DS-Systeme Kapitel 8

Kontroll-strukturen Teil 1

Inhaltsverzeichnis

Kontrollstrukturen - Einführung

In C bzw. C++ werden folgende Kontrollmechanismen verwendet:

- **if**-Anweisungen (conditionals)

- Schleifen (loops)
  - **do while**
  - **while**
  - **for**

- **switch**-Anweisungen

Dafür bietet die Maschinensprache zwei Mechanismen an. In beiden Fällen werden Werte getestet und der Kontroll- (control flow) bzw. Datenfluss (data flow) entsprechend angepasst.

Anpassung des Kontrollflusses

- Änderung der Reihenfolge der Ausführung (jump instructions)

Anpassung des Datenflusses

- Bedingte-Befehle (conditional instructions / conditional mov)

***** Die Anpassung des Datenflusses kommt aus der ARM-Architektur, ist auf den meisten Intel-Prozessoren noch vorhanden, auf anderen Plattformen gar nicht mehr.

Kontrollstrukturen – Flags (single bit condition codes)

Die CPU verwaltet einen Satz von **Flags** bzw. **single-bit condition codes** (einzelne Bits im Statusregister RFLAGS bzw. EFLAGS)

Die wichtigsten sind:

CF (carry flag)
- Detektiert einen Überlauf (overflow) für **unsigned** Datentypen. Bei einem Rechts-Shift wird es auf die zuletzt herausgeschobene Ziffer gesetzt.

ZF (zero flag)
- Zeigt an, ob das Ergebnis der letzten Operation **null** war.

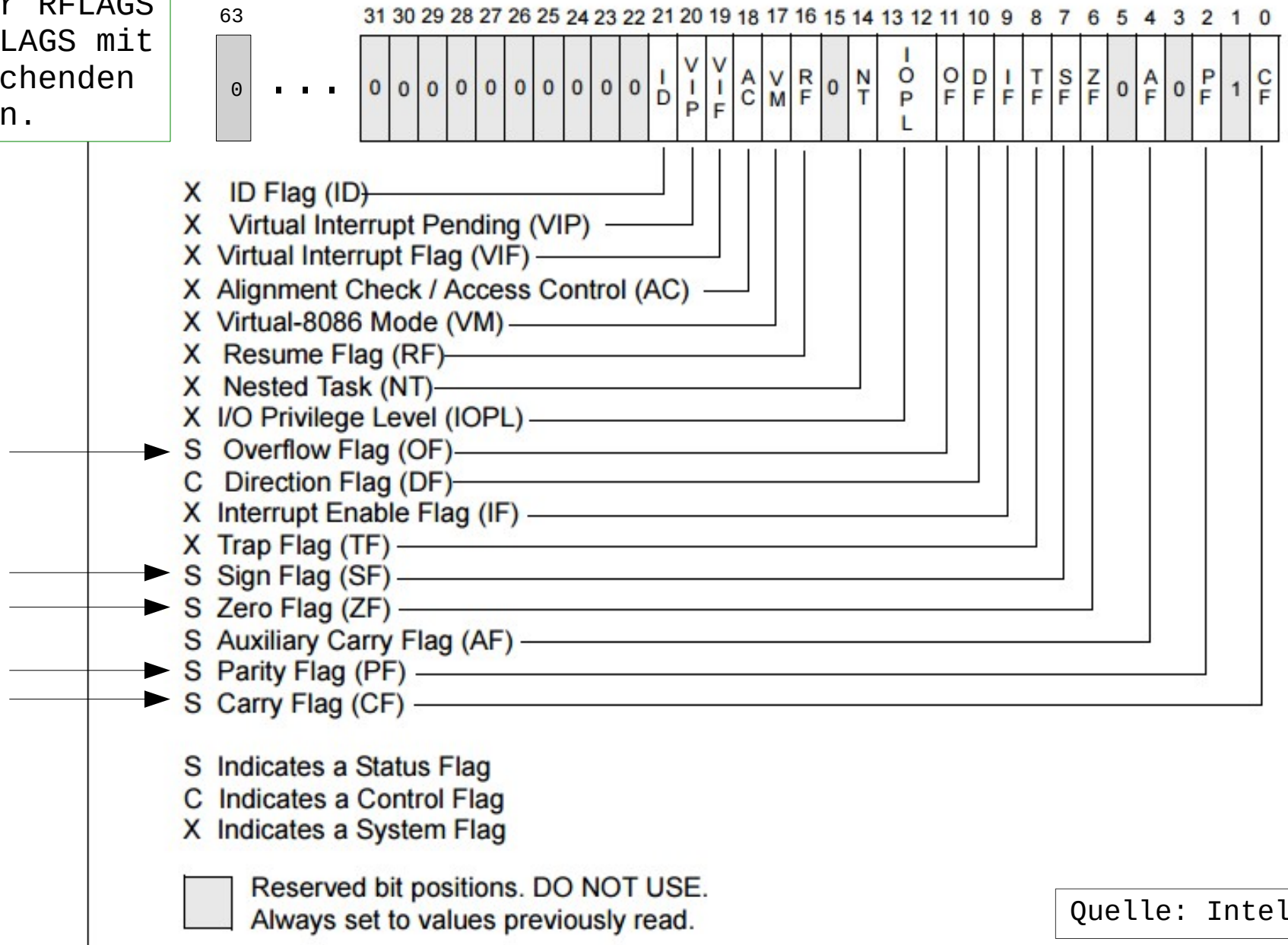
SF (sign flag)
- Zeigt an, ob die letzte Operation ein negatives Ergebnis erzeugt hat.

OF (overflow flag)
- Zeigt an, ob die letzte Operation für **signed** Datentypen einen Zweier-komplement-Überlauf (negativ oder positiv) (twos complement overflow) hatte.

PF (parity flag)
- Zeigt nach arithmetischen Operationen an, ob die Summe der 1en im **least significant Byte** gerade (PF = 1) oder ungerdade ist (PF = 0)

(siehe auch Dateien **carryOverflow.pdf** und **carrySubstraction.pdf**)

Register RFLAGS bzw. EFLAGS mit entsprechenden Inhalten.



Quelle: Intel

Kontrollstrukturen – Ablauf des Testens und Reaktionen darauf

Testen:

Durch arithmetische und logische Operationen werden folgende Flags gesetzt:

- Logische Operationen (**and**, **or**, u.s.w.) setzen **CF** und **OF** auf 0.

- Right-Shift-Operationen setzen **CF** auf den Bit-Wert, der als letztes hinausgeschoben wurde und **OF** auf 0.

- Die Operationen **inc** und **dec** setzen **OF** und **ZF** auf 0 (es sei denn es kommt zu einem Übertrag oder Overflow), lassen aber **CF** unverändert.

Hinweis:

Es gibt Befehle, die die Flags gar nicht ändern, z.B. **mov**, **lea, set.**
Und es gibt Befehle, die immer Flags setzen: z.B. **cmp**, **test**

(siehe auch Datei **testFlags.pdf**)

Reaktionen bzw. Aktionen nach dem Testen:

- Setzen eines Bytes (als Zwischenergebnis) auf 0 oder 1
  (set instruction)
- conditional jump
- conditional data transfer (conditional mov)

Kontrollstrukturen – cmp, test

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
cmpb			Compare byte
cmpw			Compare word
cmpl			Compare double word
cmpq			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
testb			Test byte
testw			Test word
testl			Test double word
testq			Test quad word

Beachten Sie die Indizees!

cmp und **test** werden explizit verwendet, wenn die Befehle selbst die RFLAGS nicht implizit ändern.

cmp funktioniert wie eine Subtraktion zweier Operanden. Das Ergebnis ist dann <, = oder > 0

test funktioniert wie ein Vergleich mit 0 bzw. auf ungleich 0 unter Verwendung der &-Verknüpfung.

z.B.:
5 & 5 = 5 (ungleich 0)
5 & 6 = 4 (ungleich 0)
--> **ZF = 0**
5 & 2 = 0 (gleich 0)
5 & 0 = 0 (gleich 0)
0 & 0 = 0 (gleich 0)
--> **ZF = 1**

Kontrollstrukturen – test

Befehl **test** – Beschreibung:

Berechnet das bitweise logische UND des ersten Operanden (Source1) und des zweiten Operanden (Source2) und setzt die Status-Flags SF, ZF und PF entsprechend des Ergebnisses. **Das Ergebnis selbst wird dann verworfen.**

```
Temporary = Source1 & Source2;
SF = MSB(Temporary);
if(Temporary == 0) ZF = 1;
else ZF = 0;
PF = BitwiseXNOR(Temporary[0:7]);
for(PF = 1, i = 0; i < 8; ++i) PF ^= Temporary[i];
CF = 0;
OF = 0;
AF = Undefined;
```

Auf **test** folgende Sprünge beziehen sich auf die Flags – **nicht** auf das Resultat von Source1 & Source2.

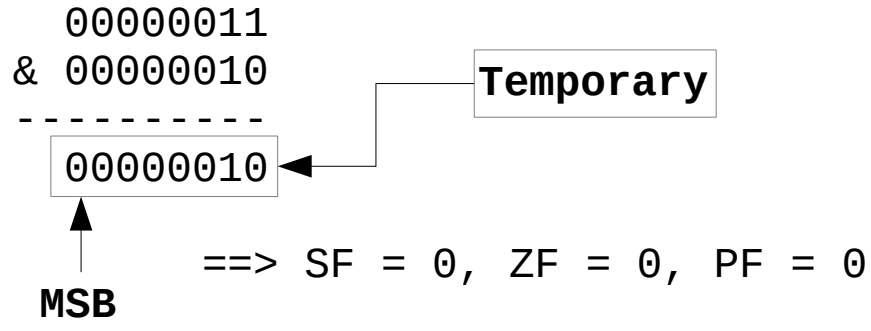
jz – jump zero
jnz – jump not zero
js – jump if sign
jns – jump if not sign
jp / jpe – jump parity even
jnp / jpo – jump parity odd

test S1, S2

S1 kann Immediate oder Register sein
S2 kann Register oder Speicher sein

Beispiel:
```
movb $3,  %al  # 00000011
testb $2, %al  # 00000010
```

Berechnung:



Kontrollstrukturen – set

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← `ZF`	Equal / zero
`setne`	*D*	`setnz`	*D* ← ~ `ZF`	Not equal / not zero
`sets`	*D*		*D* ← `SF`	Negative
`setns`	*D*		*D* ← ~ `SF`	Nonnegative
`setg`	*D*	`setnle`	*D* ← ~ (`SF` ^ `OF`) & ~ `ZF`	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← ~ (`SF` ^ `OF`)	Greater or equal (signed >=)

Bei **D** (destination) handelt es sich hier jeweils um die **8 least significant** (niederwertigsten) **Bits eines Registers**, die je nach Ergebnis auf 0 oder 1 gesetzt werden.
Das Resultat bezieht sich auf den letzten **cmp**- bzw. **test**-Befehl, der **vor** dem **set**-Befehl ausgeführt wurde.

Kontrollstrukturen – set

setl	D	setnge	D ← SF ^ OF	Less (signed <)
setle	D	setng	D ← (SF ^ OF) | ZF	Less or equal (signed <=)
seta	D	setnbe	D ← ~ CF & ~ ZF	Above (unsigned >)
setae	D	setnb	D ← ~ CF	Above or equal (unsigned >=)
setb	D	setnae	D ← CF	Below (unsigned <)
setbe	D	setna	D ← CF | ZF	Below or equal (unsigned <=)

Erklärung wie vorherige Seite.

Kontrollstrukturen – set

Beispiel:

```
  .section .text
  .globl comp
  .type comp, @function
# int comp(data_t a, data_t b)
# a in rdi, b in rsi
comp:
  cmpq %rsi, %rdi   # Compare a:b
  setl %al          # Set low order byte of eax to 0 or 1
  movzbl %al, %eax  # Clear rest of eax
  ret
```

data_t steht hier für einen beliebigen Datentyp

set less (signed) →

```
a)                  a = 7, b = 5, Überprüfung: 7 < 5?
   Berechnung:      a – b = 7 – 5 = 2, 2 > 0 ==> 7 < 5 is false
                    SF ^ OF = 0 ^ 0 = 0 ==> %al = 00000000

b)                  a = 5, b = 7, Überprüfung: 5 < 7?
   Berechnung:      a – b = 5 – 7 = -2, -2 < 0 ==> 5 < 7 = true
                    ==> SF ^ OF = 1 ^ 0 = 1 ==> %al = 00000001

c)                  a = 5, b = 5, Überprüfung: 5 < 5?
    Berechnung:     a – b = 5 – 5 = 0, 0 = 0 ==> 5 < 5 = false
                    ==> SF ^ OF = 0 ^ 0 = 0 ==> %al = 00000000
```

Kontrollstrukturen – jump

Bei den ersten beiden Befehlen handelt es sich um unbedingte, bei den restlichen um bedingte Sprungbefehle.

Der Sprung wird immer dann ausgeführt, wenn die Sprungbedingung (jump condition) wahr ist (3. Spalte).

Hinweise:

Durch Sprünge ändert sich der Inhalt des Registers **rip** (**instruction pointer**), der immer auf die Adresse des als nächstes auszuführenden Befehls zeigt.

jmp *%rax
verwendet den Wert in **rax** (Adresse) als Sprungziel

jmp *(%rax)
verwendet den Wert, auf den **rax** zeigt (Adresse) als Sprungziel (Zeiger auf Zeiger)

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	ZF	Equal / zero
`jne`	*Label*	`jnz`	~ZF	Not equal / not zero
`js`	*Label*		SF	Negative
`jns`	*Label*		~SF	Nonnegative
`jg`	*Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`jge`	*Label*	`jnl`	~(SF ^ OF)	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	SF ^ OF	Less (signed <)
`jle`	*Label*	`jng`	(SF ^ OF) | ZF	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	~CF & ~ZF	Above (unsigned >)
`jae`	*Label*	`jnb`	~CF	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	CF	Below (unsigned <)
`jbe`	*Label*	`jna`	CF | ZF	Below or equal (unsigned <=)

Kontrollstrukturen – jump

Beispiel für **unbedingten** Sprung:

```
  pushq %rdx          # save content of rdx
  movq $0, %rax       # set %rax to 0
  jmp .L1             # goto .L1
  movq (%rax), %rdx # null pointer dereference (skipped)
.L1:                  # Jump target
  popq %rdx           # any instruction
```

Der Sprungbefehl wird hier in jedem Fall ausgeführt, weil er von keiner Bedingung abhängt.

Hinweis:

Bei **.L1** handelt es sich um einen typischen vom Compiler generierten Namen – man kann auch selbst definierte Namen wählen.

Kontrollstrukturen – jump

Vorgehensweise bei **bedingten** Sprüngen:

C- bzw. C++-Code:

```
if (test-expr)
  // true-statements
else
  // false-statements
```

Assembler-Code (dargestellt wie C-Code):

```
  t = test-expr;
  if (!t) goto false;
  true-statements
  goto done;
false:
  false-statements
done:
```

Wichtiger Hinweis:

In der Zeile `if (!t) goto false;` ist das Sprungziel das **false**-Label. Im Fall, dass die Bedingung **t** **true** ist, gibt es keinen Sprung.

Deshalb wird die Bedingung im Fall einer **if**-Anweisung in Assembler **immer** negiert (aus der Bedingung **t** wird **!t**).

Kontrollstrukturen – jump

Beispiel für **bedingten** Sprung:

C- bzw. C++-Code:

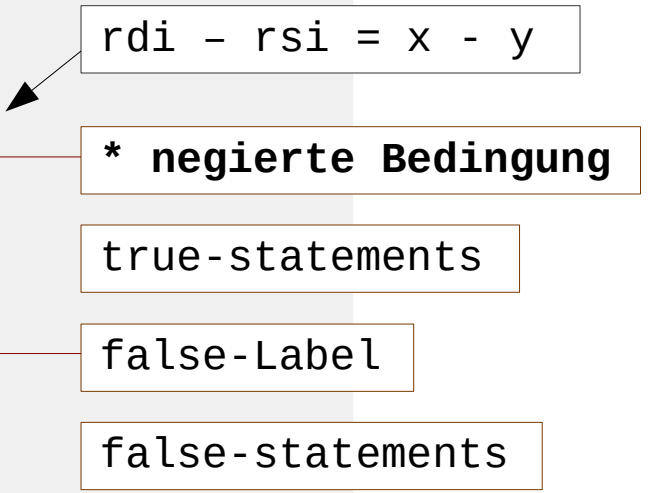
```
long absdiff(long x, long y)
{
  long result;
  if (x < y) {
    result = y - x;
  }
  else {
    result = x - y;
  }
  return result;
}
```

Assembler-Code:

```
.section .text
  .globl absdiff
  .type absdiff, @function

# long absdiff(long x, long y)
# x in rdi, y in rsi

absdiff:
  cmpq %rsi, %rdi
  jge .L2
  movq %rsi, %rax
  subq %rdi, %rax
  ret
.L2:
  movq %rdi, %rax
  subq %rsi, %rax
  ret
```



* **jge** (>=) (greater equal)
ist das Gegenteil von **jl** (<) (less)
bzgl. der Bedingung (x < y)

Kontrollstrukturen – jump

Beispiel für **bedingten** Sprung:

C- bzw. C++-Code:

```
long absdiff(long x, long y)
{
  long result;
  if (x < y) {
    result = y - x;
  }
  else {
    result = x - y;
  }
  return result;
}
```

Assembler-Code kommentiert:

```
.section .text
  .globl absdiff
  .type absdiff, @function
# long absdiff(long x, long y)
# x in rdi, y in rsi
absdiff:
  cmpq %rsi, %rdi          # Compare x:y
  jge .L2                  # If >= goto x_ge_y
  movq %rsi, %rax
  subq %rdi, %rax          # result = y-x
  ret                      # Return
.L2:                       # {x_ge_y}:
  movq %rdi, %rax
  subq %rsi, %rax          # result = x-y
  ret                      # Return
```

Aufgabe

Vervollständigen Sie den C-Code:

C- bzw. C++-Code:

```
long test(long x, long y, long z)
{
  long val = ______;
  if (______) {
      val=___;
  }
  else if (______){
      val = ___;
  }
  return val;
}
```

Hinweis:

rep; ret wird verwendet, wenn **ret** auf einen Sprungbefehl folgt.
Dabei hat **rep** eine ähnliche Bedeutung wie **nop** (no operation).
Das Semikolon trennt die beiden Befehle.
Nur notwendig z.B. bei AMD-Architektur, bei Intel unnötig.

Assembler-Code:

```
#long test(long x, long y, long z)
#x in %rdi, y in %rsi, z in %rdx
test:
  leaq (%rdi,%rsi), %rax
  addq %rdx, %rax
  cmpq $-3, %rdi
  jge .L2
  movq %rdi, %rax
  imulq %rsi, %rax
  ret
.L2:
  cmpq $2, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
.L4:
rep; ret
```

Ausführliche Begründung unter
https://repzret.org/p/repzret

Aufgabe

Vervollständigen Sie den C-Code:

C- bzw. C++-Code:

```
long test1(long x, long x) {
  if (x < y)
      return x;
  else
      return y;
}
```

Vervollständigen Sie den Assembler-Code:

```
.section .text
# long test1(long x, long y)
# x in %rdi, y in %rsi
.globl test1
.type test1, @function
test1:

...
```