Лабораторная работа № 8. Операторы выбора. Условные переходы и циклы.

Цель работы

Получить представление о реализации операторов выбора на уровне ассемблера и приобретение практических навыков по использованию команд для реализации условных переходов и ветвлений.

Теоретические сведения

Некоторые команды передачи управления

Таблица 1 – Оператор `jmp`

Команда:	`jmp операнд`
Назначение:	Безусловный переход
Процессор:	8086

`jmp` передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операндом может быть непосредственный адрес для перехода (в программах используют имя метки, установленной перед командой, на которую выполняется переход), а также регистр или переменная, содержащая адрес. В зависимости от типа перехода различают:

- ***переход типа short (короткий переход)*** — если адрес перехода находится в пределах от -127 до +128 байт от команды `jmp`;
- ***переход типа near (ближний переход)*** — если адрес перехода находится в том же сегменте памяти, что и команда `jmp`;
- ***переход типа far (дальний переход)*** — если адрес перехода находится в другом сегменте. Дальний переход может выполняться и в тот же самый сегмент, если в сегментной части операнда указано число, совпадающее с текущим значением `CS`;
- ***переход с переключением задачи*** — передача управления другой задаче в многозадачной среде.

При выполнении переходов типа short и near команда `jmp` фактически изменяет значение регистра `EIP` (или `IP`), изменяя тем самым смещение следующей исполняемой команды относительно начала сегмента кода. Если операнд — регистр или переменная в памяти, то его значение просто копируется в `EIP`, как если бы это была команда `mov`. Если операнд для `jmp` — непосредственно указанное число, то его значение суммируется с содержимым `EIP`, приводя к относительному переходу. В ассемблерных программах в качестве операнда обычно указывают имена меток, но на уровне исполнимого кода ассемблер вычисляет и записывает именно относительные смещения.

Выполняя дальний переход в реальном режиме, виртуальном режиме и в защищенном режиме (при переходе в сегмент с теми же привилегиями), команда `jmp` просто загружает новое значение в `EIP` и новый селектор сегмента кода в `CS`, используя старшие 16 бит операнда как новое значение для `CS` и младшие 16 или 32 — как значение `IP` или `EIP`.

Таблица 2 – Семейство команд `jcc`

Команда:	`jcc метка`
Назначение:	Условный переход
Процессор:	8086

Это набор команд, каждая из которых выполняет переход (типа short или near), если удовлетворяется соответствующее условие. Условием в каждом случае реально является состояние тех или иных флагов, но, если команда из набора `jcc` используется сразу после `cmp`, условия приобретают формулировки, соответствующие отношениям между операндами `cmp` (см. табл. 3). Например, если операнды `cmp` были равны, то команда `je`, выполненная сразу после этого `cmp`, осуществит переход. Операнд для всех команд из набора `jcc` — 8-битное или 32-битное смещение относительно текущей команды.
Слова «выше» и «ниже» в таблице относятся к сравнению чисел без знака, слова «больше» и «меньше» учитывают знак.

Таблица 3 – Варианты команды `jcc`

Код команды	**Реальное условие**	**Условие для CMP**
`ja` `jnbe`	CF = 0 и ZF = 0	если выше если не ниже или равно

`jae`
`jnb`
`jnc`
	CF = 0	если выше или равно
		если не ниже
		если нет переноса

`jb`
`jnae`
`jc`
	CF = 1	если ниже
		если не выше или равно
		если перенос

`jbe` `jna`	CF = 1 и ZF = 1	если ниже или равно если не выше
`je` `jz`	ZF = 1	если равно если ноль
`jg` `jnle`	ZF = 0 и SF = OF	если больше если не меньше или равно
`jge` `jnl`	SF = OF	если больше или равно если не меньше
`jl` `jnge`	SF <> OF	если меньше если не больше или равно
`jle` `jng`	ZF = 1 и SF <> OF	если меньше или равно если не больше
`jne` `jnz`	ZF = 0	если не равно если не ноль
`jno`	OF = 0	если нет переполнения
`jo`	OF = 1	если есть переполнение
`jnp` `jpo`	PF = 0	если нет четности если нечетное
`jp` `jpe`	PF = 1	если есть четность если четное
`jns`	SF = 0	если нет знака
`js`	SF = 1	если есть знак

Команды jcc не поддерживают дальних переходов, так что, если требуется выполнить условный переход на дальнюю метку, необходимо использовать команду из набора jcc с обратным условием и дальний jmp, как, например:

```
    cmp     ax,0
    jne     local_1
    jmp     far_label  ; переход, если AX = 0
local_1:
```

Таблица 4 – Команды jcxz/jecxz

Команда:	jcxz метка
Назначение:	Переход, если CX = 0
Команда:	jecxz метка
Назначение:	Переход, если ECX = 0
Процессор:	8086

Выполняет ближний переход на указанную метку, если регистр CX или ECX (для jcxz и jecxz соответственно) равен нулю. Так же как и команды из серии jcc, jcxz и jecxz не могут выполнять дальних переходов. Проверка равенства CX нулю, например, может потребоваться в начале цикла, организованного командой loopne, — если в него войти с CX = 0, то он будет выполнен 65 535 раз.

Логические операции

Формальной основой для реализации операторов IF на языке ассемблера X86 являются команды сравнения и операторы перехода условного/ безусловного в сочетании с логическими командами.

Таблица 6 – Команда «И»

Команда:	and приемник, источник
Назначение:	Логическое И
Процессор:	8086

Команда выполняет побитовое «логическое И» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, только если соответствующие биты обоих операндов

были равны 1, и равен 0 в остальных случаях. Наиболее часто and применяют для выборочного обнуления отдельных бит, например, команда

```
and     al,00001111b
```

обнулит старшие четыре бита регистра AL, сохранив неизменными четыре младших.
Флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

Таблица 7 – Команда «ИЛИ»

Команда:	or приемник, источник
Назначение:	Логическое ИЛИ
Процессор:	8086

Выполняет побитовое «логическое ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 0, только если соответствующие биты обоих операндов были равны 0, и равен 1 в остальных случаях. Команду or чаще всего используют для выборочной установки отдельных бит, например, команда

```
or      al,00001111b
```

приведет к тому, что младшие четыре бита регистра AL будут установлены в 1.
При выполнении команды or флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.

Таблица 8 – Команда «XOR»

Команда:	xor приемник, источник
Назначение:	Логическое исключающее ИЛИ
Процессор:	8086

Выполняет побитовое «логическое исключающее ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, если соответствующие биты операндов различны, и нулю, если одинаковы. xor используется для самых разных операций, например:

```
xor     ax,ax       ; обнуление регистра AX
```

или

```
xor     ax,bx
xor     bx,ax
xor     ax,bx       ; меняет местами содержимое AX и BX
```

Оба этих примера могут выполняться быстрее, чем соответствующие очевидные команды

```
mov     ax,0
```

или

```
xchg    ax,bx
```

Таблица 9 – Команда «НЕ»

Команда:	not приемник
Назначение:	Инверсия
Процессор:	8086

Каждый бит приемника (регистр или переменная), равный нулю, устанавливается в 1, и каждый бит, равный 1, сбрасывается в 0. Флаги не затрагиваются.

Таблица 10 – Команда «TEST»

Команда:	`test приемник, источник`
Назначение:	Логическое сравнение
Процессор:	8086

Вычисляет результат действия побитового «логического И» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и устанавливает флаги `SF`, `ZF` и `PF` в соответствии с полученным результатом, не сохраняя результат (флаги `OF` и `CF` обнуляются, значение `AF` не определено). `test`, так же как и `cmp`, используется в основном в сочетании с командами условного перехода (`jcc`), условной пересылки данных (`cmovcc`) и условной установки байт (`setcc`).

Циклы в ассемблере

Циклы в ассемблере создаются так же с помощью условных и безусловных переходов. Пример создания цикла со счётчиком показан ниже. Как известно синтаксис цикла со счётчиком выглядит следующим образом:

```
for (действие до начала цикла;
     условие продолжения цикла;
     действия в конце каждой итерации цикла) {
          тело цикла;
          инструкция цикла;
          инструкция цикла 2;
          инструкция цикла N;
}
```

Для взаимодействия с ассемблерной вставкой из кода на C++ создадим две переменные:

```
unsigned int N = 5;
unsigned int A = 0;
```

Для примера создадим простейший цикл со счётчиком:

```
A = 0;
for (int i = 0; i < N; i++)
{
    A += i;
}
```

Ассемблерная вставка реализующая это код будет иметь следующий вид:

```
__asm
    {
          MOV EAX, 0 ; Занесём в EAX ноль (регистр A будет
                     ; использоваться вместо переменной A)
          MOV EBX, N ; Занесём значение переменной N в регистр EBX
                     ; (чтобы ускорить работу с этим значением)

          ; Вместо переменной i используется регистр ECX
          ; (так называемый "счётчик" (counter) часто используемый в
          ; качестве переменной счетчика в цикле for)
```

```
        ; Выполняем действие, выполняемое до начала цикла
        MOV ECX, 0 ; Заносим в ECX начальное значение (i = 0)
m0:
        ; Переход к метке m0 означает переход к следующей итерации цикла

        ; Выполняем проверку условия продолжения цикла
        CMP ECX, EBX ; Сравниваем значения регистра ECX и EBX
                     ;(в котором значение N = 5)
        ; Обычно мы выполняем тело цикла когда условие продолжения
        ; выполняется. Здесь же, мы должны задать такое условие,
        ; при котором нужно выйти из цикла
        JNL m1 ; Выполняем переход если ECX при сравнении было не
               ; меньше EBX. На C++ было бы if (!(i < 5))

        ; Начало тела цикла
        ADD EAX, ECX ; Прибавляем к EAX значение ECX (A += i)
        ; Конец тела цикла

        ; Выполняем действие, выполняемое в конце каждой итерации цикла
        INC ECX ; Увеличиваем ECX на единицу (i++)

        ; Переходим к следующей итерции цикла
        JMP m0 ; Выполняем безусловный переход к метке m0
m1:
        ; Переход к метке m1 означает выход из цикла
        MOV A, EAX
    }
```

Все остальные разновидности циклов строятся аналогичным образом.

Задания

1. Вычислить действительные корни квадратного уравнения
2. Дан отрезок на прямой , заданный своими концевыми точками – *a*, *b*. Задана точка *c*. Лежит *c* внутри отрезка или снаружи?
3. Написать программу, позволяющую по последней цифре целого числа определить последнюю цифру его квадрата . Вариант – число типа float, double /не целое, дробное/.
4. Составить программу , которая по заданному году *y* и номеру месяца *m* определяет количество дней в году и этом месяце.
5. Пусть элементами круга являются радиус (1-й элемент), диаметр (2-й элемент), и длина окружности (3-й элемент). Составить программу , которая по номеру элемента вычисляла бы площадь круга.
6. Именинник пригласил на праздник N (до 10) гостей. Для каждого гостя было подготовлено отдельное место с его фамилией. Вывести на экран количество возможных способов рассаживания гостей, при которых ни один гость не будет сидеть на предназначенном для него месте.
7. Даны числа от единицы до миллиарда. У каждого числа вычисляется сумма цифр, до тех пор пока не получится однозначное число (для 8195: 8+1+9+5 = 23; затем для 23: 2+3 = 5). Вывести на экран, каких однозначных чисел получится больше.
8. Введите месяц и день вашего рождения. Выясните, какой ближайший год

будет счастливым (год называется счастливым, если остаток от деления суммы его цифр на 10 совпадает с аналогичным остатком суммы цифр месяца и дня рождения).

9. Найти наибольший общий делитель (НОД) двух введенных натуральных чисел, используя алгоритм Евклида (алгоритм Евклида: вычитаем из меньшего большее до тех пор, пока они не сравняются, полученное в результате число и есть НОД).
10. Написать программу, которая приблизительно рассчитывает значение математической константы e, используя формулу: e=1 + 1/1! + 1/2! + 1/3! …
11. Написать программу, которая приблизительно рассчитывает значение ex, используя формулу: e^x=1 + x/1! + x^2/2! + x^3/3! …
12. Найти на отрезке [n,m] натуральное число, имеющее наибольшее количество делителей.

Контрольные вопросы

1. Для чего нужен оператор `jmp`?
2. Какие типы переходов Вы знаете?
3. Какие варианты команды `jcc` Вам известны?
4. Какие команды для логических операций Вы знаете? Какие функции они выполняют?