



Grundlagen der Digitaltechnik

Foliensatz 6: Rechnerstrukturen

G. Kemnitz

Institut für Informatik, TU Clausthal (EDS_F6)

13. Juli 2023



Inhalt F6: Rechnerstrukturen

CORDIC

- 1.1 Algorithmus
- 1.2 Simulation des Algorithmus
- 1.3 Umstellung auf Festkommazahlen
- 1.4 Entwurf als Rechenwerk
- 1.5 Testrahmen

MiPro

- 2.1 Funktion und Befehlssatz
- 2.2 Assembler und Disassembler
- 2.3 Prozessorzustand
- 2.4 Simulationsmodell
- 2.5 Testrahmen
- 2.6 Testbeispiele

RISC-Prozessor

- 3.1 Pipeline-Verarbeitung
- 3.2 Pipeline-Auslastung
- 3.3 Simulationsmodell
- 3.4 Testbeispiele

Der Foliensatz behandelt 3 Beispielenwürfe

Anwendungsspezifisches Rechenwerk für Winkelfunktionen:

- Algorithmus und Optimierung mit Gleitkommaoperationen.
- Package mit algorithmenspez. Datentypen, Operationen, ...
- Ersatz der Gleitkomma- durch Festkommaoperationen.
- Operationsablauf. Simulationsmodell für die Synthese, ...

Minimalprozessor »MiPro« aus »Rechnerarchitektur«:

- Package mit algorithmenspezifischen Datentypen, ...
- Assembler, Disassembler, ... in VHDL.
- Simulationsmodell, Testrahmen und Testbeispiele.

Erweiterung des Minimalprozessors zu einem RISC-Prozessor:

- Pipeline-Erweiterung.
- Simulationsmodell, Testbeispiele.

Lernziele und Anmerkungen

Der Hauptaufwand des modernen Schaltungsentwurfs entfällt auf Simulation, Test und Fehlerbeseitigung.

Die nachfolgenden Simulationen nutzen wieder die klassische »printf-Technik«¹. Dafür werden entsprechende Packages programmiert.

Im Gegensatz zu Software ist für Hardware Optimierung wichtig. Minimierung der Schaltungsgröße, des Stromverbrauchs, ...

Das größte Optimierungspotenzial steckt im Algorithmus.

Suche eines Algorithmus mit minimaler Anzahl von Rechenschritten, Rechenwerken, ...

¹Wie bereits auf Foliensatz 2 für Anweisungen und auf Foliensatz 4 für Operationsabläufe.



CORDIC



Algorithmen für Sinus und Kosinus

Taylor-Reihen:

$$\sin(\phi) = \sum_{n=0}^{\infty} (-1)^n \frac{\phi^{2n+1}}{(2n+1)!}$$

$$\cos(\phi) = \sum_{n=0}^{\infty} (-1)^n \frac{\phi^{2n}}{(2n)!}$$

- Mehrere Multiplikationen je Summationsschritt.
- Erforderliche Anzahl der Summationsschritte hängt von ϕ ab.

CORDIC:

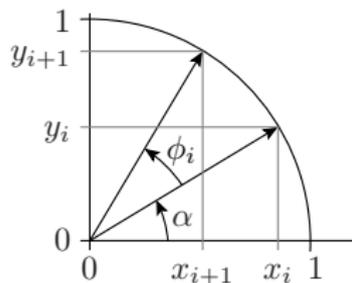
- Drei Additionen/Subtraktionen je Berechnungsschritt.
- Erforderliche Anzahl der Berechnungsschritte ist etwa Anzahl der Nachkommabits des Ergebnisses und damit konstant.



Algorithmus

Mathematische Grundlage

Rotation eines Vektors in einem Koordinatensystem:

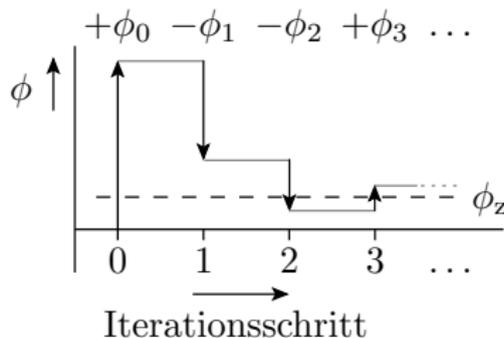
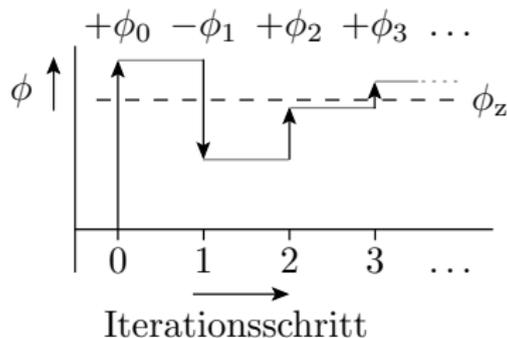


$$\begin{aligned}
 x_i &= \cos(\alpha) \\
 y_i &= \sin(\alpha) \\
 x_{i+1} &= \cos(\alpha + \phi_i) = \cos(\alpha) \cdot \cos(\phi_i) - \sin(\alpha) \cdot \sin(\phi_i) \\
 &= x_i \cdot \cos(\phi_i) - y_i \cdot \sin(\phi_i) \\
 y_{i+1} &= \sin(\alpha + \phi_i) = \cos(\alpha) \cdot \sin(\phi_i) + \sin(\alpha) \cdot \cos(\phi_i) \\
 &= x_i \cdot \sin(\phi_i) + y_i \cdot \cos(\phi_i)
 \end{aligned}$$

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos(\phi_i) & -\sin(\phi_i) \\ \sin(\phi_i) & \cos(\phi_i) \end{pmatrix} \cdot \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Beim CORDIC-Algorithmus wird der Drehwinkel aus einer festen Anzahl konstanter Winkelschritte abnehmender Größe zusammengesetzt und schrittweise der zugehörige Sinus- und Kosinus-Wert berechnet.

In jedem Schritt nächst kleinere Winkelkonstante addieren oder subtrahieren. Iterationsziel: Annäherung der Winkelsumme an den Vorgabewert ϕ_z :



Zu jeder Winkeladd./-sub. gehört eine Matrixmultiplikation

$$\begin{pmatrix} c \\ s \end{pmatrix} = \begin{pmatrix} \cos(\pm\phi_{n-1}) & -\sin(\pm\phi_{n-1}) \\ \sin(\pm\phi_{n-1}) & \cos(\pm\phi_{n-1}) \end{pmatrix} \cdot \dots \cdot \begin{pmatrix} \cos(\pm\phi_0) & -\sin(\pm\phi_0) \\ \sin(\pm\phi_0) & \cos(\pm\phi_0) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

(c, s) – Ergebnisse für $\cos(\phi)$ und $\sin(\phi)$.

Optimierung zur Hardware-Nachbildung

Aufwand je Rotationsschritt bis hierher: 4 Mult., mehrere Add., ...

Weitere Optimierung:

- Kosinus $\cos(\phi_i) = \cos(-\phi_i)$ ausklammern:

$$\begin{pmatrix} c \\ s \end{pmatrix} = SCS \cdot \begin{pmatrix} 1 & -\tan(\pm\phi_{n-1}) \\ \tan(\pm\phi_{n-1}) & 1 \end{pmatrix} \cdot \dots \cdot \begin{pmatrix} 1 & -\tan(\pm\phi_0) \\ \tan(\pm\phi_0) & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- Winkelvorzeichen unabhängige Konstante:

$$SCS = \prod_{i=0}^{n-1} \cos(\phi_i)$$

- Winkelschritte so wählen, dass die Tangens-Konstanten absteigende Zweierpotenzen sind:

$$\begin{pmatrix} c \\ s \end{pmatrix} = SCS \cdot \begin{pmatrix} 1 & \mp 2^{-n+1} \\ \pm 2^{-n+1} & 1 \end{pmatrix} \cdot \dots \cdot \begin{pmatrix} 1 & \mp 2^{-1} \\ \pm 2^{-1} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \mp 2^{-0} \\ \pm 2^{-0} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} c \\ s \end{pmatrix} = SCS \cdot \begin{pmatrix} 1 & \mp 2^{-n+1} \\ \pm 2^{-n+1} & 1 \end{pmatrix} \cdot \dots \cdot \begin{pmatrix} 1 & \mp 2^{-1} \\ \pm 2^{-1} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \mp 2^{-0} \\ \pm 2^{-0} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Berechnung von $\cos(\phi_i)$ und $\sin(\phi_i)$ mit n Nachkommabits:

- $2 \cdot n$ Bitverschiebungen um $i \in [1, n]$ Bits
- $3 \cdot n$ Additionen/Subtraktionen
- n Lesezugriffe auf eine Tabelle mit $(\arctan(2^{-i}))$
- 2 Multiplikationen mit der Konstanten SCS

statt $4 \cdot 2$ Multiplikationen und ...

Rechenwerke und Konstanten

Der Algorithmus benötigt für jeden Rotationsschritt

- zwei Additionen/Subtraktionen mit »alter Wert $\cdot 2^{-i}$ «,
- eine Addition/Subtraktion mit einer Winkelkonstanten:

i	0	1	2	3	4	5	6	$i > 6$
$\tan(\phi_i)$	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-i}
ϕ_i	0,785	0,464	0,245	0,124	0,062	0,031	0,015	$\approx 2^{-i}$

- Je eine Abschlussmultiplikation mit

$$\text{SCS} = \prod_{i=0}^{n-1} \cos\left(\arctan\left(2^{-i}\right)\right) \approx 0,607$$

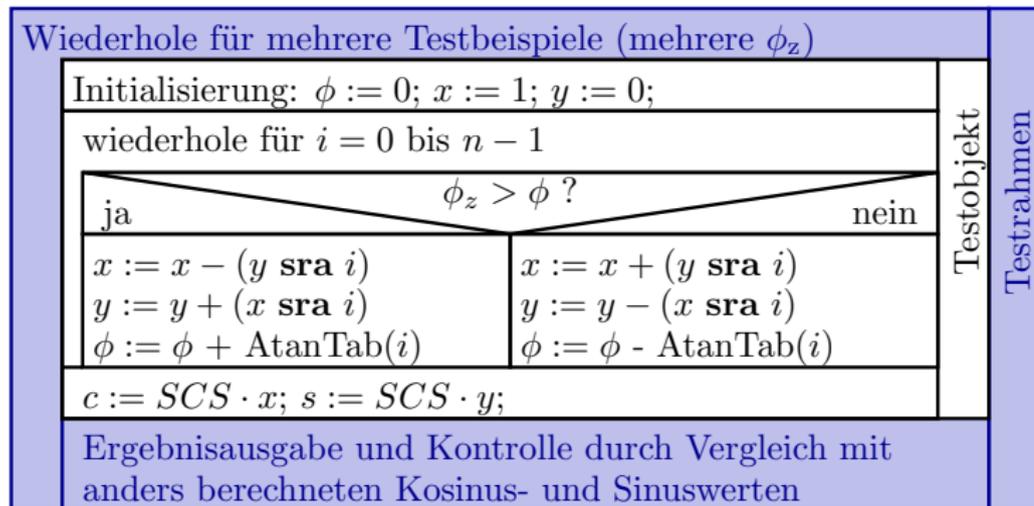
für den Sinus- und Kosinuswert.



Simulation des Algorithmus



Algorithmus und Testrahmen



$x \text{ sra } i$ arithmetische Rechtsverschiebung, identisch mit $x \cdot 2^{-i}$

Das Simulationsmodell dient auch zur Festlegung der erforderlichen Anzahl der Nachkommabits und Berechnungsschritte für die erforderliche Rechengenauigkeit.



Package mit anwendungsspezifischen Datentypen

- ```
package cordic_real_pack is
```
- Datentyp für Winkel-, Sinus-, Kosinus und Zwischenwerte<sup>2</sup>:  

```
subtype t_dat is real range -2.0 to 2.0;
```
  - Feldtyp für Winkelwerte für die Konstantentabelle:  

```
type t_dat_array is array (natural range <>) of t_dat;
```
  - Datentyp für Testbeispiele (Eingabe + Ergebnissollwerte):  

```
type t_test is record
 phi:t_dat; sin:t_dat; cos:t_dat;
end record;
```

---

<sup>2</sup>Später im Package cordic\_fix\_pack.vhd wird t\_dat durch einen Bitvektortyp für Festkommazahlen ersetzt.



## Konstanten und Testbeispiele

- Winkelkonstanten<sup>3</sup> (Berechnung siehe Folie 12):

```
constant AtanTab: t_dat_array(0 to 15) := (
 0.785398163, 0.463647609, 0.244978663,
 0.124354995, 0.062418810, 0.031239833, ...);
```

```
constant SCS: t_dat := 0.607252935;
```

- Testbeispiele als Feldkonstante von Tupeln aus Winkel- und Sollwerten für den zugehörigen Sinus- und Kosinuswert:

```
type t_test_dat is array (natural range <>
 of t_test;
```

```
constant test_dat: t_test_dat(1 to 21) := (
 1=>(-1.57080, -1.00000, 0.00000),
 2=>(-1.41372, -0.98769, 0.15643), ...);
```

---

<sup>3</sup>Erzeugt mit dem Matlab-Script GenConstReal.m auf der WEB-Seite.



## Anwendungsspezifische Funktionen

- Konvertierung von Werten vom Typ »t\_dat« in eine formatierte Textdarstellung:

```
function str(x t_dat) return string;
```

- Vorzeichenbehaftete Rechtsverschiebung um eine variable Bitanzahl (Multiplikation mit  $2^{-i}$ ):

```
function "sra"(a: t_dat; i: natural)
 return t_dat;
```

- Ende der Package-Definition:

```
end package;
```



## Funktionsimplementierungen im Package Body

```
package body cordic_real_pack is
 function str(x t_dat) return string is
 — Textkonvertierung für Werte vom Typ t_dat in
 — eine Dezimaldarstellung mit Vorzeichen, einer
 — Vor- und 4 Nachkommastellen, z.B. "-1,9765"
 ...

 function4 "sra"(a: t_dat; i: natural) return t_dat is
 — arithmetische Rechtsverschiebung
 begin
 return a/(2.0**i);
 end function;
end package body;
```

---

<sup>4</sup>Aufruf der Operatorfunktion mit `<Wert> sra <Verschiebung>`.



### Testprogramm

```
-- test_cordic_real.vhd
use std.textio.write;
use std.textio.output;
use work.cordic_real_pack.all;

entity test_cordic_real is end entity;
architecture a of test_cordic_real is
 signal x, y, c, s, phi, phi_z: t_dat;
begin
 process
 variable test: t_test;
 begin
 <Ausgabe Tabellekopf>
 for tnr in test_dat'range loop
 <Abarbeitung und Protokollierung der Testbeispiele>
 end loop;
 wait;
 end process;
end architecture;
```

Wiederhole für 21 Testbeispiele

zu testender Algorithmus

Ausgabe der Ergebnisse und  
der Soll-/Ist-Abweichungen



## Protokollierung der Testbeispiele

```

write(output, "␣Winkel␣(Delta)|Kosinus␣(Delta)|␣...")
for tnr in test_dat'range loop
 test := test_dat(tnr);
 phi_z <= test.phi;
 <Berechnung von sin(phi_z) und cos(phi_z)>
 write(output, str(phi_z) & "␣" & str(phi-test.phi)
 & "|" & str(c) & "␣" & str(c-test.cos)
 & "|" & str(s) & "␣" & str(s-test.sin));
end loop;

```

Textausgabe (Delta ist die Soll/Ist-Abweichung, im Beispiel null):

```

Winkel <Delta>|Kosinus <Delta>| Sinus <Delta>
-1,5708 -0,0000|-0,0000 -0,0000|-1,0000 +0,0000
-1,4137 +0,0000|+0,1565 +0,0000|-0,9877 +0,0000
-1,2566 +0,0000|+0,3090 +0,0000|-0,9511 +0,0000
-1,0996 +0,0000|+0,4540 +0,0000|-0,8910 +0,0000
-0,9425 +0,0000|+0,5878 +0,0000|-0,8090 +0,0000

```

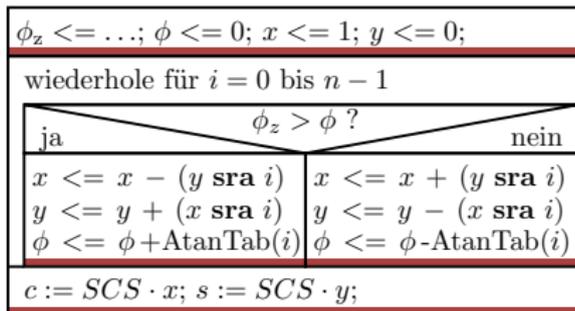


## Berechnung

```

x<=1.0; y<=0.0; phi<=0.0;
wait for 10 ns;
for i in AtanTab' range loop
 if phi_z > phi then
 x <= x - (y sra i); y <= y + (x sra i);
 phi<=phi + AtanTab(i);
 else
 x <= x + (y sra i); y <= y - (x sra i);
 phi <= phi - AtanTab(i);
 end if;
wait for 10 ns;
end loop;
c <= SCS * x; s <= SCS * y;
wait for 10 ns;

```



— Simulation wartet 10 ns

Wait-Anweisungen, damit die Signale die zugewiesenen Werte übernehmen.



## Umstellung auf Festkommazahlen



## Umstellung auf Festkommazahlen

Der nicht synthesefähige Typ

```
subtype t_dat is real range -2.0 to 2.0;
```

soll durch 16-Bit vorzeichenbehaftete Festkommazahlen ersetzt werden.

- Betrag kleiner 2 verlangt ein Vorzeichen- und ein Datenbit (insg. 2 Bit) vor dem Komma.
- Bleiben  $16-2=14$  Nachkommabits. Rundungsfehler  $> 2^{-15} \approx 3 \cdot 10^{-5}$ .

Die Zahlentypänderung verlangt fast nur Änderungen im Package:

```
package cordic_fix_pack is
 subtype t_dat is signed(15 downto 0);
```

Die Definitionen der daraus zusammengesetzten Typen »t\_dat\_array«, »t\_test« und »t\_test\_dat« ändern sich nicht.



## Die Konstanten sind umzurechnen (mit Matlab)

$$w_{\text{fix}} = \text{round}(w_{\text{real}} \cdot 2^{14})$$

```
type t_dat_array is array (natural range <>) of t_dat;
— Winkelkonstanten
constant AtanTab: t_dat_array(0 to 14) := (
 x"3244", — 0.78540
 x"1dac", — 0.46365
 x"0fae", — 0.24498
 ...
 x"0001");— 0.00006 (Wert 15 wäre null und entfällt)

— Abschlussmultiplikator
constant SCS: t_dat := x"26dd"; — 0.6073

— Testbeispiele
constant test_dat: t_test_dat(1 to 21) := (
 1=>(x"9b78", x"c000", x"0000"), —(-1.570, -1.000, 0.000)
 2=>(x"a585", x"c0c9", x"0a03"), —(-1.413, -0.988, 0.156)
 ...
```



## Umdefinieren der Package-Funktionen

Die Textkonvertierung

```
function str(x: t_dat) return string is ..
```

ist neu zu programmieren. Die Zahlendarstellung soll bleiben wie bisher:

```
Winkel <Delta>|Kosinus <Delta>| Sinus <Delta>
-1,5708 -0,0000|-0,0000 -0,0000|-1,0000 +0,0000
-1,4137 +0,0000|+0,1565 +0,0000|-0,9877 +0,0000
-1,2566 +0,0000|+0,3090 +0,0000|-0,9511 +0,0000
-1,0996 +0,0000|+0,4540 +0,0000|-0,8910 +0,0000
-0,9425 +0,0000|+0,5878 +0,0000|-0,8090 +0,0000
```

Der sra-Operator ist mit Bitverschiebungen zu realisieren:

```
function "sra"(a:t_dat; b:natural) return t_dat is
 variable y: t_dat := (others=>a(15));
begin
 assert b<16; — nur fuer die Simulation
 y(15-b downto 0) := a(15 downto b);
 return y;
end function;
```



## Neue Konstanten und Package-Funktionen

Definition von Konstanten für die Werte 0.0 und 1.0:

```
constant const_0: t_dat := x"0000"; -- 0.0
constant const_1: t_dat := x"4000"; -- 1.0
```

Funktion für die Abschlussmultiplikation mit SCS<sup>5</sup>:

```
function mult_scs(a: t_dat) return t_dat is
 variable p: signed(31 downto 0) := a * SCS;
begin
 return t_dat(p(29 downto 14));
end function;
```

---

<sup>5</sup>Das Produkt zweier Zahlen mit 2 Bit vor und 14 Bit nach dem Komma hat 4 Bit vor und 28 Bit nach dem Komma. Daraus wird der Teilbereich mit 2 Vor- und 14 Nachkommabits ausgewählt. Wegen der betragsmäßigen Wertebegrenzung auf  $< 1$  sind bei der Verringerung der Anzahl der Vorkommastellen auf 2 Wertebereichsüberläufe ausgeschlossen.

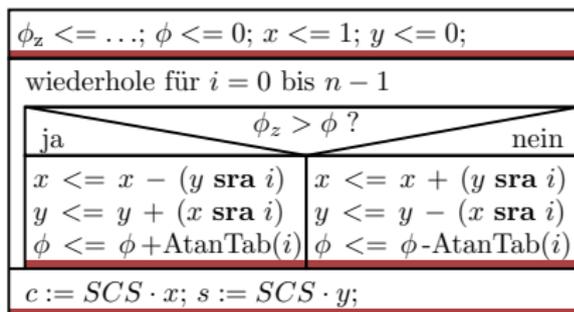


## Der simulierte Algorithmus bleibt fast unverändert

```

x<=const_1; y<=const_0;
phi<=const_0;
wait for 10 ns;
for i in AtanTab range loop
 if phi_z > phi then
 x <= x - (y sra i); y <= y + (x sra i);
 phi<=phi + AtanTab(i);
 else
 x <= x + (y sra i); y <= y - (x sra i);
 phi <= phi - AtanTab(i);
 end if;
 wait for 10 ns;
end loop;
c <= mult_scs(x); s <= mult_scs(y);

```



Simulation wartet 10 ns



## Rahmenprogramm Simulation und Ausgabe

- Zusätzlich genutzte Packages: `work.cordic_fix_pack`, `ieee.numeric_std` (Operatoren `+`, `-`, `>`).
- Rest des Rahmenprogramms wie mit »reellen Zahlen«:

Wiederhole für alle Testbeispiele:

`x`, `y`, `phi` und `phi_z` initialisieren

15 CORDIC-Schritte,

2 Abschlussmultiplikationen

Textausgabe

- Numerischer Fehler max.  $\mp 2 \cdot 10^{-4}$ :

```
-0,1571 +0,0000!+0,9876 -0,0001!-0,1565 +0,0000
+0,0000 +0,0000!+1,0001 +0,0001!-0,0001 -0,0001
+0,1570 +0,0000!+0,9877 +0,0000!+0,1564 -0,0001
+0,3141 +0,0000!+0,9510 -0,0001!+0,3089 -0,0001
+0,4712 +0,0000!+0,8911 +0,0001!+0,4537 -0,0002
+0,6283 +0,0000!+0,8091 +0,0001!+0,5876 -0,0002
+0,7853 +0,0000!+0,7072 +0,0001!+0,7070 -0,0001
+0,9424 +0,0000!+0,5876 -0,0001!+0,8091 +0,0001
+1,0995 +0,0000!+0,4539 -0,0001!+0,8910 +0,0000
+1,2566 +0,0000!+0,3090 +0,0000!+0,9510 -0,0001
+1,4137 +0,0000!+0,1564 +0,0000!+0,9877 +0,0000
+1,5707 +0,0000!+0,0000 +0,0000!+1,0000 +0,0000
```



# Entwurf als Rechenwerk

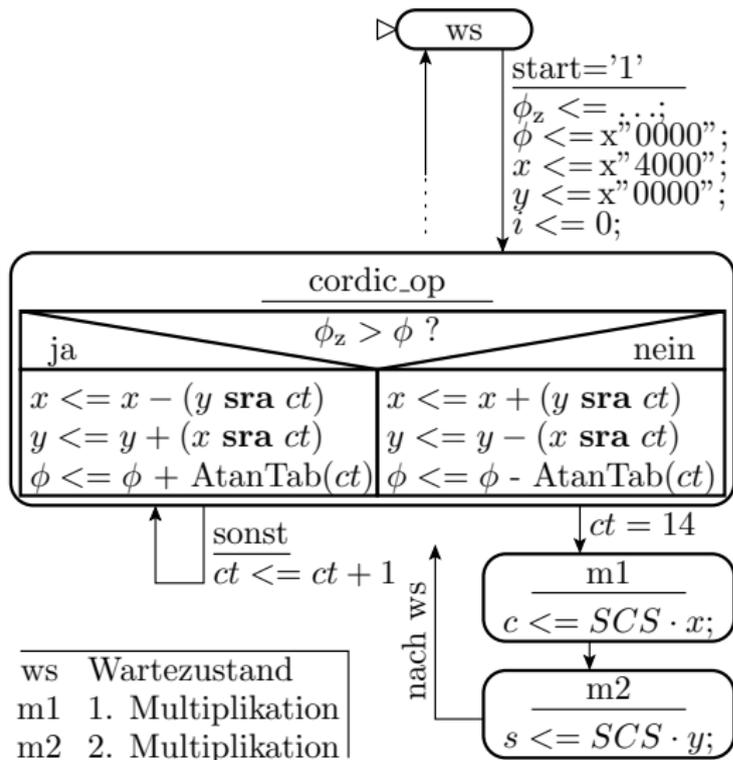
## Festlegung des Operationsablaufs

Angestrebtes Simulationsergebnis:



- Wenn busy inaktiv und Start aktiv, Datenübernahme und Berechnungsbeginn.
- 15 CORDIC-Schritte,
- Abschlussmultiplikationen nacheinander. Ein zusätzlicher Takt und ein eingesparter Multiplizierer.

## Operationsablaufgraph

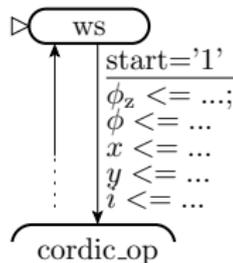


## Schnittstelle, interne Signale

```
use work.cordic_fix_pack.all; ...

entity cordic is
 port(
 T, I, Start: in std_logic; -- I, Start seien
 w: in t_dat; -- abgetastet
 busy: out std_logic;
 s, c: out t_dat);
end entity;

architecture a of cordic is
 type t_state is (ws, cordic_op, m1, m2);
 signal state: t_state;
 signal ct: natural range (AtanTab'range);
 signal x, y, phi, phi_z: t_dat;
begin
 ...
end architecture;
```



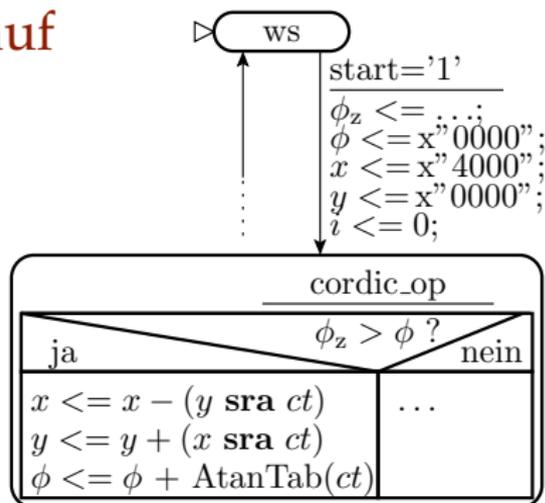


## Prozesse für Operationsablauf

```

process (T, I)
begin
 if I='1' then
 state <= ws;
 elsif rising_edge(T) then
 case state is
 when ws =>
 if Start = '1' then
 ct <= AtanTab'low;
 phi<=const_0;
 x<=const_1; y<=const_0;
 state <= cordic_op; phi_z <= w;
 end if;
 when cordic_op =>
 if phi_z > phi then
 x <= x - (y sra ct); y <= y + (x sra ct);
 phi<=phi + AtanTab(ct);

```





```

else
 x <= x + (y sra ct); y <= y - (x sra ct);
 phi <= phi - AtanTab(ct);
end if;
if ct=AtanTab'high-1 then state <= m1;
else ct <= ct +1; end if;
when m1 => -- Abschlussmultiplikation Kosinus
 c <= mult_scs(x); state <= m2;
when m2 => -- Abschlussmultiplikation Sinus
 s <= mult_scs(y); state <= ws;
end case;
end if;
end process;

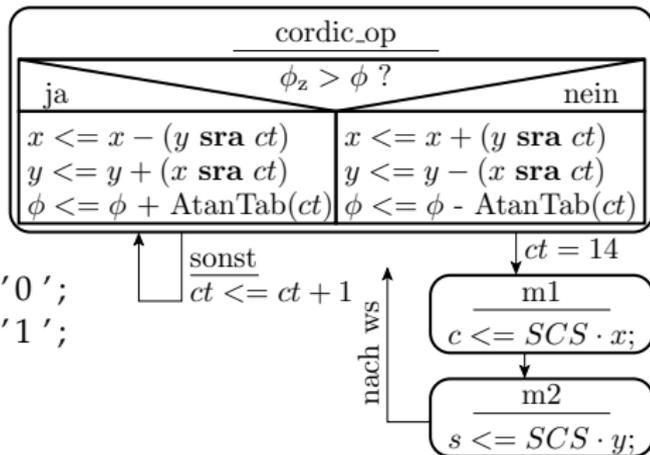
```

-- Erzeugung von busy

```

process (state) begin
 if state=ws then busy <= '0';
 else busy <= '1';
 end if;
end process;

```





# Testrahmen



```
use work.cordic_fix_pack.all;
library ieee; use ieee.std_logic_1164.all;
entity test_cordic is end entity;
architecture a of test_cordic is
 signal T, I, Start, busy: std_logic := '0';
 signal w, s, c: t_dat;
begin
 — Instanziierung des Testobjekts
 uut: entity work.cordic port map(T=>T, I=>I,
 Start=>Start, busy=>busy, w=>w, s=>s, c=>c);
 process — Erzeugung des Takt- und des Init.-Signals
 begin
 I <= '1', '0' after 20 ns;
 while now < 1 us loop
 wait for 5 ns; T <= not T;
 end loop;
 wait;
 end process;
```

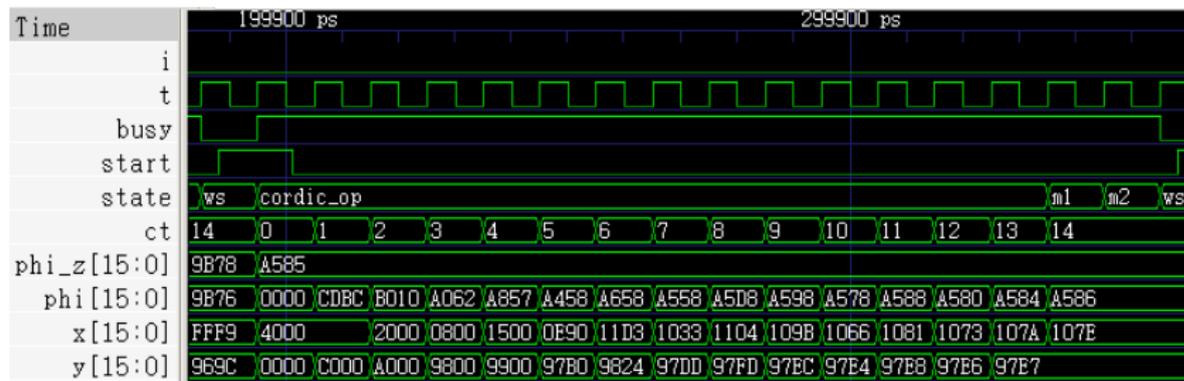


## Abarbeitung der Testbeispiele aus dem Package

```
process
 variable test: t_test;
begin
 for tnr in test_dat'range loop
 wait until busy='0'; wait for 3 ns;
 start <='1'; test:=test_dat(tnr);
 w <= test.phi;
 wait until busy = '1';
 wait for 6 ns;
 start <='0';
 end loop;
 wait;
end process;
end architecture;
```



## Simulationsergebnis



- Wenn »start=1« Eingabeübernahme und »busy<='1'.
- In den nächsten 15 Schritten folgen CORDIC-Operationen.
- Abschlussmultiplikationen nacheinander (mit demselben Multiplizierer) und Deaktivierung von »busy«.
- Kontrolle der Zahlenwerte besser nach der »printf«-Methode:

```
-0.1571 +0.0000!+0.9876 -0.0001!-0.1565 +0.0000
+0.0000 +0.0000!+1.0001 +0.0001!-0.0001 -0.0001
```



## Zusammenfassung

Entwurf eines CORDIC-Rechenwerks in den Schritten:

- Entwicklung und Simulation des Algorithmus mit Gleitkommazahlen (auch komplett in Matlab möglich).
- Ersatz der Gleitkomma- durch Festkommazahlen. Festlegung der Vor- und Nachkommabit- und Iterationsanzahl.
- Operationsablauf mit 3 Addierern, 2 Block-Shiftern, einem adressierbaren Konstantenspeicher, einem Multiplizierer, ...
- »Printf«- und Regressionstest nach jeder Verfeinerung zum Aufspüren und zur Beseitigung entstandener Fehler.

Es würden noch folgen: Synthese, Test, ...

### Tatsache:

Komplexe Systeme wie ein CORDIC-Rechenwerk lassen sich nicht mehr in einem Schritt entwerfen.



# MiPro

# Der Minimalprozessor MiPro

MiPro ist ein für die Lehrveranstaltung »Rechnerarchitektur« im 1. Semester entwickelter Minimalprozessor:

- 16-Bit Befehlswoorte, 4 Befehlsformate, 31 Befehle,
- acht 8-Bit-Register, 3-Bit-Registeradressen, 8-Bit-Konstanten,
- $2^8 = 256$  Befehls- und 256 Datenspeicherplätze.

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 | cnr       |
|---------------|-----------|--------|-------|-------|-----|-----------|
| noop          | 00000     |        |       |       |     | 0         |
| jump imm,cond | 00001     | cond   | imm   |       |     | 1         |
| cmd rd,imm    | cnr       | rd     | imm   |       |     | 2 bis 14  |
| cmd rd (retu) | cnr       | rd     |       |       |     | 15        |
| cmd rd,ra     | cnr       | rd     | ra    |       |     | 16 bis 23 |
| cmd rd,ra,rb  | cnr       | rd     | ra    | rb    |     | 24 bis 30 |

(cnr – Befehlsnummer; rd, ra, rb – Registeradressen; imm (**I**mmediate) – Direktwert ; cond (**C**ondition) – Sprungbedingung).



## 2. MiPro

Der Prozessor ist so einfach aufgebaut, dass sich der komplette Prozessorzustand in einer Textzeile darstellen lässt:

| PC | Befehl | assem.:     | hex  | r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | c | z |
|----|--------|-------------|------|----|----|----|----|----|----|----|----|---|---|
| 00 | ld_i   | r1, 4a, ..: | 294a | 00 | 4a | 00 | 00 | 00 | 00 | 00 | 00 | 0 | 0 |
| 01 | ld_i   | r0, 73, ..: | 2873 | 73 | 4a | 00 | 00 | 00 | 00 | 00 | 00 | 0 | 0 |
| 02 | move   | r2, r0, ..: | 8200 | 73 | 4a | 73 | 00 | 00 | 00 | 00 | 00 | 0 | 0 |
| 03 | move   | r3, r1, ..: | 8320 | 73 | 4a | 73 | 4a | 00 | 00 | 00 | 00 | 0 | 0 |
| 04 | andi   | r2, 0f, ..: | 620f | 73 | 4a | 03 | 4a | 00 | 00 | 00 | 00 | 0 | 0 |
| 05 | andi   | r3, f0, ..: | 63f0 | 73 | 4a | 03 | 40 | 00 | 00 | 00 | 00 | 0 | 0 |
| 06 | or_r   | r4, r2, r3: | ec4c | 73 | 4a | 03 | 40 | 43 | 00 | 00 | 00 | 0 | 0 |
| 07 | xorr   | r5, r4, r4: | f590 | 73 | 4a | 03 | 40 | 43 | 00 | 00 | 00 | 0 | 1 |

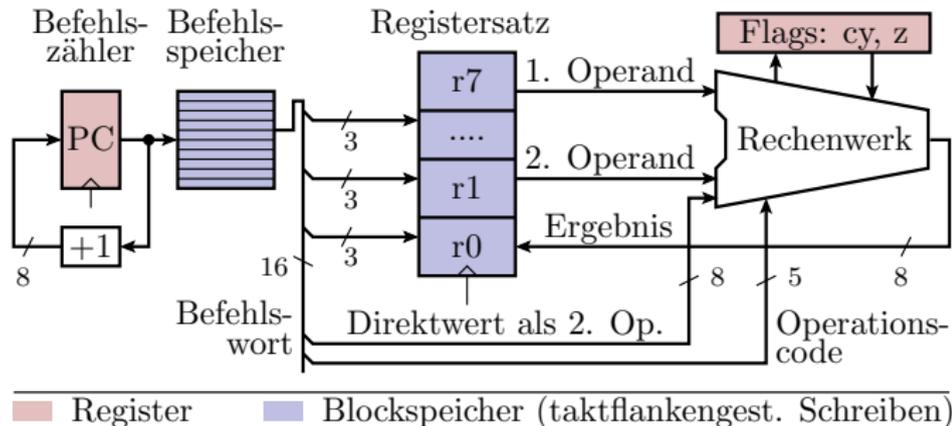
PC – Befehlszähler, r0 bis r7 – Register; c, z – Carry- und Zero-Flag.

- Spalte 1: Befehlsadresse (hex.)
- Spalte 2: Befehlswort (disassembliert und hex.)
- ab Spalte 3: Register und Flags nach Operationsausführung.

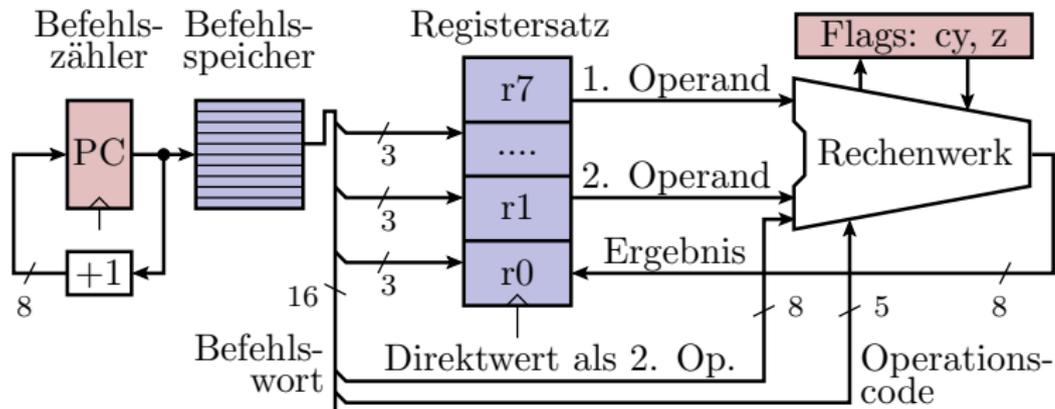


# Funktion und Befehlssatz

## Hardware für Verarbeitungsbefehle



- PC adressiert den Befehlspeicher und erhöht sich um eins.
- Aus dem gelesenen Befehlswort werden Befehlsnummer, Registeradressen und optional ein Direktwert entnommen.
- Operationausführung mit zwei Operanden und einem Ergebnis. Optionale Auswertung und Veränderung der Flags.



Mit dieser Hardware ausführbare Operationen:

- Kopieren: Register oder Konstante  $\Rightarrow$  Register;
- logische Operationen: UND-, ODER-, EXOR-Verknüpfung von zwei Operanden (bitweise). Negation eines Operanden.
- Verschiebung, Rotation rechts / links um ein Bit.
- Addition, Subtraktion mit oder ohne einlaufendem Übertrag.

Details siehe nachfolgende Folien. Mit dieser Hardware noch nicht ausführbar sind Lade-/Speicher-Operationen, Sprünge, ...



# Logikbefehle des Minimalprozessors

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 |
|---------------|-----------|--------|-------|-------|-----|
| cmd rd,ra,rb  | cnr       | rd     | ra    | rb    |     |
| cmd rd,ra     | cnr       | rd     | ra    |       |     |
| cmd rd,imm    | cnr       | rd     | imm   |       |     |

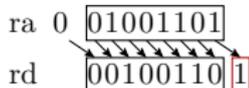
| Befehl        | Operation        | Flags | cnr |
|---------------|------------------|-------|-----|
| andr rd,ra,rb | rd := ra and rb  | z     | 28  |
| andi rd,imm   | rd := rd and imm | z     | 12  |
| or_r rd,ra,rb | rd := ra or rb   | z     | 29  |
| or_i rd,imm   | rd := rd or imm  | z     | 13  |
| xorr rd,ra,rb | rd := ra xor rb  | z     | 30  |
| xori rd,imm   | rd := rd xor imm | z     | 14  |
| notr rd,ra    | rd := not ra     | z     | 23  |

Befehle zum Initialisieren und Kopieren von Registerinhalten:

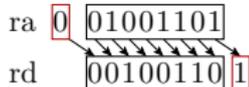
|             |           |  |    |
|-------------|-----------|--|----|
| ld_i rd,imm | rd := imm |  | 5  |
| move rd,ra  | rd := ra  |  | 16 |

## Verschiebe- und Rotationsbefehle

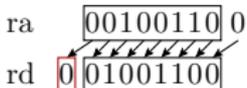
sh\_r rd, ra



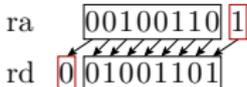
rotr rd, ra



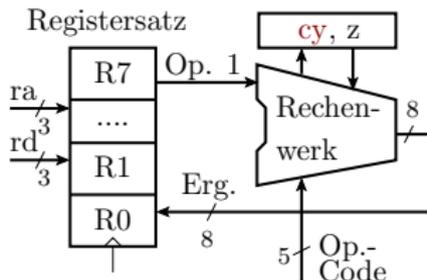
sh\_l rd, ra



rotl rd, ra



Registersatz



|             |                  |            |
|-------------|------------------|------------|
| sh_r rd, ra | rd := 0:ra >> 1  | cy, z   20 |
| rotr rd, ra | rd := cy:ra >> 1 | cy, z   22 |
| sh_l rd, ra | rd := ra:0 << 1  | cy, z   19 |
| rotl rd, ra | rd := ra:cy << 1 | cy, z   21 |

- 0:ra – Verkettung von null und ra zu einem 9-Bit-Wert.
- 1-Bit-Verschiebung, rausgeschobenes Bit in cy.
- Verschiebebefehle übernehmen in das frei werdende Bit null, Rotationsbefehle den bisherigen Wert von cy.

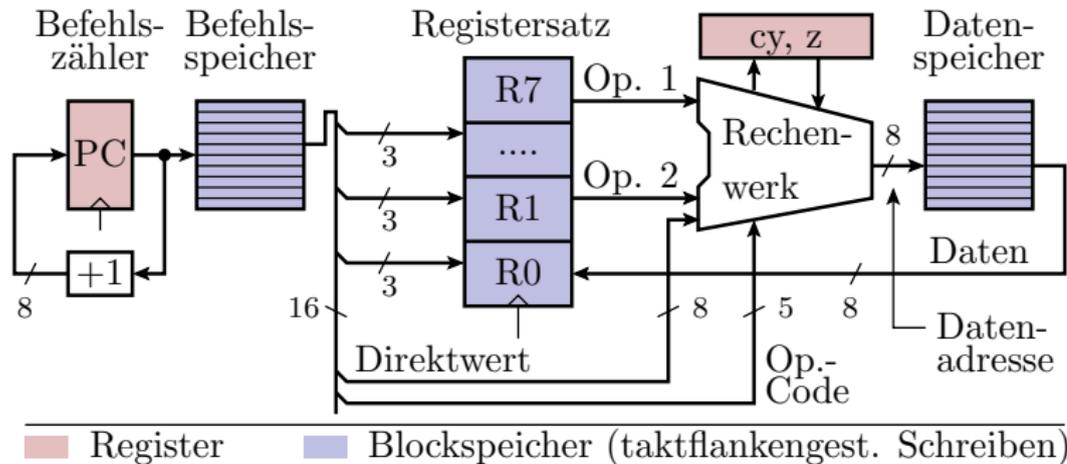


## Additions- und Subtraktionsbefehle MiPro

| Befehl           | Operation           | Flags | cnr |
|------------------|---------------------|-------|-----|
| addr rd, ra, rb  | rd := ra + rb       | cy, z | 24  |
| addi rd, imm     | rd := rd + imm      | cy, z | 8   |
| adcr rd, ra, rb  | rd := ra + rb + cy  | cy, z | 25  |
| adci rd, imm     | rd := rd + imm + cy | cy, z | 9   |
| subr rd, ra, rb  | rd := ra - rb       | cy, z | 26  |
| subi rd, imm     | rd := imm - rd      | cy, z | 10  |
| sbcrr rd, ra, rb | rd := ra - rb - cy  | cy, z | 27  |
| sbcir rd, imm    | rd := imm - rd - cy | cy, z | 11  |

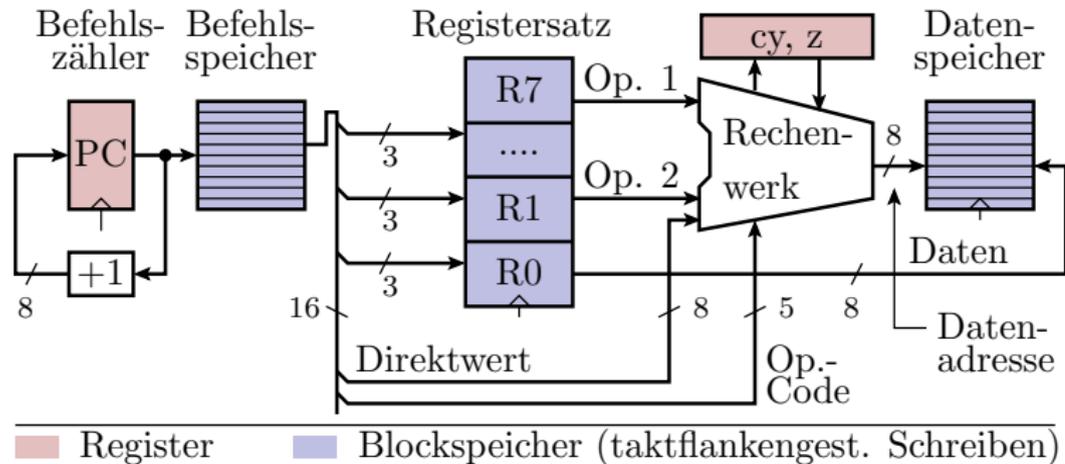
- Bei Addition mehrerer Bytes werden die niederwertigen Bytes mit add und die höherwertigen mit adc addiert. Analog bei der Subtraktion.
- Der erste Operand kann eine Variable oder eine Konstante (Direktwert) sein.

## Erweiterung um Ladeoperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das Zielregister übernimmt statt des Berechnungsergebnisses den aus dem Datenspeicher gelesenen Wert.

## Erweiterung um Speicheroperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das »Zielregister« wird gelesen und sein Wert unter der berechneten Adresse im Datenspeicher abgelegt.

## Lade- / Speicherbefehle des Minimalprozessors

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 |
|---------------|-----------|--------|-------|-------|-----|
| cmd rd,ra     | cnr       | rd     | ra    |       |     |
| cmd rd,imm    | cnr       | rd     | imm   |       |     |

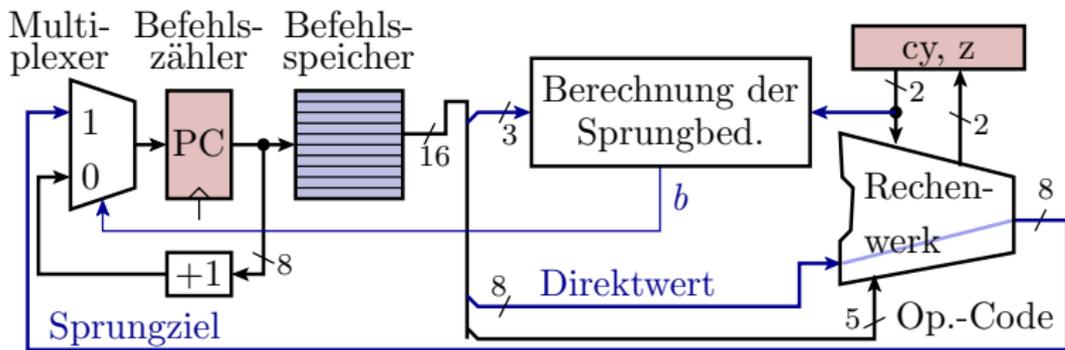
| Befehl      | Operation       | Flags | cnr |
|-------------|-----------------|-------|-----|
| load rd,imm | rd := dmem(imm) |       | 3   |
| stor rd,imm | dmem(imm) := rd |       | 4   |
| ld_r rd,ra  | rd := dmem(ra)  |       | 17  |
| st_r rd,ra  | dmem(ra) := rd  |       | 18  |

Unterstützte Adressierungsarten:

- direkt für die Adressierung von Variablen mit festen Adressen.
- indirekt für die Adressierung mit Zeigern.

Eine Adressrechnung, z.B. Registerinhalt + Konstante, in MiPro nicht implementiert, würde im Rechenwerk erfolgen.

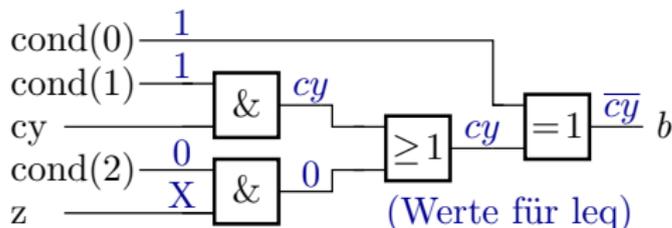
## MiPro-Erweiterungen für Sprungbefehle



jump imm,cond; if (b) pc = imm; else pc++;

- Das Rechenwerk leitet die Konstante zu einem Multiplexer (Umschalter), der gesteuert vom berechneten Bedingungsbit  $b$  zwischen »nächster Befehl« und »Sprung« umschaltet.
- Die Berechnung der Sprungbedingung erfolgt mit einer Schaltung aus 4 Gattern (siehe nächste Folie).

## Berechnung der Sprungbedingung



| cond | Wert | Bedeutung                       | Flag-Bedingung          |
|------|------|---------------------------------|-------------------------|
| nev  | 000  | jump <b>never</b>               | keine                   |
| alw  | 001  | jump <b>always</b>              | keine                   |
| gth  | 010  | jump <b>greater than</b>        | $\text{cy}=1$           |
| leq  | 011  | jump if <b>less or equal</b>    | $\text{cy}=0$           |
| equ  | 100  | jump if <b>equal</b>            | $z = 1$                 |
| neq  | 101  | jump if <b>not equal</b>        | $z = 0$                 |
| geq  | 110  | jump if <b>greater or equal</b> | $\text{cy}=1$ or $z=1$  |
| lth  | 111  | jump if <b>less than</b>        | $\text{cy}=0$ and $z=0$ |





# Assembler und Disassembler

## Textdarstellung von Befehlsbestandteilen

```
constant imem: t_imem(0 to 7) := (
 0=>cmd(ld_i, r1, x"23"),
 1=>cmd(subi, r1, x"4a"), ...
```

```
00|ld_i r1,23,..:2923|00 23 00 00 00 00 00 00|0|0|
01|subi r1,4a,..:514a|00 27 00 00 00 00 00 00|0|0|
```

Für Programmierung und Test ist es wichtig, dass Befehle und Befehlsbestandteile in Assemblernotation beschrieben werden:

- Befehlsnummern als Symbole,
- Registeradressen als r0 bis r7 und
- Konstanten als zweistellige Hexzahlen.

Programmtechnische Lösung:

- Symbolische Befehlsbestandteile als Aufzählungstypen,
- Assemblieren mit den Funktionen »cmd()«, »nop()« und »jmp()«,
- ...



## Befehlsbestandteile als Aufzählungstypen

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 | cnr       |
|---------------|-----------|--------|-------|-------|-----|-----------|
| noop          | 00000     |        |       |       |     | 0         |
| jump imm,cond | 00001     | cond   | imm   |       |     | 1         |
| cmd rd,imm    | cnr       | rd     | imm   |       |     | 2 bis 14  |
| cmd rd (retu) | cnr       | rd     |       |       |     | 15        |
| cmd rd,ra     | cnr       | rd     | ra    |       |     | 16 bis 23 |
| cmd rd,ra,rb  | cnr       | rd     | ra    | rb    |     | 24 bis 30 |

```

type t_cmd is (noop, — Assemblierfunktion
 jump, — jump(imm8, cond:=non)
 call, load, stor, ld_i, comp, — cmd(<cmd> rd, imm8)
 cmpc, addi, adci, subi, sbci,
 andi, or_i, xori,
 retu, — ret(rd)
 move, ld_r, st_r, sh_l, sh_r, — cmd(<cmd> rd ra)
 rotl, rotr, notr,
 addr, adcr, subr, sbcr, — cmd(<cmd> rd ra rb)
 andr, or_r, xorrr);

```



## Konvertierung in Texte und Bitvektoren

- Aufzählungstyp (hier vom Typ »t\_cmd«) nach Text:

```
... t_cmd 'image' (<Objekt vom Typ t_cmd>)
```

- Zahlenwert »w« nach *b*-Bit-Vektor\* :

```
function to_bitvec(w: natural; b: positive)
return bit_vector;
```

- Aufzählungstyp (hier vom Typ »t\_cmd«) nach Bitvektor:

```
... to_bitvec(t_cmd 'pos' (<Obj. vom Typ t_cmd>, 5);
```

- Bitvektor in eine vorzeichenfreie Zahl\* :

```
function uint(w: bit_vector) return natural;
```

- Zahlenwert nach Text (Hex.-Darstellung)\* :

```
function hex(dat: bit_vector) return string;
```

\* Funktionen aus dem Package »mipro\_pack«.



## Registeradressen und Sprungbedingungen

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 | cnr       |
|---------------|-----------|--------|-------|-------|-----|-----------|
| noop          | 00000     |        |       |       |     | 0         |
| jump imm,cond | 00001     | cond   | imm   |       |     | 1         |
| cmd rd,imm    | cnr       | rd     | imm   |       |     | 2 bis 14  |
| cmd rd (retu) | cnr       | rd     |       |       |     | 15        |
| cmd rd,ra     | cnr       | rd     | ra    |       |     | 16 bis 23 |
| cmd rd,ra,rb  | cnr       | rd     | ra    | rb    |     | 24 bis 30 |

```
type t_reg is (r0, r1, r2, r3, r4, r5, r6, r7);
```

```
type t_cond is (
 nev, — jump never
 alw, — jump always
 gth, — jump greater than (cy=1)
 leq, — jump if less or equal (cy=0)
 equ, — jump if equal (z =1)
 neq, — jump if not equal (z =0)
 geq, — jump if greater or equal (cy=1 or z=1)
 lth); — jump if less than (cy=0 and z=0)
```



## Die Assembler-funktionen

| Teilbitvektor | 15 ... 11 | 10 9 8 | 7 6 5 | 4 3 2 | 1 0 | cnr       |
|---------------|-----------|--------|-------|-------|-----|-----------|
| noop          | 00000     |        |       |       |     | 0         |
| jump imm,cond | 00001     | cond   | imm   |       |     | 1         |
| cmd rd,imm    | cnr       | rd     | imm   |       |     | 2 bis 14  |
| cmd rd (retu) | cnr       | rd     |       |       |     | 15        |
| cmd rd,ra     | cnr       | rd     | ra    |       |     | 16 bis 23 |
| cmd rd,ra,rb  | cnr       | rd     | ra    | rb    |     | 24 bis 30 |

```
constant nop: bit_vector(15 downto 0) := x"0000";
```

— *Sprungbefehl: Argumente Sprungziel und -bedingung*

```
function jmp(tgadr: bit_vector(7 downto 0);
 cond: t_cond:=alw) return bit_vector;
```

— *Argumente: Registeradresse und 8-Bit-Direktwert*

```
function cmd(w: t_cmd; rd:t_reg;
 imm: bit_vector(7 downto 0)) return bit_vector;
```

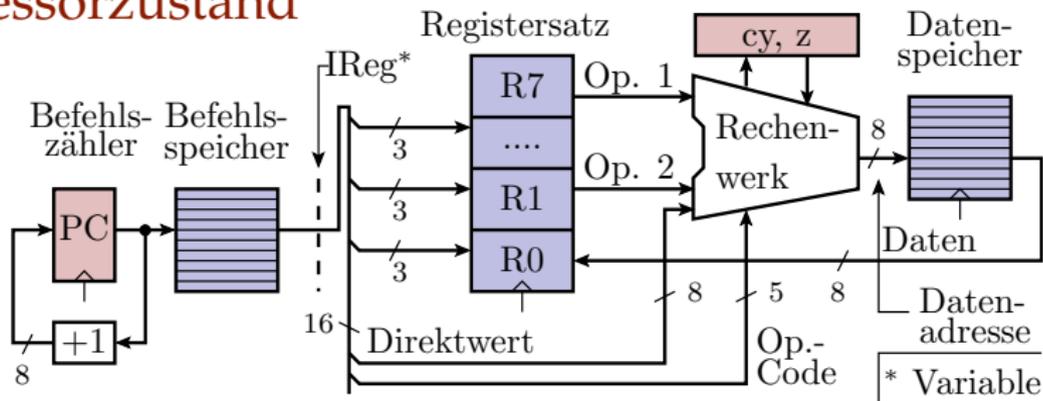
— *Argumente: 1 bis 3 Registeradressen*

```
function cmd(w: t_cmd; rd:t_reg; ra, rb: t_reg:=r0)
 return bit_vector;
```



# Prozessorzustand

## Prozessorzustand

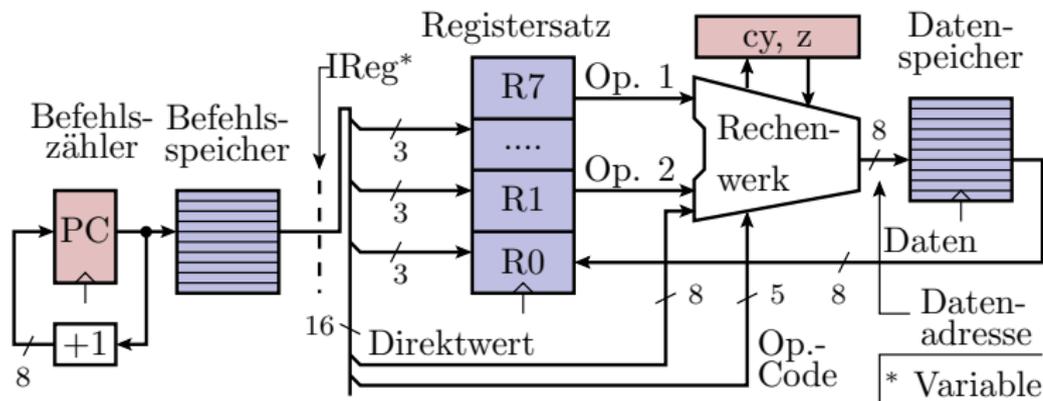


- Datentyp Befehlsspeicher:

```
type t_imem is array (natural range<>)
of bit_vector(15 downto 0);
```

- Textdarstellung für Befehle (z.B. »ld\_i r1,4a,..«):

```
function str_ir(w: bit_vector(15 downto 0))
return string;
```



- Datentyp Datenspeicher:

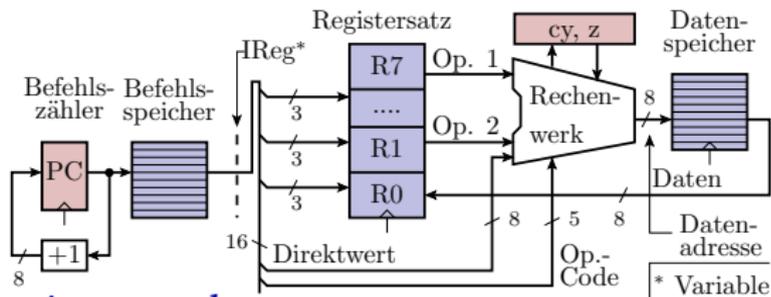
```
type t_dmem is array (natural range <>)
of bit_vector(7 downto 0);
```

- Textdarstellung für Datenspeicherbereiche:

```
function str(w: t_dmem) return string;
```



■ Prozessorzustand:



```
type t_proc_state is record
```

```
 PC : bit_vector(7 downto 0); — Befehlszähler
```

```
 IReg: bit_vector(15 downto 0); — Befehlswort
```

```
 cy: bit; — Carry-Flag
```

```
 z: bit; — Zero-Flag
```

```
 RSet: t_dmem(0 to 7); — Registersatz
```

```
end record;
```

■ Textdarstellung des Prozessorzustands:

```
constant c_proc_state: string :=
```

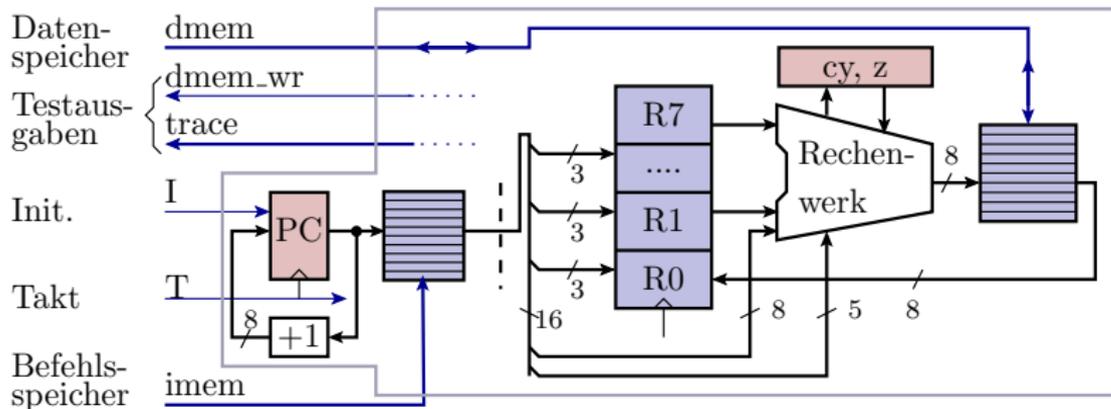
```
"PC|Befehl_assem.:_hex|r0_r1_r2_r3_r4_r5_r6_r7|c|z|";
```

```
function str(ps: t_proc_state) return string;
```



# Simulationsmodell

## Schnittstelle



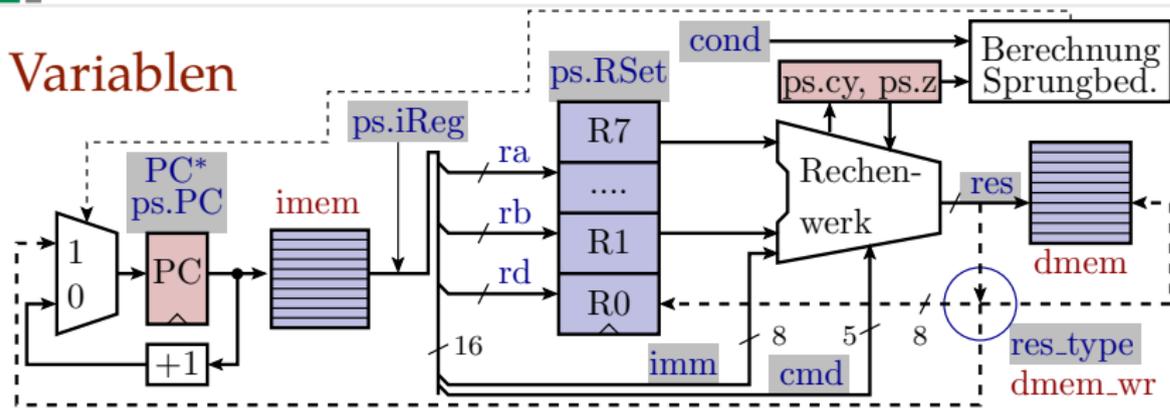
```

entity mipro is port (
 T, I: in bit;
 imem: in t_imem;
 dmem: inout t_dmem;
 trace: out t_proc_state;
 dmem_wr: out boolean);
end entity;

```



## Variablen



```
architecture a of mipro is begin
```

```
process(T, I)
```

```
variable cmd: t_cmd;
```

```
variable res: bit_vector(8 downto 0);
```

```
variable imm, PC: bit_vector(7 downto 0);
```

```
variable cond: bit_vector(2 downto 0);
```

```
variable rd, ra, rb: natural range 0 to 7;
```

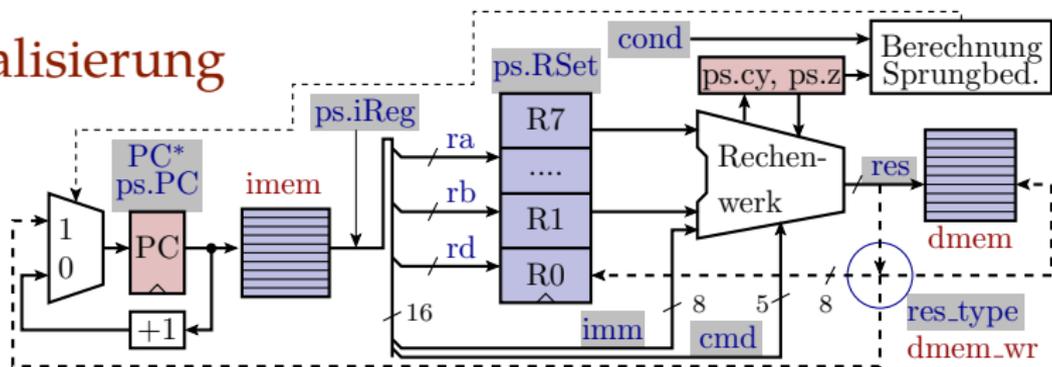
```
variable ps: t_proc_state;
```

```
type t_res_type is (non, rcz, rncz, ncz, rz);
```

```
variable res_type: t_res_type;
```

```
xx Variablen
* Folgewert
```

### Initialisierung



**begin**

**if** I='1' **then**

ps.PC := x"00";

**elsif** T'event **and** T='1' **then**

*<Register-Transfer-Funktion des Prozessors>*

**end if**;

trace <= ps; — *nach Berechnung Status mit altem*

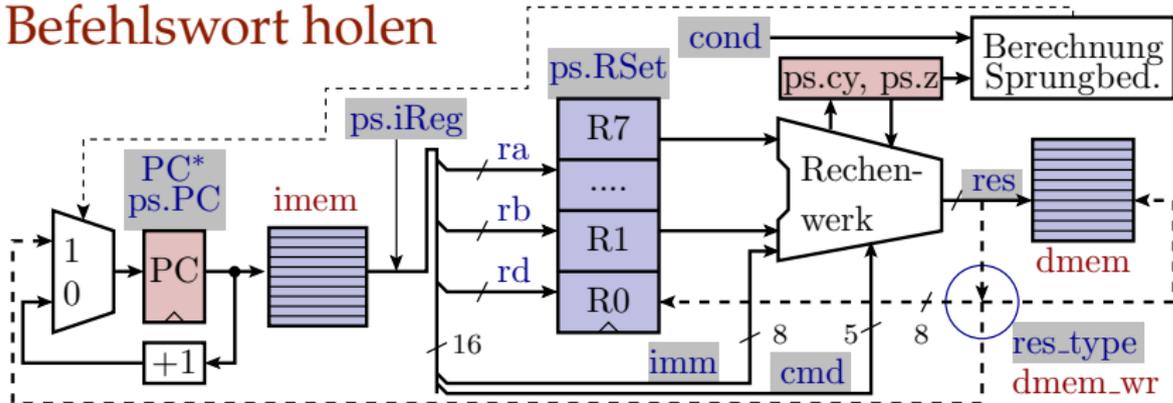
ps.PC := PC; — *PC ausgeben, dann PC aktualis.*

**end process**;

**end architecture**;

xx Variablen  
\* Folgewert

## Befehlsword holen



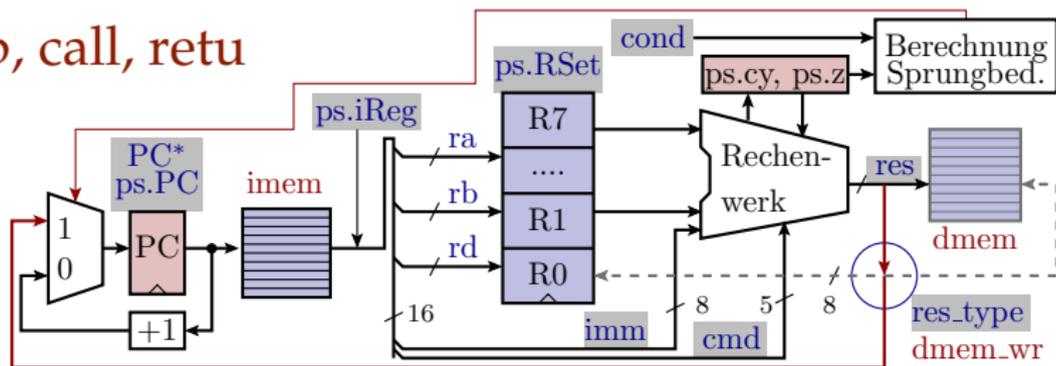
```

ps.IReg := imem(uint(ps.PC));
PC := inc(ps.PC);
cmd := t_cmd'val(uint(ps.IReg(15 downto 11)));
cond := ps.IReg(10 downto 8);
rd := uint(ps.IReg(10 downto 8));
ra := uint(ps.IReg(7 downto 5));
rb := uint(ps.IReg(4 downto 2));
imm := ps.IReg(7 downto 0);

```



## jump, call, retu

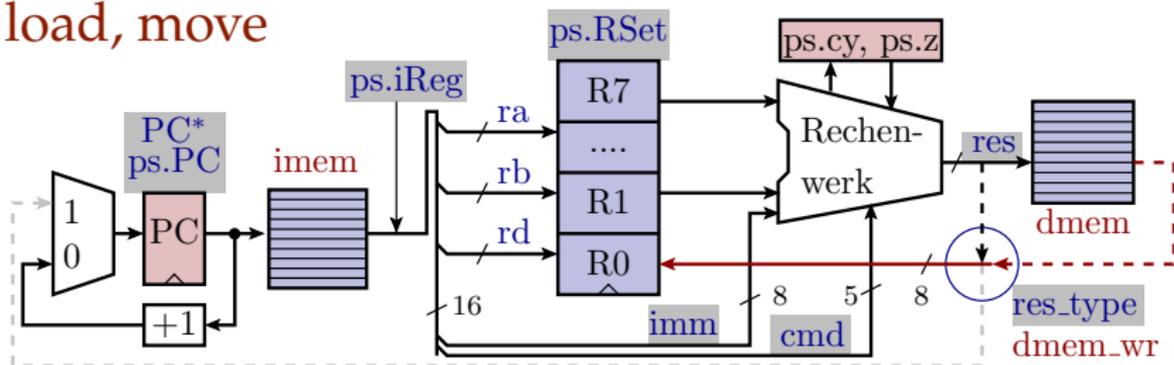


```

dmem_wr <= false; res_type := non;
case cmd is
when jump => -- Sprungbefehl
 if test_jmp_cond(cond, ps.cy, ps.z) then
 PC := imm; -- überschreibt PC+1
 end if;
when call => -- Unterprogrammaufruf
 ps.RSet(rd) := PC; PC := imm;
when retu => -- Rücksprung
 PC := ps.RSet(rd);

```

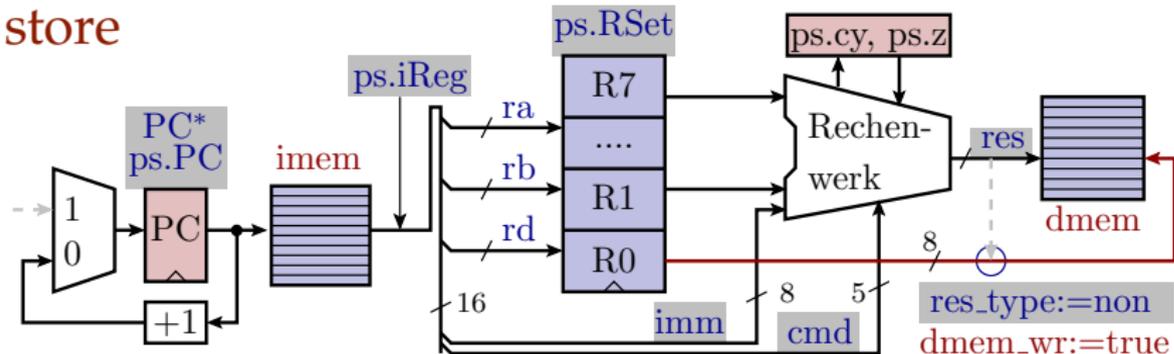
### load, move



- when** load  $\Rightarrow$  — Lade von konstanter Adresse  
 $\text{ps.RSet}(rd) := \text{dmem}(\text{uint}(\text{imm}));$
- when** ld\_r  $\Rightarrow$  — Lade von variabler Adresse  
 $\text{ps.RSet}(rd) := \text{dmem}(\text{uint}(\text{ps.RSet}(ra)));$
- when** ld\_i  $\Rightarrow$  — Lade eine Konstante  
 $\text{ps.RSet}(rd) := \text{imm};$
- when** move  $\Rightarrow$  — Kopiere von Ra nach Rd  
 $\text{ps.RSet}(rd) := \text{ps.RSet}(ra);$



## store



**when** stor => — *Speichern, konstante Adresse*

```
dmem(uint(imm)) <= ps.RSet(rd);
```

```
dmem_wr <= true;
```

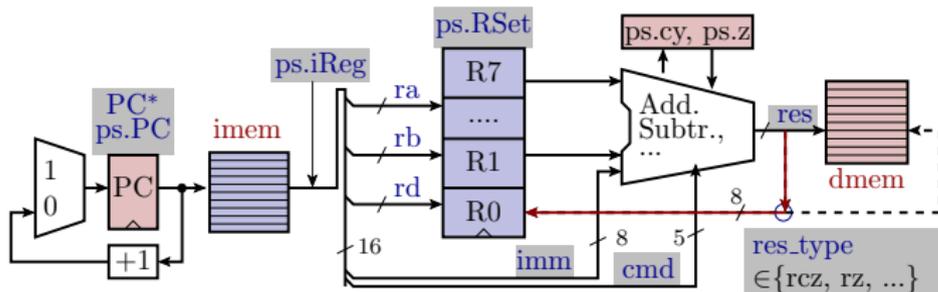
**when** st\_r => — *Speichern, variable Adresse*

```
dmem(uint(ps.RSet(ra))) <= ps.RSet(rd);
```

```
dmem_wr <= true;
```

»dmem\_wr« signalisiert dem Testrahmen eine Werteänderung des Datenspeichers, wurde vor der Fallunterscheidung auf »false« gesetzt und wird für Speicheroperationen mit »true« überschrieben.

### Addition



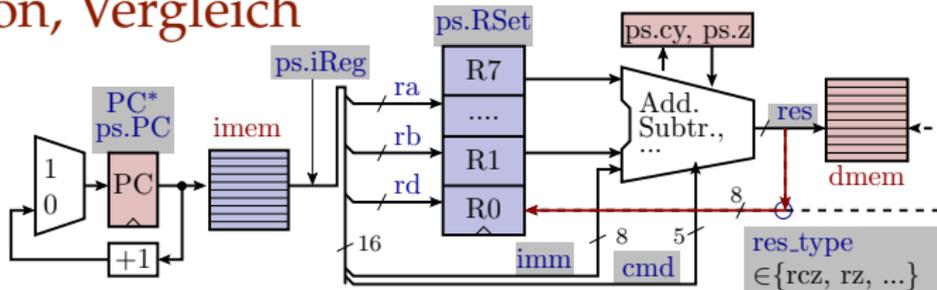
```

when addr => res_type := rcz;
 res := add(ps.RSet(ra), ps.RSet(rb), '0');
when addi => res_type := rcz;
 res := add(imm, ps.RSet(rd), '0');
when adcr => res_type := rcz;
 res := add(ps.RSet(ra), ps.RSet(rb), ps.cy);
when adci => res_type := rcz;
 res := add(imm, ps.RSet(rd), ps.cy);

```

Fallunterscheidung nach »res\_type« folgt nach der auszuführenden Operation. »rcz«: Ergebnis in Register schreiben, Carry- und Zero-Flag anpassen.

### Subtraktion, Vergleich



- Subtraktion: Addition mit dem Komplement:

**when** `subr` =>

```
res := add(ps.RSet(ra), not(ps.RSet(rb)), '1');
res_type := rncz; ...
```

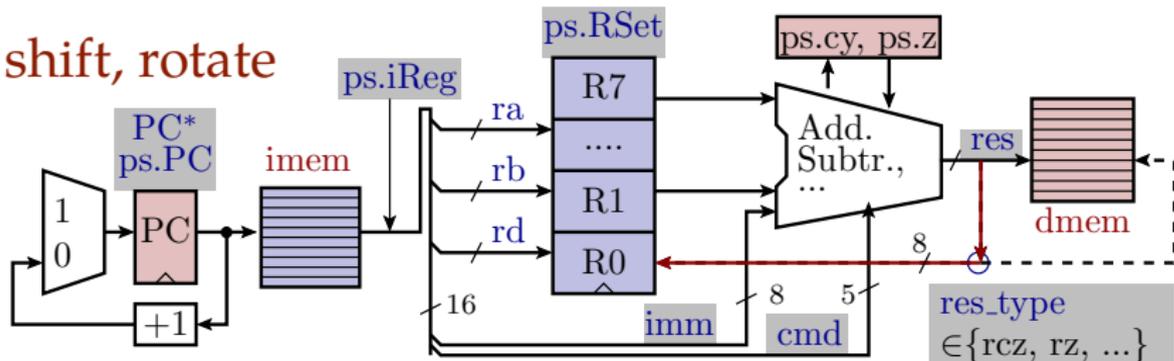
- Vergleich: Subtraktion, die nur das Carry- und Zero-Flag beeinflusst:

**when** `comp` =>

```
res := add(imm, not(ps.RSet(rd)), '1');
res_type := ncz;
```



shift, rotate

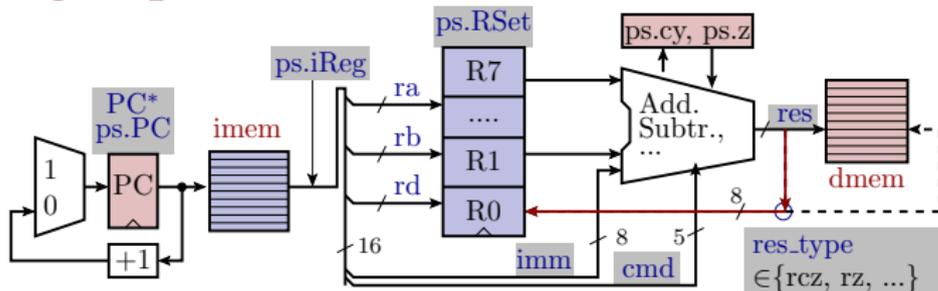


```

when sh_l => res_type := rcz;
 res := ps.RSet(ra) & '0';
when rotl => res_type := rcz;
 res := ps.RSet(ra) & ps.cy;
when sh_r => res_type := rcz;
 res := '0' & ps.RSet(ra);
 res := res(0)&res(8 downto 1);
when rotr => res_type := rcz;
 res := ps.cy & ps.RSet(ra);
 res := res(0)&res(8 downto 1);

```

## Bitweise Logikoperationen



```

when notr => res_type := rz;
 res(7 downto 0) := not ps.RSet(ra);
when andr => res_type := rz;
 res(7 downto 0) := ps.RSet(ra) and ps.RSet(rb);
when andi => res_type := rz;
 res(7 downto 0) := imm and ps.RSet(rd);
when ...
 <für or_r, or_i, xorr und xori vergleichbare Befehlsfolgen>
when others => res_type := non;
end case;

```

## Ergebnis schreiben

```
case res_type is
 when rcz => — addr, addi, adcr, adci, sh_l, ...
 ps.RSet(rd) := res(7 downto 0);
 ps.cy := res(8);
 ps.z := is_zero(res(7 downto 0));
 when rncz => — subr, subi, sbcr, sbci,
 ps.RSet(rd) := res(7 downto 0);
 ps.cy := not res(8);
 ps.z := is_zero(res(7 downto 0));
 when ncz => — comp, cmpc
 ps.cy := not res(8);
 ps.z := is_zero(res(7 downto 0));
 when rz => — notr, andr, andi, or_r, or_i, ...
 ps.RSet(rd) := res(7 downto 0);
 ps.z := is_zero(res(7 downto 0));
 when others => null;
end case;
```



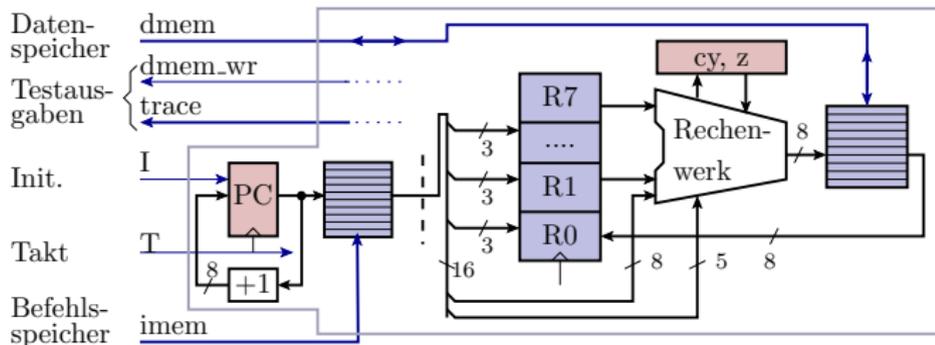
# Testrahmen

## Das zu testende Programm

```
use work.mipro_pack.all; use ...
entity test_mipro is end entity;
architecture a of test_mipro is
 constant imem: t_imem(0 to 8) := (
 0=>cmd(ld_i , r1 , x"3a"),
 1=>cmd(ld_i , r0 , x"73"),
 2=>cmd(ld_i , r3 , x"E7"),
 3=>cmd(ld_i , r2 , x"13"),
 4=>cmd(addr , r5 , r1 , r3),
 5=>cmd(adcr , r4 , r0 , r2),
 6=>cmd(subi , r5 , x"23"),
 7=>cmd(sbc_i , r4 , x"86"),
 others => nop);
```

Zuweisung der Befehlswoorte an ausgewählte Befehlsspeicherplätze.  
Alle anderen Adressen werden mit x"0000" (nop) initialisiert.

## Signal, Taktperiode, Testobjektinstanz



```

signal T, I: in bit;
signal dmem: t_dmem(0 to 7);
signal ps: t_proc_state;
signal dmem_wr: boolean;
constant tP: delay_length := 10 ns;

```

**begin**

```

DUT: entity work.mipro

```

```

port map(T, I, imem, dmem, ps, dmem_wr);

```



## Erzeugung des Takts und der Trace-Tabelle

```

process
begin
 write(output, lf & c_proc_state); — Tabellenkopf
 while uint(ps.PC)<imem'high loop — für alle Befehle
 T <= '1' after 5 ns, '0' after 10 ns; — Takt erzeugen und warten
 wait for 10 ns;
 write(output, str(ps)); — Prozessorzustand
 if dmem_wr then — nach Speicherzugriff
 write(output, "str(dmem)␣=␣" & str(dmem)); — Datenspeicherinhalt ausgeben
 end if;
 end loop;
 wait;
end process;

```

### Trace-Ausgabe:

| PC | Befehl | assem.:       | hex | r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | c z |
|----|--------|---------------|-----|----|----|----|----|----|----|----|----|-----|
| 00 | ld_i   | r1,3a,..:293a |     | 00 | 3a | 00 | 00 | 00 | 00 | 00 | 00 | 0 0 |



# Testbeispiele



## Addition und Subtraktion

### Programmieraufgabe:

```

r0:r1 = 0x733A; r2:r3 = 0x13E7;
r4:r5 = r0:r1 + r2:r3; Ergebnis: 0x8721
r6:r7 = r0:r1 - r2:r3; Ergebnis: 0x5F53

```

### Programm mit ergänzten Registerinhalten:

| PC | Befehl | assem.:     | hex  | r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | c | z |
|----|--------|-------------|------|----|----|----|----|----|----|----|----|---|---|
| 00 | ld_i   | r1, 3a, ..: | 293a | 00 | 3a | 00 | 00 | 00 | 00 | 00 | 00 | 0 | 0 |
| 01 | ld_i   | r0, 73, ..: | 2873 | 73 | 3a | 00 | 00 | 00 | 00 | 00 | 00 | 0 | 0 |
| 02 | ld_i   | r3, e7, ..: | 2be7 | 73 | 3a | 00 | e7 | 00 | 00 | 00 | 00 | 0 | 0 |
| 03 | ld_i   | r2, 13, ..: | 2a13 | 73 | 3a | 13 | e7 | 00 | 00 | 00 | 00 | 0 | 0 |
| 04 | addr   | r5, r1, r3: | c52c | 73 | 3a | 13 | e7 | 00 | 21 | 00 | 00 | 1 | 0 |
| 05 | adcr   | r4, r0, r2: | cc08 | 73 | 3a | 13 | e7 | 87 | 21 | 00 | 00 | 0 | 0 |
| 06 | subr   | r7, r1, r3: | d72c | 73 | 3a | 13 | e7 | 87 | 21 | 00 | 53 | 1 | 0 |
| 07 | sbc r  | r6, r0, r2: | de08 | 73 | 3a | 13 | e7 | 87 | 21 | 5f | 53 | 0 | 0 |

## Testbeispiel mit Speicherzugriff und Schleife

```
r0 = 1; r1 = 34;
M: dmem(r0) = r1;
r1 = r1 - r0; r0 = r0 +1;
wenn r0 ≤ 3 springe zu M
```

Sprungbedingung für r0=2 und 3 erfüllt. 3 Schleifendurchläufe.

```
0000: ld_i r0 ,01 ,..
0001: ld_i r1 ,34 ,..
0002: st_r r1 ,r0 ,..
0003: subr r1 ,r1 ,r0
0004: addi r0 ,01 ,..
0005: comp r0 ,03 ,..
0006: jump 02 ,leq..
0007: noop .. ,.. ,..
```

- In welcher Reihenfolge werden die Anweisungen abgearbeitet?



```

PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,01,...:2801|01 00 00 00 00 00 00 00|0|0|
01|ld_i r1,34,...:2934|01 34 00 00 00 00 00 00|0|0|
02|st_r r1,r0,...:9100|01 34 00 00 00 00 00 00|0|0|
 dmem = [00 34 00 00 00 00 00 00]
03|subr r1,r1,r0:d120|01 33 00 00 00 00 00 00|0|0|
04|addi r0,01,...:4001|02 33 00 00 00 00 00 00|0|0|
05|comp r0,03,...:3003|02 33 00 00 00 00 00 00|0|0|
06|jump 02,leq...:0b02|02 33 00 00 00 00 00 00|0|0|
02|st_r r1,r0,...:9100|02 33 00 00 00 00 00 00|0|0|
 dmem = [00 34 33 00 00 00 00 00]
03|subr r1,r1,r0:d120|02 31 00 00 00 00 00 00|0|0|
04|addi r0,01,...:4001|03 31 00 00 00 00 00 00|0|0|
05|comp r0,03,...:3003|03 31 00 00 00 00 00 00|0|0|
06|jump 02,leq...:0b02|03 31 00 00 00 00 00 00|0|0|
02|st_r r1,r0,...:9100|03 31 00 00 00 00 00 00|0|0|
 dmem = [00 34 33 31 00 00 00 00]
03|subr r1,r1,r0:d120|03 2e 00 00 00 00 00 00|0|0|
04|addi r0,01,...:4001|04 2e 00 00 00 00 00 00|0|0|
05|comp r0,03,...:3003|04 2e 00 00 00 00 00 00|0|1|
06|jump 02,leq...:0b02|04 2e 00 00 00 00 00 00|0|1|
07|noop 0000|04 2e 00 00 00 00 00 00|0|1|

```



# RISC-Prozessor



### RISC-Prozessoren

- Die in den letzten 30 Jahren neu entwickelten Prozessoren haben eine RISC<sup>6</sup>-Architektur.
- Die Befehlssätze sind (überwiegend) auf Befehle reduziert, die in einem Zeitslot in einer Pipeline abarbeitbar sind.
- Zuvor gebräuchliche Befehle für komplexere Operationen, die mehrere Zeitslots benötigen, wie das Inkrementieren einer Variablen im Speicher werden durch Befehlsfolgen nachgebildet.
- Pipeline-Verarbeitung hat ein nicht zu übertreffend günstiges Aufwand-Nutzen-Verhältnis.

In diesem Abschnitt wird das Simulationsmodell des Minimalprozessors »MiPro« zu dem eines RISC-Prozessors mit Pipeline erweitert.

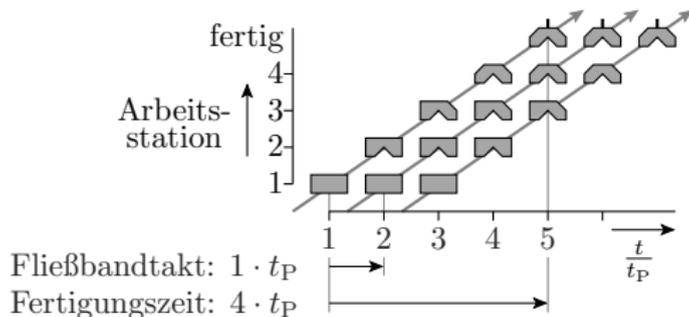
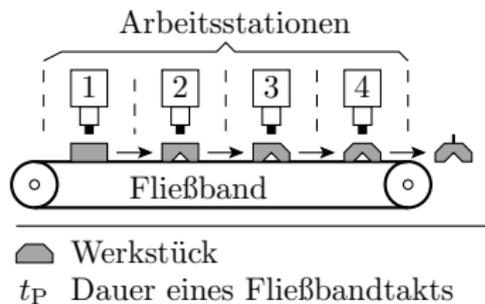
---

<sup>6</sup>RISC ist ein Akronym für **R**educed **I**nstruction **S**et **C**omputer.



# Pipeline-Verarbeitung

## Pipeline-Verarbeitung

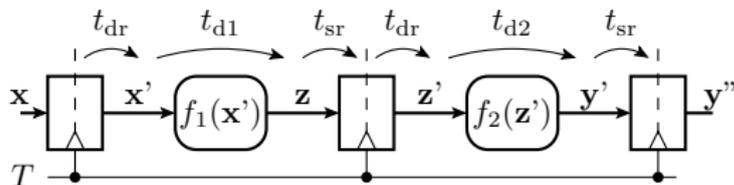
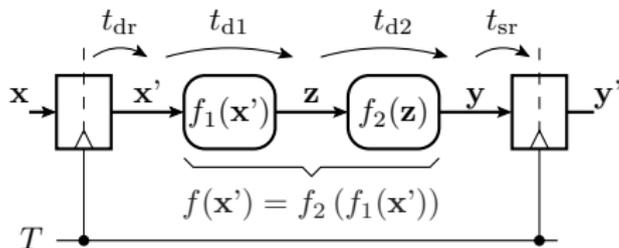


- Aufteilung einer Gesamtaufgabe in  $N_P$  Arbeitsschritte.
- Gesamtaufwand je Objekt:  $N_P \cdot t_P$  ( $t_P$  – Periode Fließbandtakt)
- Je Takt wird ein Objekt fertig.

### Tatsache:

Parallelverarbeitung ohne vervielfachten Aufwand.

## Angewandt auf Hardware



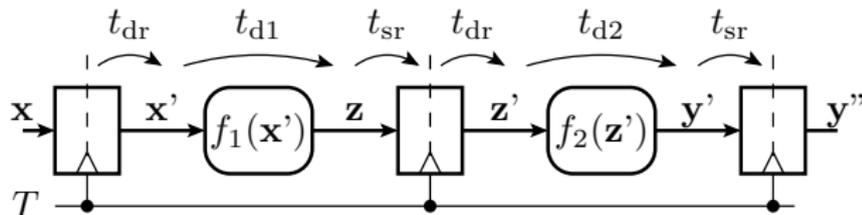
- $t_{dr}$  Registerverzögerungszeit
- $t_{d1}$  Verzögerungszeit von  $f_1$
- $t_{d2}$  Verzögerungszeit von  $f_2$
- $t_{sr}$  Registervorhaltezeit
- $x$  Eingangssignal
- $z$  Zwischensignal
- $y$  Ausgangssignal
- $\dots'$  abgetastetes Signal

### minimale Taktperiode

- ohne Pipeline  
 $t_{dr} + t_{sr} + t_{d1} + t_{d2}$
- mit Pipeline  
 $t_{dr} + t_{sr} + \max(t_{d1}, t_{d2})$

- Aufwand: ein zusätzliches Register je Pipeline-Stufe.
- Viel billiger als mehrfache Hardware. Vorzugslösung für Parallelverarbeitung.

## Beschreibung einer Pipeline in VHDL



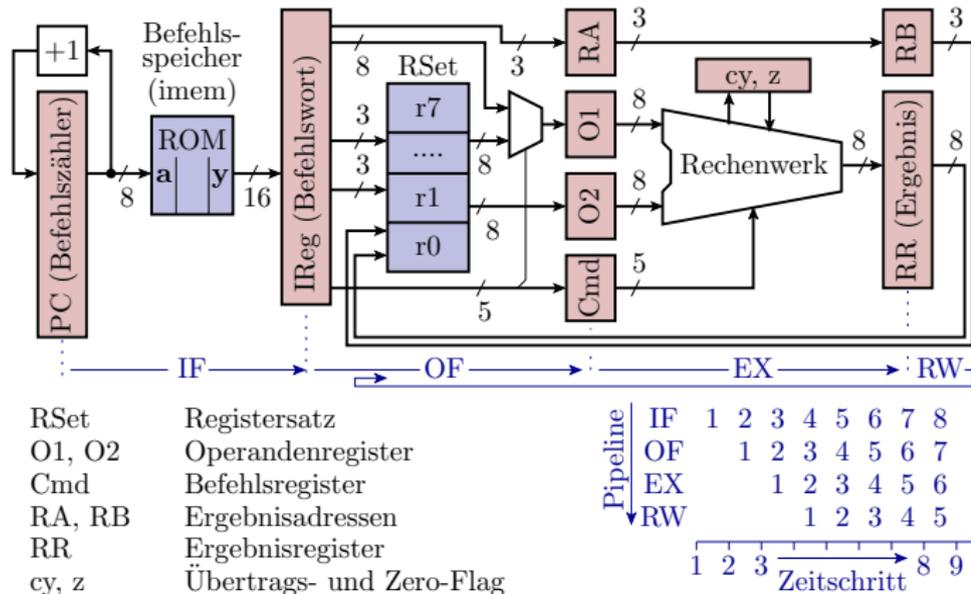
```

process (T)
 if rising_edge(T) then
 x_del <=x;
 z_del <=f1 (x_del);
 y_del2<=f2 (z_del);
 end if;
end process;

```

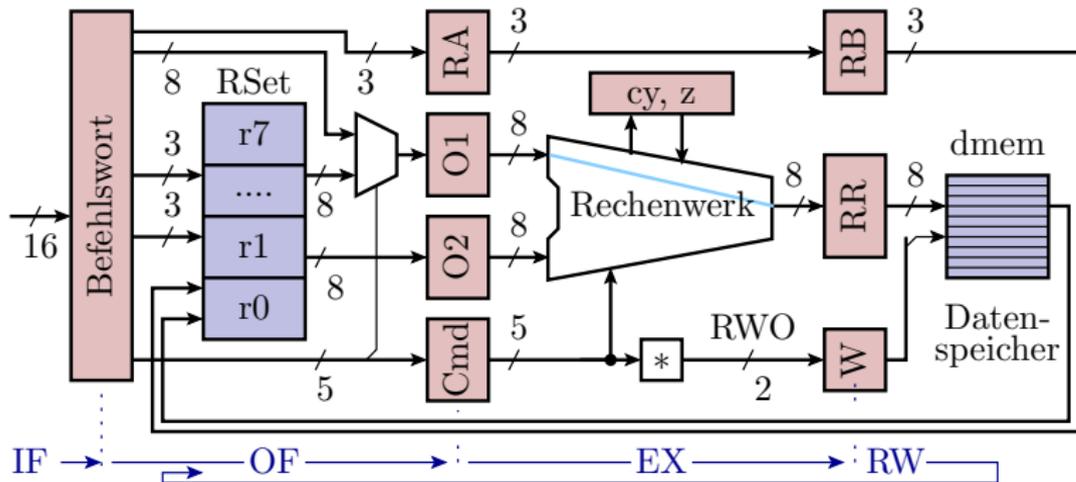
Register werden durch Signalzuweisungen, wenn »aktive Taktflanke« beschrieben.

## Verarbeitungs-Pipeline für unseren RISC-Prozessor



Pipeline-Phasen: IF – Instruction Fetch; OF – Operand Fetch; EX – Execute; RW – Result Write.

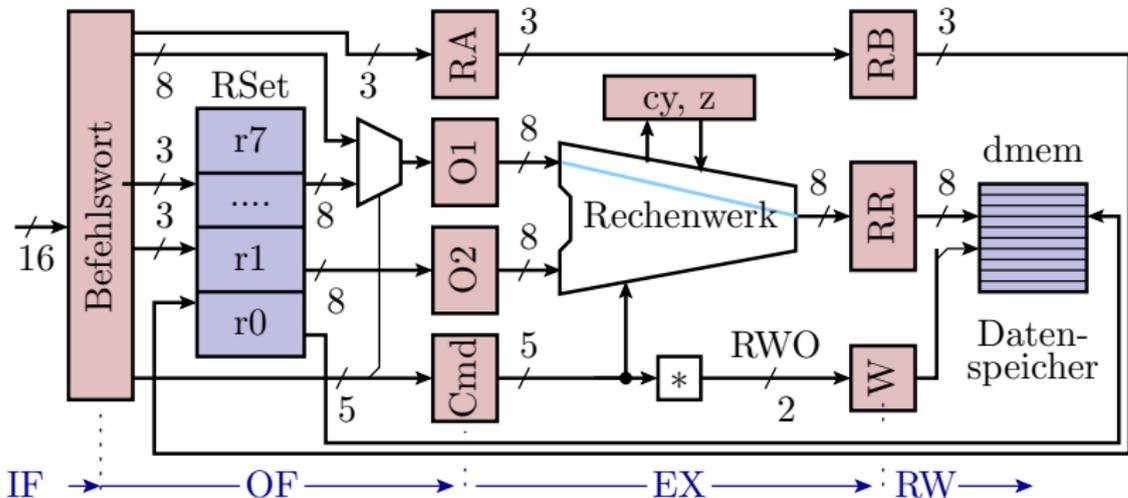
## Modifizierte Pipeline für Ladebefehle



\* Bildung des 2-Bit-RWO-Codes für die RW-Phase ( $RWO \in \{-, R, L, S\}$  für kein Ergebnis speichern, Ergebnis in Register speichern, Lade- oder Speicheroperation).

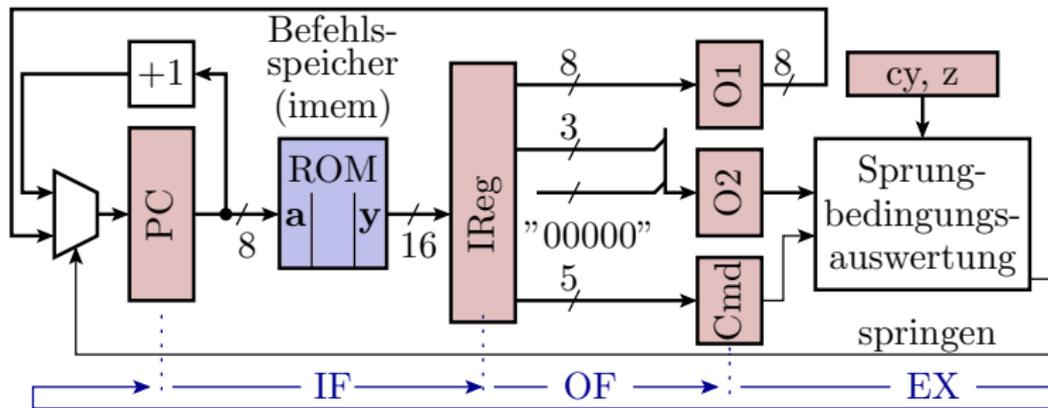
- EX-Phase: Adressrechnung, Weitergabe RWO-Code Laden.
- RW-Phase: Laden des Zielregisters mit Speicherinhalt.

## Pipeline für Speicheroperationen



- EX-Phase: Adressrechnung, Weitergabe RWO-Code Speichern.
- RW-Phase: Kopieren des »Zielregisters« in den Speicher.

## Sprung-Pipeline



jump imm,cond; if (b) pc = imm; else pc++;

- OF-Phase: Sprungziel in O1 und Sprungbedingung in O2 laden.
- EX-Phase: bedingte Übernahme von O1 in den Befehlszähler.
- RW-Phase: keine Operation.





# Pipeline-Auslastung



## Pipeline-Auslastung

Verarbeitungsergebnisse werden erst zwei Takte nach Lesen der Operanden geschrieben. Verursacht Probleme. Beispiel:

Addition von vier Registerinhalten

Adr Befehl

0 B1: addr r0,r0,r1  
 1 B2: addr r0,r0,r2  
 2 B3: addr r0,r0,r3

\* Register lesen  
 \**n* lesen und mit *n* überschreiben

BW Befehlsword

| PC | BW | <i>r</i> <sub>0</sub> | <i>r</i> <sub>1</sub> | <i>r</i> <sub>2</sub> | <i>r</i> <sub>3</sub> | O1 | O2 | RR |
|----|----|-----------------------|-----------------------|-----------------------|-----------------------|----|----|----|
| 0  |    | 7                     | 2                     | 11                    | 17                    |    |    |    |
| 1  | B1 |                       |                       |                       |                       |    |    |    |
| 2  | B2 | *                     | *                     |                       |                       | 7  | 2  |    |
| 3  | B3 | *                     |                       | *                     |                       | 7  | 11 | 9  |
| 4  |    | *9                    |                       |                       | *                     | 7  | 17 | 18 |
| 5  |    | 18                    |                       |                       |                       |    |    | 24 |
| 6  |    | 24                    |                       |                       |                       |    |    |    |

In *r*<sub>0</sub> steht in Takt 5  $r_0 + r_1$ , in Takt 6  $r_0 + r_2$  und in Takt 7  $r_0 + r_3$ . Statt  $r_0 + r_1 + r_2 + r_3$  wird  $r_0 + r_3$ , d.h. ein falsches Ergebnis berechnet.

Einfügen von noop<sup>7</sup>-Befehlen

|     |                   | IF | OF   |                |                |                | RW             | EX |    |    |
|-----|-------------------|----|------|----------------|----------------|----------------|----------------|----|----|----|
|     |                   | →  | →    |                |                |                | ←              | →  |    |    |
| Adr | Befehl            | PC | BW   | r <sub>0</sub> | r <sub>1</sub> | r <sub>2</sub> | r <sub>3</sub> | O1 | O2 | RR |
| 0   | B1: addr r0,r0,r1 | 0  |      | 7              | 2              | 11             | 17             |    |    |    |
| 1   | noop              | 1  | B1   |                |                |                |                |    |    |    |
| 2   | noop              | 2  | noop | *              | *              |                |                | 7  | 2  |    |
| 3   | B2: addr r0,r0,r2 | 3  | noop |                |                |                |                |    |    | 9  |
| 4   | noop              | 4  | B2   | 9              |                |                |                |    |    |    |
| 5   | noop              | 5  | noop | *              |                | *              |                | 9  | 11 |    |
| 6   | B3: addr r0,r0,r3 | 6  | noop |                |                |                |                |    |    | 20 |
| 7   | noop              | 7  | B3   | 20             |                |                |                |    |    |    |
| 8   | noop              | 8  | noop | *              |                |                | *              | 20 | 17 |    |
| 9   | noop              | 9  | noop |                |                |                |                |    |    | 37 |
| 10  | noop              | 10 | noop | 37             |                |                |                |    |    |    |

\* Register lesen

\**n* lesen und mit *n* überschreiben

■ Die drei Additionen benötigen bis zum Abschluss 11 Takte.

<sup>7</sup>noop – No Operation, für den Beispielprozessor Op-Code 0x0000.

## Optimierte Berechnungsreihenfolge

|   |                          | IF | OF    |       |       |       | RW | EX |    |
|---|--------------------------|----|-------|-------|-------|-------|----|----|----|
|   | PC                       | BW | $r_0$ | $r_1$ | $r_2$ | $r_3$ | O1 | O2 | RR |
| 0 | B1: addr $r_0, r_0, r_1$ | 0  | 7     | 2     | 11    | 17    |    |    |    |
| 1 | B2: addr $r_2, r_2, r_3$ | 1  | B1    |       |       |       |    |    |    |
| 2 | noop                     | 2  | B2    | *     | *     |       | 7  | 2  |    |
| 3 | noop                     | 3  | noop  |       | *     | *     | 11 | 17 | 9  |
| 4 | B3: addr $r_0, r_0, r_2$ | 4  | noop  | 9     |       |       |    |    | 28 |
|   |                          | 5  | B3    |       | 28    |       |    |    |    |
|   |                          | 6  | noop  | *     | *     |       | 9  | 28 |    |
|   |                          | 7  | noop  |       |       |       |    |    | 37 |
|   |                          | 8  | noop  | 37    |       |       |    |    |    |

\* Register lesen

\* $n$  lesen und mit  $n$  überschreiben

Die Additionen  $r_0 + r_1$  und  $r_2 + r_3$  können direkt nacheinander erfolgen. Ausführungszeit zwei Takte weniger.

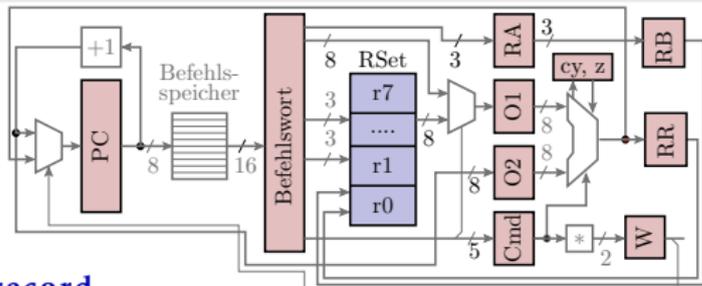
Abarbeitungszeit und Größe von Programmen hängen erheblich von der Compiler-Optimierung ab.



# Simulationsmodell



# Prozessorzustand



```
type t_proc_state is record
```

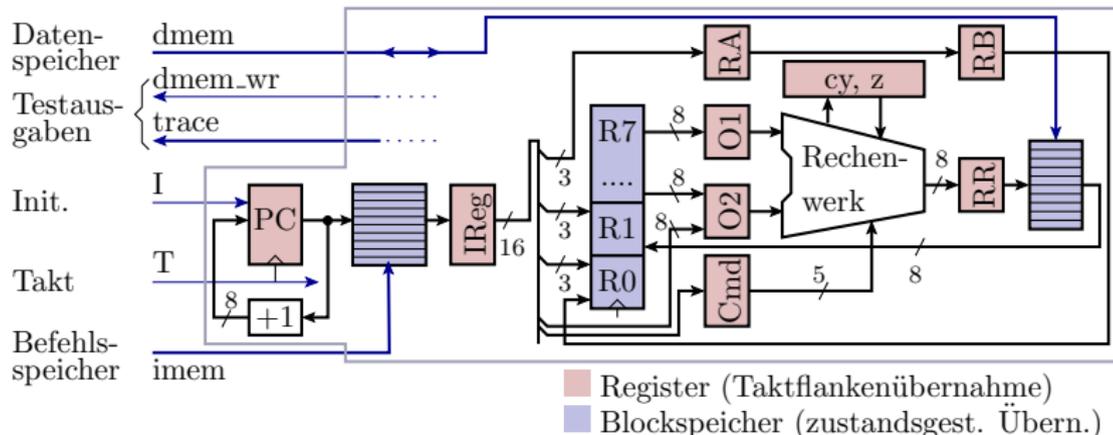
```

PC : t_dat(7 downto 0); -- Befehlszähler
IReg: t_dat(15 downto 0); -- Befehlswort
Cmd: t_dat(4 downto 0); -- Befehlsregister
O1: t_dat(7 downto 0); -- Operand 1
O2: t_dat(7 downto 0); -- Operand 2
cy: t_dat(0 downto 0); -- Übertrag
z: t_dat(0 downto 0); -- Zero-Flag
RR: t_dat(7 downto 0); -- Ergebnisregister
RA: t_dat(2 downto 0); -- Ergebnisadresse EX
RB: t_dat(2 downto 0); -- Ergebnisadresse RW
W: t_dat(1 downto 0); -- RWO-Code
RSet: t_dmem(0 to 7); -- Registersatz

```

```
end record;
```

## Schnittstelle fast wie »MiPro«



```

entity risc is port(
 T, I: in bit;
 imem: in t_imem;
 dmem: inout t_dmem;
 ps: buffer t_proc_state; --Trace, Prozessorstatus
 dmem_wr: out boolean);
end entity;

```



```
entity risc is
 generic(dm_init: t_dmem := (others => x"00"));
 port(
 T, I: in bit;
 imem: in t_imem;
 dmem: inout t_dmem;
 ps: buffer t_proc_state;
 dmem_wr: out boolean);
end entity;
```

- Befehlsspeicher (imem): Vom Simulationsmodell nur lesbar.
- Datenspeicher (dmem): Vom Simulationsmodell les- und schreibbar.
- Prozessorzustand (ps): Testausgabe, von außen nur lesbar.
- »generic dm\_init«: Optionaler Anfangswert für den Datenspeicher, um Programmbeispiele mit initialisiertem Datenspeicher simulieren zu können.



## Beschreibungsrahmen

```
architecture a of risc is
begin
 process(T, I)
 variable res: bit_vector(8 downto 0);
 variable cmd: t_cmd;
 begin
 if I='1' then
 ps.PC <= x"00";
 dmem <= dm_init;
 elsif T'event and T='1' then
 <RT-Funktion des RISC-Prozessors>
 end if;
 end process;
end architecture;
```

- Durch den Pipeline-Einbau sind aus vielen MiPro-Variablen Register geworden.

## IF- und OF-Phase

— *IF (Instruction Fetch)*

```
ps.PC <= inc(ps.PC);
ps.IReg <= imem(uint(ps.PC));
```

— *OF (Operand Fetch)*

```
ps.Cmd <= ps.IReg(15 downto 11);
ps.RA <= ps.IReg(10 downto 8);
cmd := t_cmd'val(uint(ps.IReg(15 downto 11)));
```

```
case cmd is
```

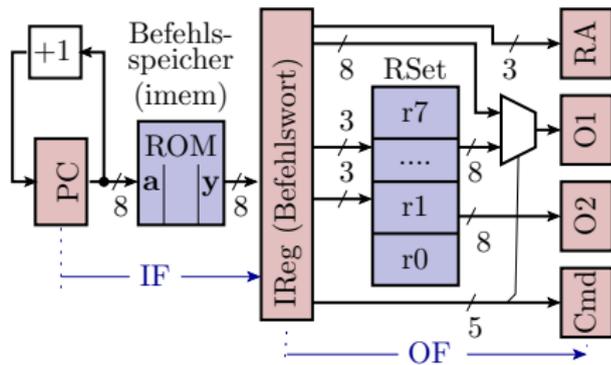
```
 when noop =>
```

```
 ps.O1 <= x"00";
```

```
 ps.O2 <= x"00";
```

*<ps.O1 und ps.O2 für die anderen Befehle siehe nächste Folie>*

```
end case;
```





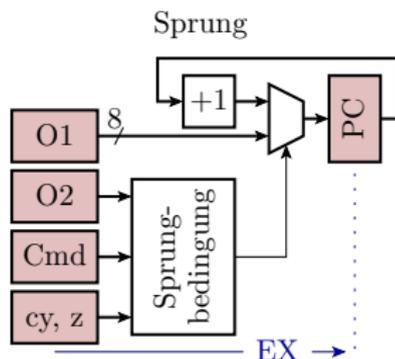
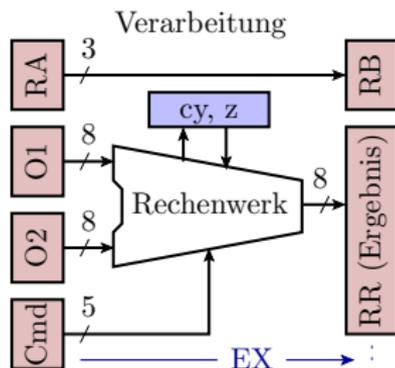
```
when jump =>
 ps.O1 <= ps.IReg(7 downto 0); --Sprungziel
 ps.O2 <= "0000" & ps.IReg(10 downto 8); --Sprungbeding.
when call =>
 ps.O1 <= ps.IReg(7 downto 0); --Sprungziel
 ps.O2 <= ps.PC; --Rücksprungadr.
when load|stor|ld_i =>
 ps.O1 <= ps.IReg(7 downto 0); --Datensp.-Adr.
 ps.O2 <= x"00"; --ungenutzt
when retu =>
 ps.O1 <= ps.RSet(uint(ps.IReg(10 downto 8))); --Sprungziel
 ps.O2 <= x"00"; --ungenutzt
when move|ld_r|st_r|sh_l|sh_r|rotl|rotr|notr =>
 ps.O1 <= ps.RSet(uint(ps.IReg(7 downto 5))); --Operand
 ps.O2 <= x"00"; --ungenutzt
when addr|adcr|subr|sbcr|andr|or_r|xorr =>
 ps.O1 <= ps.RSet(uint(ps.IReg(7 downto 5))); --R-Wert A
 ps.O2 <= ps.RSet(uint(ps.IReg(4 downto 2))); --R-Wert B
when others =>
 ps.O1 <= ps.IReg(7 downto 0); --Konstante
 ps.O2 <= ps.RSet(uint(ps.IReg(10 downto 8))); --R-Wert
```

## EX- (Ausführungs-) Phase

```

ps.RB <= ps.RA;
cmd := t_cmd'val(uint(ps.Cmd));
case cmd is
 when jump => — Sprung
 if test_jump_cond(ps.O2(2 downto
 0), ps.cy, ps.z) then
 ps.PC <= ps.O1;
 end if;
 when call => — Unterprogrammaufr.
 ps.PC <= ps.O1;
 ps.RR <= ps.O2;
 when retu => — Rücksprung
 ps.PC <= ps.O1;
 — Laden, Speichern, Kopieren
 when ld_i | load | ld_r | stor | st_r | move =>
 ps.RR <= ps.O1;

```





**when** `addr|addi` => — *Addition ohne Übertrag*

```
res := add(ps.O1, ps.O2, '0');
ps.RR <= res(7 downto 0);
ps.cy <= res(8);
ps.z <= is_zero(res(7 downto 0));
```

**when** `adcr|adci` => — *Addition mit Übertrag*

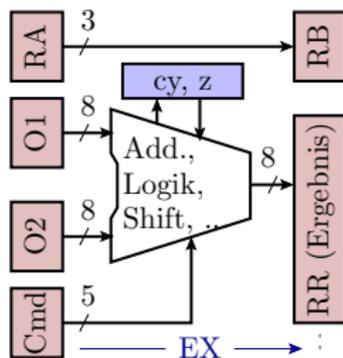
```
res := add(ps.O1, ps.O2, ps.cy);
ps.RR <= res(7 downto 0);
ps.cy <= res(8);
ps.z <= is_zero(res(7 downto 0));
```

**when** `<Subtraktion>` =>

`<Addition neg. Subtrahend und Übertrag>`

**when** `sh_l` => — *Linksverschiebung*

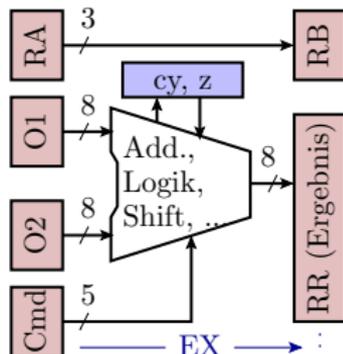
```
res := ps.O1 & '0';
ps.cy <= res(8);
ps.RR <= res(7 downto 0);
ps.Z <= is_zero(res(7 downto 0));
```



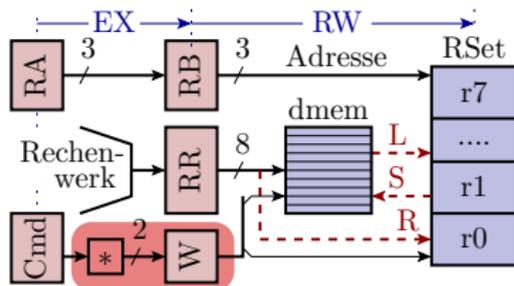
```

when <Rechtsverschiebung, Rotation> =>
 <Vergleichbare Operation wie sh_l>
when notr => — bitweise Negation
 ps.RR <= not ps.O1;
 ps.Z <= is_zero(not ps.O1);
when andr|andi => — bitweises UND
 ps.RR <= ps.O1 and ps.O2;
 ps.Z <= is_zero(ps.O1 and ps.O2);
when <ODER oder EXOR> =>
 <Vergleichbare Operation wie bei UND>
when others =>
 null;
end case;

```



## Berechnung der RW-Operation für die RW-Phase



```
case cmd is
```

```
when noop | jump | comp | cmpc | retu =>
 ps.W <= "00"; — RW-Phase: keine Operation
```

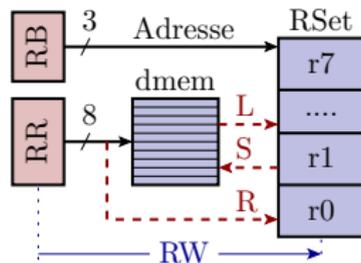
```
when load | ld_r =>
 ps.W <= "10"; — RW-Phase: RSet(RB) := dmem(RR)
```

```
when stor | st_r =>
 ps.W <= "01"; — RW-Phase: dmem(RR) := RSet(RB)
```

```
when others =>
 ps.W <= "11"; — RW-Phase: RSet(RB) := RR
```

```
end case;
```

## RW-Phase (Ergebnis schreiben)



```
dmem_wr <= false;
```

```
case ps.W is
```

```
 when "11" => — Ergebnis schreiben
```

```
 ps.RSet(uint(ps.RB)) <= ps.RR;
```

```
 when "10" => — Ladeoperation
```

```
 ps.RSet(uint(ps.RB)) <= dmem(uint(ps.RR));
```

```
 when "01" => — Speicheroperation
```

```
 dmem(uint(ps.RR)) <= ps.RSet(uint(ps.RB));
```

```
 dmem_wr <= true;
```

```
 when "00" => null; — keine Operation
```

```
end case;
```



# Testbeispiele



## Addition

r0:r1 = 733A; r2:r3 = 13E7

r4:r5 = r0:r1 + r2:r3 (Ergebnis: 8721)

r4:r5 = 8623 - r4:r5 (Ergebnis: FF02)

| PC | Befehlswort      | O1 | O2 | RR | c | z | RA | RB | W | r0 | r1 | r2 | r3 | r4 | r5 |
|----|------------------|----|----|----|---|---|----|----|---|----|----|----|----|----|----|
| 01 | ld_i r1, 3a, ..  | 00 | 00 | 00 | 0 | 0 | r0 | r0 | - | 00 | 00 | 00 | 00 | 00 | 00 |
| 02 | ld_i r0, 73, ..  | 3a | 00 | 00 | 0 | 0 | r1 | r0 | - | 00 | 00 | 00 | 00 | 00 | 00 |
| 03 | ld_i r3, e7, ..  | 73 | 00 | 3a | 0 | 0 | r0 | r1 | R | 00 | 00 | 00 | 00 | 00 | 00 |
| 04 | ld_i r2, 13, ..  | e7 | 00 | 73 | 0 | 0 | r3 | r0 | R | 00 | 3a | 00 | 00 | 00 | 00 |
| 05 | noop .., .., ..  | 13 | 00 | e7 | 0 | 0 | r2 | r3 | R | 73 | 3a | 00 | 00 | 00 | 00 |
| 06 | addr r5, r1, r3  | 00 | 00 | 13 | 0 | 0 | r0 | r2 | R | 73 | 3a | 00 | e7 | 00 | 00 |
| 07 | adcr r4, r0, r2  | 3a | e7 | 13 | 0 | 0 | r5 | r0 | - | 73 | 3a | 13 | e7 | 00 | 00 |
| 08 | noop .., .., ..  | 73 | 13 | 21 | 1 | 0 | r4 | r5 | R | 73 | 3a | 13 | e7 | 00 | 00 |
| 09 | subi r5, 23, ..  | 00 | 00 | 87 | 0 | 0 | r0 | r4 | R | 73 | 3a | 13 | e7 | 00 | 21 |
| 0a | sbc_i r4, 86, .. | 23 | 21 | 87 | 0 | 0 | r5 | r0 | - | 73 | 3a | 13 | e7 | 87 | 21 |
| 0b | noop .., .., ..  | 86 | 87 | 02 | 0 | 0 | r4 | r5 | R | 73 | 3a | 13 | e7 | 87 | 21 |
| 0c | noop .., .., ..  | 00 | 00 | ff | 1 | 0 | r0 | r4 | R | 73 | 3a | 13 | e7 | 87 | 02 |
| 0d | noop .., .., ..  | 00 | 00 | ff | 1 | 0 | r0 | r0 | - | 73 | 3a | 13 | e7 | ff | 02 |



## Testbeispiel mit Lade- und Speicherbefehlen

```
dmem(0x05) = 0x48;
r1 = 0x06; dmem(r1) = 0x31;
r3 = dmem(0x05);
r4 = dmem(r1);
```

Programmiert für Pipeline-Verarbeitung:

```
0000: ld_i r0, 48, ..; r0 = 0x48
0001: stor r0, 05, ..; dmem(0x05) = r0
0002: ld_i r1, 06, ..; r1 = 0x06 (Adresse)
0003: noop .., .., ..
0004: ld_i r2, 31, ..; r2 = 0x31 (Daten)
0005: noop .., .., ..
0006: st_r r2, r1, ..; dmem(r1) = r2
0007: load r3, 05, ..; r3 = dmem(0x05)
0008: ld_r r4, r1, ..; r4 = dmem(r1)
0009: noop .., .., ..
```



## Programm-Trace mit Lade- und Speicherbefehlen

| PC          | Befehlswort   | Cmd  | O1 | O2 | RR | c | z | RA | RB | W | r0                        | r1 | r2 | r3 | r4 |
|-------------|---------------|------|----|----|----|---|---|----|----|---|---------------------------|----|----|----|----|
| 01          | ld_i r0,48,.. | noop | 00 | 00 | 00 | 0 | 0 | r0 | r0 | - | 00                        | 00 | 00 | 00 | 00 |
| 02          | stor r0,05,.. | ld_i | 48 | 00 | 00 | 0 | 0 | r0 | r0 | - | 00                        | 00 | 00 | 00 | 00 |
| 03          | ld_i r1,06,.. | stor | 05 | 00 | 48 | 0 | 0 | r0 | r0 | R | 00                        | 00 | 00 | 00 | 00 |
| 04          | noop .....    | ld_i | 06 | 00 | 05 | 0 | 0 | r1 | r0 | S | 48                        | 00 | 00 | 00 | 00 |
| 05          | ld_i r2,31,.. | noop | 00 | 00 | 06 | 0 | 0 | r0 | r1 | R | 48                        | 00 | 00 | 00 | 00 |
| dmem[0:7] = |               |      |    |    |    |   |   |    |    |   | [00 00 00 00 00 48 00 00] |    |    |    |    |
| 06          | noop .....    | ld_i | 31 | 00 | 06 | 0 | 0 | r2 | r0 | - | 48                        | 06 | 00 | 00 | 00 |
| 07          | st_r r2,r1,.. | noop | 00 | 00 | 31 | 0 | 0 | r0 | r2 | R | 48                        | 06 | 00 | 00 | 00 |
| 08          | load r3,05,.. | st_r | 06 | 00 | 31 | 0 | 0 | r2 | r0 | - | 48                        | 06 | 31 | 00 | 00 |
| 09          | ld_r r4,r1,.. | load | 05 | 00 | 06 | 0 | 0 | r3 | r2 | S | 48                        | 06 | 31 | 00 | 00 |
| 0a          | noop .....    | ld_r | 06 | 00 | 05 | 0 | 0 | r4 | r3 | L | 48                        | 06 | 31 | 00 | 00 |
| dmem[0:7] = |               |      |    |    |    |   |   |    |    |   | [00 00 00 00 00 48 31 00] |    |    |    |    |
| 0b          | noop .....    | noop | 00 | 00 | 06 | 0 | 0 | r0 | r4 | L | 48                        | 06 | 31 | 48 | 00 |
| 0c          | noop .....    | noop | 00 | 00 | 06 | 0 | 0 | r0 | r0 | - | 48                        | 06 | 31 | 48 | 31 |



## Beispielprogramm mit Schleife

```
r0 = 0x01; r1 = 0x34;
M: dmem(r0) = r1;
r1 = r1 - r0;
r0 = r0 + 0x01;
wenn r0 <= 0x3 springe zu M; (3 Schleifen-
durchläufe)
```

### Programmiert für Pipeline-Verarbeitung:

```
0x00: ld_i r0,01,.. ; r0 = 0x01
0x01: ld_i r1,34,.. ; r1 = 0x34
0x02: noop ..,..,.. ; warte, bis r0 geladen ist
0x03: comp r0,03,.. ; vergleiche r0 mit 0x03
0x04: st_r r1,r0,.. ; dmem(r0) = r1;
0x05: jump 02,1th.. ; wenn r0 < 0x3 springe zu 0x2
0x06: addi r0,01,.. ; r0 = r0+1 (Delayslot 1)
0x07: subr r1,r1,r0 ; r1 = r1-r0 (Delayslot 2)
0x08: noop ..,..,.. ; 1. Anw. nach der Schleife
```



## Programm-Trace mit Schleife

```

PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3 r4
01|ld_i r0,01,..|noop|00|00|00|0|0|r0|r0|-|00 00 00 00 00
1. Schleifendurchlauf:
02|ld_i r1,34,..|ld_i|01|00|00|0|0|r0|r0|-|00 00 00 00 00
03|noop,..|ld_i|34|00|01|0|0|r1|r0|R|00 00 00 00 00
04|comp r0,03,..|noop|00|00|34|0|0|r0|r1|R|01 00 00 00 00
05|st_r r1,r0,..|comp|03|01|34|0|0|r0|r0|-|01 34 00 00 00
06|jump 02,lth..|st_r|01|00|02|0|0|r1|r0|-|01 34 00 00 00
07|addi r0,01,..|jump|02|07|01|0|0|r7|r1|S|01 34 00 00 00
2. Schleifendurchlauf:
02|subr r1,r1,r0|addi|01|01|01|0|0|r0|r7|-|01 34 00 00 00
Abschluss 1. Schreibop.: dmem[0:7]=[00 34 00 00 00 00 00]
03|noop,..|subr|34|01|02|0|0|r1|r0|R|01 34 00 00 00
04|comp r0,03,..|noop|00|00|33|0|0|r0|r1|R|02 34 00 00 00
05|st_r r1,r0,..|comp|03|02|33|0|0|r0|r0|-|02 33 00 00 00

```



## Fortsetzung

```

PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3 r4
05| Fortsetzung nach Laden des Spungbefehts
06| jump 02, lth...|st_r|02|00|01|0|0|r1|r0|-|02 33 00 00 00
07| addi r0, 01, ...|jump|02|07|02|0|0|r7|r1|S|02 33 00 00 00

3. Schleifendurchlauf:
02| subr r1, r1, r0|addi|01|02|02|0|0|r0|r7|-|02 33 00 00 00
Abschluss 2. Schreibop.: dmem[0:7]=[00 34 33 00 00 00 00]
03| noop ...|subr|33|02|03|0|0|r1|r0|R|02 33 00 00 00
04| comp r0, 03, ...|noop|00|00|31|0|0|r0|r1|R|03 33 00 00 00
05| st_r r1, r0, ...|comp|03|03|31|0|0|r0|r0|-|03 31 00 00 00
06| jump 02, lth...|st_r|03|00|00|0|1|r1|r0|-|03 31 00 00 00
07| addi r0, 01, ...|jump|02|07|03|0|1|r7|r1|S|03 31 00 00 00
08| subr r1, r1, r0|addi|01|03|03|0|1|r0|r7|-|03 31 00 00 00
Abschluss 3. Schreibop.: dmem[0:7]=[00 34 33 31 00 00 00]
09| noop ...|subr|31|03|04|0|0|r1|r0|R|03 31 00 00 00
0a| noop ...|noop|00|00|2e|0|0|r0|r1|R|04 31 00 00 00

```



## Unterprogrammaufruf

Das nachfolgende Unterprogramm bekommt in `dmem(1)` einen Wert und in `r1` eine Adresse übergeben und schreibt den übergebenen Wert + `0x13` in den Datenspeicher auf die Übergabeadresse:

```
0000: ld_i r0, 35, .. Unterprogramm:
0001: stor r0, 01, .. 0010: load r3, 01, ..
0002: ld_i r1, 02, .. 0013: addi r3, 13, ..
0003: call r5, 10, .. 0014: st_r r3, r1, ..
0006: ld_i r0, 46, .. 0015: retu r5, .., ..
0007: stor r0, 01, ..
0008: ld_i r1, 04, .. (Restliche Befehle noop)
0009: call r5, 10, ..
000c: jump 0c, alw.. ; Halt in Endlosschleife
```

### Testbeispiele:

- Aufruf mit `dmem(1)=0x35` und `r1=2`, Ergebnis `dmem(2)=0x48`
- Aufruf mit `dmem(1)=0x46` und `r1=4`, Ergebnis `dmem(4)=0x59`



## Programm-Trace mit einer Schleife

```

PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
01|ld_i r0,35,..|noop|00|00|00|r0|r0|-|00 00 00 00 00 00
02|stor r0,01,..|ld_i|35|00|00|r0|r0|-|00 00 00 00 00 00
03|ld_i r1,02,..|stor|01|00|35|r0|r0|R|00 00 00 00 00 00

Unterprogrammaufruf mit dmem(1)=0x35 und r1=2
04|call r5,10,..|ld_i|02|00|01|r1|r0|S|35 00 00 00 00 00
05|noop,..|call|10|04|02|r5|r1|R|35 00 00 00 00 00
Abschluss Schreibop. Aufruf 1: dmem = [00 35 00 00 00 ..]
10|noop,..|noop|00|00|04|r0|r5|R|35 02 00 00 00 00
11|load r3,01,..|noop|00|00|04|r0|r0|-|35 02 00 00 00 04
12|noop,..|load|01|00|04|r3|r0|-|35 02 00 00 00 04
13|noop,..|noop|00|00|01|r0|r3|L|35 02 00 00 00 04
14|addi r3,13,..|noop|00|00|01|r0|r0|-|35 02 00 35 00 04
15|st_r r3,r1,..|addi|13|35|01|r3|r0|-|35 02 00 35 00 04
16|retu r5,..,..|st_r|02|00|48|r3|r3|R|35 02 00 35 00 04
...
1 Zeitslot bis Rücksprung und 2 bis Speicherop. fertig

```



```
PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
17|noop .,.,.,.,.|retu|04|00|02|r5|r3|S|35 02 00 48 00 04
04|noop .,.,.,.,.|noop|00|00|02|r0|r5|-|35 02 00 48 00 04
```

Abschluss Schreibop. UP1: dmem = [00 35 48 00 00 00 ..]

```
05|noop .,.,.,.,.|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
06|noop .,.,.,.,.|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
07|ld_i r0,46,..|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
08|stor r0,01,..|ld_i|46|00|02|r0|r0|-|35 02 00 48 00 04
09|ld_i r1,04,..|stor|01|00|46|r0|r0|R|35 02 00 48 00 04
```

Unterprogrammaufruf mit dmem(1)=0x46 und r1=4

```
0a|call r5,10,..|ld_i|04|00|01|r1|r0|S|46 02 00 48 00 04
0b|noop .,.,.,.,.|call|10|0a|04|r5|r1|R|46 02 00 48 00 04
```

Abschluss Schreibop. Aufruf 2: dmem = [00 46 48 00 00 ..]

```
10|noop .,.,.,.,.|noop|00|00|0a|r0|r5|R|46 04 00 48 00 04
11|load r3,01,..|noop|00|00|0a|r0|r0|-|46 04 00 48 00 0a
12|noop .,.,.,.,.|load|01|00|0a|r3|r0|-|46 04 00 48 00 0a
13|noop .,.,.,.,.|noop|00|00|01|r0|r3|L|46 04 00 48 00 0a
14|addi r3,13,..|noop|00|00|01|r0|r0|-|46 04 00 46 00 0a
15|st_r r3,r1,..|addi|13|46|01|r3|r0|-|46 04 00 46 00 0a
16|retu r5,..,..|st_r|04|00|59|r3|r3|R|46 04 00 46 00 0a
```



```

PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
... 1 Zeitslot bis Rückspr. und 2 bis Speicherop. fertig
17|noop|retu|0a|00|04|r5|r3|S|46 04 00 59 00 0a
0a|noop|noop|00|00|04|r0|r5|-|46 04 00 59 00 0a
Abschluss Schreibop. UP2: dmem = [00 46 48 00 59 00 ..]
0b|noop|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
0c|noop|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a

Endlosschleife mit 2 Delay-Slots
0d|jump 0c,alw...|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
0e|noop|jump|0c|01|04|r1|r0|-|46 04 00 59 00 0a
0c|noop|noop|00|00|04|r0|r1|-|46 04 00 59 00 0a
0d|jump 0c,alw...|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a

```



### Abschlussbemerkungen

Der Entwurf eines Prozessors mit Pipeline ist für Studierende keine unlösbare Aufgabe:

- Festlegung der Pipeline-Abläufe für alle Befehlsarten.
- Zusammenstellen aller Register-Transfer-Operationen.
- Beschreibung in VHDL.
- Synthese. FPGA-Programmieren. Test.

Die größeren Herausforderungen sind:

- Ein strukturierter Entwurfsablauf, damit der Entwurf, die Dokumentationen, ... fertig werden.
- Ein prüfgerechter Entwurf, um den Prozessor simulieren und testen zu können.
- Die Werkzeuge für die Programmierung (Assembler, Dissassembler, ...)