Virtual Memory Allocation - Recap

Tobias Nordholm-Højskov

UNIVERSITY OF COPENHAGEN

Agenda

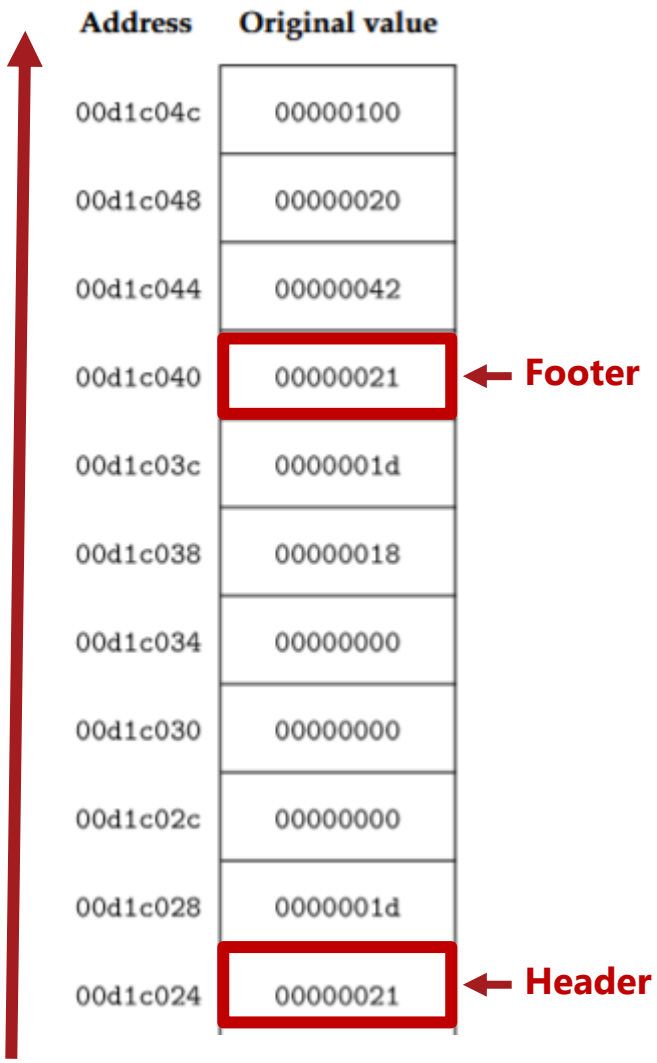
- Heap allocation basics

- Heap allocation questions from:
  - Exam 23/24
  - Exam 22/23
  - Exam 21/22

Address	Original value	After `free`	After `malloc`
00d1c06c	0BF36DFF	0BF36DFF	0BF36DFF
00d1c068	020A3400	020A3400	020A3400
00d1c064	00000041	00000041	00000041
00d1c060	00000012		
00d1c05c	00000000		
00d1c058	00000000	00000000	00000000
00d1c054	00000012		
00d1c050	00000019		
00d1c04c	00003400		
00d1c048	00C01DB0		
00d1c044	00035408		
00d1c040	0000E870		
00d1c03c	00000019	00000019	00000019
00d1c038	00000012	00000012	00000012
00d1c034	00000000	00000000	00000000
00d1c030	00000000	00000000	00000000
00d1c02c	00000012		
00d1c028	00000011		
00d1c024	0B367BD7	0B367BD7	0B367BD7
00d1c020	1A8F959E	1A8F959E	1A8F959E
00d1c01c	00000011	00000011	00000011

The Heap

- The heap grows bottom-up
- Comprised of blocks encased in a header and footer.
  - The header and footer contain information about the block, including its size and allocation status (allocated or free).
  - The header and footer of the same block will always have the same value.
  - The header and footer each take up 4 bytes and count towards the entire blocks size.
- Each address represents 4 bytes
- A blocks size must always be a multiple of 8



Decoding the values

- Usually true, but read the question to make sure
- Convert hex values to binary
  - Bit 0 - indicates whether the current block is allocated or free
  - Bit 1 - indicates whether the previous block is allocated or free
  - Bit 2 – is always 0
- As shown:

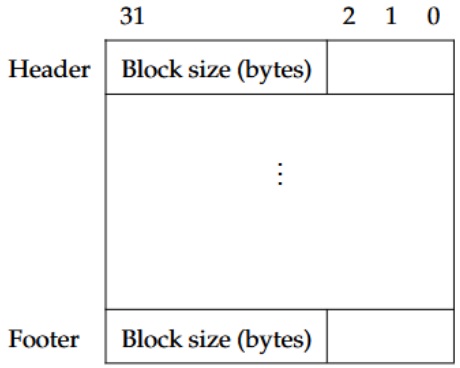
0x21

- Bit 0 – Is allocated
- Bit 1 – Previous is unallocated
- Size of block - calculated by setting lower 3 bits to 0

0010 0001 → 00100000 = 0x20 or 32

Question Overview – Exam 23/24

Question 2.1.3: Consider a memory allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size that is the number of payload words. Only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The minimum block size is 2 words. The usage of the remaining 3 lower order bits is as follows:

- Bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- Bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- Bit 2 is unused and is always set to be 0.

Given the partial contents of the heap shown on the left, show the new contents of the heap after a call `free(0xd1c040)` is executed (in the middle column), followed by a call `malloc(30)` that returns `0xd1c030` (rightmost column).

- All numbers are hexadecimal, and so should your answers be.
- Note that the address grows from bottom up.
- Some parts of the heap may lie outside the area shown.
- Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.
- Perform the minimum number of memory changes required.

- Instructions to execute are either going to be:
  - free()
  - malloc()
  - realloc()

- Read the question carefully as to not wrongly interpret what is actually asked

Address	Original value	free(0xd1c040) After free	malloc(30) returning 0xd1c030 After malloc
00d1c06c	0BF36DFF	0BF36DFF	0BF36DFF
00d1c068	020A3400	020A3400	020A3400
00d1c064	00000041	00000041	00000041
00d1c060	00000012		
00d1c05c	00000000		
00d1c058	00000000	00000000	00000000
00d1c054	00000012		
00d1c050	00000019		
00d1c04c	00003400		
00d1c048	00C01DB0		
00d1c044	00035408		
00d1c040	0000E870		
00d1c03c	00000019	00000019	00000019
00d1c038	00000012	00000012	00000012
00d1c034	00000000	00000000	00000000
00d1c030	00000000	00000000	00000000
00d1c02c	00000012		
00d1c028	00000011		
00d1c024	0B367BD7	0B367BD7	0B367BD7
00d1c020	1A8F959E	1A8F959E	1A8F959E
00d1c01c	00000011	00000011	00000011

- Begin by locating the instructed address
- Identify all blocks in the heap (find header & footers)

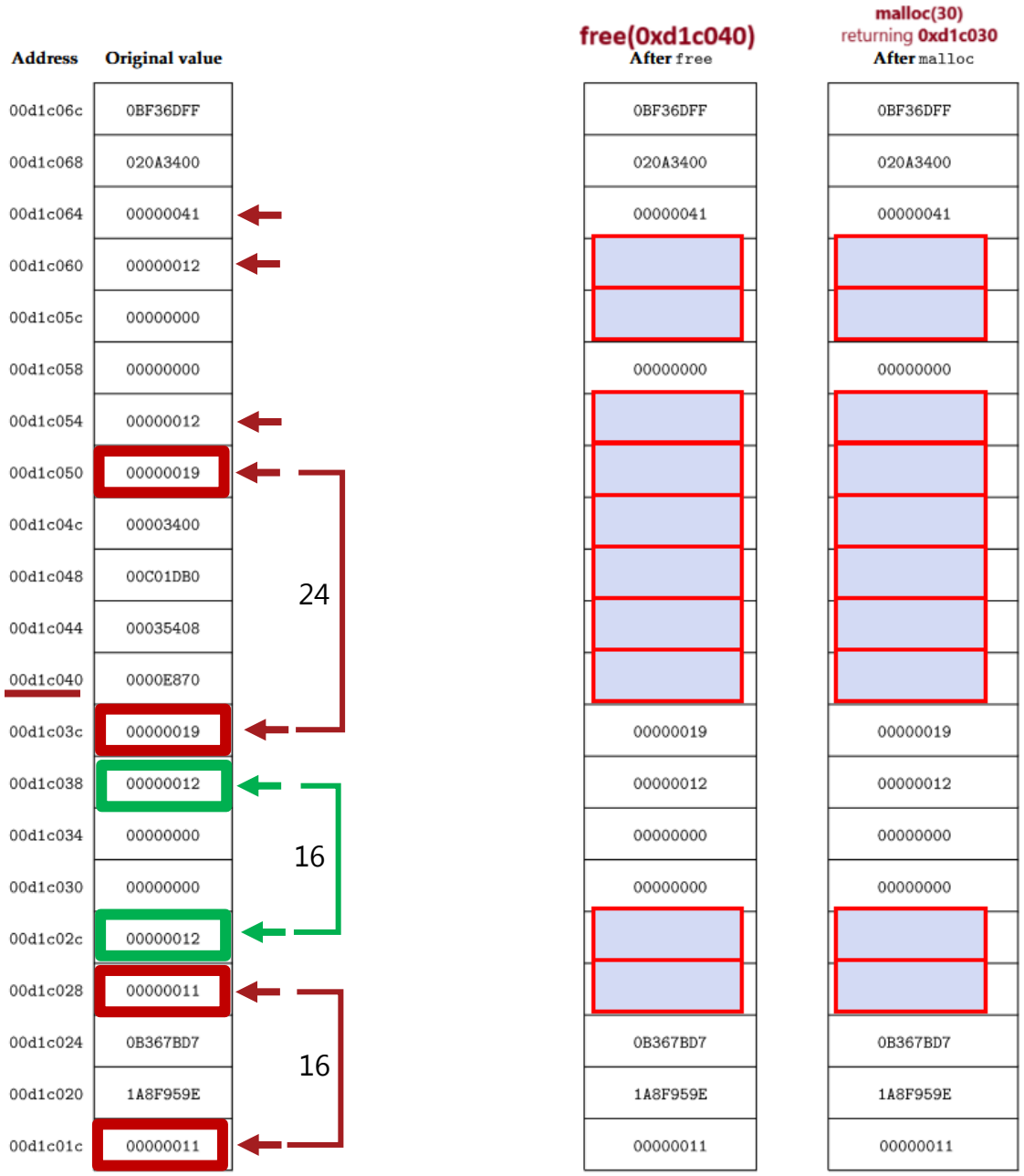
Address	Original value		free(0xd1c040) After free	malloc(30) returning 0xd1c030 After malloc
00d1c06c	0BF36DFF		0BF36DFF	0BF36DFF
00d1c068	020A3400		020A3400	020A3400
00d1c064	00000041	←Header?	00000041	00000041
00d1c060	00000012	← Footer?		
00d1c05c	00000000			
00d1c058	00000000		00000000	00000000
00d1c054	00000012	←Header?		
00d1c050	00000019	← Footer?		
00d1c04c	00003400			
00d1c048	00C01DB0			
00d1c044	00035408			
00d1c040	0000E870			
00d1c03c	00000019	←Header?	00000019	00000019
00d1c038	00000012	← Footer?	00000012	00000012
00d1c034	00000000		00000000	00000000
00d1c030	00000000		00000000	00000000
00d1c02c	00000012	←Header?		
00d1c028	00000011	← Footer?		
00d1c024	0B367BD7		0B367BD7	0B367BD7
00d1c020	1A8F959E		1A8F959E	1A8F959E
00d1c01c	00000011	←Header?	00000011	00000011

- Lets look at the header of a possible block:

0x19

0001 1001

- Allocated
- Previous Unallocated
- Size = 00011000 = 0x18 = **24**



- Lets look at the header of a possible block:

0x19 ↓ 0001 1001

- Allocated
- Previous Unallocated
- Size = 00011000 = 0x18 = **24**

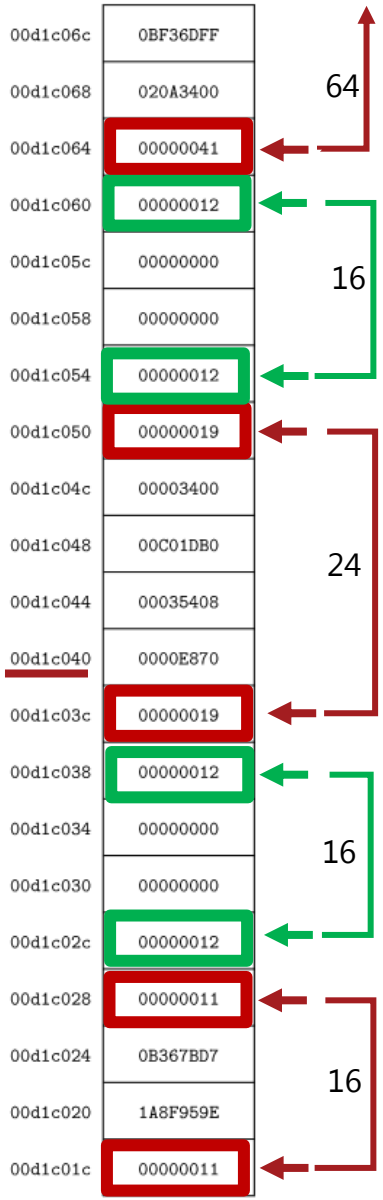
- Now we identify the remaining possible blocks:

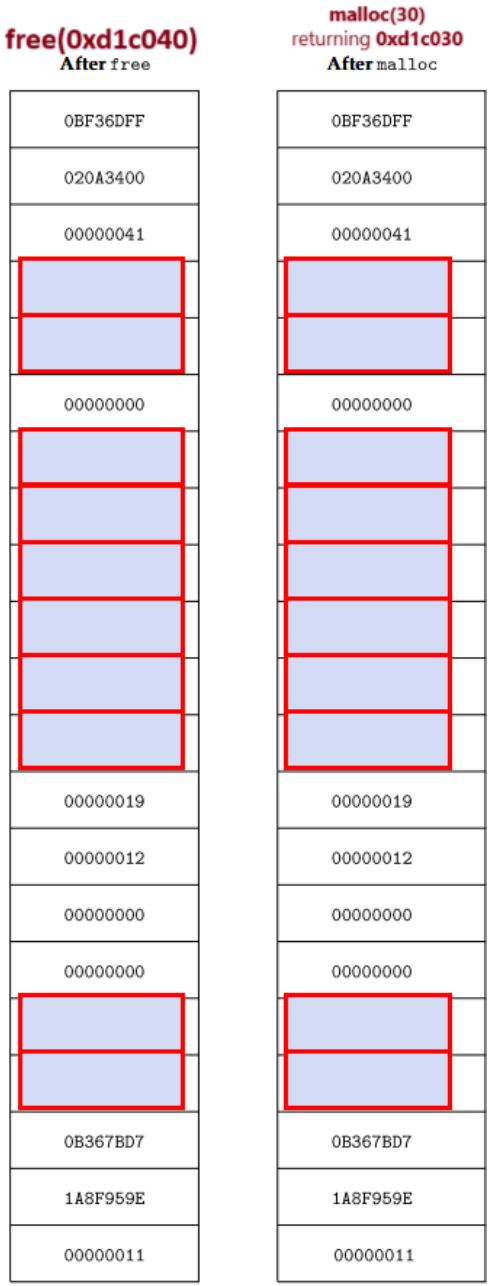
0x12 ↓ 0001 0010

- Free
- Previous Allocated
- Size = 00010000 = 0x10 = **16**

0x11 ↓ 0001 0001

- Allocated
- Previous Unallocated
- Size = 00010000 = 0x10 = **16**





0x12

0001 0010

- Free
- Previous Allocated
- Size = 00010000 = 0x10 = **16**

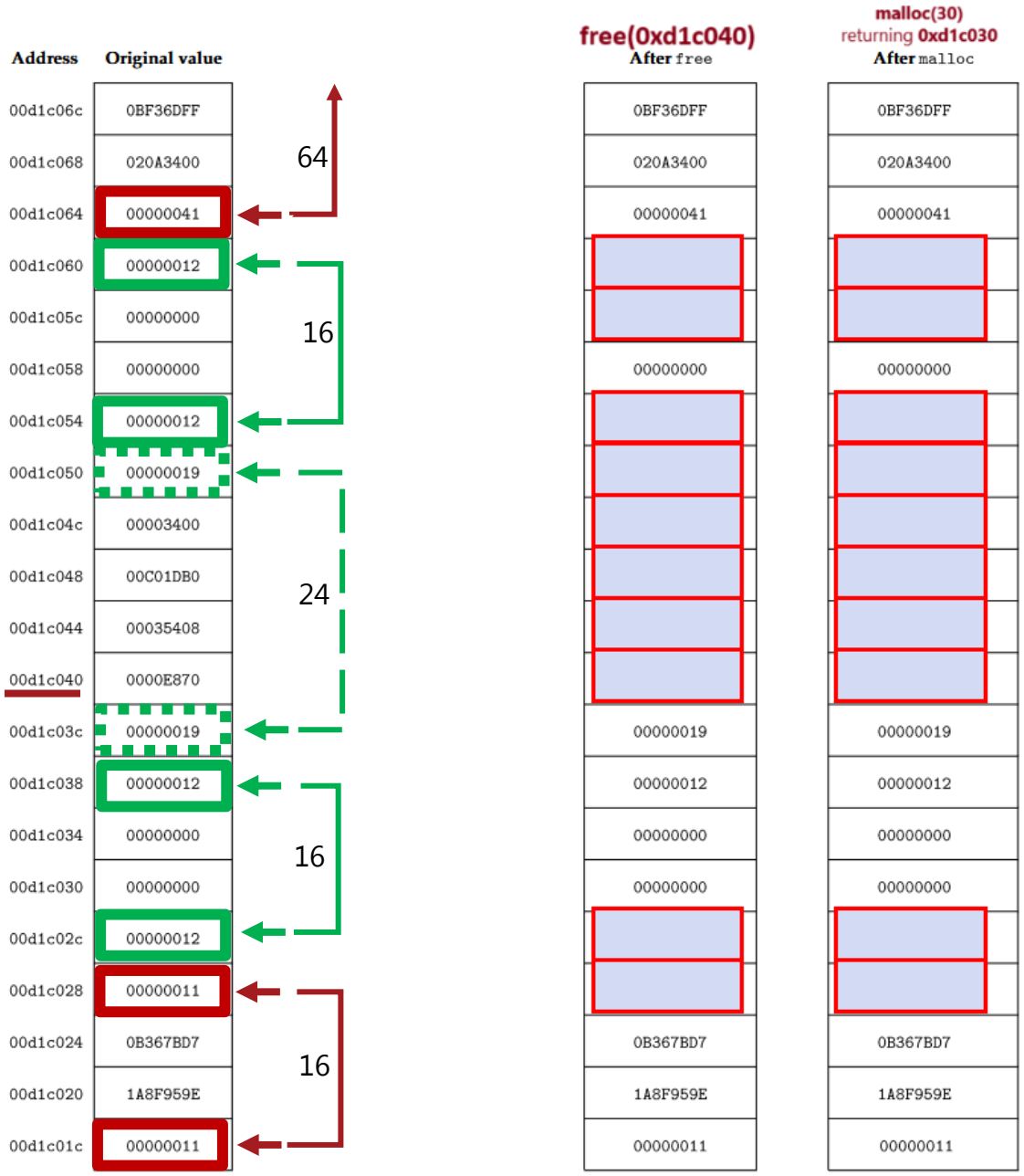
0x41

0100 0001

- Allocated
- Previous Unallocated
- Size = 01000000 = 0x40 = **64**

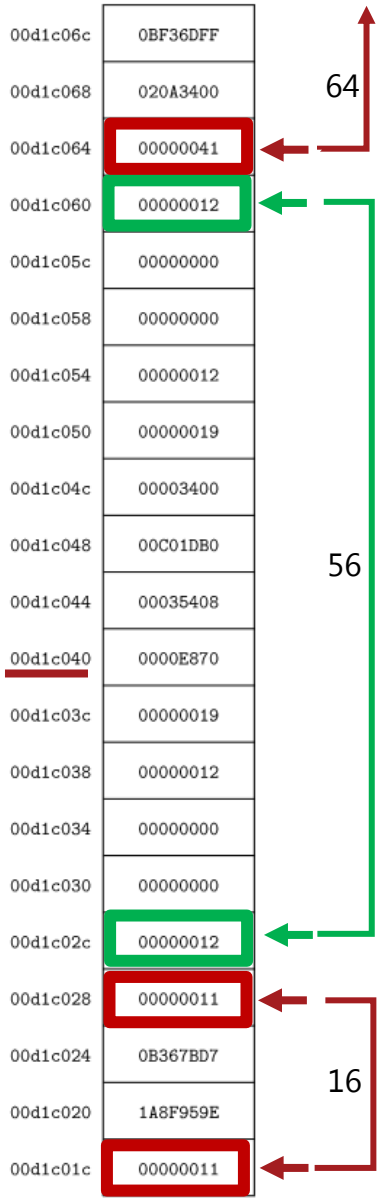
- This heap has a big block which is not fully located within scope

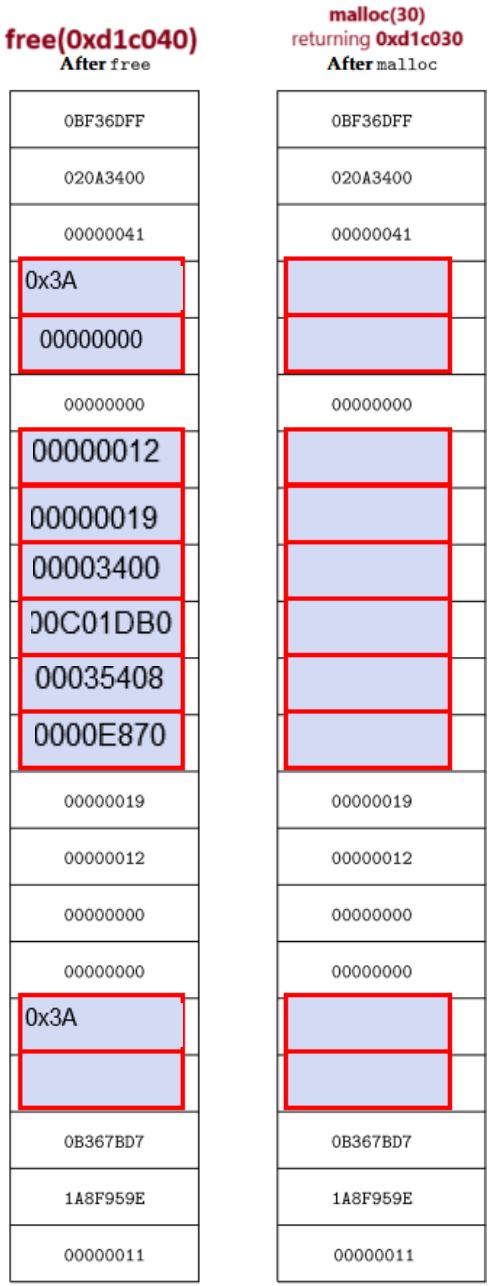
- With all blocks identified we are able to free the requested block



- In dotted green is the block we want to free, however The block is located between **2x 16 byte free blocks**, and as the question asked we are dealing with immediate coalescing

- So all the free blocks are combined into **1 block** of size:
  - 16 + 24 + 16 = **56**





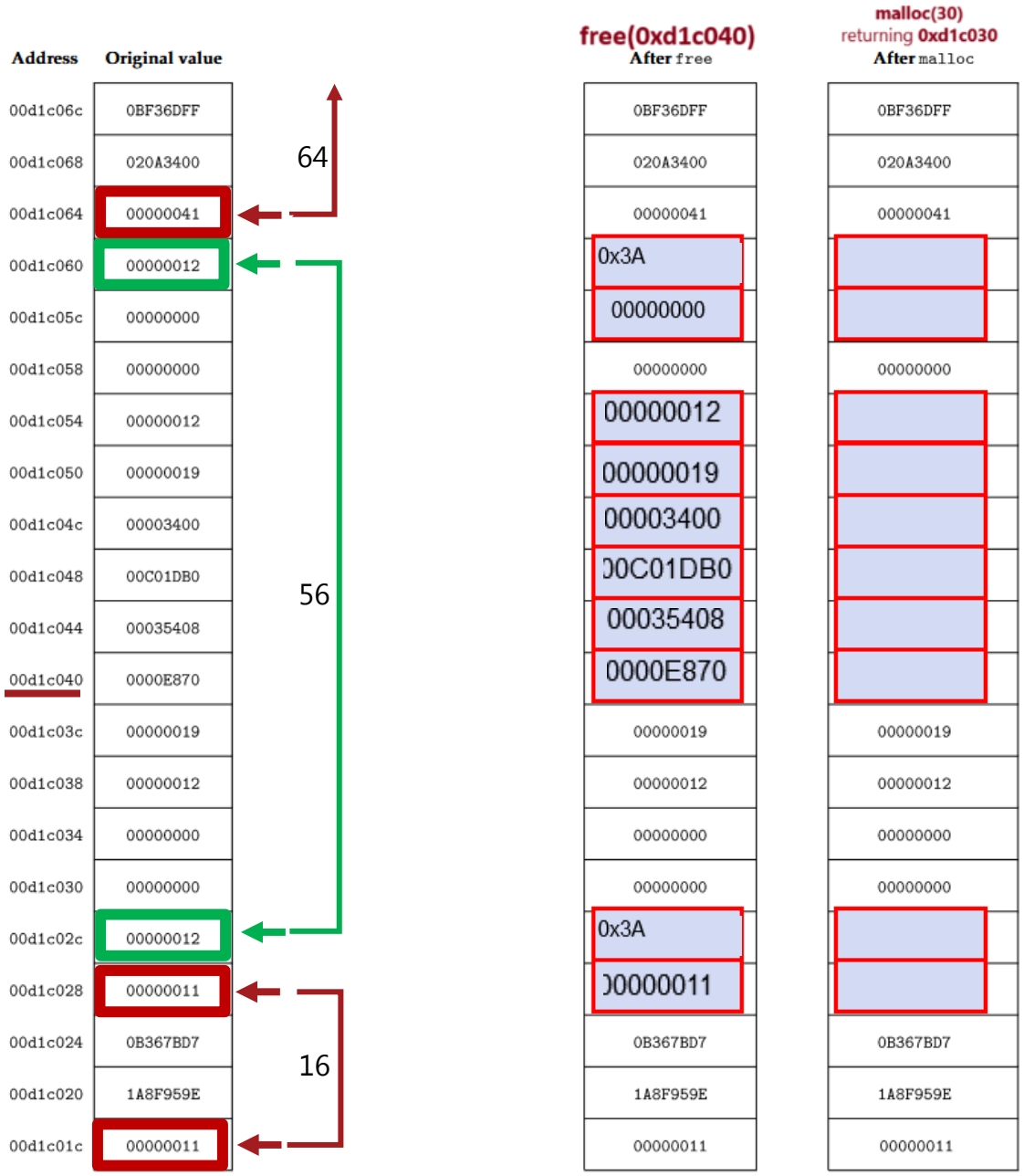
- In dotted green is the block we want to free, however
- The block is located between **2x 16 byte free blocks**, and as the question asked we are dealing with immediate coalescing

- So all the free blocks are combined into **1 block** of size:
  - 16 + 24 + 16 = **56**

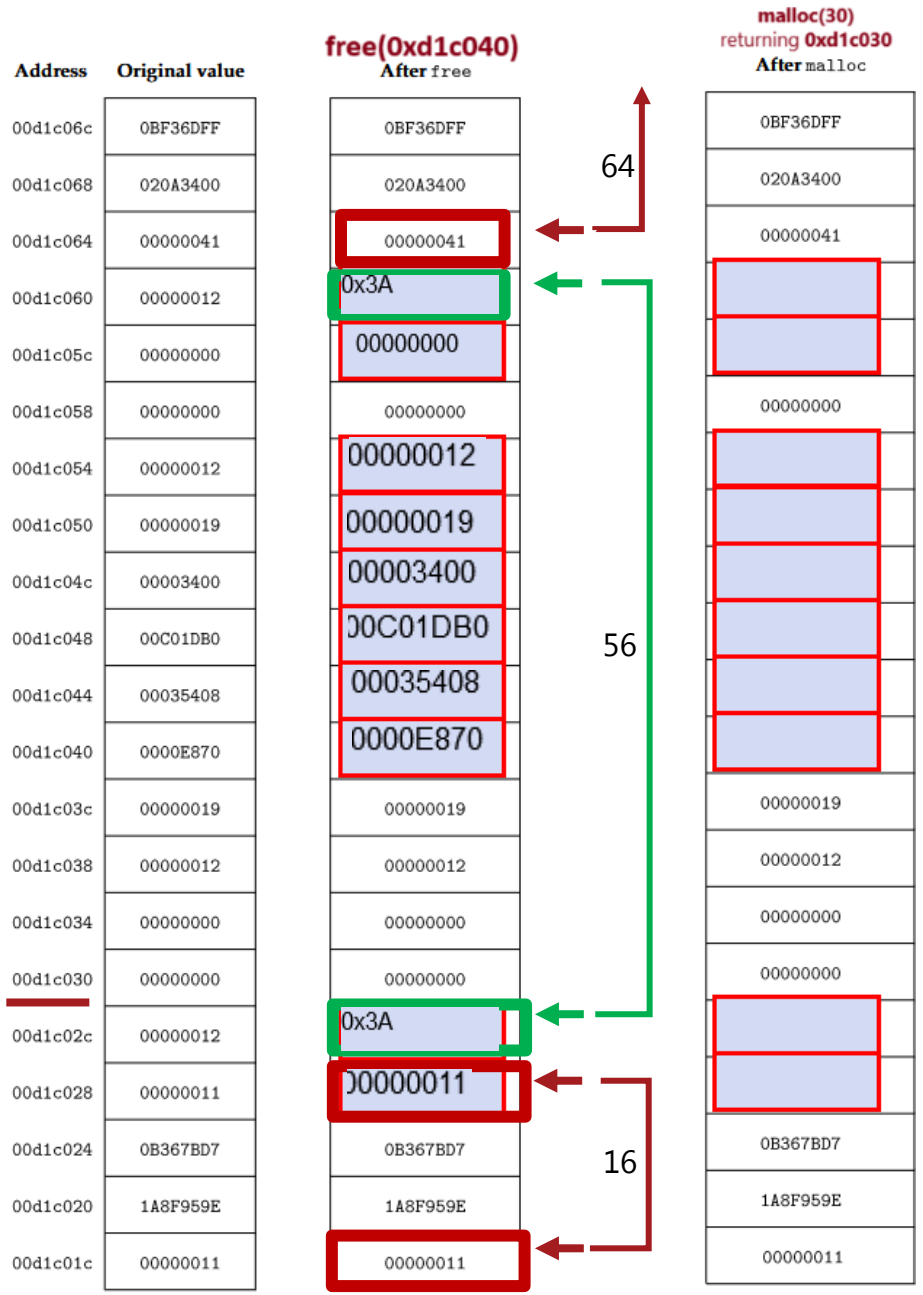
- Now we just need to update the blocks header and footer to reflect this change:
  - Bit 0 – Free
  - Bit 1 – Previous allocated
  - Size = 56

  - 00111010 = **0x3A**

- The values in-between remain unchanged, and what was the header and footer before is now just garbage data

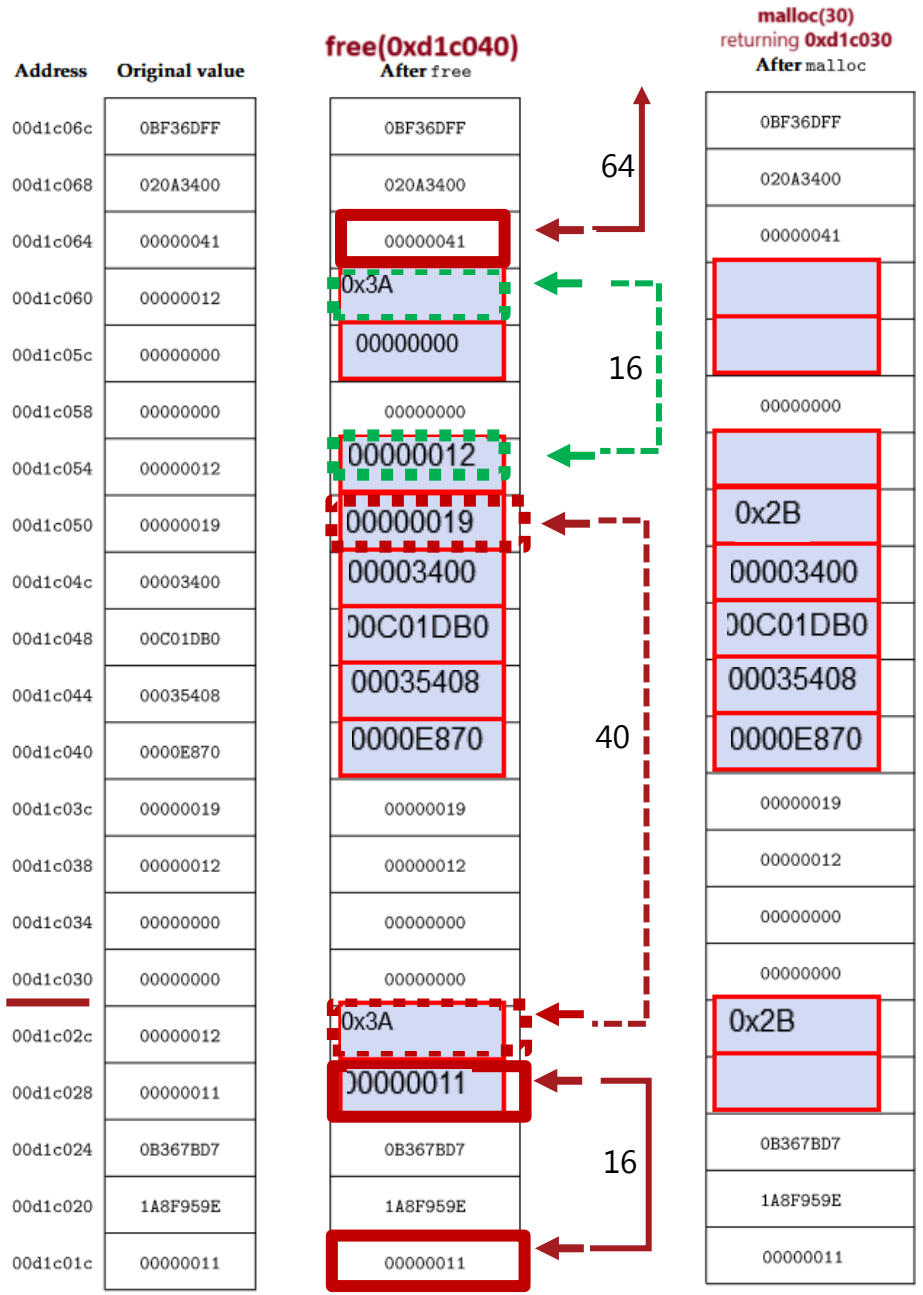


- Finally the footer of the block beneath remains unchanged as the block above is still free and its size has not changed.
- We now move onto performing the malloc request.

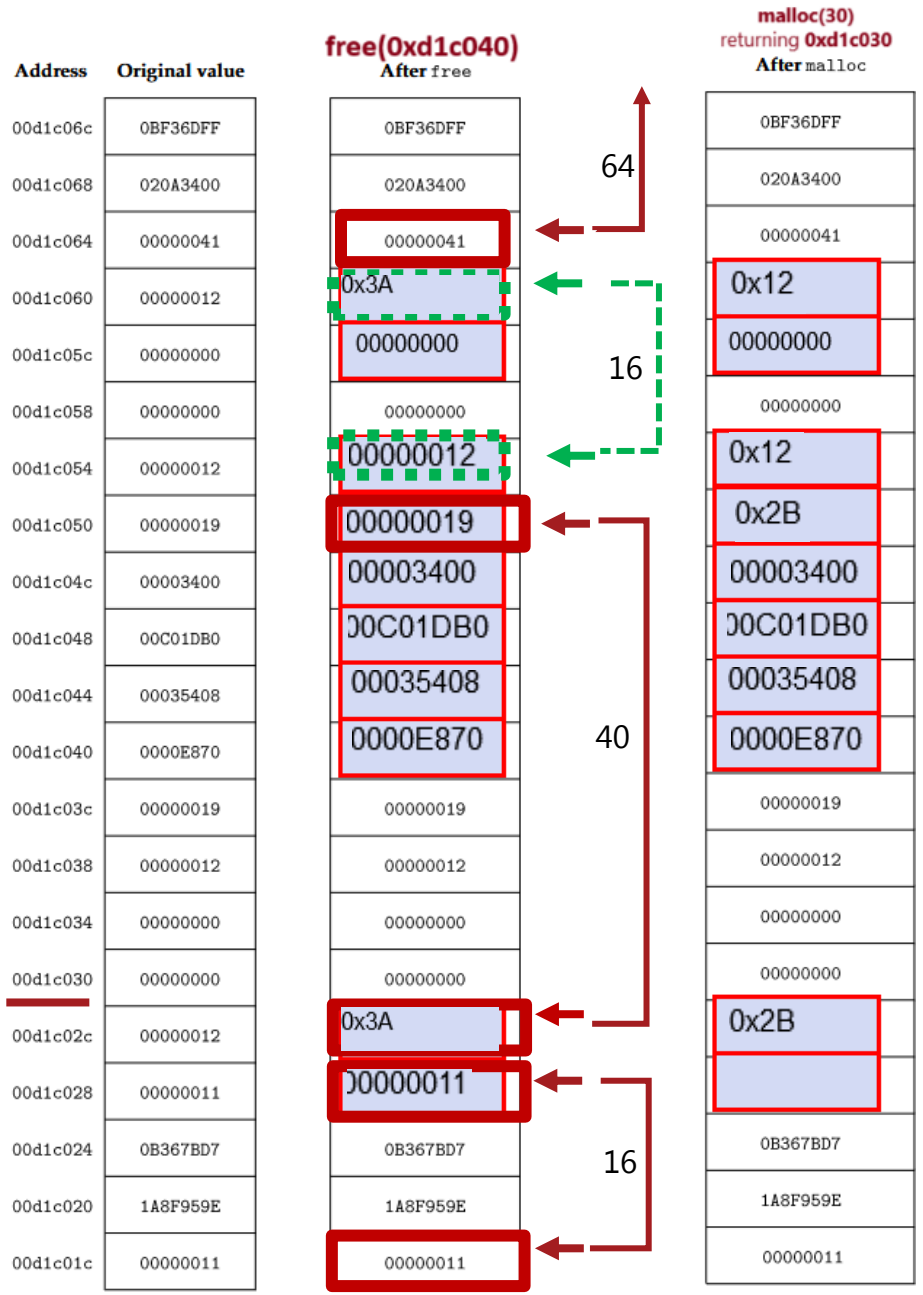


- It is important to remember we are working off how the heap looks after the free request was done, and not how it looked originally

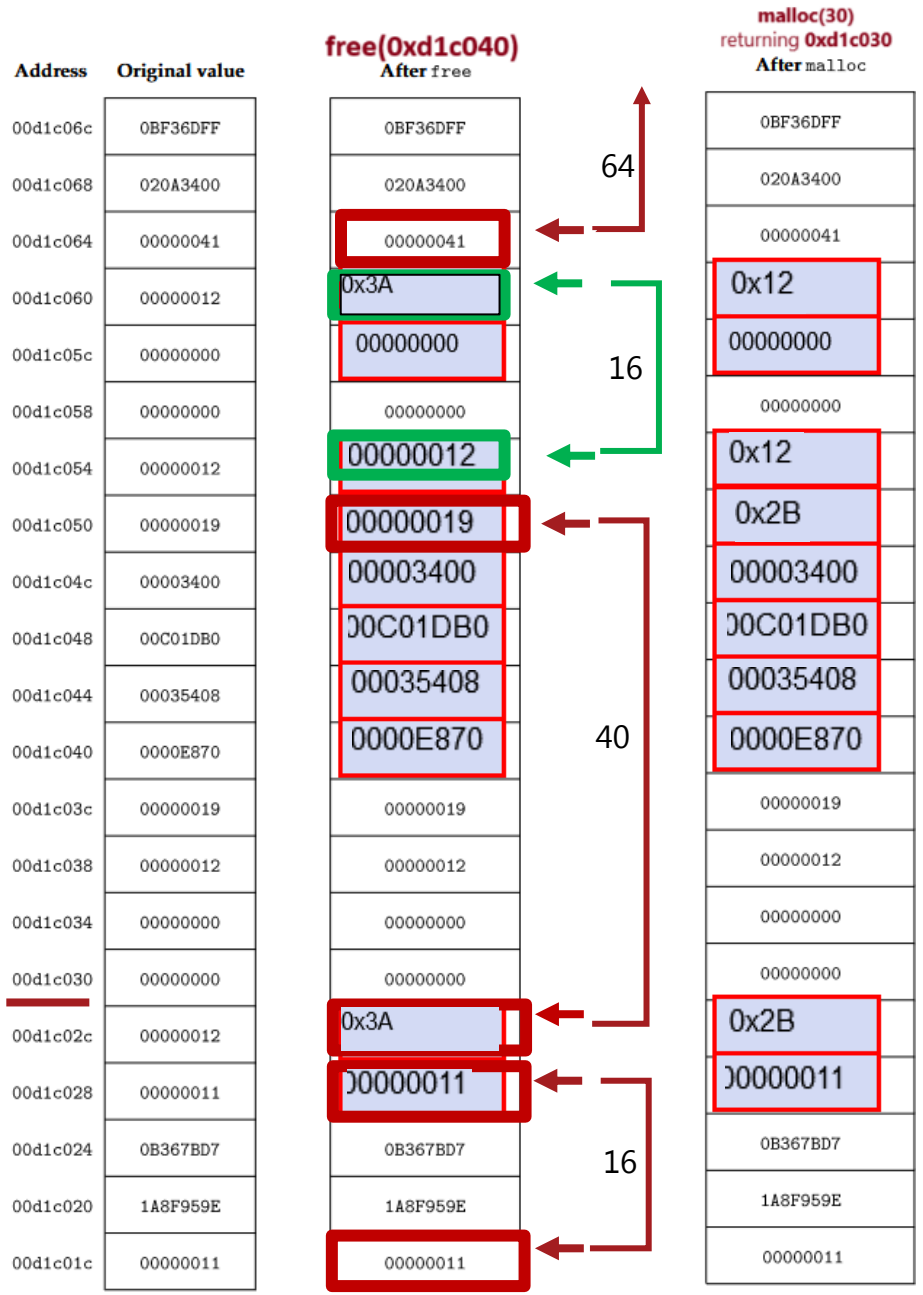
- The size of the new block is **30-bytes** + 8 for header and footer = **38**
- However 38 is not a multiple of 8 so the actual block size is going to be 40 bytes.



- We now need to update the headers of the 2 new blocks we have created, starting with our allocated space:
  - Bit 0 – Allocated
  - Bit 1 – Previous Allocated
  - Size = 40
  - 00101011 = **0x2B**

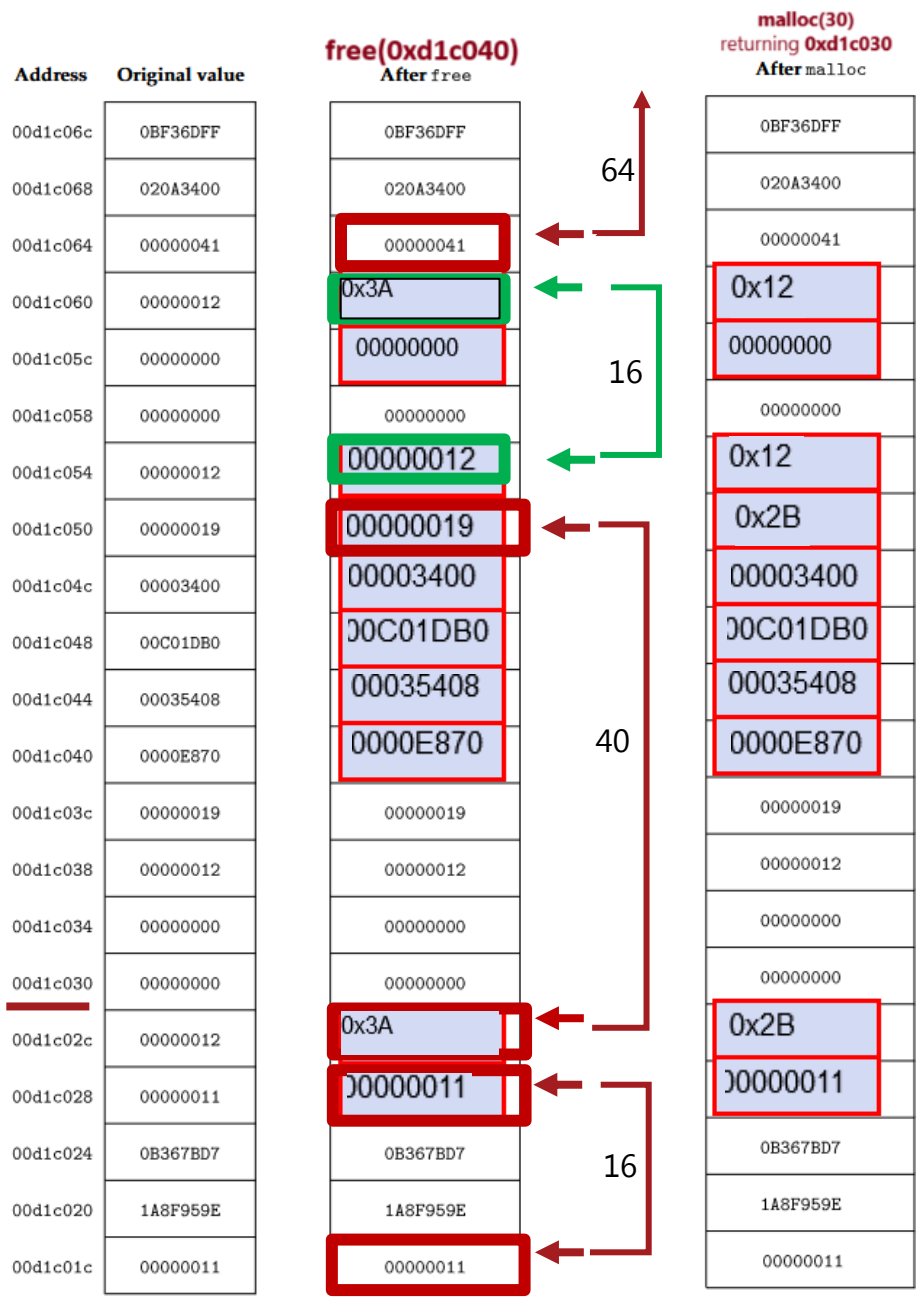


- Now the block above:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16
  - 00010010 = **0x12**



- Now the block above:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16
  - 00010010 = **0x12**

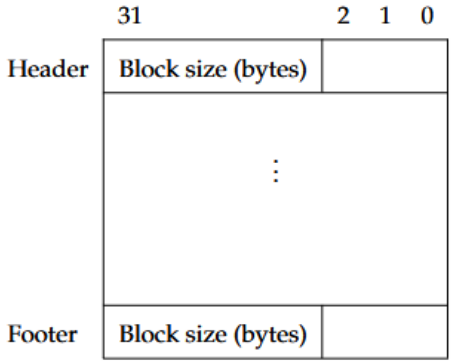
- Lastly the first block in the visible heap remains unchanged



Address	Original value	After free	After malloc
00d1c06c	0BF36DFF	0BF36DFF	0BF36DFF
00d1c068	020A3400	020A3400	020A3400
00d1c064	00000041	00000041	00000041
00d1c060	00000012	0000003A	00000012
00d1c05c	00000000	00000000	00000000
00d1c058	00000000	00000000	00000000
00d1c054	00000012	00000012	00000012
00d1c050	00000019	00000019	0000002B
00d1c04c	00003400	00003400	00003400
00d1c048	00C01DB0	00C01DB0	00C01DB0
00d1c044	00035408	00035408	00035408
00d1c040	0000E870	0000E870	0000E870
00d1c03c	00000019	00000019	00000019
00d1c038	00000012	00000012	00000012
00d1c034	00000000	00000000	00000000
00d1c030	00000000	00000000	00000000
00d1c02c	00000012	0000003A	0000002B
00d1c028	00000011	00000011	00000011
00d1c024	0B367BD7	0B367BD7	0B367BD7
00d1c020	1A8F959E	1A8F959E	1A8F959E
00d1c01c	00000011	00000011	00000011

Question Overview – Exam 22/23

Question 2.3.1: Consider an allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:

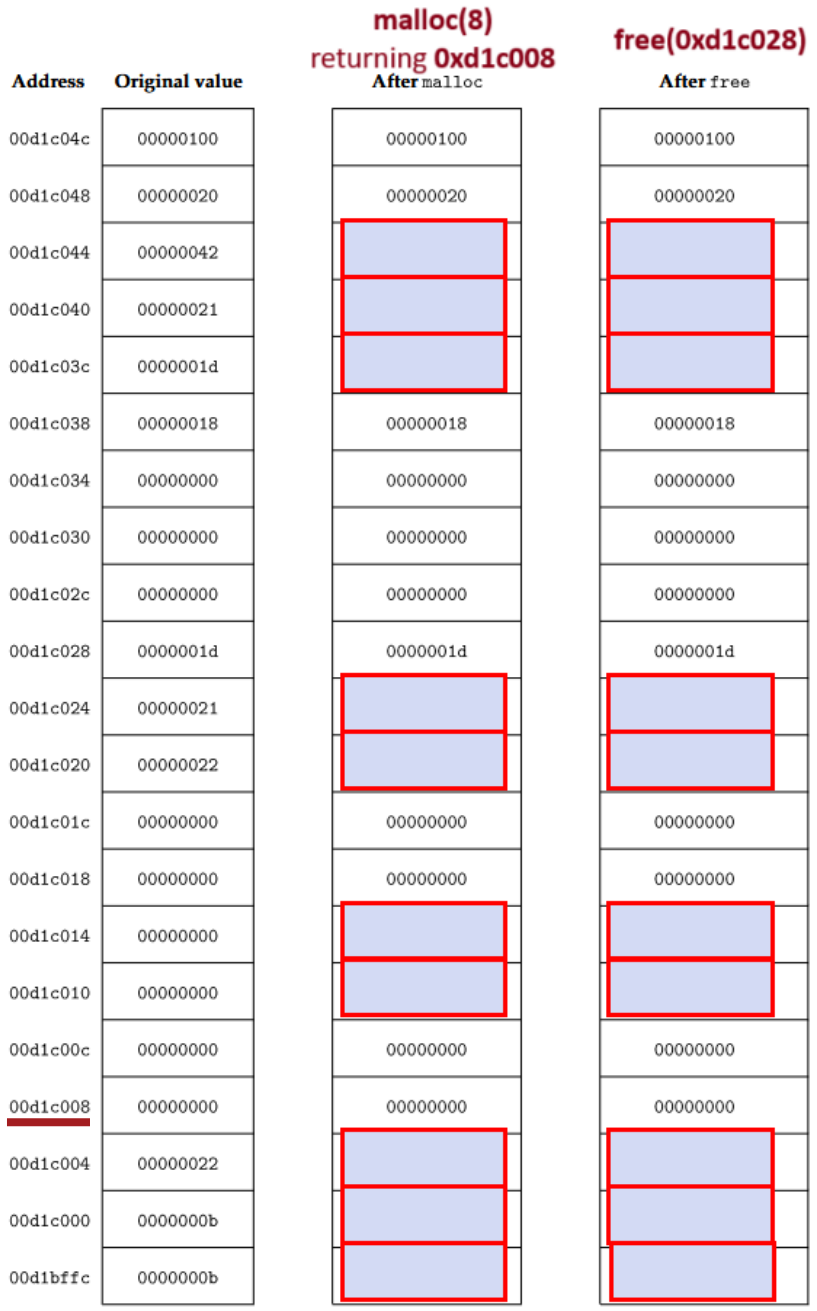


Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The minimum block size is 8. The usage of the remaining 3 lower order bits is as follows:

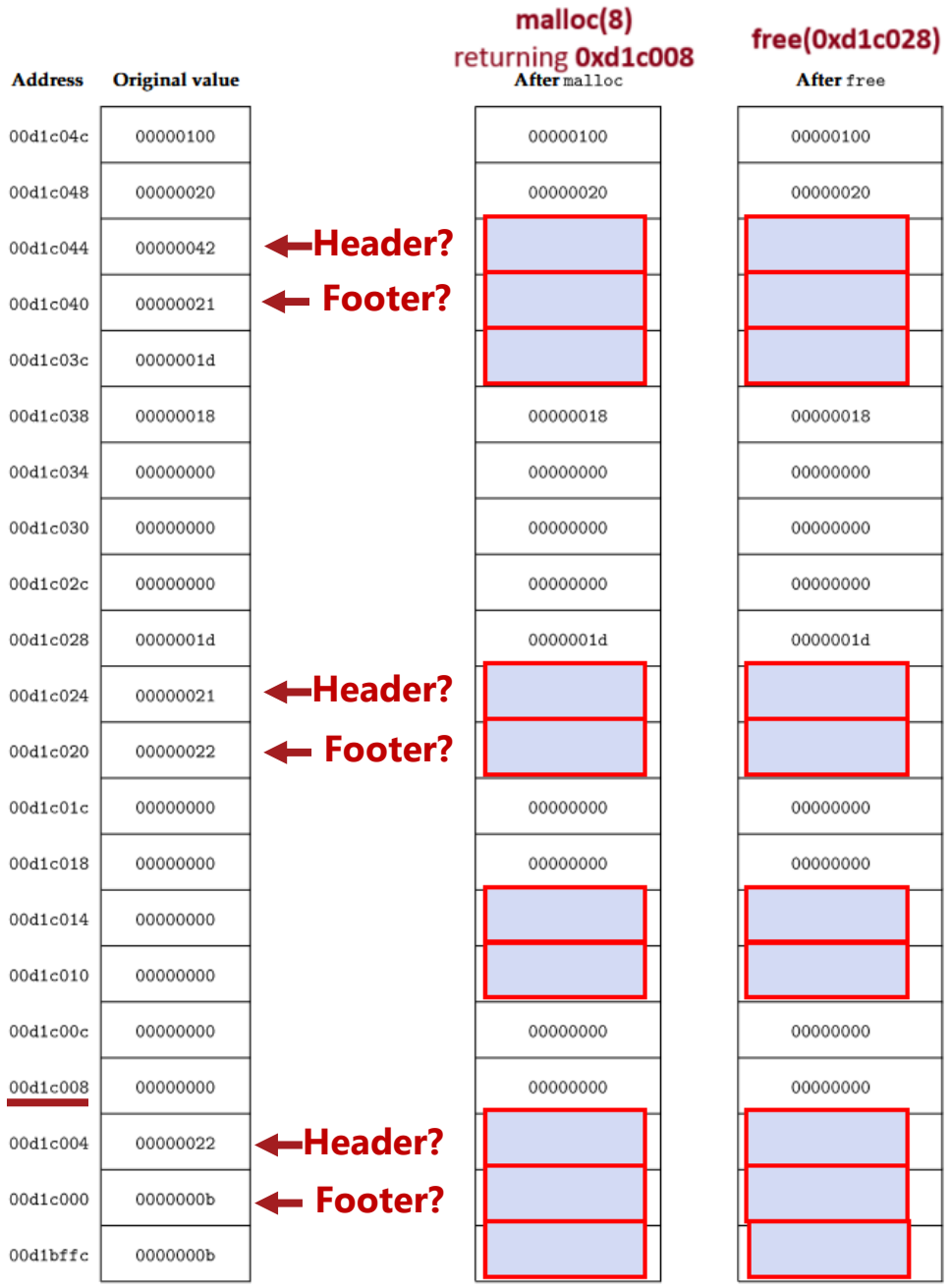
- Bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- Bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- Bit 2 is unused and is always set to be 0.

Given the partial contents of the heap shown on the left, show the new contents of the heap after a call to `malloc(8)` is executed that returns `0xd1c008` (in the middle column), followed by a call `free(0xd1c028)` (rightmost column).

- All numbers are hexadecimal, and so should your answers be.
- Note that the address grows from bottom up.
- Some parts of the heap may lie outside the area shown.
- Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.
- Perform the minimum number of memory changes required.

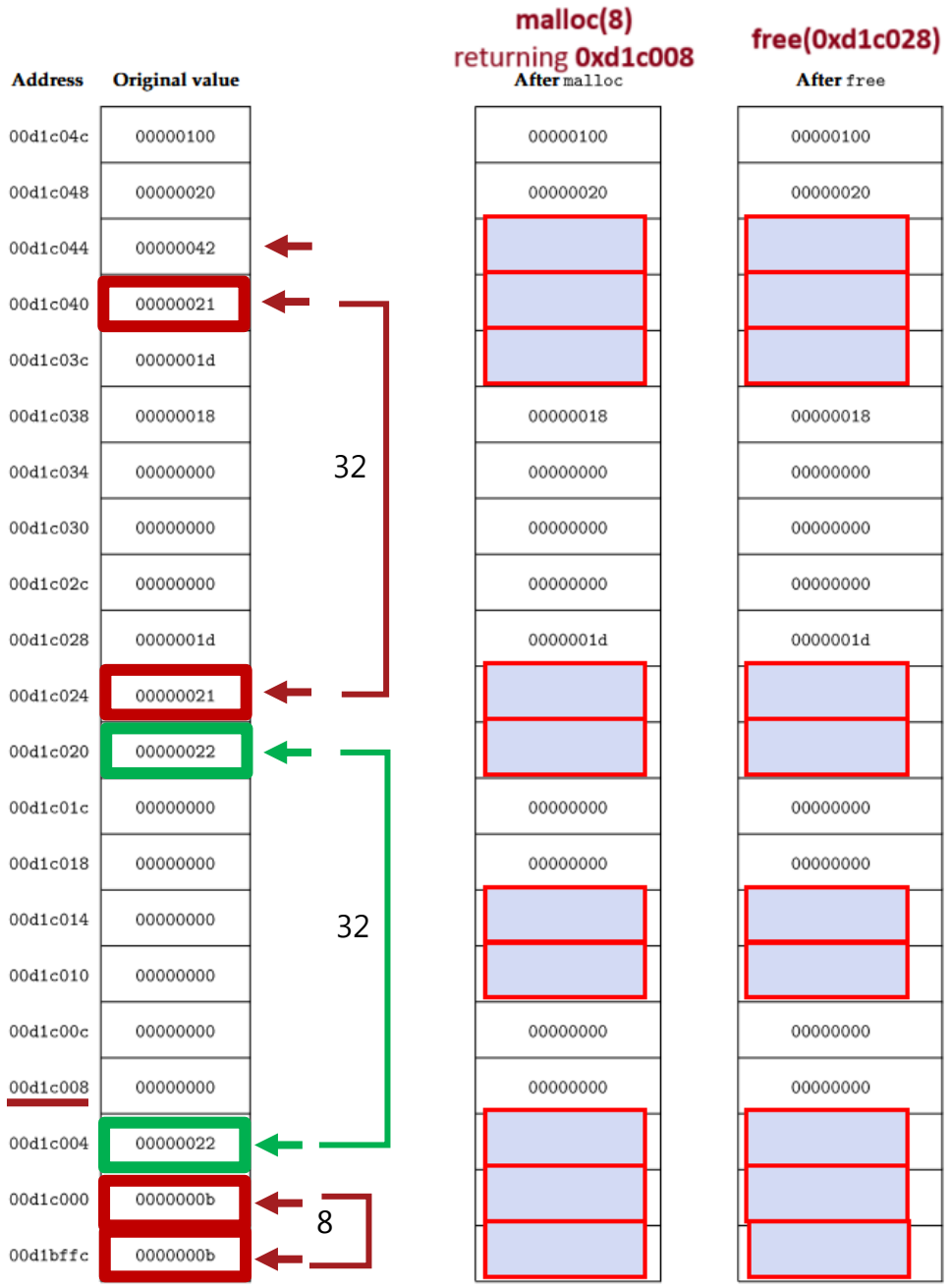


- Again begin by locating the instructed address and identify all blocks in the heap



- Again begin by locating the instructed address and identify all blocks in the heap
- Now lets verify:

- Free
- Previous Allocated
- Size = 00100000 = 0x20 = **32**



- Again begin by locating the instructed address and identify all blocks in the heap
- Now lets verify:

0x22

↓

0010 0010

- Free
- Previous Allocated
- Size = 00100000 = 0x20 = **32**

0x0b

↓

0000 1011

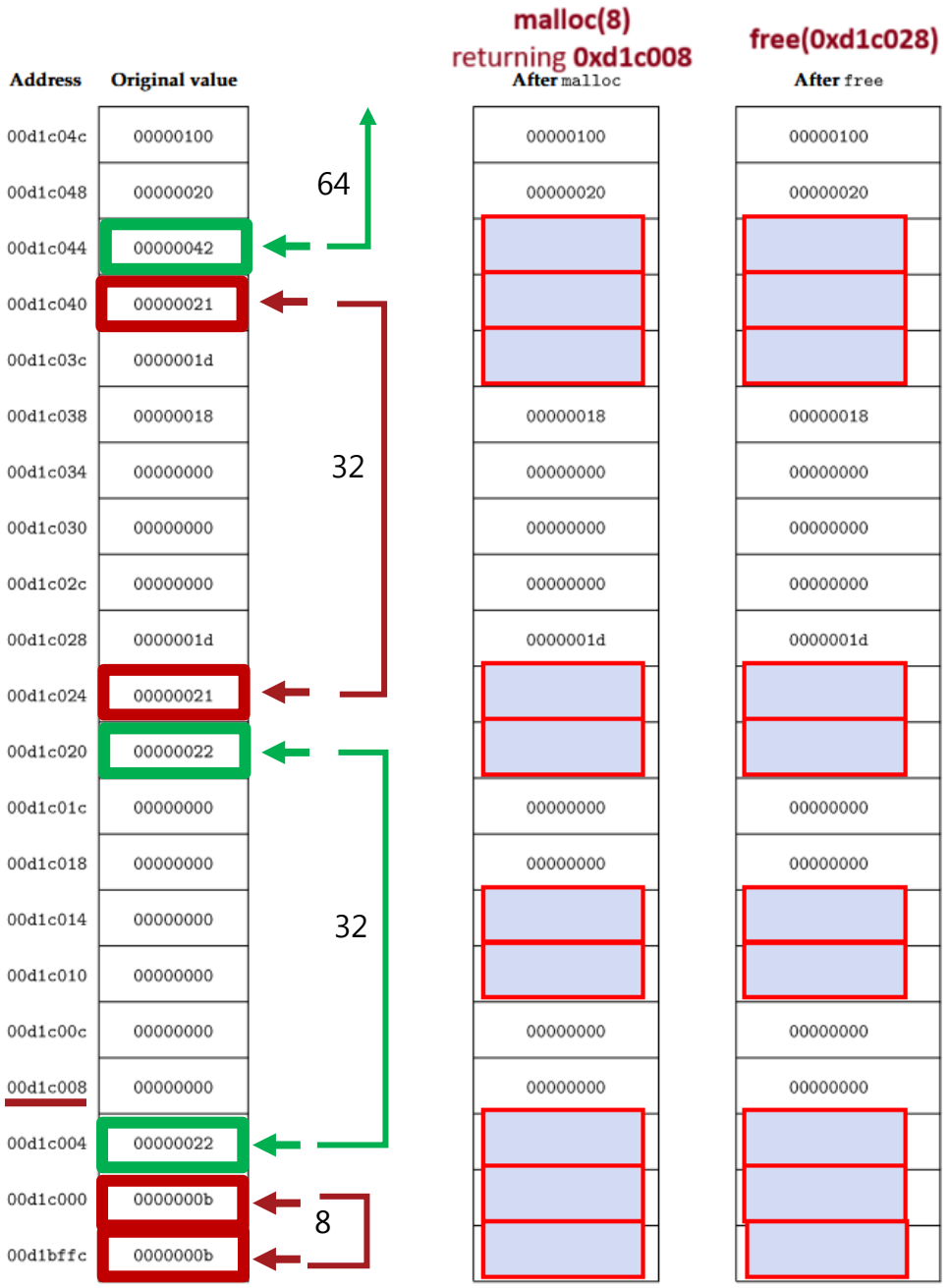
- Allocated
- Previous Allocated
- Size = 00001000 = 0x8 = **8**

0x21

↓

0010 0001

- Allocated
- Previous Unallocated
- Size = 00100000 = 0x20 = **32**



0x42

↓

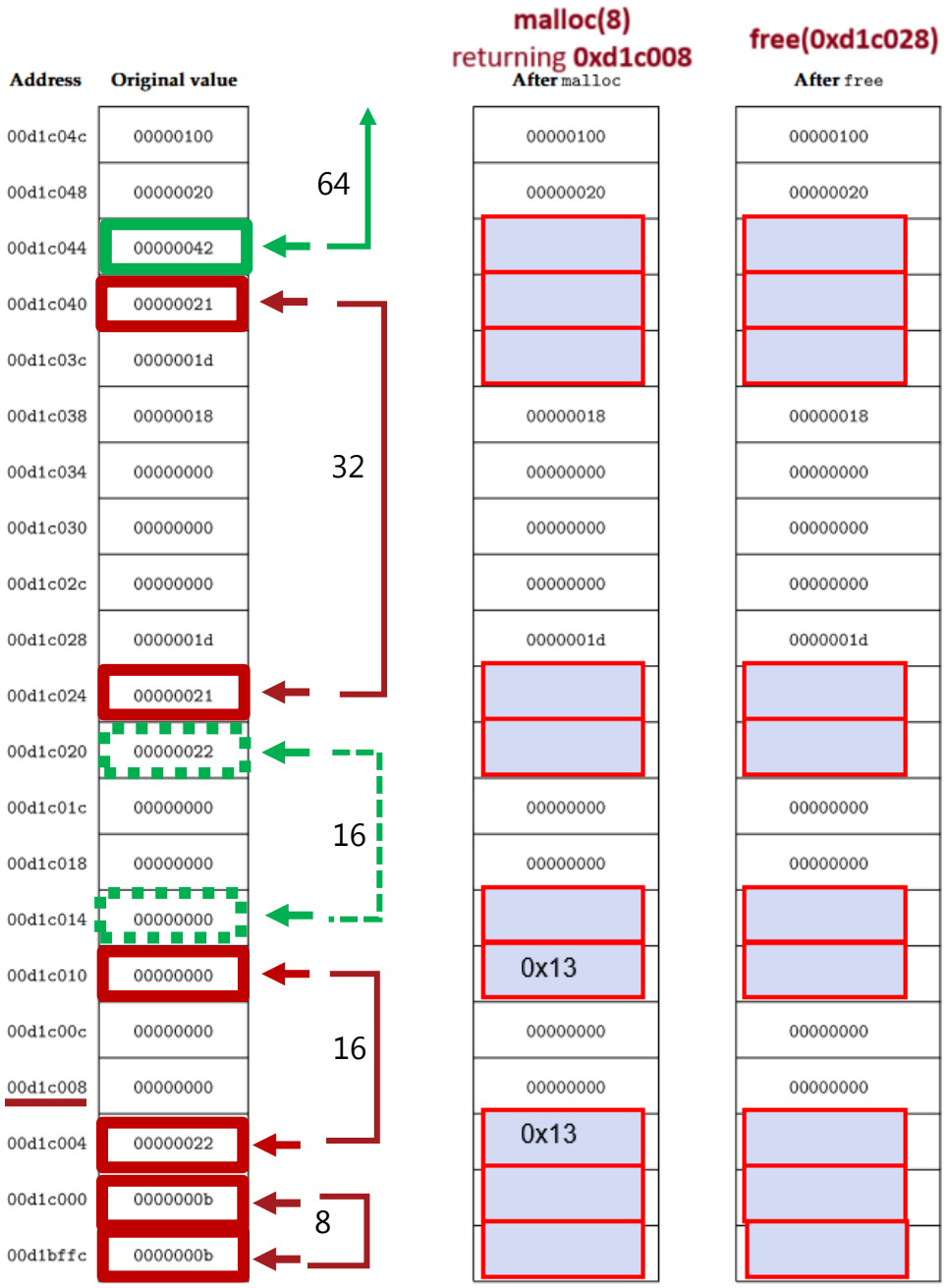
0100 0010

- Free
- Previous Allocated
- Size = 01000000 = 0x40 = **64**

- Knowing all blocks we can now malloc:
  - Bit 0 – Allocated
  - Bit 1 – Previous Allocated
  - Size = 8-bytes + header & footer = **16-bytes**

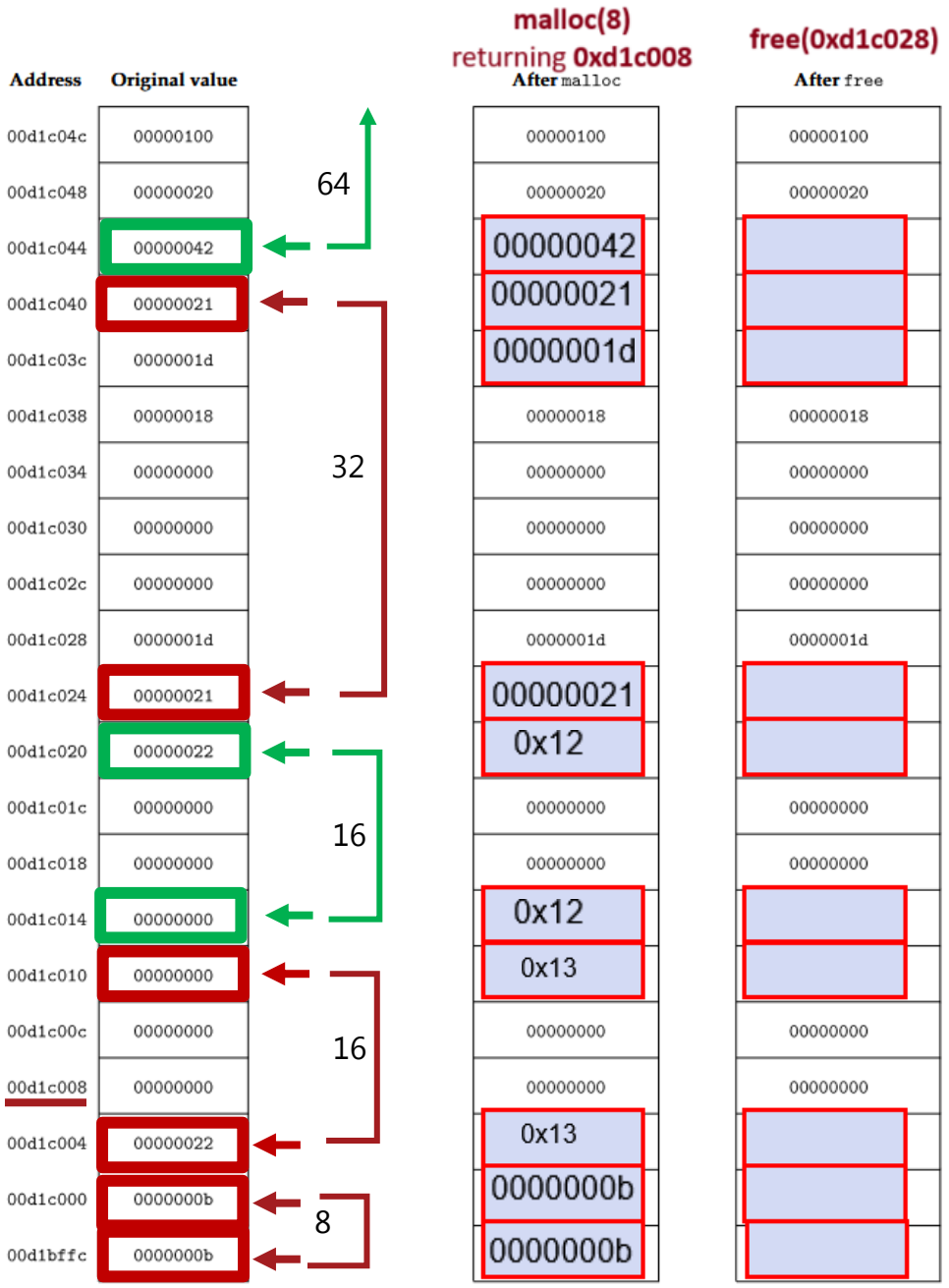
↓

- 00010011 = **0x13**



- We then update the new smaller free block:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16-bytes

  - 00010010 = **0x12**

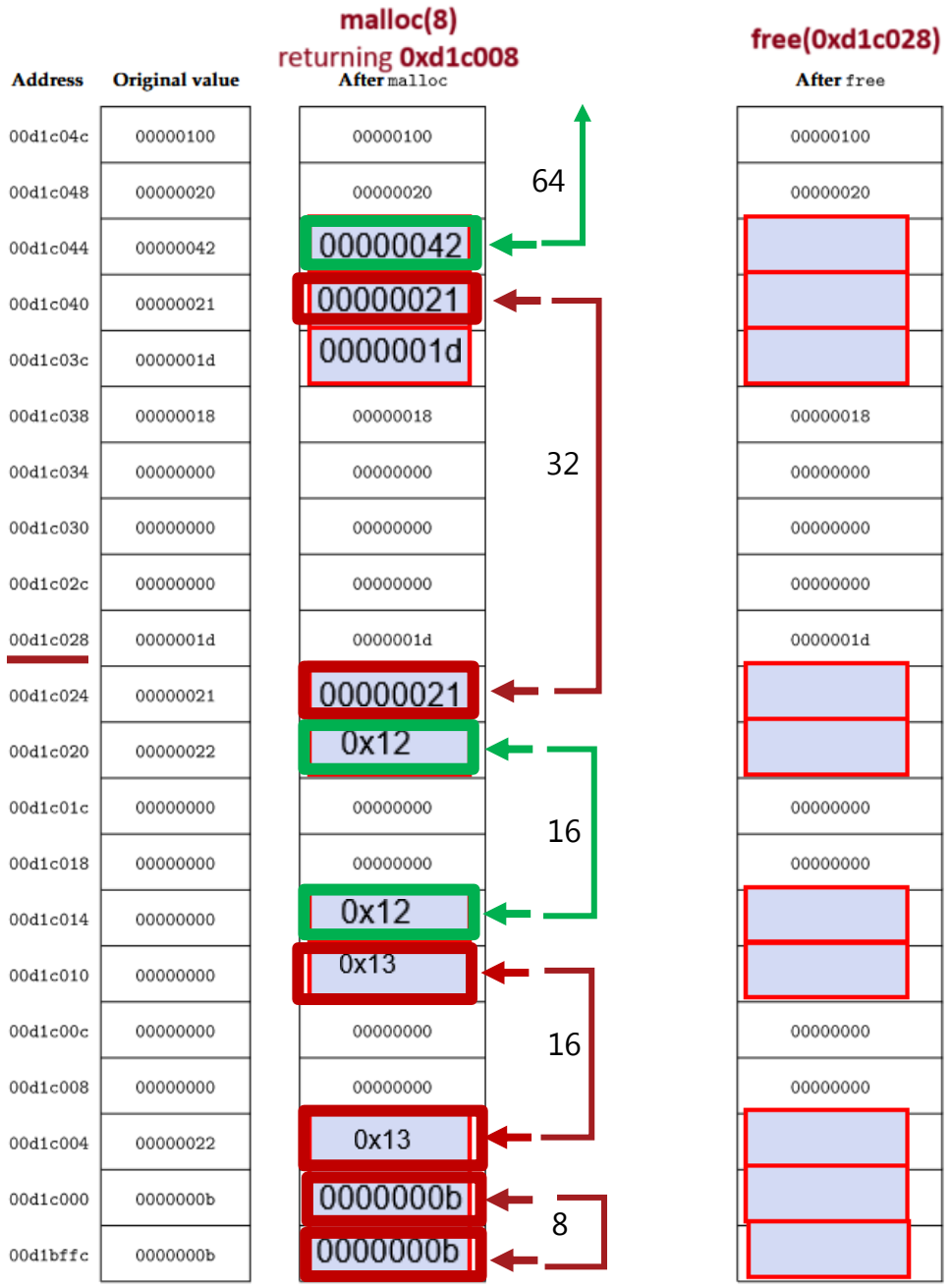


- We then update the new smaller free block:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16-bytes

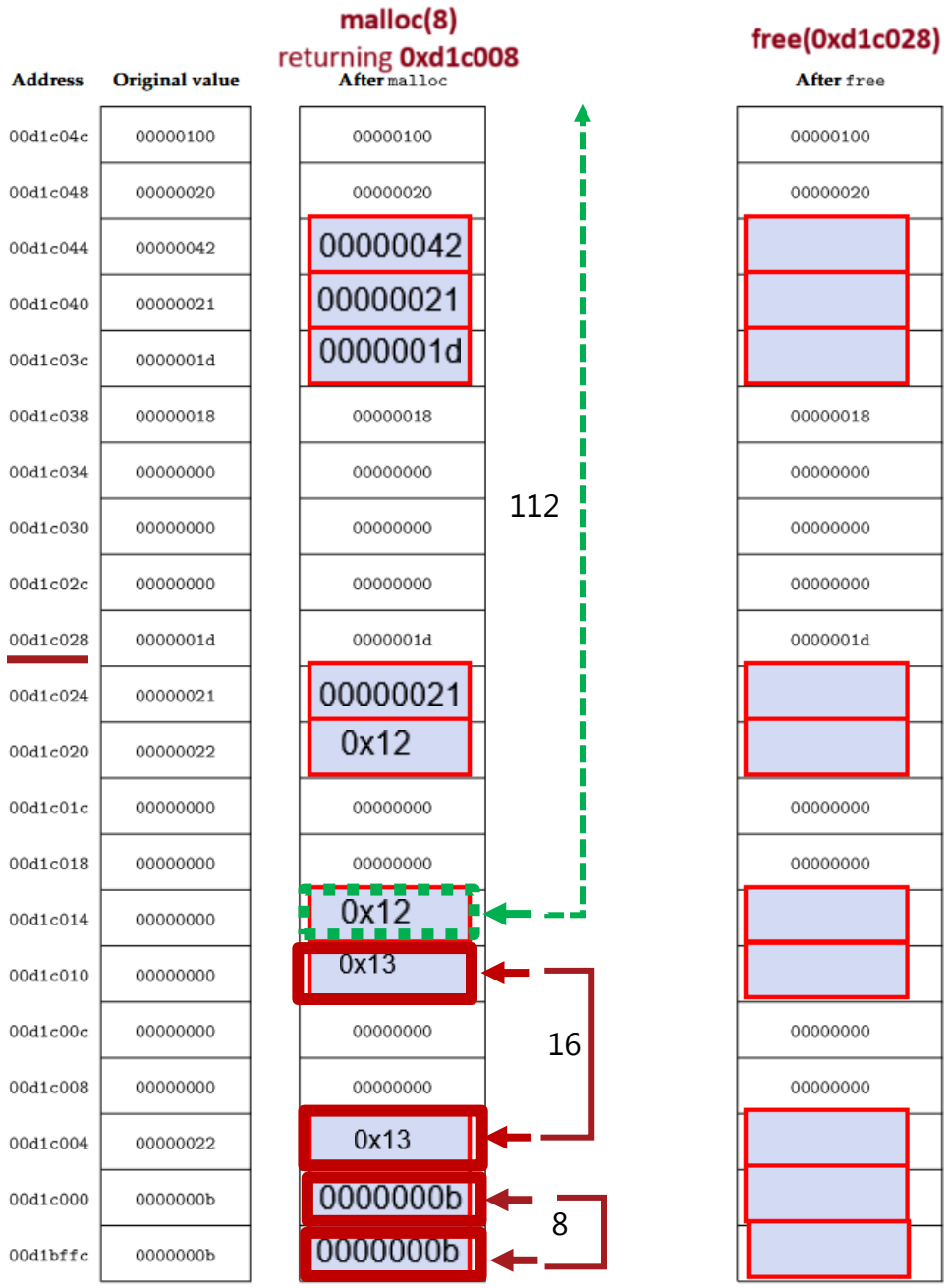
  - 00010010 = **0x12**

- The other header and footers don't require updating, so we can copy over the remaining values from the heap

- Now we move onto the free() instruction

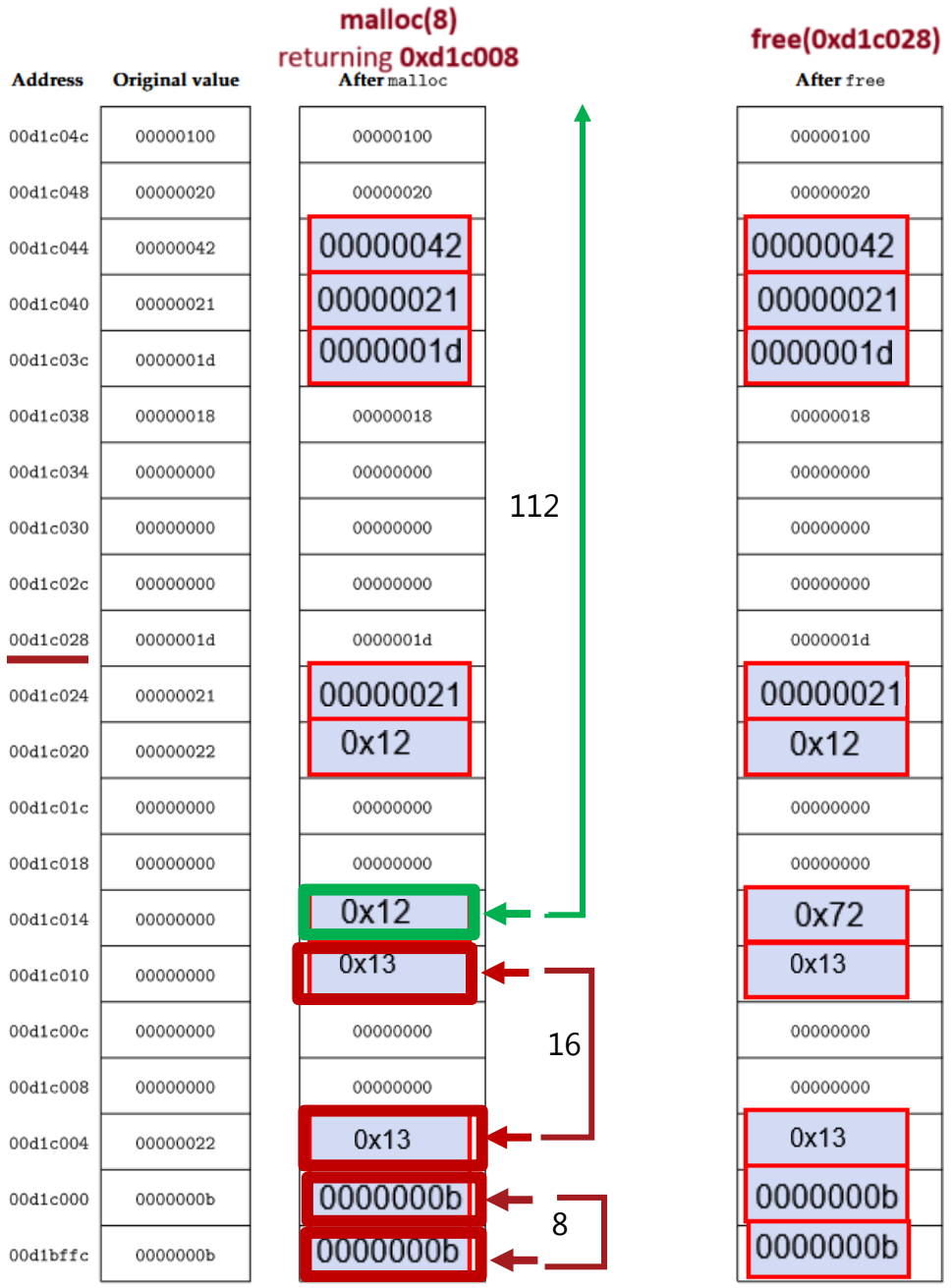


- The block to free is located between **2** other free blocks
- Remember immediate coalescing



- The block to free is located between **2** other free blocks
- Remember immediate coalescing
- Size = 64 + 32 + 16 = **112-bytes**
- Bit 0 – Free
- Bit 1 – Previous Allocated

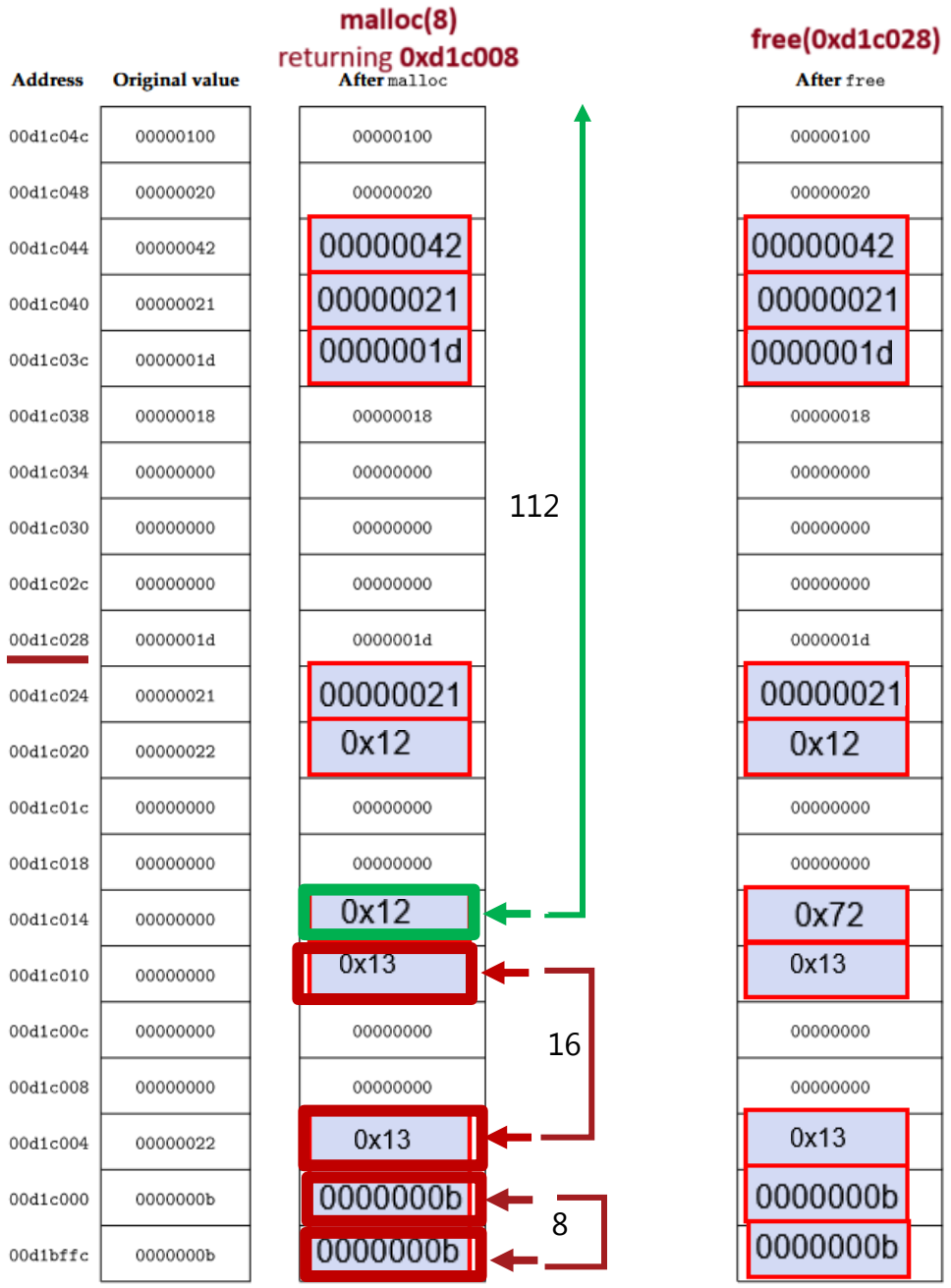
- 01110010 = **0x72**



- The block to free is located between **2** other free blocks
- Remember immediate coalescing
- Size = 64 + 32 + 16 = **112-bytes**
- Bit 0 – Free
- Bit 1 – Previous Allocated

- 01110010 = **0x72**

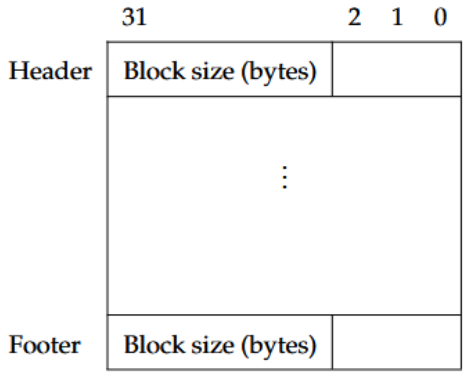
- Again the other headers and footers remain unchanged, and all values within the new free block have become garbage data and can be copied over



Address	Original value	After malloc	After free
00d1c04c	00000100	00000100	00000100
00d1c048	00000020	00000020	00000020
00d1c044	00000042	00000042	00000042
00d1c040	00000021	00000021	00000021
00d1c03c	0000001d	0000001d	0000001d
00d1c038	00000018	00000018	00000018
00d1c034	00000000	00000000	00000000
00d1c030	00000000	00000000	00000000
00d1c02c	00000000	00000000	00000000
00d1c028	0000001d	0000001d	0000001d
00d1c024	00000021	00000021	00000021
00d1c020	00000022	00000012	00000012
00d1c01c	00000000	00000000	00000000
00d1c018	00000000	00000000	00000000
00d1c014	00000000	00000012	00000072
00d1c010	00000000	00000013	00000013
00d1c00c	00000000	00000000	00000000
00d1c008	00000000	00000000	00000000
00d1c004	00000022	00000013	00000013
00d1c000	0000000b	0000000b	0000000b
00d1bffc	0000000b	0000000b	0000000b

Question Overview – Exam 21/22

Question 2.3.1: Consider an allocator that uses an implicit free list and immediate coalescing of neighbouring free blocks. The layout of each allocated and free memory block is as follows, with one 32-bit word per row:



Each memory block, either allocated or free, has a size that is a multiple of eight bytes, rounding up allocations if necessary. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
- `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- `bit 2` is unused and is always set to be 0.

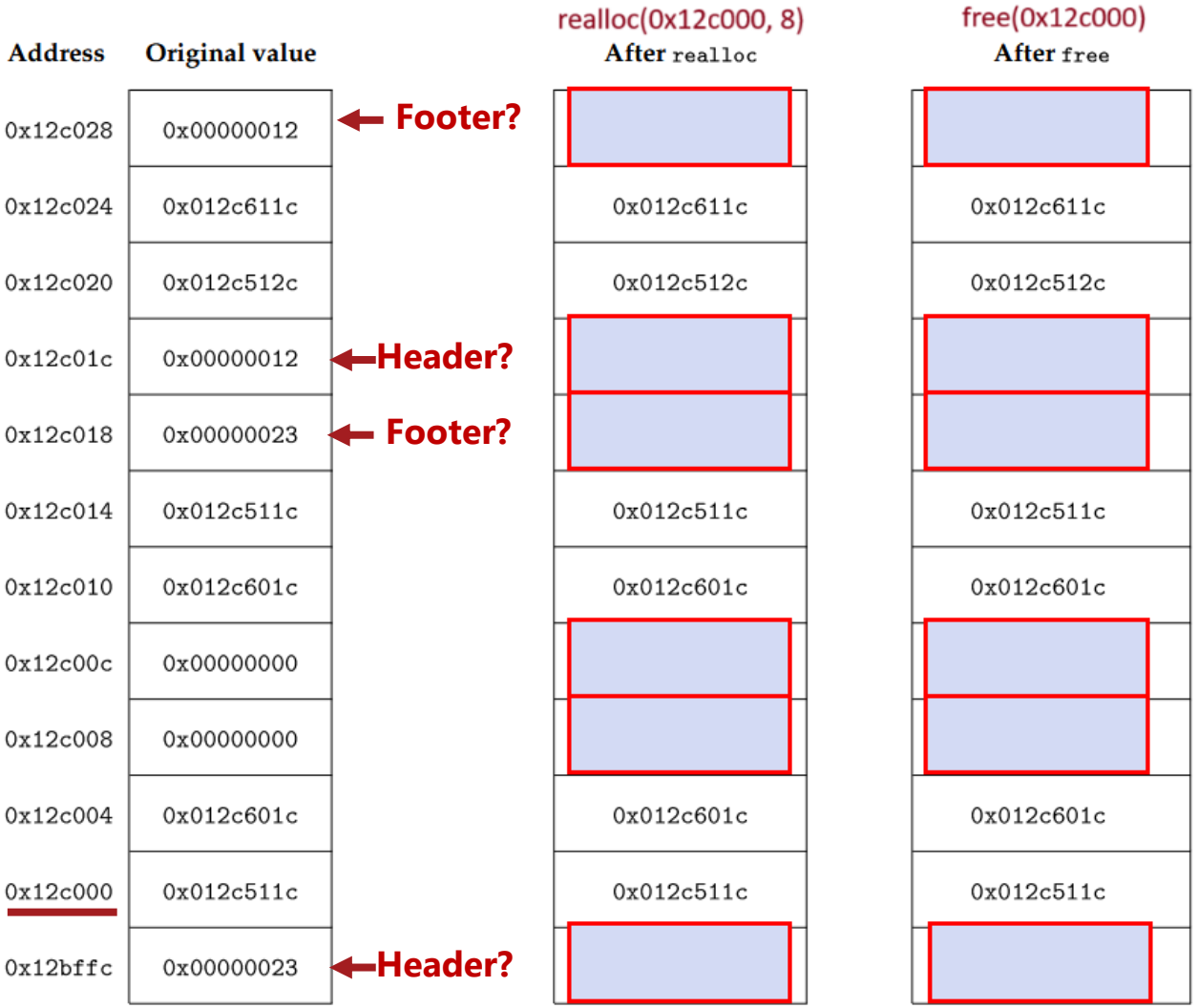
Important: We must *never* create blocks with zero payload (i.e. we must *never* create blocks with size 8).

Given the heap shown on the left, show the new heap contents after *consecutive* calls to

1. `realloc(0x12c000, 8)`. Assume the the return value is `0x12c000`, and that the existing allocation is resized to be as small as possible.
2. `free(0x12c000)`.

Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

- Again begin by locating the instructed address and identify all blocks in the heap

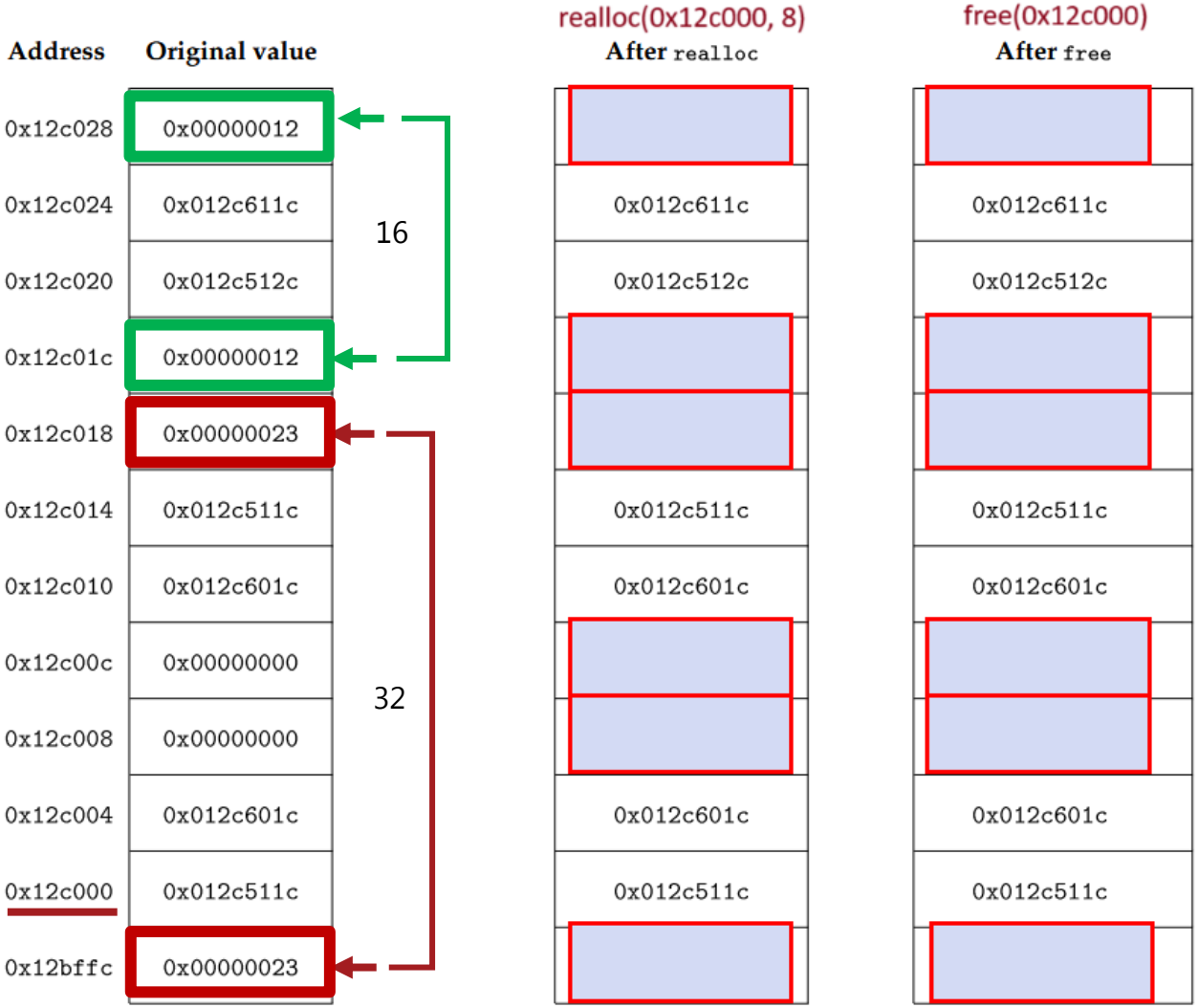


- Lets verify:

0x23

0010 0011

- Allocated
- Previous Allocated
- Size = 00100000 = 0x20 = **32**



- Lets verify:

0x23

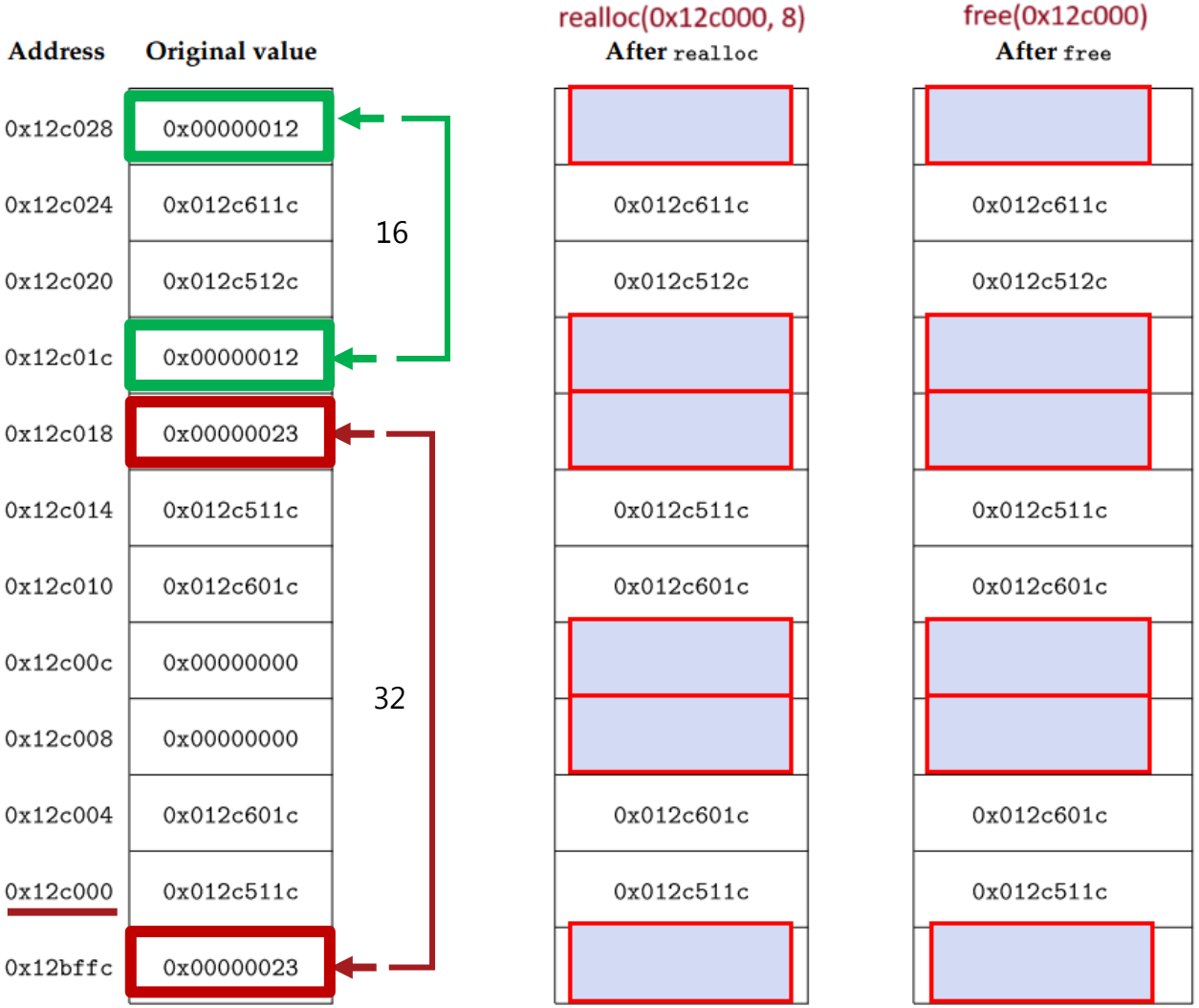
0010 0011

- Allocated
- Previous Allocated
- Size = 00100000 = 0x20 = **32**

0x12

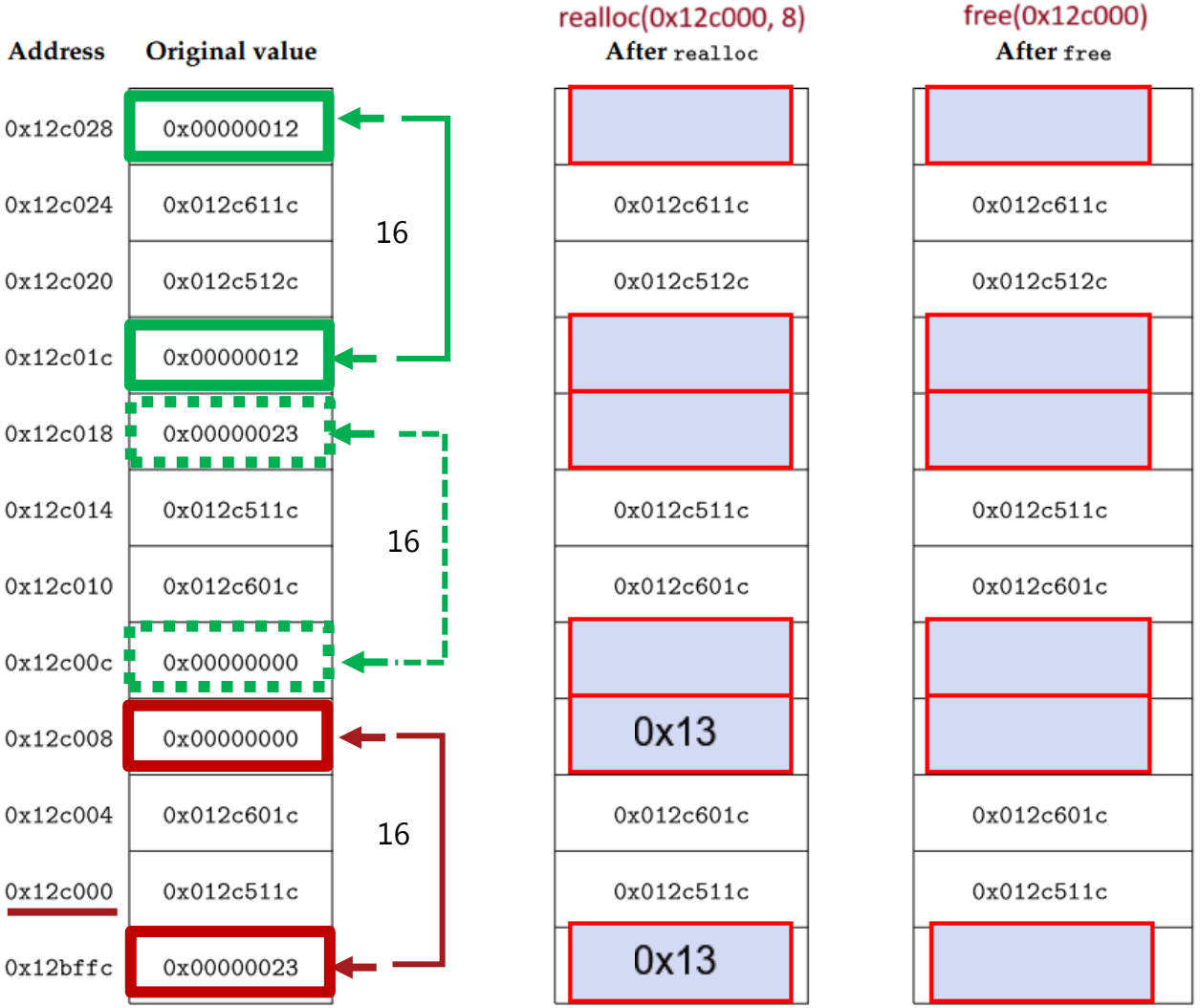
0001 0010

- Free
- Previous Allocated
- Size = 00010000 = 0x10 = **16**

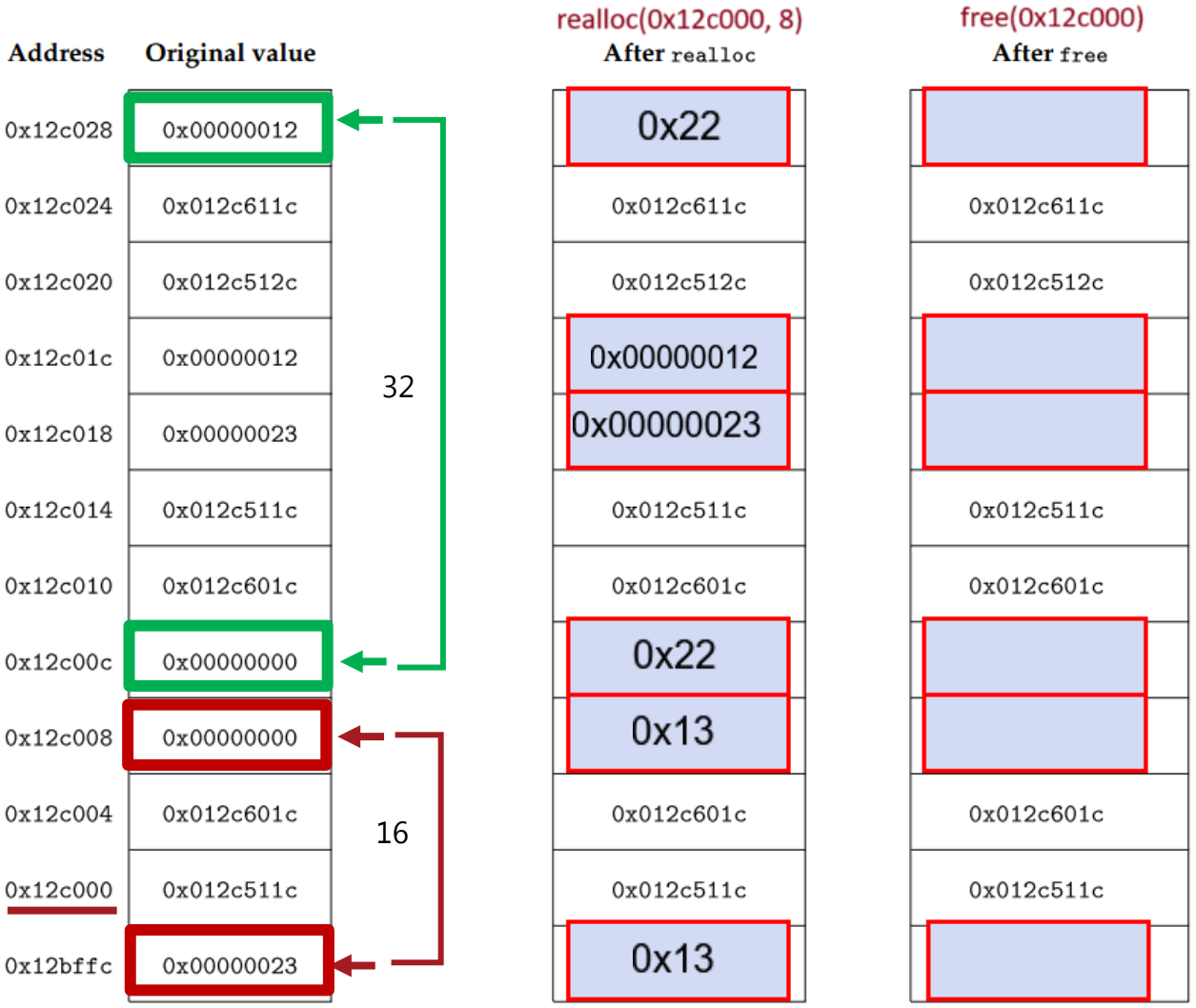


- Now we want to realloc the **32-byte** block to size **8** + footer and header = **16-bytes**
  - Bit 0 – Allocated
  - Bit 1 – Previous Allocated
  - Size = 16

  - 00010011 = **0x13**

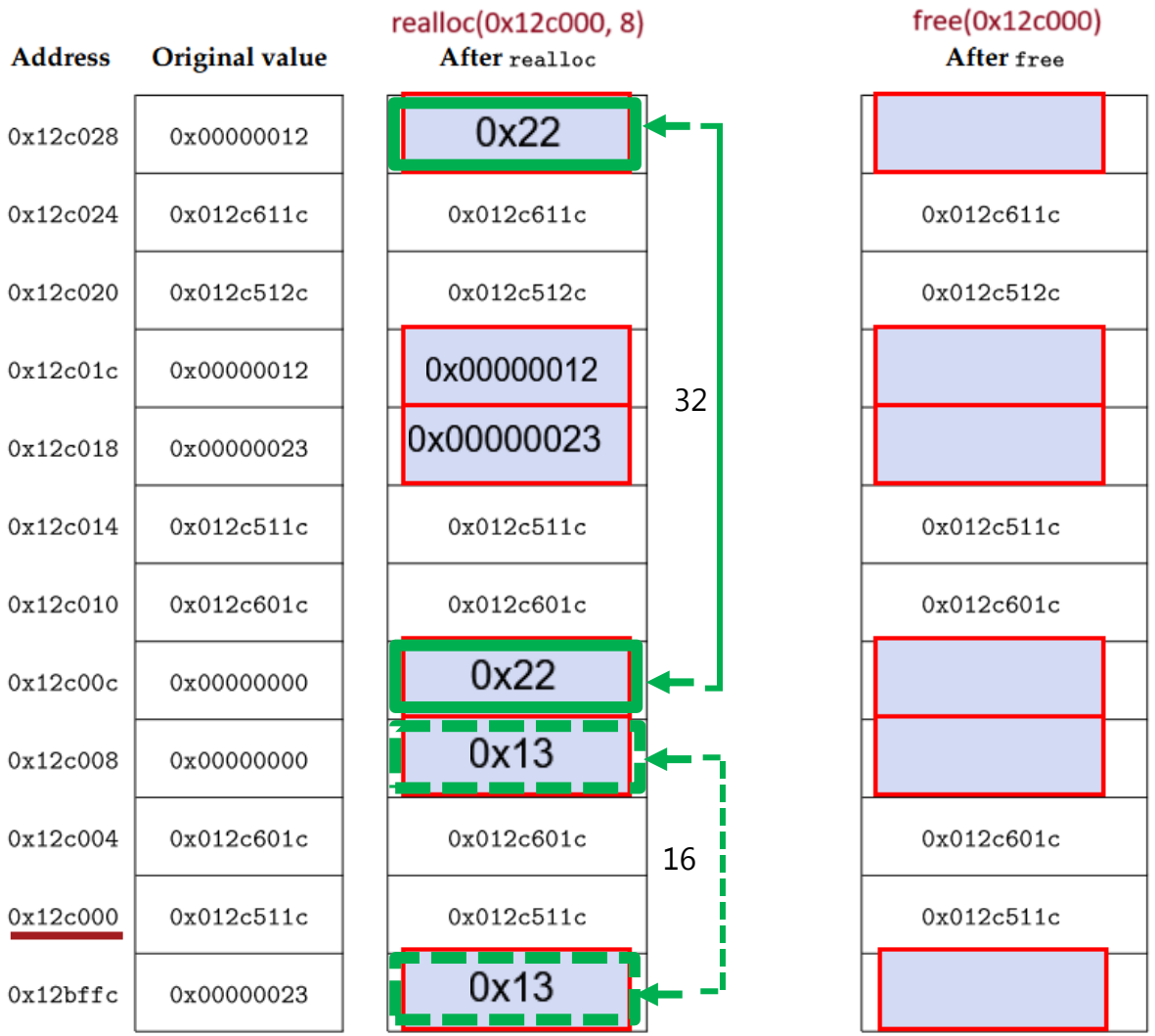


- Now we want to realloc the **32-byte** block to size **8** + footer and header = **16-bytes**
  - Bit 0 – Allocated
  - Bit 1 – Previous Allocated
  - Size = 16
  - 00010011 = **0x13**
- Due to immediate coalescing both **16-bytes** free blocks are merged:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16 + 16 = 32
  - 00100010 = **0x22**



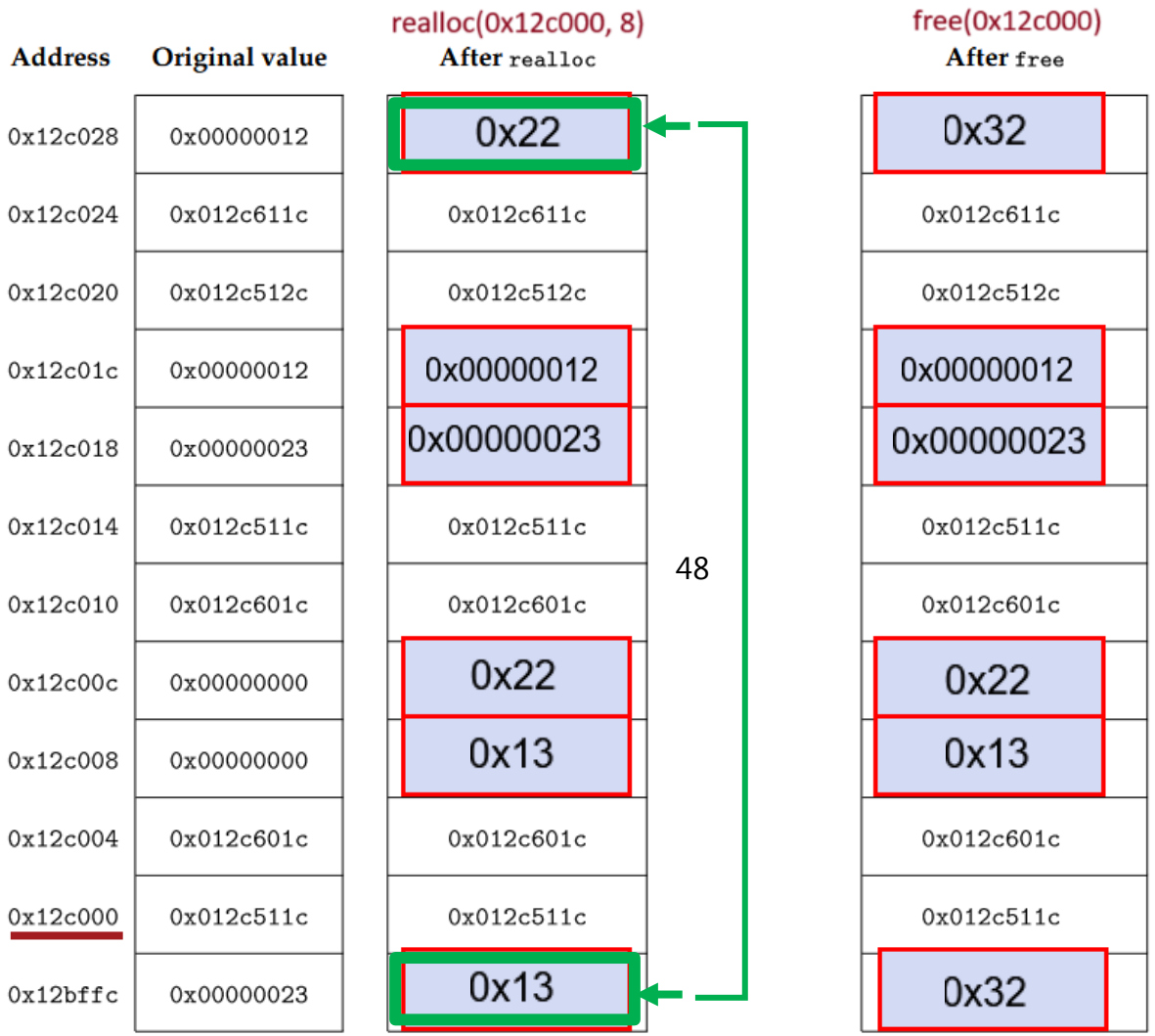
- The values in-between can just be copied over
- Next step – Free:

- We now free the only visible **16-byte** allocated block:



- We now free the only visible **16-byte** allocated block:
- Remember Immediate Coalescing, so:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16 + 32 = 48

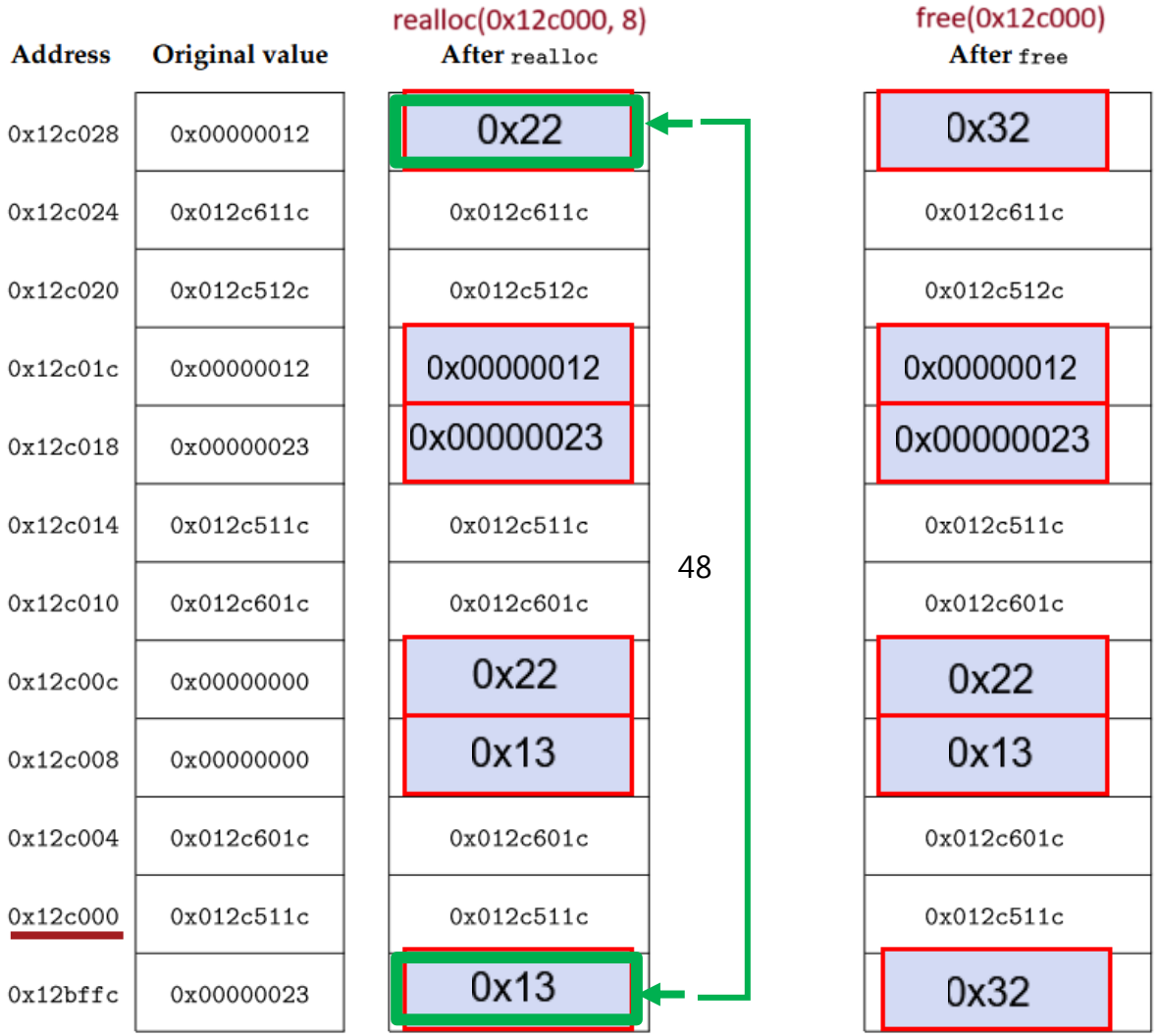
  - 00110010 = **0x32**



- We now free the only visible **16-byte** allocated block:
- Remember Immediate Coalescing, so:
  - Bit 0 – Free
  - Bit 1 – Previous Allocated
  - Size = 16 + 32 = 48

  - 00110010 = **0x32**

- Again values in-between header and footer can just get copied over



Address	Original value	After realloc	After free
0x12c028	0x00000012	0x22	0x32
0x12c024	0x012c611c	0x012c611c	0x012c611c
0x12c020	0x012c512c	0x012c512c	0x012c512c
0x12c01c	0x00000012	0x12	0x12
0x12c018	0x00000023	0x23	0x23
0x12c014	0x012c511c	0x012c511c	0x012c511c
0x12c010	0x012c601c	0x012c601c	0x012c601c
0x12c00c	0x00000000	0x22	0x22
0x12c008	0x00000000	0x13	0x13
0x12c004	0x012c601c	0x012c601c	0x012c601c
0x12c000	0x012c511c	0x012c511c	0x012c511c
0x12bffc	0x00000023	0x13	0x32

Little bonus question

- **Would it be possible for a free block to start at address 0x12c02C?**
  - No, because we use immediate coalescing, meaning free blocks are never adjacent
  - And we know the block previous to 0x12c02c is free