SEC204

Computer Architecture and Low Level Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:

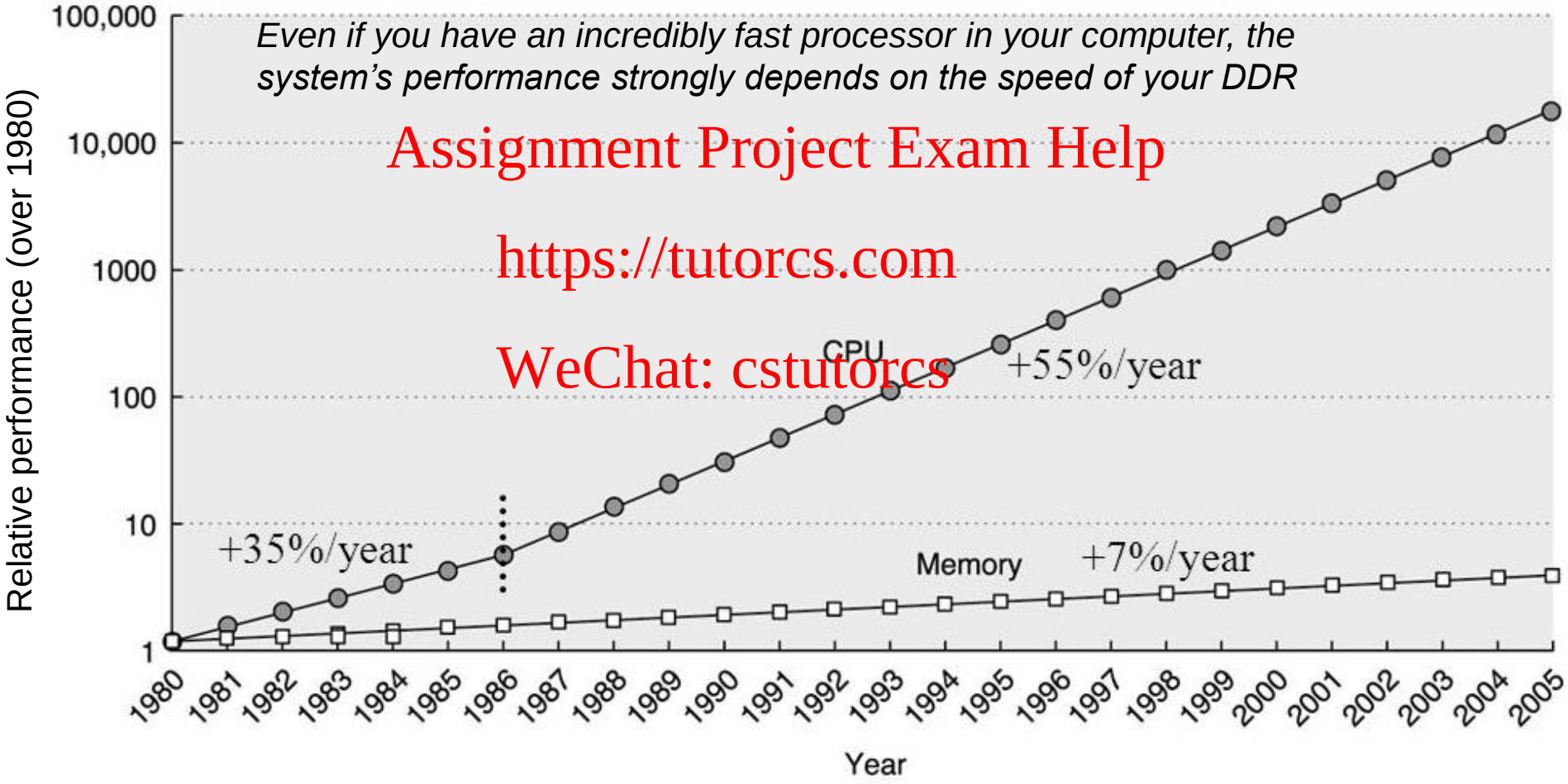
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Outline

- Memory hierarchy
- Cache memories
- Temporal and spatial locality
- Cache design
  - Cache hit/miss
  - Direct mapped, set associative, fully associative
  - Write policies
  - Replacement policy

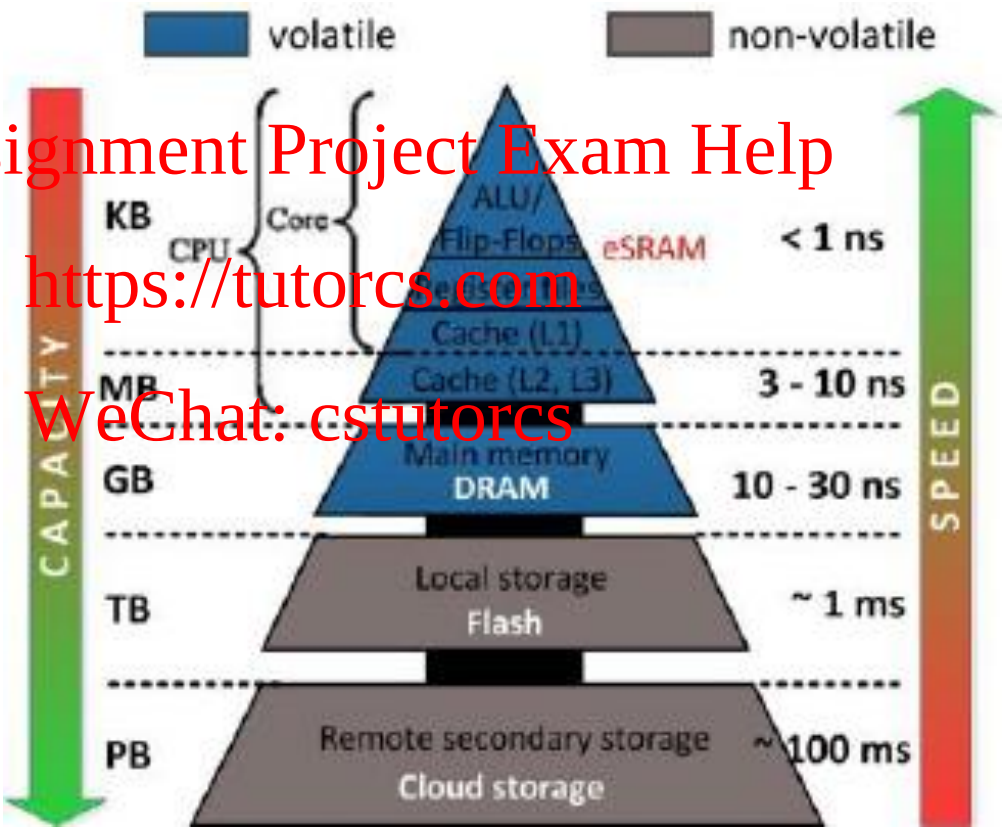
- Stack memory

Memory Wall Problem



Take from https://slideplayer.com/slide/7075269/

Memory Hierarchy



Taken from *https://www.researchgate.net/publication/281805561_MTJ-based_hybrid_storage_cells_for_normally-off_and_instant-on_computing/figures?lo=1*

Cache memories

- Wouldn't it be nice if we could find a balance between fast and cheap memory?
- The solution is to add from 1 up to 3 levels of cache memories, which are small, fast, but expensive memories
  - The cache goes between the processor and the slower, main memory (DDR)
  - It keeps a copy of the most frequently used data from the main memory
  - Faster reads and writes to the most frequently used addresses
  - We only need to access the slower main memory for less frequently used data
- Cache memories occupy the largest part of the chip area
- They consume a significant amount of the total power consumption
- Add complexity to the design
- Cache memories are of key importance regarding performance

Memory Hierarchy (2)

- Consider that CPU needs to perform a load instruction
  - First it looks at L1 data cache. If the datum is there then it loads it and no other memory is accessed (**L1 hit**)
  - If the datum is not in the L1 data cache (**L1 miss**), then the CPU looks at the L2 cache
  - If the datum is in L2 (**L2 hit**), then no other memory is accessed. Otherwise (**L2 miss**), the CPU looks at L3 etc

L1 cache access time:	1-4 CPU cycles
L2 cache access time :	6-14 CPU cycles
L3 cache access time :	40-70 CPU cycles
DDR access time :	100-200 CPU cycles

Data Locality

- Regarding **static** programs, it's very difficult and time consuming to figure out what data will be the "most frequently accessed" before a program actually runs
  - In static programs the control flow path is known at compile time
- Regarding **dynamic** programs it is impossible
- This makes it hard to know what to store into the small, precious cache memory
- But **in practice, most programs exhibit *locality*,** which the cache can take advantage of
  - **The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again**
  - **The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses**

Temporal Locality in Program Instructions

- The principle of **temporal locality** says that if a program accesses one memory address, there is a good chance that it will access the same address again
- Loops are excellent examples of temporal locality in programs
  - The loop body will be executed many times
  - The computer will need to access those same few locations of the instruction memory repeatedly
- For example:


```
Loop:   lw,   $t0, 0($s1)
add     $t0, $t0, $s2
sw      $t0, 0($s1)
addi    $s1, $s1, -4
bne     $s1, $0, Loop
```

  - Each instruction will be fetched over and over again, once on every loop iteration

Temporal Locality in Data

- Programs often access the same variables over and over, especially within loops, e.g., below, sum, i and B[k] are repeatedly read/written
- **Commonly-accessed variables can be kept in registers, but this is not always possible as there is a limited number of registers**

- Sum and i variables are a) of small size, b) reused many times, and therefore it is efficient to remain in the CPU's registers

- B[k] remains unchanged during the innermost loop and therefore it is efficient to remain in a CPU register
- The whole A[ ] array is accessed 3 times and therefore it will remain in the cache (depending on its size)

```
sum = 0;
for (k = 0; k < 3; k++)
  for (i = 0; i < N; i++)
    sum = sum + A[i] + B[k];
```

Spatial Locality in Program Instructions

- The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses

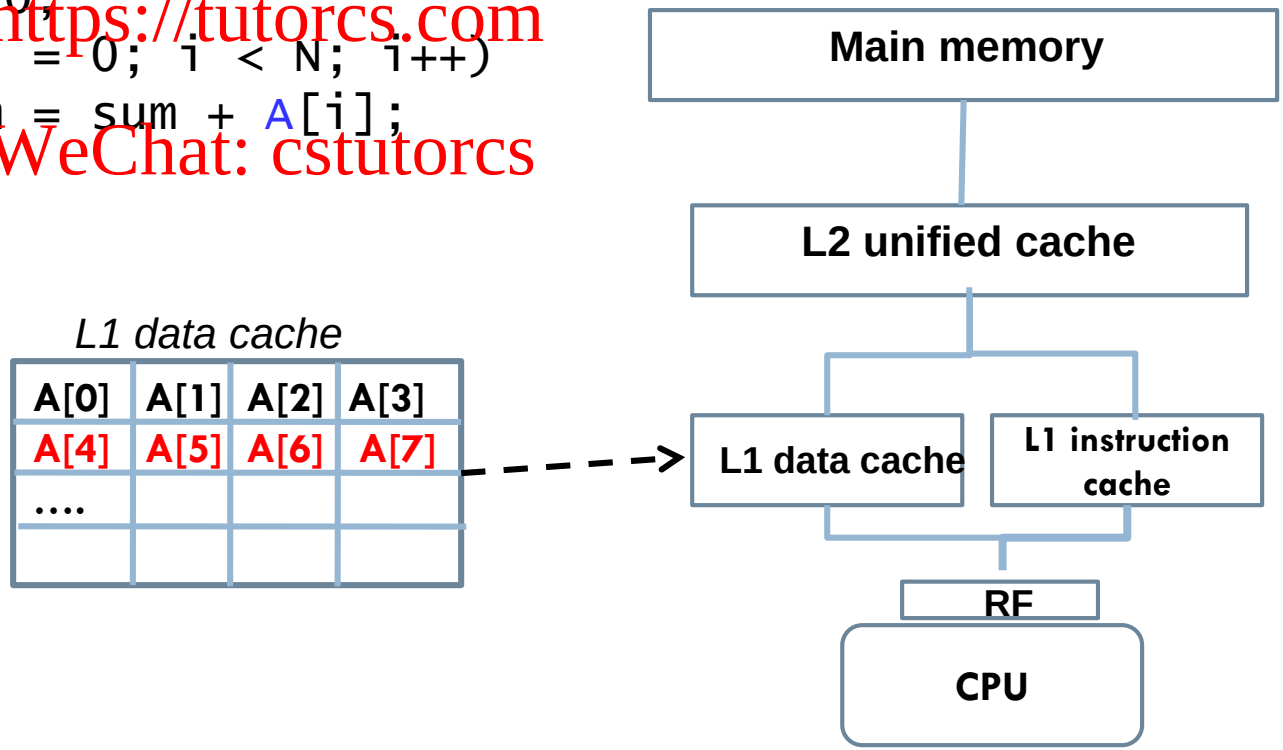

```
sub  $sp, $sp, 16
sw   $ra, 0($sp)
sw   $s0, 4($sp)
sw   $a0, 8($sp)
sw   $a1, 12($sp)
```

- **Every program exhibits spatial locality, because instructions are executed in sequence most of the time (however, branches might occur) -** if we execute an instruction at memory location *i*, then we will probably also execute the next instruction, at memory location *i+1*
- Code fragments such as loops exhibit *both* temporal and spatial locality

Spatial Locality in Data

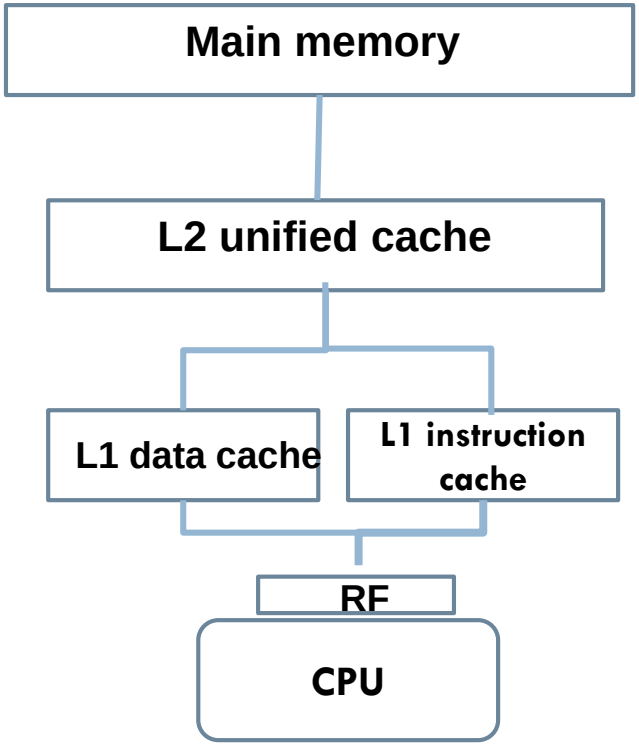
- Programs often access data that are stored in contiguous memory locations
  - Arrays, like A[ ] in the code below are always stored in memory contiguously – this task is performed by the compiler


```
sum = 0;
for (i = 0; i < N; i++)
    sum = sum + A[i];
```

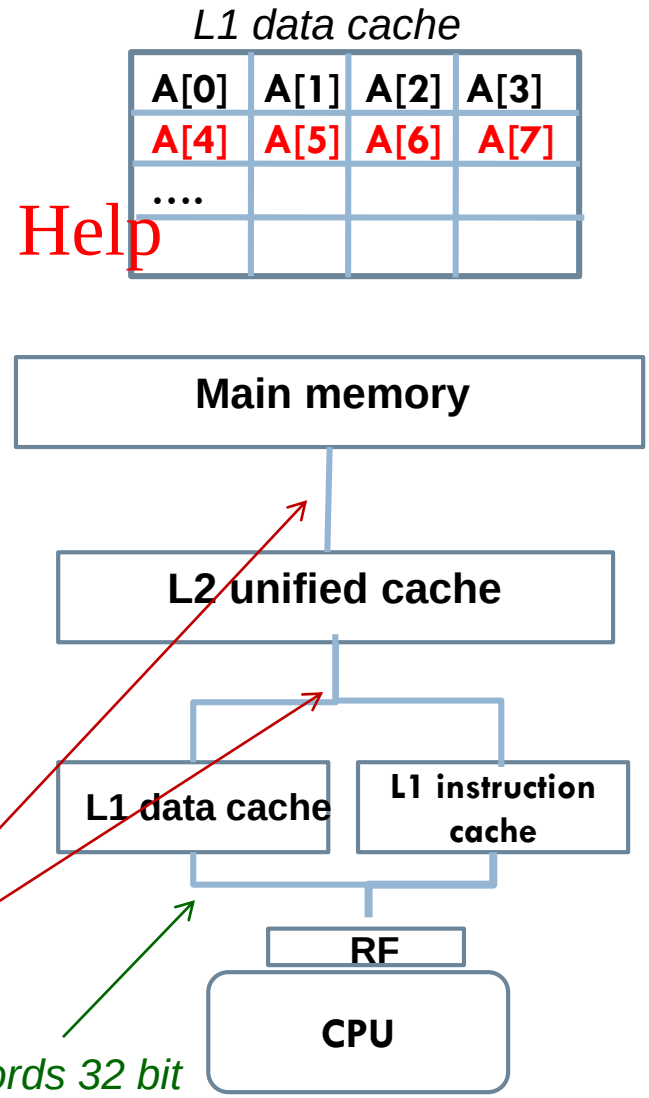
How caches take advantage of temporal locality

- Every time the processor reads from an address in main memory, a copy of that datum is also stored in the cache
  - The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower DDR
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster
- This takes advantage of temporal locality - **commonly accessed data are stored in the faster cache memory**



How caches take advantage of Spatial locality

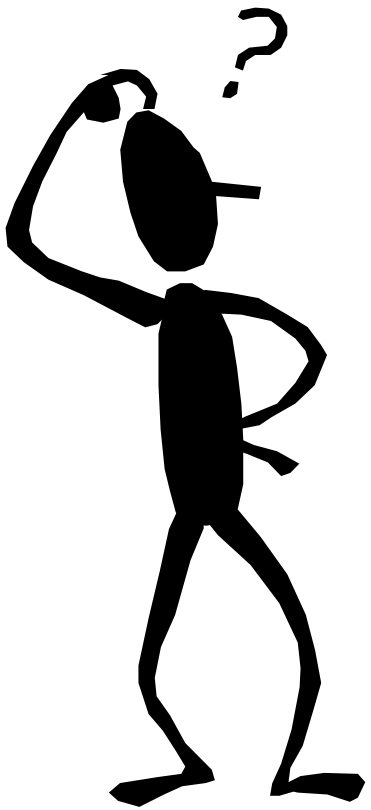
- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache
- **But instead of just copying the contents of location *i*, it copies *several* values into the cache at once (cache line)**
  - If the CPU later does need to read from a location in that cache line, it can access that data from the cache and not the slower main memory, e.g. **A[0] and A[3]**
  - For example, instead of loading just one array element at a time, the cache actually loads four /eight array elements at once
- Again, the initial load incurs a performance penalty, but we're gambling on spatial locality and the chance that the CPU will need the extra data



Definitions: Hits and misses

- A **cache hit** occurs if the cache contains the data that we're looking for. Hits are desirable, because the cache can return the data much faster than main memory
- A **cache miss** occurs if the cache does not contain the requested data. This is inefficient, since the CPU must then wait accessing the slower next level of memory
- There are two basic measurements of cache performance
  - The **hit rate** is the percentage of memory accesses that are handled by the cache
  - The **miss rate** (1 - hit rate) is the percentage of accesses that must be handled by the slower lower level memory
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster

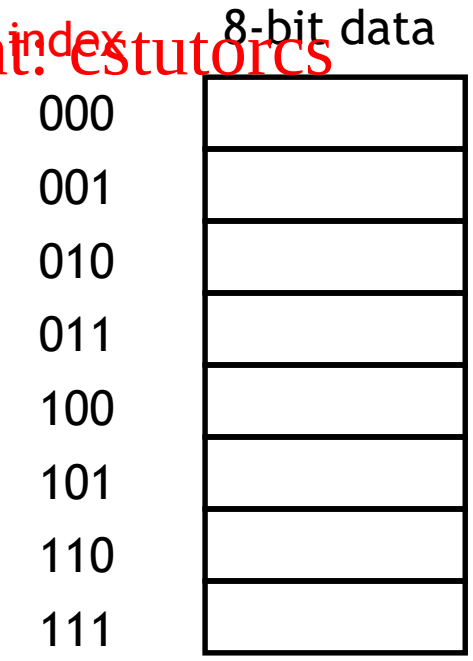
Important Questions



1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main memory, we'd have to replace one of the existing blocks in the cache... which one?
4. In a write request, are we going to write to all memories in memory hierarchy or not?

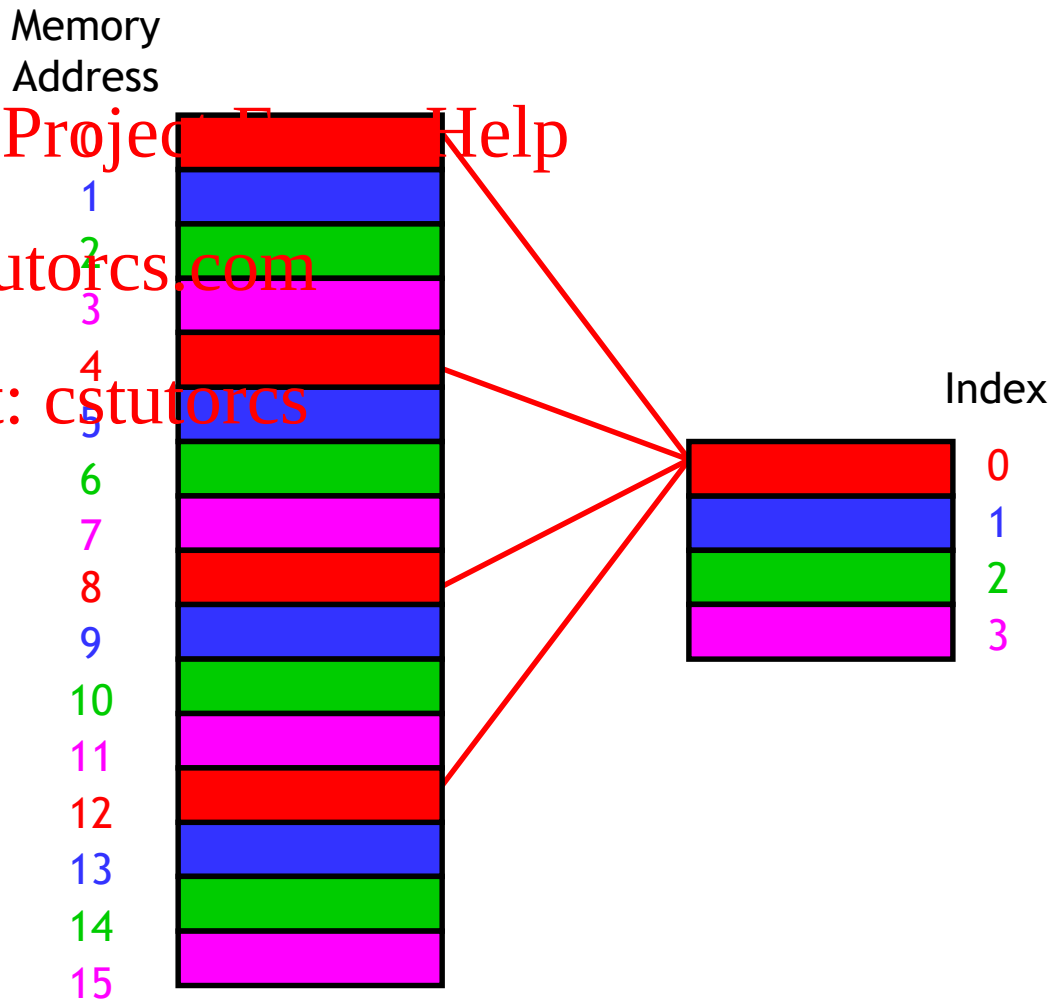
A simple cache design

- Caches are divided into blocks, which may be of various sizes
  - The number of blocks in cache memories are always in power of 2
  - For now consider that each block contains just one byte (not true in practice). Of course this cannot take advantage of spatial locality
- Here is an example of cache with eight blocks, each holding one byte



Where should we put data in the cache? (1)

- A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block
- In the following figure a 16-entry main memory and a 4-entry cache (four 1-entry blocks) are shown
- Memory locations 0, 4, 8 and 12 all map to cache block 0
- Addresses 1, 5, 9 and 13 map to cache block 1, etc



Where should we put data in the cache? (2)

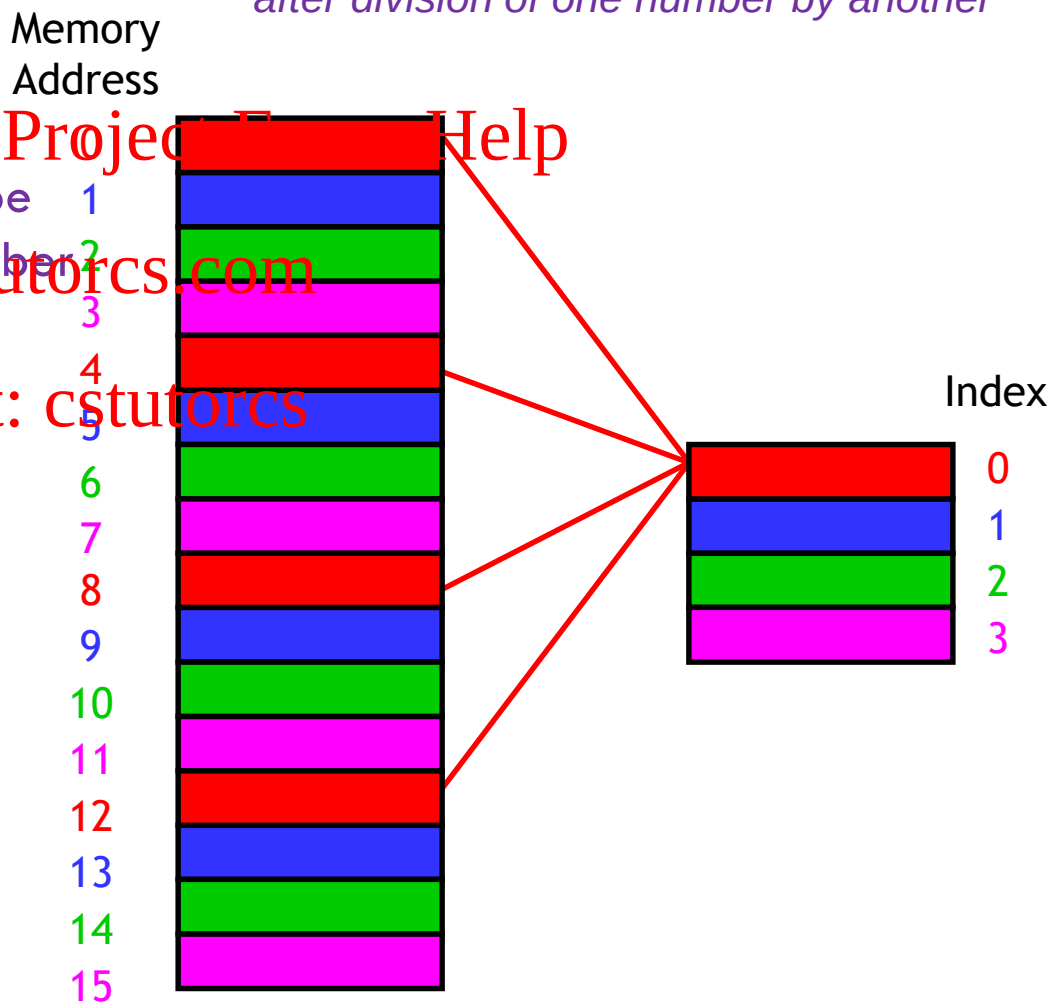
- One way to figure out which cache block a particular memory address should go to is to use the **modulo** (remainder) operator
- Let x be block number in cache, y be block number of DDR, and n be number of blocks in cache, then mapping is done with the help of the equation

$$x = y \bmod n$$

- For instance, with the four-block cache here, address 14 would map to cache block 2

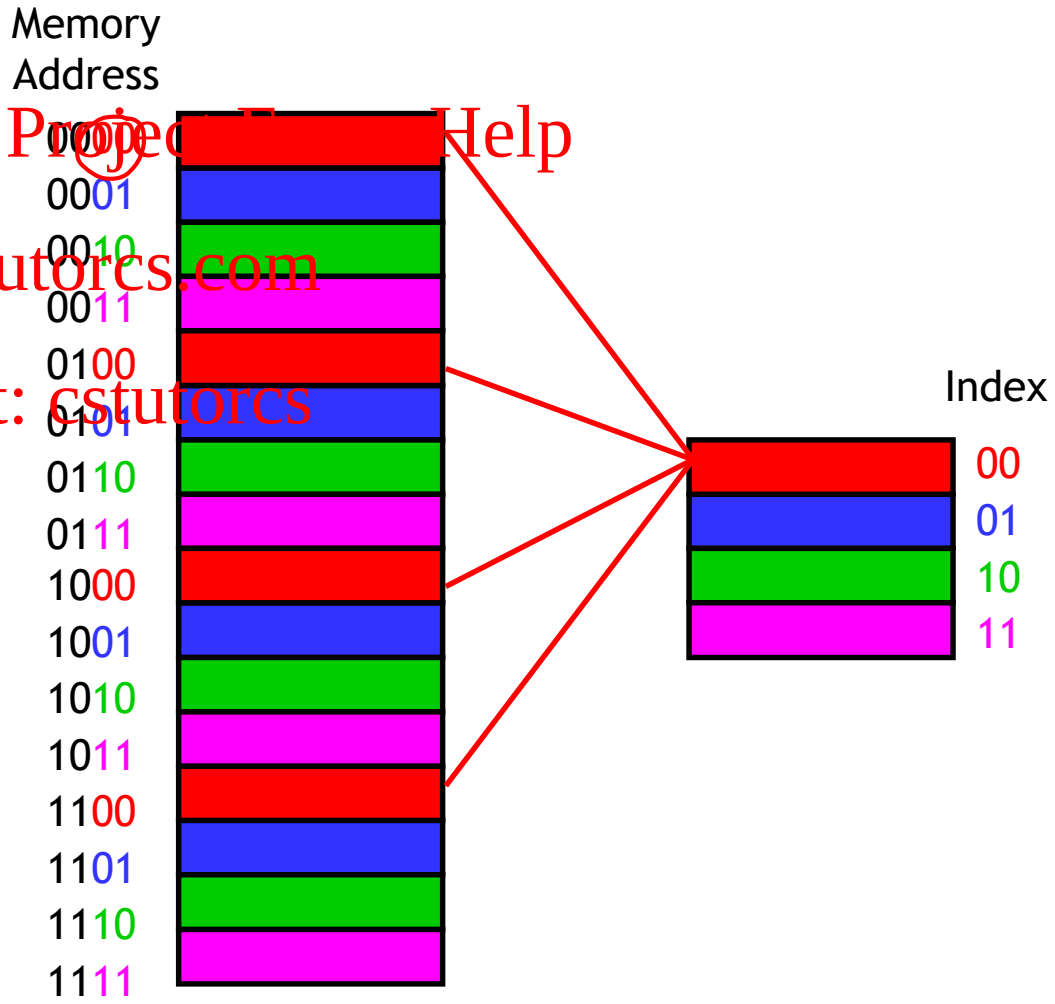
$$14 \bmod 4 = 2$$

the modulo operation finds the remainder after division of one number by another



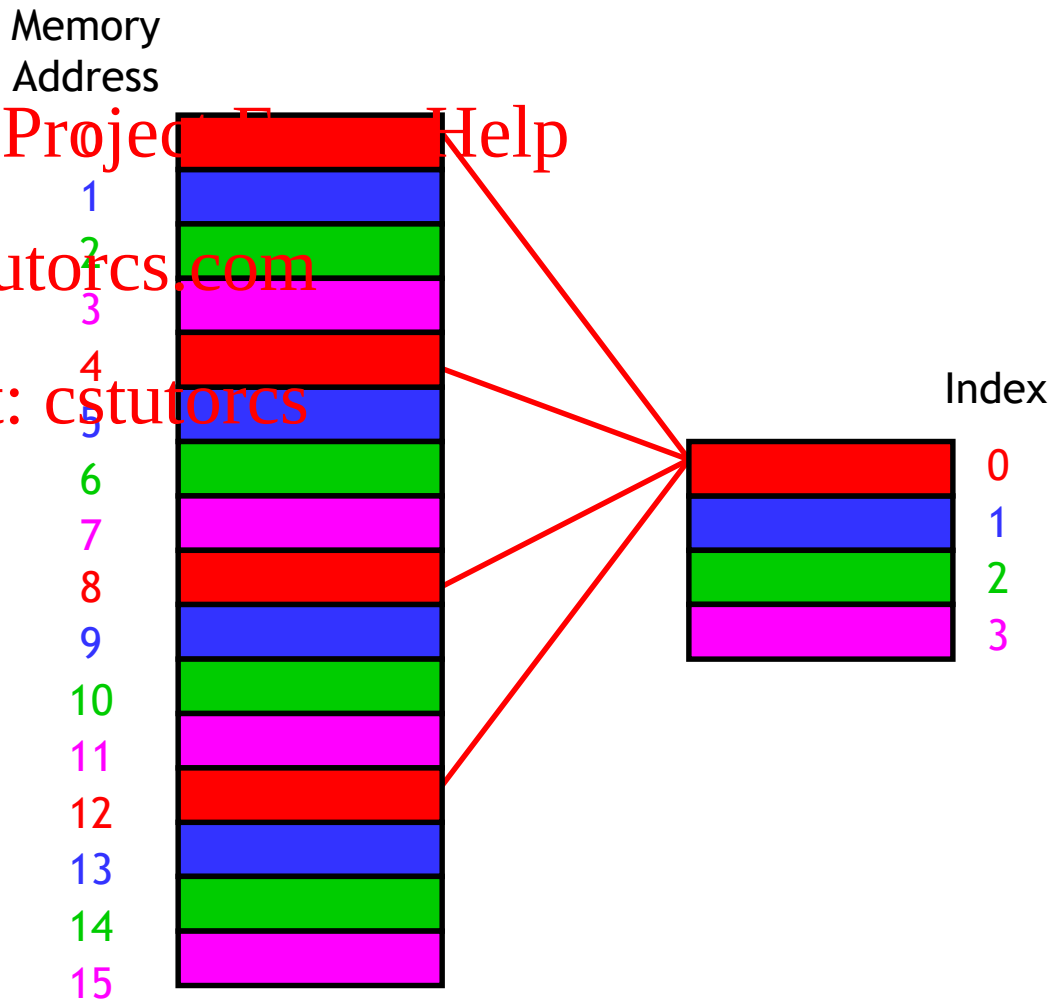
Where should we put data in the cache? (3)

- **An equivalent way to find the placement of a memory address in the cache is to look at the least significant *k* bits of the address**
- In a four-entry cache we would check the two least significant bits of our memory addresses
- Again, you can check that address 14_{10} (1110_2) maps to cache block 2_{10} (10_2)
- Taking the least *k* bits of a binary value is the same as computing that value mod n



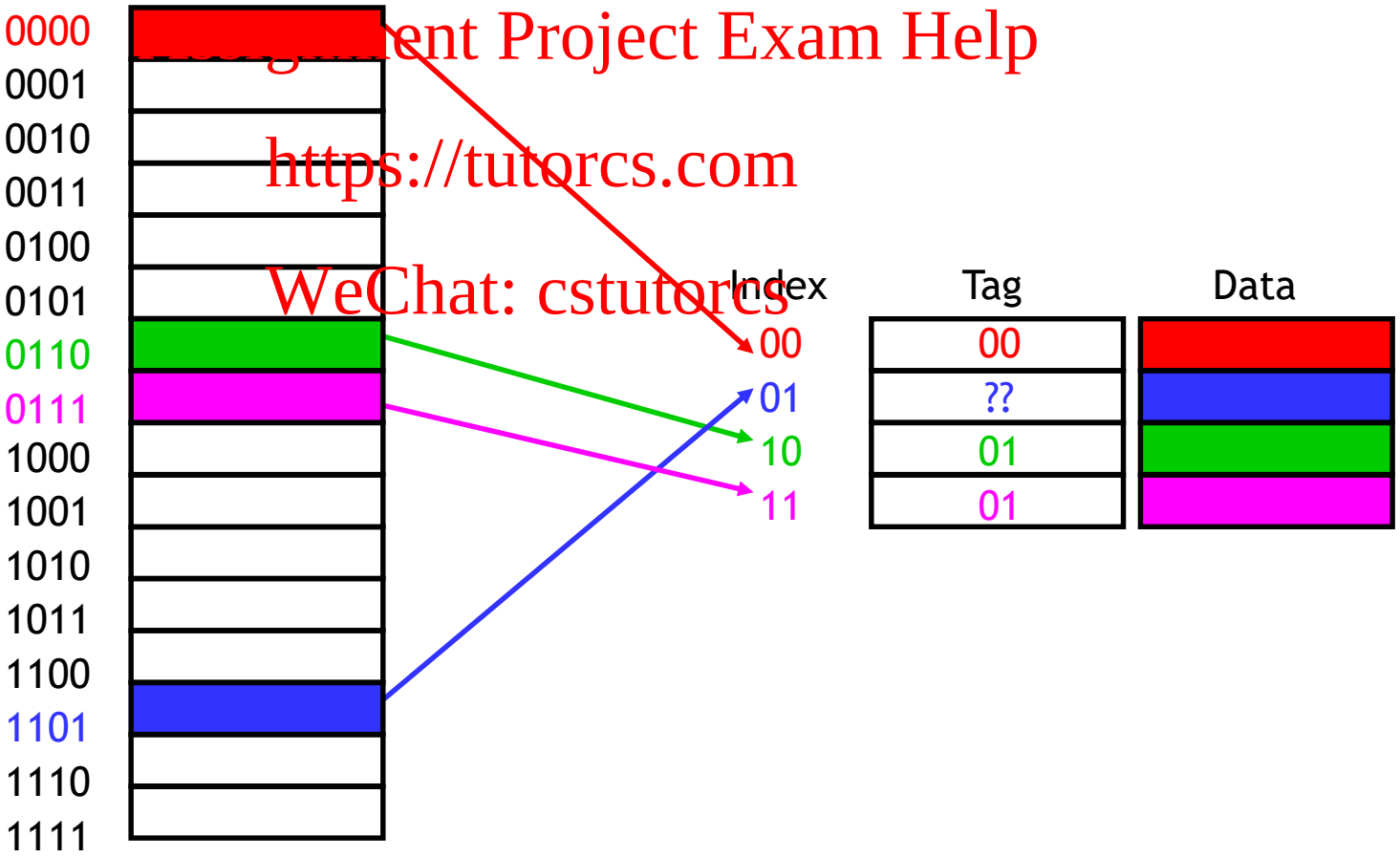
How can we find data in the cache?

- **How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?**
- If we want to read memory address *i*, we can use the mod trick to determine which cache block would contain *i*
- But other addresses might *also* map to the same cache block. How can we distinguish between them?
- For instance, cache block 2 could contain data from addresses 2, 6, 10 *or* 14



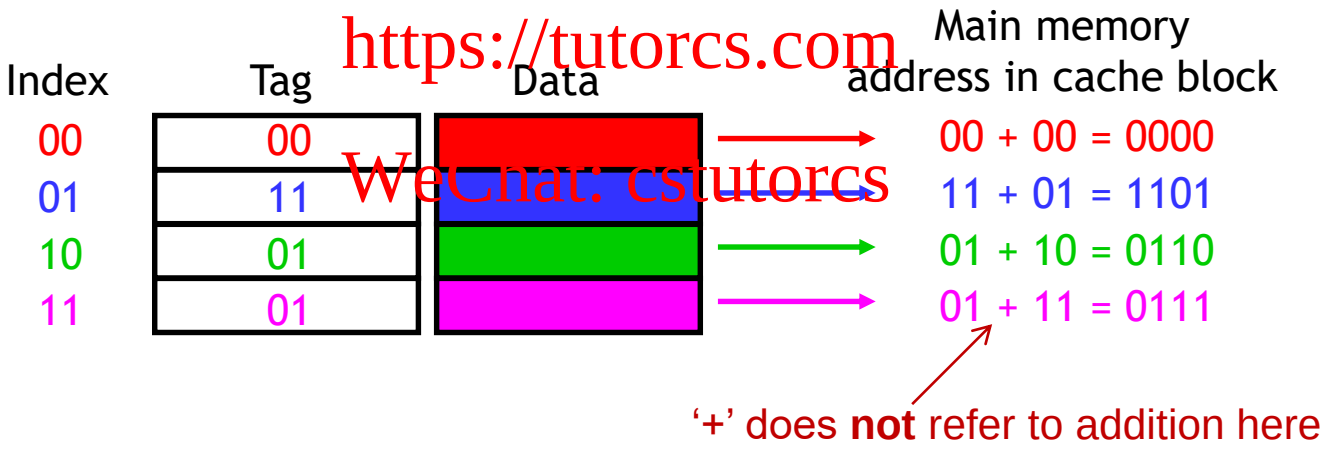
Adding tags

- **The solution is to add tags to the cache,** which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block



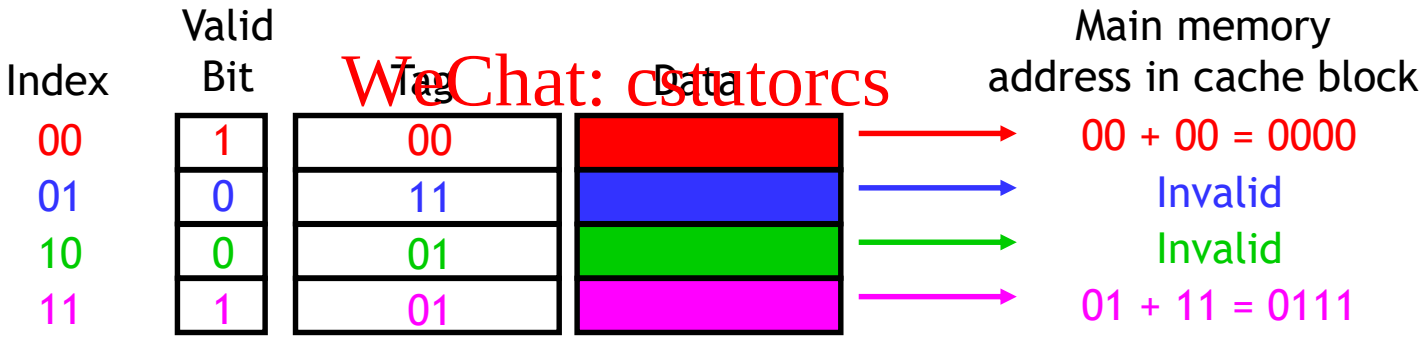
what's in the cache

- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices



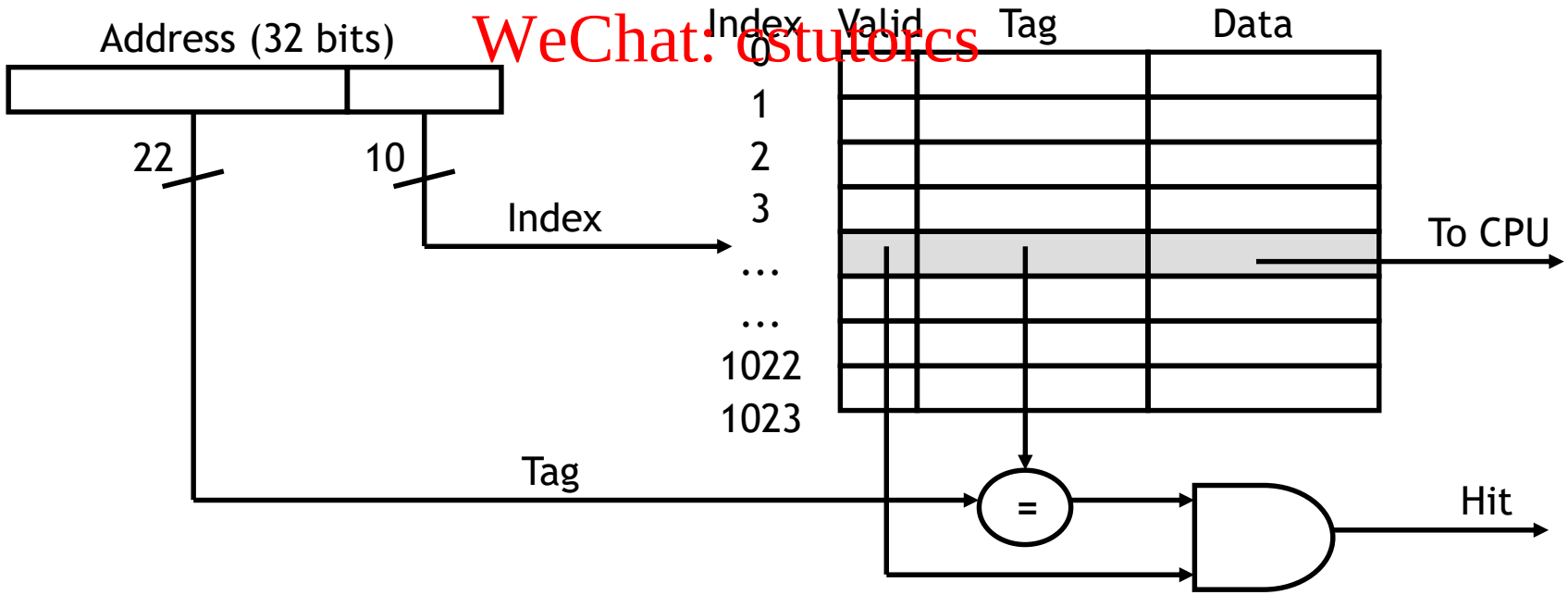
the valid bit

- **Initially, the cache is empty and does not contain valid data, but trash**
- Thus, we add a **valid bit** for each cache block
  - When the system is initialized, all the valid bits are set to 0
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1



cache hit

- **Every memory has its memory controller, a HW mechanism responsible for finding the words in memory, loading/storing etc**
- When the CPU tries to read from memory, the address will be sent to the cache controller
  - The lowest *k* bits of the address will index a block in the cache
  - If the block is valid and the tag matches the upper (*m* - *k*) bits of the *m*-bit address, then that data will be sent to the CPU
- Here is a diagram of a 32-bit memory address and a 2^{10} byte cache



cache miss

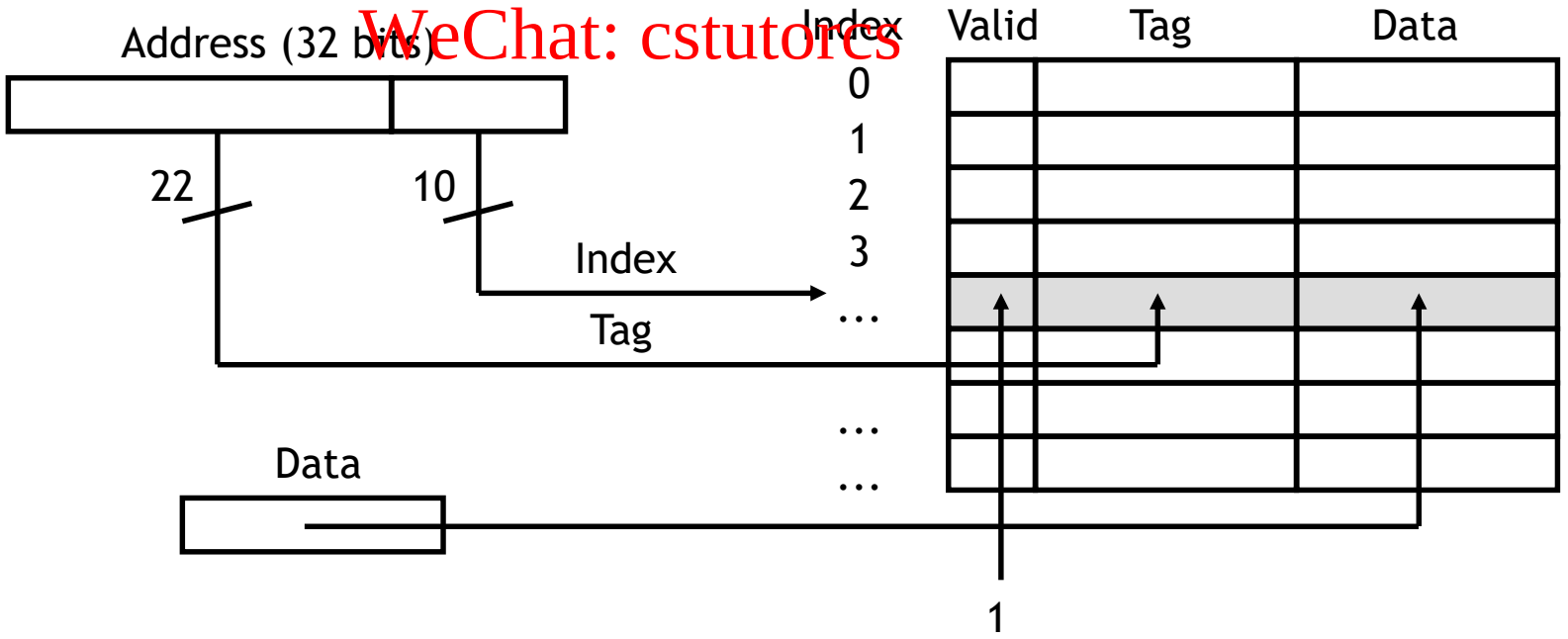
- In a two level memory hierarchy, L1 Cache misses are somehow expensive, but L2 cache misses are very expensive

- However, the slower main memory accesses are inevitable on an L3 cache miss

- **The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache)**

Copying a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward
  - The lowest k bits of the address specify a cache block
  - The upper ($m - k$) address bits are stored in the block's tag field
  - The data from main memory is stored in the block's data field
  - The valid bit is set to 1

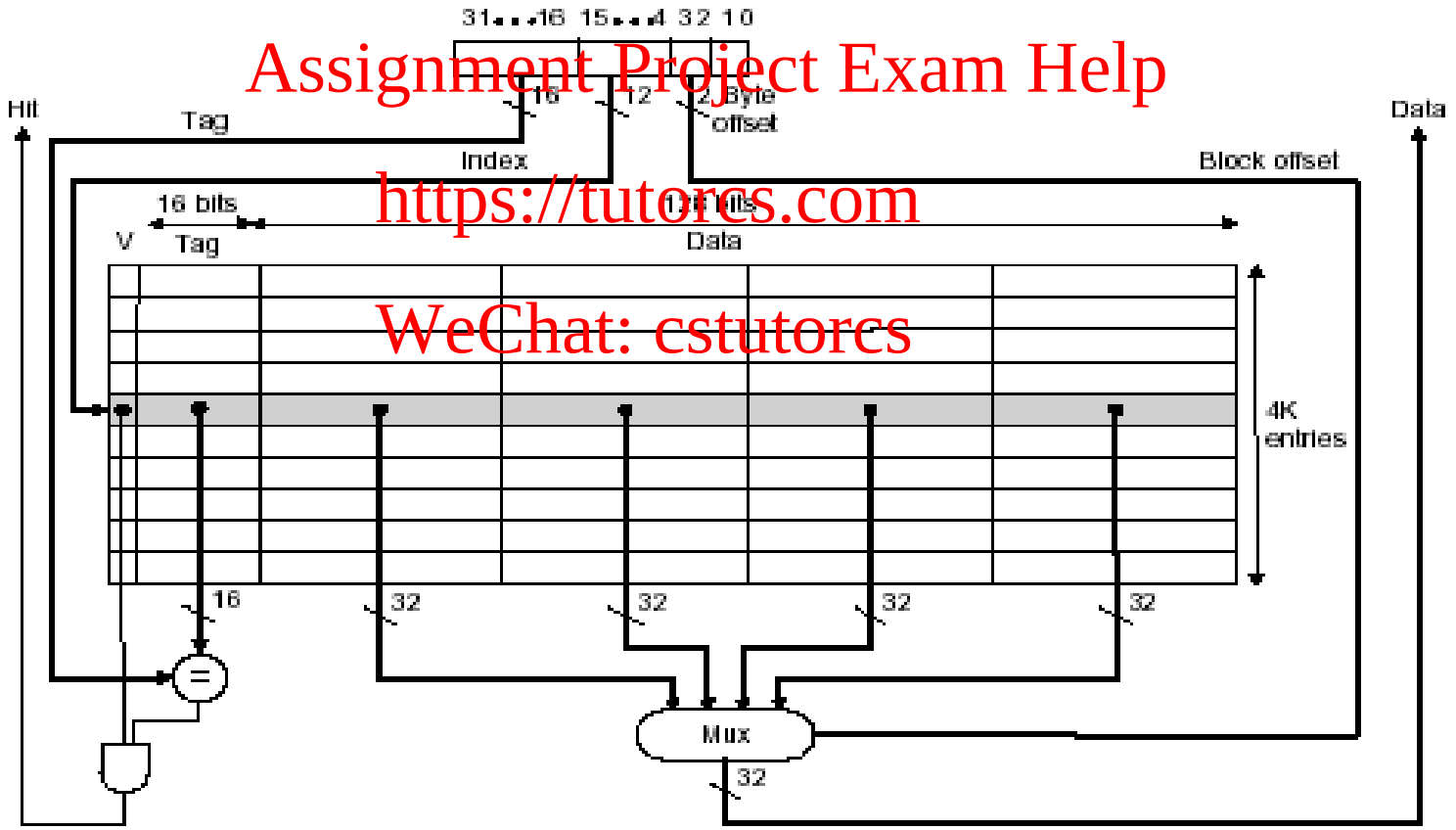


What if the cache fills up?

- Eventually, the small cache memory might fill up. To load a new block from DDR, we'd have to replace one of the existing blocks in the cache... which one?
  - A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data
  - Normally, the least recently used (LRU) replacement policy is used, which assumes that least recently used data are less likely to be requested than the most recently used ones
  - So, in a cache miss, **cache throws out the cache line that has been unused for the longest time**

A more realistic **direct mapped** cache memory

- Normally, one cache line contains 128/256 bits of data. In most programming languages, an integer uses 32 bits (4 bytes)



Associativity

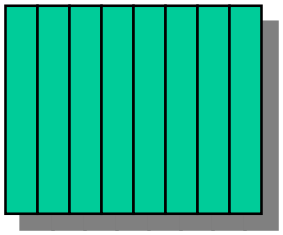
- The replacement policy decides where in the cache a copy of a particular entry of main memory will go
- **So far, we have seen directed mapped cache only**
  - each entry in main memory can go in just one place in the cache
- Although direct mapped caches are simpler and cheaper they are not performance efficient as they give a large number of cache misses
- **There are three types of caches regarding associativity**
  - **Direct mapped**
  - **N-way Associative**
  - **Fully associative**

Associative Caches

- Block 12 placed in 8 block cache:

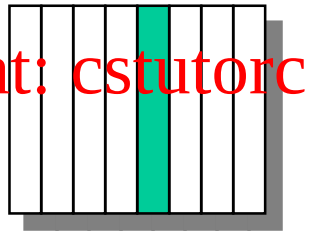
Fully associative: block 12 can go anywhere

Block no. 0 1 2 3 4 5 6 7



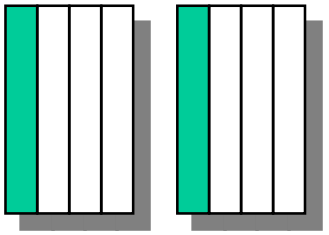
Direct mapped: block 12 can go only into block 4 (12 mod **8** = 4)

Block no. 0 1 2 3 4 5 6 7

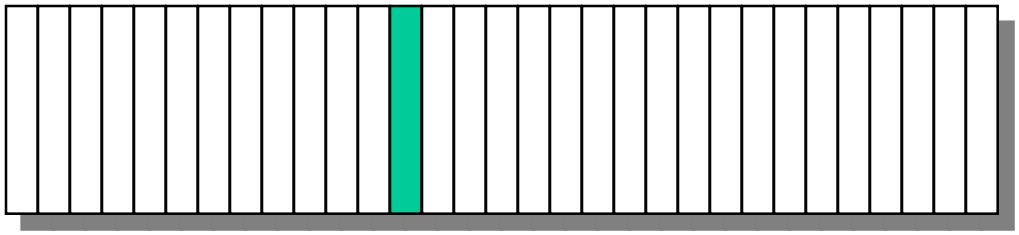


2 way Set associative (like having two half size direct mapped caches): block 12 can go in either of the two block 0 (12 mod **4** = 0)

Block no. 0 1 2 3 0 1 2 3



Block-frame address

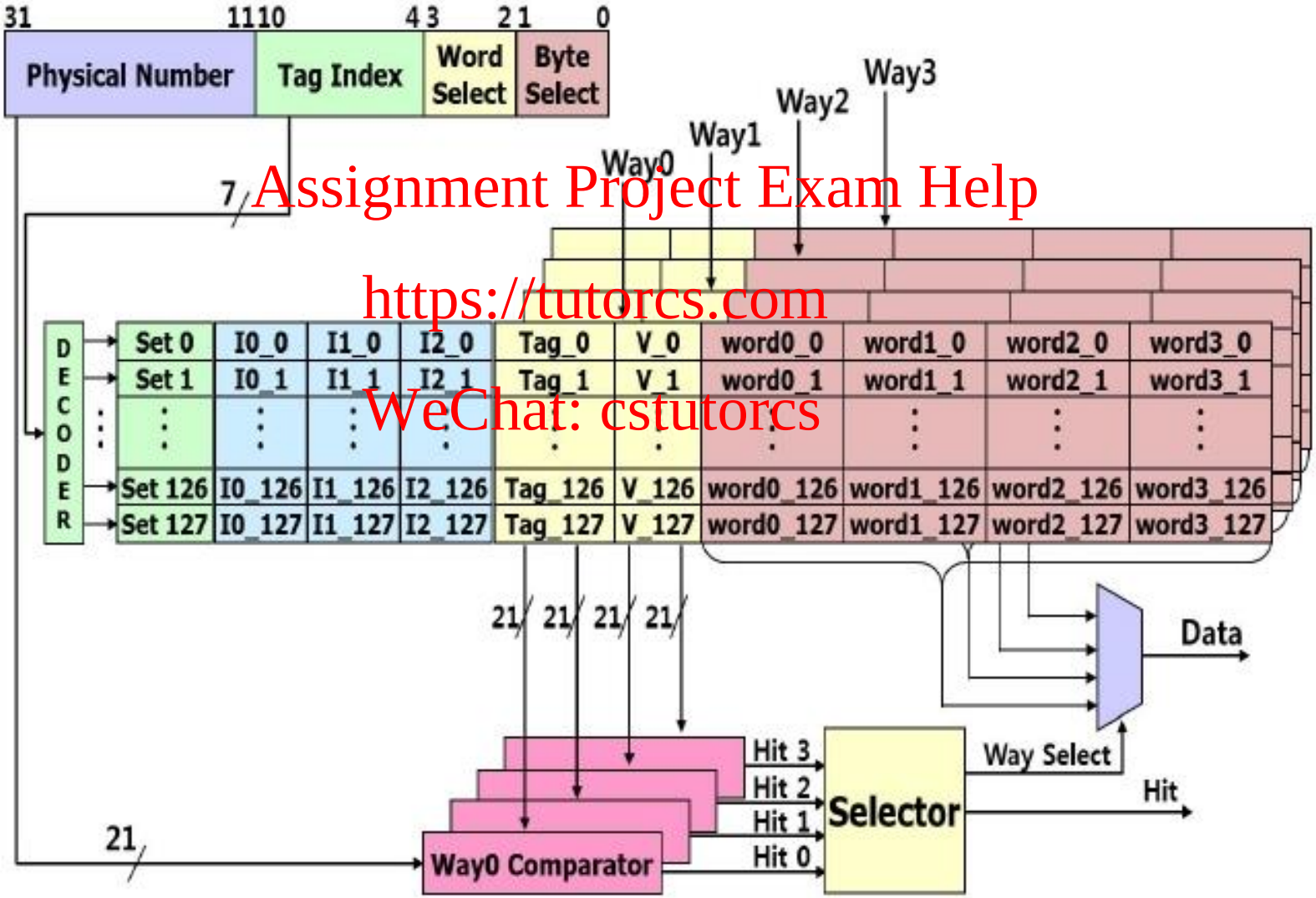


Block no. 0 1 2 3 4 5 6 7 8 9 0 1 **2** 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Set-associative cache memories

- **A 4-way associative cache consists of four direct-mapped caches that work in parallel**
- **Normally, L1 caches are of 8 way associative, while L2/L3 caches are of 16/24 way associative, i.e., 16/24 direct mapped caches in parallel**
- Data are found in one cache among four by using an address which is stored in one of the four caches
- **Set associative caches are the most used as they present the best compromise between cost and performance**

The 4-way set-associative cache architecture



Cache misses

- **Compulsory miss** (or cold miss): first access to a block
  - Normally, compulsory misses are insignificant
- **Capacity miss**:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- **Conflict miss**:
  - Multiple memory locations are mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

Write policies - In a write request, are we going to write to all memories in memory hierarchy or not?

- ***Write through*** – **Data are written to all memories in memory hierarchy**
  - Disadvantage: Data are written into multiple memories every time and thus it takes more time

- ***Write back*** – **Data are written only to L1 data cache; only when this block is replaced, it is written in L2. If this block is replaced from, then it is written in main memory**

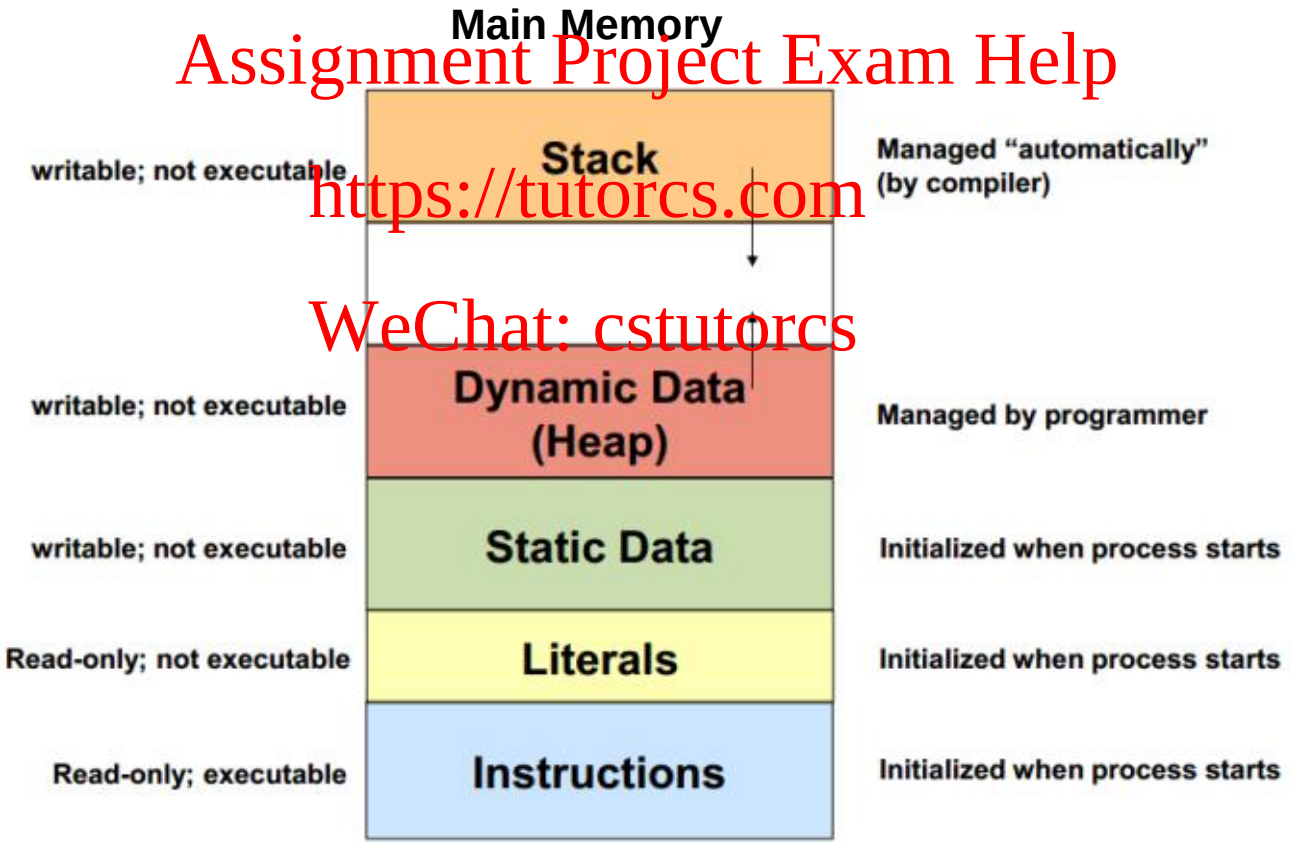
  - Requires a "dirty bit"
  - more complex to implement, since it needs to track which of its locations have been written over, and mark them as dirty for later writing to the lower lower memory

Write policies (2)

What happens on a write miss?

- a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:
  - **Write allocate** : datum at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses
  - **No-write allocate** : datum at the missed-write location is not loaded to cache, and is written directly to DDR. In this approach, data are loaded into the cache on read misses only
- Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in this way:
  - **A write-back cache uses write allocate**, hoping for subsequent writes (or even reads) to the same location, which is now cached
  - **A write-through cache uses no-write allocate**. Here, subsequent writes have no advantage, since they still need to be written directly to DDR

Memory Allocation



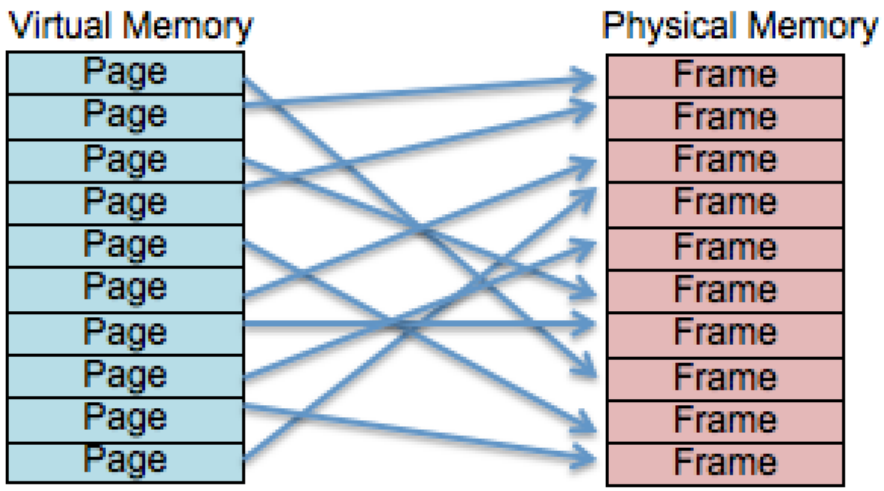
Stack

- There are 4 parts in main memory:
  - **code, global variables, stack, and heap.**
- Stack is a block of memory that is used for function calls and local variables.
- Stack size is fixed during compilation – cannot ask for more during run-time.
- Actual data in the stack grows and shrinks as local variables and other pointers are added and removed accordingly.
- **Every function call is allocated a stack frame –** a block of memory – required for holding function specific data. The size of this is determined during compile-time.
- In x64, the stack frame size is a multiple of 16 bytes
- Little-endian form is used

Virtual and Physical Memory

- A virtual address is a memory address that a process uses to access its own memory
  - The virtual address is not the same as the actual physical RAM addresss in which it is stored
  - When a process accesses a virtual address, the Memory Management Unit hardware translates the virtual address to physical
  - The OS determines the mapping from virtual address to physical address

- Some of the benefits include
  - virtual space available is huge compared to physical memory
  - increased security due to memory isolation



How can I find the memory addresses of an array?

- **We can print the virtual memory addresses only**
- The hardware addresses are managed by the memory management unit
- However, if you know how the cache works you can have a very good guess about where the data are stored


```
//print virtual memory addresses
for (i=0; i<4; i++)
 for (j=0; j<4; j++)
  printf("\nThe address of element (%d,%d) is %p",i, j, &A[i][j]);
```

Stack and Heap

- Stack memory
  - The stack is a special area of RAM that can be used by functions as temporary storage..
    - To save register state.
    - For local variables.
    - For large input parameters / return values.
    - Stack works as Last In First Out (LIFO)

- Heap memory
  - Dynamic allocation

Why Stack is needed?

- Calling subroutines
- Using registers in subroutines
- Using local variables in subroutines with the same name as in others
  - less RAM memory is used as these local variables are no longer needed when the function is ended
- Passing and returning large arguments to subroutines

Why Stack is needed?

- Every function being called has its own space in stack
- Every time a function is called, it stores into the stack the following:
  - the return address – thus the CPU knows where to return afterwards
  - its input operands (in x86, the first 6 operands are stored into registers)
  - Its local variables – this way, we can use the registers in the subroutine without being overwritten, e.g., consider using *edx* in both caller and callee

Stack related instructions and registers

- **Registers**
  - ***ESP Stack pointer*** – points to the last element used in the stack. Implicitly manipulated by instructions (e.g. push, pop, call, ret, etc.)
  - ***EBP Base pointer*** – used to reference function parameters and local variable within a stack frame.
- **Instructions**
  - ***push M/L/R*** Pushes an M/L/R value on top of the stack (ESP - 4).
  - ***pop R*** Remove and restore an R value from the top (ESP + 4).
  - ***call* label** Calls a function with a label. This results into pushing the next instruction address on the stack.
  - ***ret*** Returns to caller function – return value is usually stored in eax register.

Every time we call a function…

- The operands of the called function are stored into *rdi,rsi,rdx,rcx,r8,r9* and then into the stack (if more space is needed).
- The return value is stored to *rax* (*eax* for 32bit - it is the same register)
- Remember this when apply reverse engineering…

How Stack Works?

- Let's have a look at the 3rd question of this lab session …

Caller and Callee – example (1)

- In the C++ code below, main is the caller function, and addFunc is the callee function.
- We will convert this into assembly and see how it works with the memory.

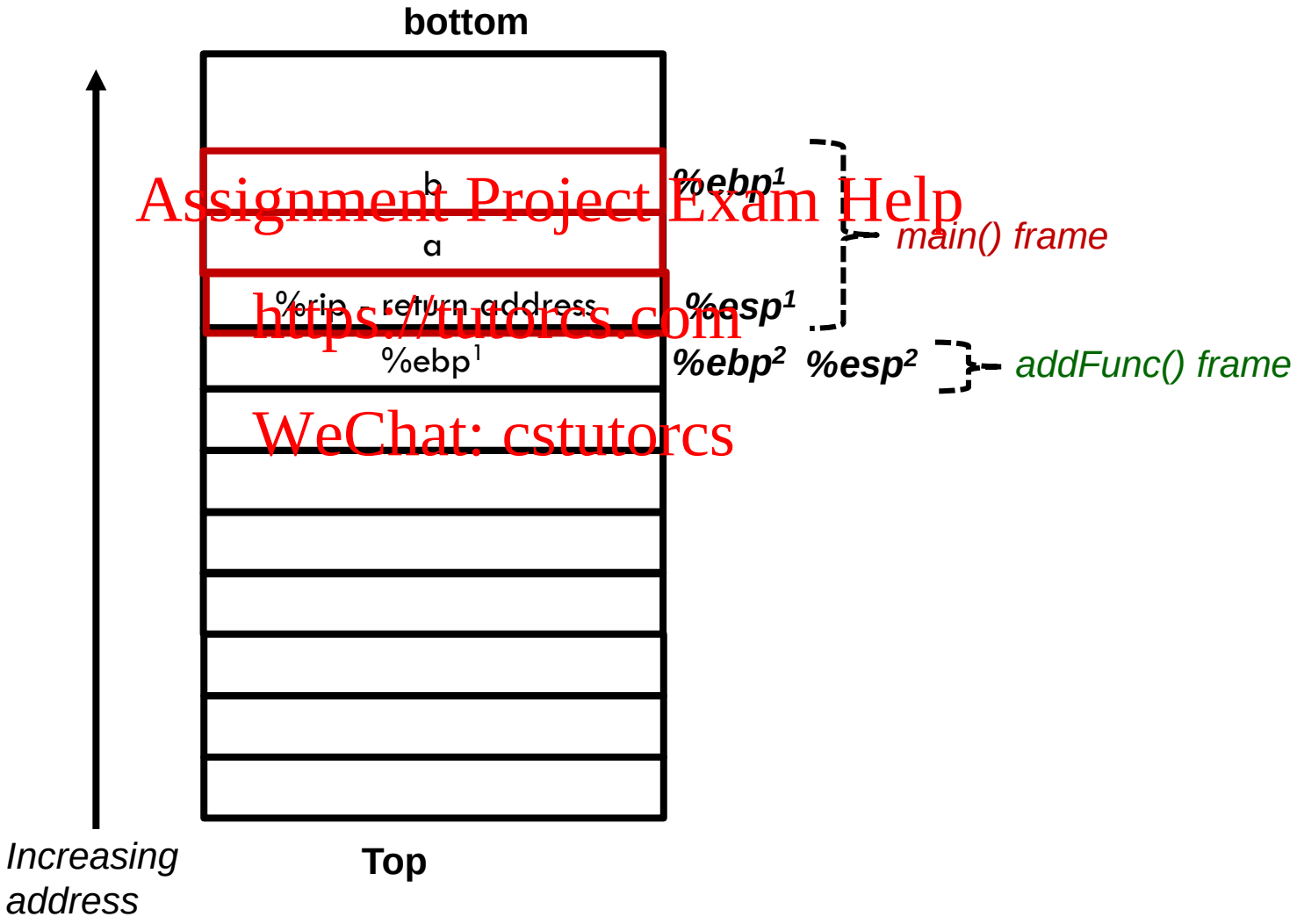

```
int main() {
    int a = 2;
    int b = 3;
    int c = addFunc(a, b);
    return 0;
}

int addFunc(int a, int b) {
    return a + b;
}
```

Caller and Callee -example (2)

47

```
6   .data ; data segment
7       ; define your variables here
8       a DWORD 2
9       b DWORD 4
10  .code   ; code segment
11      main PROC C ;   cdecl calling cconvention -- caller
12          push b  ; push b into stack
13          push a  ; push a into stack
14          call addFunc ; call addtion function
15          add esp, 8 ; remove the
16          INVOKE ExitProcess, 0 ; exit process
17      main ENDP
18      addFunc PROC C ; cdecl calling convention -- callee
19          push ebp ; store whatever ebp is for the caller
20          mov ebp, esp ; get the current stack pointer into the base pointer
21          mov ebx, [ebp + 8] ; this is a
22          mov eax, [ebp + 12] ; this is b
23          add eax, ebx
24          pop ebp ; restore ebp for the caller
25          ret ; return
26      addFunc ENDP
27  END main
```

Caller and Callee -example (3)



Walking through the code...


```
11      main PROC C ;   cdecl calling cconvention -- caller
12          push b  ; push b into stack
```

```
Registers
EAX = 8A[illegible] EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 0118101C ESP = 00DEF8C0
EBP = 00DEF8D0 EFL = 00000246
```


- We start the debugger
- Before running line 12, the ESP=0x00DEF8C0.
- Each address is pointing to 4 bytes of memory.
- Decreasing addresses has nothing in there to being with.
- Push the last parameter of the function first (LIFO)!

```
Memory 1
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  00 00 00 00  ....
0x00DEF8B4  00 00 00 00  ....
0x00DEF8B8  00 00 00 00  ....
0x00DEF8BC  00 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

Walking through the code...


```
12          push b  ; push b into stack
13          push a  ; push a into stack  ≤ 1ms elapsed
```

```
Registers
EAX = 8AC2BA1B EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 01181022 ESP = 00DEF8BC
EBP = 00DEF8D0 EFL = 00000246
```


- The value of b is pushed on the top of the stack.
- Last used address is now ESP = 0x00DEF8C0 - 4 = 0x00DEF8BC.
- Little Endian – lower bits on the lower addresses (see red highlighted line on the right).

```
Memory 1
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  00 00 00 00  ....
0x00DEF8B4  00 00 00 00  ....
0x00DEF8B8  00 00 00 00  ....
0x00DEF8BC  04 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

Walking through the code…


```
13          push a  ; push a into stack
14          call addFunc ; call addtion function ≤ 1ms elapsed
```

```
Registers
EAX = 8AC2BA10 EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 01181028 ESP = 00DEF8B8
EBP = 00DEF8D0 EFL = 00000246
```



```
Memory 1
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  00 00 00 00  ....
0x00DEF8B4  00 00 00 00  ....
0x00DEF8B8  02 00 00 00  ....
0x00DEF8BC  04 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

- The value of a is pushed on the top of the stack.
- Last used address is now ESP = 0x00DEF8BC - 4 = 0x00DEF8B8.
- Little Endian – lower bits on the lower addresses (see red highlighted line on the right).

Walking through the code...


```
18      addFunc PROC C ; cdecl calling convention -- callee
19  |       push ebp ; store whatever ebp is for the caller  ≤1ms elapsed
```

```
Registers
 EAX = 8AC2BA10 EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
   ESI = 0118100A EDI = 0118100A EIP = 01181037 ESP = 00DEF8B4
   EBP = 00DEF8D0 EFL = 00000246
```



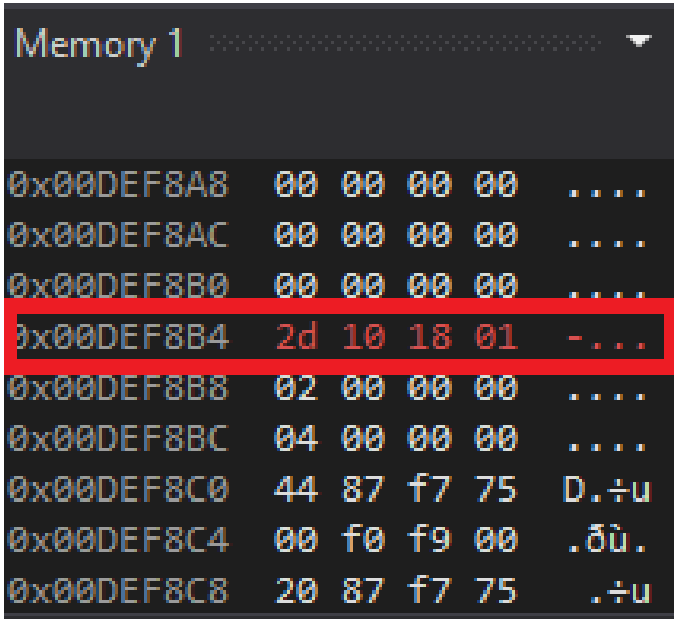
```
Memory 1
0x00DEF8A8   00 00 00 00   ....
0x00DEF8AC   00 00 00 00   ....
0x00DEF8B0   00 00 00 00   ....
0x00DEF8B4   2d 10 18 01   -...
0x00DEF8B8   02 00 00 00   ....
0x00DEF8BC   04 00 00 00   ....
0x00DEF8C0   44 87 f7 75   D.÷u
0x00DEF8C4   00 f0 f9 00   .ðù.
0x00DEF8C8   20 87 f7 75    .÷u
```

- What happened here? ⇒ We jumped into the fucntion that we called: addFunc.
- Last used address is now ESP = 0x00DEF8B8 - 4 = 0x00DEF8B4.
- Why was that weird number pushed to the stack?

Walking through the code...


```
        call addFunc ; call addtion function
01181028  call        addFunc (01181037h)
        add esp, 8 ; remove the
0118102D  add         esp,8
```


- We pushed the instruction address that comes after the function call would finish, so that we can resume normal operation after fucntion call has finished.



Walking through the code...


```
19        push ebp ; store whatever ebp is for the caller
20        mov ebp, esp ; get the current stack pointer into the base pointe
```

```
Registers
EAX = 8AC2BA10 EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 01181038 ESP = 00DEF8B0
EBP = 00DEF8D0 ESL = 00000246
```


- Now, we have saved the EBP state in the stack.
- This is because we are going to re-write it in the next statement.
- This will enable us to explicitly manipulate bits of memory.
- Last used address is now ESP = 0x00DEF8B4 - 4 = 0x00DEF8B0.


```
Memory 1
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  d0 f8 de 00  ÐøÞ.
0x00DEF8B4  2d 10 18 01  -...
0x00DEF8B8  02 00 00 00  ....
0x00DEF8BC  04 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

Walking through the code…


```
20          mov ebp, esp ; get the current stack pointer into the base pointe
21          mov ebx, [ebp + 8] ; this is a   ≤ 1ms elapsed
```

```
Registers
EAX = 8AC2BA10 EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 0118103A ESP = 00DEF8B0
EBP = 00DEF8B0 ESL = 00000246
```


- We copied the ESP into the EBP register.
- Within addFunc, we will use the EBP register to access stack rather than ESP.
- We want to access a and b values fromthe stack next. Where are they?
- Last used address is now ESP = 0x00DEF8B4 - 4 = 0x00DEF8B0.

Memory 1

```
0x00DEF8A8   00 00 00 00   ....
0x00DEF8AC   00 00 00 00   ....
0x00DEF8B0   d0 f8 de 00   ÐøÞ.
0x00DEF8B4   2d 10 18 01   -...
0x00DEF8B8   02 00 00 00   ....
0x00DEF8BC   04 00 00 00   ....
0x00DEF8C0   44 87 f7 75   D.÷u
0x00DEF8C4   00 f0 f9 00   .ðù.
0x00DEF8C8   20 87 f7 75    .÷u
```

Walking through the code…


```
20          mov ebp, esp ; get the current stack pointer into the base pointe
21          mov ebx, [ebp + 8] ; this is a  ≤1ms elapsed
```

```
Registers
EAX = 8AC2BA10 EBX = 00F9F000 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 0118103A ESP = 00DEF8B0
EBP = 00DEF8B8 EFL = 00000246
```


- Currently ESP = 0x00DEF8B0.
- a is in address 0x00DEF8B8 = ESP + 8
- b is in address 0x00DEF8BC = ESP + 12
- Why? ⇒ We pushed the next instruction address following on from the addFunc call and then the EBP address

```
Memory 1
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  d0 f8 de 00  ÐøÞ.
0x00DEF8B4  2d 10 18 01  -...
0x00DEF8B8  02 00 00 00  ....
0x00DEF8BC  04 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

Walking through the code…


```
21          mov ebx, [ebp + 8] ; this is a
22          mov eax, [ebp + 12] ; this is b
23          add eax, ebx
24          pop ebp ; restore ebp for the caller   ≤ 1ms elapsed
```


Registers

```
EAX = 00000006 EBX = 00000002 ECX = 0118100A EDX = 0118100A
ESI = 01181042 EDI = 0118100A ESP = 00DEF8B0
EBP = 00DEF8B0 EFL = 00000206
```


- We perform three steps now to add a and b.
- Stack is unaltered – only changed general purpose registers eax and ebx using the ebp to access values from the stack.
- The resulting sum is in eax = 2 + 4 = 6

Memory 1

```
0x00DEF8A8   00 00 00 00   ....
0x00DEF8AC   00 00 00 00   ....
0x00DEF8B0   d0 f8 de 00   ÐøÞ.
0x00DEF8B4   2d 10 18 01   -...
0x00DEF8B8   02 00 00 00   ....
0x00DEF8BC   04 00 00 00   ....
0x00DEF8C0   44 87 f7 75   D.÷u
0x00DEF8C4   00 f0 f9 00   .ðù.
0x00DEF8C8   20 87 f7 75    .÷u
```

Walking through the code…


```
    24          pop ebp ; restore ebp for the caller
    25          ret ; return  ≤ 1ms elapsed
Registers
EAX = 00000000 EBX = 00000024 ECX = 0118100A EDX = 0118100A
ESI = 0118100A EDI = 0118100A EIP = 01181043 ESP = 00DEF8B4
EBP = 00DEF8D0 EFL = 00000206
```


- We now restored EBP to original state, so that the caller function do not see any changes to it, and therefore can use it as if nothing happened.
- Pressing next now return to the next instruction in the main function.
- Note that ESP has changed: it now “removed” (i.e. not tracking the piece of memory) EBP.

Memory 1

```
0x00DEF8A8  00 00 00 00  ....
0x00DEF8AC  00 00 00 00  ....
0x00DEF8B0  d0 f8 de 00  ÐøÞ.
0x00DEF8B4  2d 10 18 01  -...
0x00DEF8B8  02 00 00 00  ....
0x00DEF8BC  04 00 00 00  ....
0x00DEF8C0  44 87 f7 75  D.÷u
0x00DEF8C4  00 f0 f9 00  .ðù.
0x00DEF8C8  20 87 f7 75   .÷u
```

Walking through the code...


```
15          add esp, 8 ; remove the
16          INVOKE ExitProcess, 0 ; exit process   ≤ 1ms elapsed
```

```
Registers
EAX = 00000000 EBX = 00000002 ECX = 0118100A EDX = 0118100A
  ESI = 0118100A EDI = 0118100A EIP = 01181030 ESP = 00DEF8C0
  EBP = 00DEF8D0 EFL = 00000216
```



```
Memory 1
0x00DEF8A8   00 00 00 00   ....
0x00DEF8AC   00 00 00 00   ....
0x00DEF8B0   d0 f8 de 00   ÐøÞ.
0x00DEF8B4   2d 10 18 01   -...
0x00DEF8B8   02 00 00 00   ....
0x00DEF8BC   04 00 00 00   ....
0x00DEF8C0   44 87 f7 75   D.÷u
0x00DEF8C4   00 f0 f9 00   .ðù.
0x00DEF8C8   20 87 f7 75    .÷u
```

- After the addFunc we reset the ESP to the original starting point by adding 8 (i.e. discounting the places where we put a and b).
- The values are still there in the stack, but will be overwritten if we have other functions using the stack.

Thank you