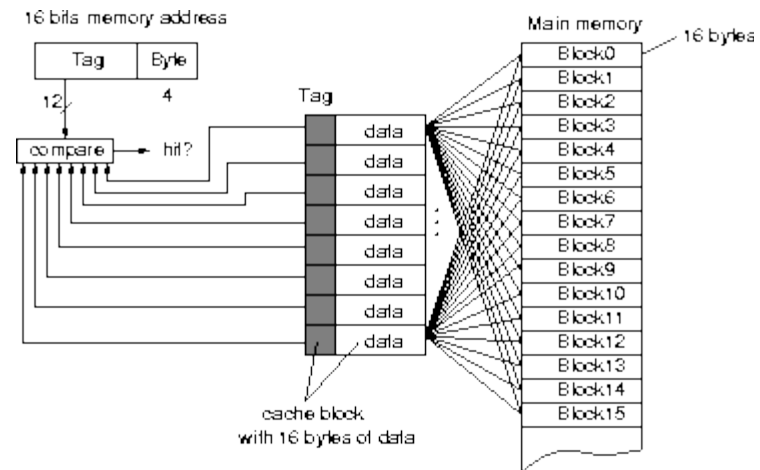


# cse5441 - parallel computing

## cache management part two



# thrashing

<u>example</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
6	32	1024	32	1	32	22	5	5

```
let @x[256] = AAAA0000
    @y[256] = AAAA0400
```

sizeof( float ) = 4 bytes

sum is in a register

```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum += x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000  
 set bits = 00 000

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000  
 set bits = 00 000

# set-associative cache

example

example

m

C

B

E

S

t

s

b

7

32

2048

32

2

32

22

5

5

```
let @x[256] = AAAA0000
    @y[256] = AAAA0400
```

sizeof( float ) = 4 bytes

sum is in a register

```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum += x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000

set bits =

00 000

fills 1<sup>st</sup> line in set

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000

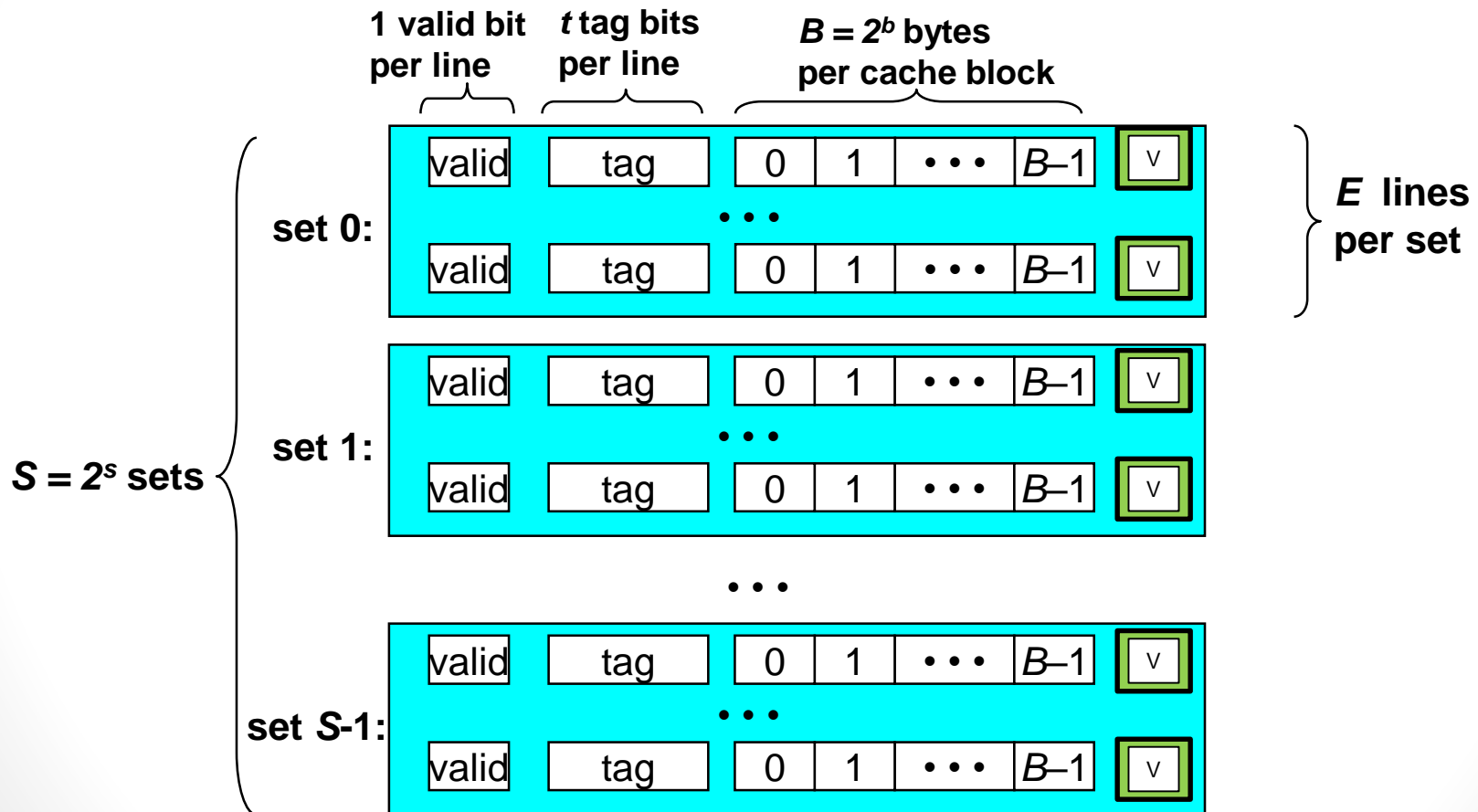
set bits =

00 000

fills 2<sup>nd</sup> line in set

# set-associative cache

structure

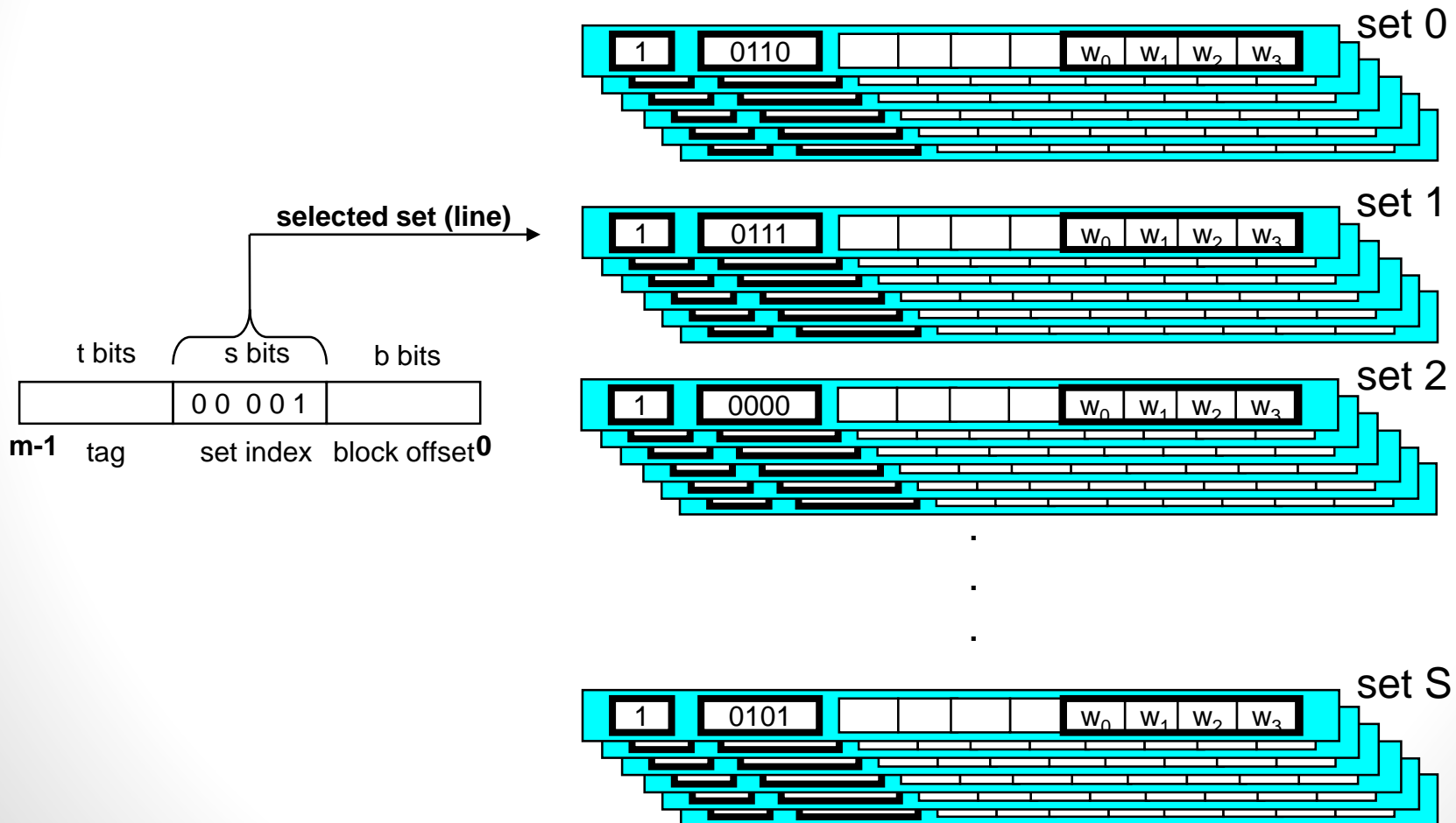


Cache size:  $C = B \times E \times S$  data bytes

# set - associative cache

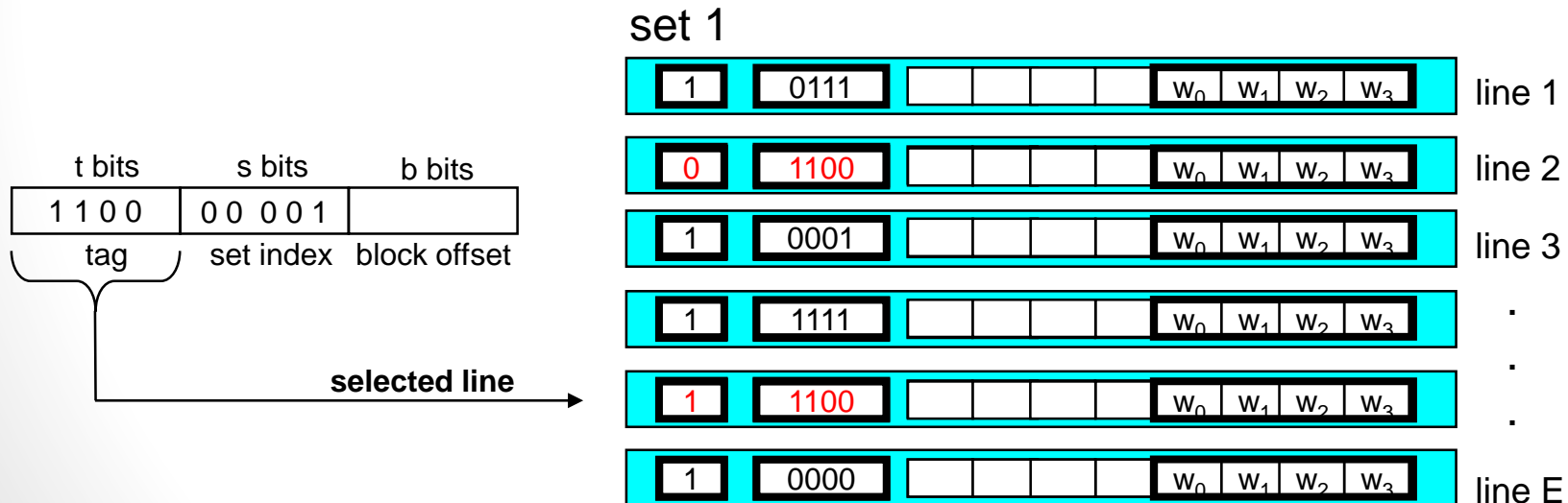
## set selection

- Use the set index bits to determine the set of interest.



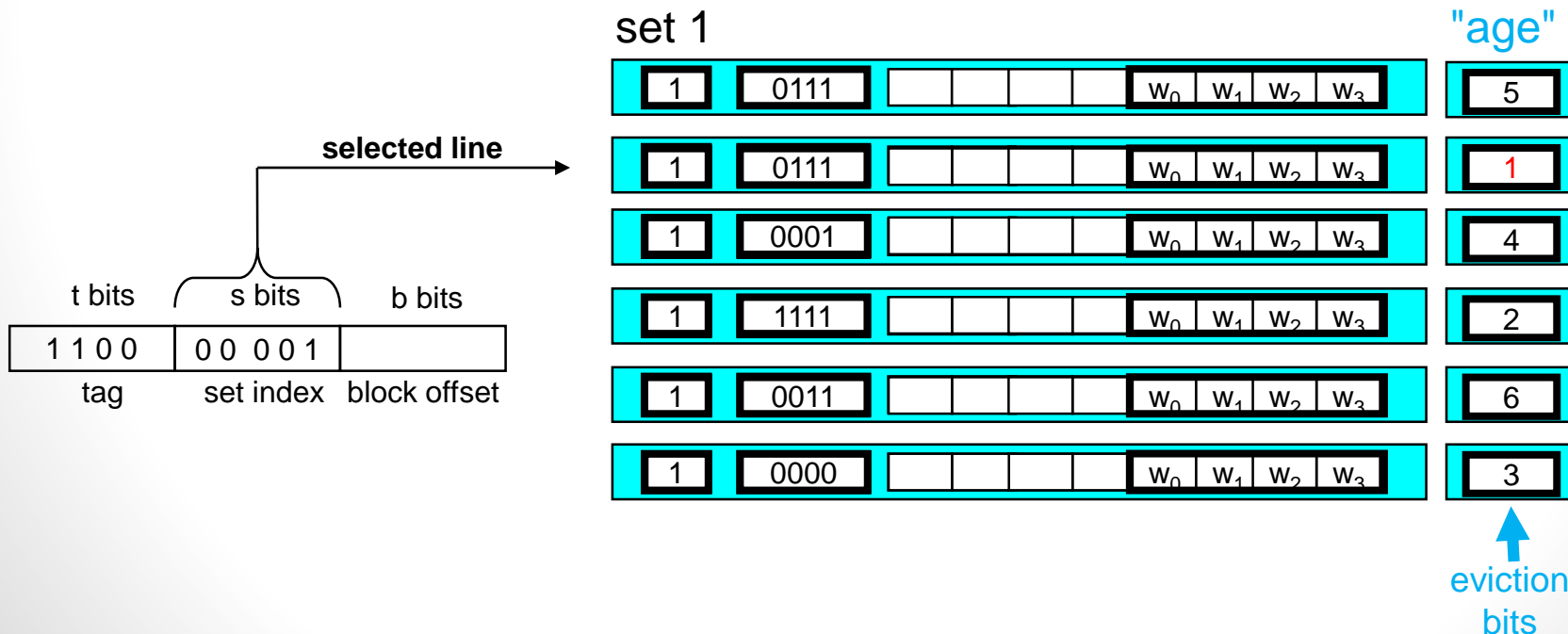
# set - associative cache line selection on read hit

- scan each cache line in set
  - if valid bit is set
    - if tag matches
      - **cache hit**



# set - associative cache line selection on read miss

- scan each cache line in set, determine is a miss
- retrieve from slower memory
- scan for empty line
  - if full, pick a loser



# LRU / LFU

LRU

sequential  
access

							age
1	0111					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	006
1	0101					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	005
1	0001					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	004
1	1111					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	003
1	0011					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	0000					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	001

LFU

equivalent  
access

							access count
1	0111					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	0101					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	0001					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	1111					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	0011					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002
1	0000					w <sub>0</sub> w <sub>1</sub> w <sub>2</sub> w <sub>3</sub>	002



# it's your turn . . .

problem      m      C      B      E      S      t      s      b

1      64      4096      32      4      32

```
let @x[1024] = 0...0AAAA0000
    @y[1024] = 0...0AAAA1000
```

`sizeof( float )` = 4 bytes

sum is in a register  
LRU eviction

```
float dotprod (float x[1024], float y[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum += x[ i ] * y[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?  
what is the hit rate for this loop?  
what is in the cache at the end of loop execution?

# set-associative cache

drawbacks

- $\text{sum} += x[i] * y[i] * z[i]$
- cache size
- where to populate new line
- what if the cache is full?
- time to determine a hit

# fully associative cache

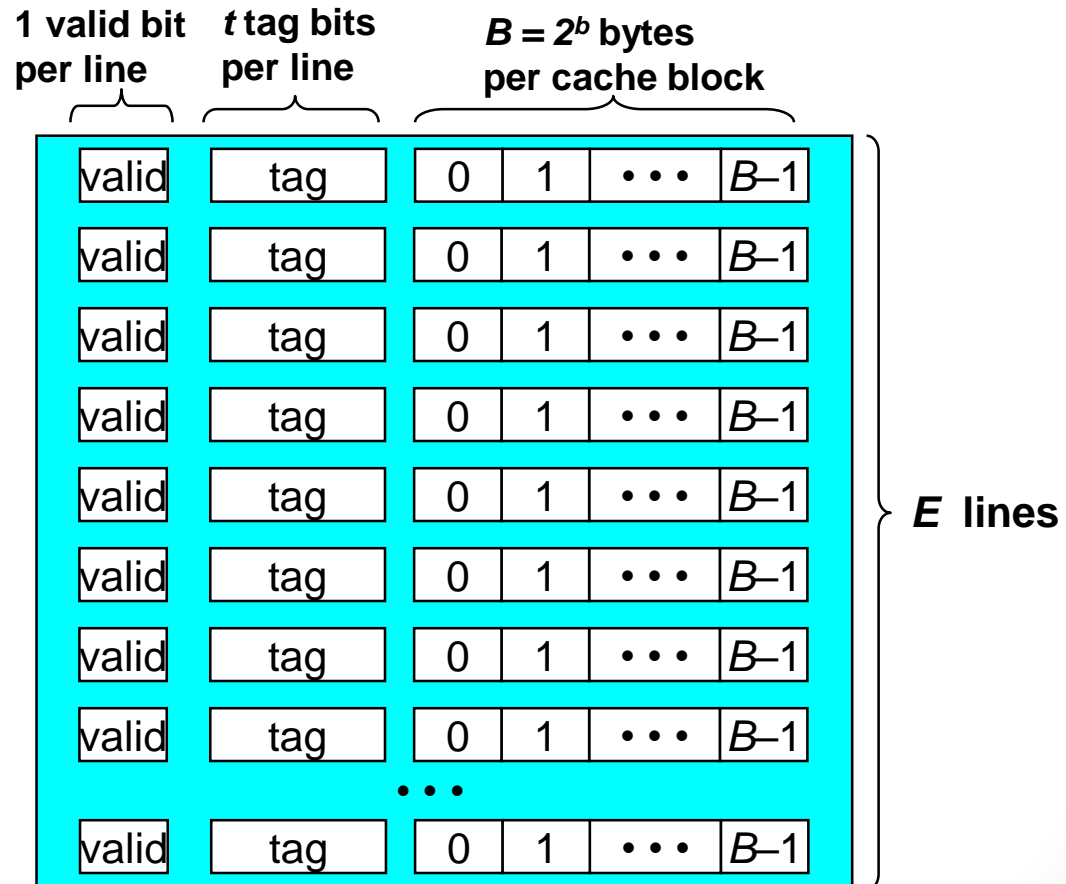
structure

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.

$S = 1$  set



Cache size:  $C = B \times E$  data bytes

# fully associative cache

<u>example</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
8	32	2048	32	64	1	27	0	5

```
let @x[256] = AAAA0000
    @y[256] = AAAA0400
```

sizeof( float ) = 4 bytes

sum is in a register

```
float dotprod (float x[256], float y[256])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 256; i++)
    {
        sum += x[ i ] * y[ i ];
    }
    return sum;
}
```

x[0] @ AAAA0000 = 1010 1010 1010 1010 0000 0000 0000 0000  
set bits =

fills 1<sup>st</sup> line in set

y[0] @ AAAA0400 = 1010 1010 1010 1010 0000 0100 0000 0000  
set bits =

fills 2<sup>nd</sup> line in set

... and so forth ...

# it's your turn . . .

<u>problem</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
8	32	2048	32	64	?	?	?	?

```
let @x[1024] = 0...0AAAA0000
    @y[1024] = 0...0AAAA1000
    @z[1024] = 0...0AAAA2000
```

`sizeof( float )` = 4 bytes

sum is in a register  
LRU eviction

```
float dotplus (float x[1024], float y[1024],
               float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum += x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?  
what is the hit rate for this loop?  
what is in the cache at the end of loop execution?

# it's your turn . . .

<u>problem</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
8	32	2048	32	2	?	?	?	?

```
let @x[1024] = 0...0AAAA0000
    @y[1024] = 0...0AAAA1000
    @z[1024] = 0...0AAAA2000
```

sizeof( float ) = 4 bytes

sum is in a register  
LRU eviction

```
float dotplus (float x[1024], float y[1024],
               float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum += x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?  
what is the hit rate for this loop?  
what is in the cache at the end of loop execution?

# it's your turn . . .

<u>problem</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
8	32	2048	32	4	?	?	?	?

```
let @x[512] = 0...0AAAAA0000
    @y[512] = 0...0AAAAA1000
    @z[512] = 0...0AAAAA2000
```

sizeof( float ) = 4 bytes

sum is in a register  
LRU eviction

```
float dotplus (float x[512], float y[512],
               float z[512])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum += x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

what are remaining cache parameters?  
what is the hit rate for this loop?  
what is in the cache at the end of loop execution?

# set associative cache

example

<u>problem</u>	<u>m</u>	<u>C</u>	<u>B</u>	<u>E</u>	<u>S</u>	<u>t</u>	<u>s</u>	<u>b</u>
1	64	4096	32	4	32	54	5	5

```
let @x[1024] = 0...0AAAA0000
    @y[1024] = 0...0AAAA1000
    @z[1024] = 0...0AAAA2000
```

sizeof( float ) = 4 bytes

sum is in a register  
LRU eviction

```
float dotplus(float x[1024], float y[1024],
              float z[1024])
{
    float sum = 0.0;
    int i;

    for (i = 0; i < 1024; i++)
    {
        sum += x[ i ] * y[ i ] + z[ i ];
    }
    return sum;
}
```

for each set:

line 1  
line 2  
line 3  
line 4

x	y	z	z
y	z	z	x
z	z	x	y
-	x	y	z

■ ■ ■

x	y	z	z
y	z	z	x
z	z	x	y
z	x	y	z



# fully associative cache

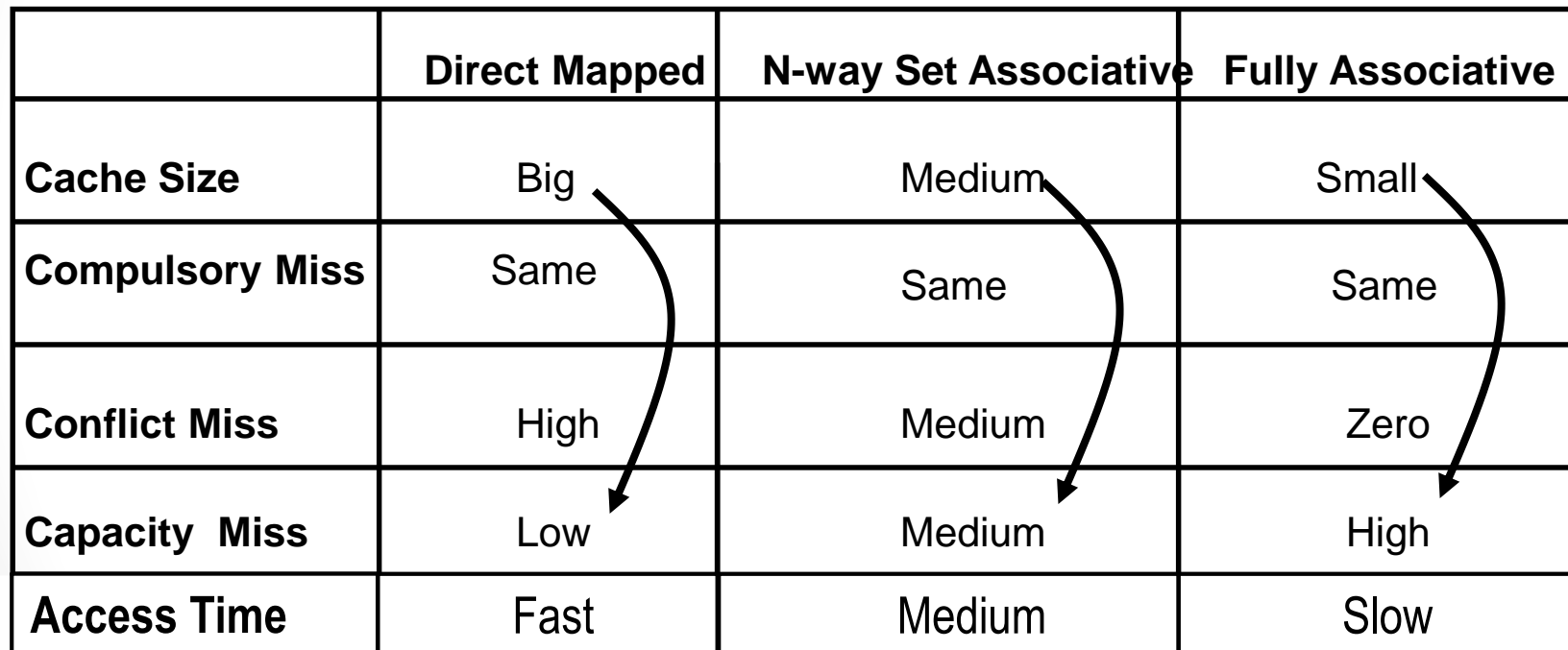
drawbacks

- more complex and expensive logic (compared to DMC)
- increased controller logic may slow low-level cache access times
- increased scan lengths may slow low-level cache access times
- preserving access times may decrease feasible cache size
- totally unnecessary for many common access patterns

# cache associativity

summary

	Direct Mapped	N-way Set Associative	Fully Associative
* <b>Cache Size</b>	Big	Medium	Small
<b>Compulsory Miss</b>	Same	Same	Same
<b>Conflict Miss</b>	High	Medium	Zero
<b>Capacity Miss</b>	Low	Medium	High
* <b>Access Time</b>	Fast	Medium	Slow



# cache management - part two

## cache properties

# simplified cache implementation size

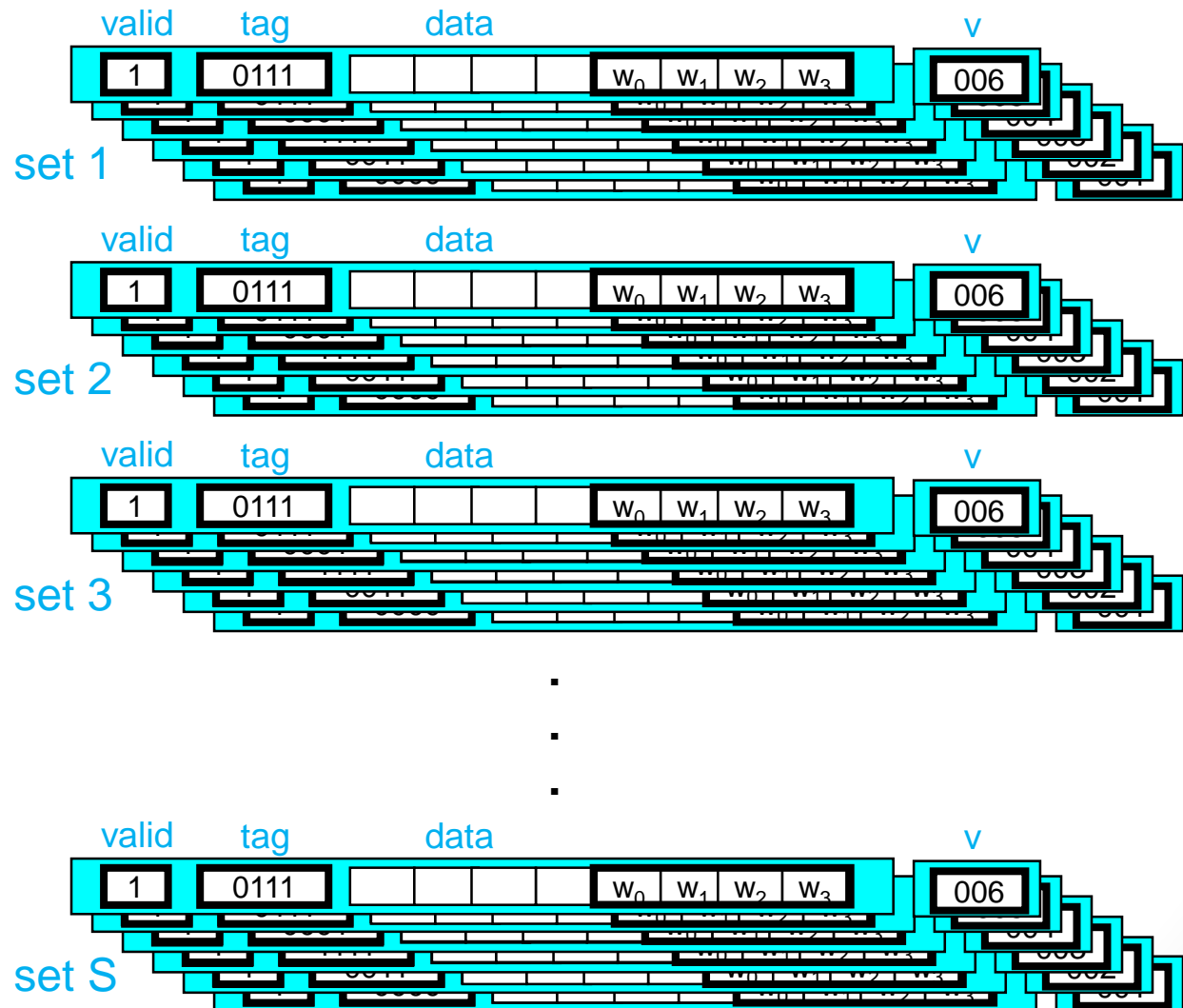
cache "size"  
 $= S \times E \times B$   
 bytes

cache  
 "implementation size"

$$= S \times E \times (1 + t + 8B + \epsilon)$$

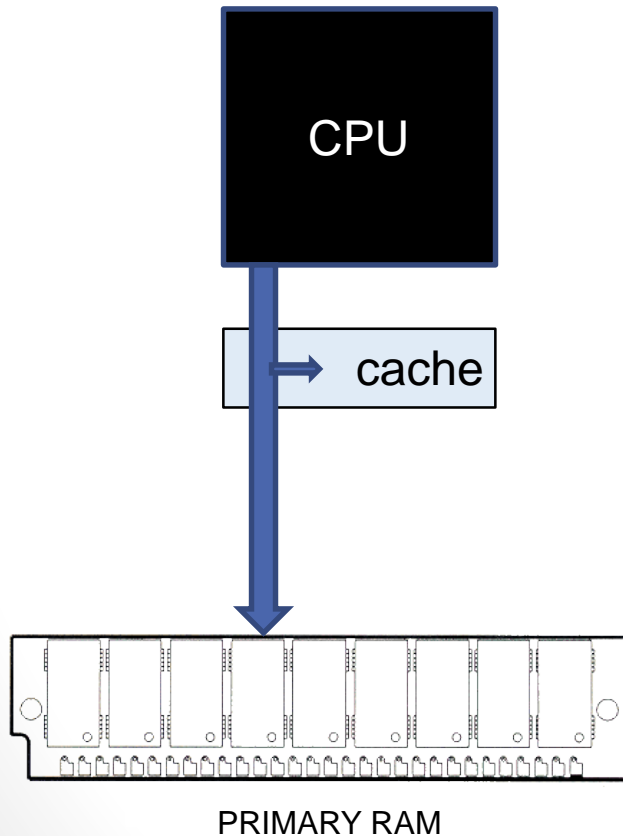
bits

$$\epsilon = |V|$$

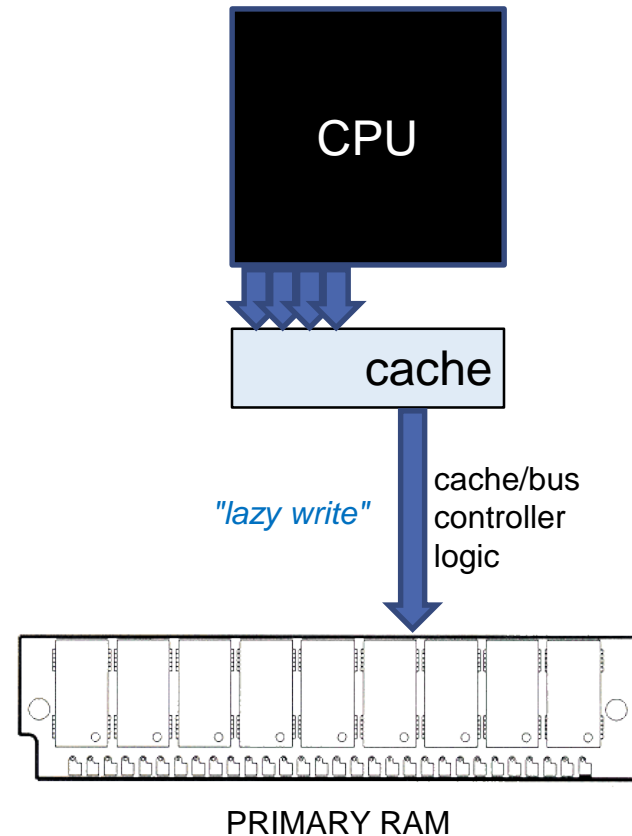


# cache writes

write through



write back



# unification and sharing

## cache content

- split instruction and data caches
- unified cache

## cache access

- private cache
- shared cache

# affects of cache size parameters

generalization

increase  
overall cache size

improved  
hit  
rate  
slower,  
(meet one-cycle target?)

increase  
block size

better  
spatial  
locality  
fewer lines,  
larger miss penalty

increase  
associativity

decreased  
thrashing  
expensive,  
slow,  
more complex,  
increased penalties

# cache management - part two

## cache performance metrics



# overall cache performance

## Average Memory Access Time (AMAT)

$t_{\text{hit}}$	hit time	cycles to complete detection and transfer of a cache hit
$\text{prob\_miss}$	miss probability	% time for cache miss
$\text{penalty\_miss}$	miss penalty	<b>additional</b> cycles incurred to process a cache miss

# overall cache performance

$$\text{AMAT} = \underset{\substack{\text{(hit time)}}}{t_{\text{hit}}} + \underset{\substack{\text{(miss rate)}}}{\text{prob\_miss}} * \underset{\substack{\text{(miss penalty)}}}{\text{penalty\_miss}}$$

- reduce hit time
- reduce miss penalty
- reduce miss rate

# overall cache performance

single-level cache

$$AMAT = t_{hit} + prob_{miss} * penalty_{miss}$$

- L1\$ hits in 1 cycle, with 80% hit rate
- main memory hits in 1000 additional cycles

$$AMAT = 1 + (1-.8)(1000) \\ = 201$$

- L1\$ hits in 1 cycle, with 85% hit rate
- main memory hits in 1000 additional cycles

$$AMAT = 1 + (1-.85)(1000) \\ = 151$$

# overall cache performance

multi-level cache

$$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$$

- L1\$ hits in 1 cycle, with 50% hit rate
- L2\$ hits in 10 cycles, with 75% hit rate
- L3\$ hits in 100 cycles, with 90% hit rate
- main memory hits in 1000 cycles

$$\begin{aligned} AMAT_1 &= 1 + (1-.5)(AMAT_2) \\ &= 1 + (1-.5)(10 + (1-.75)(AMAT_3)) \\ &= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(AMAT_m))) \\ &= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(1000))) \\ &= 31 \end{aligned}$$

what percentage of accesses hit main memory?

# it's your turn . . .

overall cache performance

$$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$$

- L1\$ hits in 1 cycle, with 25% hit rate
- L2\$ hits in 6 cycles, with 80% hit rate
- L3\$ hits in 60 cycles, with 95% hit rate
- main memory hits in 1000 cycles

$$AMAT_1 = ??$$

what percentage of accesses hit main memory?

# it's your turn . . .

overall cache performance

$$AMAT_i = t\_hit_i + prob\_miss_i * penalty\_miss_i$$

- L1\$ hits in 1 cycle, with 70% hit rate
- L2\$ hits in 20 cycles, with 70% hit rate
- L3\$ hits in 200 cycles, with 70% hit rate
- main memory hits in 1000 cycles

$$AMAT_1 = ??$$

what percentage of accesses hit main memory?

# cache management - part two

cache friendly  
code

# writing cache-friendly code

- maximize spatial locality in data organization
- maximize spatial locality in data access
- maximize temporal locality
- engineer access strides
- padding
- blocking / tiling
- computational structure (ordered fetching)



# spatial locality in data organization

```
struct part_type
{
    int      id;
    int      value;
    int      supplier_id[MAX_SUPPLIER];
    blob     image;
} parts[MAX_PARTS];
```

# spatial locality in data access

A

```
for (int i = 0; i < N; i++)  
{  
    for (int j = 0; j < N; j++)  
    {  
        sum[ j ] += matrix[ j ][ i ]  
    }  
}
```

B

```
for (int i = 0; i < N; i++)  
{  
    for (int j = 0; j < N; j++)  
    {  
        sum[ i ] += matrix[ i ][ j ]  
    }  
}
```

# temporal locality

B

```
for (int i = 0; i < cols; i++)  
{  
    for (int j = 0; j < rows; j++)  
    {  
        sum[ i ] += matrix[ i ][ j ]  
    }  
}
```

# access strides

```
int a[size], b[size], c[size];  
  
for (int i = 0; i < size; i++)  
{  
    a[i] = b[i] + c[i];  
}
```

```
let size = 2048  
sizeof(int) = 4  
direct mapped cache,  
    m = 32  
    B = 32  
    S = 64  
(E = 1)
```

# padding

```
int a[pSize], b[pSize], c[pSize];  
  
for (int i = 0; i < size; i++)  
{  
    a[i] = b[i] + c[i];  
}
```

let size = 2048  
sizeof(int) = 4  
direct mapped cache,  
m = 32  
B = 32  
S = 64  
(E = 1)

let pSize = size + B/4 = 2056  
sizeof(int) = 4  
direct mapped cache,  
m = 32  
B = 32  
S = 64  
(E = 1)

# blocking / tiling

sizeof(float) = 8

C = 64K

m = 32

B = 32

S = 64

E = 1

note border condition is finessed ...

```
float A [1024] [1024], R [1024], C [1024]
```

```
for ( i = 0; i < 1024; i++ )
```

```
    for ( j = 0; j < 1024; j++ )
```

```
        R[ i ] += A[ i ][ j ]
```

```
        C[ i ] -= A[ i ][ j ]
```

*many other computations ...*

```
float A [1024] [1024], R [1024], C [1024]
```

```
for ( i = 0; i < 1024; i++ )
```

```
    for ( jb = 0; jb < 1024; jb += 4 )
```

```
        for ( j = 0; j < 4; j++ )
```

```
            R[ i ] += A[ i ][ jb+j ]
```

```
            C[ i ] += A[ i ][ jb+j ]
```

*many other computations ...*

# ordered fetching

## approach A:

declare large data structure A  
initialize large data structure A

declare large data structure B  
initialize large data structure B

declare large data structure C  
initialize large data structure C

.  
.  
.

compute A  
compute B  
compute C

## approach B:

declare large data structure A  
initialize large data structure A  
compute A

.  
.  
.

declare large data structure B  
initialize large data structure B  
compute B

.  
.  
.

declare large data structure C  
initialize large data structure C  
compute C

# cse5441 - parallel computing

## cache management part two

