МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

#### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ

БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет автоматики и вычислительной техники

Кафедра радиоэлектронных средств

Отчет по дисциплине

«Цифровые устройства и микропроцессоры»

Лабораторная работа №1

«ПРИНЦИПЫ

ВЫПОЛНЕНИЯ КОМАНД ВЕТВЛЕНИЯ, ОРГАНИЗАЦИЯ

ЦИКЛОВ И ПОДПРОГРАММ»

Вариант №15

|  |  |  |
| --- | --- | --- |
| Выполнил: студент гр. ИНБс-3301-01-00 | \_\_\_\_\_\_\_\_\_\_\_\_ | С.А. Симахин |
| Проверил: доцент кафедры РЭС | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | М. А. Земцов |

#### Киров 2025

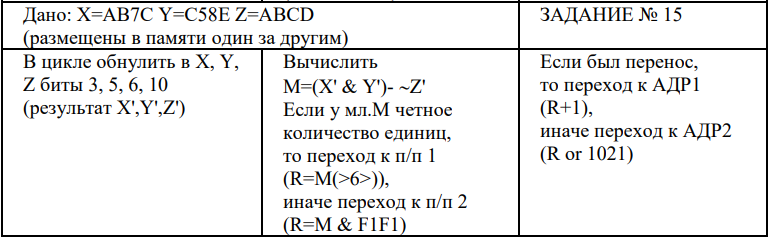
**Цель работы:** изучение принципов выполнения команд ветвления,

организации циклов и подпрограмм микропроцессоров с архитектурой x86.

**Ход работы:**

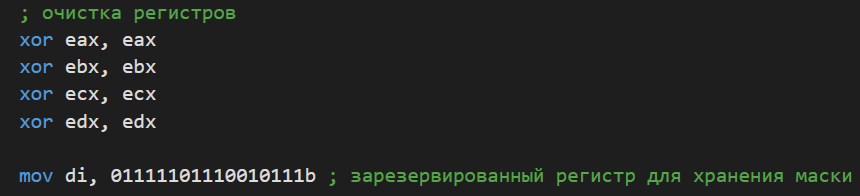
Исходные данные представлены в таблице 1 и пояснении ниже.

Таблица 1 – Исходные данные



**1.** В цикле обнулить в X, Y, Z биты 3, 5, 6, 10 (результат X’, Y’, Z’)

Для начала работы необходимо очистить регистры, с которыми будем работать, а также ввести значение в регистр для последующего использования



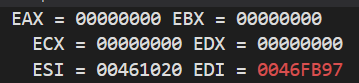


Рисунок 1 – очистка регистров и ввод значения

Ввод данного значения был обусловлен требованием обнуления 3, 5, 6, 10 бит



Рисунок 2 – «маска» для обнуления битов

Количество итераций 1 цикла loop\_s обусловлено количеством переменных, которых требуется изменить, а именно 3 (X, Y, Z)





Рисунок 3 – число итераций цикла loop\_s для обнуления битов

Для последующей работе с циклом loop\_s и выполнением задания, нам необходимо ввести в регистр EAX адрес хранения переменной X (для I итерации цикла), для этого используется функция **offset**



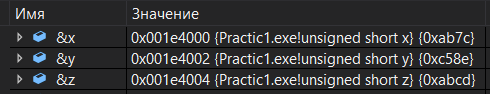






Рисунок 4 – передача адреса хранения переменной X в регистр EAX

Далее рассмотрим работу цикла loop\_s:

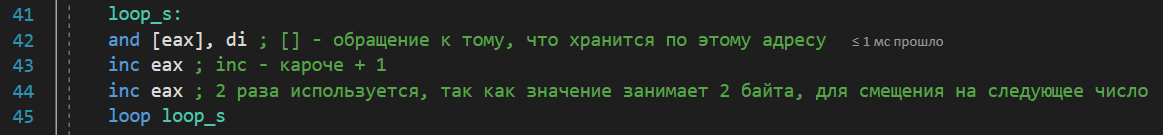


Рисунок 5 – цикл loop\_s

42 строка: логическое И (**and**) между тем, что хранится в адресе EAX (**[eax]**) и значением в регистре **di**

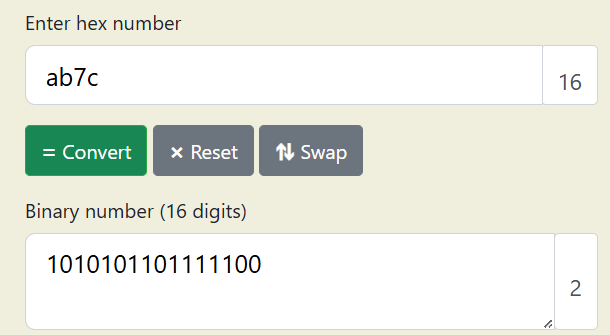
****

Рисунок 6 – перевод значения Х в двоичную систему счисления, I итерация цикла

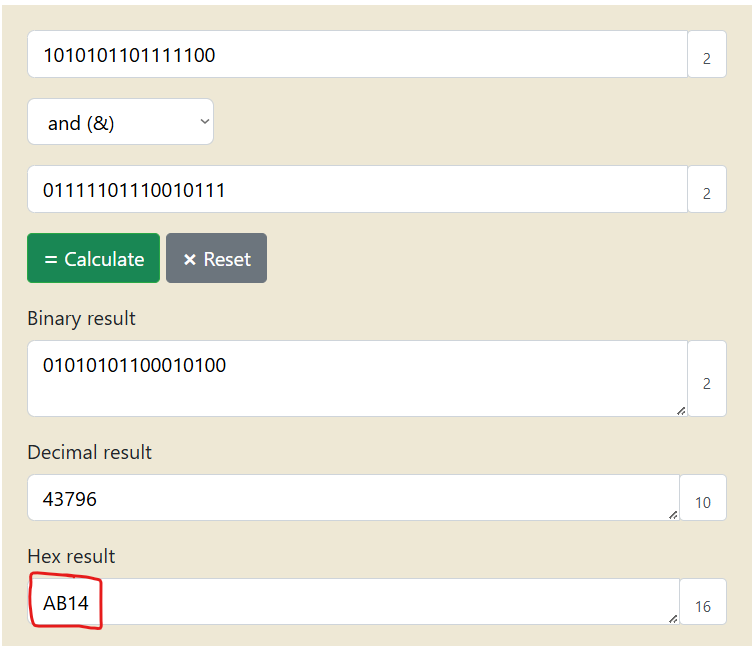
****

Рисунок 7 – операция логического «И» и его верификация, I итерация цикла

43 и 44 строки: **inc eax** используем для перехода к следующему числу в адресном пространстве, так как в нашем случае число занимает 2 байта, то используем это 2 раза



Рисунок 8 – переход на 2 байта адресного пространства регистра EAX,

I итерация цикла

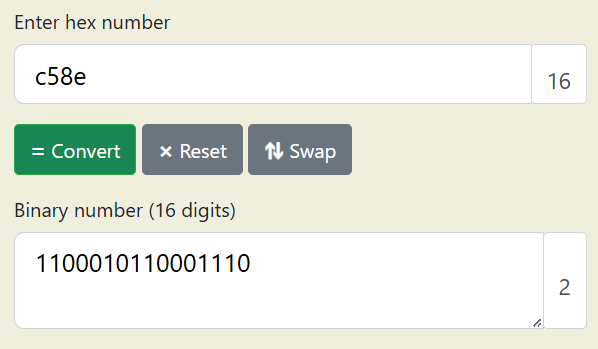


Рисунок 9 – перевод значения Y в двоичную систему счисления, II итерация цикла

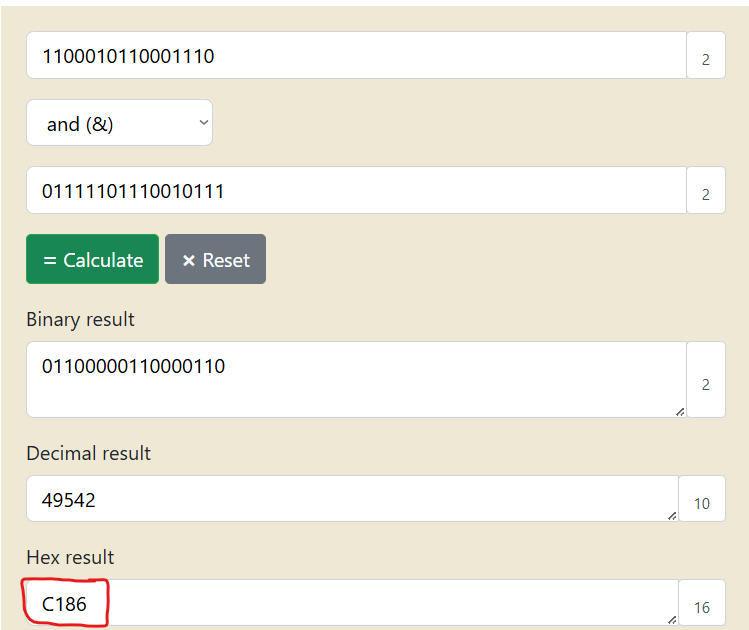




Рисунок 10 – операция логического «И» и его верификация, II итерация цикла

****

Рисунок 11 – переход на 2 байта адресного пространства регистра EAX,

II итерация цикла

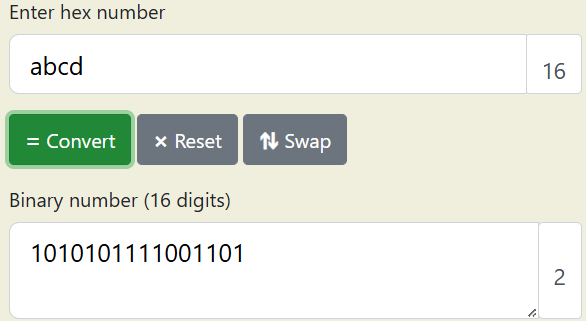
****

Рисунок 12 – перевод значения Z в двоичную систему счисления, III итерация цикла

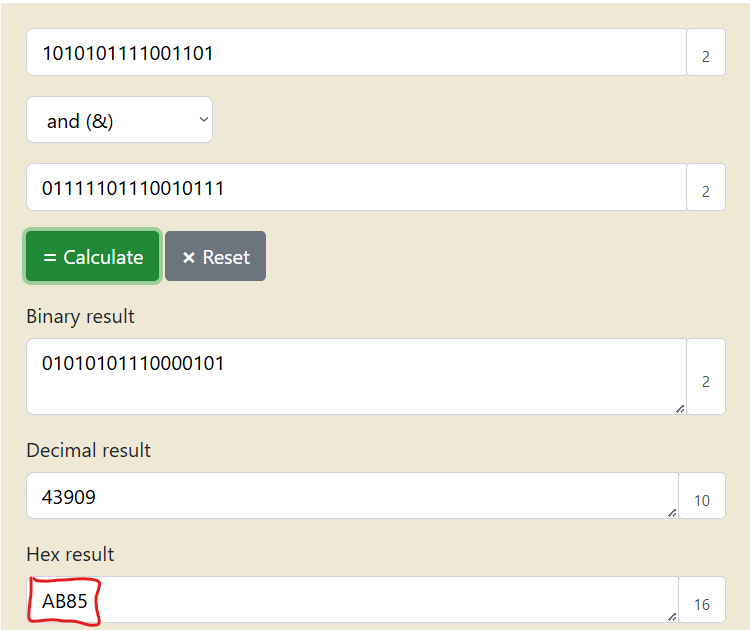
****

Рисунок 13 – операция логического «И» и его верификация, III итерация цикла

****

Рисунок 14 – переход на 2 байта адресного пространства регистра EAX,

III итерация цикла

**2.** Вычислить M = (X' & Y') - ~Z'

Если у мл. М четное количество единиц, то переход к п/п 1

(R = M (> 6 > )), иначе переход к п/п 2 (R = M & F1F1)

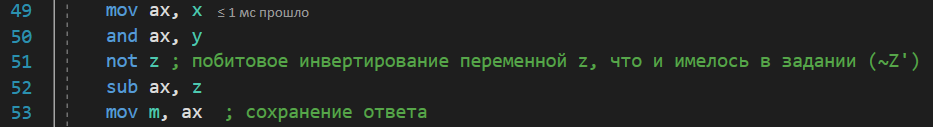
****

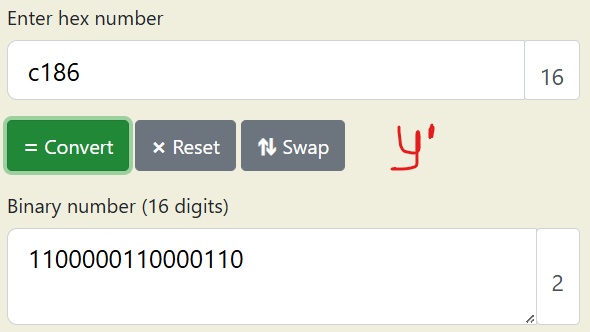
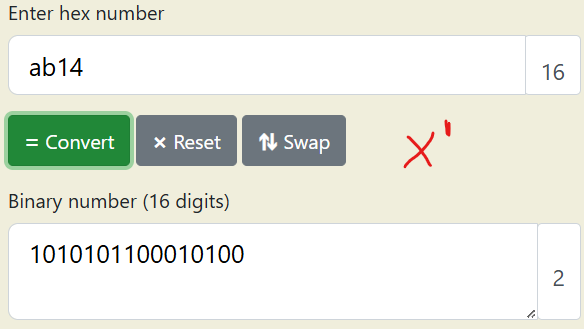
Рисунок 15 – вычисление числа M

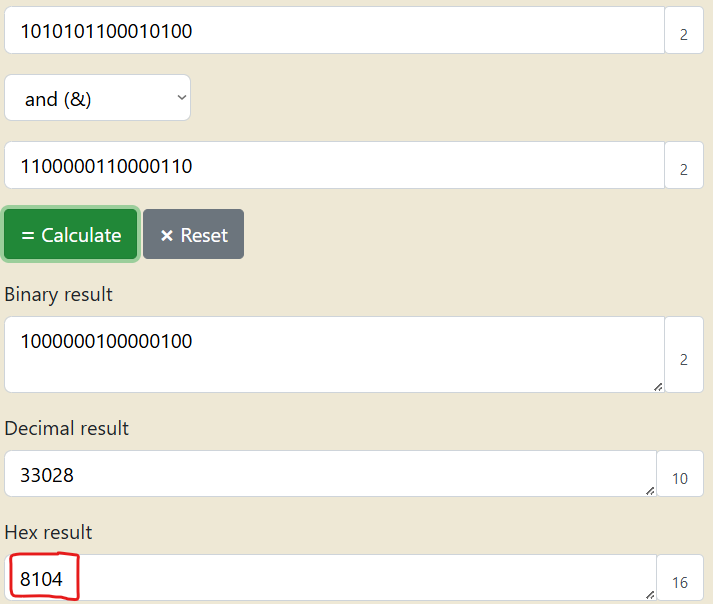
49 строка: заносим в регистр AX значение X’

50 строка: логическое «И» между регистром AX и переменной Y’

Ответ:

Верификация:

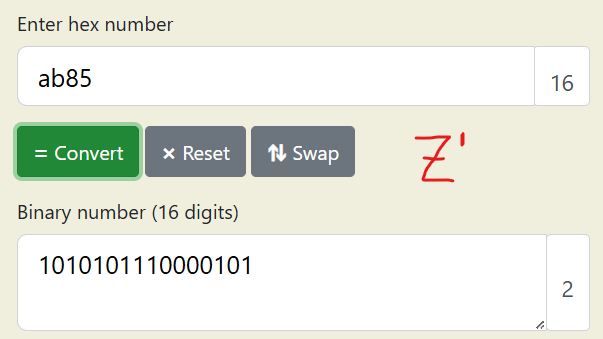




🡨 Ответ совпал

51 строка: побитовое инвертирование переменной Z, получение ~Z’

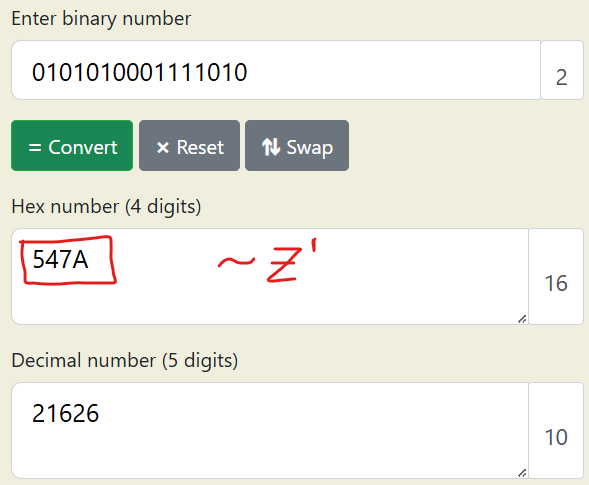
Верификация:



1010101110000101

↓

0101010001111010

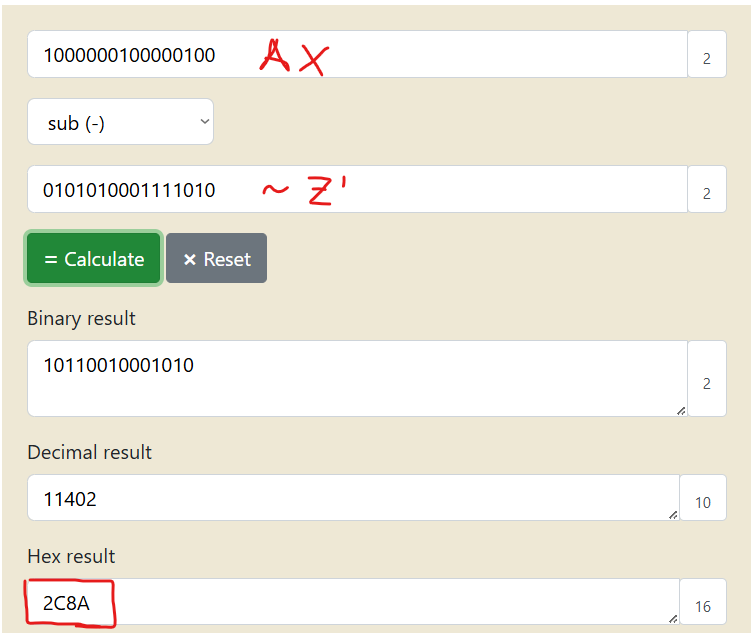
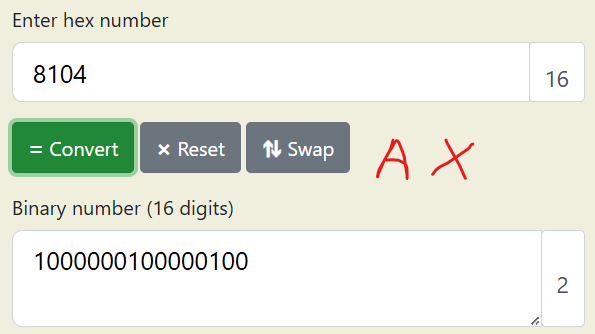


🡨 Ответ совпал

52 строка: вычитание из регистра AX переменной ~Z’



Верификация:

🡨 Ответ совпал

53 строка: сохраняем полученный ответ в переменную m

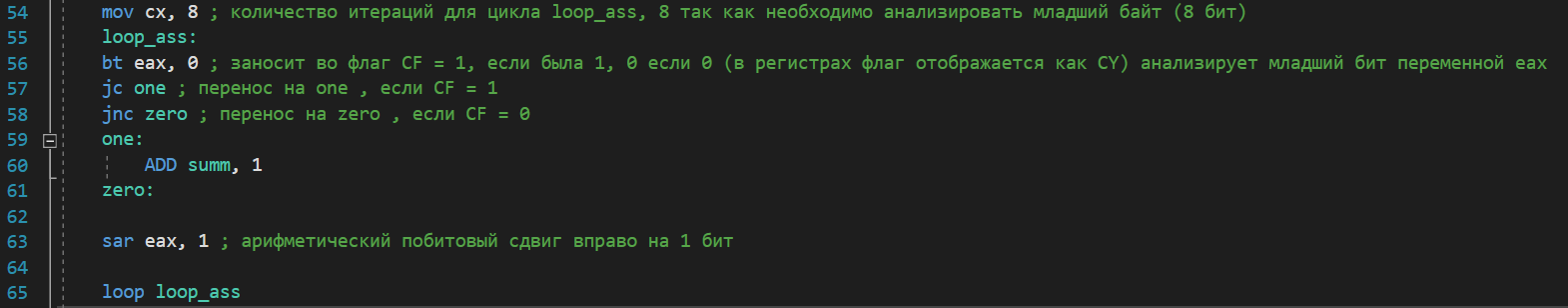


Рисунок 16 – цикл loop\_ass

Цикл loop\_ass считает количество единиц в регистре EAX

54 строка: количество итераций для цикла loop\_ass

56 строка: **bt** (Bit Test) — это команда ассемблера x86, предназначенная для проверки состояния (значения) отдельного бита в регистре или области памяти. Проверяет значение бита с указанным номером в первом операнде. Копирует это значение (0 или 1) во флаг CF (Carry Flag) процессора

57 и 58 строки: перенос в точки one и zero соответственно

**jc** (Jump if Carry) и **jnc** (Jump if Not Carry) — условные переходы, управляемые состоянием флага CF (Carry Flag).

**jc** – переход, если флаг CF = 1

**jnc** – переход, если флаг CF = 0

**Важно, что даже после захода в точку “one” он все равно после зайдет в точку “zero”**

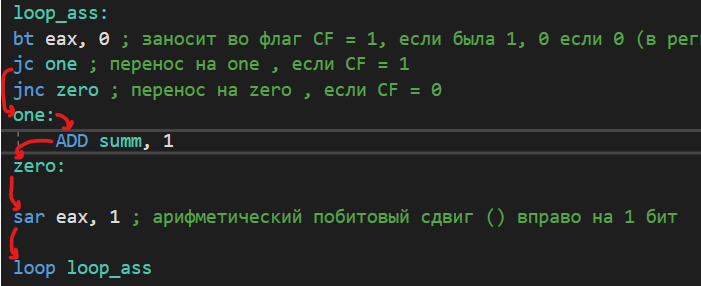


Рисунок 17 – работа обхода цикла loop\_ass

После перехода в точку «**one**» к переменной summ прибавляется 1

Сама переменная **summ** создана, что бы в себе хранить количество единиц

63 строка:

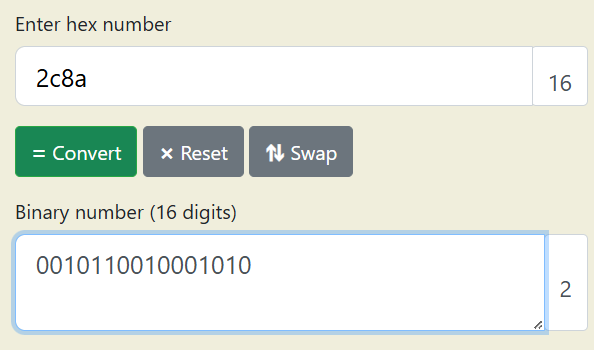
**sar** (Shift Arithmetic Right) выполняет знаковый сдвиг операнда вправо, сохраняя знаковый бит (старший бит) и заполняя освободившиеся старшие биты его значением. Это эквивалентно целочисленному делению на 2ⁿ со знаком.

Конечный результат данного цикла:



Рисунок 18 – результат работы цикла loop\_ass

Верификация:



В числе 00101100 **1**000**1**0**1**0 всего 3 (нечетное) единицы в младшем байте числа 🡪 переход в odin (71 строка)

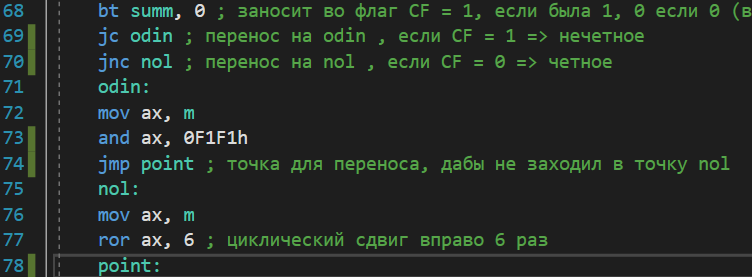


Рисунок 19 – ветвление с использованием прошлого результата (summ)

68 строка: проверка на четность / нечетность значения, анализирует младший байт значения summ

Если: нечетное 🡪 CF(CY) = 1 🡪 переход в точку **odin**

**** четное 🡪 CF(CY) = 0 🡪 переход в точку **nol**

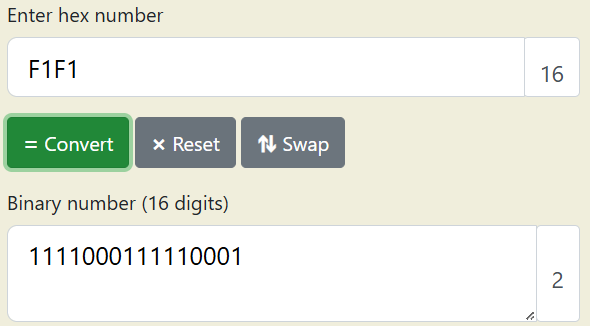
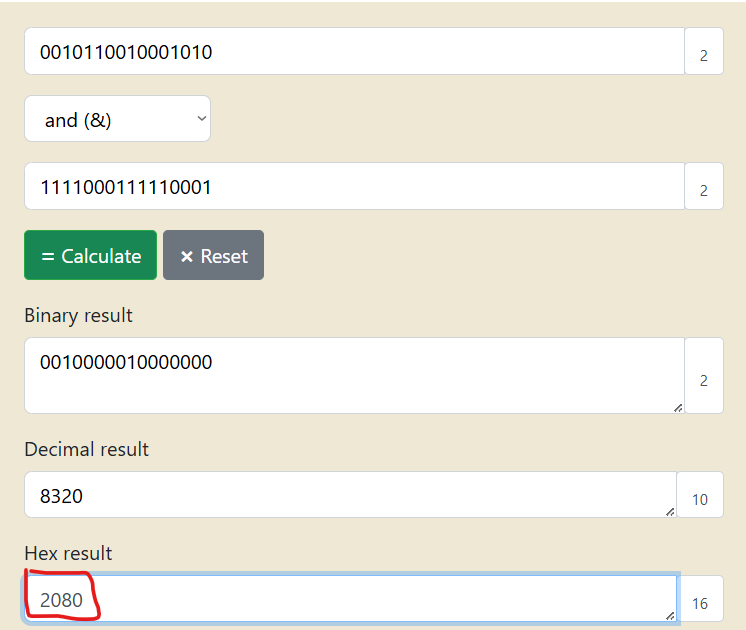
72 строка: перенос значения переменной **m** в регистр **AX**

73 строка: логическое «И» между регистр **AX** и значением F1F1

Ответ:

74 строка: точка переноса **jmp point** для обхода точки **nol**

Верификация:

 🡨 Ответ совпал

**3. Если был перенос, то переход к АДР1 (R+1), иначе переход к АДР2 (R or 1021)**

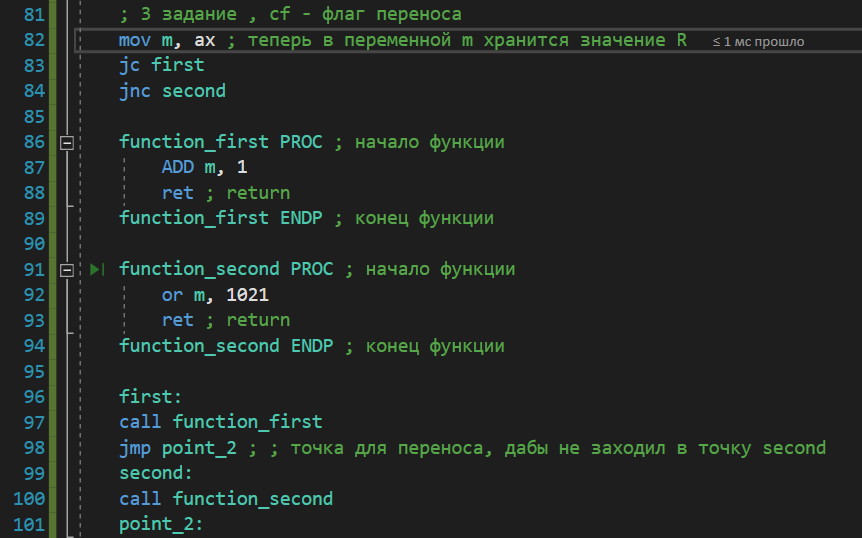


Рисунок 20 – код для 3 задания

82 строка: ввод в переменную **m** значения, хранящегося в регистре **AX**

****

**jc** – переход, если флаг CF (CY)= 1 (был переход)

**jnc** – переход, если флаг CF (CY)= 0 (не было перехода)

В нашем случае мы переходим в точку **“second”** перехода не было

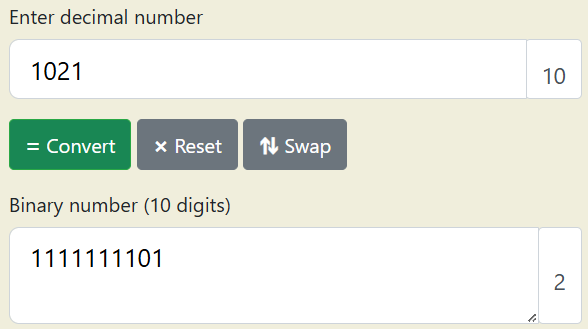
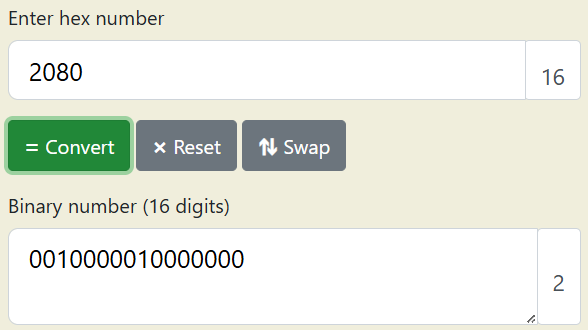
100 строка: вызов функции **function\_second**

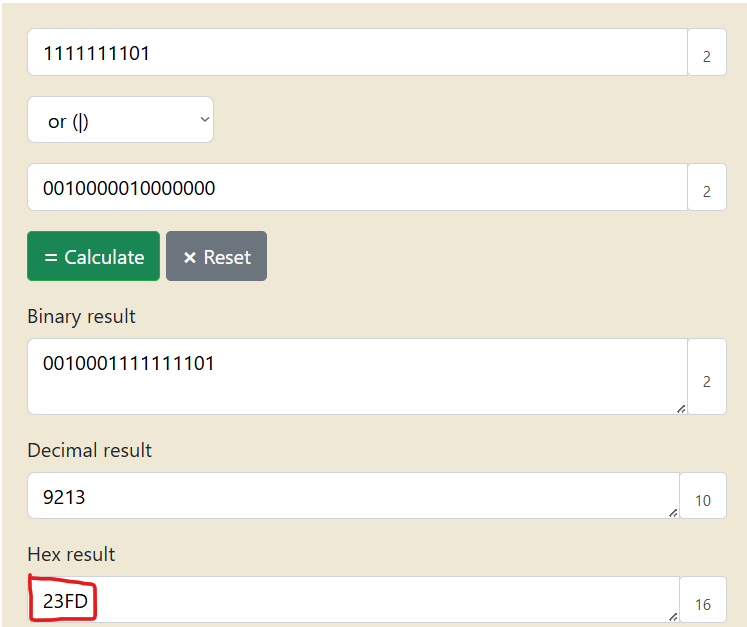
92 строка: логическое «ИЛИ» между переменной m и числом 1021

93 строка: ret (return) – взвращение обратно к точке, откуда вызывалась функция

98 строка: точка для обхода функции jmp point\_2

Верификация 92 строки:





**Код программы:**

; 15 вариант

; X = AB7C Переменные размещены

; Y = C58E в памяти

; Z = ABCD один за другим

; В цикле обнулить в X, Y, Z биты 3, 5, 6, 10 (результат X',Y',Z')

; Вычислить M = (X' & Y') - ~Z'

; Если у мл.М четное количество единиц, то переход к п/п 1

; (R = M (> 6 > )), иначе переход к п/п 2 (R = M & F1F1)

; Если был перенос, то переход к АДР 1 (R + 1),

; иначе переход к АДР 2 (R or 1021)

.686

.model flat, stdcall

.stack 100h

.data

; задача переменных

x dw 0AB7Ch ; AB7Ch == 43900 (10) == 1010101101111100b

y dw 0C58Eh ; C58Eh == 50574 (10) == 1100010110001110b

z dw 0ABCDh ; ABCDh == 43981 (10) == 1010101111001101b

summ dw 0 ; переменная для 3 задания, сумма единиц

m dw 0 ; переменная для хранения ответа

.code

ExitProcess PROTO STDCALL :DWORD

Start:

; очистка регистров

xor eax, eax

xor ebx, ebx

xor ecx, ecx

xor edx, edx

mov di, 01111101110010111b ; зарезервированный регистр для хранения маски

; обнуление в цикле X, Y, Z биты 3, 5, 6, 10

mov cx, 3 ; число итераций цикла loop\_s

mov eax, offset x ; offset - функция, передается адрес в котором хранится переменная, в данном случае х

loop\_s:

and [eax], di ; [] - обращение к тому, что хранится по этому адресу

inc eax ; inc - кароче + 1

inc eax ; 2 раза используется, так как значение занимает 2 байта, для смещения на следующее число

loop loop\_s

; Вычислить M = (X' & Y') - ~Z' 2 задание

mov ax, x

and ax, y

not z ; побитовое инвертирование переменной z, что и имелось в задании (~Z')

sub ax, z

mov m, ax ; сохранение ответа

mov cx, 8 ; количество итераций для цикла loop\_ass, 8 так как необходимо анализировать младший байт (8 бит)

loop\_ass:

bt eax, 0 ; заносит во флаг CF = 1, если была 1, 0 если 0 (в регистрах флаг отображается как CY) анализирует младший бит переменной eax

jc one ; перенос на one , если CF = 1

jnc zero ; перенос на zero , если CF = 0

one:

ADD summ, 1

zero:

sar eax, 1 ; арифметический побитовый сдвиг (откидывает мл. бит, сохраняя старшие) вправо на 1 бит

loop loop\_ass

; 00101100 10001010 , должно насчитать 3 единицы

bt summ, 0 ; заносит во флаг CF = 1, если была 1, 0 если 0 (в регистрах флаг отображается как CY) анализирует младший бит переменной summ

jc odin ; перенос на odin , если CF = 1 => нечетное

jnc nol ; перенос на nol , если CF = 0 => четное

odin:

mov ax, m

and ax, 0F1F1h

jmp point ; точка для переноса, дабы не заходил в точку nol

nol:

mov ax, m

ror ax, 6 ; циклический сдвиг вправо 6 раз

point:

; 3 задание , cf - флаг переноса

mov m, ax ; теперь в переменной m хранится значение R

jc first

jnc second

function\_first PROC ; начало функции

ADD m, 1

ret ; return

function\_first ENDP ; конец функции

function\_second PROC ; начало функции

or m, 1021

ret ; return

function\_second ENDP ; конец функции

first:

call function\_first

jmp point\_2 ; ; точка для переноса, дабы не заходил в точку second

second:

call function\_second

point\_2:

exit:

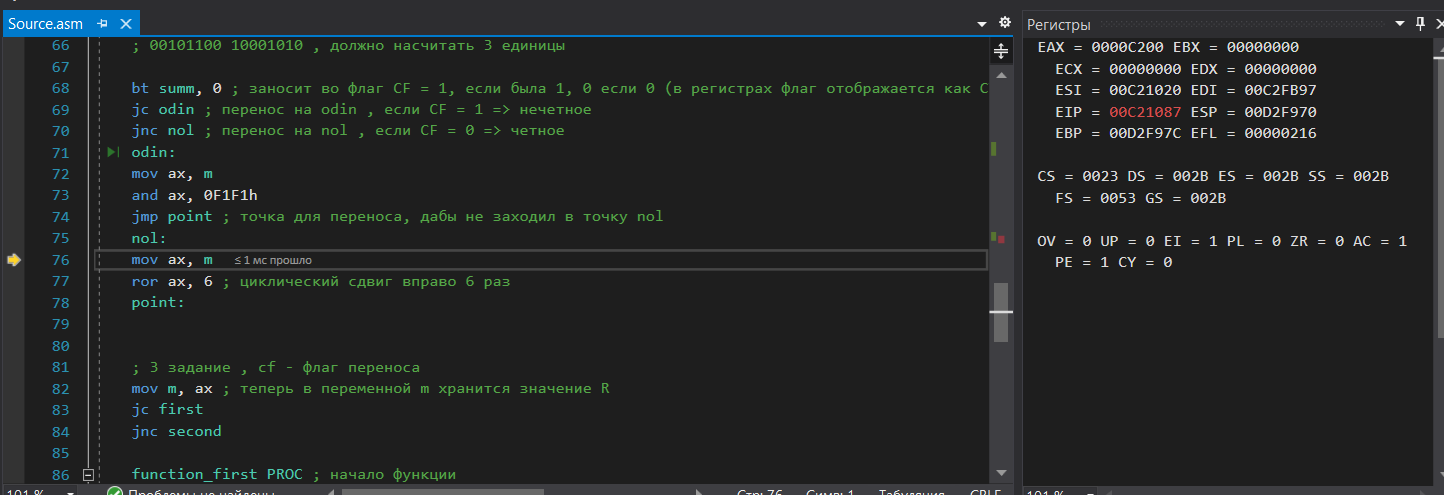
Invoke ExitProcess,1

End Start

Вывод:

Min число x при котором изменяется ветвление 2 задания: 0





Зашло в другое ветвление

В данной работе я изучил принципы выполнения команд ветвления и организации циклов и подпрограмм микропроцессоров с архитектурой х86 в MASM.

* .686 – данная директива указывает ассемблеру использовать набор команд процессора Pentium Pro или Pentium II.
* .model flat - это несегментированная модель памяти, используемая в 32-разрядных операционных системах. В этой модели все данные и код программы размещаются в одном логическом адресном пространстве, что упрощает адресацию памяти. Сегментные регистры не используются для адресации данных, а вместо них используются 32-разрядные смещения
* .model stdcall - это соглашение о вызовах процедур, которое определяет порядок передачи параметров и очистки стека. При использовании stdcall, параметры передаются через стек в обратном порядке, а очистка стека производится вызываемой процедурой
* .stack100h – данная директива определяет размер стека программы. В данном случае, стек будет иметь размер 256 байт (100h — это шестнадцатеричное представление 256 в десятичной системе)
* ExitProcess PROTO STDCALL: DWORD:
  + ExitProcess:
    - Это имя функции из Windows API, которая завершает выполнение текущего процесса.
    - Она принимает один параметр — код завершения процесса (тип DWORD).
  + PROTO:
    - Ключевое слово MASM для объявления прототипа функции.
    - Прототип позволяет ассемблеру знать, как вызывать функцию, какие параметры она принимает и в каком порядке.
  + STDCALL:
    - Указывает соглашение о вызове функции.
    - В соглашении stdcall параметры передаются через стек в обратном порядке (сначала последний параметр), а очистка стека выполняется вызываемой функцией.
    - Это стандартное соглашение для большинства функций Windows API.
  + DWORD:
    - Описывает тип параметра функции. В данном случае функция принимает один параметр типа DWORD (32-битное беззнаковое целое число).
* API (Application Programming Interface) — это набор способов и правил, по которым различные программы общаются между собой и обмениваются данными.