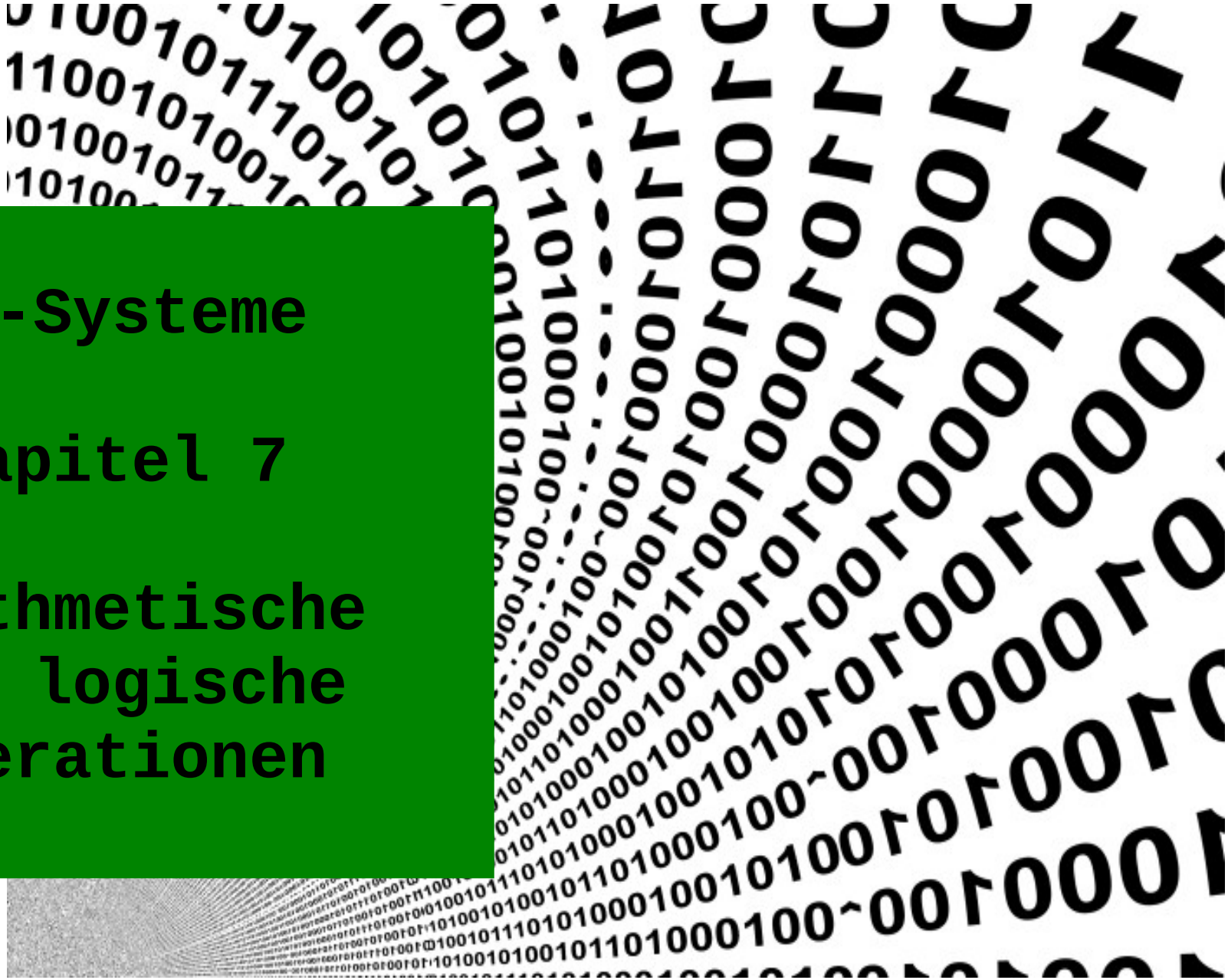

DS-Systeme

Kapitel 7

Arithmetische und logische Operationen

Inhaltsverzeichnis

Arithmetische und logische u.a. Operationen - Überblick

Instruction		Effect	Description
LEAQ	S,D	D <- &S	Load effective address
INC	D	D <- D +1	Increment
DEC	D	D <- D -1	Decrement
NEG	D	D <- -D	Negate
ADD	S,D	D <- D + S	Add
SUB	S,D	D <- D – S	Subtract
IMUL	S,D	D <- D * S	Multiply
NOT	D	D <- ~D	Complement
AND	S,D	D <- D & S	And
OR	S,D	D <- D | S	Or
XOR	S,D	D <- D ^ S	Exclusive-or
SAL	S,D	D <- D << k	Left Shift
SHL	S,D	D <- D << k	Left Shift (same as SAL)
SAR	S,D	D <- D $>>_A$ k	Arithmetic right shift
SHR	S,D	D <- D $>>_L$ k	Logical right shift

Adress-Operation

Arithmetische Operationen

Bit-Operationen

Bit-Shift-Operationen

Arithmetische und logische Operationen - lea

Instruction		Effect	Description
LEAQ	S,D	D <- &S	Load effective address

Wie **movq** – allerdings wird nicht die Source selbst, sondern ihre Adresse verschoben / kopiert. Da Adressen 8 Byte groß sind → Suffix "q".

```
.section .data
height:
  .quad 5

# in .section .text
  leaq height, %rdi
  movq (%rdi), %rax
```

Verwendung 1:

Kopieren von Adressen in ein Register.

← Adresse von **height** wird in Register **rdi** kopiert
← das, worauf **rdi** zeigt, wird nach **rax** kopiert

Verwendung 2:

Ausnutzung der Eigenschaften von **leaq** für eine Kombination aus Addition und Multiplikation.

```
# %rdx has any value x
leaq 7(%rdx, %rdx, 4), %rax # %rax = 5x + 7
```

In diesem Fall gilt $r_b = r_i$

rax = rdx * 4 + rdx + 7 = 5 * rdx + 7

Hinweis: **lea** addressiert keinen Speicherbereich, sondern verschiebt nur den (evtl. berechneten) Adresswert in ein Register.

Arithmetische und logische Operationen - lea

Ein Label repräsentiert normalerweise den ihm zugewiesenen Wert (s.u.: height = 5).

Alternativ kann bei einem Label die Adresse mit Hilfe des **$**-Operators zur Kompilationszeit bestimmt werden (**leaq** funktioniert auch zur Laufzeit).

```
.section .data
height:
  .quad 5
.section .text
  movq $height, %rdi
  movq (%rdi), %rax
```

← Adresse von **height** wird in Register **rdi** kopiert
← das, worauf **rdi** zeigt, wird nach **rax** kopiert

GAS – Arithmetische und logische Operationen – inc, dec, add, sub

Instruction		Effect	Description
INC	D	D <- D +1	Increment
DEC	D	D <- D -1	Decrement
NEG	D	D <- -D	Negate (keine Bit-Operation)
ADD	S,D	D <- D + S	Add
SUB	S,D	D <- D – S	Subtract

unäre Operatoren

binäre Operatoren

Bei den beiden binären Operatoren kann **Source**

- **Immediate**
- **Register**
- **Memory**

sein und **Destination**

- **Register**
- **Memory**

Aber in beiden zusammen darf **Memory** höchstens einmal vorkommen.

Beispiel:

```
movq $-5, %rax
negq %rax
subq $10, %rax
addq $15, %rax   # %rax has now
                 # value 10
```

Immediate Register

Aufgabe – inc, dec, add, sub

Gegeben sind folgende Werte in Registern und Speicheradressen:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0X11		

Füllen Sie die untenstehende Tabelle aus. Betrachten Sie dabei die Befehle als unabhängig voneinander.

Instruction	Destination (address/reg)	Value
addq %rcx, (%rax)		
subq %rdx, 8(%rax)		
incq 16(%rax)		
decq %rcx		
subq %rdx, %rax		

GAS – Arithmetische und logische Operationen - Bit-Operatoren

Instruction		Effect	Description
NOT	D	D <- ~D	Complement
AND	S,D	D <- D & S	And
OR	S,D	D <- D | S	Or
XOR	S,D	D <- D ^ S	Exclusive-or

not ist auch eine unäre Operation. Nicht zu verwechseln mit **neg**!

	NOT	AND	OR	XOR (excl. or)
Funktions gleichung	$y = \overline{x1}$	$y = x1 \wedge x2$	$y = x1 \vee x2$	$y = x1 \oplus x2$
C bit-level	y= ~x1;	y= x1 & x2;	y= x1 | x2;	y = x1 ^ x2;
Wahrheitstabelle	x_1 | y 0 | 1 1 | 0	x_2 | x_1 | y 0 0 0 0 1 0 1 0 0 1 1 1	x_2 | x_1 | y 0 0 0 0 1 1 1 0 1 1 1 1	x_2 | x_1 | y 0 0 0 0 1 1 1 0 1 1 1 0

GAS – Arithmetische und logische Operationen - Bit-Shift-Operatoren

Instruction		Effect	Description
SAL	S,D	D <- D << k	Left Shift
SHL	S,D	D <- D << k	Left Shift (same as SAL)
SAR	S,D	D <- D $>>_A$ k	Arithmetic right shift
SHR	S,D	D <- D $>>_L$ k	Logical right shift

S (source), also **k**, repräsentiert hier die Anzahl der zu verschiebenden Bits.

- Die Anzahl der Bit-Shift's (shift amount) wird durch den ersten Operanden angegeben (**Immediate** oder **%cl**) – es ist tatsächlich nur mit **%cl** möglich!

- Der zweite Operand bestimmt, was geschoben werden soll (value to shift).

- Die Befehle **sal** und **shl** haben die gleiche Wirkung
  - **sal** - arithmetischer Shift nach links (arithmetic left shift)
  - **shl** – logischer Shift nach links (logical left shift)

  Beide Befehle füllen von rechts Nullen nach

- **sar** füllt beim Shift nach rechts (arithmetic right shift) von links Kopien des Sign-Bits nach

- **shr** füllt beim Shift nach rechts (logical right shift) von links Nullen nach - deshalb nur für **unsigned**-Datentypen geeignet

GAS – Arithmetische und logische Operationen - Bit-Shift-Operatoren

Beispiel für Shift-Operatoren (C-Code)

```
long shift_left4_right_n(long x, long n)
{
  x <<= 4;
  x >>= n;
  return x;
}
```

Gleiche Funktion in GNU Assembler:

```
#long shift_left4_right_n(long x, long n)
#x in %rdi, n in %rsi
shift_left4_right_n:
  movq %rdi, %rax # get x
  movq %rsi, %rcx # get n
  salq $4, %rax   # x <<= 4
  sarq %cl, %rax  # x >>= n
  ret
```

Shift amount mittels Immediate angegeben

Shift amount mittels Register **cl** angegeben.
Zur Erinnerung: **cl** ist der niederwertigste 8-Bit-Anteil des Registers **rcx**.

Aufgabe - Bit-Shift-Operatoren

```
#long shift_left4_right_n(long x, long n)
#x in %rdi, n in %rsi

shift_left4_right_n:
  movq %rdi, %rax
  movq %rsi, %rcx
  salq $4, %rax
  sarq %cl, %rax
  ret
```

Aufgabe:

Überlegen Sie, welchen Wert die Funktion zurückgibt, wenn sie mit den Parametern

a) x = 20, n = 8
b) x = -20, n = 8

aufgerufen wird.

GAS – Arithmetische und logische Operationen - Bit-Operatoren

Beispiel für Bit-Operatoren (C-Code):

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0F0F0F0F;
  long t4 = t2 - t3;
  return t4;
}
```

Gleiche Funktion in GNU Assembler:

```
  .text
  .globl arith
  .type arith, @function
# long arith(long x, long y, long z)
# x in rdi, y in rsi, z in rdx
arith:
  xorq %rsi, %rdi
a) leaq (%rdx, %rdx, 2), %rax
  salq $4, %rax
  andq $0x0F0F0F0F, %rdi
  subq %rdi, %rax
  ret
```

Aufgaben:

a) Überlegen Sie, was diese beiden Befehlszeilen bewirken.

b) Welches Ergebnis würde der Aufruf **arith(3, 2, 1)** zurück liefern?

GAS – Arithmetische und logische Operationen - Bit-Operatoren

Gleicher Code noch einmal mit Kommentaren:

```
  .text
   .globl  arith
   .type arith, @function
# long arith(long x, long y, long z)
# x in rdi, y in rsi, z in rdx
arith:
   xorq  %rsi, %rdi             # t1 = x ^ y
   leaq  (%rdx,%rdx,2), %rax    # 3 * z
   salq  $4, %rax               # t2 = 2^4 * (3 * z)
                                #    = 16 * 3 * z = 48 * z
   andq  $0x0F0F0F0F, %rdi      # t3 = t1 & 0x0F0F0F0F
   subq  %rdi, %rax             # return value = t2 - t3
   ret
```

Aufgabe – arithmetische und Bit-Operatoren

Gegeben ist der folgende GNU Assembler-Code:

```
#long arith2(long x, long y, long z)
#x in %rdi, y in %rsi, z in %rdx
arith2:
  orq %rsi, %rdi
  sarq $3, %rdi
  notq %rdi
  movq %rdx, %rax
  subq %rdi, %rax
  ret # %rax holds the return
      # value of a function
```

Vervollständigen Sie den C-Code entsprechend dem o.a. GAS-Programm.

```
long arith2(long x, long y, long z)
{
  long t1  =  __________;
  long t2  =  __________;
  long t3  =  __________;
  long t4  =  __________;
  return t4;
}
```

Aufgabe – arithmetische und Bit-Operatoren

Gegeben ist der folgende GNU Assembler-Code:

```
#long arith2(long x, long y, long z)
#x in %rdi, y in %rsi, z in %rdx
arith2:
  orq %rsi, %rdi
  sarq $3, %rdi
  notq %rdi
  movq %rdx, %rax
  subq %rdi, %rax
  ret # %rax holds the return
      # value of a function
```

Aufgabe:

Welches Ergebnis würde der Aufruf **arith2(3, 4, 5)** zurückliefern?

Schreiben Sie die Funktion noch einmal, ohne lokale Variablen anzulegen.

```
long arith2(long x, long y, long z)
{
  // ?
  return ?;
}
```

GAS – Arithmetische und logische Operationen – spez. arithm. Operatoren

Instruction	Effect	Description
`imulq` *S*	R[`%rdx`]:R[`%rax`] ← *S* × R[`%rax`]	Signed full multiply
`mulq` *S*	R[`%rdx`]:R[`%rax`] ← *S* × R[`%rax`]	Unsigned full multiply
`cqto`	R[`%rdx`]:R[`%rax`] ← SignExtend(R[`%rax`])	Convert to oct word
`idivq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Signed divide
`divq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Unsigned divide

GAS – Arithmetische und logische Operationen – mul, imul

Instruction	Effect	Description
`imulq` *S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Signed full multiply
`mulq` *S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Unsigned full multiply

Das "**S**" in der ersten Spalte der Tabelle bezeichnet den einzigen Operanden der Multiplikation.
z.B.: **movq $-0x1ABC1ABC, %r8**
imulq %r8

bedeutet: **-0x1ABC1ABC** (**%r8**) wird mit dem Inhalt von **rax** multipliziert

Bei der Multiplikation von zwei 64-Bit Operanden wird das Ergebnis doppelt so lang sein:

mulq und **imulq** 64bit-Value * 64bit-Value → 128bit-Value
imul signed multiplication, **mul** unsigned multiplication

Das Ergebnis wird auf zwei Register (**rdx**, **rax**) verteilt.
Der niederwertigere Teil befindet sich in **rax**.

GAS – Arithmetische und logische Operationen – mul, imul

Einschub __int128:

(Erklärung aus cppreference)
https://en.cppreference.com/w/cpp/language/integer_literal

"If the value of the integer literal is too big to fit in any of the types allowed by suffix/base combination and **the compiler supports extended integer types** (such as **__int128**) the literal may be given the extended integer type – otherwise the program is ill-formed."

GAS – Arithmetische und logische Operationen – mul, imul

Beispiel für die Verwendung von **mul** (C-Code):

```c
#include <inttypes.h>
typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
    *dest = x * (uint128_t)y;
}
```

Gleiches Beispiel in GNU Assembler:

```
 .section .text
  .globl store_uprod
  .type store_uprod, @function
# void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
# dest in rdi, x in rsi, y in rdx
store_uprod:
  movq %rsi, %rax
  mulq %rdx
  movq %rax, (%rdi)
  movq %rdx, 8(%rdi)
  ret
```

GAS – Arithmetische und logische Operationen – mul, imul

Gleicher GNU Assembler-Code mit Kommentaren:

```
  .text
   .globl  store_uprod
   .type store_uprod, @function
# void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
# dest in rdi, x in rsi, y in rdx
store_uprod:
   movq %rsi, %rax      # Copy x to multiplicand
   mulq %rdx            # Multiply by y
   movq %rax, (%rdi)    # Store lower 8 bytes at dest
   movq %rdx, 8(%rdi)   # Store upper 8 bytes at dest + 8
   ret
```

GAS – Arithmetische und logische Operationen – div, idiv

`idivq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Signed divide
`divq` *S*	R[`%rdx`] ← R[`%rdx`]:R[`%rax`] mod *S*; R[`%rax`] ← R[`%rdx`]:R[`%rax`] ÷ *S*	Unsigned divide

idivq und **divq**:

Es gibt keinen Modulo-Befehl in GNU Assembler.

Das Ergebnis verteilt sich auf die Register **rax** und **rdx.**

rax: Quotient

rdx: Rest (remainder)

GAS – Arithmetische und logische Operationen – div, idiv

Beispiel für die Verwendung von **div** (C-Code):

```
void remdiv(long x, long y, long *pq, long *pr) {
  long q = x / y;
  long r = x % y;
  *pq = q;
  *pr = r;
}
```

Gleiches Beispiel in GNU Assembler:

```
.globl  remdiv
  .type remdiv, @function
# void remdiv(long x, long y, long *pq, long *pr)
# x in rdi, y in rsi, pq in rdx, pr in rcx
remdiv:
  movq %rdx, %r8
  movq %rdi, %rax
  cqto
  idivq %rsi
  movq %rax, (%r8)
  movq %rdx, (%rcx)
  ret
```

Aufgabe:

Erklären Sie, weshalb der Inhalt von **%rdx** nach **%r8** gesichert wird, der Inhalt von **%rcx** aber gar nicht gesichert wird.

GAS – Arithmetische und logische Operationen – div, idiv

Gleiches Beispiel mit Kommentaren:

```
  .globl remdiv
   .type remdiv, @function
# void remdiv(long x, long y, long *pq, long *pr)
# x in rdi, y in rsi, pq in rdx, pr in rcx
remdiv:
   movq %rdx, %r8      # Copy pq
   movq %rdi, %rax     # Move x to lower 8 bytes of dividend
   cqto                # Sign extend to upper 8 bytes of dividend
   idivq %rsi          # Divide by y
   movq %rax, (%r8)    # Store quotient at pq
   movq %rdx, (%rcx)   # Store remainder at pr
   ret
```

Aufgabe – div, idiv

```
.globl  remdiv
  .type remdiv, @function
# void remdiv(long x, long y, long *qp, long *rp)
# x in rdi, y in rsi, qp in rdx, rp in rcx
remdiv:
  movq %rdx, %r8
  movq %rdi, %rax
  cqto
  idivq %rsi
  movq %rax, (%r8)
  movq %rdx, (%rcx)
  ret
```

Ändern Sie das obige Beispiel von **signed** in **unsigned** Division. Beachten Sie dabei, dass x und/oder y auch negative Zahlen enthalten können.

```
.globl  uremdiv
  .type uremdiv, @function
# void uremdiv(unsigned long x, unsigned long y,
#              unsigned long *qp, unsigned long *rp)
# x in rdi, y in rsi, qp in rdx, rp in rcx
uremdiv:
```