ассемблер — это просто [хадкорный ассемблер]

крис касперски ака мыщъх, no-email

эта рубрика открывает двери в удивительный мир, расположенный за фасадом высокоуровневого программирования. ~~здесь~~ где вращаются те шестеренки, ~~что~~которые приводят в движение все остальное. ассемблер — это разговор с компьютером на естественном для него языке, радость общение с голым железом, высший пилотаж особого полета свободной мысли и безграничное пространство для самовыражения. ассемблер намного проще большинства языков высоко уровня, он значительно проще, чем Си++ и овладеть им можно буквально за несколько месяцев, главное взять правильный старт, уверенно продвигаясь в нужном направлении, а не петляя впотьмах наугад.

ассемблер — это не только ценный мех

Хакер, не знающий ассемблера, все равно, что гребец без весла. На языках высокого уровня далеко не уплывешь. Чтобы взломать приложение, исходные тексты которого недоступны (а в подавляющем большинстве случаев дело обстоит именно так), необходимо проанализировать его алгоритм, растворенный в абракадабре машинного кода. Существует множество переводчиков с машинного кода на ассемблер (такие штуки называются дизассемблерами), но автоматически восстановить исходный текст по машинному коду невозможно!

Добыча недокументированных возможностей из недр операционной системы так же ведется на ассемблере. Поиск закладок, обезвреживание вирусов, адоптация приложений под собственные нужды, заимствование чужих идей, рассекречивание засекреченных алгоритмов… Перечисление можно продолжать бесконечно. Сфера применения ассемблера настолько широка, что проще перечислить области к которым он не имеет никакого отношения.

Ассемблер — это мощное оружие, дающее безграничную власть над системой. Это седьмое чувство и второе зрение. Когда выскакивает хорошо известное окно с воплем о критической ошибке, прикладники лишь матерятся и разводят руками (это карма у программы такая). Все эти буковки и циферки для них китайская грамота. Но не для ассемблерщиков! Эти ребята спокойно идут по указанному адресу, правят баг, зачастую даже без потери не сохраненных данных!

философия ассемблера

Ассемблер — это низкоуровневый язык, оперирующий машинными понятиями и концепциями. Не ищите команду вывода строки "hello, world!". Здесь ее нет. Вот кратный перечень действий, которые может выполнить процессор: сложить/вычесть/разделить/умножить/сравнить два числа и в зависимости от полученного результата передать управление на ту или иную ветку, переслать число с одного места в другое, записать число в порт или прочитать его оттуда. Управление периферией осуществляется именно через порты или через специальную область памяти (например, видеопамять). Чтобы вывести символ на терминал, необходимо обратиться к технической документации на видеокарту, а чтобы прочитать сектор с диска — к документации по накопителю. К счастью, эту часть работы берут на себя драйвера и выполнять ее вручную обычно не требуется (к тому же, в нормальных операционных системах, таких, например, как Windows NT с прикладного уровня порты недоступны).

Другой машинной концепцией является **регистр**. Объяснить, что это такое, не погрешив против истины, невозможно. Регистр это нечто такое, что выглядит как регистр, но таковым в действительно не является. В древних машинах регистр был частью устройства обработки данных. Процессор не может сложить два числа, находящихся в оперативной памяти. Сначала он должен взять их в руки (регистры). Это — на микро уровне. Поверх микро уровня расположен интерпретатор машинных кодов, без которого не обходится не один современных процессор (да! да! машинные коды интерпретируются!). PDP-11 уже не требовал от программиста предварительной загрузки данных в регистры, делая вид, что он берет их прямо из памяти. На самом же деле, данные скрыто загружались во внутренние регистры, а после выполнения арифметических операций результат записывался в память или в… "логический" регистр, представляющий собой ячейку очень быстрой памяти.

В x86 регистры так же виртуальны, но в отличие от PDP, частично сохранили свою специализацию. Некоторые команды (например, MUL) работают со строго определенным набором регистров, который не может быть изменен. Это — плата за совместимость со старыми версиями. Другое неприятное ограничение состоит в том, что x86 не поддерживает адресации типа память – память и одно из обрабатываемых чисел обязательно должно находится в регистре или представлять собой непосредственное значение. Фактически, ассемблерная программа наполовину состоит из команд пересылки данных.

Все эти действия происходят на арене, называемой адресным пространством. Адресное пространство — это просто совокупность ячеек виртуальной памяти, доступной процессору. Операционные системы типа Windows 9x и большинство UNIX'ов создают для каждого приложения свой независимый 4 Гбайтный регион, в котором можно выделить по меньшей мере три области: область кода, область данных и стек.

Стек — это такой способ хранения данных. Что-то среднее между списком и массивом (читайте Кнута, от рулез). Команда PUSH кладет новую порцию данных на верхушку стека, а команда POP — снимает. Это позволяет сохранять данные в памяти не заботясь об их абсолютных адресах. Очень удобно! Вызов функций происходит именно так. Команда CALL func забрасывает в стек адрес следующей за ней команды, а RET стягивает его со стека. Указатель на текущую вершину хранится в регистре ESP, а дно… формально стек ограничен лишь протяженностью адресного пространства, а так — количеством выделенной ему памяти. Направление роста стека: от больших адресов — к меньшим. Еще говорят, что стек растет снизу вверх.

Регистр EIP содержит указатель на следующую выполняемую команду и непосредственно недоступен для модификации. Регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP называются регистрами общего назначения и могут свободно участвовать в любых математических операциях или операциях обращения к памяти. Их всего семь. Семь 32-разрядных регистров. Четыре первых из них (EAX, EBX, ECX и EDX) допускают обращения к своим 16-разрядным половинкам, хранящим младшее слово — AX, BX, CX и DX. Каждый из них в свою очередь делиться на старший и младший байты — AH/AL, BH/BL, CH/CL и DH/DL. Важно понять, что AL, AX и EAX это не три разных регистра, а разные части одного и того же регистра!

Существуют так же и другие регистры — сегментные, мультимедийные, регистры математического сопроцессора, отладочные регистры… Без хорошего справочника в них легко запутаться и утонуть, однако, на первых порах мы их касаться не будем.

объяснение ассемблера на сишных примерах

Основной ассемблерной командой является команда пересылки данных MOV, которую можно уподобить оператору присвоения. `c = 0x333` на языке ассемблера записывается так: `MOV EAX, 333h` (обратите внимание на разницу записи шестнадцатеричных чисел!). Можно так же записать `MOV EAX, EBX` (записать в регистр EAX значение регистра EBX).

Указатели заключаются в квадратные скобки. Сишное `a = *b` на ассемблере записывается так: `MOV EAX, [EBX]`. При желании, к указателю можно добавить смещение: `a = b[0x66]` эквивалентно: `MOV EAX, [EBX + 0x66]`.

Переменные объявляются директивами DB (переменная в один байт), DW (переменная в одно слово), DD (переменная в двойное слово) и т.д. Знаковость переменных при их объявлении не указывается. Одна и та же переменная в различных участках программы может интерпретироваться и как число со знаком и как число без знака. Для загрузки переменной в указатель применяется либо команда LEA, либо MOV с директивой offset. Покажем это на следующем примере:

```
LEA EDX,b      ;// регистр EDX содержит указатель на переменную b
MOV EBX,a      ;// регистр EBX содержит значение переменной a
MOV ECX, offset a // регистр ECX содержит указатель на переменную a

MOV [EDX],EBX  ;// скопировать переменную a b

MOV b, EBX     ;// скопировать переменную a b

MOV b, a       ;// !!!ошибка!!! так делать нельзя!!!
               ;// оба аргумента команды MOV не могут быть в памяти!

a DD 66h;// объявляем переменную a типа двойного слова и инициализируем ее числом 66h
b DD ?  ;// объявляем неинициализированную переменную b типа двойного слова
```

Листинг 1 основные типа пересылок данных

Теперь перейдем к условным переходам. Никакого "if" на ассемблере нет и эту операцию приходится осуществлять в два этапа. Команда CMP позволяет сравнить два числа, сохраняя результат своей работы во флагах. Флаги — это биты специального регистра, описание которого заняло бы слишком много места и поэтому здесь не рассматривается. Достаточно запомнить три основных состояния: меньше (bellow или less), больше (above или great) и равно (equal). Семейство команд условного перехода Jxx проверяют условие xx и, если оно истинно, совершают прыжок по указанному адресу. Например, JE прыгает, если числа равны (Jump if Equal), а JNE если неравны (Jump if Not Equal). JB/JA работают с беззнаковыми числами, а с JL/JG — со знаковыми. Любы два не противоречащих друг другу условия могут быть скомбинированы друг с другом, например: JBE — переход если одно беззнаковое число меньше другого или равно ему. Безусловный переход осуществляется командой JMP.

Конструкция CMP/Jxx больше всего похожа на Бейсковское IF xxx GOTO, чем на Си. Вот несколько примеров ее использования:

```
CMP EAX, EBX          ;// сравнить EAX и EBX
JZ xxx                ;// если они равны переход на xxx

CMP [ECX], EDX ;// сравнить *ECX и EDX
JAE yyy               ;// если беззнаковый *ECX >= EDX перейти на yyy
```

Листинг 2 основные типы условных переходов

Вызов функций на ассемблере реализуется намного сложнее, чем на Си. Во-первых, существует по меньшей мере два типа соглашений — Си и Паскаль. В Си-соглашении параметры в функцию передаются справа налево, а из стека их вычищает вызывающий функцию код. В Паскаль соглашении все происходит наоборот! Аргументы передаются слева направо, а из стека из вычищает сама функция. Большинство API-функций операционный системы Windows придерживаются комбинированного соглашения stdcall, при котором аргументы заносятся в соответствии с Си-соглашением, а из стека вычищаются по соглашению Паскаль. Возвращаемое функцией значение помещается в регистр EAX или (для передачи 64-разрядных значений используется регистровая пара EDX:EAX). Разумеется, этих соглашением необходимо придерживаться только при вызове внешних функций (API, библиотек и т. д.)."Внутренние" функции им следовать не обязаны и могут передавать аргументы любым мыслимым способом, например, через регистры.

Вот простейший пример вызовафункции:

```
PUSH offset LibName          ;// засылаем в стек смещение строки
CALL LoadLibrary             ;// вызов функции
MOV h, EAX                   ;// EAX содержит возращенное значение
```

Листинг 3 вызов API-функции операционной системы

ассемблерные вставки как тестовый стенд

Как же сложно программировать на чистом ассемблере! Минимально работающая программа содержит чертовую уйму разнообразных конструкций, сложным образом взаимодействующих друг с другом и открывающих огонь без предупреждения. Одним махом мы отрезаем себя от привычного окружения. Сложить два числа на ассемблере не проблема, но вот вывести их результат на экран…

Ассемблерные вставки — другой дело. В то время как классические руководства по ассемблеру (Зубков, Юров), буквально с первых же строк буквально бросают читателя в пучину системного программирования, устрашая его ужасающей сложностью архитектуры процессора и операционной системы, ассемблерные вставки оставляет читателя в привычном ему окружении языков Си (и/или Паскаль) и постепенно, безо всяких резких скачков, знакомит его с внутренним миром процессора. Они же позволяют начать изучение непосредственно ассемблера с 32-разрядного защищенного режима процессора: дело в том, что в чистом виде защищенный режим настолько сложен, что не может быть усвоен даже гением, и потому практически все руководства начинают изложение с описание морально устаревшего 16-разрядного реального режима, что не только оказывается бесполезным балластом, но и замечательным средством запутывания ученика (помните, "забудьте все, чему вас учили раньше…"); По своему личному опыту и опыту моих друзей могу сказать, что такая методика

обучения превосходят все остальные как минимум по двум категориям: а) **скорость** – буквально через три-четыре дня интенсивных занятий человек, ранее никогда не знавший ассемблера, начинает сносно на нем программировать; б) **легкость освоения** – изучение ассемблера происходит практически безо всякого напряжения и усилий, ни в какой момент обучения ученика не заваливают ворохом неподъемной и непроходимой информации: каждый последующий шаг интуитивно понятен и с дороги познания заботливо убраны все потенциальные препятствия.

Ну так чего же мы ждем? Для объявления ассемблерных вставок в Microsoft Visual C++ служит ключевое слово __asm, а простейшая ассемблерная программа выглядит так:

```
main()
{
        int a = 1;              // объявляем переменную a и кладем туда значение 1
        int b = 2;              // объявляем переменную a и кладем туда значение 1
        int c;                  // объявляем переменную c, но не инициализируем ее

        // начало ассемблерной вставки
        __asm{
                mov eax, a      ;// загружаем значение переменной a в регистр EAX
                mov ebx, b      ;// загружаем значение переменной b в регистр EBX
                add eax, ebx    ;// складываем EAX с EBX, записывая результат в EAX
                mov c, eax      ;// загружаем значение EAX в переменную c
        }
        // конец ассемблерной вставки

        // выводим содержимое c на экран
        // с помощью привычной нам функции printf
        printf("a + b = %x + %x = %x\n",a,b,c);
}
```

Листинг 4 ассемблерная вставка складывающая два числа

о планах на будущее

В следующих статьях этой рубрики мы докажем, что ассемблер это не заумная теоретическая муть, а самый настоящий хардкор. Самомодифицирующийся код, технологии полиморфизма, противодействие отладчикам и дизассемблерам, эксплоиты, генетически модифицированные черви, шпионаж за системными событиями, перехват паролей… Это и многое другое станет нашим!

>>> врезка. инструментарий

Программируя методами ассемблерных вставок, достаточно иметь компилятор с его IDE (например, Microsoft Visual Studio). Ассемблерные вставки отлаживаются точно так же, как и весь остальной высокоуровневый код. Удобно!

Программы, целиком написанные на ассемблере, транслируются в машинный код при помощи ассемблера. Под DOS'ом большой популярностью пользовался пакет TASM от компании Borland, но на Windows его позиция выглядит неубедительной и большинство программистов использует транслятор MASM от Microsoft, входящий в состав DDK (Device Driver Kit – набор инструментов разработчика драйверов), который можно бесплатно скачать с сервера www.Microsoft.com (для каждоый версии Windows он свой). С ним конкурирует некоммерческий транслятор FASM (http://flatassembler.net/), заточенный под нужды системных программистов и поддерживающий более естественный синтаксис. Существуют ассемблеры и под UNIX, например, NASM, входящий в штатный комплект поставки большинства дистрибьютивов. В общем, какой ассемблер выбрать — дело вкуса.

Прежде чем ассемблированная программа заработает ее необходимо скомпоновать. Для этого вполне подойдет стандартный линкер, выдернутый из той же Microsoft Visual Studio или Platform SDK. Из нестандартных можно порекомендовать ulink от Юрия Харона, поддерживающего большое количество форматов файлов и множество тонких настроек, которых другие линкеры крутить не дают. Его можно скачать с сайта фирмы Стикс: ftp://ftp.styx.cabel.net/pub/UniLink/ulnbXXXX.zip. Для некоммерческого использования он бесплатен.

Еще нам понадобиться отладчик и дизассемблер. Отладчик это инструмент для поиска ошибок в своих собственных приложениях и взламывания чужих. Их много разных:

Microsoft Visual Debugger, интегрированный в состав Microsoft Visual Studio, Microsoft Windows Debugger (сокращенно WDB), и Kernel Debugger, входящие в состав SDK и DDK, soft-ice от NuMega, OllyDbg от Олега Яшкина и т. д. Самый мощный — soft-ice, самый расширяемый — WDB, самый простой и неприхотливый — OllyDbg. Дизассемблер — это, конечно, IDA Pro. Другие и рядом не лежали.

Мелочь типа hex-редакторов, сравнивателей файлов, дамперов памяти, упаковщиков/распаковщиков так же должна быть все время под рукой. Скачать полный комплект необходимого инструментария можно, например, с сайта www.wasm.ru