

CS 33

Architecture and Optimization (3)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

What About Branches?

- Challenge

- **instruction control unit** must work well ahead of **execution unit** to generate enough operations to keep EU busy

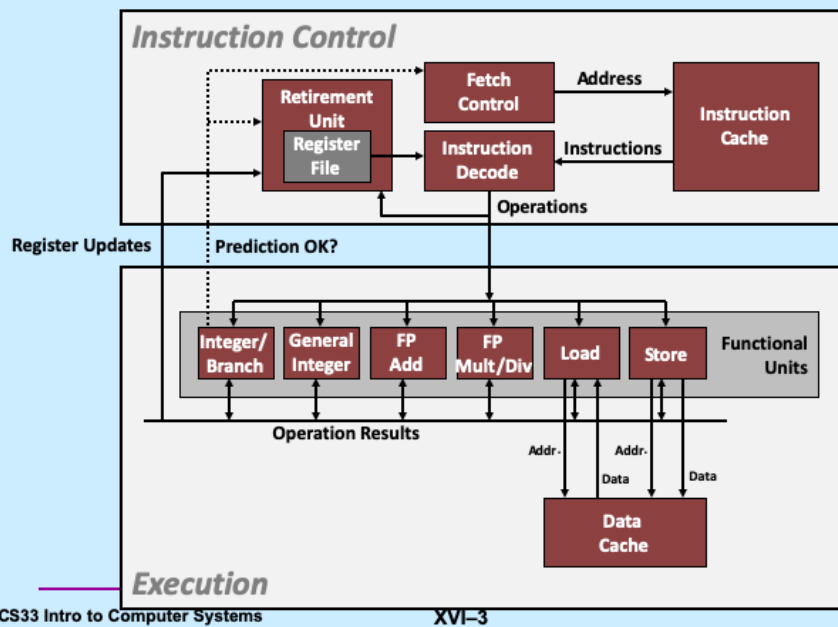
```
80489f3: movl    $0x1,%ecx
80489f8: xorq    %rdx,%rdx
80489fa: cmpq    %rsi,%rdx
80489fc: jnl     8048a25
80489fe: movl    %esi,%edi
8048a00: imull   (%rax,%rdx,4),%ecx
```

} Executing

← How to continue?

- when it encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Supplied by CMU.

Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
 - branch taken: transfer control to branch target
 - branch not-taken: continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
80489f3: movl    $0x1,%ecx
80489f8: xorq    %rdx,%rdx
80489fa: cmpq    %rsi,%rdx
80489fc: jnl     8048a25
80489fe: movl    %esi,%esi
8048a00: imull   (%rax,%rdx,4),%ecx
```

Branch not-taken

Branch taken

```
8048a25: cmpq    %rdi,%rdx
8048a27: jl      8048a20
8048a29: movl    0xc(%rbp),%eax
8048a2c: leal    0xffffffff8(%rbp),%esp
8048a2f: movl    %ecx,(%rax)
```

Supplied by CMU.

Branch Prediction

- Idea

- guess which way branch will go
- begin executing instructions at predicted position
 - » but don't actually modify register or memory data

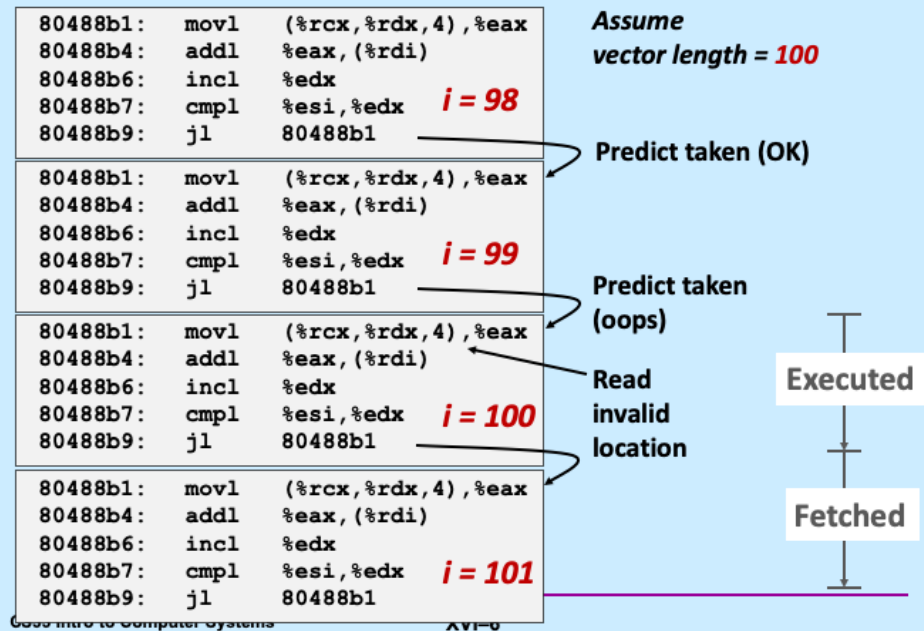
```
80489f3: movl    $0x1,%ecx
80489f8: xorq    %edx,%edx
80489fa: cmpq    %rsi,%rdx
80489fc: jnl     8048a25
. . .
```

Predict taken

```
8048a25: cmpq    %rdi,%rdx
8048a27: jl      8048a20
8048a29: movl    0xc(%rbp),%eax
8048a2c: leal    0xffffffff8(%rbp),%esp
8048a2f: movl    %ecx,(%rax)
```

} Begin
execution

Branch Prediction Through Loop



Supplied by CMU.

Branch Misprediction Invalidation

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 98
80488b9: j1l     80488b1
```

Assume
vector length = **100**

Predict taken (OK)

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 99
80488b9: j1l     80488b1
```

Predict taken (oops)

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 100
80488b9: j1l     80488b1
```

Invalidate

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax, (%rdi)
80488b6: incl    %edx    i = 101
```

Branch Misprediction Recovery

```
80488b1: movl    (%rcx,%rdx,4),%eax
80488b4: addl    %eax,(%rdi)
80488b6: incl    %edx
80488b7: cmpl    %esi,%edx    i = 99
80488b9: jnl     80488b1
80488bb: leal    0xffffffffe8(%rbp),%esp
80488be: popl    %ebx
80488bf: popl    %esi
80488c0: popl    %edi
```

Definitely not taken

- **Performance Cost**

- multiple clock cycles on modern processor
- can be a major performance limiter

Latency of Loads

```
typedef struct ELE {  
    struct ELE *next;  
    long data;  
} list_ele, *list_ptr;
```

```
int list_len(list_ptr ls) {  
    long len = 0;  
    while (ls) {  
        len++;  
        ls = ls->next;  
    }  
    return len;  
}
```

```
# len in %rax, ls in %rdi
```

```
.L11:                # loop:  
    addq    $1, %rax    # incr len  
    movq    (%rdi), %rdi # ls = ls->next  
    testq   %rdi, %rdi  # test ls  
    jne     .L11        # if != 0  
                    # go to loop
```

• 4 CPE

This example is from the textbook (Figure 5.31). Here we can't execute the loads in parallel, since each load is dependent on the result of the previous load. The point is that loads (fetching data from memory) have a latency of 4 cycles.

Clearing an Array ...

```
#define ITERS 100000000
void clear_array() {
    long dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<100; i++)
            dest[i] = 0;
    }
}
```

• 1 CPE

This is adapted from Figure 5.32 of the textbook. There are no data dependencies and thus the stores can be pipelined.

Store/Load Interaction

```
void write_read(long *src, long *dest, long n) {  
    long cnt = n;  
    long val = 0;  
  
    while(cnt--) {  
        *dest = val;  
        val = (*src)+1;  
    }  
}
```

This code is from the textbook.

Store/Load Interaction

```
void write_read(long *src,
               long *dest, long n){
    long cnt = n;
    long val = 0;
    while(cnt--){
        *dest = val;
        val = (*src)+1;
    }
}
```

```
long a[] = {-10, 17};
```

Example A: `write_read(&a[0], &a[1], 3)`

	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	-10 17	-10 0	-10 -9	-10 -9
val	0	-9	-9	-9

• CPE 1.3

Example B: `write_read(&a[0], &a[0], 3)`

	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	-10 17	0 17	1 17	2 17
val	0	1	2	3

• CPE 7.3

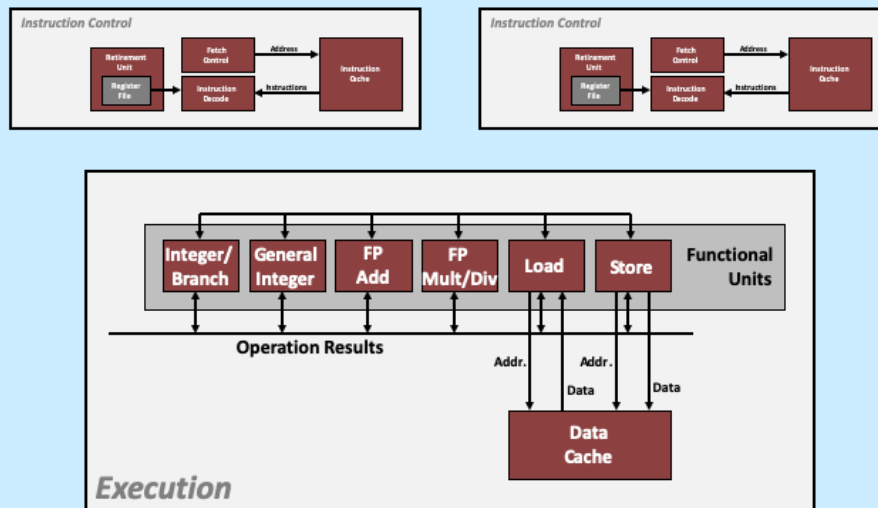
This is Figure 5.33 of the textbook. Performance depends upon whether *src* and *dest* are the same location. If they are different locations, they don't interact that loads and stores can be pipelined. If they are the same locations, then they do interact and pipelining is not possible.

Getting High Performance

- **Good compiler and flags**
- **Don't do anything stupid**
 - watch out for hidden algorithmic inefficiencies
 - write compiler-friendly code
 - » watch out for optimization blockers:
procedure calls & memory references
 - look carefully at innermost loops (where most work is done)
- **Tune code for machine**
 - exploit instruction-level parallelism
 - avoid unpredictable branches
 - make code cache friendly (covered soon)

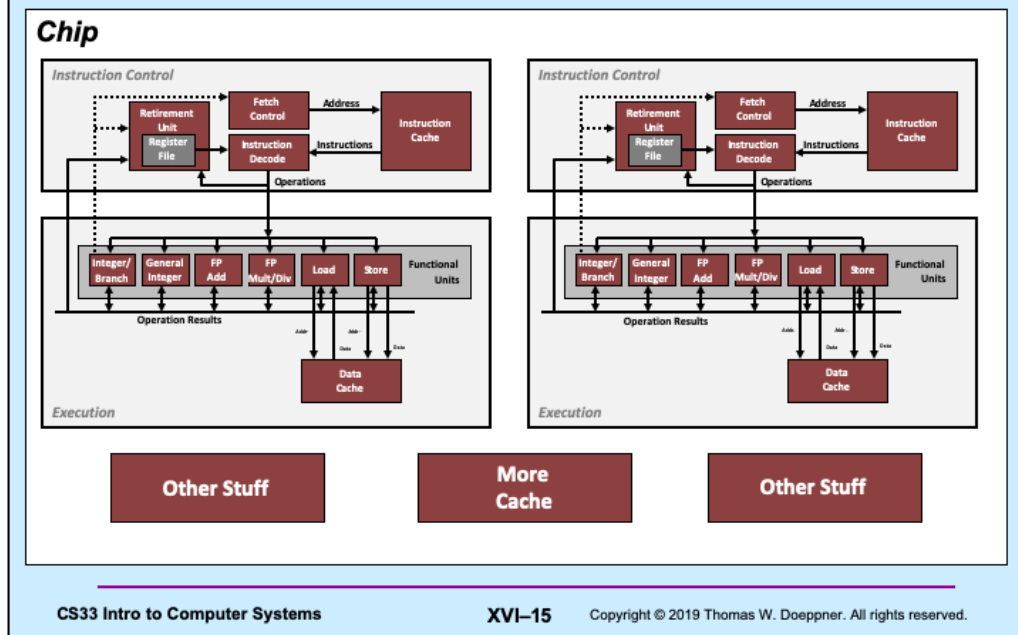
Supplied by CMU.

Hyper Threading



One way of improving the utilization of the functional units of a processor is hyperthreading. The processor supports multiple instruction streams ("hyper threads"), each with its own instruction control. But all the instruction streams share the one set of functional units.

Multiple Cores



Going a step further, one can pack multiple complete processors onto one chip. Each processor is known as a core and can execute instructions independently of the other cores (each has its private set of functional units). In addition to each core having its own instruction and data cache, there are caches shared with the other cores on the chip. We discuss this in more detail in a subsequent lecture.

In many of today's processor chips, hyperthreading is combined with multiple cores. Thus, for example, a chip might have four cores each with four hyperthreads. Thus the chip might handle 16 instruction streams.

CS 33

Memory Hierarchy I

Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Random-Access Memory (RAM)

- **Key features**
 - **RAM** is traditionally packaged as a chip
 - basic storage unit is normally a **cell** (one bit per cell)
 - multiple RAM chips form a memory
- **Static RAM (SRAM)**
 - each cell stores a bit with a four- or six-transistor circuit
 - retains value indefinitely, as long as it is kept powered
 - relatively insensitive to electrical noise (EMI), radiation, etc.
 - faster and more expensive than DRAM
- **Dynamic RAM (DRAM)**
 - each cell stores bit with a capacitor; transistor is used for access
 - value must be refreshed every 10-100 ms
 - more sensitive to disturbances (EMI, radiation,...) than SRAM
 - slower and cheaper than SRAM

Supplied by CMU.

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

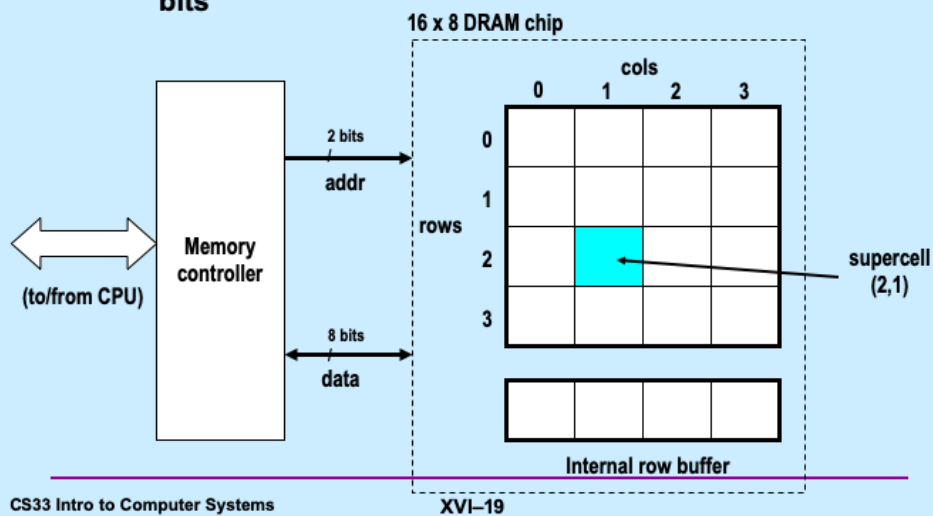
- **EDC = error detection and correction**
 - to cope with noise, etc.

Supplied by CMU.

Conventional DRAM Organization

- $d \times w$ DRAM:

- dw total bits organized as d **supercells** of size w bits



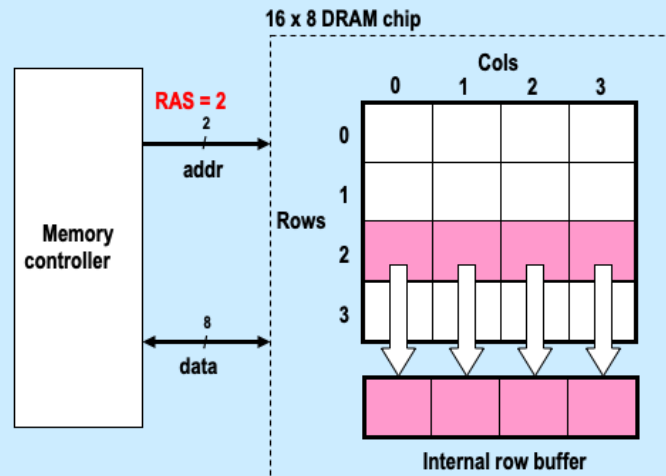
Supplied by CMU.

Note that the chip in the slide contains 16 supercells of 8 bits each. The supercells are organized as a 4x4 array.

Reading DRAM Supercell (2,1)

Step 1(a): row access strobe (**RAS**) selects row 2

Step 1(b): row 2 copied from DRAM array to row buffer

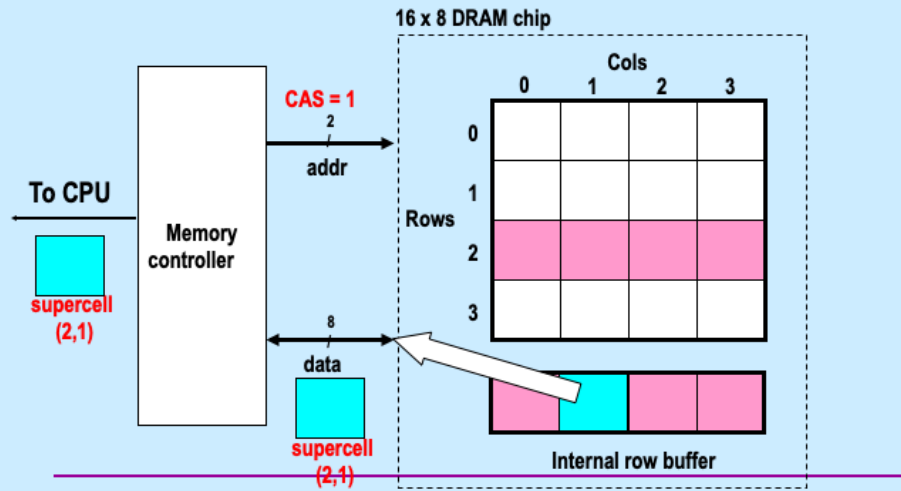


Supplied by CMU.

Reading DRAM Supercell (2,1)

Step 2(a): column access strobe (**CAS**) selects column 1

Step 2(b): supercell (2,1) copied from buffer to data lines, and eventually back to the CPU

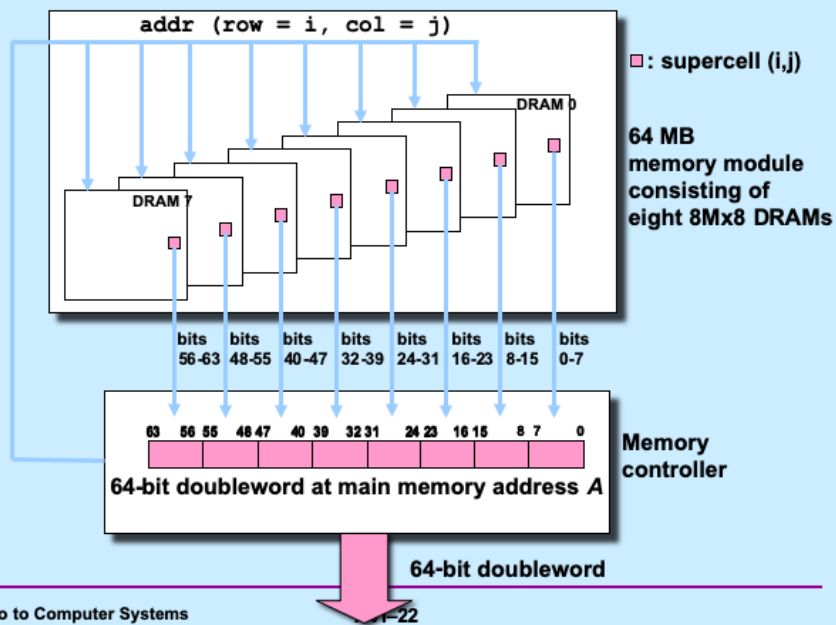


CS33 Intro to Computer Systems

XVI-21

Supplied by CMU.

Memory Modules



CS33 Intro to Computer Systems

F-22

Supplied by CMU.

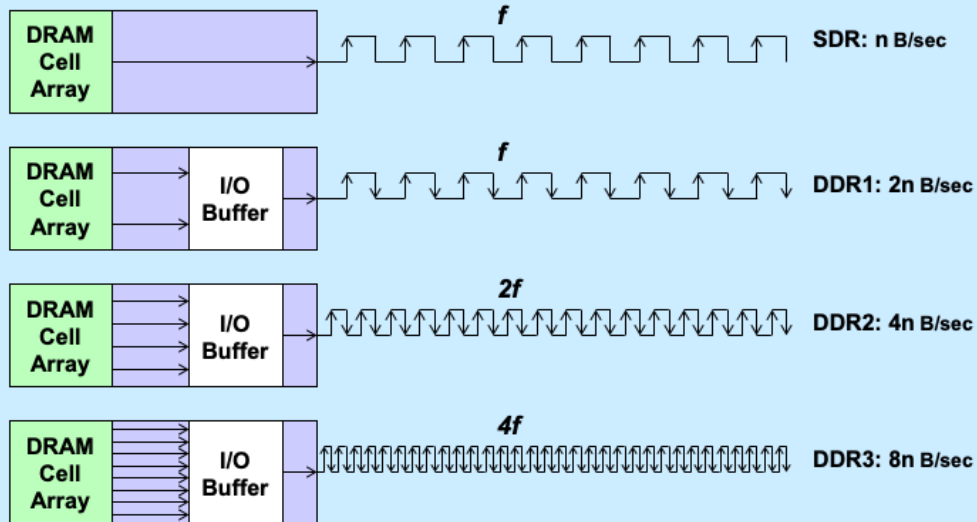
The memory controller pulls in eight supercells from eight DRAM modules and transfers them to the processor over the memory bus.

Enhanced DRAMs

- **Basic DRAM cell has not changed since its invention in 1966**
 - commercialized by Intel in 1970
- **DRAMs with better interface logic and faster I/O:**
 - **synchronous DRAM (SDRAM)**
 - » uses a conventional clock signal instead of asynchronous control
 - » allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
 - **double data-rate synchronous DRAM (DDR SDRAM)**
 - » **DDR1**
 - twice as fast
 - » **DDR2**
 - four times as fast
 - » **DDR3**
 - eight times as fast

Supplied by CMU.

Enhanced DRAMs



This slide is based on figures from *What Every Programmer Should Know About Memory* (<http://www.akkadia.org/drepper/cpumemory.pdf>), by Ulrich Drepper. It's an excellent article on memory and caching.

It is costly to make DRAM cell arrays run at a faster rate. Thus rather than speed up the operation of the individual modules, they are organized to transfer in parallel. Thus all that needs to be sped up is the bus that carries the data (something that is relatively inexpensive to do).

With SDR (Single Data-Rate DRAM), the DRAM cell array produces data at the same frequency as the memory bus, sending data on the rising edge of the signal.

With DDR1 (double data-rate), data is sent twice as fast by "double-pumping" the bus: sending data on both the rising and falling edges of the signal. To get data out of the cell array at this speed, data from two adjacent supercells are produced at once. These are buffered so that one doubleword at a time can be transmitted over the bus.

With DDR2, the frequency of the memory bus is doubled, and four supercells are produced at once. DDR3 takes this one step further, with eight supercells being produced at once.

Note that the processor fetches and stores 64 bytes of data at a time (for reasons having to do with caching, which we cover later in this lecture).

Summary

- **Memory transfer speed increased by a factor of 8**
 - no increase in DRAM Cell Array speed
 - 8 times more data transferred at once
 - » 64 adjacent bytes fetched from DRAM

Quiz 1

A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR3 system at a speed that's n times that of an SDR system, where n is:

- a) 1**
- b) 2**
- c) 4**
- d) 8**

A Mismatch

- **A processor clock cycle is ~0.3 nsecs**
 - SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- **Basic operations take 1 – 10 clock cycles**
 - .3 – 3 nsecs
- **Accessing memory takes 70-100 nsecs**
- **How is this made to work?**

Caching to the Rescue

CPU

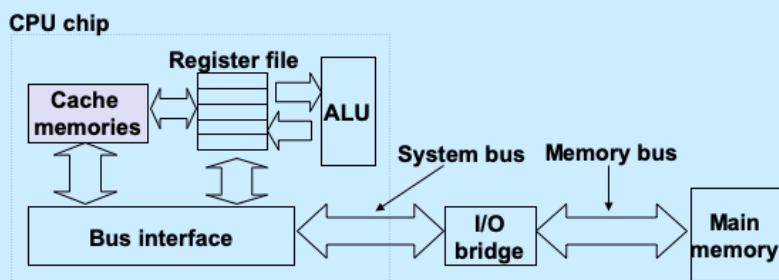
Cache



Sitting between the processor and RAM are one or more caches. (They actually are on the chip along with the processor.) Recently accessed items by the processor reside in the cache, where they are much more quickly accessed than directly from memory. The processor does a certain amount of pre-fetching to get things from RAM before they are needed. This involves a certain amount of guesswork, but works reasonably well, given well behaved programs.

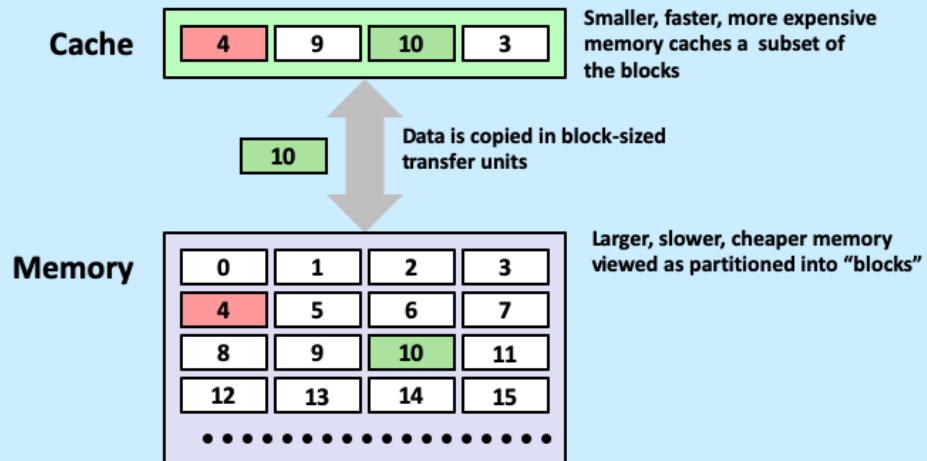
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
 - hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory
- Typical system structure:



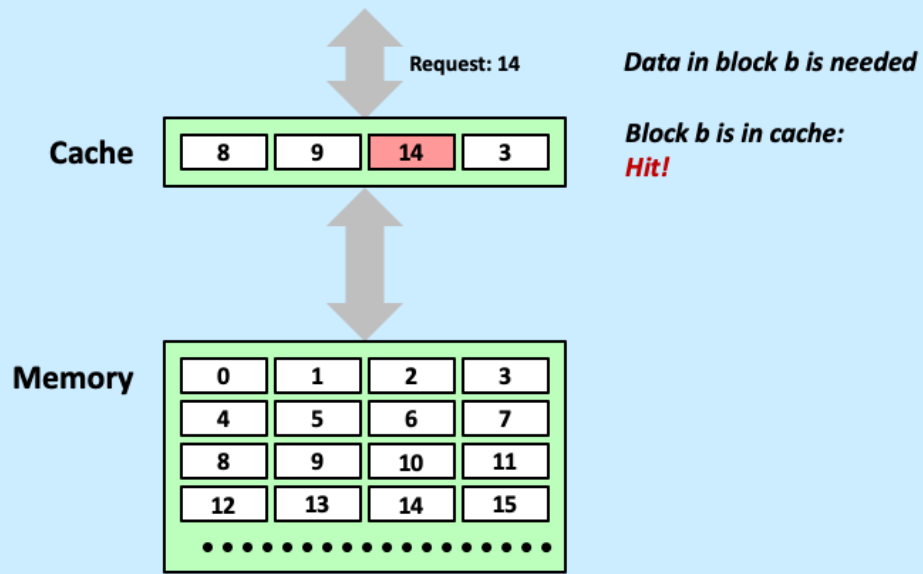
Supplied by CMU.

General Cache Concepts



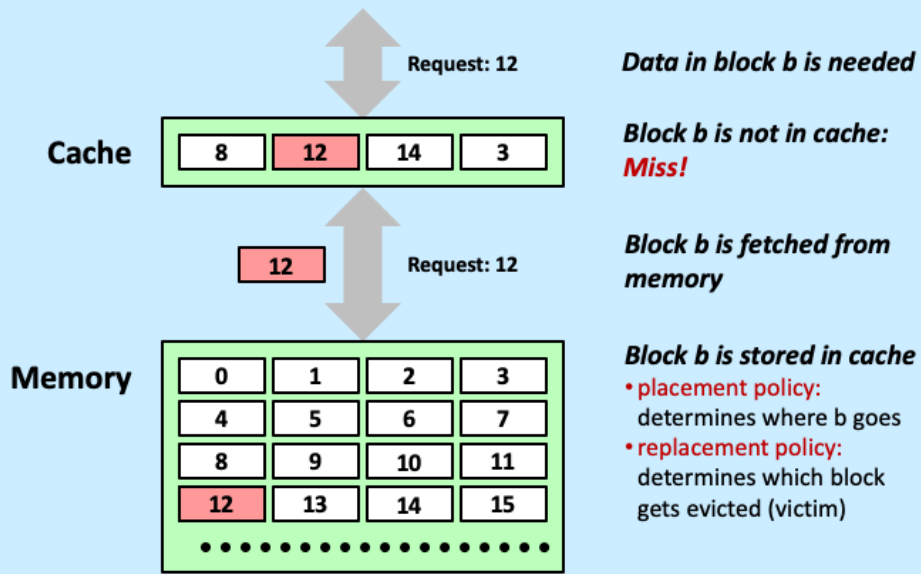
Supplied by CMU.

General Cache Concepts: Hit



Supplied by CMU.

General Cache Concepts: Miss

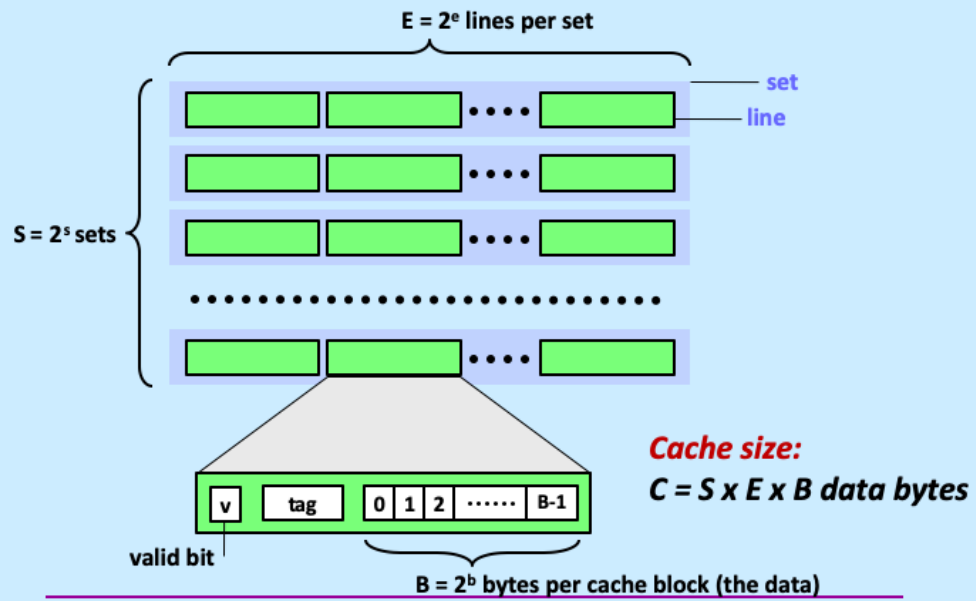


Supplied by CMU.

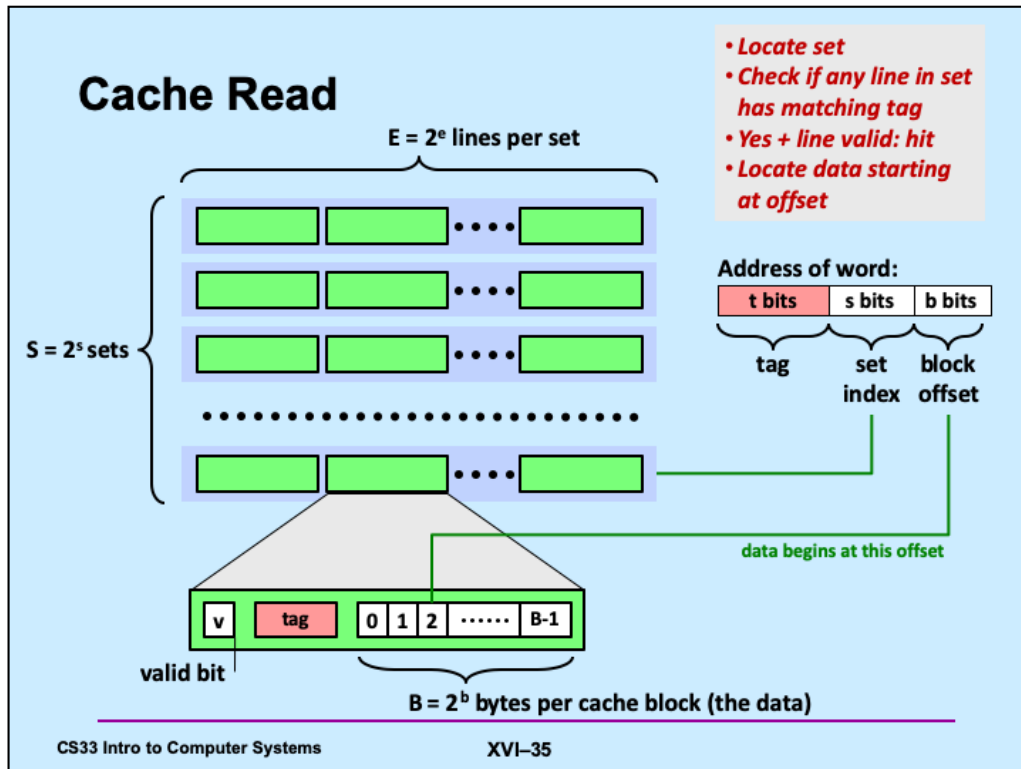
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
 - cold misses occur because the cache is empty
- **Conflict miss**
 - most caches limit blocks to a small subset (sometimes a singleton) of the block positions in RAM
 - » e.g., block i in RAM must be placed in block $(i \bmod 4)$ in the cache
 - conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache block
 - » e.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time
- **Capacity miss**
 - occurs when the set of active cache blocks (**working set**) is larger than the cache

General Cache Organization (S, E, B)



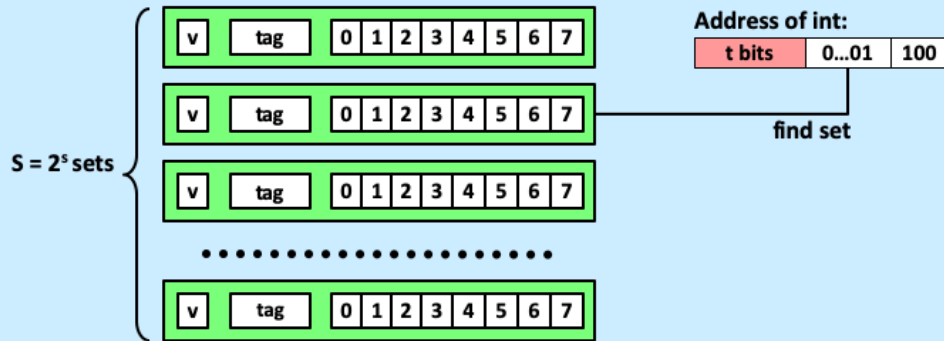
Supplied by CMU.



Supplied by CMU.

Example: Direct Mapped Cache (E = 1)

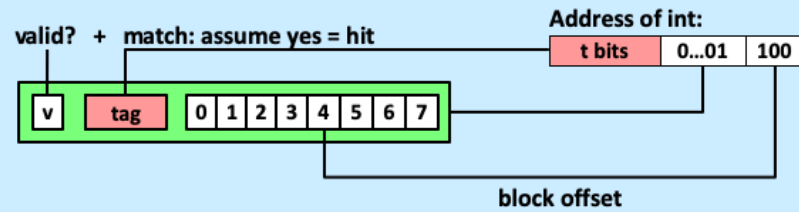
Direct mapped: one line per set
Assume: cache block size 8 bytes



Supplied by CMU.

Example: Direct Mapped Cache (E = 1)

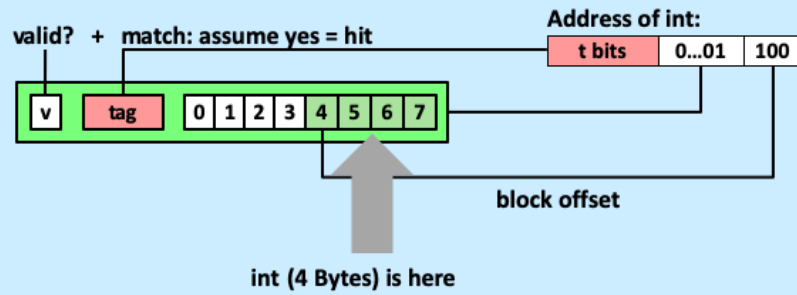
Direct mapped: one line per set
Assume: cache block size 8 bytes



Supplied by CMU.

Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Supplied by CMU.

Direct-Mapped Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Supplied by CMU.

A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

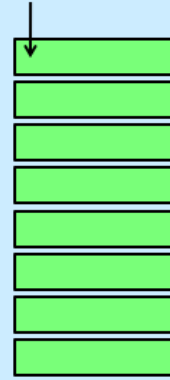
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{0,8}$	$a_{0,9}$	$a_{0,10}$	$a_{0,11}$
$a_{0,12}$	$a_{0,13}$	$a_{0,14}$	$a_{0,15}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{1,8}$	$a_{1,9}$	$a_{1,10}$	$a_{1,11}$
$a_{1,12}$	$a_{1,13}$	$a_{1,14}$	$a_{1,15}$

32 B = 4 doubles

Note that the cache holds two rows of the matrix; each cache block holds four doubles.

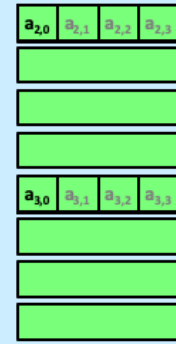
A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```



32 B = 4 doubles

For each reference to an element of the matrix, its entire row is brought into the cache, even though the rest of the row is not immediately used.

Conflict Misses: Aligned

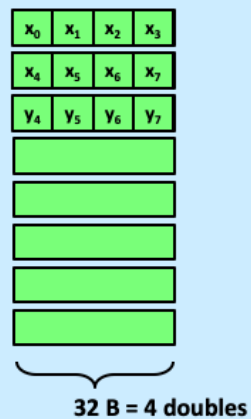
```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```



If arrays x and y have the same alignment, i.e., both start in the same cache set, then each access to an element of y replaces the cache line containing the corresponding element of x , and vice versa. The result is that loop is executed very slowly — each access to either array results in a conflict miss.

Different Alignments

```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```

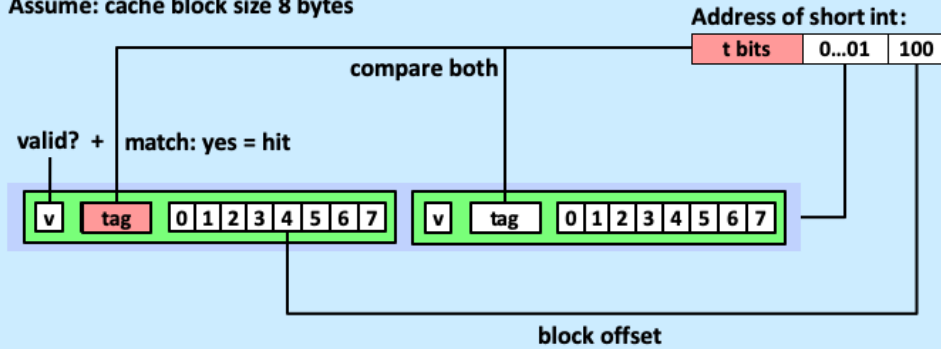


However, if the two arrays start in different cache sets, then the loop executes quickly — there is a cache miss on just every fourth access to each array.

E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

Assume: cache block size 8 bytes

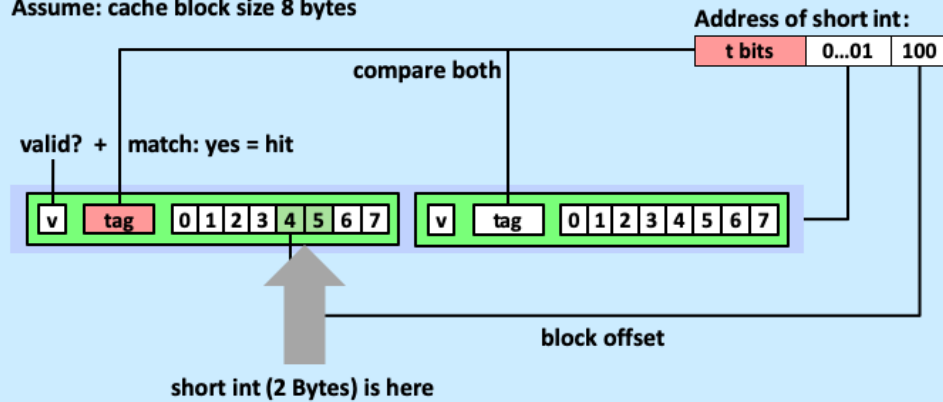


Supplied by CMU.

E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Supplied by CMU.

Quiz 2

Address of int:

100	01	100
-----	----	-----

0	v	tag=0	0	0	0	0	1	1	1	1
	v	tag=2	2	2	2	2	3	3	3	3
1	v	tag=0	4	4	4	4	5	5	5	5
	v	tag=4	6	6	6	6	7	7	7	7
2	v	tag=2	8	8	8	8	9	9	9	9
	v	tag=3	a	a	a	a	b	b	b	b
3	v	tag=4	c	c	c	c	d	d	d	d
	v	tag=a	e	e	e	e	f	f	f	f

Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

- a) 1111
- b) 3333
- c) 4444
- d) 7777

2-Way Set-Associative Cache Simulation

t=2	s=1	b=1
XX	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[0000 ₂],	miss
1	[0001 ₂],	hit
7	[0111 ₂],	miss
8	[1000 ₂],	miss
0	[0000 ₂]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Supplied by CMU.

A Higher-Level Example

Ignore the variables `sum`, `i`, `j`

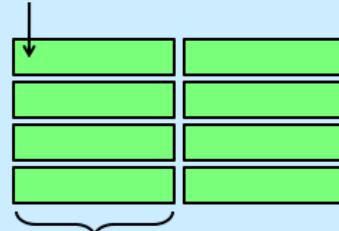
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
`a[0][0]` goes here



32 B = 4 doubles

A Higher-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

a _{0,0}	a _{0,1}	a _{0,2}	a _{0,3}	a _{1,0}	a _{1,1}	a _{1,2}	a _{1,3}
a _{0,4}	a _{0,5}	a _{0,6}	a _{0,7}	a _{1,4}	a _{1,5}	a _{1,6}	a _{1,7}
a _{0,8}	a _{0,9}	a _{0,10}	a _{0,11}	a _{1,8}	a _{1,9}	a _{1,10}	a _{1,11}
a _{0,12}	a _{0,13}	a _{0,14}	a _{0,15}	a _{1,12}	a _{1,13}	a _{1,14}	a _{1,15}

32 B = 4 doubles

The cache still holds two rows of the matrix, but each row may go into one of two different cache lines. In the slide, the first row goes into the first lines of the cache sets, the second row goes into the second lines of the cache sets.

A Higher-Level Example

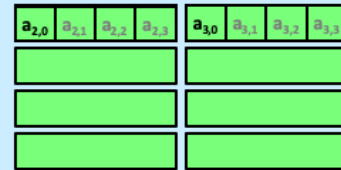
Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

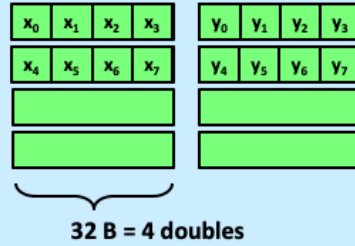


32 B = 4 doubles

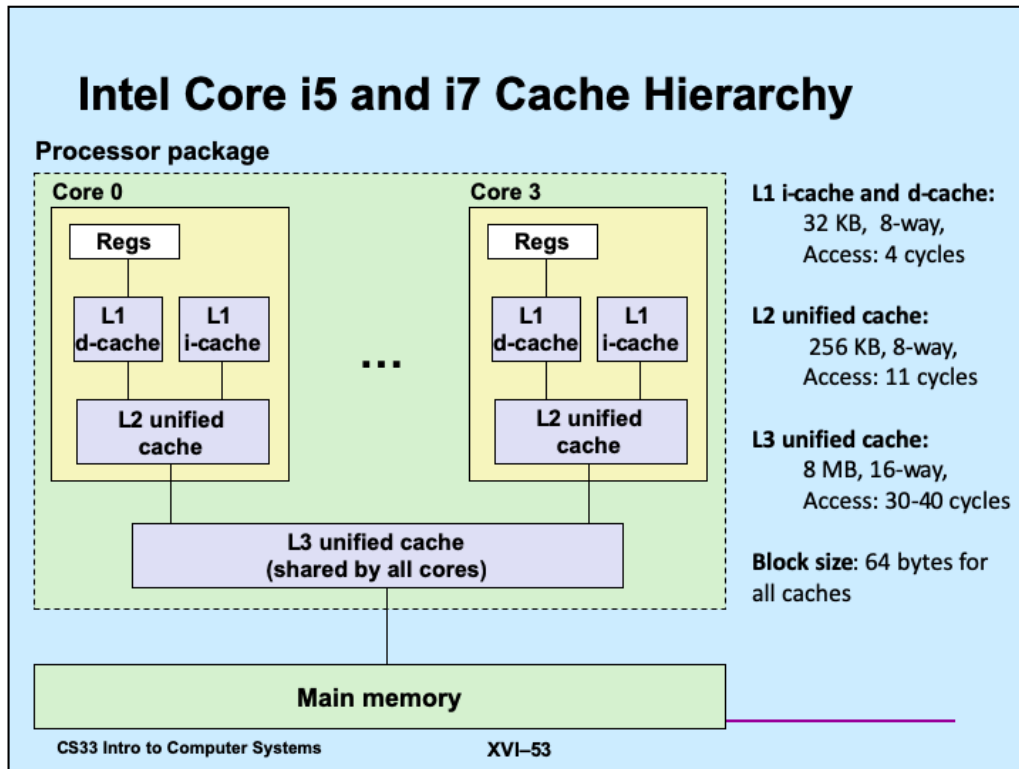
There is still a cache miss on each access.

Conflict Misses

```
double dotprod(double x[8], double y[8]) {  
    double sum = 0.0;  
    int i;  
  
    for (i=0; i<8; i++)  
        sum += x[i] * y[i];  
  
    return sum;  
}
```



With a 2-way set-associative cache, our dot-product example runs quickly even if the two arrays have the same alignment.



Supplied by CMU.

The L3 cache is known as the *last-level cache* (LLC) in the Intel documentation.

One concern is whether what's contained in, say, the L1 cache is also contained in the L2 cache. If so, caching is said to be *inclusive*. If what's contained in the L1 cache is definitely not contained in the L2 cache, caching is said to be *exclusive*. An advantage of exclusive caches is that the total cache capacity is the sum of the sizes of each of the levels, whereas for inclusive caches, the total capacity is just that of the largest. An advantage of inclusive caches is that what's been brought into the cache hierarchy by one core is available to the other core.

AMD processors tend to have exclusive caches; Intel processors tend to have inclusive caches.

What About Writes?

- **Multiple copies of data exist:**
 - L1, L2, main memory, disk
- **What to do on a write-hit?**
 - **write-through** (write immediately to memory)
 - **write-back** (defer write to memory until replacement of line)
 - » need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
 - **write-allocate** (load into cache, update line in cache)
 - » good if more writes to the location follow
 - **no-write-allocate** (writes immediately to memory)
- **Typical**
 - write-through + no-write-allocate
 - write-back + write-allocate

Supplied by CMU.

Most current processors use the write-back/write-allocate approach. This causes some (surmountable) difficulties for multi-core processors that have a separate cache for each core.