

Hinweise – C-Programmierung

Dr.-Ing. Volkmar Sieh

Department Informatik 4
Systemsoftware
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2025/2026



das n -te Bit gesetzt:

`1 << n`

Maske für alle außer dem n -ten Bit:

`~(1 << n)`

Bit setzen → Verodern:

`x |= 0x01;`

Bit löschen → Verunden:

`x &= ~0x01;`

Bit umkehren → Exklusives Oder

`x ^= 0x01;`

niederwertiges Byte:

`x & 0xff`

nur niederwertiges Byte verändern:

`x &= ~0xff; x |= b & 0xff;`



C garantiert keine Größe von `short int`, `int`, `long int`, aber `inttypes.h` enthält folgende Definitionen:

- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`
- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`



Format-String für inttypes

Problem: %d bezieht sich auf int, %ld auf long int usw.

Für Variablen fester Größe können die Macros PRI_d8, PRI_u16, PRI_o32, PRI_x64 etc. verwendet werden.

Beispiel:

```
uint32_t x;  
  
x = 1234;  
printf("x hat den Wert %" PRIu32 "\n", x);
```



- Eine Funktion ist eine Adresse im .text-Segment
- Diese Adresse kann in C genutzt werden (z.B. für Callbacks)
- Deklaration für Funktionspointer:
Rückgabe-Typ (*Name)(Parameter-Typ, Parameter-Typ, ...)
- Beispiel: `void (*funptr)(int, long int);`
- Zuweisung: `funptr = myfunc;`
- Aufruf mittels: `(*funptr)(23, 42);`
- Einfacher: `funptr(23, 42);`



Problem: „Virtuelle Maschinen sind immer (zu) langsam.“

Fakten:

- moderne CPUs sind sehr schnell
(bis zu 4.000.000.000 Instruktionen pro Sekunde)
- Speicherzugriffe langsam (u.U. mehrere Tausend Instruktionen)
- Sprünge flushen ggf. die Instruktions-Pipeline (damit werden ggf. bis zu 50 Instruktionen verworfen)
- ...



Was heißt das für die Programmierung einer Virtuellen Maschine?

- Cache gut ausnutzen
 - Zustand der virtuellen CPU (Register inkl. Status-Wort und Inst.-Pointer)
 - Zustandsinfos der virtuellen I/O-Geräte
 - ...
 - häufig durchlaufener Simulations-Code sollte in möglichst wenigen Cache-Zeilen stehen
- Sprünge sollten vermieden werden
 - Inlining
 - „rechnen statt springen“
 - ...



Z.B.:

```
struct cpu_state {
    uint32_t ip;
    uint32_t flags;
    uint32_t eax;
    uint32_t ebx;
    ...
} __attribute__((aligned(128)));
```

Ergebnis: CPU-Zustandsinfo liegt (vermutlich) in ein oder zwei Cache-Zeilen.



schlecht

```
void
cpu_step(void) {
    if (! power) {
        /* Do nothing... */
    } else if (irq_pend) {
        cpu_step_irq();
    } else {
        cpu_step_inst();
    }
}
```

C-Compiler generiert vermutlich zwei (i.A. genommene) Sprungbefehle
(\Rightarrow Code vermutlich verteilt auf mehrere Cache-Zeilen)

gut

```
#define BE(a, b) \
    __builtin_expect(a, b)

void
cpu_step(void) {
    if (BE(! power, 0)) {
        /* Do nothing... */
    } else if (BE(irq_pend, 0)) {
        cpu_step_irq();
    } else {
        cpu_step_inst();
    }
}
```

C-Compiler generiert vermutlich zwei (i.A. nicht genommene) Sprungbefehle
(\Rightarrow Code vermutlich in ein/zwei Cache-Zeilen)



schlecht

```
uint32_t  
add(uint32_t a, uint32_t b) {  
    uint64_t res  
        = (uint64_t) a + b;  
    if (0x100000000 <= res) {  
        carry = 1;  
    } else {  
        carry = 0;  
    }  
    return (uint32_t) res;  
}
```

gut

```
uint32_t  
add(uint32_t a, uint32_t b) {  
    uint64_t res  
        = (uint64_t) a + b;  
    carry = (res >> 32) & 1;  
    return (uint32_t) res;  
}
```

Sprungvorhersage unzuverlässig,
da Carry-Bit nicht vorhersagbar
(=> viele Pipeline-Flushes).

Linearer Code
(=> keine Pipeline-Flushes).



Problematisch:

```
exec_inst() {  
    uint8_t inst = load(ip++);  
    switch (inst) {  
        case 0x00: ...; break;  
        case 0x01: ...; break;  
        ...  
        case 0xff: ...; break;  
    }  
}
```

Sprungvorhersage unzuverlässig, da immer neue Instuktionen
(=> viele Pipeline-Flushes).



Performance – Sprünge

schlecht

Datei x.c

```
...  
void  
bar(...)  
{  
    ...  
}  
...
```

gut

```
...  
static void  
bar(...)  
{  
    ...  
}
```

eine Datei

Datei y.c

```
...  
void  
foo(...)  
{  
    ...  
    bar(...);  
    ...  
}  
...
```

```
...  
void  
foo(...)  
{  
    ...  
    bar(...);  
    ...  
}
```

Nicht „static“ und andere Datei
(=> Inlining i.A. unmöglich).

„static“ und gleiche Datei
(=> Inlining möglich).



Performance – Sprünge

eventuell besser

```
uint32_t  
alu(  
    int cmd,  
    uint32_t a,  
    uint32_t b  
)  
{  
    uint32_t res;  
    switch (cmd) {  
        case 0: res = a + b; break;  
        case 1: res = a - b; break;  
        case 2: res = a * b; break;  
        case 3: res = a / b; break;  
    }  
    return res;  
}
```

inst immer wieder anders
(=> viele Pipeline-Flushes).

```
uint32_t  
alu(  
    int cmd,  
    uint32_t a,  
    uint32_t b  
)  
{  
    uint32_t res[4];  
    res[0] = a + b;  
    res[1] = a - b;  
    res[2] = a * b;  
    res[3] = a / b;  
    return res[cmd];  
}
```

linearer Code
(=> keine Pipeline-Flushes).

