Лабораторная работа № 4

Побитовая обработка на языке Ассемблер

Цель работы:

Практическое овладение навыками разработки программного кода на языке Ассемблер Изучение логических командах и команд сдвига. Практическое освоение основных функций отладчика TD.

Задачи:

Разработка простой программы использующей логические команды для приобретения практических навыков программирования ветвлений.

Порядок выполнения работы:

1. Создать рабочую папку для текстов программ на ассемблере и записать в нее файлы tasm.exe, tlink.exe, rtm.exe и td.exe из пакета tasm, а также файл с исходным текстом программы на ассемблере, который сохранить с именем prog№.asm.
2. Создать загрузочный модуль, загрузить его в отладчик и выполнить программу в пошаговом режиме.

Содержание отчета:

1. Цель работы.
2. Постановка задачи.
3. Листинг программ, с подробными комментариями каждой строки.
4. Окно отладчика, демонстрирующий КОНЕЧНЫЙ результат состояния регистров.
5. Вывод.

Постановка задачи:

Задание 1. Полагая, что DL содержит 11110001_2, а переменная BOOL типа Byte содержит $1110'0011_2$, определите значение регистра DL после выполнения каждой отдельно взятой команды:

a) AND DL, BOOL b) OR DL, BOOL
c) XOR DL, BOOL d) AND DL, C
e) XOR DL, $ FF f) NOT DL.

После выполнения каких команд будет установлен флаг ZF?

Задание 2 (Пример Приложение_1).
ВАРИАНТЫ

1. Дано число в двоичном виде. Умножить его на 16. Результат перевернуть следующим образом: нулевой разряд становится седьмым, 1-ый становится 6-ым и т.д.
2. Даны два числа в двоичном виде. В первом числе 3,5,7 разряды обнулить и результат разделить на 4, полученное значение логически умножить на 2-ое число.
3. Дано число в двоичном виде. Поменять местами старшую и младшую части числа. Полученное значение разделить на 32 и проинвертировать.
4. Даны два числа в двоичном виде. В первом числе старшие (4 разряда) разряды обнулить. Во втором числе сделать единицами 2,4,6 разряды. Полученные результаты логически перемножить.
5. Дано число в двоичном виде. Разделить его на 16, занести в 1,3,7 разряды нули. Полученное значение логически сложить с числом 19.

6. Даны два числа в двоичном виде. Первое число умножить на 9, второе разделить на 4. результаты логически перемножить и старшую часть поменять местами с младшей.
7. Дано число в двоичном виде. Поменять местами третий бит с пятым. Результат умножить на 8 и проинвертировать.
8. Дано число в двоичном виде. Логически перемножить его с числом 28. Проинвертировать результат и умножить на 4. В полученном значении 4,5,6 разряды заменить на противоположные.
9. Дано число в двоичном виде. Поменять местами четные разряды с нечетным. Результат проинвертировать и умножить на 4.
10. Дано число в двоичном виде. Вывернуть число «наизнанку» (разряды стоящие в середине сделать крайними). Результат разделить на 16.
11. Даны два числа в двоичном виде. Из первого числа взять четыре младших разряда и поменять местами с четырьмя старшими разрядами второго числа. Результаты логически сложить и разделить на 8.
12. Даны два числа в двоичном виде. Поменять местами 7,6,5,1- разряды первого числа с 0,2,3,4 разрядами второго числа соответственно. Результаты логически сложить и умножить на 8.
13. Дано число в двоичном виде. Все нечетные разряды числа обнулить, а четные заменить на противоположные. Результат разделить на 4 и проинвертировать.
14. Даны два числа в двоичном виде (первое число размером в байт, второе число размером в слово). Первое число умножить на 16 и в полученном значении обнулить 3,5 разряды. Результат сложить со старшей частью второго числа.
15. Даны два числа в двоичном виде. Обнулить в первом числе 3,5,6 разряды и разделить полученное значение на 8, второе число умножить на 2 и логически сложить с первым. Результат проинвертировать.
16. Дано двоичное число. В старшей части числа все четные биты заменить на противоположные. В младшей части числа все нечетные биты обнулить. Результат разделить на 16.
17. Даны два числа в двоичном виде. В первом числе поменять местами старшую и младшую части числа. Во втором 1-ый и 4-ый разряды поменять местами с 3,7-мым разрядами соответственно. Результаты логически сложить и умножить на 4
18. Дано число в двоичном виде. Разделить его на две составляющие: в первую войдут только четные разряды, во вторую только нечетные разряды. Их логически перемножить и результат умножить на 16.
19. Даны два числа в двоичном виде. Первое число умножить на 4. второе разделить на 2 . Результаты логически сложить. 0-ой и 7-ой разряды, в полученном значении, поменять местами.
20. Дано число в двоичном виде. Поменять местами значения четных и нечетных разрядов. Полученное число проинвертировать и умножить на 8.
21. Даны четыре числа в двоичном виде. Составить пятое число, которое состоит из 0-го и 1-го битов первого числа, 2-го и 3-го битов второго числа, 4,5-ые биты из третьего числа, 6,7-ой биты из четвертого числа. Полученное значение проинвертировать и разделить на 16.
22. Дано двоичное число. Поменять местами 3-ий разряд с 7-ым. Полученное значение разделить на 8 и логически сложить с числом 56.
23. Дано число в двоичном виде. 0,1,4,5-ые разряды заменить на противоположные. Остальные занести в отдельный регистр, поставив их на 0,1,4,5- ые биты соответственно. Полученные значения логически перемножить.

Теоретическая часть

Битовые операции

1. **Булевские команды**

Эти команды оперируют с отдельными битами ячейки, независимо от других битов ячейки.

Инвертировать	**not opr**	$opr \leftarrow \neg opr$
"ИЛИ" (дизъюнкция)	**or dst,src**	$dst \leftarrow dst \vee src$
"И" (конъюнкция)	**and dst,src**	$dst \leftarrow dst \wedge src$
"Исключающее ИЛИ"	**xor dst,src**	$dst \leftarrow dst \oplus src$
Логическое сравнение	**test opr1,opr2**	$opr1 \wedge opr2$

Команда **not** на флаги не действует, остальные команды сбрасывают CF и OF, а флаги SF, ZF, PF изменяют по обычным правилам. Рассмотрим серию примеров, иллюстрирующих работу этих команд.

Пример. Команда инвертирования битов. Операция НЕ применяется к каждому биту.

```
mov al, 10011101b  ;   AL = 9Dh
not al             ;   AL = 01100010b = 62h
```

Остальные команды двухоперандные. Обычно они используются по следующей схеме. В источник src помещается так называемая "маска" — непосредственный операнд. Маска указывает, как изменять биты приемника.

Пример. Установка и сброс битов в соответствии с маской.

1) В регистре DL установить 6-й, 3-й и 1-й биты. Применяем дизъюнкцию с маской, используя правила поглощения $1 \vee b = 1, 0 \vee b = b$.

маска	0	1	0	0	1	0	1	0
DL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
DL $\vee$ маска	b_7	1	b_5	b_4	1	b_2	1	b_0

Команда имеет вид **or dl, 01001010b** (или **or dl, 4Ah**).

2) В регистре CH сбросить 4-й и 3-й биты. Применяем конъюнкцию с маской, используя правила поглощения $1 \wedge b = b, 0 \wedge b = 0$.

маска	1	1	1	0	0	1	1	1
CH	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
CH $\wedge$ маска	b_7	b_6	b_5	0	0	b_2	b_1	b_0

Команда **and ch,11100111b** (или **and ch,0E7h**).

3) Инвертировать 4-й и 3-й биты регистра BH. Применяем "исключающее ИЛИ", используя правила $1 \oplus b = \neg b, 0 \oplus b = b$.

маска	0	0	0	1	1	0	0	0
CH	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
CH $\wedge$ маска	b_7	b_6	b_5	$\neg b_4$	$\neg b_3$	b_2	b_1	b_0

Команда **xor ch,11000b** (или **xor ch,18h**).
Команда **xor** применяется для обнуления регистров: код **xor ax,ax** короче, чем **mov ax, 0.**

Пример Вычисление абсолютной величины для содержимого AX без ветвлений.

cwd
xor ax,dx
sub ax,dx

4) Вместо команды **cmp ax,0 ;** AX = 0 ?
предпочитают использовать более короткие команды **and ax,ax** или **or ax,ax**. Они оставляют содержимое AX неизменным, но устанавливают или сбрасывают флаг ZF.

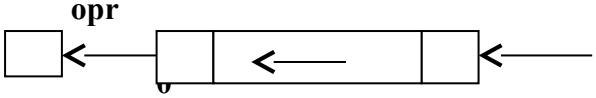
5) Команду **test** используют, чтобы проверить, установлен ли определенный бит операнда, и при этом не "испортить" операнд.

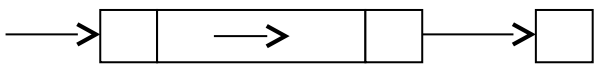
Пример. Установлен ли 5-й бит DL? Если ДА, перейти на метку m.
test dl,00100000b
jnz m
С помощью этой команды легко проверить делимость числа на степени двойки. Например, если два младших бита числа — нули, то оно делится на 4:
test cx, 11b
jz yes

2. Команды сдвигов

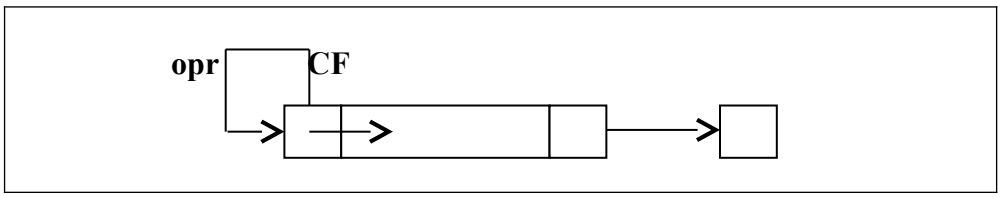
Эти команды перемещают содержимое ячейки влево или вправо. Одним из операндов этих команд является количество сдвигов cnt. Оно либо равно 1, либо определяется содержимым регистра CL (при этом CL сохраняет свое содержимое после операции). Начиная с 286 процессора количество сдвигов, отличное от единицы, можно указывать как непосредственный операнд.
Естественно, возникает вопрос (для определенности говорим о сдвиге на одну позицию вправо): куда девается младший бит (точнее, его содержимое)? что записывается в старший бит? В командах сдвига эти вопросы решаются по-разному.

2.1. **Логические и арифметические сдвиги**.

Логический/арифметический сдвиг влево (Shift Logical/Arithmetic Left)	**shl/sal opr,cnt**
CF **opr**  **0**	

Логический сдвиг вправо (Shift Logical Right)	**shr opr,cnt**
opr **CF** **0** 	

Арифметический сдвиг вправо (Shift Arithmetic Right)	**sar opr,cnt**



Флаг CF устанавливается в соответствии с рисунками. Флаг OF имеет смысл , если cnt = 1. Сначала остановимся на формальном синтаксисе команд.

Неправильно	Правильно
1) **sar ax,2** (в 286 и выше — правильно!)	**1) mov cl,2** **sar ax,cl**
2) mov bl,3 **shr dx,bl** (количество сдвигов только в CL)	2) **shl ax,1**

Арифметические сдвиги предназначены для выполнения умножения и деления на степени двойки (табл.1).

Таблица .1

операнд	умножение	деление
знаковый	**sal/shl**	**sar**
беззнаковый		**shr**

Умножение на 2 — сдвиг влево и приписывание справа нуля. Докажите, что OF = CF $\oplus$ SF.

Деление на 2 — сдвиг вправо, в CF попадает остаток от деления. При арифметическом сдвиге вправо дублируется знаковый бит: поэтому отрицательное число остается отрицательным.

Пример. Пусть AL = –120 = 10001000. **sar al,1** дает AL = –60 = 11000100. Проверьте.

К сожалению, результат деления с помощью команды **sar** не всегда совпадает с результатом деления, полученным с помощью команды **idiv**. Сравните результат от деления –5 на 2, полученный с помощью и той и другой команды.

3. Средства Ассемблера для битовых операций

На этапе ассемблирования над числами возможны следующие логические операции: **NOT**, **AND**, **OR**, **XOR**, **SHL**, **SHR**.
Приведем примеры.

```
L = 101b
L = L SHL 2            ; L = 10100b
mov al, NOT 3          ; AL = 0FCh
mov bh, 1100b OR 1010b ; BH = 1110b
```

Когда требуется плотная упаковка нескольких числовых значений, диапазон изменения которых невелик, их помещают в заранее определенные битовые поля ячейки памяти. Рассмотрим пример.
Разместим в слове информацию о работнике (табл.2).

Таблица .2

Биты	Длина поля	Имя поля	Назначение
15	1	sex	пол (0 — мужчина, 1 — женщина)
14	1	married	семейное положение (0 — холост, 1 — женат)
13:10	4	children	количество детей (0 — 15)

9	1	xxx	зарезервировано
8:2	7	age	возраст (0 — 127)
1:0	2	school	уровень образования (0 — 3)

Пусть в программе имеется описание:

Johnson DW 0100100010001011b

Нужно получить количество детей Джонсона (чтобы предоставить ему налоговые льготы). Сначала дадим "лобовое" решение задачи.

mov ax, Johnson ; поместить данные в AX
and ax, 0011110000000000b ; выделить поле children
shr ax, 10 ; прижать поле к правому краю
; (в AX — количество детей)

Стоит нам ошибиться хотя бы на один бит в маске и количестве сдвигов и мы получим неверный результат. Нельзя ли поручить этот скрупулезный подсчет битов Ассемблеру? Да, TASM предоставляет нам такую возможность.

Приведем полный текст программы

```
.MODEL small
.286
.STACK 100h
.DATA
person RECORD sex:1=0,married:1,children:4,xxx:1, \
        age:7,school:2=1
Johnson person <,1,2,,34,3>
.CODE
start:
  mov ax,@data
  mov ds,ax
  mov ax, Johnson
  and ax, MASK children
  shr ax, children
  mov ax,4c00h
  int 21h
  END start
```

Прокомментируем текст программы. Директива **.286** разрешает использовать инструкции 286-го процессора. Эта директива нужна, чтобы TASM сгенерировал код инструкции **shr ax,10** (посмотрите в Turbo Debugger, что получится, если не указать эту директиву). Директива **RECORD** задает шаблон описания битовых полей. Этот шаблон носит имя **person**. В шаблоне указаны имя битового поля и его ширина, а не биты, которое занимает поле, — из-за этого пришлось включить в шаблон зарезервированное поле **xxx**. В двух полях (**sex** и **age**) после знака равенства дано значение "по умолчанию", которое может быть переопределено при описании с использованием шаблона. В шаблоне необязательно описывать все биты ячейки: можно было описать, например, только поля **age** и **school**. Само описание шаблона памяти не выделяет, зато описание выделяет в памяти слово в соответствии с шаблоном. Использован символ \ для продолжения директивы на следующей строке. В этом описании заданы значения всех полей, кроме **xxx** (значение которого безразлично), и **sex** (которое берется по умолчанию из шаблона). Три строки в программе после загрузки DS полностью эквивалентны трем строкам, приведенным ранее. Операция (выполняемая на этапе ассемблирования) **MASK поле_записи** возвращает маску, необходимую для выделения поля записи. Значение **поля_записи** — число битов, на которые нужно осуществить сдвиг вправо, чтобы прижать поле к правому краю ячейки.

Если нужно прижать поле к левому краю слова, можно воспользоваться командой
shl ax, 16 – WIDTH children – children.

Здесь в вычислениях участвует ширина поля **WIDTH поле**.

Команда двойного сдвига применяется для быстрого перемещения строки битов на новое место в памяти. (В англоязычной литературе используется сокращение "bit blt" (BIT Block Transfer — перемещение блока бит).
Пример. Вставить битовую подстроку длиной 16 бит в строку битов, начиная с 8-го бита (т.е. заменить биты 23:8 заданной строкой из 16 бит). Вставляемая подстрока выровнена к левому краю.

```
        MODEL small
        .386
        .STACK 100h
        .DATA
bit_str DD      12345678h       ;  строка-приемник
p_str   DD      0ABCD0000h   ;   вставляемая подстрока 0ABCDh
        .CODE
start:  mov     ax, @data
        mov     ds, ax

        mov     eax, p_str        : Поместить подстроку в EAX
; Правый край места вставки циклически переместить к краю
; строки bit_str (т.е. слева поместить правые биты)
        ror     bit_str,8           ; (bit_str = 78123456)
; Сдвинуть строку вправо на длину подстроки
        shr     bit_str,16          ; (bit_str = 00007812)
; Поместить подстроку в строку-приемник (16 – размер подстроки)
        shld    bit_str,eax,16   : (bit_str = 7812ABCD)
; Восстановить младшие 8 бит (вернуть правые биты на место)
        rol     bit_str,8            : (bit_str=12ABCD78)

        mov     ax,4C00h
        int     21h
        END     start
```

Битовые операции, появившиеся в 386-м процессоре

1. Команды сканирования битов bsf и bsr.

Сканирование бита вперед	**bsf dst,src**	в dst — индекс первого справа единичного бита в src
Bit Scan Forward		ZF = 1, если все биты src нулевые ZF = 0, если в src есть единичный бит
форматы операндов: bsf r16, r/m16, bsf r32, r/m32		

Сканирование бита назад	**bsr dst,src**	в dst — индекс первого слева единичного бита в src
Bit Scan Reverse		ZF = 1, если все биты src нулевые ZF = 0, если в src есть единичный бит
форматы операндов: bsr r16, r/m16, bsf r32, r/m32		

Итак, "вперед" — в направлении увеличения нумерации битов, "назад" — в направлении уменьшения. Индекс — номер разряда (бита).

Пример. Выяснить индекс самого старшего установленного бита в регистре DX (пусть DX содержит число 5).

```
bsr     ax, dx  ;  ax = 2
```

Пример. Сдвинуть содержимое регистра EBX вправо так, чтобы младший бит EBX был установлен; если это невозможно, перейти на метку null (Пусть EBX = 84h = 0…0 1000 0100b.)

```
bsf     ecx, ebx  ; ecx = 2
jz      null
shr     ebx, cl     ; ebx = 21h = 0…0 0010 0001b
```

2. Команды проверки битов bt, bts, btr, btc

Сначала разберем команду bt, остальные команды на нее похожи.

Проверка бита	**bt src, index**	сохраняет бит из src с номером index в CF
Bit Test		CF = 1, если бит src установлен, CF = 0, если бит src сброшен
форматы операндов: bt r/m16, r16, bt r/m32, r32, bt r/m16, i8, bt r/m32, i8		

Пример. Поместить второй бит регистра CX в флаг CF.

```
        mov     cx, 5  ; cx = 101b
        bt      cx, 2  ; CF = 1
```

Пример. Решим уже знакомую задачу: сосчитать количество единичных битов в регистре AX.

```
        xor     cx, cx
        xor     bx, bx
n:      bt      ax, cx
        adc     bx, 0
        inc     cx
        cmp     cx, 16
        jnae    n
```

Для оставшихся трех команд не будем приводить подробного описания — оно вполне аналогично команде bt.

btc — проверка и инвертирование (bit test and complement) — бит помещается в CF, сам бит инвертируется.

btr — проверка и сброс (bit test and reset) — бит помещается в CF, сам бит сбрасывается.

bts — проверка и установка (bit test and set) — бит помещается в CF, сам бит устанавливается.

Упражнение. Придумайте примеры для каждой из приведенных команд и проверьте их работу в TD.

3. Команды двойного сдвига

Двойной сдвиг влево	**shld dst,src, count**	описание приведено ниже
Double precision Shift Left		SF, ZF, PF по результату; в CF последний выдвинутый бит; OF = 1, если произошло изменение знака (имеет смысл при count = 1)
форматы операндов: shld r/m16, r16, i8/cl shld r/m32, r32, i8/cl		

dst и src объединяются и происходит сдвиг влево на count бит объединенного содержимого. src остается без изменений.

Аналогично работает команда двойного сдвига вправо shrd dst,src, count.

Команда двойного сдвига применяется для быстрого перемещения строки битов на новое место в памяти. (В англоязычной литературе используется сокращение "bit blt" (BIT Block Transfer — перемещение блока бит).
Пример. Вставить битовую подстроку длиной 16 бит в строку битов, начиная с 8-го бита (т.е. заменить биты 23:8 заданной строкой из 16 бит). Вставляемая подстрока выровнена к левому краю.

```
        MODEL small
        .386
        .STACK 100h
        .DATA
bit_str DD      12345678h       ;  строка-приемник
p_str   DD      0ABCD0000h   ;   вставляемая подстрока 0ABCDh
        .CODE
start:  mov     ax, @data
        mov     ds, ax

        mov     eax, p_str        : Поместить подстроку в EAX
; Правый край места вставки циклически переместить к краю
; строки bit_str (т.е. слева поместить правые биты)
        ror     bit_str,8            ; (bit_str = 78123456)
; Сдвинуть строку вправо на длину подстроки
        shr     bit_str,16           ; (bit_str = 00007812)
; Поместить подстроку в строку-приемник (16 – размер подстроки)
        shld    bit_str,eax,16   : (bit_str = 7812ABCD)
; Восстановить младшие 8 бит (вернуть правые биты на место)
        rol     bit_str,8             : (bit_str=12ABCD78)

        mov     ax,4C00h
        int     21h
        END     start
```

```
a db 10110101b
b db 00110111b
c db 0
```

```
start:
mov ax,ds                ; инициализация сегмента данных
mov ds,ax
not a                    ; инвертируем первое число и делим его на 4
shr a,2
shl b,1                   ; второе число умножаем на 2
mov al , a                ; полученные результаты складываем
add al , b
xor al, 00001111b        ; меняем первые четыре разряда на противоположные
mov c,al
mov ax,4c00h              ;завершаем работу программы
int 21h
```