1. Понятие и задачи системного программного обеспечения (СПО): Системное программное обеспечение — это совокупность программных средств, обеспечивающих работу компьютера и предоставляющих услуги и ресурсы для прикладных программ. Основные задачи СПО включают:
    a. Управление ресурсами компьютера (процессором, памятью, устройствами ввода-вывода).
    b. Обеспечение интерфейса между пользователем и аппаратными средствами.
    c. Предоставление базовых сервисов для прикладных программ (например, файловые системы, сетевые протоколы).
    d. Обеспечение безопасности и надежности работы системы.

2. Структура и назначение микропроцессора и системной шины: Микропроцессор — это центральное устройство компьютера, выполняющее вычисления и управление. Он состоит из арифметико-логического устройства (АЛУ), устройства управления, регистров и шин. Системная шина — это набор проводников, связывающих микропроцессор с памятью и периферийными устройствами, обеспечивающих передачу данных, адресов и команд.

3. Структура и назначение шин адреса, данных и управления:

    a. Шина адреса: Передает адреса памяти и периферийных устройств. Её ширина определяет максимальное количество адресуемых ячеек памяти.
    b. Шина данных: Передает данные между микропроцессором, памятью и периферийными устройствами. Её ширина определяет размер данных, передаваемых за один такт.
    c. Шина управления: Передает управляющие сигналы, такие как сигналы прерывания, синхронизации, управления режимами работы.

4. Структура памяти: Память компьютера состоит из нескольких уровней, включая оперативную память (RAM), кэш-память, память на жестких дисках и других устройствах хранения. Оперативная память делится на сегменты (код, данные, стек) и страницы. Кэш-память ускоряет доступ к часто используемым данным.

5. Назначение и формирование имен регистров микропроцессора: Регистры — это небольшие, быстрые области памяти внутри микропроцессора, используемые для временного хранения данных и адресов. Имена регистров обычно определяются

архитектурой процессора (например, EAX, EBX, ECX, EDX в x86). Имена регистров выбираются для удобства программирования и отражают их специализированные функции.

6. Пользовательские регистры микропроцессора: Пользовательские регистры — это регистры, доступные программисту для использования в программе. В архитектуре x86 это включает регистры общего назначения (EAX, EBX, ECX, EDX), сегментные регистры (CS, DS, SS, ES), указатель стека (ESP), указатель базы стека (EBP) и регистр флагов (EFLAGS).

7. Регистры общего назначения: Регистры общего назначения используются для хранения временных данных и адресов. В архитектуре x86 это EAX, EBX, ECX, EDX. Например, EAX часто используется для хранения результатов арифметических операций, а ECX — для счетчиков циклов.

8. Сегментные регистры: Сегментные регистры (CS, DS, SS, ES, FS, GS) содержат базовые адреса сегментов памяти. CS указывает на сегмент кода, DS — на сегмент данных, SS — на сегмент стека, а ES, FS, GS — на дополнительные сегменты данных.

9. Назначение и использование сегмента кода: Сегмент кода содержит исполняемый код программы. Микропроцессор использует сегментный регистр CS и указатель команд (EIP) для адресации и выполнения инструкций.

10. Назначение и использование сегмента данных: Сегмент данных содержит глобальные и статические переменные программы. Микропроцессор использует сегментный регистр DS для доступа к данным.

11. Назначение и использование сегмента стека: Сегмент стека используется для временного хранения данных и адресов, таких как локберы стековые данные (например, параметры функций, локальные переменные). Микропроцессор стека (ESP) указывает на вершинний элемент элемент стека, а указатель базы стека (EBP) используется для сохранения адреса базы стека.
    Конечно, я постараюсь ответить на каждый пункт максимально подробно, как если бы я был студентом на экзамене.

12. Назначение и использование дополнительных сегментов данных.

Назначение: Дополнительные сегменты данных используются для организации и разделения данных в памяти процессора. Они позволяют улучшить управление памятью, обеспечивая более эффективное использование ресурсов и упрощая разработку программ.

Использование: В ассемблере, например, в x86 архитектуре, можно использовать дополнительные сегменты данных, такие как ES (Extra Segment), DS (Data Segment), SS (Stack Segment) и CS (Code Segment). Каждый сегмент имеет свой сегментный регистр, который указывает на базовый адрес сегмента в памяти. Это позволяет программисту организовать данные по функциональному назначению, например, разместить данные для входных и выходных операций в разных сегментах.

13. Назначение и структура регистра флагов.

Назначение: Регистр флагов (Flags Register) используется для хранения информации о состоянии процессора после выполнения арифметических и логических операций. Эти флаги помогают в принятии решений в программах, например, для условных переходов.

Структура: В x86 архитектуре регистр флагов (EFLAGS для 32-битных процессоров и RFLAGS для 64-битных) имеет следующие основные биты:
- **CF (Carry Flag):** Указывает на перенос или заём при арифметических операциях.
- **PF (Parity Flag):** Указывает на чётность количества установленных битов в младшем байте результата.
- **AF (Auxiliary Carry Flag):** Указывает на перенос из младшего полубайта (биты 3-0) в старший полубайт (биты 7-4).
- **ZF (Zero Flag):** Указывает, что результат операции равен нулю.
- **SF (Sign Flag):** Указывает на знак результата (1 — отрицательное, 0 — положительное).
- **OF (Overflow Flag):** Указывает на переполнение при операциях с знаковыми числами.

14. Флаги состояния. Флаг переноса. Флаг четности.

Флаги состояния: Флаги состояния — это биты в регистре флагов, которые отражают состояние процессора после выполнения операций. Они используются для условных переходов и других логических операций.

Флаг переноса (CF): Указывает на перенос или заём при арифметических операциях. Например, при сложении двух чисел, если результат выходит за пределы диапазона представимых значений, флаг переноса устанавливается.

Флаг четности (PF): Указывает на чётность количества установленных битов в младшем байте результата. Если количество установленных битов чётное, флаг четности устанавливается; если нечётное — сбрасывается.

15. Флаги состояния. Вспомогательный флаг переноса. Флаги нуля, знака, переполнения.

Вспомогательный флаг переноса (AF): Указывает на перенос из младшего полубайта (биты 3-0) в старший полубайт (биты 7-4). Используется при выполнении операций с BCD (Binary-Coded Decimal) числами.

Флаг нуля (ZF): Указывает, что результат операции равен нулю. Используется для условных переходов, например, `JZ` (Jump if Zero).

Флаг знака (SF): Указывает на знак результата. Если результат отрицательный, флаг знака устанавливается; если положительный — сбрасывается.

Флаг переполнения (OF): Указывает на переполнение при операциях с знаковыми числами. Например, при сложении двух положительных чисел, если результат выходит за пределы диапазона положительных чисел, флаг переполнения устанавливается.

16. Системные флаги.

Системные флаги: Это биты в регистре флагов, которые управляют режимами работы процессора и его поведением. Они включают:
- **IF (Interrupt Flag):** Управляет разрешением или запретом прерываний. Если флаг установлен, прерывания разрешены; если сброшен — запрещены.
- **DF (Direction Flag):** Определяет направление обработки строк. Если флаг установлен, строки обрабатываются в обратном направлении (с большего адреса к меньшему); если сброшен — в прямом направлении.
- **TF (Trap Flag):** Включает режим отладки, при котором процессор выполняет каждую инструкцию в режиме прерывания.

17. Регистр указателя команд.

Назначение: Регистр указателя команд (Instruction Pointer, IP) содержит адрес следующей инструкции, которая будет выполнена процессором.

Структура: В 16-битной архитектуре этот регистр называется IP, в 32-битной — EIP, а в 64-битной — RIP. Он автоматически обновляется после выполнения каждой инструкции, указывая на следующую инструкцию в памяти. Инструкции `CALL`, `RET`, `JMP` и условные переходы могут изменять значение IP.

18. Назначение системного таймера.

Назначение: Системный таймер (системный таймер) — это аппаратное устройство, которое генерирует прерывания с фиксированным интервалом. Он используется для управления временными интервалами в системе, таких как планирование задач, обновление операций и синхронизацию процессов.

19. Понятие и классификация прерываний.

Понятие: Прерывание — это прерывание нормального потока выполнения программы, которое вызывается аппаратно или программно для обработки специальных событий.

Классификация:
- **Аппаратные прерывания (Hardware Interrupts):** Генерируются внешними устройствами, такими как клавиатура, мышь, сетевые адаптеры.
- **Программные прерывания (Software Interrupts):** Генерируются программно, например, инструкцией `INT` в ассемблере. Используются для вызова системных вызов и обработки исключений.

20. Понятие вектора прерывания.

Понятие: Вектор прерывания — это адрес начального адреса обработчика прерывания. Каждому типу прерывания соотетствует свой вектор, который содержится в таблице векторов прерываний (Interrupt Vector Table, IVT).

21. Назначение обработчика прерывания.

Назначение: Обработчик прерывания — это программный модуль, который вызывается при возникновении прерывания. Он отвечает за обработку специального события, например, чтение данных из устройства ввода-вывода или обработку исключения.

22. Контроллер прерываний. Схемы соединения контроллеров прерываний.

Контроллер прерываний: Контроллер прерываний (Interrupt Controller) — это аппаратное устройство, которое управляет прерываниями, поступающими от различных устройств. Он приоритизирует прерывания и передает их процессору.

Схемы соединения:
- **Каскадное соединение:** Несколько контроллеров прерываний соединяются последовательно, что позволяет увеличить количество обрабатываемых прерываний.
- **Матричное соединение:** Контроллеры прерываний соединяются в матричной структуре, что позволяет обрабатывать большое количество прерываний с высокой приоритетной гибкойностью.

23. Системные программы: BIOS.

Назначение: BIOS (Basic Input/Output System) — это специальная программа, встроенная в микросхему на материнской плате компьютера. Она отвечает за инициализацию аппаратных ресурсов при включении компьютера, загрузку операционной системы и обеспечивает базовую поддержку ввода-вывода.

24. Системные программы: операционная система.

Назначение: Операционная система (OS) — это программное обеспечение, которое управляет аппаратными ресурсами компьютера и предоставляет пользователю и приложениям интерфейс для взаимодействия с аппаратной частью. Она обеспечивает многозадачность, управление памятью, сетевыми соединениями и другие системные функции.

25. Системные программы: службы.

Назначение: Службы (Services) — это фоновые процессы, которые запускаются операционной системой при загрузке и выполняют специальные задачи, такие как управление сетевыми соединениями, управление устройствами, обеспечение безопасности и другие системные функции.

26. Системные программы: драйверы.

Назначение: Драйверы (Device Drivers) — это программные модули, которые обеспечивают взаимодействие операционной системы с аппаратными устройствами. Они переводят общие команды операционной системы в специальные команды, понятные устройству.

27. Структура программы на Ассемблере.

Структура программы на Ассемблере включает:
- **Сегменты:** Область программы, такие как сегмент данных, сегмент кода и сегмент стека.
- **Директивы:** Команды ассемблера, такие как `ORG`, `END`, `SECTION`, которые определяют структуру и параметры программы.

- **Инструкции:** Ассемблерные команды, такие как `MOV`, `ADD`, `JMP`, которые выполняют арифметические, логические и управляющие операции.
- **Макросредства:** Макрокоманды, которые позволяют определять повторяющиеся блоки кода.

28. Облашение сегментов в TASM и NASM.

TASM (Turbo Assembler):
- **Сегмент данных:** `data segment`
- **Сегмент кода:** `code segment`
- **Сегмент стека:** `stack segment`

NASM (Netwide Assembler):
- **Сегмент данных:** `section .data`
- **Сегмент кода:** `section .text`
- **Сегмент стека:** `section .bss`

29. Облашение сегментов в EMU8086 и Turbo Debugger.

EMU8086:
- **Сегмент данных:** `model small`
- **Сегмент кода:** `code segment`
- **Сегмент стека:** `stack segment`

Turbo Debugger:
- **Сегмент данных:** `model small`
- **Сегмент кода:** `code segment`
- **Сегмент стека:** `stack segment`

30. Предложения языка Ассемблера.

Предложения языка Ассемблера включают:
- **Директивы:** Команды, которые управляют процессом ассемблирования, такие как `ORG`, `END`, `SECTION`.
- **Псевдооперации:** Специальные команды, которые имеют особое значение для ассемблера, такие как `EQU`, `=`.
- **Макросредства:** Команды, которые позволяют определять макрокоманды и блоки кода.

31. Идентификаторы в Ассемблере. Правила записей идентификаторов.

Идентификаторы: Идентификаторы — это имена переменных, меток, меток и других элементов программы.

Правила записей:
- Идентификаторы могут содержать буквы, цифры и символы подчеркивания.
- Идентификаторы не могут начинаться с цифры.
- Идентификаторы чувствительны к регистру (в некоторых ассемблерах).
- Идентификаторы не могут совпадать с зарезервированными словами ассемблера.

32. Цепочки символов (строки) в Ассемблере.

Цепочки символов (строки): Строки в ассемблере представляются последовательностями символов, заключенными в кавычки.

Пример:
```assembly
message db 'Hello, World!', 0
```

33. Целые числа в Ассемблере. Правила записи десятичных и двоичных чисел.

Десятичные числа: Записываются без префиксов.
- Пример: `123`

Двоичные числа: Записываются с префиксом `0b` или суффиксом `b`.
- Пример: `0b1101` или `1101b`

34. Целые числа в Ассемблере. Правила записи шестнадцатеричных чисел.

Шестнадцатеричные числа: Записываются с префиксом `0x` или суффиксом `h`.
- Пример: `0x1A3F` или `1A3Fh`

35. Операнды в Ассемблере. Классификация операндов.

Операнды: Операнды — это данные, на которых выполняются операции.

Классификация:
- **Константы (непосредственные операнды):** Числовые значения, например, `123`.
- **Регистровые операнды:** Значения в регистрах процессора, например, `AX`.
- **Памятные операнды:** Значения в памяти, указанные по адресу, например, `[BX]`.
- **Символьные операнды:** Значения символов, например, `'A'`.

36. Операнды в Ассемблере. Постоянные (непосредственные) операнды. Адресные операнды.

Постоянные (непосредственные) операнды: Значения, которые напрямую указываются в инструкции.
- Пример: `MOV AX, 123`

Адресные операнды: Значения, которые находятся по адресу, указанному в инструкции.
- Пример: `MOV AX, [1000h]`

37. Операнды в Ассемблере. Перемещаемые операнды. Счетчик адреса.

Перемещаемые операнды: Значения, которые могут быть перемещены между регистрами и памятью.
- Пример: `MOV AX, BX`

Счетчик адреса: Регистр, который хранит текущий адрес в памяти.
- Пример: `MOV SI, 1000h`

38. Операнды в Ассемблере. Регистровые операнды. Базовые и индексные операнды.

Регистровые операнды: Значения, хранящиеся в регистрах процессора.
- Пример: `MOV AX, BX`

Базовые операнды: Значения, хранящиеся по адресу, указываемому базовым регистром.
- Пример: `MOV AX, [BX]`

Индексные операнды: Значения, хранящиеся по адресу, указываемому базовым и индексным регистрами.
- Пример: `MOV AX, [BX + SI]`

39. Выражения в Ассемблере. Типы операторов.

Выражения: Выражения в ассемблере могут быть арифметическими, логическими или битовыми.

Типы операторов:
- **Арифметические операторы:** `+`, `-`, `*`, `/`, `%`
- **Логические операторы:** `AND`, `OR`, `XOR`, `NOT`
- **Битовые операторы:** `SHL`, `SHR`, `ROL`, `ROR`

40. Описание сегментов в Ассемблере: выравнивание и комбинирование сегмента.

Выравнивание: Определяет, как сегмент выравнивается в памяти. Например, `align 4` выравнивает сегмент по границе 4 байта.

Комбинирование сегмента: Определяет, как сегменты могут быть объединены. Например, `combine type` указывает, как сегменты могут быть объединены в один.

41. Описание сегментов в Ассемблере: класс и размер сегмента.

Класс сегмента: Определяет тип сегмента, например, `CODE`, `DATA`, `BSS`.

Размер сегмента: Определяет размер сегмента, например, `use16`, `use32`.

42. Упрощенные директивы сегментации.

Упрощенные директивы:
- **NASM:** `section .data`, `section .text`, `section .bss`
- **TASM:** `data segment`, `code segment`, `stack segment`

43. Типы данных в Ассемблере. Размерность данных простого типа. Логическая интерпретация простых данных.

Типы данных:
- **Байт (BYTE):** 8 бит
- **Слово (WORD):** 16 бит
- **Двойное слово (DWORD):** 32 бит
- **Квадрупл слово (QWORD):** 64 бит

Логическая интерпретация: Простые данные могут интерпретироваться как числа, символы или логические значения.

44. Директивы резервирования и инициализации данных в Ассемблере.

Директивы:
- **DB (Define Byte):** Резервирует и инициализирует байты.
- **DW (Define Word):** Резервирует и инициализирует слова.
- **DD (Define Double Word):** Резервирует и инициализирует двойные слова.
- **DQ (Define Quad Word):** Резервирует и инициализирует квадрупл слова.
- **RESB, RESW, RESD, RESQ:** Резервируют память без инициализации.

45. Сложные типы данных в Ассемблере: массивы. Описание и инициализация массива.

Массивы: Массивы — это последовательности элементов одного типа.

Описание и инициализация:
```assembly
array db 1, 2, 3, 4, 5
array dw 10h, 20h, 30h, 40h, 50h
```

46. Доступ к элементам массива в Ассемблере. Индексная адресация со смещением.

Доступ к элементам:
```assembly
mov bx, 0  ; Индекс элемента
mov al, [array + bx]  ; Доступ к элементу массива
```

47. Масштабирование индекса в Ассемблере. Понятие базово-индексной адресации.

Масштабирование индекса: Умножение индекса на размер элемента.

Базово-индексная адресация: Адрес вычисляется как сумма базового адреса и смещения, умноженного на размер элемента.
```assembly
mov bx, 0  ; Индекс элемента
mov al, [array + bx*2]  ; Доступ к элементу массива слов
```

48. Представление двумерных массивов в Ассемблере. Формирование адреса элемента двумерного массива.

Двумерные массивы:
```assembly
matrix dw 10h, 20h, 30h, 40h, 50h, 60h
```

Формирование адреса:
```assembly
mov bx, 0  ; Строка
mov cx, 0  ; Столбец
mov ax, [matrix + bx*4 + cx*2]
```

49. Структуры в Ассемблере. Использование структур.

Структуры: Структуры — это агрегатные типы данных, состоящие из нескольких полей.

Описание и использование:
```assembly
struc point
    .x dw 0
    .y dw 0
endstruc

point1 point <10h, 20h>
```

50. Различие между описание и определением структуры в Ассемблере. Описание шаблона структуры. Определение экземпляра структуры.

Описание шаблона:
```assembly
struc point
    .x dw 0
    .y dw 0
endstruc
```

Определение экземпляра:
```assembly
point1 point <10h, 20h>
```

51. Работа со структурами в Ассемблере. Обращение к элементам структуры.

Обращение к элементам:
```assembly
mov ax, [point1 + point.x]
mov bx, [point1 + point.y]
```

52. Массивы структур в Ассемблере. Пример использования массива структур.

Массив структур:
```assembly
```

```
struc point
    .x dw 0
    .y dw 0
endstruc

points point 3 dup (<10h, 20h>)
```

Пример использования:
```assembly
mov bx, 0  ; Индекс элемента
mov ax, [points + bx*4 + point.x]
mov bx, [points + bx*4 + point.y]
```

53. Записи в Ассемблере. Использование записей.

Записи: Записи — это структуры данных, состоящие из полей с фиксированными типами и размерами.

Описание и использование:
```assembly
record struct
    .name db 10 dup (0)
    .age db 0
endstruct

person record <'John Doe', 25>
```

54. Описание и определение записей в Ассемблере. Различие между {} и <> при определении записей.

Описание:
```assembly
record struct
    .name db 10 dup (0)
    .age db 0
endstruct
```

Определение:
```assembly
person record <'John Doe', 25>
```

```

**Различие:**
- `{}`: Используется для инициализации полей структуры.
- `<>`: Используется для инициализации полей записи.

### 55. Работа с записями в Ассемблере. Оператор WIDTH. Оператор MASK.

**Оператор WIDTH:** Возвращает размер поля записи.
```assembly
record struct
    .name db 10 dup (0)
    .age db 0
endstruct

mov ax, record.name WIDTH
```

Оператор MASK: Возвращает маску для поля записи.
```assembly
mov ax, record.name MASK
```

56. Алгоритмы выделения и изменения элемента записи. Дополнительные возможности обработки элементов записи.

Выделение элемента:
```assembly
mov bx, [person + record.name]
```

Изменение элемента:
```assembly
mov [person + record.age], 30
```

Дополнительные возможности:
- **COPY:** Копирование значений полей.
- **FILL:** Заполнение полей значениями.

57. Макросредства Ассемблера: псевдооператоры EQU и =.

EQU: Определяет константу.
```assembly

```
const1 EQU 100
```

=: Определяет переменную. Значение может меняться.
```assembly
var1 = 100
```

58. Макросредства Ассемблера: директивы слияния и выделения строк.

Слияние строк:
```assembly
%define str1 "Hello"
%define str2 "World"
%define str3 str1 str2
```

Выделение строк:
```assembly
%define str1 "Hello"
%define str2 "World"
%define str3 %substr(str1, 1, 5) %substr(str2, 1, 5)
```

59. Макросредства Ассемблера: директивы выделения подстроки в строке и определения длины строки.

Выделение подстроки:
```assembly
%define str1 "HelloWorld"
%define str2 %substr(str1, 1, 5)
```

Определение длины строки:
```assembly
%define str1 "HelloWorld"
%define len %strlen(str1)
```

60. Понятие макрокоманды Ассемблера. Макроопределение.

Макрокоманда: Макрокоманда — это блок кода, который можно вставить в программу с помощью одного имени.

Макроопределение:
```assembly
%macro print_string 1
    mov ah, 09h
    mov dx, %1
    int 21h
%endmacro

print_string message
```