C nach Maschine

Sebastian Hack

Inhalt

- RISC Architekturen
- x86
- Aufbau von Binärdateien
- ABI/Funktionsaufruf
- Zahlenformate

RISC Architekturen

- Viele Register, meist 32
- Fixe Befehlslänge meist 32 Bit
- Drei-Operanden-Befehle: `add r1, r2, r3`
- Nur simple arithmetische & logische Befehle
- Speicherzugriff nur über `load`, `store`
- Prominente Vertreter: PowerPC, Arm, Mips, Sparc, Alpha
- Effizient implementierbar mit Fließband
- Ermöglicht Superskalarität

x86

- Architektur aus Kompatibilitätsgründen CISC
- Intern aber RISC Kern
- Umwandlung der CISC-Befehle in RISC-Befehle durch “Emulator” in Hardware
- Fazit: RISC hat sich durchgesetzt

x86

- Im wesentlichen acht Register

<table>
<tr><td>%eax</td><td rowspan="4">General Purpose (GP)</td></tr>
<tr><td>%ebx</td></tr>
<tr><td>%ecx</td></tr>
<tr><td>%edx</td></tr>
<tr><td>%esi</td><td rowspan="2">GP, aber spezielle Bedeutung bei String-Befehlen</td></tr>
<tr><td>%edi</td></tr>
<tr><td>%ebp</td><td>Schachtelzeiger</td></tr>
<tr><td>%esp</td><td>Kellerzeiger</td></tr>
</table>

- Gleitkomma-Register als Keller organisiert
- Zusätzl. Register: MMX, SSE, SSE2, etc.

x86

- Zwei-Operanden-Befehle:

  `add %eax, %ebx` ➞ `%ebx = %ebx + %eax`

- Fast alle Befehle können komplexe Speicheroperanden haben:

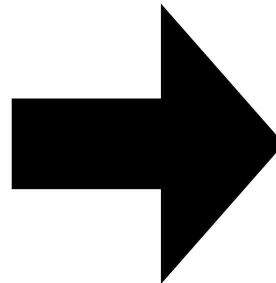
  `add %eax, 12(%ecx, %ebx, 4)`

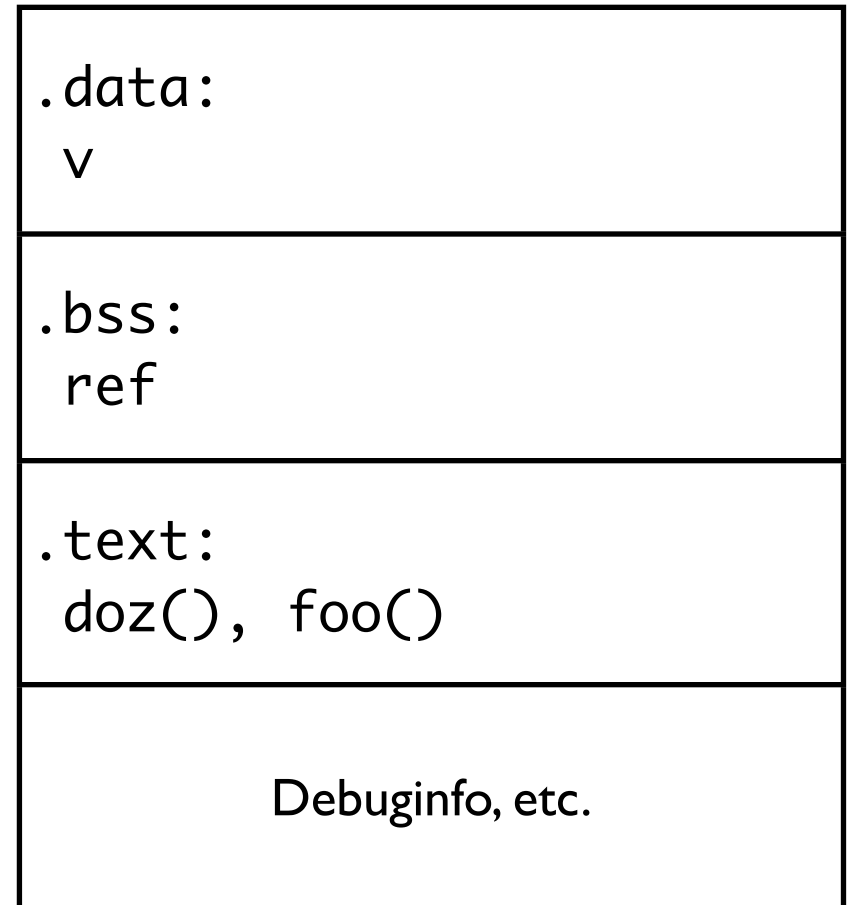
  ➞ `*(%eax + %ebx * 4 + 12) = %eax`

- Befehlslänge variabel

Übersetzungsprozess

```
extern int w;

int v = 3;

static int ref;

int doz(int a, int b) {
    int d = a - b;
    return d > 0 ? d : 0;
}

int foo(int x) {
    ref++;
    return v + w;
}
```





Abschnitte einer .o Datei

- Tabelle mit definierten & benötigten Symbolen
- data: (Konstant) initialisierte Daten;

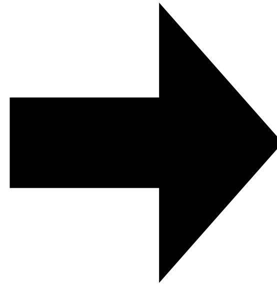
  Initialisierer werden im Binary gespeichert
- bss: Uninitialisierte Daten.

  Werden vom OS beim Start mit 0 initialisiert
- text: Maschinencode aller Funktionen
- Debuginfo, Read-Only-Data

Symbole (nm)

```
extern int w;

int v = 3;

static int ref;

int doz(int a, int b) {
    int d = a - b;
    return d > 0 ? d : 0;
}

int foo(int x) {
    ref++;
    return v + w;
}
```

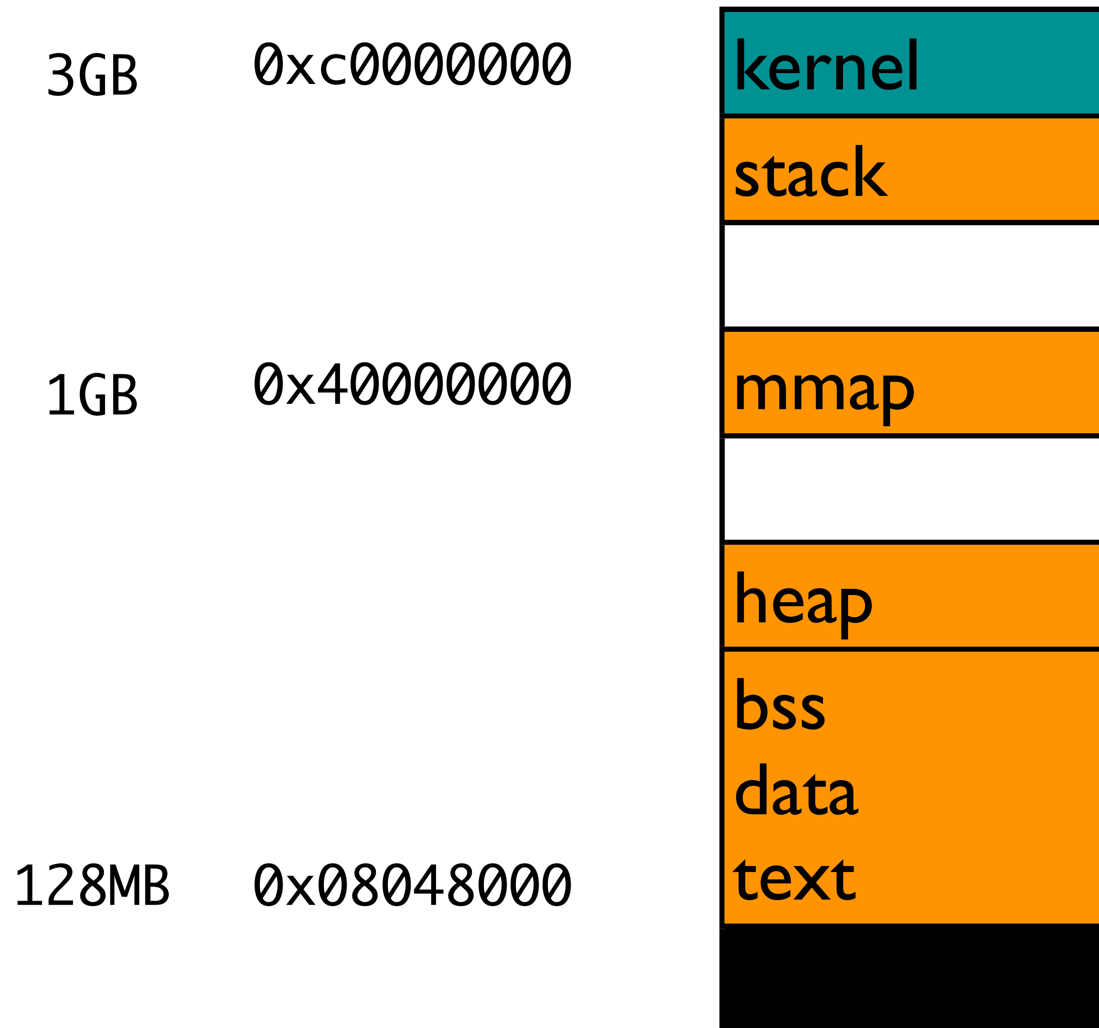
```
$ nm test.o
```



Adresse	Art	Symbol
00000000	T	doz
0000002b	T	foo
00000000	b	ref
00000000	D	v
	U	w

Art	Sichtbarkeit	Beschr.
T	global	Code
b	lokal	BSS
D	global	Data def.
U	global	Data undef.

Speicherbelegung in Linux zur Laufzeit



ABI

- Definiert den Aufruf von Funktionen
- Ermöglicht Interoperabilität von Binärcode
- Hier betrachtet: Linux/x86

Funktionsaufruf x86

- Übergabe der Parameter auf dem Keller
- Keller wächst von oben nach unten
- Letztes Argument zuerst (wegen . . .)
- Rückgabewert in %eax (%edx:%eax bei 64-bit)

doz

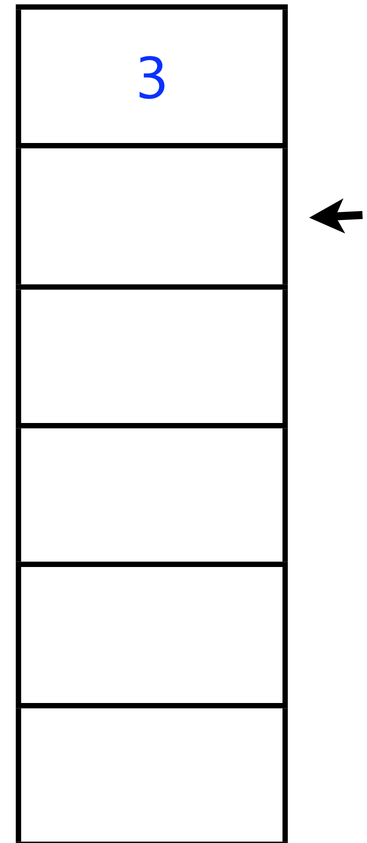
```
int doz(int a, int b) {
   int d;
   int res;

   d = a - b;

   if(d > 0)
      goto L2;
   res = 0;
L2:
   res = d;
   return res;
}
```

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    12(%ebp), %eax
        movl    8(%ebp), %edx
        ---------------------
        subl    %eax, %edx
        movl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, -8(%ebp)
        ---------------------
        cmpl    $0, -8(%ebp)
        jns     .L2
        ---------------------
        movl    $0, -8(%ebp)
        ---------------------
.L2:
        ---------------------
        movl    -8(%ebp), %eax
        ---------------------
        leave
        ret
```

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

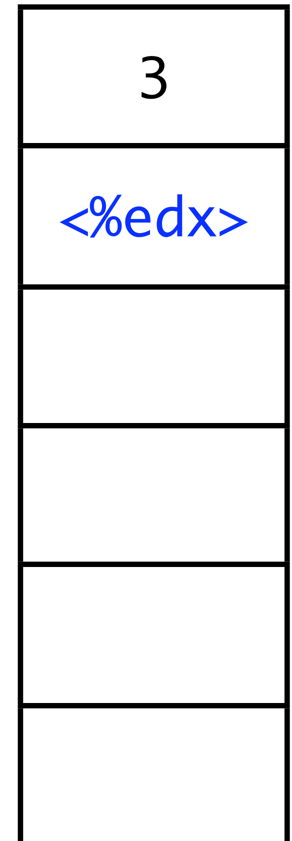


Letztes Argument auf den Keller

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

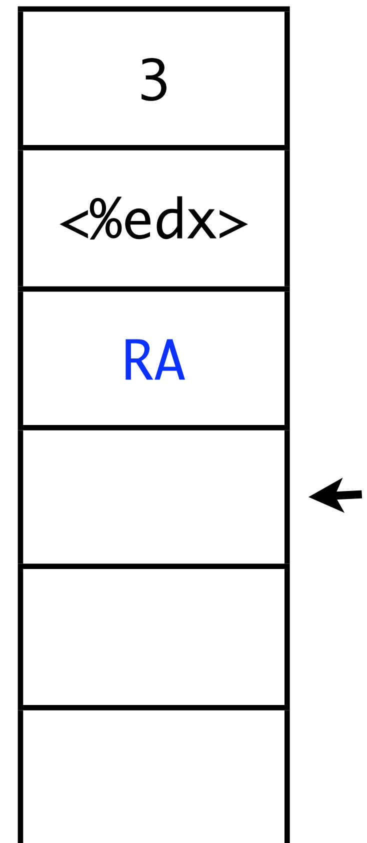


Näachstes Argument auf den Keller

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

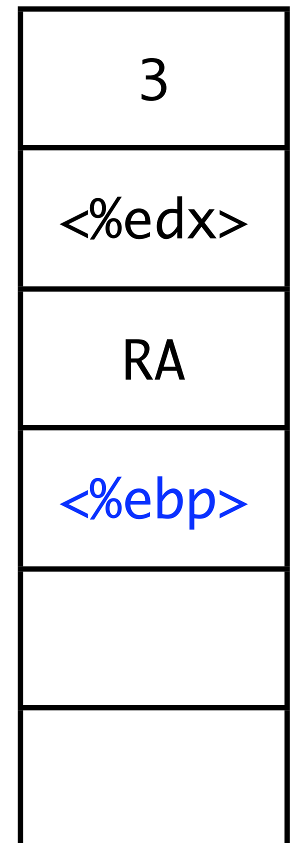


Rüchsprungadresse auf den Keller (implizit durch `call`)

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

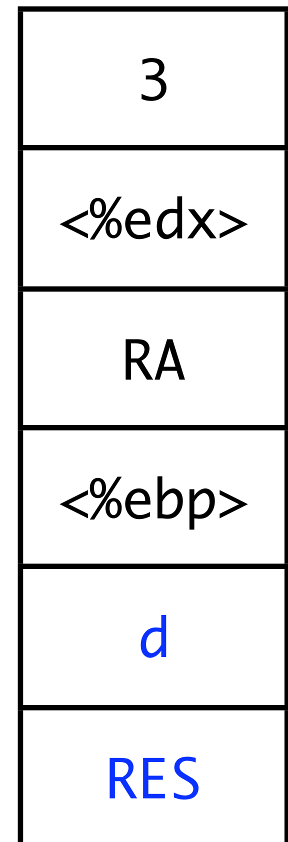


Sichern der Schachtelmarke des Aufrufers

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

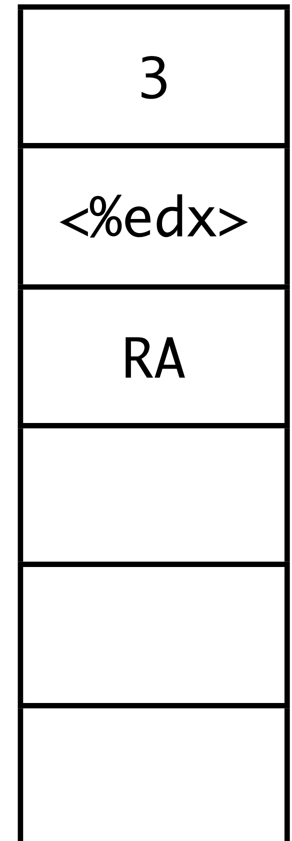


8 Bytes Platz für lokale Variable

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

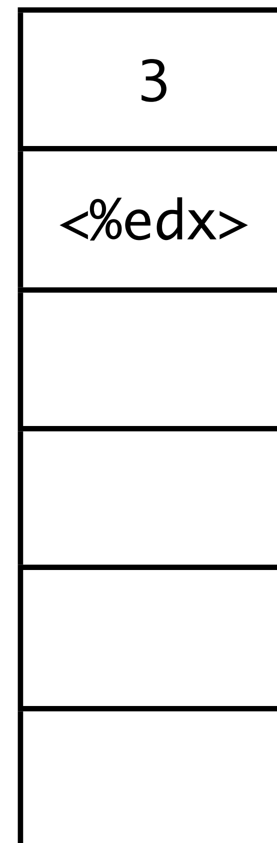


Laden des Ergebnisses in %eax

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
```

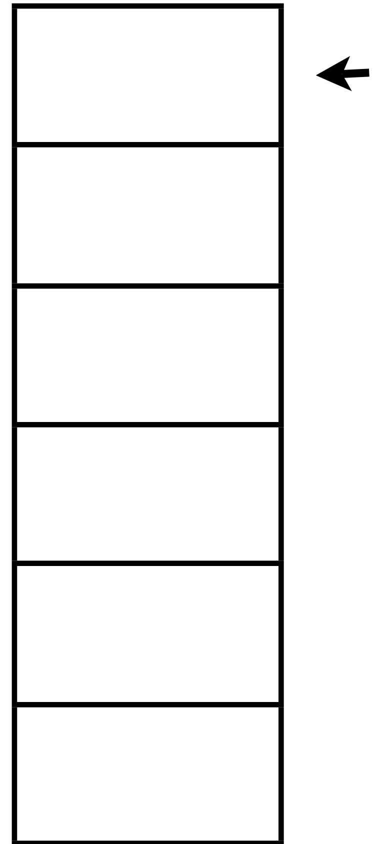


Holen der Rücksprungadresse vom Keller und Sprung

Funktionsaufruf

```
int doz(int a, int b) {
   int d = a - b;
   return d > 0 ? d : 0;
}
...
int bar(int x) {
  int a;
  ...
  a = doz(x, 3);
  ...
}
```

```
...
pushl $3
pushl %edx
call  doz
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
...
movl  -8(%ebp), %eax
leave
ret
...
add   $8, %esp
```



Freigabe des Argumente auf dem Keller

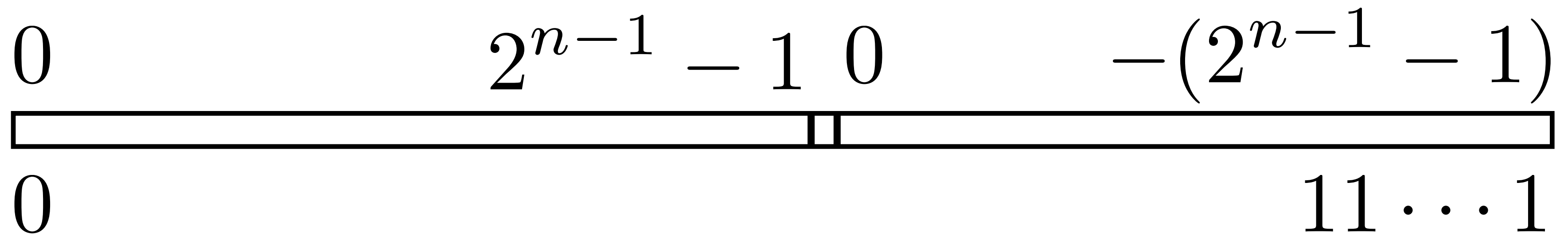
doz optimiert

```
movl    8(%esp), %ecx     2. Arg nach ecx
movl    4(%esp), %eax     1. Arg nach eax
subl    %ecx, %eax        eax = eax - ecx
movl    %eax, %ecx        Sichere Differenz nach ecx
xorl    $-1, %ecx         Invertiere ecx
sarl    $31, %ecx
andl    %ecx, %eax        Ergebis in eax
ret                       Kehre zurück
```

$$\texttt{ecx} = \begin{cases} 11\ldots1 & \texttt{ecx} \geq 0 \\ 0 & \texttt{ecx} < 0 \end{cases}$$

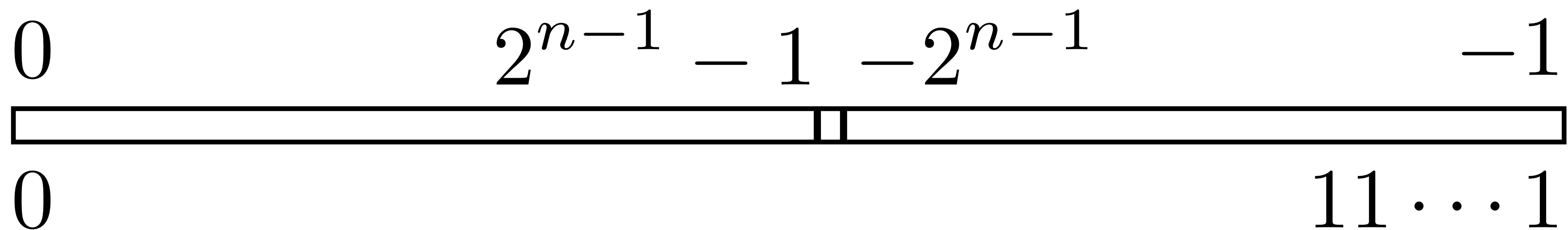
- Kein Anlegen der Schachtel
- Zwischenergebnisse in Registern
- Sprung durch Arithmetik ersetzt

Einerkomplement



- Zwei Nullen
- Addieren/Subtrahieren bei vorzeichenlos/-behaftet unterschiedlich!

Zweierkomplement



- Nur eine Null
- Addieren/Subtrahieren bei vorzeichenlos/-behaftet gleich
- Aber: Multiplikation/Division unterschiedlich!

Basis -2

000	0
001	1
010	-2
011	-1
100	4
101	5
110	2
111	3

- Algorithmen für Addition, Subtraktion und Multiplikation gleich
- Nur theoretisch von Interesse