Стек

Стек продолжение

Числа с плавающей точкой

Числа с плавающей точкой, векторные инструкции

Динамическая память

Строковые операции

Макросы

If Else

стек

вычитаем - увеличиваем память прибавляем к esp уменьшаем

Задачи

1. Сохранить что то \ достать
2. Механизм передачи параметров функции

ИНСТРУКЦИЯ PUSH и POP

1. Принимает word и double word аргументы
2. синтаксис: push(pop) op1 $op1 \in m16/m32/r16/r32/imm$ уточнение - при push imm (константа) всегда выделяется 4 байта
3. Равносильно добавить
   sub esp, 4
   mov [ss:esp], op1
4. забрать из стека
   sub op1, [ss:esp]
   add esp, 4

ещё ИНСТРУКЦИИ

pusha/popa	16 бит
pushad/popad	32 бит

Берёт все регистры общего назначения и добавляет их на вершину стека

pushf/popf	16 бит
pushfd/popfd	32 бит

Берёт регистры флагов eflags и добавляет их на вершину стека (но не все но нам это неважно)

в 32 битной системе все 16 битные инстуркции работаю также как и 32 битные

1. call метка равносильно:

   sub esp, 4

   mov ss:esp, eip

   jmp метка

   или можно передать регистр eax → call [eax+eps:eax]

2. ret op1 равносильно

   mov тень, [esp]; тень - теневой регистр скрытый от нас

   add esp, 4+op1

   jmp тень

   Объеснение работы: Программа доходит до call, она ложит в вершину стека eip (точка возврата) выполнили функцию по метке и дошли до ret который берёт из стека eip и возвращается

   call и ret обязательно добавляют и забирают память стека.

3. call foo

   foo:

   mov eax, [esp]; в eax будет сохранено значение eip это единственный способ его достать

   ret

стековый кадр

используем регистр ebp для сохранения изначального значения esp для дальнейши и call

выполнили call на некоторую метку (функцию) имеющую вид:

push ebp

mov ebp, esp

...; расширили стек на сколько надо для локальных переменных

mov [ebp-8], eax

mov [ebp-8+4], edx

локальные переменные ближе к esp внешние параметры ниже по положительеым смещениям от ebp

обращение к переменным что вызывали

mov eax, [ebp]

mov [eax-8], ebx; сохранили в локальные переменные вышестоящей функции значение ebx

мы можем рекурсивно забираться дальше пока ebx $\neq 0$

$$\begin{cases} \text{push ebp} \\ \text{mov ebp, esp} \end{cases} = \text{enter - в одну инстуркцию на экзамене не использовать по аналогии } \exists \text{ leave}$$

соглашения о передаче переменных

call conventions

1. зависят от языка программмирования
2. описывает способ передачи параметров
3. описывает способ возвращения результатов
4. кто чистит стек от параметров
5. какие регистры не меняются функцией

Соглашения:

1. systemV ABI - Unix системы
2. cdecl - windows C программы
3. stdcall - winAPI, Freepascal
4. syscall - система вызова windows
5. fastcall - linux версия(3 параметра регистр) и windows версия(2 параметра регистр) вроде
6. pascal - turbopascal
7. thiscall - C++

среди всего этого множества мы изучаем stdcall

способы передачи параметров

1. через регистры
2. через стек $\begin{cases} \text{в прямом порядке - первый элемент ближе к началу памяти} \\ \text{в обратном порядке - первый элемент ближе к концу памяти} \end{cases}$

параметры в обратном порядке через стек cdecl stdcall

al - 8бит $\mathbb{Z}$

ax - 16бит $\mathbb{Z}$

eax - 32бит $\mathbb{Z}$

edx:eax - 64бит $\mathbb{Z}$

st$\emptyset$ - с плавающей точкой $\mathbb{R}$

cdecl, systemV ABC: СТЕК чистит тот кто вызывает функцию

stdcall, pascal: стек чистит сама функция

cdecl, stdcall: можно менять eax, ecx edx, eflags

fastcall: ecx, edx

Продолжение со стеком

имя_функции proc public (stdcall)

⋮

имя_функции endp

на паскале:

```
1        function sum_numbs(a, b: integer): longintege;
2            sum_numbs := a+b
```

на ассемблере:

```
1        sum_numbs proc
2            push ebp
3            mov ebp, esp
4            mov ax, [ebp+8]; взали a
5            mov dx, [ebp+12]; взали b
6            movsx eax, ax
7            movsx edx, dx
8            add eax, edx
9            pop ebp
10           ret 8; 2 параметра
11           sum_numbs endp
12   код Функции:
13   .data
14       a dw 3
15       b dw 4
16   .code
17   ...
18   mov ax, a
19   mov bx, b
20   push ebx
21   push eax
22   call sum_numbs
```

```
1        function sum_abc(a, b: integer; var c:longint): boolean;
2            c := a+b
```

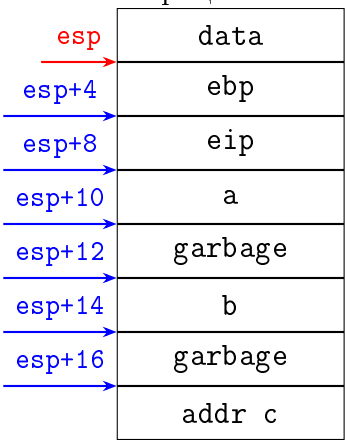
на ассемблере:

```
        sum_abc proc
            push ebp
            mov ebp, esp
            mov ax, [ebp+8]; взали a
            mov dx, [ebp+12]; взали b
            add ax, dx
            mov ecx, 0
            jno @F; прыгнуть на ближа'шую анонимную метку ниже
            mov ecx, 1
            @@: ; анонимнаа метка преобразуетса в ?? 0001
            mov edx, [ebp+16]; взали адрес C
            cwde; расширили ax
            adc eax, 0; дла корректности в случае переполнениа
            mov [edx], eax; записали по ссылке
            mov eax, ecx
            pop ebp
            ret 12; 3*4
            sum_abc endp
    ;код Среды:
    .data
        a dw 3
        b dw 4
        c dd ?
    .code
    ...
        push offset c; передадим ссылку
        mov ax, b
        push eax
        mov ax, a
        push eax
        call sum_abc
```

код с лекции:

print_number знаковое значение 32-битное number - выводим в десятичной системе значение

Иллюстрация стека памяти для задачи 2



Числа с плавающей точкой

1. FPU - floating point unit - 8087 сопроцессор (чуть инфы 1 потока)

   позволяет вычислять 10 байтные Числа

   ! в FPU существуеют регистры $st(\emptyset), st(1), st(2), \ldots, st(7)$, где $st(\emptyset)$ - вершина стека

   ! Каждый регистр имеет 80 бит (10 байт) размер, Freepascal реализует использование 80 битных чисел в типе extended (на C long double) на порядок выделяется 15 бит

   * fld dword ptr mem; single precision (одинарная точность) float

   * fld qword ptr mem; double precision (двойная точность) double

   * fld tword ptr mem; long double|extended

   ! помещает в вершину стека st0 байты из памяти

   * fstp dword|qword|tword ptr mem - инструкция кладёт в указнный адрес в памяти значение st0

   * По соглашению stdcall возвращаем значение в st0

   ! Вся остальная инфа в лекциях первого потока, мы не этим пользуемся

2. Хронология перехода набора инструкций 3DNOW → MMX → SSE → SSE2 → SSE3 (pentium pro)→ SSE4 но с появлением AMD64 → AVX → AVX2 → AVX512
   Мы работает с SSE2, SSE3

3. SSE2

   * есть 7 регистров $xmm0, xmm1, \ldots, xmm7$ каждый по 128 бит

   ! Различные способы интерпретации регистров

   $$|_{32}|_{32}|_{32}|_{32}|$$
   $$|_{64}\quad|_{64}\quad|$$

   * Обращение может

     (a) Скалярное → $|_{32}|$ - к блоку в мнемонике scalar

(b) Векторное обращение к всем 128 битам в мнемонике **p**acked

в продолжении sse3 число регистров выросло до 16 в AVX 32 регистра ymm в AVX 512 регистры zmm 512 битности

Скалярные инстуркции

(a) **movss** op1, op2 ; 32 битное обращение в память|регистр op1 - память → xmm[0 . . . 7] или наоборот как укажешь

(b) **movsd** op1, op2 ; 64 битное обращение в память|регистр op1 - память → xmm[0 . . . 7] или наоборот как укажешь

addss	addsd
subss	subsd
mulss	mulsd
divss	divsd

ss работа в в скалярном виде с числом с одинарной точностью sd работа в скалярном виде с числом 2 точностью

(c) **comicss**

(d) **comicsd**

! выставляют ZF, SF, PF (PF в случае ошибки) операнд это регистр

4. Векторные инстуркции

movups(movupd) op1, op2; берут все 128 бит op1, op2 - (память|xmm[0 . . . 7])

movaps(movapd); где a-aligned требует выравненнного стека по 16 бит работает быстрее

Пример (мнемоника инструкций по аналогии)

addps xmm0, xmm1; сложит векторно каждый блок сложит векторно

$$\begin{array}{|c|c|c|} \hline \__{32} & |\__{32}|\__{32}|\__{32}| & xmm\emptyset \\ \hline \__{32} & |\__{32}|\__{32}|\__{32}| & xmm1 \\ \hline xmm1[3] \underset{32}{+} xmm0[3] & |\__{32}|\__{32}|\__{32}| & xmm\emptyset \\ \hline \end{array}$$

сложит 4 числа одновременно

xorps(d)

xorps xmm0, xmm0; занулит xmm0

5. Конвертация

* cvt[p,s]$_{\text{что}}$[s,d,i] 2 [p,s]$_{\text{куда}}$[s,d,i] op1, op2; op1 - куда, op2 - что

! Пример cvtss2sd xmm0, xmm1;

! первая буква [ps] - векторно или скалярно (весь регистр или лишь один блок)

! вторая буква [sdi] - одинарная точность 32бита, 2 точность 64 бита, в целые числа знаковые

Продолжение чисел с плавающей точкой

```
1        cvtsd2si xmm, reg32\64
2        cvtsd2ss xmm|mem, xmm
3        cvtsi2sd mem32|64(reg32|64), xmm
4        cvtsi2ss mem32|64(reg32|64), xmm
5        cvtss2sd xmm|mem32, xmm
6        ; замечаниа по переводу в целочисленны'ї
7        cvtss2si xmm|mem64, xmm; если число вылезло за диапозон то
8        ; положит в память 100000...
9        ; после запато'і часть округлаетса к ближа'їшему чётному
10       ; см 2 бита последние
11       cvttsd2si xmm|mem64, reg32|64; t-урезание
12       ; если число вышло за диапозон допустим порадок 2^2^10
13       ; то заполнит единицами (как максимальное)
14       ; часть после запато'і выкинет сразу
15       cvttss2si xmm|mem32, reg32|64
```

команда cvt[t]s - скалярная cvt[t]p - векторное кладёт в 128 битное место очевидно

в AVX инструкциях: мнемоника инструкций начинается с V - VCVTSD2SI к примеру

регистры ymm

вычислить скалярное произведение

$$A: |_{\square}|_{\bigcirc}|_{\triangle}|_{\bigtriangledown}| \; | \; | \; | \; | \; | \; | \; | \; | \; | \; | \; | \; |$$

$$B: |_{\square}|_{\bigcirc}|_{\triangle}|_{\bigtriangledown}| \; | \; | \; | \; | \; | \; | \; | \; | \; | \; | \; | \; |$$

A и B выравнены в памяти по 16 то есть адрес начала

внутри числа одинарной точности 32 бита

мы положим в xmm0 командой **movaps** произведём умножение **mulps**

$$|_{\square}|_{\bigcirc}|_{\triangle}|_{\bigtriangledown}|$$

сделаем также для обоих массивов и положим в xmm0 произведение с посощью векторных инструкций

Сложить все произведения

Способ 0 - просто выгрузить в память и сложить по сути скалярно потребует много взаимодействий с памятью

Способ 1 инстурукция **haddps** - горизаонтальное сложение появилось в SSE3

$$|_{\square_1}|_{\bigcirc_1}|_{\triangle_1}|_{\triangledown_1}| \qquad xmm0$$

$$|_{\square_2}|_{\bigcirc_2}|_{\triangle_2}|_{\triangledown_2}| \qquad xmm1$$

haddps xmm0, xmm1, xmm0;

$$|_{\square_1+\bigcirc_1}|_{\triangle_1+\triangledown_1}|_{\square_2+\bigcirc_2}|_{\triangle_2+\triangledown_2}| \qquad xmm1$$

Способ 2 с помоощью инстуркции **shuffps** xmm0, xmm1, imm8(байт с маской); появилось в SSE2

$$|_{3(11)}\square|_{2(10)}\bigcirc|_{1(01)}\triangle|_{0(00)}\triangledown| \qquad xmm0$$

- снизу подписаны адреса соответсвенно 11_10_01_00

второй регистр адрес назначения

инстуркция сделает перестановку по байту (поделим на 4) смотрим двойкам

для примерера 00_11_10_01

значит 11 $\rightarrow$ 00,10 $\rightarrow$ 11,01 $\rightarrow$ 10,00 $\rightarrow$ 01 будто сдвиг по циклу влево если указать несколько позиций конечных одинаковых то поведение непредсказуемо shuffps xmm0, xmm1 00111001b

$$|_{10\rightarrow11}\bigcirc|_{01\rightarrow10}\triangle|_{00\rightarrow01}\triangledown|_{01\rightarrow00}\square| \qquad xmm0$$

по итогу мы последовательно циклом пройдёмся сложим и не будем общаться через шину памяти много раз

* перед просмотром примера можно рассмотреть obj файл через objdump -d main.obj; что то на nm (Linux) для того чтобы разобраться с линковкой нескольких объектных файлов с модификатором Public T, нет t, также есть u - undefined

! важен порядок линковки сначала main.obj а потом multiply.obj

Работа с динамической память

код с комментариями в ТГ!! в итоговой версии конспекта будут ссылки

Строковые команды

1. флаг **DF** будет отвечать за сторону обхода строки(массива)

   (a) **cld** - сбросить DF

   (b) **std** - выставить DF

   Будем считать что **DF** изначально не определён

   (a) **movsb** ; байты 8бит

   (b) **movsw** ; слова 16бит

   (c) **movsd** ; 2 слова 32бит

2. операции передачи строки

   △ копирует массив ds:esi в es:edi, ds, es - сегментные регистры, для **rep** (чуть ниже объяснение) в **ecx** длину массива

   ◯ операндов нет, как и все последующие инструкции работают без операндов

   □ использует ecx, esi, edi, ds, es, eflags при условии что ecx $\neq 0$

```
1 mov [edi], [esi]; псевдокод
2 dec ecx
3 DF = 1: ; то есть если DF выставлен то копирует на -- адресов
4     dec esi
5     dec edi
6 DF = 0: ; то есть если DF не выставлен то копирует на ++ адресов
7     inc esi
8     inc edi
```

3. чтобы многократно вызвать **movsb**|**movsw**|**movsd** нужно использовать **rep movsb**|**movsw**|**mo**
   пока **ecx** $\neq 0$ Тогда скопируется массив ECX символов согласно DF из [ESI] в [EDI]

4. операции сравнения строк:

   (a) **cmpsb**

   (b) **cmpsw**

   (c) **cmpsd**

△ по аналогии с movsb зависит от DF будет сравнивать попарно байты/слова/двойные слова

◯ вызов с **repe** повторять пока ecx $\neq 0$ и пока вхождение символов равны так что в [edi] и [esi] останутся байты разные или строки идентичны

□ вызов с **repne** повторять пока ecx $\neq 0$ и пока вхождение символов не равны

если строки идентичны то **ZF** $= 0$

5. Строковые операции уступают в производительности векторным*
6. Операции сравнения с определённым набором байтов
   (a) **scasb**; образец в al
   (b) **scasw**; образец в ax
   (c) **scasd**; образец в eax

   ◯ DF аналогично, при 1 адреса на уменьшение, при 0 увеличение

   □ Взаимодействует с массивом по адресу es:edi, длина массива всё также в ecx

   △ вызов с **repe** повторять пока ecx $\neq 0$ и пока вхождение символа с образцом равны

   ▽ вызов с **repne** повторять пока ecx $\neq 0$ и пока вхождение символа с образцом не равны
7. операция сохранения в регистр
   (a) **lodsb**
   (b) **lodsw**
   (c) **lodsd**

   сохраняет содержимое [edi] в регистре al|ax|eax
8. операция заполнения
   (a) **stosb**
   (b) **stosw**
   (c) **stosd**

   ◯ DF логика та же

△ берёт из al|ax|eax и кладёт в [es:edi]

□ работает с **rep**, заполнит длиной ecx массив образцами из eax

смотреть пример в ТГ!!

Макросы

1. макроподстановка ⟶ 2. компиляция ⟶ 3. объектный файл

1. Ассесмблер
2. Сюда подставить код - код от макроса
3. текст

Ассемблер прочитаем ASM файл и вставит текст из макроса
причём рекурсивно то есть он и в текст дополнит макросами
сначала он соберёт код полностью с учётом вставок а только потом

```
1 ml ... prog.asm
2 поавитса фа'йл:
3 prog.lst
```

В этом файле хранится информация после макроподстановки
Директивы

1. .listmacro;
2. .nolistmacro;

в макросах console.inc включени .notlistmacro, что значит что после подстановки в <prog>.lst макроса не будет

```
1 има equ выражение; има = вырежение
2 ARR_PARM_OFF = 8+4+4; выражение =(equ) лучше пользоваться = если это ма
     кропеременнаа
3 ARR_PARM_OFF[ebp]; подставит 8+4+4
4 %ARR_PARM_OFF[ebp]; подставит 16
```

Макрос синтаксис

```
1 <Има> macro <список_параметров>;
2 ; <список_параметров> := <параметр> {, <параметр>}
```

Виды параметров:

1. просто <параметр> := **<Имя>**
2. обязательный <параметр> := **<Имя>: req**
3. параметр со значением по умолчанию := **<Имя>:= текст**

как передовать параметры:

```
.BIAKA a, , 12 ; макрос с 3 параметрами 2 параметр - пустое слово|значе
   ние по умолчанию|Выдаст ошибку если параметр req
!, - экранирует запатую
.BIAKA a, <большо'i текст>, b; конструкциа в скобочках воспринимаетса ед
   инственным параметром - строко'i
```

Подстановка в текст

```
    YEAR = 2025
    "Сегодна &YEAR& год"
```

подставит 2025 как число

```
<Има> macro <список_параметров>;
    ;; Это коментари'i с одинарно'i ; не получитса
    тело макроса
endm
```

Конструкция досрочного выхода: **exitm**

метка, конструкция

```
    :after_loop_in_macros ;;метка начинаетса с :

    goto after_loop_in_macros
```

Метки Ассемблера внутри макроса:

```
    local список_меток_дла_макроса;
;; пример
    local mark_err
```

```
4
5       mark_err:
```

название метки будет удалено сохранится лишь число ??000017

конструкция .err, дойдя до блока будет ошибка

```
1       .err текст
2       ...
3       exitm
4       ;; выведет текст компилатором в случае ошибки
```

конструкция echo текст

и %echo выражение соответственно выведет значение или текст во время компиляции

СМ пример в тг

Макросы продолжение

меня не было

Многомодульное программирование

a.asm - головной файл

```
1 ...
2 Start:
3     ...
4     mov var_b, 100
5     ...
6 end Start
```

b.asm

```
1 ...
2     add eax, var_b
3 ...
4     end
```

c.asm

```
1 ...
2     sub var_b, 16
3 ...
4     end
```

1 var_b -> имя предоставляется

2 var_b >- имя испрользуемой в модуле

ИМЯ = символ для линковки

1. **strong** - сильные имена, имя должно быть указано лишь однажды
2. **weak** - слабые имена, имя можно будет переопределить

Конструкция для предоставляения:

```
1 public има переменно'і|функции|метки;
```

Конструкция для получения:

```
1 extern список имён
```

для переменных: имя_переменной: [byte|word|...]

для функций: имя_переменной: [near|proc] - первый смотрит на соглашение

типичный пролог asm файла:

```
1 .686
2 .model flat cdecl|stdcall|...
```

при написании stdcall, cdecl ассемблер переименует названия под стандарт для паскаля ничего не пишем что имена не менялись

.asm → .obj

Секции:

1. .data
2. .data?
3. .code

при компиляции изначально не понять смещение offset var_a, так как всё будет слеплено в один файл после потому существуют заголовки:

имя | размер | где встречается ... ничерта не понял ... неебические иллюстрации не для моего уровня теха

1. Архитектура
2. Размер Секций|их позиций в файле
3. Смещение до точки входа
4. Список динамических подгружаемых библиотек
5. Размер стека
6. С какого адреса размещать программу

Динамически подгружаемые данные.

таблицы этих DLL лок: GOT - Global offset table как правило переменных

PLT - f123: ...

ну тут пиздец

взаимодействие с Pascal

способы взаимодействия:

1. Ассемблерные вставки 2. Интринтики 3. Слинковать с obj Ассемблера

```
1 asm
2 ...
3 ... набор инстукци'i
4 ...
5 end ['eax', 'ecx']
```

в конце список регистров изменённых вставкой

для линковки с obj файлом: {L путь_до_объектного_файла}

Макроархитектура

Теневые реистры

в процессоре существует множество регистров, скрытые от пользователя

Пример когда мы обращаемся к eax нат самом деле мы обращаемся к теневому регистру

Блок команд

Все инструкции попадают в назначенный блок команд

Допустим представим команду так как она хранится в блоке

add byte ptr[edx], 45

преобразуется в

mov тень$_i$, 45

load тень$_j$, edx; тень$_j$ $\leftarrow$ [edx]

add тень$_k$, тень$_j$, тень$_i$

store edx, тень$_k$

команды load и store - самые долгие, тк работают с памятью

***работа с регистрами быстрее чем с константами**

Конвеер * иллюстрация с сборкой самолёта коннвер с сцеплением

груду мусора собрать

собрали элементы

собрали частично

собрали

выкатили

дверки для дополнительных если двигатель в самолёт надо что то стороннее вставить

проблемы - данные на вход неполны ждём, зависимости - ждём кучу пузырей чем дальше переход тем хуже

разгон конвеера - время прохождения от начала и до конца

Доступ в память

load|store

короче общается через иерархию кэшей

кэш комманд и кэш данных отличаются

Предсказание переходов

запихать, но если мы не угадали, то окажется что то что мы вычислили окажется никому не нужным и придётся пустить в конвеер по метке Существует 2 способа ускорения

1. Таблица переходов
2. спекулитивное выполнение

Спекулитивное выполнение возможно при нескольких АЛУ

выполнение происходит одновремнно на двух ветках (то что после jne и после метки) после сравнения подставит нужную ветку, ветка просчитывается в рамках буфера

Глубина предсказаний - число различных исходов

Таблица переходов

Откуда	куда	счётчик	счётчик неуспеха

В тот момент когда процессор доходит до нового перехода от вычисляет по некоторой функции что вытащить из таблички для нового перехода по числу срабатываний и неуспеха
Благодаря табличке мы будем чаще предсказывать что действительно произойдёт по отношению числу срабатываний и неуспехов.

Прерывания

существуют 2 вида прерывания:

1. Внешние прерывания - вызвано внешним устройством
2. Внутренние - вызвано схемой процессора (пример обращение к несуществующему адресу)

У каждого прерывания есть номер прерывания - всего их 256, чего очень мало так как внешних устройств много допустима ситуация, когда от 10 подключенных жётских дисков прерывания будут сливаться в один номер
Прерывания и Исключения - различия

1. Прерывание предразумевает что ошибочная ситуация не произошла
2. Исключение содержит помимо номера прерывания сведения об ошибки

когда случилось прерывание - процессор естественный ход прерывает и выполняет обработку этого прерывания
Регистр прерываний (вектор прерываний) - 256 битный регистр, при происхождение прерывания, он заносится в вектор после обработки соответсвующий бит обнуляется обратно

изначально

контроллер	шина	
IC	ISA	interrupt controller ранее внешние устр
PIC	PCI	Pragramm I
APIC	PCI	Advanced создан с учётом работы между ядрами и про
MSI	PCI|PCI express	устройство отправялет некоторое сообщение

регистр флагов - IF (Interrupt Flags) IF = 0 - на прерывение не реагирует
IF = 1 - на прерывание регирует
cli sti - ставит 0 и 1 соответсвенно
Idtr Idt - таблицы
Idtr - interrupt description table register - содержит адрес таблицы и размер (регистр)
idt - interrupt description table
поля IDT:

1 используется или нет

2 адрес обработчика прерываний(виртуальный)

3 тип обработчика прерываний

4‘ номер сегмента кода (где лежит обработчик прерываний) в GDT

4“ Номер TSS (task save segment) сегмента

5 DPL - desired privilege level

6 размер адреса

при возникновении прерывания:

1. на старом стеке
2. на новом стеке

В зависимости от уровни привелегий процессор сам может сбрасывать IF или нет
на старом стеке: программа выполнялась дошла до N иструкции произошло прерывание

1. старый CS, eflags и EIP кладём на стек в случае ошибки - информация о ней.
2. сменили сегмент на тот что в табличке (CS)
3. сменили EIP на тот что в табличке
4. CLI - аппаратно

*иллюстрация
на новом стеке - всё то же самое но сохраняется SS и ESP сохраним: SS ESP CS eflags EIP *сообщение об ошибке*
также используется iret новый стек возьмём из TSS
в прерывании необходимо сохранить все регистры в случае использования TSS не надо.