Assembly

January 24, 2024

Übersicht

- Setup

  Dockerfile: github.com/Kernware/Presentations
- Einleitung per Beispiel
- Verschiedene Architekturen
- x86_64 Architektur
- Instruktionen
- Programmier Beispiele

Kurt Nistelberger

HTL Weiz
Informatik Studium TU Graz (LosFuzzys)

Denuvo by Irdeto (irdeto.com/denuvo)
Kernware (kernware.at)

github.com/gast04
kurt@kernware.at

C - Beispiel

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

C - Beispiel

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Kompilieren mit:

clang++ 1_sample.c -> ./a.out

C - Beispiel

```
1   ; godbolt.org
2   .data:
3       .string "Hello World"
4   main:
5       push    rbp
6       mov     rbp, rsp
7       mov     edi, OFFSET FLAT:.data
8       call    puts
9       mov     eax, 0
10      pop     rbp
11      ret
```

C - Beispiel

```
1   ; godbolt.org
2   .data:
3       .string "Hello World"
4   main:
5       push    rbp
6       mov     rbp, rsp
7       mov     edi, OFFSET FLAT:.data
8       call    puts
9       mov     eax, 0
10      pop     rbp
11      ret
```

Und nun?

C - Beispiel

```
1  .text:01140 ; int __cdecl main(int argc, const char **argv, const char **envp)
2  .text:01140
3  .text:01140 55                      push    rbp
4  .text:01141 48 89 E5                mov     rbp, rsp
5  .text:01144 48 83 EC 10             sub     rsp, 10h
6  .text:01148 C7 45 FC 00 00 00 00    mov     [rbp+var_4], 0
7  .text:0114F 48 8D 3D AE 0E 00 00    lea     rdi, format     ; "Hello World\n"
8  .text:01156 B0 00                   mov     al, 0
9  .text:01158 E8 D3 FE FF FF          call    _printf
10 .text:0115D 31 C0                   xor     eax, eax
11 .text:0115F 48 83 C4 10             add     rsp, 10h
12 .text:01163 5D                      pop     rbp
13 .text:01164 C3                      ret
```

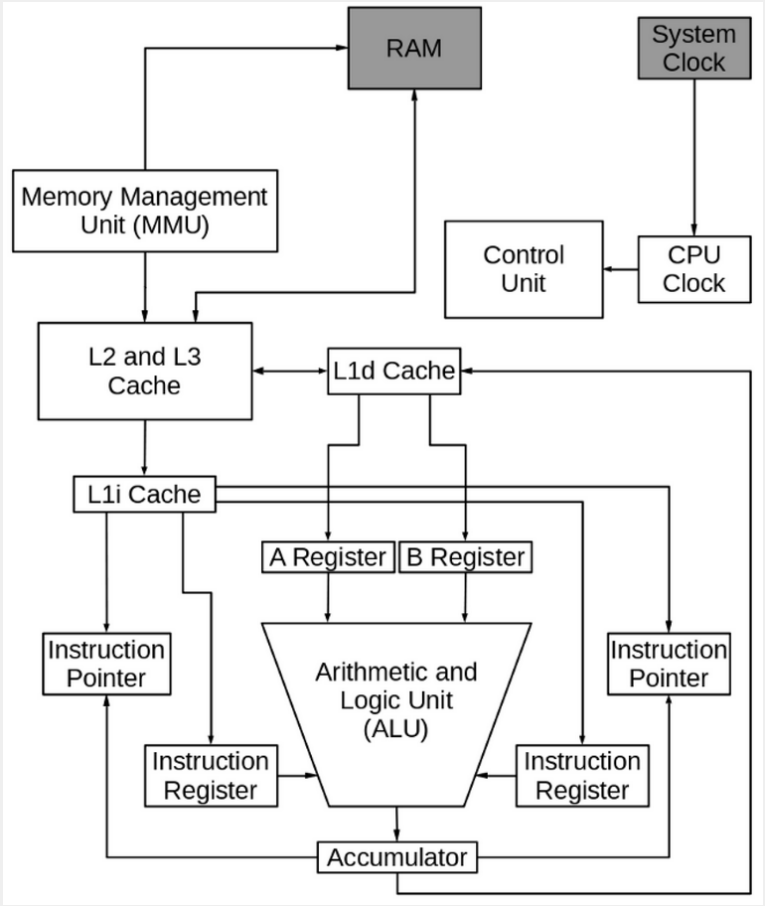
C - Beispiel

```
1  .text:01140 ; int __cdecl main(int argc, const char **argv, const char **envp)
2  .text:01140
3  .text:01140 55                      push    rbp
4  .text:01141 48 89 E5                mov     rbp, rsp
5  .text:01144 48 83 EC 10             sub     rsp, 10h
6  .text:01148 C7 45 FC 00 00 00 00    mov     [rbp+var_4], 0
7  .text:0114F 48 8D 3D AE 0E 00 00    lea     rdi, format     ; "Hello World\n"
8  .text:01156 B0 00                   mov     al, 0
9  .text:01158 E8 D3 FE FF FF          call    _printf
10 .text:0115D 31 C0                   xor     eax, eax
11 .text:0115F 48 83 C4 10             add     rsp, 10h
12 .text:01163 5D                      pop     rbp
13 .text:01164 C3                      ret
```

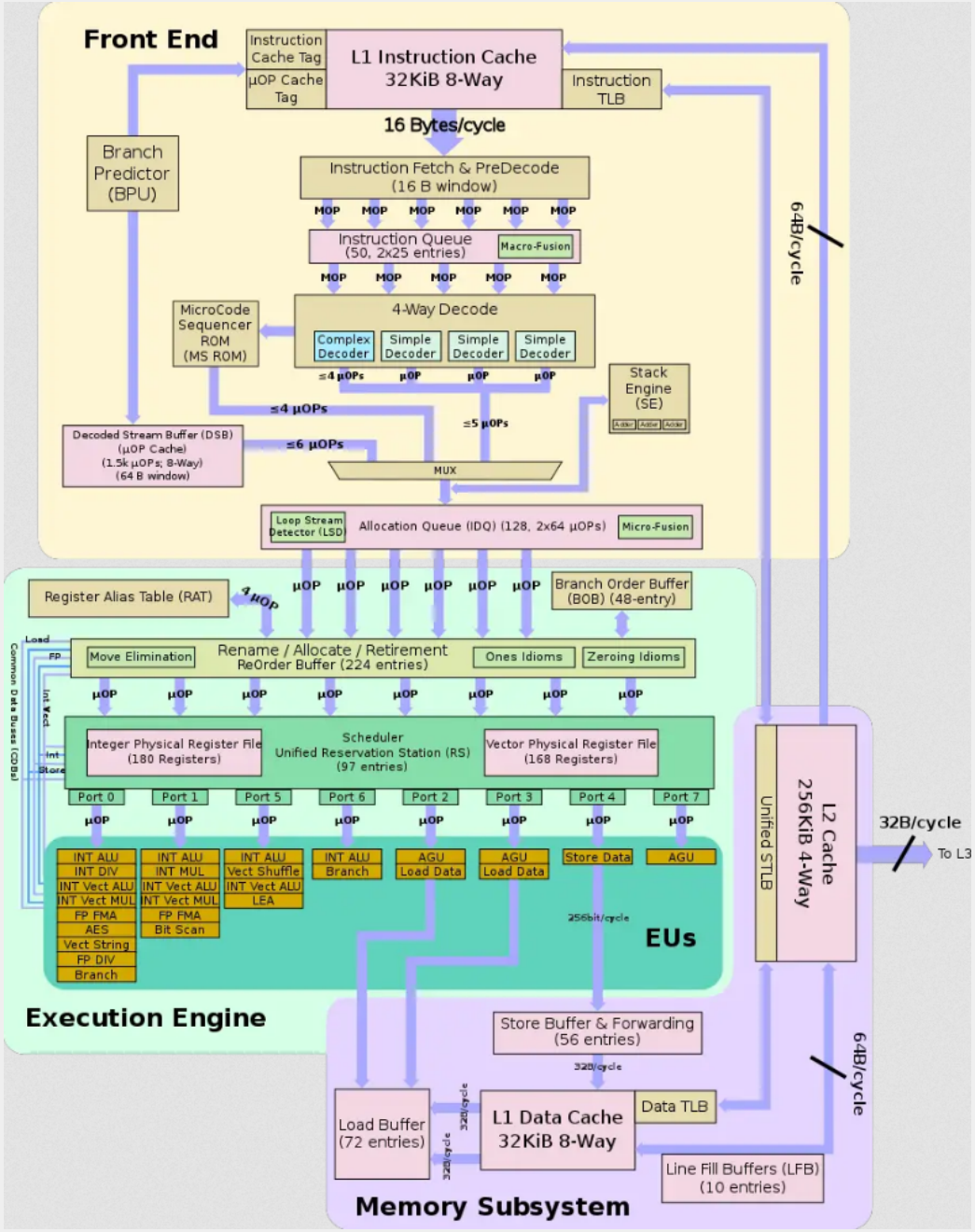
CPU macht: Fetch -> Decode -> Execute

CPU Architektur

```
1  55                          push   rbp
2  48 89 E5                    mov    rbp, rsp
3  48 83 EC 10                 sub    rsp, 10h
4  C7 45 FC 00 00 00 00        mov    [rbp+var_4], 0
5  48 8D 3D AE 0E 00 00        lea    rdi, format_str
6  B0 00                       mov    al, 0
7  E8 D3 FE FF FF              call   _printf
8  31 C0                       xor    eax, eax
9  48 83 C4 10                 add    rsp, 10h
10 5D                          pop    rbp
11 C3                          ret
```



[1] https://www.redhat.com/sysadmin/cpu-components-functionality



[1] https://en.wikichip.org/wiki/intel/microarchitectures/skylake

CPU Architektur(en)

```
1     addi    sp,sp,-16                ; RISC-V
2     sd      ra,8(sp)
3     sd      s0,0(sp)
4     addi    s0,sp,16
5     lui     a5,%hi(.LC0)
6     addi    a0,a5,%lo(.LC0)
7     call    puts
8     li      a5,0
9     mv      a0,a5
10    ld      ra,8(sp)
11    ld      s0,0(sp)
12    addi    sp,sp,16
13    jr      ra
```

CPU Architektur(en)

```
1     addi    sp,sp,-16                 ; RISC-V
2     sd      ra,8(sp)
3     sd      s0,0(sp)
4     addi    s0,sp,16
5     lui     a5,%hi(.LC0)
6     addi    a0,a5,%lo(.LC0)
7     call    puts
8     li      a5,0
9     mv      a0,a5
10    ld      ra,8(sp)
11    ld      s0,0(sp)
12    addi    sp,sp,16
13    jr      ra
```

```
1     push    {r11, lr}                 ; Arm-v7
2     mov     r11, sp
3     sub     sp, sp, #8
4     mov     r0, #0
5     str     r0, [sp]
6     str     r0, [sp, #4]
7     ldr     r0, .LCPI0_0
8     bl      printf
9     ldr     r0, [sp]
10    mov     sp, r11
11    pop     {r11, lr}
12    bx      lr
```

CPU Architektur(en)

```
1     stp     x29, x30, [sp, -16]!        ; Arm64
2     mov     x29, sp
3     adrp    x0, .LC0
4     add     x0, x0, :lo12:.LC0
5     bl      puts
6     mov     w0, 0
7     ldp     x29, x30, [sp], 16
8     ret
```

CPU Architektur(en)

```
1     stp     x29, x30, [sp, -16]!        ; Arm64
2     mov     x29, sp
3     adrp    x0, .LC0
4     add     x0, x0, :lo12:.LC0
5     bl      puts
6     mov     w0, 0
7     ldp     x29, x30, [sp], 16
8     ret
```

```
1     push r28                            ; AVR
2     push r29
3     in r28,__SP_L__
4     in r29,__SP_H__
5     ldi r24,lo8(.LC0)
6     ldi r25,hi8(.LC0)
7     rcall puts
8     ldi r24,0
9     ldi r25,0
10    pop r29
11    pop r28
12    ret
```

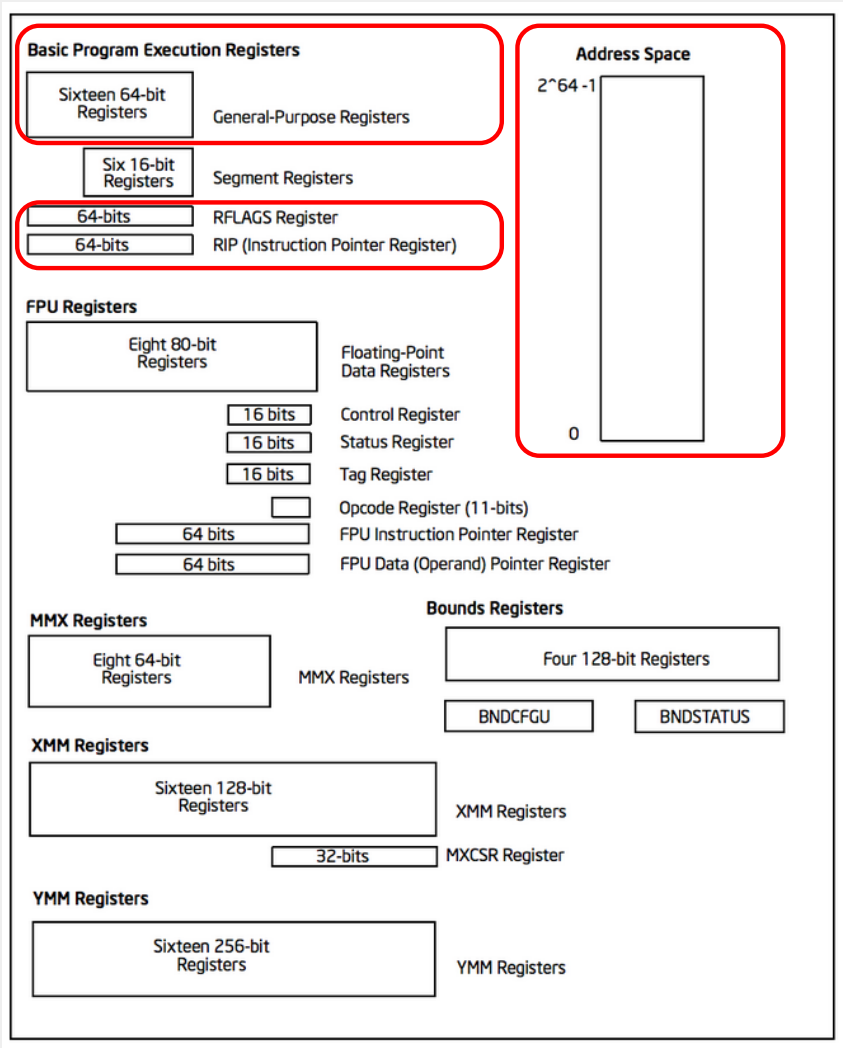
Architektur x86_64

- weitesten verbreitet

[1] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

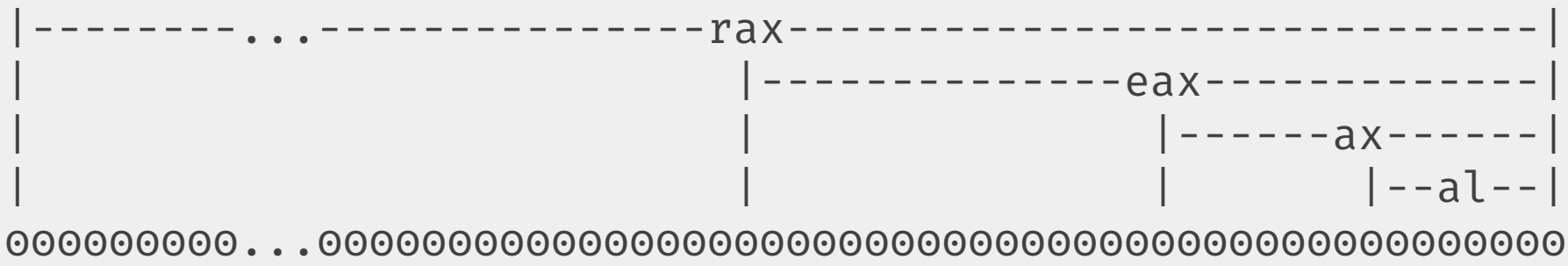
Architektur x86_64

- weitesten verbreitet



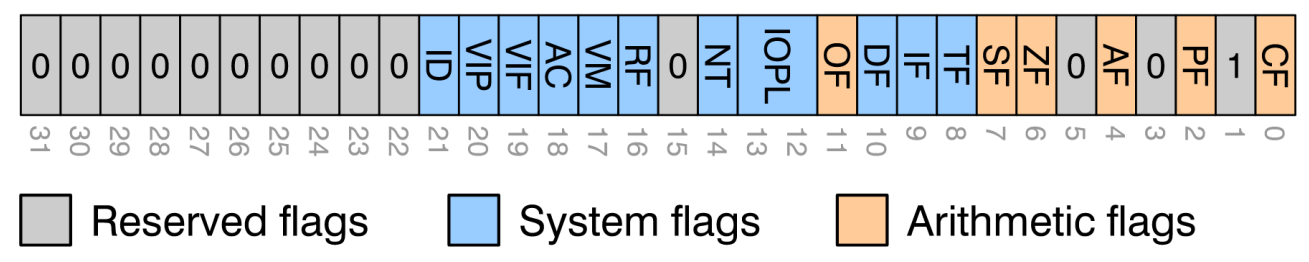
[1] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

Architektur x86_64 General Purpose Register



64bit	32bit	16bit	8bit
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rbp	ebp	bp	bpl
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b

Flag Register



ZF - Zero Flag

CMP Instruction

```
cmp rax, rbx
CF, OF, SF, ZF, AF, PF
```

CF = 1	unsigned rbx > unsigned rax
OF = 1	MSB rax != MSB result
SF = 1	MSB result == 1
ZF = 1	rax == rbx

JE, JZ	ZF == 1	rax == rbx
JNE, JNZ	ZF == 0	rax != rbx
JG	ZF == 0 & SF == OF	rax > rbx

Instruction Set

- Data Transfer (mov, ...)

```
1    mov rax, 0
2    mov rax, rbx
3    mov [rbp], 1
4    lea rax, [2]
```

Instruction Set

- Data Transfer (mov, ...)

```
1    mov rax, 0
2    mov rax, rbx
3    mov [rbp], 1
4    lea rax, [2]
```

- Binary Arithmetic (add, sub, inc, dec, mul, div, ...)

```
1    inc rax
2    add rax, rbx
3    sub rax, rbx
4    imul rax, rcx
5    idiv rbx  ; (RDX:RAX)/RBX, quotient in RAX, remainder in RDX
```

Instruction Set

- Data Transfer (mov, ...)

```
1    mov rax, 0
2    mov rax, rbx
3    mov [rbp], 1
4    lea rax, [2]
```

- Binary Arithmetic (add, sub, inc, dec, mul, div, ...)

```
1    inc rax
2    add rax, rbx
3    sub rax, rbx
4    imul rax, rcx
5    idiv rbx  ; (RDX:RAX)/RBX, quotient in RAX, remainder in RDX
```

- Logical (and, or, xor, ...)

```
1    and rdx, rcx
2    or  rax, rbx
3    xor rax, rax
4    not rcx
```

Instruction Set

- Data Transfer (mov, ...)

```
1    mov rax, 0
2    mov rax, rbx
3    mov [rbp], 1
4    lea rax, [2]
```

- Binary Arithmetic (add, sub, inc, dec, mul, div, ...)

```
1    inc rax
2    add rax, rbx
3    sub rax, rbx
4    imul rax, rcx
5    idiv rbx  ; (RDX:RAX)/RBX, quotient in RAX, remainder in RDX
```

- Logical (and, or, xor, ...)

```
1    and rdx, rcx
2    or  rax, rbx
3    xor rax, rax
4    not rcx
```

- Shift and rotate (sal, shl, sar, shr, ...)

```
1    shl rax, 3
2    shr rbx, 1
3    rol rdx, 4
```

Instruction Set

- Control transfer (jmp, jl, call, ret, ...)

```
1     cmp rax, rbx              ; sub rax, rbx
2
3     je equal_target
4     jne not_equal_traget
5     jg greater_target         ; Jump if RAX is greater than RBX
6     jle less_target           ; Jump if RAX is less than or equal to RBX
7
8     jz zero_target            ; Jump if ZF (zero flag) is set
9     jnz not_zero_target       ; Jump if ZF (zero flag) is not set
10
11    jmp unconditional_target
```

Instruction Set

- Control transfer (jmp, jl, call, ret, ...)

```
1     cmp rax, rbx              ; sub rax, rbx
2
3     je equal_target
4     jne not_equal_traget
5     jg greater_target         ; Jump if RAX is greater than RBX
6     jle less_target           ; Jump if RAX is less than or equal to RBX
7
8     jz zero_target            ; Jump if ZF (zero flag) is set
9     jnz not_zero_target       ; Jump if ZF (zero flag) is not set
10
11    jmp unconditional_target
```

- Viele mehr (Bit and Byte, I/O, Decimal Arithmetic, ...)

PUSH UND POP

```
1     mov rax, 5
2     push rax
3     pop rax
```

PUSH UND POP

```
1    mov rax, 5
2    push rax
3    pop rax
```

```
1    mov rax, 5
2    sub rsp, 8 ; push rax
3    mov [rsp], rax
4    xor rax, rax
5    mov rax, [rsp] ; pop rax
6    add rsp, 8
```

push und pop, Beispiel

```
1 -> mov rax, 5
2    sub rsp, 8 ; push rax
3    mov [rsp], rax
4    xor rax, rax
5    mov rax, [rsp] ; pop rax
6    add rsp, 8
7    nop
```

Register File:

RSP	0x100108
RAX	0x123456

Memory:

0x100100	0	
0x100108	0	(RSP)

push und pop, Beispiel

```
1     mov rax, 5
2  -> sub rsp, 8 ; push rax
3     mov [rsp], rax
4     xor rax, rax
5     mov rax, [rsp] ; pop rax
6     add rsp, 8
7     nop
```

Register File:

RSP	0x100108
RAX	5

Memory:

0x100100	0	
0x100108	0	(RSP)

push und pop, Beispiel

```
1     mov rax, 5
2     sub rsp, 8 ; push rax
3  -> mov [rsp], rax
4     xor rax, rax
5     mov rax, [rsp] ; pop rax
6     add rsp, 8
7     nop
```

Register File:

RSP	0x100100
RAX	5

Memory:

0x100100	0	(RSP)
0x100108	0	

push und pop, Beispiel

```
1     mov rax, 5
2     sub rsp, 8 ; push rax
3     mov [rsp], rax
4  -> xor rax, rax
5     mov rax, [rsp] ; pop rax
6     add rsp, 8
7     nop
```

Register File:

RSP	0x100100
RAX	5

Memory:

0x100100	5	(RSP)
0x100108	0	

push und pop, Beispiel

```
1     mov rax, 5
2     sub rsp, 8 ; push rax
3     mov [rsp], rax
4     xor rax, rax
5  -> mov rax, [rsp] ; pop rax
6     add rsp, 8
7     nop
```

Register File:

RSP	0x100100
RAX	0

Memory:

0x100100	5	(RSP)
0x100108	0	

push und pop, Beispiel

```
1     mov rax, 5
2     sub rsp, 8 ; push rax
3     mov [rsp], rax
4     xor rax, rax
5     mov rax, [rsp] ; pop rax
6  -> add rsp, 8
7     nop
```

Register File:

RSP	0x100100
RAX	5

Memory:

0x100100	5	(RSP)
0x100108	0	

push und pop, Beispiel

```
1      mov rax, 5
2      sub rsp, 8 ; push rax
3      mov [rsp], rax
4      xor rax, rax
5      mov rax, [rsp] ; pop rax
6      add rsp, 8
7  ->  nop
```

Register File:

RSP	0x100108
RAX	5

Memory:

0x100100	5	
0x100108	0	(RSP)

Erstes Beispiel

Schreib ein Schleife die genau 5 ma ausgeführt wird.
(http://asmdebugger.com/)

C PseudoCode:

```
int a = 5;
while(a > 0) {
    a--;
}
```

Achtung: Code muss mit ‘start:‘ anfangen und ‘nop‘ Instructions sind nicht unterstützt.

Erstes Beispiel

```
1     start:
2     mov rax, 0
3     mov rbx, 5
4     loop_start:
5         inc rax             ; Increment RAX
6         cmp rax, rbx        ; Compare RAX with loop count
7         jl loop_start       ; Jump if RAX < loop_count
8     exit:
```

Erstes Beispiel

```
1     start:
2     mov rax, 5
3     loop_start:
4         dec rax           ; Decrement RAX, sets ZF if o
5         jnz loop_start
6     exit:
```

Erstes Beispiel

```
1     start:
2     mov rcx, 5
3     loop_start:
4         loop loop_start ; Alias for "dec rcx; jnz"
5     exit:
```

Was macht folgender Code?

```
1     xor eax, eax
2     lea rbx, [o]
3     mov rdx, o
4     and esi, o
5     sub edi, edi
```

How to do it local?

```
1   ; hello.asm
2   global _start
3
4   _start:
5   mov rax, 0
6   mov rbx, 5
7
8   loop_start:
9       inc rax
10      cmp rax, rbx
11      jl loop_start
12  nop
```

How to do it local?

```
1     ; hello.asm
2     global _start
3
4     _start:
5     mov rax, 0
6     mov rbx, 5
7
8     loop_start:
9         inc rax
10        cmp rax, rbx
11        jl loop_start
12    nop
```

```
nasm -felf64 hello.asm & ld hello.o & ./a.out
```

Beispiel

C PseudoCode:

```
int a = 5;
int b = 3;
if (a > b) {
    b = a;
}
else {
    a = 0;
}
while(1){};
```

Beispiel

```
1   ; if.asm
2   global _start
3
4   _start:
5   mov rax, 5
6   mov rbx, 3
7
8   cmp rax, rbx
9   jg if_target
10  ; else_target
11  mov rax, 0
12  jmp while_target
13
14  if_target:
15  mov rbx, rax
16
17  while_target:
18  jmp while_target
```

Beispiel

C PseudoCode:

```
int getOne(int arg) {
    if (arg > 5) {
        return 1;
    }
    return 0;
};

int main() {
    int a = 42;
    int b = getOne(a);
    while(1){};
};
```

Beispiel

```
1   ; call.asm
2   global _start
3
4   getOne: ; argument stored in rax
5   cmp rax, 5
6   jg if_target
7   mov rax, 0
8   ret
9
10  if_target:
11  mov rax, 1
12  ret
13
14  _start:
15  mov rax, 42
16  call getOne
17  mov rbx, rax
18
19  while_target:
20  jmp while_target
```

Calling Convention

```
void callMe(int a, int b, int c, int d,
            int e, int f, int g, int h)
{
    // do something
}
```

Calling Convention

```
// Microsoft x64
func(int a, int b, int c, int d, int e, int f);
// RCX, RDX, R8, R9, everything else pushed on stack

// Linux
func(int a, int b, int c, int d, int e, int f);
// RDI, RSI, RDX, RCX, R8, R9
```

Beispiel

```
1   global _start ; challenge.asm
2   verify:
3   push  rbp
4   mov   rbp, rsp
5   sub   dword [rbp + 16], 0x18
6   xor   dword [rbp + 16], 0x7a69
7   not   dword [rbp + 16]
8   xor   dword [rbp + 16], 0x11e61
9   xor   dword [rbp + 16], 0x37
10  add   dword [rbp + 16], 0x26
11  mov   eax, dword [rbp + 16]
12  pop   rbp
13  ret
14
15  _start:
16  ; ... TODO
17  call verify
18  mov rbx, 0xfffe8906
19  cmp rax, rbx
20  jne while_target
21  lea rsi, [msg]
22  mov rdi, 1
23  mov rdx, 2
24  mov rax, 1
25  syscall
26
27  while_target:
28  jmp while_target
29
30  msg db ':', ')', 0
```

Beispiel

```
1   global _start ; challenge.asm
2   verify:
3   push  rbp
4   mov   rbp, rsp
5   sub   dword [rbp + 16], 0x18
6   xor   dword [rbp + 16], 0x7a69
7   not   dword [rbp + 16]
8   xor   dword [rbp + 16], 0x11e61
9   xor   dword [rbp + 16], 0x37
10  add   dword [rbp + 16], 0x26
11  mov   eax, dword [rbp + 16]
12  pop   rbp
13  ret
14
15  _start:
16  mov rax, 0x1338
17  push rax
18  call verify
19  mov rbx, 0xfffe8906
20  cmp rax, rbx
21  jne while_target
22  lea rsi, [msg]
23  mov rdi, 1
24  mov rdx, 2
25  mov rax, 1
26  syscall
27
28  while_target: jmp while_target
29
30  msg db ':', ')', 0
```

Beispiel

```
1   int verify(int arg) {
2       int ret = arg;
3       ret -= 0x18;
4       ret ^= 0x7a69;
5       ret = ~ret;
6       ret ^= 0x11e61;
7       ret ^= 0x37;
8       ret += 0x26;
9       return ret;
10  }
11
12  int main() {
13      int a = 0x1338;
14      int b = verify(a);
15      if (b == 0xfffe8906) {
16          printf(":)");
17      }
18      while(1){};
19  }
```

Zu guter Letzt

Vielen Dank!