Zusammenfassung GI4

zur Vorlesung von Prof. Bemmerl, Sommersemester 2012, angefertigt von Martin Gritzan

Fragen oder Fehler bitte an martin.gritzan@web.de

Die Zusammenschrift behandelt hauptsächlich die Themen, die auswendig beherrscht werden müssen. Um Programmieren in Assembler zu lernen, muss man das wiederholt praktisch üben!

Kapitel 2

Stack

Der Stack wächst von hohen zu niedrigen Adressen. Das Regiser EBP zeigt dabei immer auf die Basis des aktuellen Stackframes, ESP auf die Stelle, an der gepusht/gepoppt wird.

Ein Funktionsaufruf mit CALL legt die Rücksprungadresse für den Befehlszähler auf den Stack, bevor zum nach CALL spezifizierten Label gesprungen wird.

C-Calling-Convenvtion

Die Funktionsparameter werden in UMGEKEHRTER REIHENFOLGE auf den Stack gepusht, dann wir die Funktion mit CALL aufgerufen. Der Rückgabewert wird im Register EAX erwartet.
Die Funktion legt sich mit

push ebp
mov ebp, esp

einen neuen Stackframe an und stellt hinterher (nachdem sie „ihren“ Stack leergemacht hat) mit

pop ebp
ret

den alten wieder her.
Da der Stack von hohen zu niedrigen Adressen wächst, liegen die Parameter an höheren Adressen als der Stack der aktuellen Funktion. Da CALL die Rücksprungadresse für RET auf den Stack legt, liegt der erste Parameter bei [ebp+8], die weiteren dann entsprechend bei noch größeren Offsets bezüglich des neuen Stackframes.

SSE-Register

Eine Methode zur echt parallelen Verarbeitung von Daten sind die Streaming-SIMD-Extensions des x86-Prozessors. Sie bieten die Möglichkeit, die gleiche arithmetische Operation auf 4 Gruppen von Integer- oder Single-Werten oder 2 Gruppen von Double-Werten parallel durchzuführen (quasi Vektoroperationen). Dies wird mit acht je 128bit breiten Spezialregistern realisiert, die auf die Adressen xmm0 bis xmm7 hören.

Dazu gibt es eine Reihe von besonderen Befehlen. Insbesondere muss beachtet werden, dass die SSE-Register nur miteinander oder mit dem Hauptspeicher kommunizieren können und NIEMALS mit „normalen“ Registern.

Die Mnemonics basieren auf den normalen Mnemonics, sind aber um „Anhängsel“ erweitert: Erstens, „s“ für Zugriff auf den ersten Slot im Register oder „p“ für Zugriff auf alle Slots. Zweitens ein „s“ für single oder integer oder „d“ für double. Mov-Befehle, die vom Speicher laden, haben noch ein „a“ für aligned oder „u“ für unaligned, je nachdem, ob die Speicheradresse durch 16 teilbar ist (besonders schnelles Laden). Mov-Befehle zwischen den Registern und Mov-Befehle aus den Registern heraus brauchen dieses Infix nicht.

Beispiele:

*movupd xmm0, [ebp + ecx*8]* läd die beiden Double-Werte von der spezifizierten Stelle (nicht durch 16 teilbar) in das Register xmm0.

addps xmm1, xmm5 addiert die 4 Single-Werte in xmm5 zu denen in xmm1

mulpd xmm0, xmm2 multipliziert xmm0 und xmm2 (double).

Eine Besonderheit stellt der Shuffle-Befehl dar:

shufpd xmm0, xmm0, 0x1 tauscht die beiden double-Werte aus, damit sie mit einem movsd-Befehl nacheinander rauskopiert oder mit einem addsd aufaddiert werden können.

shufps xmm0, xmm0, 0x39 schaltet rückt alle Werte einen Slot auf und eignet sich damit zum „Durchschalten“ der Werte.

Diese beiden sollte man auswendig wissen.

Die hex-Konstante legt fest, welche Werte wo hin bewegt werden. Wenn man die mal selbst berechnen muss, dann macht man das so:

xmm0:	a3	a2	a1	a0
xmm1:	b3	b2	b1	b0
	11	10	01	00

Der shufps-Befehl soll hinterher in xmm0 b2, b3, a0, a1 stehen haben. An höheren Bytes stehen immer Elemente des hinteren Operanden. Will man das nicht, muss man das Register hinterher noch mal mit sich selbst shuffeln.

Man wählt in unserem Fall die Konstante zu 0b10110001 = 0xB1

shufps xmm0, xmm1, 0xB1 b2b3a0a1

sollte dann der Shuffle-Befehl sein.

Thread-Programmierung

Eine Methode zu Ausnutzung mehrerer Prozessorkerne ist das Erstellen mehrerer Ausführungsfäden (Threads) in einem Prozess, die das OS dann auf die Kerne verteilt und parallel abarbeitet. Eine Bibliothek zum Arbeiten mit Threads nach **IEEE-POSIX-1003.c-Standard** steht in C nach dem

#include <pthread.h>

zur Verfügung. Außerdem legt man sich eine Präprozessor-Konstante mit der Anzahl der Threads an, die man höchstens erzeugen möchte. Genau oder doppelt so viele (wegen Hyperthreading) Threads wie Prozessorkerne machen Sinn, mehr nicht.

#define MAX_THREADS 8 //4 Prozessorkerne auf meinem Intel i3

Ein Thread braucht eine Einstiegsfunktion. Mit dieser Funktion werden dann neue Trheads gestartet. Die Einstiegsfunktion wird mit

```
void* einstiegsfkt(void* parameter){
    <Anweisungen>
    return 0;}
```

definiert. Als Parameter kann nur ein Integer-Wert übergeben werden, der standardmäßig als Zeiger interpretiert wird. Bei Bedarf kann man beliebig viele Werte übergeben, die über einen einzigen Zeiger dereferenziert werden müssen.

Um einen neuen Thread zu starten, benutzt man

```
pthread_create(pthread_t* thread, NULL, einstiegsfkt, void* parameter);
```

wobei pthread_t* thread vorher irgendwo deklariert werden muss.

Bevor das Programm beendet oder die Ergebnisse aller Threads ausgewertet werden können, muss auf die Beendigung aller Threads gewartet werden. Dazu setzt man

```
pthread_join(pthread_t thread);
```

ein, was natürlich in einer Schleife für jeden laufenden Thread einmal gemacht werden muss.

Manchmal kann es nötig sein, alle Threads in ihrer Ausführung an einer Stelle auf den langsamsten warten zu lassen, um sie dann mit einem gemeinsamen Zwischenergebnis wieder getrennt weiterarbeiten zu lassen. Dazu benutzt man **Barrieren**. Eine Barriere definiert man global

```
pthread_barrier_t barriere;
```

und initialisiert sie vor Start der Threads mit

```
pthread_barrier_init(&barriere, NULL, MAX_TRHEADS);
```

Dann kann man in der Threadfunktion mit dem Aufruf

```
pthread_barrier_wait(&barriere);
```

alle Threads an dieser Stelle synchronisieren.

Bei Threadprogrammen kann es sein, dass mehrere Threads mehr oder weniger gleichzeitig auf die selbe Ressource zugreifen wollen. Das kann zu undefiniertem Verhalten führen, beispielsweise, wenn Thread A eine globale Variable einliest und sie im nächsten Schritt verändert. Derweil könnte Thread B aber noch die „alte“ globale Variable einlesen. Thread A schreibt den neuen Wert, der aber sofort von Thread B wieder überschrieben wird, weil der auch einen neuen Wert erzeugt hat. So eine Codestelle heißt **kritischer Abschnitt** und muss mit einem „**Mutex**“ gesichert werden.
Man definiert global

```
pthread_mutex_t CS = PTHREAD_MUTEX_INITIALIZER;
```

und dann in der Threadfunktion vor Beginn des kritischen Abschnitts

```
pthread_mutex_lock(&CS);
```

Jetzt kann außer dem Thread, der das Mutex verriegelt hat, kein Thread mehr in diese Codestelle einsteigen. Alle anderen Thread warten, bis „unser“ Thread

```
pthread_mutex_unlock(&CS);
```

am Ende des kritischen Abschnitts aufgerufen hat.
Ein Mutex ist also so etwas wie ein Signal bei der Eisenbahn: Es kann immer nur ein Zug (Thread) im gleichen Streckenabschnitt (Codesegment) sein. Will ein weiterer Zug in den Streckenabschnitt einfahren, sieht er ein rotes Signal (verriegeltes Mutex) und muss warten, bis der vorausfahrende Zug am wiederum nächsten Streckensensor (pthread_mutex_unlock)

vorbeigefahren ist.

Kapitel 3

Assemblieren von Hand

In den Aufgaben ist meist ein Assembler-Quellcode gegeben, der in eine Objektdatei umgewandelt werden soll. Dazu folgende Vorgehensweise:

Phase 1: Quellcode durchgehen und hinter jede Zeile die Länge des zugehörigen Maschinenbefehls (in Byte) oder bei Pseudobefehlen „Pseudo“ schreiben.
Die Länge berechnet sich zu:

- ein Byte für den Opcode
- je ein Byte für jeden vorhandenen Operanden (Register/Adressbez.)
- pro Operand optional noch je 4 Byte für Offset/Immediate/Displacement

Anschließend den location counter vor jeden Befehl schreiben (Längen bis zum Anfang (!) des aktuellen Befehls aufaddieren). Pseudocode NICHT mitzählen.

ALLE ZAHLEN IN HEXADEZIMAL AUFSCHREIBEN!

Phase 2: Objektdatei erstellen
Prof. Bemmerl hat sich ein realitätsloses, vereinfachtes Objektdateiformat ausgedacht. Eine Objektdatei besteht demnach aus Datensätzen, die zeilenweise codiert sind. Jede Zeile beginnt mit einem Buchstaben, der den Inhalt der Zeile definiert. Die Zahlen werden in hex codiert. Die Ziffern in den Erläuterungen stehen für die hexadezimalen Ziffern, die dieser Block lang ist. Nicht für jeden Befehl muss eine neue Zeile angelegt werden. Mehrere gleichartige Befehle können zusammen codiert werden. Dabei muss beachtet werden, dass nie mehr als 69 Ziffern (ohne Trennzeichen) in einer Zeile stehen dürfen. „Anfang“, „Länge“ und „Adresse“ beziehen sich im Folgenden immer auf den location counter.

Die Objektdatei beginnt mit (genau) einem Header
H|<Anfangsadresse, 8>|<Länge der Datei, 8>
Dann werden die SECTIONS benannt. I.A. gibt es eine SECTION .text und eine SECTION .data mit getrennten Adressräumen
S|.text |<Anfang, 8>|<Länge, 8>|.data |<Anfang, 8>|<Länge, 8>|...
Dann kommen meist die D- und R-Datensätze. D-Datensätze geben die Position von Labels aus diesem Modul an, die öffentlich sind (global-Schlüsselwort im Quelltext), R-Datensätze benennen externe Funktionen, die benutzt werden.
Bei der Benennung von Labels/Funktionsnamen wird ein Unterstrich vorangestellt und dann nur die ersten fünf Buchstaben berücksichtigt.
D|_todo|<Adresse der ersten Anweisung nach dem Label, 8>|...
R|_exter|...
Dann kommen die Maschinenanweisungen, in T-Datensätzen. Davon gibt es normalerweise weitaus die meisten.
T|<Anfang, 8>|<Länge der Zeile, 2>|<Datensätze, 59>

Dabei werden die Datensätze durch Leerzeichen voneinander getrennt. Die T-Datensätze von .data werden einfach hinten angehängt.

Zur Erzeugung eines T-Datensatzes sind normalerweise Tabellen gegeben. Ein einzelner Maschinenbefehl hat folgendes Format
Opcode_Reg/Addr1_Reg/Addr2_Aux1_Aux2 ohne die Unterstriche.
Beispiel: *mov ebp, esp* = 0C4050, Länge 6
(0C = Opcode für mov, 4 = Register ebp, 0 = Adressmodus „Register“,
5 = Register esp, 0 = Adressmodus „Register“)
Beispiel: *mov dword [ebp-4], 0x30* = 0C4501FFFFFFFC00000030, Länge B
(0C = Opcode für mov, 4 = Register ebp, 5 = Adressmodus „indirekt mit Displacement“, 0 = don‘t care weil kein Register erforderlich, 1 = Adressmodus „immediate“, FFFFFFFC = -4, 00000030 = 0x30)

Nach den T-Datensätzen kommen die M-Datensätze, die etwas komplizierter zu verstehen sind. Sie geben an, welche Werte nach dem Laden der Objektdatei noch angepasst werden müssen.
M|<Adresse des anzupassenden Wertes, 8>|<Länge des Wertes, 4>|<+ oder ->|<Name des Labels, um dessen logische Adresse der Wert modifiziert werden muss>|...
Und zum Schluss noch ein
E
damit der Lader weiß, dass die Datei zu Ende ist.

Kapitel 4

Paging

Eine virtuelle Speicherverwaltung ist in vielerlei Hinsicht nützlich. Eine Möglichkeit, physikalischen und logischen Speicher zu trennen, ist das Paging. Die Speicherverwaltung wird von der MMU (memory management unit) vorgenommen.

Der physikalische Speicher wird in logische Adressräume, sogenannte „Frames“ (Rahmen) fester Größe eingeteilt. Die zugehörigen logischen Adressblöcke heißen „Pages“ (Seiten).
Um RAM zu sparen, können nicht benötigte Seiten auf den Swap (UNIX-Derivate) bzw. die Auslagerungsdatei (Windoof) ausgelagert werden. Es muss also nicht für jede Seite immer ein Frame reserviert sein.
Alle Seiten werden in der Seitentabelle von der MMU verwaltet. Länger nicht benötigte Seiten werden ausgelagert und in der Seitentabelle entsprechend gekennzeichnet (“gesperrt“).
Wenn eine solche Seite aufgerufen wird, erkennt die MMU das und erzeugt einen Seitenfehler. Das OS fängt den Seitenfehler ab, unterbricht die Ausführung des Programms, das den Fehler verursacht hat, und lässt die geforderte Seite in ein freies Frame einlagern. Dann wird das Programm fortgesetzt. Das Programm „merkt“ von alldem nichts.

Beim Laden eines neuen Programms wird erstmal nur eine Reihe von Seiten angelegt, die erst bei der ersten Verwendung eingelagert werden. So wird Zeit gespart.

Bindender Lader

Phase 1: Alle zu ladenden Module werden hintereinandergelegt.
Dann werden die logischen Adressen in den H, D und S-Datensätzen ausgerechnet und angepasst.
Diese Adressen bilden die ESTAB-Liste.

Beachte: Die .data-Segmente werden hinter das letzte .text-Segment gelegt.

PROGADDR = erste zum Programm gehörende Adresse
ECEXADDR = Adresse der ersten Maschinenanweisung, die ausgeführt werden soll. (meist = PROGADDR)

Phase 2: Alle absoluten Verweise werden mithilfe der M-Datensätze und der ESTAB angepasst.
Dann wird das fertig gebundene Programm in den RAM geladen.

Dynamisches Binden mit Shared Libraries

Zuerst wird nur das Hauptprogramm geladen. Für dieses Modul werden alle externen Referenzen in eine GOT (global offset table) eingetragen, die im Datensegment des Moduls abgelegt wird. Jeder Referenz wird eine (global eindeutige) ID zugeordnet.

Dann durchsucht der Lader die (fertig modifizierten) D-Datensätze aller bisher geladenen Module nach Kandidaten für die Referenzen. Wenn er einen Kandidaten findet, trägt er in die GOT des neu geladenen Moduls neben der ID die zugehörige logische Adresse ein.

Ist noch kein Kandidat für die Referenz geladen, versucht der Lader, ein Modul mit der referenzierten Funktion drin zu laden. Klappt das nicht, scheitert das dynamische Binden.

Funktionsaufruf bei dynamisch gebundenen Modulen

Von einer jeden GOT ist die relative Position zum Anfang „ihres“ Moduls bekannt. Wenn also eine externe Funktion aufgerufen werden soll, muss in der GOT die logische Adresse der Funktion nachgeschlagen werden.
Dazu wird zum aktuellen Befehlszäherstand (GOT_offset - aktuelle_Pos_im_Modul) hinzuaddiert. Dann wird linear nach der ID der Funktion gesucht und dann dorthin gesprungen.

Anmerkung: Das Auslesen des Befehlszählregisters EIP ist verboten. Man behilft sich mit

```
    call LABEL
LABEL:
    pop ebx
```

was die gleiche Funktionalität bietet wie *mov ebx, eip*. Diese Methode dauert ihre Zeit, daher sollte man in zeitkritischen Abschnitten auf Funktionen aus anderen Modulen verzichten.

Dynamisches Binden mit PLT und lazy binder

Bei Binden und Laden wird im Prinzip wie beim normalen dyn. Binden vorgegangen, nur werden hier die Abhängigkeiten zwischen den Modulen erst zur Laufzeit des Programms (und nicht schon direkt nach dem Laden) aufgelöst. Damit geht das Starten von Programmen, die umfangreiche Bibliotheken benutzen können, aber nicht alles davon direkt am Anfang brauchen, wesentlich schneller.

In die GOT wird statt der Adresse einer externen Funktion (die unbekannt bleibt) die Adresse einer kleinen internen Ersatzfunktion gelegt.
Die Erstatzfunktion für jede externe Funktion steht in der „procedure lookup table“ PLT, die im Codesegment des Moduls abgelegt wird.
Diese Ersatzfunktion pusht ihre ID auf den Stack und ruft den „lazy binder“ auf.

Der lazy binder „sucht“ dann die zur ID auf dem Stack passende Funktion (die eigentlich aufgerufen werden sollte) und springt dorthin. Ab da weiter wie beim dyn. Binden ohnte PLT.

Beispiel: Aufruf der externen Funktion runthis.

irgendwo im Code von main steht

```
...
call runthis_PLT
nop
...
```

in der PLT steht
runthis_PLT:
*jmp [ebx + runthis_GOT*4] ;* in der GOT: Adr. von runthis_DUMMY

und bei den Erstazfunktionen ist die folgende dabei:
runthis_DUMMY:

```
push runthis_PLT_ID
jmp lazy_binder
```

Der lazy binder springt zu runthis. runthis terminiert mit RET und nimmt als Rücksprungadresse wieder die von dem Befehl nach *call runthis_PLT* an, in diesem Fall das NOP.
Damit das klappt, müssen alle anderen „Aufrufe“ unterwegs mit JMP gemacht werden und nicht mit CALL!

Der lazy binder schreibt die gefundene Startadresse von runthis in die GOT von main. Damit wird beim nächsten Aufruf *call runthis_PLT* von der GOT aus ohne den Umweg über die Dummyfunktion direkt zu runthis gesprungen.

Kapitel 5

Verarbeitungstechniken

Wie kommt man vom Code zum laufenden Programm?

Zweistufige Verarbeitung: Ein Compiler übersetzt den Code (z.B. Assemblercode), der dann auf einem Hardware-Prozessor (z.B. µC) interpretiert wird.

Dreistufige Verarbeitung: Ein Compiler übersetzt den Code in eine andere Sprache, die von einem weiteren Compiler in Maschinencode umgewandelt wird. Dieser wird dann auf der Hardware interpretiert. Paradebeispiel: Ein C-Programm für den µC, das erst in Assemblercode umgewandelt wird, der dann als Maschinencode auf den µC übertragen wird.

Übersetzung und Interpretation: Ein Compiler übersetzt den Quellcode in einen anderen (nicht-Assambler-) Code, der dann von einem Softwareinterpreter interpretiert wird. Beispiel: Java-Programme für die Java-VM unter Windows.

Übersetzung und Laufzeitsystem: Häufigster Anwendungsfall. Der Quellcode wird teilweise in Maschinencode übersetzt (der dann auf dem Hardwareprozessor läuft) und teilweise in eine von einem Laufzeitsystem (=Betriebssystem) interpretierbare Sprache übersetzt. Das OS stellt dann kompliziertere Funktionalitäten zur Verfügung, die der Befehlssatz des Prozessors nicht direkt hergibt.

Emulation: Ein Maschinencode, der für Maschine A gedacht ist, wird von einem Softwareinterpreter interpretiert, der selbst auf Maschine B läuft. Beispiel: Nintendo64-Emulator für Windows auf der x86-Maschine.

Makroprozessor

Der Makroprozessor expandiert Makros im Quelltext. Vielen Compilern ist ein Makroprozessor vorgeschaltet, ein prominentes Beispiel ist der C/C++ Präprozessor.

Ein vereinfachter Makroprozessor ist klausurrelevant.

Ein Makro hat die beispielhafte Form

```
&DEFINE &ADD (&Par1, &Par2) &AS
# &Par1 + &Par2 #;
```

Dieses Makro addiert die beiden Parameter. Die Expansion des Makros läuft wie folgt ab, die Phasen 1 und zwei laufen quasi parallel, insbesondere, wenn es um die Erstellung der MD-Einträge geht.

Phase 1, lexikalische Analyse: Der „Scanner“ unterteilt das gesamte Programm in „Token“, also Abschnitte entsprechend ihrer Funktion. Der behandelte Makroprozessor unterscheidet

Namen in Makros (&ADD, &Par1...)
Schlüsselwörter für Makros (&DEFINE, &AS...)
Terminalzeichen wie # oder (oder ;
Quelltext, der nicht zum Makro gehört
Quelltext, der zur Makrodefinition gehört und expandiert werden muss
überflüssige Leerzeichen

voneinander.

Der Scanner legt die Programmtabelle **PT** an, die angibt, was an welcher Stelle gemacht werden muss. Tabellenaufbau:
Tätigkeit | Anfangspos | Länge | Verweis auf MD (nur bei Tätigkeit M)
Tätigkeiten sind

K: Kopieren von Zeichenketten. Wird auf den überwiegenden Teil des Quellcodes angewendet.

M: Makroexpansion. Eine Stelle, an der ein Makro aufgerufen wird und expandiert werden muss. Der Verweis auf MD gibt einen Hinweis darauf, wo die Makrodefinition zu finden ist. Anfangsposition und Länge werden nicht spezifiziert.

P: Parameterdefinition. Folgen auf einen M-Eintrag, wenn das zugehörige Makro Parameter besitzt. Die Einträge definieren dann, was als Parameter benutzt werden soll.

Makrodefinitionen werden nicht in die PT aufgenommen, da sie nach der Expansion nicht mehr benötigt werden und entfallen.

Phase 2, syntaktische Analyse: Hier werden die Makrodefinitionstabelle MD und die Makrorumpftabelle MR erstellt.

Makrodefinitionen werden in die **MD** eingetragen. Ihr Aufbau ist
Index | Name | Anzahl Parameter | Verweis auf MR | Anzahl Token
Die Makrorumpftabelle enthält dann Informationen darüber, was bei einem Aufruf einer Zeile in der MD eigentlich passieren soll. Der Aufbau der **MR** ist
Index | Tätigkeit | Anfangspos | Länge | Parameter
Tätigkeiten sind

K: Kopieren eines Teils aus dem Makrorumpf. Parameternamen werden nicht mit kopiert, sondern umgangen und mit einem E-Eintrag behandelt.

E: Einsetzen eines aktuellen Parameters. Anfangspos und Länge entfallen. In die Parameter-Spalte wird die Nummer des Parameters lt. Definition eingetragen (und nicht etwa der symbolische Name).

Phase 3, Programmgenerierung: Jetzt wird anhand der vorhergehenden Tabellen makrofreier Code erzeugt.
Beispielsweise wird *&ADD(a, (4+3b));* durch *a + (4+3b);* ersetzt. Die PT wird durchgesehen und alle Anweisungen befolgt. Bei M-Einträgen werden die Makros entsprechend MD und MR expandiert.

Compiler

dienen der Übersetzung zwischen zwei Sprachen. Der Compiler geht erstmal wie ein Makroprozessor vor und analysiert den Quellcode mit einem Scanner. Dabei werden folgende Datenstrukturen erzeugt:

uniform symbol table UST: Enthält Symbole (Variablennamen, Konstantennamen, Sprungmarken) und die zugehörigen Zahlen / Adressen.

identifiers and names IDN: Verwaltet alle Bezeichner innerhalb des Programms.

literals LIT: Verwaltet alle Konstanten und ihre Werte.

unknown tokens ANY: Hier wird alles abgelegt, womit der Scanner nichts anfangen kann. Nötig für Fehleranalyse.

Zusätzlich kennt der Scanner noch die

terminals TRM: Liste aller Terminalsymbole wie ; oder : oder (

und die

key KEY: Liste aller Schlüsselworte der Quellsprache, wie *while* oder *if* oder *int* und so weiter.

Im nächsten Schritt erzeugt der „Parser“ anhand der UST, LIT und IDN und der ihm bekannten Grammatikregeln (in Backus-Naur-Form gegeben) für die Quellsprache einen Syntaxbaum, der durch die Matrixdarstellung repräsentiert wird. Eine solche Matrix für das Programm

*kosten = rate * (ende - anfang / 12) + 50;*

wird so aussehen:

Zeile	**1. Operand**	**2. Operand**	**Operator**
0	anfang	12	/
1	ende	M(0)	-
2	rate	M(1)	*
3	M(2)	50	+
4	kosten	M(3)	=

Dabei muss die logische Rangfolge von Operationen bekannt sein. Sie wird in einer Präzedenzmatrix abgelegt. Auf den Achsen der Matrix sind alle Operatoren aufgezählt, die Zellen geben mit Relationszeichen die Rangfolge oder mit „0“ an, dass eine Verkettung der Operatoren sinnlos / unzulässig ist.

Bevor jetzt Zielcode (Assemblercode) erzeugt werden kann, muss erstmal der Speicher für die Variablen und Konstanten angelegt werden. Für die Einträge der Tabellen IDN und LIT und die Zwischenergebnisse aus der Auswertungsmatrix (z.B. M(0)) müssen im Datensegment Speicherplätze reserviert werden.
Ein Zwischenergebnis dazu ist die Tabelle „Data“, die einfach alle nötigen Variablen enthält. Uninitialisierter Speicher kann in Assembler mit RESx (RESD für Integer) angefordert werden. Die einzelnen Tabellen werden geschlossen als Arrays angefordert.

Anschließend wird die Matrix zeilenweise abgearbeitet und der Inhalt in Assembler-Anweisungen (oder allgemein Anweisungen in der Zielsprache) übersetzt.

Einige Compiler können Optimierungen durchführen:
Maschinenunabhängig:

> Mehrfach benötigte Zwischenergebnisse werden nur einmal berechnet und dann im Speicher abgelegt (mehrfache Nutzung einer Matrixzeile als Operand in anderen Matrixzeilen).
> Konstanten („3+4“) können schon bei der lexikalischen Analyse berechnet und ersetzt werden.
Maschinenabhängig:
> Ausnutzung effizienterer Befehle, z.B. *xor eax, eax* statt *mov eax, 0*
> Vermeidung unnötiger Speicherzugriffe
> Ausnutzung der SSE-Fähigkeit des Zielprozessors (bei Übersetzung in Assembler).