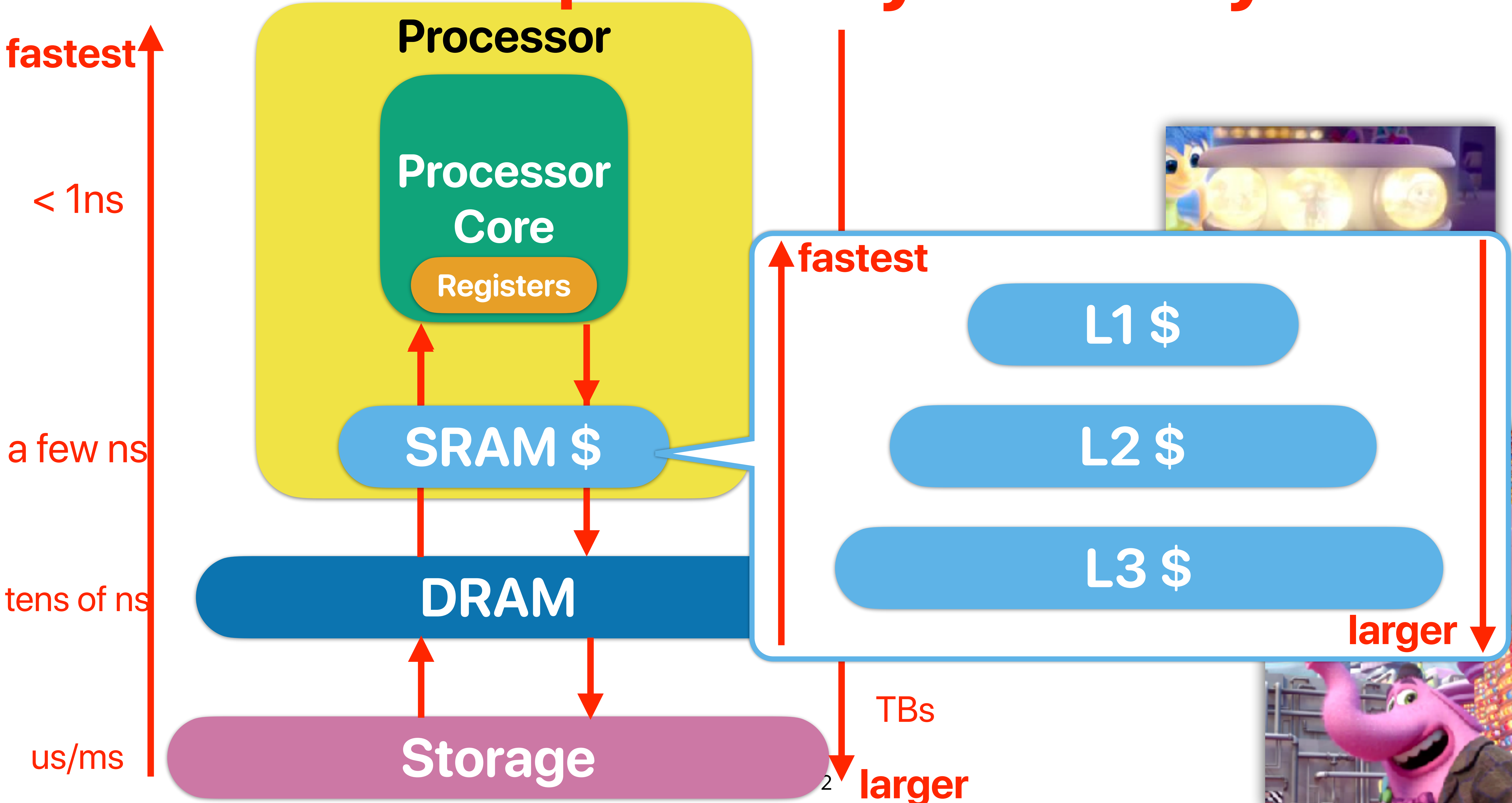
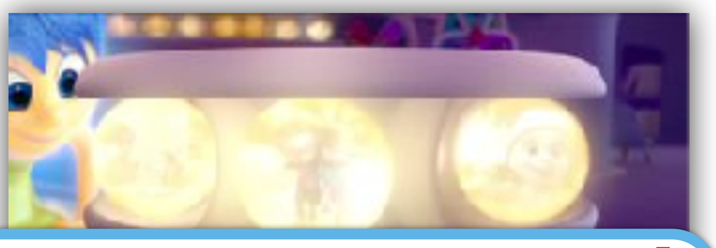
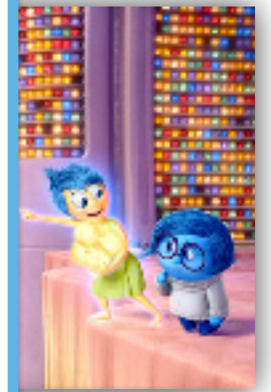
Lab 3: Memory

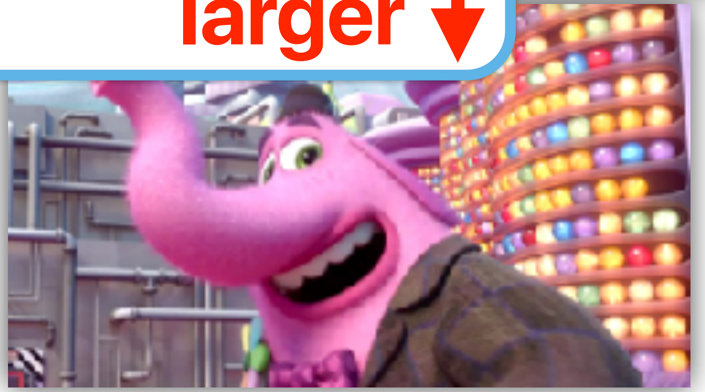
Hung-Wei Tseng

Recap: Memory Hierarchy

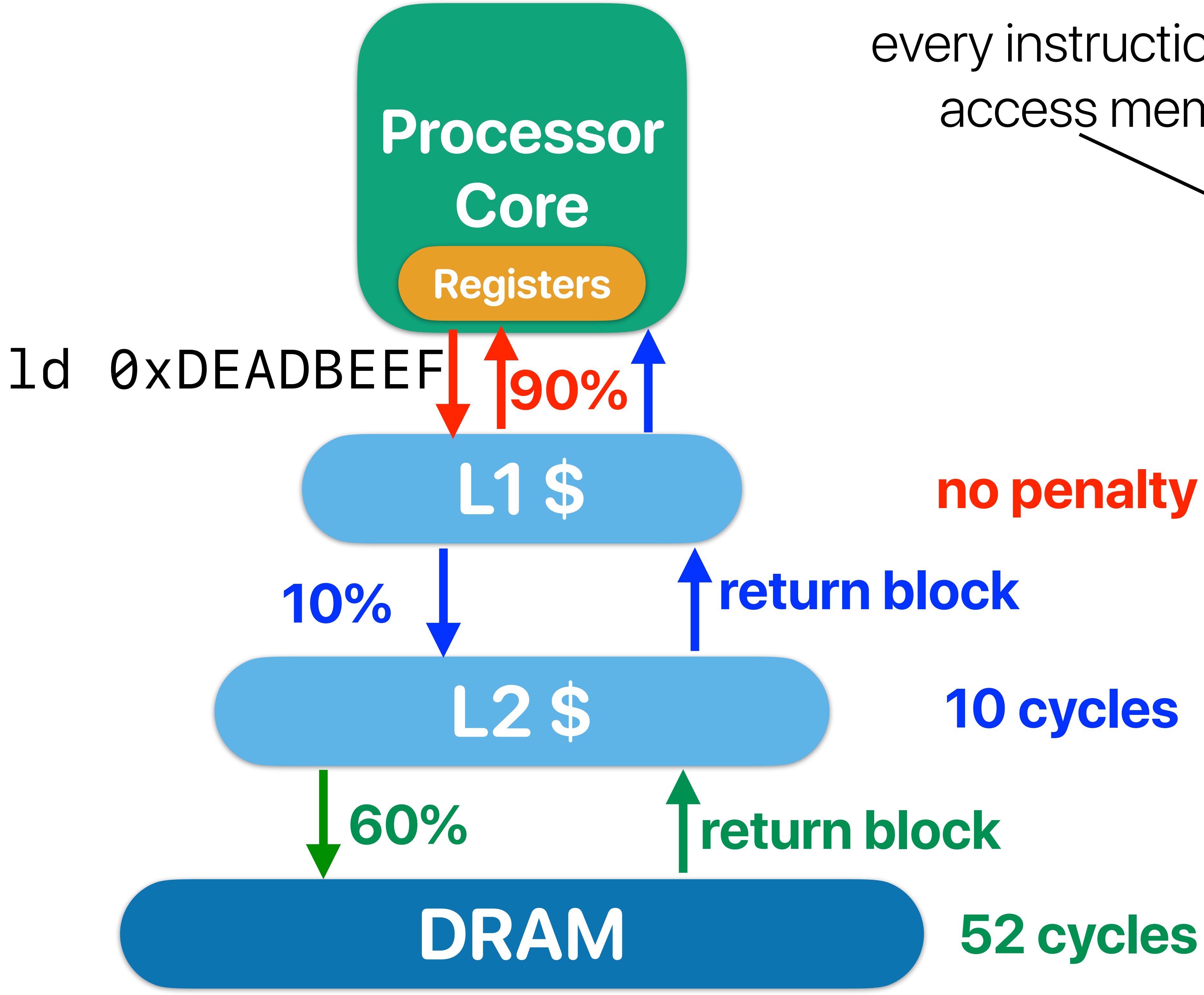








Performance evaluation with multi-level $



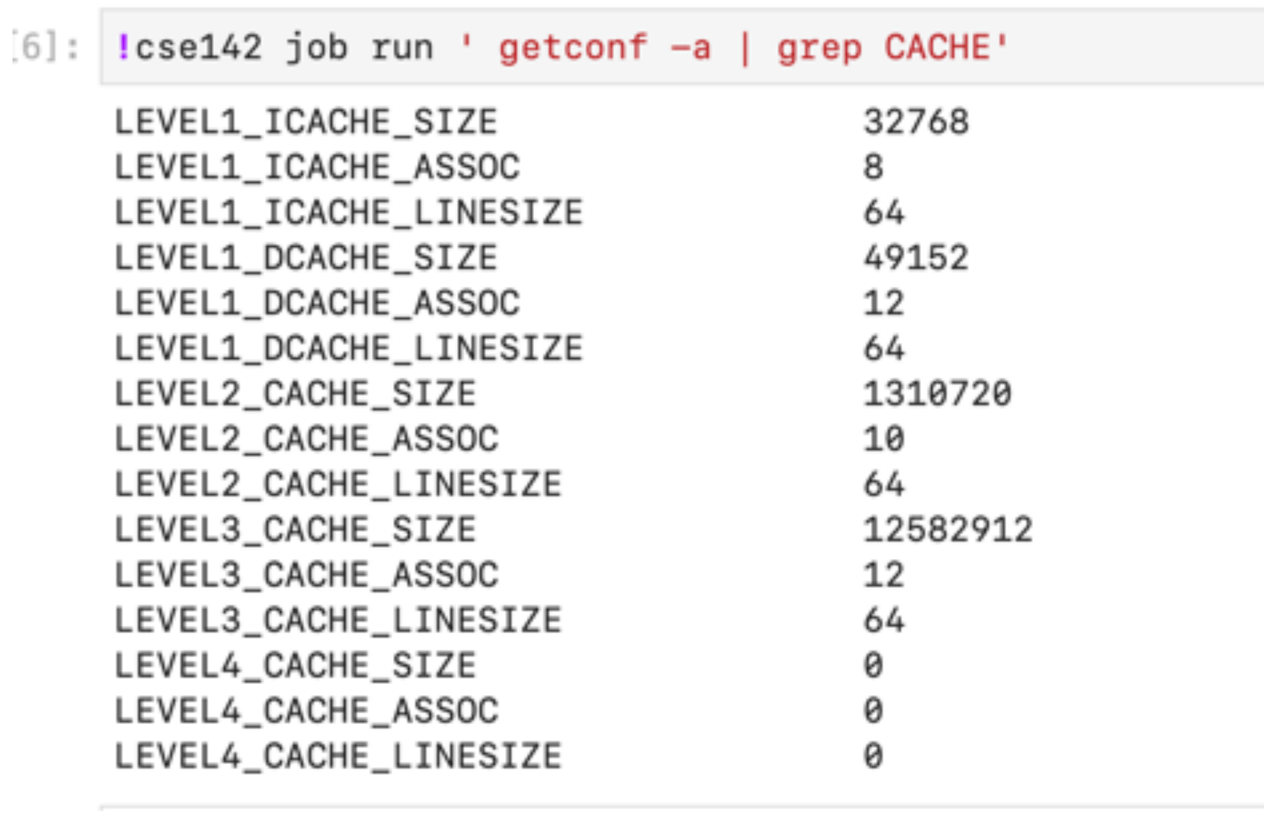
every instruction fetch access memory

assume 30% data accesses

$$1 + (1 + 30\%) \times (1 - 90\%) \times [10 + (1 - 60\%) \times 52] = 5\ cycles$$

Know your cache configuration

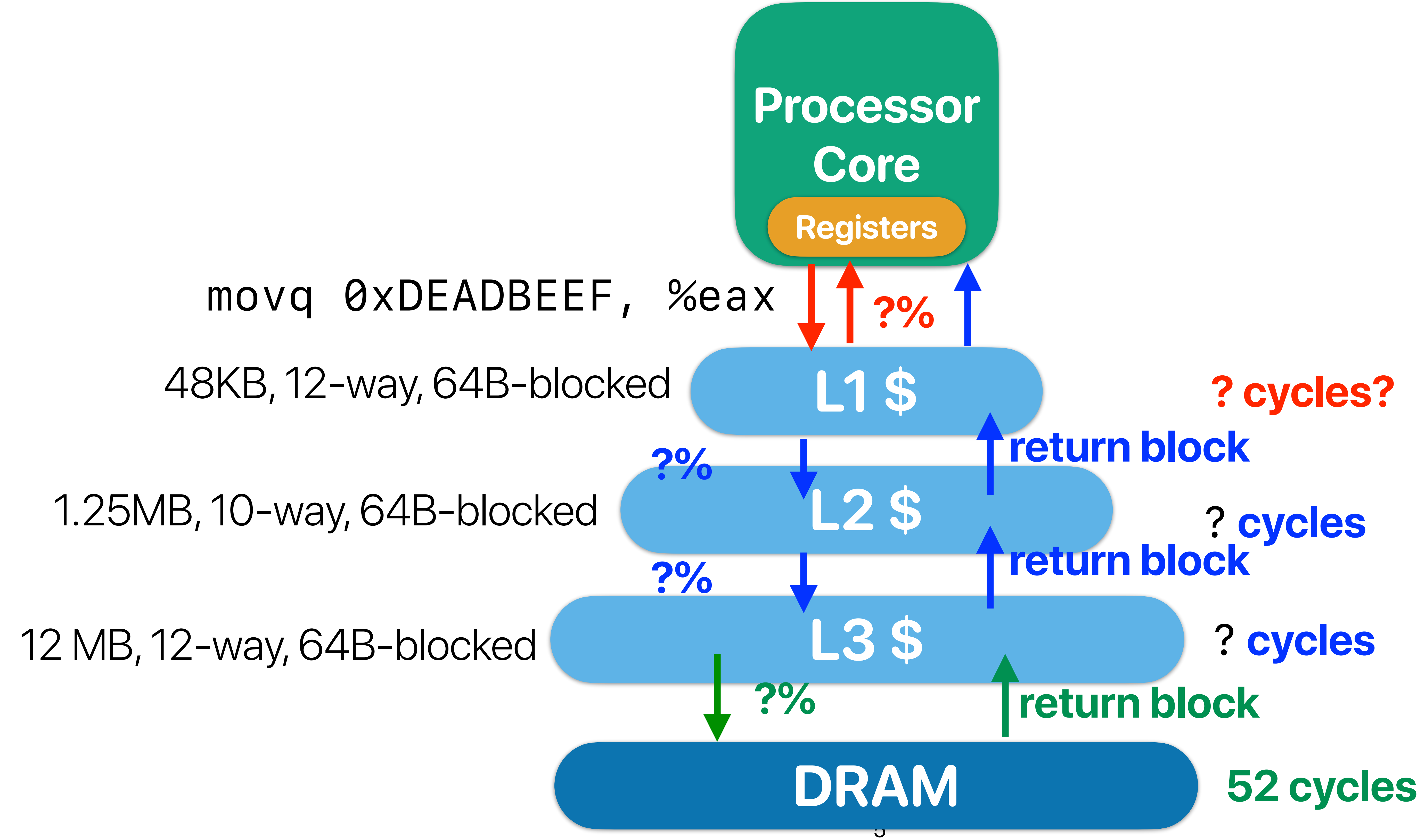
- getconf -a | grep CACHE



```
[6]: !cse142 job run ' getconf -a | grep CACHE'
```

```
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 10
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  12582912
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Q4 & Q5: Performance with multi-level $



How fast is my memory hierarchy?

How are you going to implement a tool that can test cache performance?

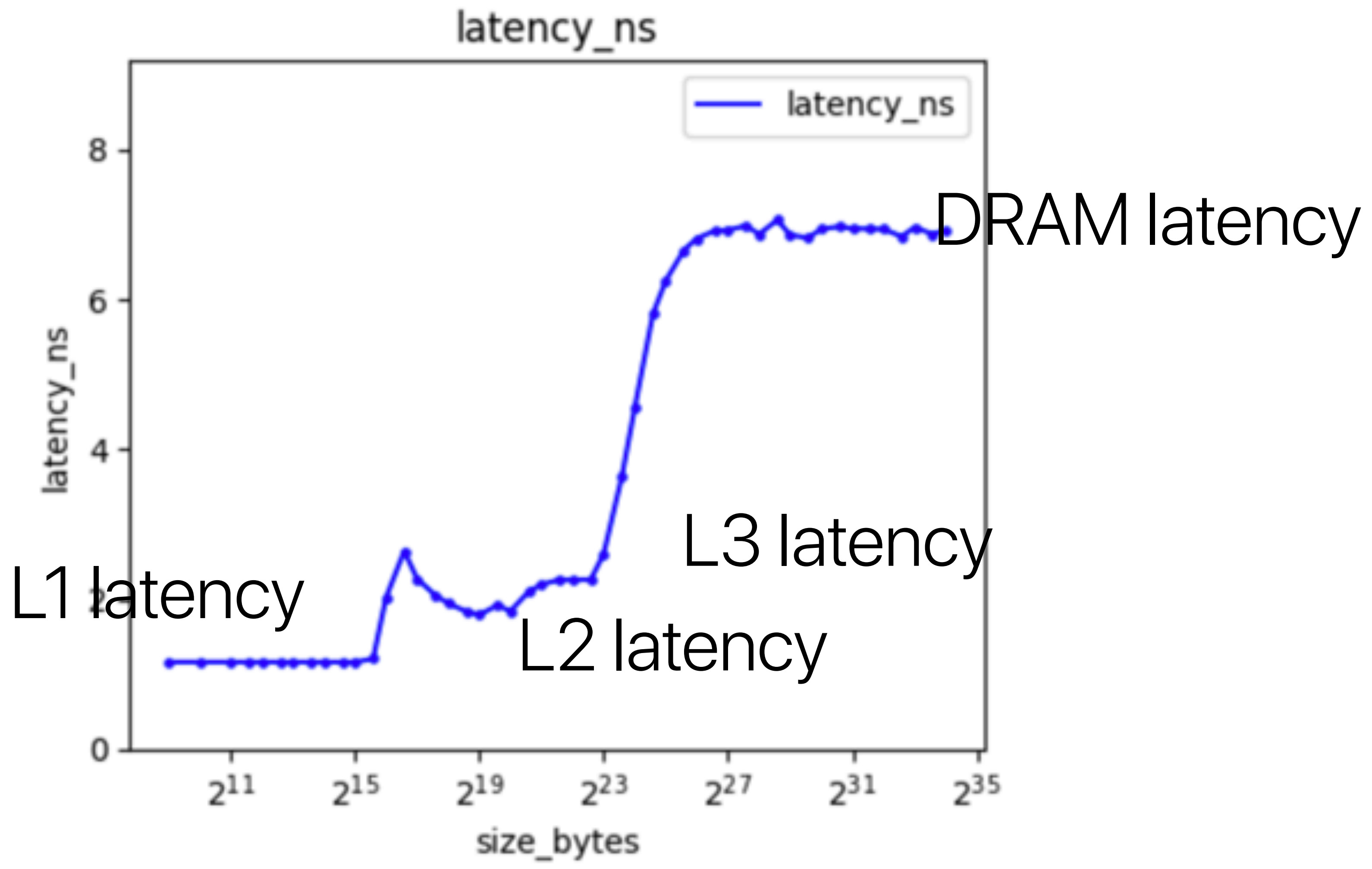
- create cache accesses that
  - Does not miss — L1 access latency
  - Only miss on L1, but not L2 — L2 access latency
  - Misses on L1 & L2, but not L3 — L3 access latency
  - Misses on L3 — DRAM access latency\
- But we don’t know the size of each
  - We need to test difference sizes
  - We need to?

The tool

- Memory Latency
  - https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html
- lat_mem_rd
- Spec
  - dmidecode — for DRAM parameters
  - lscpu — for on-CPU caches

lat_mem_rd

```
[10]: #df = render_csv("lat_mem_rd.csv")
      df = pd.read_csv("lat_mem_rd.csv", sep=",")
      df["size_bytes"] = df["size_MB"]*1024*1024
      plotPE(df=df, what=[("size_bytes", "latency_ns")], lines=True, logx=2,  log_autoscale_x=False, log_autoscale_y=False)
```



```
[6]: !cse142 job run ' getconf -a | grep CACHE'

LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 10
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  12582912
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Memory performance when running applications

How big is a struct?

Q3: The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

  What's the output of

  `printf("%lu\n",sizeof(struct student))?`

  A. 20

  B. 28

  C. 32

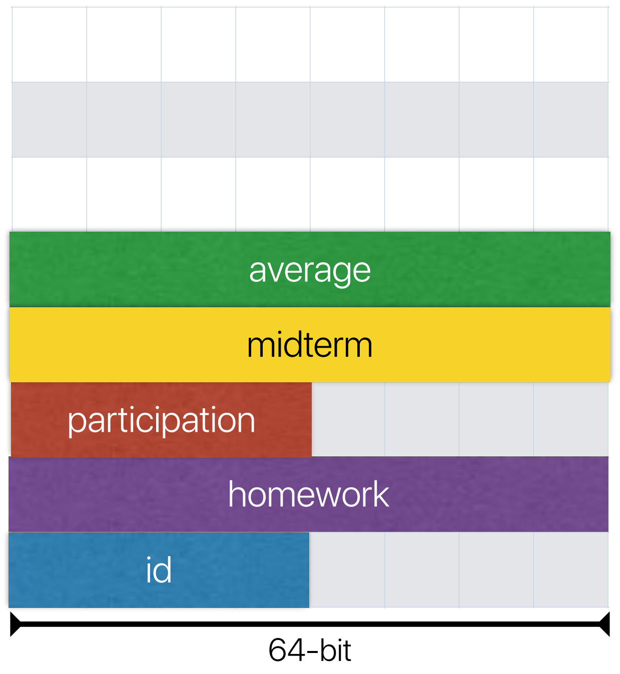
  D. 36

  E. 40

The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```



What's the output of
`printf("%lu\n",sizeof(struct student))?`

A. 20

B. 28

C. 32

D. 36

[✓ E. 40]

Memory addressing/alignment

- Almost every popular ISA architecture uses "byte-addressing" to access memory locations
- Instructions generally work faster when the given memory address is aligned
  - Aligned — if an instruction accesses an object of size n at address X, the access is **aligned** if X **mod** n **= 0**.
  - Potentially incurs two cache misses for an access
  - Some architecture/processor does not support aligned access at all
  - Therefore, compilers only allocate objects on "aligned" address

Memory addressing/alignment

- Unaligned accesses are sometimes inefficient: one load can cause 2 cache misses
- I haven't been able to demonstrate this on our machines
  - No effect for L1.
  - Measurement noise for L2/L3/DRAM is too large
- Some versions of ARM's ISA have very complicated semantics for unaligned accesses
- Other ISAs
  - Unaligned access can cause an interrupt.

Locality

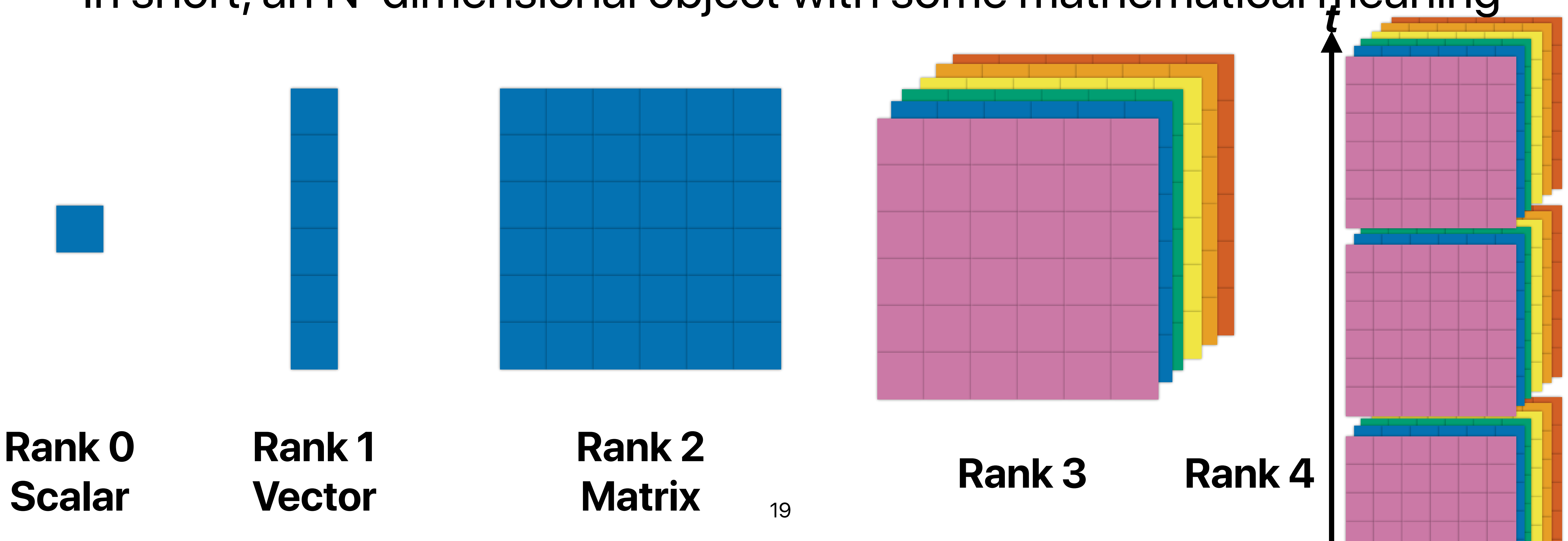
- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — the current element in an array, then the next
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

If we want to improve the memory performance of our code, we have to make our code better exploit "localities"

How “Tensors” are layed out?

Tensors

- A tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space.
- In short, an N-dimensional object with some mathematical meaning



Q6: Tensor Layout

- Internally, tensors are stored a linear array
  - size = x*y*z*b
- Here's the code to translate coordinates to a linear index (tensor_t.hpp)
  - Increment x moves the index by 1
  - Incrementing y moves by size.x
  - Incrementing z moves by size.x*size.y
  - Incrementing b moves by size.x*size.y*size.z

```
index = b * (size.x * size.y * size.z) +
z * (size.x * size.y) +
y * (size.x) +
x;
```

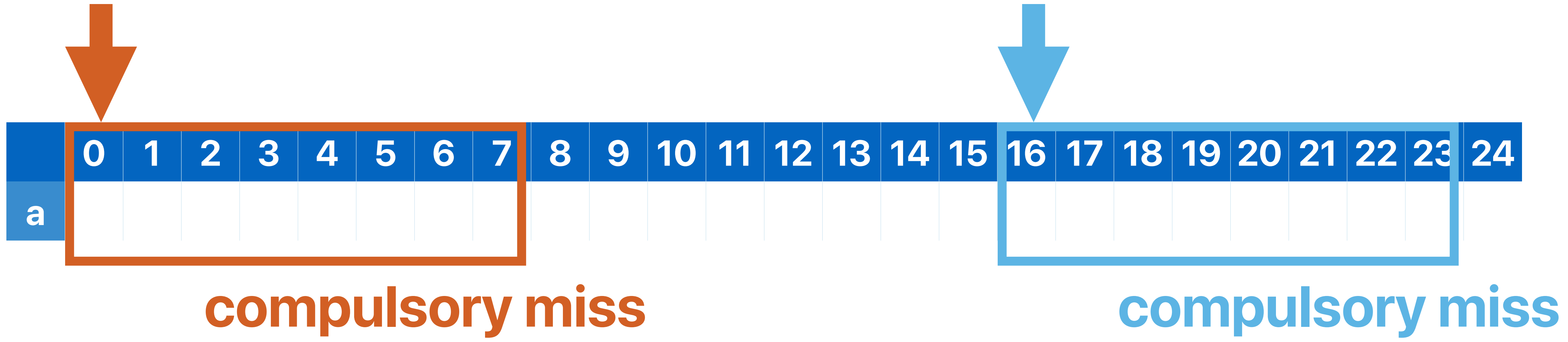
Q7: cache performance on the following code

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(i = 0; i < size; i+=stride) {
    sum += a[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code when stride = 16 and size = 8192?

A. 0%
B. 6.25%
C. 25%
D. 50%
E. 100%



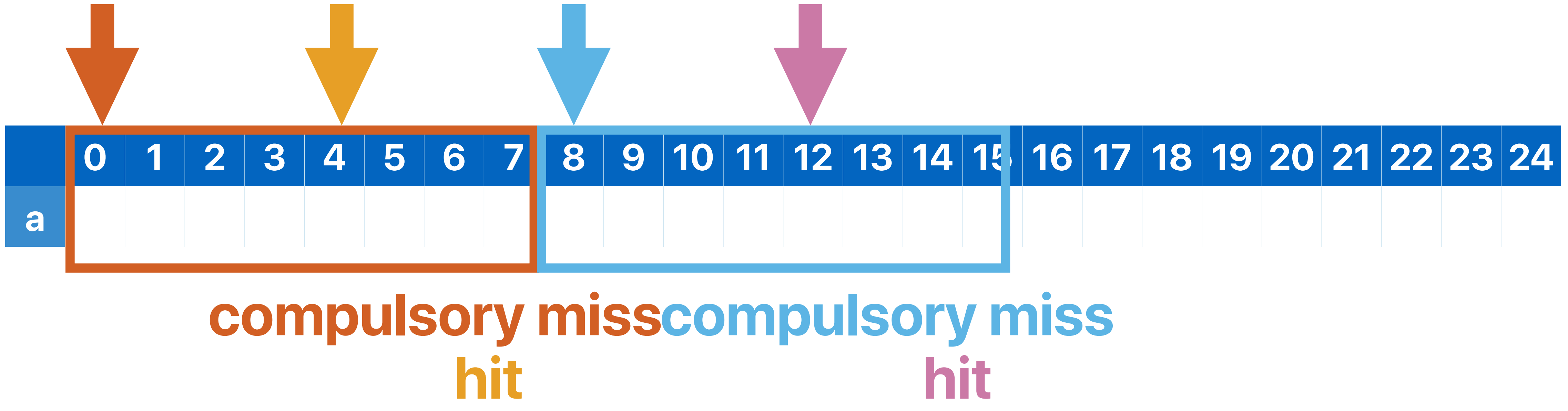
Q7: cache performance on the following code

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(i = 0; i < size; i+=stride) {
    sum += a[i];
    //load a, b, and then store to c
}
```

What's the data cache miss rate for this code when stride = 4 and size = 8192?

A. 0%
B. 6.25%
C. 25%
D. 50%
E. 100%



Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — the current element in an array, then the next
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

If we want to improve the memory performance of our code, we have to make our code better exploit "localities"

Run more…

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(j = 0; j < 1000)
    for(i = 0; i < size; i+=stride) {
        sum += a[i];
        //load a, b, and then store to c
}
```

What's the data cache miss rate for this code when stride = 16 and size = 8192?

A. 0%

B. 6.25%

C. 25%

D. 50%

E. 100%

data visited $\dfrac{\dfrac{8192 \times 8}{64}}{\dfrac{16}{8}} = 512$ blocks

$\dfrac{49152}{64} = 768$ blocks

Go larger...

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(j = 0; j < 1000)
    for(i = 0; i < size; i+=stride) {
        sum += a[i];
        //load a, b, and then store to c
}
```

What's the data cache miss rate for this code when stride = 16 and size = 32768?

A. 0%

B. 6.25%

C. 25%

D. 50%

E. 100%

Go larger...

- D-L1 Cache configuration of intel Core i7 processor
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384];
/* a = 0x20000 */
for(j = 0; j < 1000)
    for(i = 0; i < size; i+=stride) {
        sum += a[i];
        //load a, b, and then store to c
}
```

What's the major cause of data cache misses this code when stride = 16 and size = 32768?

A. Compulsory misses

B. Conflict misses

C. Capacity misses

Working set size

- The portion of memory that the program is currently using

Conflict misses

Q11 & Q12: Demo

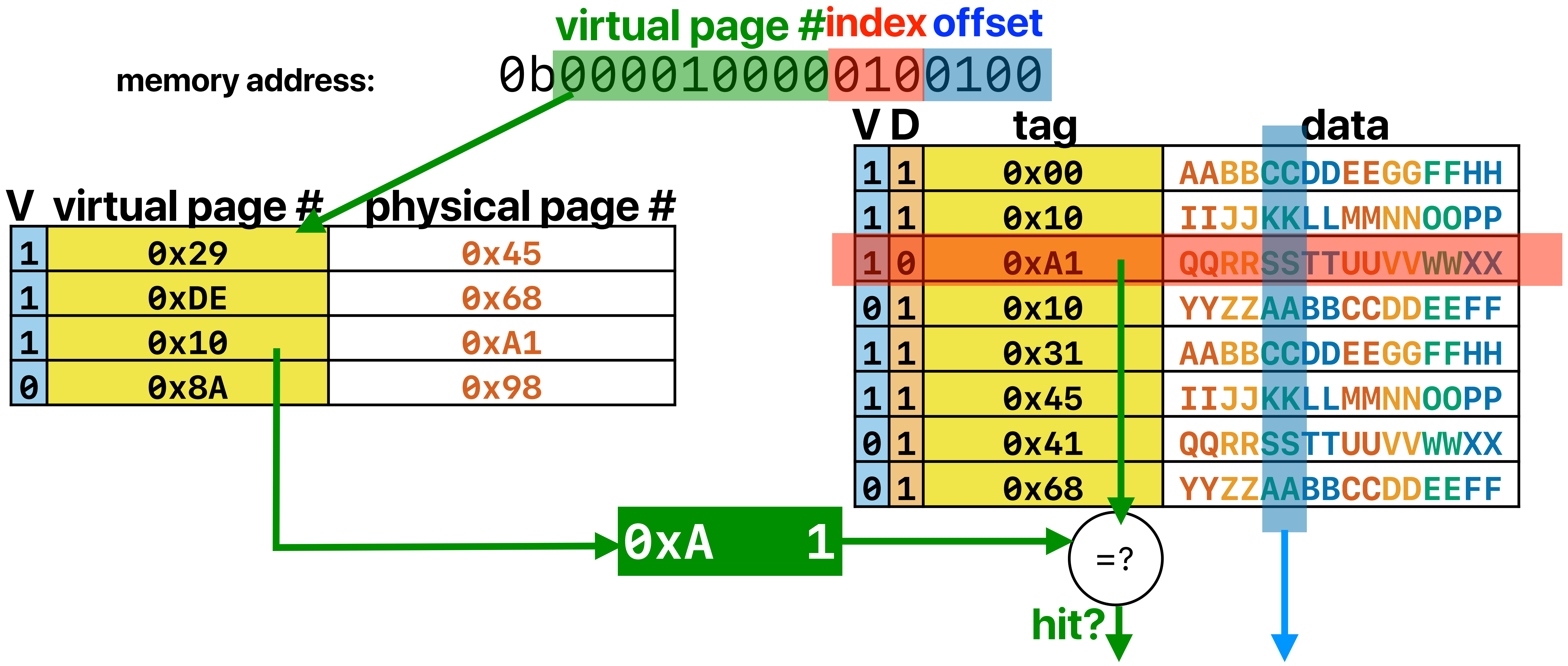
- Conflict misses occurs because blocks are mapped to the same index

Remember, we also have virtual memory

Memory hierarchy in practice: TLB + Cache

memory address: 0x0 8 2 4

memory address: 0b00001000000100100

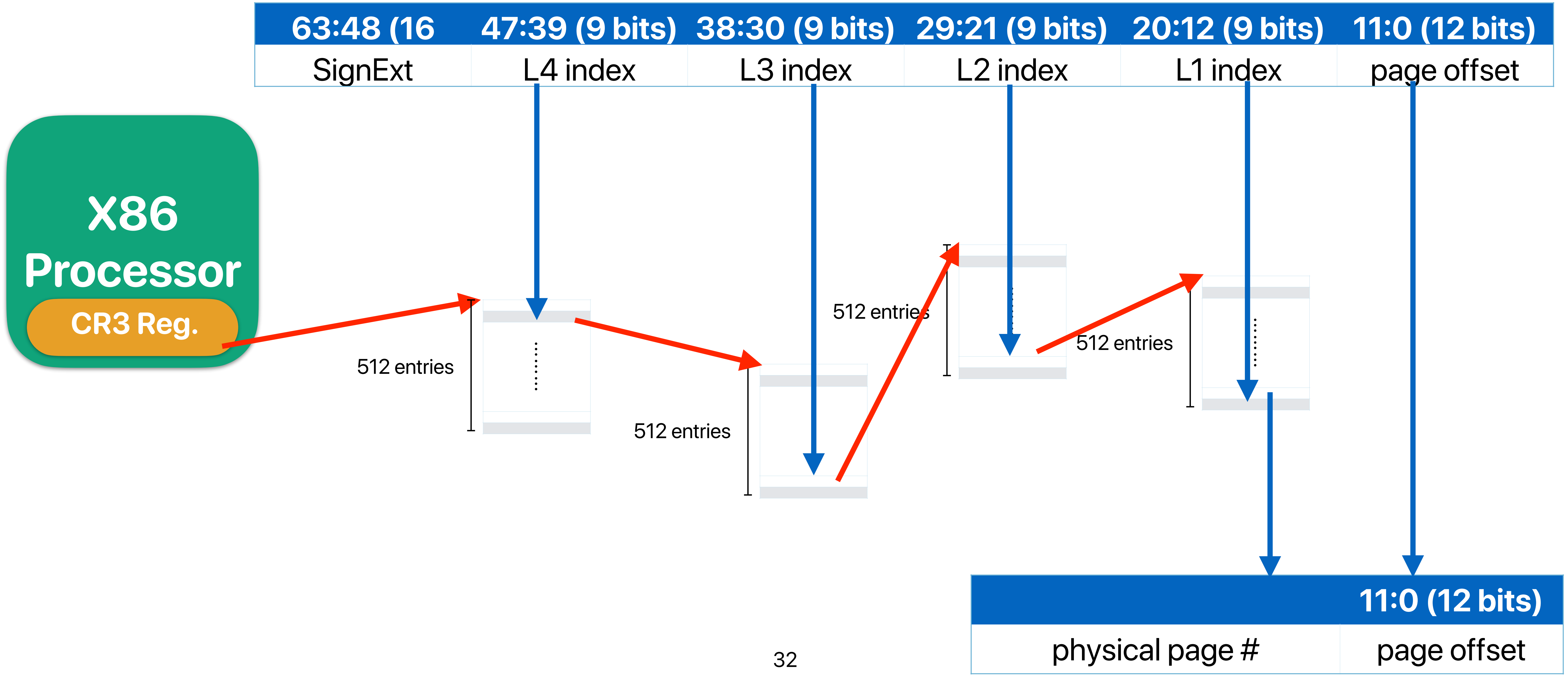


V	virtual page #	physical page #
1	0x29	0x45
1	0xDE	0x68
1	0x10	0xA1
0	0x8A	0x98

V	D	tag	data
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF

What happen on a TLB miss?

- Page table look up — how many memory accesses?



TLB.cpp

```
//START
#include<cstdint>
#include<cstdlib>
#include<vector>
#include<algorithm>
#include"function_map.hpp"
#include <sys/mman.h>

template<size_t BYTES>
struct MM {
    struct MM* next;  // I know that pointers are 8 bytes
    uint64_t junk[BYTES/8 - 1]; // This forces the struct
};


template<class MM>
MM *  __attribute__ ((noinline))  miss(MM * start, uint64
    for(uint64_t i = 0; i < count; i++) { // Here's the l
        start = start->next;
    }
    return start;
}
```

```
//
template<size_t BYTES>
uint64_t* TLB(uint64_t * data, uint64_t size, uint64_t arg1) {
    struct MM<BYTES> * array = NULL;
    int r =  posix_memalign(reinterpret_cast<void**>(&array), 4096, size);
     if (r == -1) {
         std::cerr << "posix_memalign() failed.  Exiting: " << strerror(errno) << "\n";
         exit(1);
    }

    r = madvise(reinterpret_cast<void*>(array), size, MADV_NOHUGEPAGE);
     if (r == -1) {
         std::cerr << "madvise() failed.  Exiting: " << strerror(errno) << "\n";
         exit(1);
    }

    std::cout << "array alignment is " << (reinterpret_cast<uintptr_t>(array) % 4096) << "\n";
    std::cout << "array size is " << size/BYTES << " element; " << size << "B\n";

    // This is clever part  'index' is going to determine where the pointers go.  We fill it consecutive integers.
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < size/BYTES; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < size/BYTES; i++) {
        array[index[i]].next = &array[index[(i + 1) % (size/BYTES)]];
    }

    MM<BYTES> * start = &array[0];
    start = miss(start, arg1); // 128 million accesses.
    return reinterpret_cast<uint64_t*>(start); // This is a garbage value, but if we don't return it, the compiler will optimize out the call to miss.
}
```

Optimizations for locality

Loop interchange/renesting

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

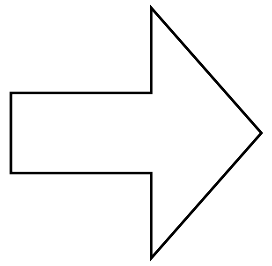
B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

$O(n^2)$	Complexity	$O(n^2)$
Same	Instruction Count?	Same
Same	Clock Rate	Same
Better	CPI	Worse

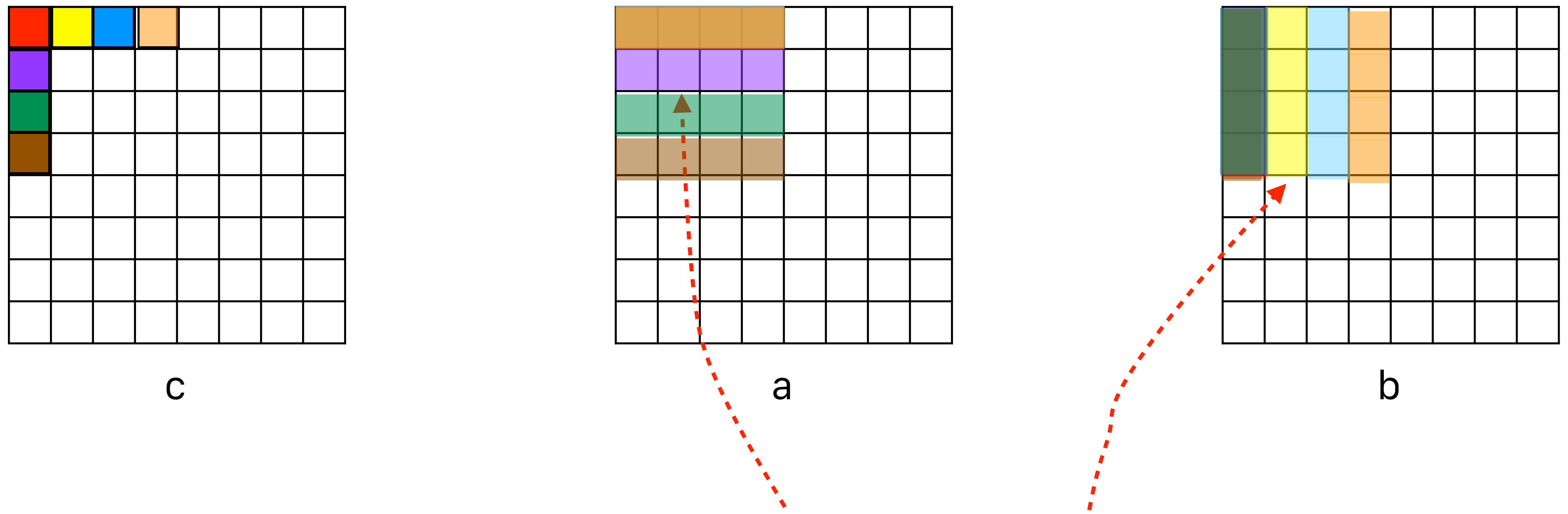
Tiling

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```



```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
              c[ii][jj] += a[ii][kk]*b[kk][jj];

    }
  }
}
```



You only need to hold these sub-matrices in your cache

Programming assignment

Matrix exp: $B = A^N$

```
#include "tensor_t.hpp"
#include "pin_tags.h"


template<typename T>
void __attribute__((noinline)) matexp_solution(tensor_t<T> & dst, const tensor_t<T> & A, uint32_t power){
              // parameters you can use for whatever purpose you want (e.g., tile sizes)) {



    // In psuedo code this just
    //
    // dst = I
    // for(i = 0..p)
    //     dst = dst * A

    // Start off with the identity matrix, since M^0 == I
    // The result will end up in dst when we are done.
    for(int32_t x = 0; x < dst.size.x; x++) {
        for(int32_t y = 0; y < dst.size.y; y++) {
            if (x == y) {
                dst.get(x,y) = 1;
            } else {
                dst.get(x,y) = 0;
            }
        }
    }


    for(uint32_t p = 0; p < power; p++) {
        tensor_t<T> B(dst); // Copy dst, since we are going to modify it.
        mult_solution(dst,B,A); // multiply!

    }



}

#endif
```

```
template<typename T>
void  __attribute__((noinline)) mult_solution(tensor_t<T> &C, const tensor_t<T>
&A, const tensor_t<T> &B,) {

    // This is just textbook matrix multiplication.

    for(int i = 0; i < C.size.x; i++) {
        for(int j = 0; j < C.size.y; j++) {
            C.get(i,j) = 0;
            for(int k = 0; k < B.size.x; k++) {
                C.get(i,j) += A.get(i,k) * B.get(k,j);
            }
        }
    }
}
```

What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss

B. Capacity miss

[✓ C. Conflict miss]

D. Capacity & conflict miss

E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
      b_t[i][j] += b[j][i];
  }
}


for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++
              // Compute on b_t
              c[ii][jj] += a[ii][kk]*b_t[jj][kk];
    }
  }
}
```

Computer
Science &
Engineering

142L

つづく