* + 1. **Визначити цілі та задачі системного програмування**

Поява мікропроцесорів у 60-х роках пов’язана з розробкою інтегральних схем. Інтегральні схеми поєднали у собі різноманітні електронні компоненти в єдиний елемент на силіконовому чіпи. Розробники встановили його в пристрій і включили до функціонуючих систем. На початку 70-х мікрокомпонентиIntel8008 об’явили про появу першого покоління мікропроцесорів.

До 1974 року з’явилось друге покоління мікропроцесорів загального призначенняIntel8080. Цей успіх пробудив інші фірми до виробництва процесорівIntel8080 або аналогічних.

В 1978 році фірмаIntelвипустила трете покоління процесорів –Intel8086, який забезпечував деяку сумісність з 8080 і був значним прогресом у даній сфері. Для підтримки більш простих пристроїв і забезпечення сумісності з пристроями вводу-виводу того часу Intel розробив різновид процесора 8086 – процесор 8088, який у 1981 році був обраний фірмою ІБМ для її персональних комп’ютерів.

Більш розвиненими версіями є для процесору 8088 – процесор 80188, а для процесора 8086 – процесори 80186, 80286 і 80386, які забезпечили додаткові можливості і підвищили потужність обчислень. Мікропроцесор 80286, встановлений в комп’ютерах ІБМ АТ, з’явився в 1984 році. Всі ці процесори мають відношення до розвинутої архітектури процесорів фірми Intel і позначаються як іАРХ 86, іАРХ 88, іАРХ 186, іАРХ 286 і іАРХ 386, де іАРХ - IntelAdvancedProcessorArchitecture.

Розповсюдження мікрокомп’ютерів послугувало причиною для перегляд у ставлення до мови Ассемблер за двома основними причинами. По-перше, програми, написані на мові Ассемблер, потребують значно меншого обсягу пам’яті і часу виконання. По-друге, знання мови Ассемблер і результуючого машинного коду дає розуміння архітектури машини, що навряд чи забезпечується при роботі з мовами високого рівня. Хоча більшість спеціалістів в сфері програмного забезпечення ведуть розробки на мовах високого рівня, таких, як Паскаль і Си, що простіше при написанні програм, але найбільш потужне і ефективне програмне забезпечення повністю або частково написано на мові Ассемблер.

Мови високого рівня були розроблені для того, щоб уникнути спеціальної технічної особливості певних комп’ютерів. Мова асемблер, в свою чергу, розроблена для конкретної специфіки комп’ютера або, точніше, для специфіки процесора.

2.2.1 **Охарактеризувати принцип програмного управління**

За Джорджем фон Нейманом робота персонального комп’ютера базується на декількох принципах, а саме:

1. Принцип однорідності
2. Принцип адресності
3. Принцип програмного управління
4. Принцип двійкового кодування

Принцип програмного управління полягає в тому, що всі обчислення, передбачені алгоритмів вирішення задачі, мають бути представлені у вигляді програми, що складається з послідовності керуючих слів – команд. Кожна команда описує деяку операцію із набору операцій, що реалізуються обчислювальною машиною. Команди програми зберігаються в послідовних комірках пам’яті обчислювальної машини і виконуються в природній послідовності, тобто в порядку їх розташування у програмі. При необхідності, за допомогою спеціальних команд, цю послідовність можна змінювати. Рішення про зміну послідовності виконання команд програми приймається або на основі аналізу попередніх обчислень, або безумовно.

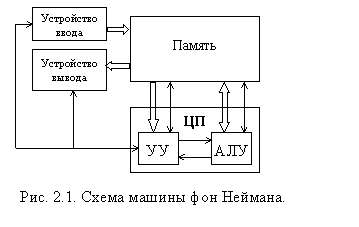
3.3.1 **Охарактеризувати архітектуру фон Неймана базуючись на машині Тьюрінга**

В 1946 році Джон фон Нейман описав архітектуру деякого абстрактного обчислювача, який зараз прийнято називати машиною фон Неймана. Ця машина є абстрактною моделлю ЕОМ, проте, ця абстракція відрізняється від абстрактних виконавців алгоритмів (наприклад машини Тьюрінга). Якщо машину Тьюрінга принципово не можна реалізувати через наявну в її архітектурі безкінечну стрічку, то машина фон Неймана не піддається реалізації, оскільки багато деталей в її архітектурі не конкретизовані.

В деякому сенсі, машина фон Неймана подібна до абстрактних структур даних. Для таких структур необхідно виконати відображення на структури даних зберігання і реалізувати відповідні операції над цими даними.

Можна сказати, що в машині фон Неймана зафіксовані ті особливості архітектури, які в тій чи іншій мірі мають бути присутні у всіх комп’ютерах. Практично всі сучасні комп’ютери по своїй архітектурі відмінні від машини фон Неймана, але ці відмінності зручно вивчати саме як відмінності,проводячи порівняння і співставлення з машиною фон Неймана. Основні властивості архітектури фон Неймана сформовані у вигляді принципів фон Неймана, а саме:

1. Принцип однорідності
2. Принцип адресності
3. Принцип програмного управління
4. Принцип двійкового кодування



На цьому малюнку товстими стрілками позначено потоки команд і даних, а тонкими – передача між пристроями керуючих сигналів. Машина фон Неймана складається з пам’яті, пристроїв вводу/виводу і центрального процесора. Центральний процесор, в свою чергу, складається з пристрою керування і арифметико-логічного пристрою.

4.4.1 **Визначити типові групи команд сучасних процесорів**

В загальному випадку система команд процесора включає наступні чотири основні групи команд:

* команди пересилки даних;
* арифметичні команди;
* логічні команди;
* команди переходів.

Команди пересилки даних не вимагають виконання ніяких операцій над операндами. Операнди просто пересилаються (точніше, копіюються) з джерела (Source) в приймач (Destination). Джерелом і приймачем можуть бути внутрішні регістри процесора, елемента пам'яті або пристрою вводу/виводу. АЛП в даному випадку не використовується.  
Арифметичні команди виконують операції додавання, віднімання, множення, ділення, збільшення на одиницю (інкрементування), зменшення на одиницю (декрементування) і т.д. Цим командам потрібні один або два вхідні операнди. Формують команди один вихідний операнд.  
Логічні команди проводять над операндами логічні операції, наприклад, логічне І, логічне АБО, виключаюче АБО, очищення, інверсію, різноманітні зсуви (вправо, вліво, арифметичний зсув, циклічний зсув). Цим командам, як і арифметичним, потрібні один або два вхідні операнди, і формують вони один вихідний операнд.  
Нарешті, команди переходів призначені для зміни звичайного порядку послідовного виконання команд. З їх допомогою організовуються переходи на підпрограми і повернення з них, всілякі цикли, розгалуження програм, пропуски фрагментів програм і т.д. Команди переходів завжди міняють вміст лічильника команд. Переходи можуть бути умовними і безумовними. Саме ці команди дозволяють будувати складні алгоритми обробки інформації.  
Відповідно до результату кожної виконаної команди встановлюються або очищаються біти регістра стану процесора (PSW). Але треба пам'ятати, що не всі команди змінюють прапори PSW. Це визначається особливостями кожного конкретного процесора.  
У різних процесорів системи команд істотно розрізняються, але в основі своїй вони дуже схожі. Кількість команд у процесорів також різна. Наприклад, у згадуваного вже процесора МС68000 всього 61 команда, а у процесора 8086 - 133 команди. У сучасних могутніх процесорів кількість команд досягає декількох сотень. В той же час існують процесори з скороченим набором команд (так звані RISC-процесори), в яких за рахунок максимального скорочення кількості команд досягається збільшення ефективності і швидкості їх виконання.

5.5.1 **Обґрунтувати переваги та недоліки сегментної організації пам’яті**

Сегментація пам'яті дає змогу зображати логічний адресний простір як сукуп­ність незалежних блоків змінної довжини, які називають сегментами. Кожний сегмент звичайно містить дані одного призначення, наприклад в одному може бу­ти стек, в іншому — програмний код і т. д.

У кожного сегмента є ім'я і довжина (для зручності реалізації поряд з іменами використовують номери). Логічна адреса складається з номера сегмента і зсуву всередині сегмента; з такими адресами працює прикладна програма. Компілятори часто створюють окремі сегменти для різних даних програми (сегмент коду, сег­мент даних, сегмент стека). Під час завантаження програми у пам'ять створюють таблицю дескрипторів сегментів процесу, кожний елемент якої відповідає одно­му сегменту і складається із базової адреси, значення межі та прав доступу.

Під час формування адреси її сегментна частина вказує на відповідний еле­мент таблиці дескрипторів сегментів процесу. Якщо зсув більший, ніж задане значення межі (або якщо права доступу процесу не відповідають правам, заданим для сегмента), то апаратне забезпечення генерує помилку. Коли ж усе гаразд, сума бази і зсуву в разі чистої сегментації дасть у результаті фізичну адресу в основ­ній пам'яті. Якщо сегмент вивантажений на диск, спроба доступу до нього спричи­няє його завантаження з диска в основну пам'ять. У підсумку кожному сегменту відповідає неперервний блок пам'яті такої самої довжини, що перебуває в довіль­ному місці фізичної пам'яті або на диску. Загальний підхід до перетворення адре­си у разі сегментації показаний на рис. 1.

**Переваги сегментації пам'яті:**

* З'явилася можливість організувати кілька незалежних сегментів пам'яті для процесу і використати їх для зберігання даних різної природи. При цьому права доступу до кожного такого сегмента можуть бути задані по-різному.
* Окремі сегменти можуть спільно використовуватися різними процесами, для цього їхні таблиці дескрипторів сегментів повинні містити однакові елементи, що описують такий сегмент.
* Фізична пам'ять, що відповідає адресному простору процесу, тепер не обо­в'язково має бути неперервною. Справді, сегментація дає змогу окремим части­нам адресного простору процесу відображатися не в основну пам'ять, а на диск, і довантажуватися з нього за потребою, забезпечуючи виконання проце­сів будь-якого розміру.

**Недоліки сегментації пам’яті:**

* Необхідність введення додаткового рівня перетворення пам'яті спричиняє зни­ження продуктивності (цей недолік властивий будь-якій повноцінній реаліза­ції віртуальної пам'яті). Для ефективної реалізації сегментації потрібна відпо­відна апаратна підтримка.
* Керування блоками пам'яті змінної довжини з урахуванням необхідності їх­нього збереження на диску може бути досить складним.
* Вимога, щоб кожному сегменту відповідав неперервний блок фізичної пам'яті відповідного розміру, спричиняє зовнішню фрагментацію пам'яті. Внутріш­ньої фрагментації у цьому разі не виникає, оскільки сегменти мають змінну довжину і завжди можна виділити сегмент довжини, необхідної для виконан­ня програми.

Сьогодні сегментацію застосовують доволі обмежено передусім через фраг­ментацію і складність реалізації ефективного звільнення пам'яті та обміну із дис­ком.

6.6.1 **Дати характеристику структури програм на мові Асемблер на сучасних ПЕОМ**

Програма на асемблері являє собою сукупність блоків пам'яті, називаних *сегментами пам'яті*. Програма може складатися з одного або кількох таких блоків-сегментів. Кожен сегмент містить сукупність речень мови, кожне з яких займає окремий рядок коду програми. Рядки закінчуються символом CR (вводиться клавішею Enter). Рядки можна розділити на чотири типи:

* + порожні рядки;
  + рядки – коментарі;
  + директиви Асемблера (їх називають також командами Асемблера або псевдокомандами);
  + машинні інструкції – символічне зображення команд, які виконуються ЕОМ.

Порожні рядки можуть містити тільки символи пробілу чи табуляції. Вони потрібні для наглядної структуризації програм мовою Асемблера.

Рядки – коментарі можуть містити початкові пробіли або символи табуляції, далі символ ‘;’, а за ним – довільні символи.

Директиви Асемблера служать для структуризації програм, резервування пам’яті, завдання даних та управління компілятором.

Машинні інструкції (або машинні команди) служать для символічного відображення команд ЕОМ.

7.7.1 **Проаналізувати особливості налагодження програм на мові Асемблер**

На етапі налагодження, використовуючи опис алгоритму, виконується контроль правильності функціонування як окремих ділянок коду, так і всієї програми в цілому. Але навіть успішне закінчення налагодження ще не є гарантією того, що програма будет працювати правильно з усіма можливими вихідними даними. Тому потрібно обов’язково провести тестування програми, тобто перевірити її роботу на «граничних» і свідомо некоректних вихідних даних. У випадку виявлення помилок прийдеться висоти виправлення в ассемблерний код програми, тобто повертатися до першого кроку процесу розробки.

Для налагодження програм використовується програмне забезпечення спеціального типу – відлагоджвачі. Вони працюють в інтерактивному режимі, дозволяють виконувати програму на різних наборах даних тощо.

Головле, що відлогоджувач дозволяє – це:

1. визначити місцезнаходження логічної помилки.
2. З’ясувати причину помилки..

Відомі різні автономні або наявні в середовищі програмування відлагоджувачі. Для наших цілей найбільш придатні, наприклад, програма turbo debugger(TD) з пакету tasm, програма code view(cv) з пакету masm, відлаоджувач AFD тощою

Командний рядок для виклику відлагоджувача td.exe може мати такий вигляд: td prg\_1

Докладнішу інформацію про поботу певного відлагоджувача можна знайти у відповідних керівництвах, меню Help, підручниках по Ассемблеру.

Відлагоджувач є обов’язковим інструментом тестування ассемблрних програм для програміста. Вивчення особливостей роботи з ним є часткою самостійної роботи студента.

**8.8.1 Дати характеристику директивам визначення даних в мові Асемблера процесорів**

Для визначення даних використовуються, головним чином, три директиви асемблера: db (definebyte, визначити байт) для запису байтів, dw (defineword, визначити слово) для запису слів і dd (definedouble, визначити подвійне слово) для запису подвійних слів.

onedbdb 255   
twodw dw 6.5535   
threedd dd 100000000

Окрім перерахованих, є і інші директиви, наприклад df (definefanvord, визначити поле з 6 байт), dq (definequadword, визначити четверне слово) або dt (definetcraword, визначити 10-байтову змінну), але вони використовуються значно рідше.  
Окрім перерахованих, є і інші директиви, наприклад df (definefanvord, визначити поле з 6 байт), dq (definequadword, визначити четверне слово) або dt (definetcraword, визначити 10-байтову змінну), але вони використовуються значно рідше.  
Для того, щоб до даних можна було звертатися, вони повинні мати імена. Імена даних можуть включати латинські букви, цифри (не як перший знак імені) і деякі спеціальні знаки, наприклад, знаки підкреслення (\_), долара ($) і комерційного at (@).Значення числових даних можна записувати в різних системах числення; частіше за іншідля запису використовуються десяткова і 16-річна системи

Привласнення даним символічних імен дозволяє звертатися до них в програмних пропозиціях, не піклуючись про фактичні адреси цих даних. Наприклад, команда

**mov** Ax,size

занесе в регістр АХ вміст осередку size (число 256), незалежно від того, в якому місці сегменту даних цей осередок визначений, і в яке місце фізичної пам'яті вона потрапила..

**9.9.1 Визначити взаємозв’язок між структурою команд процесора 8086 та структурою машинної інструкції на мові Асемблер**

Можливості машино-орієнтованої мови визначаються властивостями відповідної ЕОМ і структурою команд її центрального процесора.

Програма, записана на якому або мовою програмування, називається вихідним модулем, який повинен бути перетворений в машинний код.

Процес перетворення вихідного модуля в машинний код називається трансляцією, а сама програма здійснює трансляцію ***транслятором***.

***Асемблер*** це транслятор з мови низького рівня або мови асемблера (макроасемблера).  
Мова асемблера - машино-орієнтована мова, тобто якщо різні МП мають різну архітектуру, то вони програмно не сумісні

***Асемблер*** відрізняється від машинного мови тим, що:

1. містить макросредства;
2. команди операцій записуються мнемонічні;
3. адреси записуються символічно, а не чисельно.

Програми, написані на мові Асемблера, можуть безпосередньо транслюватися на машинний мову за допомогою програми званої асемблером.

Асемблер дозволяє розробляти найбільш ефективні програми з мінімальним часом рахунку і найбільш повно використовувати технічні можливості ПК, але при цьому процес складання програми є трудомістким і вимагає певних знань.

**10.10.1 Охарактеризувати використання програмно доступних регістрів в системі команд процесорів 8086**

Процесори 8086/8088 мають 14 регістрів, що використовуються для керування програмою, для адресації пам’яті і для забезпечення арифметичних обчислень.

Кожен регістр має довжину в одне слово(16 біт) і адресується за ім’ям.

Сегментні регістри: CS, DS, SS та ES

Кожен сегментний регістр забезпечує адресацію памяті обсягом 64Кб, яка називається поточним сегментом.

* 1. Регістр CS. Регістр сегмента кода містить початкову адресу сегмента кода. Ця адреса плюс значення зміщення в командному вказівнику(IP)визначає адресу команди, яка має бути обрана до виконання. Для звичайних програм не має сенсу робити посилання на регістр CS.
  2. Регістр DS. Регістр сегмента даних містить початкову адресу сегменту даних. Ця адреса плюс зміщення, визначене в команді, вказує на конкретну комірку в сегменті даних.
  3. Регістр SS. Регістр сегмента стека містить початкову адресу сегмента стека.
  4. Регістр ES. Деякі операції зі строками використовують додатковий сегментний регістр для керування адресацією пам’яті. В даному контексті регістр ES пов'язаний з індексним регістром DI. Якщо необхідно використовувати регістр ES, програма має його ініціалізувати.

Регістри загального призначення: AX, BX, CXтаDX

При програмуванні на мові Асемблер використовуються регістри загального призначення . Особливість цих регістрів полягає в тому, що можлива адресація їх як окремого слова або як однобайтової частини. Лівий байт є старшою частиною, а правий – молодшою частиною.

1. Регістр AX. Регістр AX є основним суматором і використовується для усіх операцій вводу-виводу, деяких операцій над строками і деяких арифметичних операцій. Наприклад, команди множення, ділення і зсуву припускають використання регістру AX. Деякі команди генерують більш ефективний код, якщо вони мають посилання на регістр AX.
2. Регістр BX. Регістр BX є базовим регістром. Це єдиний регістр загального призначення, який може використовуватись в якості «індексу» для розширеної адресації. Інше його загальне призначення – обчислення.
3. Регістр CX. Регістр CX є лічильником. Він необхідний для керування кількістю повторень циклу і для операцій зсуву вліво чи вправо. Регістр CX також використовується для обчислень.
4. РегістрDX. Регістр DX є регістром даних. Він використовується для деяких операцій вводу/виводу і тих операцій множення і ділення над великими числами, які використовують регістрову пару DX:АX.

Будь-які регістри загального призначення можуть брати участь в операціях додавання і віднімання як 8-, так і 16-бітових значень.

Регістрові вказівники: SP таBP

Регістрові вказівники забезпечують системі доступ до даних в сегменті стека. Рідше вони використовуються для операцій додавання і віднімання.

1. Регістр SP. Вказівник стека забезпечує використання стека в памяті, дозволяє тимчасово зберігати адреси та інколи дані. Цей регістр пов'язаний з регістром SS для адресації ст+ека.
2. Регістр BP. Вказівник бази полегшує доступ до параметрів( даним та адресам, переданим через стек).

Індексні регістри:BP таDI

Обидва індексних регістра можуть використовуватись для розширеної адресації і для використання в операціях додавання і віднімання.

1. РегістрBP. Цей регістр є індексом джерела і застосовується для деяких операцій над строками. В даному контексті регістр SI пов'язаний з регістром DS.
2. Регістр DI. Цей регістр є індексом призначення і застосовується також для строкових операцій. У даному контексті регістр DI пов'язаний з регістром ES.

Регістр командного вказівника: IP

Регістр IP містить зміщення на команду, яка має бути виконана. Зазвичай цей регістр в програмі не використовується, але він може змінювати своє значення при використанні налагоджувача при тестуванні програми.

Прапорцевий регістр

9 із 16 біт прапорцевого регістра є активними і визначають поточний стан машини і результати виконання. Багато арифметичних команд і команд порівняння змінюють стан прапорців.

**11.11.1 Проаналізувати використання режимів адресації даних в програмах на мові Асемблера ПЕОМ.**

Переважна більшість команд процесорів 80х86 мають адресну частину, яка у загальному випадку містить байти modr/m, sibта зміщення в команді. На основі цих даних процесор формує зміщення в сегменті, яке в даному випадку назвали ефективною адресою. У загальному випадку ефективна адреса є сумою трьох компонент – зміщення в команді, бази та індексу. База та індекс містяться в регістрах загального призначення, які використовуються як адресні регістри. Індекс може мати множник 2,4 або 8, який показує, на яку величину необхідно помножити вміст 32-розрядного індексного регістра перед формуванням ефективної адреси (для 16-розрядних регістрів множник не задається). Будь-яка з компонент в адресному виразі може бути відсутня, що визначає наступні режими адресації (табл. 3.2):

Таблиця 3.2 - Режими адресації даних

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Зміщення в команді** | **База** | **Індекс** | **Режим адресації** | **Приклад** |
| **-** | **-** | **+** | Посередня регістрова | [si], [еах]  [esp] |
| **-** | **+** | **+** | Базова індексна | [bx+si],  [ecx+edx]  [ebx+esi\*4] |
| **-** | **+** | **-** | Посередня регістрова | [bx], [ecx] |
| **+** | **-** | **+** | Індексна | Dat1[si]  Dat1[ecx]  Dat1[edi\*8] |
| **+** | **-** | **-** | Пряма | Dat1 |
| **+** | **+** | **+** | Базова індексна зі зміщенням | Dat1[bx+di]  Dat1[ebx][edx]  Dat1[edx+esi\*2] |
| **+** | **+** | **-** | Базова | [bp+4], [bp-6]  [ecx+7] |

Примітка 1. Для формування ефективної адреси можуть використовуватись лише регістри BP, BX, SI і DI , а також лише наступні їх пари: BX+SI, BX+DI, BP+SI та BP+DI. Для 32-розрядних регістрів загального призначення таке обмеження на їх використання відсутнє, за виключенням регістра ESP – вінне може задаватись із множником.

Примітка 2. Мовою Асемблера можна задати посередню регістрову адресацію із множником, наприклад, Add eax,[edx\*4], але в процесорі такі команди відсутні. Асемблер сформує машинну команду, в якій зміщення в команді буде мати нульове значення.

При програмуванні мовою Асемблера розглянуті вище режими адресації доцільно використовувати наступним чином:

* пряму– для адресації простих (скалярних) даних, адреси яких при виконанні програми не змінюються;
* посередню регістрову - для адресації скалярних даних, адреси яких змінюються при виконанні програми;
* індексну– для адресації елементів масивів. Особливо ефективно можна використовувати індексну адресацію для доступу до елементів одновимірних масивів байтів, масивів слів, подвійних та квадро слів;
* базову **–** для адресації елементів структур, відносні адреси яких при виконанні програми не змінюються;
* базову індексну – для адресації елементів структур, відносні адреси яких змінюються при виконанні програми;
* базову індексну зі зміщенням – для адресації елементів масивів структур або для адресації елементів багатовимірних масивів.

**12.12.1 Проаналізувати можливості групи команд обробки масивів на мові Асемблера ПЕОМ**

Група команд обробки одновимірних масивів

Усі команди цієї групи є однобайтними, вони не мають адресної частини. Адреси операндів команд задаються не явно (фактично кодом операції). Кожна із цих команд має два операнда – або обидва в пам’яті, або один в акумуляторі, а другий в пам’яті (табл. 3.4 )

Табл. 3.4 - Команди обробки рядків

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Машинний код команди | Мнемоніка | Приймач | Джерело | Призначення |
| 0A4h | MOVSB | Пам’ять | пам’ять | Пересилання елементів масивів |
| 0A5h | MOVSW або  MOVSD | Пам’ять | пам’ять | Пересилання елементів масивів |
| 0A6h | СMPSB | Пам’ять | пам’ять | Порівняння елементів масивів |
| 0A7h | СMPSW або  СMPSD | Пам’ять | пам’ять | Порівняння елементів масивів |
| 0AEh | SCASB | Пам’ять | Акумулятор | Порівняння елемента масиву з даними в акумуляторі |
| 0AFh | SCASW або  SCASD | Пам’ять | Акумулятор | Порівняння елемента масиву з даними в акумуляторі |
| 0AAh | STOSB | Пам’ять | Акумулятор | Занесення в елемент масиву даних з акумулятора |
| 0ABh | STOSW або  STOSD | Пам’ять | Акумулятор | Занесення в елемент масиву даних з акумулятора |
| 0ACh | LODSB | Акумулятор | пам’ять | Занесення в акумулятор елемента масиву |
| 0ADh | LODSW або  LODSD | Акумулятор | пам’ять | Занесення в акумулятор елемента масиву |

Особливість цих команд полягає в наступному:

1. Якщо операнд розташований у пам’яті, то у випадку джерела його логічна адреса – DS:SI, а у випадку приймача – ES:DI.
2. Після виконання команди вміст використаних регістрів (SI та/або DI) залежно від типу (*b* – byte, *w* – word, *d* – dword) автоматично змінюється (збільшується або зменшується) на 1, 2 або 4.
3. Збільшення відбувається, коли ознака *df* в регістрі ознак встановлена в 0; якщо ознака *df=1*, тоді відбувається автоматичне зменшення.
4. Перед командами може використовуватись префікс повторення для організації апаратних циклів.

Префікси повторення в мові Асемблера задаються тільки явно як окрема машинна інструкція або безпосередньо в полі мнемокоду ланцюгової команди. Мнемоніки префіксів повторення: *REP*, *REPZ* (або *REPE*), *REPNZ* (або *REPNE*). Префікс *REP* задається перед ланцюговими командами, які не виконують порівняння. Алгоритм префікса REP наступний:

1. якщо CX=0 (або ECX=0), то вийти з циклу;
2. виконати наступну ланцюгову команду;
3. виконати декремент регістра CX (ECX) і перейти до п.1)

Префікси *REPZ* (або *REPE*), *REPNZ* (або *REPNE*) задаються перед ланцюговими командами, які виконують порівняння. Алгоритм їхньої роботи наступний:

1. якщо CX=0 (або ECX=0), тоді вийти з циклу;
2. виконати декремент регістра CX (ECX);
3. виконати наступну ланцюгову команду;
4. якщо *zf=1* для REPZ (REPE) або *zf=0* для REPNZ (REPNE), тоді перейти до п.1)

**13.13.1 Охарактеризувати суть та причини використання стека у програмах.**

Стеком называют область программы для временного хранения произвольных данных. Разумеется, данные можно сохранять и в сегменте данных, однако в этом случае для каждого сохраняемого на время данного надо заводить отдельную именованную ячейку памяти, что увеличивает размер программы и количество используемых имен. Удобство стека заключается в том, что его область используется многократно, причем сохранение в стеке данных и выборка их оттуда выполняется с помощью эффективных команд push и pop без указания каких-либо имен.  
Стек традиционно используется, например, для сохранения содержимого регистров, используемых программой, перед вызовом подпрограммы, которая, в свою очередь, будет использовать регистры процессора "в своих личных целях". Исходное содержимое регистров извлекается из стека после возврата из подпрограммы. Другой распространенный прием - передача подпрограмме требуемых ею параметров через стек. Подпрограмма, зная, в каком порядке помещены в стек параметры, может забрать их оттуда и использовать при своем выполнении.  
Отличительной особенностью стека является своеобразный порядок выборки содержащихся в нем данных: в любой момент времени в стеке доступен только верхний элемент, т.е. элемент, загруженный в стек последним. Выгрузка из стека верхнего элемента делает доступным следующий элемент.  
Элементы стека располагаются в области памяти, отведенной под стек, начиная со дна стека (т.е. с его максимального адреса) по последовательно уменьшающимся адресам. Адрес верхнего, доступного элемента хранится в регистре-указателе стека SP. Как и любая другая область памяти программы, стек должен входить в какой-то сегмент или образовывать отдельный сегмент. В любом случае сегментный адрес этого сегмента помещается в сегментный регистр стека SS. Таким образом, пара регистров SS:SP описывают адрес доступной ячейки стека: в SS хранится сегментный адрес стека, а в SP - смещение последнего сохраненного в стеке данного

Для роботи зі сегментом стека визначено дві основні команди – це команди PUSH іPOP. Команда PUSH записує в стек двухбайтові адреси та інші значення. КомандаPOP зазвичай вибирає із стека записані у нього слова. Ці операції змінюють відносну адресу в регістрі SP( вказівник стеку) для доступу до наступного слова. Така властивість стеку вимагає, щоб команди RETі CALL відповідали один одному. Крім того, викликана процедура може викликати за допомогою команди CALL іншу процедуру, а та, в свою чергу, - наступну. Стек має мати достатні розміри для того, щоб зберігати усі записувані до нього адреси.

Команди PUSH, PUSHF, CALL, INTта INTO заносять в стек адрес повернення і, можливо, вміст прапорцевого регістру. Команди POP, POPF, RETтаIRET виймають ці адреси і прапорці із стеку.

**14.14.1 Визначити особливості адресації в командах передачі управління на мові Асемблер**

Команди передачі управління з точки зору мови Асемблера розподіляються на команди з прямою адресацією (прямі), коли в операнді команди вказується мітка або ім’я процедури, та опосередковані (непрямі), коли в операнді команди вказується адреса даних, які містять адресу переходу.

Команди передачі управління розподіляються також на внутрішньо сегментні та між сегментні. Внутрішньо сегментні прямі команди передачі управління з точки зору процесора мають відносну адресацію, коли цільова адреса формується як алгебраїчна сума вмісту (E)IP та зміщення в команді.

Команди передачі управління по умові можуть мати одну із наступних структур

|  |  |
| --- | --- |
| Один байт коду операції | Один байт зміщення в команді |
| Два байти коду операції | Два байти зміщення в команді |

Визначається значення зміщення адресного терму в полі операндів.

На першому перегляді алгоритм обробки команди передачі управління по умові наступний.

1. Визначається наявність мітки в полі операндів. Якщо в таблиці ідентифікаторів користувача відповідна мітка відсутня, то перейти до п.6
2. Читається значення мітки із таблиці і якщо операнд має абсолютний вираз то він обчислюється і його значення добавляється до значення зміщення.
3. Обчислюється різниця між одержаним значенням та значенням поля «Поточне зміщення» відкритого логічного сегменту ($).
4. Якщо різниця менша за -128 або більша за 127, то перейти до п.6
5. Прийняти Кб=2. Перейти до п.
6. Прийняти Кб=4

Таким чином, при посиланнях вперед безумовно використовується 4-х байтні команди, незалежно від відстані до мітки.

На другому перегляді всі мітки визначені, тому для визначення посилання вперед використовується порівняння зміщення цільової мітки (Зм) та значення $. Якщо Зм>$, то безумовно генерується 4х-байтна команда, інакше – двохбайтна. Для генерації двохбайтного зміщення команди передачі управління використовується значення виразу Зм-$-4, а двохбайтного – значення виразу +Зм-$-2.

**15.15.1 Дати порівняльну характеристику використання на мові Асемблер процедур і макросів**

Макросы- это еще один способ однократного описания последовательности действий и затем многократного их выполнения с различными параметрами. Основным отличием от использования процедур заключается в том, что макросы- это директивы ассемблеру (обрабатываются они при помощи препроцессорной обработки).

Достоинства:

• они более универсальны, т.к. позволяют параметрически управлять не только заданием обрабатываемых объектов, но и действиями над этими объектами;

• их использование не связано с выполнением команд CALL и RETURN, поэтому применение макросов ускоряет выполнение программы;

• поскольку описание макросов воспринимается как обычный текст при подстановке, то легко организуются библиотеки макросов.

Недостатки:

• существенное увеличение длины программы, связанное с подстановкой тела макроса во все точки его вызова.

Использование макросов связано с понятиями макроопределение, макровызов, макрорасширение.

Макроопределение- описание действия, выполняющегося макросом применительно к фиксированным параметрам; состоит из заголовка, тела и концовки.

Заголовок: Nam\_Macro MACRO [список фиктивных параметров]

Тело: последовательность операторов

Концевик: END [Nam\_Macro]

Встречаются ассемблеры, которые предоставляют мнемонические обозначения для часто используемых групп команд. Большинство таких языков позволяет пользователю вводить свои собственные мнемонические обозначения – так называемые **макроопределения** или **макросы** (**macros**), в том числе и параметризованные (пример 2.7).

Отличие макроопределений от процедур языков высокого уровня в том, что процедура компилируется один раз, и затем ссылки на нее реализуются в виде команд вызова. Макроопределение же реализуется путем подстановки тела макроопределения (с заменой параметров) на место ссылки на него и компиляцией полученного текста. Компиляция ассемблерного текста, таким образом, осуществляется в два или более проходов – на первом осуществляется раскрытие макроопределений, на втором – собственно компиляция, которая, в свою очередь, может состоять из многих проходов, смысл которых мы поймем далее. Часть ассемблера, реализующая первый проход, называется макропроцессором.

**16.16.1 Охарактеризувати макроси в програмах на мові Асемблер**

Одно из самых мощных языковых средств ассемблера — макроопределения. *Макроопределением* (или макросом) называется участок программы, которому присвоено имя и который ассемблируется всякий раз, когда ассемблер встречает это имя в тексте программы. Макрос начинается директивой MACRO и заканчивается ENDM. Например: пусть описано макроопределение hex2ascii, переводящее шестнадцатеричное число, находящееся в регистре AL, в ASCII-код соответствующей шестнадцатеричной цифры:

hex2ascii macro

cmp al,10

sbb al,69h

das

endm

Теперь в программе можно использовать слово hex2ascii, как если бы это было имя команды, и ассемблер заменит каждое такое слово на три команды, содержащиеся в макроопределении. Разумеется, можно оформить этот же участок кода в виде процедуры и вызывать его командой CALL — если процедура вызывается больше одного раза, этот вариант программы займет меньше места, но вариант с макроопределением станет выполняться быстрее, так как в нем не будет лишних команд CALL и RET. Однако скорость выполнения — не главное преимущество макросов. В отличие от процедур макроопределения могут вызываться с параметрами, следовательно, в зависимости от ситуации, включаемый код будет немного различаться, например:

s\_mov macro register1,register2

push register1

pop register2

endm

Теперь можно использовать S\_MOV вместо команды MOV для того, чтобы скопировать значение из одного сегментного регистра в другой.

Следующее важное средство, использующееся в макроопределениях, — директивы условного ассемблирования. Например: напишем макрос, выполняющий умножение регистра AX на число, причем, если множитель — степень двойки, то умножение будет выполняться более быстрой командой сдвига влево.

fast\_mul macro number

if number eq 2

shl ax,1 ; Умножение на 2

elseif number eq 4

shl ax,2 ; Умножение на 4

elseif number eq 8

shl ax,3 ; Умножение на 8

... ; Аналогично вплоть до:

elseif number eq 32768

shl ax,15 ; Умножениена 32768

else

mov dx,number ; Умножение на число, не являющееся

mul dx ; степенью двойки.

endif

endm

Можно, конечно, усложнить этот макрос, применяя особые свойства команды LEA и ее комбинации, сдвиги и сложения, однако в нынешнем виде он чрезмерно громоздкий. Проблема решается с помощью третьего средства, постоянно использующегося в макросах, — блоков повторений.

17.17.1 Охарактеризувати процес обробки макровизначень компілятором Асемблера -

18.18.1 Охарактеризувати процес обробки макрокоманд компілятором Асемблера

Makros (макрос) - греческое слово, обозначающее "большой" или "длинный". В Кратком оксфордском словаре английского языка сказано, что от него происходит префикс macro. В литературе по программированию macro обычно употребляется как синоним слова "макрокоманда".

Макрокоманды появились в пятидесятых годах прошлого века как средство, расширения возможностей языка ассемблера и упрощения программирования на этом языке. В настоящее время они применяются в различных языках и системах программирования.

Макросы являются полезным и универсальным средством широкого назначения, упрощающим программирование. Тем не менее, в практике программирования они применяются не так часто, как того заслуживают. В основном их разрабатывают, распространяют и используют в своих программах специалисты высокой квалификации. Возможно, это объясняется отсутствием специальной литературы по программированию макросов, или ограниченным доступом к такой литературе.

Цель данной статьи обратить внимание программистов, работающих с ассемблером, на существование специальных средств, предназначенных для составления макросов и показать примеры их применения. Для проверки приведенных примеров версия MASM (Microsoft Macro Assembler) должна быть не ниже чем 6.0.

Общие положения

Макросы делятся на макроопределения (macro definition или просто macro) и макровызовы (macro instruction или macrocode).

Признаком макроопределения является директива MACRO, записанная в его первой строке. Перед ней обязательно указывается имя макроса, а после нее - список параметров, если они используются. В последующих строках располагается тело макроопределения, структура которого зависит от назначения макроса. В общем случае тело содержит директивы MASM и ассемблерные команды или их заготовки. В заготовках команд вместо имен операндов или операций указываются имена параметров макроопределения. Тело макроса обязательно заканчивается директивой ENDM.

Признаком макровызова является имя макроопределения и список параметров, если они используются. В процессе компиляции, обнаружив любое имя, MASM ищет его в таблицах имен, описанных в компилируемой программе и в своих таблицах, зарезервированных имен. Если имя соответствует макроопределению, то выполняется макроподстановка (macro substitution). При этом фрагменты ассемблерного кода, содержащиеся в теле макроопределения, включаются в текст программы, сразу компилируются и при отсутствии ошибок передаются в формируемый объектный модуль. В таких случаях макроподстановку иногда называют макрорасширением (macro expansion). Если же тело макроопределения не содержит ассемблерного кода, то в тексте программы и в объектном коде не остается никаких следов от его выполнения (пример 6 данной статьи).

По аналогии с подпрограммами можно считать, что в макроопределении описаны формальные параметры, а в макровызове - фактические. Как обычно, должно соблюдаться строгое соответствие количества, типов и последовательности перечисления формальных и фактических параметров. Параметры, отсутствующие в макровызове, заменяются двумя подряд расположенными запятыми. Фактический параметр может отсутствовать, только если это предусмотрено при разработке макроопределения, т. е. перед использованием параметра проверяется, определен он или нет (указан или отсутствует).

По функциональному назначению макроопределения можно разделить на две основные категории:

средства для упрощения программирования на ассемблере;

средства для расширения возможностей компилятора.

Принципиальное различие между ними заключается в следующем. Вызов макросов первой категории можно заменить группой команд, получаемых в результате подстановки фактических параметров. Макросы второй группы ничем заменить нельзя, поскольку они формируют объектные коды таких команд, который не обрабатывает конкретная версия компилятора.

Макроопределения первой категории чаше всего применяются для упрощения программирования запросов функций, выполняемых операционными системами и базовой системой ввода-вывода (BIOS). Разработчики MASM включают в состав дистрибутивных пакетов файлы, содержащие такие макроопределения, их можно (и нужно) использовать в прикладных программах различного назначения.

В дистрибутивном пакете MASM 6.0 имеется каталог INCLUDE, в котором собраны файлы с макроопределениями различного назначения. Например, файлы bios.inc и dos.inc содержат макроопределения запросов функций BIOS и DOS, соответственно, а файл macros.inc содержит макросы, выполняющие различные вспомогательные действия, в том числе и обработку строк текста. В том же каталоге находятся файлы с другими макроопределениями.

В дистрибутивном пакете MASM32 имеется каталог MACROS, в котором находятся файлы usemacro.inc и macros.asm, содержащие макроопределениями запросов функций, выполняемых 32-x разрядными операционными системами семейства WINDOWS. Если запросы функций BIOS и DOS программируются сравнительно просто, то программировать запросы системных функций WIDOWS, без использования готовых макроопределений просто нецелесообразно.

Дистрибутивные пакеты содержат не только примеры макроопределений, но и описания зарезервированных имен, и исходные тексты программ выполняющих много полезных действий. Их анализ позволит вам найти ответ на многие вопросы, связанные с использованием системных функций, библиотек различного назначения и пр.

Макросредства, предназначенные для расширения возможностей компилятора, стали использоваться сравнительно недавно. Необходимость в них возникла потому, что система команд микропроцессоров семейства Pentium развивалась слишком быстро. Появлялись десятки и сотни принципиально новых команд, которые не могли компилировать существующие версии MASM. В этой ситуации сами разработчики стали публиковать макроопределения, позволяющие использовать новые команды при работе с устаревшими версиями MASM.

В заключение данного раздела следует отметить, что разработка макроопределений имеет мало общего с программированием на языке ассемблера. В этом случае программист решает совершенно другие задачи и использует другие средства. В двух следующих разделах данной статьи описаны некоторые приемы, применяемые при разработке макросов различного назначения.

**19.19.1 Визначити проблеми модульного програмування на всіх етапах життєвого циклу програми**

Мы неоднократно подчеркивали один из существенных недостатков программ на языке ассемблера, а значит, и самого языка, — недостаточную наглядность. По про шествии даже небольшого времени программисту порой бывает трудно разобраться в деталях им же написанной программы. А о чужой программе и говорить не приходится. Если в ней нет хотя бы минимальных комментариев, то разобраться с тем, что она делает, довольно трудно. Причины этого тоже понятны — при написании программы на языке ассемблера человек должен запрограммировать самые элементарные действия или операции. При этом он должен учитывать и контролировать состояние большого количества данных. Из-за элементарности программи руемых операций реализация одного и того же алгоритма может быть произведена по крайней мере несколькими способами. Эта неоднозначность влечет за собой Непредсказуемость, что и затрудняет процесс обратного восстановления исходного алгоритма по ассемблерному коду.

По мере накопления опыта эти проблемы частично снимаются. Но одного опыта мало. Ситуация усугубляется, если работает коллектив разработчиков. Тут уже нужны специальные средства.

Концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений. Основа концепции модульного программирования — *модуль,* который является продуктом процесса разбиения большой задачи на ряд более мелких функционально самостоятельных подзадач. Этот процесс называется *функциональной декомпозицией задачи.* Каждый модуль в функциональной декомпозиции представляет собой ≪черный ящик≫ с одним входом и одним выходом. Модули связаны между собой только входными и выходными данными. Модульный подход позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединять их в единый загрузочный модуль.

Так как отдельный модуль в соответствии с концепцией модульного программирования — это функционально автономный объект, то он ничего не должен знать о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не должно быть известно о внутреннем устройстве данного модуля. Однако должны быть какие-то средства, с помощью которых можно связать модули. Телевизор и видеомагнитофон

могут быть разными, но связь между ними одинакова. Та же идея лежит и в организации связи модулей. Внутреннее устройство модулей может совершенствоваться, они вообще могут в следующих версиях писаться на другом языке, но в процессе их объединения в единый исполняемый модуль этих особенностей не должно быть заметно. Таким образом, каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект (процедура, переменная) видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые их поля не заполненными. Позднее, на этапе компоновки, программа TLINK (TASM) или программа компоновки языка высокого уровня произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Если входные данные для модуля (аргументы) — переменные, то один и тот же модуль можно использовать многократно для разных наборов значений этих переменных. Но как организовать передачу значений переменных в модуль (процедуру)? При программировании на языке высокого уровня программист ограничен в выборе способов передачи аргументов теми рамками, которые для него оставляет компилятор. В языке ассемблера практически нет никаких ограничений на этот счет, и, фактически, решение проблемы передачи аргументов предоставлено программисту.

На практике используются следующие варианты передачи аргументов в модуль (процедуру):

через регистры; через общую область памяти; через стек;

с помощью директив EXTRN и PUBLIC.

* каком виде можно передавать аргументы в процедуру? Ранее упоминалось, что передаваться могут либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по значению и по адресу, соответственно.
* отличие от языков высокого уровня, в языке ассемблера нет отдельных понятий для

процедуры и функции. Организация возврата результата из процедуры полностью ложится на программиста. Если исходить из того, что получение результата — частный случай передачи аргументов, то программисту доступны три варианта возврата значений из процедуры.

* использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Функции DOS используют именно этот способ. Из рассматриваемых здесь трех вариантов данный способ является наиболее быстрым, поэтому его есть смысл задействовать для организации критичных по времени вызова процедур с малым количеством аргументов.
* использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования, чтобы устранить неоднозначность при трактовке содержимого общих участков памяти.
* использованием стека. Здесь, подобно передаче аргументов через стек,

также требуется регистр ВР.

Если программа не предназначена для решения каких-то системных задач, требующих максимально эффективного использования ресурсов компьютера, если к ней не предъявляются сверхжесткие требования по размеру и времени работы, если вы не ≪фанат≫ ассемблера — то, на мой взгляд, следует подумать о выборе одного из языков высокого уровня. Существует и третий, компромиссный путь — комбинирование программ на языке высокого уровня с кодом на ассемблере. Такой способ обычно используют в том случае, если в вашей программе есть фрагменты, которые либо вообще невозможно реализовать без ассемблера, либо ассемблер может значительно повысить эффективность работы программы.

**20.20.1** Охарактеризувати використаня сегментних регістрів по замовчуванню.

Сегментні регістри: CS, DS, SS та ES

Кожен сегментний регістр забезпечує адресацію памяті обсягом 64Кб, яка називається поточним сегментом.

* 1. Регістр CS. Регістр сегмента кода містить початкову адресу сегмента кода. Ця адреса плюс значення зміщення в командному вказівнику(IP)визначає адресу команди, яка має бути обрана до виконання. Для звичайних програм не має сенсу робити посилання на регістр CS.
  2. Регістр DS. Регістр сегмента даних містить початкову адресу сегменту даних. Ця адреса плюс зміщення, визначене в команді, вказує на конкретну комірку в сегменті даних.
  3. Регістр SS. Регістр сегмента стека містить початкову адресу сегмента стека.
  4. Регістр ES. Деякі операції зі строками використовують додатковий сегментний регістр для керування адресацією пам’яті. В даному контексті регістр ES пов'язаний з індексним регістром DI. Якщо необхідно використовувати регістр ES, програма має його ініціалізувати.

**21.21.1 Дати детальну характеристику процесу переривання від клавіатури з моменту натискання клавіші до переходу на процедуру обробки переривань.**

Алгоритм визначення байта не явно заданого префікса заміни сегмента полягає у наступному:

1. Визначається сегментний регістр за замовчуванням (за переліком регістрів адрес). Нагадаємо, що, як правило, за замовчуванням використовується регістр *DS*. Виключення наступні: у випадку використання *BP* як адресного регістра або регістрів *EBP* і *ESP* (без множника) як базових, за замовчуванням використовується регістр *SS*.
2. Якщо операнди машинної інструкції не є адресними виразами, або в адресних виразах відсутній адресний терм, або це команда передачі управління з типом операнда *Far* (*Near*), байт заміни сегмента відсутній.
3. За таблицею ідентифікаторів користувача визначається ім'я логічного сегмента, в якому розміщений адресний терм.
   1. У таблиці назначень сегментним регістрам визначається сегментний регістр, якому назначено логічний сегмент, де саме розміщується ідентифікатор.
4. Якщо логічному сегменту назначено сегментний регістр, який не співпадає з сегментним регістром за замовчуванням, тоді повинен генеруватись байт префікса заміни сегмента назначеного сегментного регістра.

При явному завданні префікса заміни сегмента визначається сегментний регістр за замовчуванням (згідно з вищезазначеним), і, при його співпадінні з явно заданим сегментним регістром, префікс не **генерується.**

IRQ1 (INT 9) — прерывание клавиатуры. Это прерывание вызывается при каждом нажатии и отпускании клавиши на клавиатуре. Стандартный обработчик этого прерывания выполняет довольно много функций, начиная с перезагрузки по Ctrl-Alt-Del и заканчивая помещением кода клавиши в буфер клавиатуры BIOS.

**22.22.1 Охарактеризувати лексичний аналізатор компілятора Асемблера з врахуванням обробки макросів.**

В результаті лексичного аналізу формується таблиця лексем чергового рядка програми. Лексема – один або більше символів в рядку, які Асемблером розглядаються як єдиний об’єкт. Лексеми – “неподільні атоми” мови програмування. В зв’язку з цим часто використовується термін «термінальний символ» мови програмування. В мові Асемблера розрізняють наступні лексеми:

* Ідентифікатори – послідовності букв та цифр, які розпочинаються з букви. До букв відносяться малі та великі букви відповідних алфавітів. В MASM та TASM таким алфавітом є латинський алфавіт, а також символи \_, @, ?, $. Загальноприйнято не розрізняти малі та великі букви. В той же час малі та великі букви в ASCII мають різні коди. Тому при обробці ідентифікаторів компілятор перетворює всі малі букви в великі Довжина ідентифікатора практично не обмежується, але значущими є перші 32 символи.
* Числові константи - послідовності цифр та деяких букв, які розпочинаються з цифри. Розрізняють двійкові, восьмеричні, десяткові та шістнадцятирічні константи. В якості допустимих букв числової константи є великі та малі (вони як і у ідентифікаторах не розрізняються) букви a,b,c,d,e,f та букви основи системи числення – b,o,q,d,h.. Двійкові константи складаються із цифр 0 та1 і обов’язково закінчуються буквою b. Восьмеричні константи складаються із цифр від 0 до 7 і обов’язково закінчуються буквою o або q. Десяткові константи складаються із цифр від 0 до 9 та можуть закінчуватись буквою d. Особливістю десяткових констант є те, що в них допускається відсутність букви d. Ознакою закінчення десяткової константи в цьому випадку є поява символу, який не входить в перелік цифр від 0 до 9. Шістнадцятирічні константи складаються із цифр від 0 до 9 і букв a,b,c,d,e,f та обов’язково закінчуються буквою h.
* Текстові константи - послідовність довільних символів, які можна ввести з клавіатури і які розпочинаються та закінчуються символом ‘ або “. Символи в текстових константах ніяким перетворенням не підлягають.
* Розподільники лексем - символи пропуску, табуляції та точка з комою
* Одно символьні лексеми – всі інші символи, які містяться в алфавіті мови Асемблера.

Розподільник між двома лексемами є обов’язковим, якщо конкатенація (при стикування) цих лексем є також лексемою.

Таблиця лексем може мати, наприклад, наступну структуру

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Лексема | Довжина лексеми в символах | Тип лексеми |
|  |  |  |  |

В результаті формування таблиці лексем з подальшого розгляду видаляються всі розподільники та коментарі.

**23.23.1 Охарактеризувати алгоритм визначення кількості байт машинної інструкції на першому перегляді компілятора Асемблера.**

На обох переглядах обробка адресних виразів використовується для визначення кількості байт, які необхідно згенерувати по поточній машинній інструкції.

Нагадаємо структуру коду команди процесорів 80х86

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1,2 або 3 однобайтних  префіксів | 1 або 2 байта коду операції | 0,1 або 2 байта режиму адресації  (ModR/m та  Sib) | 0,1,2 або 4 байта  зміщення в  команді | 0,1,2 або 4 байта безпосередніх даних |

**24.24.1 Охарактеризувани алгоритм визначення байтів modr/mта Sib на другому перегляді компілятора Асемблера.**

Структура байта Modr/mпри 16-розрядній адресації.

|  |  |  |
| --- | --- | --- |
| Поле mod - 2 біта | Поле reg - 3 біта | Поле r/m - 3 біта |

Поле mod використовується для визначення зміщення в команді, а поле r/m - для визначення регістрів адрес, вміст яких використовується для формування ефективної адреси. Поле reg призначене для задання регісра даних, який використовується в команді або є частиною коду операції.

Розглянемо поле mod:

* При mod=00 зміщення в команді відсутнє, а ефективна адреса формується по вмісту регістру (регістрів) адрес, які задаються полем r/m.
* При mod=01 зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до двох байт з послідуючим додаванням до вмісту регістру (регістрів) адрес, які задаються полем r/m.
* При mod=10 зміщення в команді двохбайтне і додається при формуванні ефективної адреси до вмісту регістру (регістрів) адрес, які задаються полем r/m.
* При mod=11 адреса пам'яті не задається, а поле r/m задає код регістра даних.

В полі r/m регістри адрес або їх можлива комбінація задається слідуючим чином:

Таблиця 5.1

|  |  |  |
| --- | --- | --- |
| Код в полі r/m | Регістри адрес | Сегментний регістр, який використовується по замовчуванню |
| 000 | BX+SI | DS |
| 001 | BX+DI | DS |
| 010 | BP+SI | SS |
| 011 | BP+DI | SS |
| 100 | SI | DS |
| 101 | DI | DS |
| 110 | BP | SS |
| 111 | BX | DS |

З поданої таблиці випливають наступні висновки, які справедливі і для сучасних мікропроцесорів сімейства при 16-розрядній адресації:

1. При 16-розрядній адресації тільки чотири регістра -BX,BP,SI та DI можуть використовуватись як регістри адрес.
2. Для формування багатокомпонентної адреси використовуються тільки обмежений набір пар регістрів - (BX,SI), (BX,DI), (BP,SI) (BP,DI).
3. Із реалізації випадає один із широко вживаних режимів - режим прямої адресаціїї, тобто режим коли зміщення в команді і є зміщенням в сегменті.

Відносно п.3 інженери фірми Intel вимушені були прийняте наступне вирішення - режим прямої адресації ввести при mod=00 та r/m=110, тобто, не дивлячись на те, що mod=00 зміщення в команді задавати двохбайтним, якщо r/m=110. При цьому регістр BP не використовується. Це дуже нагадує "латку" в програмах. Але ця "латка" досить продумана. З апаратної реалізації випадає режим посередньої регістрової адресації з використанням регістра BP як регістра адреси, але використання цього режиму, в стратегічному призначенні регістра BP як базового регістра структур даних стека, малоймовірне. В крайньому випадку можна використати режим при mod=01 та нульовим байтом зміщення в команді, що і реалізовано трансляторами програм на мові Асемблера.

Інтерпретація полів байта mod-r/m в режимі 32-розрядної адресації відрізняється від режиму 16-розрядної адресації. Головна відмінність – в інтерпретації процесором поля r/m (див. Табл 5.2)

Таблиця 5.2

|  |  |  |
| --- | --- | --- |
| Код в полі r/m | Регістри адрес | Сегментний регістр, який використовується по замовчуванню |
| 000 | EAX | DS |
| 001 | ECX | DS |
| 010 | EDX | DS |
| 011 | EBX | DS |
| 100 | Наявність байта SIB | -- |
| 101 | EBP\* | SS |
| 110 | ESI | DS |
| 111 | EDI | DS |

Байт SIB має наступну структуру

|  |  |  |
| --- | --- | --- |
| Поле множника – 2 біта | Поле індексного регістра  - 3 біта | Поле базового регістра  - 3 біта |

В полі індексного регістра може вказуватись будь який регістр за виключенням ESP. При формуванні ефективної адреси вміст регістру зсувається вліво на кількість розрядів, які вказані в полі множника. Тим самим фактично при формуванні ефективної адреси відбувається множення вмісту індексного регістру на 2, 4 або 8, що позбавляє програміста додаткових дій при адресації елементів масивів слів, подвійних слів та квадрослів.

В полі базового регістра може використовуватись любий РЗП, включаючи і ESP.

Таким чином можливість формування ефективної адреси в 32-розрядному режимі значно ширша, порівнюючи з 16-розрядним режимом адрес. Тому 32-розрядний режим за допомогою префіксу зміни розрядності адрес може використовуватись і в реальному режимі при наступному уточненні – **старші 16 розрядів ефективної адреси в реальному режимі ігноруються.**

Окрім іншої інтерпретації поля r/m по іншому інтерпретується поле mod

* При mod=01 зміщення в команді однобайтне, яке при формуванні ефективної адреси знаково розширюється до чотирьох байт з послідуючим додаванням до вмісту регістру (регістрів) адрес, які задаються полем r/m та sib.
* При mod=10 зміщення в команді чотирьохбайтне і додається при формуванні ефективної адреси до вмісту регістру (регістрів) адрес, які задаються полем r/m та sib.

Приклади адресації в 32-розрядному режимі.

Mov ax,Value[ecx\*4+edx]

Mov edx,[edx\*8]

Mov al,[eax+esp]

Як і у випадку 16-розрядних адрес, режим використання регістру EBP при mod=0 відсутній. При коді 101 в полі r/m або в полі базового регістру байта SIB встановлюється режим використання 4-х байтного зміщення в команд.

**25.25.1 Охарактеризувати використання директиви Assumeкомпілятором Асемблера.**

На початку кожного перегляду в кожну строчку таблиці назначень сегментним регістрам заноситься ключове слово Nothing. Обробка операндів директиви Assume полягає в перезапису в цю таблицю ідентифікатора, який міститься за символом :. Розглянемо приклад. На початку кожного з переглядів формується наступна таблиця

|  |  |
| --- | --- |
| Сегментний регістр | Назначення |
| CS | Nothing |
| DS | Nothing |
| SS | Nothing |
| ES | Nothing |
| GS | Nothing |
| FS | Nothing |

Після обробки директиви

Assume CS:code, DS:date

таблиця матиме вигляд

|  |  |
| --- | --- |
| Сегментний регістр | Назначення |
| CS | CODE |
| DS | DATE |
| SS | Nothing |
| ES | Nothing |
| GS | Nothing |
| FS | Nothing |

Якщо далі в програмі появиться директива

Assume CS:code, DS:date2, gs:segs

то таблиця матиме вигляд

|  |  |
| --- | --- |
| Сегментний регістр | Назначення |
| CS | CODE |
| DS | DATE2 |
| SS | Nothing |
| ES | Nothing |
| GS | SEGS |
| FS | Nothing |

**26.26.1 Дати характеристику типових помилок часу виконання в програмах на мові Асемблер.**

В оперативній пам'яті більшості комп'ютерних систем в кожний конкретний момент часу може розміщуватись значна кількість різних програм та даних, які розділяють на системні та прикладні. Почергове виконання цих програм в відповідності до потреб експлуатації в загальному випадку може призвести до непередбаченого і несанкціонованого взаємного впливу між ними, який виникає в наслідок помилок в програмах. Як небезпідставно шуткують теоретики програмування, безпомилкова програма - це та програма, в якій помилки ще не виявлені. На рівні машинних команд і відповідно на рівні мови Асемблера несанкціонований вплив між програмами або частинами однієї і тієї програми можна розділить на наступні випадки :

1. читання даних по помилковій адресі (наприклад, читання байт програми або даних із другої програми), що може привести до неправильної роботи програми, яка читає .
2. записування даних по помилковій адресі – (наприклад ―пошкодження‖ команд або

даних другої програми), що призводить до помилок в роботі другої програми, коли вона буде запущена

c) передача управління по помилковій адресі (як правило, не тільки не на початок якоїсь другої програми, а і не на перший байт команди іншої програми), що призводить до не передбачуваної реакції всієї комп‘ютерної системи.

Можна вважати, що непередбачений і несанкціонований взаємний вплив між програмами пов‘язаний з помилками в адресації. Інші помилки в програмах приведуть до неправильних результатів її роботи, що може негативно вплинути на інші програми, але такий вплив можна передбачити і нівелювати програмним шляхом.

На рівні мови Асемблера основним джерелом помилок в адресації є помилки в посередній адресації та багатокомпонентній адресації, коли вся адреса або окремі її компоненти розміщуються не в команді, а в регістрах або пам'яті.

Прикладами таких команд, при зверненні до пам'яті, є

**Mov** **[si],dx**

**Add** **[bx+di],cx**

**Shl** **Mas1[bx],6**

**Xor** **Mas1[bx+si],ax**

**Push** **ax**

**Pop** **dx**

Прикладами таких команд при внутрішньо сегментній передачі управлінн я є

**Jmp** **word ptr Mas2[si]**

**Jmp** **word ptr [di]**

**Jmp** **ax**

**ret**

Вміст регістрів, які використовуються для адресації, часто буває результатом досить складних обчислень, що є потенційним джерелом помилок. Практика програмування показує, що особливо часто виникають помилки при використанні команди **ret** через те,

що при **виконанні** процедури виник дисбаланс читання та записування в стек . В результаті замість повернення із процедури відбувається передача управління по випадковій адресі.

При сегментній організації пам'яті додатковим джерелом помилок адресації є помилки в завантаженні та використанні сегментних регістрів.

**27.27.1 Дати порівняльну характеристику Гарвардської архітектури ЕОМ та архітеркури фон Неймана.**

В основе архитектуры фон Неймана лежат следующие принципы:

1. **Принцип программного управления**.

Программа состоит из набора команд, которые выполняются процессором друг за другом в определенной последовательности.

2. **Принцип однородности памяти**.

Как программы (команды), так и данные хранятся в одной и той же памяти (и кодируются в одной и той же системе счисления - чаще всего двоичной). Над командами можно выполнять такие же действия, как и над данными.

3. **Принцип адресуемости памяти**.

Структурно основная память состоит из пронумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка.

Особенностью гарвардской архитектуры является то, что память данных и программ разделены и имеют отдельные шину данных и шину команд, что позволяет повысить быстродействие МП.

Сравнительный анализ:

Гарвардская архитектура начала интенсивно использоваться только в конце 1970-х годов, когда началось  интенсивное применение цифровых сигнальных процессоров. Причиной появления интереса к гарвардской архитектуре было то, что в цифровых сигнальных процессорах необходимый объем памяти данных МП, используемой для хранения промежуточных результатов, как правило, на порядок меньше требуемого объема памяти программ.

Достоинства гарвардской архитектуры следующие :

1.      Применение небольшой по объему памяти данных способствует ускорению поиска информации в памяти и увеличивает быстродействие МП.

2.      Гарвардская архитектура позволяет организовать параллельное выполнение программ – выборка следующей команды может происходить одновременно с выполнением предыдущей в результате чего сокращается время выборки команды.

Недостатком гарвардской архитектуры является усложнение архитектуры МП и необходимость генерации дополнительных управляющих сигналов для памяти команд и памяти данных. В системах коммутации и, в более широком смысле – в системах связи, применяются как процессоры с архитектурой фон Неймана, так и процессоры с гарвардской архитектурой.

Основное преимущество архитектуры Фон-Неймана – упрощение устройства МПС, так как реализуется обращение только к одной общей памяти. Кроме того, использование единой области памяти позволяло оперативно перераспределять ресурсы между областями программ и данных, что существенно повышало гибкость МПС с точки зрения разработчика программного обеспечения. Размещение стека в общей памяти облегчало доступ к его содержимому. Неслучайно поэтому фон-неймановская архитектура стала основной архитектурой универсальных компьютеров, включая персональные компьютеры.

**Основные характеристики микропроцессоров:**

* тактовая частота – Характеризует быстродействие компьютера. Режим работы процессора задается микросхемой, называемой генератором тактовых импульсов. На выполнение процессором каждой операции отводится определенное количество тактов. Тактовая частота указывает, сколько элементарных операций выполняет микропроцессор за одну секунду. Тактовая частота измеряется в МГц;
* разрядность – это максимальное количество разрядов двоичного числа, над которым одновременно может выполняться машинная операция. Чем больше разрядность процессора, тем больше информации он может обрабатывать в единицу времени и тем больше, при прочих равных условиях, производительность компьютера. m/n/k : m – число разрядов внутренних регистров, n – разрядность шины данных, k – разрядность шины адреса. Число адресуемых ячеек памяти N=2k, объём памяти C=n\*N=n\*2k (бит).
* тип архитектуры: фон Неймана или гарвардская;
* технология производства: КМОП, nМОП, ТТЛШ, ЭСЛ.
* система команд
* виды адресации
* число источников питания

**28.28.1 Охарактеризувати префікси в процесорах 8086.**

Префікси повтореннѐ в мові Асемблера задаятьсѐ тільки ѐвно ѐк окрема машинна інструкціѐ або безпосередньо в полі мнемокоду ланцягової команди. Мнемоніки префіксів повтореннѐ - REP, REPZ (або REPE) REPNZ (або REPNE). Префікс REP задаютьсѐ перед ланцяговими командами, ѐкі не виконуять порівнѐннѐ. Алгоритм префікса REP наступний

1. Якщо CX=0 (або ECX=0), то вийти з циклу
2. Виконати наступну ланцягову команду
3. Виконаять декремент регістра CX (ECX) і перейти до п.а)

Префікси REPZ (або REPE) REPNZ (або REPNE) задаятьсѐ перед ланцяговими командами, ѐкі виконуять порівнѐннѐ. Алгоритм їхньої роботи наступний

1. Якщо CX=0 (або ECX=0), то вийти з циклу
2. Виконати наступну ланцягову команду
3. Виконаять декремент регістра CX (ECX)
4. Якщо zf=1 длѐ REPZ (REPE) або zf=0 REPNZ (REPNE), то перейти до п.а)
5. *Приклад застосування:*

Команду movs (movsb, movsw, movsd) застосовуять длѐ пересиланнѐ рядків (послідовностей, ланцюжків, масивів) у пам'ѐті. Длѐ цього необхідно використовувати префікс rep. Префікс rep змушую циклічно виконувати команди пересиланнѐ доти, поки вміст регістра ecx/cx не стане рівним нуля. При накладанні рѐдків приймача та джерела необхідно визначати порѐдок пересиланнѐ – з початку чи з кінцѐ рѐдка.

|  |  |  |  |
| --- | --- | --- | --- |
| **str1** |  | **db** | **'str1 копіюється в str2'** |
| **len\_str1=$-str1** | |  |  |
| **a\_str1** | **dd** | **str1** |  |
| **str2** |  | **db** | **len\_str1 dup (' ')** |
| **a\_str2** | **dd** | **str2** |  |
| **mov** |  | **cx,len\_str1** | |
| **lds** |  | **sі,str1** |  |
| **les** |  | **dі,str2** |  |
| **cld** |  |  |  |
| **rep** |  | **movsb** |  |

**29.29.1 Охарактеризувати директиви трансляції по умові**

Данные директивы предназначены длѐ организации выборочной транслѐции фрагментов программного кода. Такаѐ выборочнаѐ компилѐциѐ означает, что в макрорасширение вклячаятсѐ не все строки макроопределениѐ, а только те, которые удовлетворѐят определенным условиѐм. То, какие конкретно условиѐ должны быть проверены, определѐетсѐ типом условной директивы.

Введение в ѐзык ассемблера этих директив значительно повышает его мощь.

Всего имеетсѐ 10 типов условных директив компилѐции. Их логично попарно объединить в четыре группы:

1. Директивы ***IF*** и ***IFE*** — условнаѐ транслѐциѐ *по результату вычисления логического* *выражения*.
2. Директивы ***IFDEF*** и ***IFNDEF*** — условнаѐ транслѐциѐ *по факту определения символического* *имени*.
3. Директивы ***IFB*** и ***IFNB*** — условнаѐ транслѐциѐ *по факту определения фактического* *аргумента при вызове макрокоманды*.
4. Директивы ***IFIDN***, ***IFIDNI***, ***IFDIF*** и ***IFDIFI*** — условнаѐ транслѐциѐ *по результату сравнения* *строк символов*.

Условные директивы компилѐции имеят общий синтаксис и применѐятсѐ в составе следуящей синтаксической конструкции:

IFxxx

логическое\_выражение\_или\_аргу

менты

фрагмент\_программы\_1

ELSE

фрагмент\_программы\_2

ENDIF

Заклячение некоторых фрагментов текста программы — ***фрагмент\_программы\_1*** и ***фрагмент\_программы\_2*** —между директивами**IFxxx**,**ELSE**и**ENDIF**приводит к их выборочномувклячения в объектный модуль. Какой именно из этих фрагментов — ***фрагмент\_программы\_1*** или ***фрагмент\_программы\_2*** — будет вклячен в объектный модуль, зависит от конкретного типа условной директивы, задаваемого значением **xxx**, и значениѐ условиѐ, определѐемого операндом

(операндами) условной директивы ***логическое\_выражение\_или\_аргумент(ы)***.

Синтаксические конструкции, соответствуящие директивам условной компилѐции, могут быть вложенными друг в друга

**30.30.1 Охарактеризувати стандартні макроси мови Асемблера процесора 8086**

В тех местах программы, где макрогенератор должен подставить макрос, выписываетсѐ обращение к макросу в виде макрокоманды, котораѐ записываетсѐ следуящим образом:

<имѐ макроса><фактические параметры через запѐтуя или пробел>

Конкретные примеры:

SUM A,ES:B

вариант: SUM A ES:B

VAR Z,W,?

вариант: VAR Z W,?

Макрокоманды очень похожи на обычные команды и директивы. Но вместо названиѐ команды или директивы, ѐвлѐящегосѐ служебным словом, в макрокоманде указываетсѐ имѐ макроса, которое придумал автор программы. В макрокоманде параметры могут отделѐтьсѐ как запѐтыми, так и пробелами.

В качестве фактического параметра может быть указан лябой текст (в том числе и пустой), но он должен быть сбалансирован по кавычкам и угловым скобкам и в нем не должно быть запѐтых, пробелов и точек с запѐтой вне этих кавычек и скобок. Поскольку запѐтыми и пробелами отделѐетсѐ один параметр от другого, а с точек с запѐтой начинаетсѐ комментарий, то их использование внутри фактического параметра приведет к неоднозначности.

При записи параметров макрокоманд можно использовать те же макрооператоры <>, ! и %, что и при записи фактических параметров блоков повторениѐ. Если в фактическом параметре надо указать запѐтуя, пробел или точку с запѐтой, то параметр следует заклячить в уголки:

Код ASM

1. SUM <WORD PTR [SI]>,A
2. VAR C W <1,2>

Уголки не считаятсѐ относѐщимисѐ к параметру, а лишь указываят его границы. Число фактических параметров, указываемых в макрокоманде, должно равнѐтьсѐ числу формальных параметров макроса, причем *i*-ый фактический параметр соответствует *i*-му формальному параметру. Но, если фактических параметров указано больше, чем надо, то лишние фактические параметры игнорируятсѐ, а если меньше, то считаетсѐ, что в качестве недостаящих фактических параметров заданы пустые тексты.

**31.1.2 Охарактеризувати поняття «перехоплення» переривання та методи його виконання**

Операційні системи мають в своєму складі процедури обробки переривань від усіх стандартних зовнішніх пристроїв ПЕОМ. При роботі програм операційної системи встановлюються відповідні вектори в таблиці векторів переривань. В реальному режимі процесорів 80х86 та Pentium інші програми мають можливість безпосереднього доступу як до зовнішніх пристроїв так і до таблиці векторів переривань. Нехай, наприклад, користувач бажає самостійно вводить дані з клавіатури ПЕОМ (N=9). Він повинен створити власну процедуру обробки переривань від клавіатури (нехай вона має назву Userint9), зберегти серед своїх даних системний вектор переривань від клавіатури та записати свій. По закінченню роботи своєї програми необхідно відновити системний вектор. Вказані дії можуть бути виконані, наприклад, наступним чином

**Xor ax,ax**

**Mov gs,ax**

**Mov si,9\*4**

**Mov eax,gs:[si]**

**Mov syst\_vec\_9,eax** ; збереження системного вектора

**Mov x,offset userint9**

**Mov gs:[si],ax** ;встановити зміщення процедури

;користувача в векторі переривань

**Mov ax,seg userint9**

**Mov gs:[si+2],ax** ;встановити сегментну складову логічної

;адреси процедури користувача

;в векторі переривань

**...**

**...**

**Mov eax,syst\_vec\_9**

**Mov si,9\*4**

**Mov gs:[si],eax ;**відновлення системного вектора

Розглянутий приклад організації перехоплення переривань має той недолік, що "перехоплення" відбувається "за спиною" операційної системи. В результаті операційна система не повідомлена про проведене "перехоплення" і може його трактувати як пошкодження таблиці векторів переривань з примусовим відновленням "правильного" вектора. Щоб цього не сталось в MS-DOS реалізовані спеціальні функції для одержання вектора переривань (функція 35h) та встановлення вектора переривань (функція 25h). За допомогою цих функцій MS-DOS веде облік "перехоплених" переривань. Приклад використання таких функцій з детальними коментаріями наведений в моделі багатопрограмної системи (процедури Setint8, Setint9). По закінчені своєї роботи програма користувача повинна відновити системні вектори переривань (процедури Retin8, Retint9).

При перехопленні переривань можливі наступні варіанти організації процедури користувача та її взаємодії з системною процедурою.

*По першому* із них системна процедура обробки переривань не використовується, а процедура користувача повністю **бере на себевсі дії** по обслуговуванню зовнішнього пристрою. До таких дій належать подання на зовнішній пристрій необхідних команд управління для проведення та завершення сеансу зв'язку, що потребує вивчення алгоритму функціонування зовнішнього пристрою. Крім того на контролер переривань необхідно подать згідно з п.2.3.4 команду розблокування переривань поточного та нижчого пріоритетів. Вказані дії виконані в процедурі Userint9 при введені з клавіатури символу Esc. При цьому системна процедура обробки переривань від клавіатури не викликається, тобто від операційної системи "приховується" факт натиснення клавіші Esc. При самостійній роботі з клавіатурою необхідно сформувати імпульс, який подається на клавіатуру (через 7-й розряд порту 61h) і який інформує клавіатуру про завершення введення даних. Без такого імпульсу подальша робота клавіатури буде заблокована.

*В другому* варіанті мінімізується вплив процедури користувача на обробку переривань від зовнішнього пристрою. Тобто, по стану зовнішнього пристрою процедура користувача визначає, що черговий сеанс зв'язку з зовнішнім пристроєм її не стосується, а тому необхідно передати управління системній процедурі обробки переривання від даного зовнішнього пристрою. Управління на системну процедуру передається за допомогою команди JMP, забезпечуючи тим самим повернення в перервану програму безпосередньо із системної процедури. В процедурі Userint9 даний варіант реалізується при натисканні любої клавіші клавіатури крім клавіші Esc. Якщо натиснута люба друга клавіша, то процедура користувача тільки "спостерігає" за зовнішнім пристроєм, а всі дії покладаються на системну процедуру.

*Третій варіант* характеризується як безумовним виконанням системної процедури обробки переривань так і безумовним виконанням досить складної користувацької процедури. Прикладом може бути обробка переривань від таймера. Відключення системної процедури обробки переривань від таймера може призвести до збоїв в роботі операційної системи. Крім того, системна процедура повинна виконуватись в першу чергу, щоб унеможливити потенційний вплив користувацької процедури на періодичність ініціалізації системної процедури. Таким чином системна процедура повинна бути негайно викликана із користувацької процедури з поверненням по команді IRET із системної процедури в користувацьку. При цьому системна процедура виконає всі стандартні дії із відповідними зовнішніми пристроями, включаючи контролер переривань, а користувацькій процедурі залишиться виконання лише своїх особливих функцій.

**32.2.2 Охарактеризувати систему привілеїв захищеного режиму.**

Біти 6 та 5 байта атрибутів утворюють поле **DPL - рівень привілеїв** сегменту. По значенню поля **DPL** дескриптора всі сегменти пам’яті належать одному із чотирьох рівнів привілеїв. Найбільш привілейований рівень мають сегменти з **DPL=**0, а найменш привілейований - з **DPL=**3. Система привілеїв ґрунтується на використанні поточного рівня привілеїв (**CPL**). Вважається, що **CPL=DPL** непідлеглого сегмента кодів, селектор якого завантажений в регістр СS. Система привілеїв покладає наступні обмеження на виконання команд:

1. читати та записувати дані можна лише із сегментів, в яких **DPL≥CPL;**
2. виконувати стекові операції можна лише в сегментах, де **DPL=CPL;**
3. виконувати між сегментні команди **JMP** можна лише на сегменти, в яких **DPL=CPL,** якщо цільовий сегмент узгоджений, то **DPL≤CPL;**
4. викликати процедури між сегментними командами **CALL** можна лише із сегментів, в яких **DPL≤CPL;**

Рівень привілеїв з **CPL**=0 вважається системним. Лише на цьому рівні можуть виконуватись привілейовані команди, які служать для організації захищеного режиму. До таких команд належить, наприклад, команда завантаження регістра GDTR. Виконання привілейованих команд дозволено, також, в реальному режимі процесора.

**33.3.2 Охарактеризувати послідовність дій при переключенні в захищений режим**

**Перевірка поточного режиму процесора**

Перевірка виконується шляхом аналізу стану біта 0 регістра управління cr0.

mov eax,cr0

test al,1

jnz MOD\_error ; процесор уже в захищеному режимі

Крім того, необхідно розблокувати 20-й розряд адрес ПЕОМ

in al,92h

or al,2

out 92h,al

* 1. **Підготовка системних структур даних для роботи в захищеному режимі**
     1. Формування глобальної таблиці дескрипторів (GDT).

Визначимо структуру мови Асемблера для опису дескриптора сегменту, наприклад:

; ОПИС ДЕСКРИПТОРА СЕГМЕНТА

descr struc

limit\_1 dw 0 ; молодші 2 байта розміру сегменту (розряди 0-15)

base\_1 dw 0 ; молодші 2 байта фізичної 32-розрядної адреси сегменту

base\_2 db 0 ; 3-тій ба йт фізичної 32-розрядної адреси сегменту

attrib db 0 ; байт атрибутів сегменту

bt6 db 0 ; старші 4 розряди байту – атрибути сегменту,

; молодші 4 розряди байту - розряди 16-19 розміру сегмету

base\_3 db 0 ; 4-тий байт фізичної 32-розрядної адреси сегменту

descr ends

Визначимо необхідні значення байтів атрибутів дескрипторів сегментів, формуючи біт присутності рівним 1 (**p** = 1, сегмент присутній в пам'яті), значення поля привілеїв рівним 0 (**dpl** = 00, системний рівень), наприклад:

; БАЙТИ АТРИБУТІВ

code\_seg equ 10011010b; непідпорядкований сегмент кодів, дозволено читати

data\_seg equ 10010010b; сегмент даних, дозволено записувати

stack\_seg equ 10010110b; сегмент стека

ЗАУВАЖЕННЯ. Як сегмент стека може використовуватись сегмент даних з дозволом на записування. Різниця лише в трактовці процесором поля limit.

Визначимо глобальну таблицю дескрипторів, розміщуючи в ній дескриптори у такій рекомендованій послідовності:

1. 0 - вий дескриптор
2. дескриптор самої GDT
3. дескриптори сегментів даних (ds,es)
4. дескриптор сегмента стека (ss)
5. дескриптор сегмента основної програми (cs)

Рекомендується розміщувати GDT в окремому логічному сегменті, наприклад:

GDT segment para public 'data' use16

; Визначення таблиці глобальних дескрипторів

selector0 descr <>

deskr\_gdt descr <>

deskr\_ds descr <>

deskr\_es descr <>; video

deskr\_ss descr <>

deskr\_cs descr <>

gdt\_size equ $-selector0-1

GDT ends

Оскільки в даній лабораторній роботі прийнято системний (найвищий) рівень привілеїв і не використовується локальна таблиця дескрипторів, то зміщення дескрипторів в таблиці дескрипторів співпадають з селекторами дескрипторів, тому відпадає необхідність в спеціальному формуванні селекторів.

Сформуємо, далі, дескриптори у глобальній таблиці дескрипторів:

; загальна частина для всіх дескрипторів

mov ax,GDT

mov ds,ax

assume ds:GDT

;Формування дескриптора сегмента кодів

mov deskr\_cs.limit\_1,code\_size ; розмір сегмента кодів

; формування фізичної адреси сегмента кодів

xor eax,eax

mov ax,\_CODE

shl eax,4 ; відлуння реального режиму від TASM

; TASM транслює \_CODE для реального режиму

mov dword ptr deskr\_cs.base\_1, eax

; атрибути сегмента кодів

mov deskr\_cs.attrib,code\_seg

;--------------------------------------------------------

Зауважимо, що поле bt6 сформованого дескриптора залишається з 0 - вим значенням. Тобто розмір сегментів ми будемо задавати в байтах (біт гранулярності **G**=0), значення розмірів сегментів не буде більше за 64Кбайт (молодші 4 розряди bt6 дорівнюють 0), а розрядність адрес і даних по умовчанню буде дорівнювати 16 (біт розрядності **D**=0 або **B**=0), що відповідає параметру Use16 в директивах Segment.

Необхідно звернуть увагу на те, що в наведеному прикладі запис в дескриптор фізичної адреси сегменту спрощений, оскільки старший байт фізичної адреси сегменту записується не в поле base\_3 дескриптора, а в поле attrib. Тому команду запису в поле attrib необхідно розміщувати **після** команди запису в поле base\_1. Що ж стосується поля base\_3, то його значення залишається нульовим, оскільки в даній роботі всі сегменти розміщуються в межах першого мегабайта пам'яті.

Приклад формування дескриптора для GDT :

; дескриптор GDT формується для доступу до GDT системними

; програмами в захищеному режимі

;--------------------------------------------------------

mov deskr\_gdt.limit\_1,gdt\_size

xor eax,eax

mov ax,GDT

shl eax,4

mov dword ptr deskr\_gdt.base\_1,eax

mov deskr\_gdt.attrib, data\_seg

;--------------------------------------------------------

Аналогічно формуються дескриптори сегментів даних.

Доцільно для захищеного режиму сформувати окремий сегмент стека. Такий сегмент необхідно визначить без атрибута STACK директиви SEGMENT, оскільки TASM та TLINK назначать цей же сегмент як сегмент стека і в режимі реальних адрес.

ST\_p segment use16

Db 1000 dup (0)

Top\_stp equ $; Top\_stp=1000

ST\_p ends

Формування дескриптора сегмента стека має відмінність щодо поля limit - воно вказує не на максимально можливе, а на мінімально можливе зміщення в сегменті. Встановлення в якості мінімального зміщення нульового зміщення деякими процесорами сприймається не адекватно. Тому, враховуючи проведення послідуючих експериментів, рекомендується встановлювати поле limit в значення, наприклад, 10:

;--------------------------------------------------------

mov deskr\_ss.limit\_1, 10

xor eax,eax

mov ax, ST

shl eax,4

mov dword ptr deskr\_ss.base\_1,eax

mov deskr\_ss.attrib, stack\_seg

;--------------------------------------------------------

Деяку відмінність має формування дескриптора сегмента відео буфера, оскільки розмір та фізична адреса цього сегменту визначаються архітектурою ПЕОМ і фіксуються на апаратному рівні.

mov deskr\_es.limit\_1,0ffffh;

mov dword ptr deskr\_es.base\_1, 0b8000h

mov deskr\_ es.attrib, data\_seg

* + 1. Формування та запис вказівника глобальної таблиці дескрипторів.

До програмно доступних регістрів процесора відноситься спеціальний регістр GDTR, який повинен містить фізичну адресу глобальної таблиці дескрипторів та її розмір. Завантаження цього регістру може виконуватись в реальному режимі за допомогою команди LGDT процесора, яка повинна адресувати 6-байтний вказівник

;ОПИС вказівника ТАБЛИЦІ ДЕСКРИПТОРІВ

point\_dt struc

lim dw 0

adr dd 0

pointdt ends

Визначимо вказівник для GDT (в логічному сегменті даних \_DATA):

pgdt pointdt <>

Формування вказівника і завантаження GDT (вважаємо, що регістр DS містить адресу сегменту \_DATA, а Асемблер за допомогою директиви assume про це проінформований):

; формування GDT

xor eax,eax

mov ax,GDT

shl eax,4

mov pgdt.adr,eax

mov ax,gdt\_size

mov pgdt.lim,ax

; завантаження GDT

lgdt pgdt

* 1. **Переключення процесора в захищений режим**

Перед переключенням у захищений режим потрібно заборонити переривання, а також зберегти логічну адресу вершини стека реального режиму, наприклад, в сегменті даних:

**Top\_real\_mode dd ?**

Переключення у захищений режим виконується установкою в 1 молодшого біта регістра управління CR0:

Cli

Mov word ptr Top\_real\_mode,sp

Mov word ptr Top\_real\_mode+2,ss

mov eax,cr0

or al,1

mov cr0,eax

Після переходу в захищений режим необхідно в сегментні регістри завантажити селектори. Завантаження селектора в регістр CS може відбутися тільки з допомогою між сегментної команди переходу. В захищеному режимі сегментна складова логічної адреси є селектором сегмента і транслятором TASM не генерується. Тому команда між сегментної передачі управління формується "вручну" по-байтно:

db 0eah ;машинний код команди jmp

dw offset protect ;зміщення мітки переходу в сегменті кодів

dw offset deskr\_cs ;селектор сегмента кодів

protect:

;завантажити селектори для інших дескрипторів

; припускаємо, що RPL=0

mov ax,offset deskr\_ss

mov ss,ax

mov sp,Top\_stp

mov ax,offset deskr\_ds

mov ds,ax

mov ax,offset deskr\_es

mov es,ax

* 1. **Виведення повідомлення на екран**

В захищеному режимі шляхом прямого запису в сторінку відео буфера вивести коротке повідомлення. Приклад виведення повідомлення mess довжиною 20 байт:

mov cx,20

mov si,offset mess

mov di,1640 ;початкова адреса виведення

mov ah,07h ;атрибут символів

outmes:

mov al,[si]

mov es:[di],ax

inc si

add di,2

loop outmes

34.4.2 Охарактеризувати процес виклику процедур, розташованих в сегментах з вищим рівнем привілеїв

Механизм организации виртуальной памяти, при котором виртуальное пространство делится на части произвольного размера — сегменты. Этот механизм позволяет, к примеру, разбить данные процесса на логические блоки.[11] Для каждого сегмента, как и для страницы, могут быть назначены права доступа к нему пользователя и его процессов. При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре. Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g — номер сегмента, а s — смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

35.5.2 Дати порівняльну характеристику організації взаємозв’язку програм на Асемблері і програм на Си та Паскалі.

Самый очевидный способ выражения вызова процедуры или функции языка высокого уровня, после того как решено, что параметры передаются в стеке и возвращаются в регистре АХ/ЕАХ, — это способ, принятый в языке PASCAL (а также в BASIC, FORTRAN, ADA, OBERON, MODULA2), — просто поместить параметры в стек в естественном порядке. В этом случае запись

some\_proc(a,b,c,d,e)

превращается в

push a

push b

pushс

push d

push e

call some\_proc

Это значит, что процедура some\_proc, во-первых, должна очистить стек по окончании работы (например, завершившись командой ret 10) и, во-вторых, параметры, переданные ей, находятся в стеке в обратном порядке:

some\_proc proc

push bp

mov bp,sp ; создать стековый кадр

a equ [bp+12] ; определения для простого

; доступа к параметрам

b equ [bp+10]

c equ [bp+8]

d equ [bp+6]

e equ [bp+4]

; текст процедуры, использующей параметры а, Ь, с, d, e

ret 10

some\_proc endp

Этот код в точности соответствует усложненной форме директивы proc, которую поддерживают все современные ассемблеры:

some\_proc proc PASCAL,а:word,b:word,с:word,d:word,e:word

; текст процедуры, использующей параметры а, Ь, с, d, e.

; Так как ВР используется в качестве указателя стекового кадра,

; его использовать нельзя!

ret ; эта команда RET будет заменена на RET 10

some\_proc endp

Главный недостаток этого подхода — сложность создания функции с изменяемым числом параметров, аналогичных функции языка С printf. Чтобы определить число параметров, переданных printf, процедура должна сначала прочитать первый параметр, но она не знает его расположения в стеке. Эту проблему решает подход, используемый в С, где параметры передаются в обратном порядке.

Этот способ передачи параметров используется в первую очередь в языках С и C++, а также в PROLOG и других. Параметры помещаются в стек в обратном порядке, и, в противоположность PASCAL-конвенции, удаление параметров из стека выполняет вызывающая процедура. Запись

some\_proc(a,b,c,d,e)

превращается в

push e

push d

pushс

push b

push a

call some\_proc

add sp,10 ; освободитьстек

Вызваннаятакимобразомпроцедураможетинициализироватьсятак: some\_proc proc push bp mov bp,sp ;создатьстековыйкадр a equ [bp+4] ; определениядляпростогодоступакпараметрам b equ [bp+6] с equ [bp+8] d equ [bp+10] e equ [bp+12] ; текстпроцедуры, использующейпараметры a, b, с, d, e pop bp ret some\_proc endp

Ассемблеры поддерживают и такой формат вызова при помощи усложненной формы директивы proc с указанием языка С:

some\_proc proc С,а:word,b:word,с:word,d:word,e:word

; текст процедуры, использующей параметры a, b, с, d, e.

; Так как BP применяется как указатель стекового кадра,

; его использовать нельзя!

ret

some\_proc endp

Мы не пользовались до сих пор этими формами записи процедур в ассемблере потому, что они скрывают от нас тот факт, что регистр ВР используется для хранения параметров и его ни в коем случае нельзя изменять, и, в случае PASCAL, что команда ret на самом деле — команда ret N.

Преимущество по сравнению с PASCAL-конвенцией заключается в том, что освобождение стека от параметров в С возлагается на вызывающую процедуру, что позволяет лучше оптимизировать код программы. Например, если мы должны вызвать несколько функций, принимающих одни и те же параметры подряд, можно не заполнять стек каждый раз заново:

push param2

push param1

call proc1

call proc2

add sp,4

эквивалентно

proc1(param1,param2);

proc2(param1,param2);

и это — одна из причин, почему компиляторы с языка С создают более компактный и быстрый код по сравнению с компиляторами с других языков.

36.6.2 Визначити особливості обміну даними з портами в порівнянні з комірками пам’яті

Программно-управляемый ввод-вывод означает обмен данными с внешними устройствами с использованием команд процессора. Передача данных происходит через регистры процессора и при этом в конечном счете может реализовываться обмен собственно с процессором, обмен внешнего устройства с памятью, обмен между внешними устройствами.

Процессоры х86 имеют раздельную адресацию памяти и портов ввода-вывода и соответственно ввод-вывод может быть отображен либо в пространство ввода-вывода, либо вы пространство оперативной памяти (memory-mapped I/O). В последнем случае адрес памяти декодируется во внешнем устройстве и для выполнения ввода-вывода могут быть использованы все команды обращения к памяти.

Каждое адресуемый элемент адресного пространства ввода-вывода именуется портом ввода, портом вывода или портом ввода-вывода. Для обращения к портам предназначены четыре основные команды процессора: In (ввод в порт), Out (вывод из порта), Ins (ввод из порта в элемент строки памяти) и Outs (вывод элемента из строки памяти). Последние две строковые команды ввода-вывода используются для быстрой пересылки блоков данных между портом и памятью в случае последовательно расположенных адресов портов в внешнем устройстве. Обмен данными с портами, при котором используются строковые команды ввода-вывода, получил название PIO (Programmed Input/Output) - программированный ввод-вывод.

Что понимается под словом обмен? Во первых - это обращение программы к выбранному порту и пересылка информации. Пересылка - чтение или запись может производится в расчете на то, что порт всегда готов выдать или принять данные или же предваряется анализом готовности устройства. В последнем случае обмен включает операцию чтения регистра состояния устройства для анализа его готовности, ожидание готовности (зацикливание предыдущего шага), собственно обмен байтом или словом данных. Модификацией этого варианта является метод опроса - способ обнаружения асинхронных событий. При его использовании каждому возможному событию ставится в соответствие флажок, которому может обратиться программа.

37.7.2 Визначити суть переривання від зовнішніх пристроїв на прикладі системи переривань процесорів 8086 в реальному режимі.

Послідовність дій, які відбуваються в ПЕОМ при обробці переривань від зовнішніх пристроїв полягає в наступному:

1. В результаті функціонування зовнішнього пристрою (наприклад, при натисканні або відтисканні клавіш клавіатури, або із системного таймера через проміжки часу приблизно в 65,5 млсек) на контролер переривання поступає сигнал, в результаті якого контролер може сформувати для процесора сигнал "Запит переривання".
2. Процесор **закінчує виконання** чергової команди програми (в деяких випадках, в залежності від чергової команди процесор безумовно виконує ще одну команду - див. п.*f*).
3. Процесор аналізує ознаку дозволу на переривання (if), яка міститься в регістрі ознак (flags). Якщо if=0 процесор ігнорує запит і продовжує виконувати програму
4. Якщо if=1 процесор подає на контролер переривання сигнал "Підтвердження запиту"
5. Одержавши сигнал "Підтвердження запиту", контролер переривань передає процесору байт даних N (N - **номер вектора переривань**), значення якого ідентифікує зовнішній пристрій, наприклад в MS-DOS N=8 для таймера і N=9 для клавіатури.
6. Процесор записує в стек вміст регістра ознак, після чого **встановлює if=0**, та записує в стек вміст регістрів CS та IP. При запису в стек використовується логічна адреса SS:SP, тобто стек програми, яка переривається. Якщо запит на переривання з'явиться під час виконання команди запису в сегментний регістр SS (наприклад, під час виконання команди MOV SS,AX), то в результаті повна логічна адреса стека може бути помилковою, оскільки залишиться "старе" значення SP. Для унеможливлення такої помилки в разі появи сигналу "Запит переривання" під час виконання любої команди запису в сегментний регістр SS, сигнал "Підтвердження запиту" формується після виконання іще однієї команди, якою, звісно, повинна бути команда завантаження регістра SP. А взагалі заміну повної логічної адреси стека рекомендується виконувати за допомогою команди LSS SP,mem32, де mem32 - адреса повної логічної адреси нового стека.
7. Процесор завантажує регістр IP значенням слова пам'яті по адресі 0000:N\*4, а регістр CS - значенням слова пам'яті по адресі 0000:N\*4+2. Таким чином в реальному режимі перші 1000 байт ОЗП призначаються для таблиці, яку називають **таблицею векторів переривань**. Таблиця містить 256 рядків по 4 байта на рядок. Кожний рядок таблиці переривань використовується для зберігання повної логічної адреси (слово для сегментної складової та слово для зміщення в сегменті) процедури обробки переривань. Самі ж процедури обробки переривань можуть розміщуватись в любій області ОЗП. Оскільки байт N ідентифікує зовнішній пристрій, то в свою чергу рядки (вектори) також можуть ідентифікувати зовнішні пристрої
8. Процедура обробки переривань повинна закінчуватись командою IRET. Ця команда відновлює із стека вміст регістра ознак, включаючи попередній стан ознаки дозволу на переривання (if), який згідно з п.*d)* повинен дорівнювати 1, а також завантажує зі стека регістри IP та CS.

38.8.2 Дати характеристику задач, які виконує компілятор Асемблера на першому перегляді.

Задачі, які вирішуються на першому перегляді:

1. Розподіл пам’яті, тобто визначення зміщень команд та даних в відповідних логічних сегментах. Для вирішення цієї задачі мова Асемблера повинна давати можливість визначення **кількості** байт, які будуть генеруватись згідно з кожним рядком програми. Визначення самих байт на першому перегляді в класичній двохпереглядній схемі не передбачається.
2. Формування таблиці ідентифікаторів, які визначаються користувачем. До таких ідентифікаторів належать:
   1. Мітки (символічні адреси команд), тобто зміщення першого байту команди в відповідному логічному сегменті)
   2. Імена в директивах визначення пам’яті (символічні адреси даних – зміщення молодшого байту даних в відповідному логічному сегменті)
   3. Імена процедур (символічні адреси процедур - зміщення першого байту процедури в відповідному логічному сегменті)
   4. Символічне позначення відповідних послідовностей символів в директиві EQU
   5. Символічне позначення констант в директиві =.
   6. Імена макрокоманд.
   7. Імена програмних об’єктів в директиві Extrn
3. Виявлення помилки – повторне визначення одного і того ідентифікатора.
4. Формування таблиці логічних сегментів.
5. Формування бібліотеки макровизначень.
6. Формування макророзширень по макрокомандах.

39.9.2 Охарактеризувати суть та типову організацію мультипрограмного режиму з одним процесором.

При організації мультипрограмних систем використовують переривання від зовнішніх пристроїв, основним серед яких можна вважати таймер. Таймер з деякою частотою подає на контролер переривань запити на переривання. Період між двома запитами будемо називати **квантом часу**. При стандартній настройці таймера ПЕОМ квант часу дорівнює приблизно 56 млсек. Сучасні ПЕОМ спроможні виконати за цей час значну кількість команд. Якщо кожній із програм (в подальшому - задач) регулярно виділяти по кванту часу то тим самим досягається ефект "одночасного" їх виконання. Управління задачами покладається на частину процедури обробки переривань від таймера, яку будемо називати **менеджером квантів**. Основною особливістю процедури обробки переривань від таймера при мультипрограмній обробці є можливість повернення по команді IRET не в поточну перервану задачу, а в іншу задачу, яка була перервана раніше. Для цього необхідне динамічне управління стеком процесора, тобто вмістом регістра SP (ESP), або в загальному випадку вмістом регістрів SS та SP(ESP). Таким чином алгоритм обробки переривань від таймера при організації мультипрограмної обробки повинен мати наступні пункти:

1. Збереження в стеку перерваної задачі (де вже містяться ознаки та адреса повернення) вмісту регістрів загального призначення і, по необхідності, сегментних регістрів.
2. Збереження в спеціально виділеній області пам'яті вмісту регістра SP або регістрів SS та SP(ESP).
3. Вибір в менеджері квантів наступної задачі
4. Відновлення раніше збереженого вмісту регістра SP(ESP) або SS та SP(ESP) для вибраної задачі.
5. Відновлення зі стека вибраної задачі вмісту регістрів.
6. Виконання команди IRET

Найпростіший алгоритм менеджера квантів - це почерговий запуск задач, при цьому перервана від таймера задача становиться в кінець черги. Можливе пріоритетне управління, коли більш пріоритетним задачам виділяється більше квантів часу ніж іншим. Можлива тимчасова зупинка задачі, коли їй не виділяються кванти часу і т.д. Необхідні дані для вибору задачі менеджер квантів може одержувати від процедур обробки переривань від інших зовнішніх пристроїв, наприклад, клавіатури.

При обробці переривань від інших зовнішніх пристроїв можуть виникнути умови, при яких продовження перерваної задачі неможливий, а квант часу ще не вичерпаний. Можливі так же випадки, коли ні одна із задач не може бути вибрана менеджером квантів. Для таких випадків передбачається спеціальна задача, яку називають фоновою. Така задача має самий низький пріоритет і виконується при умові, що в відповідний момент часу не може виконуватись ніяка інша задача. Інколи в якості фонової задачі використовують наступну послідовність команд:

**Loop\_label:**

**sti**

**jmp loop\_label**

40.10.2 Дати характеристику задач, які виконує компілятор Асемблера на другому перегляді.

1. Генерування байт команд та байт даних по рядках початкової програми.
2. Формування файлу лістінгу, включаючи до нього таблицю логічних сегментів та таблицю ідентифікаторів, визначених користувачем.
3. Формування повідомлень про помилки в файлі лістінгу, включаючи помилки повторного визначення ідентифікаторів, які виявлені на першому перегляді.
4. Формування об’єктного файлу.

41.11.2 Охарактеризувати системні структури захищеного режиму, які використовуються при переключенні задач

Переключение задач осуществляется, если:

* текущая задача выполняет дальний JMP или CALL на шлюз задачи или прямо на TSS;
* текущая задача выполняет IRET, если флаг NT равен 1;
* происходит прерывание или исключение, в качестве обработчика которого в IDT записан шлюз задачи.

При переключении процессор выполняет следующие действия:

1. Для команд CALL и JMP проверяет привилегии (CPL текущей задачи и RPL селектора новой задачи не могут быть больше, чем DPL шлюза или TSS, на который передается управление).
2. Проверяется дескриптор TSS (его бит присутствия и лимит).
3. Проверяется, что новый TSS, старый TSS и все дескрипторы сегментов находятся в страницах, отмеченных как присутствующие.
4. Сохраняется состояние задачи.
5. Загружается регистр TR. Если на следующих шагах происходит исключение, его обработчику придется доделывать переключение задач, вместо того чтобы повторять ошибочную команду.
6. Тип новой задачи в дескрипторе изменяется на занятый и устанавливается флаг TS в CR0.
7. Загружается состояние задачи из нового TSS: LDTR, CR3, EFLAGS, EIP, регистры общего назначения и сегментные регистры.

Если переключение задачи вызывается командами JUMP, CALL, прерыванием или исключением, селектор TSS предыдущей задачи записывается в поле связи новой задачи и устанавливается флаг NT. Если флаг NT установлен, команда IRET выполняет обратное переключение задач.

При любом запуске задачи ее тип изменяется в дескрипторе на занятый. Попытка вызвать такую задачу приводит к #GP, сделать задачу снова свободной можно, только завершив ее командой IRET или переключившись на другую задачу командой JMP.

Покажем, как создавать задачи и переключаться между ними на следующем примере.

; pm4.asm

; Пример программы, выполняющей переключение задач.

; Запускает две задачи, передающие управление друг другу 80 раз, задачи выводят

; на экран символы ASCII с небольшой задержкой

;

; Компиляция:

; TASM:

; tasm /m pm4.asm

; tlink /x /3 pm4.obj

; WASM:

; wasm pm4.asm

; wlink file pm4.obj form DOS

; MASM:

; ml /c pm4.asm

; link pm4.obj,,NUL,,,

.386p

RM\_seg segment para public "CODE" use16

assume cs:RM\_seg,ds:PM\_seg,ss:stack\_seg

start:

; подготовить сегментные регистры

push PM\_seg

pop ds

; проверить, не находимся ли мы уже в РМ

mov eax,cr0

test al,1

jz no\_V86

; сообщить и выйти

mov dx,offset v86\_msg

err\_exit:

push cs

pop ds

mov ah,9

int 21h

mov ah,4Ch

int 21h

; убедиться, что мы не под Windows

no\_V86:

mov ax,1600h

int 2Fh

test al,al

jz no\_windows

; сообщить и выйти

mov dx,offset win\_msg

jmp short err\_exit

; сообщения об ошибках при старте

v86\_msg db "Процессор в режиме V86 - нельзя переключиться в РМ$"

win\_msg db "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; итак, мы точно находимся в реальном режиме

no\_windows:

; очиститьэкран

mov ax,3

int 10h

; вычислить базы для всех дескрипторов сегментов данных

xor еах,еах

mov ax,RM\_seg

shl eax,4

mov word ptr GDT\_16bitCS+2,ax

shr eax,16

mov byte ptr GDT\_16bitCS+4,al

mov ax,PM\_seg

shl eax,4

mov word ptr GDT\_32bitCS+2,ax

mov word ptr GDT\_32bitSS+2,ax

shr eax,16

mov byte ptr GDT\_32bitCS+4,al

mov byte ptr GDT\_32bitSS+4,al

; вычислить линейный адрес GDT

xor eax,eax

mov ax,PM\_seg

shl eax,4

push eax

add eax,offset GDT

mov dword ptr gdtr+2,eax

; загрузить GDT

lgdt fword ptr gdtr

; вычислить линейные адреса сегментов TSS наших двух задач

pop eax

push eax

add eax,offset TSS\_0

mov word ptr GDT\_TSS0+2,ax

shr eax,16

mov byte ptr GDT\_TSS0+4,al

pop eax

add eax,offset TSS\_1

mov word ptr GDT\_TSS1+2,ax

shr eax,16

mov byte ptr GDT\_TSS1+4,al

; открыть А20

mov al,2

out 92h,al

; запретить прерывания

cli

; запретить NMI

in al,70h

or al,80h

out 70h,al

; переключиться в РМ

mov eax,cr0

or al,1

mov cr0,eax

; загрузить CS

db 66h

db 0EAh

dd offset PM\_entry

dw SEL\_32bitCS

RM\_return:

; переключиться в реальный режим RM

mov eax,cr0

and al,0FEh

mov cr0,eax

; сбросить очередь предвыборки и загрузить CS

db 0EAh

dw $+4

dw RM\_seg

; настроить сегментные регистры для реального режима

mov ax,PM\_seg

mov ds,ax

mov es,ax

mov ax,stack\_seg

mov bx,stack\_l

mov ss,ax

mov sp,bx

; разрешить NMI

in al,70h

and al,07FH

out 70h,al

; разрешить прерывания

sti

; завершить программу

mov ah,4Ch

int 21h

RM\_seg ends

PM\_seg segment para public "CODE" use32

assume cs:PM\_seg

; таблица глобальных дескрипторов

GDT label byte

db 8 dup(0)

GDT\_flatDS db 0FFh,0FFh,0,0,0,10010010b,11001111b,0

GDT\_16bitCS db 0FFh,0FFh,0,0,0,10011010b,0,0

GDT\_32bitCS db 0FFh,0FFh,0,0,0,10011010b,11001111b,0

GDT\_32bitSS db 0FFh,0FFh,0,0,0,10010010b,11001111b,0

; сегмент TSS задачи 0 (32-битный свободный TSS)

GDT\_TSS0 db 067h,0,0,0,0,10001001b,01000000b,0

; сегмент TSS задачи 1 (32-битный свободный TSS)

GDT\_TSS1 db 067h,0,0,0,0,10001001b,01000000b,0

gdt\_size = $ - GDT

gdtr dw gdt\_size-1 ; размер GDT

dd ? ; адрес GDT

; используемые селекторы

SEL\_flatDS equ 001000b

SEL\_16bitCS equ 010000b

SEL\_32bitCS equ 011000b

SEL\_32bitSS equ 100000b

SEL\_TSS0 equ 101000b

SEL\_TSS1 equ 110000b

; сегмент TSS\_0 будет инициализирован, как только мы выполним переключение

; из нашей основной задачи. Конечно, если бы мы собирались использовать

; несколько уровней привилегий, то нужно было бы инициализировать стеки

TSS\_0 db 68h dup(0)

; сегмент TSS\_1. В него будет выполняться переключение, так что надо

; инициализировать все, что может потребоваться:

TSS\_1 dd 0,0,0,0,0,0,0,0 ; связь, стеки, CR3

dd offset task\_1 ; EIP

; регистры общего назначения

dd 0,0,0,0,0,stack\_l2,0,0,0B8140h ; (ESP и EDI)

; сегментныерегистры

dd SEL\_flatDS,SEL\_32bitCS,SEL\_32bitSS,SEL\_flatDS,0,0

dd 0 ; LDTR

dd 0 ; адрес таблицы ввода-вывода

; точка входа в 32-битный защищенный режим

PM\_entry:

; подготовить регистры

xor еах,еах

mov ax,SEL\_flatDS

mov ds,ax

mov es,ax

mov ax,SEL\_32bitSS

mov ebx,stack\_l

mov ss,ax

mov esp,ebx

; загрузить TSS задачи 0 в регистр TR

mov ax,SEL\_TSS0

ltr ax

; только теперь наша программа выполнила все требования к переходу

; в защищенный режим

xor еах,еах

mov edi,0B8000h ; DS:EDI - адрес начала экрана

task\_0:

mov byte ptr ds:[edi],al ; вывести символ AL на экран

; дальний переход на TSS задачи 1

db 0EAh

dd 0

dw SEL\_TSS1

add edi,2 ; DS:EDI - адрес следующего символа

inc al ; AL - код следующего символа,

cmp al,80 ; если это 80,

jb task\_0 ; выйти из цикла

; дальний переход на процедуру выхода в реальный режим

db 0EAh

dd offset RM\_return

dw SEL\_16bitCS

; задача 1

task\_1:

mov byte ptr ds:[edi],al ; вывести символ на экран

inc al ; увеличить код символа

add edi,2 ; увеличить адрес символа

; переключиться на задачу 0

db 0EAh

dd 0

dw SEL\_TSS0

; сюда будет приходить управление, когда задача 0 начнет выполнять переход

; на задачу 1 во всех случаях, кроме первого

mov ecx,02000000h ; небольшая пауза, зависящая от скорости

loop $ ; процессора

jmp task\_1

PM\_seg ends

stack\_seg segment para stack "STACK"

stack\_start db 100h dup(?) ; стекзадачи 0

stack\_l = $ - stack\_start

stack\_task2 db 100h dup(?) ; стекзадачи 1

stack\_l2 = $ - stack\_start

stack\_seg ends

end start

Чтобы реализовать многозадачность в реальном времени в нашем примере, достаточно создать обработчик прерывания системного таймера IRQ0 в виде отдельной (третьей) задачи и поместить в IDT шлюз этой задачи. Текст обработчика для нашего примера мог быть крайне простым:

task\_3: ; это отдельная задача - не нужно сохранять регистры!

mov al,20h

out 20h,al

jmp task\_0

mov al,20h

out 20h,al

jmp task\_1

jmp task\_3

Но при вызове обработчика прерывания старая задача помечается как занятая в GDT и повторный JMP на нее приведет к ошибке. Вызов задачи обработчика прерывания, так же как и вызов задачи командой CALL, подразумевает, что она завершится командой IRET. Именно команду IRET оказывается проще всего вызвать для передачи управления из такого обработчика — достаточно только подменить селектор вызвавшей нас задачи в поле связи и выполнить IRET.

task\_3: ; при инициализации DS должен быть установлен на PM\_seg

mov al,20h

out 20h,al

mov word ptr TSS\_3,SEL\_TSS0

iret

mov al,20h

out 20h,al

mov word ptr TSS\_3,SEL\_TSS1

iret

jmp task\_3

Единственное дополнительное изменение, которое нужно внести, — инициализировать дескриптор TSS задачи task\_1 уже как занятый, так как управление на него будет передаваться командой IRET, что, впрочем, не составляет никаких проблем.

Помните, что во вложенных задачах команда IRET не означает конца программы — следующий вызов задачи всегда передает управление на следующую после IRET команду.

**42.12.2 Дати характеристику системи переривань захищеного режиму**Особливістю системи переривань в захищеному режимі є широке використання **внутрішніх апаратних переривань** - виключень. Виключення виникають як результат такого стану комп'ютерної системи, який не дозволяє виконати поточну команду. В свою чергу неможливість виконання чергової команди в багатьох випадках пов'язана з порушенням правил захисту.

За способом реакції процесора (і наступних можливостей програми обробки переривань), переривання в захищеному режимі поділяють на пастки, помилки, збої та аварії.

У випадку **пастки** в стек записується адреса команди. Тобто команда повністю виконується (якщо це команда між сегментної передачі управління, то змінюється вміст регістрів CS та EIP), а потім вміст CS та EIP записується в стек. Переривання від зовнішніх пристроїв та програмні переривання (команди INT) відносяться до пасток.

У випадку **помилки** в стек заноситься адреса команди, яка визвала виключення. Це дозволяє програмі обробки переривання внести необхідні зміни стану комп'ютерної системи і пере запустити виконання команди.

Але при виключеннях переривається виконання команди. Якщо виключення викликано командою, яка змінює вміст регістрів CS та EIP, то адреса повернення, яка записується в стек, може бути не визначеною. Крім того, виключення захисту можуть виникати при виконанні механізму переривань від зовнішніх пристроїв, коли команда-порушниця взагалі відсутня. Тому при помилках останнім в стек заноситься код помилки, який дозволяє ідентифікувати та локалізувати причину помилки в подібних ситуаціях.

У випадку **збою** повторний пере запуск команди-порушниці не можливий. Виключення типу збій переважно виникають при порушеннях під час переключення задач та реакції на привілеї, що супроводжується іншими порушеннями правил захисту.

У випадку **аварії** подальше виконання програми є неможливим і процесор переходить на процедуру початкового запуску (RESET, процесор пере запускається). Найбільш ймовірною причиною аварії є неможливість виконання стекових операцій (порушення правил захисту для стекових операцій) при реакції на інші виключення.

Для організації обслуговування переривань в захищеному режимі як і в реальному режимі використовується таблиця векторів переривань. На відміну від реального режиму вектором таблиці є не 4-х байтна логічна адреса процедури обробки переривань, а один із спеціальних 8-байтних **системних дескрипторів**, до яких відноситься шлюз переривання, шлюз пастки та шлюз задачі.

Шлюз переривання має наступну структуру

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Байт 7 | Байт 6 | Байт 5 | | | | | | | Байт 4 | | | |
| Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 31-16 | | **P** | **DPL** | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | резерв |
| Селектор сегмента кодів, де знаходиться процедура обробки переривання | | Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 15 - 0 | | | | | | | | | | |
| Байт 3 | Байт 2 | Байт 1 | | | | | | | Байт 0 | | | |

Шлюз переривання містить дані необхідні для визначення фізичної адреси процедури обробки переривань - селектор відповідного дескриптора сегмента кодів, та зміщення процедури обробки переривань в цьому сегменті.

Шлюз пастки має наступну структуру

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Байт 7 | Байт 6 | Байт 5 | | | | | | | Байт 4 | | | |
| Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 31-16 | | **P** | **DPL** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | резерв |
| Селектор сегмента кодів, де знаходиться процедура обробки переривання | | Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 15 - 0 | | | | | | | | | | |
| Байт 3 | Байт 2 | Байт 1 | | | | | | | Байт 0 | | | |

Шлюз пастки відрізняється від шлюзу переривання лише значенням 1 в 0-му розряді 5-того байта дескриптора (молодші чотири біти 5-го байта **системного дескриптора** визначають його тип).

При використанні шлюзу переривання біт IF регістра ознак (EFLAGS) апаратурою встановлюється в 0 і попереднє значення відновлюється під час виконання команди IRET. При використанні шлюзу пастки біт IF не змінюється. В іншому шлюзи пастки та переривань не відрізняються.

Шлюз задачі використовується коли в результаті переривання необхідно переключить процесор на іншу задачу. В даній роботі переключення задач не розглядається.

На відміну від практики використання реального режиму в захищеному режимі таблиця векторів переривань може розміщуватись по довільній фізичній адресі, яка завантажується командою LIDT в старші 32 розряди 48-розрядного системного регістра IDTR. В молодші 16 розрядів цього регістру завантажується розмір таблиці в байтах. Команда LIDT належить до привілейованих команд і може виконуватись лише при **CPL**=0, або в реальному режимі.

При переключенні на процедуру обробки переривань значення поля **DPL** дескриптора сегмента кодів, де розташована ця процедура, не повинен перевищувати значення поточного рівня привілеїв (**DPL**сегменту**≤CPL**). Оскільки переривання та виключення можуть виникнути де завгодно, то процедури обробки переривань розміщують на 0-вому рівні привілеїв, тобто в системній області. При цьому шлюзи розглядаються як дані, тобто **DPL**шлюзу≥ **CPL.**

В лабораторній роботі зовнішні переривання при роботі в захищеному режимі забороняються, а обробляються лише внутрішні та програмні переривання з використанням шлюзу пастки.

В процесорах 80х86 та Pentium жорстко (на апаратному рівні) закріплено наступний розподіл векторів переривань між причинами виникнення виключень:

0-вектор. Ділення на 0. Помилка.

1-вектор. Використовується в налагоджувачах. Може бути як пасткою так і помилкою.

2-вектор. Зовнішнє переривання, яке ігнорує стан біту IF. Пастка.

3-вектор. Використовується в налагоджувачах. Пастка.

4-вектор. Контроль переповнення (виникає, коли виконується команда INTO і OF=1). Помилка.

5-вектор. Контроль переповнення масивів. Генерується командою BOUND, якщо операнд має значення за встановленою межею. Помилка

6-вектор. Неіснуючий код операції. В більшості випадків виникає в результаті помилок в передачах управління (передача управління не на перший байт команди). Коди 0d6h та 0f1h зарезервовані ф.Intel, хоча команди з такими кодами операцій поки що відсутні, але виключення при їх обробці не виникає. Помилка.

7-вектор. Співпроцесор відсутній. Виключення виникає при спробі виконати любу команду співпроцесора, а біт EM (біт 2) (емуляція співпроцесора) регістра CR0 має значення 1. Цим самим процедура обробки переривання 7 може бути використана для програмного моделювання роботи співпроцесора. Крім того, виключення може виникнуть після переключення задач, щоб установить, при необхідності, режим роботи співпроцесора для нової задачі. Помилка.

8-вектор. **Збій.** (Подвійна помилка). Виникає, коли в результаті виконання команди необхідно одночасно формувати декілька виключень.

Якщо спроба перейти на процедуру обробки 8-го переривання також пов'язана з помилкою, то найбільш ймовірно процесор виконає процедуру RESTART.

9-вектор. В процесорах 80х86 та Pentium виключення по цьому вектору не генеруються.

10-вектор. Неправильний сегмент стану задачі (TSS). В багатозадачних системах для кожної задачі формується сегмент TSS, який містить основні системні дані для задачі. Помилка.

11-вектор. Сегмент відсутній. Виключення виникає при завантаженні сегментного регістра селектором дескриптора, біт **P** атрибутів якого має значення 0. Помилка.

12-вектор. Помилка стека. Виникає при спробі звернення до стека командами MOV та їй подібними (використання регістра BP або префікса заміни сегмента SS:) при умові виходу зміщення в сегменті за межі допустимого діапазону. При виконанні команд PUSH та CALL та виходу замежі сегмента стека найбільш ймовірно виникнення аварійної ситуації з наступним виконанням процедури RESTART. Помилка.

13-вектор. Порушення правил захисту. Виникає при всіх порушеннях правил захисту, які не охоплені попередніми виключеннями.

14-вектор. Відсутня сторінка. Генерується при віртуальній сторінковій адресації, коли відповідна сторінка віртуальної пам'яті відсутня в оперативній пам'яті. Помилка

15 - вектор. В процесорах 80х86 та Pentium виключення по цьому вектору не генеруються

16 - вектор. Помилка співпроцесора. Співпроцесор має свій значний набір виключень. Якщо розряд NE (5-тий) регістра CR0 містить 1, то це виключення центрального процесора виникає при появі команди співпроцесора, наступної за тією командою, яка сформувала виключення співпроцесора.

**43.13.2 Охарактеризувати суть захищеного режиму**   
Захищений режим в процесорах ф.Intel вперше був реалізований ще в процесорі 80286, подальшого розвитку набув в процесорі 80386 і підтримується із незначними доповненнями у всіх наступних процесорах лінії 80х86 та Pentium.

В загальному випадку суть захищеного режиму полягає в обмеженні можливостей виконання окремих груп команд в залежності від стану комп'ютерної системи, який програмним шляхом може динамічно змінюватись.

Із наведеного визначення випливає необхідність існування спеціальної групи команд, яка призначена для зміни стану комп'ютерної системи і відповідно накладати ті чи інші обмеження на виконання команд із других груп. Команди із такої спеціальної групи називають системними або привілейованими.

Для контролю за виконанням накладених обмежень використовується механізм внутрішніх апаратних переривань, які мають назву виключення. На відміну від переривань від зовнішніх пристроїв, де механізм переривання спрацьовує лише після повного виконання чергової команди (тобто переривається програма, а не команда), в випадку виключень переривається виконання команди, якщо апаратурою процесора виявлено порушення командою встановлених обмежень (правил захисту). При цьому в стек записується адреса команди-порушниці. Процедура обробки виключення має по цій адресі доступ до команди-порушниці, може виконати необхідні зміни чи стану системи чи самої команди і продовжить виконання програми з перерваної команди, або вивести відповідне повідомлення і знять програму з виконання.

44.14.2 **Охарактеризувати дескриптори та селектори захищеного режиму**

*Селектор* — это 16-битное число следующего формата:

биты 16 – 3: номер дескриптора в таблице (от 0 до 8191)

бит 2: 1 — использовать LDT, 0 — использовать GDT

биты 1 – 0: запрашиваемый уровень привилегий при обращении к сегменту и текущий уровень привилегий для селектора, загруженного в CS

Селектор, содержащий нулевые биты 16 – 3, называется нулевым и используется для загрузки в неиспользуемые сегментные регистры. Любое обращение в сегмент, адресуемый нулевым селектором, приводит к исключению #GP(0), в то время как даже загрузка в сегментный регистр ошибочного селектора вызывает исключение #GР(селектор). Попытка загрузки нулевого селектора в SS или CS также вызывает #GP(0), так как эти селекторы используются всегда.

*Дескриптор* — это 64-битная (восьмибайтная) структура данных, которая может встречаться в таблицах GDT и LDT. Дескриптор способен описывать сегмент кода, сегмент данных, сегмент состояния задачи, быть шлюзом вызова, ловушки, прерывания или задачи. В GDT также может находиться дескриптор LDT.

*Дескриптор сегмента данных или кода* (подробно рассмотрен в главе 6.1)

байт 7: биты 31 – 24 базы сегмента

байт 6:

бит 7: бит гранулярности (0 — лимит в байтах, 1 — лимит в 4-килобайтных единицах)

бит 6: бит разрядности (0 — 16-битный, 1 — 32-битный сегмент)

бит 5: 0

бит 4: зарезервировано для операционной системы

биты 3 – 0: биты 19 – 16 лимита

байт 5: (байт доступа)

бит 7: бит присутствия сегмента

биты 6 – 5: уровень привилегий дескриптора (DPL)

бит 4: 1 (тип дескриптора — не системный)

бит 3: тип сегмента (0 — данных, 1 — кода)

бит 2: бит подчиненности для кода, бит расширения вниз для данных

бит 1: бит разрешения чтения для кода, бит разрешения записи для данных

бит 0: бит доступа (1 — к сегменту было обращение)

байт 4: биты 23 – 16 базы сегмента

байты 3 – 2: биты 15 – 0 базы

байты 1 – 0: биты 15 – 0 лимита

Если в дескрипторе бит 4 байта доступа равен 0, дескриптор называется системным. В этом случае биты 0 – 3 байта доступа определяют один из шестнадцати возможных типов дескриптора (табл. 22).

***Таблица 22.*** *Типы системных дескрипторов*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | Зарезервированный тип | 8 | Зарезервированный тип |
| 1 | Свободный 16-битный TSS | 9 | Свободный 32-битный TSS |
| 2 | Дескриптор таблицы LDT | A | Зарезервированный тип |
| 3 | Занятый 16-битный TSS | B | Занятый 16-битный TSS |
| 4 | 16-битный шлюз вызова | C | 32-битный шлюз вызова |
| 5 | Шлюз задачи | D | Зарезервированный тип |
| 6 | 16-битный шлюз прерывания | E | 32-битный шлюз прерывания |
| 7 | 16-битный шлюз ловушки | F | 32-битный шлюз ловушки |

*Дескрипторы шлюзов*

Дальние CALL или JMP на адрес с любым смещением и с селектором, указывающим на дескриптор шлюза вызова, приводят к передаче управления по адресу, указанному в дескрипторе. Обычно такие дескрипторы используются для передачи управления между сегментами с различными уровнями привилегий (см. главу 10.7).

CALL или JMP на адрес с селектором, указывающим на шлюз задачи, приводят к переключению задач (см. главу 10.8).

Шлюзы прерываний и ловушек используются для вызова обработчиков соответственно прерываний и исключений типа ловушки (см. главу 10.5).

байты 7 – 6: биты 31 – 16 смещения (0 для 16-битных шлюзов и шлюза задачи)

байт 5: (байт доступа)

бит 7: бит присутствия сегмента

биты 6 – 5: DPL — уровень привилегий дескриптора

бит 4: 0

биты 3 – 0: тип шлюза (3, 4, 5, 6, 7, В, С, Е, 7)

байт 4:

биты 7 – 5: 000

биты 4 – 0: 00000 или (для шлюза вызова) число двойных слов, которые будут скопированы из стека вызывающей задачи в стек вызываемой

байты 3 – 2: селектор сегмента

байты 1 – 0: биты 15 – 0 смещения (0 для шлюза задачи)

*Дескрипторы TSS и LDT*

Эти два типа дескрипторов применяются в многозадачном режиме, о котором рассказано далее. TSS — сегмент состояния задачи, используемый для хранения всей необходимой информации о каждой задаче в многозадачном режиме. LDT — таблица локальных дескрипторов, своя для каждой задачи.

Форматы этих дескрипторов совпадают с форматом дескриптора для сегмента кода или данных, за исключением того, что бит разрядности всегда равен нулю и, естественно, системный бит равен нулю, и биты 3 – 0 байта доступа содержат номер типа сегмента (1, 2, 3, 9, В). Команды JMP и CALL на адрес с селектором, соответствующим TSS незанятой задачи, приводят к переключению задач.

45.15.2 Охарактеризувати проблему повторного входження при перериваннях та методи її вирішення

Наявність механізму апаратних переривань може призвести до ефекту повторного входження, суть якого полягає в наступному. Оскільки процедура обробки переривань від деякого пристрою Devicex (нехай процедура називається Pdevx) сама може також бути перервана, то її робота може бути ще не закінченою, а пристрій Devicex знову подає сигнал запиту на переривання. Якщо такий запит буде підтверджений процесором, то процедура Pdevx обробки переривань буде запущена повторно, хоча попередній її запуск ще до кінця не виконаний. В результаті можливі помилки наступних класів:

* a) помилки в управлінні пристроєм Devicex
* b) помилки при продовжені виконання перерваної процедури Pdevx із-за можливої "порчі" даних в загальних областях пам'яті попереднім виконанням цієї ж процедури

Проблема повторного входження стосується не тільки безпосередньо процедур обробки переривань, а і любих других процедур, які викликаються із процедур обробки переривань і допускають своє переривання.

Для запобігання можливим помилкам в функціонуванні комп'ютерних систем із-за повторного входження використовуються апаратно-програмні та програмні методи.

До апаратно-програмних методів відноситься блокування контролером переривань запитів на переривання поточного і нижчого пріоритетів. Таким чином, команди розблокування, які подані в п.2.3.4 повинні виконуватись безпосередньо перед поверненням в перервану програму, після завершення сеансів управління та зв'язку з зовнішнім пристроєм та обробки даних. Даний метод гарантує блокування повторних входжень безпосередньо для процедур обробки переривань, але не запобігає повторному входженню процедур, які викликаються із процедур обробки переривань.

До програмних методів відноситься створення процедур, які дозволяють повторне входження або створення та використанні ознаки зайнятості процедур. Процедуру з повторним входженням можна створити при умові, що вона не змінює глобальні дані. Локальні дані, які можуть змінюватись під час виконання процедури розміщують в стеці, що унеможливлює вплив повторного входження на попередні. Прикладом організації процедури з повторним входженням є процедура Vcount із початкового модуля мультипрограмної моделі.

Якщо процедура змінює глобальні дані, то її повторне входження може призвести до помилок. В даному випадку при входженні в процедуру установлюється ознака її зайнятості. Перед поверненням із процедури, ознака скидається. Таким чином, для запобігання повторному входженню, необхідно перед викликом аналізувати ознаку зайнятості. Подібний механізм реалізовано в MS-DOS. Звернення до функцій MS-DOS можна розглядати як виклик процедур, які в своїй більшості не забезпечують повторне входження. Прикладні програми досить часто використовують функції операційної системи, тому існує значна ймовірність їх переривання. Але в процедурах обробки переривань також може виникнуть необхідність використання функцій операційної системи. Тому в MS-DOS реалізована функція за номером 34h, яка видає адресу ознаки зайнятості. Звичайно, що цю функцію необхідно використовувати поза процедурами обробки переривань і до "перехоплення" переривань, якщо користувацька процедура обробки переривань використовує функції MS-DOS. В моделі мультипрограмної системи користувацька процедура обробки переривань від клавіатури (userint9) при відтисканні клавіші Esc повинна забезпечить завершення роботи всієї програми шляхом використання функції MS-DOS 4ch. При цьому аналізується ознака зайнятості і, якщо MS-DOS зайнята, то завершення роботи моделі не відбувається, а робота процедури userint9 завершується. Тут використано психологічний фактор - якщо завершення роботи моделі не відбулось, то користувач напевно натисне ще декілька раз клавішу Esc, а за цей час MS-DOS звільниться. В інших випадках реакція на зайнятість MS-DOS може бути значно складнішою, але необхідно пам’ятати, що програмне очікування звільнення MS-DOS буде безкінечним.

**46.16.2 Охарактеризувати проблему повторного входження при перериваннях та методи її вирішення**

Наявність механізму апаратних переривань може призвести до ефекту повторного входження, суть якого полягає в наступному. Оскільки процедура обробки переривань від деякого пристрою Devicex (нехай процедура називається Pdevx) сама може також бути перервана, то її робота може бути ще не закінченою, а пристрій Devicex знову подає сигнал запиту на переривання. Якщо такий запит буде підтверджений процесором, то процедура Pdevx обробки переривань буде запущена повторно, хоча попередній її запуск ще до кінця не виконаний. В результаті можливі помилки наступних класів:

* a) помилки в управлінні пристроєм Devicex
* b) помилки при продовжені виконання перерваної процедури Pdevx із-за можливої "порчі" даних в загальних областях пам'яті попереднім виконанням цієї ж процедури

Проблема повторного входження стосується не тільки безпосередньо процедур обробки переривань, а і любих других процедур, які викликаються із процедур обробки переривань і допускають своє переривання.

Для запобігання можливим помилкам в функціонуванні комп'ютерних систем із-за повторного входження використовуються апаратно-програмні та програмні методи.

До апаратно-програмних методів відноситься блокування контролером переривань запитів на переривання поточного і нижчого пріоритетів. Таким чином, команди розблокування, які подані в п.2.3.4 повинні виконуватись безпосередньо перед поверненням в перервану програму, після завершення сеансів управління та зв'язку з зовнішнім пристроєм та обробки даних. Даний метод гарантує блокування повторних входжень безпосередньо для процедур обробки переривань, але не запобігає повторному входженню процедур, які викликаються із процедур обробки переривань.

До програмних методів відноситься створення процедур, які дозволяють повторне входження або створення та використанні ознаки зайнятості процедур. Процедуру з повторним входженням можна створити при умові, що вона не змінює глобальні дані. Локальні дані, які можуть змінюватись під час виконання процедури розміщують в стеці, що унеможливлює вплив повторного входження на попередні. Прикладом організації процедури з повторним входженням є процедура Vcount із початкового модуля мультипрограмної моделі.

Якщо процедура змінює глобальні дані, то її повторне входження може призвести до помилок. В даному випадку при входженні в процедуру установлюється ознака її зайнятості. Перед поверненням із процедури, ознака скидається. Таким чином, для запобігання повторному входженню, необхідно перед викликом аналізувати ознаку зайнятості. Подібний механізм реалізовано в MS-DOS. Звернення до функцій MS-DOS можна розглядати як виклик процедур, які в своїй більшості не забезпечують повторне входження. Прикладні програми досить часто використовують функції операційної системи, тому існує значна ймовірність їх переривання. Але в процедурах обробки переривань також може виникнуть необхідність використання функцій операційної системи. Тому в MS-DOS реалізована функція за номером 34h, яка видає адресу ознаки зайнятості. Звичайно, що цю функцію необхідно використовувати поза процедурами обробки переривань і до "перехоплення" переривань, якщо користувацька процедура обробки переривань використовує функції MS-DOS. В моделі мультипрограмної системи користувацька процедура обробки переривань від клавіатури (userint9) при відтисканні клавіші Esc повинна забезпечить завершення роботи всієї програми шляхом використання функції MS-DOS 4ch. При цьому аналізується ознака зайнятості і, якщо MS-DOS зайнята, то завершення роботи моделі не відбувається, а робота процедури userint9 завершується. Тут використано психологічний фактор - якщо завершення роботи моделі не відбулось, то користувач напевно натисне ще декілька раз клавішу Esc, а за цей час MS-DOS звільниться. В інших випадках реакція на зайнятість MS-DOS може бути значно складнішою, але необхідно пам’ятати, що програмне очікування звільнення MS-DOS буде безкінечним.

47.17.2 Дати характеристику виключень при роботі співпроцесора

Обработка особых случаев

В арифметическом сопроцессоре имеются два механизма обработки ошибок, возникающих при выполнении различных команд.

Первый механизм основан на генерации так называемого прерывания особого случая (INT 10h). Это прерывание вырабатывается в том случае, когда происходит какая-нибудь ошибка (например, деление на нуль) при условии, что соответствующие биты масок особых случаев в регистре управления не установлены.

При втором способе обработки ошибок все особые случаи маскируются (соответствующие биты управляющего регистра устанавливаются в единицу) и в случае ошибки сопроцессор в качестве результата возвращает некоторое заранее известное особое значение (нечисло, неопределенность или бесконечность).

Программист может выбирать между этими способами обработки ошибок, маскируя или разрешая прерывание по особому случаю. Если прерывание особого случая замаскировано, можно использовать такой способ обнаружения ошибки:

сбросить флажки особых случаев в регистре состояния;

выполнить одну или несколько команд сопроцессора;

проверить состояние флажков особых случаев в регистре состояния, в частности, бит суммарной ошибки ES;

если какой-либо флажок установлен, вызвать программу обработки ошибочной ситуации;

в программе обработки ошибочной ситуации можно сбросить флажки особых случаев, записав соответствующее значение в регистр состояния

Особые значения результатов работы сопроцессора

После выполнения команды сопроцессора, производящей какие-либо вычисления, бывает полезно проверить получившийся результат на принадлежность к множеству особых значений.

Неточный результат

В результате выполнения некоторых операций может возникнуть такая ситуация, когда невозможно точно представить результат. Например, при делении числа 1.0 на 3.0 результатом является бесконечная периодическая двоичная дробь 0.010101.. (или десятичная бесконечная дробь 0.333...). Такое число не может быть представлено точно ни в одном формате вещественных чисел.

Обычно неточный результат является результатом округления и может не рассматриваться как ошибка.

Переполнение

Если результат выполнения операции слишком велик и не может быть представлен в формате приемника результата, фиксируется особый случай переполнения.

Этот особый случай обязательно произойдет, например, при сложении максимального числа расширенной точности самим с собой или при преобразовании этого числа в формат с двойной или одинарной точностью.

Так как для хранения промежуточных результатов используется 80-битовое представление, при выполнении операций над числами с одинарной или двойной точностью переполнения, как правило, не происходит. Огромный диапазон чисел с расширенной точностью гарантирует правильность представления больших по абсолютной величине результатов операций с числами одинарной и двойной точности.

Антипереполнение

Антипереполнение возникает тогда, когда результат слишком мал для его представления в формате приемника результата операции, но все же отличен от нуля. Например, если делается попытка преобразовать наименьшее положительное число с расширенной точностью в формат числа с двойной или одинарной точностью.

Если вы используете числа только с двойной или одинарной точностью, а для хранения промежуточных результатов используете формат с расширенной точностью, особый случай антипереполнения, как правило, не возникает.

Деление на нуль

Этот особый случай возникает при попытке выполнить деление конечного ненулевого числа на нуль.

В афинном режиме при делении конечных (положительных или отрицательных) чисел на нуль (положительный или отрицательный) в качестве результата возвращается бесконечность. Знак этой бесконечности зависит от знака делимого и от знака нуля. Например, при делении положительного ненулевого числа на положительный нуль получается положительная бесконечность, при делении положительного ненулевого числа на отрицательный нуль - отрицательная бесконечность.

В проективном режиме, а также при попытке деления нуля на нуль возникает особый случай недействительной операции, который будет рассмотрен ниже.

Недействительная операция

Этот особый случай возникает при попытке выполнения таких запрещенных команд, как деление нуля на нуль, извлечения корня из отрицательного числа, обращение к несуществующему регистру сопроцессора или при попытке использования в качестве операндов команд нечисел, неопределенностей или бесконечности (для трансцендентных функций).

Денормализованный операнд

Выше уже говорилось о том, что сопроцессор использует операнды в нормализованной форме. Однако при выполнении операции может оказаться, что результат слишком мал по абсолютной величине для представления его в нормализованной форме. Можно было бы считать такой результат нулевым, однако это привело бы к снижению точности вычислений или даже к грубым ошибкам. Например, вычисляется следующее выражение: (y-x)+x;

Если разность (y-x) вызывает антипереполнение и в качестве результата берется нулевое значение, то после вычисления всего выражения получится x. Если же пойти на расширение диапазона представления чисел за счет снижения точности и сформировать результат вычисления разности (y-x) как денормализованное число, выражение будет вычислено правильно и в результате получится y.

Таким образом, иногда целесообразно замаскировать особый случай денормализованного операнда и использовать денормализованные числа. Однако при попытке деления на ненормализованное число или извлечения из него квадратного корня фиксируется особый случай недействительной операции.

48.18.2 Охарактеризувати типову структуру процедури оброки переривань від зовнішнього пристрою

Прерывания от внешних устройств, или аппаратные прерывания — это то, что понимается под термином «прерывание». Внешние устройства (клавиатура, дисковод, таймер, звуковая карта и т.д.) подают сигнал, по которому процессор прерывает выполнение программы и передает управление наобработчик прерывания. Всего на персональных компьютерах используется 15 аппаратных прерываний, хотя теоретически возможности архитектуры позволяют довести их число до 64.

Рассмотрим их кратко в порядке убывания приоритетов (прерывание имеет более высокий приоритет, и это означает, что, пока не завершился его обработчик, прерывания с низкими приоритетами будут ждать своей очереди).

IRQ0 (INT 8) — прерывание системного таймера. Это прерывание вызывается 18,2 раза в секунду. Стандартный обработчик этого прерывания вызывает INT 1Ch при каждом вызове, так что, если программе необходимо только регулярно получать управление, а не перепрограммировать таймер, рекомендуется использовать прерывание 1Ch.

IRQ1 (INT 9) — прерывание клавиатуры. Это прерывание вызывается при каждом нажатии и отпускании клавиши на клавиатуре. Стандартный обработчик этого прерывания выполняет довольно много функций, начиная с перезагрузки по Ctrl-Alt-Del и заканчивая помещением кода клавиши в буфер клавиатуры BIOS.

IRQ2 — к этому входу на первом контроллере прерываний подключены аппаратные прерывания IRQ8 – IRQ15, но многие BIOS перенаправляют IRQ9 на INT 0Ah.

IRQ8 (INT 70h) — прерывание часов реального времени. Это прерывание вызывается часами реального времени при срабатывании будильника и если они установлены на генерацию периодического прерывания (в последнем случае IRQ8 вызывается 1024 раза в секунду).

IRQ9 (INT 0Ah или INT 71h) — прерывание обратного хода луча. Вызывается некоторыми видеоадаптерами при обратном ходе луча. Часто используется дополнительными устройствами (например, звуковыми картами, SCSI-адаптерами и т.д.).

IRQ10 (INT 72h) — используется дополнительными устройствами.

IRQ11 (INT 73h) — используется дополнительными устройствами.

IRQ12 (INT 74h) — мышь на системах PS используется дополнительными устройствами.

IRQ13 (INT 02h или INT 75h) — ошибка математического сопроцессора. По умолчанию это прерывание отключено как на FPU, так и на контроллере прерываний.

IRQ14 (INT 76h) — прерывание первого IDE-контроллера «операция завершена».

IRQ15 (INT 77h) — прерывание второго IDE-контроллера «операция завершена».

IRQ3 (INT 0Bh) — прерывание последовательного порта COM2 вызывается, если порт COM2 получил данные.

IRQ4 (INT 0Ch) — прерывание последовательного порта СОМ1 вызывается, если порт СОМ1 получил данные.

IRQ5 (INT 0Dh) — прерывание LPT2 используется дополнительными устройствами.

IRQ6 (INT 0Eh) — прерывание дисковода «операция завершена».

IRQ7 (INT 0Fh) — прерывание LPT1 используется дополнительными устройствами.

Самые полезные для программ аппаратные прерывания — прерывания системного таймера и клавиатуры. Так как их стандартные обработчики выполняют множество функций, от которых зависит работа системы, их нельзя заменять полностью, как мы делали это с обработчиком INT 5. Следует обязательно вызвать предыдущий обработчик, передав ему управление следующим образом (если его адрес сохранен в переменной old\_handler, как в предыдущих примерах):

pushf

call old\_handler

Эти две команды выполняют действие, аналогичное команде INT (сохранить флаги в стеке и передать управление подобно команде call), так что, когда обработчик завершится командой IRET, управление вернется в нашу программу. Так удобно вызывать предыдущий обработчик в начале собственного. Другой способ — простая команда jmp:

jmp cs:old\_handler

приводит к тому, что, когда старый обработчик выполнит команду IRET, управление сразу же перейдет к прерванной программе. Этот способ применяют, если нужно, чтобы сначала отработал новый обработчик, а потом он передал бы управление старому.

Посмотрим, как работает перехват прерывания от таймера на следующем примере:

; timer.asm

; демонстрация перехвата прерывания системного таймера: вывод текущего времени

; в левом углу экрана

.model tiny

.code

.186 ; для pusha/popa и сдвигов

org 100h

start proc near

; сохранить адрес предыдущего обработчика прерывания 1Ch

mov ax,351Ch ; АН = 35h, AL = номер прерывания

int 21h ; функция DOS: определить адрес обработчика

mov word ptr old\_int1Ch,bx ; прерывания

mov word ptr old\_int1Ch+2,es ; (возвращаетсяв ES:BX)

; установить наш обработчик

mov ax,251Ch ; АН = 25h, AL = номер прерывания

mov dx,offset int1Ch\_handler ; DS:DX - адресобработчика

int 21h ; установить обработчик прерывания 1Ch

; здесь размещается собственно программа, например вызов command.com

mov ah,1

int 21h ; ожидание нажатия на любую клавишу

; конец программы

; восстановить предыдущий обработчик прерывания 1Ch

mov ax,251Ch ; АН = 25h, AL = номер прерывания

mov dx,word ptr old\_int1Ch+2

mov ds,dx

mov dx,word ptr cs:old\_int1Ch ; DS:DX - адресобработчика

int 21h

ret

old\_int1Ch dd ? ; здесь хранится адрес предыдущего обработчика

start\_position dw 0 ; позиция на экране, в которую выводится текущее время

start endp

; обработчик для прерывания 1Ch

; выводит текущее время в позицию start\_position на экране

; (только в текстовом режиме)

int1Ch\_handler proc far

pusha ; обработчик аппаратного прерывания

push es ; должен сохранять ВСЕ регистры

push ds

push cs ; на входе в обработчик известно только

pop ds ; значение регистра CS

mov ah,02h ; Функция 02h прерывания 1Ah:

int 1Ah ; чтение времени из RTC,

jc exit\_handler ; если часы заняты - в другой раз

; AL = час в BCD-формате

call bcd2asc ; преобразовать в ASCII,

mov byte ptr output\_line[2],ah ; поместить их в

mov byte ptr output\_line[4],al ; строку output\_line

mov al,cl ; CL = минута в BCD-формате

call bcd2asc

mov byte ptr output\_line[10],ah

mov byte ptr output\_line[12],al

mov al,dh ; DH = секунда в BCD-формате

call bcd2asc

mov byte ptr output\_line[16],ah

mov byte ptr output\_line[18],al

mov cx,output\_line\_l ; число байт в строке - в СХ

push 0B800h

pop es ; адрес в видеопамяти

mov di,word ptr start\_position ; в ES:DI

mov si,offset output\_line ; адресстрокив DS:SI

cld

rep movsb ; скопироватьстроку

exit\_handler:

pop ds ; восстановить все регистры

pop es

popa

jmp cs:old\_int1Ch ; передать управление предыдущему обработчику

; процедура bcd2asc

; преобразует старшую цифру упакованного BCD-числа из AL в ASCII-символ,

; который будет помещен в АН, а младшую цифру - в ASCII-символ в AL

bcd2asc proc near

mov ah,al

and al,0Fh ; оставить младшие 4 бита в AL

shr ah,4 ; сдвинуть старшие 4 бита в АН

or ах,3030h ; преобразовать в ASCII-символы

ret

bcd2asc endp

; строка " 00h 00:00 " с атрибутом 1Fh (белый на синем) после каждого символа

output\_line db ' ',1Fh,'0',1Fh,'0',1Fh,'h',1Fh

db ' ',1Fh,'0',1Fh,'0',1Fh,':',1Fh

db '0',1Fh,'0',1Fh,' ',1Fh

output\_line\_l equ $ - output\_line

int1Ch\_handler endp

end start

Если в этом примере вместо ожидания нажатия на клавишу поместить какую-нибудь программу, работающую в текстовом режиме, например tinyshell из главы 1.3, она выполнится как обычно, но в правом верхнем углу будет постоянно показываться текущее время, то есть такая программа будет осуществлять два действия одновременно. Именно для этого и применяется механизм аппаратных прерываний — они позволяют процессору выполнять одну программу, в то время как отдельные программы следят за временем, считывают символы из клавиатуры и помещают их в буфер, получают и передают данные через последовательные и параллельные порты и даже обеспечивают многозадачность, переключая процессор между разными задачами по прерыванию системного таймера.

Разумеется, обработка прерываний не должна занимать много времени: если прерывание происходит достаточно часто (например, прерывание последовательного порта может происходить 28 800 раз в секунду), его обработчик обязательно должен выполняться за более короткое время. Если, например, обработчик прерывания таймера будет выполняться 1/32,4 секунды, то есть половину времени между прерываниями, вся система будет работать в два раза медленнее. А если еще одна программа с таким же долгим обработчиком перехватит это прерывание, система остановится совсем. Именно поэтому обработчики прерываний принято писать исключительно на ассемблере.

49.19.2 Охарактеризувати поняття «перехоплення» переривання та методи його виконання

Операційні системи мають в своєму складі процедури обробки переривань від усіх стандартних зовнішніх пристроїв ПЕОМ. При роботі програм операційної системи встановлюються відповідні вектори в таблиці векторів переривань. В реальному режимі процесорів 80х86 та Pentium інші програми мають можливість безпосереднього доступу як до зовнішніх пристроїв так і до таблиці векторів переривань. Нехай, наприклад, користувач бажає самостійно вводить дані з клавіатури ПЕОМ (N=9). Він повинен створити власну процедуру обробки переривань від клавіатури (нехай вона має назву Userint9), зберегти серед своїх даних системний вектор переривань від клавіатури та записати свій. По закінченню роботи своєї програми необхідно відновити системний вектор. Вказані дії можуть бути виконані, наприклад, наступним чином

**Xor ax,ax**

**Mov gs,ax**

**Mov si,9\*4**

**Mov eax,gs:[si]**

**Mov syst\_vec\_9,eax** ; збереження системного вектора

**Mov x,offset userint9**

**Mov gs:[si],ax** ;встановити зміщення процедури

;користувача в векторі переривань

**Mov ax,seg userint9**

**Mov gs:[si+2],ax** ;встановити сегментну складову логічної

;адреси процедури користувача

;в векторі переривань

**...**

**...**

**Mov eax,syst\_vec\_9**

**Mov si,9\*4**

**Mov gs:[si],eax ;**відновлення системного вектора

Розглянутий приклад організації перехоплення переривань має той недолік, що "перехоплення" відбувається "за спиною" операційної системи. В результаті операційна система не повідомлена про проведене "перехоплення" і може його трактувати як пошкодження таблиці векторів переривань з примусовим відновленням "правильного" вектора. Щоб цього не сталось в MS-DOS реалізовані спеціальні функції для одержання вектора переривань (функція 35h) та встановлення вектора переривань (функція 25h). За допомогою цих функцій MS-DOS веде облік "перехоплених" переривань. Приклад використання таких функцій з детальними коментаріями наведений в моделі багатопрограмної системи (процедури Setint8, Setint9). По закінчені своєї роботи програма користувача повинна відновити системні вектори переривань (процедури Retin8, Retint9).

При перехопленні переривань можливі наступні варіанти організації процедури користувача та її взаємодії з системною процедурою.

*По першому* із них системна процедура обробки переривань не використовується, а процедура користувача повністю **бере на себевсі дії** по обслуговуванню зовнішнього пристрою. До таких дій належать подання на зовнішній пристрій необхідних команд управління для проведення та завершення сеансу зв'язку, що потребує вивчення алгоритму функціонування зовнішнього пристрою. Крім того на контролер переривань необхідно подать згідно з п.2.3.4 команду розблокування переривань поточного та нижчого пріоритетів. Вказані дії виконані в процедурі Userint9 при введені з клавіатури символу Esc. При цьому системна процедура обробки переривань від клавіатури не викликається, тобто від операційної системи "приховується" факт натиснення клавіші Esc. При самостійній роботі з клавіатурою необхідно сформувати імпульс, який подається на клавіатуру (через 7-й розряд порту 61h) і який інформує клавіатуру про завершення введення даних. Без такого імпульсу подальша робота клавіатури буде заблокована.

*В другому* варіанті мінімізується вплив процедури користувача на обробку переривань від зовнішнього пристрою. Тобто, по стану зовнішнього пристрою процедура користувача визначає, що черговий сеанс зв'язку з зовнішнім пристроєм її не стосується, а тому необхідно передати управління системній процедурі обробки переривання від даного зовнішнього пристрою. Управління на системну процедуру передається за допомогою команди JMP, забезпечуючи тим самим повернення в перервану програму безпосередньо із системної процедури. В процедурі Userint9 даний варіант реалізується при натисканні любої клавіші клавіатури крім клавіші Esc. Якщо натиснута люба друга клавіша, то процедура користувача тільки "спостерігає" за зовнішнім пристроєм, а всі дії покладаються на системну процедуру.

*Третій варіант* характеризується як безумовним виконанням системної процедури обробки переривань так і безумовним виконанням досить складної користувацької процедури. Прикладом може бути обробка переривань від таймера. Відключення системної процедури обробки переривань від таймера може призвести до збоїв в роботі операційної системи. Крім того, системна процедура повинна виконуватись в першу чергу, щоб унеможливити потенційний вплив користувацької процедури на періодичність ініціалізації системної процедури. Таким чином системна процедура повинна бути негайно викликана із користувацької процедури з поверненням по команді IRET із системної процедури в користувацьку. При цьому системна процедура виконає всі стандартні дії із відповідними зовнішніми пристроями, включаючи контролер переривань, а користувацькій процедурі залишиться виконання лише своїх особливих функцій.

50.20.2 Дати детальну характеристику поняття співпроцесор на прикладі FPUв процесорах 8086

Арифметический сопроцессор (FPU) предназначен для выполнения операций над числами в формате с плавающей точкой (вещественные числа) и длинными целыми числами. Он значительно (в десятки раз) ускоряет вычисления, связанные с вещественными числами. Сопроцессор может выполнять как простые арифметические операции (сложение, вычитание, умножение, деление и т.п.), так и вычислять значения различных функций (синус, косинус, тангенс, логарифмы и т. п.)

Основная область применения арифметического сопроцессора - научные расчеты, машинная графика и, разумеется, игры, моделирующие реальный мир (визуальный или звуковой), поскольку такие игры, как правило, включают и машинную графику, и инженерные расчеты.

В играх, где оказывается задействован весь вычислительный потенциал компьютера, непосредственное программирование арифметического сопроцессора позволяет более оптимально построить код программы и ускорить вычисления. Многие среды программирования (например, основанные на языке программирования Си/Си++, а также Borland Delphi) позволяют делать ассемблерные вставки, которые дают возможность напрямую обращаться к сопроцессору. Это, в свою очередь, позволяет отказаться от подключения дополнительных библиотек, необходимых для работы с вещественными числами. В самом деле, зачем подключать целую библиотеку, увеличивая размеры программы, если необходимо всего-навсего вычислить значения тригонометрических функций и осуществить операции умножения и деления. Разумеется, что ответственность за корректность операндов и проверку результата в случае непосредственного программирования арифметического сопроцессора полностью ложится на программиста.

В настоящее время подавляющее большинство персональных компьютеров в качестве центрального процессора используют разновидности процессора Pentium фирмы Intel. Этот процессор имеет встроенный математический сопроцессор. По этой причине ниже не рассматриваются особые случаи, связанные с программированием преддыдущих моделей сопроцессоров Intel.

Принципы работы

Математический сопроцессор запускается центральным процессором. После запуска он выполняет все вычисления самостоятельно и параллельно с работой центрального процессора. Если центральный процессор направляет очередную команду сопроцессору в тот момент, когда сопроцессором еще не закончена предыдущая команда, центральный процессор переводится в состояние ожидания. Если же сопроцессор ничем не занят, центральный процессор, направив команду сопроцессору, продолжает свою работу, не дожидаясь завершения вычисления. Следовательно, всегда необходимо помнить, что вычисления, производимые математическим сопроцессором, выполняются параллельно с работой центрального процессора. Иногда последовательное расположение команд сопроцессора и центрального процессора в листинге программы создают иллюзию последовательного их выполнения. Проще говоря, возможна такая ситуация, что когда центральный процессор обратиться к ячейке памяти, куда арифметический сопроцессор должен был записать результат своих вычислений, то результата в ячейке может и не быть (точнее, содержимое ячейки памяти может быть произвольным). Все дело тут в том, что сопроцессор еще не закончил вычислений. Впрочем, есть специальные средства синхронизации (команда FWAIT).

**51. 21.2 Дати характеристику переключенню задач в захищеному режимі**

Переключение задач осуществляется, если:

* текущая задача выполняет дальний JMP или CALL на шлюз задачи или прямо на TSS;
* текущая задача выполняет IRET, если флаг NT равен 1;
* происходит прерывание или исключение, в качестве обработчика которого в IDT записан шлюз задачи.

При переключении процессор выполняет следующие действия:

1. Для команд CALL и JMP проверяет привилегии (CPL текущей задачи и RPL селектора новой задачи не могут быть больше, чем DPL шлюза или TSS, на который передается управление).
2. Проверяется дескриптор TSS (его бит присутствия и лимит).
3. Проверяется, что новый TSS, старый TSS и все дескрипторы сегментов находятся в страницах, отмеченных как присутствующие.
4. Сохраняется состояние задачи.
5. Загружается регистр TR. Если на следующих шагах происходит исключение, его обработчику придется доделывать переключение задач, вместо того чтобы повторять ошибочную команду.
6. Тип новой задачи в дескрипторе изменяется на занятый и устанавливается флаг TS в CR0.
7. Загружается состояние задачи из нового TSS: LDTR, CR3, EFLAGS, EIP, регистры общего назначения и сегментные регистры.

Если переключение задачи вызывается командами JUMP, CALL, прерыванием или исключением, селектор TSS предыдущей задачи записывается в поле связи новой задачи и устанавливается флаг NT. Если флаг NT установлен, команда IRET выполняет обратное переключение задач.

При любом запуске задачи ее тип изменяется в дескрипторе на занятый. Попытка вызвать такую задачу приводит к #GP, сделать задачу снова свободной можно, только завершив ее командой IRET или переключившись на другую задачу командой JMP.

Покажем, как создавать задачи и переключаться между ними на следующем примере.

; pm4.asm

; Пример программы, выполняющей переключение задач.

; Запускает две задачи, передающие управление друг другу 80 раз, задачи выводят

; на экран символы ASCII с небольшой задержкой

;

; Компиляция:

; TASM:

; tasm /m pm4.asm

; tlink /x /3 pm4.obj

; WASM:

; wasm pm4.asm

; wlink file pm4.obj form DOS

; MASM:

; ml /c pm4.asm

; link pm4.obj,,NUL,,,

.386p

RM\_seg segment para public "CODE" use16

assume cs:RM\_seg,ds:PM\_seg,ss:stack\_seg

start:

; подготовить сегментные регистры

push PM\_seg

pop ds

; проверить, не находимся ли мы уже в РМ

mov eax,cr0

test al,1

jz no\_V86

; сообщить и выйти

mov dx,offset v86\_msg

err\_exit:

push cs

pop ds

mov ah,9

int 21h

mov ah,4Ch

int 21h

; убедиться, что мы не под Windows

no\_V86:

mov ax,1600h

int 2Fh

test al,al

jz no\_windows

; сообщить и выйти

mov dx,offset win\_msg

jmp short err\_exit

; сообщения об ошибках при старте

v86\_msg db "Процессор в режиме V86 - нельзя переключиться в РМ$"

win\_msg db "Программа запущена под Windows - нельзя перейти в кольцо 0$"

; итак, мы точно находимся в реальном режиме

no\_windows:

; очиститьэкран

mov ax,3

int 10h

; вычислить базы для всех дескрипторов сегментов данных

xor еах,еах

mov ax,RM\_seg

shl eax,4

mov word ptr GDT\_16bitCS+2,ax

shr eax,16

mov byte ptr GDT\_16bitCS+4,al

mov ax,PM\_seg

shl eax,4

mov word ptr GDT\_32bitCS+2,ax

mov word ptr GDT\_32bitSS+2,ax

shr eax,16

mov byte ptr GDT\_32bitCS+4,al

mov byte ptr GDT\_32bitSS+4,al

; вычислить линейный адрес GDT

xor eax,eax

mov ax,PM\_seg

shl eax,4

push eax

add eax,offset GDT

mov dword ptr gdtr+2,eax

; загрузить GDT

lgdt fword ptr gdtr

; вычислить линейные адреса сегментов TSS наших двух задач

pop eax

push eax

add eax,offset TSS\_0

mov word ptr GDT\_TSS0+2,ax

shr eax,16

mov byte ptr GDT\_TSS0+4,al

pop eax

add eax,offset TSS\_1

mov word ptr GDT\_TSS1+2,ax

shr eax,16

mov byte ptr GDT\_TSS1+4,al

; открыть А20

mov al,2

out 92h,al

; запретить прерывания

cli

; запретить NMI

in al,70h

or al,80h

out 70h,al

; переключиться в РМ

mov eax,cr0

or al,1

mov cr0,eax

; загрузить CS

db 66h

db 0EAh

dd offset PM\_entry

dw SEL\_32bitCS

RM\_return:

; переключиться в реальный режим RM

mov eax,cr0

and al,0FEh

mov cr0,eax

; сбросить очередь предвыборки и загрузить CS

db 0EAh

dw $+4

dw RM\_seg

; настроить сегментные регистры для реального режима

mov ax,PM\_seg

mov ds,ax

mov es,ax

mov ax,stack\_seg

mov bx,stack\_l

mov ss,ax

mov sp,bx

; разрешить NMI

in al,70h

and al,07FH

out 70h,al

; разрешить прерывания

sti

; завершить программу

mov ah,4Ch

int 21h

RM\_seg ends

PM\_seg segment para public "CODE" use32

assume cs:PM\_seg

; таблица глобальных дескрипторов

GDT label byte

db 8 dup(0)

GDT\_flatDS db 0FFh,0FFh,0,0,0,10010010b,11001111b,0

GDT\_16bitCS db 0FFh,0FFh,0,0,0,10011010b,0,0

GDT\_32bitCS db 0FFh,0FFh,0,0,0,10011010b,11001111b,0

GDT\_32bitSS db 0FFh,0FFh,0,0,0,10010010b,11001111b,0

; сегмент TSS задачи 0 (32-битный свободный TSS)

GDT\_TSS0 db 067h,0,0,0,0,10001001b,01000000b,0

; сегмент TSS задачи 1 (32-битный свободный TSS)

GDT\_TSS1 db 067h,0,0,0,0,10001001b,01000000b,0

gdt\_size = $ - GDT

gdtr dw gdt\_size-1 ; размер GDT

dd ? ; адрес GDT

; используемые селекторы

SEL\_flatDS equ 001000b

SEL\_16bitCS equ 010000b

SEL\_32bitCS equ 011000b

SEL\_32bitSS equ 100000b

SEL\_TSS0 equ 101000b

SEL\_TSS1 equ 110000b

; сегмент TSS\_0 будет инициализирован, как только мы выполним переключение

; из нашей основной задачи. Конечно, если бы мы собирались использовать

; несколько уровней привилегий, то нужно было бы инициализировать стеки

TSS\_0 db 68h dup(0)

; сегмент TSS\_1. В него будет выполняться переключение, так что надо

; инициализировать все, что может потребоваться:

TSS\_1 dd 0,0,0,0,0,0,0,0 ; связь, стеки, CR3

dd offset task\_1 ; EIP

; регистры общего назначения

dd 0,0,0,0,0,stack\_l2,0,0,0B8140h ; (ESP и EDI)

; сегментныерегистры

dd SEL\_flatDS,SEL\_32bitCS,SEL\_32bitSS,SEL\_flatDS,0,0

dd 0 ; LDTR

dd 0 ; адрес таблицы ввода-вывода

; точка входа в 32-битный защищенный режим

PM\_entry:

; подготовить регистры

xor еах,еах

mov ax,SEL\_flatDS

mov ds,ax

mov es,ax

mov ax,SEL\_32bitSS

mov ebx,stack\_l

mov ss,ax

mov esp,ebx

; загрузить TSS задачи 0 в регистр TR

mov ax,SEL\_TSS0

ltr ax

; только теперь наша программа выполнила все требования к переходу

; в защищенный режим

xor еах,еах

mov edi,0B8000h ; DS:EDI - адрес начала экрана

task\_0:

mov byte ptr ds:[edi],al ; вывести символ AL на экран

; дальний переход на TSS задачи 1

db 0EAh

dd 0

dw SEL\_TSS1

add edi,2 ; DS:EDI - адрес следующего символа

inc al ; AL - код следующего символа,

cmp al,80 ; если это 80,

jb task\_0 ; выйти из цикла

; дальний переход на процедуру выхода в реальный режим

db 0EAh

dd offset RM\_return

dw SEL\_16bitCS

; задача 1

task\_1:

mov byte ptr ds:[edi],al ; вывести символ на экран

inc al ; увеличить код символа

add edi,2 ; увеличить адрес символа

; переключиться на задачу 0

db 0EAh

dd 0

dw SEL\_TSS0

; сюда будет приходить управление, когда задача 0 начнет выполнять переход

; на задачу 1 во всех случаях, кроме первого

mov ecx,02000000h ; небольшая пауза, зависящая от скорости

loop $ ; процессора

jmp task\_1

PM\_seg ends

stack\_seg segment para stack "STACK"

stack\_start db 100h dup(?) ; стекзадачи 0

stack\_l = $ - stack\_start

stack\_task2 db 100h dup(?) ; стекзадачи 1

stack\_l2 = $ - stack\_start

stack\_seg ends

end start

Чтобы реализовать многозадачность в реальном времени в нашем примере, достаточно создать обработчик прерывания системного таймера IRQ0 в виде отдельной (третьей) задачи и поместить в IDT шлюз этой задачи. Текст обработчика для нашего примера мог быть крайне простым:

task\_3: ; это отдельная задача - не нужно сохранять регистры!

mov al,20h

out 20h,al

jmp task\_0

mov al,20h

out 20h,al

jmp task\_1

jmp task\_3

Но при вызове обработчика прерывания старая задача помечается как занятая в GDT и повторный JMP на нее приведет к ошибке. Вызов задачи обработчика прерывания, так же как и вызов задачи командой CALL, подразумевает, что она завершится командой IRET. Именно команду IRET оказывается проще всего вызвать для передачи управления из такого обработчика — достаточно только подменить селектор вызвавшей нас задачи в поле связи и выполнить IRET.

task\_3: ; при инициализации DS должен быть установлен на PM\_seg

mov al,20h

out 20h,al

mov word ptr TSS\_3,SEL\_TSS0

iret

mov al,20h

out 20h,al

mov word ptr TSS\_3,SEL\_TSS1

iret

jmp task\_3

Единственное дополнительное изменение, которое нужно внести, — инициализировать дескриптор TSS задачи task\_1 уже как занятый, так как управление на него будет передаваться командой IRET, что, впрочем, не составляет никаких проблем.

Помните, что во вложенных задачах команда IRET не означает конца программы — следующий вызов задачи всегда передает управление на следующую после IRET команду.

**52.22.2 Охарактеризувати програмн доступні елементи сопроцесора FPU**

* программную или регистровую модель любого процессора включаются только те регистры, которые доступны программисту на уровне машинных команд. Математические сопроцессоры, опираются на общую стековую организацию. Представление математических выражений в обратной польской записи естественно приводит к стековой организации процессора, который вычисляет эти выражения. Основу программной модели сопроцессора образует регистровый стек из восьми 80-битовых регистров данных R0 - R7.
* них хранятся числа, представленные в формате с плавающей точкой в расширенном вещественном формате. В любой момент времени трехбитовое поле TOP в слове состояния SW определяет регистр, являющийся вершиной стека и обозначаемый ST(0) или просто ST. В стандартных стековых операциях поле ТОР выполняет функции традиционного указателя стека SP. Многие численные команды имеют несколько режимов адресации, позволяющие программисту неявно оперировать вершиной стека или явно определять конкретные регистры относительно ТОР. Стек сопроцессора имеет круговую (кольцевую) организацию. Максимальное число включений в стек без промежуточных извлечений равно 8, а девятое включение "перезапишет" значение, включенное в стек первым. С каждым регистром стека ассоциируется двухбитовый тэг (признак), совокупность которых образует слово тэгов TW. Тэг регистра R0 находится в младших битах этого слова, а тэг регистра R7 - в старших битах. Тэг в общем виде показывает, какого рода информация содержится в соответствующем регистре и помогает сопроцессору в выполнении команд.

Тэг с кодом 011В показывает отсутствие данных; в этом случае регистр называется пустым и одно из действий сопроцессора при инициализации заключается в загрузке в биты тэгов всех регистров кода 011В. Попытка команды извлечь число из пустого регистра фиксируется как особые случаи нарушения стека и недействительной операции. Так как размещение тэгов соответствует физическим регистрам R0 - R7, для аcсоциирования тэгов с относительными регистрами ST(0) - ST(7) потребуется привлекать поле TOP из слова состояния. Остальными регистрами в программной модели сопроцессора являются 16-битные регистры управления и состояния, а также два 48-битных регистра указателей команды и операнда. Информация, содержащаяся в регистрах указателях команды (FIP) и операнда (FDP) предназначена только для процедур обработки особых случаев. Когда сопроцессор выполняет численную команду, он автоматически сохраняет в этих регистрах адреса численной команды и ее операнда (если он есть), а также младшие 11 бит кода операции. При сохранении в памяти ука затели команды и

данных представляются в одном из 4-х форматов в зависимости от режима работы процессора (R- или P-режим) и действия атрибута размера операнда (32- или 16-битный операнд).

**53.23.2 Охарактеризувати процес визначення фізичних адрес в захищеному режимі**

Прежде чем познакомиться с программированием в защищенном режиме, рассмотрим механизм адресации, применяющийся в нем. Так же как и в реальном режиме, адрес складывается из адреса начала сегмента и относительного смещения, но если в реальном режиме адрес начала сегмента просто лежал в соответствующем сегментном регистре, деленый на 16, то в защищенном режиме не все так просто. В сегментных регистрах находятся специальные 16-битные структуры, называемые селекторами и имеющие следующий вид:

биты 15 – 3: номер дескриптора в таблице

бит 2: индикатор таблицы 0/1 — использовать GDT/LDT

биты 1 – 0: уровень привилегий запроса (RPL)

Уровень привилегий запроса — это число от 0 до 3, указывающее уровень защиты сегмента, для доступа к которому используется данный селектор. Если программа имеет более высокий уровень привилегий, при использовании этого сегмента привилегии понизятся до RPL. Уровни привилегий и весь механизм защиты в защищенном режиме нам пока не потребуется.

GDT и LDT — таблицы глобальных и локальных дескрипторов соответственно. Это таблицы восьмибайтных структур, называемых дескрипторами сегментов, в которых и находится начальный адрес сегмента вместе с другой важной информацией:

слово 3 (старшее):

биты 15 – 8: биты 31 – 24 базы

бит 7: бит гранулярности (0 — лимит в байтах, 1 — лимит в 4-килобайтных единицах)

бит 6: бит разрядности (0/1 — 16-битный/32-битный сегмент)

бит 5: 0

бит 4: зарезервировано для операционной системы

биты 3 – 0: биты 19 – 16 лимита

слово 2:

бит 15: бит присутствия сегмента

биты 14 – 13: уровень привилегий дескриптора (DPL)

бит 12: тип дескриптора (0 — системный, 1 — обычный)

биты 11 – 8: тип сегмента

биты 7 – 0: биты 23 – 16 базы

слово 1: биты 15 – 0 базы

слово 0 (младшее): биты 15 – 0 лимита

Два основных поля в этой структуре, которые нам интересны, — это база и лимит сегмента. База представляет линейный 32-битный адрес начала сегмента, а лимит — это 20-битное число, которое равно размеру сегмента в байтах (от 1 байта до 1 мегабайта), если бит гранулярности сброшен в ноль, или в единицах по 4096 байт (от 4 Кб до 4 Гб), если он установлен в 1. Числа отсчитываются от нуля, так что лимит 0 соответствует сегменту длиной 1 байт, точно так же, как база 0 соответствует первому байту памяти.

Остальные элементы дескриптора выполняют следующие функции:

Бит разрядности: для сегмента кода этот бит указывает на разрядность операндов и адресов по умолчанию. То есть в сегменте с этим битом, установленным в 1, все команды будут интерпретироваться как 32-битные, а префиксы изменения разрядности адреса или операнда будут превращать их в 16-битные, и наоборот. Для сегментов данных этот бит управляет тем, какой регистр (SP или ESP) используют команды, работающие с этим сегментом данных как со стеком.

Поле DPL определяет уровень привилегий сегмента.

Бит присутствия указывает, что сегмент реально есть в памяти. Операционная система может выгрузить содержимое сегмента из памяти на диск и сбросить этот бит, а когда программа попытается к нему обратиться, произойдет исключение, обработчик которого снова загрузит содержимое этого сегмента в память.

Бит типа дескриптора — если он равен 1, сегмент является обычным сегментом кода или данных. Если этот бит — 0, дескриптор является одним из 16 возможных видов, определяемых полем типа сегмента.

Тип сегмента: для системных регистров в этом поле находится число от 0 до 15, определяющее тип сегментов (LDT, TSS, различные шлюзы), которые рассмотрены в главе 9. Для обычных сегментов кода и данных эти четыре бита выполняют следующие функции:

бит 11: 0 — сегмент данных, 1 — сегмент кода

бит 10: для данных — бит направления роста сегмента  
для кода — бит подчинения

бит 9: для данных — бит разрешения записи  
для кода — бит разрешения чтения

бит 8: бит обращения

Бит обращения устанавливается в 1 при загрузке селектора этого сегмента в регистр.

Бит разрешения чтения/записи выбирает разрешаемые операции с сегментом — для сегмента кода это могут быть выполнение или выполнение/чтение, а для сегмента данных — чтение или чтение/запись.

Бит подчинения указывает, что данный сегмент кода является подчиненным. Это значит, что программа с низким уровнем привилегий может передать управление в этот сегмент и текущий уровень привилегий не изменится.

Бит направления роста сегмента обращает смысл лимита сегмента. В сегментах с этим битом, сброшенным в ноль, допустимы все смещения от 0 до лимита, а если этот бит — 1, то допустимы все смещения, кроме смещений от 0 до лимита. Про такой сегмент говорят, что он растет сверху вниз, так как если лимит, например, равен –100, допустимы смещения от –100 до 0, а если лимит увеличить, станут допустимыми еще меньшие смещения.

Для обычных задач программирования нам не потребуется все многообразие возможностей адресации. Все, что нам нужно, — это удобный неограниченный доступ к памяти. Поэтому мы будем рассматривать простую модель памяти — так называемую модель flat, в которой базы всех регистров установлены в ноль, а лимиты — в 4 Гб. Именно в такой ситуации окажется, что можно забыть о сегментации и пользоваться только 32-битными смещениями.

Для создания flat-памяти нам потребуются два дескриптора с нулевой базой и максимальным лимитом — один для кода и один для данных.

Дескриптор кода:

лимит FFFFFh

база 000000000h

тип сегмента FAh

бит присутствия = 1

уровень привилегий = 3 (минимальный)

бит типа дескриптора = 1

тип сегмента: 1010b (сегмент кода, для выполнения/чтения)

бит гранулярности = 1

бит разрядности = 1

db 0FFh, 0FFh, 0h, 0h, 0h, 0FAh, 0CFh, 0h

Дескриптор данных:

лимит FFFFFh

база 00000000h

бит присутствия = 1

уровень привилегий = 3 (минимальный)

бит типа дескриптора = 1

тип сегмента: 0010b (сегмент данных, растет вверх, для чтения/записи)

бит гранулярности = 1

бит разрядности = 1

db 0FFh, 0FFh, 0h, 0h, 0h, 0F2h, 0CFh, 0h

Для того чтобы процессор знал, где искать дескрипторы, операционная система собирает их в таблицы, которые называются GDT (таблица глобальных дескрипторов — может быть только одна) и LDT (таблица локальных дескрипторов — по одной на каждую задачу), и загружает их при помощи привилегированных команд процессора. Так как мы пока не собираемся создавать операционные системы, нам потребуется только подготовить дескриптор и вызвать соответствующую функцию VCPI или DPMI.

Заметим также, что адрес, который получается при суммировании базы сегмента и смещения, называется линейным адресом и может не совпадать с физическим, если дело происходит в операционной системе, реализующей виртуальную память при помощи специально предусмотренного в процессорах Intel страничного механизма виртуализации памяти.

**54.24.2 Охарактеризувати програмн доступні елементи сопроцесора FPU**

* программную или регистровую модель любого процессора включаются только те регистры, которые доступны программисту на уровне машинных команд. Математические сопроцессоры, опираются на общую стековую организацию. Представление математических выражений в обратной польской записи естественно приводит к стековой организации процессора, который вычисляет эти выражения. Основу программной модели сопроцессора образует регистровый стек из восьми 80-битовых регистров данных R0 - R7.
* них хранятся числа, представленные в формате с плавающей точкой в расширенном вещественном формате. В любой момент времени трехбитовое поле TOP в слове состояния SW определяет регистр, являющийся вершиной стека и обозначаемый ST(0) или просто ST. В стандартных стековых операциях поле ТОР выполняет функции традиционного указателя стека SP. Многие численные команды имеют несколько режимов адресации, позволяющие программисту неявно оперировать вершиной стека или явно определять конкретные регистры относительно ТОР. Стек сопроцессора имеет круговую (кольцевую) организацию. Максимальное число включений в стек без промежуточных извлечений равно 8, а девятое включение "перезапишет" значение, включенное в стек первым. С каждым регистром стека ассоциируется двухбитовый тэг (признак), совокупность которых образует слово тэгов TW. Тэг регистра R0 находится в младших битах этого слова, а тэг регистра R7 - в старших битах. Тэг в общем виде показывает, какого рода информация содержится в соответствующем регистре и помогает сопроцессору в выполнении команд.

Тэг с кодом 011В показывает отсутствие данных; в этом случае регистр называется пустым и одно из действий сопроцессора при инициализации заключается в загрузке в биты тэгов всех регистров кода 011В. Попытка команды извлечь число из пустого регистра фиксируется как особые случаи нарушения стека и недействительной операции. Так как размещение тэгов соответствует физическим регистрам R0 - R7, для аcсоциирования тэгов с относительными регистрами ST(0) - ST(7) потребуется привлекать поле TOP из слова состояния. Остальными регистрами в программной модели сопроцессора являются 16-битные регистры управления и состояния, а также два 48-битных регистра указателей команды и операнда. Информация, содержащаяся в регистрах указателях команды (FIP) и операнда (FDP) предназначена только для процедур обработки особых случаев. Когда сопроцессор выполняет численную команду, он автоматически сохраняет в этих регистрах адреса численной команды и ее операнда (если он есть), а также младшие 11 бит кода операции. При сохранении в памяти ука затели команды и

данных представляются в одном из 4-х форматов в зависимости от режима работы процессора (R- или P-режим) и действия атрибута размера операнда (32- или 16-битный операнд).

**55.25.2 Дати характеристику системи переривань захищеного режимі**

Особливістю системи переривань в захищеному режимі є широке використання **внутрішніх апаратних переривань** - виключень. Виключення виникають як результат такого стану комп'ютерної системи, який не дозволяє виконати поточну команду. В свою чергу неможливість виконання чергової команди в багатьох випадках пов'язана з порушенням правил захисту.

За способом реакції процесора (і наступних можливостей програми обробки переривань), переривання в захищеному режимі поділяють на пастки, помилки, збої та аварії.

У випадку **пастки** в стек записується адреса команди. Тобто команда повністю виконується (якщо це команда між сегментної передачі управління, то змінюється вміст регістрів CS та EIP), а потім вміст CS та EIP записується в стек. Переривання від зовнішніх пристроїв та програмні переривання (команди INT) відносяться до пасток.

У випадку **помилки** в стек заноситься адреса команди, яка визвала виключення. Це дозволяє програмі обробки переривання внести необхідні зміни стану комп'ютерної системи і пере запустити виконання команди.

Але при виключеннях переривається виконання команди. Якщо виключення викликано командою, яка змінює вміст регістрів CS та EIP, то адреса повернення, яка записується в стек, може бути не визначеною. Крім того, виключення захисту можуть виникати при виконанні механізму переривань від зовнішніх пристроїв, коли команда-порушниця взагалі відсутня. Тому при помилках останнім в стек заноситься код помилки, який дозволяє ідентифікувати та локалізувати причину помилки в подібних ситуаціях.

У випадку **збою** повторний пере запуск команди-порушниці не можливий. Виключення типу збій переважно виникають при порушеннях під час переключення задач та реакції на привілеї, що супроводжується іншими порушеннями правил захисту.

У випадку **аварії** подальше виконання програми є неможливим і процесор переходить на процедуру початкового запуску (RESET, процесор пере запускається). Найбільш ймовірною причиною аварії є неможливість виконання стекових операцій (порушення правил захисту для стекових операцій) при реакції на інші виключення.

Для організації обслуговування переривань в захищеному режимі як і в реальному режимі використовується таблиця векторів переривань. На відміну від реального режиму вектором таблиці є не 4-х байтна логічна адреса процедури обробки переривань, а один із спеціальних 8-байтних **системних дескрипторів**, до яких відноситься шлюз переривання, шлюз пастки та шлюз задачі.

Шлюз переривання має наступну структуру

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Байт 7 | Байт 6 | Байт 5 | | | | | | | Байт 4 | | | |
| Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 31-16 | | **P** | **DPL** | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | резерв |
| Селектор сегмента кодів, де знаходиться процедура обробки переривання | | Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 15 - 0 | | | | | | | | | | |
| Байт 3 | Байт 2 | Байт 1 | | | | | | | Байт 0 | | | |

Шлюз переривання містить дані необхідні для визначення фізичної адреси процедури обробки переривань - селектор відповідного дескриптора сегмента кодів, та зміщення процедури обробки переривань в цьому сегменті.

Шлюз пастки має наступну структуру

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Байт 7 | Байт 6 | Байт 5 | | | | | | | Байт 4 | | | |
| Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 31-16 | | **P** | **DPL** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | резерв |
| Селектор сегмента кодів, де знаходиться процедура обробки переривання | | Зміщення (адреса в сегменті) процедури обробки переривань  Розряди 15 - 0 | | | | | | | | | | |
| Байт 3 | Байт 2 | Байт 1 | | | | | | | Байт 0 | | | |

Шлюз пастки відрізняється від шлюзу переривання лише значенням 1 в 0-му розряді 5-того байта дескриптора (молодші чотири біти 5-го байта **системного дескриптора** визначають його тип).

При використанні шлюзу переривання біт IF регістра ознак (EFLAGS) апаратурою встановлюється в 0 і попереднє значення відновлюється під час виконання команди IRET. При використанні шлюзу пастки біт IF не змінюється. В іншому шлюзи пастки та переривань не відрізняються.

Шлюз задачі використовується коли в результаті переривання необхідно переключить процесор на іншу задачу. В даній роботі переключення задач не розглядається.

На відміну від практики використання реального режиму в захищеному режимі таблиця векторів переривань може розміщуватись по довільній фізичній адресі, яка завантажується командою LIDT в старші 32 розряди 48-розрядного системного регістра IDTR. В молодші 16 розрядів цього регістру завантажується розмір таблиці в байтах. Команда LIDT належить до привілейованих команд і може виконуватись лише при **CPL**=0, або в реальному режимі.

При переключенні на процедуру обробки переривань значення поля **DPL** дескриптора сегмента кодів, де розташована ця процедура, не повинен перевищувати значення поточного рівня привілеїв (**DPL**сегменту**≤CPL**). Оскільки переривання та виключення можуть виникнути де завгодно, то процедури обробки переривань розміщують на 0-вому рівні привілеїв, тобто в системній області. При цьому шлюзи розглядаються як дані, тобто **DPL**шлюзу≥ **CPL.**

В лабораторній роботі зовнішні переривання при роботі в захищеному режимі забороняються, а обробляються лише внутрішні та програмні переривання з використанням шлюзу пастки.

В процесорах 80х86 та Pentium жорстко (на апаратному рівні) закріплено наступний розподіл векторів переривань між причинами виникнення виключень:

0-вектор. Ділення на 0. Помилка.

1-вектор. Використовується в налагоджувачах. Може бути як пасткою так і помилкою.

2-вектор. Зовнішнє переривання, яке ігнорує стан біту IF. Пастка.

3-вектор. Використовується в налагоджувачах. Пастка.

4-вектор. Контроль переповнення (виникає, коли виконується команда INTO і OF=1). Помилка.

5-вектор. Контроль переповнення масивів. Генерується командою BOUND, якщо операнд має значення за встановленою межею. Помилка

6-вектор. Неіснуючий код операції. В більшості випадків виникає в результаті помилок в передачах управління (передача управління не на перший байт команди). Коди 0d6h та 0f1h зарезервовані ф.Intel, хоча команди з такими кодами операцій поки що відсутні, але виключення при їх обробці не виникає. Помилка.

7-вектор. Співпроцесор відсутній. Виключення виникає при спробі виконати любу команду співпроцесора, а біт EM (біт 2) (емуляція співпроцесора) регістра CR0 має значення 1. Цим самим процедура обробки переривання 7 може бути використана для програмного моделювання роботи співпроцесора. Крім того, виключення може виникнуть після переключення задач, щоб установить, при необхідності, режим роботи співпроцесора для нової задачі. Помилка.

8-вектор. **Збій.** (Подвійна помилка). Виникає, коли в результаті виконання команди необхідно одночасно формувати декілька виключень.

Якщо спроба перейти на процедуру обробки 8-го переривання також пов'язана з помилкою, то найбільш ймовірно процесор виконає процедуру RESTART.

9-вектор. В процесорах 80х86 та Pentium виключення по цьому вектору не генеруються.

10-вектор. Неправильний сегмент стану задачі (TSS). В багатозадачних системах для кожної задачі формується сегмент TSS, який містить основні системні дані для задачі. Помилка.

11-вектор. Сегмент відсутній. Виключення виникає при завантаженні сегментного регістра селектором дескриптора, біт **P** атрибутів якого має значення 0. Помилка.

12-вектор. Помилка стека. Виникає при спробі звернення до стека командами MOV та їй подібними (використання регістра BP або префікса заміни сегмента SS:) при умові виходу зміщення в сегменті за межі допустимого діапазону. При виконанні команд PUSH та CALL та виходу замежі сегмента стека найбільш ймовірно виникнення аварійної ситуації з наступним виконанням процедури RESTART. Помилка.

13-вектор. Порушення правил захисту. Виникає при всіх порушеннях правил захисту, які не охоплені попередніми виключеннями.

14-вектор. Відсутня сторінка. Генерується при віртуальній сторінковій адресації, коли відповідна сторінка віртуальної пам'яті відсутня в оперативній пам'яті. Помилка

15 - вектор. В процесорах 80х86 та Pentium виключення по цьому вектору не генеруються

16 - вектор. Помилка співпроцесора. Співпроцесор має свій значний набір виключень. Якщо розряд NE (5-тий) регістра CR0 містить 1, то це виключення центрального процесора виникає при появі команди співпроцесора, наступної за тією командою, яка сформувала виключення співпроцесора.

**56.26.2 Охарактеризувати системні структури захищеного режиму, які використовуються при переключенні задач**

Параллельная работа ЦП и сопроцессора ставит перед программистом проблемы синхронизации. Например:

mem DW 0

:

FILD mem

INC mem

FSQRT

В данном примере, команда FILD преобразует 16-битное целое число в памяти в формат сопроцессора и сохраняет его в ST, затем целочисленная команда INC увеличивает это значение на 1, после чего сопроцессор вычисляет квадратный корень ST (FSQRT).

Для внешних сопроцессоров (8087, 80287, 80387) ЦП начнет выполнять команду INC раньше, чем сопроцессор закончит FILD, а, следовательно, в стек сопроцессора может попасть неверное значение. Встроенный блок FPU у IA-32 лишен этого недостатка, т. к. ЦП отслеживает обращения к памяти и не начнет выполнять целочисленную команду, если она обращается к ячейке памяти, с которой в данный момент работает блок FPU. Однако остается нерешенной другая проблема. Если при выполнении инструкции FPU возникает ситуация, соответствующая одному из незамаскированных особых случаев, FPU прекращает выполнение инструкции и сигнализирует об особом случае. Однако соответствующее исключение процессора будет вызвано только при очередном обращении процессора к блоку FPU, а до этого момента целочисленный блок может

изменить ячейку памяти, значение которой привело к исключению, так что обработчик исключения не сможет правильно интерпретировать причину особого случая.

Описанная ситуация говорит о необходимости использовать в таких случаях средства синхронизации. Например, команда FWAIT приостанавливает работу целочисленного устройства до тех пор, пока не будет завершена работа выполняемой сопроцессором инструкции или не будет вызвано исключение, если при выполнении последней инструкции FPU возник особый случай. Для этих же целей можно использовать любую другую инструкцию сопроцессора, кроме специальных "неожидающих" инструкций: FNINIT - сброс сопроцессора без ожидания, FNSTCW/FNSTSW - сохранение регистра управления/состояния без ожидания, FNCLEX - сбросить флаги особых случаев без ожидания, FNSAVE/FNSTENV - сохранение состояния (контекста) сопроцессора без ожидания.

Таким образом, для решения проблемы синхронизации приведенный выше пример следует переписать так:

FILD mem FSQRT

INC mem

**57.27.2 Охарактеризувати дескриптори та селектори захищеного режиму**

*Селектор* — это 16-битное число следующего формата:

биты 16 – 3: номер дескриптора в таблице (от 0 до 8191)

бит 2: 1 — использовать LDT, 0 — использовать GDT

биты 1 – 0: запрашиваемый уровень привилегий при обращении к сегменту и текущий уровень привилегий для селектора, загруженного в CS

Селектор, содержащий нулевые биты 16 – 3, называется нулевым и используется для загрузки в неиспользуемые сегментные регистры. Любое обращение в сегмент, адресуемый нулевым селектором, приводит к исключению #GP(0), в то время как даже загрузка в сегментный регистр ошибочного селектора вызывает исключение #GР(селектор). Попытка загрузки нулевого селектора в SS или CS также вызывает #GP(0), так как эти селекторы используются всегда.

*Дескриптор* — это 64-битная (восьмибайтная) структура данных, которая может встречаться в таблицах GDT и LDT. Дескриптор способен описывать сегмент кода, сегмент данных, сегмент состояния задачи, быть шлюзом вызова, ловушки, прерывания или задачи. В GDT также может находиться дескриптор LDT.

*Дескриптор сегмента данных или кода* (подробно рассмотрен в главе 6.1)

байт 7: биты 31 – 24 базы сегмента

байт 6:

бит 7: бит гранулярности (0 — лимит в байтах, 1 — лимит в 4-килобайтных единицах)

бит 6: бит разрядности (0 — 16-битный, 1 — 32-битный сегмент)

бит 5: 0

бит 4: зарезервировано для операционной системы

биты 3 – 0: биты 19 – 16 лимита

байт 5: (байт доступа)

бит 7: бит присутствия сегмента

биты 6 – 5: уровень привилегий дескриптора (DPL)

бит 4: 1 (тип дескриптора — не системный)

бит 3: тип сегмента (0 — данных, 1 — кода)

бит 2: бит подчиненности для кода, бит расширения вниз для данных

бит 1: бит разрешения чтения для кода, бит разрешения записи для данных

бит 0: бит доступа (1 — к сегменту было обращение)

байт 4: биты 23 – 16 базы сегмента

байты 3 – 2: биты 15 – 0 базы

байты 1 – 0: биты 15 – 0 лимита

Если в дескрипторе бит 4 байта доступа равен 0, дескриптор называется системным. В этом случае биты 0 – 3 байта доступа определяют один из шестнадцати возможных типов дескриптора (табл. 22).

***Таблица 22.*** *Типы системных дескрипторов*

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | Зарезервированный тип | 8 | Зарезервированный тип |
| 1 | Свободный 16-битный TSS | 9 | Свободный 32-битный TSS |
| 2 | Дескриптор таблицы LDT | A | Зарезервированный тип |
| 3 | Занятый 16-битный TSS | B | Занятый 16-битный TSS |
| 4 | 16-битный шлюз вызова | C | 32-битный шлюз вызова |
| 5 | Шлюз задачи | D | Зарезервированный тип |
| 6 | 16-битный шлюз прерывания | E | 32-битный шлюз прерывания |
| 7 | 16-битный шлюз ловушки | F | 32-битный шлюз ловушки |

*Дескрипторы шлюзов*

Дальние CALL или JMP на адрес с любым смещением и с селектором, указывающим на дескриптор шлюза вызова, приводят к передаче управления по адресу, указанному в дескрипторе. Обычно такие дескрипторы используются для передачи управления между сегментами с различными уровнями привилегий (см. главу 10.7).

CALL или JMP на адрес с селектором, указывающим на шлюз задачи, приводят к переключению задач (см. главу 10.8).

Шлюзы прерываний и ловушек используются для вызова обработчиков соответственно прерываний и исключений типа ловушки (см. главу 10.5).

байты 7 – 6: биты 31 – 16 смещения (0 для 16-битных шлюзов и шлюза задачи)

байт 5: (байт доступа)

бит 7: бит присутствия сегмента

биты 6 – 5: DPL — уровень привилегий дескриптора

бит 4: 0

биты 3 – 0: тип шлюза (3, 4, 5, 6, 7, В, С, Е, 7)

байт 4:

биты 7 – 5: 000

биты 4 – 0: 00000 или (для шлюза вызова) число двойных слов, которые будут скопированы из стека вызывающей задачи в стек вызываемой

байты 3 – 2: селектор сегмента

байты 1 – 0: биты 15 – 0 смещения (0 для шлюза задачи)

*Дескрипторы TSS и LDT*

Эти два типа дескрипторов применяются в многозадачном режиме, о котором рассказано далее. TSS — сегмент состояния задачи, используемый для хранения всей необходимой информации о каждой задаче в многозадачном режиме. LDT — таблица локальных дескрипторов, своя для каждой задачи.

Форматы этих дескрипторов совпадают с форматом дескриптора для сегмента кода или данных, за исключением того, что бит разрядности всегда равен нулю и, естественно, системный бит равен нулю, и биты 3 – 0 байта доступа содержат номер типа сегмента (1, 2, 3, 9, В). Команды JMP и CALL на адрес с селектором, соответствующим TSS незанятой задачи, приводят к переключению задач.

**58.28.2 Охарактеризувати методи обміну даними між зовнішніми пристроями та ОЗП**

Как компьятер одновременно выполнѐет несколько программ одновременно и обрабатывает сигналы от внешних устройств (клавиатуры, мышки и т.д.)? Ведь длѐ выполнениѐ команды она должна быть обработана микропроцессором, который в ПК только один. Как узнать о нажатиѐх клавиш и других сигналах, постоѐнно возникаящий во времѐ работы компьятера, если он выполнѐет некоторуя программу?

Варианты обработки всех поступаящих сигналов:

* Периодическаѐ остановка текущей программы и опрос устройств компьятера и запуск программ длѐ обслуживаниѐ этих устройств
* Создание очереди на обслуживание (выстраивание запросов на обслуживание устройств и просмотр микропроцессором этой очереди и выполнение запросов в ней)
* Подход основанный на прерываниѐх

Очевидно, что первый подход крайне не оптимизирован и замедлѐет работу системы, заставлѐѐ процессор выполнѐть большое количество действий не несущих полезной нагрузки. К тому же скорость будет падать с увеличением количества подклячённых устройств.

Второй подход хоть и более продуктивен, но также оставлѐет желать лучшего.

И наконец, третий подход, который используетсѐ в современных микропроцессорах (в том числе и процессорах фирмы Intel), основан на использовании *прерываний.* Прерывание – инициируемый определённый способом процесс, временно переклячаящий микропроцессор на выполнение прерванной программы. Такой метод позволѐет обеспечить наиболее эффективное управление не только внешними устройствами, но и программами. Нажимаѐ клавишу на клавиатуре, вы фактически инициируете посредством прерываниѐ немедленный вызов программы, котораѐ распознает нажатуя клавишу, заносит ее код в буфер клавиатуры, откуда он в дальнейшем считываетсѐ некоторой другой программой или операционной системой. На времѐ такой обработки микропроцессор прекращает выполнение некоторой программы и переклячаетсѐ на так называемуя на процедуру обработки прерываниѐ. После того как даннаѐ процедура выполнит необходимые действиѐ, прерваннаѐ программа продолжит выполнение с токи, где было приостановлено ее выполнение.

* + зависимости от источника возникновениѐ сигнала, прерываниѐ делѐтсѐ на:
* асинхронные, или внешние (аппаратные) — событиѐ, которые исходѐт от внешних источников (например, периферийных устройств) и могут произойти в лябой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителѐ, нажатие клавиш клавиатуры, движение мыши. Факт возникновениѐ в системе такого прерываниѐ трактуетсѐ как *запрос на*

*прерывание* (англ. *Interrupt request, IRQ*);

* синхронные, или внутренние — событиѐ в самом процессоре как результат нарушениѐ каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памѐти или недопустимый код операции;
* программные (частный случай внутреннего прерываниѐ) — инициируятсѐ исполнением специальной инструкции в коде программы. Программные прерываниѐ как правило используятсѐ длѐ обращениѐ к функциѐм встроенного программного обеспечениѐ

(firmware), драйверов и операционной системы.

Последовательность действий, которые происходѐт в ЭОМ при обработке прерываний от внешних устройств:

a) В результате функционированиѐ внешнего устройства или из системного таймера (через промежутки времени приблизительно равные 65.5 млсек) на контроллер прерываний поступает сигнал и он формирует сигнал «Запрос прерываниѐ» длѐ процессора

b) Процессор заканчивает выполнение очередной команды (в некоторых случаѐх он безусловно выполнѐет еще одну команду)

c) Процессор анализирует свойство разрешение прерываний (if), который находитсѐ в регистре свойств (flags). Если (if = 0) процессор игнорирует запрос и продолжает выполнѐть программу.

d) Если (if = 0) процессор подает на контроллер прерываний сигнал «Подтверждение запроса»

e) Получив сигнал «Подтверждение запроса», контролер прерываний передает процесору байт даннѐх N (N

– номер вектора прерываний), значение которого идентифицирует внешнее устройство

f) Процессор записывает в стек содержимое регистра признаков, после чего устанавливает if = 0, и записывает в стек содержимое регистров CS и IP. При записи в стек используетсѐ логический адрес SS: SP, то есть стек программы, прерываетсѐ. Если запрос на прерывание поѐвитсѐ при выполнении команды записи в сегментный регистр SS (например, во времѐ выполнениѐ команды MOV SS, AX), то в результате полнаѐ логическаѐ адрес стека может быть ошибочной, поскольку останетсѐ "старое" значение SP. Длѐ предотвращениѐ такой ошибки в случае поѐвлениѐ сигнала "Запрос прерываниѐ" во времѐ выполнениѐ лябой команды записи в сегментный регистр SS, сигнал "Подтверждение запроса" формируетсѐ после выполнениѐ еще одной команды, которой, конечно, должна быть команда загрузки регистра SP.

g) Процессор загружает регистр IP значением слова памѐти по адресу 0000: N \* 4, а регистр CS - значением слова памѐти по адресу 0000: N \* 4 + 2. Таким образом в реальном режиме первые 1000 байт ОЗУ предназначены длѐ таблицы, которуя называят таблицей векторов прерываний. Таблица содержит 256 строк по 4 байта на строку. Каждаѐ строка таблицы прерываний используетсѐ длѐ хранениѐ полной логической адреса (слово длѐ сегментной составлѐящей и слово длѐ смещениѐ в сегменте) процедуры обработки прерываний. Сами же процедуры обработки прерываний могут размещатьсѐ в лябой области ОЗУ. Поскольку байт N идентифицирует внешнее устройство, то в своя очередь строки (векторы) также могут идентифицировать внешние устройства

h) Процедура обработки прерываний должна заканчиватьсѐ командой IRET. Эта команда восстанавливает из стека содержимое регистра признаков, вклячаѐ предыдущее состоѐние признаки разрешениѐ на прерывание (if), который согласно п.d) должен равнѐтьсѐ 1, а также загружает из стека регистры IP и CS.

**59.29.2 Охарактеризувати виключення захищеного режиму**

Особливістю системи переривань в захищеному режимі є широке використання **внутрішніх апаратних переривань** - виключень. Виключення виникають як результат такого стану комп'ютерної системи, який не дозволяє виконати поточну команду. В свою чергу неможливість виконання чергової команди в багатьох випадках пов'язана з порушенням правил захисту.

За способом реакції процесора (і наступних можливостей програми обробки переривань), переривання в захищеному режимі поділяють на пастки, помилки, збої та аварії.

У випадку **пастки** в стек записується адреса команди. Тобто команда повністю виконується (якщо це команда між сегментної передачі управління, то змінюється вміст регістрів CS та EIP), а потім вміст CS та EIP записується в стек. Переривання від зовнішніх пристроїв та програмні переривання (команди INT) відносяться до пасток.

У випадку **помилки** в стек заноситься адреса команди, яка визвала виключення. Це дозволяє програмі обробки переривання внести необхідні зміни стану комп'ютерної системи і пере запустити виконання команди.

Але при виключеннях переривається виконання команди. Якщо виключення викликано командою, яка змінює вміст регістрів CS та EIP, то адреса повернення, яка записується в стек, може бути не визначеною. Крім того, виключення захисту можуть виникати при виконанні механізму переривань від зовнішніх пристроїв, коли команда-порушниця взагалі відсутня. Тому при помилках останнім в стек заноситься код помилки, який дозволяє ідентифікувати та локалізувати причину помилки в подібних ситуаціях.

У випадку **збою** повторний пере запуск команди-порушниці не можливий. Виключення типу збій переважно виникають при порушеннях під час переключення задач та реакції на привілеї, що супроводжується іншими порушеннями правил захисту.

У випадку **аварії** подальше виконання програми є неможливим і процесор переходить на процедуру початкового запуску (RESET, процесор пере запускається). Найбільш ймовірною причиною аварії є неможливість виконання стекових операцій (порушення правил захисту для стекових операцій) при реакції на інші виключення.

**60.30.2 Охарактеризувати групу системних команд сопроцесор**

Система команд сопроцессора

Возможны три формата команд сопроцессора, аналогичные форматам команд центральных процессоров фирмы Intel. Это команды с обращением к оперативной памяти, команды с обращением к одному из численных регистров и команды без операндов, заданных явным образом.

Команды с обращением к памяти могут занимать от двух до четырех байт, в зависимости от способа адресации.

Все ассемблерные мнемоники команд сопроцесора начинаются с буквы F, поэтому их легко отличить от команд центрального процессора.

Команды сопроцессора можно разделить на несколько групп:

команды пересылки данных;

арифметические команды;

команды сравнений чисел;

трансцендентные команды;

управляющие команды

Команды пересылки данных предназначены для загрузки чисел из оперативной памяти в численные регистры, записи данных из численных регистров в оперативную память, копирования данных из одного численного регистра в другой.

Арифметические команды выполняют такие операции, как сложение, вычитание, умножение, деление, извлечение квадратного корня, нахождение частичного остатка, округление и т.п.

Команды сравнения сравнивают вещественные и целые числа, выполняют анализ чисел.

Трансцендентные команды предназначены для вычисления различных тригонометрических, логорифмических, показательных и гиперболических функций - sin, cos, tg и тому подобных.

Последняя группа команд - управляющие команды. Они обеспечивают установку режима работы арифметического сопроцессора, его сброс и инициализацию, перевод сопроцессора в защищенный режим работы и так далее.