# Grundbegriffe der Informatik

## Kapitel 22: Mima-X

Thomas Worsch

KIT, Institut für Theoretische Informatik

Wintersemester 2015/2016

# MIMA-X — eine Erweiterung der MIMA

MIMA aus Vorlesung «Rechnerorganisation»

- *sehr* einfach gehalten, damit z. B.
  auch Mikroprogrammierung im Detail erläuterbar

MIMA-X für «Programmieren» und «Softwaretechnik»

- komfortabler, damit z. B. auch
  Grundzüge der Übersetzung im Detail erläuterbar

alle fundamentalen Bestandteile und Konzepte
wie in realen Prozessoren

# Überblick

Motivation

Stapel

Mima-X

# Wo sind wir?

Motivation

Stapel

Mima-X

# Erinnerung: die Ackermann-Funktion $A : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$$\begin{aligned}
\forall y \in \mathbb{N}_0 : &\quad A(0, y) = y + 1 \\
\forall x \in \mathbb{N}_0 : &\quad A(x + 1, 0) = A(x, 1) \\
\forall x, y \in \mathbb{N}_0 : &\quad A(x + 1, y + 1) = A(x, A(x + 1, y))
\end{aligned}$$

Vorsicht beim Rechnen!

- schon bei kleinen Argumenten viele Rekursionsschritte

# Ackermann-Funktion Beispielberechnung für $A(2,2)$

$$
\begin{aligned}
& A(2,2) \\
&= A(1, A(2,1)) \\
&= A(1, A(1, A(2,0))) \\
&= A(1, A(1, A(1,1))) \\
&= A(1, A(1, A(0, A(1,0)))) \\
&= A(1, A(1, A(0, A(0,1)))) \\
&= A(1, A(1, A(0,2))) \\
&= A(1, A(1,3)) \\
&= A(1, A(0, A(1,2))) \\
&= A(1, A(0, A(0, A(1,1)))) \\
&= A(1, A(0, A(0, A(0, A(1,0))))) \\
&= A(1, A(0, A(0, A(0, A(0,1))))) \\
&= A(1, A(0, A(0, A(0,2))))
\end{aligned}
$$

$$
\begin{aligned}
&= A(1, A(0, A(0,3))) \\
&= A(1, A(0,4)) \\
&= A(1,5) \\
&= A(0, A(1,4)) \\
&= A(0, A(0, A(1,3))) \\
&= A(0, A(0, A(0, A(1,2)))) \\
&= A(0, A(0, A(0, A(0, A(1,1))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(1,0)))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0,1)))))) \\
&= A(0, A(0, A(0, A(0, A(0,2))))) \\
&= \cdots \\
&= A(0,6) \\
&= 7
\end{aligned}
$$

# Ackermann-Funktion $A(2, 2)$ kompakt notiert

$$
\begin{array}{ll}
\phantom{\rightsquigarrow}\ 2, 2 & \rightsquigarrow 1, 0, 0, 0, 2 \\
\rightsquigarrow 1, 2, 1 & \rightsquigarrow \cdots \\
\rightsquigarrow 1, 1, 2, 0 & \rightsquigarrow 1, 5 \\
\rightsquigarrow 1, 1, 1, 1 & \rightsquigarrow 0, 1, 4 \\
\rightsquigarrow 1, 1, 0, 1, 0 & \rightsquigarrow 0, 0, 1, 3 \\
\rightsquigarrow 1, 1, 0, 0, 1 & \rightsquigarrow 0, 0, 0, 1, 2 \\
\rightsquigarrow 1, 1, 0, 2 & \rightsquigarrow 0, 0, 0, 0, 1, 1 \\
\rightsquigarrow 1, 1, 3 & \rightsquigarrow 0, 0, 0, 0, 0, 1, 0 \\
\rightsquigarrow 1, 0, 1, 2 & \rightsquigarrow 0, 0, 0, 0, 0, 0, 1 \\
\rightsquigarrow 1, 0, 0, 1, 1 & \rightsquigarrow \cdots \\
\rightsquigarrow 1, 0, 0, 0, 1, 0 & \rightsquigarrow 0, 6 \\
\rightsquigarrow 1, 0, 0, 0, 0, 1 & \rightsquigarrow 7
\end{array}
$$

Liste von Werten

Änderungen nur am rechten Ende

ein/zwei Werte werden durch andere ersetzt, d. h.

- ein/zwei Werte entfernt
- neue Werte angefügt

Datenstruktur *Stapel* (vgl. Aufgabenblatt 5)

# Wo sind wir?

Motivation

Stapel

Mima-X

# Stapel oder Keller — Zugriff nur auf das oberste Element

Menge $S$ aller Stapel über Grundmenge $V$

- es gibt den leeren Stapel

Operationen, die einen Stapel verändern

$$\text{push}: S \times V \to S$$

$$\text{pop}: S \dashrightarrow S \quad \text{(für nichtleere Stapel)}$$

Operation, die Informationen aus Stapel extrahieren

$$\text{top}: S \dashrightarrow V \quad \text{(für nichtleere Stapel)}$$

Spezifikation

$$\forall s \in S\ \forall v \in V : \text{pop}(\text{push}(s, v)) = s$$

$$\forall s \in S\ \forall v \in V : \text{top}(\text{push}(s, v)) = v$$

# Stapel — eine mögliche „Implementierung“

Listen/Tupel von Elementen in $V$

$$S = \bigcup_{i \in \mathbb{N}_0} V^i$$

- leerer Stapel: leere Liste ()

Operationen

$$\begin{aligned}
\text{push}(\,()\,, v) &= (v) \\
\text{push}(\,(x_1, \ldots, x_n), v\,) &= (x_1, \ldots, x_n, v) \\
\text{pop}(\,()\,) &= \text{undefiniert} \\
\text{pop}(\,(x_1, \ldots, x_n)\,) &= (x_1, \ldots, x_{n-1}) \\
\text{top}(\,()\,) &= \text{undefiniert} \\
\text{top}(\,(x_1, \ldots, x_n)\,) &= x_n
\end{aligned}$$

# Stapel — bequeme Verallgemeinerung

in Prozessoren oft auch möglich:

$$\text{peek}\colon S \times \mathbb{N}_0 \dashrightarrow V$$

„Implementierung“

$$\text{peek}(\,(), k\,) = \text{undefiniert}$$

$$\text{peek}(\,(x_1, \ldots, x_n), k\,) = \begin{cases} x_{n-k}, & \text{falls } 0 \leq k < n \\ \text{undefiniert}, & \text{sonst} \end{cases}$$

# Berechnung der Ackermann-Funktion mit einem Stapel

$V = \mathbb{N}_0$

$$\begin{aligned} \forall y \in \mathbb{N}_0 : &\quad A(0, y) = y + 1 \\ \forall x \in \mathbb{N}_0 : &\quad A(x+1, 0) = A(x, 1) \\ \forall x, y \in \mathbb{N}_0 : &\quad A(x+1, y+1) = A(x, A(x+1, y)) \end{aligned}$$

definiere $A' : S \dashrightarrow S$ entsprechend falls $n \geq 2$:

$$\begin{aligned} A'(\,(x_1, \ldots, 0, x_n)\,) &= (\,x_1, \ldots, x_{n-2}, x_n + 1\,) \\ A'(\,(x_1, \ldots, x_{n-1} + 1, 0\,) &= (\,x_1, \ldots, x_{n-1}, 1\,) \\ A'(\,(x_1, \ldots, x_{n-1} + 1, x_n + 1\,) &= (\,x_1, \ldots, x_{n-1}, x_{n-1} + 1, x_n\,) \end{aligned}$$

Stapel-Manipulationen mit push, etc. ausdrückbar

# Jede $k$-stellige Operation auf $V$ ist auf Stapel mit mindestens $k$ Einträgen übertragbar

Idee

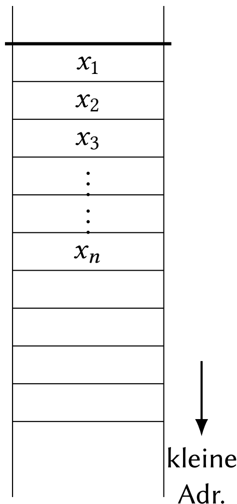
- Argumente mittels top und pop „vom Stapel holen“
- Operation anwenden
- Ergebnis mittels push auf den Stapel legen

Vorsicht bei nichtkommutativen Operationen

Beispiele

- Auswertung großer arithmetischer Ausdrücke in Programmiersprachen
- Taschenrechner mit „umgekehrter polnischer Notation“
  - nach dem polnischen Logiker Jan Lukasiewicz (1878–1956)
- Postscript

# Stapel — Implementierung in einem Rechner




Liste von Werten () oder $(x_1, \ldots, x_n)$
an aufeinander folgenden Adressen gespeichert

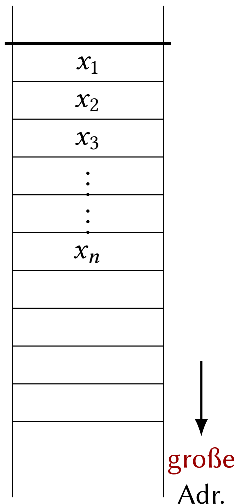
zwei Varianten

- Wachstum Richtung kleine Adressen
- Wachstum Richtung große Adressen (hier verwendet)
- genau lesen, was Autoren annehmen!

für Arbeit mit Stapel eine Adresse gespeichert
zwei Varianten

- Adresse, an der oberstes Element des Stapels liegt
- Adresse der nächsten freien Stelle
  (hier verwendet)
- genau lesen, was Autoren annehmen!

# Stapel — Implementierung in einem Rechner




Liste von Werten () oder $(x_1, \ldots, x_n)$
an aufeinander folgenden Adressen gespeichert

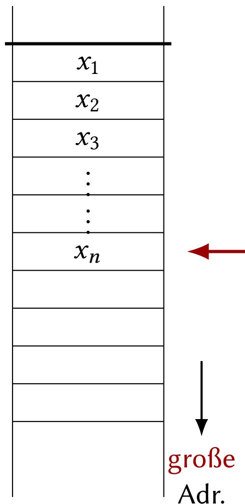
zwei Varianten

- Wachstum Richtung kleine Adressen
- Wachstum Richtung große Adressen (hier verwendet)
- genau lesen, was Autoren annehmen!

für Arbeit mit Stapel eine Adresse gespeichert
zwei Varianten

- Adresse, an der oberstes Element des Stapels liegt
- Adresse der nächsten freien Stelle
  (hier verwendet)
- genau lesen, was Autoren annehmen!

# Stapel — Implementierung in einem Rechner




Liste von Werten () oder $(x_1, \ldots, x_n)$
an aufeinander folgenden Adressen gespeichert

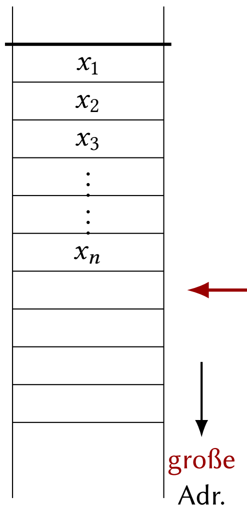
zwei Varianten

- Wachstum Richtung kleine Adressen
- Wachstum Richtung große Adressen (hier verwendet)
- genau lesen, was Autoren annehmen!

für Arbeit mit Stapel eine Adresse gespeichert
zwei Varianten

- Adresse, an der oberstes Element des Stapels liegt
- Adresse der nächsten freien Stelle
  (hier verwendet)
- genau lesen, was Autoren annehmen!

# Stapel — Implementierung in einem Rechner




Liste von Werten () oder $(x_1, \ldots, x_n)$
an aufeinander folgenden Adressen gespeichert

zwei Varianten

- Wachstum Richtung kleine Adressen
- Wachstum Richtung große Adressen (hier verwendet)
- genau lesen, was Autoren annehmen!

für Arbeit mit Stapel eine Adresse gespeichert
zwei Varianten

- Adresse, an der oberstes Element des Stapels liegt
- Adresse der nächsten freien Stelle
  (hier verwendet)
- genau lesen, was Autoren annehmen!

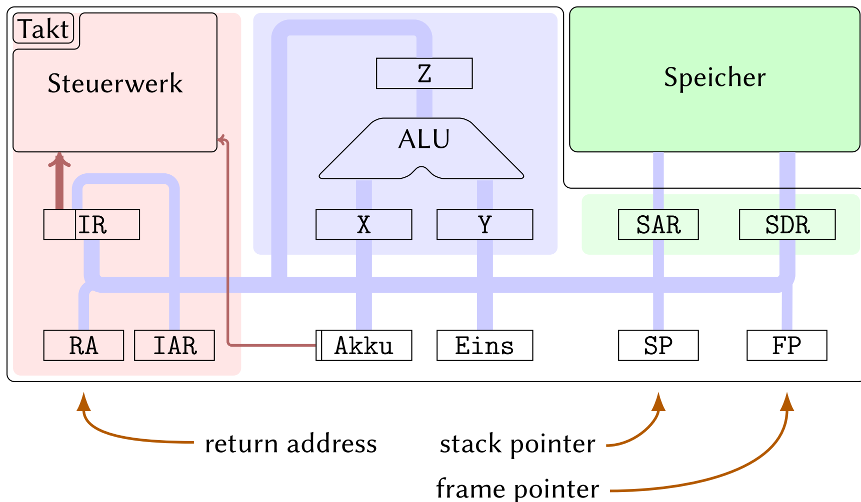
# Wo sind wir?

Motivation

Stapel

MIMA-X

# Mimax — drei zusätzliche Register für Adressen



# Register RA speichert eine Rückkehradresse

zwei neue Maschinenbefehle

`CALL` *adr* «*call subroutine at address*»

- ähnlich `JMP` *adr*
- zusätzlich wird der Inhalt von `IAR` in `RA` gespeichert

`RET` «*return from subroutine*»

- ähnlich `JMP` *adr*
- neuer Inhalt von `IAR` wird aus `RA` geholt

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | |
|---|---|
| *teil1:* | … Arbeit … |
| | `CALL` *sub* |
| *cont1:* | … weiter … |
| | |
| *teil2:* | … Arbeit … |
| | `CALL` *sub* |
| *cont2:* | … weiter … |
| | |
| *sub:* | … Arbeit … |
| | `RET` |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

```
teil1:   ... Arbeit ...
         CALL sub        RA ← cont1
cont1:   ... weiter ...

teil2:   ... Arbeit ...
         CALL sub
cont2:   ... weiter ...

sub:     ... Arbeit ...
         RET
```

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | CALL *sub* | RA ← *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | CALL *sub* | |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | RET | |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | `RA` $\leftarrow$ *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | |

# `CALL` und `RET` — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | `RA` $\leftarrow$ *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | Sprungziel aus `RA`, also *cont1* |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | `RA` $\leftarrow$ *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | |

# `CALL` und `RET` — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | |
|---|---|
| *teil1:* | … Arbeit … |
| | `CALL` *sub* |
| *cont1:* | … weiter … |
| | |
| *teil2:* | … Arbeit … |
| | `CALL` *sub* |
| *cont2:* | … weiter … |
| | |
| *sub:* | … Arbeit … |
| | `RET` |

# `CALL` und `RET` — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | |
|---|---|
| *teil1:* | … Arbeit … |
| | `CALL` *sub* |
| *cont1:* | … weiter … |
| | |
| *teil2:* | … Arbeit … |
| | `CALL` *sub* |
| *cont2:* | … weiter … |
| | |
| *sub:* | … Arbeit … |
| | `RET` |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | `RA` ← *cont2* |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | `RA` ← *cont2* |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

| | | |
|---|---|---|
| *teil1:* | … Arbeit … | |
| | `CALL` *sub* | |
| *cont1:* | … weiter … | |
| | | |
| *teil2:* | … Arbeit … | |
| | `CALL` *sub* | `RA` ← *cont2* |
| *cont2:* | … weiter … | |
| | | |
| *sub:* | … Arbeit … | |
| | `RET` | Sprungziel aus `RA`, also *cont2* |

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

```
teil1:   … Arbeit …
         CALL sub
cont1:   … weiter …

teil2:   … Arbeit …
         CALL sub
cont2:   … weiter …

sub:     … Arbeit …
         RET
```

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

|  |  |
|---|---|
| *teil1:* | … Arbeit … |
|  | CALL *sub* |
| *cont1:* | … weiter … |
|  |  |
| *teil2:* | … Arbeit … |
|  | CALL *sub* |
| *cont2:* | … weiter … |
|  |  |
| *sub:* | … Arbeit … |
|  | RET |

funktioniert so nur,
wenn zwischen *sub* und RET
kein CALL

# CALL und RET — Wiederverwendung von Codestücken durch primitiven Unterprogrammaufruf

```
teil1:  … Arbeit …
        CALL sub
cont1:  … weiter …

teil2:  … Arbeit …
        CALL sub
cont2:  … weiter …

sub:    … Arbeit …
        RET
```

funktioniert so nur,
wenn zwischen *sub* und `RET`
kein `CALL`

„Parameterübergabe“
nur in `Akku`

# SP und FP

FP: „frame pointer“ oder „Umgebungszeiger“

- technisch: genau so verwendbar wie SP
- inhaltlich:
  - Vorlesungen Programmieren und Softwaretechnik
  - hier keine weiteren Erläuterungen

SP: „stack pointer“ oder „Stapelpegel“

- zeigt auf freie Speicherstelle,
  in der nächstes push einen Wert ablegen würde
- Realisierung eines push
  - Ablegen eines Wertes an richtiger Speicherstelle und
  - Verändern des Inhalts von SP
- Realisierung eines top
  - Lesen eines Wertes von richtiger Speicherstelle und
- Realisierung eines pop
  - Verändern des Inhalts von SP

# Speicherzugriffe mittels SP

„relative Adressierung“ mittels Summe von

- Inhalt von SP
- vorzeichenbehaftetem konstanten Aneil

`LDVR` *disp*`(SP)` «*load value relative with displacement*»

- Beispiel: `LDVR -1(SP)`
- holt oberstes Stapelelement in den `Akku`

`STVR` *disp*`(SP)` «*store value relative with displacement*»

- Beispiel: `STVR 0(SP)`
- speichert Inhalt von `Akku` als neues oberstes Stapelelement

# Veränderungen des SP-Registers

Befehle für

- Transport zwischen `SP` und `Akku`
  - `LDSP` «*load from stack pointer*»
  - `STSP` «*load to stack pointer*»
- analog für `FP`
- analog für `RA`

Befehl zum Addieren einer vorzeichenbehafteten Konstanten

- `ADC` *const* «*add constant to* `Akku`»

damit zum Beispiel Erhöhung des Stack Pointers durch

```
LDSP
ADC 1
STSP
```

# Realisierung von push, top und pop

push

```
STVR 0(SP)
LDSP
ADC 1
STSP
```

top

```
LDVR -1(SP)
```

pop

```
LDSP
ADC -1
STSP
```

# push und pop von RA — für ineinander geschachtelte CALL

*main:* … Arbeit …
`CALL` *sub1*
*cont1:* … weiter …

*sub1:* … Arbeit …
lege `RA` auf Stapel
`CALL` *sub2*
*cont2:* hole `RA` von Stapel
… weiter …
`RET`

*sub2:* … Arbeit …
`RET`

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | `RA` ← *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | |
| | `CALL` *sub2* | |
| *cont2:* | hole `RA` von Stapel | |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von `RA` — für ineinander geschachtelte `CALL`

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | `RA` ← *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | |
| | `CALL` *sub2* | |
| *cont2:* | hole `RA` von Stapel | |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | `RA` ← *cont1* |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | Stack: *cont1* |
| | `CALL` *sub2* | |
| *cont2:* | hole `RA` von Stapel | |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | Stack: *cont1* |
| | `CALL` *sub2* | `RA` ← *cont2* |
| *cont2:* | hole `RA` von Stapel | |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | CALL *sub1* | |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege RA auf Stapel | Stack: *cont1* |
| | CALL *sub2* | RA ← *cont2* |
| *cont2:* | hole RA von Stapel | |
| | … weiter … | |
| | RET | |
| | | |
| *sub2:* | … Arbeit … | |
| | RET | |

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | Stack: *cont1* |
| | `CALL` *sub2* | `RA` ← *cont2* |
| *cont2:* | hole `RA` von Stapel | |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | Sprungziel aus `RA`, also *cont2* |

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | Stack: *cont1* |
| | `CALL` *sub2* | |
| *cont2:* | hole `RA` von Stapel | `RA` ← *cont1* |
| | … weiter … | |
| | `RET` | |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von RA — für ineinander geschachtelte CALL

```
main:   … Arbeit …
        CALL sub1
cont1:  … weiter …

sub1:   … Arbeit …
        lege RA auf Stapel
        CALL sub2
cont2:  hole RA von Stapel    RA ← cont1
        … weiter …
        RET

sub2:   … Arbeit …
        RET
```

# push und pop von RA — für ineinander geschachtelte CALL

| | | |
|---|---|---|
| *main:* | … Arbeit … | |
| | `CALL` *sub1* | |
| *cont1:* | … weiter … | |
| | | |
| *sub1:* | … Arbeit … | |
| | lege `RA` auf Stapel | |
| | `CALL` *sub2* | |
| *cont2:* | hole `RA` von Stapel | `RA` $\leftarrow$ *cont1* |
| | … weiter … | |
| | `RET` | Sprungziel aus `RA`, also *cont1* |
| | | |
| *sub2:* | … Arbeit … | |
| | `RET` | |

# push und pop von RA — für ineinander geschachtelte CALL

| | |
|---|---|
| *main:* | … Arbeit … |
| | `CALL` *sub1* |
| *cont1:* | … weiter … |
| | |
| *sub1:* | … Arbeit … |
| | lege `RA` auf Stapel |
| | `CALL` *sub2* |
| *cont2:* | hole `RA` von Stapel |
| | … weiter … |
| | `RET` |
| | |
| *sub2:* | … Arbeit … |
| | `RET` |

# Wir halten fest

## Das sollten Sie mitnehmen:

- zusätzliche Register und
- zusätzliche Maschinenbefehle
- erlauben Realisierung eines Stapels
  - z. B. von Rücksprungadressen
    für geschachelte Unterprogrammaufrufe