Übung zu Betriebssysteme

Aufrufkonvention

Wintersemester 2021/22

Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

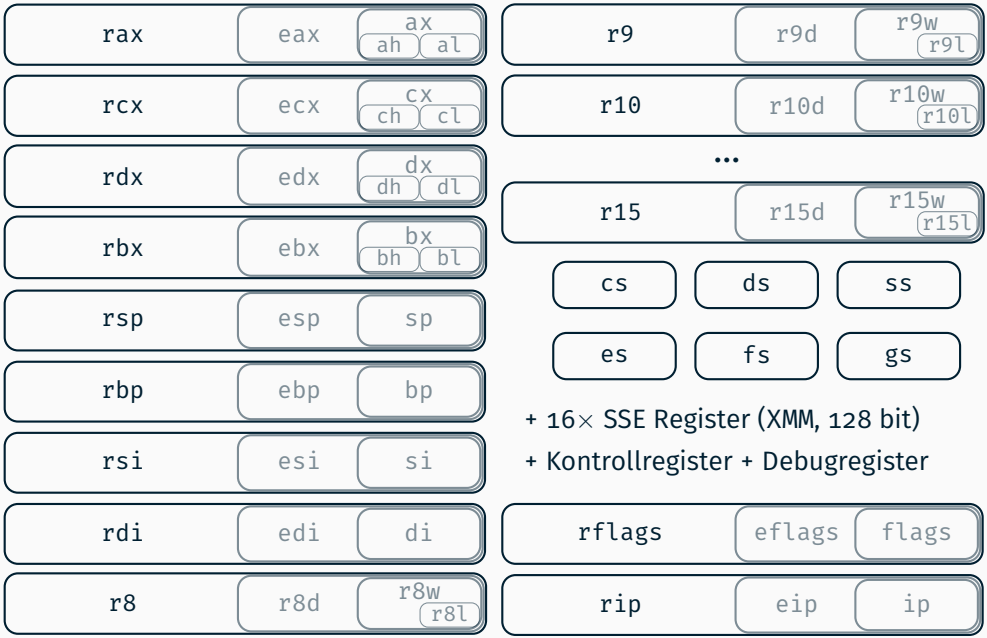
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Kontextsicherung gemäß Konvention

```
void baz(){
  ...

  // flüchtige Register
  // sichern

  func();

  // flüchtige Register
  // wiederherstellen

  ...
}
```

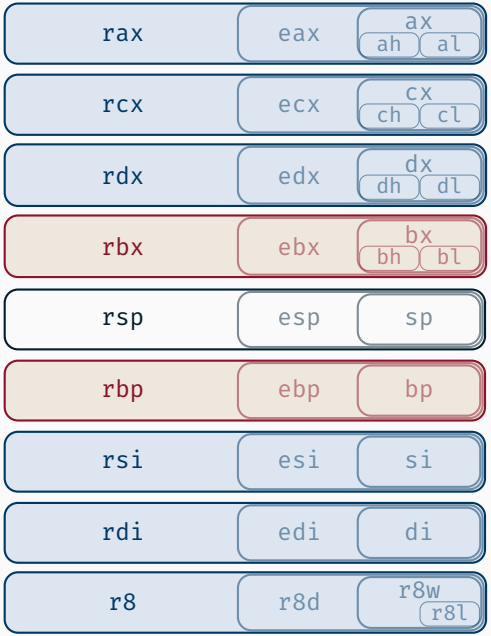
```
void func(){
  // nicht-flüchtige
  // Register
  // sichern

  ...

  // nicht-flüchtige
  // Register
  // wiederherstellen

  return;
}
```


rax
eax
ax
ah
al
rcx
ecx
cx
ch
cl
rdx
edx
dx
dh
dl
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
rdi
edi
di
r8
r8d
r8w
r8l
r9
r9d
r9w
r9l
r10
r10d
r10w
r10l
...
r15
r15d
r15w
r15l
cs
ds
ss
es
fs
gs
+ 16× SSE Register (XMM, 128 bit)
+ Kontrollregister + Debugregister
rflags
eflags
flags
rip
eip
ip

flüchtige (*scratch* / *caller-save*) Register

nicht-flüchtig (*non-scratch* / *callee-save*)

Wieso nicht gleich die **flüchtigen Register** beim Funktionsaufruf nutzen?


rax
eax
ax
ah
al
Rückgabewert
rcx
ecx
cx
ch
cl
4. Parameter
rdx
edx
dx
dh
dl
3. Parameter
rbx
ebx
bx
bh
bl
rsp
esp
sp
rbp
ebp
bp
rsi
esi
si
2. Parameter
rdi
edi
di
1. Parameter
r8
r8d
r8w
r8l
5. Parameter

Beispiel: Parameterübergabe gemäß Konvention

```
int i = func(23, 42);
```

```
; ggf. Register Sicherung
push r9

; 2. Parameter in Register rsi
mov esi, 0x2a

; 1. Parameter in Register rdi
mov edi, 0x17

; Funktionsaufruf
call func
```

Syntaxunterschiede bei x86-Assembler

Intel:

```
mov edi, 0x17
```

(Ziel, Quelle)

`objdump -M intel`

AT&T:

```
mov $0x17, %edi
```

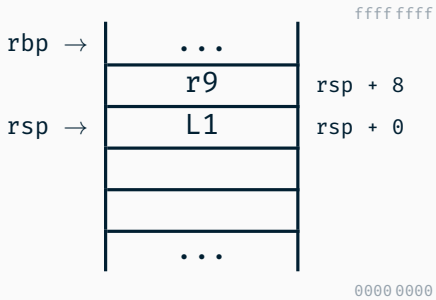
(Quelle, Ziel)

Standard bei `objdump`

```
int i = func(23, 42);
```

```
  ; ggf. Register Sicherung
  push r9

  ; 2. Parameter in Register rsi
  mov esi, 0x2a

  ; 1. Parameter in Register rdi
  mov edi, 0x17

  ; Funktionsaufruf
  call func
  ; pushed implizit die
  ; Rücksprungadresse
L1:
```

Stack beim Funktionsaufruf:



Zur Erinnerung – für den Stack auf x64 gilt

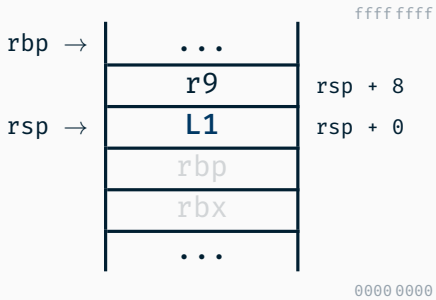
- Der Stack „wächst“ von *oben* (hohe Adresse) nach *unten* (niedrige Adresse)
- Der Stackzeiger (`rsp`) zeigt auf das zuletzt hinzugefügte Datum

  `push X` verringert zuerst den Wert von `rsp` um 8 Byte und legt dann `X` an die resultierende Adresse

  `pop X` liest den Inhalt des Speichers, auf das `rsp` zeigt, in `X` und erhöht anschließend den Wert von `rsp` um 8 Byte.

- Bei Funktionsaufrufen muss der Stack an 16 Byte ausgerichtet sein

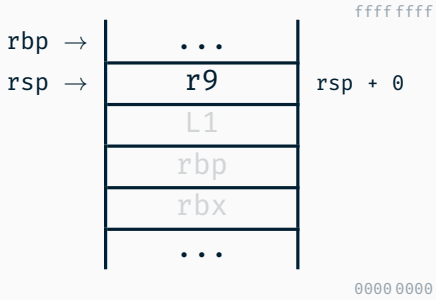
```
func:

  ; alten Rahmenzeiger sichern
  push rbp

  ; aktuellen Rahmenzeiger setzen
  mov rbp, rsp

  ; ggf. weitere Register sichern
  push rbx

  ...

  ; Rückgabewert nach rax
  mov eax, 0xd

  ; Register wiederherstellen
  pop rbx
  pop rbp

  ; Rücksprung
  ret
```



```
int i = func(23, 42);
```

```
  ; ggf. Register Sicherung
  push r9

  ; 2. Parameter in Register rsi
  mov esi, 0x2a

  ; 1. Parameter in Register rdi
  mov edi, 0x17

  ; Funktionsaufruf
  call func
  ; pushed implizit die
  ; Rücksprungadresse
L1:
  ; Rückgabewert in rax
  mov rsi, rax

  ; ggf. Register wiederherstellen
  pop r9
```

Stack nach Funktionsaufruf:



```
  ; Stackpointer setzen
  sub rsp, 0x18

  ; ggf. Register Sicherung
  mov [rsp + 0x8], r9

  ; 2. Parameter in Register rsi
  mov esi, 0x2a

  ; 1. Parameter in Register rdi
  mov edi, 0x17

  ; Funktionsaufruf
  call func
  ; pushed implizit die
  ; Rücksprungadresse
L1:
  ; Rückgabewert in rax
  mov rsi, rax

  ; ggf. Register wiederherstellen
  mov r9, [rsp + 0x8]
  add rsp, 0x18
```