x86 Assembly Crash Course

Introduction

The assembly language is the lowest level of human-readable language. It is also the highest level of language into which a binary can be reliably decompiled. When learning malware reverse engineering, knowing the basics of assembly language is essential. This is because when we get a malware sample to analyze, it is most likely a compiled binary. We cannot view this binary's C/C++ or other language code because that is not available to us. What we can do, however, is to decompile the code using a decompiler or a disassembler. The problem with decompiling is that a lot of information in the written code is removed while it is compiled into a binary; hence we won't see variable names, function names, etc., as we do while writing code. So the most reliable code we have for a compiled binary is its assembly code. In this room, we will learn the basics of assembly that we can use in the malware analysis rooms to understand what a binary is doing while looking at its assembly code.

Learning Objectives

We will be covering the following topics:

- Opcodes and operands
- General assembly instructions
- Arithmetic and logical instructions
- Conditionals
- Branching instructions

Prerequisites

Before starting this room, it is highly recommended that you complete the x86 Architecture Overview room first.

Opcodes and Operands

The code for a program, as written on the disk and understood by the CPU, is in binary format. This means that the actual code is a sequence of 1s and 0s. To make it understandable, we often club a series of 8 bits into hex digits, called a byte. For example, if we have the binary number 10100101, the first 4 bits can be converted to A in hex and the next 4 into 5 in hex, making the binary number A5 in hex, also written as 0xA5 to denote that it is a hex number. So the instructions that a computer is executing will be just a sequence of random numbers in hex to a human. Among these random numbers are opcodes and operands. Opcodes denote the hex for actual operations, and operands are the registers or memory locations on which the operations are performed.

Opcodes

Opcodes are numbers that correspond to instructions performed by the CPU. When we use a disassembler (we will learn about disassemblers in the upcoming rooms) to disassemble a program, it reads the opcodes. It translates them into assembly instructions to make them human-readable. For example, the instruction for moving 0x5F to the eax register is:

- mov eax, 0x5f

When looking at it in a disassembler, we will see:

- 040000:   b8 5f 00 00 00   mov eax, 0x5f

Here, the 040000: corresponds to the address where the instruction is located. b8 refers to the opcode of the instruction mov eax, and 5F 00 00 00 indicates the other operand 0x5f. Please note that due to endianness, the operand 0x5f is written as 5f 00 00 00, which is actually 00 00 00 5f but in little-endian notation. Similarly, there is an opcode for each instruction in the assembly language. There are references for converting opcodes into assembly instructions. Still, unless we are writing a disassembler, we will not need them, as a disassembler does that work pretty well. However, it is good to understand what is happening under the hood for a better picture overall.

We saw that in the above operation, we had three parts, an instruction, mov, and two operands, eax and 0x5f. In this particular instruction, the value 0x5f is being moved into eax; however, we can also have other kinds of operands in the assembly language.

Types of Operands

In general, there are three types of operands in the assembly language.

- Immediate Operands can also be considered constants. These are fixed values like we had the 0x5f in the above example.
- Registers can also be operands. The above example shows eax as a register where the immediate operand is stored.
- Memory operands are denoted by square brackets, and they reference memory locations. For example, if we see [eax] as an operand, it will mean that the value in eax is the memory location on which the operation has to be performed.

Now that we have learned about the operands and opcodes, we will learn about common assembly instructions in the next task.

****

Answer the questions below:

What are the hex codes that denote the assembly operations called?
Answer: Opcodes

Which type of operand is denoted by square brackets?
Answer: Memory Operand

General Instructions

Instructions tell the CPU what operation to perform. Instructions often use operands from registers, memory, or immediate operands to perform operations and then store the results in either registers or memory. In this task, we will learn the most common instructions that we might come across while reverse engineering malware.

General Instructions

These instructions perform simple operations, such as moving a value from one type of storage to another.

The MOV Instruction

The mov instruction moves a value from one location to another. Its syntax is as follows:

- mov destination, source

The mov instruction can move a fixed value to a register, a register to another register, and a value in a memory location to a register. The following examples will help explain.

The following instruction copies a fixed value to a register. In this particular instruction, 0x5f is being moved to eax:

- mov eax, 0x5f

In this particular case, the value stored in eax is being moved to ebx:

- mov ebx, eax

The following instruction copies the value stored in a memory location to a register:

- mov eax, [0x5fc53e]

As seen above, we use square brackets when referencing memory. Similarly, suppose we see a register in square brackets. In that case, that will mean that the value in that register will be treated as a memory location, and the value in that memory location will be moved to the destination. This means that the example mov eax, [0x5fc3e] and the below example will have the same result.

- mov ebx, 0x5fc53e
- mov eax, [ebx]

We can use the mov instruction to perform arithmetic calculations when referencing memory addresses. For example, the below instruction calculates ebp+4 (adding an offset of 4 bytes to the memory location) and moves the value in the resulting memory address into eax:

- mov eax, [ebp+4]

The LEA Instruction

The lea instruction stands for "load effective address." The format of this instruction is as follows:

- lea destination, source

While the mov instruction moves the data at the source memory address to the destination, the lea instruction moves the address of the source into the destination. In the example below, the ebp value will be increased by four and moved to eax. However, if we had used a mov instruction here instead of lea, it would have moved the value in the memory location ebp+4.

- lea eax, [ebp+4]

Here, we can notice that we have performed an arithmetic operation on a register and saved the result in another register using a single instruction. The lea instruction is often used by compilers not for referencing memory locations but so that an arithmetic operation is performed on a register and saved to another using a single instruction. This is true, especially if the arithmetic operations are more complex, like adding and multiplying in a single instruction. As we will see in the next task, using arithmetic operations for this operation will need several instructions.

The NOP Instruction

The nop instruction stands for no operation. This instruction exchanges the value in eax with itself, resulting in no meaningful operation. Hence, the execution moves to the next instruction without changing anything. The nop instructions are used for consuming CPU cycles while waiting for an operation or other such purposes. It has the following syntax:

- nop

The nop instruction is used by malware authors when redirecting execution to their shellcode. The exact location where the execution will redirect is often unknown, so the malware author uses a bunch of nop instructions to ensure that the shellcode execution does not start from the middle. This padding of nop instructions is called a nop sled.

Shift Instructions

The CPU uses shift instructions to shift each register bit to the adjacent bit. There are two shift instructions for shifting either to the right or left. The shift instructions have the following syntax:

- shr destination, count
- shl destination, count

Here the shr instruction is for the shift right operation, and the shl is for the shift left operation. This instruction shifts the bits in the destination operand. The count operand decides the number of bits to be shifted. The bits which are shifted out of their location are filled with zeroes. So, if we have 00000010 in eax and shift it left, it will become 00000100.

The carry flag (CF) is used to augment the destination, as it is filled by the last bit overflowing the destination. For example, if we have 00000101 in eax and shift it right by 1 bit, the result will have 00000010 in eax, and the carry flag will be set, i.e. it will have a value of 1.

Shift instructions are used instead of multiplication and division by two or powers of two (2n where n is the count in the shift instruction). This saves execution time by not having to manipulate values in registers before performing multiplication or division. For example, If eax has 00000010, and we shift right by 1 bit, we get 00000001, which is the same result as dividing eax by 2. Similarly, if eax has 00000001, and we shift left by 1 bit, the result is 00000010, the same as multiplying eax by 2.

Rotate Instructions

The rotate instructions are similar to the shift instructions. The only difference is that the bits are rotated back to the other end of the register instead of moving the overflowing bit into the carry flag or adding zeros instead of shifted-out bits. The rotate instruction has the following syntax:

- ror destination, count
- rol destination, count

Here, the ror instruction rotates the destination to the right, and rol rotates the destination to the left. The rest of the syntax remains the same. As an example, if we have 10101010 in al, and we rotate it right by 1 bit, it will result in 01010101. Similarly, rotating this result to the left by 1 bit will result in 10101010 again.

****

Answer the questions below:

In mov eax, ebx, which register is the destination operand?

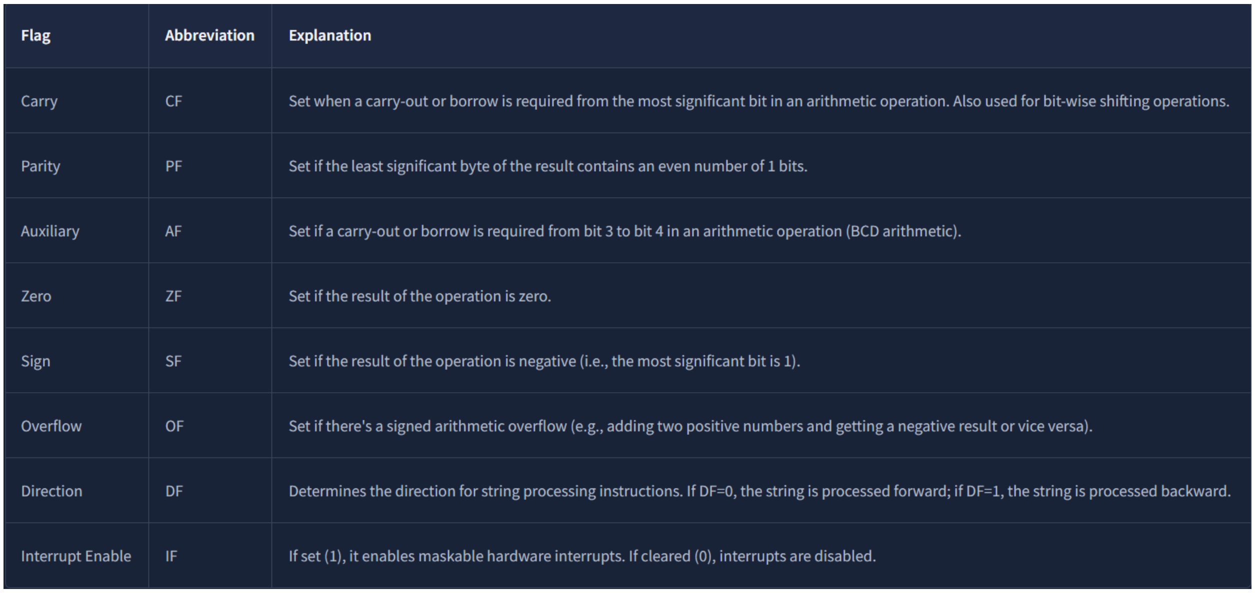
Answer: eax

What instruction performs no action?

Answer: nop

Flags

In x86 assembly language, the CPU has several flags that indicate the outcome of certain operations or conditions. These flags are bits in a special register known as the flags register or EFLAGS register. Each flag represents a specific condition or result of the most recent arithmetic or logical operation. Here’s a table with the most common flags in x86 assembly and their explanations:



Flag	Abbreviation	Explanation
Carry	CF	Set when a carry-out or borrow is required from the most significant bit in an arithmetic operation. Also used for bit-wise shifting operations.
Parity	PF	Set if the least significant byte of the result contains an even number of 1 bits.
Auxiliary	AF	Set if a carry-out or borrow is required from bit 3 to bit 4 in an arithmetic operation (BCD arithmetic).
Zero	ZF	Set if the result of the operation is zero.
Sign	SF	Set if the result of the operation is negative (i.e., the most significant bit is 1).
Overflow	OF	Set if there's a signed arithmetic overflow (e.g., adding two positive numbers and getting a negative result or vice versa).
Direction	DF	Determines the direction for string processing instructions. If DF=0, the string is processed forward; if DF=1, the string is processed backward.
Interrupt Enable	IF	If set (1), it enables maskable hardware interrupts. If cleared (0), interrupts are disabled.

Flags can be used in conditional jumps and are crucial for implementing conditional branching in assembly code. For example, you might only jump to a specific address if a certain flag is set or cleared.

****

Answer the questions below:

Which flag will be set if the result of the operation is zero? (Answer in abbreviation)

Answer: ZF

Which flag will be set if the result of the operation is negative? (Answer in abbreviation)

Answer: SF

Arithmetic and Logical Instructions

Arithmetic Operations

Arithmetic Operations are performed by a CPU using arithmetic instructions. In this task, we will go through these instructions.

Addition and Subtraction Instructions
The syntax for the addition instruction is as follows. The value is added to the destination, and the result is stored in the destination.

- add destination, value

The subtraction instruction follows a similar syntax. In the below syntax, the value is subtracted from the destination, and the result is stored in the destination.

- sub destination, value

In the above examples, the value can be either a fixed value constant or a register. For the subtraction operation, Zero Flag (ZF) is set if the result of the subtraction is zero. If the destination is smaller than the subtracted value, then the Carry Flag (CF) is set.

Multiplication and Division Instructions
The multiplication and division operations use the eax and the edx registers. Therefore, we will have to look at the last instruction that manipulated these registers for every multiply and divide operation.

The multiply instruction has the following format. It multiplies the value with eax and stores the result in edx:eax as a 64-bit value. Two registers are required here because the result of multiplying 2 32-bit values can often be higher than 32 bits. The lower 32 bits of the result are stored in the eax register, and the higher 32 bits are stored in the edx register.

- mul value

The value can be another register or a constant as an immediate operand.

For the division instruction, the case is the opposite. It divides the 64-bit value in edx:eax and saves the result in eax and the remainder in edx.

- div value

Increment and Decrement Instructions
As the name suggests, the increment and decrement instructions increment or decrement the operand register by 1. The syntax to increment eax by 1 is as follows:

- inc eax

Similarly, the syntax to decrement eax by 1 using the decrement instruction is as follows:

- dec eax

Logical Instructions

Logical instructions are used to perform logical operations. Let's go through a few common logical operations performed by the CPU.

AND Instruction

The AND instruction performs a bitwise AND operation on the operands. An AND operation returns an output of 1 when both the inputs are 1; otherwise, it returns 0. An example instruction is below:

- and al, 0x7c

In this example, 0x7c converts to 01111100 in binary. Suppose al had a value of 0xfc, which is 11111100 in binary. In this case, the output of the above instruction will be 01111100. However, if al has a value of 0x8c, 10001100 in binary, the result of the above instruction will be 00001100 or 0xc.

OR Instruction

The OR instruction performs a bitwise OR operation. An OR operation returns 1 if at least one of the operands is 1. It returns 0 if none of the operands is 1. An example instruction is below:

- or al, 0x7c

In this example, if al had a value of 0xfc or 11111100 in binary, the output of the above instruction will be 11111100. Similarly, if al has a value of 0x8c or 10001100 in binary, the result will still be 11111100 in binary or 0xfc.

NOT Instruction

The NOT instruction takes one operand. It simply inverts the operand bits, replacing 1s with 0s and vice versa. In the following example, if al had a value of 11110000, it would result in 00001111.

- not al

XOR Instruction

The XOR operation returns 1 if both the inputs are opposite. It returns 0 when both inputs are the same. This operation is performed by the XOR instruction in assembly language, which performs a bitwise XOR operation on the operands. It has the following syntax.

- xor al, 0x7c

If al has a value of 0xfc, which is 11111100, the result of this instruction will be 10000000 or 0x80. Similarly, if al has a value of 0x8c, which is 10001100, the result of this instruction will be 11110000 or 0xf0. If al has a value of 0x7c, the result will be 0x00. This shows that XORing a register with itself results in 0. Therefore, the XOR instruction is often used to zero a register, which is more optimized than a MOV instruction.

****

Answer the questions below:

In a subtraction operation, which flag is set if the destination is smaller than the subtracted value?
Answer: Carry Flag

Which instruction is used to increase the value of a register
Answer: inc

Do the following instructions have the same result? (yea/nay)

- **xor eax, eax**
- **mov eax, 0**

Answer: yea

Conditionals and Branching

Conditionals

A CPU often must determine if two values are equal to, greater than, or less than each other. To perform such operations, the CPU uses some conditional instructions. This task will discuss conditional instructions in the x86 assembly language.

The TEST Instruction

The test instruction performs a bitwise AND operation, and instead of storing the result in the destination operand as the AND instruction does, it sets the Zero Flag (ZF) if the result is 0. This instruction is often used to check if an operand has a NULL value, for example, by testing the operand against itself. This is done because it takes fewer bytes to use the test instruction than by comparing to 0. The test instruction has the following syntax:

- test destination, source

The CMP Instruction

Based on the result, the CMP instruction compares the two operands and sets the Zero Flag (ZF) or the Carry Flag (CF). It has the following syntax:

- cmp destination, source

The compare instruction works similarly to a subtract instruction. The only difference is that the operands are not modified. The Zero Flag (ZF) is set if both operands are equal. If the source operand is greater than the destination operand, the Carry Flag (CF) is set. The ZF and the CF are cleared if the destination operand is greater than the source operand.

Branching

When there is no branching, the Instruction Pointer goes from one instruction to the other in the order they are placed in memory. The control flow remains in a straight line unless there is a branching operation. Branching operations change the value of the Instruction Pointer and change the program's control flow from linear to branching out.

The JMP Instruction

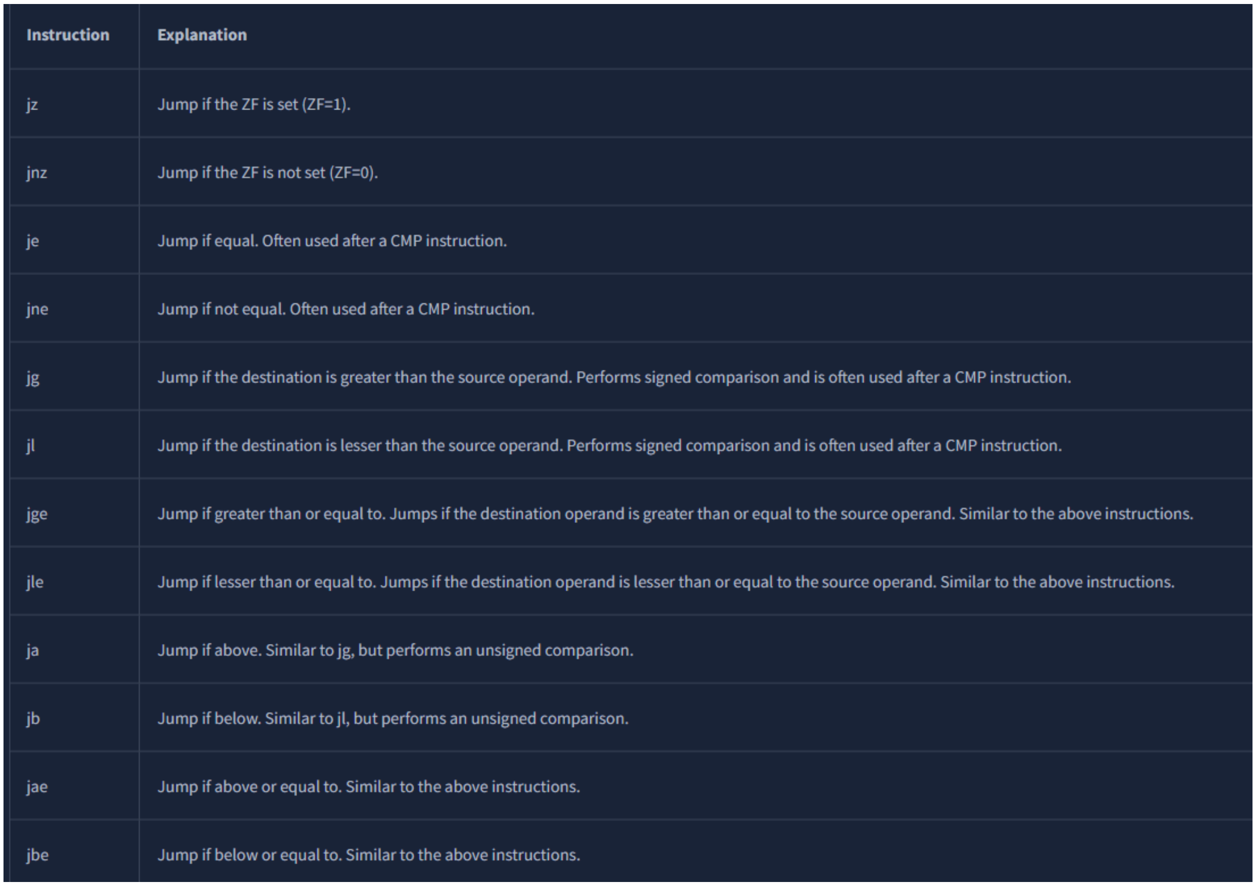
The JMP instruction makes the control flow jump to a specified location. It has the following syntax:

- jmp location

Here, the location operand will be moved to the Instruction Pointer, making it the address where the next instruction will be fetched for execution.

Conditional Jumps

Often, the code is required to move if a specific condition is met. In higher-level languages, there are if conditions that help fulfil this requirement. However, there is no if statement in the assembly language. This requirement is fulfilled using conditional jumps. Conditional jumps decide whether to jump based on the value of the Flag Registers. Their syntax is similar to the jump instruction. The following table shows some of the common conditional jumps.



Instruction	Explanation
jz	Jump if the ZF is set (ZF=1).
jnz	Jump if the ZF is not set (ZF=0).
je	Jump if equal. Often used after a CMP instruction.
jne	Jump if not equal. Often used after a CMP instruction.
jg	Jump if the destination is greater than the source operand. Performs signed comparison and is often used after a CMP instruction.
jl	Jump if the destination is lesser than the source operand. Performs signed comparison and is often used after a CMP instruction.
jge	Jump if greater than or equal to. Jumps if the destination operand is greater than or equal to the source operand. Similar to the above instructions.
jle	Jump if lesser than or equal to. Jumps if the destination operand is lesser than or equal to the source operand. Similar to the above instructions.
ja	Jump if above. Similar to jg, but performs an unsigned comparison.
jb	Jump if below. Similar to jl, but performs an unsigned comparison.
jae	Jump if above or equal to. Similar to the above instructions.
jbe	Jump if below or equal to. Similar to the above instructions.

****

Answer the questions below:

Which flag is set as a result of the test instruction being zero?
Answer: Zero Flag

Which of the below operations uses subtraction to test two values? 1 or 2?
1. cmp eax, ebx
2. test eax, ebx
Answer: 1

Which flag is used to identify whether a jump will be taken or not after a jz or jnz instruction?
Answer: Zero Flag

Stack and Function Calls

The Stack

In the x86 Architecture Overview room, we learned about the stack and its significance. We also learned about some of the registers used to reference the location of the stack in the memory. The stack is a Last In, First Out (LIFO) memory. This means the last variable pushed onto the stack is the first to pop out. These push and pop operations are performed by following instructions in the assembly language.

The PUSH Instruction

The push instruction has the following syntax:

- push source

As mentioned earlier, the push instruction will push the source operand onto the stack. The value of the operand is stored at the memory location pointed to by the stack pointer (ESP), effectively becoming the new top of the stack. The stack pointer is then adjusted (decremented) to reflect the updated top position of the stack. The following instructions also push all the general-purpose registers to the stack.

- pusha (push all words): Pushes all the 16-bit general purpose registers to the stack, i.e. AX, BX, CX, DX, SI, DI, SP, BP
- pushad (push all double words): Pushes all the 32-bit general purpose registers to the stack, i.e. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

When we encounter these instructions, it is often a sign of someone manually injecting assembly instructions to save the state of registers, as is often the case with shellcode.

The POP Instruction

The pop instruction has the following syntax:

- pop destination

The pop instruction retrieves the value from the top of the stack and stores it in the destination operand. As a result, the stack pointer (ESP) is incremented to reflect the adjustment made after popping the value. The following instructions also pop all the general-purpose registers from the stack.

- popa (pop all words): Pops the values sequentially from the top of the stack to general-purpose registers in the following order: DI, SI, BP, BX, DX, CX, AX. The SP or ESP is adjusted to reflect the new top position of the stack.
- popad (pop all double words): Pops the values sequentially from the top of the stack to general-purpose registers in the following order: EDI, ESI, EBP, EBX, EDX, ECX, EAX. The SP or ESP is adjusted to reflect the new top position of the stack.

The CALL Instruction

The call instruction is used in the assembly language for performing a function call operation to perform a specific task. It has the following syntax:

- call location

Depending on the calling convention, the arguments are placed on the stack or in the registers in a function call. The function prologue prepares the stack by adjusting the EBP and ESP and pushing the return address on the stack. Similarly, when the function returns, the epilogue restores the stack for the caller function. We will learn more about calling conventions, prologue, and epilogue in the upcoming rooms.

In the next task, we will practice using assembly instructions.

****

Answer the questions below:

Which instruction is used for performing a function call?

Answer: call

Which instruction is used to push all registers to the stack?
Answer: pusha

Practice Time

So far, we have learned about some common assembly instructions. It's time to practice those instructions using our Assembly Emulator.
Click on the View Site button on the top-right side of this task. It will open a lab in the split screen. For more convenience using full screen, you can click here to open it in a new tab. Please take the guided tour for an overview. After that, follow and run the instructions to observe the stack, memory, registers, and flags.

The options on the top bar can be used to:

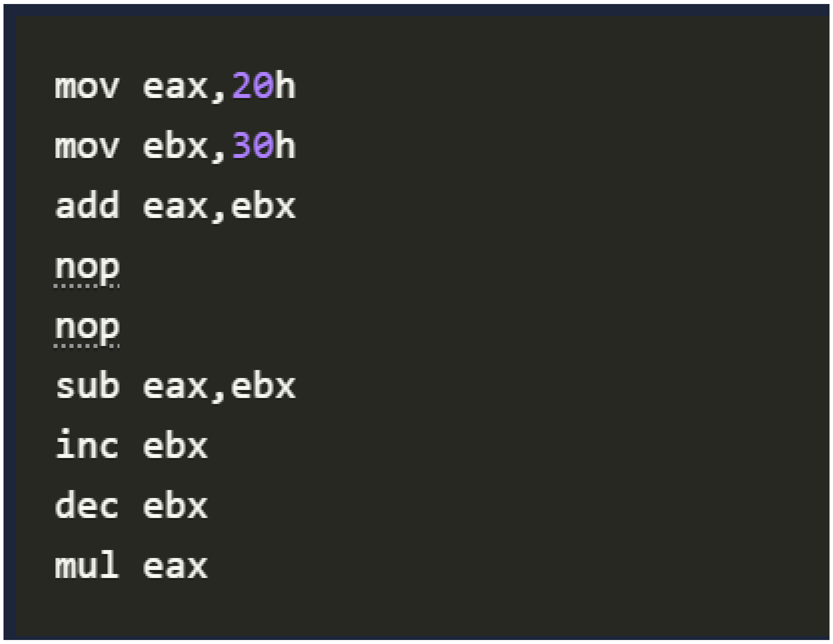
- Run the program
- Execute the next instruction
- Stop the execution
- Restart the program

Assembly Code

The lab contains the following assembly code instructions. Run these instructions and observe how these instructions impact registers, flags, memory, and the stack.

Arithmetic Code

The following instructions demonstrate how different arithmetic instructions work:



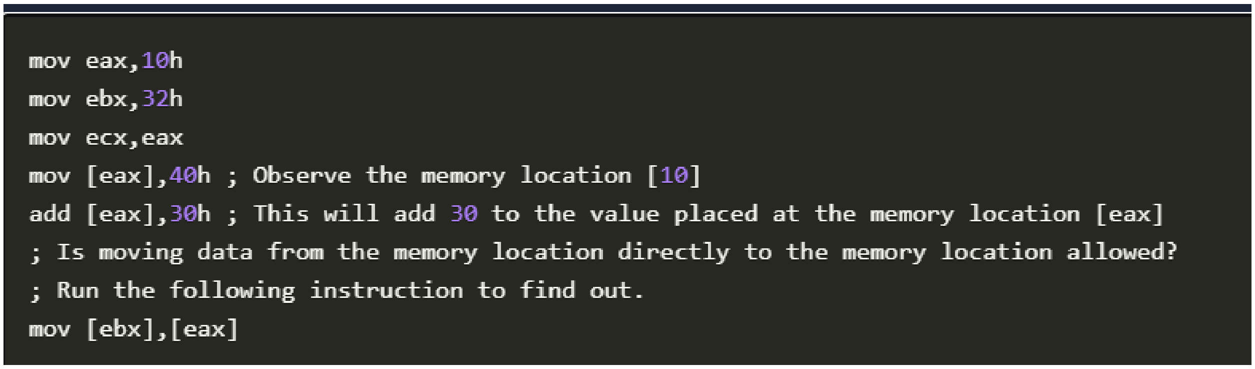
```
mov eax,20h
mov ebx,30h
add eax,ebx
nop
nop
sub eax,ebx
inc ebx
dec ebx
mul eax
```

MOV Instructions

The following instructions demonstrate how the data can be moved:

- Into registers
- From register to register
- From memory to register
- From register to memory

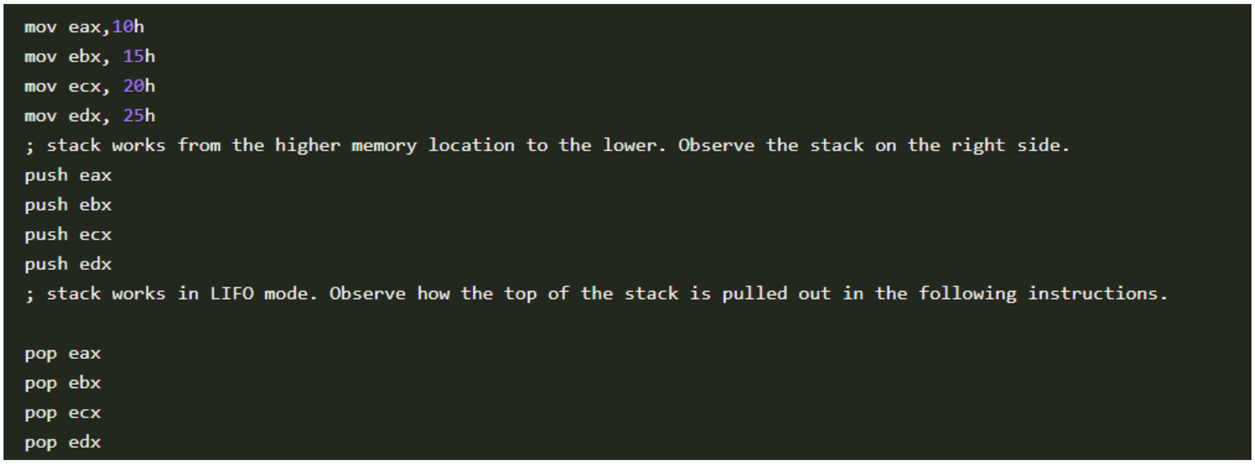
Choose the Mov Instruction code from the emulator's dropdown, run each instruction, and observe the registers and memory.



```
mov eax,10h
mov ebx,32h
mov ecx,eax
mov [eax],40h ; Observe the memory location [10]
add [eax],30h ; This will add 30 to the value placed at the memory location [eax]
; Is moving data from the memory location directly to the memory location allowed?
; Run the following instruction to find out.
mov [ebx],[eax]
```

Stack

The following instructions demonstrate two instructions, push and pop used to push data into the stack and pop the data out. Choose the Stack code from the emulator's dropdown and observe the outcome.



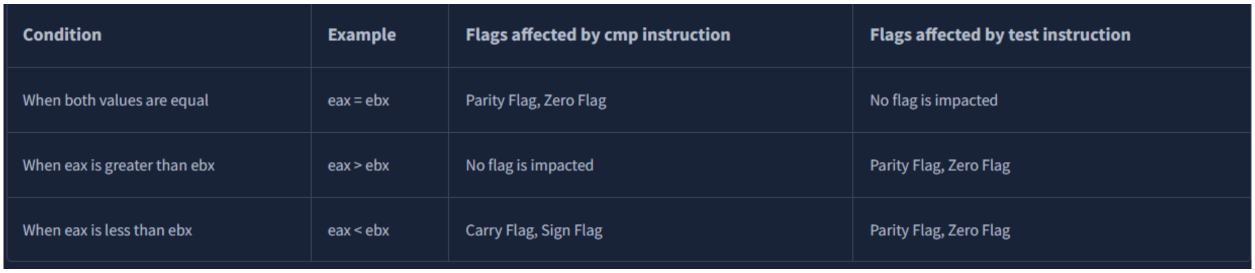
```
mov eax,10h
mov ebx, 15h
mov ecx, 20h
mov edx, 25h
; stack works from the higher memory location to the lower. Observe the stack on the right side.
push eax
push ebx
push ecx
push edx
; stack works in LIFO mode. Observe how the top of the stack is pulled out in the following instructions.

pop eax
pop ebx
pop ecx
pop edx
```

CMP and TEST Instructions

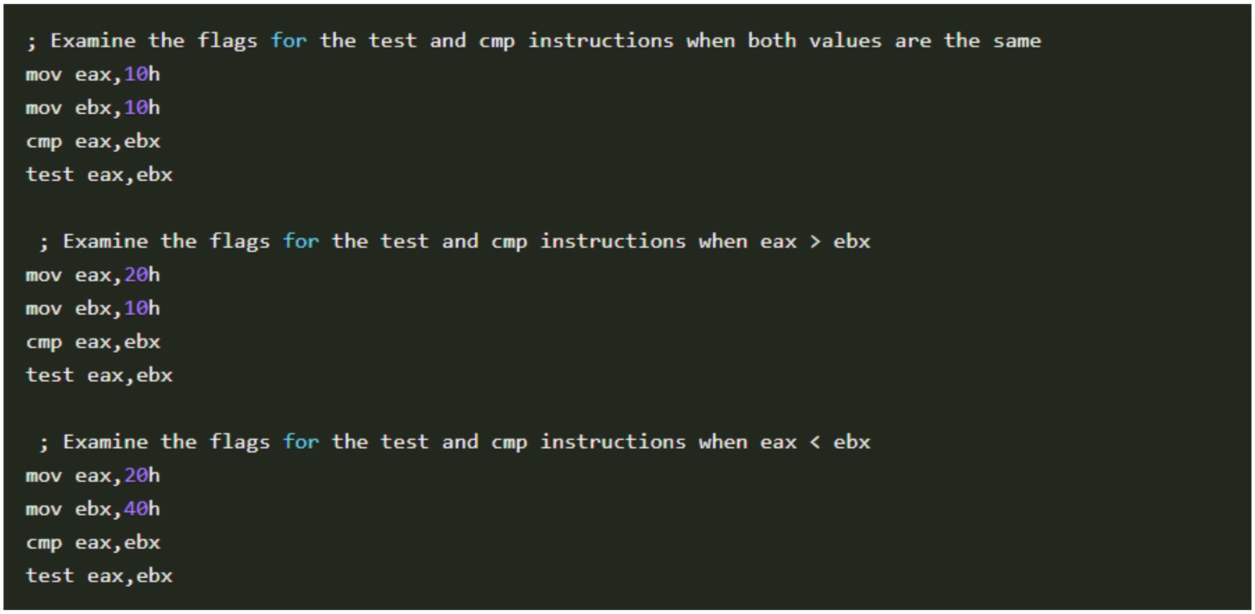
In the previous tasks, we explored two conditional instructions, test and cmp, which are used to compare the two values and set the flags based on the result. Let's visualize how the flags are changed based on the results.

While comparing two values, there are only three possible results; each result has a different impact on the critical flags like ZF and CF. The value of the flags determines the program flow. Therefore, it is essential to understand which flags are impacted by each comparison condition.



Condition	Example	Flags affected by cmp instruction	Flags affected by test instruction
When both values are equal	eax = ebx	Parity Flag, Zero Flag	No flag is impacted
When eax is greater than ebx	eax > ebx	No flag is impacted	Parity Flag, Zero Flag
When eax is less than ebx	eax < ebx	Carry Flag, Sign Flag	Parity Flag, Zero Flag

Choose the Cmp and Test Instructions code from the emulator's dropdown and verify the information mentioned in the table above.



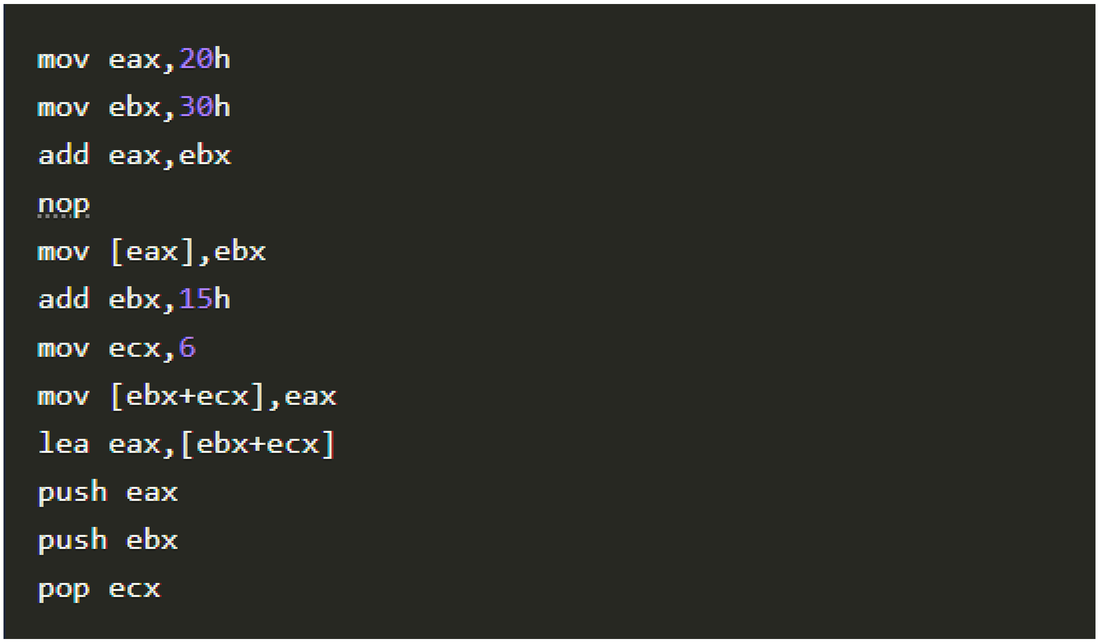
```
; Examine the flags for the test and cmp instructions when both values are the same
mov eax,10h
mov ebx,10h
cmp eax,ebx
test eax,ebx

 ; Examine the flags for the test and cmp instructions when eax > ebx
mov eax,20h
mov ebx,10h
cmp eax,ebx
test eax,ebx

 ; Examine the flags for the test and cmp instructions when eax < ebx
mov eax,20h
mov ebx,40h
cmp eax,ebx
test eax,ebx
```

LEA Instruction

Choose the Lea Instruction code from the emulator's dropdown and observe the movement of the data.



```
mov eax,20h
mov ebx,30h
add eax,ebx
nop
mov [eax],ebx
add ebx,15h
mov ecx,6
mov [ebx+ecx],eax
lea eax,[ebx+ecx]
push eax
push ebx
pop ecx
```

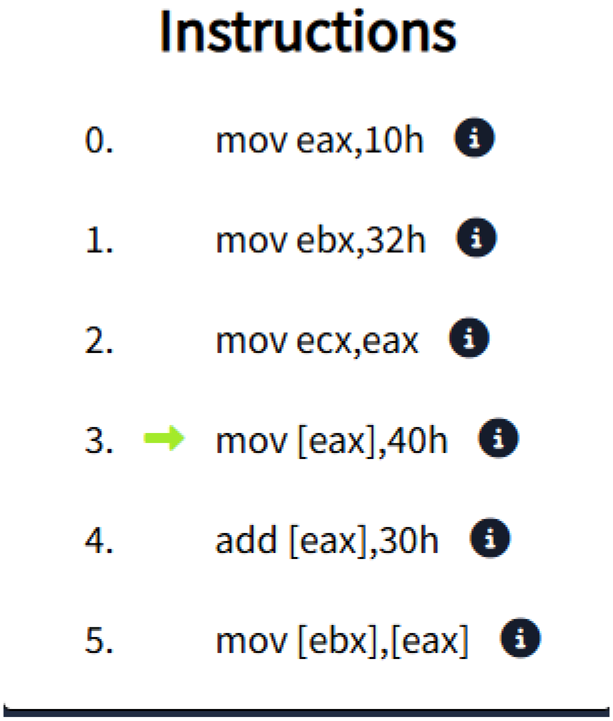
Understanding assembly instructions is a critical part of being a profound malware analyst. Practice assembly instructions and get comfortable with the instructions.

Note: The first instruction is at index 0 in the Instructions Block.

****

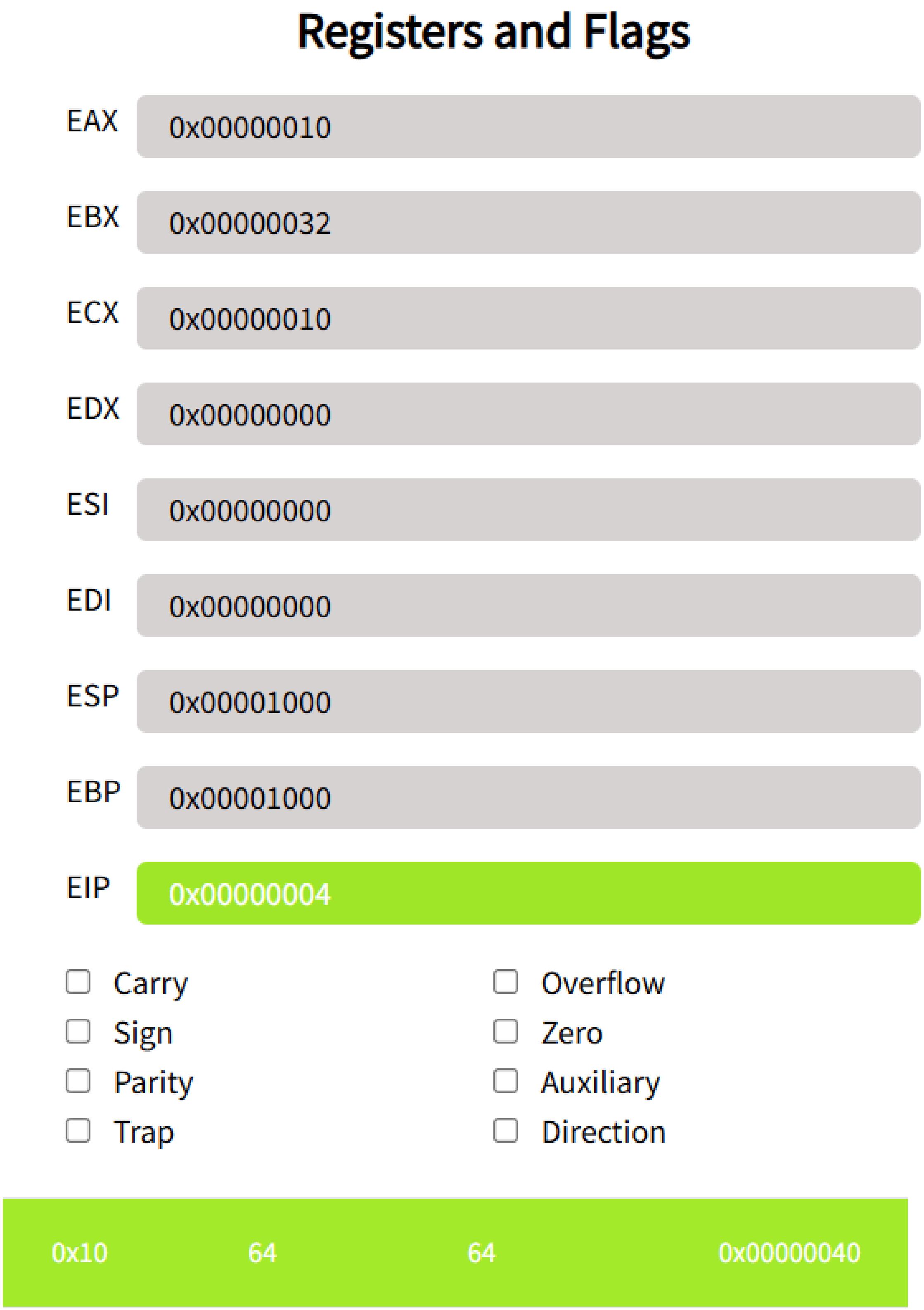
Answer the questions below:

While running the MOV instructions, what is the value of [eax] after running the 4th instruction? (in hex)



Instructions

```
0.     mov eax,10h
1.     mov ebx,32h
2.     mov ecx,eax
3.  →  mov [eax],40h
4.     add [eax],30h
5.     mov [ebx],[eax]
```



Registers and Flags

EAX	0x00000010
EBX	0x00000032
ECX	0x00000010
EDX	0x00000000
ESI	0x00000000
EDI	0x00000000
ESP	0x00001000
EBP	0x00001000
EIP	0x00000004

☐ Carry ☐ Overflow

☐ Sign ☐ Zero

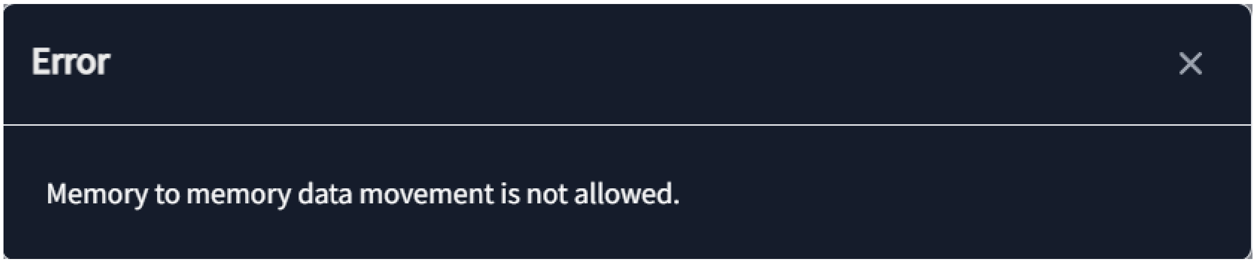
☐ Parity ☐ Auxiliary

☐ Trap ☐ Direction

0x10	64	64	0x00000040

Answer: 0x00000040

What error is displayed after running the 6th instruction from the MOV instruction section?

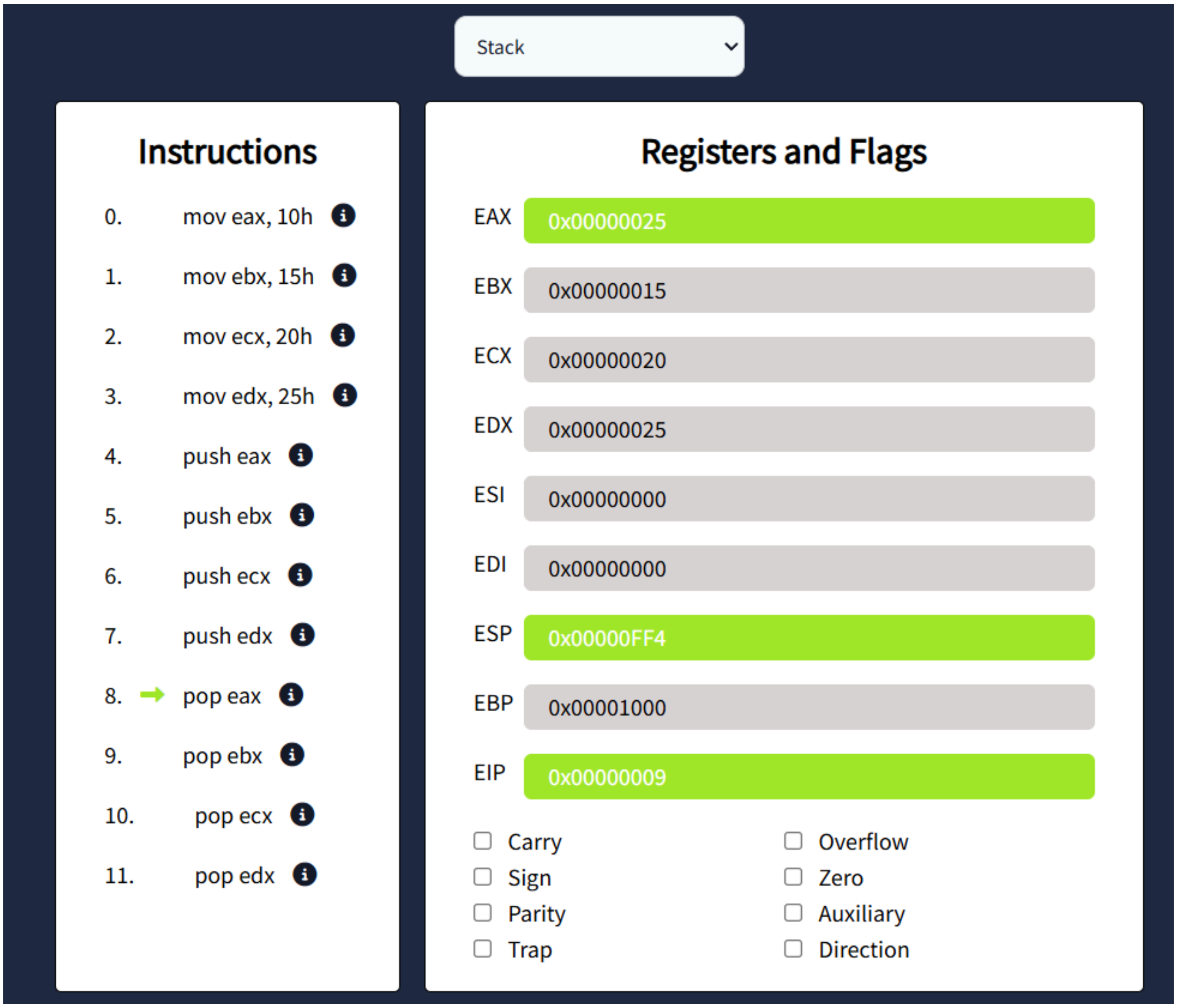


Error ✕

Memory to memory data movement is not allowed.

Answer: Memory to memory data movement is not allowed.

Run the instructions from the stack section. What is the value of eax after the 9th instruction? (in hex)



Instructions

0. mov eax, 10h
1. mov ebx, 15h
2. mov ecx, 20h
3. mov edx, 25h
4. push eax
5. push ebx
6. push ecx
7. push edx
8. → pop eax
9. pop ebx
10. pop ecx
11. pop edx

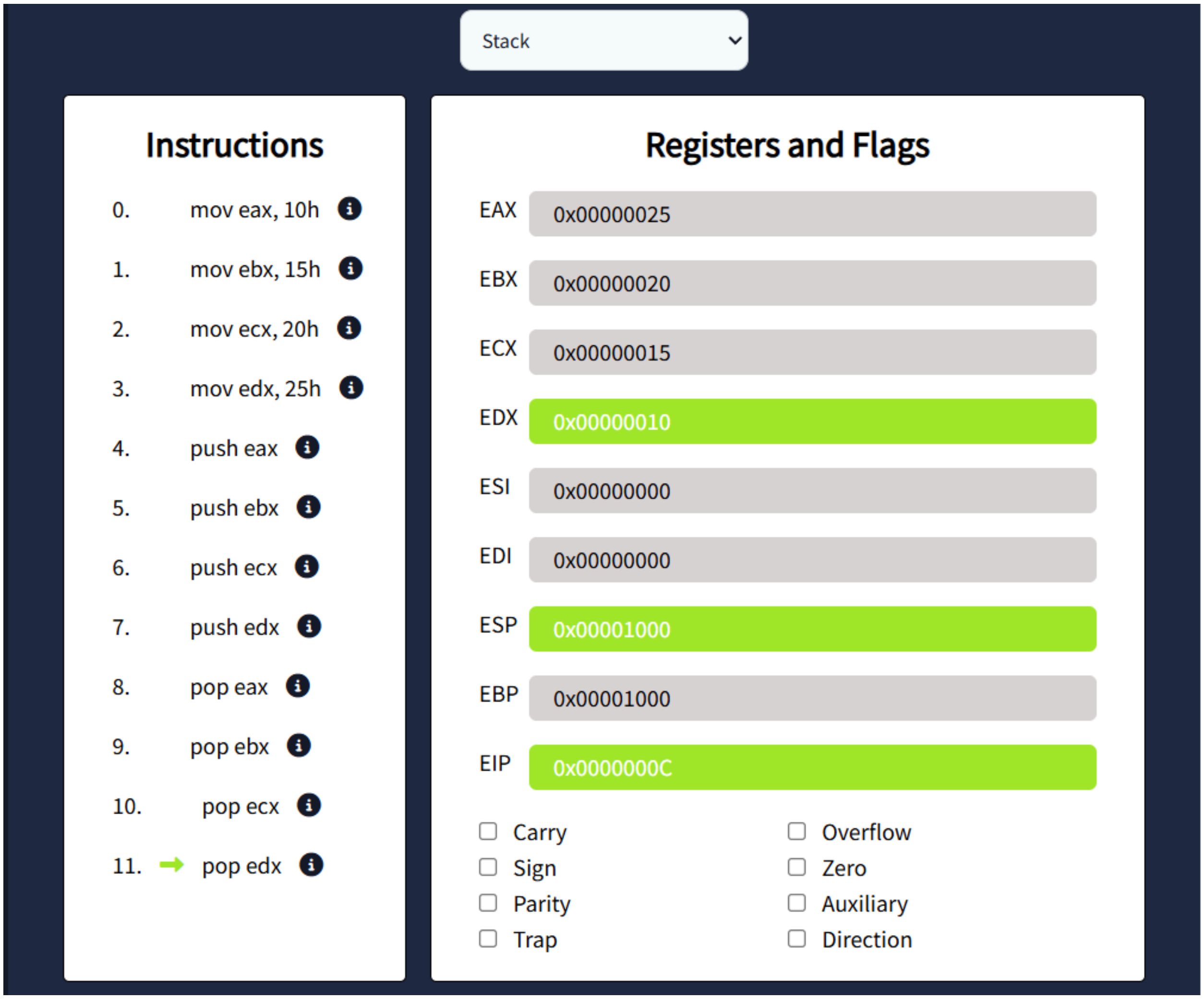
Registers and Flags

EAX	0x00000025
EBX	0x00000015
ECX	0x00000020
EDX	0x00000025
ESI	0x00000000
EDI	0x00000000
ESP	0x00000FF4
EBP	0x00001000
EIP	0x00000009

□ Carry □ Overflow
□ Sign □ Zero
□ Parity □ Auxiliary
□ Trap □ Direction

Answer: 0x00000025

Run the instructions from the stack section. What is the value of edx after the 12th instruction? (in hex)



Instructions

0. mov eax, 10h
1. mov ebx, 15h
2. mov ecx, 20h
3. mov edx, 25h
4. push eax
5. push ebx
6. push ecx
7. push edx
8. pop eax
9. pop ebx
10. pop ecx
11. → pop edx

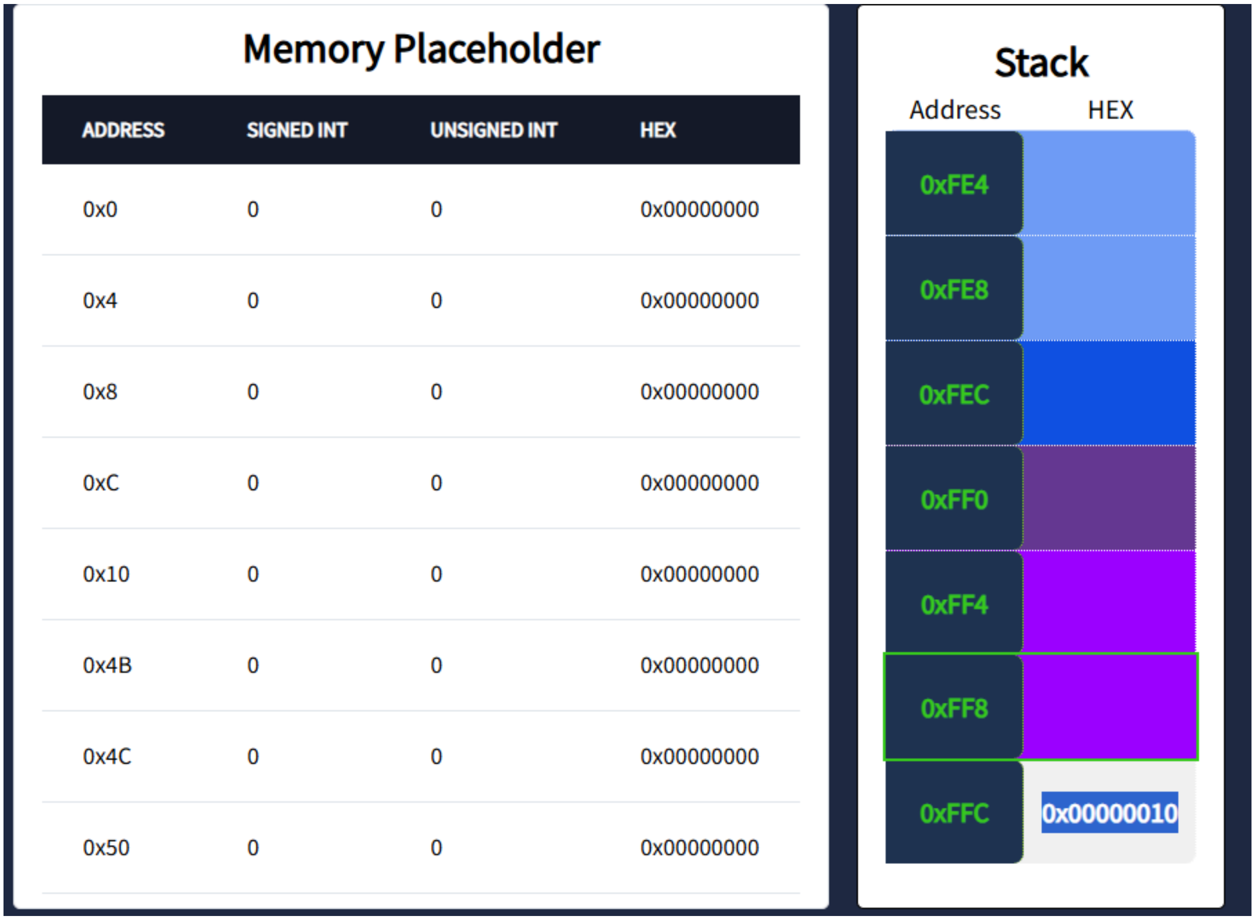
Registers and Flags

EAX	0x00000025
EBX	0x00000020
ECX	0x00000015
EDX	0x00000010
ESI	0x00000000
EDI	0x00000000
ESP	0x00001000
EBP	0x00001000
EIP	0x0000000C

□ Carry　□ Overflow
□ Sign　□ Zero
□ Parity　□ Auxiliary
□ Trap　□ Direction

Answer: 0x00000010

Run the instructions from the stack section. After POP ecx, what is the value left at the top of the stack? (in hex)



Memory Placeholder

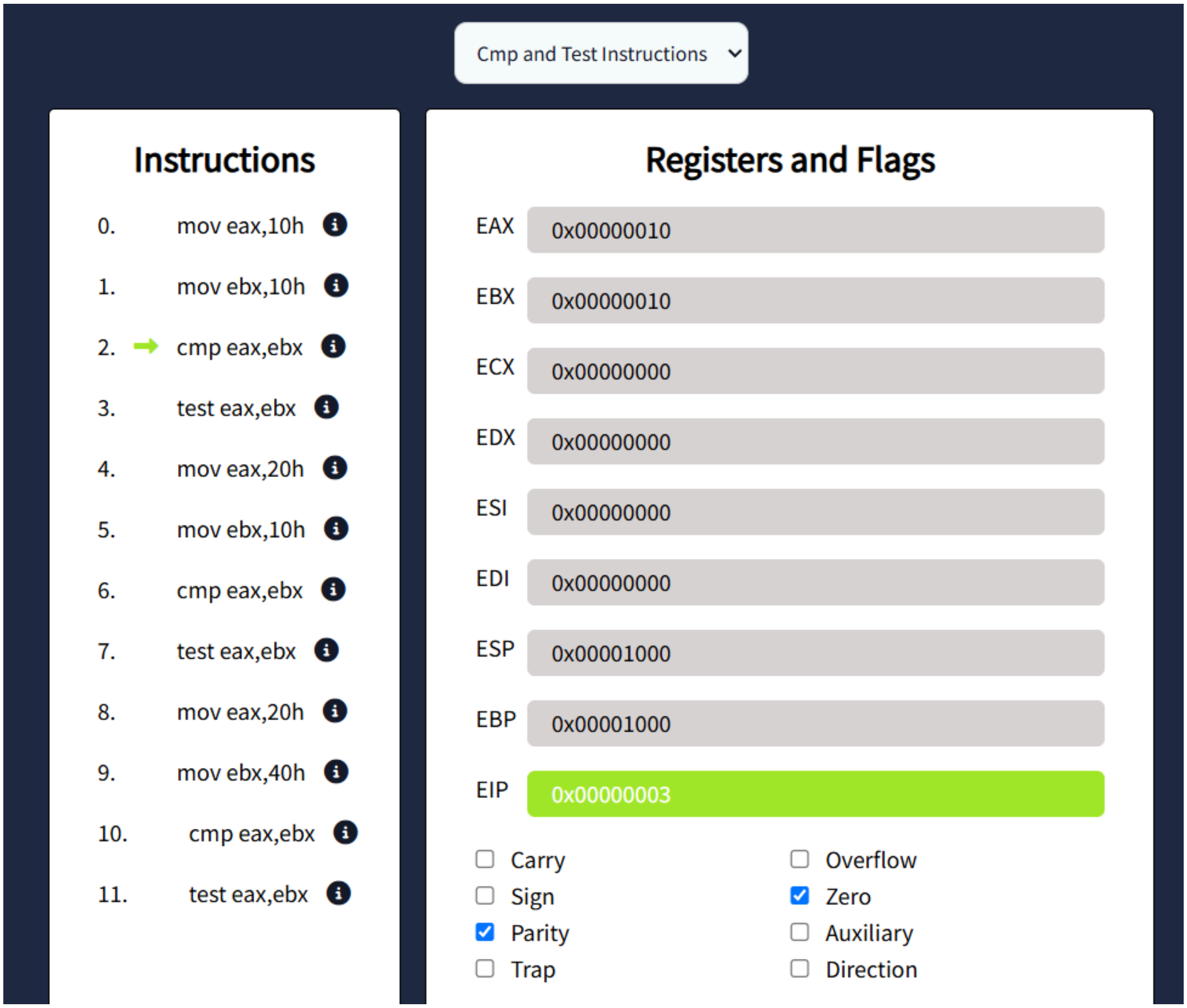
ADDRESS	SIGNED INT	UNSIGNED INT	HEX
0x0	0	0	0x00000000
0x4	0	0	0x00000000
0x8	0	0	0x00000000
0xC	0	0	0x00000000
0x10	0	0	0x00000000
0x4B	0	0	0x00000000
0x4C	0	0	0x00000000
0x50	0	0	0x00000000

Stack

Address	HEX
0xFE4	
0xFE8	
0xFEC	
0xFF0	
0xFF4	
0xFF8	
0xFFC	0x00000010

Answer: 0x00000010

Run the cmp and test instructions. Which flags are triggered after the 3rd instruction?
(Note: Use these abbreviations in alphabetical order with no spaces: CF,PF,SF,ZF)



Cmp and Test Instructions

Instructions

0. mov eax,10h
1. mov ebx,10h
2. ➡ cmp eax,ebx
3. test eax,ebx
4. mov eax,20h
5. mov ebx,10h
6. cmp eax,ebx
7. test eax,ebx
8. mov eax,20h
9. mov ebx,40h
10. cmp eax,ebx
11. test eax,ebx

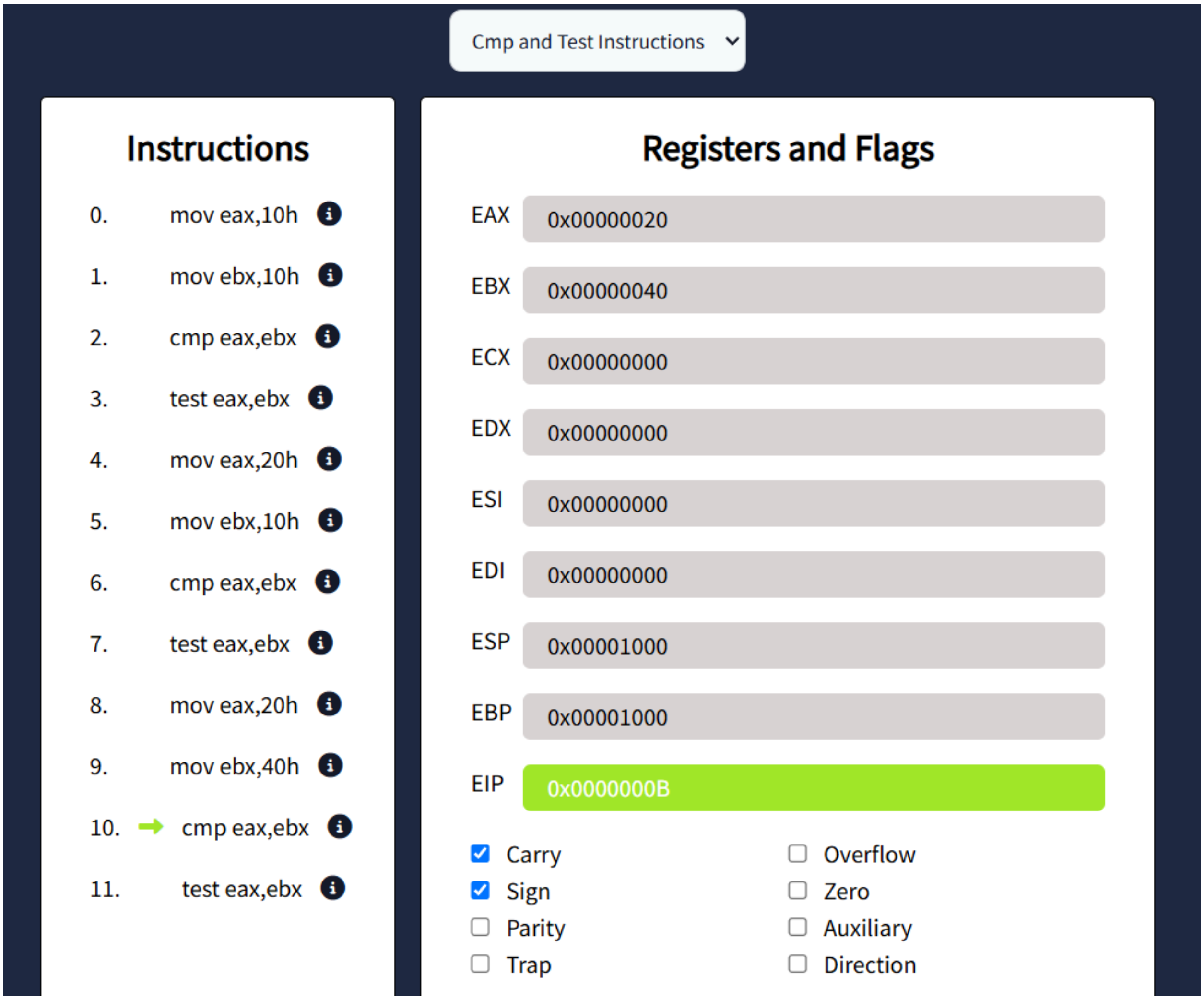
Registers and Flags

EAX	0x00000010
EBX	0x00000010
ECX	0x00000000
EDX	0x00000000
ESI	0x00000000
EDI	0x00000000
ESP	0x00001000
EBP	0x00001000
EIP	0x00000003

□ Carry	□ Overflow
□ Sign	☑ Zero
☑ Parity	□ Auxiliary
□ Trap	□ Direction

Answer: PF,ZF

Run the test and the cmp instructions. Which flags are triggered after the 11th instruction?
(Note: Use these abbreviations in alphabetical order with no spaces: CF,PF,SF,ZF)



Instructions

0. mov eax,10h
1. mov ebx,10h
2. cmp eax,ebx
3. test eax,ebx
4. mov eax,20h
5. mov ebx,10h
6. cmp eax,ebx
7. test eax,ebx
8. mov eax,20h
9. mov ebx,40h
10. ➡ cmp eax,ebx
11. test eax,ebx

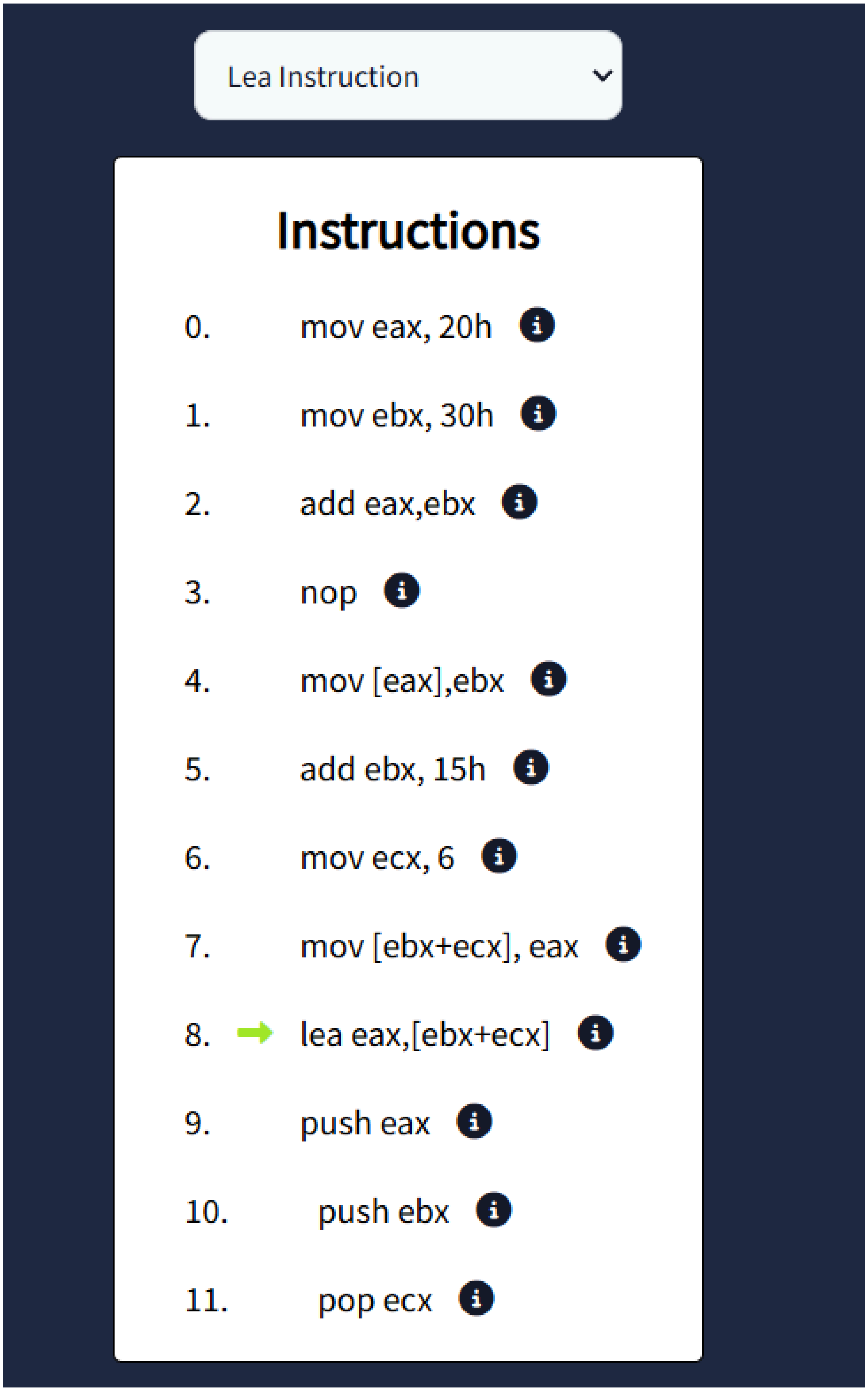
Registers and Flags

EAX 0x00000020
EBX 0x00000040
ECX 0x00000000
EDX 0x00000000
ESI 0x00000000
EDI 0x00000000
ESP 0x00001000
EBP 0x00001000
EIP 0x0000000B

☑ Carry ☐ Overflow
☑ Sign ☐ Zero
☐ Parity ☐ Auxiliary
☐ Trap ☐ Direction

Answer: CF,SF

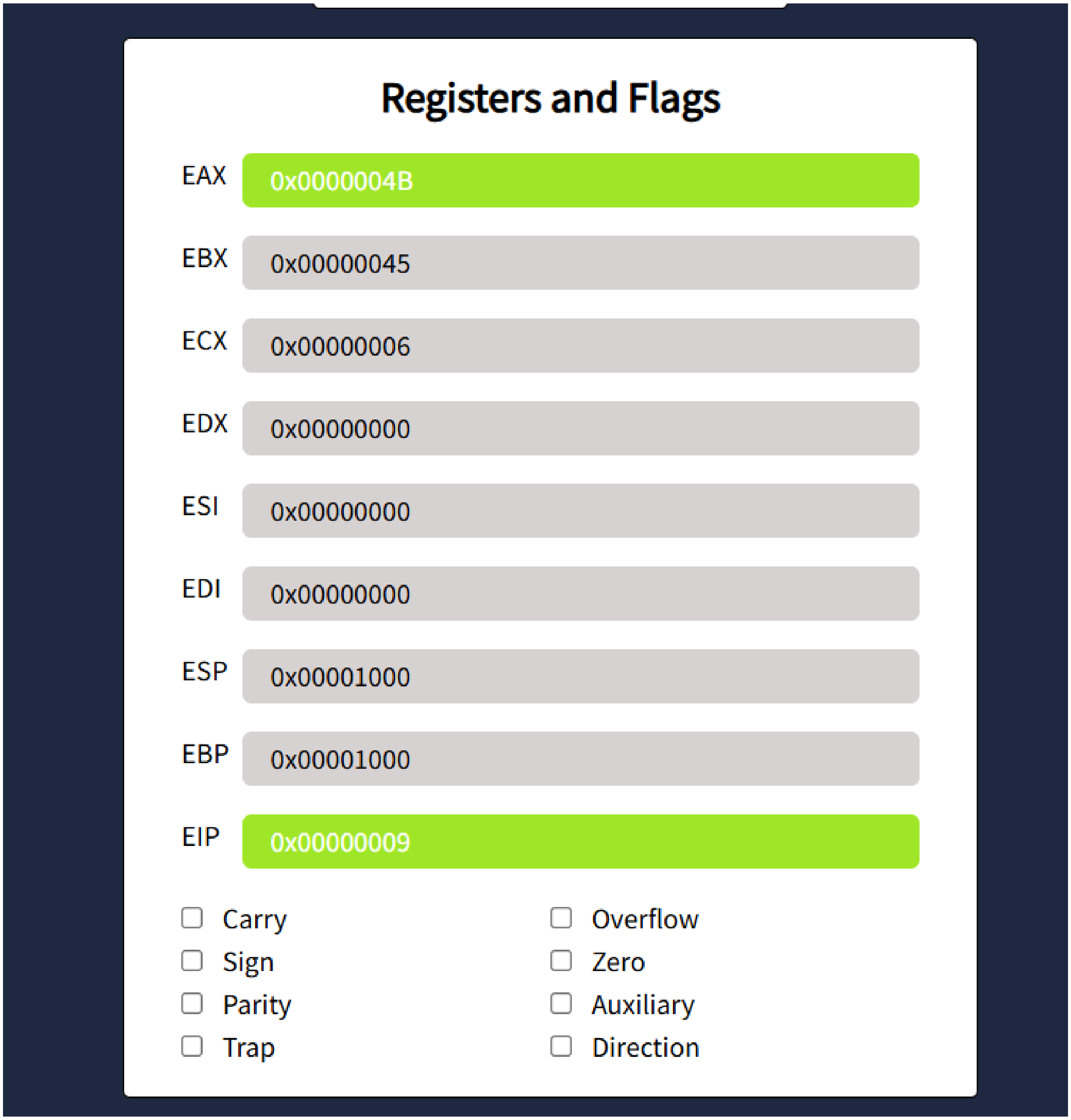
Run the instructions from the lea section. What is the value of eax after running the 9th instruction? (in hex)



Lea Instruction

Instructions

```
0.      mov eax, 20h
1.      mov ebx, 30h
2.      add eax,ebx
3.      nop
4.      mov [eax],ebx
5.      add ebx, 15h
6.      mov ecx, 6
7.      mov [ebx+ecx], eax
8.  →   lea eax,[ebx+ecx]
9.      push eax
10.     push ebx
11.     pop ecx
```



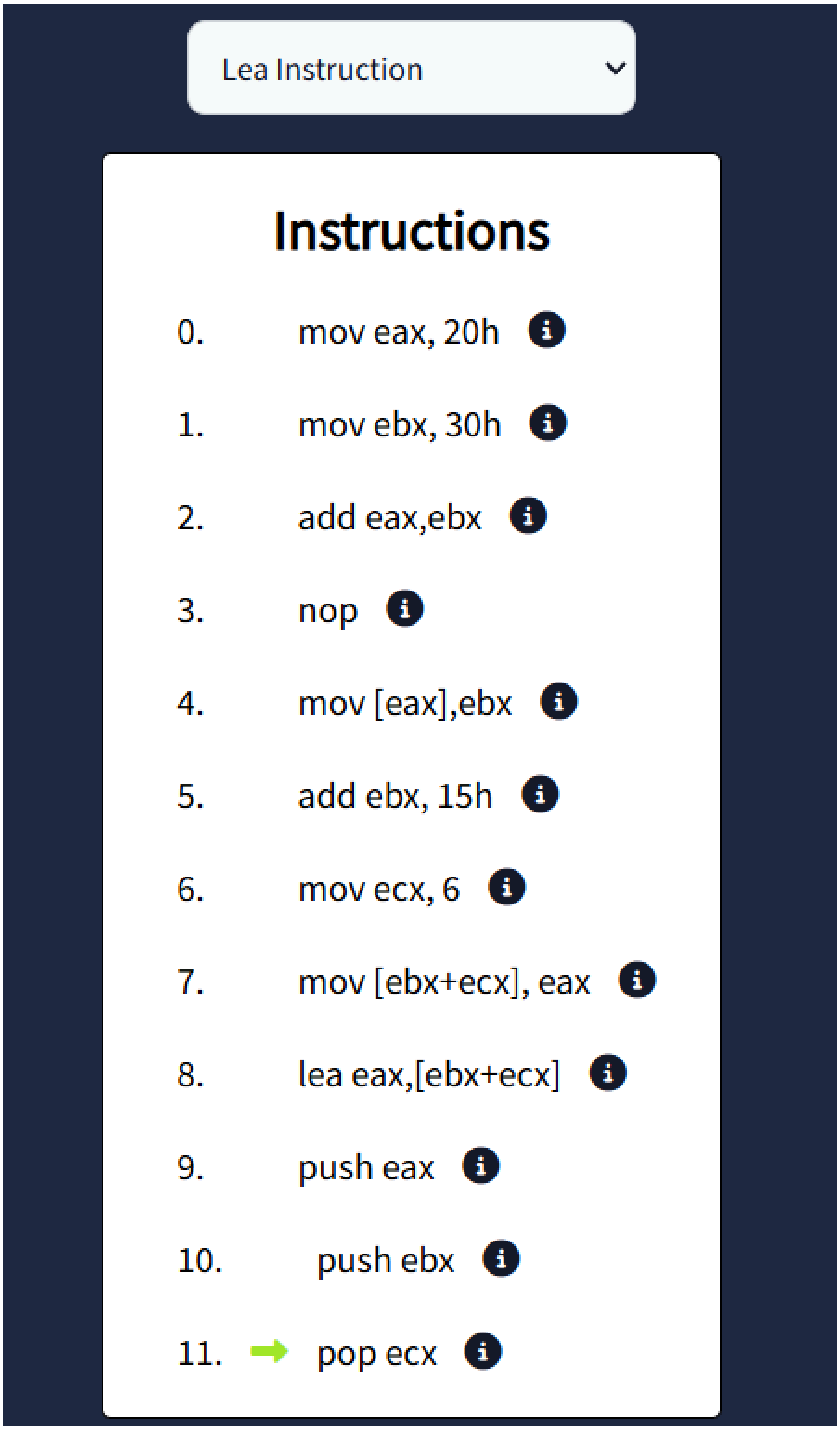
Registers and Flags

```
EAX  0x0000004B
EBX  0x00000045
ECX  0x00000006
EDX  0x00000000
ESI  0x00000000
EDI  0x00000000
ESP  0x00001000
EBP  0x00001000
EIP  0x00000009
```

☐ Carry ☐ Overflow
☐ Sign ☐ Zero
☐ Parity ☐ Auxiliary
☐ Trap ☐ Direction

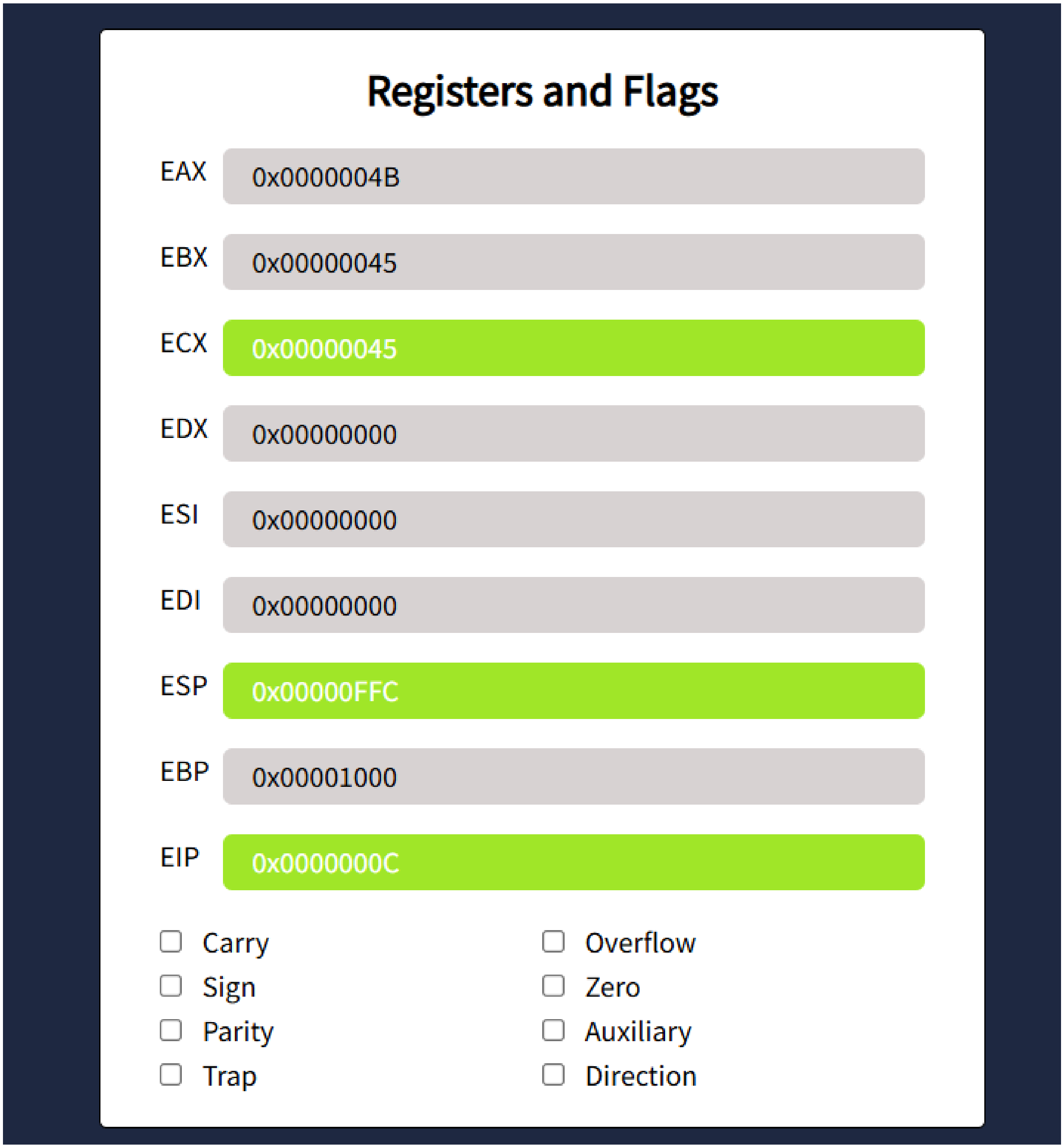
Answer: 0x0000004B

Run the instructions from the lea section. What is the final value found in the ECX register? (in hex)



Instructions

```
0.      mov eax, 20h
1.      mov ebx, 30h
2.      add eax,ebx
3.      nop
4.      mov [eax],ebx
5.      add ebx, 15h
6.      mov ecx, 6
7.      mov [ebx+ecx], eax
8.      lea eax,[ebx+ecx]
9.      push eax
10.     push ebx
11. →   pop ecx
```



Registers and Flags

EAX	0x0000004B
EBX	0x00000045
ECX	0x00000045
EDX	0x00000000
ESI	0x00000000
EDI	0x00000000
ESP	0x00000FFC
EBP	0x00001000
EIP	0x0000000C

□ Carry
□ Sign
□ Parity
□ Trap
□ Overflow
□ Zero
□ Auxiliary
□ Direction

Answer: 0x00000045

Conclusion

In this room, we covered some of the most commonly used instructions in x86 assembly language. Though there are many other instructions, we covered the ones that briefly overview how operations are performed in the assembly language. As we move forward to other rooms in this module, we will learn about some more instructions as per requirements. In this room, we have covered the following:

- Opcodes and how we convert them into the assembly language.
- General instructions such as move, load effective address, shift and rotate.
- Arithmetic instructions such as addition, subtraction, multiplication, and division.
- Conditionals and how we use them for branching.
- Stack, the push and pop instructions, and function calls.