

Má smysl rozumět assembleru?

LinuxDays 2026

Karel Kočí

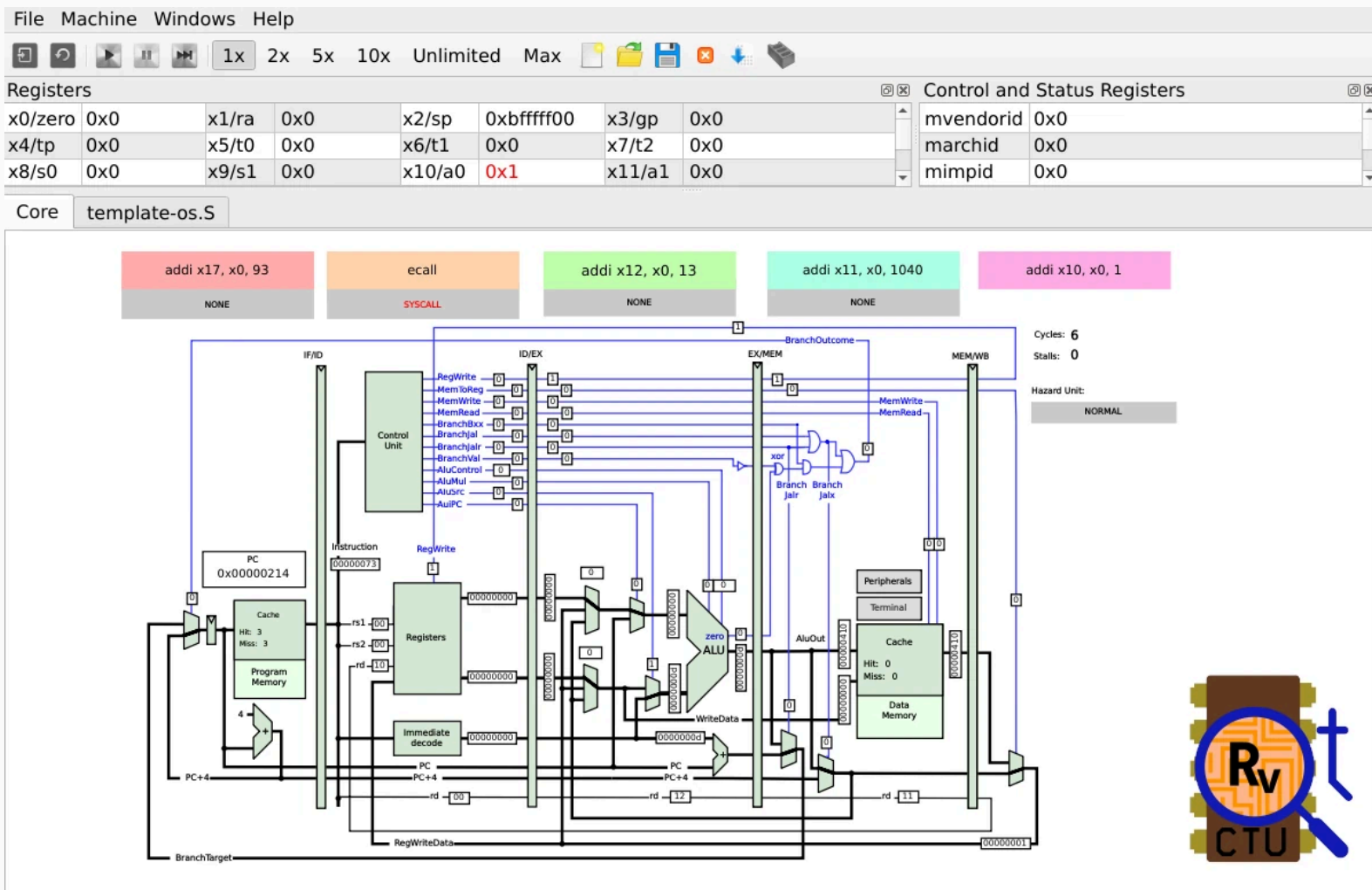
4.10.2026

<https://git.cynerd.cz/presentations/tree/2026-linuxdays>

Co je Assembler?

- Programovací jazyk
 - Syntaxe
 - Komentáře
 - Makra
- Čitelný zápis programového kódu
 - 1:1 asociace mezi instrukcemi
 - rozhraní mezi programem a procesorem

Crash course CPU



Má smysl rozumět assembleru?

Crash course Assembler

```
.globl _start  
.option norelax
```

```
.text
```

```
_start:
```

```
loop:
```

```
    lw      x2, 0x400(x0)
```

```
    sw      x2, 0x404(x0)
```

```
    beq     x0, x0, loop
```

```
    nop
```

```
    nop
```

```
    ebreak
```

```
.data
```

```
.org 0x400
```

```
src_val:
```

```
    .word   0x12345678
```

```
dst_val:
```

```
    .word   0
```

Proč tohle nefunguje?

```
void *buf = malloc(BUFSIZ);  
uint64_t *num = buf + 1;  
*num = 42;
```

```
$ gcc align.c -o align
```

```
$ ./align
```

```
$
```

```
$ armv7l-unknown-linux-gnueabihf-gcc -ggdb align.c -o align
```

```
$ ./align
```

```
qemu: uncaught target signal 7 (Bus error) - core dumped
```

```
$
```

Disassembly

```
$ armv7l-unknown-linux-gnueabihf-objdump -S align
```

```
...
```

```
void *buf = malloc(BUFSIZ);
```

```
6d4: e3a00a02  mov r0, #8192 @ 0x2000
```

```
6d8: ebffff97  bl  53c <malloc@plt>
```

```
6dc: e1a03000  mov r3, r0
```

```
6e0: e50b3008  str r3, [fp, #-8]
```

```
uint64_t *num = buf + 1;
```

```
6ec: e51b3008  ldr r3, [fp, #-8]
```

```
6f0: e2833001  add r3, r3, #1
```

```
6f4: e50b3010  str r3, [fp, #-16]
```

```
*num = 42;
```

```
6f8: e51b1010  ldr r1, [fp, #-16]
```

```
6fc: e3a0202a  mov r2, #42 @ 0x2a
```

```
700: e3a03000  mov r3, #0
```

```
704: e1c120f0  strd r2, [r1]
```

```
...
```

Má smysl rozumět assembleru?

Malá oprava

```
typedef uint64_t uint64_unaligned_t [[gnu::aligned(1)]];  
void *buf = malloc(BUFSIZ);  
uint64_unaligned_t *num = buf + 1;  
*num = 42;
```

```
uint64_unaligned_t *num = buf + 1;  
6ec: e51b3008 ldr r3, [fp, #-8]  
6f0: e2833001 add r3, r3, #1  
6f4: e50b3010 str r3, [fp, #-16]  
*num = 42;  
6f8: e51b3010 ldr r3, [fp, #-16]  
6fc: e3a0202a mov r2, #42 @ 0x2a  
700: e5832000 str r2, [r3]  
704: e3a02000 mov r2, #0  
708: e5832004 str r2, [r3, #4]
```

Použití speciálních instrukcí

Například SIMD:

```
typedef int v4si [[gnu::vector_size(16)]];

v4si v4add(v4si a, v4si b) {
    asm("padd %[%b], %[a]" : [a] "+x"(a) : [b] "x"(b));
    return a;
}
```

TLDR; Ano, vyplatí se ho znát.

Ale SIMD...

```
typedef int v4si [[gnu::vector_size(16)]];
v4si v4add2(v4si a, v4si b) { return a + b; }
```

000000000000000025 <v4add2>:

```
25: 55                push    %rbp
26: 48 89 e5          mov     %rsp,%rbp
29: 0f 29 45 f0       movaps  %xmm0,-0x10(%rbp)
2d: 0f 29 4d e0       movaps  %xmm1,-0x20(%rbp)
31: 66 0f 6f 45 f0    movdqa -0x10(%rbp),%xmm0
36: 66 0f fe 45 e0    paddb  -0x20(%rbp),%xmm0
3b: 5d                pop     %rbp
3c: c3                ret
```

A zarovnání...

- Python, JavaScript, ...
- Ehm... RUST! RUST! RUST! RUST! RUST!
- Btw.. v Go to taky bez unsafe nejde

A kdo dneska píše programy v assembleru?

TLDR; Takže ho nemusím znát?

Pochopení jak fungují struktury

```
struct px { int x, y;};  
int x_plus_y(struct px *p) {  
    return p→x + p→y;  
}
```

p→x + p→y;

66: 411c

68: 4148

6a: 9d3d

lw a5,0(a0)

lw a0,4(a0)

addw a0,a0,a5

Jak funguje volání funkce

```
add2(1, 2);
```

```
fc: 4589          li  a1,2
fe: 4505          li  a0,1
100: 00000097      auipc ra,0x0
104: 000080e7      jalr  ra # 100 <call_add+0xc>
```

```
add9(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

```
108: 47a5          li  a5,9
10a: e03e          sd  a5,0(sp)
10c: 48a1          li  a7,8
10e: 481d          li  a6,7
110: 4799          li  a5,6
112: 4715          li  a4,5
114: 4691          li  a3,4
116: 460d          li  a2,3
118: 4589          li  a1,2
11a: 4505          li  a0,1
11c: 00000097      auipc ra,0x0
120: 000080e7      jalr  ra # 11c <call_add+0x28>
```

Má smysl rozumět assembleru?

Staré zvyky

```
int div2_a(unsigned x) {  
    return x / 2;  
}
```

```
0: 1101          addi  sp,sp,-32  
2: ec06          sd    ra,24(sp)  
4: e822          sd    s0,16(sp)  
6: 1000          addi  s0,sp,32  
8: 87aa          mv    a5,a0  
a: fef42623      sw    a5,-20(s0)  
e: fec42783      lw    a5,-20(s0)  
12: 0017d79b     srliw a5,a5,0x1  
16: 2781          sext.w a5,a5  
18: 853e          mv    a0,a5  
1a: 60e2          ld    ra,24(sp)  
1c: 6442          ld    s0,16(sp)  
1e: 6105          addi  sp,sp,32  
20: 8082          ret
```

```
int div2_b(unsigned x) {  
    return x >> 1;  
}
```

```
22: 1101          addi  sp,sp,-32  
24: ec06          sd    ra,24(sp)  
26: e822          sd    s0,16(sp)  
28: 1000          addi  s0,sp,32  
2a: 87aa          mv    a5,a0  
2c: fef42623      sw    a5,-20(s0)  
30: fec42783      lw    a5,-20(s0)  
34: 0027d79b     srliw a5,a5,0x1  
38: 2781          sext.w a5,a5  
3a: 853e          mv    a0,a5  
3c: 60e2          ld    ra,24(sp)  
3e: 6442          ld    s0,16(sp)  
40: 6105          addi  sp,sp,32  
42: 8082          ret
```

Opravdu optimální?

```
int abs_a(int x) {  
    return x < 0 ? -x : x;  
}
```

```
52: fec42783          lw  a5,-20(s0)  
56: 41f7d79b          sraiw a5,a5,0x1f  
5a: fec42703          lw  a4,-20(s0)  
5e: 8f3d              xor  a4,a4,a5  
60: 40f707bb          subw a5,a4,a5  
64: 2781              sext.w a5,a5
```

```
int abs_b(int x) {  
    int m = x >> (sizeof(x) * CHAR_BIT - 1);  
    return (x + m) ^ m;  
}
```

```
7e: fdc42783          lw  a5,-36(s0)  
82: 41f7d79b          sraiw a5,a5,0x1f  
86: fef42623          sw  a5,-20(s0)  
8a: fdc42783          lw  a5,-36(s0)  
8e: 873e              mv  a4,a5  
90: fec42783          lw  a5,-20(s0)  
94: 9fb9              addw a5,a5,a4  
96: 2781              sext.w a5,a5  
98: fec42703          lw  a4,-20(s0)  
9c: 8fb9              xor  a5,a5,a4  
9e: 2781              sext.w a5,a5
```

Takže k čemu se hodí?

- Pochopení jak procesory fungují
- Optimalizují, nebo to kazím kompilátoru?
- -> Porozumění kódu (<https://godbolt.org/>)
- Debuggování kompilovaných jazyků, když už dojde na nehorší
- Implementace kritických sekcí v OS a embedded
- Vývoj kompilátoru (i JIT)

Děkuji za pozornost

Karel Kočí

<https://git.cynerd.cz>

<https://gitlab.com/cynerd>

<https://git.cynerd.cz/presentations/tree/2026-linuxdays>