Асемблер

1. Схема трансляції, компанування і виконання програми.

Схема трансляції, компанування і виконання програми включає кілька основних етапів:

1. **Трансляція (компіляція)**:
   - **Синтаксичний аналіз**: Аналіз вихідного коду для виявлення синтаксичних помилок.
   - **Семантичний аналіз**: Перевірка смислової правильності коду.
   - **Оптимізація**: Поліпшення коду для ефективності.
   - **Генерація коду**: Перетворення вихідного коду у проміжний код або безпосередньо у машинний код.
2. **Компанування (лінкування)**:
   - **Збір об'єктних файлів**: Об'єктні файли (.obj або .o), створені на етапі трансляції, збираються разом.
   - **Резолюція символів**: Зв'язування зовнішніх символів та функцій.
   - **Створення виконуваного файлу**: Об'єднання всіх об'єктних файлів у один виконуваний файл (.exe або аналогічний).
3. **Виконання**:
   - **Завантаження**: Завантаження виконуваного файлу у пам'ять.
   - **Ініціалізація**: Підготовка середовища виконання (налаштування стеку, ініціалізація змінних).
   - **Виконання коду**: Відпрацювання інструкцій виконуваного файлу процесором.
   - **Завершення**: Повернення управління операційній системі після завершення програми.

2. Основні регістри мікропроцесора

- **AX, BX, CX, DX**: Використовуються для загальних цілей, таких як зберігання тимчасових даних, лічильників, індексів тощо.
- **EAX, EBX, ECX, EDX**: Розширені версії для 32-бітних мікропроцесорів.
- **RAX, RBX, RCX, RDX**: Розширені версії для 64-бітних мікропроцесорів.

3.Позиційна незалежність програми в однопрограмних ОС

Позиційна незалежність програми (Position-Independent Code, PIC) в однопрограмних операційних системах означає, що програмний код може виконуватися незалежно від того, де в пам'яті він розташований. Це досягається шляхом написання та компіляції коду так, щоб він не залежав від конкретних адрес пам'яті.

4.Режими адресування операндів в однопрограмних ОС.

1. **Пряме адресування**: Операнд міститься безпосередньо в інструкції.
   - Приклад: `MOV AX, 1234h` (завантажити значення 1234h в AX)
2. **Непряме адресування**: Адреса операнда вказана в регістрі або пам'яті.
   - Приклад: `MOV AX, [BX]` (завантажити значення з адреси, що міститься в регістрі BX)
3. **Реєстрове адресування**: Операнд знаходиться в регістрі.
   - Приклад: `MOV AX, BX` (завантажити значення з регістру BX в AX)
4. **Базове адресування**: Адреса обчислюється як базова адреса плюс зміщення.
   - Приклад: `MOV AX, [BP + 4]` (завантажити значення з адреси, що обчислюється як BP + 4)
5. **Індексне адресування**: Адреса обчислюється як індекс плюс зміщення.
   - Приклад: `MOV AX, [SI + 4]` (завантажити значення з адреси, що обчислюється як SI + 4)
6. **Базово-індексне адресування**: Адреса обчислюється як базовий регістр плюс індексний регістр плюс зміщення.
   - Приклад: `MOV AX, [BX + SI + 4]` (завантажити значення з адреси, що обчислюється як BX + SI + 4)
7. **Пряме адресування з сегментом**: Адреса обчислюється з використанням сегментного регістру.
   - Приклад: `MOV AX, [ES:BX]` (завантажити значення з адреси, що обчислюється як ES:BX)

5. Моделі адресування операндів в командах мікропроцесора.

Моделі адресування визначають способи визначення адреси операндів в інструкціях. Основні моделі включають:

1. **Імпліцитне адресування**: Операнд явно не вказується, він імпліцитно визначається інструкцією.
    - Приклад: `STC` (Set Carry Flag, прапорець Carry змінюється без явного зазначення операнда)
2. **Імедіатне адресування**: Операнд безпосередньо вказаний у самій інструкції.
    - Приклад: `MOV AX, 1234h` (значення 1234h є операндом)
3. **Реєстрове адресування**: Операнд знаходиться в регістрі.
    - Приклад: `MOV AX, BX`
4. **Пряме адресування**: Інструкція містить адресу операнда.
    - Приклад: `MOV AX, [1234h]`
5. **Непряме адресування**: Адреса операнда знаходиться в регістрі або іншій пам'яті.
    - Приклад: `MOV AX, [BX]`
6. **Індексне адресування**: Адреса обчислюється з використанням індексного регістру.
    - Приклад: `MOV AX, [SI + 4]`
7. **Базово-індексне адресування**: Адреса обчислюється з використанням базового та індексного регістрів.
    - Приклад: `MOV AX, [BX + SI + 4]`

6.Моделі структури програм: головна програма, підпрограма, COM-програма (команди + дані).

1. **Головна програма**:
    - Основний блок коду, який виконується першим при запуску програми.
    - Включає виклики підпрограм і управління потоком виконання.
2. **Підпрограма**:
    - Окремі блоки коду, які виконують конкретні задачі і можуть викликатися з різних частин головної програми або інших підпрограм.
    - Зазвичай мають точки входу і виходу, можуть приймати параметри і повертати значення.

3. **COM-програма**:
    - Проста модель для DOS, де код і дані знаходяться в одному сегменті.
    - Програма обмежена 64 КБ пам'яті, що включає як код, так і дані.
    - Відсутність сегментації спрощує адресування, але обмежує розмір програми.

7.Загальний формат бітової структури команди процесора Intel. Поля коду команди

Загальний формат команди процесора Intel складається з кількох полів:

1. **Opcode** (код операції): Визначає, яку операцію виконує інструкція.
2. **ModRM**: Використовується для визначення режиму адресування і регістрів.
3. **SIB** (Scale-Index-Base): Використовується для складних режимів адресування (наприклад, базово-індексне з масштабом).
4. **Displacement** (зміщення): Використовується для адресування з прямим або непрямим зміщенням.
5. **Immediate**: Містить безпосереднє значення операнда.

Поля коду команди (Opcode) можуть бути одно- або двобайтовими і визначають конкретну інструкцію процесора. ModRM і SIB поля використовуються для детальнішого адресування і специфікації регістрів.

8.Будова байта ModRM способу адресування для 16- і 32-бітових режимів.

Байт **ModRM** використовується для визначення режиму адресування і регістрів. Його структура:

- **Mod**: Два біти, які визначають режим адресування.
- **Reg/Opcode**: Три біти, які визначають регістр або частину коду операції.
- **RM**: Три біти, які визначають другий регістр або спосіб адресування.

Для 16-бітового режиму:

- **Mod**: 00 - непряме адресування через регістр; 01 - з 8-бітним зміщенням; 10 - з 16-бітним зміщенням; 11 - реєстрове адресування.

- **Reg/Opcode**: Вказує на один з 8 регістрів (AX, CX, DX, BX, SP, BP, SI, DI).
- **RM**: Визначає регістр або спосіб адресування (наприклад, [BX + SI], [BX + DI]).

Для 32-бітового режиму:

- **Mod**: 00 - непряме адресування через регістр; 01 - з 8-бітним зміщенням; 10 - з 32-бітним зміщенням; 11 - реєстрове адресування.
- **Reg/Opcode**: Вказує на один з 8 регістрів (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).
- **RM**: Визначає регістр або спосіб адресування, включаючи можливість використання SIB-байта для складних режимів адресування.

9. Схеми виконання команд мікропроцесора: команди без операндів (нуль-операндні), команди операцій і команди дії з одним операндом.

Команди без операндів (нуль-операндні)

Ці команди не мають явних операндів і зазвичай використовують стек або внутрішні регістри процесора.

Приклад:

- `NOP` (No Operation): Не виконує ніякої дії, використовується для затримки або вирівнювання коду.
- `RET` (Return): Повернення з підпрограми, використовуючи адресу, збережену на вершині стеку.

Схема виконання:

1. Зчитування команди з пам'яті.
2. Виконання команди без потреби у зверненні до операндів у пам'яті чи регістрах.
3. Перехід до наступної інструкції.

Команди операцій і дії з одним операндом

Ці команди використовують один операнд, який може бути вказаний безпосередньо або непрямо.

Приклад:

- `INC AX`: Збільшення значення в регістрі AX на 1.
- `NEG [BX]`: Інвертування значення за адресою, вказаною в регістрі BX.

Схема виконання:

1. Зчитування команди з пам'яті.
2. Визначення адреси операнда (якщо непряме адресування).
3. Виконання операції над операндом.
4. Збереження результату (якщо потрібно) і перехід до наступної інструкції.

10.Схеми виконання команд мікропроцесора: команди з двома операндами, команди з трьома операндами.

Команди з двома операндами

Ці команди використовують два операнди: джерело і призначення.

Приклад:

- `MOV AX, BX`: Копіювання значення з регістру BX до регістру AX.
- `ADD AX, 5`: Додавання значення 5 до регістру AX.

Схема виконання:

1. Зчитування команди з пам'яті.
2. Визначення адреси або значення обох операндів.
3. Виконання операції з використанням обох операндів.
4. Збереження результату в призначеному місці.
5. Перехід до наступної інструкції.

Команди з трьома операндами

Ці команди використовуються в деяких архітектурах і дозволяють мати три окремі операнди: два джерела і одне призначення.

Приклад:

- `ADD R1, R2, R3`: Додавання значень з R2 і R3, результат зберігається в R1.

Схема виконання:

1. Зчитування команди з пам'яті.
2. Визначення адреси або значення трьох операндів.
3. Виконання операції з використанням двох джерел.
4. Збереження результату в місці призначення.
5. Перехід до наступної інструкції.

11.Директиви асемблера та їх застосування.

Директиви асемблера використовуються для керування процесом трансляції, визначення даних та організації коду.

Основні директиви:

- **`.data`**: Визначає сегмент даних, де розміщуються змінні.
  - Приклад: `.data var1 DW 10`
- **`.code`**: Визначає сегмент коду, де розміщуються інструкції.
  - Приклад: `.code start: MOV AX, var1`
- **`.stack`**: Визначає сегмент стеку.
  - Приклад: `.stack 100h`
- **`.macro`**: Визначення макросів.
  - Приклад: `.macro INC_VAR var ADD var, 1 .endm`
- **`.include`**: Включення файлів.
  - Приклад: `.include "filename.asm"`

12.Трансляція програм ASM за першим і другим переглядом.

Перший перегляд

Під час першого перегляду асемблер:

1. Аналізує кожну інструкцію і директиву.
2. Визначає мітки і адреси.
3. Створює таблицю символів, яка містить адреси міток та змінних.

Другий перегляд

Під час другого перегляду асемблер:

1. Виконує дійсну трансляцію інструкцій в машинний код, використовуючи таблицю символів.
2. Замінює мітки і змінні на відповідні адреси або значення.
3. Генерує вихідний машинний код і об'єктний файл.

13.Алгоритм першого перегляду асемблера при трансляції.

1. Ініціалізація таблиці символів.
2. Читання кожного рядка вихідного коду.
3. Якщо рядок містить мітку, додати мітку і поточну адресу в таблицю символів.
4. Оцінка розміру інструкції чи директиви.
5. Оновлення поточної адреси.
6. Перехід до наступного рядка коду.
7. Після обробки всього коду, завершення першого перегляду.

14.Алгоритм другого перегляду асемблера при трансляції.

1. Ініціалізація буфера для зберігання машинного коду.
2. Читання кожного рядка вихідного коду.
3. Замінювання міток і символів на відповідні адреси з таблиці символів.
4. Трансляція інструкцій в машинний код.
5. Збереження машинного коду у вихідний файл або буфер.
6. Перехід до наступного рядка коду.
7. Після обробки всього коду, завершення другого перегляду.

15.Загальні принципи компонування програм. Об’єктні файли.

Компонування (лінкування) - процес об'єднання одного або більше об'єктних файлів для створення виконуваного файлу.

Принципи компонування:

1. **Збір об'єктних файлів**: Компонувальник збирає всі об'єктні файли, які мають бути об'єднані.

2. **Резолюція символів**: Визначення адрес зовнішніх символів та функцій, які використовуються в об'єктних файлах.
3. **Створення таблиць переривань та інших системних структур**.
4. **Формування виконуваного файлу**: Об'єднання всіх об'єктних файлів в один виконуваний файл.

Об’єктні файли:

- Містять машинний код, дані та інформацію для лінкування.
- Формати можуть включати ELF, COFF, PE тощо.
- Містять символи, мітки і зовнішні посилання.

16.Компонувальники і принципи їх роботи.

Компонувальники (лінкери) - це програми, які об'єднують об'єктні файли для створення виконуваного файлу.

Принципи роботи:

1. **Завантаження об'єктних файлів**: Лінкер завантажує всі об'єктні файли та бібліотеки.
2. **Аналіз таблиць символів**: Перевірка символів на відповідність, визначення адрес функцій та змінних.
3. **Резолюція зовнішніх символів**: Визначення адрес для зовнішніх символів, використовуваних в об'єктних файлах.
4. **Перестановка коду та даних**: Розміщення коду і даних у відповідні сегменти виконуваного файлу.
5. **Створення виконуваного файлу**: Об'єднання всіх компонентів і запис у виконуваний файл

Бібліотеки DLL

1. Переваги і недоліки статичного компонування програм.

Переваги:

- **Швидкість виконання**: Оскільки всі необхідні бібліотеки вже включені у виконуваний файл, не потрібно завантажувати додаткові модулі під час виконання.

- **Простота розгортання**: Всі залежності включені у виконуваний файл, що спрощує розгортання програми на кінцевих системах.
- **Безпека**: Меньша ймовірність зловмисного втручання, оскільки немає зовнішніх бібліотек, які можуть бути замінені або змінені.

Недоліки:

- **Розмір файлу**: Виконувані файли стають більшими, оскільки вони включають всі необхідні бібліотеки.
- **Оновлення**: Щоб оновити бібліотеку, потрібно перекомпілювати та перевипустити весь виконуваний файл.
- **Пам'ять**: Кожна копія програми, що працює, має власну копію бібліотек в пам'яті, що неефективно використовує ресурси.

2.Визначення динамічної бібліотеки. Схема динамічного завантаження в адресний простір процесу

Динамічна бібліотека (Dynamic Link Library, DLL) - це бібліотека, яка завантажується та зв'язується з програмою під час її виконання.

Схема динамічного завантаження:

1. **Завантаження програми**: При запуску програми завантажувач операційної системи читає заголовок виконуваного файлу і знаходить список динамічних бібліотек.
2. **Завантаження DLL**: Завантажувач завантажує необхідні DLL у віртуальний адресний простір процесу.
3. **Зв'язування функцій**: Завантажувач знаходить адреси функцій в DLL і заповнює таблицю імпорту в процесі.
4. **Виконання програми**: Після завантаження та зв'язування всіх необхідних DLL, програма починає виконуватись.

3.Переваги і недоліки використання динамічних бібліотек.

Переваги:

- **Зменшення розміру виконуваних файлів**: Код, який використовується багатьма програмами, розміщується в DLL, що зменшує розмір виконуваних файлів.

- **Оновлення**: Легше оновлювати програму, оскільки бібліотеки можуть бути замінені без перекомпіляції всього коду.
- **Спільне використання коду**: Кілька програм можуть використовувати одну і ту ж бібліотеку, що зменшує споживання пам'яті.

Недоліки:

- **Залежності**: Програми можуть не працювати, якщо необхідні DLL відсутні або не сумісні.
- **Продуктивність**: Завантаження DLL під час виконання може займати більше часу в порівнянні зі статичним компонуванням.
- **Безпека**: Зловмисник може замінити DLL на шкідливу версію.

4.Зворотня сумісність динамічних бібліотек.

Зворотня сумісність означає, що нові версії динамічних бібліотек повинні працювати з існуючими програмами без їх змін.

Забезпечення зворотної сумісності:

- **Збереження інтерфейсу**: Нові версії DLL повинні зберігати існуючі функції та їх сигнатури.
- **Нова функціональність**: Нові функції додаються, але старі не видаляються або не змінюються.
- **Версії**: Використання номерів версій для DLL, що дозволяє програмам визначати сумісні версії.

5.Загальні принципи зв’язування з DLL в алгоритмічних мовах для неявного і явного зв’язування.

Неявне зв’язування:

- **Директиви під час компіляції**: Вказуються під час компіляції або лінкування (наприклад, у Visual Studio).
- **Таблиці імпорту**: Створюється таблиця імпорту, яка вказує функції, що імпортуються з DLL.
- **Завантаження під час старту**: DLL завантажується автоматично при запуску програми.

Явне зв’язування:

- **Функції завантаження**: Використовуються функції завантаження DLL під час виконання (наприклад, `LoadLibrary` в Windows).
- **Отримання адрес функцій**: Використовуються функції для отримання адрес функцій (наприклад, `GetProcAddress`).
- **Динамічне зв'язування**: Програма сама контролює завантаження і зв'язування DLL під час виконання.

6. Взаємодія динамічної бібліотеки з адресним простором процесу. Особливості об'єктного коду динамічних бібліотек.

Взаємодія з адресним простором:

- **Завантаження в пам'ять**: DLL завантажуються в адресний простір процесу і можуть бути спільно використані кількома процесами.
- **Віртуальні адреси**: Адреси функцій і даних в DLL є віртуальними і транслюються у фізичні адреси під час виконання.

Особливості об'єктного коду DLL:

- **Переносимість**: Код в DLL повинен бути переносимим, оскільки DLL може бути завантажена в будь-яку область пам'яті.
- **Релокації**: Адреси в коді можуть потребувати релокації під час завантаження DLL, щоб відповідати адресному простору процесу.

7. Точка входу динамічної бібліотеки і приклади її раціонального використання.

Точка входу (entry point) в DLL - це функція, яка виконується при завантаженні, розвантаженні або інших подіях, що стосуються DLL. У Windows ця функція зазвичай називається `DllMain`.

Приклад `DllMain`:

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved) {
    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            // Ініціалізація DLL при завантаженні в процес
            break;
        case DLL_PROCESS_DETACH:
            // Очищення ресурсів при розвантаженні DLL з процесу
            break;
        case DLL_THREAD_ATTACH:
            // Дія при створенні нового потоку
            break;
        case DLL_THREAD_DETACH:
            // Дія при завершенні потоку
            break;
    }
    return TRUE;
}
```

Раціональне використання:

- **Ініціалізація ресурсів**: Ініціалізація ресурсів, таких як файли, мережеві з'єднання, які потрібні DLL.
- **Очищення ресурсів**: Звільнення ресурсів при розвантаженні DLL.
- **Обробка подій**: Обробка специфічних подій, таких як створення або завершення потоків.

8. Структура виконуваних файлів для Windows. Формат PE.

PE (Portable Executable) - це формат виконуваних файлів, що використовується в Windows для виконуваних програм, DLL, драйверів та інших видів файлів.

Основні компоненти PE-файлу:

- **Заголовок DOS**: Історичний заголовок для сумісності з DOS.
- **Заголовок PE**: Містить інформацію про тип і структуру файлу.
- **Таблиця секцій**: Містить інформацію про різні секції файлу (код, дані, ресурси тощо).
- **Секції**: Містять код, дані, таблиці імпорту/експорту, ресурси тощо.

9. Схема процесу компонування для Windows у разі неявного зв'язування.

1. **Компіляція**: Вихідний код компілюється в об'єктні файли.
2. **Лінкування**: Лінкер використовує таблицю імпорту для визначення DLL і функцій, які повинні бути використані.
3. **Створення виконуваного файлу**: Лінкер створює виконуваний файл, включаючи таблицю імпорту, яка вказує функції, що імпортуються з DLL.
4. **Завантаження**: Під час запуску програми операційна система завантажує всі необхідні DLL та заповнює таблицю імпорту.

10. Схема процесу компонування для Windows у разі явного зв'язування.

1. **Компіляція**: Вихідний код компілюється в об'єктні файли.
2. **Лінкування**: Створюється виконуваний файл без вказівки на конкретні DLL.
3. **Завантаження**: Під час виконання програми, вона самостійно завантажує необхідні DLL за допомогою `LoadLibrary`.
4. **Зв'язування**: Програма використовує `GetProcAddress` для отримання адрес функцій з завантажених DLL.
5. **Виконання**: Програма виконує функції з завантажених DLL.

11. Механізми передавання параметрів до процедур і функцій: за значенням; за посиланням (за адресою); за поверненим значенням; за результатом; за іменем; відкладеним обчисленням. Загальні визначення.

1. **За значенням**:
    - **Принцип**: Передається копія аргументу. Зміни до параметру всередині функції не впливають на оригінальний аргумент.
    - **Приклад**:
      cpp

```
void func(int value) {

    value = 5;
```

```
}
```

-

2. **За посиланням (за адресою)**:
   - **Принцип**: Передається адреса аргументу, дозволяючи функції змінювати оригінальний аргумент.
   - **Приклад**:
     cpp

```
void func(int* value) {
    *value = 5;
}
```

-

3. **За поверненим значенням**:
   - **Принцип**: Функція повертає значення як результат своєї роботи.
   - **Приклад**:
     cpp

```
int func() {
    return 5;
}
```

-

4. **За результатом**:
   - **Принцип**: Використовується для вихідних параметрів, результат передається назад через аргументи.
   - **Приклад**:
     cpp

```
void func(int& result) {
    result = 5;
}
```

    -
5. **За іменем**:
    - **Принцип**: Аргументи передаються у вигляді виразів, які обчислюються кожного разу, коли використовуються.
    - **Приклад**: Поширено в мовах, як-от Algol.
6. **Відкладене обчислення**:
    - **Принцип**: Вирази аргументів обчислюються тільки тоді, коли вони реально використовуються всередині функції.
    - **Приклад**: Легко реалізується в функціональних мовах програмування.

12. Способи передавання параметрів до процедур і функцій: в регістрах; в глобальних зміних; в стеку. Погодження(конвенції) для передавання параметрів у функцію через стек.

В регістрах:

- **Принцип**: Параметри передаються через регістри процесора. Це швидше, але обмежена кількість параметрів.
- **Приклад**:
  assembly

```
mov eax, param1

mov ebx, param2
```

-

В глобальних змінних:

- **Принцип**: Параметри зберігаються у глобальних змінних, доступних всім функціям.
- **Недолік**: Потенційно небезпечне, оскільки може призвести до конфліктів.

В стеку:

- **Принцип**: Параметри передаються через стек, що дозволяє передавати багато параметрів незалежно від їх кількості.
- **Приклад**:
  assembly

```
push param1

push param2
```

  -

Погодження (конвенції) для передавання параметрів через стек:

- **cdecl**: Параметри передаються справа наліво, викликаюча функція очищує стек.
- **stdcall**: Параметри передаються справа наліво, функція, що викликається, очищує стек.

13. Модель конвенції stdcall передавання параметрів до процедур і функцій.

stdcall:

- **Порядок параметрів**: Справа наліво.
- **Очищення стека**: Функція, що викликається, очищує стек після завершення.
- **Приклад**:
  assembly

```
push param2

push param1

call function
```

14. Стандартні типи даних Windows та їх еквіваленти в мові C.

- BOOL: `int`
- CHAR: `char`
- DWORD: `unsigned long`
- HANDLE: `void*`
- HINSTANCE: `void*`
- HWND: `void*`
- LPSTR: `char*`
- LPWSTR: `wchar_t*`
- UINT: `unsigned int`
- VOID: `void`

15. Процедура підготовки функцій для DLL на прикладі алгоритмічної мови (C++ чи іншої).

Приклад на C++:

cpp Copy code

```cpp
// mydll.h
#ifdef MYDLL_EXPORTS
#define MYDLL_API __declspec(dllexport)
#else
#define MYDLL_API __declspec(dllimport)
#endif

extern "C" MYDLL_API void myFunction();

// mydll.cpp
#include "mydll.h"

extern "C" MYDLL_API void myFunction() {
    // Реалізація функції
}
```

16. Компіляція функцій в бібліотеку DLL (створення DLL). Налаштування компілятора і завантажувача.

Кроки:

1. **Створення проекту**: Виберіть тип проекту DLL у середовищі розробки (наприклад, Visual Studio).
2. **Додавання файлів**: Додайте вихідні файли та заголовки з директивами експорту.
3. **Конфігурація**: Встановіть опції компіляції для створення DLL.
4. **Комппіляція**: Скомпілюйте проект, що створить DLL-файл.

17. Використання DLL в прикладних програмах методом явного зв’язування (на прикладі мови С чи іншої).

Приклад на C:

cpp Copy code

```cpp
#include <windows.h>
#include <iostream>

typedef void (*MYFUNCTION)();

int main() {
    HINSTANCE hDLL = LoadLibrary("mydll.dll");
    if (hDLL) {
        MYFUNCTION func = (MYFUNCTION)GetProcAddress(hDLL, "myFunction");
        if (func) {
            func();
        }
        FreeLibrary(hDLL);
    }
    return 0;
}
```

18. Використання DLL в прикладних програмах методом неявного зв’язування (на прикладі мови C чи іншої).

Приклад на C++:

cpp Copy code

```cpp
// main.cpp
#include "mydll.h"

int main() {
    myFunction();
    return 0;
}

// mydll.h
#ifdef MYDLL_EXPORTS
#define MYDLL_API __declspec(dllexport)
#else
#define MYDLL_API __declspec(dllimport)
#endif

extern "C" MYDLL_API void myFunction();
```

Конфігурація проекту:

- **Імпортування**: Вкажіть проект, який використовує DLL, щоб імпортувати функції з DLL.
- **Лінкування**: Вкажіть файл .lib, створений під час компіляції DLL, для лінкування з проектом.

Графічні редактори

1. Піксел. Роздільна здатність екрана. Палітра кольорів. Принцип малювання на екрані.

Піксел: Найменша одиниця зображення на екрані комп'ютера, що має свій колір та положення.

Роздільна здатність екрана: Кількість пікселів, які можуть бути відображені на екрані горизонтально та вертикально. Вимірюється у пікселях на дюйм (dpi).

Палітра кольорів: Сукупність всіх доступних кольорів, які можуть бути використані для відображення зображень на екрані. Вона може бути обмеженою (наприклад, 256 кольорів) або безліччю кольорів (true color).

Принцип малювання на екрані: Малювання на екрані здійснюється шляхом зміни значень пікселів у відповідних координатах. Програма, що малює, може звертатися до графічного API (Application Programming Interface), щоб реалізувати зміни кольору та значення пікселів.

2. Графічні примітиви. Графічні бібліотеки. Принципи використання.

Графічні примітиви: Основні форми та об'єкти, які можна малювати на екрані, такі як лінії, кола, прямокутники тощо.

Графічні бібліотеки: Набір програмних інструментів та функцій, які дозволяють розробникам малювати графічні примітиви на екрані. Приклади включають OpenGL, DirectX, Cairo тощо.

Принципи використання: Розробники використовують графічні бібліотеки для створення зображень та малювання графічних примітивів

на екрані. Вони викликають функції бібліотеки, щоб ініціалізувати зображення та виконати малювання на екрані.

3. Графічний інтерфейс редактора з користувачем. Клавіатурний інтерфейс редактора з користувачем.

Графічний інтерфейс (GUI): Інтерфейс, що використовує графічні елементи, такі як кнопки, меню, поля вводу тощо, для взаємодії з користувачем. Наприклад, вікна, кнопки, меню.

Клавіатурний інтерфейс: Спосіб взаємодії з програмою за допомогою клавіатури. Це може включати введення тексту, виклик команд за допомогою гарячих клавіш, переміщення курсора тощо.

4. Прокручування поля, багатовіконність редактора, коректне припинення роботи.

Прокручування поля: Функціональність, яка дозволяє користувачам переміщатися по області зображення, яка перевищує розміри екрану.

Багатовіконність редактора: Можливість відкривати одночасно декілька вікон або документів для редагування.

Коректне припинення роботи: Завершення роботи програми без втрати даних або порушення цілісності файлів, які були відкриті.

5. Сервісні можливості редактора: логічні групи команд; розтягання, стиснення, повороти; масштабування малюнка; контекстні і випливаючі меню; підказки про події на екрані; довідкова і навчальна підсистема редактора.

- **Логічні групи команд**: Групування схожих функцій або операцій разом для зручності використання.
- **Розтягання, стиснення, повороти**: Маніпулювання розміром та орієнтацією об'єктів на екрані.

- **Машстабування:** Збільшення або зменшення розміру малюнка без втрати якості.
- **Контекстні і випливаючі меню**: Меню, яке з'являється в редакторі під час виконання певних дій або під час взаємодії з певними об'єктами. Випливаючі меню з'являються під час натискання правої кнопки миші.
- **Підказки про події на екрані**: Інформація, яка з'являється при наведенні курсора на певні об'єкти або елементи інтерфейсу, щоб пояснити їх функції або значення.
- **Довідкова і навчальна підсистема редактора**: Включає довідкову інформацію та інструкції щодо використання редактора, яка може бути доступною через вбудовану довідку або онлайн-ресурси.

6. Графічний програмний інструментарій: перо, пензель, шрифт. Загальні характеристики.

- **Перо**: Інструмент для малювання ліній або обводок. Може мати різні товщини та стилі.
- **Пензель**: Інструмент для зафарбовування областей зображення. Може мати різні форми, розміри та типи текстури.
- **Шрифт**: Набір символів та знаків, які використовуються для відображення тексту на зображенні. Має різні розміри, стилі та типи.

7. Операції читання/запису для графічних зображень. Формати збереження у файлах.

- **Операції читання/запису**: Можливість завантажувати та зберігати графічні зображення з іншими форматами даних.
- **Формати збереження**: Включають BMP, JPEG, PNG, GIF, TIFF тощо. Кожен формат має свої унікальні характеристики та переваги, такі як стиснення, якість та підтримка прозорості.

8. Растрові і векторні методи малювання. Зберігання растрових і векторних зображень. Формати растрові і векторні.

- **Растрові методи малювання**: Базуються на малюванні кожного пікселя окремо. Розмір файлу залежить від роздільної здатності та кількості кольорів.
- **Векторні методи малювання**: Базуються на використанні математичних об'єктів, таких як лінії та криві. Розмір файлу не залежить від роздільної здатності та кількості кольорів.
- **Зберігання растрових і векторних зображень**: Растрові зображення зазвичай зберігаються в BMP, JPEG, PNG тощо. Векторні зображення зазвичай зберігаються в SVG, PDF, AI тощо.

9. Принципи будови графічних редакторів на основі растрових і на основі векторних зображень. Характеристика особливостей растрового і векторного малювання, переваги і недоліки кожного методу.

- **Растрові редактори**: Спеціалізуються на маніпулюванні растровими зображеннями, такими як фотографії. Вони надають більшу гнучкість у редагуванні піксельних даних, але можуть втрачати якість при масштабуванні.
- **Векторні редактори**: Спеціалізуються на маніпулюванні векторними об'єктами, такими як лінії та фігури. Вони забезпечують високу якість зображення незалежно від розміру, але можуть бути обмежені у тому, які типи ефектів можуть приймати

10. Події Windows, зв'язані з малюванням і відновленням зображень у вікні. Перелік подій та їх характеристика.

1. **WM_PAINT**: Подія, що відбувається, коли система вимагає перемалювання вмісту вікна.
2. **WM_ERASEBKGND**: Сигнал, що викликається перед WM_PAINT, для стирання фону вікна.
3. **WM_SIZE**: Виникає при зміні розміру вікна.
4. **WM_MOVE**: Сповіщає про переміщення вікна.
5. **WM_CREATE**: Сповіщає про створення вікна.

11. Стандартні класи системи програмування для малювання. Класи без зберігання малюнка (без поля пам'яті) і класи зі зберіганням малюнка у власній пам'яті.

- **Класи без зберігання малюнка**: Наприклад, GDI (Graphics Device Interface) для Windows, який просто робить малювання на екрані без зберігання малюнка.
- **Класи зі зберіганням малюнка у власній пам'яті**: Наприклад, класи відомі в області веб-програмування, такі як Canvas у HTML5 або контекст малювання у більшості графічних бібліотек.

12. Поняття про пересування та зміну розмірів видимих елементів вікна під час виконання програми.

Пересування та зміна розмірів елементів вікна можуть відбуватися шляхом зміни їх положення та розміру відповідно. Це може викликати перемалювання елементів у вікні.

13. Події миші і клавіатури, які можна використати для пересування і зміни розмірів видимих елементів вікна під час виконання програми. Параметри подій.

- **Mouse Events**:
  - **WM_MOUSEMOVE**: Подія руху миші.
  - **WM_LBUTTONDOWN**, **WM_RBUTTONDOWN**: Події натискання лівої або правої кнопки миші.
  - **WM_MOUSEWHEEL**: Подія прокрутки колеса миші.
- **Keyboard Events**:
  - **WM_KEYDOWN**, **WM_KEYUP**: Події натискання та відпускання клавіші на клавіатурі.
  - **WM_CHAR**: Подія введення символу.

14. Об'єкт пересування та зміни розміру, зміщення об'єкта в процесі пересування, зовнішні розміри об'єкта, поточна канва малювання.

- **Об'єкт пересування**: Це графічний об'єкт, який можна пересувати відносно його поточного місцезнаходження.
- **Зовнішні розміри об'єкта**: Розміри об'єкта, які відображаються на екрані.
- **Поточна канва малювання**: Робоча область, де відбувається малювання графічних об'єктів

15. Особливості малювання многокутників. Фіксування вершин полігону. Відображення проміжних полігонів. Динаміка малювання ребер полігона. Події, придатні до малювання полігона і відображення динаміки малювання.

- **Малювання многокутників**: Процес малювання фігур з багатьох сторін.
- **Фіксування вершин полігону**: Визначення та малювання кожної вершини полігона.
- **Відображення проміжних полігонів**: Малювання ліній між вершинами для утворення фігури.
- **Динаміка малювання ребер полігона**: Анімаційний ефект під час малювання полігона, наприклад, згортання чи розгортання.

16. Масштабування зображень в процесі малювання. Проблеми, які можуть виникати при ручному програмуванні масштабування.

Масштабування зображень в процесі малювання включає збільшення або зменшення розміру зображення. Проблеми, які можуть виникати при ручному програмуванні масштабування, включають:

1. **Втрата якості**: При збільшенні розміру зображення може виникнути розтягнення або розмазування, що призводить до втрати деталей.
2. **Розмитість**: При зменшенні розміру зображення може виникнути розмитість або втрата чіткості, особливо при низькій роздільній здатності.
3. **Артефакти**: Можуть виникати артефакти, такі як піксельна артифактість або аліасинг (середній ефект), які погіршують якість зображення.

4. **Витрати обчислювальних ресурсів**: Масштабування зображень може бути витратним процесом, особливо для великих зображень або при використанні складних алгоритмів масштабування.

17. Опрацювання зображень способом фільтрування. Приклади фільтрів та їх формули. Застосування фільтрів до частини загального зображення.

Опрацювання зображень способом фільтрування включає застосування певних математичних фільтрів або ядер до пікселів зображення для отримання певного ефекту або покращення якості зображення. Деякі приклади фільтрів та їх формули:

Розмивання (Blur): Зменшує різкість контурів зображення.

- Формула: $\text{new\_pixel} = \frac{1}{9} \sum_{i=1}^{3} \sum_{j=1}^{3} \text{pixel}[i, j]$

Підвищення різкості (Sharpen): Підкреслює різкість контурів та деталей зображення.

- Формула: $\text{new\_pixel} = \text{pixel} + \text{factor} \times (\text{pixel} - \text{blurred\_pixel})$

Ефект обрізання (Crop): Обрізає зображення до заданих розмірів.

Фільтр Собеля (Sobel Filter): Використовується для виявлення контурів на зображенні.

- Формула: $G_x = \sum_{i=1}^{3} \sum_{j=1}^{3} \text{kernel\_x}[i, j] \times \text{pixel}[i, j]$

18. Контекстні підказки про хід виконання графічних операцій. Перемикання контекстом підказки.

Контекстні підказки про хід виконання графічних операцій відображаються, коли користувач виконує певні дії, такі як малювання, переміщення або зміна розміру об'єктів. Ці підказки можуть вказувати користувачеві, які дії він може виконати або які клавіші він може

натиснути, щоб виконати певні операції. Перемикання контекстом підказки може відбуватися автоматично в залежності від поточного стану програми або може бути ініційоване користувачем, наприклад, за допомогою гарячих клавіш або контекстного меню.

Електронні таблиці

1. Означення термінів "активна комірка", "впливаюча комірка", "залежна комірка", "ряд даних", "властивості комірки". Алгоритм розпізнавання статусу комірки.

- **Активна комірка**: Комірка в електронній таблиці, яка вибрана або виділена для редагування або виконання певних операцій.
- **Впливаюча комірка**: Комірка, значення якої використовується в обчисленні значення іншої комірки.
- **Залежна комірка**: Комірка, значення якої залежить від значень інших комірок. Це можуть бути як впливаючі комірки, так і комірки, які використовуються в формулах.
- **Ряд даних**: Структурована послідовність комірок у вертикальному напрямку. Ряди даних зазвичай представляють різні параметри або значення.
- **Властивості комірки**: Це характеристики або атрибути, що визначають поведінку та вигляд комірки, такі як формат, стиль, розмір і т. д.

2. Формули в комірках ЕТ. Структура формули на рівні змістового призначення. Типи можливих синтаксичних і семантичних помилок в формулах. Критерії визначення помилки.

Формула в комірці електронної таблиці - це вираз, який використовується для обчислення значення комірки на основі значень інших комірок або констант. Структура формули включає оператори, функції, посилання на комірки та інші операнди. Помилки в формулах можуть виникати через невідповідність синтаксису або некоректність логіки обчислення.

3. Граматики для розпізнавання типу комірки. Алгоритм розпізнавання.

Граматика для розпізнавання типу комірки може включати правила, які описують синтаксичну структуру та можливі значення комірки. Наприклад, типи можуть визначатися за допомогою ключових слів, чисел, текстових рядків або формул.

4. Поняття лексеми формули. Перелік лексем на прикладі граматики формули. Формальне визначеня лексем.

Лексема формули - це найменша одиниця коду, яка може бути ідентифікована і оброблена сканером. Приклади лексем включають числа, оператори, функції, посилання на комірки та роздільники.

5. Загальна схема сканера. Визначення структур або класів для розпізнавання лексем сканером. Перелік параметрів лексем для їх опрацювання.

Сканер - це компонент програми, який аналізує вхідний текст та розпізнає лексеми формули. Він може використовувати різні методи, такі як регулярні вирази або алгоритми сканування, для розпізнавання лексем. Структури або класи для розпізнавання лексем можуть включати токени, які представляють різні типи лексем, а також методи для їх обробки. Параметри лексем можуть включати значення, позицію в тексті та інші властивості.

6. Алгоритми розпізнавання лексем формул у вигляді діаграми станів скінченного автомата.

Алгоритми розпізнавання лексем можна представити у вигляді діаграм станів скінченного автомата (ДСА). Кожен стан ДСА відповідає певній лексемі, а переходи між станами відбуваються залежно від зчитаного символу.

7. Схема основного циклу сканування формули і кодування лексем.

Основний цикл сканування формули включає наступні кроки:

1. Зчитування символів вхідного рядка по одному.
2. Визначення типу символу (цифра, оператор, посилання на комірку тощо).
3. Побудова лексеми на основі зчитаних символів.

4. Кодування лексеми (наприклад, за допомогою токенів або числових кодів).
5. Збереження закодованих лексем для подальшого використання.

8. Програмовані функції для кодування і декодування адресів комірок ЕТ. Зображення комірок як лексем.

Програмовані функції для кодування і декодування адрес комірок ЕТ дозволяють перетворювати адреси комірок у числові або символьні коди, що легше обробляти. Кожна комірка може бути представлена як лексема зі своїм унікальним ідентифікатором.

9. Перетворення формул ЕТ з інфіксної форми у постфіксну. Правила перетворення. Алгоритми, необхідні для перетворення.

Перетворення формул ЕТ з інфіксної форми у постфіксну може виконуватися за допомогою алгоритмів перетворення, таких як алгоритм Шунта або алгоритм зворотного поля. Правила перетворення включають у себе приоритети операторів і правила асоціативності.

10. Побудова постфіксної форми виразів на основі граматики виразів. Нисхідний граматичний розбір виразів методом рекурсивного спуску.

Побудова постфіксної форми виразів може виконуватися шляхом нисхідного граматичного розбору виразів методом рекурсивного спуску. Кожному нетермінальному символу в граматиці відповідає функція розбору, яка реалізується рекурсивно. Результатом розбору є постфіксна форма виразу, яка може бути обчислена за допомогою стека або інших структур даних.

11. Модифікація граматичних правил виразів для випадку розбору методом рекурсивного спуску. Питання ліворекурсивних правил і факторизації.

При реалізації рекурсивного спуску важливо уникати ліворекурсивних правил, що може призвести до зациклення. Факторизація граматичних правил може бути використана для розбиття правил на менші підправила, що сприяє легшому реалізації рекурсивного спуску.

12. Алгоритм взаємодії функцій розпізнавання елементів виразів для перетворення формул ЕТ з інфіксної форми у постфіксну.

Взаємодія функцій розпізнавання полягає у послідовному виклику функцій для кожного елементу виразу. Під час перетворення формул ЕТ з інфіксної форми у постфіксну, функції розпізнавання операторів та операндів викликаються відповідно до правил граматики.

13. Алгоритм фіксування помилок у формулах і друкування діагностики в процесі перетворення до постфіксної форми.

Під час перетворення формул можуть виникати помилки, такі як синтаксичні помилки або помилки в обчисленні. Для фіксації помилок і виведення діагностики можна використовувати механізми винятків або ж побудувати систему обробки помилок.

14. Інтерпретація формул ЕТ на основі постфіксної форми зображення. Структура основного алгоритму інтерпретації.

Основний алгоритм інтерпретації формул на основі постфіксної форми полягає у послідовному обчисленні операндів згідно з порядком, визначеним постфіксною формою. Для цього можна використовувати стек або інші структури даних для збереження проміжних результатів.

15. Алгоритм застосування стеку і правила опрацювання постфіксної форми виразів в процесі інтерпретації.

Під час інтерпретації постфіксної форми виразів, операнди зберігаються в стеку, а оператори оброблюються згідно з правилами операцій. Кожен оператор видаляється зі стеку після виконання операції над відповідними операндами.

16. Алгоритм опрацювання зв'язаних формул в процесі інтерпретації (залежна-впливаюча).

Зв'язані формули можна опрацьовувати шляхом інтерпретації кожної формули окремо, при цьому враховуючи залежності між ними. Наприклад, для обчислення значення комірки залежної від інших, спочатку визначаються значення впливаючих комірок.

17. Організація алгоритму інтерпретації пряморекурсивної формули. Необхідні допоміжні параметри для коректної інтерпретації та їх використання

Для інтерпретації пряморекурсивних формул може бути використана рекурсія. Допоміжні параметри можуть включати лічильники або інші структури даних для контролю процесу рекурсії і забезпечення коректної роботи алгоритму.