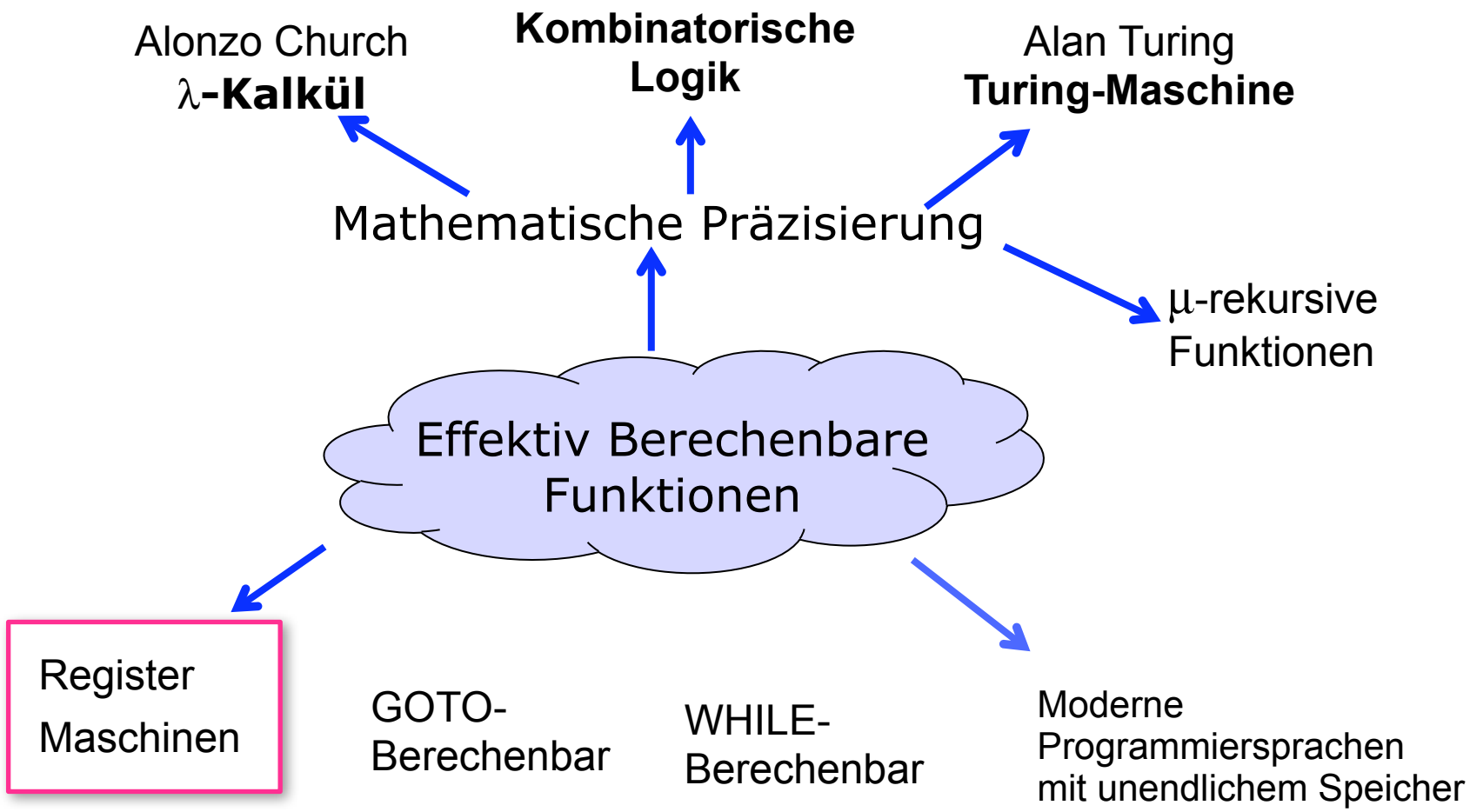
ALP II

Register Maschinen

SS 2012

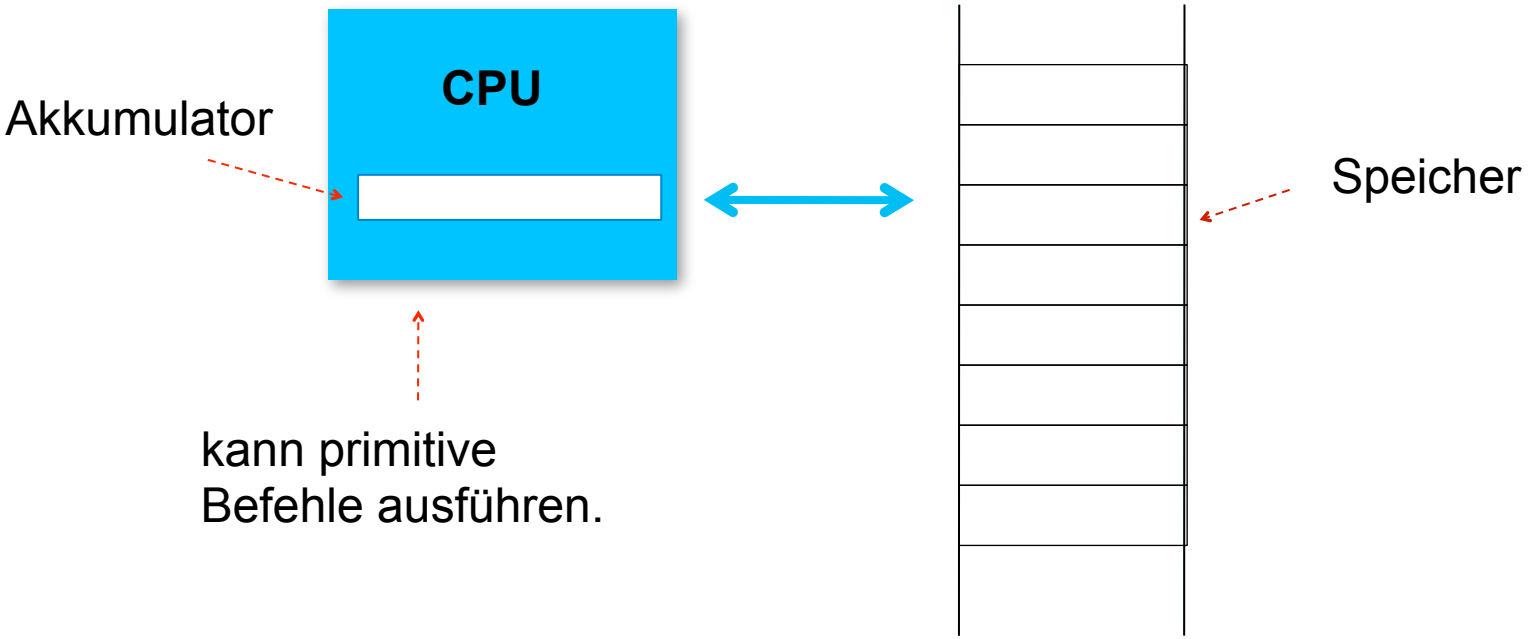
Prof. Dr. Margarita Esponda

Äquivalenz vieler Berechnungsmodelle



Register-Maschinen

- sind eine weitere Alternative der Lösung der effektiven berechenbaren Funktionen.
- angenommen, wir haben immer genug Speicher



Register-Maschinen

Was ist der minimale Befehlssatz?

- Patterson and Hennessy (Stanford University)
  - ein Befehl

    subleq ***a, b, c***

  - subtrahiert die Zahlen, die sich in den Speicheradressen ***a*** und ***b*** befinden, und wenn das Ergebnis kleiner gleich Null ist, springt die Ausführung des Programms zur Adresse ***c***.

Register-Maschinen

Beispiel:

R. Rojas *"Minimal Instruction Set Computers and the Power of Self-Modifying Programs."*

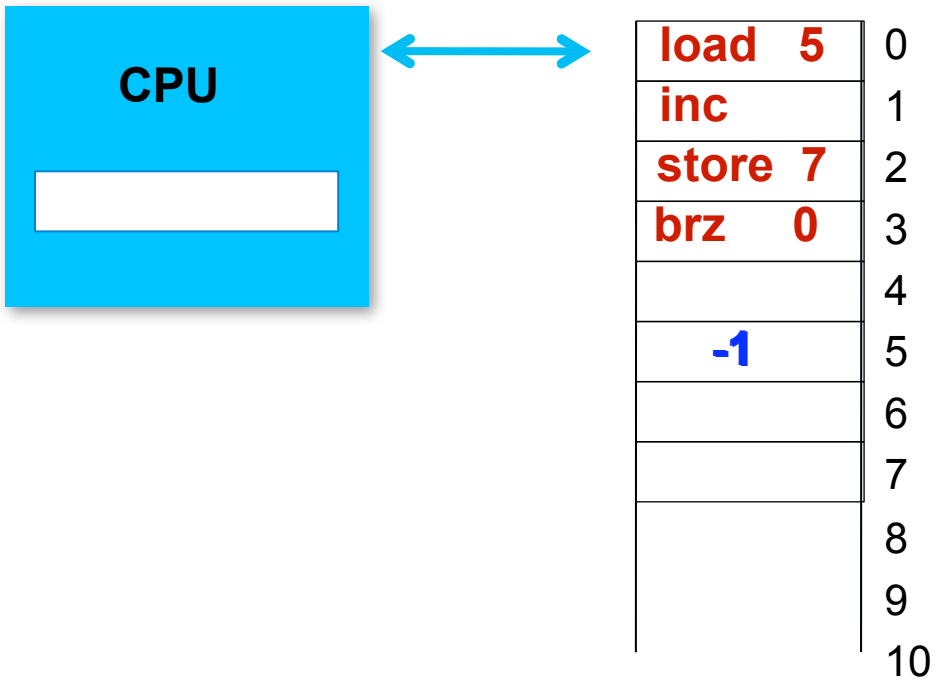
- Akkumulator mit **n**-Bits (endliche Große)
- **5** Befehle
- davon 3 Befehle für Speicheradressierung
- als einzige arithmetische Operation **(+1)**
- und **CLR** (schreibt eine Null im Akkumulator)

Register-Maschine

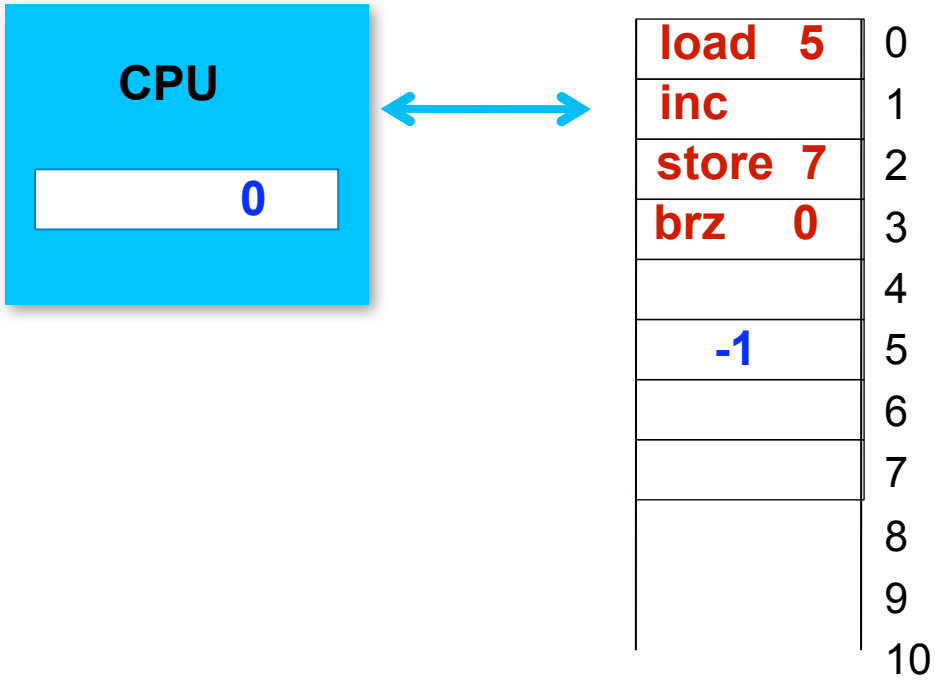
- 5 Befehle
  - load A -- kopiert den Inhalt der Speicheradresse A zum CPU-Akkumulator
  - store A -- kopiert den Inhalt des Akkumulators zur Speicheradresse A
  - inc -- inkrementiert den Akkumulator um 1
  - brz B -- wenn der Akkumulator gleich Null ist, springt die Ausführung des Programms zur Adresse B
  - clr -- schreibt eine Null in dem Akkumulator

Die Programme werden sequenziell ausgeführt, bis ein Sprungbefehl stattfindet (brz).

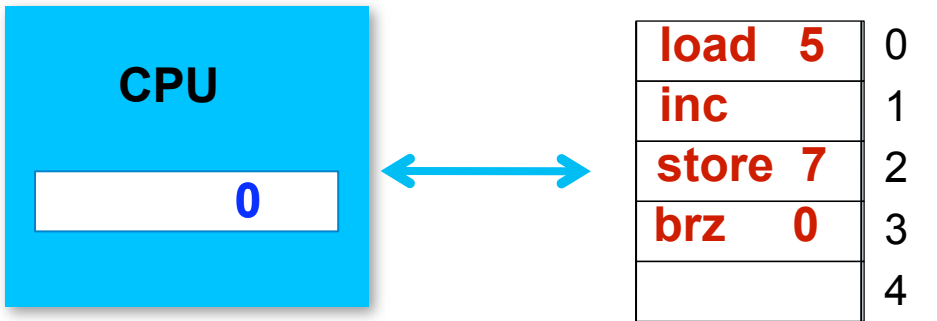
Register-Maschine



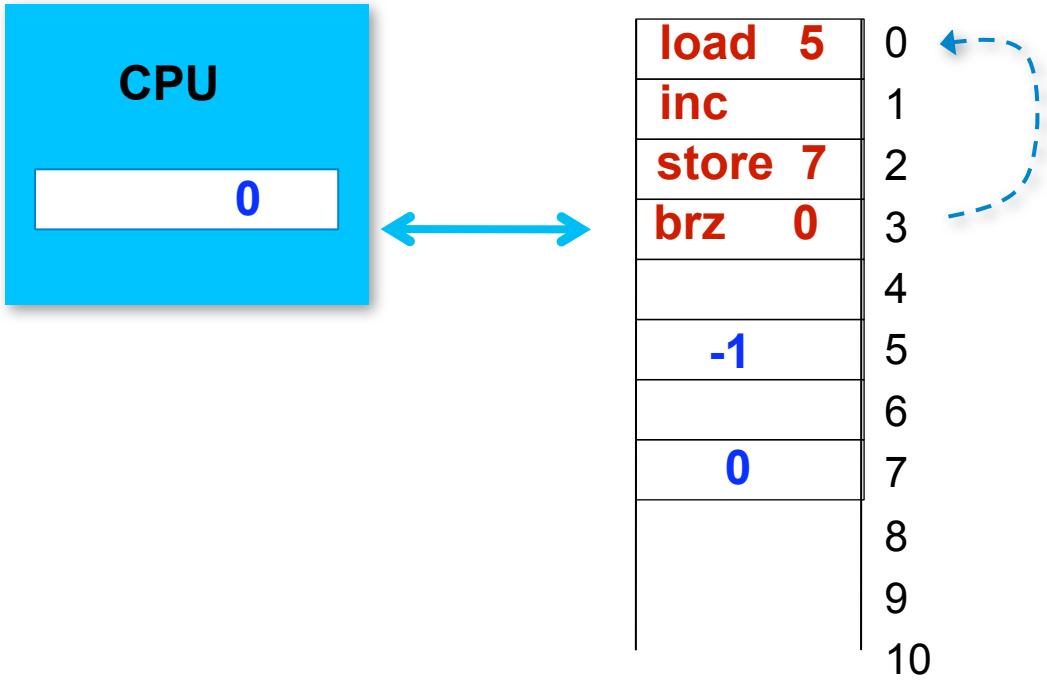
Register-Maschine



Register-Maschine



Register-Maschine



Macros

-- löscht eine beliebige Speicheradresse

```
CLR  A   ≡  |  CLR
            |  STORE  A
```

-- kopiert den Inhalt von Adresse A in Adresse B

```
MOVE  A, B  ≡  |  LOAD   A
               |  STORE  B
```

... weitere Macros

-- die Ausführung des Programms springt zur Adresse B,
-- wenn der Inhalt von Adresse A gleich Null ist.

```
BRZ  A,  B  ≡  |  LOAD  A
               |  BRZ   B
```

-- unbedingter Sprung

```
GOTO  X   ≡  |  CLR
             |  BRZ   X
```

… weitere Macros

Eine Reihe Adressen können mit konstanten Zahlen belegt werden oder als Hilfsadressen für Zwischenergebnisse benutzt werden.

T0, T1, …. , TN

Ein Befehl, der nichts tut, kann dann wie folgt definiert werden:

NOP ≡ | STORE T0

Wir werden Etiketten (*label:*) verwenden, um Programmadressen in den Programmen übersichtlicher zu machen.

. . . weitere Macros

-- berechnet das Komplement der Zahl in Adresse A

```
CMPL  A   ≡      CLR   T1
          loop:  INC   A
                 BRZ   A, end
                 INC   T1
                 GOTO  loop
          end:   MOVE  T1, A
```

. . . weitere Macros

-- berechnet **-n** aus einer beliebigen Zahl **n**

```
NEG   A    ≡  |  CMPL  A
              |  INC   A
```

-- dekrementiert die Zahl in Adresse A (n-1)

```
DEC   A    ≡  |  NEG   A
              |  INC   A
              |  NEG   A
```

Programmbeispiele

Addition:

ADD A, B ≡

```
loop:  BRZ  A,  end
       INC  B
       DEC  A
       GOTO loop
end:   NOP
```

Programmbeispiele

Um die Multiplikation und die Division zu implementieren, definieren wir zuerst die SHIFT-Operationen.

Eine Bit-Verschiebung der Zahl in Adresse A nach links ist:

SHIFTL A = A*2 = A+A

SHIFTL A ≡

```
MOVE  A  T1
ADD   T1 A
```

Programmbeispiele

Eine Bit-Verschiebung nach rechts kann wie folgt implementiert werden:

SHIFTR A = A/2

SHIFTR A ≡

```
        CLR   T3
loop:   BRZ   A, end
        DEC   A
        BRZ   A, end
        DEC   A
        INC   T3
        GOTO  loop
end:    MOV   T3, A
```

Selbstmodifizierende Programme

Um Speicherbereiche zu kopieren, brauchen wir entweder indirekte Adressierung oder ein Programm, das sich selber verändern kann.

Indirekte Adressierung:

der Code des LOAD-Befehls wird in T4 gespeichert.

```
LOAD  (A)  ≡   |      MOVE  "LOAD",  T4
               |      ADD   A,   T4
               |      MOVE  T4,  inst
               | inst:  0;     -- Null als Platzfüller
```

Selbstmodifizierende Programme

Indirekte Adressierung:

der Code des STORE-Befehls wird in T4 gespeichert.

```
STORE (A) ≡ |   MOVE "STORE", T4
            |   ADD  A,  T4
            |   MOVE  T4,  inst
            |   . . .
            | inst:  0;    -- Null als Platzfüller
```

Selbstmodifizierende Programme

Jetzt können wir einen Block aus n Zahlen ab Adresse A in einen zweiten Bereich ab Adresse B kopieren

COPY A, B, N ≡

```
loop:  BRZ    N,  end
       DEC    N
       LOAD   (A)
       STORE  (B)
       INC    A
       INC    B
       GOTO   loop
end:   NOP
```

Ist diese Register-Maschine universell?

Ja, weil die Turing-Maschine damit simuliert werden kann.

Diese Register-Maschine kann um einen Befehl reduziert werden, weil der CLR-Befehl wie folgt implementiert werden kann.

Wir können eine konstante Position Z im Speicher haben, die immer eine Null beinhaltet.

$$\text{CLR} \quad \equiv \quad \text{LOAD} \quad Z$$

Einfache Register-Maschine in Haskell

```
-- Register-Maschine

data Inst  =  CLR  | HALT  |  INC  |  BRZ  Int  |
                        LOAD  Int | STORE  Int | NOP
                        deriving (Show, Eq)


--       ( memory, acc, prog, pc )
type RM = ([Int], Int, [Inst], Int)


run :: RM -> RM
run (mem, acc, prog, (-1))   = (mem, acc, prog, (-1))
run (mem, acc, prog, pc) = run ( runInst (mem, acc, prog, pc))
```

Register-Maschine in Haskell

```
runInst (mem, acc, prog, pc) | prog!!pc == NOP    = ( mem,  acc, prog, pc+1 )
runInst (mem, acc, prog, pc) | prog!!pc == CLR    = ( mem,    0, prog, pc+1 )
runInst (mem, acc, prog, pc) | prog!!pc == HALT   = ( mem,  acc, prog, (-1) )
runInst (mem, acc, prog, pc) | prog!!pc == INC    = ( mem,acc+1, prog, pc+1 )

runInst (mem, acc, prog, pc) | is_Load(prog!!pc)  =
                        ( mem, mem!!(getAddr(prog!!pc)), prog, pc+1 )
                                where
                                  getAddr (LOAD a) = a

runInst (mem, acc, prog, pc) | is_Store(prog!!pc) =
                  (( wm mem (getAddr(prog!!pc)) acc), acc, prog, pc+1 )
                                where
                                  getAddr (STORE a) = a
```

Register-Maschine in Haskell

```
runInst (mem, acc, prog, pc) | is_Brz(prog!!pc) =
                    if acc==0 then (mem, acc, prog, getAddr(prog!!pc))
                              else (mem, acc, prog, pc+1)
                      where
                        getAddr (BRZ a) = a
```

Register-Maschine in Haskell

```
-- Hilfsfunktionen
wm  mem  a  acc =  take a mem ++ [acc] ++ drop (a+1) mem

is_Load :: Inst -> Bool
is_Load (LOAD a) = True
is_Load _        = False

is_Store :: Inst -> Bool
is_Store (STORE a) = True
is_Store _         = False

is_Brz :: Inst -> Bool
is_Brz (BRZ a)   = True
is_Brz  _        = False
```

Register-Maschine in Haskell

```
-- Macros
clr a     = [ CLR, STORE a ]
inc a     = [ LOAD a, INC, STORE a]
move a b  = [ LOAD a, STORE b ]
brz a b   = [ LOAD a, BRZ b ]
goto x    = [ CLR, BRZ x ]

-- Programmbeispiel
prog_1 = [ CLR,   INC,   INC,  STORE 3,   CLR,  HALT  ]
```