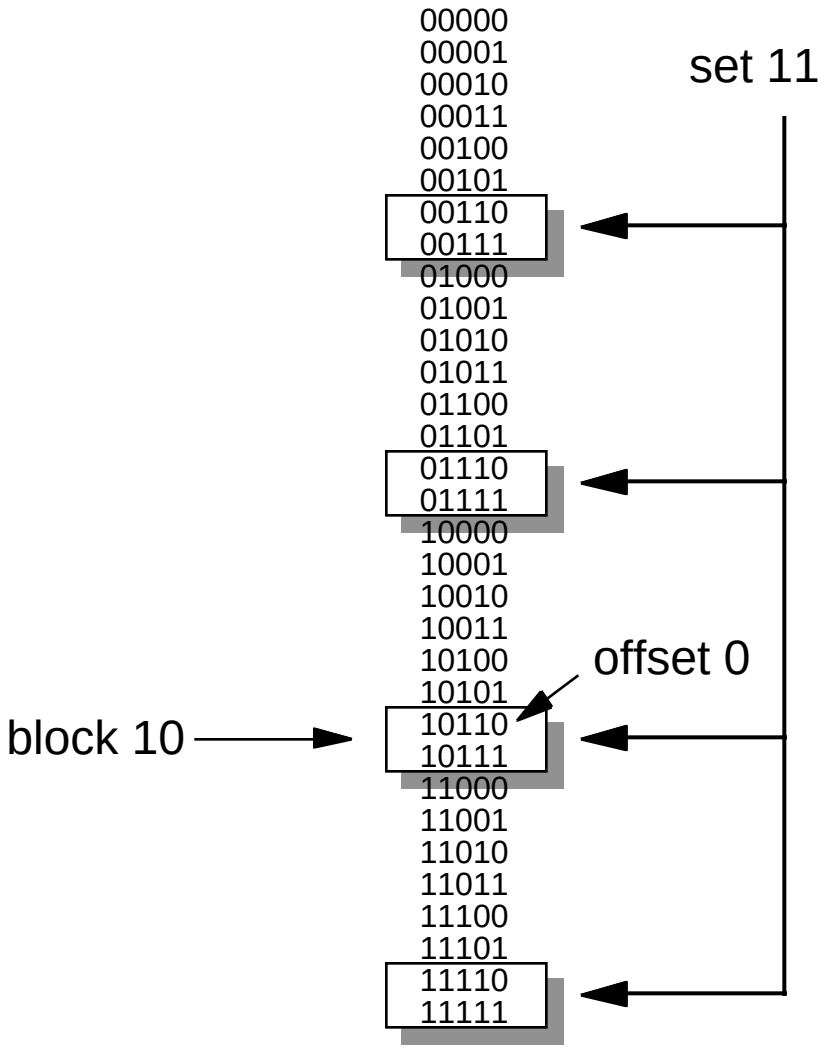
2^s = 8 sets of blocks
2^t = 2 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=2	s=2	b=1
10	11	0

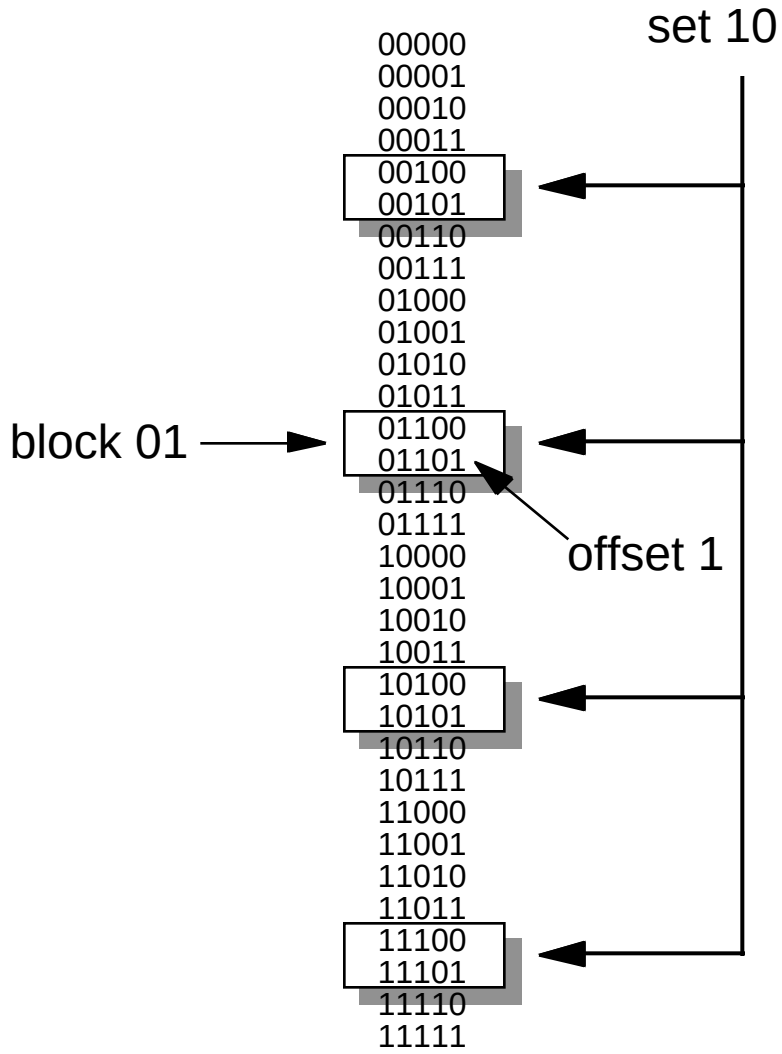
2^s = 4 sets of blocks
2^t = 4 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=2	s=2	b=1
01	10	1

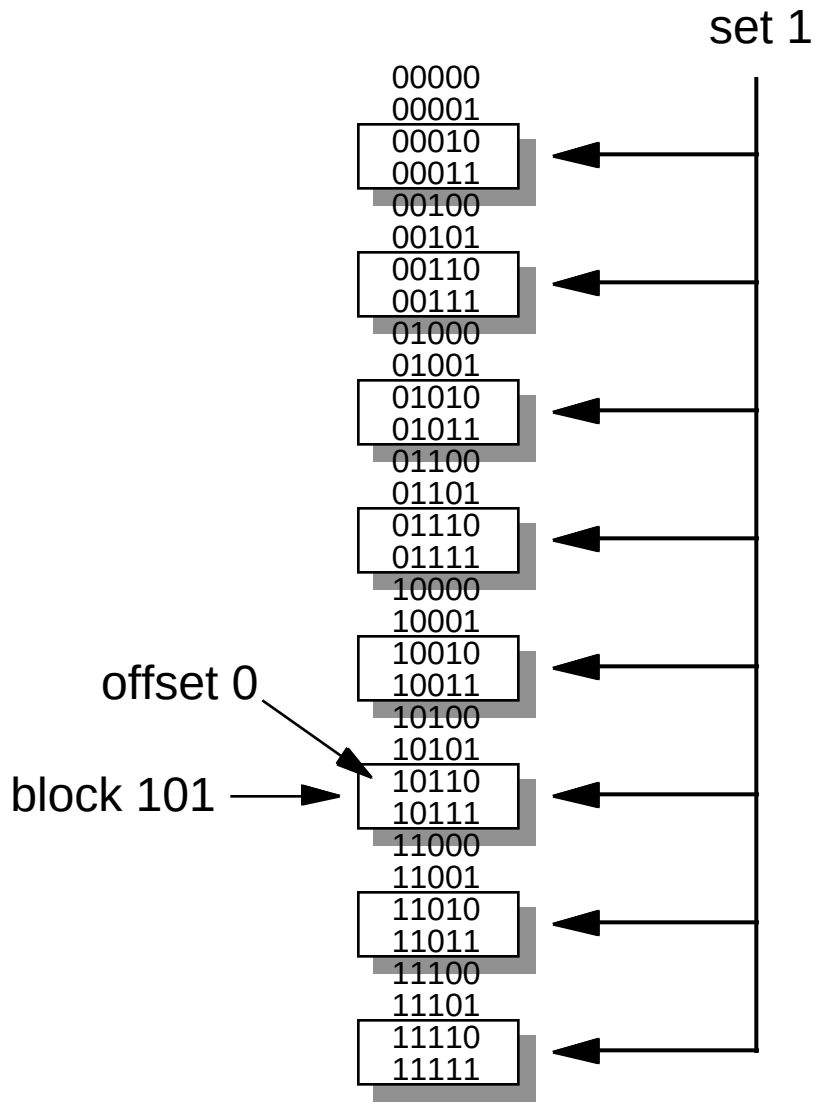
2^s = 4 sets of blocks
2^t = 4 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=3	s=1	b=1
101	1	0

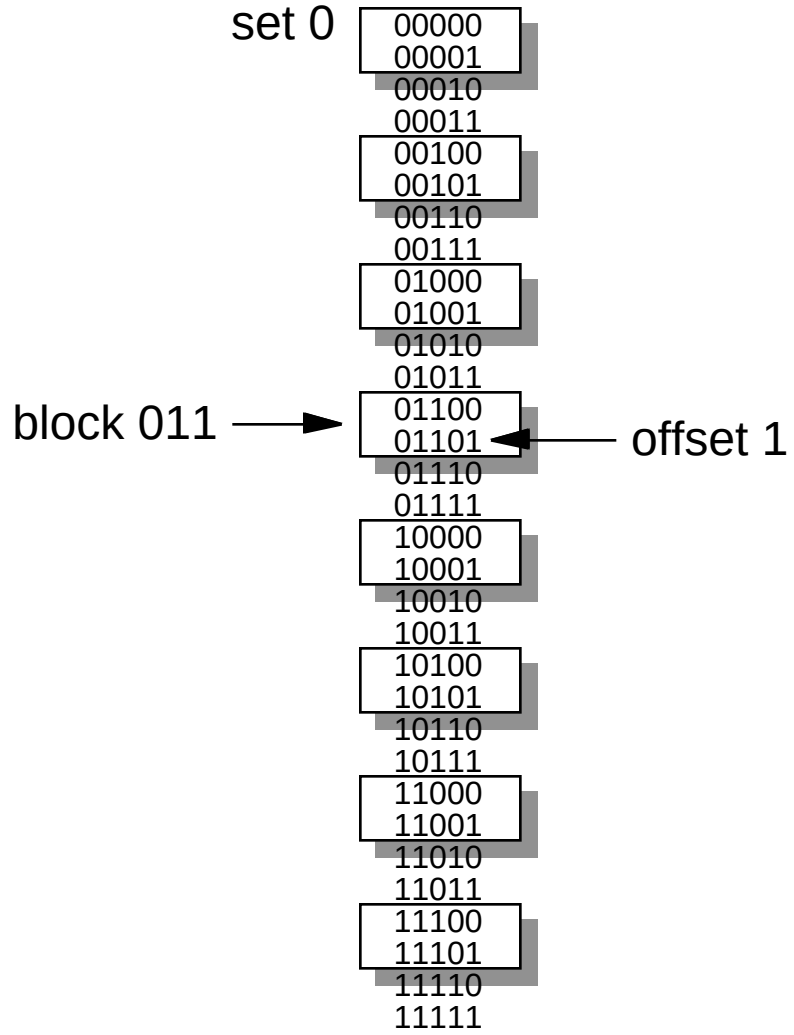
2^s = 2 sets of blocks
2^t = 8 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=3	s=1	b=1
011	0	1

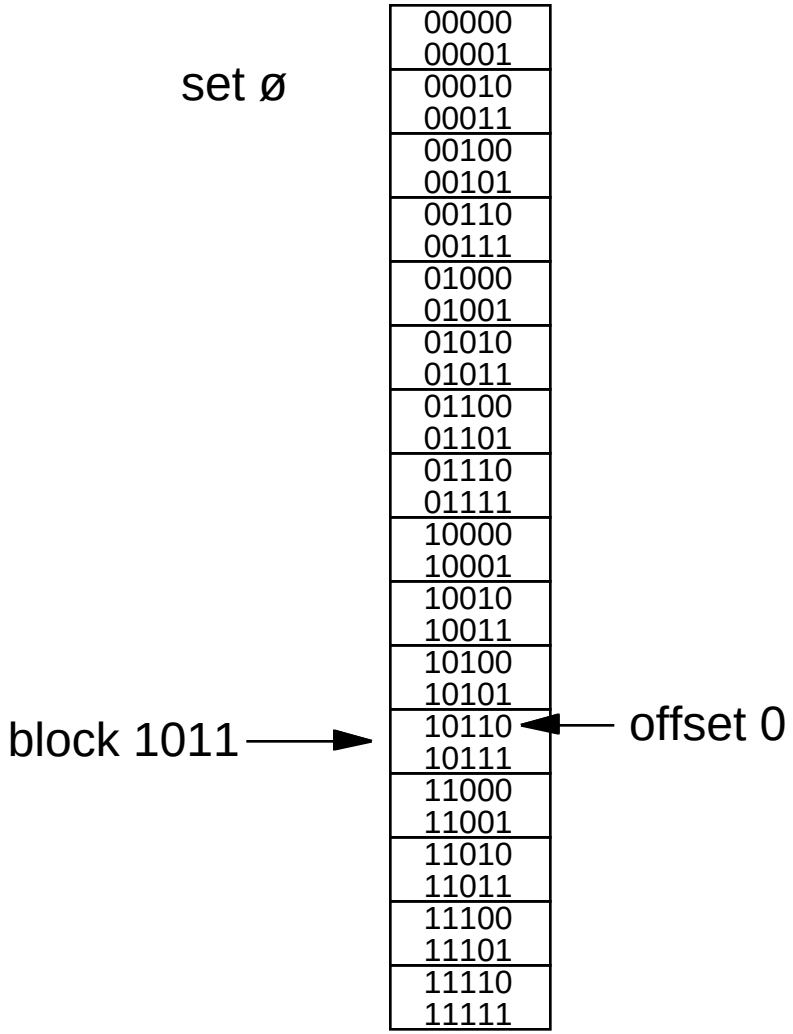
2^s = 2 sets of blocks
2^t = 8 blocks/set
2^b = 2 addresses/block.



Partitioning address spaces

t=4	s=0	b=1
1011		0

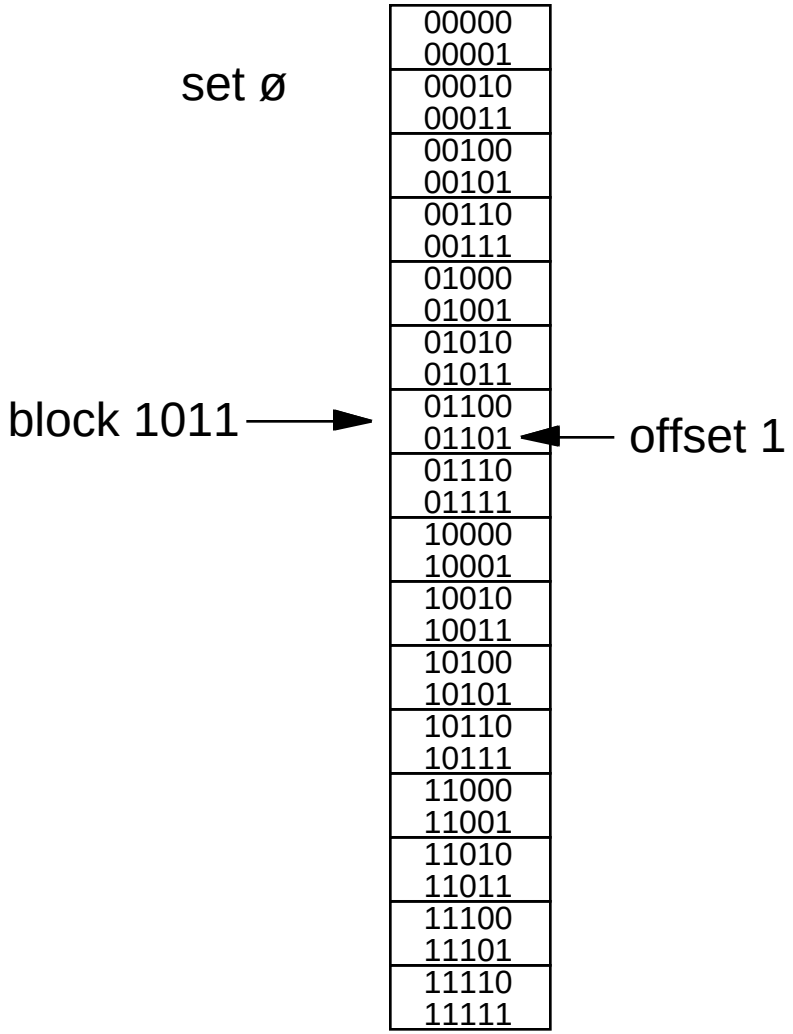
2^s = 1 set of blocks
2^t = 16 blocks/set
2^b = 2 addresses/block.



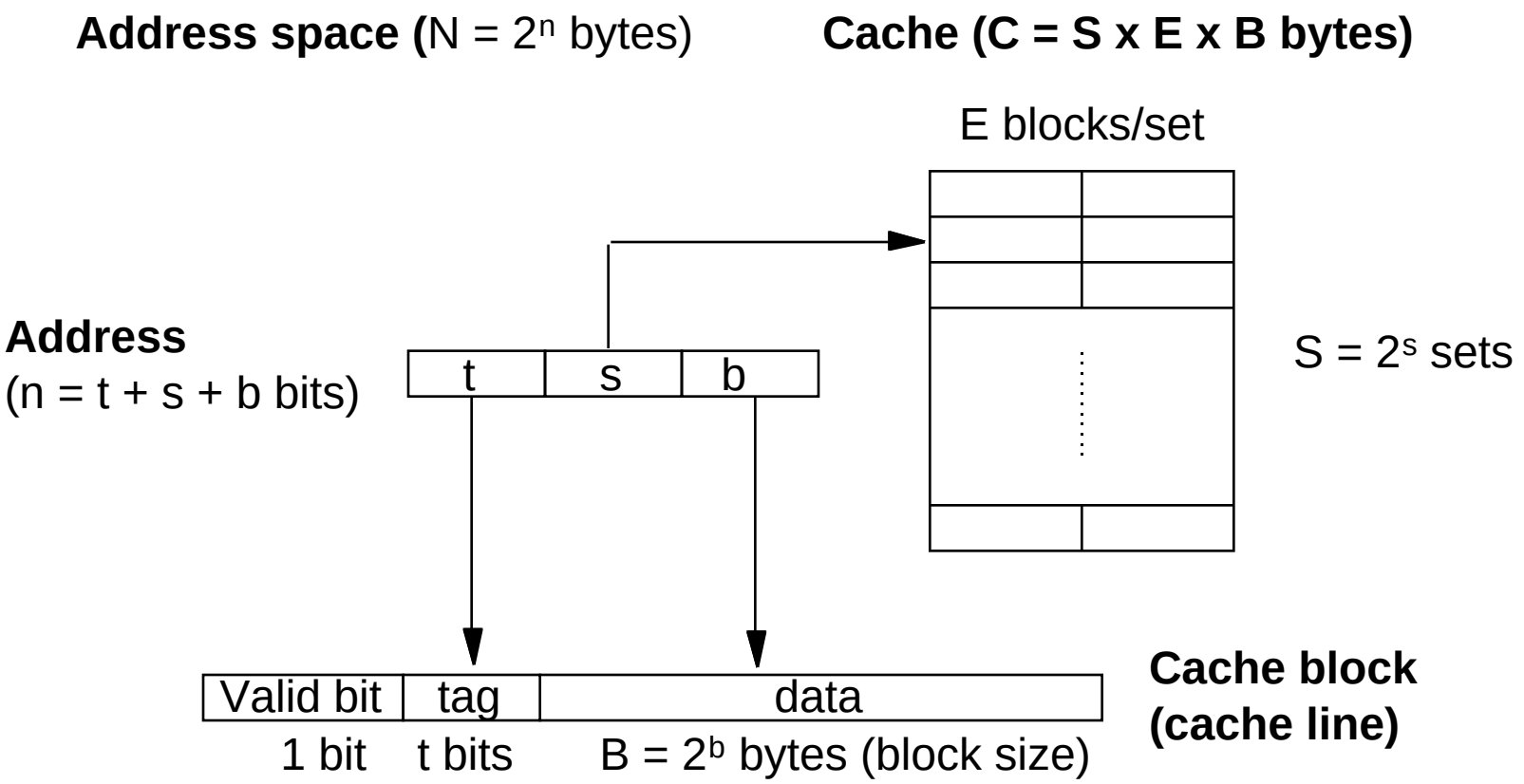
Partitioning address spaces

t=4	s=0	b=1
0110		1

2^s = 1 set of blocks
2^t = 16 blocks/set
2^b = 2 addresses/block.



Basic cache organization



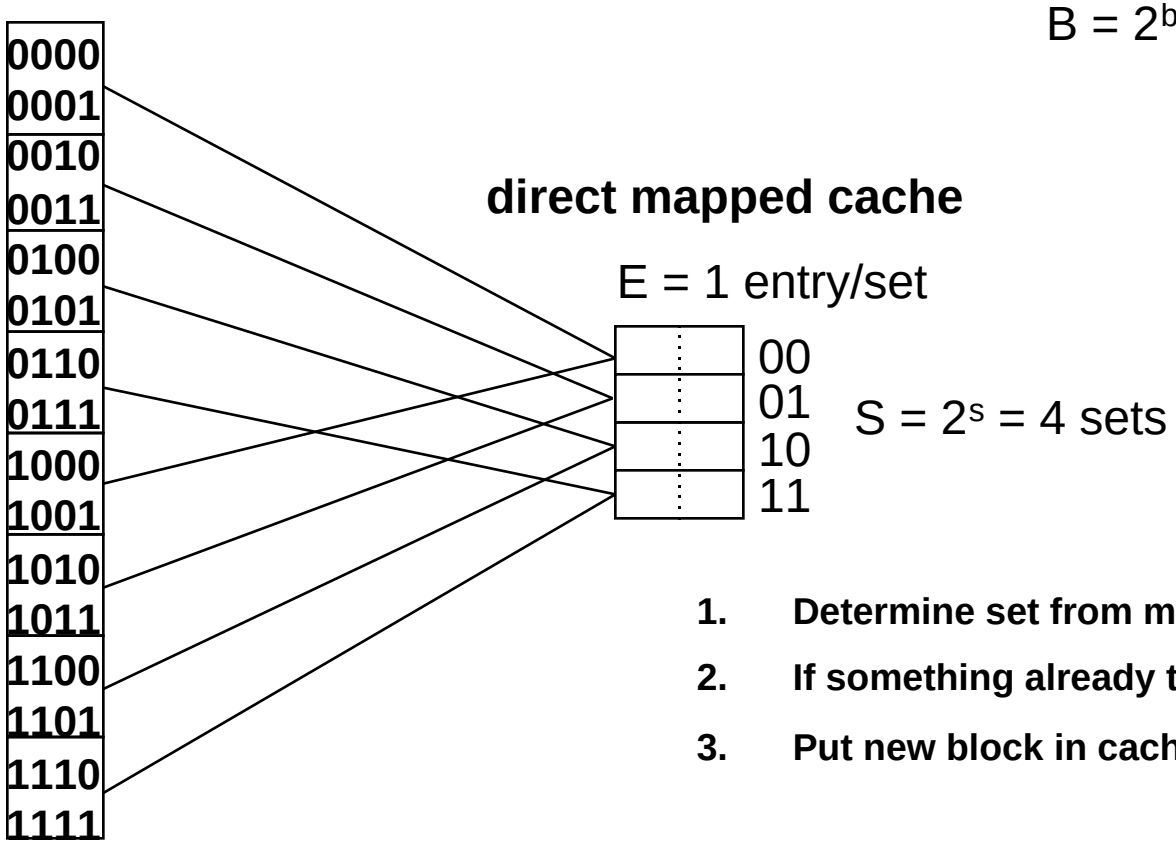
E: Describes *associativity*: how many blocks in set can reside in cache simultaneously

Direct mapped cache (E = 1)

N = 16 byte addresses (n=4)

t=1	s=2	b=1
x	xx	x

cache size:
C = 8 data bytes
line size:
B = 2^b = 2 bytes/line



1. **Determine set from middle bits**
2. **If something already there, knock it out**
3. **Put new block in cache**

Direct Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

N=16 byte addresses B=2 bytes/block S=4 sets E=1 entry/set

Address trace (reads):

0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

(1)

0 (miss)

v	tag	data

(2)

13 (miss)

v	tag	data

(3)

8 (miss)

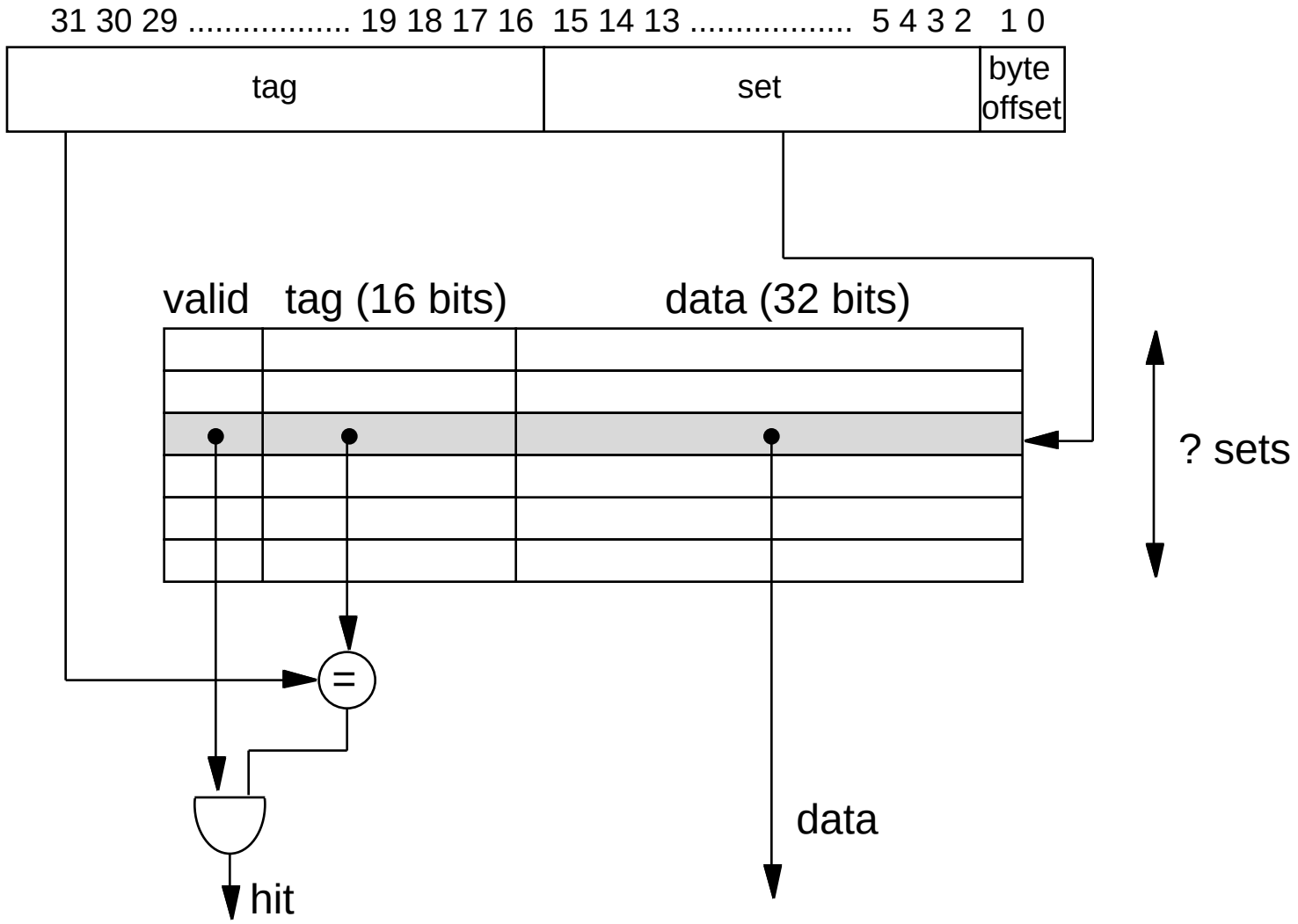
v	tag	data

(4)

0 (miss)

v	tag	data

Direct Mapped Cache Implementation (DECStation 3100)

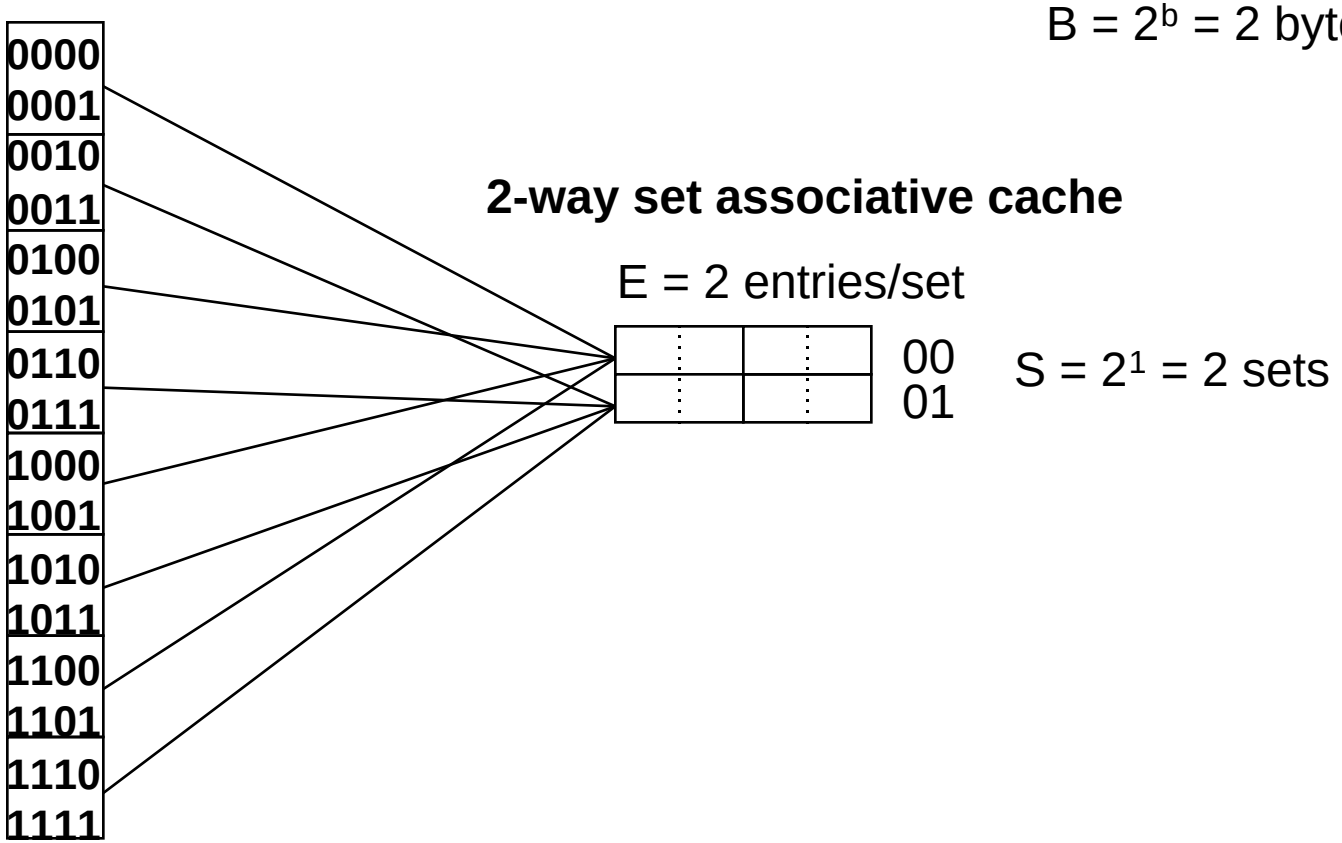


E-way Set-Associative Cache

N = 16 addresses (n=4)

t=2	s=1	b=1
xx	x	x

Cache size:
C = 8 data bytes
Line size:
B = 2^b = 2 bytes



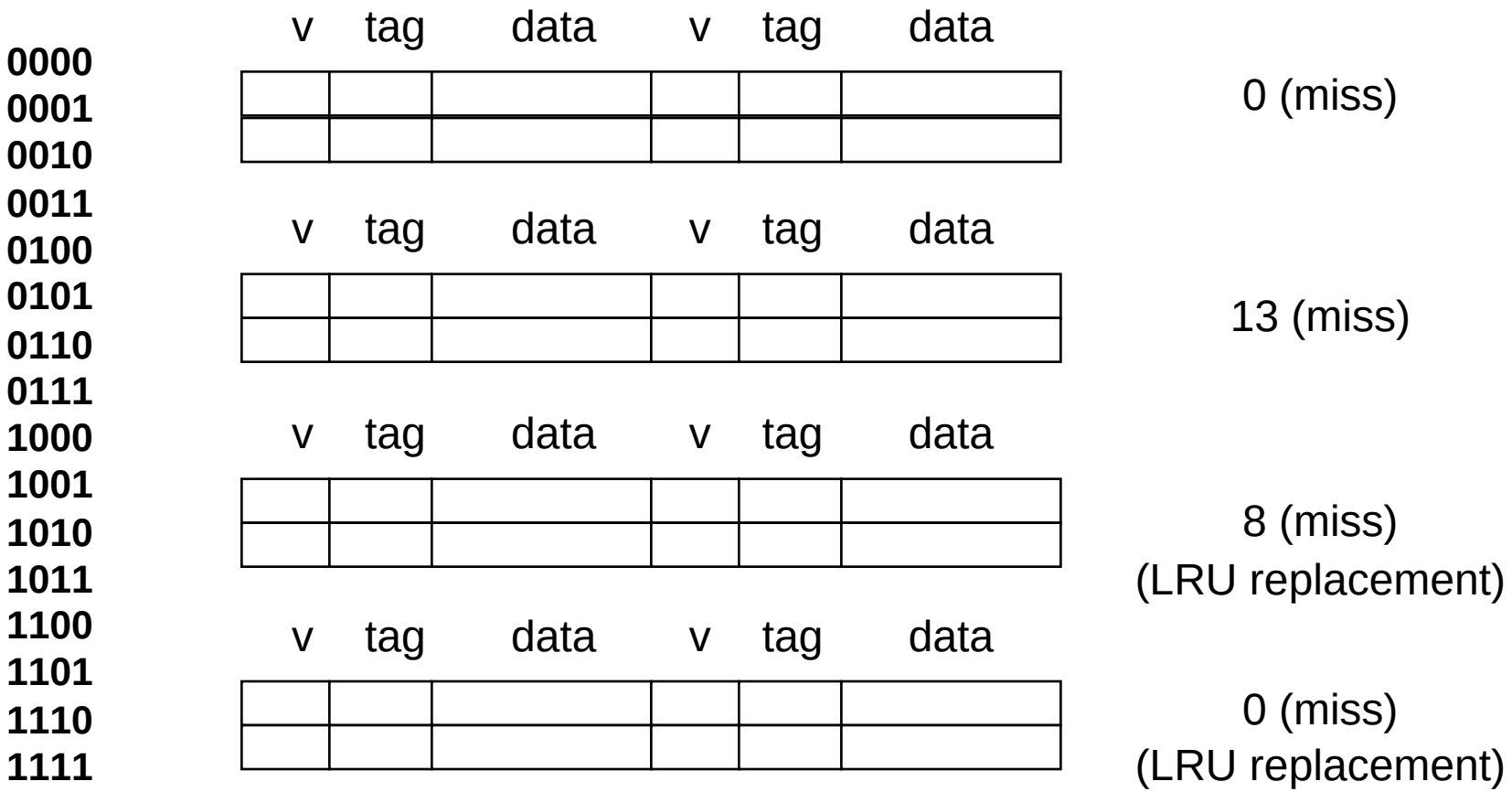
2-Way Set Associative Simulation

t=2	s=1	b=1
xx	x	x

N=16 addresses B=2 bytes/line S=2 sets E=2 entries/set

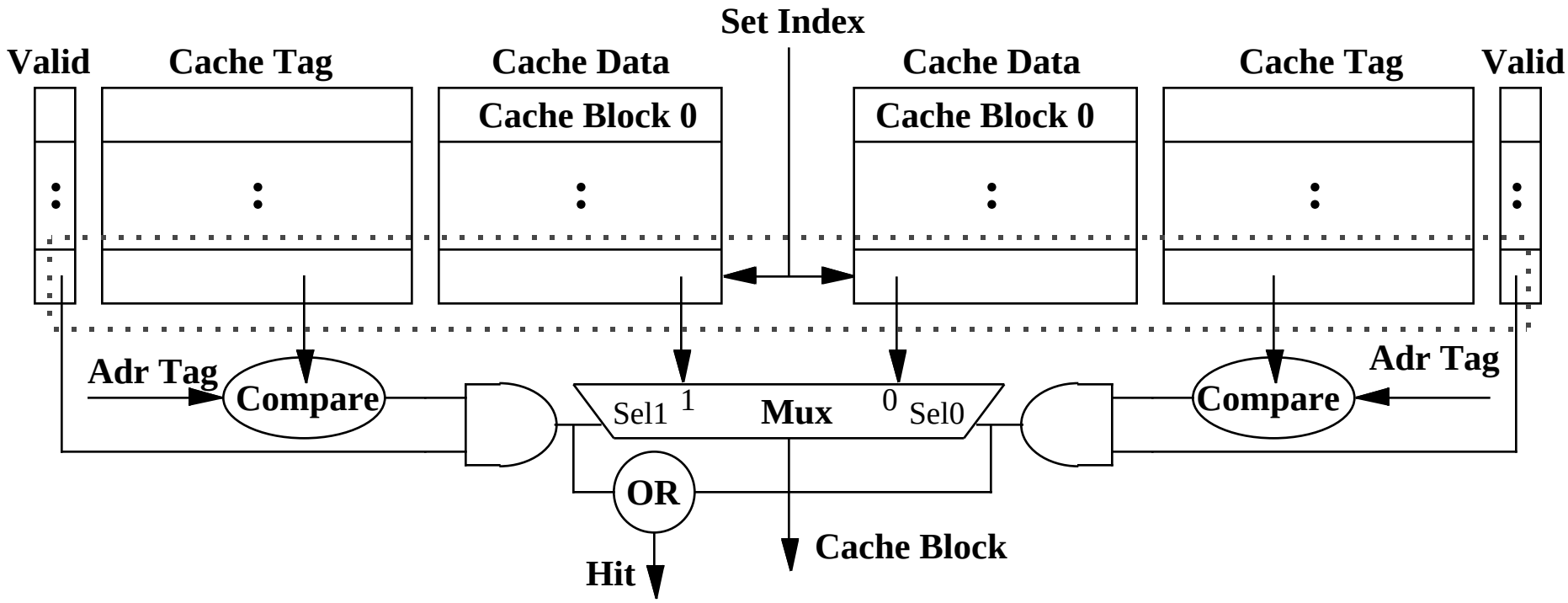
Address trace (reads):

0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]



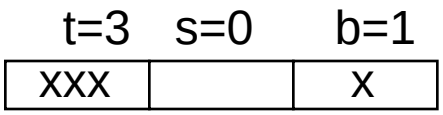
Two-Way Set Associative Cache Implementation

- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result

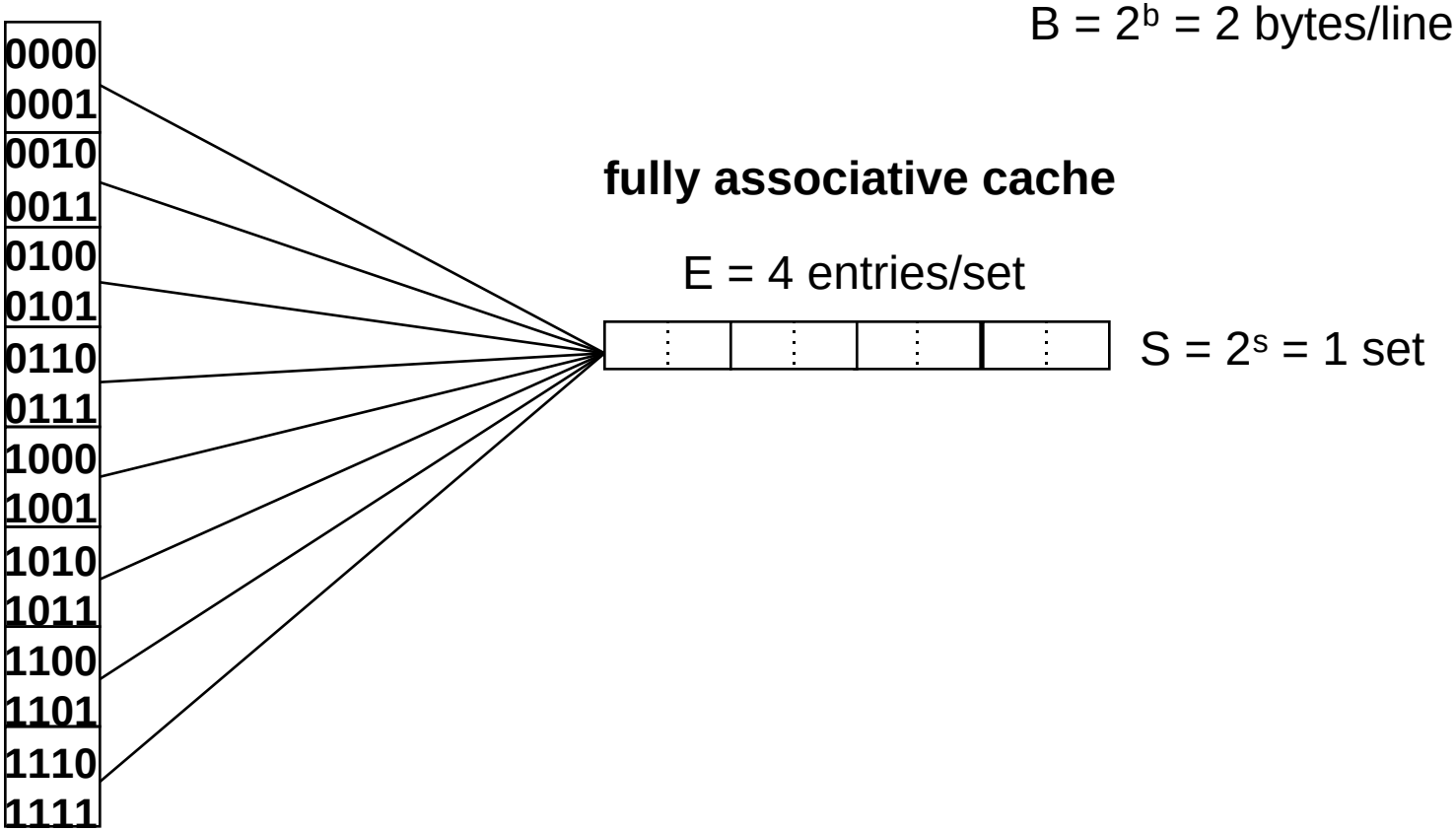


Fully associative cache (E = C/B)

N = 16 addresses (n=4)



cache size:
C = 8 data bytes
line size:
B = 2^b = 2 bytes/line



Fully associative cache simulation

N=16 addresses B=2 bytes/line S=1 sets E=4 entries/set
Address trace (reads):
0 [0000] 1 [0001] 13 [1101] 8 [1000] 0 [0000]

t=3	s=0	b=1
xxx		x

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

0 (miss)

(1)

v	tag	data

set ø

13 (miss)

(2)

v	tag	data

8 (miss)

(3)

v	tag	data

Replacement Algorithms

- ***When a block is fetched, which block in the target set should be replaced?***

Usage based algorithms:

- **Least recently used (LRU)**
  - replace the block that has been referenced least recently
  - hard to implement

Non-usage based algorithms:

- **First-in First-out (FIFO)**
  - treat the set as a circular queue, replace block at head of queue.
  - easy to implement
- **Random (RAND)**
  - replace a random block in the set
  - even easier to implement

Implementing RAND and FIFO

FIFO:

- **maintain a modulo E counter for each set.**
- **counter in each set points to next block for replacement.**
- **increment counter with each replacement.**

RAND:

- **maintain a single modulo E counter.**
- **counter points to next block for replacement in any set.**
- **increment counter according to some schedule:**
  - each clock cycle,
  - each memory reference, or
  - each replacement anywhere in the cache.

Implementing LRU

Create an ExE bit matrix for each set (only use E(E-1) bits.)
When block i is referenced, set row i and clear column i.
The LRU block is the row with all zeros.

All other blocks have been referenced more recently than this one

Example trace (E=4): 1 2 3 4 3 2 1

1

	1	1	1
0		0	0
0	0		0
0	0	0	

2

	0	1	1
1		1	1
0	0		0
0	0	0	

3

	0	0	1
1		0	1
1	1		1
0	0	0	

4

	0	0	0
1		0	0
1	1		0
1	1	1	

3

	0	0	0
1		0	0
1	1		1
1	1	0	

2

	0	0	0
1		1	1
1	0		1
1	0	0	

1

	1	1	1
0		1	1
0	0		1
0	0	0	

Setting Row

My reference most recent

Clearing Column

I was referenced after you were

Write Strategies

Write Policy

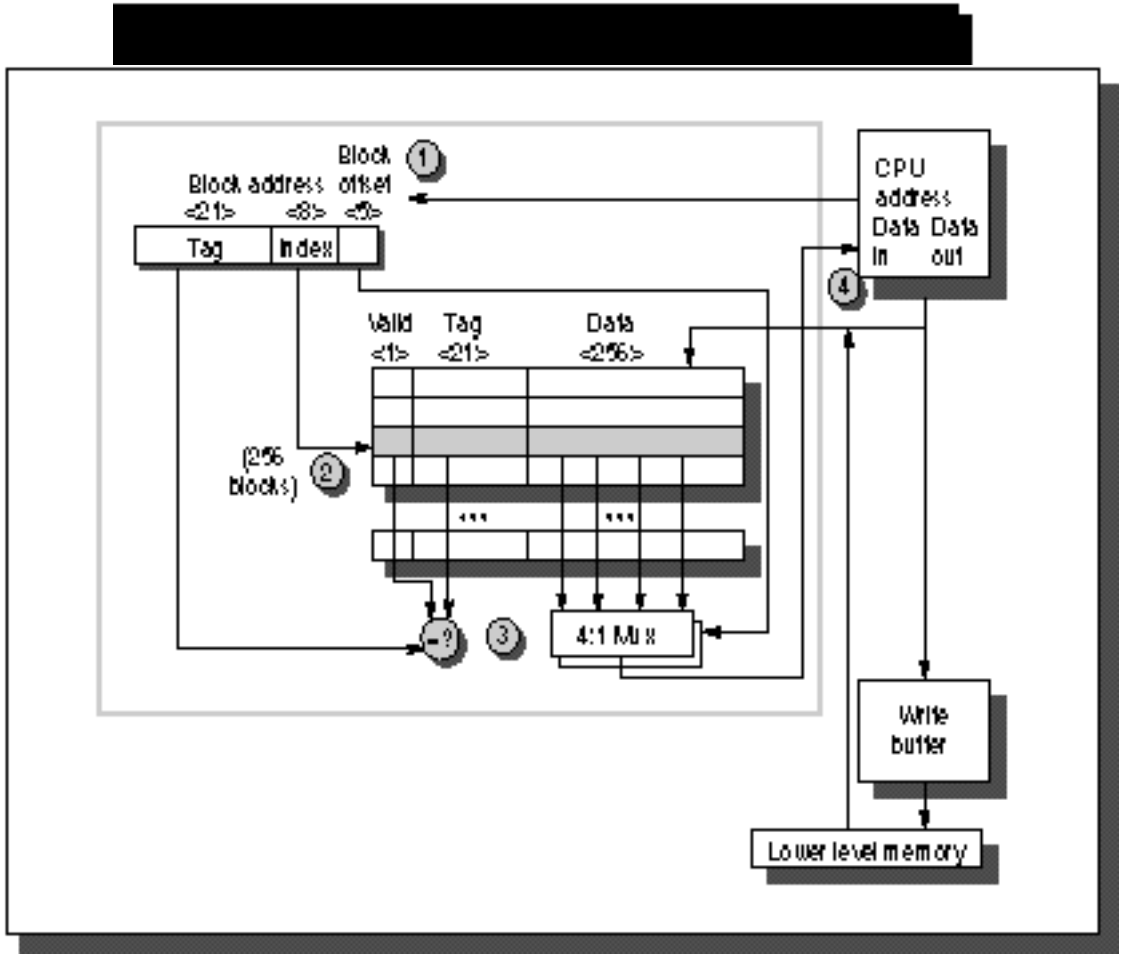
- **What happens when processor writes to the cache?**
- ***write through***
  - information is written to the block in cache *and* memory.
  - memory always consistent with cache
  - Can overwrite cache entry
- ***write back***
  - information is written only block in cache. Modified block written to memory only when it is replaced.
  - requires a dirty bit for each block
    - » To remove dirty block from cache, must write back to main memory
  - memory not always consistent with cache

Write Buffer

- **Common optimization for write-through caches**
- **Overlaps memory updates with processor execution**

Alpha AXP 21064 direct mapped data cache

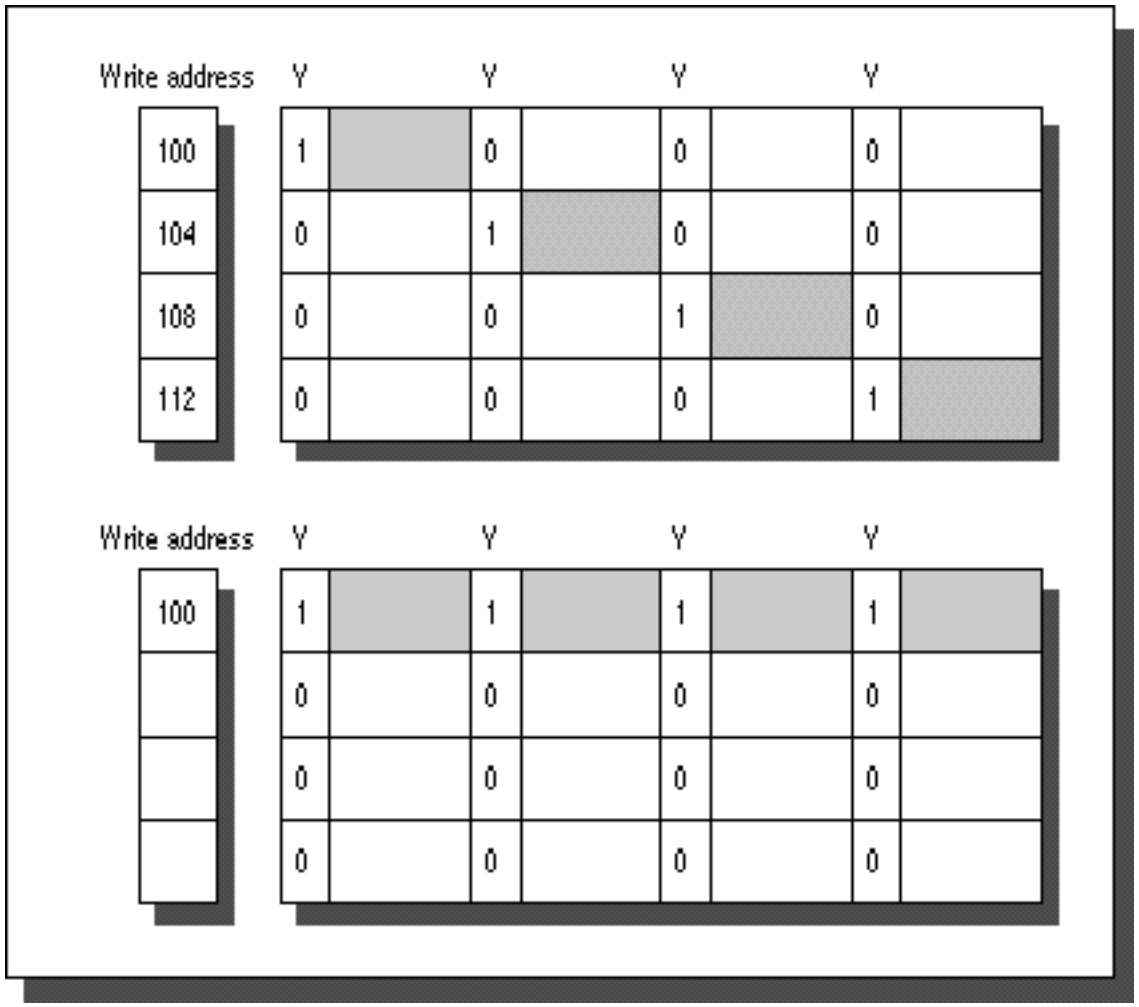
34-bit address
256 blocks
32-bytes/block



Write merging

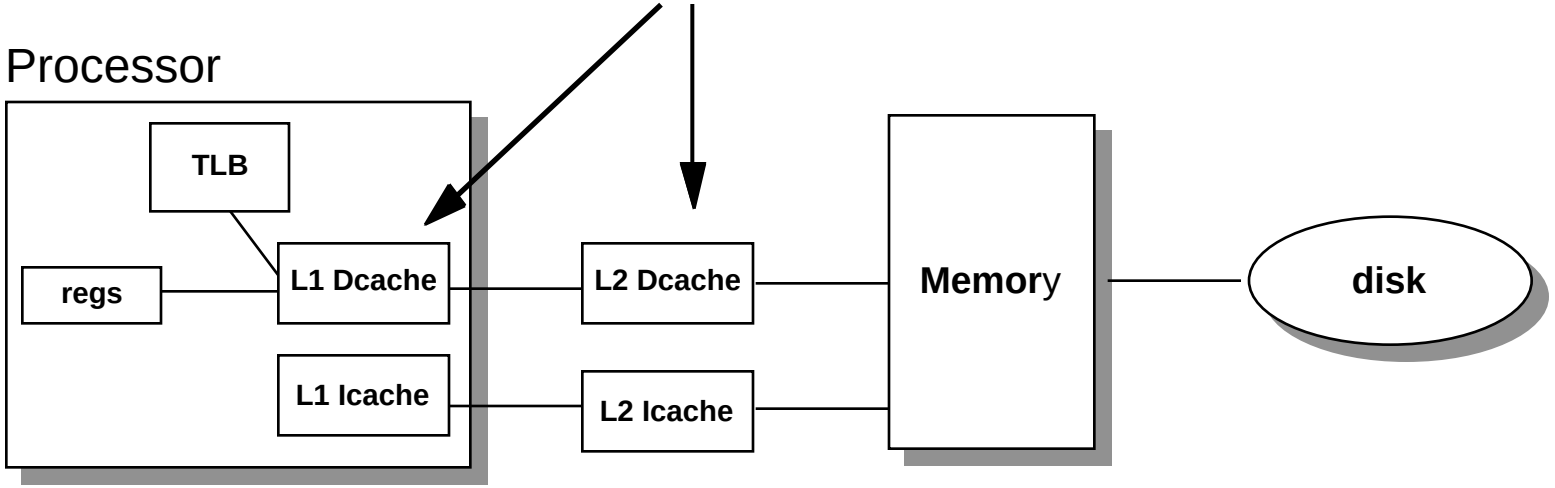
A write buffer that does not do write merging

A write buffer that does write merging



Multi-level caches

Can have separate Icache and Dcache or *unified* Icache/Dcache

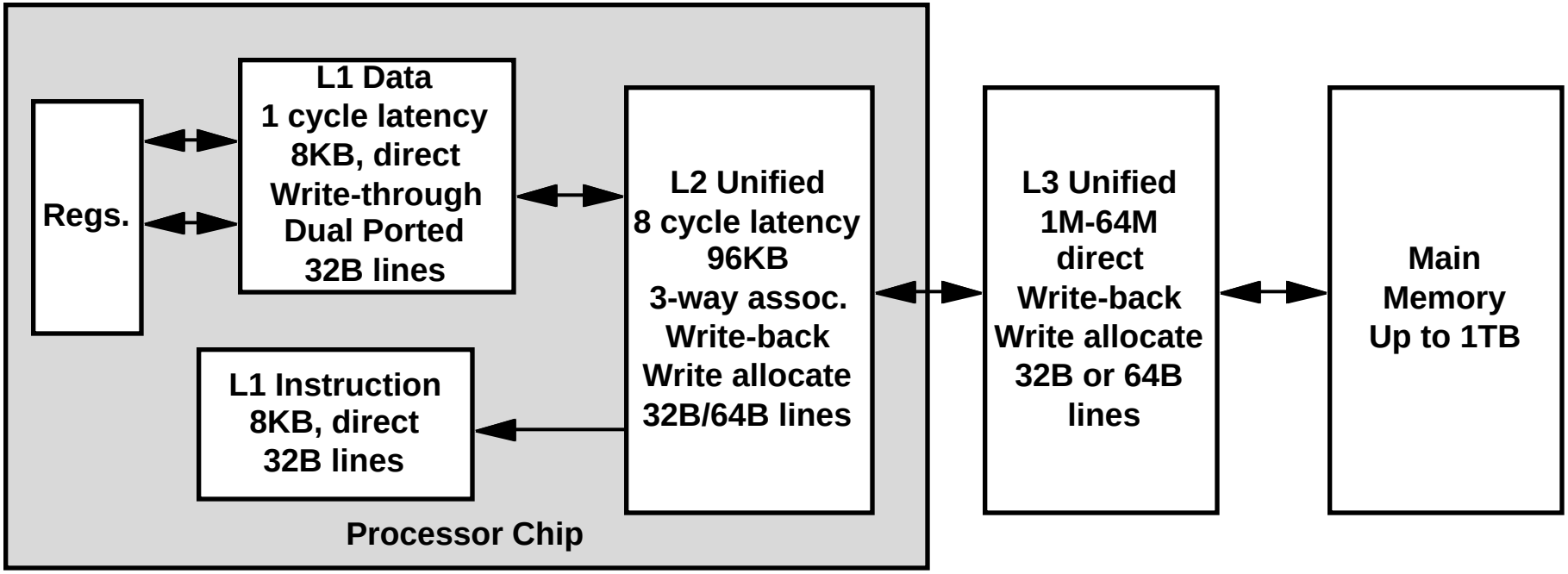


size:	200 B	8 KB	1M SRAM	128 MB DRAM	10 GB
speed:	5 ns	5 ns	6 ns	100 ns	10 ms
$/Mbyte:			$256/MB	$2/MB	$0.30/MB
block size:	4 B	16 B	32 KB	4 KB	

larger, slower, cheaper

larger block size, higher associativity, more likely to write back

Alpha 21164 Hierarchy



- **Improving memory performance was main design goal**
- **Earlier Alpha's CPUs starved for data**