CS 33

Architecture and Optimization (3)

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.

Eliminate Unneeded Memory References

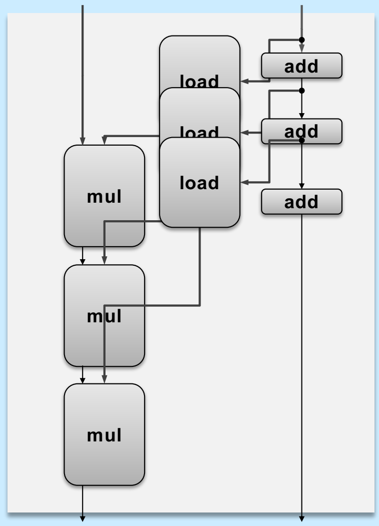
```
void combine4(vec_ptr_t v, data_t *dest){
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

Supplied by CMU.

Finally, we recognize that we don't need to update ***dest** on each iteration, but only when we're done.

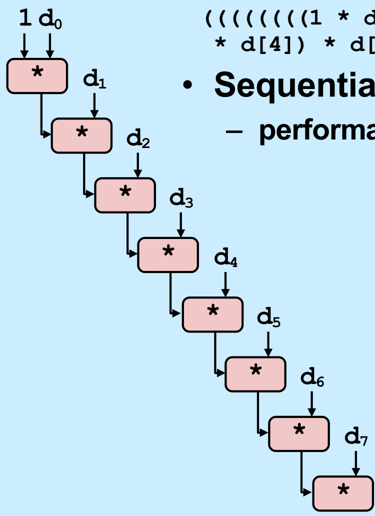
Pipelined Data-Flow Over Multiple Iterations



The loads depend only on the computation of the array index, which is quickly done by addition units. Thus, the loads can be pipelined.

It’s clear that the multiplies form the critical path, since they use the results of the previous multiplies.

Combine4 = Serial Computation (OP = *)



- **Computation (length=8)**

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence**
  - **performance: determined by latency of OP**

Supplied by CMU.

Since the multiplies form the critical path, here we focus only on them. In what's shown here, only one multiply can be done at a time, since the result of the one multiply is needed for the next.

Loop Unrolling

```
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

- **Perform 2x more useful work per iteration**

Supplied by CMU.

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	0.25	1.0	1.0	0.5

- Helps integer add
  – reduces loop overhead
- Others don't improve. *Why?*
  – still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```


Supplied by CMU.

Loop Unrolling with Reassociation

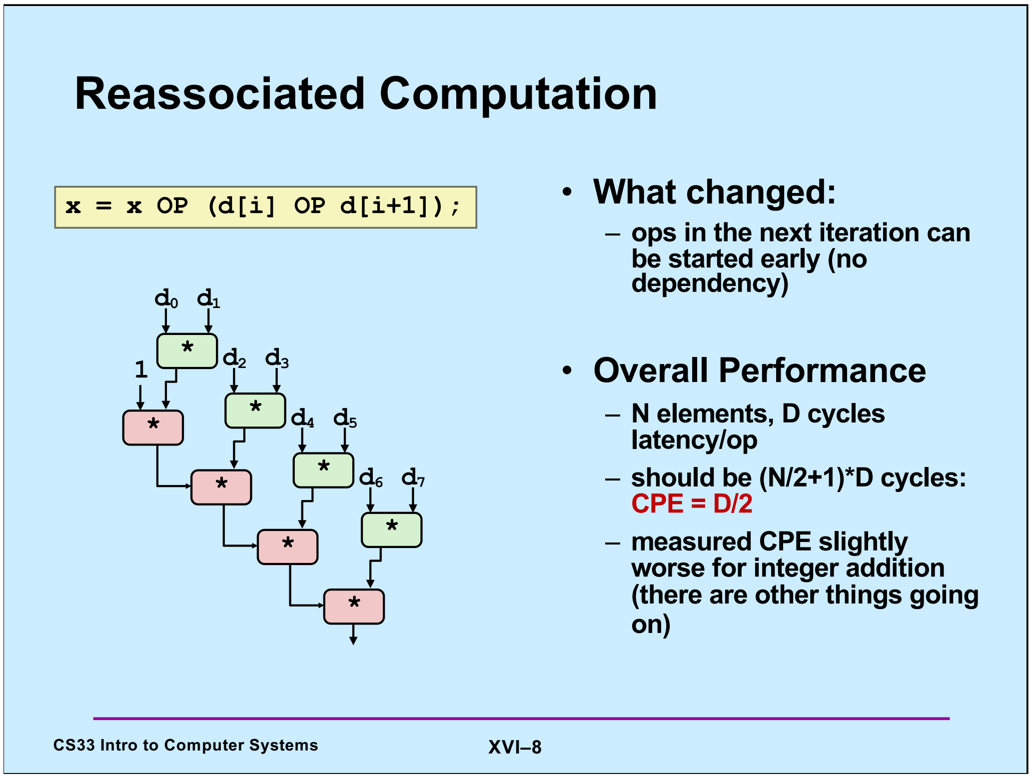
```
void unroll2xra(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- **Can this change the result of the computation?**
- **Yes, for FP. *Why?***

Supplied by CMU.



Supplied by CMU.

How much time is required to compute the products shown in the slide? The multiplications in the upper right of the tree, directly involving the d_i, could all be done at once, since there are no dependencies; thus, computing them can be done in D cycles, where D is the latency required for multiply. This assumes we have a sufficient number of functional units to do this, thus this is a lower bound. The multiplications in the lower left must be done sequentially, since each depends on the previous; thus, computing them requires (N/2)*D cycles. Since first of the top right multiplies must be completed before the bottom left multiplies can start, the overall performance has a lower bound of (N/2 + 1)*D.

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

- Nearly 2x speedup for int *, FP +, FP *
  - reason: breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```


Supplied by CMU.

Loop Unrolling with Separate Accumulators

```
void unroll2xp2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

Supplied by CMU.

Here one "accumulator" (x0) is summing the array elements with even indices, the other (x1) is summing array elements with odd indices.

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.00	3.00	5.00
Unroll 2x	1.01	3.00	3.00	5.00
Unroll 2x, reassociate	1.01	1.51	1.51	2.01
Unroll 2x parallel 2x	.81	1.51	1.51	2.51
Latency bound	1.0	3.0	3.0	5.0
Throughput bound	.25	1.0	1.0	.5

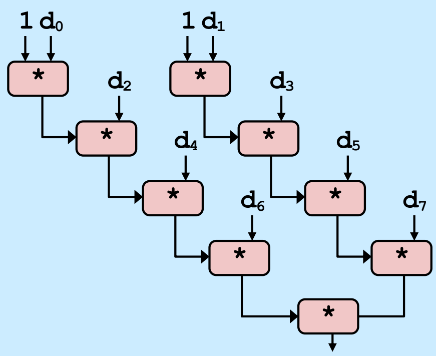
- 2x speedup (over unroll 2x) for int *, FP +, FP *
  - breaks sequential dependency in a "cleaner," more obvious way

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```


Supplied by CMU.

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - two independent “streams” of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - should be (N/2+1)*D cycles: **CPE = D/2**
  - Integer addition improved, but not yet at predicted value

What Now?

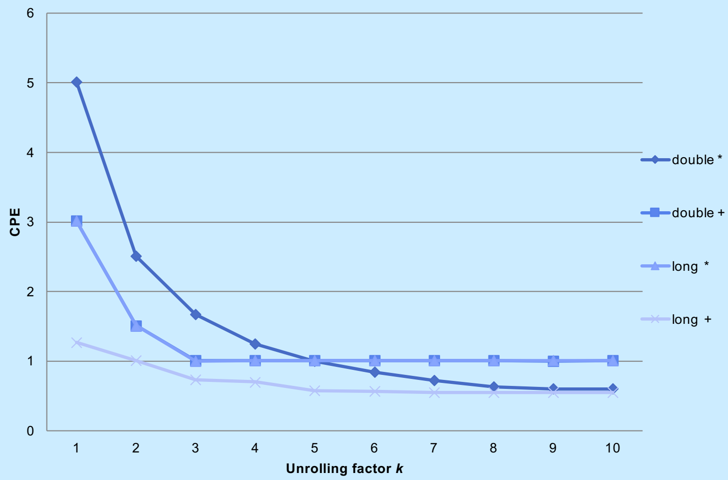
Supplied by CMU.

Quiz 1

We're making progress. With two accumulators we get a two-fold speedup. With three accumulators, we can get a three-fold speedup. How much better performance can we expect if we add even more accumulators?

a) It keeps on getting better as we add more and more accumulators
b) It's limited by the latency bound
c) It's limited by the throughput bound
d) It's limited by something else

Performance



- **K-way loop unrolling with K accumulators**
  - **limited by number and throughput of functional units**

This is Figure 5.30 from the textbook.

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5

Based on a slide supplied by CMU.

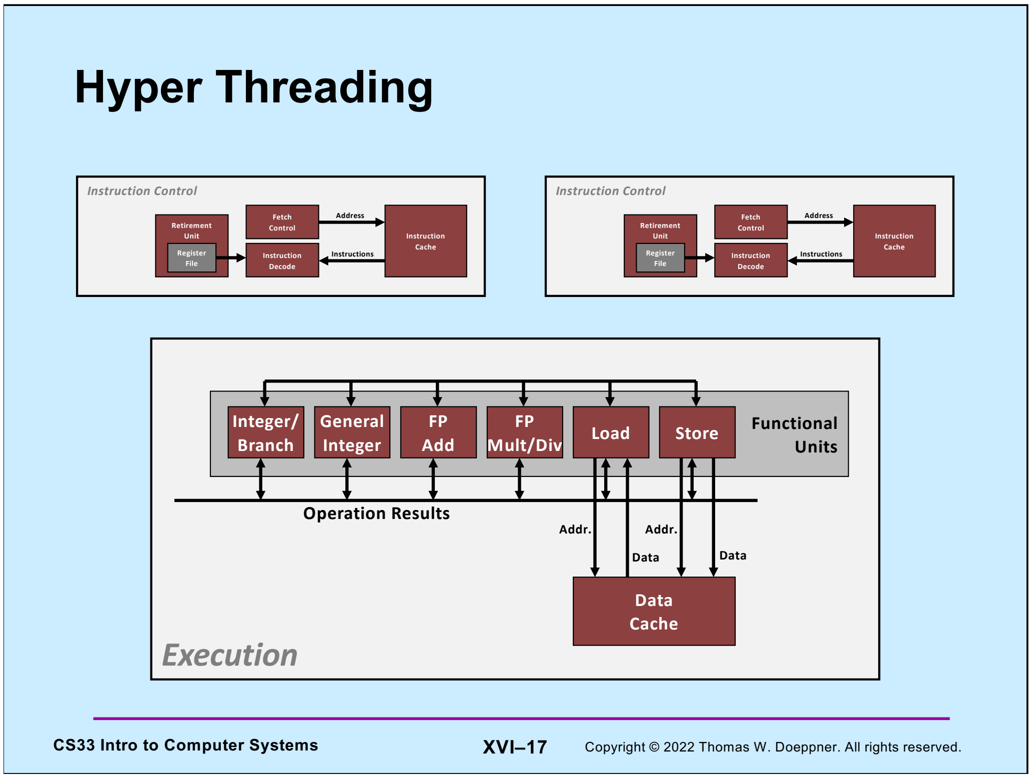
Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.0	3.0	5.0
Achievable Scalar	.52	1.01	1.01	.54
Latency bound	1.00	3.00	3.00	5.00
Throughput bound	.25	1.00	1.00	.5
Achievable Vector	.05	.24	.25	.16
Vector throughput bound	.06	.12	.25	.12

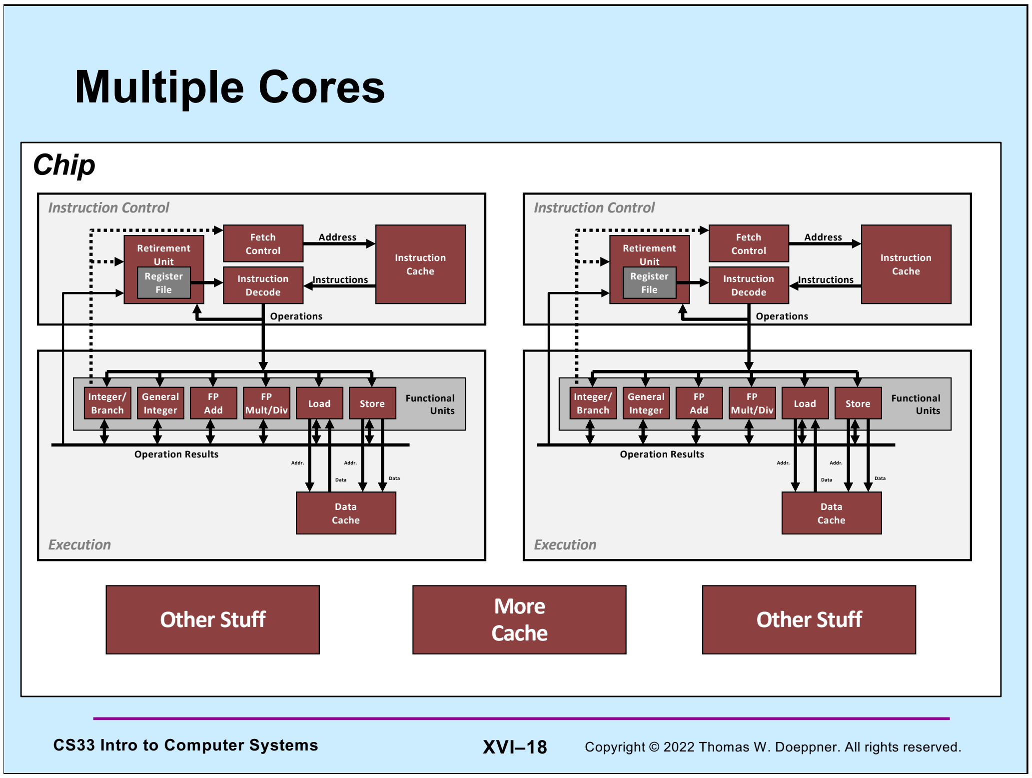
- **Make use of SSE Instructions**
  - parallel operations on multiple data elements

Based on a slide supplied by CMU.

SSE stands for "streaming SIMD extensions". SIMD stands for "single instruction multiple data" – these are instructions that operate on vectors.



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.



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.

This is the first of two lectures on memory hierarchy. The second, covering secondary storage (disk, etc.) will be given in a few weeks.

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

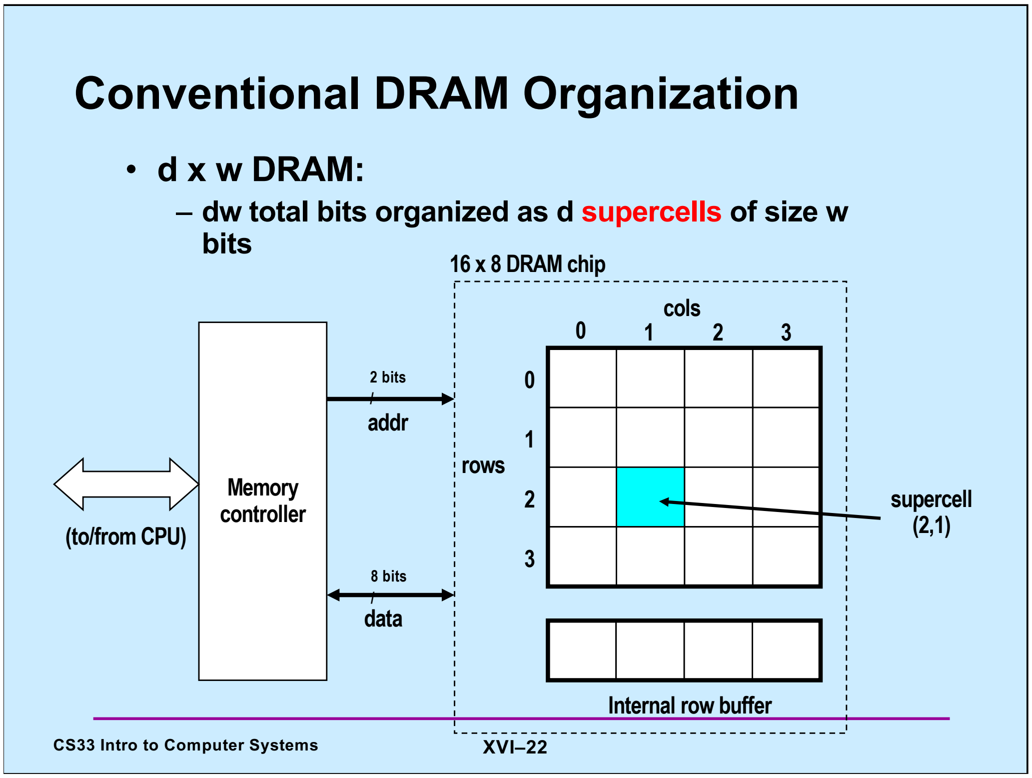
Supplied by CMU.

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.



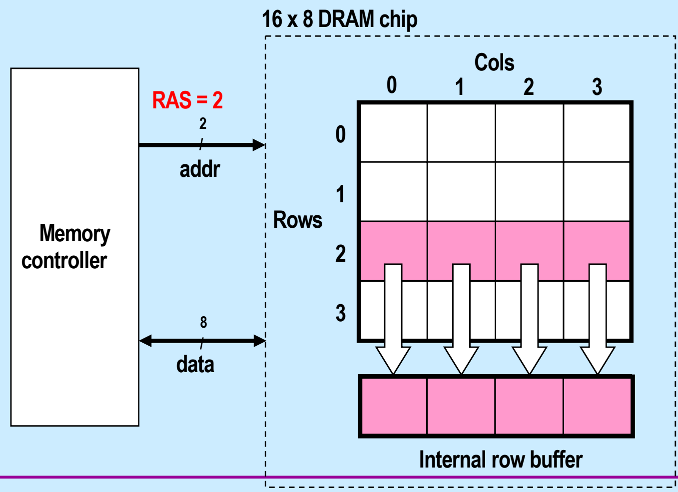
Supplied by CMU.

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

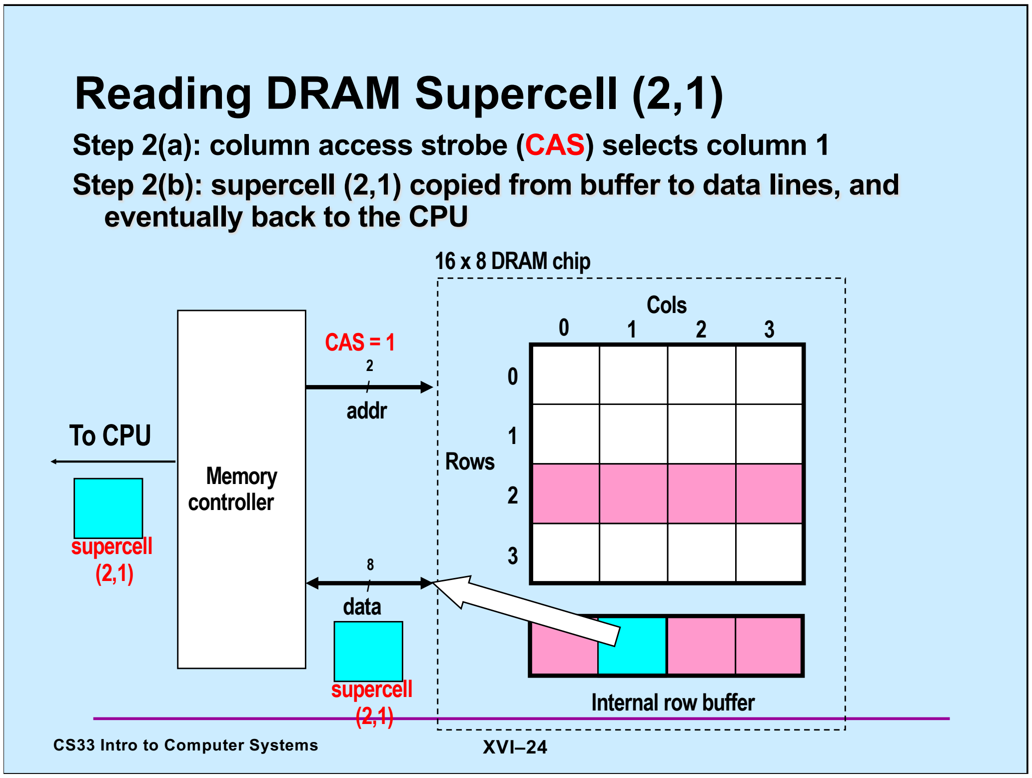
Reading DRAM Supercell (2,1)

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

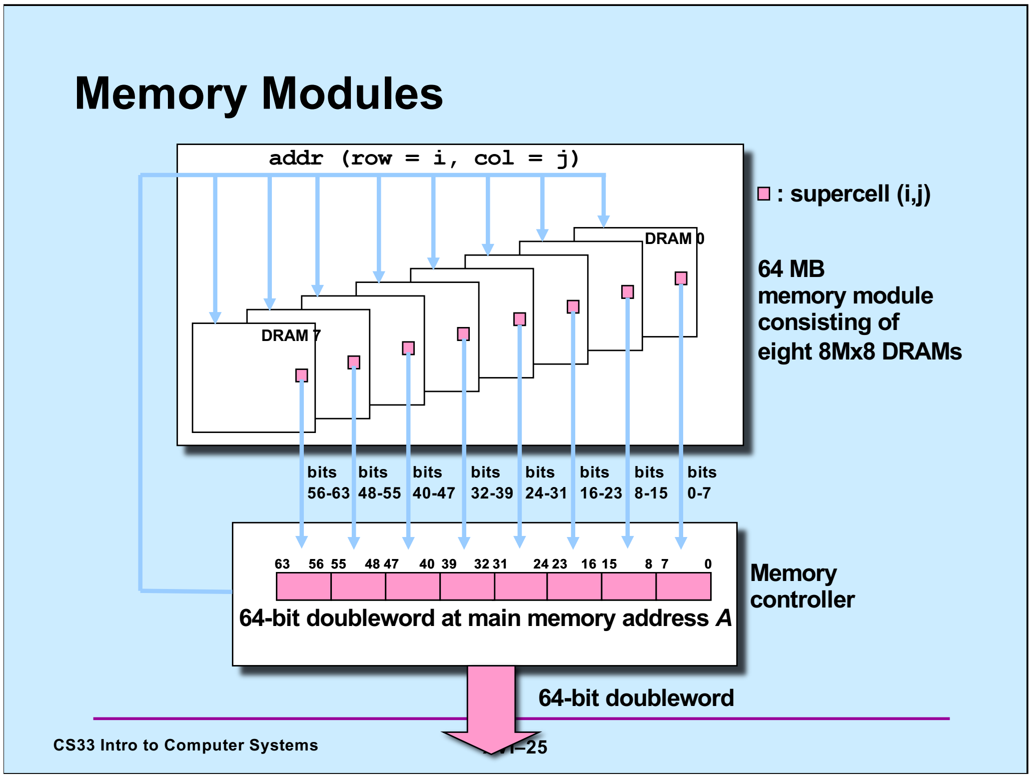
Step 1(b): row 2 copied from DRAM array to row buffer



Supplied by CMU.



Supplied by CMU.



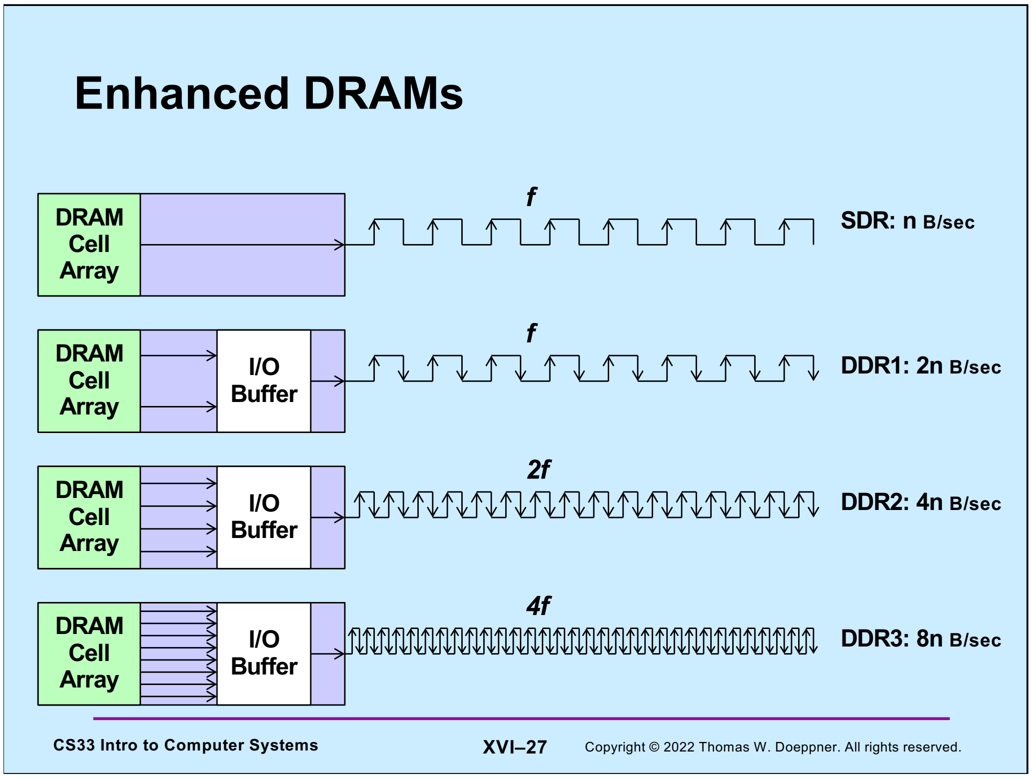
Supplied by CMU.

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 or SDR)
    - » 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: 16 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR2
      - 4 times as fast: 32 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR3
      - 8 times as fast: 64 consecutive bytes xfr'd as fast as 8 in SDR

Adapted from a slide supplied by CMU.



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. DDR4 takes this a step further and delivers 16 supercells 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).

DDR4

- **Memory transfer speed increased by a factor of 16 (twice as fast as DDR3)**
  - **no increase in DRAM Cell Array speed (same as SDR)**
  - **16 times more data transferred at once**
    - » **64 adjacent bytes fetched from DRAM**
      - **just like DDR3**

DDR4 memory became available in 2015. It's 16 times as fast as SDRAM, but transfers 64 consecutive bytes at a time, the same as DDR3. DDR5 is currently being discussed.

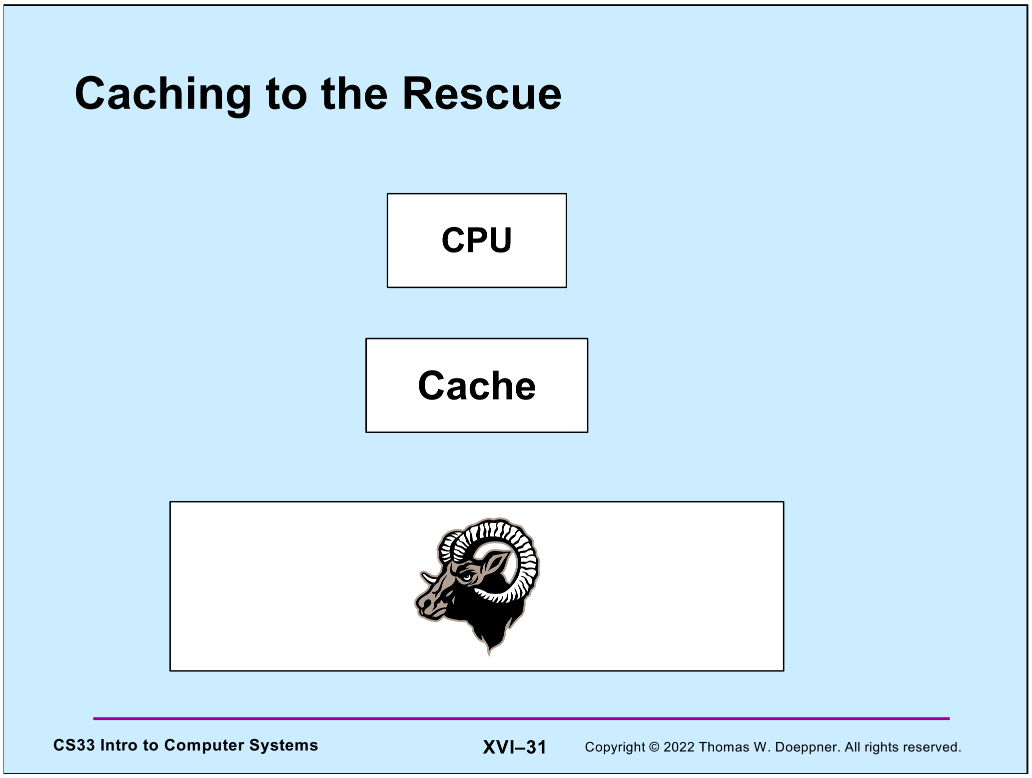
Quiz 2

A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR4 system at a speed that's n times that of an SDR system, where n is:

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

A Mismatch

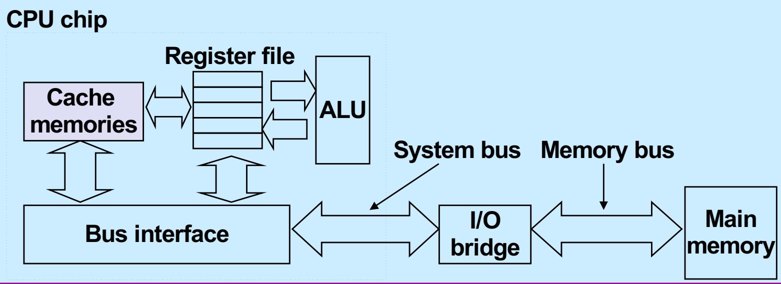
- A processor clock cycle is ~0.3 nsecs
  - SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?



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.

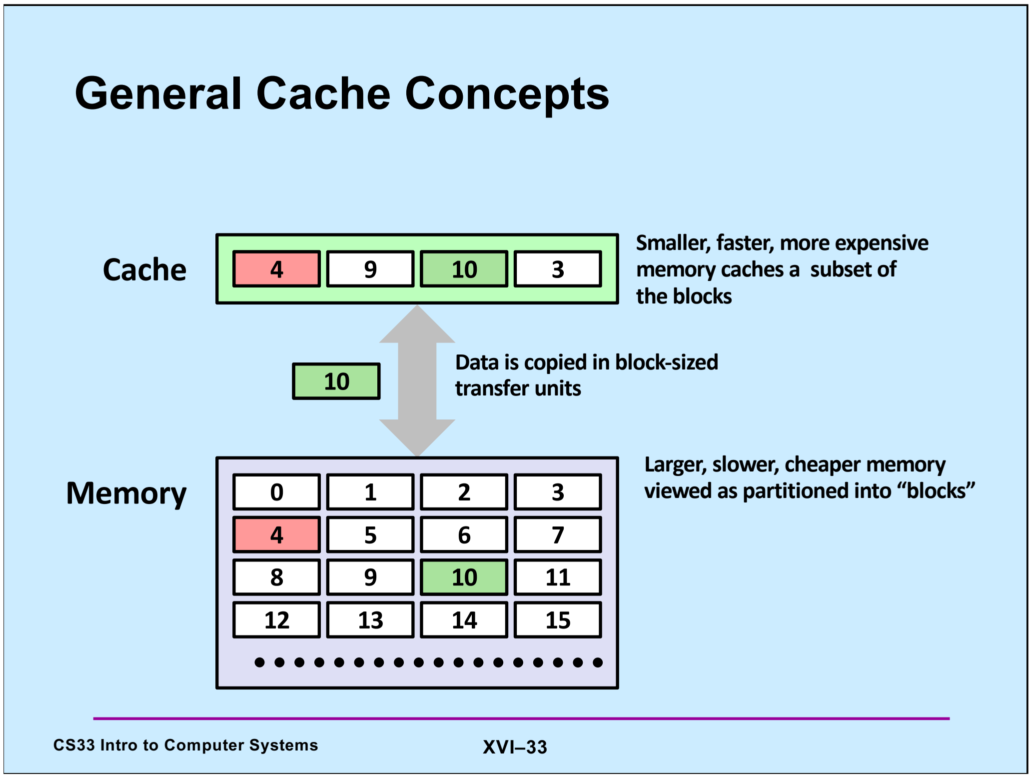
Cache Memories

- 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:

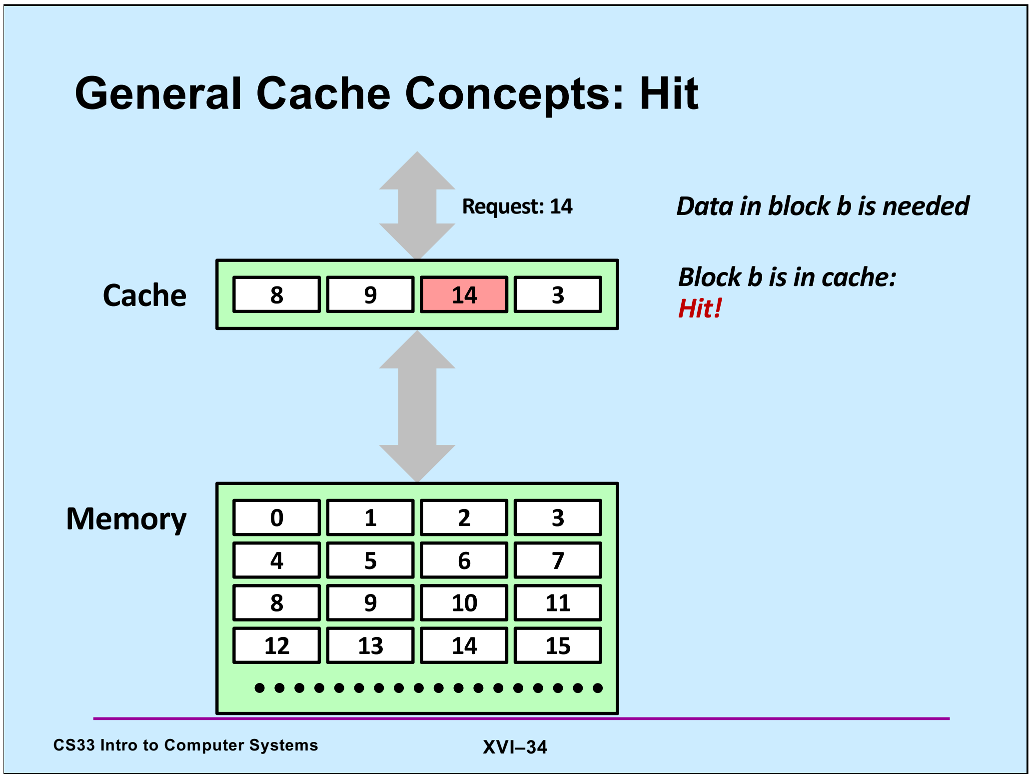


Supplied by CMU.

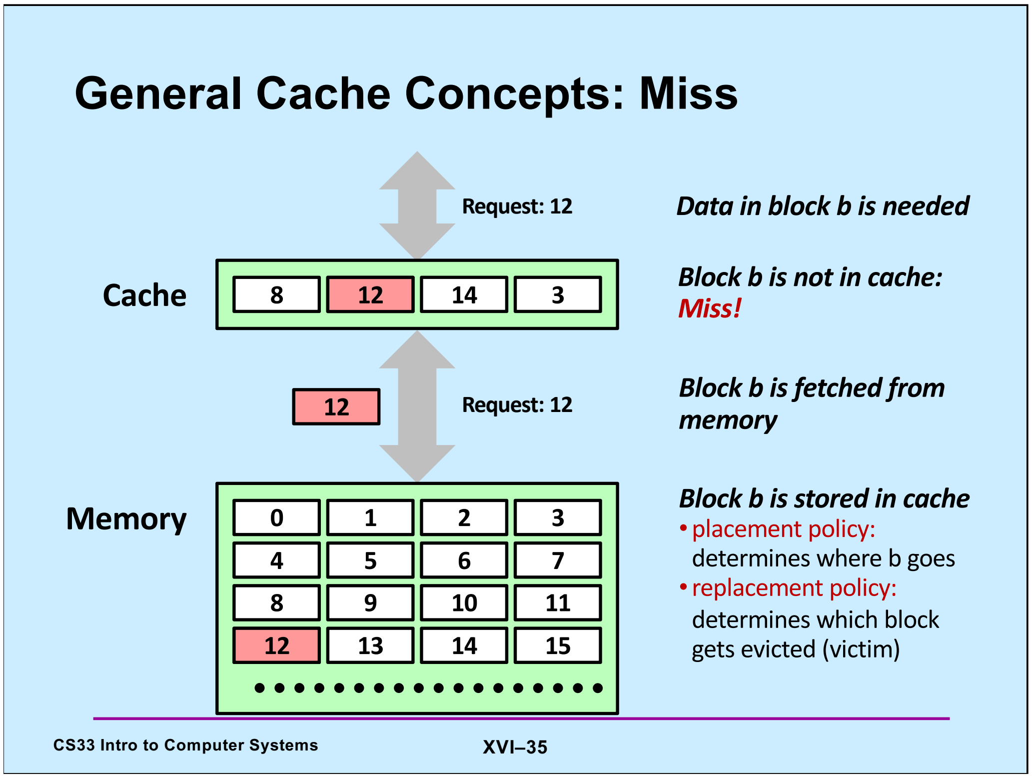
"ALU" (arithmetic and logic unit) is a traditional term for the instruction and execution units of a processor.



Supplied by CMU.



Supplied by CMU.

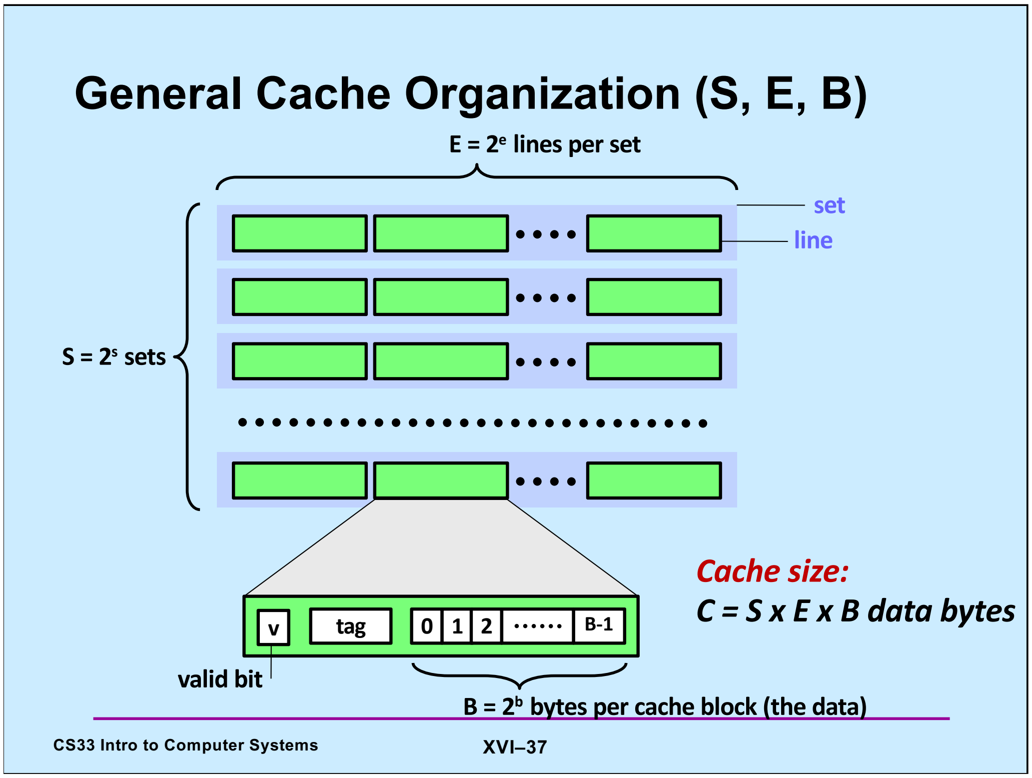


Supplied by CMU.

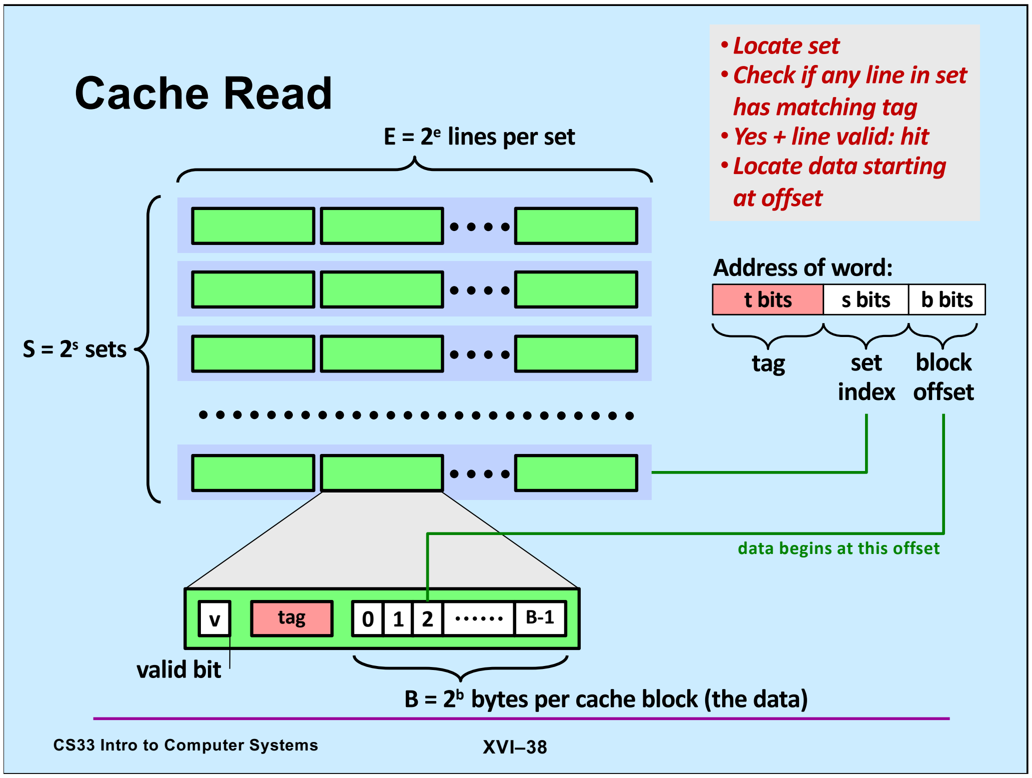
General Caching Concepts: Types of Cache Misses

- **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 mod 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

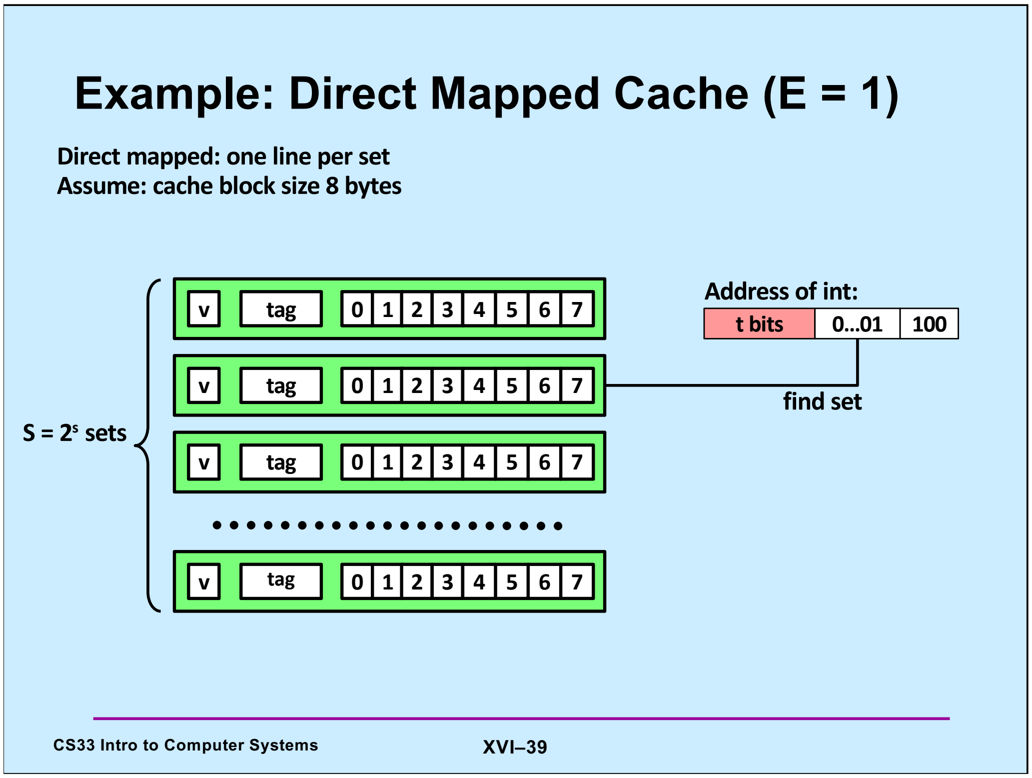
Supplied by CMU.



Supplied by CMU.



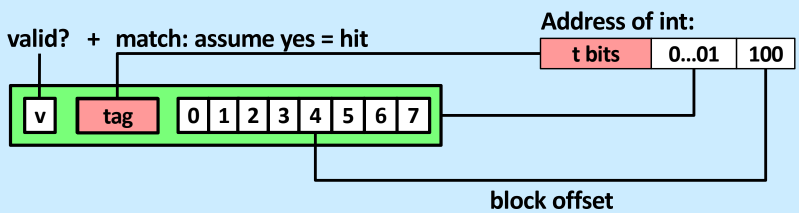
Supplied by CMU.



Supplied by CMU.

Example: Direct Mapped Cache (E = 1)

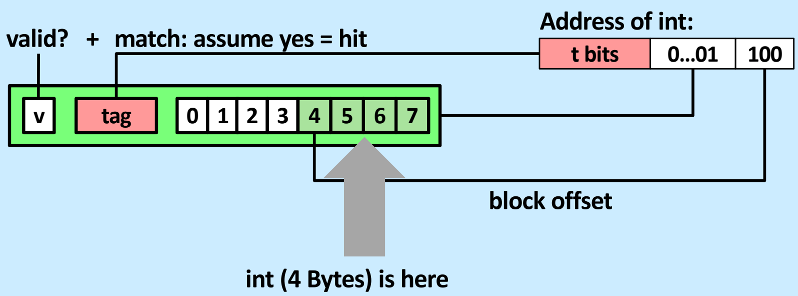
Direct mapped: one line per set
Assume: cache block size 8 bytes



Supplied by CMU.

Example: Direct Mapped Cache (E = 1)

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



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	**$[0\underline{00}0_2]$,**	miss
1	**$[0\underline{00}1_2]$,**	hit
7	**$[0\underline{11}1_2]$,**	miss
8	**$[1\underline{00}0_2]$,**	miss
0	**$[0\underline{00}0_2]$**	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

Ignore the variables sum, i, j

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


```
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;
}
```

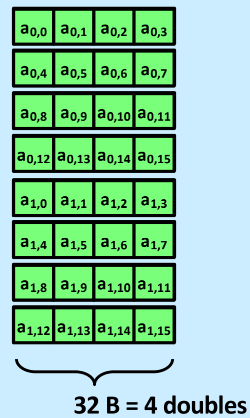
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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


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



Note that the cache holds two rows of the matrix; each cache block holds four doubles. When a[0][0] is read, so are a[0][1] through a[0][3]. Thus, after one cache miss, we get three hits.

A Higher-Level Example

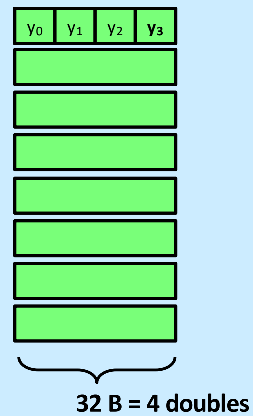
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

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

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

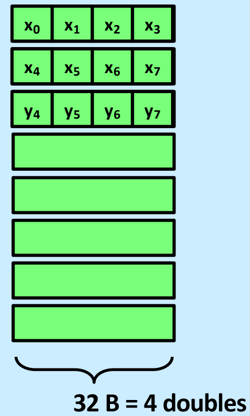
```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  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 the loop is executed very slowly — each access to either array results in a conflict miss.

Different Alignments

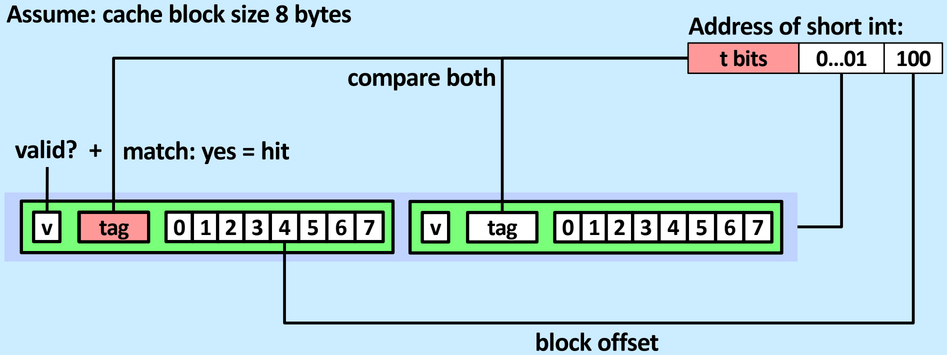
```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  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)

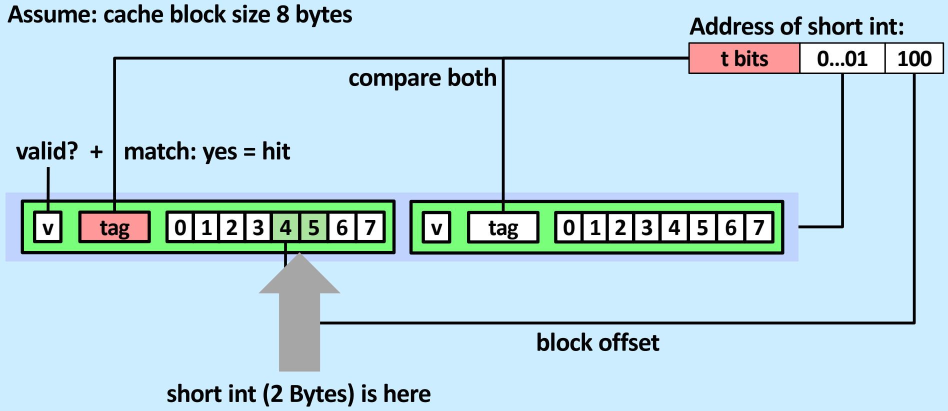
E = 2: two lines per set
Assume: cache block size 8 bytes



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:

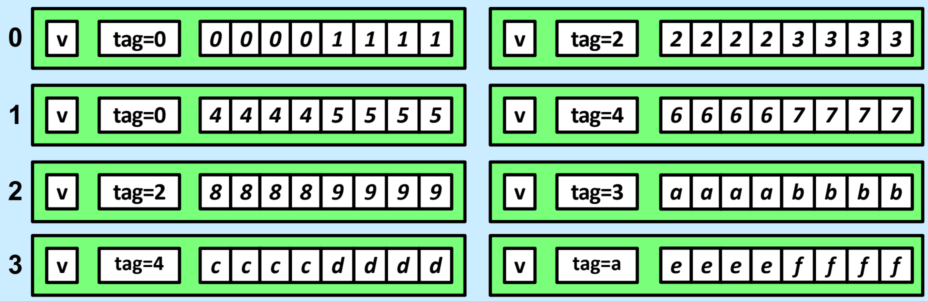
- **One line in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU), …**

Supplied by CMU.

Quiz 3

Address of int:

100	01	100



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	**$[00\underline{0}0_2]$,**	miss
1	**$[00\underline{0}1_2]$,**	hit
7	**$[01\underline{1}1_2]$,**	miss
8	**$[10\underline{0}0_2]$,**	miss
0	**$[00\underline{0}0_2]$**	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.

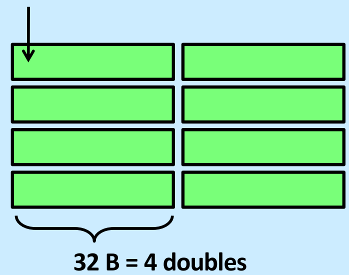
A Higher-Level Example

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

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



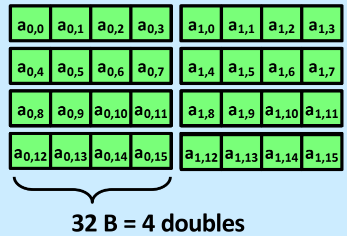
Supplied by CMU.

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

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



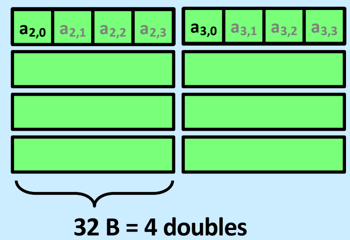
The cache still holds two rows of the matrix, but each row may go into one of two different cache lines. In the slide, the first row goes into the first lines of the cache sets, the second row goes into the second lines of the cache sets.

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;
}
```

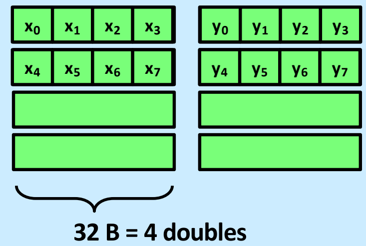
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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



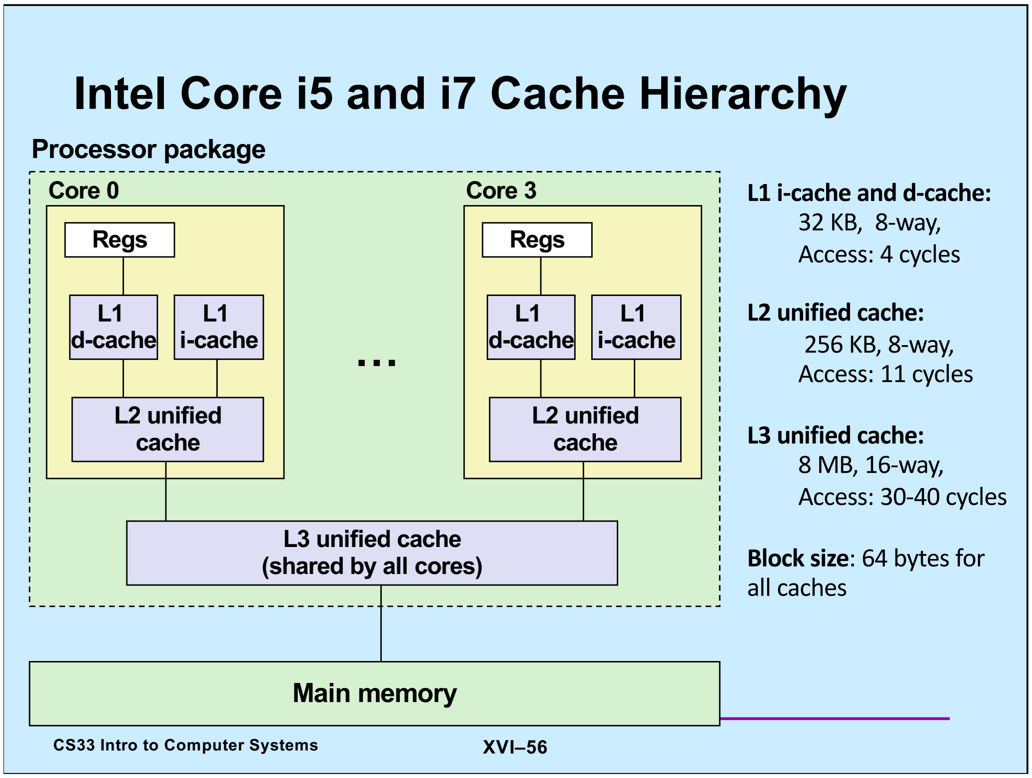
There is still a cache miss on each access.

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 cores.

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.