Vorlesung: Rechnerstrukturen, Teil 2 (Modul IP7)

J. Zhang

zhang@informatik.uni-hamburg.de

Universität Hamburg

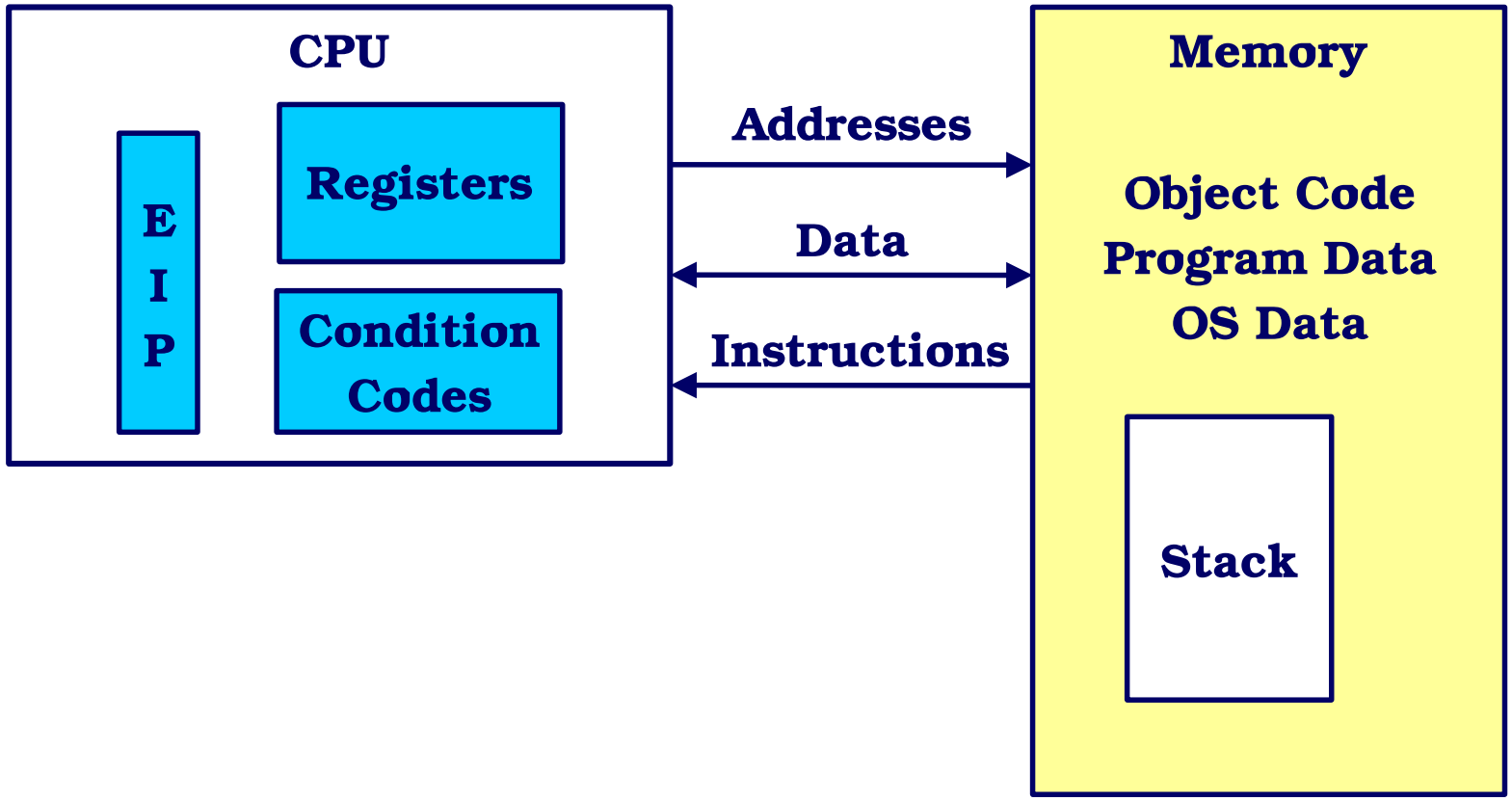
Fachbereich Informatik

AB Technische Aspekte Multimodaler Systeme

Inhaltsverzeichnis

Assembler-Programmierung

Assembler aus der Sicht des Programmierers:



Beobachtbare Zustände (Assemblersicht)

- EIP Programmzähler
  - ✦ Adresse der nächsten Anweisung

Beobachtbare Zustände (Assemblersicht)

- EIP Programmzähler
  - ✦ Adresse der nächsten Anweisung
- Register-Bank
  - ✦ Stark benutzte Programmdaten

Beobachtbare Zustände (Assemblersicht)

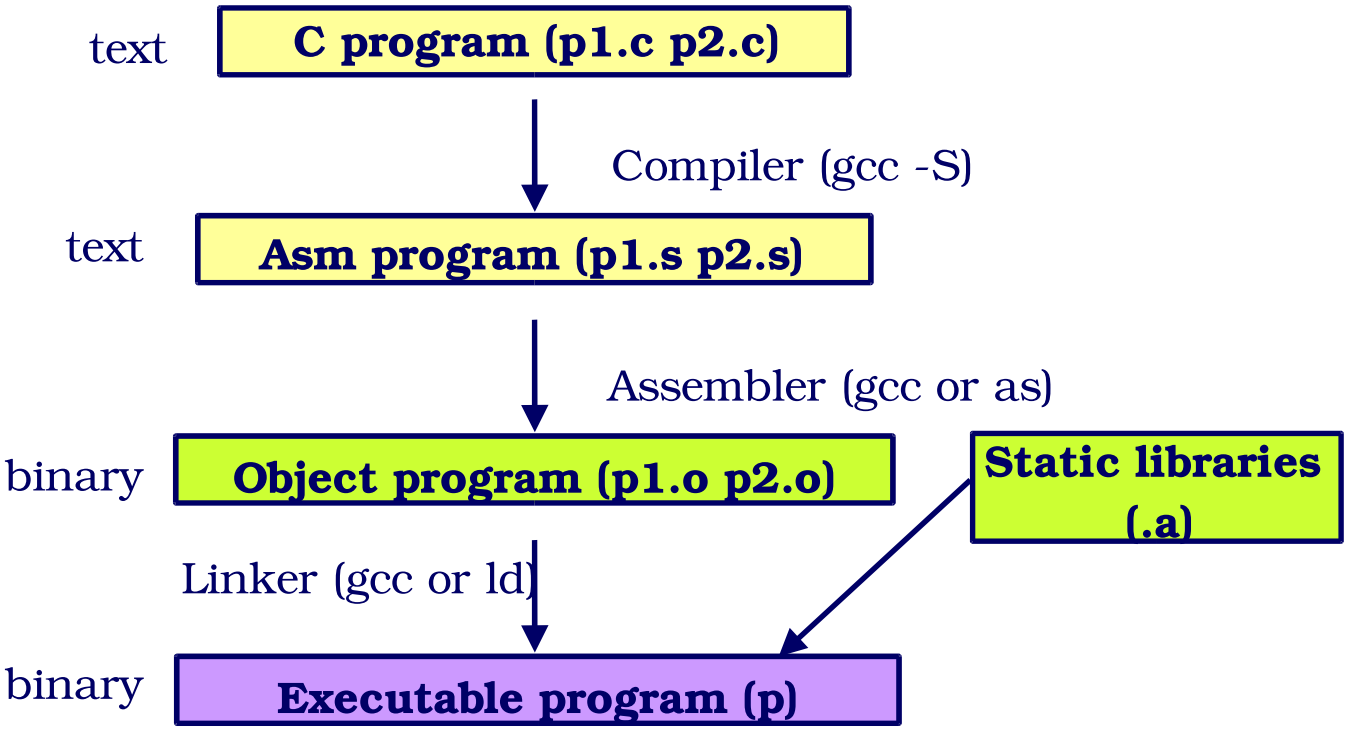
- EIP Programmzähler
  - ✦ Adresse der nächsten Anweisung
- Register-Bank
  - ✦ Stark benutzte Programmdaten
- Zustandscodes
  - ✦ Speichern Statusinformationen über die letzte arithmetische Operation
  - ✦ Werden für Conditional Branching benutzt

Beobachtbare Zustände (Assemblersicht)

- EIP Programmzähler
  - Adresse der nächsten Anweisung
- Register-Bank
  - Stark benutzte Programmdaten
- Zustandscodes
  - Speichern Statusinformationen über die letzte arithmetische Operation
  - Werden für Conditional Branching benutzt
- Speicher
  - Durch Bytes addressierbarer Array
  - Code, Nutzerdaten, (einige) OS Daten
  - Beinhaltet Kellerspeicher zur Unterstützung von Abläufen

Assembler und Disassembler

Umwandlung von C in Objektcode:



Kompilieren in Assembler

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated Assembly

```
_sum:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%e
        addl 8(%ebp),%ea
        movl %ebp,%esp
        popl %ebp
        ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Assembler Charakteristika – I

Minimale Datentypen

- Ganzzahl- Daten mit 1,2 oder 4 Bytes
  - Datenwerte
  - Adressen
- Gleitkomma-Daten mit 4,8, oder 10 Bytes
- Keine Aggregatstypen wie Arrays oder Strukturen
  - Nur fortlaufend zugeteilte Bytes im Speicher

Assembler Charakteristika – II

Primitive Operationen

- Führen am Register oder den Speicherdaten arithmetische Funktionen durch
- Transferieren Daten zwischen dem Speicher und dem Register
  - ✦ Laden Daten aus dem Speicher ins Register
  - ✦ Legen Registerdaten im Speicher ab
- Transferieren Kontrolle
  - ✦ “Unconditional Jumps” zu/von Prozeduren
  - ✦ “Conditional Branches”

Objektcode

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

Assembler und Binder

Assembler

- Übersetzt .s zu .o
- Binäre Kodierung jeder Anweisung
- Fast vollständiges Bild des ausführbaren Codes
- Fehlende Verknüpfungen zwischen Codes in verschiedenen Dateien

Assembler und Binder

Assembler

- Übersetzt .s zu .o
- Binäre Kodierung jeder Anweisung
- Fast vollständiges Bild des ausführbaren Codes
- Fehlende Verknüpfungen zwischen Codes in verschiedenen Dateien

Binder (Linker)

- Löst Referenzen zwischen Dateien auf
- Kombiniert mit statischen Laufzeit-Bibliotheken
  - ✦ Z.B. Code für malloc, printf
- Manche Bibliotheken sind *dynamisch verknüpft*
  - ✦ Die Verknüpfung tritt auf, wenn das Programm mit der Durchführung beginnt

Beispiel eines Maschinenbefehls

```
int t = x+y;
```

```
    addl 8(%ebp),%eax
```

Similar to expression

```
x += y
```

```
0x401046:     03 45 08
```

C Code: `int t = x+y;`

- Fügt zwei Ganzzahlen mit Vorzeichen hinzu

C Code: `int t = x+y;`

- Fügt zwei Ganzzahlen mit Vorzeichen hinzu

Assembler: `addl 8(%ebp), %eax`

- Fügt 2 4-Byte Ganzzahlen hinzu
  - ✦ "Lange" Wörter in GCC Sprache
  - ✦ Der gleiche Befehl, egal ob mit oder ohne Vorzeichen
- Operanden:

```
x:    Register      %eax
y:    Speicher M[%ebp+8]
t:    Register     %eax
Resultatwert in  %eax
```

C Code: `int t = x+y;`

- Fügt zwei Ganzzahlen mit Vorzeichen hinzu

Assembler: `addl 8(%ebp), %eax`

- Fügt 2 4-Byte Ganzzahlen hinzu
  - ✦ "Lange" Wörter in GCC Sprache
  - ✦ Der gleiche Befehl, egal ob mit oder ohne Vorzeichen
- Operanden:

```
x:    Register      %eax
y:    Speicher M[%ebp+8]
t:    Register     %eax
Resultatwert in  %eax
```

Objektcode: `0x401046: 03 45 08`

- 3-Byte Befehl, gespeichert unter der Adresse 0x401046

Objektcode Disassembler

```
00401040 <_sum>:
   0:   55                   push   %ebp
   1:   89 e5                mov    %esp,%ebp
   3:   8b 45 0c             mov    0xc(%ebp),%eax
   6:   03 45 08             add    0x8(%ebp),%eax
   9:   89 ec                mov    %ebp,%esp
   b:   5d                   pop    %ebp
   c:   c3                   ret
   d:   8d 76 00             lea    0x0(%esi),%esi
```

objdump -d p

- Nützliches Werkzeug zur Untersuchung des Objektcodes
- Untersucht das Bit-Muster von Anweisungsserien und rekonstruiert Assemblerquelle weitgehend
- Kann entweder auf einer a.out (vollständiges, ausführbares Programm) oder einer .o Datei ausgeführt werden

Datentransfers ("move")

Beispiel:
movl

movl *Source*, *Dest*:

- Transferiert ein 4-Byte ("long") Wort
- Kommt in typischem Code häufig vor

movl *Source, Dest*:

- Transferiert ein 4-Byte ("long") Wort
- Kommt in typischem Code häufig vor

Typen von Operanden

- Immediate: Konstante, ganzzahliges Datum
  - Wie C Konstante, aber mit dem Präfix '$'
  - Z.B., $0x400, $-533
  - Kodiert mit 1,2 oder 4 Bytes
- Register: Eins von 8 ganzzahligen Registern
  - Aber %esp und und %ebp für speziellen Gebrauch reserviert
  - Andere haben spezielle Verwendungen für bestimmte Anweisungen
- Speicher: 4 konsekutive Speicherbytes
  - Zahlreiche "Adressmodi"

movl Operanden-Kombinationen

	Source	Destination		C Analogon
movl	*Imm*	*Reg*	`movl $0x4, %eax`	`temp = 0x4;`
		Mem	`movl $-147, (%eax)`	`*p = 147`
	Reg	*Reg*	`movl %eax, %edx`	`temp2 = temp1;`
		Mem	`movl %eax, (%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax), %edx`	`temp = *p;`

Einfache Addressierungsmodi (Speicherreferenzen)

`Normal (R) Mem[Reg[R]]`

- Register R spezifiziert die Speicheradresse

```
movl (%ecx), %eax
```

Einfache Addressierungsmodi (Speicherreferenzen)

`Normal (R) Mem[Reg[R]]`

- Register R spezifiziert die Speicheradresse

```
movl (%ecx), %eax
```

`Displacement D(R) Mem[Reg[R+D]]`

- Register R
- Konstantes "Displacement" D spezifiziert den "offset"

```
movl 8 (%ebp), %edx
```

Benutzen einfacher Adressierungsmodi

```
void swap(int *xp, int *yp)
{
        int t0 = *xp;
        int t1 = *yp;
        *xp = t1;
        *yp = t0;
}
```

```
swap:
        pushl   %ebp
        movl    %esp,       %ebp
        pushl   %ebx

        movl    12(%ebp), %ecx
        movl    8(%ebp),    %edx
        movl    (%ecx),     %eax
        movl    (%edx),     %ebx
        movl    %eax,       (%edx)
        movl    %ebx,       (%ecx)

        movl    -4(%ebp), %ebx
        movl    %ebp,       %esp
        popl    %ebp
        ret
```

Indizierte Adressierungmodi

Gebräuchlichste Form
Imm(Rb, Ri, S) Mem[Reg[Rb]+S*Reg[Ri]+Imm]

- Imm : Offset
- Rb : Basisregister: Eins von 8 ganzzahligen Registern
- Ri : Indexregister: jedes außer %esp
  - ✦ Unwarscheinlich auch, dass %ebp benutzt wird
- S : "Scaling" Faktor, 1,2,4 oder 8

Indizierte Adressierungmodi

Gebräuchlichste Form
Imm(Rb, Ri, S) Mem[Reg[Rb]+S*Reg[Ri]+Imm]

- Imm : Offset
- Rb : Basisregister: Eins von 8 ganzzahligen Registern
- Ri : Indexregister: jedes außer %esp
  - ✦ Unwarscheinlich auch, dass %ebp benutzt wird
- S : "Scaling" Faktor, 1,2,4 oder 8

Spezielle Fälle

```
(Rb,Ri)        Mem[Reg[Rb] + Reg[Ri]]
Imm(Rb,Ri)     Mem[Reg[Rb] + Reg[Ri] + Imm]
(Rb,Ri,S)      Mem[Reg[Rb] + S*Reg[Ri]]
```

Beispiel für Adressenberechnung

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

Arithmetische Operationen

Two Operand Instructions

Format		Operation	
addl	Src,Dest	Dest = Dest + Src	
subl	Src,Dest	Dest = Dest - Src	
imull	Src,Dest	Dest = Dest * Src	
sall	Src,Dest	Dest = Dest << Src	Also called shll
sarl	Src,Dest	Dest = Dest >> Src	Arithmetic
shrl	Src,Dest	Dest = Dest >> Src	Logical
xorl	Src,Dest	Dest = Dest ^ Src	
andl	Src,Dest	Dest = Dest & Src	
orl	Src,Dest	Dest = Dest | Src	

Einige Arithmetische Operationen

One Operand Instructions

Format	Operation
`incl Dest`	`Dest = Dest + 1`
`decl Dest`	`Dest = Dest - 1`
`negl Dest`	`Dest = - Dest`
`notl Dest`	`Dest = ~ Dest`

Ein Beispiel

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
movl 8(%ebp),%eax        eax = x
xorl 12(%ebp),%eax       eax = x^y          (t1)
sarl $17,%eax            eax = t1>>17       (t2)
andl $8185,%eax          eax = t2 & 8185
```

Kontrollfluss

Themen

- Zustandscodes
  - Setzen
  - Testen

Kontrollfluss

Themen

- Zustandscodes
  - Setzen
  - Testen

- Ablaufsteuerung
  - "If-then-else" (Verzweigungen)
  - "Loop"-Variationen
  - "Switch Statements"

Zustandscodes

Einzel- Bit Register

`CF Carry Flag` `SF Sign Flag`
`ZF Zero Flag` `OF Overflow Flag`

Implizit gesetzt durch Arithmetische Operationen, z.B. `addl Src, Dest`
C analog: `t = a + b`

- CF setzen, wenn höchstwertigstes Bit Übertrag generiert
- Benutzt um Überlauf ohne Vorzeichen zu entdecken
- ZF setzen wenn $t == 0$
- SF setzen wenn $t < 0$
- OF setzen wenn das Zweierkomplement überläuft

$(a > 0 \;\&\&\; b > 0 \;\&\&\; t < 0) \;||\; (a < 0 \;\&\&\; b < 0 \;\&\&\; t >= 0)$

Zustandscodes setzen (Forts.)

Explizit Setzen bei Vergleichsanweisung `cmpl Src2, Src1`

- cmpl b, a wie Berechnung von a-b ohne Abspeichern des Resultats
- CF setzen, falls höchstwertigstes Bit Übertrag generiert
  - ✦ Für Vergleiche ohne Vorzeichen benutzt
- ZF setzen wenn $a = b$
- SF setzen wenn $(a - b) < 0$
- OF setzen wenn das Kompliment von zwei überläuft

$$(a > 0 \&\& b > 0 \&\& (a - b) < 0) || (a < 0 \&\& b > 0 \&\& (a - b) > 0)$$

Zustandscodes setzen (Forts.)

Explizit Setzen bei Testanweisung `testl Src2, Src1`

- Setzt Zustandscodes basierend auf dem Wert von Src1 & Src2
  - Hilfreich, wenn einer der Operanden eine Bitmaske ist
  - testl b, a wie Berechnung von a&b ohne Einstellungs-Ziel
  - ZF setzen wenn $a\&b = 0$
  - Sf setzen wenn $a\&b < 0$

Zustandscodes Lesen

SetX Anweisungen

- Setzt einzelnes Byte (Ein-Byte Registerelemente) basierend auf Kombinationen von Zustandscodes

SetX	Zustand	Beschreibung
sete	ZF	Gleich/Null
setne	~ZF	Nicht gleich / Nicht Null
sets	SF	Negativ
setns	~SF	Nicht-Negativ
setg	~(SF^OF) & ~ZF	Größer (mit Vorzeichen)
setge	~(SF^OF	Größer oder Gleich (mit Vorzeichen)
setl	(SF^OF)	Kleiner (mit Vorzeichen)
setle	(S\^OF)| ZF	Kleiner oder Gleich (mit Vorzeichen)
seta	~CF&~ZF	Darüber (ohne Vorzeichen)
setb	CF	Darunter (ohne Vorzeichen)

Zustandscodes Lesen (Forts.)

SetX Anweisungen

- Setzt einzelnes Byte basierend auf Kombinationen von Zustandscodes
- Ein addressierbares Byte aus 8 wird erfasst
  - Eingebettet in die ersten 4 ganzzahligen Register
  - Verändert die 3 übrigen Bytes nicht
  - Gebraucht typischerweise movzbl um die Aufgabe zu vollenden


```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Compare x : y
setg %al              # al = x > y
movzbl %al,%eax       # Zero rest of %eax
```

Sprungbefehle ("Jump")

JX Anweisungen: "Jumping" zu verschiedenen Teilen des Codes abhängig von Zustandscodes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signe
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

"Conditional Branch" Beispiel

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:
        movl %ebp,%esp
        popl %ebp
        ret
```

Set Up

Body

Finish

"Conditional Branch" Beispiel (Forts.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

```
    movl 8(%ebp),%edx # edx = x
    movl 12(%ebp),%eax  # eax = y
    cmpl %eax,%edx      # x : y
    jle L9              # if <= goto L9
    movl %edx,%eax      # eax = x
L9:                 # Done:
```

Skipped when

- C erlaubt "goto" als Mittel der Kontrollübertragung
  - ✦ Näher an Maschinen-Level Programmierungsstil
- Allgemein als schlechter Kodierungsstil angesehen

"Do-While" Loop Beispiel

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Assembly

```
_fact_goto:
        pushl %ebp    # Setup
        movl %esp,%ebp       # 
        movl $1,%eax  # eax = 1
        movl 8(%ebp),%edx    # 

L11:
        imull %edx,%eax      # 
x
        decl %edx     # x--
        cmpl $1,%edx  # Compare
        jg L11 # if > goto loop

        movl %ebp,%esp       # 
        popl %ebp     # Finish
        ret    # Finish
```

- Benutzt "Backward branch" um "Looping" fortzusetzen
- Nimmt "Branch" nur wenn "while" Bedingung besteht

Allgemeine "Do-While" Übersetzung

C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- "Body" kann beliebige Folge von C "Statements" sein
  - ✦ Test ist zurückkommende Ganzzahl
    = 0 als falsch interpretiert ≠0 als wahr interpretiert

“Switch Statements”

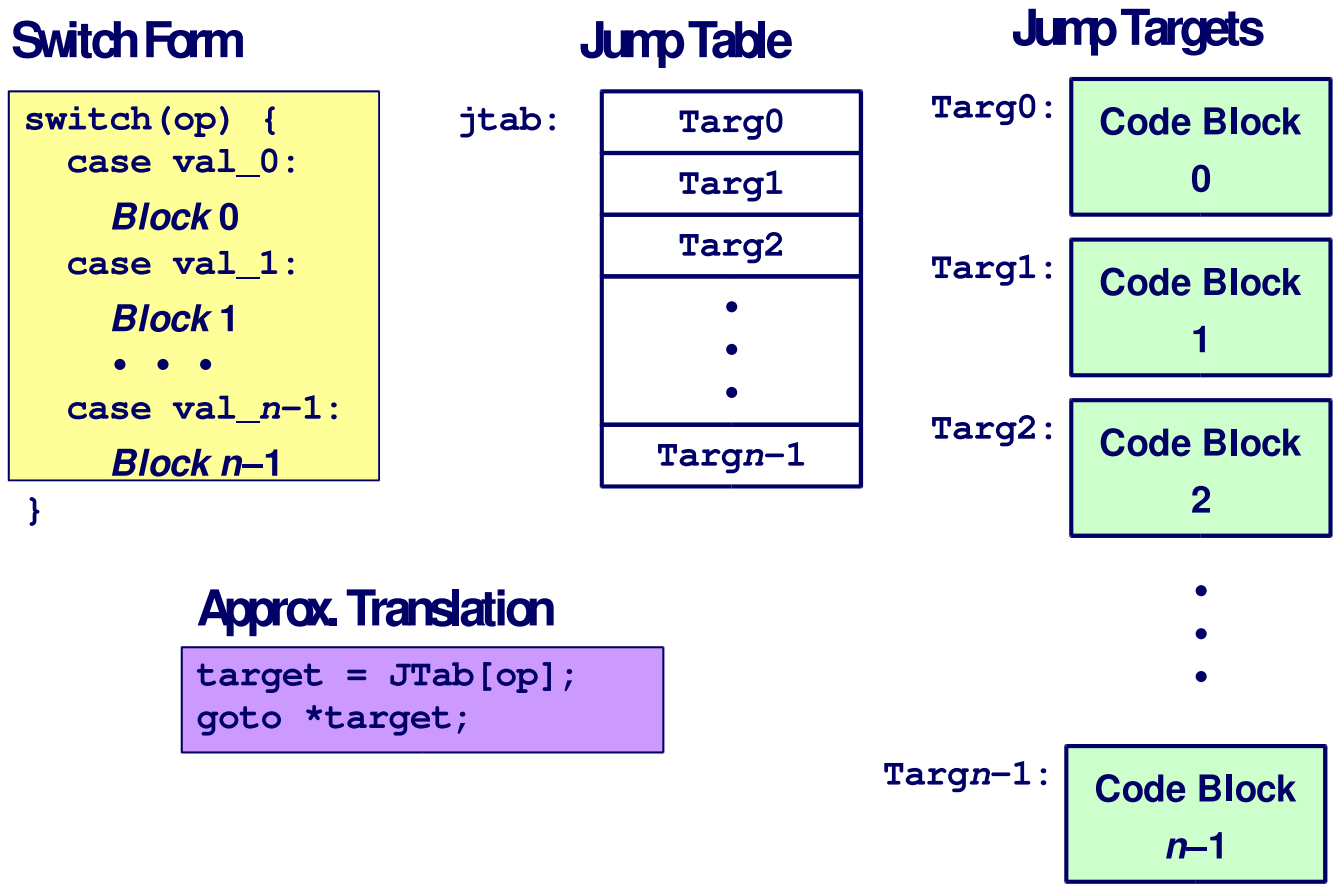
```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Implementierungsoptionen

- Serie von "Conditionals"
  - ✦ Gut falls wenige Fälle
  - ✦ Langsam bei vielen Alternativen
- "Jump Table"
  - ✦ "Lookup Branch Target"
  - ✦ Vermeidet "Conditionals"
  - ✦ Möglich falls Alternativen kleine ganzzahlige Konstanten sind
- GCC
  - ✦ Wählt eine der beiden Varianten basierend auf der Fallstruktur
- Bug im Beispielcode
  - ✦ keine Vorgabe gegeben

"Jump Table" Struktur



"Switch Statement" Beispiel

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Setup:

```
unparse_symbol:
        pushl %ebp      # Setup
        movl %esp,%ebp          # Setup
        movl 8(%ebp),%eax       # eax = op
        cmpl $5,%eax    # Compare op : 5
        ja .L49         # If > goto done
        jmp *.L57(,%eax,4)      # goto Table
[op]
```

Assembler-Setup Erkärung

Symbolische Labels

- Labels der Form .LXX werden von Assembler in Adressen übersetzt

"Table" Struktur

- Jedes Ziel erfordert 4 Bytes
- Basisadresse bei .L57

"Switch Statement" (Abschluss)

Jumping
jmp .L49

- "Jump Target" wird mit Label .L49 gekennzeichnet
  jmp * :L57(,%eax, 4)
- Start des "Jump Table" wird mit Label .L57 gekennzeichnet
- Register %eax hält op
- Muss mit dem Faktor 4 skaliert werden um "offset" ins " Table" zu bekommen
- Holt "Target" von der effektiven Adresse .L57 + op*4

Vorteil des "Jump Table"

- Kann k-way branch in O(1) Operationen durchführen

“Jump Table” aus “Binary” Extrahieren

“Jump Table” in “Read Only” Datensegment gespeichert (.rodata)

- Zahlreiche feste Werte von Ihrem Code benötigt

Kann mit obdjump untersucht werden:
obdjump code-examples -s –section=.rodata

- Zeigt alles im angegebenen Segment.

Schwer zu lesen

- “Jump Table” Einträge mit umgekehrter Byte-Anordnung gezeigt

Zusammenfassung

C Kontrollstrukturen

- "If-then-else"
- "do-while"
- "while"
- "switch"

Assembler Kontrolle

- "Jump"
- "Conditional Jump"

Compiler

- Muss Assembler Code generieren, um komplexere Kontrolle zu implementieren

Standard Techniken

- Alle loops konvertiert zu "do-while" Form
- Große "Switch Statements" verwenden "Jump Tables"

CISC Bedingungen

- CISC Maschinen haben im Allgemeinen Zustandscode-Register

RISC Bedingungen

- Benutzen Universalregister um Zustandsinformationen zu speichern
- Spezielle Vergleichs-Anweisungen
- Z.B. auf Alpha
  cmple $16, 1, $1
  - ✦ Stellt Register $1 auf 1 wenn Register $\$16 <= 1$

Ergänzende Literatur

Zur Rechnerarchitektur (3. Termin):

Literatur

[1] Randal E. Bryant and David O'Hallaron. *Computer Systems*. Pearson Education, Inc., New Jersey, 2003.

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design. The Hardware / Software Interface*. Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[3] Andrew S. Tanenbaum. *Computerarchitektur*. Pearson Studium München, 2006.