

Systemprogrammierung

Rechnerorganisation: Assemblersprache

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

Ergänzende Materialien

Gliederung

- 1 Grundlagen
 - Abstraktionsebene
 - Syntax
- 2 Pseudobefehle
 - Überblick
 - Textabschnitte
 - Datenabschnitte
- 3 Maschinenbefehle
 - Überblick
 - Entwurfsmuster
- 4 Zusammenfassung

Vorbemerkung

Lehrziel

Vermittlung grundlegender Kenntnisse einer Assemblersprache, um die darin formulierten Programme (der Vorlesung) besser verstehen zu können

- es wird strikt „von oben nach unten“ (engl. *top down*) vorgegangen
- Ausgangspunkt bilden Konstrukte der Programmiersprache C
- betrachtet wird deren Repräsentation auf Assemblersprachenebene
- dazu wird der Kompilervorgang vor dem Assembliervorgang beendet
 - gcc `-S` `-fomit-frame-pointer`, plus `-O6`, `-Os` oder `-O0`
 - Version 4.3.2 (Debian 4.3.2-1.1)

Kein Lehrziel

Vermittlung von Assemblersprache von Grund auf, um Programme für die Assemblersprachenebene manuell entwickeln zu können

- nur in vereinzelten Fällen sind solche Programmierkenntnisse nötig

Einleitung

Assemblersprache reflektiert im Wesentlichen das **Programmiermodell** der Befehlssatzebene

- ① Befehls- und Registersatz des (realen) Prozessors [GRAO]
- ② Adressierungsarten für den Operandenzugriff [GRAO]

Maschinenbefehle und Prozessorregister werden in für Menschen besser lesbare Kürzel repräsentiert

Mne|mo|nik (von gr. *mnēmoniká* „Gedächtnis“, engl. *mnemonic*)

- „linguistische Gedächtnisstütze“, **symbolischer Maschinenkode**
- in **Maschinensprache** (Binärkode) zu übersetzende Konstrukte

Generierung von Maschinenprogrammen bedingt weitere Sprachkonstrukte, die als **Pseudobefehle** realisiert sind

as|sem|blier|en zusammenfügen, versammeln, vereinigen

- Informatik: Objektmodule erzeugen und zu einem Lademodul ~

Strukturelemente von Assemblersprache

Leerzeichen, Tabulatoren, Zeilenumbrüche (engl. *whitespace*)

- „syntaktischer Zucker“: Trennung von Symbolen, Strukturierung

Kommentare sind für Assemblersprachenprogramme besonders wichtig!

- (i) mehrzeilig (wie aus C bekannt): von `/*` bis `*/`
- (ii) einzeilig (x86_64 GNU/Linux)¹: von `#` bis Zeilenende

Symbole sind Folgen kasus-sensitiver Zeichen ohne Längenbeschränkung

- Buchstaben (Groß-/Kleinschreibung), Ziffern, `'_'`, `'.'` und `'$'`

Anweisungen setzen sich aus zwei Klassen von Befehlen zusammen:

- (i) Pseudobefehl, der den Assemblier- und Bindevorgang steuert
- (ii) Maschinenbefehl, der die Ausführungsvorgänge in der CPU steuert

Direktwerte (engl. *constants*)

- Zahlen, deren Werte kontextfrei durch bloße Ansicht feststellbar sind

¹Abhängig vom Zielprozessor [3]. Für andere Plattformen: `;`, `!`, `|`, `--`.

Vorverarbeitung von Programmtext

Funktion vom **Vorprozessor** (engl. *preprocessor*) von `as(1)`:

- Leerraum (typographischer Weißraum, engl. *whitespace*) reduzieren
 - nur ein Leer-/Tabulatorzeichen vor einem Schlüsselwort hinterlassen
 - restlichen Leerraum einer Zeile in ein einzelnes Leerzeichen umwandeln
- Kommentare entfernen
 - einzeilig: in einzelnes Leerzeichen umwandeln
 - mehrzeilig: in entsprechende Anzahl von Zeilenumbrüchen umwandeln
- Zeichenkonstante in numerische Werte (Zahlen) umwandeln

Makroverarbeitung, Dateieinschluss oder andere von `cpp(1)` her bekannte Konzepte zur Vorverarbeitung sind (in der Form) nicht möglich, aber:

- `.include` erlaubt den Einschluss von Assemblersprachentext
- Dateisuffix `.S` aktiviert in `gcc(1)` den CPP-Arbeitsgang
 - ebenso kann direkt `cpp(1)` auf `.S`-Dateien angewendet werden

ASM für as [3] in EBNF [5]

ASM = { *Kommentar* | *Direktive* };

Kommentar = '#', *Prosa* – *Zeilenende*, *Zeilenende* | ('/*', *Prosa*, '*/');

Prosa = (* Text aus dem ASCII-Zeichensatz *);

Zeilenende = '\n';

Direktive = { *Kennsatz* }, { *Anweisung* };

Kennsatz = *Symbol*, ':';

Symbol = { *Zeichen* } – (* leere Menge *);

Zeichen = *Buchstabe* | *Ziffer* | *Sonderzeichen*;

Buchstabe = 'a' | (* ... *) | 'z' | 'A' | (* ... *) | 'Z';

Ziffer = '0' | (* ... *) | '9';

Sonderzeichen = '_' | '.' | '\$';

Anweisung = { ';' }, { *Leerraum* }, *Befehl*;

Befehl = *Pseudobefehl* | *Maschinenbefehl*;

Pseudobefehl = (* Liste der Pseudobefehle *);

Maschinenbefehl = (* Liste der Maschinenbefehle *);

Gliederung

- 1 Grundlagen
 - Abstraktionsebene
 - Syntax
- 2 Pseudobefehle
 - Überblick
 - Textabschnitte
 - Datenabschnitte
- 3 Maschinenbefehle
 - Überblick
 - Entwurfsmuster
- 4 Zusammenfassung

Auswahl von as-Pseudobefehlen [3] in EBNF

Von gcc(1) zu den Beispielen abgesetzte Pseudobefehle

```
Segment  = Text | Daten | BSS;  
Text     = ('.text', [Ziffer]) | ('.section', Name, ['.', Merker, ['.', Typ]]);  
Daten    = '.data', [Ziffer];  
BSS     = ('.comm' | '.lcomm'), Symbol, ',', Länge, ['.', Abgleich];  
Ausrichtung = '.align' | '.p2align', Anpassung;  
Anpassung  = Abgleich, ['.', [Füllwert], ['.', Übersprungsgröße]];  
Sichtbarkeit = ('.globl' | '.global'), Symbol;  
Wert      = ('.long', Ausdruck) | ('.string', '','', Zeichenfolge, '','',);  
  
Zugabe    = Assemblat, COFF, Aufkleber;a  
Assemblat = '.file', ['','',], Name, ['','',];  
COFF     = ('.size', Symbol, ',', Länge) | ('.type', Symbol, ',', Typ);  
Aufkleber = '.ident', '','', Kennzeichen, '','',;
```

^aDiese Pseudobefehle werden aus Gründen der Kompatibilität mit alten oder besonderen Bindern abgesetzt, sie sind für unsere Zwecke ohne weiteren Belang.

Leeres Festland: Mehr Schein als Sein...

-O6

```
main () {}
```

`.text` Textsegment

`.p2align` Adresszähler richten

- `xxx...xxx0000`
- Füllwert (NOP)
- max. 15 Bytes überspringen

`.globl` nach außen sichtbar

`main:` Adresszählermarke

- engl. *label*

```
.file "main.c"
.text
.p2align 4,,15
.globl main
.type main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ecx
    popl    %ecx
    leal    -4(%ecx), %esp
    ret
.size main, .-main
.ident "GCC: (Debian 4.3.2-1.1) 4.3.2"
.section .note.GNU-stack,"",@progbits
```

- Maschinenbefehle zwischen `main:` und einschließlich `ret` $\equiv$ `{}`

NOP: Nulloperation (engl. *no-operation*)

Maschinenbefehl, der funktional nichts tut, sich aber nichtfunktional (Speicherplatzbedarf, Ausführungszeit, Energieverbrauch) auswirkt

- lässt den Prozessorstaus (Speicher- und Registerinhalte) invariant
 - Ausnahme ist der Befehlszähler (engl. *program counter*, PC)
- benötigt bei Ausführung eine bestimmte Anzahl von Taktzyklen

Verwendungszweck

- zur **Ausrichtung** (engl. *alignment*) von Programmtextadressen
- für zeitliche Koordinierung und Regulierung (engl. *timing*)
- um Gefährdungen in der Befehlsbearbeitung^a zuvorzukommen
- als Verzögerungsfenster bei Sprüngen (engl. *branch delay slot*)
- als Platzhalter für später einzufügende „aktive“ Maschinenbefehle

^aGenauer: innerhalb der CPU-Pipeline.

NOP: Beispiele für x86 [2, 6]

# Bytes	Maschinenbefehle ²		Bedeutung
	Mnemonic	Operanden	
1	xchg	%ax, %ax	nop-Alias, siehe 2-Byte-xchg
2	mov	R_s, R_d	$R_d = R_s$, s gleich d
2	xchg	R_s, R_d	$x = R_d, R_d = R_s, R_s = x$, s gleich d
3	lea	$0(R_s), R_d$	$R_d = R_s + 0$, s gleich d
4	shl	$0, R$	$R = R \ll 0$
5	shrd	$R_s, 0, R_d$	$R_d = (R_s \rightarrow R_d) \gg 0$
6	leal	$0(R_s), R_d$	siehe 2-Byte-lea

Anmerkung

- die x86-Architektur liefert weitere Maschinenbefehle dieser Art
- wichtig für NOP ist ihre **funktional nebeneffektfreie Ausführung**

²Die erste Hälfte nennt Beispiele für den 16/32-Bit-Betrieb, die zweite nur für den 32-Bit-Betrieb (also nicht für Intel-Prozessoren älter als 80386).

Platzhalter globaler Programmvariablen

-O6 / -Os

```
char c;
char *cp;
float f;

short int si;
long int li;
long long int lli;
```

```
unsigned int i = 42;
char *s = "Reihung";
double d = 3.14159;
```

```
.globl s
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "Reihung"
.data
.align 4
.type s, @object
.size s, 4
s:
.long .LC0
```

```
.globl i
.data
.align 4
.type i, @object
.size i, 4
i:
.long 42
```

```
.globl d
.align 8
.type d, @object
.size d, 8
d:
.long -266631570
.long 1074340345
```

```
.comm c,1,1
.comm cp,4,4
.comm f,4,4
.comm si,2,2
.comm li,4,4
.comm lli,8,8
```

.section schreibgeschütztes (engl. *read-only*) Datensegment $\mapsto$ COFF

.data Datensegment

.align Ausrichtung: Abgleich
des Adresszählers auf
 n -Byte Grenze, $n = 4, 8$

.long Zahlenwert absetzen

.string Zeichenfolge (*C string*)

.comm BSS: Symbol, Länge,
Ausrichtung (s. **.align**)

Gliederung

- 1 Grundlagen
 - Abstraktionsebene
 - Syntax
- 2 Pseudobefehle
 - Überblick
 - Textabschnitte
 - Datenabschnitte
- 3 **Maschinenbefehle**
 - Überblick
 - Entwurfsmuster
- 4 Zusammenfassung

Ausgewählte Programmbeispiele mit Systembezug

- ① Wertezuweisung an `errno(3)`
 - Übernahme des Fehlerkodes von einem Systemaufruf
- ② Systemaufrufzustellung (engl. *system call dispatching*)
 - Verzweigung über eine Sprungtabelle
- ③ Möglichkeiten einer Wettlaufsituation (engl. *race condition*)
 - Inkrementieren von Zählerwerten
- ④ gezielter Einsatz von Assemblersprache zur Leistungsverbesserung
 - Kopieren von Speicherbereichen (`memcpy(3)`)

Entwurfsmuster (engl. *design patterns*)

- bewährte Lösungsschablonen für wiederkehrende Softwareprobleme
- im gegebenen Fall:
 - (a) typische Problemstellungen der Systemprogrammierung allgemein
 - (b) in der Vorlesung verwendete Programmfragmente im Speziellen

Übernahme des Fehlerkodes vom Systemaufruf

-06

```
extern int errno;

#define ERRNO_HIGH 4095

extern inline
int sys_call (int scn) {
    int scr;

    asm volatile ("int $0x80"
        : "=a" (scr) : "a" (scn));

    if (scr >= -ERRNO_HIGH) {
        errno = -scr;
        return -1;
    }

    return scr;
}
```

```
int getpid () { return sys_call(20); }
```

```
.text
.p2align 4,,15
.globl getpid
.type getpid, @function
getpid:
    movl    $20, %eax
    int     $0x80
    cmpl    $-4095, %eax
    jge     .L5
    rep
    ret
    .p2align 4,,7
    .p2align 3
.L5:
    negl    %eax
    movl    %eax, errno
    movl    $-1, %eax
    ret
```

Erklärung folgt...

Übernahme des Fehlercodes vom Systemaufruf (Forts.)

asm **einfügender Assembler** (engl. *in-line assembler*) [4]

- zum Absetzen des Maschinenbefehls eines Systemaufrufs
 - `int $0x80` im Falle von Linux
- die Operanden für diesen Befehl in Register `%eax` halten:
`Ausgabespezifikation` $\mapsto$: `"=a"` (`scr`)
`Eingabespezifikation` $\mapsto$: `"a"` (`scn`)
- `volatile` verhindert Befehlsumplanung des Kompilers

`%eax` Akkumulator (Prozessorregister), enthält hier zweierlei Dinge:

- (a) Systemaufrufnummer (engl. *system call number*): `scn`
- (b) Systemaufrufergebnis (engl. *system call result*): `scr`

rep ret Doppelbytebefehl zum nahen Prozedurrücksprung

- Einzelbytevariante verhindert Sprungvorhersage [1, S. 128]
 - mit Schalter `-O6` wollen wir „performanten“ Kode haben!
- Problem: bedingter Sprung (z.B. `jge`) gefolgt von `ret`

Verzweigung über eine Sprungtabelle

-Os

```
#define NCALLS 190 /* Linux: #1..190 */
#define ENOSYS 38

extern int (*sys_call_table[])();

extern inline
void sys_emit (int scn, word_t *scr) {
    if (scn <= NCALLS)
        *scr = (*sys_call_table[scn})();
    else
        *scr = -ENOSYS;
}
```

```
.text
.globl system_call
.type system_call, @function
system_call:
    ...
    cmpl $190, %eax
    jg    .L2
    call *sys_call_table(,%eax,4)
    movl %eax, 24(%esp)
    jmp  .L3
.L2:
    movl $-38, 24(%esp)
.L3:
    ...
```

Einbettung umseitig...

`*sys_call_table(,%eax,4)` indirekt-indizierte Adressierung

- Adresse der Sprungzieladresse: `sys_call_table + (%eax * 4)`
 - die Tabelleneinträge sind Funktionszeiger von vier Bytes Länge
- Dereferenzieren ('*'): Wert als **effektive Sprungzieladresse** lesen

Verzweigung über eine Sprungtabelle (Forts.)

Systemaufrufzustellung

```
typedef long word_t;

#define PSR_EAX 6

void __attribute__((interrupt)) system_call () {
    register word_t *psr asm("esp");    /* assume psr in register %esp */
    register int scn      asm("eax");    /* assume scn in register %eax */

    asm volatile ("...");                /* intentionally left invalid! */

    sys_emit(scn, &psr[PSR_EAX]);

    asm volatile ("...");                /* intentionally left invalid! */
}
```

`__attribute__((interrupt))` Funktionsattribut

- Programmtext passend für Unterbrechungsbehandlung auslegen
- kompletten **Prozessorstatus invariant halten**, `iret` anstatt `ret`

Inkrementieren von Zählerwerten

```
extern int foo;
```

```
void action () {
    foo++;
}
```

-O6

```
.p2align 4,,15
action:
    addl $1, foo
    ret
```

-Os

```
action:
    incl foo
    ret
```

-O0

```
action:
    movl foo, %eax
    addl $1, %eax
    movl %eax, foo
    ret
```

- Schalter -O0 übersetzt foo++ nicht mehr in eine Elementaroperation
- die Ausführung von i++ ist (plötzlich) kritisch im Überlappungsfall
- Deklaration von foo als **volatile int** bewirkt den gleichen Effekt:



-O6

```
.p2align 4,,15
action:
    movl foo, %eax
    addl $1, %eax
    movl %eax, foo
    ret
```

-Os

-O0

```
action:
    movl foo, %eax
    incl %eax
    movl %eax, foo
    ret
```

Kopieren von Speicherbereichen: Naive Fassung

-Os

```
typedef unsigned long size_t;

void *memcpy (void *dest, const void *src, size_t n) {
    char *from = (char*)src, *to = (char*)dest;

    while (n-- > 0)
        *to++ = *from++;

    return dest;
}
```

Funktionsprolog

```
memcpy:
    pushl %esi
    xorl  %edx, %edx
    pushl %ebx
    movl  12(%esp), %ebx
    movl  16(%esp), %esi
    movl  20(%esp), %ecx
    jmp   .L2
```

Kopierschleife

```
.L3:
    movb  (%esi,%edx), %al
    decl  %ecx
    movb  %al, (%ebx,%edx)
    incl  %edx
.L2:
    testl %ecx, %ecx
    jne   .L3
```

Funktionsepilog

```
    movl  %ebx, %eax
    popl  %ebx
    popl  %esi
    ret
```

Kopieren von Speicherbereichen: Naive Fassung (Forts.)

Implementierung von `memcpy` (S. 21) mit gravierenden Unzulänglichkeiten:

- ① zeichenweises (`char`) Kopieren, kein wortweises
 - `short` halbiert, `long` viertelt, `long long` achtelt den Aufwand!
- ② unausgeschöpfte Optimierungsmöglichkeiten des Kompilierers
 - Schalter `-O6` lässt `gcc(1)` die Semantik der Schleife erkennen
 - das Kompilat enthält Operationen für wortweises (`long`) kopieren
 - abgesetzt wird u.a. Code zur Anpassung von Zeiger- und Zählerwerten
 - Folge: 102 Zeilen generierter Programmtext in Assemblersprache
- ③ unausgeschöpfte Maschinenbefehle des (realen) Prozessors
 - der x86 hat **Spezialbefehle** zum Kopieren ganzer Speicherbereiche
 - Wiederholungspräfix (`rep`) für `movs`-Befehle

Spezialbefehle des Prozessors „abrufen“

- (a) in Reihe (engl. *in-line*) mit Anweisungen der Hochsprache einfügen
- (b) auf Techniken der prozeduralen Abstraktion zurückgreifen:
 - zu optimierende Anweisungen der Hochsprache **herausfaktorisieren**
 - herausfaktorierte Anweisungen in Assemblersprache **handkodieren**

Kopieren von Speicherbereichen: Optimierte Fassung (1)

```
typedef unsigned long size_t;

void *memcpy (void *dest, const void *src, size_t n) {
    void *aux = dest;          // save return value
    size_t num;

    num = n / 4;                // double-word moves

    asm volatile ("cld");       // incremental...

    asm volatile ("rep movsd"
        : "=c" (num), "=D" (dest), "=S" (src)
        : "c" (num), "D" (dest), "S" (src)
        : "memory");

    num = n & (4 - 1);          // byte moves

    asm volatile ("rep movsb"
        : "=c" (num), "=D" (dest), "=S" (src)
        : "c" (num), "D" (dest), "S" (src)
        : "memory");

    return aux;
}
```

-O6

```
.text
.p2align 4,,15
.globl memcpy
memcpy:
    subl $16, %esp
    movl %ebp, 12(%esp)
    movl 28(%esp), %ebp
    movl %ebx, (%esp)
    movl 20(%esp), %edx
    movl %esi, 4(%esp)
    movl %edi, 8(%esp)
    movl %ebp, %eax
    shrl $2, %eax
    cld
    movl 24(%esp), %esi
    movl %eax, %ecx
    movl %edx, %edi
    rep movsd
    andl $3, %ebp
    movl %ebp, %ecx
    rep movsb
    movl %edx, %eax
    movl (%esp), %ebx
    movl 4(%esp), %esi
    movl 8(%esp), %edi
    movl 12(%esp), %ebp
    addl $16, %esp
    ret
```

-Os

```
.text
.globl memcpy
memcpy:
    pushl %edi
    pushl %esi
    subl $12, %esp
    movl 32(%esp), %edx
    movl %edx, %eax
    shrl $2, %eax
    cld
    movl 28(%esp), %esi
    movl %eax, %ecx
    movl 24(%esp), %edi
    rep movsd
    andl $3, %edx
    movl %edi, 4(%esp)
    movl %edx, %ecx
    rep movsb
    movl 24(%esp), %eax
    addl $12, %esp
    popl %esi
    popl %edi
    ret
```

- Abstraktion durchbrechen:
 - einfügender Assemblierer [4]

Kopieren von Speicherbereichen: Optimierte Fassung (2)

```
.text
.p2align 4,,15
.globl memcpy
memcpy:
    movl %edi, %eax      # save %edi          (%eax: volatile register)
    movl %esi, %edx      # save %esi          (%edx: volatile register)
    movl 4(%esp), %edi    # get 1st parameter: dest
    movl 8(%esp), %esi    # get 2nd parameter: src
    movl 12(%esp), %ecx   # get 3rd parameter: n    (%ecx: volatile register)
    shrl $2, %ecx        # calculate for double-word moves
    cld                  # automatically increment %ecx, %esi, %edi
    rep movsd
    movl 12(%esp), %ecx   # re-read original n
    andl $3, %ecx        # calculate for byte moves
    rep movsb
    movl %edx, %esi      # restore %esi
    movl %eax, %edi      # restore %edi
    movl 4(%esp), %eax    # pass return value
    ret
```

Handkodierte in Assemblersprache

Kopieren von Speicherbereichen: Ergänzungen

zu S. 23 asm-Konstrukt

- Aus- und Eingabespezifikationen weisen den Variablen Registern zu:
 - **C** lokale Variable `num` $\mapsto$ Register `%ecx`
 - **D** aktueller Parameter `dest` $\mapsto$ Register `%edi`
 - **S** aktueller Parameter `src` $\mapsto$ Register `%esi`
- sie setzen damit die Operanden für die „rep movs“-Operation auf
 - der rep-Präfix bewirkt die `%ecx`-malige Ausführung des `movs`-Befehls

zu S. 24 „volatile register“ (engl. *volatile*: flüchtig)

- gcc(1) unterscheidet zwischen flüchtige und nichtflüchtige Register:
 - **flüchtige** $\sim$ sind inhaltsvariant bei Prozeduraufrufen
 - `%eax`, `%ecx`, `%edx` sind in Prozeduren frei verfügbar
 - **nichtflüchtige** $\sim$ sind inhaltsinvariant bei Prozeduraufrufen
 - `%ebx`, `%ebp`, `%esi`, `%edi` sind **nicht** frei verfügbar
- hier: nichtflüchtige Register `%edi` und `%esi` in flüchtige sichern

Gliederung

- 1 Grundlagen
 - Abstraktionsebene
 - Syntax
- 2 Pseudobefehle
 - Überblick
 - Textabschnitte
 - Datenabschnitte
- 3 Maschinenbefehle
 - Überblick
 - Entwurfsmuster
- 4 Zusammenfassung

Resümee

- Assemblersprache reflektiert das Programmiermodell einer CPU
 - Befehls-/Registersatz, Adressierungsarten
- Maschinenbefehle werden in „verständliche“ Kürzel dargestellt
 - Mnemonik, symbolischer Maschinenkode
- Pseudobefehle steuern den Assemblier- und Bindevorgang
 - Objektmodule erzeugen, zum Lademodul zusammenfügen

Gründe für die Programmierung in Assemblersprache

- Unzulänglichkeiten einer Hochsprache korrigieren [SP]
- Schwachstellen eines Kompilers umgehen
- Elementaroperationen des Prozessors erzwingen [SP]
- Spezialbefehle der Hardware absetzen [SP]
- ⋮
- Funktionsweise einer CPU besser verstehen und beurteilen [GRAO]

Literaturverzeichnis

- [1] AMD:
Software Optimization Guide for AMD64 Processors / Advanced Micro Devices, Inc.
2005 (25112). –
Handbuch
- [2] ECKERDAL, J. :
Doing a NOP: ASM/8086.
http://www.df.lth.se/~john_e/gems/gem0008.html, März 1998
- [3] ELSNER, D. ; FENLASON, J. :
Using as: The GNU Assembler.
Boston, MA, USA: Free Software Foundation, Inc., Jan. 1994
- [4] FREE SOFTWARE FOUNDATION, INC. (Hrsg.):
Using Inline Assembly with gcc.
Boston, MA, USA: Free Software Foundation, Inc., Jan. 2000
- [5] ISO/IEC 14677:
Information technology — Syntactic metalanguage — Extended BNF.
<http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>, 1996
- [6] *Intel Pentium Instruction Set Reference.*
<http://faydoc.tripod.com/cpu>, 2010