Lecture 4: Branching and Looping

Table of contents

1 Objectives **2**

2 Branching **2**

2.1 Instruction Pointer 2
2.1.1 Relative vs Absolute Offset Address 3
2.2 `JMP` Instruction 4
2.2.1 Types of Jumps: 4
2.2.2 Example 5
2.3 `LOOP` Instruction 5
2.3.1 More Examples 7
2.3.2 Nested Loop 9
2.4 `TEST` Instruction 10
2.5 `CMP` Instruction 11
2.6 `Jcc` Instruction (Conditional Jump) 13
2.6.1 Types of Conditional Jump 14
2.6.2 Jump Based on Specific Flag Values 15
2.6.3 Jump Based on Equality or The Value of CX/ECX/RCX . 15
2.6.4 Jump Based on Comparisons of Unsigned Operands . 15
2.6.5 Jump Based on Comparisons of Signed Operands . . . 16
2.6.6 Affected Flags 16

2.6.7 Examples 16

3 Implementing High-Level Flow Structures 17
3.1 Implementing IF statements 18

1 Objectives

1. Understand relative and absolute offset addresses
2. Learn all instructions provided by the Intel Microprocessors that you need for decision-making logic.
3. Learn how to translate IF statement, switch statement, and conditional loop found in high-level languages into assembly language.

2 Branching

- By default, the CPU loads and executes programs sequentially.
- A branch, or transfer of control, is a way of altering the order in which statements are executed.
- There are two basic types of transfer:
    1. **Unconditional transfer**, in which a transfer is occurred unconditionally.
    2. **Conditional transfer**, in which a transfer is occurred based on a certain condition.

2.1 Instruction Pointer

- The EIP, or instruction pointer , register contains the address of the next instruction to be executed. Certain machine instructions manipulate EIP, causing the program to *branch* to a new location. These instructions are JMP, Jcc, LOOP, CALL, RET, or IRET.

2.1.1 Relative vs Absolute Offset Address

- The EIP can be updated based on a target address.
- The target address, within the instruction stream, can be **relative offset** or **absolute offset**
- A **relative offset** is a signed *displacement* to the current value of the EIP.

$$\text{EIP} = \text{EIP} + \text{Target displacement}$$

- An **absolute address** is an offset from the base of the code segment (i.,e., offset from address 0 of a segment).

$$\text{EIP} = \text{Target Address}$$

- Relative address can be specified in either of the following ways

Address format	Description
`rel8`	A constant value in the range from -128 to 127 bytes.
`rel16`	A relative address within the same code segment.
`rel32`	A relative address within the same code segment.

> **Note**
>
> A relative offset (`rel8`, `rel16`, or `rel32`) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value.

- Absolute address can be specified as an address in general-purpose register or an address specified using memory addressing mode.

2.2 JMP Instruction

Syntax

```
        JMP    <destination>
```

- The JMP instruction causes an unconditional transfer to a destination address.
- The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

  In this course, we only consider the immediate value for the destination operand.

- Thus, the JMP instruction takes the following types (for this course):

```
JMP    rel8          ; relative address
JMP    rel16         ; relative address (not supported in 64-b
JMP    rel32         ; relative address
JMP    reg/mem16     ; absolute address (not supported in 64-b
JMP    reg/mem32     ; aboslute address (not supported in 64-b
```

2.2.1 Types of Jumps:

1. **Near Jump**: A jump to instruction within the current code segment (intra-segment jump). Here the displacement is either `rel16`, `rel32`, `reg/mem16`, `reg/mem32`.
2. **Short jump**: A near jump where the jump is limited to -128 to +127 from the current EIP value. Here the displacement is `rel8`.
3. **Far Jump**: A jump to an instruction located in a different segment than the current code segment but at the same privilege level (inter-segment jump) (NOT covered).

Note

There is no difference in the coding for a relative short jump and for a relative near jump. The assembler uses a short jump if the displacement is within the small range in order to generate more compact code. A near jump is used automatically if the displacement is more than 128 bytes away.

2.2.2 Example

```
        mov   ax, 0
top:
        mov   bx, 5
        add   bx, 1
        .
        .
        .
        jmp   top
```

2.3 LOOP Instruction

Syntax

```
        LOOP    <destination>
```

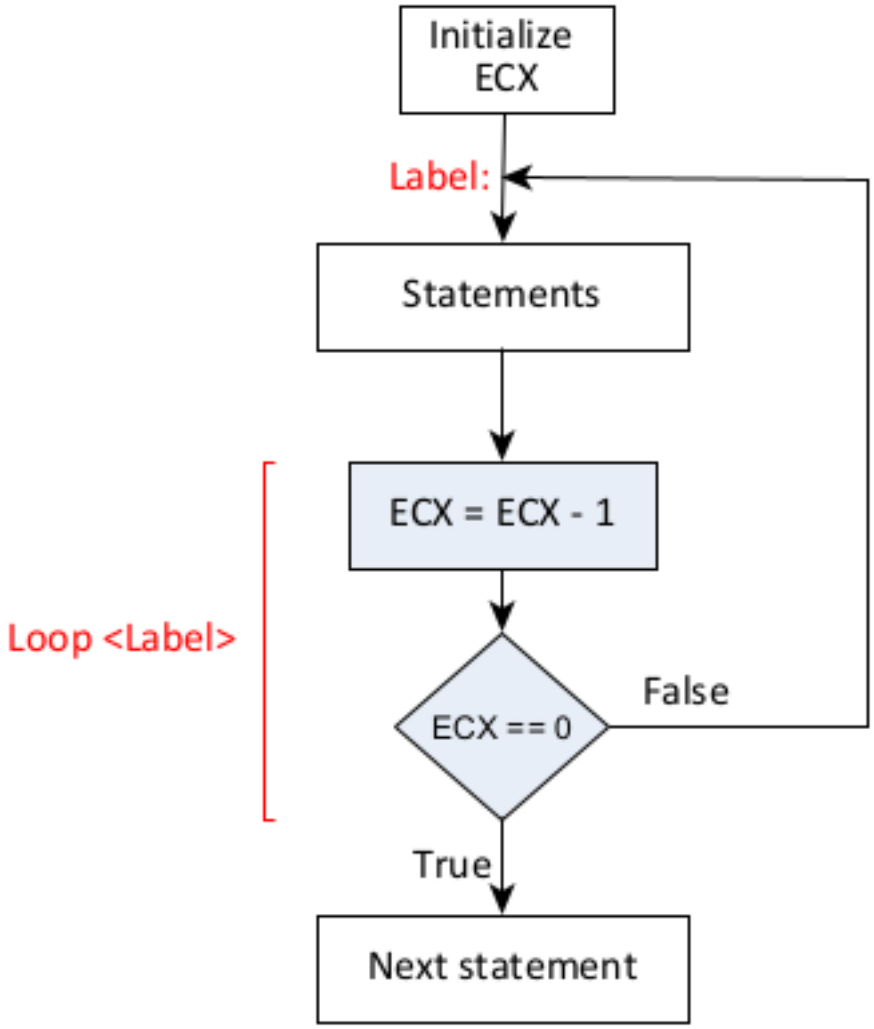
Rule

- The destination operand must be `rel8`.

- The loop instruction repeats a block of statements a specific number of times (determined by `ECX` register).

- ECX is automatically used as a counter and is decremented each time the loop repeats.
- Example

```
        mov   ax, 1
        mov   ecx, 5
top:
        add   ax, ax
        loop  top
```



2.3.1 More Examples

Example 2.1. Write an assembly program to compute the sum of the following series

$$S = 1 + 2 + 4 + 8 + \cdots + 1024$$

Solution:

Here, we can re-write the series as follows:

$$S = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{10}$$

Therefore, we can set ECX to 10. However, setting ECX to 10 implies that the loop starts from 10 and ends at 1. This also implies that 2^0 won't be added. To handle it, we can add 2^0 when we initialize the sum S.

```
        section .data
varS:   dd      0

        section .text
_main:
        mov     eax, 1          ; first elm is 1
        add     [varS], eax     ; add 1 to S
        mov     ecx, 10         ; set loop counter to 10
top:
        add     eax, eax        ; set eax = 2 * eax
        add     [varS], eax     ; add eax to S
        loop    top

        ret
```

In the previous program, we are accessing a memory location in each iteration. Therefore, if the number of iterations is quite high, then we have too many memory accesses.
Most CPUs have a cache memory, which is a smaller, faster memory and

located closer to the processor core.
When accessing the memory location `varS` for the first time, the content is stored into the main memory (RAM) (cache miss) and into the cache memory. However, in the subsequent memory accesses, the CPU is accessing the cache memory rather than the main memory (we call it temporal locality).
To avoid too many memory accesses, we can store the sum in a CPU register and then we update `varS` after the loop has been completely executed. This will definitely improve the program performance (in terms of time)

Another solution:

```
        section .bss
varS:   resd    1

        section .text
_main:
        ; we store the sum in EBX register
        mov     eax, 1          ; first elm is 1
        mov     ebx, eax        ; set sum (ebx) to 1
        mov     ecx, 10         ; set loop counter to 10
top:
        add     eax, eax        ; set eax = 2 * eax
        add     ebx, eax        ; add eax to sum (ebx)
        loop    top

        mov     [varS], ebx     ; store the result

        ret
```

Example 2.2. Suppose you have the following signed 16-bit integer array hard-coded in your program:

```
arrX[] = {13, 22, 18, 10, 25}
```

Write an assembly program to store the above array in the data section and then calculate the sum of its elements. Store the sum in `sum` variable.

Solution:

```
        section .data
arrX:   dw      13, 22, 18, 10, 25
ARRSIZ  equ     ($ - arrX) / 2
sum:    dw      0

        section .text
_main:
        mov     edi, arrX       ; edi points to 1st elm
        mov     ecx, ARRSIZ     ; set counter
        mov     ax, 0           ; set sum to 0
L1:
        add     ax, [edi]       ; load next elm and add it to sum
        add     edi, 2          ; set edi to point to next elm
        loop    L1              ; loop

        mov     [sum], ax         ; store sum
        ret
```

2.3.2 Nested Loop

- How to implement the following nested loop:

```
for (int i=100; i != 0; i--)
  for (int j=50; j != 0; j--)
    ...
```

  in assembly language?

- Unfortunately, the `LOOP` instruction has only *one counter*, which is `ECX`.
- To tackle this problem, we must preserve the outer loop counter before executing the inner loop, and restore the outer loop counter before executing the outer's `LOOP` instruction. Here is an a solution:

```
        section .bss
count:  resd    1           ; to store outer loop counter

        section .text

        .
        .
        mov     ecx, 100    ; # of iterations for outer loop
OUTER:
        mov     [count], ecx    ; preserve outer loop counter
        mov     ecx, 50     ; # of iterations for inner loop
INNER:
        .
        .
        loop    INNER
        mov     ecx, [count]
        loop    OUTER
```

- In future lecture, we can enhance this solution by using ***run-time stack***.

2.4 TEST Instruction

Syntax

```
        TEST    <destination>, <source>
```

- TEST instruction performs an ***implied*** AND operation of two operands and set the SF, ZF, and PF flags according to the results.
- The flags can then be tested by the conditional jump or loop instruction.
- TEST instruction differs from the AND instruction in that it does not alter either of the operands

Rules

- Both operands must be of the same size
- Both operands cannot be memory locations
- Destination operand cannot be immediate

2.5 CMP Instruction

Syntax

```
CMP     <destination>, <source>
```

- CMP instruction is used to compare between two integral values (integers or characters).
- It performs an ***implied*** subtraction of a source operand from a destination operand.
- It affects ALL status flags according to the value the destination would have had if actual subtraction had taken place. Through these flags, we can compare the two operands:
  - If the two operands are treated as unsigned, then

ZF	CF	Comparison Result
0	0	destination > source
0	1	destination < source
1	*	destination = source

  - If the two operands are treated as signed, then

ZF	SF	OF	Comparison Result
0	0	0	destination > source

ZF	SF	OF	Comparison Result
0	0	1	destination < source
0	1	0	destination < source
0	1	1	destination > source
1	*	*	destination = source

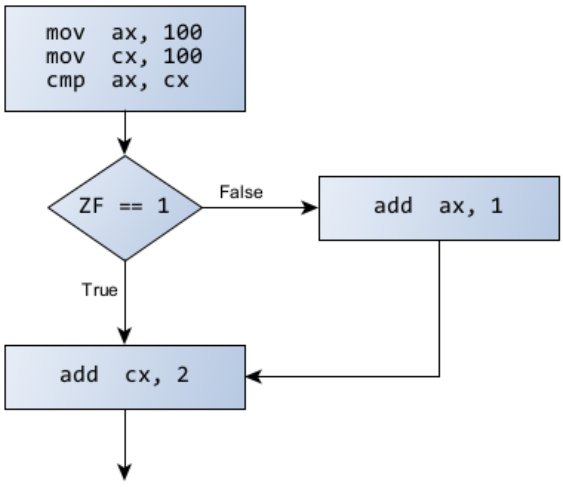
- `CMP` is a valuable tool for creating conditional logic structures. When you follow CMP with a conditional jump instruction, the result is the assembly language equivalent of a flow control statement (such as `if`, `if else`, `switch case`, `while`, `do while`).
- Example:

```
      mov   ax, 100
      mov   cx, 100
      cmp   ax, cx      ; ZF = 1

      jz    skip        ; JUMP to label "skip" only if ZF is set

      add   ax, 1
skip:
      add   cx, 2
```

- The flow control of the previous code is shown below:



2.6 `Jcc` Instruction (Conditional Jump)

Syntax

```
        Jcc    <destination>
```

- The `Jcc` instruction causes a ***conditional transfer*** to a destination address, which can be specified relatively or absolutely.
- The `Jcc` instruction supports near jump only.
- A conditional jump instruction ***branches*** to a destination label when a condition is true. Otherwise, if the condition is false, the instruction immediately following the conditional jump is executed.

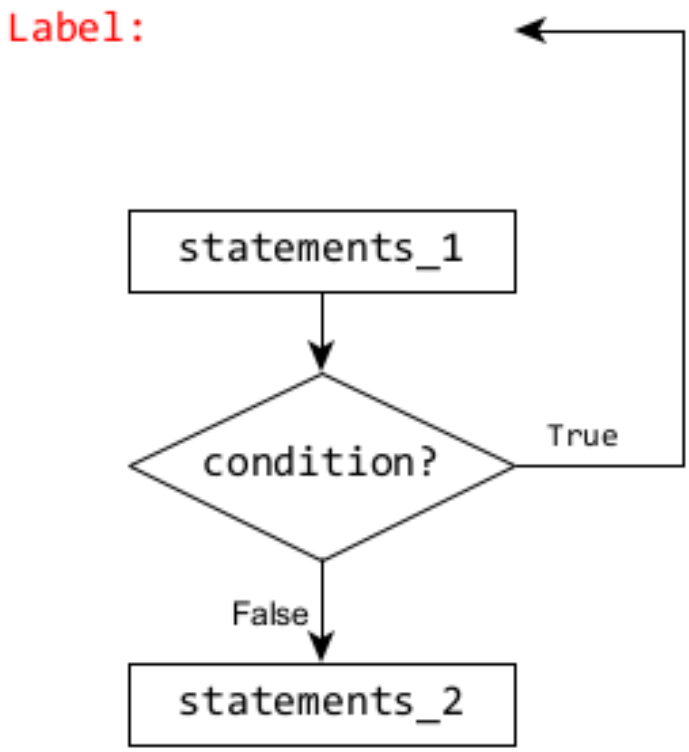


Figure 1: Backward Reference

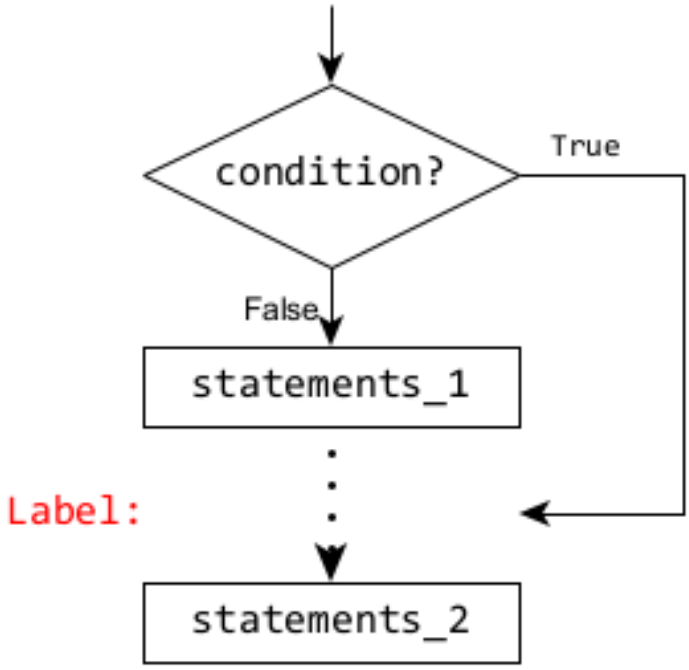


Figure 2: Forward Reference

2.6.1 Types of Conditional Jump

- The x86 instruction set has a large number of conditional jump instructions. They are able to compare signed and unsigned integers and perform actions based on the values of individual CPU flags.
- The conditional jump instructions can be divided into four groups:

  1. Jumps based on specific flag values
  2. Jumps based on equality between operands or the value of RCX, ECX, or CX
  3. Jumps based on comparisons of unsigned operands
  4. Jumps based on comparisons of signed operands

2.6.2 Jump Based on Specific Flag Values

Mnemonic	Description	Flags/Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP or JPE	Jump if parity (even)	PF = 1
JNP or JPO	Jump if not parity (odd)	PF = 0

2.6.3 Jump Based on Equality or The Value of CX/ECX/RCX

Mnemonic	Description
JE	Jump if equal
JNE	Jump if not equal
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX = 0 (64-bit mode)

2.6.4 Jump Based on Comparisons of Unsigned Operands

Mnemonic	Description	Flags
JA	Jump if above	CF = 0 and ZF = 0
JNBE	Jump if not below of equal (same as JA)	CF = 0 and ZF = 0
JAE	Jump if above or equal	CF = 0
JNB	Jump if not below (same as JAE)	CF = 0
JB	Jump if below	CF = 1 and ZF = 0

Mnemonic	Description	Flags
JNAE	Jump if not above or equal (same as JB)	CF = 1 and ZF = 0
JBE	Jump if below of equal	CF = 1 or ZF = 1
JNA	Jump if not above (same as JBE)	CF = 1 or ZF = 1

2.6.5 Jump Based on Comparisons of Signed Operands

Mnemonic	Description	Flags
JG	Jump if greater	ZF=0, SF=OF
JNLE	Jump if not less or equal (same as JG)	ZF=0, SF=OF
JGE	Jump if greater or equal	SF=OF
JNL	Jump if not less (same as JGE)	SF=OF
JL	Jump if less	SF != OF
JNGE	Jump if not greater or equal (same as JL)	SF != OF
JLE	Jump if less or equal	ZF=1 or SF != OF
JNG	Jump if not greater (same as JLE)	ZF=1 or SF != OF

2.6.6 Affected Flags

- No conditional jump instruction changes any flag value.

2.6.7 Examples

Example 2.3.

```
    mov   edx, -1
    cmp   edx, 0
    jnl   L5       ; jump if EDX .nl. 0 (false) => no jump
    jnle  L5       ; jump if EDX .nle. 0 (false) => no jump
    jl    L1       ; jump if EDX < 0 (true) => jump to L1
```

Example 2.4.

```
      mov   bx, 35
      cmp   bx, 32
      jb    L4      ; jump if BX < 32 (false) => no jump
      jbe   L4      ; jump if BX <= 32 (false) => no jump
      jae   L3      ; jump if BX >= 32 (true) => jump to L3
```

Example 2.5.

```
      ; repeat 10 times
      mov   ecx, 0
top:

      ; loop statement

      inc   ecx
      cmp   ecx, 10
      jb    top       ; jump if ECX < 10
```

3 Implementing High-Level Flow Structures

In this section, you will learn how to translate:

- if statement
- if-else statement
- switch statement
- for loop
- while loop
- do while loop

into assembly language.

3.1 Implementing IF statements

Example 3.1. Translate the following `if` statement into assembly language. Assume the variable `x` is `unsigned int`.

```
if (x > 2000)
  x = x/2;
```

Solution:

```
    mov   eax, [x]
    cmp   eax, 2000
    jbe   ENDIF
    mov   edx, 0
    mov   ebx, 2
    div   ebx
    mov   [x], eax
ENDIF:
```

Example 3.2. Let us translate the following `if` statement, assuming the variables are declared as `int`:

```
if (value < 10)
  smallCount++;
else
  largeCount++;
```

Solution:

```
    mov   eax, [value]
    cmp   eax, 10
    jnl   ELSE

    inc   DWORD [smallCount]
```

```
    jmp   ENDIF
ELSE:
    inc   DWORD [largeCount]
ENDIF:
```

Example 3.3. Translate the following statement (all variables are declared as `int`):

```
if ( (total >= 100) and (count = 10) )
  total += value;
```

Solution:

```
    mov   eax, [total]
    cmp   eax, 100
    jl    ENDIF
    cmp   DWORD [count], 10
    jnq   ENDIF

    add   eax, [value]
    mov   [total], eax
ENDIF:
```

Example 3.4. Translate the following statement (all variables are declared as `int`):

```
if ( (total >= 100) or (count = 10) )
  total += value;
```

Solution:

```
    mov   eax, [total]
    cmp   eax, 100
    jge   THEN
    cmp   DWORD [count], 10
    jnq   ENDIF
THEN:
    add   eax, [value]
    mov   [total, eax]
ENDIF:
```

Example 3.5. Find the smallest of three integers:

```
if (a <= b and a <= c)
  smallest = a;
else if (b <= c)
  smallest = b;
else
  smallest = c;
```

Solution:

```
    mov   eax, [a]
    mov   ebx, [b]
    mov   ecx, [c]
    cmp   eax, ebx
    jg    ELSE1
    cmp   eax, ecx
    jg    ELSE1

    mov   [smallest], eax
    jmp   ENDIF

ELSE1:
    cmp   ebx, ecx
```

```
    jg    ELSE2

    mov   [smallest], ebx
    jmp   ENDIF

ELSE2:
    mov   [smallest], ecx

ENDIF:
```

Example 3.6. Translate the following `switch` statement:

```
switch (x) {
case 0:
  y = 1;
  break;
case 2:
  y = 10;
  break;
case 4:
  y = 100;
  break;
default:
  y = 0;
}
```

Solution:

```
      mov   eax, [x]
      cmp   eax, 0
      jnq   CASE2

      mov   ebx, 1
      jmp   ENDCASE
CASE2:
```

```
      cmp   eax, 2
      jnq   CASE3

      mov   ebx, 10
      jmp   ENDCASE
CASE3:
      cmp   eax, 4
      jnq   CASE4

      mov   ebx, 100
      jmp   ENDCASE
CASE4:
      mov   ebx, 0
ENDCASE:
      mov   [y], ebx
```

Example 3.7. Translate the following `for` loop:

```
sum = 0;
for (int i=1; i<100; i+=2)
  sum += i;
```

Solution:

```
    mov   eax, 0
    mov   ecx, 1
top:
    add   eax, ecx
    add   ecx, 2
    cmp   ecx, 100
    jl    top
    mov   [sum], eax
```

Example 3.8. Translate the following `while` loop:

```
d1 = 100;
d2 = 0;
t = 0;
while (d2 <= d1) {
  t = t + 1;
  d1 = d1 - 3;
  d2 = 2*t - 1;
}
```

Solution:

```
    mov   eax, 100
    mov   ebx, 0
    mov   ecx, 0;
WHILE:
    cmp   ebx, eax
    jg    ENDWHILE

    inc   ecx
    sub   eax, 3
    imul  ebx, ecx, 2
    sub   ebx, 1
    jmp   WHILE
ENDWHILE
    mov   [d1], eax
    mov   [d2], ebx
    mov   [t], ecx
```

Example 3.9. Translate the following `do while` loop:

```
count = 1;
sum = 0;
do {
  sum = sum / 2 + count * 2
  count++;
} while (sum < 1000 or count <= 100);
```

Solution:

```
    mov   ecx, 1;
    mov   eax, 0
DOWHILE:
    cdq
    mov   ebx, 2
    idiv  ebx
    add   eax, ecx
    add   eax, ecx
    inc   ecx

    cmp   eax, 1000
    jl    DOWHILE
    cmp   ecx, 100
    jle   DOWHILE

    mov   [count], ecx
    mov   [sum], eax
```