Санкт-Петербургский политехнический университет
Петра Великого

А.В. Жуков

Программирование микропроцессора i8080

Дополнение к циклу лабораторных работ
по курсу "Архитектура ЭВМ"

Санкт-Петербург
2021

Жуков А.В. Программирование микропроцессора i8080 / Дополнение к циклу лабораторных работ по курсу “Архитектура ЭВМ”, 2021. 11 с.

Лабораторная работа по программированию на ассемблере, в дополнение к циклу работ по курсу “Архитектура ЭВМ”, адресованному бакалаврам по направлению "Информатика и вычислительная техника" — 09.03.01. Программная модель, способы адресации и система команд микропроцессора i8080 рассмотрены по ходу разбора примеров простых программ. В качестве среды разработки используется сетевой сервис www.asm80.com.

Стр. 11, библиогр. — 3 назв.

Основы программирования микропроцессора i8080

Процессор i8080, выпущенный в 1975 г., предназначался для обработки 8-битных данных в 16-битном адресном пространстве[1]. Имеет восемь 8-битных регистров: A (accumulator), B, C, D, E, F (flags)[2], H (high), L (low). Для адресации 64 кБайт памяти имеется пять 16-битных регистров: PC (указатель текущей инструкции), SP (указатель вершины стека) и регистровые пары B/C, D/E и H/L, составленные из перечисленных 8-битных регистров.

Рассмотрим пример использования регистров. В качестве инструментальной среды используем сервис на странице www.asm80.com.

Откройте страницу и введите в рабочей области[3]:

```
mvi d,0
mvi e,10h

ldax d
inr e
ldax d
hlt

org 10h
db 55h, -2
```

Нажмите кнопку **Save** справа и задайте имя с расширением a80[4], например, x.a80. Выполните **Compile** (F9) и, если все удачно, **Format**. Код после форматирования:

```
        MVI     d,0
        MVI     e,10h

        LDAX    d
        INR     e
        LDAX    d
        HLT

        .ORG    10h
        DB      55h,-2
```

Текст надо перенести (Copy-Paste) к себе и сохранить[5].

Вызовите **Emulator** (F10). Код теперь выглядит так:

```
0000   16 00                 MVI   d,0
0002   1E 10                 MVI   e,10h
0004   1A                    LDAX   d
0005   1C                    INR   e
0006   1A                    LDAX   d
0007   76                    HLT
0010                         .ORG   10h
0010   55 FE                 DB   55h,-2
```

В окне слева видны: память, регистры (все в hex-формате) и кнопки для отладки. Код виден по адресам 0–9, а данные (55h, 0feh) — с адреса 10h, согласно директиве ORG. Сейчас

---

[1] На i8080 был сделан первый персональный компьютер Altair 8800. Без консоли, имея 256 байт памяти. он программировался в машинных кодах при помощи тумблеров. Потом для полноценных машин на базе i8080/85 (и, впоследствии, Zilog Z80) была разработана ОС CP/M, а для нее — множество прикладных программ.

[2] Регистр F (flags) хранит *байт* состояния с признаками Z, S, P, C, AC. Имя PSW (processor status *word*), допустимое только в командах PUSH PSW и POP PSW, обозначает пару A/F.

[3] Объявление наверху можно стереть.

[4] Это важно, т.к. вариант ассемблера выбирается по расширению.

[5] Файлы, сохраненные online, находятся в кэш-памяти браузера — хранилище ненадежное.

PC = 0. Первая команда MVI (MoVe Immediate) должна записать число (0) в 8-битный регистр (D). Прежде чем выполнить MVI, введите в поле D не ноль, чтобы можно было оценить действие команды. Дважды выполните **Single Step** (F8). Теперь в паре D/E — адрес данных: старшая часть в D, младшая в E.

Следующая команда — LDAX d (LoaD Accumulator eXtended) — загружает в регистр-аккумулятор (A) байт из памяти; его адрес задан парой D/E. Буква X в команде всегда означает, что используется 16-битный регистр: PC, SP или пара B/C, D/E, H/L.

В результате первой команды LDAX в A запишется 55h — копия памяти по адресу 10h. Дальше — инкремент 8-битного регистра E, где сейчас младшая часть адреса данных. Вторая команда LDAX запишет в A число 0feh — копию памяти по адресу 11h.

Пройдите по шагам до команды HLT (остановка процессора).

Недостаток этой программы в том, что старшая часть адреса в D/E всегда равна 0 и поэтому весь AREA должен размещаться в первых 256 байтах памяти (в адресах 0–0ffh).

Для выхода из симулятора нажмите **F10**, затем исправьте число 10h на 0ffh в двух местах: в ORG и во второй команде MVI. Данные теперь размещены с адреса 0ffh. После компиляции пройдите программу по шагам. Какое число прочтет вторая команда LDAX, и откуда оно взялось?

Примечание: чтобы увидеть ячейки по адресам 00ff и 0100, надо либо покрутить колесо мыши над областью **Memory**, либо ввести ff в поле **Go there!**

Вот вариант, не привязанный к 256-байтной странице памяти:

```
            LXI     d,area

            LDAX    d
            INX     d
            LDAX    d
            HLT

            .ORG    0ffh
AREA:       DB      55h,-2
```

Команда LXI (Load eXtended Immediate) — это загрузка числа в регистровую пару. В данном случае число 00ff (адрес массива AREA) загружается в пару D/E. Команда INX D увеличивает на 1 двухбайтное слово D/E, учитывая возможный перенос из E в D. Перед ее выполнением D = 0, E = 0ffh, а после — D = 1, E = 0. Убедитесь, что вторая команда LDAX считывает 0feh — второй элемент массива AREA.

Массив у нас всего из двух элементов, и проще было бы прочесть AREA *напрямую*:

```
            LDA     area
            LDA     area+1
            HLT

            .ORG    0ffh
AREA:       DB      55h,-2
```

Команда LDA, в отличие от LDAX, использует фиксированный адрес памяти (00ff в первой команде, 0100 во второй); результат чтения попадает в аккумулятор (A). Этот двухбайтный адрес в машинном коде следует за командным байтом.

Для записи в память используем команды STA m8 (STore Accumulator) и STAX r16 (STore Accumulator eXtended) — с прямой и косвенной адресацией соответственно[6]. Чтобы

[6] Во многих командах i8080 регистр A не задан явно, но подразумевается. Сокращения: m8 — 8-bit memory; r16 — 16-bit register.

перед записью задать число в аккумуляторе, применяем уже знакомую команду MVI. Запрограммируйте запись в AREA чисел 0aah и 1 напрямую.

Определим по адресу AREA массив из 8 байт и заполним его шахматным кодом, используя косвенную адресацию в цикле:

```
            LXI     d,area
            MVI     a,55h       ; or 01010101b, or 125q
            MVI     c,size      ; counter
M1:
            STAX    d           ; A -> [D/E]

            INX     d           ; increment D/E
            DCR     c           ; decrement counter
            JNZ     m1          ; jump if not zero

            HLT

            .ORG    0ffh
AREA:       DS      8           ; Define Sequence, length 8
SIZE        EQU     $-area
```

В i8080 нет специальной команды для организации счетного цикла, как loop в i80x86. Поэтому используем DCR с последующим переходом JNZ. Команды INX-DCR-JNZ выполняют в цикле служебные функции: продвижение указателя, уменьшение и проверку счетчика итераций.

Примечания:

- Пройдя одну итерацию по шагам **F8**, выполните затем **Animate** (F12)[7].
- Директивы распределения памяти: **db** (define bytes), **dw** (define 16-bit words), **ds** (define sequence); в **ds** указывается *количество* байт, а в **db**/**dw** — *значения* через запятую.
- Знак **$** — это счетчик адресов при трансляции, а при выполнении — это PC (Program Counter). Директива ORG 0ffh устанавливает $ = 0ffh, а выражение $ - AREA дает расстояние в байтах между текущей точкой и началом AREA.

После проверки сохраните программу.

Теперь закодируйте инвертирование элементов массива, заданного так:

```
AREA:       DB      55h, 0aah, 0feh, 1, 0fh, 0f0h, 0, 0ffh
```

В каждом цикле надо: прочесть в A очередной элемент массива, выполнить команду CMA (CoMplement Accumulator) и записать A обратно в память. За основу возьмите предыдущую программу; достаточно что-то сделать с выделенными строками — и только. После проверки сохраните программу.

Как вы заметили, обмен с памятью в рассмотренных примерах всегда байтовый. Таково ограничение i8080 при доступе к памяти. Но при работе со стеком, наоборот, обмен всегда 16-битный. Операндом PUSH/POP может быть только пара B/C, D/E, H/L или A/F (flags)[8].

Интересно, что PUSH и POP — самые быстрые из команд i8080, обращающихся к памяти. Под впечатлением от этой диковины я придумал пример того, как не надо программировать доступ к массиву: при помощи PUSH (запись) или POP (чтение). Но в i8080 этот трюк дает двукратный выигрыш в скорости, в чем мы сейчас убедимся.

Восстановите предыдущий пример: заполнение AREA кодом 55h. Пройдя по шагам до входа в цикл (до STAX), запомните значение T (число затраченных машинных тактов).

[7] Только не RUN. В нем ошибка: вы проскочите HLT и затормозить сможете только кнопкой STOP.

[8] A и F объединяются в пару *только* при выполнении PUSH PSW и POP PSW.

Пройдите программу до HLT[9] и оцените прирост T. Затем выполните аналогичный тест для другого решения, использующего PUSH вместо STAX. Вот оно:

```
                DI                      ; disable hardware interrupts!
                LXI     sp,area_end     ; move stack to the end of array
                LXI     d,5555h         ; D/E := word to fill the array
                MVI     c,size/2        ; number of words
M1:
                PUSH    d               ; SP := SP - 2; D/E -> (SP)
                DCR     c
                JNZ     m1

                LXI      sp,0       ; move stack back to initial location
                EI                  ; enable hardware interrupts
                HLT

                .ORG    0ffh
AREA:           DS      8
SIZE            EQU     $-area
AREA_END:
```

Мы совместили стек с массивом AREA и командой PUSH *записали* в него 16-битное число из пары D/E, причем в *обратном* порядке, поскольку эта команда автоматически *уменьшает* SP. Вариант с PUSH должен работать быстрей, т.к. эта команда не только опережает по скорости пару (!) STAX-INX, но и пишет сразу по два байта.

Примечания:

- Так как стек используется при обработке прерываний, его перенос допустим только при запрещенных прерываниях (DI).
- Если в стеке что-то есть (например, мы вошли в подпрограмму), то перемещать его уже нельзя — мы не сможем восстановить SP, т.к. в i8080 нет команды чтения этого регистра. Можем только установить SP в исходное (стартовое) значение.
- Команда POP годится для *чтения* массива 16-битных слов в *прямом* направлении.

Вернемся к программе, которая инвертирует байты в AREA. В инверсии участвуют аккумулятор (A) и команда CMA (CoMplement Accumulator). Заметим: все арифметико-логические вычисления, кроме инкремента-декремента [10], обязательно используют A. Арифметика — это сложение, вычитание и сравнение (умножения и деления нет), а логические команды — это "и", "или" и "исключающее или", выполняемые над парами одноименных бит. Эти команды устанавливают все признаки в регистре F: **s** (sign), **z** (zero), **a** (auxilary carry[11]), **p** (parity) и **c** (carry).

Проверим некоторые из этих команд. Эмулятор показывает флаги внизу в виде строки из восьми литер, где заглавная буква означает 1. Например, **sZ0a0P1c** говорит о том, что флаг **s** — это 7-й бит в F и он равен 0, флаг **z** занимает 6-й бит и равен 1, а 5-й бит всегда 0 и т. д.

```
                XRA     a       ; A = 0, z = p = 1 (c = 0)
                ADI     0feh    ; A = 0fe, s = 1
                CPI     80h     ; c = 0 (80h is not above A)
                JC      above1
                NOP
above1:
```

[9] Режим RUN не работает, пользоваться им нельзя. **Animate** (F12) всем бы хорош, но при попадании на HLT счетчик T продолжается увеличиваться. Чтобы T остановился, надо поставить *точку останова* на HLT. Для этого, запомнив адрес команды, нажать **Breakpoints**, вписать адрес в графу слева от кнопки **+break** и нажать ее. Либо, без всех этих ухищрений, ползти по шагам до HLT.

[10] Инкремент (INR) и декремент (DCR) допустимы с *любым* 8-битным регистром, не обязательно A.

[11] Используется при обработке BCD (binary coded decimal). В i8080 предусмотрена только одна команда на эту тему — DAA (коррекция после сложения).

```
            CPI     0ffh      ; c = 1 (0ffh is above A)
            JC      above2
            NOP
above2:
            ADI     3         ; A = 1, c = 1
            HLT
```

Первая команда — поразрядное "исключающее или" над A и указанным регистром (A). В таком варианте XRA используется для обнуления и регистра A, и флага C[12].

Команда ADI задает сложение; I (Immediate) означает непосредственный операнд (число или метка). Результат накапливается в аккумуляторе.

Команда CPI (ComPare Immediate) — сравнение A с числом. Сравнение выполняется как вычитание копии A и числа. Если вычитаемое число больше A, то возникает перенос: C = 1. То есть, последующая команда JC выполнит переход, если число > аккумулятора.

Итак, все *арифметические* команды воздействуют на флаги. Логические — тоже, за одним исключением: инверсия (CMA) их не затрагивает. Что касается инструкций INR и DCR, они (как в i80x86) воздействуют на S, Z и P, но не влияют на флаг C[13].

Команды 16-битного инкремента и декремента INX/DCX флагов не касаются[14]. В следующем примере (сложение с многократной точностью [1, стр. 196]) на флаг C в теле цикла воздействует *только* команда ADC.

```
            LXI     h,ay1     ; H/L <- ptr to 1-st array
            LXI     d,ay2     ; D/E <- ptr to 2-nd array
            MVI     b,szays   ; length in bytes
            CALL    mpbadd
            HLT

MPBADD:     MOV     a,b
            ANA     a         ; clear C, check length (Z = 0 or Z = 1?)
            RZ                ; return from procedure if length = 0
LOOP:       LDAX    d         ; read byte from ay2 into A
            ADC     m         ; A := A + (H/L) + flag C
            MOV     m,a       ; store part of sum: A -> (H/L)

            INX     h         ; ptr1
            INX     d         ; ptr2
            DCR     b         ; counter
            JNZ     loop
            RET               ; return from procedure

            ORG     40h
AY2:        DB      067h,045h,023h,1       ; 01_23_45_67h   (19088743)
AY1:        DB      0efh,0cdh,0abh,089h    ; 89_ab_cd_efh (2309737967)
                                ; Result = 8a_cf_13_56h (2328826710)
SZAYS       EQU     $-Ay1
```

Здесь есть несколько незнакомых обозначений:

- Команда RZ — это возврат из подпрограммы при Z = 1. В системе команд i8080 есть возвраты по всем возможным условиям[15]: RNZ, RC, RNC и т. д.
- Буква M — так обозначается ячейка памяти, адресуемая парой H/L. Этот M называют еще *псевдорегистром*, потому что он уместен везде, где допускается *произвольный* 8-

[12] Отдельной команды для обнуления флага C нет. Для этого используют ORA A или ANA A. Для обнуления аккумулятора предпочитали XRA A (команда MVI A, 0 вдвое длинней и медленней).

[13] О переполнении при отработке INR можно узнать по Z = 1 (0ffh -> 0).

[14] Intel рассудил здраво: т.к. *основное* назначение INX и DCX — продвигать 16-битный указатель при обработке массива, эти команды не должны влиять на вычисления в теле цикла, т.е. не трогать флаги.

[15] Казалось бы, удобно, но в i80x86 их не взяли. А условные вызовы C<cond> оказались практически бесполезны, т.к. рассчитаны на подпрограммы без параметров.

битный регистр (команды, где допускается только A, не в счет). Команд LDAX **H** и STAX **H** не существует, вместо них — MOV A, M и MOV M, A соответственно[16]. В примере M появляется также в команде ADC.

Команды сдвигов в i8080 представлены по минимуму: есть только *вращения* влево и вправо, 8-битные (RLC/RRC) или 9-битные — то есть через перенос (RAL/RAR). Чтобы сделать с их помощью логический сдвиг, надо обнулить флаг C (командой ORA A) и выполнить RAR или RAL. В следующем примере, где подсчитывается число единичных бит в байте data, команда ORA A не только сбрасывает флаг C, но и проверяет A = 0 для выхода из цикла.

```
            LDA     data        ; read byte into A
            MVI     b,0         ; number of 1's
M1:
            ORA     a           ; flag c = 0
            JZ      exit        ; if A = 0 exit
            RAR
            JNC     m2
            INR     b
M2:         JMP     m1
EXIT:
            HLT
DATA:       DB      10010101b
```

В обзор не попали некоторые команды i8080, уместные в более сложных задачах[17]. Кстати, обработка *знаковых* данных в i8080 вообще не предусмотрена — нет даже флага арифметического переполнения.

Полная информация по командам доступна, например, в [2].

Рассмотренные примеры программ помогут вам выполнить первую работу из [3].

Остановимся также на программировании вывода на семисегментные индикаторы во второй работе из [3].

Режим OUTS/D

Вход в режим OUTS/D (статическая и динамическая индикация) выполняется из режима WORKS по клавише "0". Потом тип индикации переключается клавишей "#". В этом режиме можно двигаться по шагам клавишей Right.

Внимание: если не включить режим OUTS/D, то индикаторы вообще не работают.

Статическая индикация

Сначала через порт 3 задаем режим для первых трех индикаторов, а через порт 7 — для четвертого. После этого индикаторы доступны через порты 0–2 и 4.

Вот пример вывода "нуля" в первый индикатор, через порт 0.

```
; set static mode, output          code     adr

mvi    a, 200q      ; 3e         0011_1110   0000
                    ;   80       1000_0000   0001
out    3            ; d3         1101_0011   0010
                    ;   03       0000_0011   0011
out    7            ; d3         1101_0011   0100
                    ;   07       0000_0111   0101
```

[16] Команда MOV предназначена для передачи между *произвольными регистрами*, но M (хоть это и ячейка памяти, адресуемая парой H/L) здесь тоже допустима, как если бы M был восьмым программно доступным байтовым регистром: A, B, C, D, E, H, L, **M**.

[17] В частности, 16-битное сложение DAD, работающее с парой H/L.

```
; show zero at the rightmost lamp

mvi    a, 111111b   ; 3e        0011_1110   0110
                    ;   3f      0011_1111   0111
out    0            ; d3        1101_0011   1000
                    ;   00      0000_0000   1001
hlt                 ; 76        0111_0110   1010
```

Динамическая индикация

Вывод "0" в первую лампу справа (номер лампы 1-4 задается в позиционном коде: 1, или 2, или 4, или 8) выглядит следующим образом:

```
        MVI   a,80h
        OUT   13q       ; init device

        MVI   a,3fh     ; "0"
        OUT   10q       ;   -> reg "A"

        MVI   a,1       ; bit n_lamp
        OUT   11q       ;   -> reg "B"
        HLT

0000   3E 80     MVI   a,80h
0002   D3 0B     OUT   13q     ; init device
0004   3E 3F     MVI   a,3fh   ; "0"
0006   D3 08     OUT   10q     ;   -> reg A
0008   3E 01     MVI   a,1     ; bit n_lamp
000A   D3 09     OUT   11q     ;   -> reg B
000C   76        HLT
```

Можно проверить все лампы, исправляя байт по адресу 9 (выделен). Чтобы вывести в другую лампу, надо заменить 1 по адресу 9 на 2, 4 и 8.

Вот как выглядит вывод во все четыре лампы по очереди:

```
        LXI   sp,100h
        MVI   a,80h
        OUT   13q       ; init device

        call  putch
        call  putch
        call  putch
        call  putch

        HLT

putch:  RLC           ; a: 80->40->20->10->8->4->2->1->80, etc.
        OUT   10q     ; a -> reg "A"
        OUT   11q     ; a -> reg "B"
        ret

0000   31 00 01            LXI   sp,100h
0003   3E 80               MVI   a,80h
0005   D3 0B               OUT   13q      ; init device
0007   CD 14 00            CALL   putch
000A   CD 14 00            CALL   putch
000D   CD 14 00            CALL   putch
0010   CD 14 00            CALL   putch
0013   76                  HLT
0014   07          PUTCH:  RLC
0015   D3 08               OUT   10q      ;   -> reg "A"
0017   D3 09               OUT   11q      ;   -> reg "B"
0019   C9                  RET
```

Эту программу придется набрать заново. Затем надо пройти ее по шагам, иначе мы увидим что-то лишь на последнем индикаторе, а все остальные уже погаснут.

В этом примере выводимый код равен позиционному коду лампы. Двигаясь по шагам, вы увидите, как сначала в первой лампе загорается верхний сегмент, потом в ней же — другой; этот сегмент (справа наверху) должен гореть в следующей лампе, но появляется сначала в текущей. Затем первая лампа гаснет и загорается вторая.

Теперь понятно, почему индикация *динамическая*. В каждый момент времени работает *одна* лампа. Надо вывести символ на одну (текущую) лампу, немного подождать и перейти к следующей; текущая при этом погаснет. При достаточно (но не слишком) быстром переходе между индикаторами создается иллюзия, что светятся все.

Итак, надо ввести задержку, и это можно сделать прямо в putch, добавив код со смещения 0019 (перезаписав команду ret):

```
putch:  RLC
        OUT   10q       ;   -> reg "A"
        OUT   11q       ;   -> reg "B"

        mov   b, a      ; save a
        XRA   a         ; a := 0 (max delay, 256 cycles)
M1:     SUI   1
        JNZ   m1
        mov   a, b      ; restore a
        ret
```

```
0019   47               MOV   b,a
001A   AF               XRA   a
001B   D6 01        M1: SUI   1
001D   C2 1B 00         JNZ   m1
0020   78               MOV   a,b
0021   C9               RET
```

И зациклить вызовы putch:

```
0007   3E 88                MVI   a,88h
0009   CD 14 00     M2:     CALL   putch
000C   C3 09 00             JMP   m2
```

Отлично видны все цифры. Хотя немного подрагивают. Надо уменьшить задержку. Мы вычитали в цикле единицу, а теперь будем вычитать 2 (код по адресу 001c надо исправить с 1 на 2):

```
001B   D6 02        M1:     SUI   2
```

Теперь не моргает.

Можно еще уменьшить задержку — записать вместо цикла нули (NOP):

```
001B   00       NOP
001C   00       NOP
001D   00       NOP
001E   00       NOP
001F   00       NOP
0020   78       MOV   a,b
```

А вот теперь уже видно перекрытие сегментов соседних ламп. Задержка слишком мала.

На этом закончим опыты с индикацией. Вариант с вычитанием 2 в цикле можно считать рабочим.

Дальше возникает вопрос: как отлаживать программу вне стенда? Ответ: для этого надо убрать команды для вывода на индикатор из процедуры putch, а заодно определиться с ее входными параметрами.

Пусть подпрограмма putch принимает, в регистрах A и B, такие параметры:

- A — код для записи в лампу;
- B — номер лампы 0–3.

На время отладки основной программы (вне стенда) putch будет пустой:

```
putch:  ret
```

Рабочая подпрограмма putch может быть, например, такой:

```
putch:  push   psw          ; save a/f
        push   b

        out    10q          ; a -> reg "a"
        mvi    a,80h
        inr    b            ; 1-4
m2:
        rlc                 ; rotate a (80 -> 1 -> 2 -> 4 -> 8)
        dcr    b
        jnz    m2
        out    11q          ; a -> reg "b"

        xra    a            ; delay 256/2 cycles
m1:     sui    2
        jnz    m1

        pop    b
        pop    psw
        ret
```

Этот код следует проверить. По-видимому, самая простая проверка такая:

```
loop:   mvi    b, 3         ; n_lamp (3..0)
four:   mov    a, b         ; code = n_lamp + 1
        inr    a            ; (4..1)
        call   putch
        dcr    b            ; --n_lamp
        jp     four         ; >= 0?
        jmp    loop
```

Программа работает в бесконечном цикле, и на каждой итерации вывод идет слева направо (от третьего к нулевому индикатору). Инкремент добавлен для того, чтобы не возникла ситуация code = 0 — тогда на индикаторе вообще ничего не светится.

Список источников

1. Левенталь, Ланс А. Программирование на языке ассемблера для микропроцессоров 8080 и 8085 / Л. Левенталь, У. Сэйвилл ; пер. с англ. А. А. Батнера. Москва : Радио и связь, 1987, 448 с.

2. Lance A. Leventhal. 8080A/8085 Assembly Language Programming. Osborne & Associates Inc. 1978, 495 p.

3. Е. Г. Павловский, В. А. Жвариков, А. А. Кузьмин. Организация и основы программирования микропроцессоров: Учебное пособие и методические указания к лабораторному практикуму. СПб.: Санкт-Петербургский государственный политехнический университет, 2014. 130 с., ил.