Kurzanleitung zu Assembler x86

Alexander Ginschel

1. Grundlegendes

1.1. Größen

8 bits sind ein Byte. Zwei Bytes sind ein Word (16bit). Zwei Words/Wörter sind ein dword(Doubleword/32bit) und zwei dwords sind ein qword (Quadroword/64bit).
0x80 bzw 80H -> Hexadezimal für 128

2. C-Programme in Assembler konvertieren

gcc -S -masm=intel cprogramm.c -o konvertiert.s
Kleine Randnotiz! Der Compiler benutzt nach der Compilierung die Variablennamen nicht mehr. Nur ihre Werte und Funktionennamen bleiben im Assemblercode erhalten!

2.1. Speicher

Der Arbeitsspeicher wächst in zwei Richtungen. Der Heap, der Speicherplatz zur Zeit der Laufzeit zuweist, wächst nach unten (also bis zum letzten physischen Arbeitsspeicher).
Der Stack, der die statisch zugewiesenen Daten speichert, wächst nach oben (in Nullrichtung). Dessen Daten genutzter Speicherplatz kann nicht erweitert werden, wie beim viel größeren Heap.

3. Maschinencode bzw ausführbare Dateien in Assembler konvertieren

objdump -M intel -S ausführbaredatei > dumpdatei

4. Assemblersektionenarchitektur

section .text -> dort sind die ganzen Befehle
section .data -> initialiserte,statische Variablen, die auf den Stack kommen
section .rdata -> wie .data nur schreibgeschützt
section .bss (block starting symbol) -> für unitialisierte Variablen, die noch 0 sind

5. Register

RAX/EAX/AX/AH(High-Byte von AX)/AL(Low Byte von AX) -> Akkumulatorregister, was oft für arithmetische Operationen und Returns/Rückgabewerten von Funktionen genutzt wird und Input/Output bei Syscalls

RBX/EBX/BX/BH/BL -> Baseregister

RCX/ECX/CX/CH/CL -> Countregister, Schleifenzähler

RDX/EDX/DX/DH/DL -> Datenregister

RSP/ESP/SP/SPL -> Stackpointer; speichert momentane Position im Stack, hat bei jedem Programmstart einen anderen Startwert

RBP/EBP/BP/BPL -> wird dazu benutzt bei Funktionsaufrufen den Stackpointer dort zu speichern

RSI/ESI/SI/SIL -> Quell-/Sourceindex von Stringoperationen

RDI/EDI/DI/DIL -> Ziel-/Destinationindex für Stringoperationen

RIP/EIP/IP -> enthält Adresse des Befehls, der als nächstes ausgeführt werden sollte

R8-R15 -> 64bit Register für alles

CS/DS/SS/ES/FS/GS -> 16bit Segmentregister von den Segmenten Code (.text)/Data (.data)/Stack/Extra/General/General [DS:ip wäre übrigens zb Datasegment:Offset und würde somit eine physikalische Adresse ergeben]

RFLAGS/EFLAGS -> Statusflaggen, die jeweils ein Bit enthalten, beispielsweise ZF (zero-flag), CF (carry-flag), SF (sign-flag), TF (trap-flag)

6. x86 Grundbefehle

mov eax, ebx; mov eax, 0x13, mov eax, [0x4000000] -> Kopiert Wert eines Registers oder wo es hinzeigt in ein anderes Register

lea eax, [ebx+esi*4] -> (load effective address) Kopiert die berechnete Adresse in eax

add eax, 0x2; sub eax, 0x1; inc eax; dec eax -> Addieren, Subtrahieren, um Eins erhöhen oder erniedrigen (dec)

mul eax, 0x5; div eax, 0x5; imul eax, 0x5; cdq; idiv eax,0x5; -> mul multipliziert den operanden mit eax und speichert in eax. div dividiert operanden durch eax, speichert das Ergebnis in eax und den Rest in edx (imul und idiv sind die gleichen Befehle nur für vorzeichenlose Zahlen und cdq kommt immer vor idiv)

xor eax, eax; or eax, ebx; and eax, ebx; not eax -> vergleicht Werte in Binär und speichert im ersten Operanden

shl bl, 0x4; shr bl, 0x4; rol bl, 0x4; ror bl, 0x4; -> shl bl, 0x4 heißt: verschiebe bl vier Mal nach links und füge hinten vier Nullen an; shr -> shift right; und rol ist rotiere nach links, ergo platziere das erste Bit ans Ende des Bytes beispielsweie; ror-> das selbe nur umgekehrt

nop -> tue nichts (no operation)

jz 0x4000000; jnz ...; je ...; jne ..; jg ...; jge ...; jl ...; jle ...; ja ...; jb ...; jae ...; jbe ...; jo ...; js ...; jmp -> Alle machen Sprünge, die davon abhängen, ob Flaggen gesetzt sind, die diese Befehle erfordern. jz->wenn Ergebnis 0 ist; jnz -> not zero; jg -> größer als; jge -> größer als oder gleich; jl -> kleiner als; jle...; ja ->größer als 0; jae -> großer bzw gleich 0; jb -> unter Null also signed; jbe...; jo -> Overflow -> Ergebnis passt nicht ins Register; js -> prüft, ob der Wert signiert ist; jmp -> konditionslos

test eax, eax; cmp eax, 0x4 -> test ist ein logisches und und prüft, ob eax gleich 0 ist, wohingegen cmp 4 von eax subtrahiert und prüft, ob dies Null ergibt

rep; repe; repz; repne; repnz -> Stringbefehlprefixe, die ecx dekrementieren und esi und edi inkrementieren; rep -> repeat; repe -> repeat while equal; repz -> repeat while zero; repne -> repeat while not equal; repnz -> repeat while not zero

repe cmpsb ->(compare string byte, gibt auch cmpsw) vergleicht die zwei Buffer/Strings solange ecx nicht 0 ist, da ecx die Bufferlänge in diesem Fall darstellt

rep stosb -> (store string byte) überschreibt edi mit dem Wert von AL

rep movsb -Y (move string byte) kopiert esi in edi solange ecx, die Bufferlänge, nicht 0 ist

repne scasb (scan string byte) durchsucht den von edi referenzierten Buffer und subrahiert Al solange von dem referenzierten Wert, wie das Ergebnis nicht 0 ist oder ecx, die Bufferlänge, nicht 0 ist

push eax; pop ebx; pusha; pushad; popa; popad -> push und pop sind die normalen Stackbefehle. push, popa machen diese Operationen für 16bit-Werte, wohingegen pushad, popad dies nur für 32bit Werte machen

call 0x41001000 -> Funktionsaufruf, EIP kriegt als Wert den Befehl nach dem Call und wird auf den Stack gepusht

ret -> poppt Stack und springt zur Adresse, wohin der neue Wert zeigtNH Kompilation (32bit x86-Architektur)

nasm -f elf32 assembler.s -o kompilat.o && ld -m elf_i386 kompilat.o -o programm
elf32 -> executable linking format 32; ld -> linker

7. Assembler x86 Programmierung

7.1. Einfacher Output

```
global _start ; Hauptfunktion wird für den Linker benannt
_start:
        mov ecx, 4
        mov eax, 1 ; Betriebssystem weiß, dass nach syscall das Programm beendet wird
        add ebx, ecx ; ebx wird 4, also wird der Fehlercode 4 ausgegeben
        int 0x80 ; int heißt interrupt und int 0x80 ruft den 32bit syscall auf
```

Mit echo $? bekommt man den Fehlercode bzw das Ergebnis von ebx ausgegeben.

7.2. String/Variablenoutput

```
global _start ; Hauptfunktion wird für den Linker benannt
section .data
var db "Var:  " ; db = define byte
len equ $- var ; bestimmt Länge von var
section .text
_start:
        mov byte [var+6], 0x0a ; fügt Zeilenumbruch am Ende des String ein (byte ptr wäre zb masm syntax statt der jetzigen von Netwideassembler)
; mov byte sagt dem Prozessor, was genau kopiert wird
        mov eax, 4 ; bestimmt fürs Betriebssystem bzw Linux, dass ausgegeben wird
        mov ebx, 1 ; bestimmt stdoutput
        mov ecx, var ; String zum Ausgeben
        mov edx, len ; Stringlänge zur Ausgabe
        add edx, 48 ; 6 nimmt Dezimalwert des Chars 6 an, und kann so ausgegeben werden
        mov dword [var+5],edx ; An die fünften Stelle von var wird der Wert aus dex reinkopiert
        sub edx, 3 ; edx ist wieder 5
        int 80H ; Abschicken der Registerwerte ans Betriebssystem
```

7.3. Schleifen

```
global _start

_start:
        mov edx, 5 ; Startwert
        mov ecx, 3 ; Schleifenstartwert
schleife:
        dec ecx ; decrease ecx
        add edx, edx
        cmp ecx, 0 ; Flagge im Register wird gesetzt, auf welche jg reagiert
        jg schleife ; Schleife geht solange bis ecx nicht 0 ist
        mov eax, 1
        mov ebx, edx ; Der Wert von edx wird als Fehlercode ausgegeben
        int 0x80 ; syscall
```

7.4. Stack

Die Stackwerte werden standardmäßig durch den Stackpointer adressiert, da die push und pop Befehle nur Einfluss auf den Stackpointer esp(32bit) bzw sp(16bit) haben.
Bei jedem Push wird der Prozessorarchitektur entsprechend die Adresse in esp mit 4 subtrahiert, da es auf ein 32bit Register zeigt und somit um vier ganze Bytes verschoben werden muss, um auf einen neuen Eintrag im Stack zu zeigen. Es wird übrigens subtrahiert, da der Stack in Nullrichtung wächst!
push 1337 -> ist das selbe wie: sub esp, 4 | mov [esp], dword 1337

Beim Pop wird hingegen der momentane Stackwert einem im Operanden festgelegten Wert zugewiesen und der Stackpointer wieder auf die vorherige Adresse gesetzt. Er wird nicht gelöscht, sondern beim nächsten Push überschrieben.
pop ebx -> entspricht: mov ebx, dword [esp] | add esp, 4

Genz nebenbei: Var im letzten Stringbeispiel ist nichts anderes als ein Pointer, der auf einen Array (einen Chararray um genau zu sein) zeigt, weswgegen man genauso gut auch in dem Beispiel den esp hätte nehmen können, der ecx übergeben wird, wobei edx die Stringlänge bestimmt. Man müsste jedoch dann mühsam die einzelnen Bytes den Adressen nach [esp] zuweisen.

7.5. Funktionen

Funktionen realisiert man in Assembler mit einem call Befehl. Dieser puht die Adresse vom direkt nächsten Befehl nach call auf den Stack und sprint zum Label bzw der Instructionpointeradresse, die im gegeben wurde. Zurück springt man mit einem pop eax | jmp eax, insofern der momentane Stackwert noch der Befehlsadresse unter dem call entspricht.
pop eax | jmp eax kann man übrigens mit ret abkürzen.

7.5.1. Prolog und Epilog

Mithilfe von Basepointern kann man recht einfach den Stackwert mit der Adresse direkt nach dem call Befehl sichern und so dafür sorgen, dass der ret Befehl auch richtig funktioniert.
Dafür verwendet man den Prolog und den Epilog.

```
---Prolog---
push ebp ; sorgt dafür, dass man falls man in einer Funktion eine andere Funktion aufruft, nicht den ursprünglichen Basepointerwert verliert
mov ebp, esp ; speichert Stackpointer
 ... ; irgendeine Befehlsabfolge
---Epilog---
mov esp, ebp ; gibt Stackpointer den alten Wert
pop ebp ; gibt Basepointer den alten Wert zurück
ret ; pop eax | jmp eax
```

8. Nutzungen von C-Funktionen

Man kann C-Funktionen in Assembler verwende, nur sollte man darauf achten, dass man das Assemblerkompilat am Ende mit gcc zu einem Programm zusammenfasst, damit zu den C-Headern gelinkt wird. -> gcc -m32 kompilat.o -o programm

```
global main ; c braucht main Funktion
extern printf
section .data
msg db "Eine Zahl %i", 0x0a,0x00 ; C-konformer String für printf mit Zeilenumbruch und Nullbyte

section .text
main:
push ebp
mov ebp, esp ; bis jetzt nur Prolog
push 17 ; jetzt fängt die umgekehrte Parameterangabe für printf
push msg ; die Parameter werden nacheinander in printf vom Stack genommen, jedoch nicht gepoppt, was der Programmierer machen soll
call printf
mov eax, 0 ; Main muss immer 0 ausgeben
mov esp, ebp ; jetzt fängt der Epilog an
pop ebp
ret
```

9. Assemblerfunktionen in C nutzen

9.1. Assemblerdatei erstellen

mal3.s (wird mit nasm -f el32 mal2.s -o mal.o kompiliert)

```
global mal3 ; Funktionsname
mal2:
push ebp
mov ebp, esp ; Prolog
mov eax, [ebp+8] ; der von C übegebene Parameter ist zwei Adressen entfernt, wegen push ebp und dem Funktioncall
add eax, eax
mov esp, ebp : Epilog
pop ebp
ret
```

9.2. Headerdatei für C erstellen

mal2.h

```
int mal2(int c); //bloße Deklairerung der Funktion
```

9.3. Auf Assemblerfunktion in C zugreifen

main.c (am Ende wird mit gcc -m32 mal2.o main.c -o mal3 kompiliert und verlinkt)

```
#include "mal2.h" //mal.o findet der Header schon selbst, wenn er von gcc verlinkt wird
#include <stdio.h>
int main() {
printf("Zahl: %i\n", mal2(3); //gibt 10 aus
}
```

10. Reverseengineeringnotizen

X DW ? -> unititialisierte Variable X
mov byte ptr [var], 5 -> kopiere 5 in das Byte der Adresse, worauf var zeigt (in nasm ohne ptr, denn masm hat es in der Syntax)

mov eax, byte ptr 5 -> kopiere das byte 5 in das 32bit Register eax (als Lowbyte, da eax viermal größer als ein Byte ist)
mov eax,0 ist das selbe wie xor eax, eax nur schneller in der Laufzeit
Calling conventions
cdecl -> Stack für den Functioncall/Funktionsaufruf wird nicht von der Funktion gesäubert, sondern von der aufrufenden Funktion durch beispielsweise ein [esp+18]. Des Weiteren wird der Rückgabewert in eax gespeichert
stdcall -> wie cdecl nur, dass die aufgerufene Funktion den Stack mit einem leave (was dem Epilog entspricht, den gcc mit leave einfach abkürzt [der Prolog wäre übrigens enter) säubert
fastcall -> Windows tendiert dazu, dass Funktionsparameter in ECX, RDX und in R8 gespeichert werden