1 билет

3 января 2023 г. 20:00

1. Укажите основные различия процессоров 8086 и 80386.
2. Дайте определение регистров общего назначения, перечислите их и приведите примеры их явного и неявного использования.
3. Дайте определение дескриптора, опишите, из каких основных полей он состоит. Для чего используются дескрипторы?

1. Различия 8086 80386:

1) Архитектура расширена до 32 бит - все регистры стали 32-битными с префиксом "E" (EAX, EBX, EIP…)
2) Регистр указателя команды и регистр флагов также стали 32-битными (EIP и EFLAGS). В регистре флагов добавлена новая группа флажков, к 16-битным сегментным регистрам (ES, CS, SS, DS) добавлены ещё 2 16-битных регистра (FS и GS)
3) Добавлены несколько групп 32-битных регистров: 3 регистра управления (CR0, CR2, CR3), 8 регистров отладки (DR{0..7}), 2 тестовых регистра (TR6, TR7)
4) В качестве расширенной поддержки реального режима, i386 позволяет одной или нескольким задачам работать в виртуальном режиме - режиме эмуляции режима реального адреса. Это режим работы задачи в многозадачном окружении защищенного режима.
5) Выполнение задач в виртуальном режиме практически идентично реальном, за несколькими исключениями, обусловленными защищенным режимом:

- Виртуальная задача не может выполнять привилегированные команды, так как имеет низший уровень привилегий
- Все прерывания и исключения обрабатываются операционной системой защищенного режима
- При выполнении нескольких задач виртуального режима, каждая из них может выполняться абсолютно независимо друг от друга, чего невозможно достигнуть в реальном режиме

6) В i386 механизмы защиты и многозадачности значительно расширены и улучшены. В зависимости от характера нарушений они могут тихо игнорироваться (например некоторые биты регистра EFLAGS нельзя заменить загрузкой файла из стека). Серьезные ошибки на уровне ОС (или в реальном режиме) могут привести процессор в режим аварийной остановки, из которого можно выйти только аппаратным сбросом процессора.

2. Регистры общего назначения

Регистр - это опр. участок памяти внутри самого процессора, от 8 до 32 бит длиной, которая используется для промежуточного хранения информации, обрабатываемой процессором. Некоторые регистры содержат только опр. информацию.
Регистры общего использования - *EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP* - они 32 битные и делятся на 2 части, нижние из которых *AX, BX, CD, DX* - 16 битные и делятся еще на 2 8-ми битных регистра. Так, AX делится на AH и AL, DX на DH и DL и т.д. Буква H - верхний регистр.

Так, AH и AL каждый по одному байту, AX - 2 байта (или word), EAX - 4 байта (или dword). Эти регистры используются для операций с данными, такими как сравнение, мат. операции, или запись данных в память.
Регистр CX - чаще всего счетчик в циклах.
AH в DOS программах используется как определитель, какой сервис будет использоваться при вызове INT.
Регистры EDI, ESI - индексные регистры, играют особую роль в строковых операциях.
Регистр EBP обычно используется для адресации в стеке параметров локальных переменных.
Регистр ESP - указатель стека, автоматически модифицируется командами PUSH, POP, RET, CALL, но явно используется реже.

3. Дескрипторы

Дескриптор - 8-байтовая структура, описывающая сегмент. У каждого сегмента он собственный.

Поля:

- *Адрес базы* - адрес нулевого байта описываемого сегмента в 4 Гб линейном адресном пространстве (т.е. адрес начала сегмента). Процессор образует единый 32-битный адрес.
- *Лимит сегмента* - определяет размер сегмента, конечный 20-битовый размер. Реальный лимит сегмента зависит от бита гранулярности.
- *Тип* - определяет тип сегмента, права доступа к нему, и направление роста сегмента.

Доп. поля:

- *P* - флаг присутствия сегмента. Установлен, если сегмент присутствует в памяти, сброшен - отсутствует.
- *G* - флаг гранулярности - влияет на лимит сегмента. Если сброшен - лимит измеряется в байтах, если установлен - в 4 Кб единицах.
- *AVL* - зарезервировано: это 2 бита, 20 используется по усмотрению, 21 равен 0.
- *D/B* - зависит от типа сегмента, и называется в зависимости от этого D или B. Если установлен, сегмент 32-х разрядный, если сброшен - 16.
  Сегмент кода - D, сегмент стека - B.

Основная задача дескриптора - описывать сегмент данных для правильного нахождения и использования.

2 билет

4 января 2023 г. 13:11

1. Сегментные регистры и примеры их использования.
2. Способы адресации: непосредственная, прямая, косвенная, адресация по базе. Регистры-модификаторы.
3. Блоки повторений REPT, IRP, IRPC. Синтаксис, примеры использования.

1. Сегментные регистры

Регистр - это опр. участок памяти внутри самого процессора, от 8 до 32 бит длиной, которая используется для промежуточного хранения информации, обрабатываемой процессором. Некоторые регистры содержат только опр. информацию.

Регистры сегментов - это 16-битные регистры, и содержат в себе первую половину адреса "оффсет:сегмент".

- CS - сегмент кода (страница памяти) исполняемой в данный момент программы.
  Процессор использует его всякий раз, когда надо считать из памяти очередную инструкцию для выполнения.
- DS, ES - сегмент данных исполняемой программы, т.е. константы, строковые ссылки и т.д.
  Процессор обращается сюда, если инструкция считывает или сохраняет данные. ES обычно используется при обработке массивов индексацией при помощи DI или явл. дополнительным.
- SS - сегмент стека исполняемой программы. Используется для инструкций PUSH, POP, CALL, RET.
- FS, GS - дополнительные сегменты, могут не использоваться программой.

3. Блоки повторений REPT, IRP, IRPC

1) Простейший блок повторения REPT выполняет ассемблирование участка программы заданное число раз. Этот тип блоков повторения записывается следующим образом:
   REPT k
   …
   ENDM
   Вычислив значение k, макрогенератор создаёт k точных копий тела блока и подставляет их в окончательный текст программы.
   Простейший пример:
   REPT 10
   db 0
   endm
   Эквивалентно db 10 dup(0)
2) Блоки повторений типа IRP имеют следующий вид:
   IRP p, <v1, v2, …, vn>
   <тело>
   ENDM
   p - некоторое имя, играет роль формального параметра.
   vi - фактические параметры (аргументы)
   Встречая такой блок, макрогенератор изменяет его на m копий тела, причём в i-ой копии все вхождения имени p заменяются на Vi.
   Например:
   IRP REG, <AX, CX, SI>
   PUSH REG
   ENDM
   Преобразуется в:
   PUSH AX
   PUSH CX
   PUSH SI
   Формальный параметр локализируется в теле блока (вне блока он будет недоступен) и может быть любым именем. Если оно совпадает с именем другого объекта программы, то в теле блока оно обозначает именно параметр, а не этот объект.
3) IRPC-блоки Блоки этого типа записываются так:
   IRPC p,s1…sk
   <тело>
   ENDM.
   Тут p – формальный параметр, а вот si – это символы
   Это могут быть любые символы кроме пробелов и точек с запятой.
   Встречая IRPC блок, макрогенератор заменяет его на k копий тела блока (по одной на каждый символ), причём в i-й копии все вхождения параметра p будут заменены на символ из si. Например:
   IRPC D,17W
   ADD AX, D
   ENDM
   Заменится на:
   ADD AX, 1
   ADD AX, 7
   ADD AX, W

2. Способы адресации

1) Непосредственная:
   Все арифметические операции кроме деления позволяют указывать один из операндов непосредственно в тексте программы: mov ax, 2
2) Прямая:
   В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию: mov DL, meml - Содержимое байта памяти с символическим именем meml пересылается в DL.
3) Косвенная:
   Предполагает, что во внутреннем регистре процессора находится не сам операнд, а его адрес в памяти. До процессора 80386 можно было использовать только BX, SI, DI, BP.
   Например, след. команда помещает в регистр AX слово из ячейки памяти, сегмент которой находится в DS, а смещение - в BX:
   mov ax, [bx]
4) Адресация по базе:
   Этот вид адресации является дополнением предыдущего и предназначен для доступа к данным с известным смещением относительно некоторого базового адреса
   mov ax,mas[dx]

5) Косвенная базовая индексная адресация:
   При этом виде адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра.
   Например: mov eax,[esi][edx]

   Регистры модификаторы: BX, BP, SI, DI
   При модификации адреса по двум регистрам, регистры-модификаторы должны быть из разных групп!
   При модификации адреса по одному регистру, регистром-модификатором может быть любой регистр из четырёх

3 билет

5 января 2023 г. 3:49

1. Способ адресации памяти в процессоре 8086. Каким образом он может адресовать до 1 Мб памяти?
2. Синтаксис описания макросов. Что такое макроопределение и макрокоманда?
3. Уровни привилегий в защищенном режиме. Как они используются в ОС Windows?

1. Способ адресации памяти в процессоре 8086

Режим реальной адресации или сегментная адресация памяти

Сегментная адресация памяти использует тот факт, что обращения к памяти со стороны процессора можно разделить на обращение к коду программы, обрабатываемым данным и стеку. Обращение к разным типам содержимого памяти отображаются на независимые области памяти в расширенном адресном пространстве - сегменты. В процессоре 8086 используются четыре 16-битных сегментных регистра:
CS - сегмент кода, DS - сегмент данных, ES - доп. сегмент, SS - сегмент стека.
В пределах сегмента используются 16-битные исполнительные адреса, хранящиеся в счетчике команд, указателе стека, либо вычисляемые в соответствии с видом адресации, т.е. процессору всегда доступны 4 области по 64 Кбайт каждая. Если такой объем недостаточен, процессор включает логику управления сегментными регистрами. Физический 20-битный адрес, позволяющий адресовать до 1Мбайт памяти, получается сложением исполнительного адреса и значения сегментного регистра, умноженного на 16, сегмент всегда начинается на границе блока в 16 байт и называется параграфом.

Для того, чтобы с помощью 16-разрядных регистров можно было обращаться в любую точку 20-разрядного адресного пространства, введён двухкомпонентный логический адрес из двух 16-разрядных компонент:
Segment (сегмент) : Offset (смещение)
Пример: *13DF:0100*
Где Segment - адрес сегмента, а Offset - смещение от начала этого сегмента.

3. Уровни привилегий в защищенном режиме. Как они используются в ОС Windows?

Всего есть 4 уровня привилегий: от 0 до 3.
0 - самый привилегированный, 3- наименее.
1 и 2 кольцо - для драйверов и сервисных программ, однако Windows использует только 0 и 3 кольцо. В третьем - пользовательские приложения и служебные сервисы ОС, в нулевом - всё остальное.
За уровни привилегий отвечают CPL (Current Privilege Level) и DPL (Descriptor Privilege Level).
CPL - это текущий уровень привилегий, т.е. уровень привилегий кода (задачи), исполняемой в данный момент. Хранится в битах 1-0 селектора - 4 значения. Селектор находится в CS.
DPL - уровень привилегий сегмента. Хранится в дескрипторе каждого сегмента в таблице дескрипторов. Под него также отведено 2 бита.

Немного про работу с уровнями *привилегий:*
Процессор смотрит на поле DPL только при обращении к сегменту. Например, исполняется какая-то пользовательская программа. Вдруг процессор встречает инструкцию: mov dword ptr DS:[00000000h],0 ; программа хочет записать ноль по адресу DS:00000000h (а DS равно все равно чему, например, 1234h) Тут процессор задумается: а можно ли этой подозрительной программе с CPL=3 записывать данные туда, куда она намеревается это сделать? Тут же из селектора 1234h извлекается индекс (старшие 13 бит - 246h) и в таблице дескрипторов (в данном случае – текущей LDT) находится нужный дескриптор, в котором процессор ищет в первую очередь поле DPL – допустим, там ноль. Т.е. уровень привилегий сегмента, в который хочет писать программа, явно выше, чем тот, который у программы. Тут же генерируется #GP, обработчик которого должным образом разрешит ситуацию. Может возникнуть вопрос: а если программа «успеет» записать нолик до тех пор, пока процессор производит необходимые проверки? Ответ: не успеет. #GP — это исключение типа ошибки (fault), оно всегда генерируется до исполнения инструкции

2. Синтаксис описания макросов. Макроопределение и макрокоманда

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. Описание макроса, т.е макроопределение, имеет следующий вид:
<имя макроса> MACRO <формальные gараметры>
<тело макроса>
ENDM

Макрокоманда - обращение к макроопределению. Или указание макрогенератору на то, что на указанном месте необходимо подставить тело макроопределения. Итак, одна макрокоманда заменяется на группу команд, поэтому она и называется макро (большая). Синтаксис макрокоманды:
<имя макроса> [<фактические параметры>]
Замечание. Фактические параметры можно разделять запятыми или пробелами.

Формальные параметры макроопределения заменяются соответствующими фактическими параметрами макрокоманды. i-тый фактический параметр соответствует i-тому формальному параметру Число фактических параметров должно быть равно числу формальных параметров, если фактических параметров больше, то лишние игнорируются, если формальных больше, считается что в качестве недостающих фактических указаны пустые тексты.

4 билет

7 января 2023 г. 17:23

1. Строковые инструкции ассемблера. Префиксы повторений.
2. Перечислите команды условного и безусловного перехода, циклы. Приведите примеры их использования.
3. Исключение #GP в защищенном режиме. Приведите примеры ситуаций, в которых оно возникнет.

1. **Строковые инструкции ассемблера. Префиксы повторений.**
   Префиксы повторений строковых операций:
   Количество повторений - в регистре CX, кроме того, REPZ/REPE повторяет, пока ZF равен нулю, а REPNZ/REPNE - пока не равен.
   1-й префикс - REPE (repeat if equal; повторять пока равно), REPEZ (повторять пока 0), REP - повторять.
   2-й префикс - REPNE (repeat if not equal; повторять пока не равно)
   REPNZ (повторять пока не 0).
   Все перечисленные команды явл. префиксами для операции над строками. Любой из префиксов выполняет следующую за ним команду строковой обработки столько раз, сколько указано в регистре CX, уменьшая его при каждом выполнении на 1.
   При этом REPZ и REPE прекращают выполнение команды если флаг ZF сброшен в 0.
   А REPNZ и REPNE прекращают повторения, если флаг ZF установлен в 1.
   Префикс REP обычно используется с командами MOVS, LODS, STOS, а префиксы REPE, REPNE, REPZ и REPNZ - с командами CMPS и SCAS.

   Строковые инструкции:
1) Сравнение строк CPMS. CMPS сравнивает пару элементов DS:SI с ES:DI. Также автоматически производит инкремент обоих указателей, при DF = 0. Команда CMPS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сравнение строки длиной в CX байтов или слов. В первом случае сравнение продолжается до первого совпадения в строках, а во втором — до первого несовпадения.
2) Поиск в строке: SCAS. REPNE SCASB ищет в строке элемент, равный AL. REPE SCASB ищет не равный. Команда SCAS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сканирование строки длиной в CX байтов или слов. В первом случае сканирование продолжается до первого элемента строки, совпадающего с содержимым аккумулятора, а во втором - до первого отличного.
3) Сохранение строки: STOS. Сохраняет содержимое AL/AX по адресу ES:DI. Изменяет указатели. После выполнения команды регистр DI увеличивается на 1 или 2 (если копируется байт или слово), когда флаг DF = O, и уменьшается, когда DF = 1. Команда STOS с префиксом REP заполнит строку длиной в CX числом, находящимся в аккумуляторе.
4) Загрузка строки: LODS. Считывает из адреса DS:SI в AL/AX. Команда LODS с префиксом REP выполнит копирование строки длиной в CX, и в аккумуляторе окажется последний элемент строки.

3. **Исключение #GP в защищенном режиме. Привести примеры ситуаций возникновения.**
   #GP (general protection) - исключение, генерируемое при запрете доступа к сегменту. Оно генерируется в след. случаях:
   - В случае нарушения границ. Например, обращение к байту по смещению большему, чем эффективный лимит.
   - Если происходит обращение к дескриптору, лежащему за пределами таблицы дескрипторов.
   - Также если происходит несовпадение типов при записи селектора в регистр. Например, в регистр CS загрузить селектор на дескриптор, который описывает отличный от сегмента кода сегмент;
   - При нарушении уровня привилегий. Например, при выполнении пользовательской программы процессор встречает инструкцию mov dword ptr DS:[00000000h],0 ; программа хочет записать ноль по адресу DS:00000000h.
     Тогда процессор проверяет, может ли программа с CPL=3 записывать данные туда, куда намеревается? По таблице дескрипторов смотрим поле DPL. Если, допустим, там 0, то уровень привилегий у этой программы выше, и генерируется #GP.

2. **Команды условного и безусловного перехода, циклы. Примеры использования.**
   Команды перехода:
   - JMP op - безусловный переход JMP передает управление в другую точку программы, не сохраняя какой-либо информации для возврата. Операндом может быть непосредственный адрес для перехода (в программах используют имя метки, установленной перед командой, на которую выполняется переход), а также регистр или переменная, содержащая адрес.
   - Косвенный переход JMP r16 или JMP m16. Тут берётся содержимое регистра или адреса слова памяти и по нему совершается переход. Причём, этот адрес рассматривается как полный, а не отсчитанный от команды перехода.
     Переход например по переменной L следует выполнять через оператор WORD PTR L.
   - jcc op1 - условный переход. Обычно используется после CMP в виде:
     jne local_1
     jmp far_label ; Переход, если AX = 0.
     local_1:
     …
     Циклы:
   - JCXZ op1 - переход, если CX=0 Выполняет ближний переход на указанную метку, если регистр CX равен нулю. Проверка равенства CX нулю, например, может потребоваться в начале цикла, организованного командой LOOPNE, - если в него войти с CX = 0, то он будет выполнен 65 535 раз.
   - LOOP op1 - цикл
     Уменьшает регистр CX на 1 и выполняет переход типа short на метку (которая не может быть дальше расстояния -128...+ 127 байт от команды loop), если CX не равен нулю. Эта команда используется для организации циклов, в которых регистр CX играет роль счетчика. Так, в следующем фрагменте команда ADD выполнится 10 раз:
     mov cx, 0Ah
     loop_start:
     add ax,cx
     loop loop_start
   - LOOPE op1 - цикл, пока равно
     LOOPZ op1 - цикл, пока ноль
     LOOPNE op1 - цикл, пока не равно
     LOOPNZ - цикл, пока не ноль
     Все перечисленные команды уменьшают регистр CX на один, после чего выполняют переход типа short, если ECX не равен нулю и если выполняется условие.

5 билет

7 января 2023 г. 19:36

1. Регистр флагов. Перечислите флаги процессора и их назначения.
2. Логические инструкции ассемблера, распространенные способы их применения.
3. Программные сегменты. Определение, упрощенные и стандартные директивы сегментации.

1. **Регистр флагов. Флаги процессора и их назначение**

Регистр флагов FLAGS. В FLAGS каждый бит является флагом, то есть устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора.

CF - флаг переноса, устанавливается в 1 если результат предыдущей операции не уместился в приемнике и произошел перенос старшего бита и если требуется заем (при вычитании), в противном - 0;

PF - флаг четности. Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1, и в 0, если нечетное. Это не то же самое, что делимость на два. Число делится на два без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1.

AF - флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции.

ZF - флаг нуля. Устанавливается в 1, если результат предыдущей команды – ноль;

SF - флаг знака. Он всегда равен старшему биту результата;

TF - флаг ловушки. Он был предусмотрен для работы отладчиков, не использующих защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой программной команды управление временно передается отладчику (прерывание 1);

IF - флаг прерываний. Сброс этого флага в 0 приводит к тому, что процессор перестает обрабатывать прерывания от внешних устройств. Обычно его сбрасывают на короткое время для выполнения критических участков кода.

DF — флаг направления. Он контролирует поведение команд обработки строк: когда он установлен в 1, строки обрабатываются в сторону уменьшения адресов, когда DF = 0 - наоборот;

OF - флаг переполнения. Он устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.

2. **Логические инструкции ассемблера, распространенные способы их применения.**

Логические операции AND, OR, NOT, XOR

1) *and op1, op2* - логическое И Команда выполняет побитовое «логическое И» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, только если соответствующие биты обоих операндов были равны 1, и равен 0 в остальных случаях.
   Наиболее часто AND применяют для выборочного обнуления отдельных битов. Например, команда and al,00001111b обнулит старшие четыре бита регистра AL, сохранив неизменными четыре младших. Флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.
2) *or op1, op2* - логическое ИЛИ Выполняет побитовое «логическое ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 0, только если соответствующие биты обоих операндов были равны 0, и равен 1 в остальных случаях.
   Команду OR чаще всего используют для выборочной установки отдельных битов. Например, команда or al,00001111b приведет к тому, что младшие четыре бита регистра AL будут установлены в 1. При выполнении команды OR флаги OF и CF обнуляются, SF, ZF и PF устанавливаются в соответствии с результатом, AF не определен.
3) *xor op1, op2* - логическое исключающее ИЛИ. Выполняет побитовое «логическое исключающее ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная; источник и приемник не могут быть переменными одновременно) и помещает результат в приемник. Любой бит результата равен 1, если соответствующие биты операндов различны, и нулю - в противном случае.
   XOR используется для самых разных операций, например:

```
xor ax,ax ; Обнуление регистра AX.
или
xor ax,bx
xor bx,ax
xor ax,bx ; Меняет местами содержимое AX и BX.
```

3. **Программные сегменты. Определение, упрощенные и стандартные директивы сегментации.**

Физически, сегмент это область памяти, занятая командами или данными, адреса которых вычисляются относительно значения в соотв. сегментном регистре.

Каждая программа содержит 3 типа сегментов:

- Сегмент кодов - содержит машинные команды для выполнения. Обычно первая выполненная команда находится в начале этого сегмента, и ОС передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента CS его адресует.
- Сегмент данных - содержит данные, константы и рабочие области, необходимые программе. Регистр сегмента DS адресует этот сегмент.
- Сегмент стека - содержит адреса возврата в операционную систему для программы, адреса для вызовов подпрограмм (для возврата в главную программу), а также используется для передачи параметров в процедуры. Регистр стека - SS - адресует данный сегмент. Адрес текущей вершины стека - SS:SP.

Упрощенные директивы сегментации

Для задания сегментов в тексте программы можно пользоваться упрощенными директивами:

.CODE — для указания начала сегмента кода;

.DATA — для указания начала сегмента данных;

.STACK — для указания начала сегмента стека.

Стандартные директивы сегментации

Наряду с упрощенными директивами сегментации может также использоваться стандартная директива SEGMENT, которая определяет начало любого сегмента. Синтаксис:

```
ИмяСегмента SEGMENT Параметры
…
ИмяСегмента ENDS
```

Директива ENDS определяет конец сегмента.

Если в программе есть несколько сегментов с одним именем, то они соединяются в один

6 билет

6 января 2023 г. 14:46

1. Представление целых чисел в памяти компьютера.
2. Инструкции пересылки в ассемблере.
3. Работа со стеком. По какому принципу идёт адресация локальных переменных подпрограмм и её аргументов в стеке?

1. Представление целых чисел в ПК

Принцип представления достаточно прост:

- Число должно быть переведено в двоичную систему счисления
- Должен быть определен объем памяти для этого числа

Если представить число в шестнадцатеричной системе счисления, станет понятно, сколько оно займет памяти – байт, слово (два байта), двойное слово (четыре байта). Шестнадцатеричные числа, которые начинаются с «буквенной» цифры, должны предваряться нулём, иначе компилятор не сможет отличить число от идентификатора

Числа хранятся как последовательность байт в перевернутом виде.

Знаковые целые числа хранятся, используя код со сдвигом.

При использовании **кода со сдвигом** (англ. *Offset binary*) целочисленный отрезок от нуля до 2^n (n — количество бит) сдвигается влево на 2^{n-1}, а затем получившиеся на этом отрезке числа последовательно кодируются в порядке возрастания кодами от 000…0 до 111…1. Например, число -5 в восьмибитном типе данных, использующем код со сдвигом, превратится в -5+128=123, то есть будет выглядеть так: 01111011.

По сути, при таком кодировании:

- К кодируемому числу прибавляют 2^{n-1};
- Переводят получившееся число в двоичную систему счисления.

Можно получить диапазон значений [-2^{n-1}; $2^{n-1} - 1$].

Программист должен учитывать в своей программе, что одно и то же число может быть знаковым и беззнаковым.

2. Инструкции пересылки в ассемблере

Одна из основных команд языка ассемблер – это команда пересылки. С её помощью можно записать в регистр значение другого регистра, константу или значение ячейки памяти, а также можно записать в ячейку памяти значение регистра или константу. Команда имеет следующий синтаксис:

MOV <операнд1>, <операнд2>

По команде MOV значение второго операнда записывается в первый операнд. Операнды должны иметь одинаковый размер. Команда не меняет флаги.

```
mov eax, ebx    ; Пересылаем значение регистра EBX в регистр EAX
mov eax, 0ffffh    ; Записываем в регистр EAX шестнадцатеричное значение ffff
mov x, 0        ; Записываем в переменную x значение 0
mov eax, x      ; Переслать значение из одной ячейки памяти в другую нельзя.
mov y, eax      ; Но можно использовать две команды MOV.
```

На самом деле процессор имеет много команд пересылки – код команды зависит от того, куда и откуда пересылаются данные. Но компилятор языка ассемблера сам выбирает нужный код в зависимости от операндов, так что, с точки зрения программиста, команда пересылки только одна.

Для перестановки двух величин используется команда обмена:

XCHG <операнд1>, <операнд2>

Каждый из операндов может быть регистром или ячейкой памяти. Однако переставить содержимое двух регистров можно, а двух ячеек памяти – нет. Операнды должны иметь одинаковый размер. Команда не меняет флаги.

3. Работа со стеком. По какому принципу идёт адресация локальных переменных подпрограмм и её аргументов в стеке?

Стек располагается в оперативной памяти в сегменте стека, и поэтому адресуется относительно сегментного регистра SS. Шириной стека называется размер элементов, которые можно помещать в него или извлекать. В нашем случае ширина стека равна двум байтам или 16 битам. Регистр SP (указатель стека) содержит адрес последнего добавленного элемента. Этот адрес также называется вершиной стека. Противоположный конец стека называется дном.

Дно стека находится в верхних адресах памяти. При добавлении новых элементов в стек значение регистра SP уменьшается, то есть стек растёт в сторону младших адресов. Как вы помните, для COM-программ данные, код и стек находятся в одном и том же сегменте, поэтому если постараться, стек может разрастись и затереть часть данных и кода (надеюсь, с вами такой беды не случится).

Для стека существуют всего две основные операции:

- добавление элемента на вершину стека (PUSH);
- извлечение элемента с вершины стека (POP);

Существуют ещё 2 команды для добавления в стек. Команда PUSHF помещает в стек содержимое регистра флагов. Команда PUSHA помещает в стек содержимое всех регистров общего назначения в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI (значение DI будет на вершине стека). Значение SP помещается то, которое было до выполнения команды. Обе эти команды не имеют операндов.

Соответственно, есть ещё 2 команды. POPF помещает значение с вершины стека в регистр флагов. POPA восстанавливает из стека все регистры общего назначения (но при этом значение для SP игнорируется).

Чтобы создать локальные переменные в процедуре, необходимо выделить для них память. Эта память выделяется в стеке. Сделать это очень просто — достаточно вычесть из регистра SP значение, равное суммарному размеру всех локальных переменных в процедуре. Так как ширина стека равна 16 бит, то это значение должно быть кратно 2 байтам. При выходе из процедуры нужно восстановить указатель стека. Обычно это выполняется командой mov sp, bp (В bp сохраняется значение sp при входе в процедуру, как в случае с параметрами, передаваемыми через стек). Код процедуры с локальными переменными будет выглядеть следующим образом:

```
;Процедура с локальными переменными
myproc:
push bp ;Сохранение BP
mov bp,sp ;Копирование указателя стека в BP
sub sp,locals_size ;Выделение памяти для локальных переменных
...
mov sp,bp ;Восстановление указателя стека
pop bp ;Восстановление BP
ret ;Возврат из процедуры
```

Код, выполняемый при входе в процедуру, называют также кодом пролога, а код, выполняемый при выходе, — кодом эпилога.

Если функция имеет параметры и происходит стандартный тип вызова процедуры. В стеке сохраняется адрес возврата и старое значение bp. BP запоминает старый размер стека, мог бы быть любой регистр, но bp можно модифицировать и высчитывать точный адрес

7 билет

6 января 2023 г. 16:10

1. Типы данных в ассемблере и их преобразование. Оператор явного указания типа и примеры его использования.
2. Инструкции арифметического, логического и циклического сдвига. Их применение.
3. Синтаксис описания процедур. Инструкции CALL и RET.

2. Инструкции арифметического, логического и циклического сдвига. Их применение.

Сдвиговые операции состоят в одновременном смещении цифр кода на фиксированное количество разрядов влево или вправо. В цифровых вычислительных устройствах обычно используются логический, циклический и арифметический сдвиги.

При логическом сдвиге происходит смещение в разрядной сетке всех цифр числового кода, включая и разряд знака. При этом в освободившимся при сдвиге разряда сетки устанавливаются нули, а не уместившиеся в сетке при сдвиге теряются. Количество разрядов на которые сдвигаются исходный набор цифр называют константой сдвига. Н-р: в 8-ми разрядную сетку исходный код 10010111 при логическом сдвиге на два разряда вправо будет иметь вид 00100101

Циклический сдвиг отличается от логического тем, что при смещении всей числовой последовательности цифры выходящие за пределы разрядной сетки снова вводятся в освобождающиеся разрядные позиции. Н-р: в 8-ми разрядной сетке исходный код 10101001 при циклическом сдвиге на 2 разряда вправо будет иметь вид 01101010

При арифметическом сдвиге производится смещение всей числовой последовательности (вправо или влево) без изменения позиции знака числа. Для положительного и отрицательного чисел представленных в прямом коде при сдвиге влево или вправо освободившиеся разряды сетки заполняются нулями. Для отрицательных чисел представленных в обратном коде, освобождающиеся разряды сетки при сдвиге числа вправо или влево заполняются единицами. Особенностью сдвига отрицательных чисел представления в дополнительном коде является то, что при сдвиге вправо освобождающиеся старшие разряды сетки заполняются единицами, а при сдвиге влево младшие разряды заполняются нулями. Н-р: при сдвиге исходного отрицательного числа представленного в дополнительном коде Хдоп = 11100100 на три разряда вправо и влево соответственно получим: Хдоп вправо = 11111100 Хдоп влево = 00100000

Логический сдвиг:
shr - правый, shl - левый. Сдвиг может быть использован для более быстрого деления или умножения на степени двойки. «Выдвинутый» байт помещается во флаг CF.
Пример: mask_b equ 10111011

```
…
mov al,mask_b shr 3 ;
```

Арифметический сдвиг:
sar - правый, sal - левый. Команда sal не сохраняет знака, но устанавливает флаг cf в случае смены знака очередным выдвигаемым битом. Команда sar сохраняет знак, восстанавливая его после сдвига каждого очередного бита

Циклический сдвиг:
ror - правый, rol - левый. Команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие, а именно: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становиться значением флага cf.

3. Синтаксис описания процедур. Инструкции CALL и RET.

Процедура (подпрограмма) — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого приоритета и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Процедуру можно определить и как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.
Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP. Последовательность вызова процедуры проста: передать параметры, передать управление на начало процедуры и по её окончанию вернуться в место её вызова и продолжить выполнение программы. Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);
  Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив PROC и ENDP, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы END, обозначающей конец программы:

```
…
.code
myproc proc near
ret
myproc endp
start proc
call myproc
…
start endp
end start
```

- в конце (после команды, возвращающей управление операционной системе);
  Размещение процедуры в конце программы предполагает, что последовательность команд, ограниченная директивами PROC и ENDP, будет размещена после команды, возвращающей управление операционной системе.

```
…
.code
start proc
call myproc
…
start endp
myproc proc near
ret
myproc endp
end start
```

- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы;
  Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае требуется предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью команды безусловного перехода jmp:

```
…
.code
start proc
jmp ml
myproc proc near
ret
myproc endp
ml:
…
start endp
end start
```

- в другом модуле.

Поскольку имя процедуры обладает теми же атрибутами, что и метка в команде перехода, то обратиться к процедуре можно с помощью любой команды условного или безусловного перехода. Но благодаря специальному механизму вызова процедур можно сохранить информацию о контексте программы в точке вызова процедуры. Под контекстом понимается информация о состоянии программы в точке вызова процедуры. В системе команд микропроцессора есть две команды, осуществляющие работу с контекстом. Это команды call и ret:

- call ИмяПроцедуры — вызов процедуры (подпрограммы).
- ret число — возврат управления вызывающей программе. число — необязательный параметр, обозначающий количество байт, удаляемых из стека при возврате из процедуры.

1. Типы данных в ассемблере и их преобразование. Оператор явного указания типа и примеры его использования.

Типы данных:
Бит - принимает 2 значения, 0 и 1.
Байт - 8 бит, минимальный объем данных, который реально может использовать комп. программа. Биты нумеруются справа налево, от 0 до 7. Принимает 256 различных значений.
Слово - 2 байта. Принимает 65 536 значений.
Двойное слово - 4 байта, учетверенное слово - 8 байт. Запись данных таким образом, что младший бит находится по младшему адресу называется little-endian нотацией.

Для описания переменных, с которыми работает программа, в ЯА используются директивы определения данных. Такие директивы называются псевдокомандами. Все они записываются в виде d* , вместо *один из предопределенных символов. Поле значения может содержать одно или несколько чисел, строк символов, операторов ? и DUP.

- ? (знак неопределённого значения: переменная считается неинициализированной и её значение на момента запуска может оказаться каким угодно)
- Константное выражение со значением от -128 до 255

Константы описываются с помощью директивы эквивалентности - EQU:
<имя> EQU <операнд>
Например, само по себе имя регистра AX ничем не говорит, но если мы в нём храним сумму, можно сделать так: SUM EQU AX, тогда вместо “AX” можно будет писать SUM.

Директива присваивания = определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличие от констант, определённых по директиве EQU, данная константа может менять своё значение, обозначая в разных частях текста программы разные числа. Например:

```
K=10
A DW K ;A DW 10
K = K+4;
B DB K ;B DB 14
```

Оператор переопределения типа ptr применяется для переопределения или уточнения типа метки или переменной, определяемых выражением: new_type ptr
new_type: byte 1, word 2, dword 4, qword 8, tword 10, near и far - ближние и дальние указатели.
Пример:

```
d_wrd dd 0
mov al, byte ptr d_wrd+1 ;пересылка второго байта из двойного слова
```

Переменная d_wrd имеет тип двойного слова. Что делать, если возникнет необходимость обращения не ко всей переменной, а только к одному из входящих в нее байтов? Если попытаться сделать это командой mov al,d_wrd+1, то транслятор выдаст сообщение о несовпадении типов операндов.
Оператор ptr позволяет непосредственно в команде переопределить тип и выполнить команду.

8 билет

8 января 2023 г. 18:41

1. Режимы работы процессора Intel 80386.
2. Арифметика повышенной точности. Опишите процедуру сложения и вычитания двух 32-битных чисел, находящихся в парах 16-битных регистров.
3. Прерывания в реальном и защищённом режимах работы процессора Intel.

1. **Режимы работы Intel 80386**

Реальный режим (Real Mode)

После инициализации центральный процессор находится в реальном режиме. В реальном режиме центральный процессор работает как очень быстрый i8086 с возможностью использования 32-битных расширений. Реальный режим не поддерживает никакой многозадачности и способен выполнять только одну программу за раз. Во многих современных операционных системах отдельные программы можно закрывать по желанию, потому что каждая задача является своим отдельным процессом, запечатанным в своей области памяти со своими ресурсами, инкапсулировано. В DOS такой инкапсуляции нет – каждая программа является единственной выполняющейся и имеет полную власть над системой, поэтому хоть программу и можно завершить посреди её выполнения, это оставит систему в очень неопределенном состоянии. Именно поэтому и нужно прерывание на завершение программы, потому что это выполняет все необходимые действия перед её завершением и позволяет запускать следующую.

Режим системного управления (System Management Mode)

Он предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы: приостанавливается выполнение любого другого кода и запускается специальная программа-обработчик. Микропроцессор переходит в этот режим только аппаратно с помощью прерываний SMI, которые не могут быть вызваны программно; к примеру, по сигналу от чипсета или периферии. Микропроцессор возвращается из режима системного управления в тот режим, при работе в котором был получен соответствующий сигнал по команде RSM. Эта команда работает только в режиме системного управления. Среди возможных применений SMM:
Обработка системных ошибок, таких как ошибки памяти и чипсета;
Функции защиты, например выключение процессоров при сильном перегреве;
Управление питанием — например, схемами изменения напряжения;
Эмуляция периферии, которая не была реализована на материнской плате или реализация которой содержит ошибки;

Защищенный режим (Protected Mode)

Защищенный режим является основным режимом работы микропроцессора. *Ключевые особенности защищенного режима: виртуальное адресное пространство, защита и многозадачность*. В защищенном режиме программа оперирует с адресами, которые могут относиться к физически отсутствующим ячейкам памяти, поэтому такое адресное пространство называется виртуальным. Размер виртуального адресного пространства
программы может превышать емкость физической памяти и достигать 64Тбайт.

Виртуальный режим i8086 (V86)

В режим V86 процессор может перейти из защищённого режима, если установить в регистре флагов бит виртуального режима. Когда процессор находится в виртуальном режиме, его поведение во многом напоминает поведение процессора i8086. В частности, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт. Виртуальный режим – это не реальный режим процессора i8086. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима. В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый.

2. **Арифметика повышенной точности. Опишите процедуру сложения и вычитания двух 32-битных чисел, находящихся в парах 16-битных регистров.**

Сложение/вычитание с учётом переноса/заёма. Допустимые типы операндов как и при ADD/SUB.

op1	*op2*	
r8	i8, r8, m8	сложение/вычитание байтов
m8	i8, r8	
r16	i16, r16, m16	сложение/вычитание слов
m16	i16, r16	

ADC аналогична ADD, только к сумме операндов ещё прибавляется значение флага CF. OP1 = op1+op2+CF. А в команде разности SBB вычитается ещё 1. OP1 = op1-op2 – CF.

adc op1, op2 - сложение с переносом

Эта команда аналогична ADD, но при этом выполняет арифметическое сложение приемника,

источника и флага CF. Пара команд ADD/ADC используется для сложения чисел повышенной

точности. Сложим, например, два 32-битных целых числа. Пусть одно из них находится в паре

регистров DX:AX (младшее слово (биты 0-15) - в AX и старшее (биты 16-31) - в DX), а другое - в

паре регистров BX:CX:

add ax, cx

adc dx, bx

Если при сложении младших двойных слов произошел перенос из старшего разряда (флаг CF = 1), то

он будет учтен следующей командой ADC.

```
X = 1204        F003
  +               +
Y = 8052        300F
    ----        ----
    9256        12012
  +             ↓
      1  ←  CF
    ----        ----
    9257        2012
```

sbb op1, op2 - вычитание с займом

Эта команда аналогична SUB, но она вычитает из приемника значение источника и дополнительно

вычитает значение флага CF. Ее можно использовать для вычитания 32-битных чисел в DX:AX и BX:CX

аналогично ADD/ADC:

sub ax, cx

sbb dx, bx

Если при вычитании младших двойных слов произошел заем, то он будет учтен при вычитании

старших слов.

3. **Прерывания в реальном и защищённом режимах работы процессора Intel.**

Прерывание — одна из базовых концепций вычислительной техники, которая заключается в том, что при наступлении какого-либо события происходит передача управления специальной процедуре, называемой обработчиком прерываний (ISR, англ. Interrupt Service Routine). Прерывание может быть вызвано в любом месте программы. После выполнения необходимых действий, обработчик прерываний, как правило, возвращает управление прерванной программе. Как правило, прерывания используются для работы с периферийными устройствами. С помощью прерываний также может быть реализована многозадачность, отладка программ и т.д.

В зависимости от источника возникновения сигнала прерывания делятся на:

асинхронные, или внешние (аппаратные) — события, которые исходят от внешних аппаратных устройств и могут произойти в любой произвольный момент: сигнал от таймера, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание (англ. Interrupt request, IRQ) - устройства сообщают, что они требуют внимания со стороны ОС;

синхронные, или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти;

программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения, драйверов и операционной системы.

Внешние прерывания, в зависимости от возможности запрета, делятся на:

маскируемые — прерывания, которые можно запрещать установкой соответствующих битов в регистре маскирования прерываний (в x86-процессорах — сбросом флага IF в регистре флагов);

немаскируемые (англ. Non-maskable interrupt, NMI) — обрабатываются всегда, независимо от запретов на другие прерывания. К примеру, такое прерывание может быть вызвано сбоем в микросхеме памяти. До окончания обработки прерывания обычно устанавливается запрет на обработку этого типа прерывания, чтобы процессор не входил в цикл обработки одного прерывания.

Все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание.

В процессорах архитектуры x86 для явного вызова синхронного прерывания имеется инструкция Int, аргументом которой является номер прерывания (от 0 до 255). В IBM PC-совместимых компьютерах обработку некоторых прерываний осуществляют подпрограммы BIOS. В отличие от реального режима, в защищённом режиме x86-процессоров обычные программы не могут обслуживать прерывания, эта функция доступна только системному коду.

MS-DOS использует для взаимодействия со своими модулями и прикладными программами прерывания с номерами от 20h до 3Fh.

Команду *int n* могут выполнять как программы, работающие в режиме ядра (наиболее привилегированном), так и программы, работающие в пользовательском режиме (наименее привилегированном). **Для пользовательских программ вызов программного прерывания — это способ воспользоваться услугами операционной системы — т. н. системными вызовами.**

Для того, чтобы обрабатывать прерывания, нужно рассмотреть сущность – IDT, interrupt descriptor table.

И в реальном и в защищенном режиме существует регистр IDTR, в котором процессор хранит адрес и лимит таблицы прерываний.

В реальном режиме база IDTR = 00000h, а лимит - 3FFh. По адресу 00000 находится так называемая таблица векторов прерываний (Interrupt Vector Table), состоящая из 256 векторов. Каждый вектор содержит смещение и сегмент своего обработчика. Обе компоненты занимают 2 байта.

В защищенном режиме дело обстоит совершенно по-другому. IDTR должен указывать на так называемую дескрипторную таблицу прерываний (Interrupt Descriptor Table, IDT), состоящую из 8-байтных дескрипторов для каждого прерывания, которая может содержать шлюзы задачи, прерывания. Мы рассмотрим только шлюз прерывания.

Он описывается следующей структурой:

P (Present) - бит присутствия. Если =1, прерывание обрабатывается, если =0 генерируется исключение общей защиты.

DPL (Descriptor Privilege Level) - уровень привилегий.

D - разрядность.

При генерации прерывания происходит следующее.

Из IDTR извлекается база таблицы дескрипторов прерываний.

В этой таблице по номеру прерывания находится дескриптор шлюза прерывания.

Если его бит Present сброшен, генерируется исключение общей защиты. Если текущий уровень привилегий отличается от уровня привилегий обработчика, происходит переключение стека и в стеке обработчика сохраняется указатель на стек прерванной задачи (SS и ESP).

В стек помещаются регистры EFLAGS, CS, EIP.

При вызове обработчика через шлюз прерывания очищается бит IF, блокируя дальнейшие маскируемые аппаратные прерывания.

После обработки прерывания обработчик должен вытолкнуть из стека код ошибки, если он там есть, и выполнить инструкцию IRETD (для 32-разрядных операндов, IRET для 16), которая восстанавливает регистр флагов из стека.

Если уровень привилегий прерванной задачи не равен уровню привилегий обработчика, выталкиваются регистры SS и ESP (обратное восстановление стека).

9 билет

9 января 2023 г. 0:03

1. Арифметические инструкции в ассемблере.
2. Перечислите макрооператоры в ассемблере и их назначения.
3. Регистры GDTR/LDTR. Чем селектор защищенного режима отличается от адреса сегмента реального?

1. Арифметические инструкции в ассемблере.

Беззнаковые числа складываются как обычно, только в двоичной системе счисления. Однако, бывают ситуации, когда сумма получается большой и не влезает в отведённую ячейку, тогда левая единица (единица переноса) отбрасывается и в качестве ответа выдаётся то, что влезло – а единица переходит в флаг CF.
(к - размер ячейки)

$$\text{сумма}(x,y)=(x+y) \bmod 2^k=\begin{cases} x+y, & \text{если } x+y< 2^k, \ CF=0 \\ x+y-2^k, & \text{если } x+y\geq 2^k, \ CF=1 \end{cases}$$

По аналогии для беззнакового вычитания:

$$\text{разность}(x,y)=(x-y) \bmod 2^k=\begin{cases} x-y, & \text{если } x\geq y, \ CF=0 \\ (2^k+x)-y, & \text{если } x<y, \ CF=1 \end{cases}$$

При сложении и вычитании знаковых чисел делается следующее: числа представляются как будто они беззнаковые, производится соответствующая операция и, от неё берётся модуль по 2^k. Пример: 3 + (-1). Их дополнительные коды: 3 и 2^k-1 = 255, при k = 8. Получается (3+255) % 256 = 2.

add op1, op2 - сложение

Op1 = op1 + op2

Приемник может быть регистром или переменной, источник - числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда ADD никак не различает числа со знаком и без знака, но, потребляя значения флагов CF (перенос при сложении чисел без знака), OF (перенос при сложении чисел со знаком) и SF (знак результата), разрешается применять ее и для тех, и для других.

sub op1, op2 - вычитание

Op1 = op1-op2

Приемник может быть регистром или переменной, источник - числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Точно так же, как и команда ADD, SUB не делает различий между числами со знаком и без знака, но флаги позволяют использовать её и для тех, и для других.

inc op1 - инкремент

Увеличивает приемник (регистр или переменная) на 1. Единственное отличие этой команды от ADD приемник,1 состоит в том, что флаг CF не затрагивается.

dec op1 - декремент

Уменьшает приемник (регистр или переменная) на 1. Единственное отличие этой команды от SUB заключается в том, что флаг CF не затрагивается.

NEG OP - Команда рассматривает свой операнд как число со знаком и изменяет его знак

Если приемник равен нулю, флаг CF устанавливается в 0, иначе - в 1.

Сложение/вычитание с учётом переноса/заёма.

Допустимые типы операндов как и при ADD/SUB. ADC аналогична ADD, только к сумме операндов ещё прибавляется значение флага CF. OP1 =op1+op2+CF. А в команде разности SBB. Op1 = op1-op2 – CF.

Сложим, например, два 32-битных целых числа. Пусть одно из них находится в паре

регистров DX:AX (младшее слово (биты 0-15) - в AX и старшее (биты 16-31) - в DX), а другое - в паре регистров BX:CX:

add ax, cx

adc dx, bx

Если при сложении младших двойных слов произошел перенос из старшего разряда (флаг CF = 1), то он будет учтен следующей командой ADC.

```
X = 1204        F003
  +             +
Y = 8052        300F
    ------      ------
    9256        12012
  +             ↓
      1  ←  CF
    ------      -----
    9257        2012
```

Команды умножения и деления

Умножение

MUL: Выполняет умножение содержимого источника (регистр или переменная) и регистра AL, AX (в зависимости от размера источника) и помещает результат в AX, DX:AX соответственно. Если старшая половина результата (AH, DX) содержит только нули, флаги CF и OF устанавливаются в 0, иначе - в 1.

IMUL: источник (регистр или переменная) умножается на AL или AX (в зависимости от размера операнда), и результат располагается в AX или DX:AX.

Деление

2. Перечислите макрооператоры в ассемблере и их назначения.

Макрооператор &

Рассмотрим пример:

IRP W,< 1, 6>

VAR**W** D**W** ?

ENDM

Получается неоднозначность. Для её устранения, непосредственно, перед символом, заменяющим фактические параметры, нужно подставить макрооператор &.

IRP W,< 1, 6>

VAR**&W** DW ?

ENDM

Оттранслируется так:

VAR1 DW ?

VAR6 DW ?

Этот макрооператор нужен для того, чтобы параметр, переданный в качестве операнда макроопределению или блоку повторений, заменялся значением до обработки ассемблером.

Макрооператор <>

Этот макрооператор используется при передаче текстовых строк в качестве параметров для макросов. Другое частое применение угловых скобок - передача списка параметров вложенному макроопределению или блоку повторений.

IRP VAL,<<1,2>,3>

DB VAL

ENDM

Преобразуется в

DB 1,2

DB 3

Или

IRPC S,<A;B>

DB '&S'

ENDM

Преобразуется в

DB 'A'

DB ';'

DB 'B'

Макрооператор !

Является экранирующим символом, который нужно подставить перед служебным (типа точки с запятой или угловой скобки), чтобы он означал не своё служебное значение, а часть параметра. Например:

IRP X, <A!>B, Hello world!>

DB '&X'

ENDM

Оттранслируется в

DB 'A>B'

DB 'Hello world!'

Макрооператор %

%<константное выражение> - вычисляет значение этого самого константного выражения. Здесь вложенность не допускается. Указывает, что находящийся за ним текст является выражением и должен быть вычислен. Обычно это требуется для того, чтобы передавать в качестве параметра в макрос не само выражение, а его результат.

K EQU 4

IRP A,<K+1,%K+1,W%K+1>

DW A

ENDM

Оттранслируется в

DW K+1

DW 5

DW W5

Макрооператор ;;

Если в теле блока повторения (и макроса) имеются комментарии, то они переносятся во все копии блока. Однако бывает так, что комментарий полезен при описании самого блока повторения, но не нужен в его копиях. В таком случае, следует

CF и OF устанавливаются в 0, иначе - в 1.

IMUL: источник (регистр или переменная) умножается на AL или AX (в зависимости от размера операнда), и результат располагается в AX или DX:AX.

Деление

DIV: Выполняет целочисленное деление без знака AX или DX:AX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL или AX, а

остаток - в AH или DX соответственно. Результат всегда округляется в сторону нуля. Флаги CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0-в реальном.

IDIV: Выполняет целочисленное деление со знаком AX или DX:AX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL или AX, а остаток - в AH или DX соответственно. Результат всегда округляется в сторону нуля.

Переполнение или деление на ноль вызывает исключение #DE (ошибка при делении) в защищенном режиме и прерывание 0 - в реальном.

Макрооператор ;;

Если в теле блока повторения (и макроса) имеются комментарии, то они переносятся во все копии блока. Однако бывает так, что комментарий полезен при описании самого блока повторения, но не нужен в его копиях. В таком случае, следует начинать комментарий не с одной точки с запятой, а с двух. Это сэкономит память при ассемблировании программы с большим количеством макроопределений.

IRP R,<AX,BX>

;;восстановление регистров

;восстановить R

POP R ;;стек -> R

ENDM

Приведётся в:

;восстановить AX

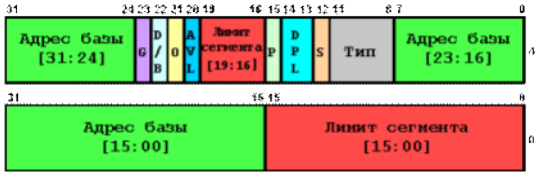
POP AX

;восстановить BX

POP BX

3. Регистры GDTR/LDTR. Чем селектор защищенного режима отличается от адреса сегмента реального?

Дескриптор – это структура, описывающая сегмент, у каждого сегмента он собственный.



Адрес базы: адрес, с которого начинается сегмент.
Лимит сегмента: определяет размер сегмента.
(G-granularity);

- если бит гранулярности сброшен (0), то 20-битное значение и будет тем самым лимитом сегмента

- если бит гранулярности установлен (1), то всё 20-битное значение автоматически увеличивается в 1000h раз.

Тип:

Определяет тип сегмента, определяет права доступа к сегменту и направление роста сегмента.

S (descriptor type) – флаг «тип дескриптора»:

Означает только одно: если сброшен (0), то описуемый сегмент – системный, если установлен (1) – это сегмент данных или кода.

DPL (descriptor privilege level) – уровень привилегий дескриптора:

P (segment present flag) – флаг присутствия сегмента:

Если установлен, значит сегмент присутствует непосредственно в памяти; если сброшен –

соответственно, отсутствует.

D/B. Зависит от типа сегмента он определяет разрядность сегмента. Если установлен – значит сегмент 32-х

разрядный, если сброшен – 16-ти. Это общий случай.

Находиться дескриптор может в трех местах:

Глобальная таблица дескрипторов (GDT - Global Descriptor Table)

Локальная таблица дескрипторов (LDT - Local Descriptor Table)

Таблица дескрипторов прерываний (IDT – Interrupt Descriptor Table)

Каждая ОС должна иметь одну таблицу GDT. Ей могут пользоваться все программы и задачи системы.

Начало таблицы GDT храниться в регистре GDTR.

GDTR:

32-битный линейный базовый адрес|16-битный лимит таблицы

Весит 48 бит. Лимит таблицы – 16-битное значение, показывает величину таблицы в байтах + 1. Сегментный дескриптор всегда занимает 8 байт. Следовательно, лимит таблицы дескрипторов – величина, равная 8N-1 байт. Первый дескриптор в GDT не используется и называется «нулевой дескриптор» (null descriptor). Загрузить/считать значение регистра GDTR можно командами LGDT/SGDT. По умолчанию база GDT равна нулю, а лимит – FFFFh

Локальная таблица дескрипторов (LDT):

Наличие LDT опционально (GDT должна быть только одна). Каждая задача может иметь свою собственную LDT, в то же время несколько задач могут использовать одну LDT на всех. LDT – это сегмент (GDT – структура данных). Это принципиально, потому что так как LDT – это сегмент, то значит у нее тоже есть свой дескриптор в той же глобальной таблице дескрипторов. Так же, как и у GDT, у LDT тоже есть свой регистр – LDTR. В отличии от GDTR этот регистр, помимо инфы про базу и лимит LDT, содержит еще одно поле – сегментный селектор.

LDTR:

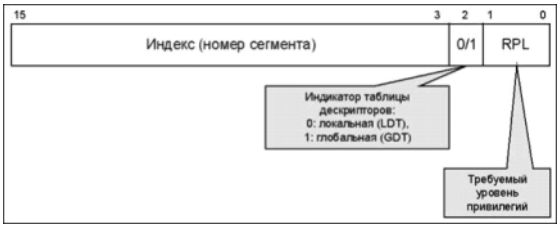
Сегм. Селектор (16 бит)|32-битный линейный базовый адрес|16-битный лимит сегмента

Инструкции LLDT и SLDT позволяют писать/читать регистр LDTR. Точно так же, при reset-е значение базы в LDTR падает в ноль, а лимит – в FFFFh.

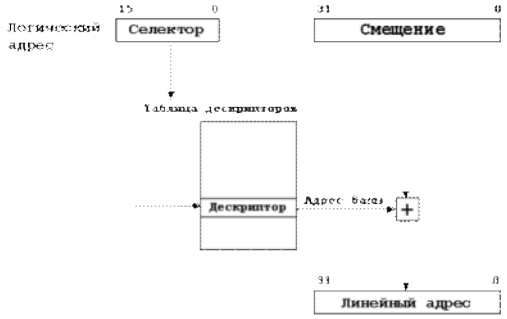
Чем селектор защищенного режима отличается от адреса сегмента реального?

Селектор это 16-битная структура данных, которая является идентификатором сегмента. Селектор указывает не на сам сегмент в памяти, а на его дескриптор, в

таблице дескрипторов. Выглядит так:



Поле индекс (биты 3-15): указывает на один из дескрипторов в таблице дескрипторов (GDT или LDT).



Преобразование логического адреса в линейный

В реальном режиме:

В сегментных регистрах указывается не весь адрес, а только первые 16 битов начального адреса сегмента. Например, если началом сегмента будет адрес 0x12340, то в сегментном регистре будет храниться число 0x1234. Начальный адрес сегмента (безпоследнего 0x0) называется номером сегмента и обозначается как seg. Тогда вычисление абсолютного (физического) адреса памяти A по значению пары SR:OFF,

где SR – сегментный регистр, а OFF – смещение, будет иметь следующий вид:

A = SR*16 + OFF.

10 билет

Tuesday, January 10, 2023 6:34 PM

1. Запись идентификаторов, целых чисел, символьных данных, комментариев, команд в языке ассемблера.
2. Директивы условного ассемблирования.
3. Прерывания: определение, обработка, роль регистра IDTR.

1. Запись идентификаторов, целых чисел, символьных данных, комментариев, команд в языке ассемблера.

1. **Идентификаторы**
   Идентификаторы нужны для обозначения различных объектов программы – переменных, меток, названий операций и т.п. В ассемблере идентификатор – это последовательность из латинских букв (больших и малых), цифр и следующих знаков: ? . @ _ $. На эту последовательность накладываются следующие ограничения:
   Длина идентификатора может быть любой, но значащими являются только первые 31 символ
   Идентификатор не должен начинаться с цифры
   Точка может быть только первым символом идентификатора
   В идентификаторах одноимённые и большие и малые буквы считаются эквивалентными (AX, aX, Ax и тд – одно и то же)
   Также символы $ и ? имеют специальные значения для ассемблера и к использованию не рекомендуются. В идентификаторах разрешается использовать буквы английского алфавита. Идентификаторы делятся на служебные слова и имена. Служебные слова имеют заранее определённый смысл, они используются для обозначения таких объектов, как регистры (ax, si и тд), названия команд (ADD, NEG и тд). Все остальные идентификаторы называются именами, программист может пользоваться ими по своему усмотрению, обозначая ими переменные, метки и другие объекты программы. В качестве имён можно использовать и служебные слова, однако крайне не рекомендуется этого Делать.
2. **Целые числа**
   В ЯА имеются средства записи целых и вещественных чисел. Целые числа можно записывать в двоичной, восьмеричной, десятичной или шестнадцатеричной системах счисления. Другие СС не допускаются. При записи чисел в десятичной форме ничего дополнительного указывать не нужно. При записи в двоичной системе, в конце ставится буква b (binary), в восьмеричной – o (octal). При записи в шестнадцатеричной СС ставится буква h (hex). Если число в шестнадцатеричной СС начинается с буквы, то перед ней нужно поставить незначащий 0.
3. **Символьные данные**
   Символы заключаются в одинарные или двойные кавычки. Естественно, что левая и правая кавычки должны быть одинаковыми. Строки (последовательности символов) также заключаются в одинарные или двойные кавычки. В качестве символов допускается использовать кириллицу. В строках и символах, в отличие от идентификаторов, большие и маленькие буквы не отождествляются. Если в строке должна находиться такая же кавычка, которыми обрамлена строка, то её необходимо просто удвоить.
4. **Комментарии**
   Комментарии не влияют на смысл программы, так как при трансляции, ассемблер игнорирует их. Они предназначены для людей, они поясняют смысл программы. Однострочный комментарий идёт после знака «точки с запятой». Также есть возможность определить многострочный комментарий. Для этого он должен начинаться со строчки: COMMENT <маркер> <текст>. Маркер, который по совместительству является операндом директивы COMMENT - любой символ, который будет считаться концом комментария: весь участок текста вплоть до следующего появления этого символа будет ассемблером игнорироваться.
5. **Команды**
   Предложения-команды – это символьная форма записи машинных команд, то есть непосредственных ассемблерных инструкций. Общий синтаксис этого типа предложений таков (все, что в квадратных скобках, не является обязательным):
   [<метка>:]<мнемокод> [<операнд>][;<комментарий>]

3. Прерывания: определение, обработка, роль регистра IDTR.
   a. Прерывание — одна из базовых концепций вычислительной техники, которая заключается в том, что при наступлении какого-либо события происходит передача управления специальной процедуре, называемой обработчиком прерываний (ISR, англ. Interrupt Service Routine). В отличие от условных и безусловных переходов, прерывание может быть вызвано в любом месте программы, в том числе если выполнение программы приостановлено, и обусловлено обычно внешними по отношению к программе событиями. После выполнения необходимых действий, обработчик прерываний, как правило, возвращает управление прерванной программе.
   b. Обработка.
   Обработчики прерываний обычно пишутся таким образом, чтобы время их обработки было как можно меньшим, поскольку во время их работы не могут обрабатываться другие прерывания, а если их будет много (особенно от одного источника), то они могут теряться.
   До окончания обработки прерывания обычно устанавливается запрет на обработку этого

2. **Директивы условного ассемблирования.**
   Каждая директива условного ассемблирования IFxxx задает конкретное условие, при вычислении которого получается истинное (true) или ложное (false) значение. Если условие имеет значение true, то выполняется ассемблирование и помещение в объектный файл блока ассемблируемого кода "тело_условия_true". Если при вычислении условия получается значение false, то Турбо Ассемблер пропускает "тело_условия_true" и не включает его в объектный файл. При наличие директивы ELSE, если условие имеет значение false, то ассемблируется и выводится в объектный файл блок "тело_условия_false". Если условие имеет значение true, то этот блок игнорируется. Условный блок завершается директивой ENDIF.
   Два описанных блока кода являются взаимноисключающими: в объектный файл включается либо "тело_условия_true", либо "тело_условия_false", но не оба блока одновременно. Кроме того, если вы используете форму IFxxx.ELSE.ENDIF, один из блоков будет обязательно включаться в объектный файл. Если используется форма IFxxx.ENDIF, то "тело_условия_true" может включаться или не включаться в объектный файл, в зависимости от условия.
   При использовании вложенных директив IF и ELSE директива ELSE всегда соответствует ближайшей директиве IF.
   Вы можете использовать директивы условного ассемблирования ELSEIFxxx как сокращенную форму, когда требуется использовать несколько директив IF. Директива ELSEIFxxx представляет собой эквивалент директивы ELSE, за которой следует вложенная директива IFxxx, но дает более компактный код.
   Кроме общих директив IF и ELSEIF ассемблеры поддерживают набор специальных команд, каждая из которых проверяет специальное условие:

   a. **IF, IFE**
   IF <константное выражение>
   IFE <константное выражение>
   <константное выражение> != 0
   <константное выражение> == 0
   Т.е. IFE/ELSEIFE проверяют, если выражение ложно (равно нулю), в отличие от IF, который проверяет его истинность.
   b. **IFIDN, IFDIF (сравниваются тексты в параметрах)**
   Эти и следующие директивы применяются в макроопределениях для проверки параметров.
   IFIDN <t1>, <t2>
   IFDIF <t1>, <t2>
   t1, t2 – любые тексты
   IFIDN <a+b>, <a+b> true
   IFIDN <a+b>, <a> false
   IFIDN <a+b>, <a+B> false
   c. **IFB, IFNB (проверка параметра на пустоту)**
   IFB <аргумент>/ELSEIFB <аргумент> - если значение аргумента - пробел;
   IFNB <аргумент>/ELSEIFNB <аргумент> - если значение аргумента - не пробел;
   IFDEF, IFNDEF – определен ли идентификатор.
   IFDEF метка/ELSEIFDEF метка - если метка определена;
   IFNDEF метка/ELSEIFNDEF метка - если метка не определена;
   d. **Директивы условного ассемблирования позволяют вам генерировать во время ассемблирования сообщения об ошибках при наступлении определенных событий. Ассемблер выводит сообщение об на экран и в файл листинга и предотвращает создание объектного файла. То есть, иногда директивы условного ассемблирования используются для того, чтобы прервать ассемблирование программы, если обнаружилась какая-нибудь ошибка.**
   Для таких случаев предназначены директивы условной генерации ошибок.
   Директива ERRxxx генерирует при удовлетворении определенных условий сообщения пользователя об ошибке. Она имеет следующий общий синтаксис:
   ERRxxx [аргументы] [сообщение]
   В этом случае директива ERRxxx представляет какую-либо из директив условной генерации сообщения об ошибке (такие как ERRIFB и т.д.).
   "Аргументы" представляют аргументы, которые могут потребоваться в директиве для вычисления условия. Некоторые директивы требуют выражения, другие требуют символьного выражения, а некоторые - одно или два текстовых выражений. Некоторые из директив вовсе не требуют аргументов.
   Если указано "сообщение", то оно задает необязательное сообщение, которое выводится с ошибкой. Сообщение должно быть заключено в кавычки (' или ").
   Директивы генерации сообщений об ошибке генерируют пользовательское сообщение об ошибке, которое выводится на экран и включается в файл листинга (если он имеется) в месте расположения директивы в исходном коде. Если директива задает сообщение, оно выводится на той же строке непосредственно за ошибкой. Например, директива:
   ERRIFNDEF foo "foo не определено!"
   если идентификатор foo не определен при обнаружении ошибки, приведет к генерации ошибки:
   User error: "foo не определено!"

   ERRE выражение - ошибка, если выражение равно нулю (ложно);
   ERRNZ выражение - ошибка, если выражение не равно нулю (истинно);
   ERRDEF метка - ошибка, если метка определена;
   ERRNDEF метка - ошибка, если метка не определена;
   ERRB <аргумент> - ошибка, если аргумент пуст;
   ERRNB <аргумент> - ошибка, если аргумент не пуст;
   ERRDIF <arg1>,<arg2> - ошибка, если аргументы различны;
   ERRDIFI <arg1>,<arg2> - ошибка, если аргументы отличаются (сравнение не различает большие и маленькие буквы);
   ERRIDN <arg1>,<arg2> - ошибка, если аргументы совпадают;
   ERRIDNI <arg1>,<arg2> - ошибка, если аргументы совпадают (сравнение не различает большие и маленькие буквы).

типа прерывания, чтобы процессор не входил в цикл обработки одного прерывания. Приоритизация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание.

Более того, существует два типа обслуживания прерываний, отличающиеся по действиям при поступлении во время обработки прерываний более приоритетных прерываний. Относительное обслуживание прерываний означает, что если во время обработки прерывания поступает более приоритетное прерывание, то это прерывание будет обработано только после завершения текущей процедуры обработки прерывания.

Абсолютное обслуживание прерываний означает, что если во время обработки прерывания поступает более приоритетное прерывание, то текущая процедура обработки прерывания вытесняется, и процессор начинает выполнять обработку вновь поступившего более приоритетного прерывания. После завершения этой процедуры процессор возвращается к выполнению вытесненной процедуры обработки прерывания.

c. IDTR
Для того, чтобы обрабатывать прерывания, нужно рассмотреть новую сущность – IDT, interrupt descriptor table.

И в реальном и в защищенном режиме существует регистр IDTR, в котором процессор хранит адрес и лимит таблицы прерываний.

В реальном режиме база IDTR = 00000h, а лимит - 3FFh (размер 400h байт минус единица). По адресу 00000 находится так называемая таблица векторов прерываний (Interrupt Vector Table), состоящая из 256 векторов. Каждый вектор содержит смещение и сегмент своего обработчика. Обе компоненты занимают 2 байта, таким образом общий размер таблицы составляет 256*2*2 = 1024 = 400h байт.

В защищенном режиме дело обстоит совершенно по-другому. IDTR должен указывать на так называемую дескрипторную таблицу прерываний (Interrupt Descriptor Table, IDT), состоящую из 8-байтных дескрипторов для каждого прерывания, которая может содержать шлюзы задачи, прерывания и ловушки.

11 билет

Wednesday, January 11, 2023 1:51 AM

1. Чем отличается директива присваивания от директивы эквивалентности?
2. Директивы определения данных. Оператор DUP.
3. Схема преобразования виртуального адреса в логический в защищенном режиме.

1. Чем отличается директива присваивания от директивы эквивалентности?

Мы рассмотрели, как в ЯА описываются переменные. Теперь рассмотрим, как в этом языке описываются константы. Это делается с помощью директивы эквивалентности – директивы EQU (equal), имеющей синтаксис:
<имя> EQU <операнд>

Например, само по себе имя регистра AX ничем не говорит, но если мы в нём храним сумму, можно сделать так: SUM EQU AX, тогда вместо “AX” можно будет писать SUM. Также константы полезны при объявлении длины массивов, в общем, всё тоже самое, как и в ЯВУ.

Теперь рассмотрим ещё одну директиву ЯА, похожую на директиву EQU – директиву присваивания:
<имя> = <константное выражение>
Эта директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличие от констант, определённых по директиве EQU, данная константа может менять своё значение, обозначая в разных частях текста программы разные числа.

Если с помощью директивы EQU можно определить имя, обозначающее не только число, но и другие конструкции, то по директиве присваивания можно определить только числовую константу. Кроме того, если имя указано в левой части директивы EQU, то оно не может появляться в левой части других директив (его нельзя переопределять). А вот имя, появившееся в левой части директивы присваивания, может снова появиться в начале другой такой директивы (но только такой!).

Другими словами, при директиве EQU значение слева – синоним значения справа и ассемблер будет подставлять значение справа вместо встретившегося синонима слева. При директиве присваивания, константа слева вычисляется сразу.

2. Директивы определения данных. Оператор DUP.

Для описания переменных, с которыми работает программа, в ЯА используются директивы определения данных. Одна из них предназначена для описания данных размером в байт, вторая – для описания данных размером в слово, а третья – для описания данных размером в двойное слово, также есть директивы для учетверенного слова (8 байт) и для размера данных в 10 байт, хотя две последние отсутствуют в некоторых вариациях языка ассемблера. В остальном эти директивы практически не отличаются друг от друга.
Такие директивы называются псевдокомандами: они приводят к включению в программу каких-либо данных или кода, но сами никаким командам и инструкциям процессора не соответствуют. Директивы определения данных указывают ассемблеру, что в соответствующем месте программы располагается переменная, устанавливают её тип, задают начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Все они записываются в виде <имя> d* <значение>, вместо *один из предопределенных символов. Поле значения может содержать одно или несколько чисел, строк символов, операторов ? и DUP.

Операнд – конструкция повторения DUP
Довольно часто в директиве DB нужно указывать повторяющиеся элементы (одинаковые операнды). Например, если мы хотим описать байтовый массив R из 8 элементов с начальным значением 0 для каждого из них, то это можно сделать так:
R DB 0,0,0,0,0,0,0,0
Но, можно сделать и так:
R DB 8 DUP(0).
Здесь в качестве операнда использована так называемая конструкция повторения, в которой сначала указывается коэффициент повторения, затем – служебное слово DUP (duplicate), а за ним в круглых скобках – повторяемая величина.
В общем случае эта конструкция имеет следующий вид:
count dup (p1, p2, p3, …, pn)
Где count – константное выражение, count >=1, pi – любой допустимый операнд директивы DB (значения, разделенные запятыми, в том числе даже вложенные операторы DUP). Данная запись является сокращением для count раз повторенной последовательности указанных в скобках операндов.

Вложенность конструкций DUP можно использовать для описания многомерных массивов.
A DB 20 DUP (30 DUP (?)) можно рассматривать как матрица 20x30.
С директивами DW и DD всё тоже самое, за исключением того, что нужно будет учитывать перевёртывание байт в памяти.

3. Схема преобразования виртуального адреса в логический в защищенном режиме.

Правило получения линейного адреса из виртуального в реальном режиме:

shl сегм.регистр, 4

add сегм.регистр, смещение

Пример 1:

1234h:5678h

Сдвинуть 1234h на 4 бита влево. Получаем 12340h.

Прибавляем 5678h. Получается: 12340h+5678h=179B8h

Т.е. вирт. адрес 1234h:5678h = лин. адресу 179B8h

Пример 2:

0001h:0000h

Сдвинуть 0001h на 4 бита влево. Получаем 0010h.

Прибавляем 0000h. Получается: 0010h+0000h=0010h

Т.е. вирт. адрес 0001h:0000h = лин. адресу 0010h

Всё в целом похоже на защищенный режим, только там вместо сдвига берется база и т.д.

12 билет

Monday, January 9, 2023 3:15 AM

1. Перечислите операторы ассемблера
2. Команды, используемые в ассемблере для поиска в строке.
3. Страничная адресация. Отличия от сегментной, основные структуры данных, схема преобразования линейного адреса в физический

1. Операторы ассемблера

- Арифметические операторы

>>>	• унарные “+” и “–”; • бинарные “+” и “–”; • умножения “*”; • целочисленного деления “/”; • получения остатка от деления “mod”.

- Операторы сдвига

>>>	***Операторы сдвига*** выполняют сдвиг выражения на указанное количество разрядов • Shr *кол-во разрядов* • Shl *кол-во разрядов*

- Операторы сравнения

>>>	***Операторы сравнения*** (возвращают значение “истина” или “ложь”) предназначены для формирования логических выражений (*см. табл.1*). Логическое значение “истина” соответствует цифровой единице, а “ложь” — нулю. • eq • ne • lt • le • gt • ge

- Логические операторы

>>>	***Логические операторы*** выполняют над выражениями побитовые операции. Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором. • and • or • xor • not

- Индексный оператор

>>>	***Индексный оператор*** **[ ]**. Не удивляйтесь, но скобки тоже являются оператором, и транслятор их наличие воспринимает как указание сложить значение ***выражение_1*** за этими скобками с ***выражение_2***, заключенным в скобки.

- Оператор переопределения типа

>>>	***Оператор переопределения типа*** **ptr** применяется для переопределения или уточнения типа метки или переменной, определяемых выражением. Тип может принимать одно из следующих значений: **byte, word, dword, qword, tbyte, near, far**.

- Оператор переопределения сегмента

>>>	***Оператор переопределения сегмента*** **:** (**двоеточие**) заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей: “имя сегментного регистра”, “имя сегмента” из соответствующей директивы SEGMENT или “имя группы”. Перед оператором может стоять: • cs • ds • ss • es • fs • gs • Имя сегмента • Имя группы

- Оператор именования типа структуры

>>>	***Оператор именования типа*** **.** (**точка**) также заставляет транслятор производить определенные вычисления, если он встречается в выражении.

- Оператор получения сегментной составляющей адреса выражения

>>>	***Оператор получения сегментной составляющей адреса выражения*** **seg** возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.

- Оператор получения смещения выражения

>>>	***Оператор получения смещения выражения*** **offset** позволяет получить значение смещения выражения в байтах относительно начала того сегмента, в котором выражение определено.

2. Команды, используемые в ассемблере для поиска в строке

Поиск в строке: *SCAS.*

REPNE SCASB ищет в строке элемент, равный AL.

REPE SCASB ищет не равный.

Сравнивает содержимое регистра AL (SCASB) или AX (SCASW) с байтом, словом или двойным словом из памяти по адресу ES:DI и устанавливает флаги аналогично команде CMP.
При использовании формы записи SCAS ассемблер сам определяет по типу указанного операнда (принято указывать имя сканируемой строки, но можно использовать любой операнд подходящего типа), какую из двух форм этой команды (SCASB или SCASW) выбрать.

После выполнения команды регистр DI увеличивается на 1 или 2 (если сканируются байты или слова), когда флаг DF = 0, и уменьшается, когда DF = 1. Команда SCAS с префиксами REPNE/REPNZ или REPE/REPZ выполняет сканирование строки длиной в CX байтов или слов. В первом случае сканирование продолжается до первого элемента строки, совпадающего с содержимым аккумулятора, а во втором - до первого отличного.

3. Страничная адресация. Отличия от сегментной, основные структуры данных, схема преобразования линейного адреса в физический

Важно:

При использовании страничной адресации структуры из сегментной адресации (как то – таблицы дескрипторов, селекторы, регистры таблиц дескрипторов) никуда не деваются. Все остается на своих местах.

Различия:

- При использовании страничной адресации линейный адрес не совпадает с физическим, как в случае с сегментной адресацией. Т.е. мы имеем дело с виртуальной памятью.

  Процессор делит линейное адресное пространство на страницы фиксированного размера (длиной 4Кб, 2Мб или 4Мб), которые, в свою очередь, уже отображаются в физической памяти (или на диске).

  Когда программа (или задача) обращается к памяти через логический адрес, процессор переводит его в линейный и затем, используя механизму страничной адресации, переводит его в соответствующий физический адрес.

  Если страницы в данный момент нет в физической памяти, то возникает исключение #PF.

  Это, по сути, кульминационный момент: обработчик этого исключения (#PF) должен выполнить соответствующие манипуляции по устранению данной проблемы, т.е. подгрузить страницу с жесткого диска (или наоборот – скинуть ненужную страницу на диск).

- Страничная организация отличается от сегментной еще и тем, что все страницы имеют фиксированный размер (в сегментной размер сегментов абсолютно произволен).

- Также при сегментной адресации все сегменты обязательно должны присутствовать непосредственно в оперативной памяти, а при страничной возможна ситуация, когда кусок сегмента находится в памяти, а другой кусок фактически того же сегмента – на жестком диске (т.е. другими словами – часть страниц сегмента находится в памяти, а часть – валяется в то же время на диске).

При трансляции линейного адреса в физический (при включенной страничной адресации) процессор использует **4 структуры данных:**

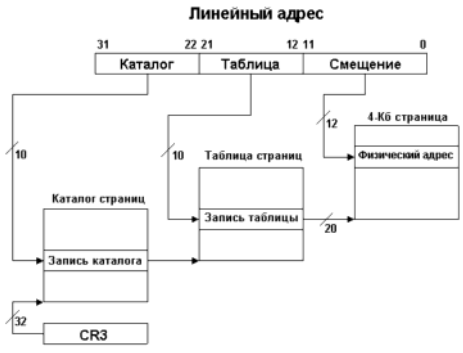
- Каталог страниц – массив 32-битных записей (PDE – page-directory entry), которые хранятся в 4Кб странице. Напоминает таблицу дескрипторов. В 4 Кб 32-битных записи помещается 1024 штуки, значит, всего 1024 PDE.

- Таблица страниц – массив 32-битных записей (PTE – page-table entry), которые также все расположены в одной 4Кб странице. Т.е. PTE-шек тоже может быть всего 1024 штуки. Забегая вперед – для 2Мб и 4Мб страниц таблица страниц вообще никак не используется – все решают только PDE- шки.

- Сама страница – 4Кб, 2Мб или 4Мб кусок памяти :)

- Указатель на каталог страниц – массив 64-битных записей, каждая из которых указывает на каталог страниц. Эта структура данных используется процессором только при использовании 36-битной адресации.

Линейный >>> Физический адрес (для 32-х разрядной адресации):



13 билет

Monday, January 9, 2023 6:48 AM

1. Команды, используемые в ассемблере для сравнения строк.
2. В чём различия между макросами и подпрограммами?
3. Опишите процедуру переключения процессора из реального режима в защищенный

1. Команды, используемые в ассемблере для сравнения строк.

Сравнение строк: *CMPS*

CMPS сравнивает пару элементов DS:SI с ES:DI. Также автоматически производит инкремент обоих указателей, при DF – 0.

Команда *CMPS* с префиксами *REPNE/REPNZ* или *REPE/REPZ* выполняет сравнение строки длиной в CX байтов или слов. В первом случае сравнение продолжается до первого совпадения в строках, а во втором — до первого несовпадения.

2. В чём различия между макросами и подпрограммами?

Процедура (подпрограмма) — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого приоритета и возврата управления в эту точку.

Макрос — символьное имя, заменяющее несколько команд языка ассемблера.

Отличие от подпрограмм:

- Оттранслированный макрос подставляется во все места, где происходил его вызов
- После работы макроса не требуется осуществлять возврат
- Макрос можно переопределить или уничтожить

Сравнительный анализ процедур и макросредств:

- Применение процедур делает код более компактным, т.е. **экономим память**
- Применение макросредств **экономит время** выполнения программы**.**
- **Большие участки кода** рекомендуется описывать как процедуры, а **маленькие** - как макроопределения.

Когда что использовать:

Процедуры дают выигрыш по памяти, а макросы - выигрыш по времени. Поэтому нельзя сказать, что процедуры лучше макросов или наоборот. Каждое из этих средств хорошо в определенных условиях, в одних случаях лучше процедуры, в других - макросы.
Если в повторяющемся фрагменте программы много команд (десяток и более), то лучше описать его как процедуру.
Но если в повторяющемся фрагменте мало команд, тогда его лучше описать как макрос. Конечно, и здесь произойдет "разбухание" программы, но не так уж и сильно.

Итак, большие фрагменты рекомендуется описывать как процедуры, а маленькие - как макросы.

3. Опишите процедуру переключения процессора из реального режима в защищенный

1. Запретить маскируемые прерывания сбросом флага IF, а возникновение немаскируемых прерываний блокировать внешней логикой.
   Программный код на время «переходного периода» должен гарантировать отсутствие исключений и не использовать программных прерываний. Это требование вызвано сменой механизма вызова обработчиков прерываний.
2. Загрузить в GDTR базовый адрес GDT (инструкцией LGDT).
3. Инструкцией MOV CR0 установить флаг PE, а если требуется страничное управление памятью, то и флаг PG.
4. Сразу после этого должна выполняться команда межсегментного перехода (JMP Far) или вызова (CALL Far) для очистки очереди инструкций, декодированных в реальном режиме, и выполнения сериализации процессора.
   Если включается страничное преобразование, то коды инструкций MOV CR0 и JMP или CALL должны находиться в странице, для которой физический адрес совпадает с логическим (для кода, которому передается управление, это требование не предъявляется).
5. Если планируется использование локальной таблицы дескрипторов, инструкцией LLDT загрузить селектор сегмента для LDT в регистр LDTR.
6. Инструкцией LTR загрузить в регистр задач селектор TSS для начальной задачи защищенного режима.
7. Перезагрузить сегментные регистры (кроме CS), содержимое которых еще относится к реальному режиму, или выполнить переход или вызов другой задачи (при этом перезагрузка регистров произойдет автоматически).
   В неиспользуемые сегментные регистры загружается нулевое значение селектора.
8. Инструкцией LIDT загрузить в регистр IDTR адрес и лимит IDT — таблицы дескрипторов прерываний защищенного режима.
9. Разрешить маскируемые и немаскируемые аппаратные прерывания.

14 билет

10 января 2023 г. 19:58

1. Инструкции двоично-десятичной коррекции.
2. Использование встроенного ассемблера (ассемблерных вставок) в языках более высокого уровня.
3. Отличия между синтаксисами ассемблера Intel и AT&T.

1. Инструкции двоично-десятичной коррекции.

Вспомним про флаг AF - флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции. Процессоры Intel поддерживают операции с двумя форматами десятичных чисел: неупакованное двоично-десятичное число - байт, принимающий значения от 00 до 09h, и упакованное двоичнодесятичное число - байт, принимающий значения от 00 до 99h. Все обычные арифметические операции над такими числами приводят к неправильным результатам. Например, если увеличить 19h на 1, то получится число 1Ah, а не 20h. Для коррекции результатов арифметических действий над двоично-десятичными числами используются приведенные ниже команды.
DAA - коррекция после сложения Если эта команда выполняется сразу после ADD (ADC, INC) и в регистре AL находится сумма двух упакованных двоично-десятичных чисел, то в AL записывается упакованное двоично-десятичное число, которое должно было стать результатом сложения. Например, если AL содержит число 19h, последовательность команд inc al daa приведет к тому, что в AL окажется 20h (а не 1 Ah, как было бы после INC). DAA выполняет следующие действия: 1. Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL увеличивается на 6, CF устанавливается, если при этом сложении произошел перенос, и AF устанавливается в 1. 2. Иначе AF = 0. 3. Если теперь старшие четыре бита AL больше 9 или флаг CF = 1, то AL увеличивается на 60h и CF устанавливается в 1. 4. Иначе CF = 0. Флаги AF и CF устанавливаются, если в ходе коррекции происходил перенос из первой или второй цифры. SF, ZF и PF устанавливаются в соответствии с результатом, флаг OF не определен.
DAS - коррекция после вычитания Если эта команда выполняется сразу после SUB (SBB или DEC) и в регистре AL находится разность двух упакованных двоично-десятичных чисел, то в AL записывается упакованное двоичнодесятичное число, которое должно было быть результатом вычитания. Например, если AL содержит число 20h, последовательность команд
dec al
das
приведет к тому, что в регистре окажется 19h (а не 1Fh, как было бы после DEC). DAS выполняет следующие действия:
1. Если младшие четыре бита AL больше 9 или флаг AF = 1, то AL уменьшается на 6, CF устанавливается, если при этом вычитании произошел заем, и AF устанавливается в 1.
2. Иначе AF = 0. 3. Если теперь старшие четыре бита AL больше 9 или флаг CF — 1,то AL уменьшается на 60h и CF устанавливается в 1. 4. Иначе CF = 0. Известный пример необычного использования этой команды - самый компактный вариант преобразования шестнадцатеричной цифры в ASCII-код соответствующего символа:
cmp al,10
sbb al,69h
das
После SBB числа 0-9 превращаются в 96h - 9Fh, а числа 0Ah - 0Fh - в 0A1h - 0A6h. Затем DAS вычитает 66h из первой группы чисел, переводя их в 30h - 39h, и 60h из второй группы чисел, переводя их в 41h — 46h.

Пример к вопросу №2
Даны целые числа a и b. Вычислить выражение a+5b.

```
#include <stdio.h>
#include <windows.h>
#include <tchar.h>
void main()
{
  char s[20];
  int a, b, sum;
  printf("a: ");
  scanf("%d",&a);
  printf("\nb: ");
  scanf("%d",&b);
  _asm
  {
    mov eax, a;
    mov ecx, 5
m: add eax, b
    loop m
    mov sum, eax
  }
  printf(«\n %d + 5*%d = %d», a, b, sum);
  getchar(); getchar();
}
```

3. Отличия между синтаксисами ассемблера Intel и AT&T

Имена регистров в AT&T начинаются с символа «%», например:
• EAX – Intel;
• %EAX – AT&T.
Числовые константы в Intel имеют суффикс «h» для 16-ричных значений и не имеют суффикса для 10-чных. Числовые константы в AT&T оформляются по правилам языка Си и имеют префикс «$». Например:
• 123h – Intel;
• $0x123 – AT&T.
Команды AT&T, имеющие операнды, используют суффикс для указания размера операнда (-ов): «B» - один байт; «W» - слово; «D» - двойное слово; «Q» - четверное слово. Команды Intel могут для этих же целей (если один из операндов находится в памяти) использовать ключевые слова «BYTE PTR», «WORD PTR», «DWORD PTR» и «QWORD PTR». Например:
• INC AL - Intel;
• INCB %AL – AT&T.
Если команда использует два операнда, то по правилам Intel сначала указывается приемник, потом источник; в синтаксисе AT&T наоборот. Например:
• MOV EAX, 123h – Intel;
• MOVD $0x123, EAX – AT&T.
Для прямой адресации в синтаксисе Intel используются квадратные скобки, в AT&T число или метка без суффиксов и префиксов. Пример:
• MOV AL, BYTE PTR [123456h] – Intel;
• MOVD 0x123456, %AL – AT&T.
Для доступа к адресу переменной в Intel используется ключевое слово «OFFSET», в AT&T просто имя метки. Пример:
• MOV EAX, OFFSET METKA - Intel;
• MOVD METKA, %EAX – AT&T.

2. Использования встроенного ассемблера в языках более высокого уровня

1) Зачем?
- Когда скорость выполнения определенного участка программы имеет большое значение.
- Некоторые алгоритмы проще реализуемы на ассемблере, чем на языке более высокого уровня.
- Если есть уже готовый код на ассемблере - проще просто его использовать.

2) Мы можем указать компилятору вставить код функции в тот участок кода, где она вызывается, то есть в то место, где выполняется вызов функции. Такая функция называется встроенной. Это похоже на вызов макроса. Преимущество - снижение использования ресурсов.
3) Форма использования ассемблера в языках более высокого уровня - ассемблерные вставки.
Ассемблерные коды в виде команд ассемблера вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как команды ассемблера и без изменений включает в формируемый им объектный код.
Эта форма удобна, если надо вставить небольшой фрагмент.
4) Использование внешних процедур и функций.
Это более универсальная форма комбинирования. У нее есть ряд преимуществ:
— написание и отладку программ можно производить независимо;
— написанные подпрограммы можно использовать в других проектах;
— облегчаются модификация и сопровождение подпрограмм.
5) Встроенный ассемблер.
При написании ассемблерных вставок используется следующий синтаксис:
_asm КодОперации операнды ; // комментарии
КодОперации задает команду ассемблера, операнды – это операнды команды.
6) Если требуется в текст программы на языке Си вставить несколько идущих подряд команд ассемблера, то их объединяют в блок:
_asm
{
текст программы на ассемблере ; комментарии
}
Внутри блока текст программы пишется с использованием синтаксиса ассемблера, при необходимости можно использовать метки и идентификаторы. Комментарии в этом случае можно записывать как после ;, так и после //.