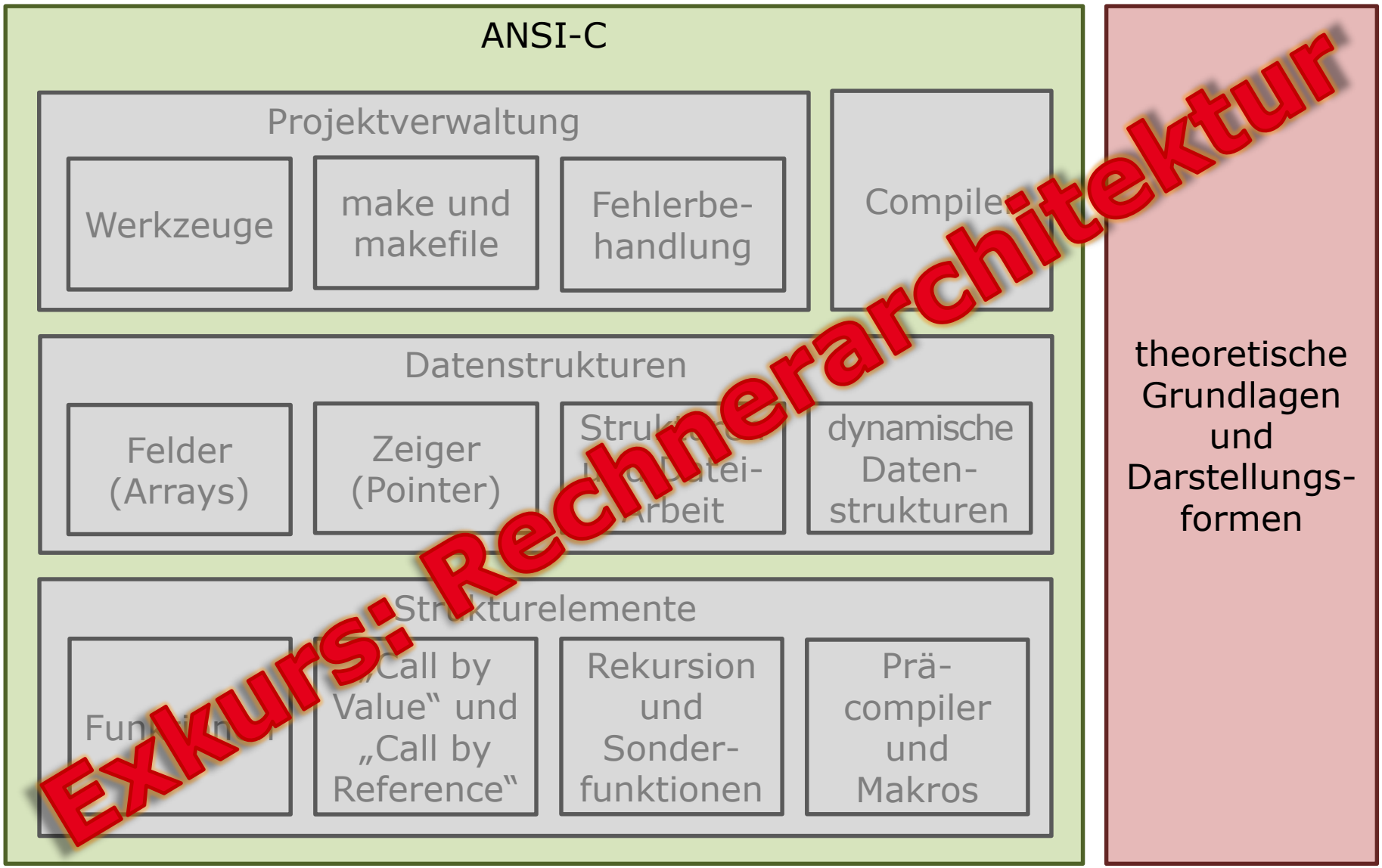
BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

Imperative Programmierung

Exkurs: Rechnerarchitektur

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

Aufbau der Lehrveranstaltung

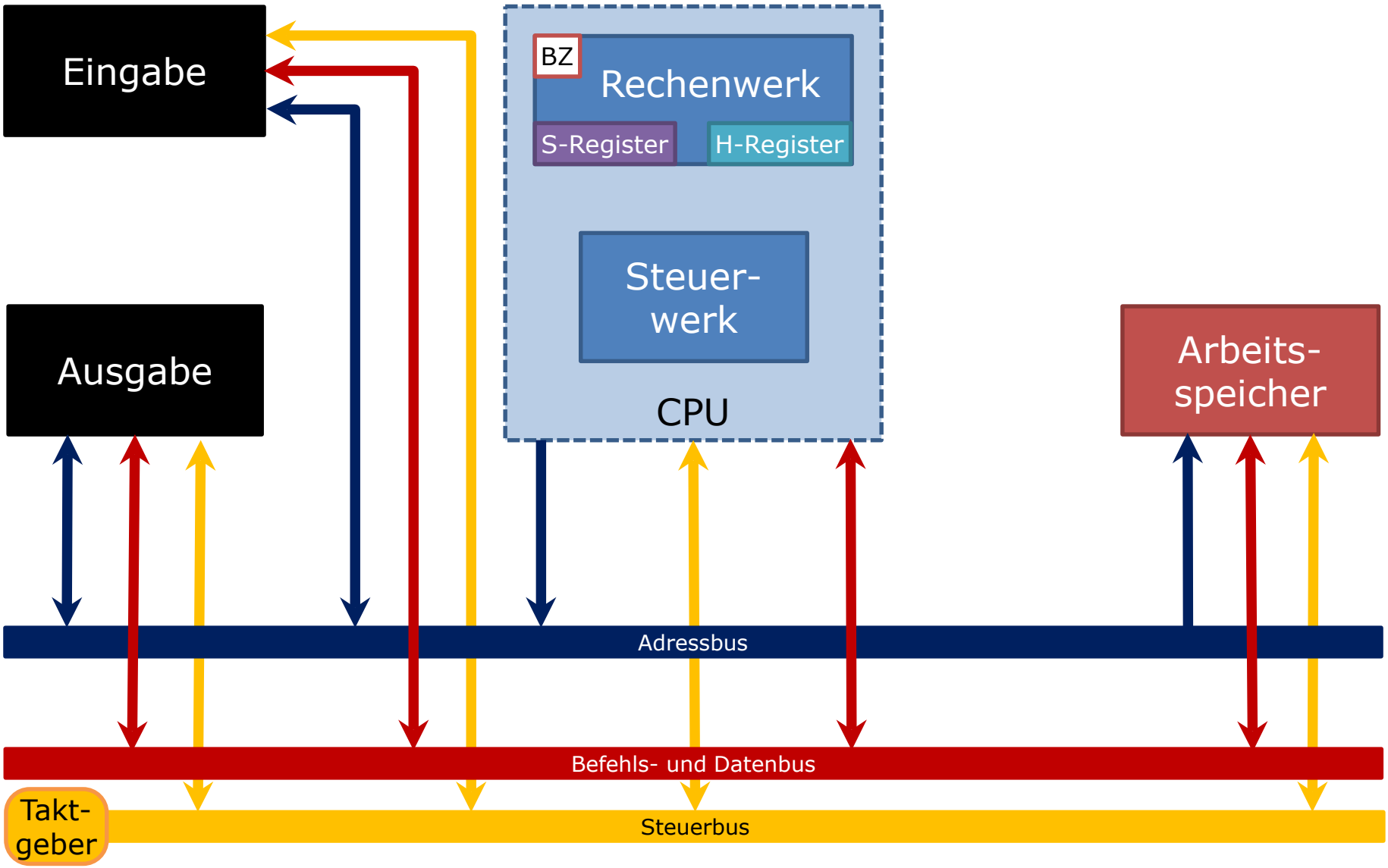


Hauptelemente eines Computers

- zentrale Verarbeitungseinheit (CPU)
  - Rechenwerk
    - Funktionseinheiten (Addierer, Multiplizierer, …)
    - arithmetisch-logische Einheit (ALU)
    - Hilfs- und Status-Register
  - Steuerwerk
- Buseinheit (inkl. Taktgenerator)
- Speicherwerk
  - Programmspeicher
  - Datenspeicher
- Eingabe-/Ausgabe-Werke

von-Neumann-Architektur

KLASSISCHE VON-NEUMANN-ARCHITEKTUR



Grundprinzipien des von-Neumann-Rechners (1/2)

Prinzipien des gespeicherten Programms:

- Befehle werden geladen, Steuersignale an Funktionseinheiten gesendet
- Befehle liegen in einem Arbeitsspeicher mit linearem Adressraum
- Befehlszähler zeigt auf den nächsten auszuführenden Befehl (manchmal auch als „Programmzähler“ bezeichnet)
- Befehle können wie Daten geändert werden

Grundprinzipien des von-Neumann-Rechners (2/2)

Prinzipien der sequentiellen Programm-Ausführung:

1. Befehlsabruf:
   nächster zu bearbeitende Befehl wird entsprechend der Adresse im Befehlszähler aus Arbeitsspeicher in Befehlsregister geladen; Befehlszähler wird um Länge des Befehls erhöht
2. Dekodierung:
   Steuerwerk löst Befehl in Schaltinstruktionen für Rechenwerk auf
3. Operandenabruf:
   Operanden (Werte oder Parameter) aus Arbeitsspeicher holen
4. Befehlsausführung:
   Rechenwerk führt arithmetische oder logische Operation aus; bei Sprungbefehlen wird Befehlszähler ggf. verändert
5. Rückschreiben:
   sofern notwendig, wird Ergebnis der Berechnung in Register oder Arbeitsspeicher zurückgeschrieben

Einfaches Beispiel: Das Programm

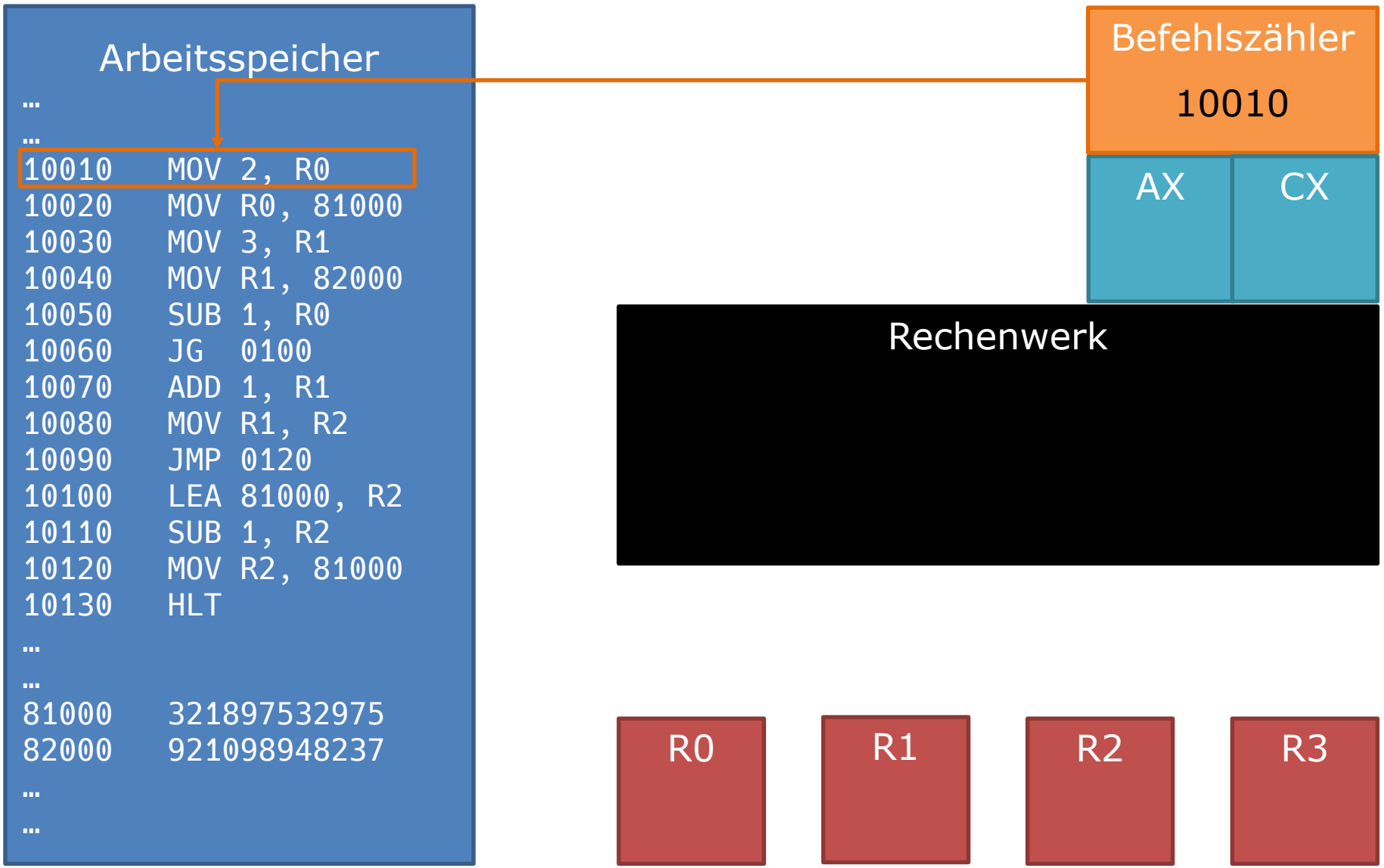
```
begin
   x := 2;
   y := 3;
   if (x > 1)
      x := y + 1;
   else
      x := x - 1;
end.
```

Einfaches Beispiel: erster Assembly-Kode

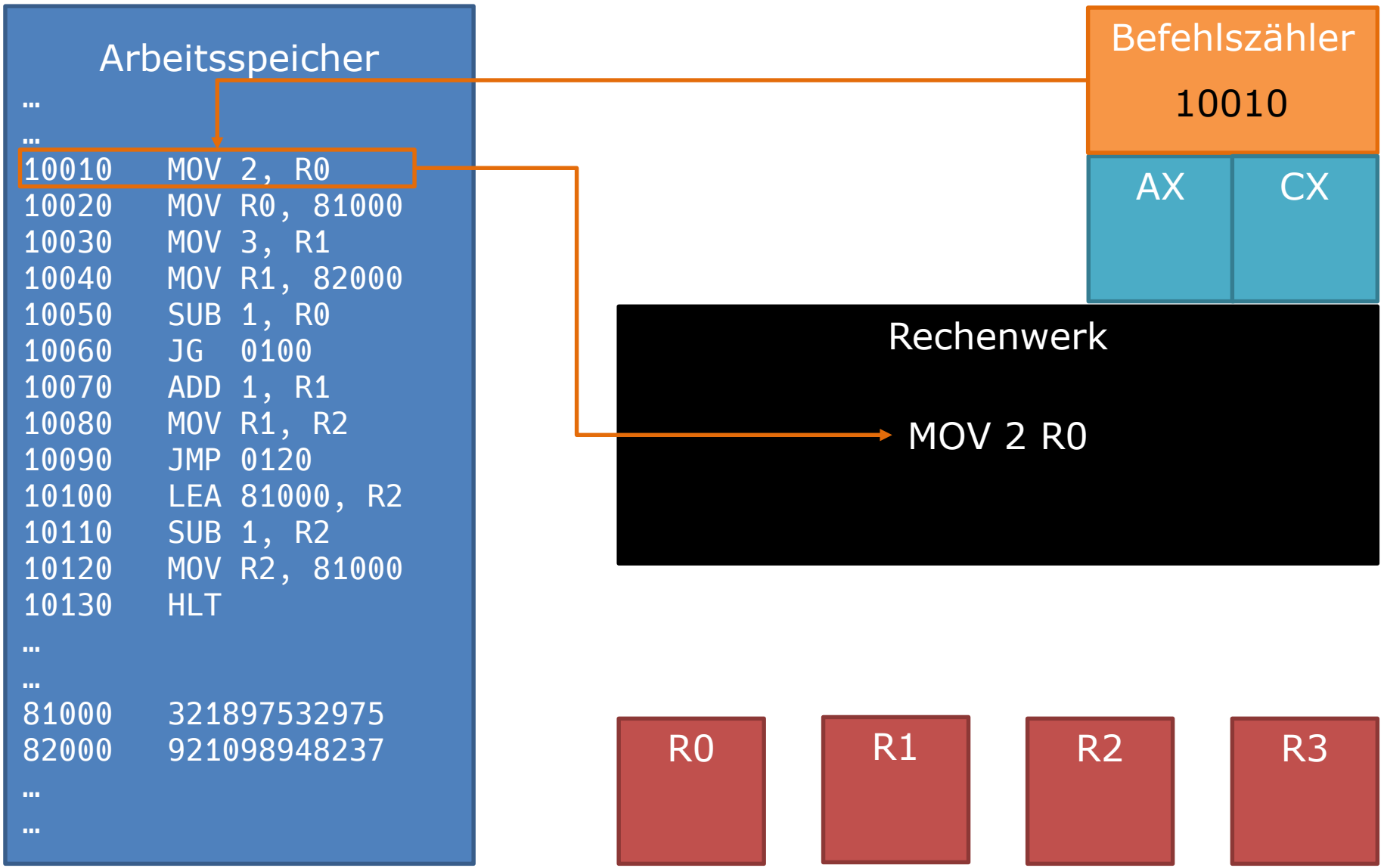
```
# Hinweise:
# * Der IBM DX8086 hat 4 Register (R0…R3, wobei R0 besonders ist).
# * Der Compiler weist den Variablen beim Erzeugen des Assembly-Kodes
#   Speicheradressen zu und legt diese in einer Variablentabelle ab;
#   für unser Beispiel seien dies die Adresse: x=81000 und y=82000

0010   MOV 2, R0
0020   MOV R0, 81000
0030   MOV 3, R0
0040   MOV R0, 82000
0050   LEA 81000, R0
0060   SUB 1, R0
0070   JG  0120
0080   LEA 82000, R0
0090   ADD 1, R0
0100   MOV R0, 81000
0110   JMP 0150
0120   LEA 81000, R0
0130   SUB 1, R0
0140   MOV R0, 81000
0150   HLT
```

```
# Optimierung des Assembly-Kodes
# unter Nutzung von 3 Registern:
0010    MOV 2, R0
0020    MOV R0, 81000
0030    MOV 3, R1
0040    MOV R1, 82000
0050    SUB 1, R0
0060    JG  0100
0070    ADD 1, R1
0080    MOV R1, R2
0090    JMP 0120
0100    LEA 81000, R2
0110    SUB 1, R2
0120    MOV R2, 81000
0130    HLT
```

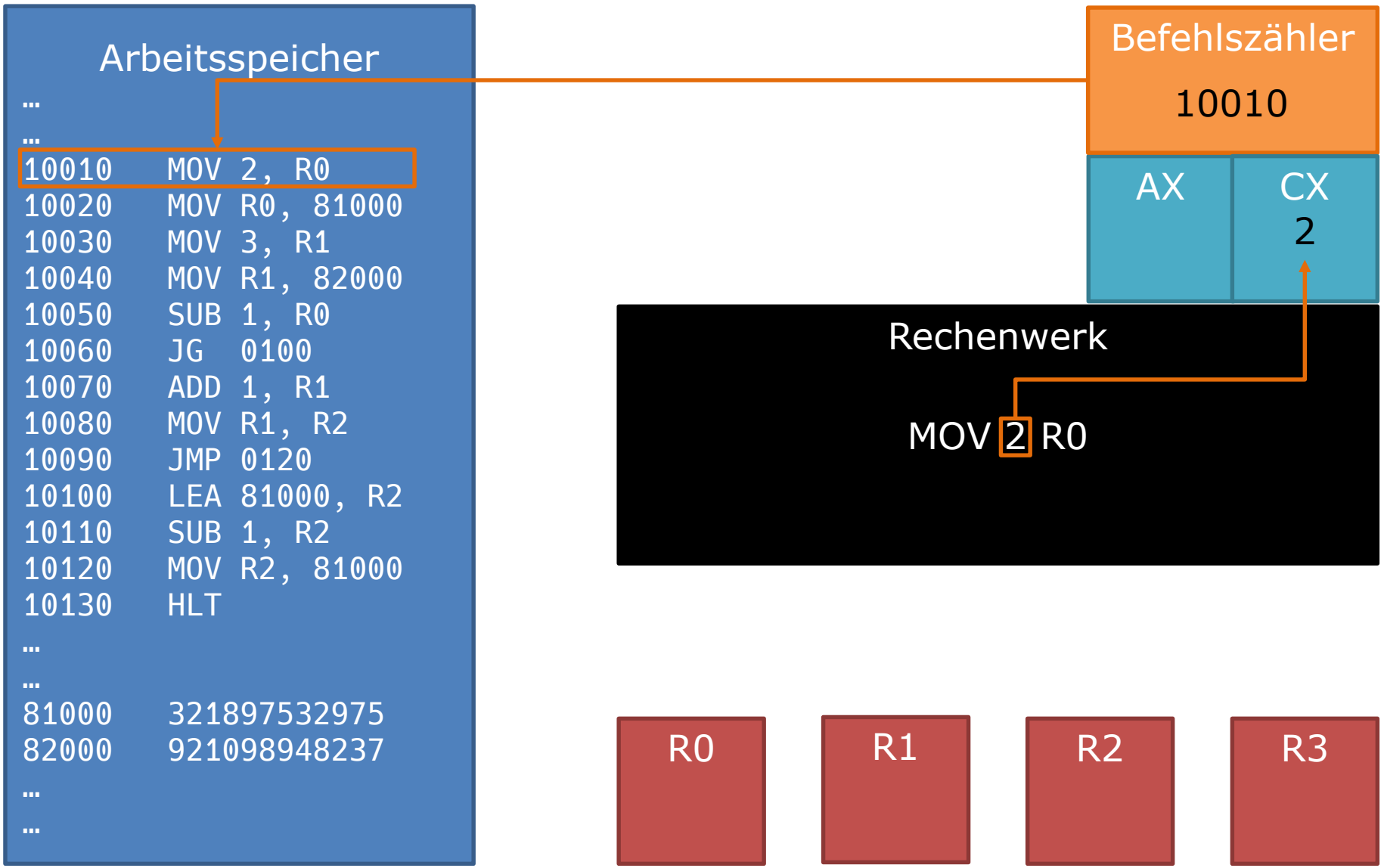
Einfaches Beispiel: nach der Initialisierung



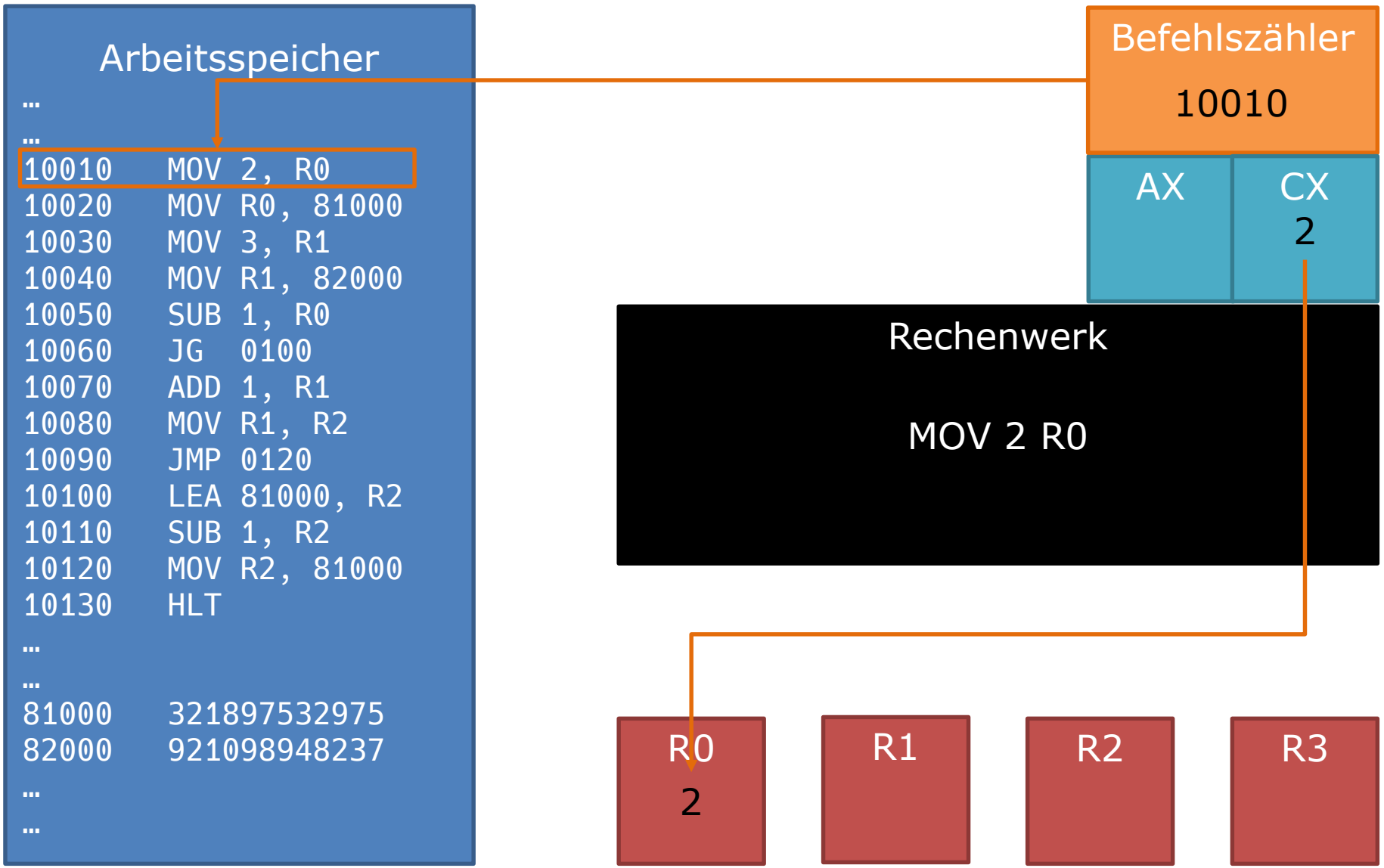
EINFACHES BEISPIEL: 1. TAKT



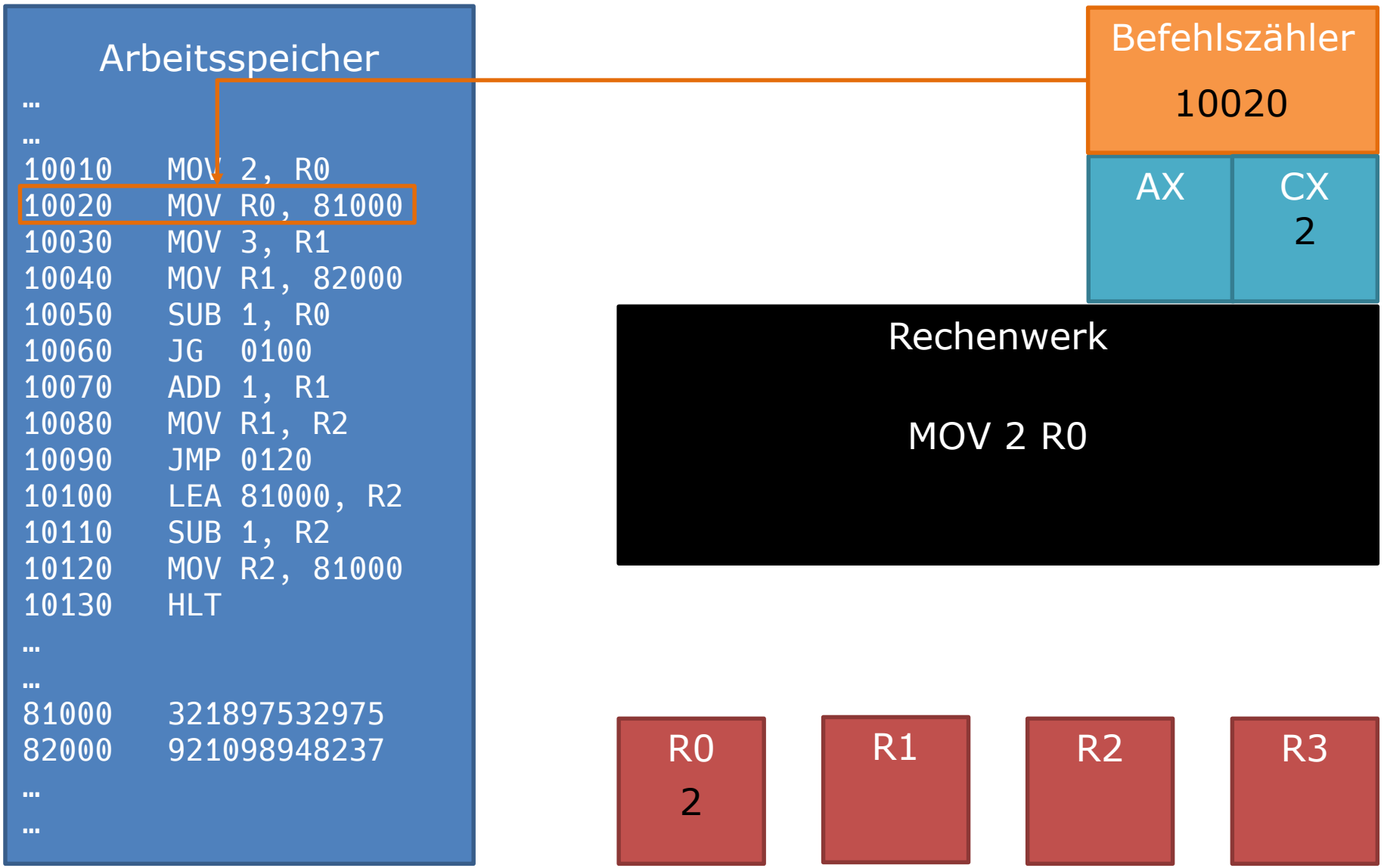
Einfaches Beispiel: 2. Takt



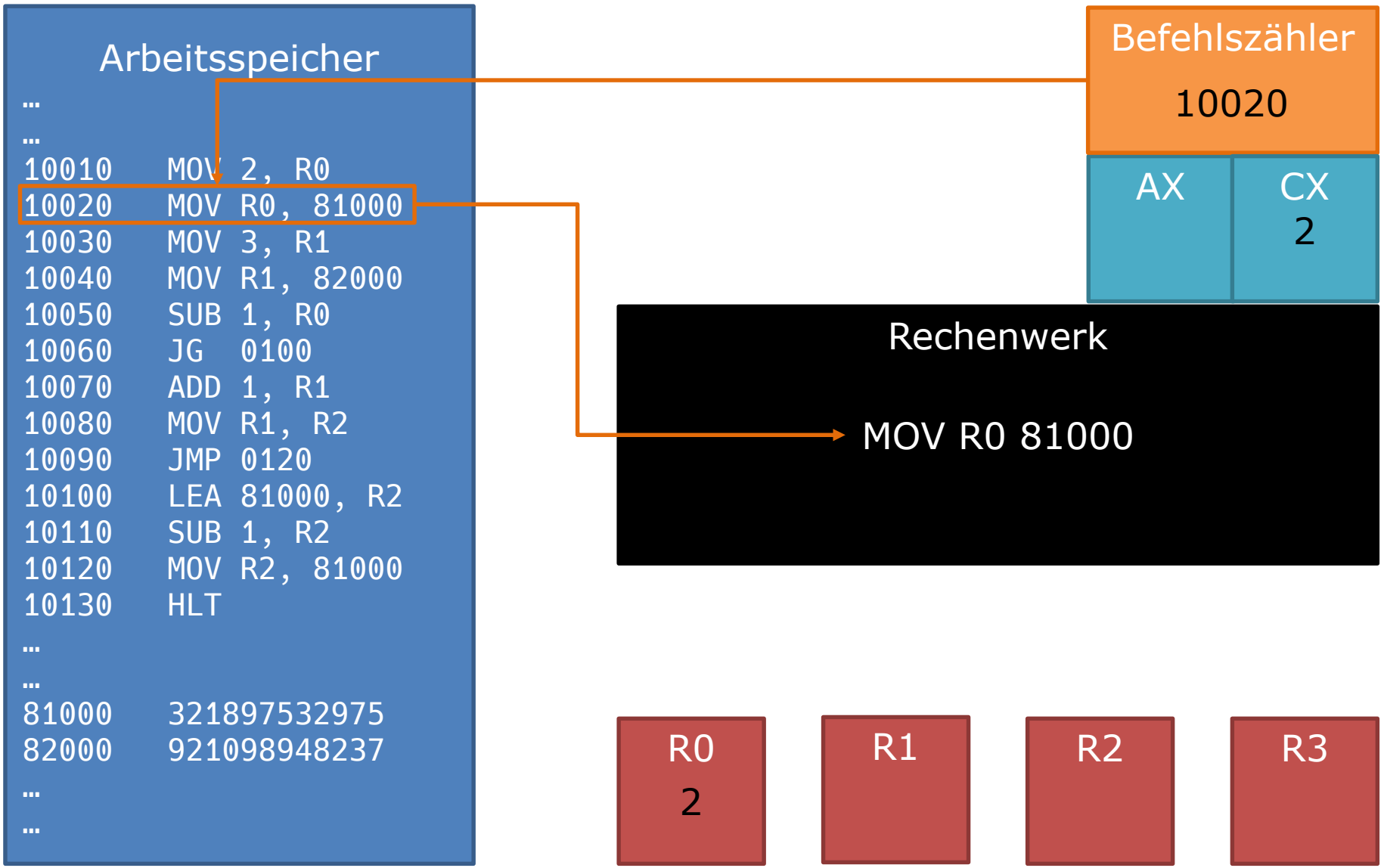
EINFACHES BEISPIEL: 3. TAKT



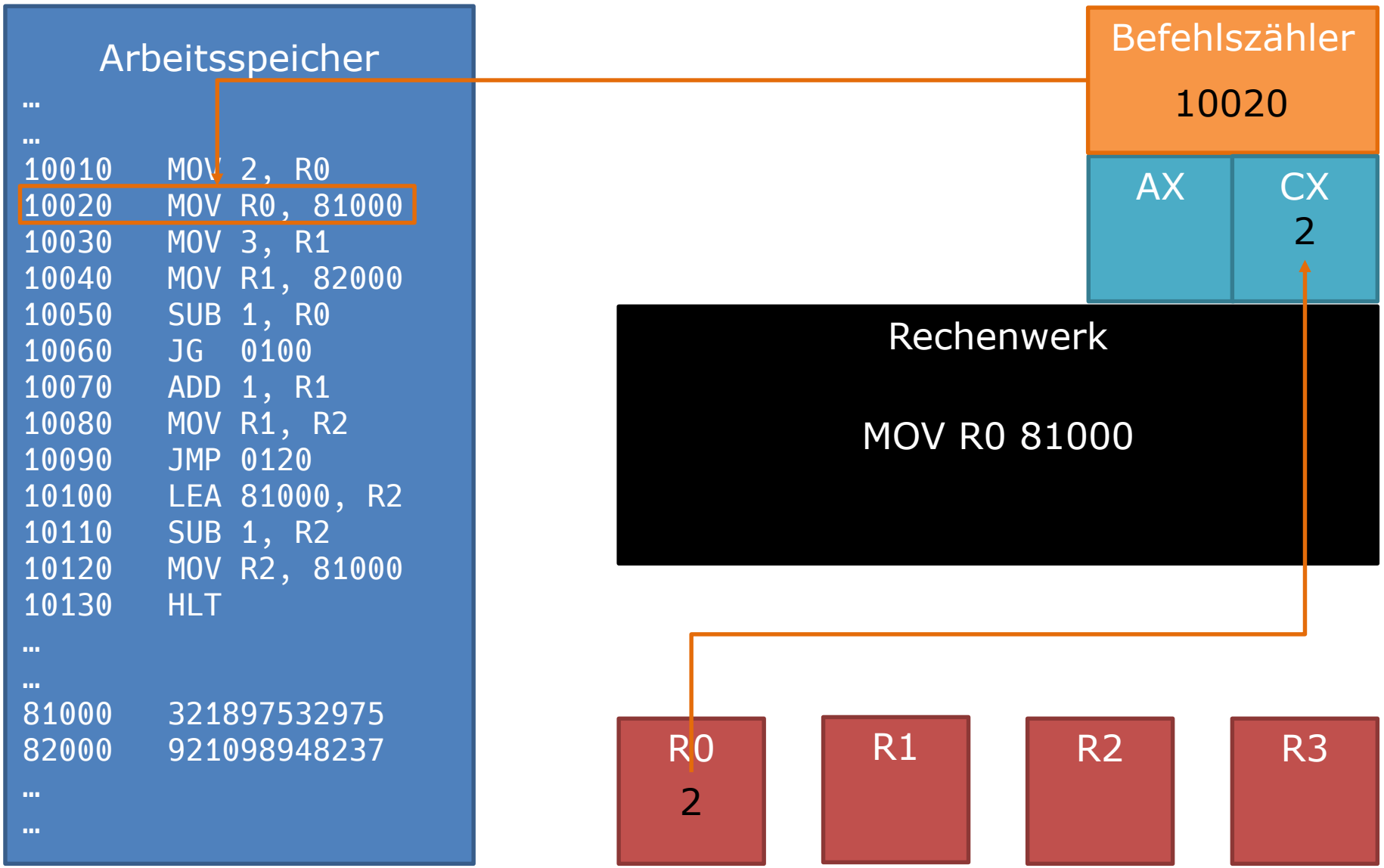
Einfaches Beispiel: 4. Takt



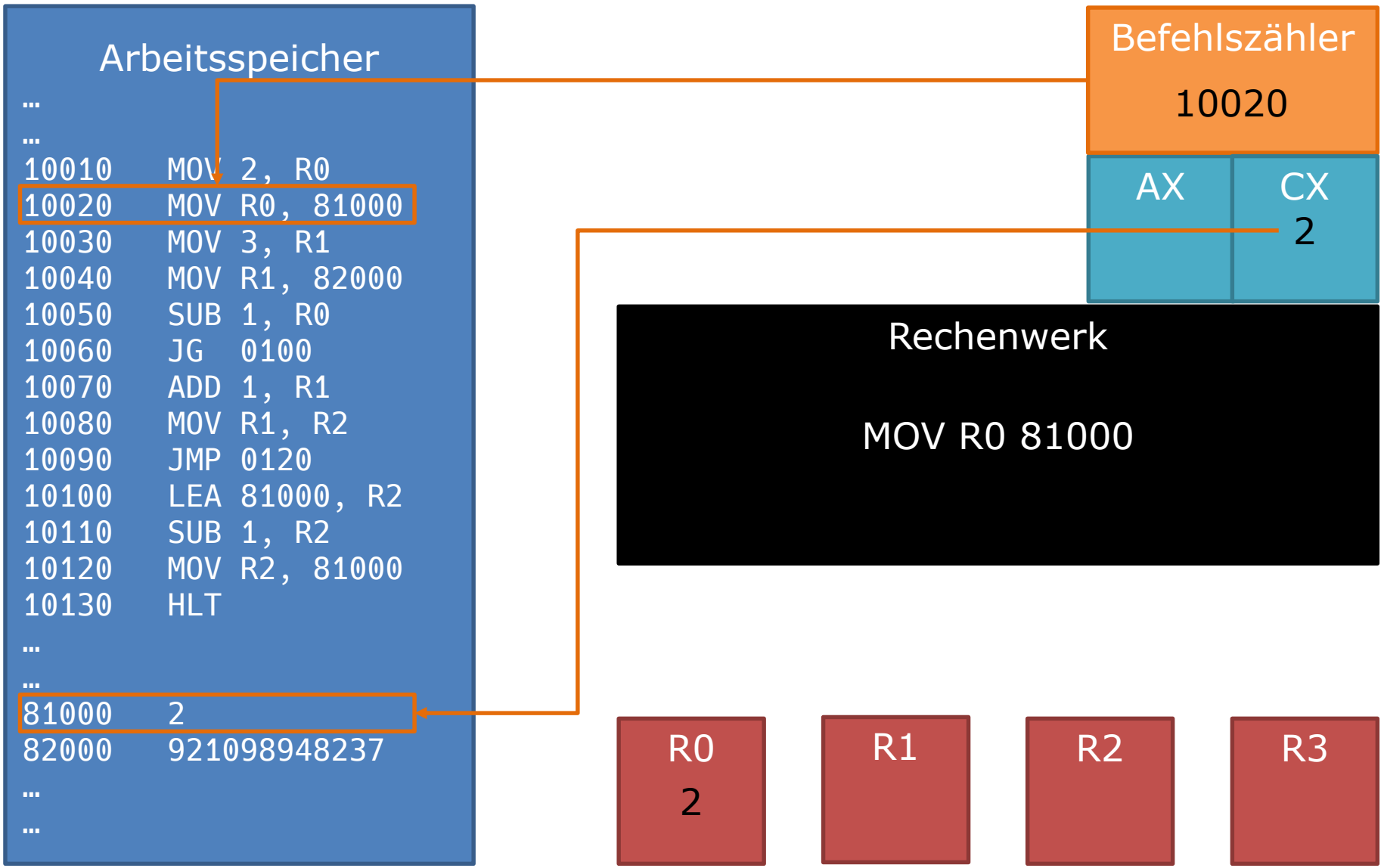
EINFACHES BEISPIEL: 5. TAKT



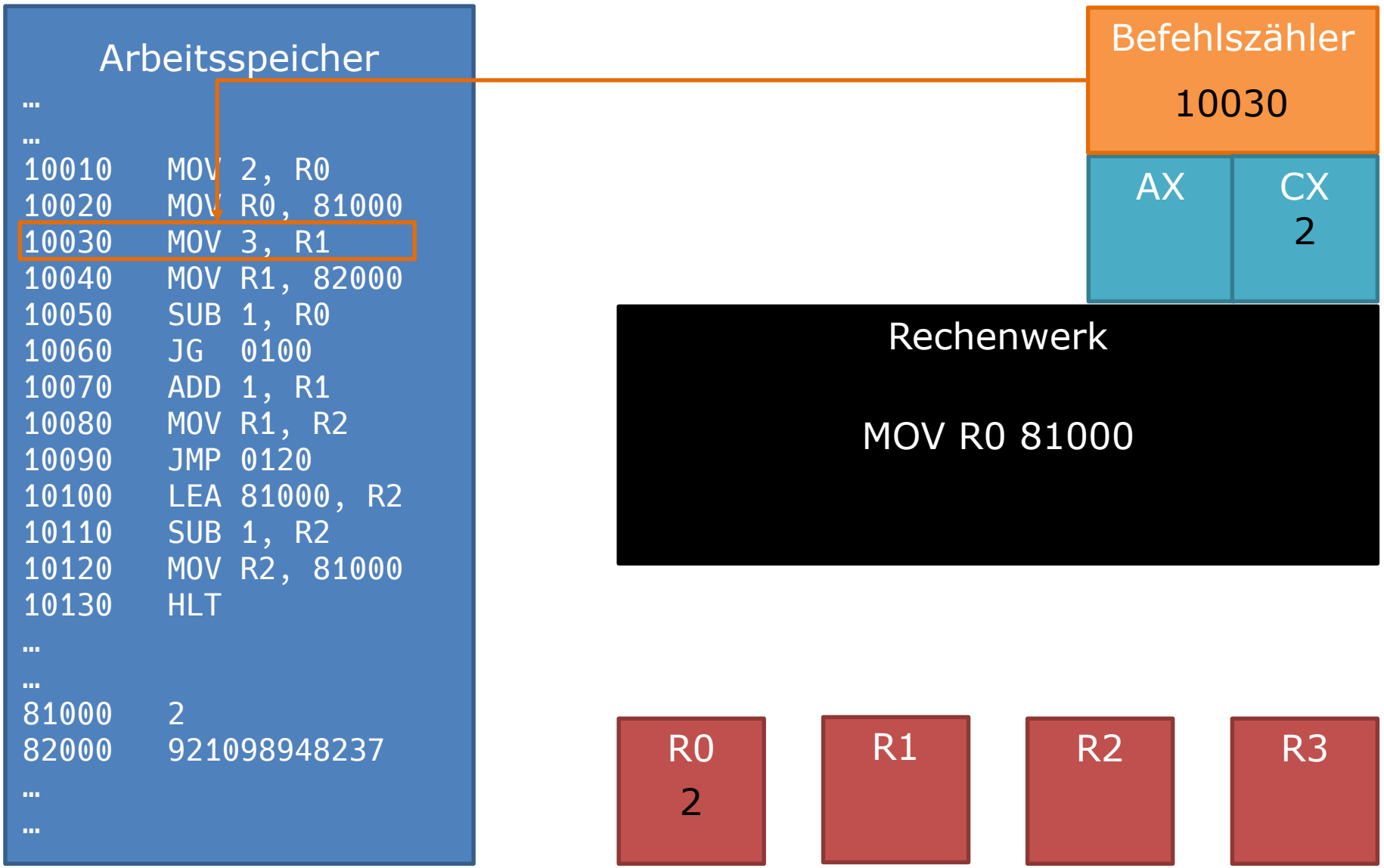
EINFACHES BEISPIEL: 6. TAKT



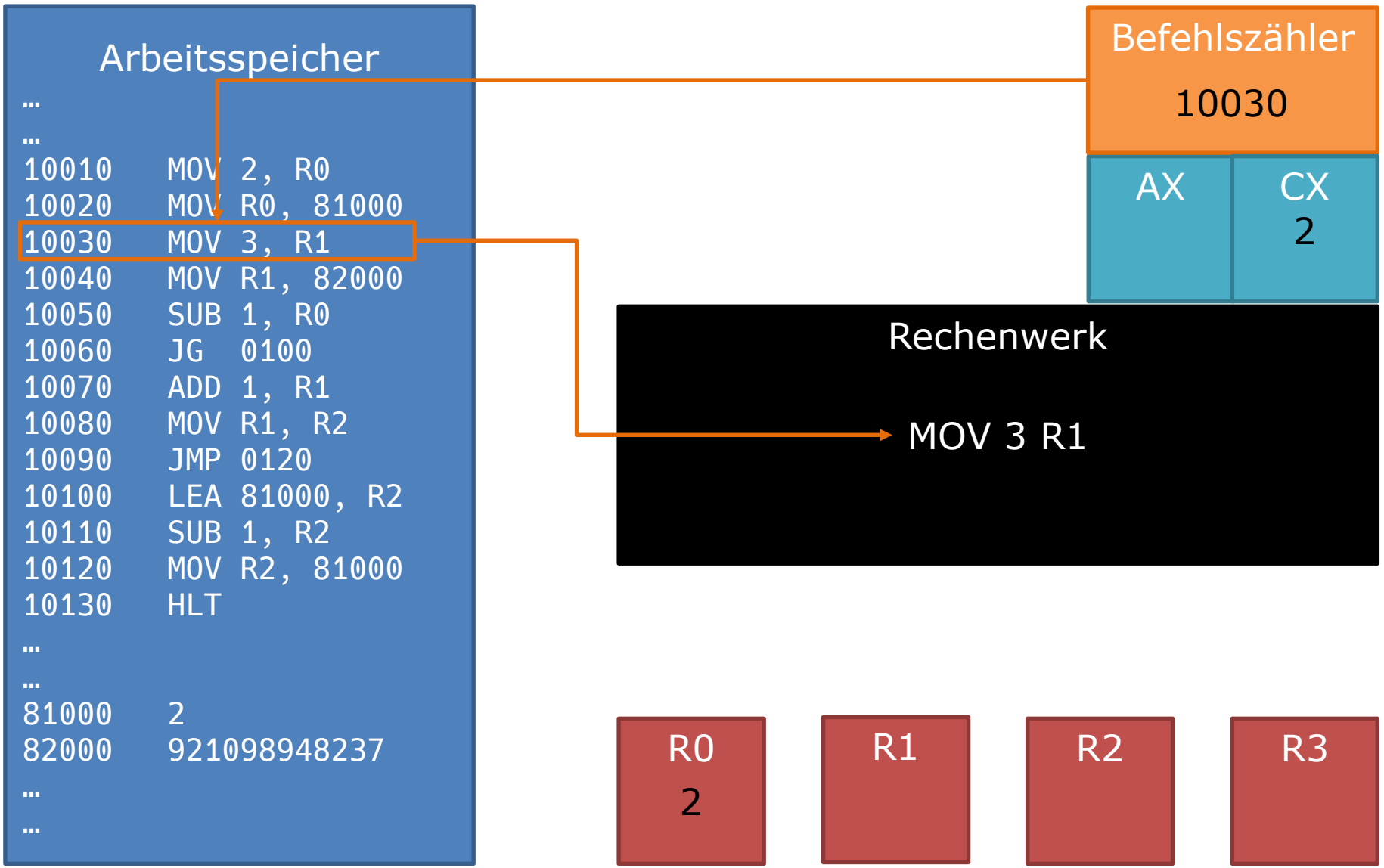
EINFACHES BEISPIEL: 7. TAKT



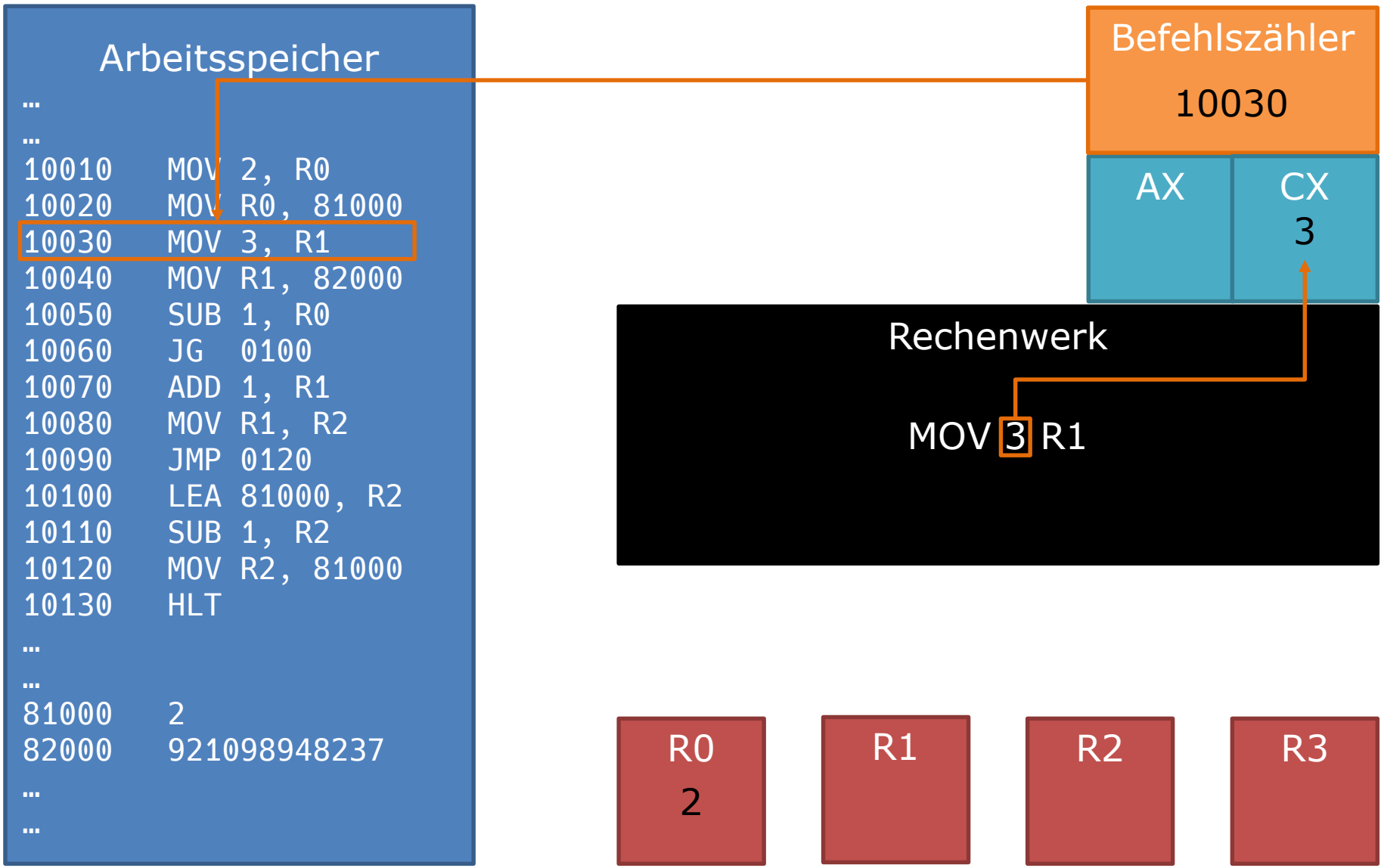
EINFACHES BEISPIEL: 8. TAKT



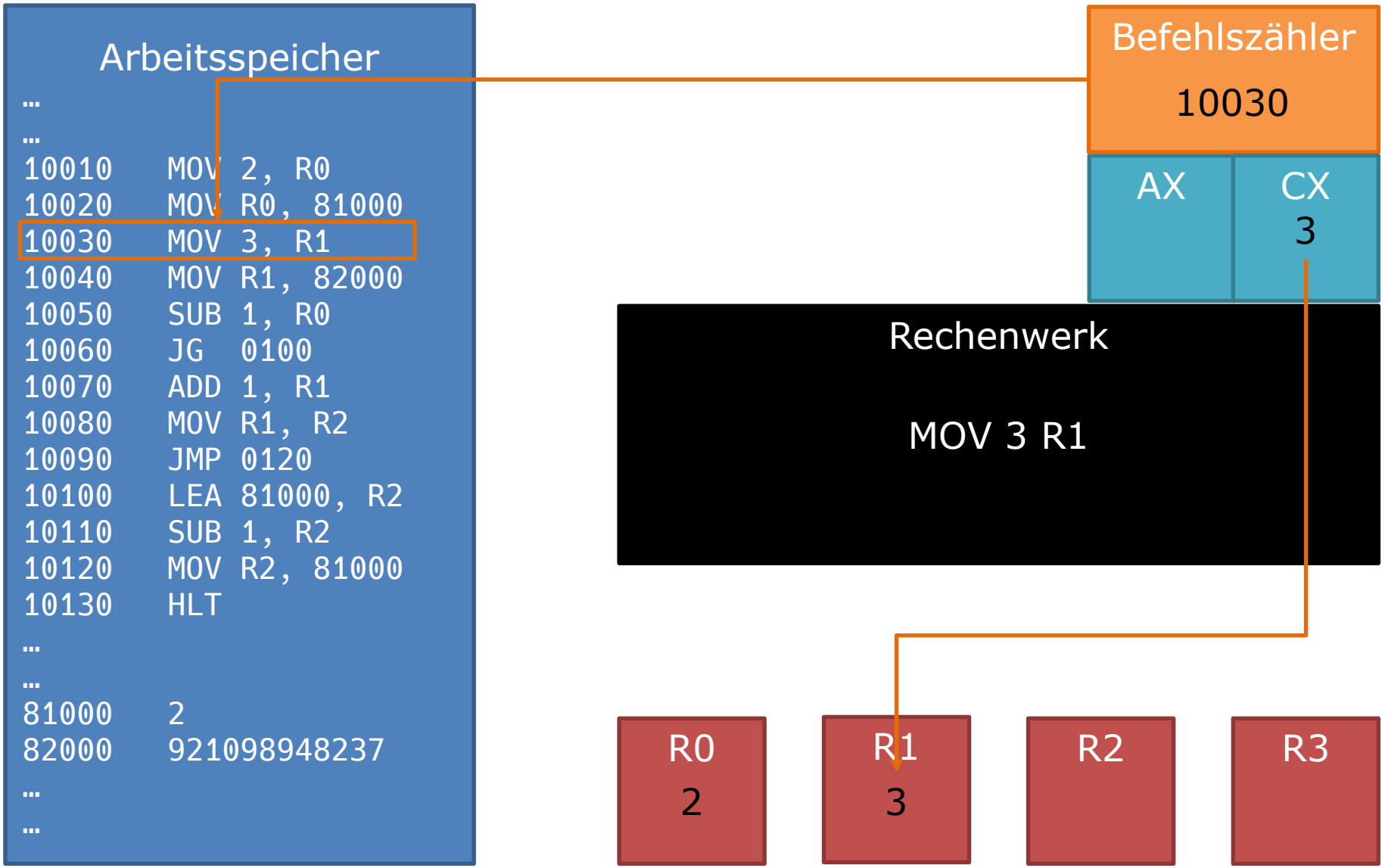
EINFACHES BEISPIEL: 9. TAKT



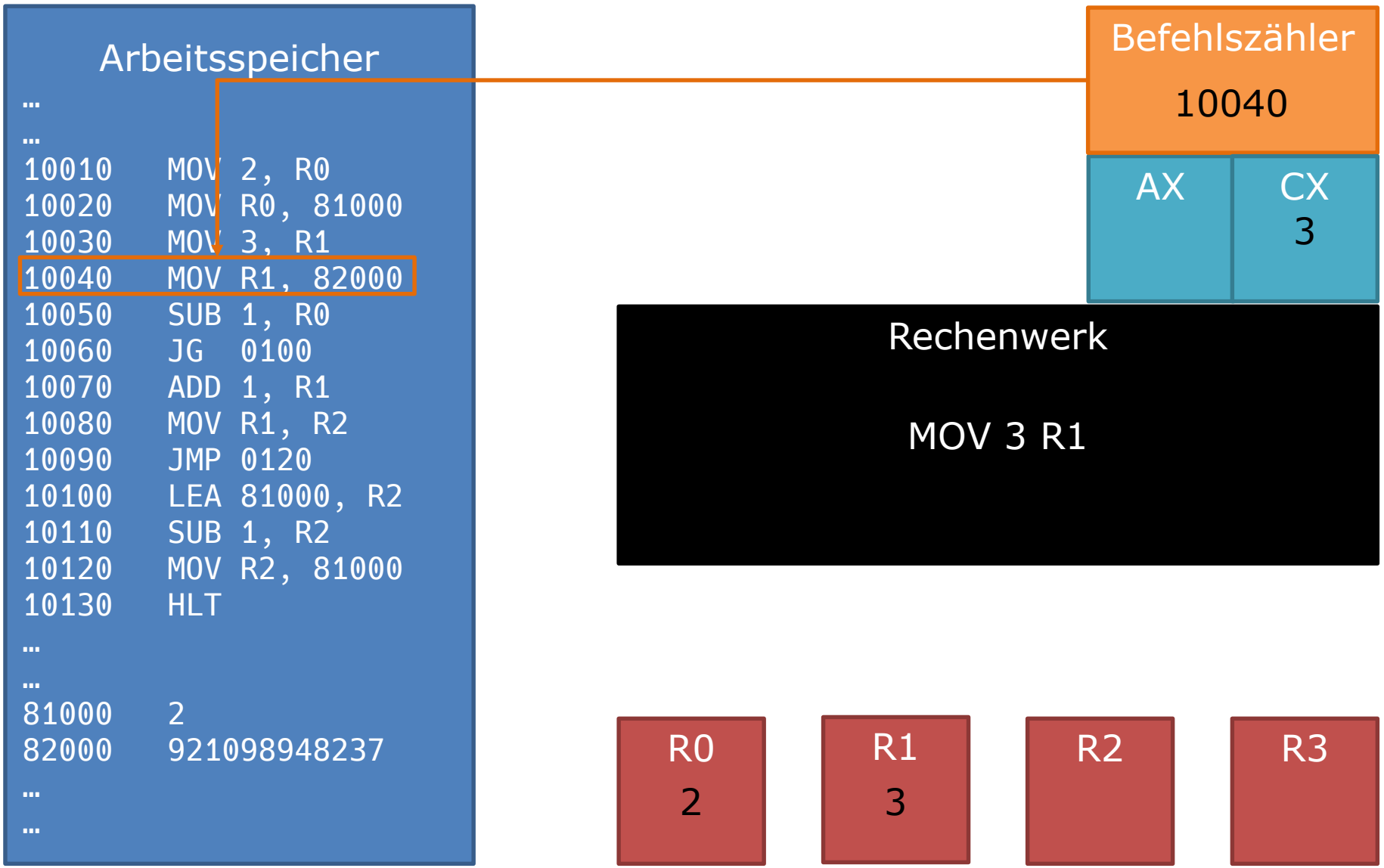
EINFACHES BEISPIEL: 10. TAKT



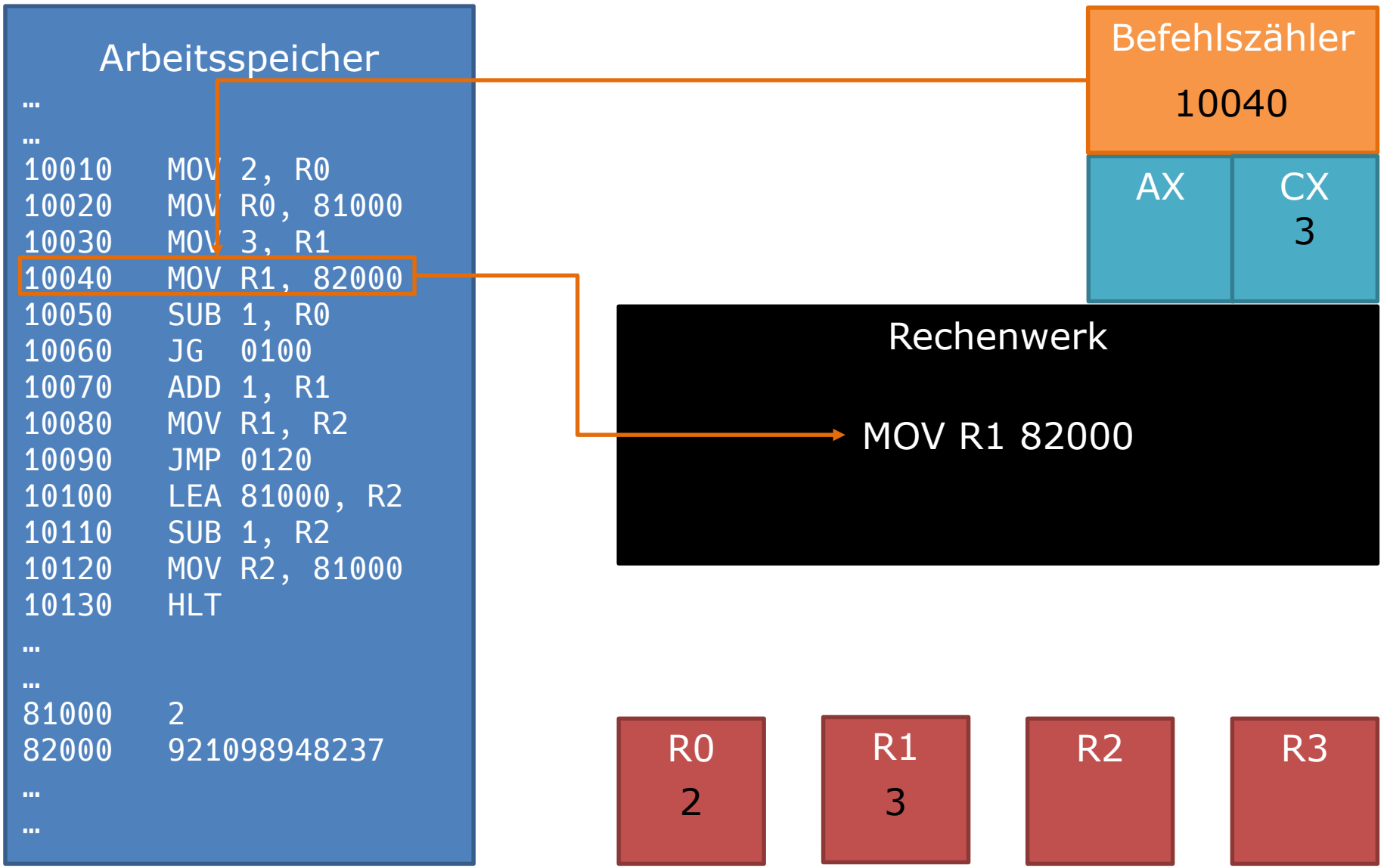
Einfaches Beispiel: 11. Takt



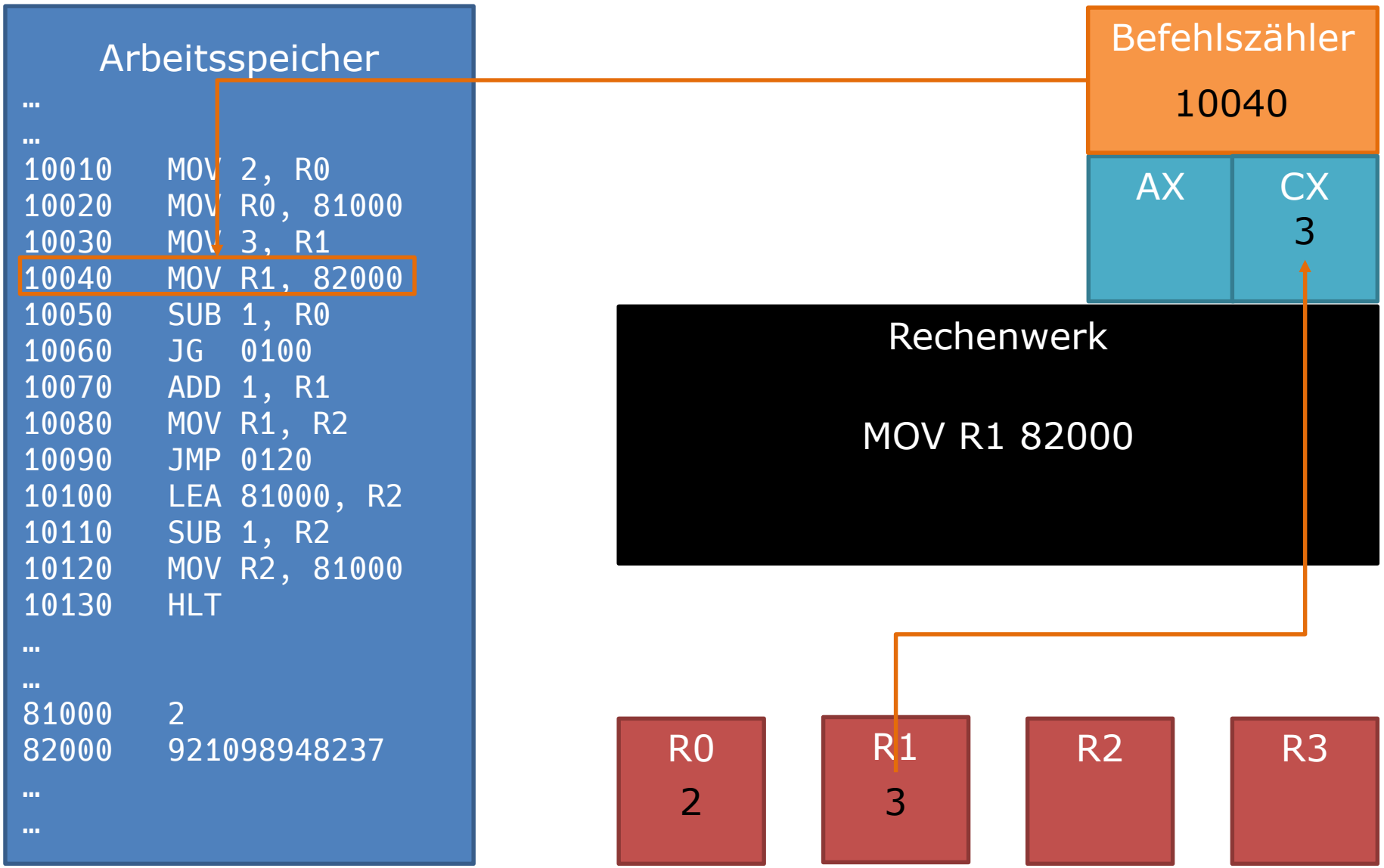
EINFACHES BEISPIEL: 12. TAKT



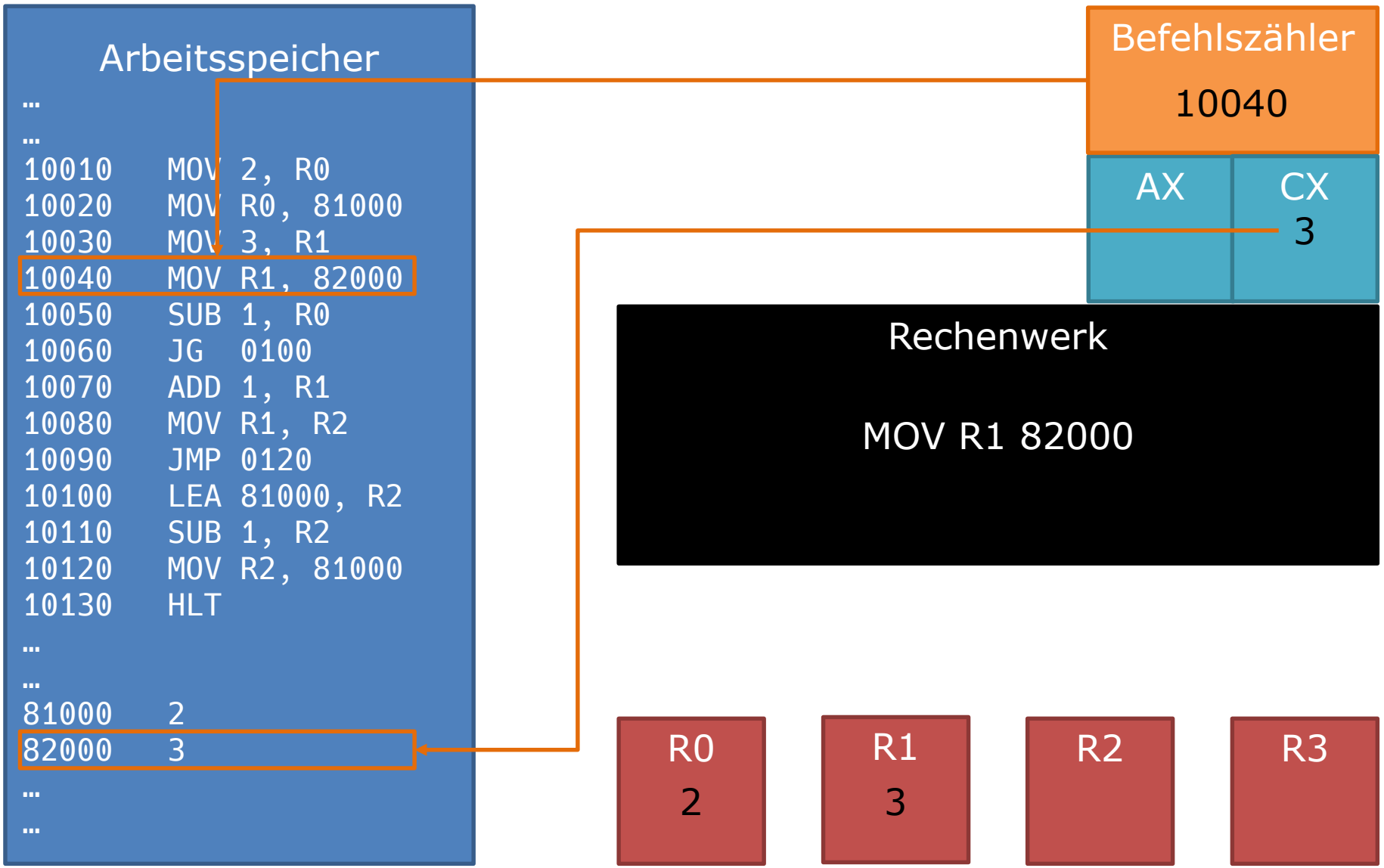
EINFACHES BEISPIEL: 13. TAKT



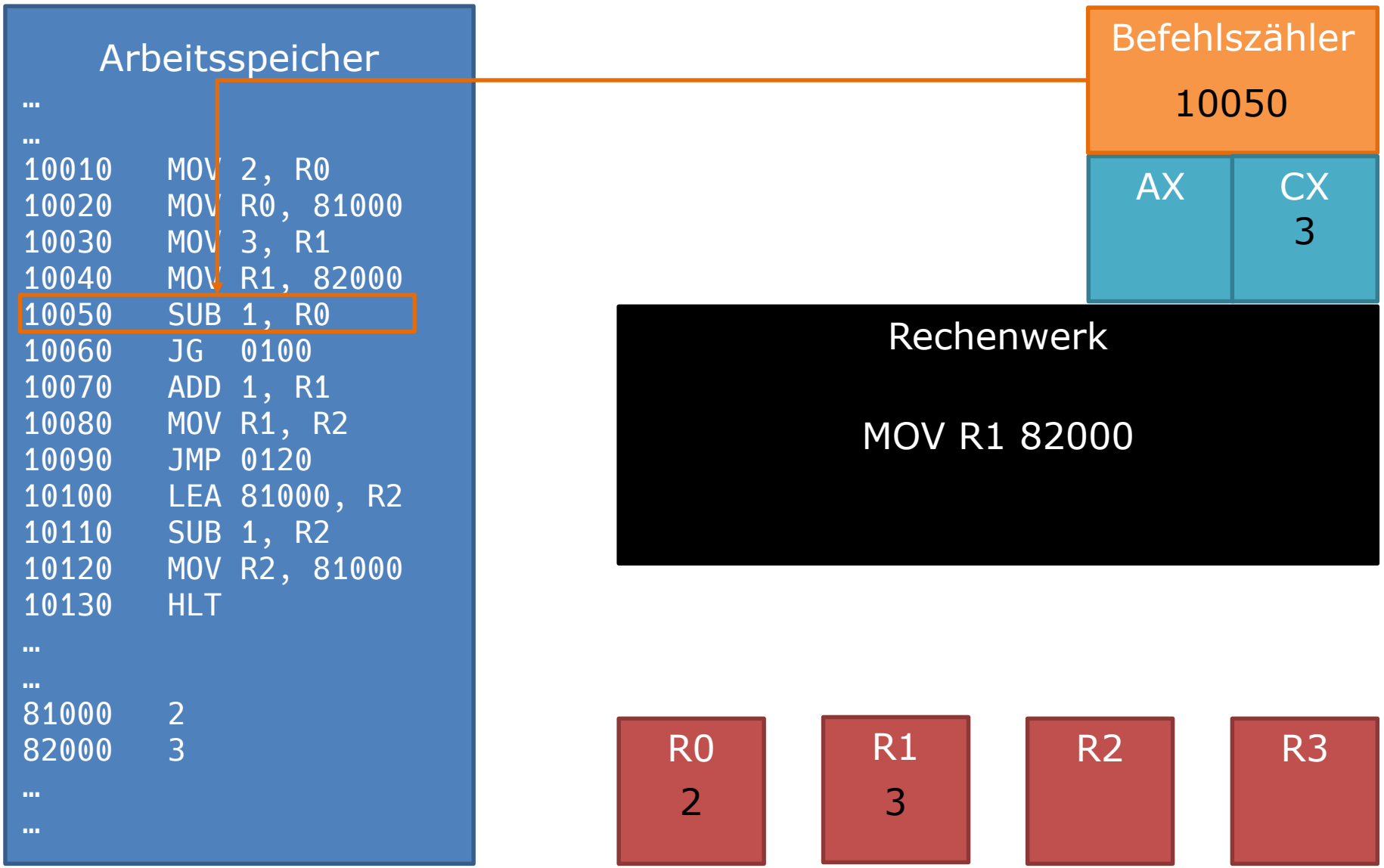
EINFACHES BEISPIEL: 14. TAKT



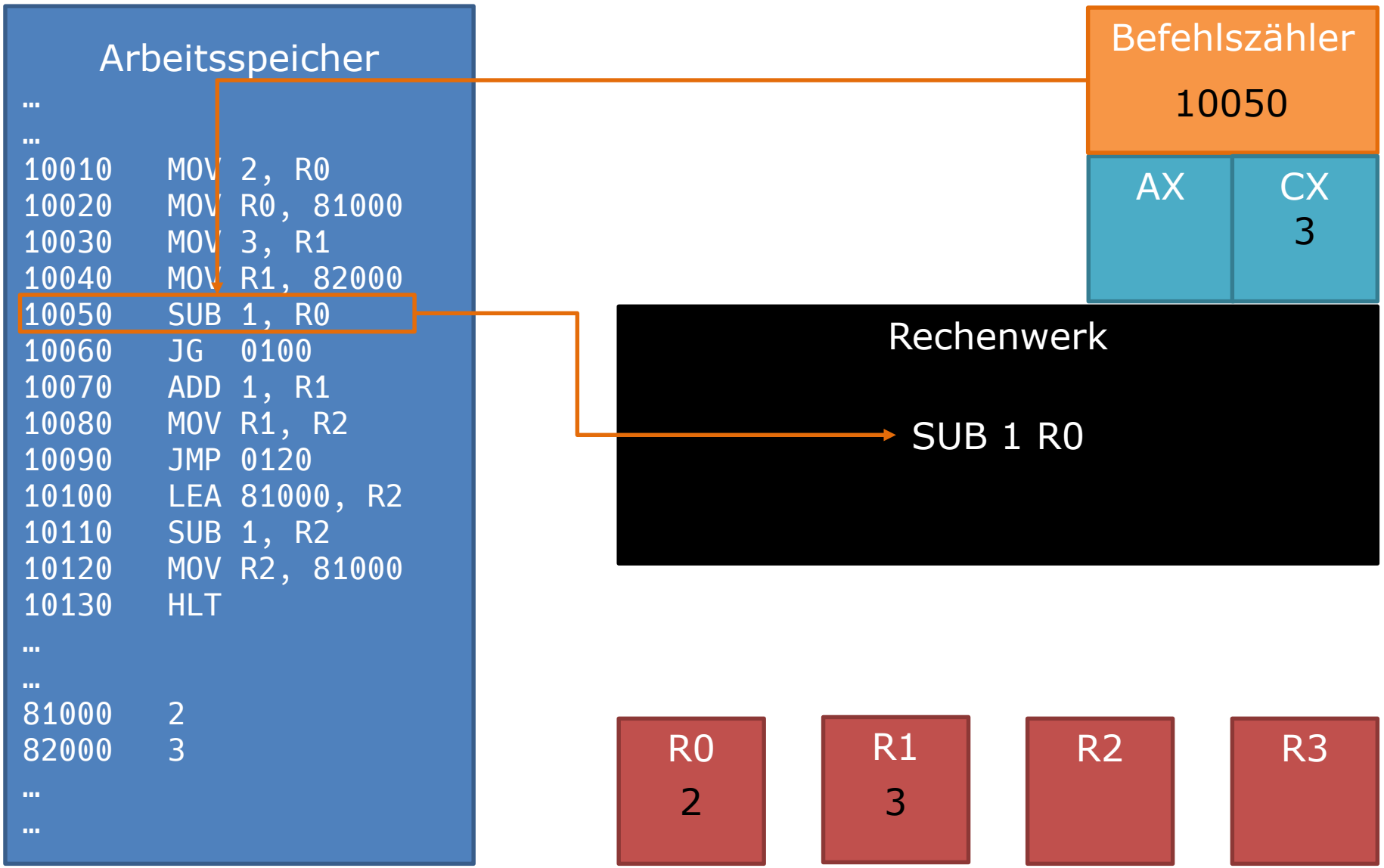
EINFACHES BEISPIEL: 15. TAKT



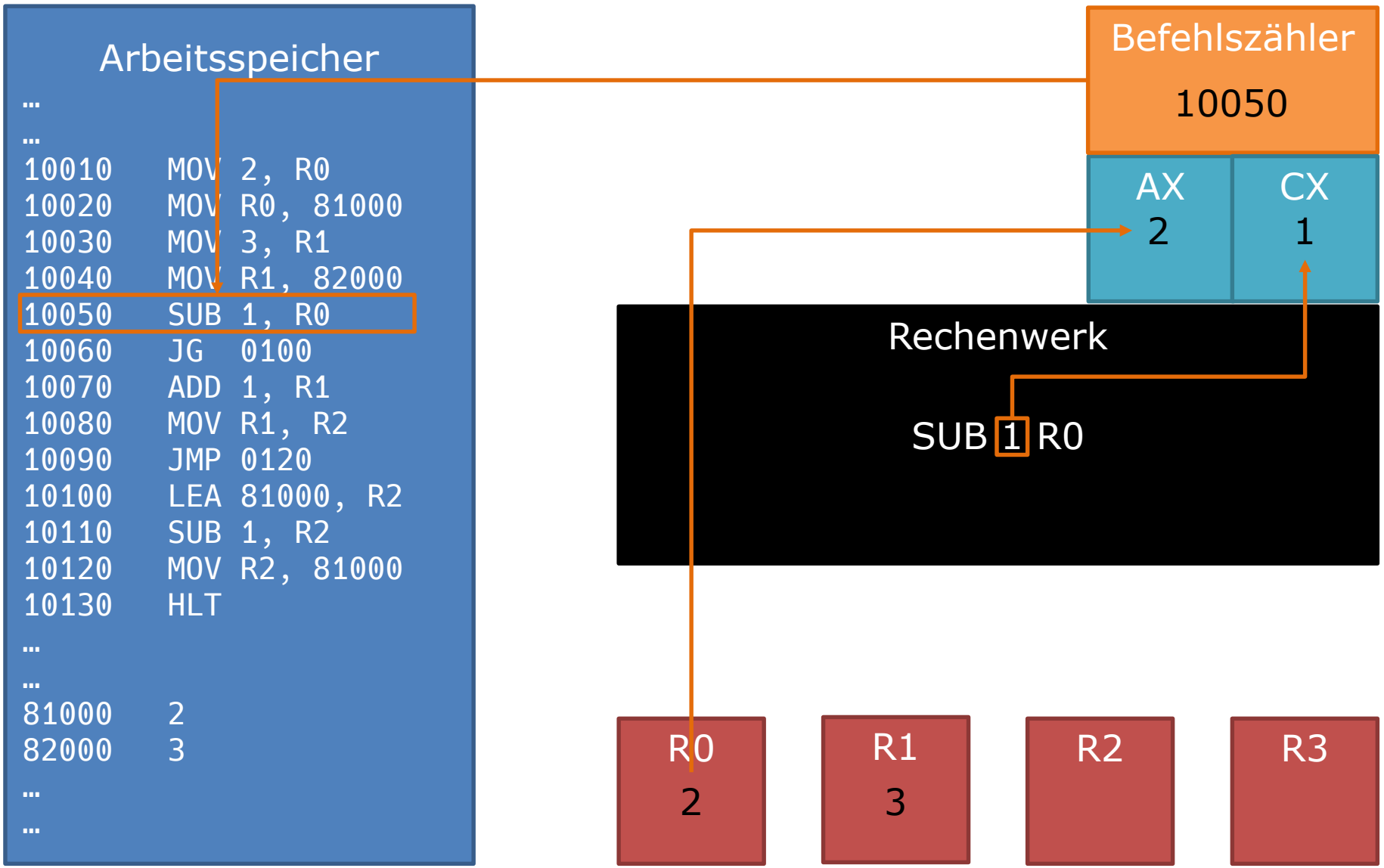
EINFACHES BEISPIEL: 16. TAKT



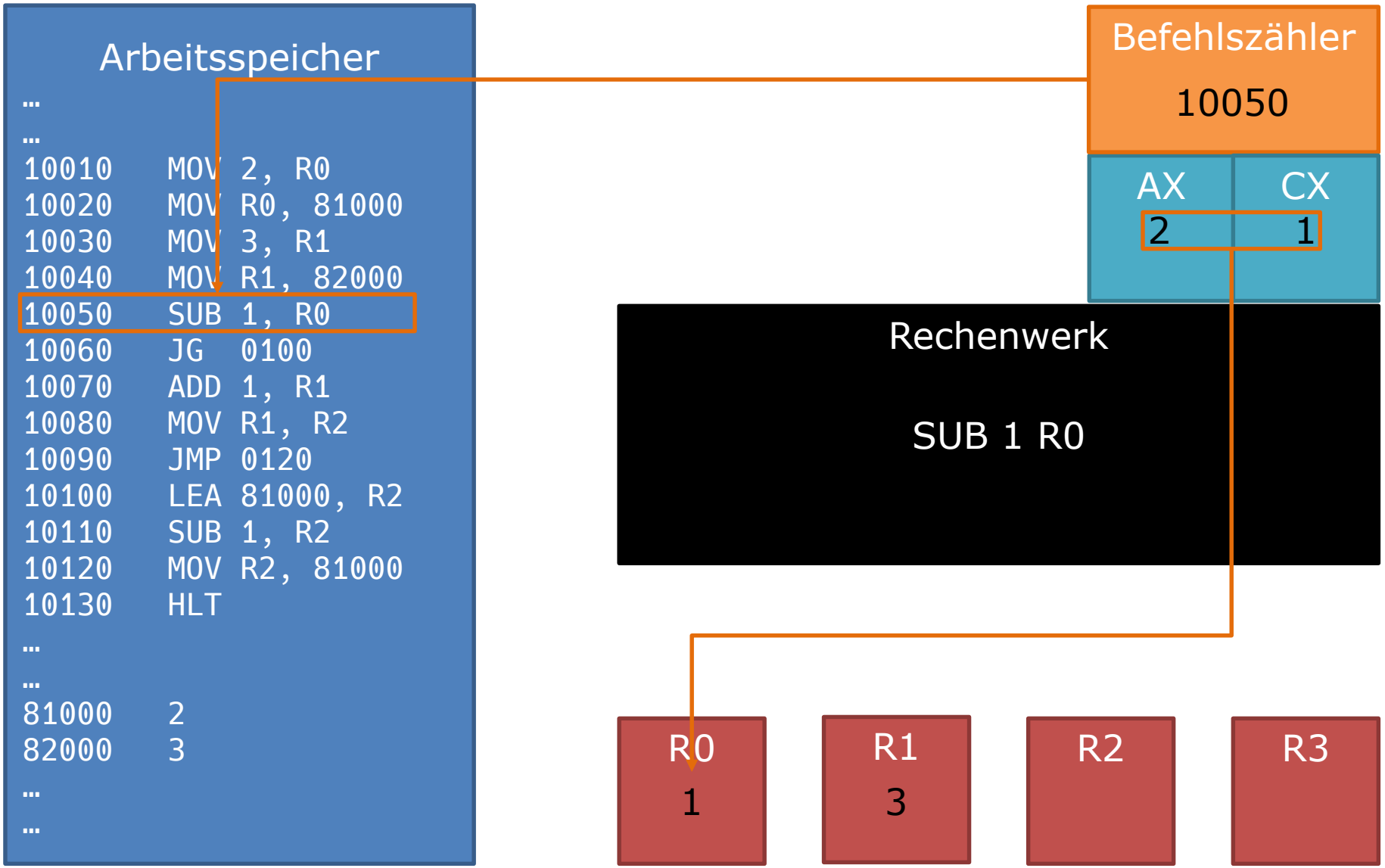
Einfaches Beispiel: 17. Takt



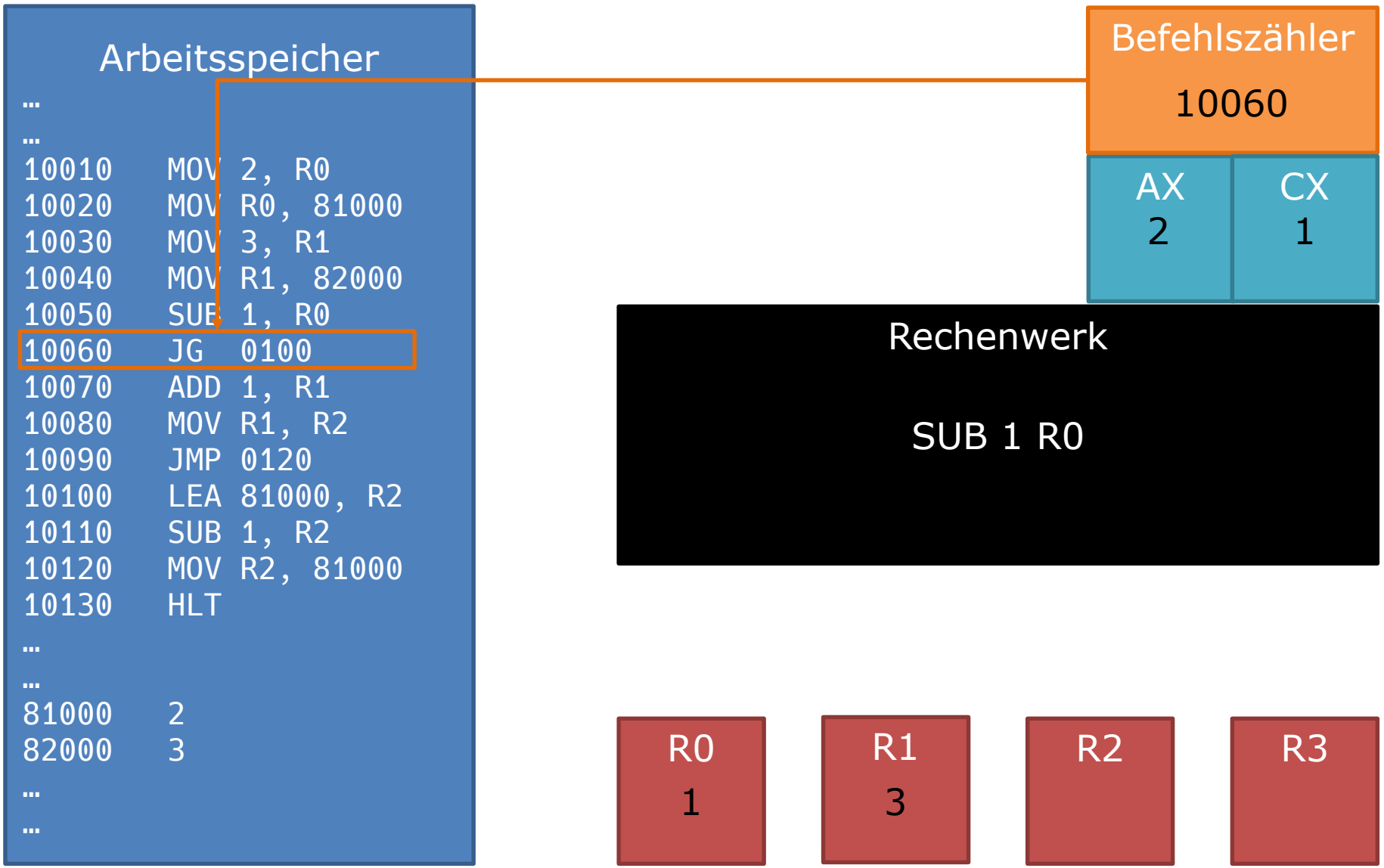
EINFACHES BEISPIEL: 18. TAKT



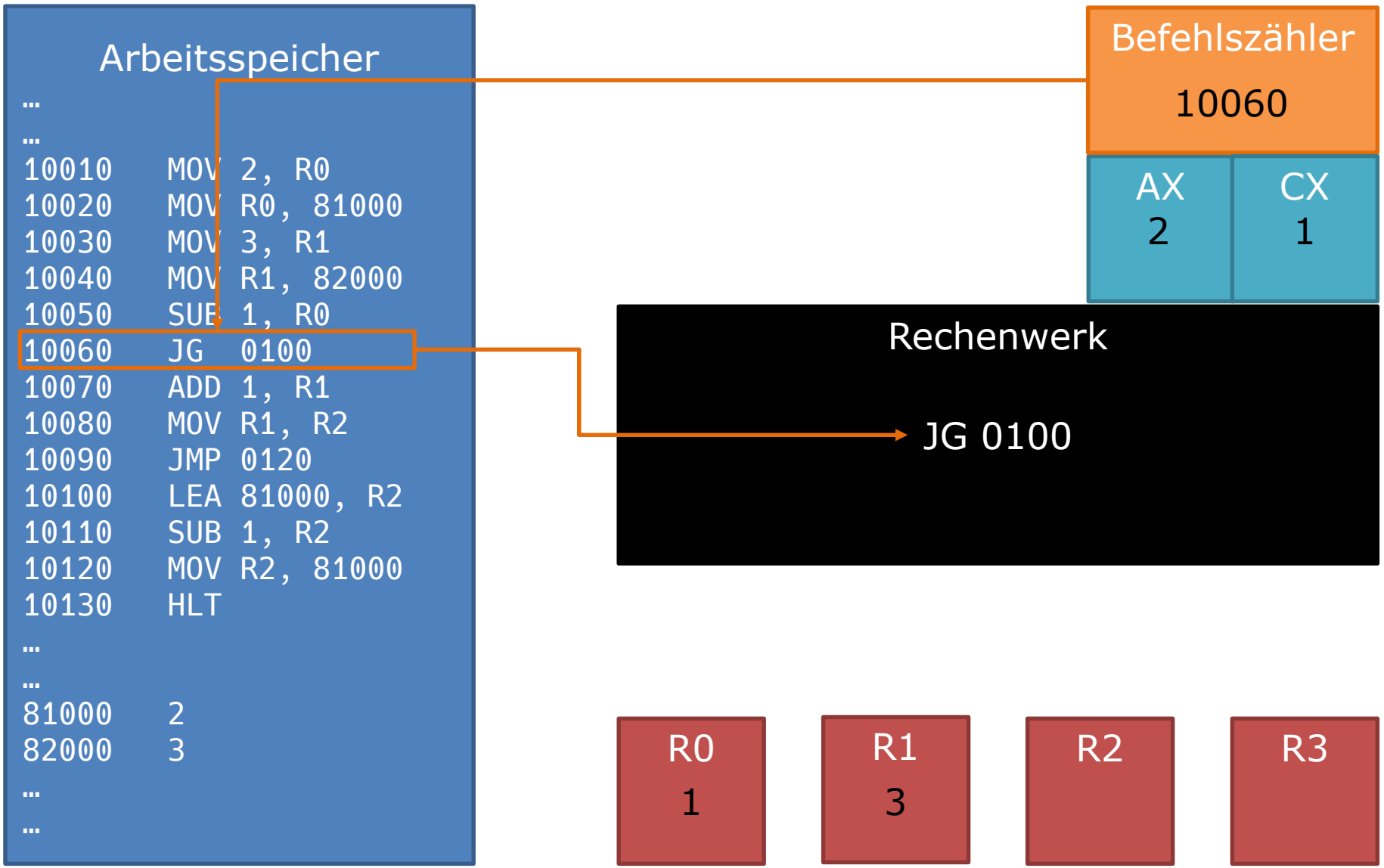
EINFACHES BEISPIEL: 19. TAKT



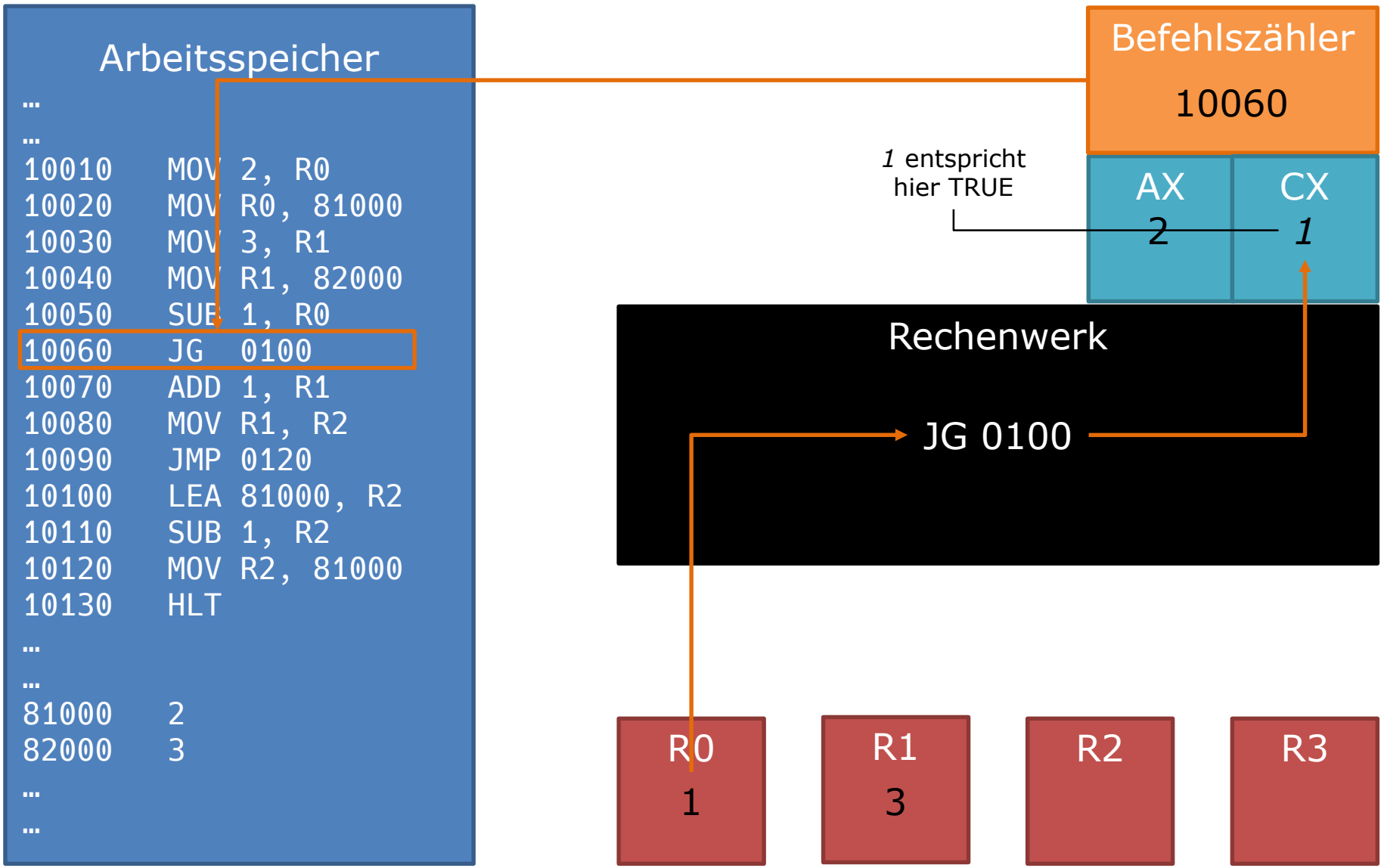
EINFACHES BEISPIEL: 20. TAKT



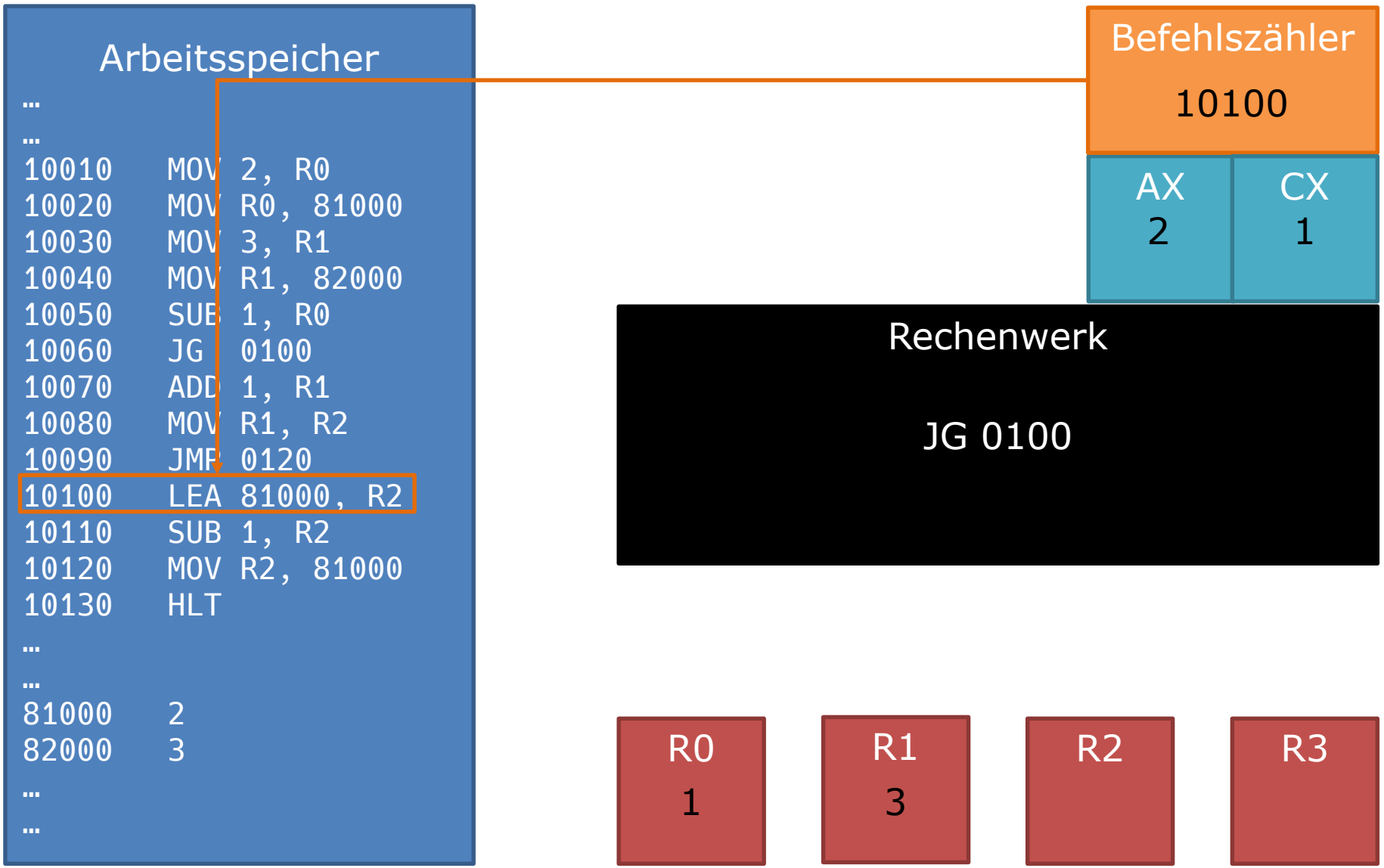
EINFACHES BEISPIEL: 21. TAKT



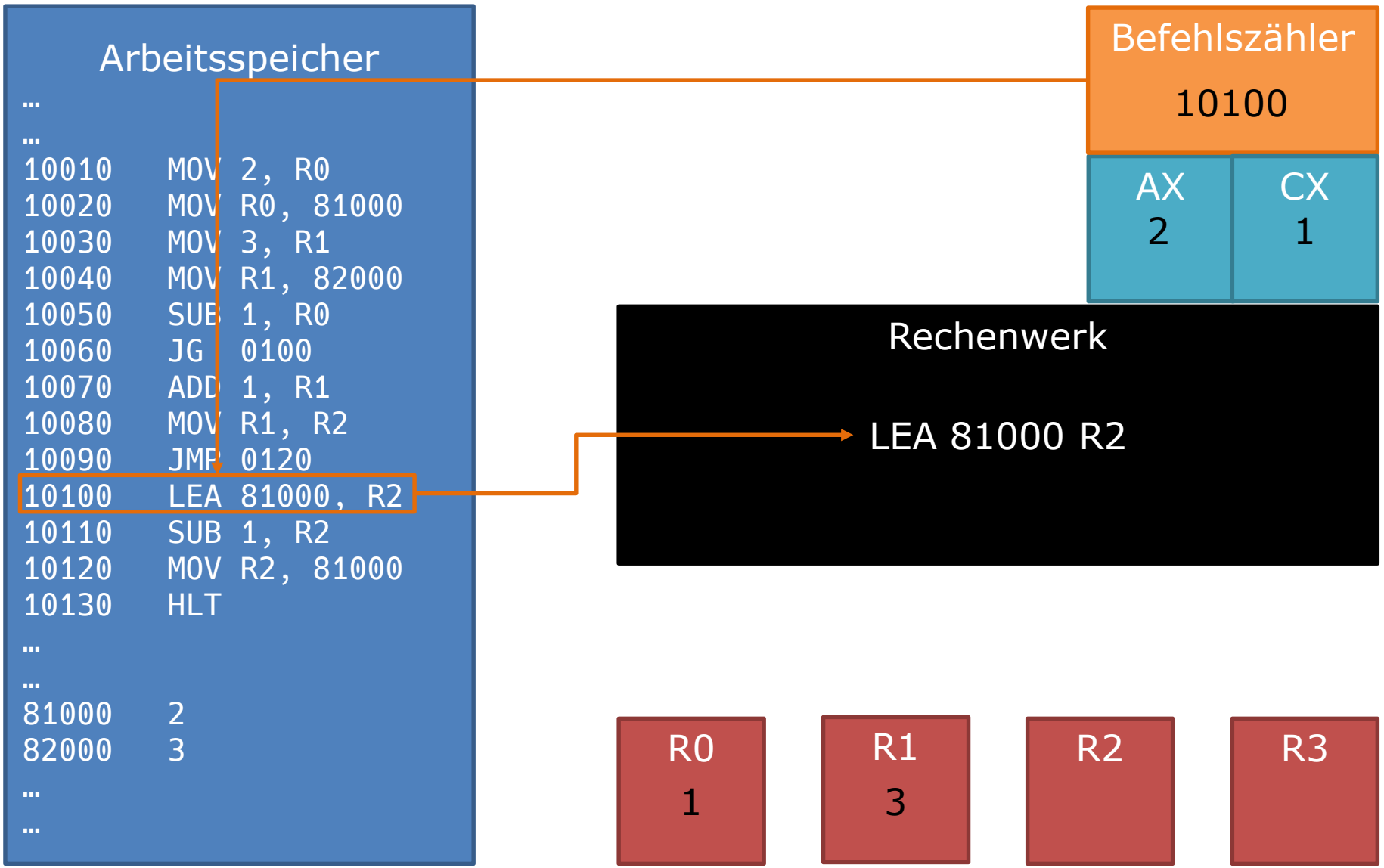
EINFACHES BEISPIEL: 22. TAKT



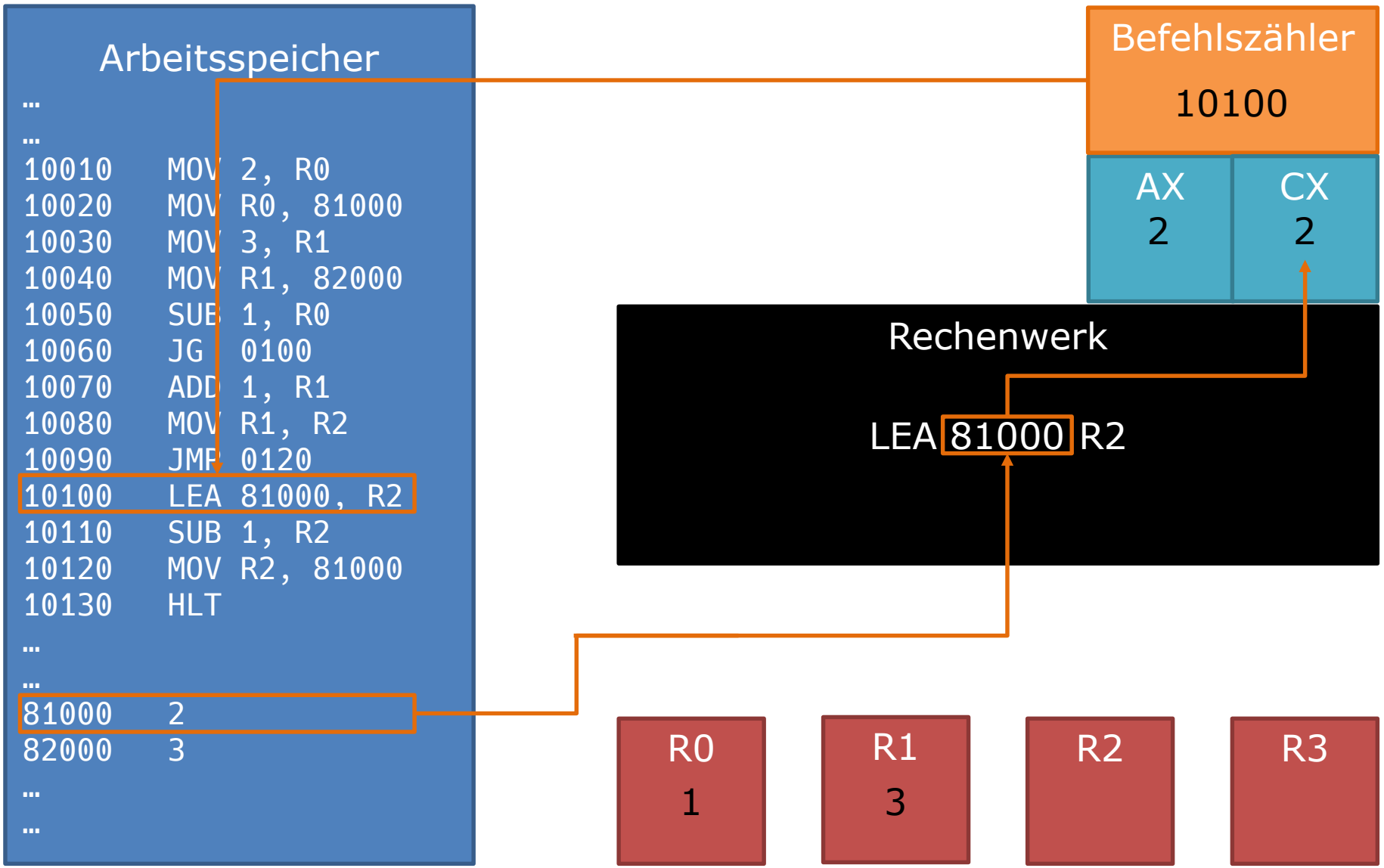
EINFACHES BEISPIEL: 23. TAKT



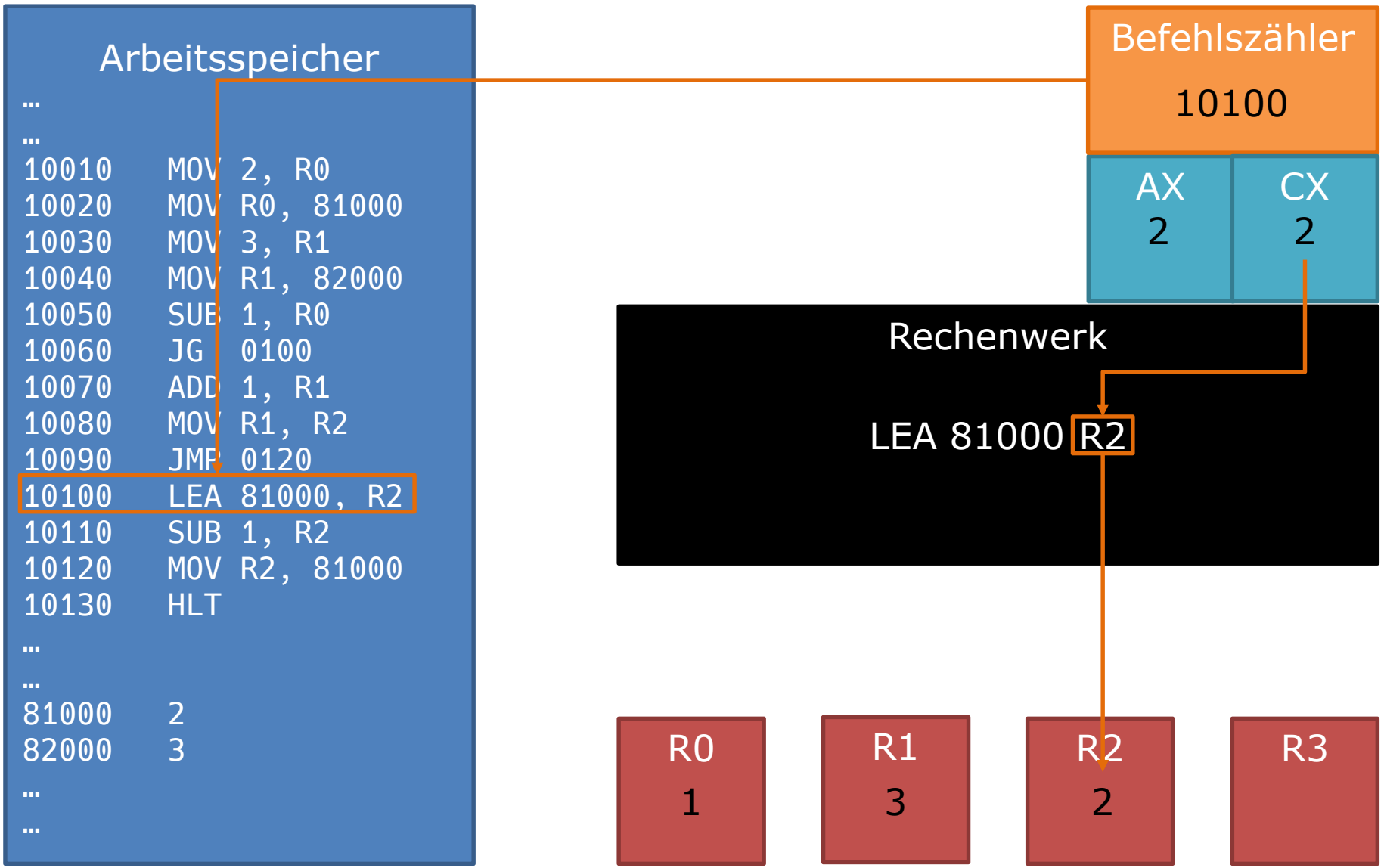
EINFACHES BEISPIEL: 24. TAKT



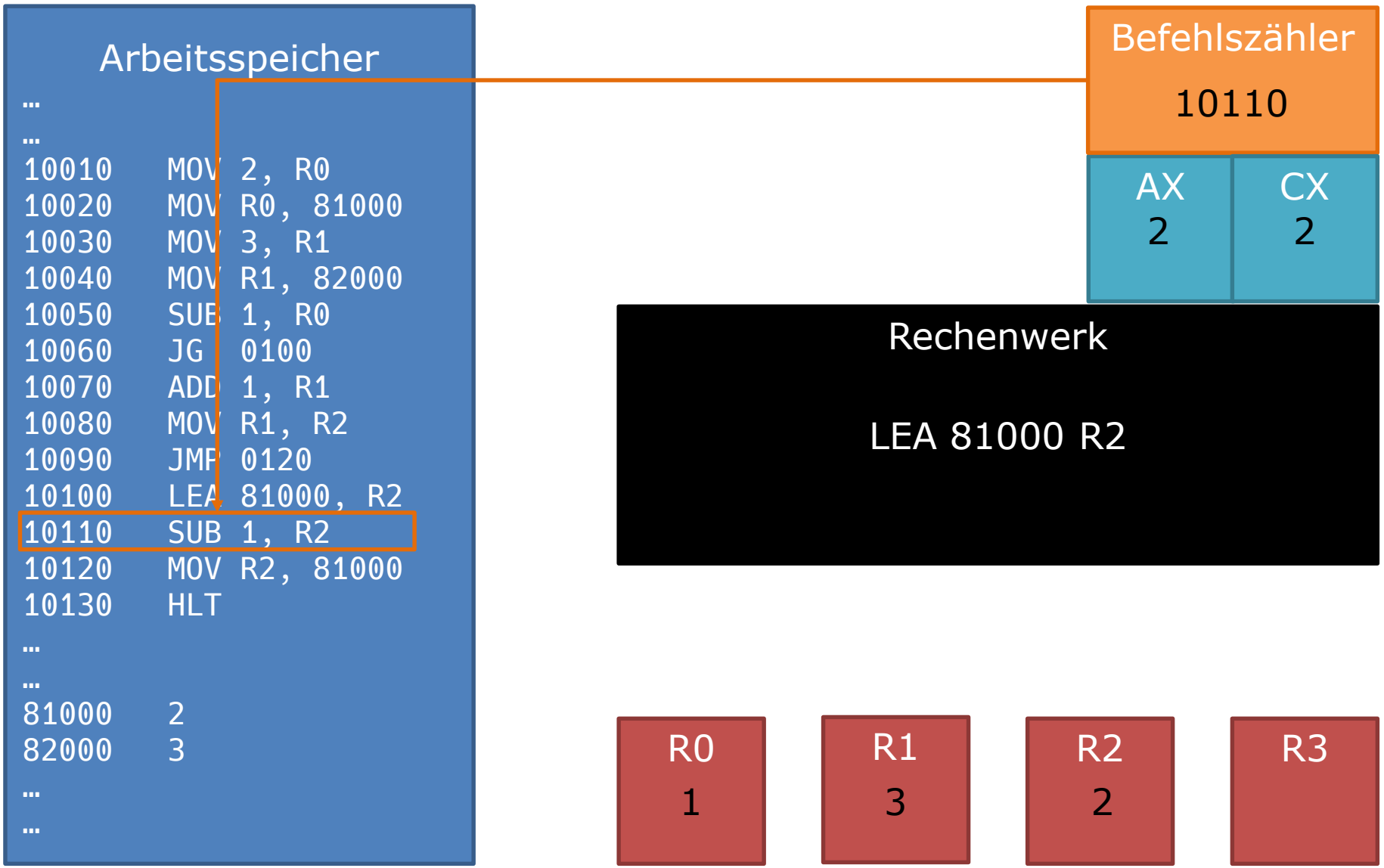
EINFACHES BEISPIEL: 25. TAKT



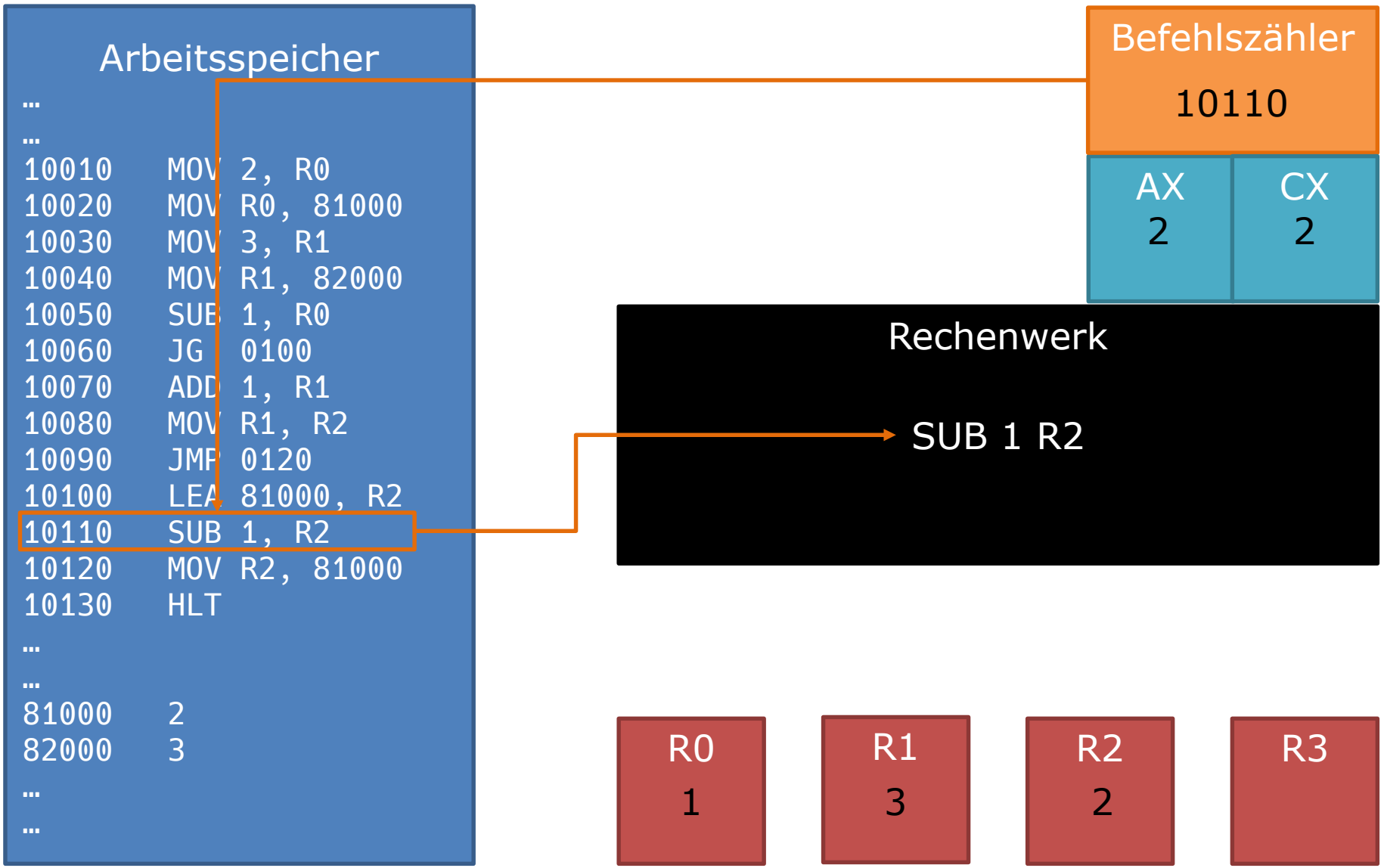
EINFACHES BEISPIEL: 26. TAKT



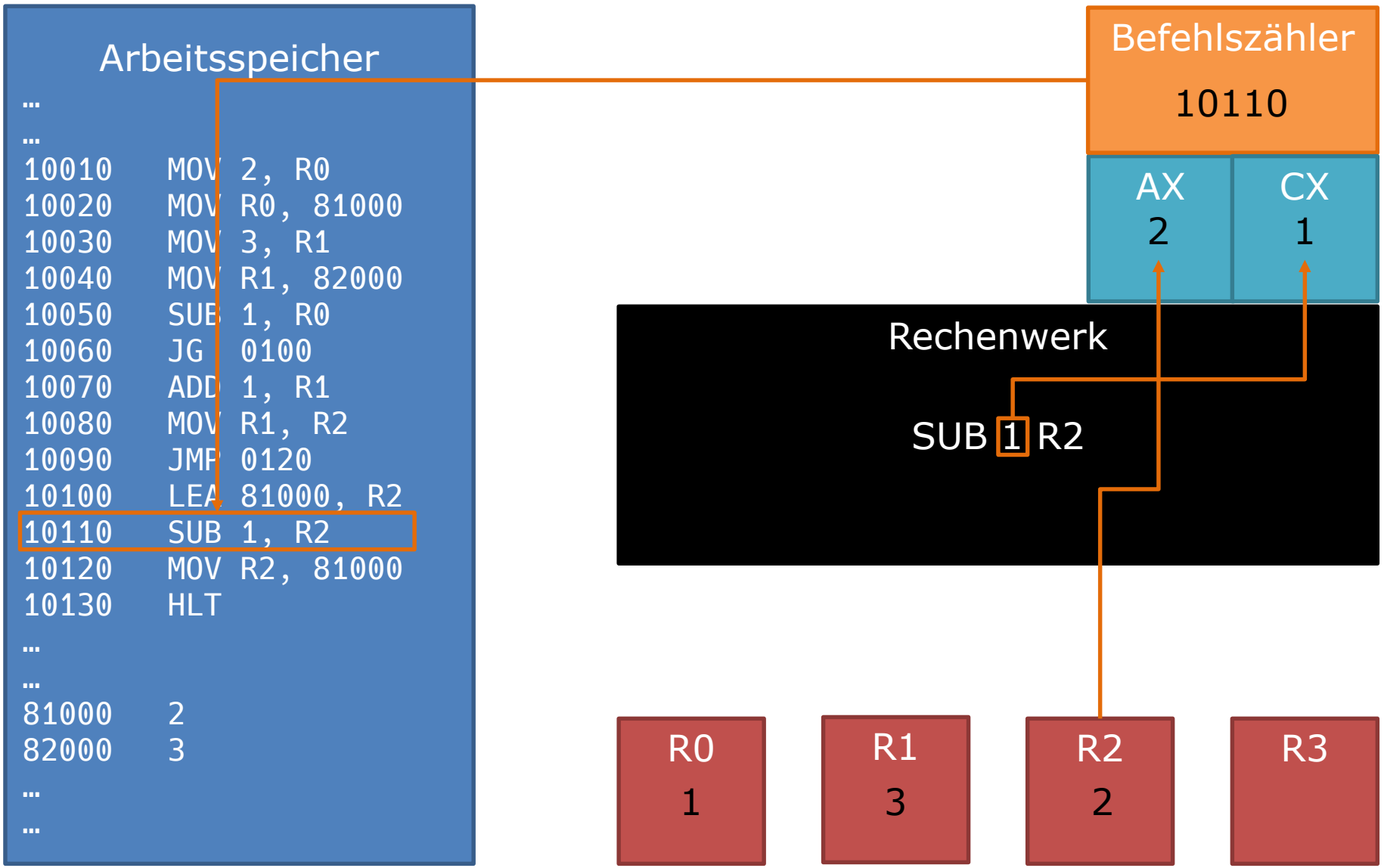
EINFACHES BEISPIEL: 27. TAKT



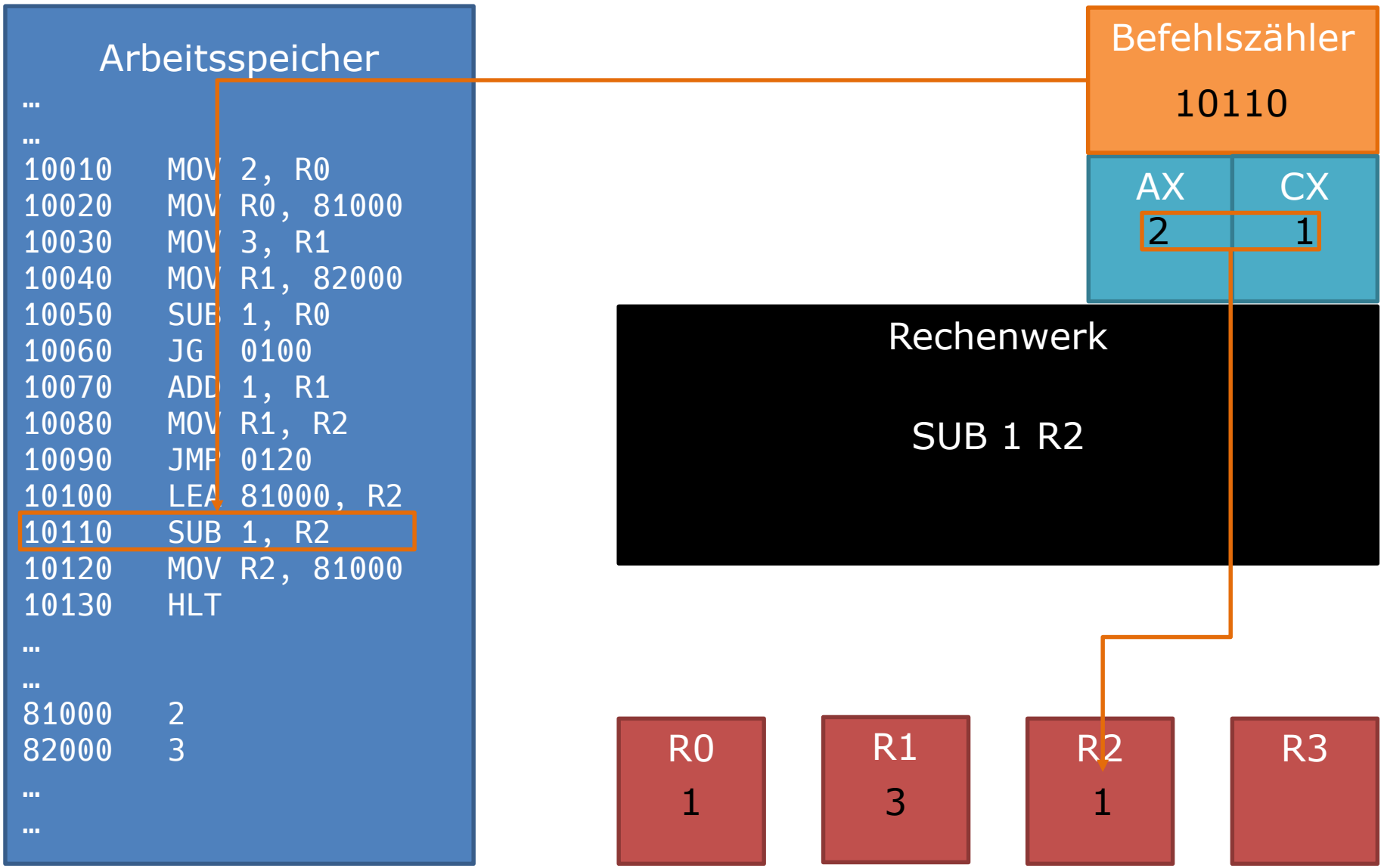
EINFACHES BEISPIEL: 28. TAKT



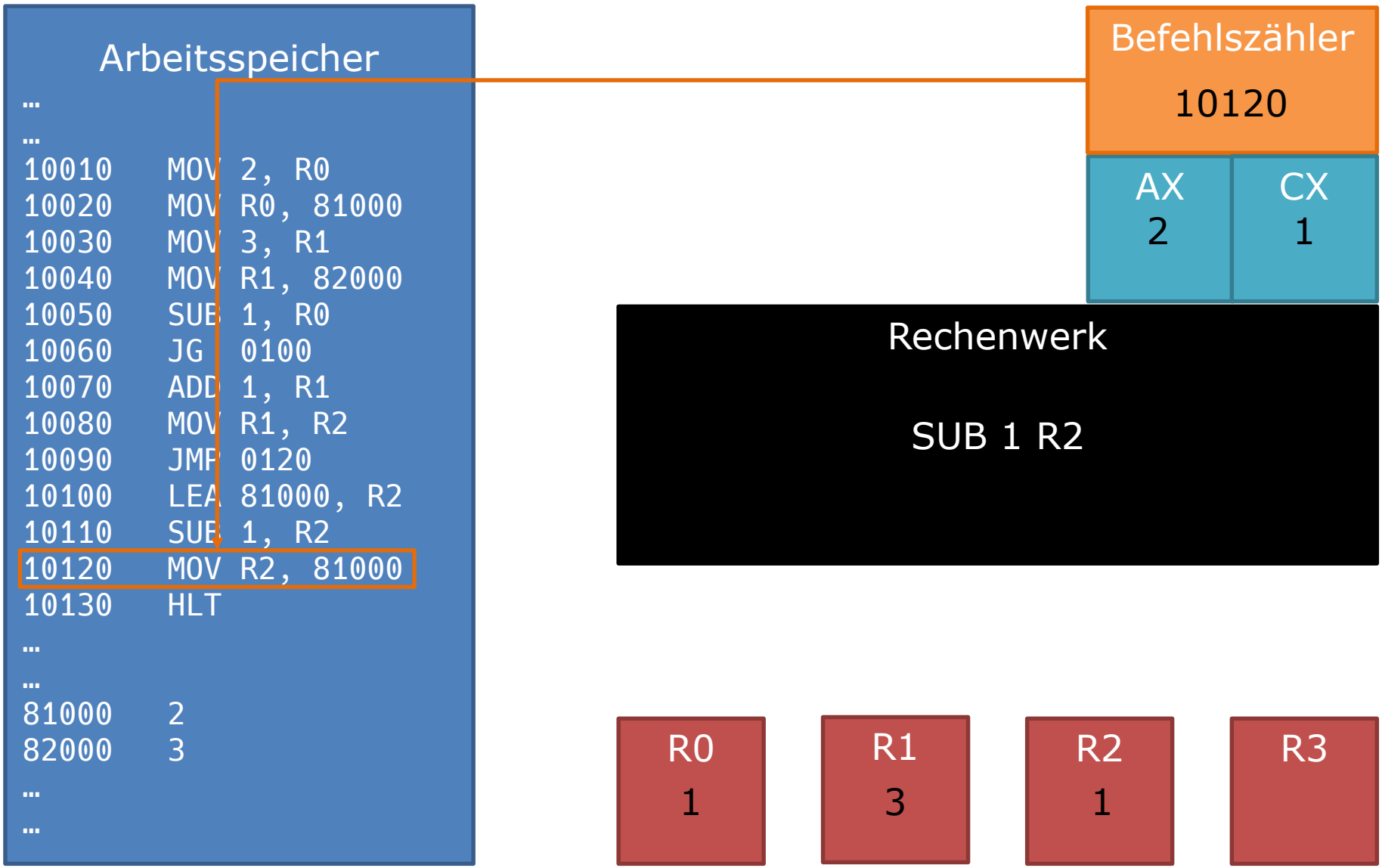
EINFACHES BEISPIEL: 29. TAKT



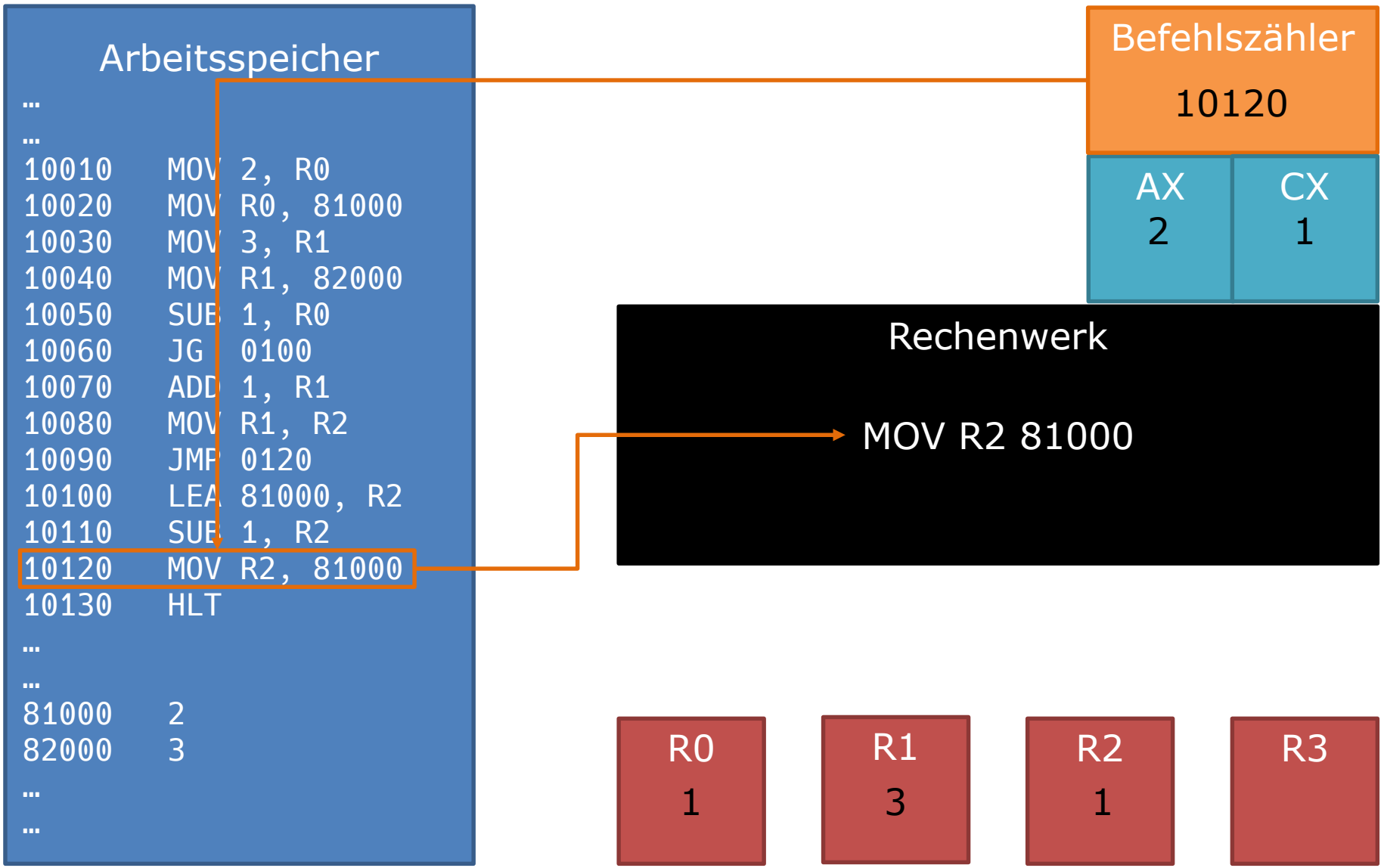
EINFACHES BEISPIEL: 29. TAKT



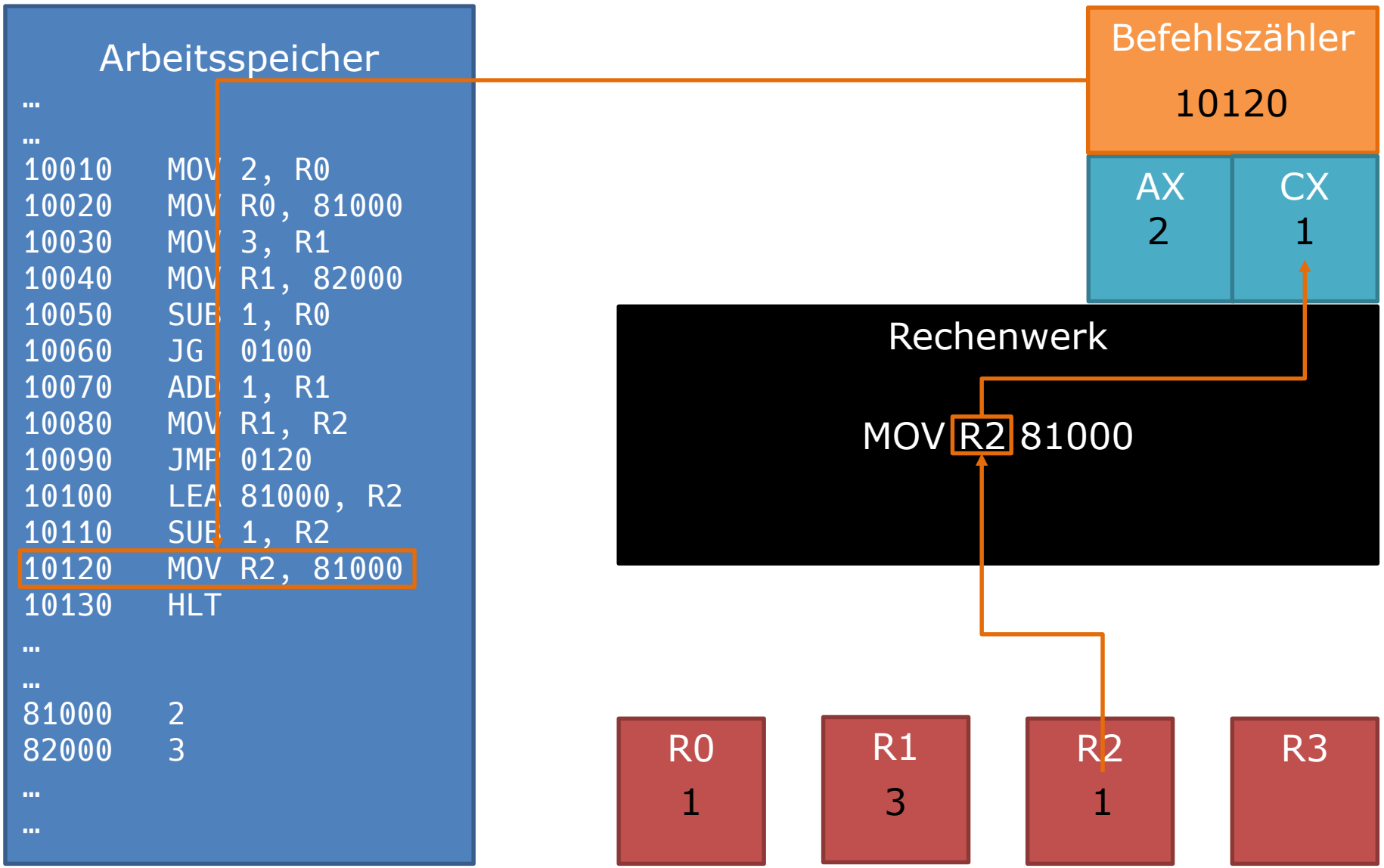
EINFACHES BEISPIEL: 30. TAKT



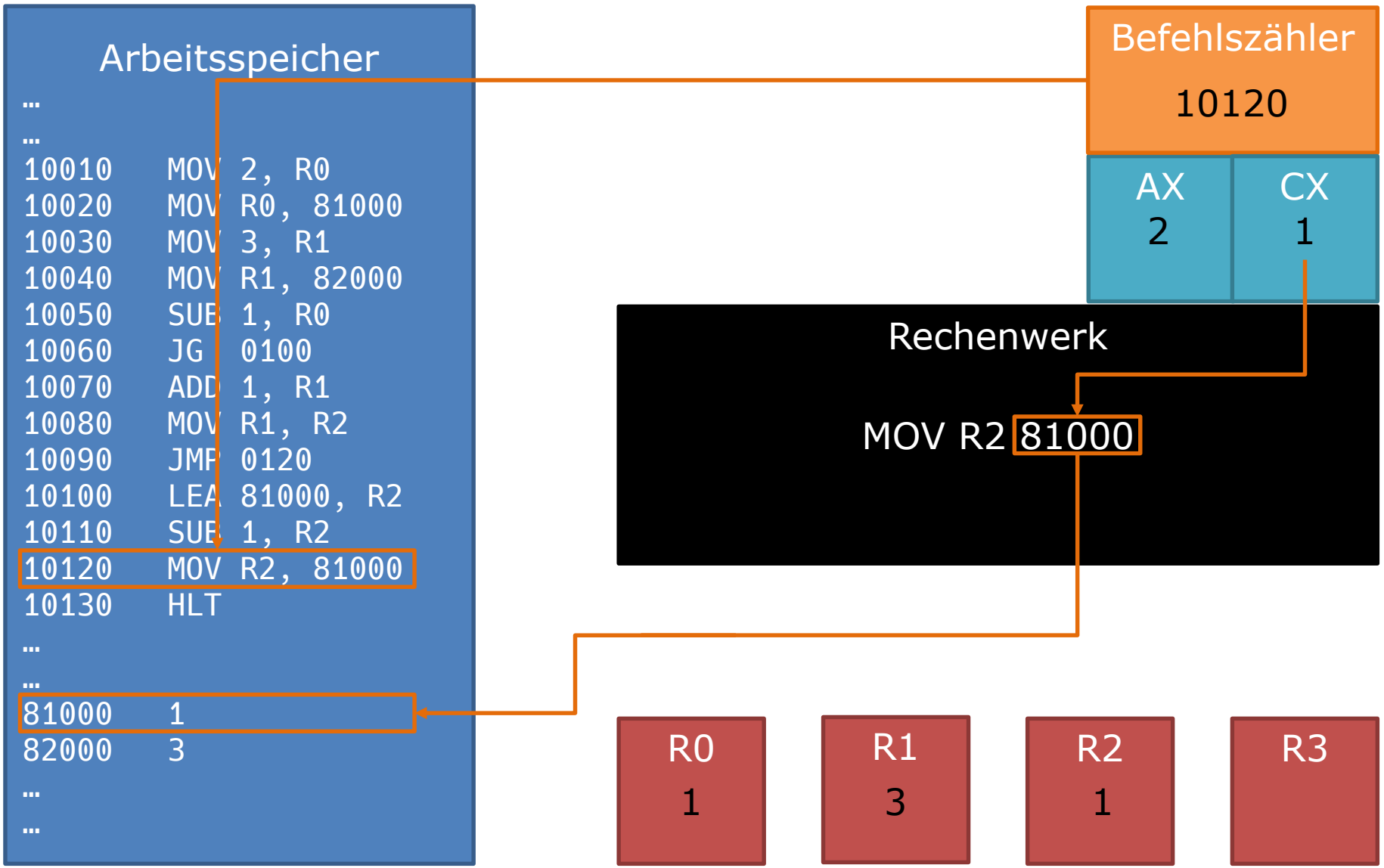
EINFACHES BEISPIEL: 31. TAKT



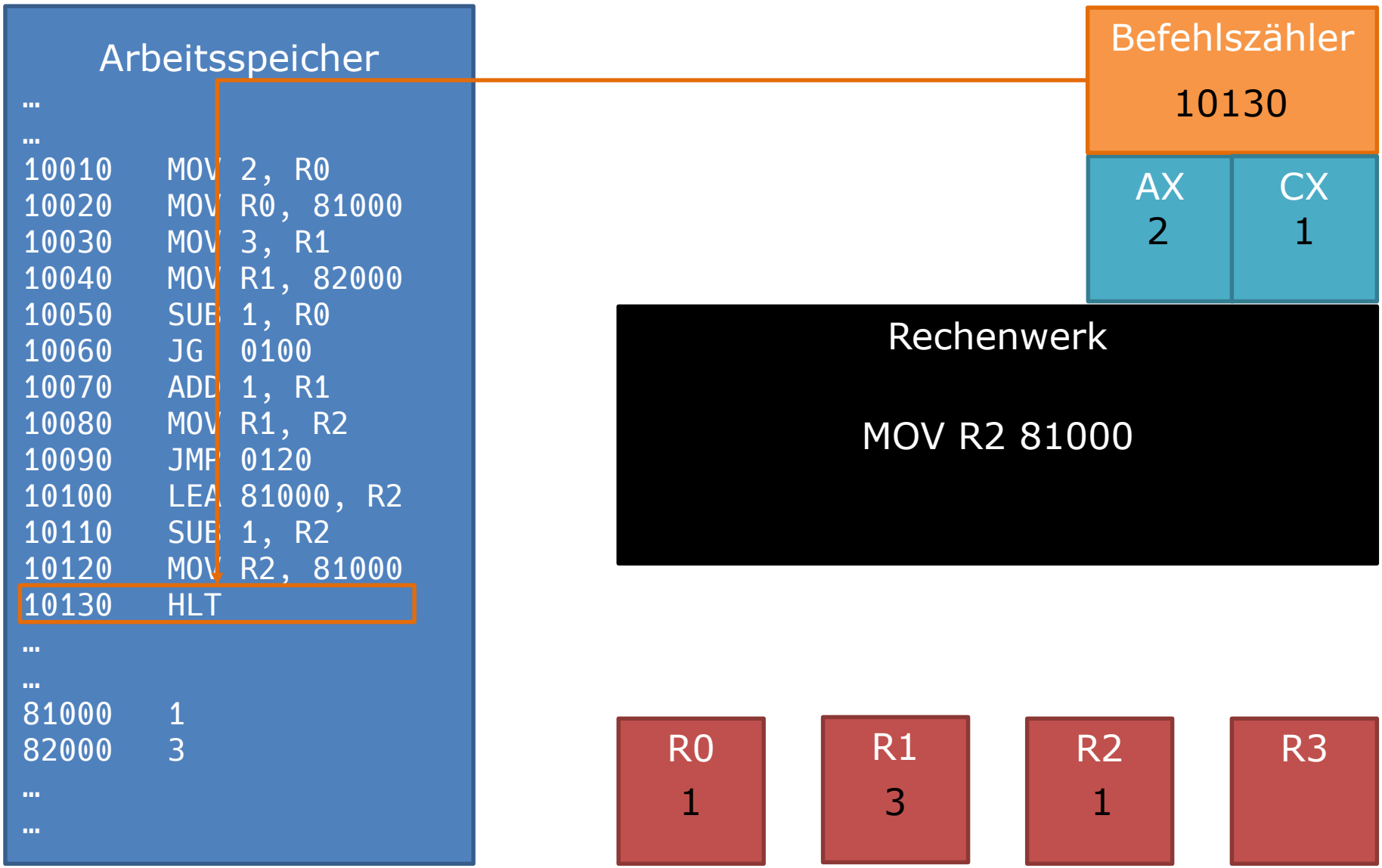
EINFACHES BEISPIEL: 32. TAKT



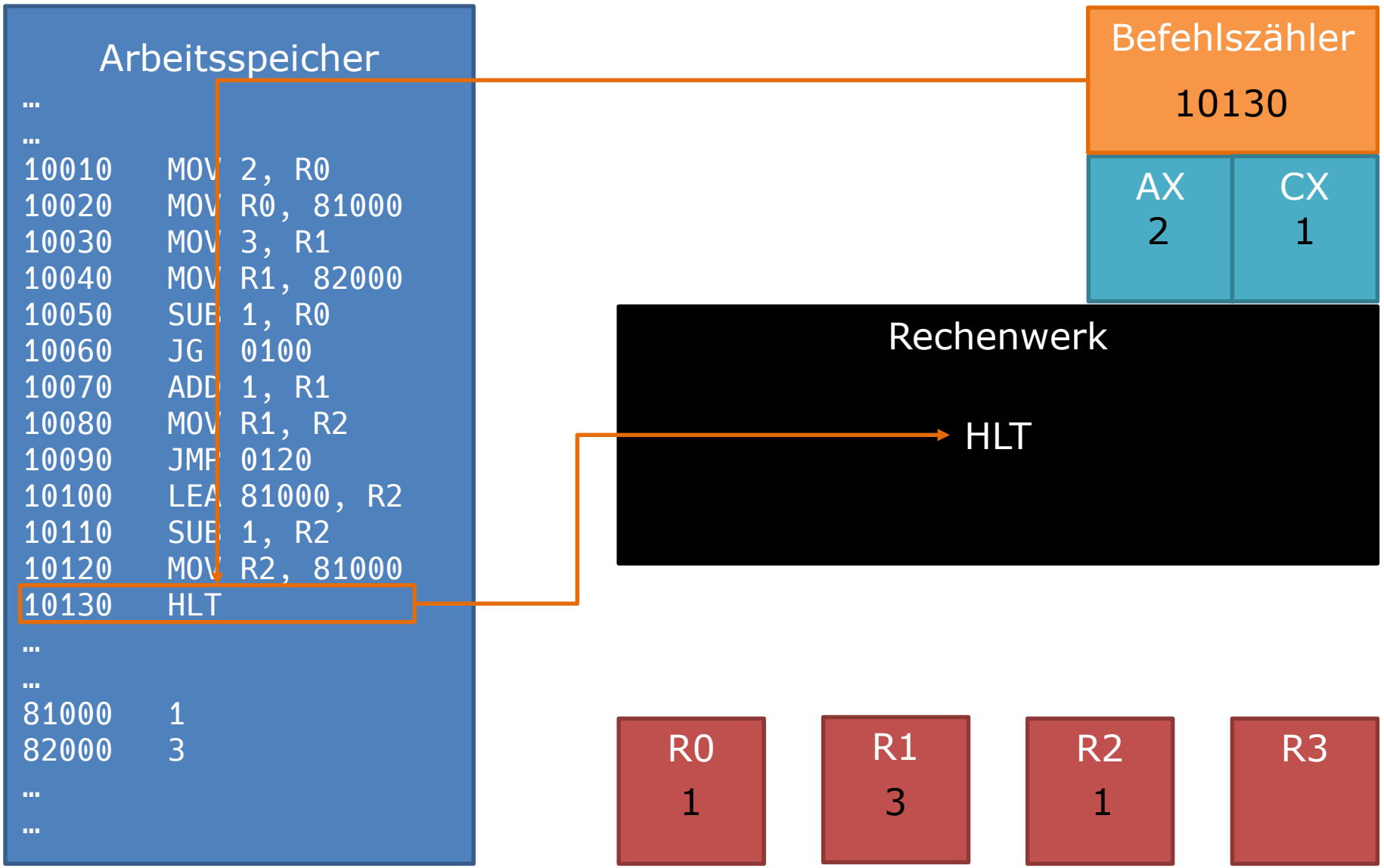
EINFACHES BEISPIEL: 33. TAKT



Einfaches Beispiel: 34. Takt



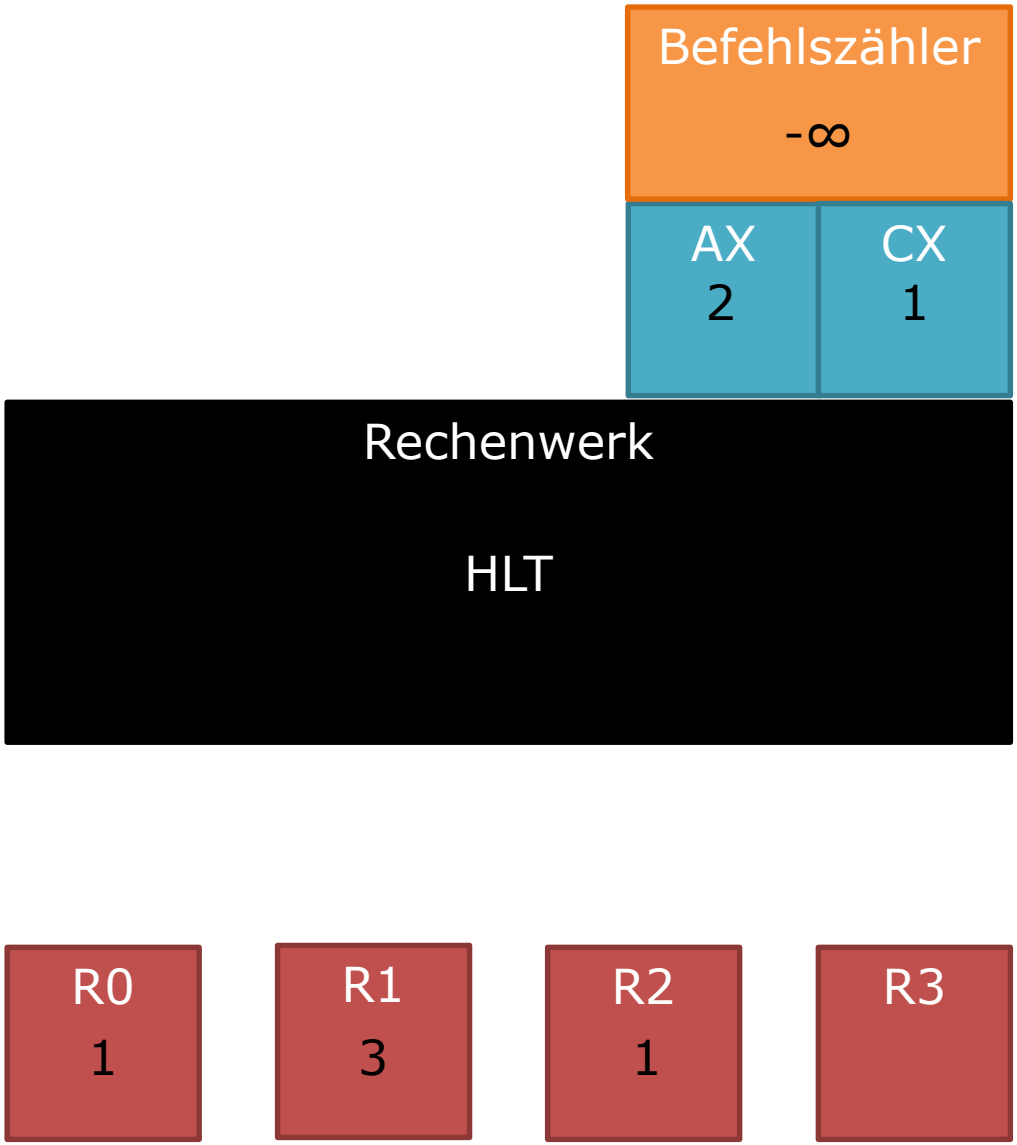
EINFACHES BEISPIEL: 35. TAKT



Einfaches Beispiel: ab 36. Takt

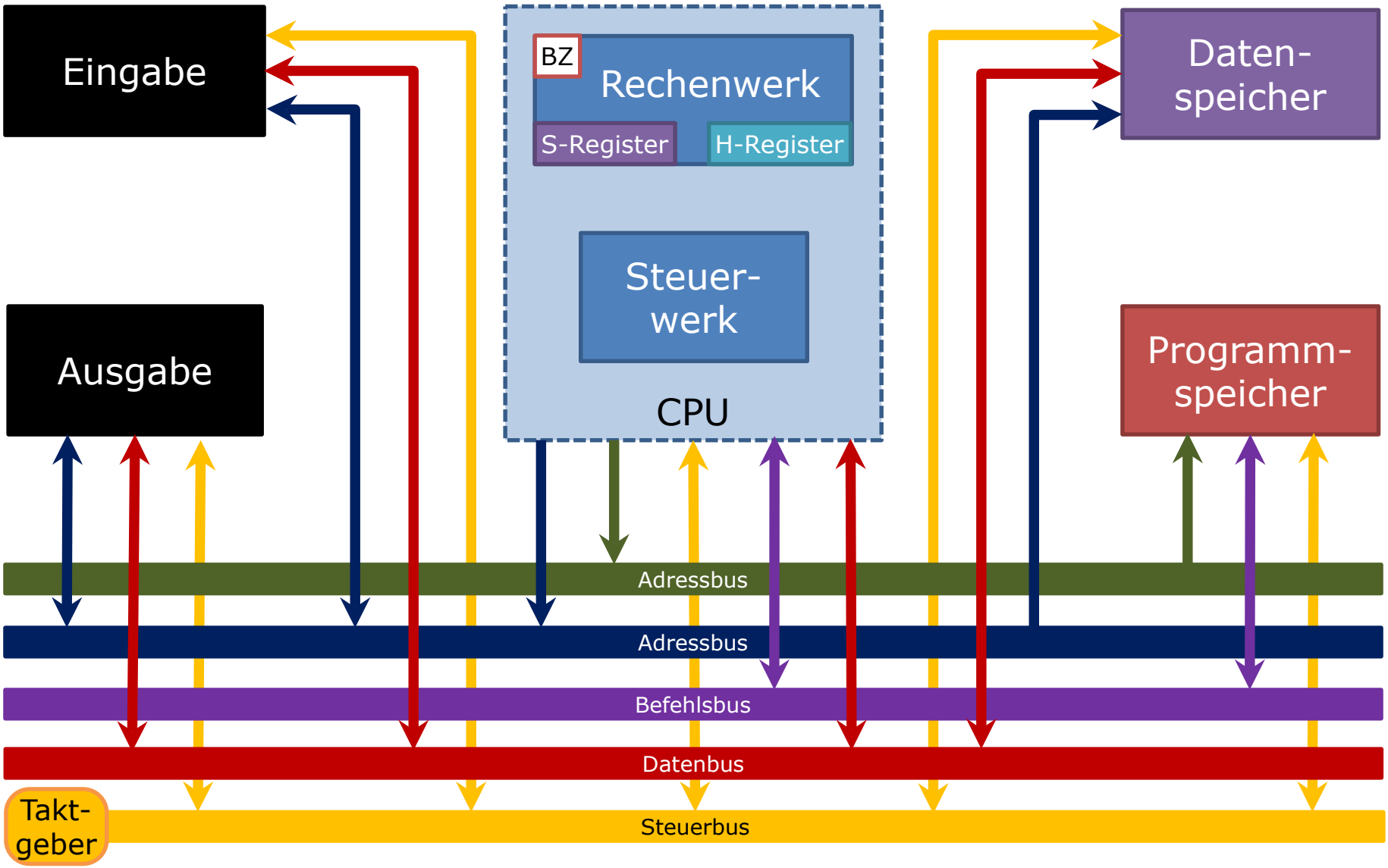
Arbeitsspeicher

```
…
…
10010   MOV 2, R0
10020   MOV R0, 81000
10030   MOV 3, R1
10040   MOV R1, 82000
10050   SUB 1, R0
10060   JG  0100
10070   ADD 1, R1
10080   MOV R1, R2
10090   JMP 0120
10100   LEA 81000, R2
10110   SUB 1, R2
10120   MOV R2, 81000
10130   HLT
…
…
81000   1
82000   3
…
…
```



Harvard-Architektur

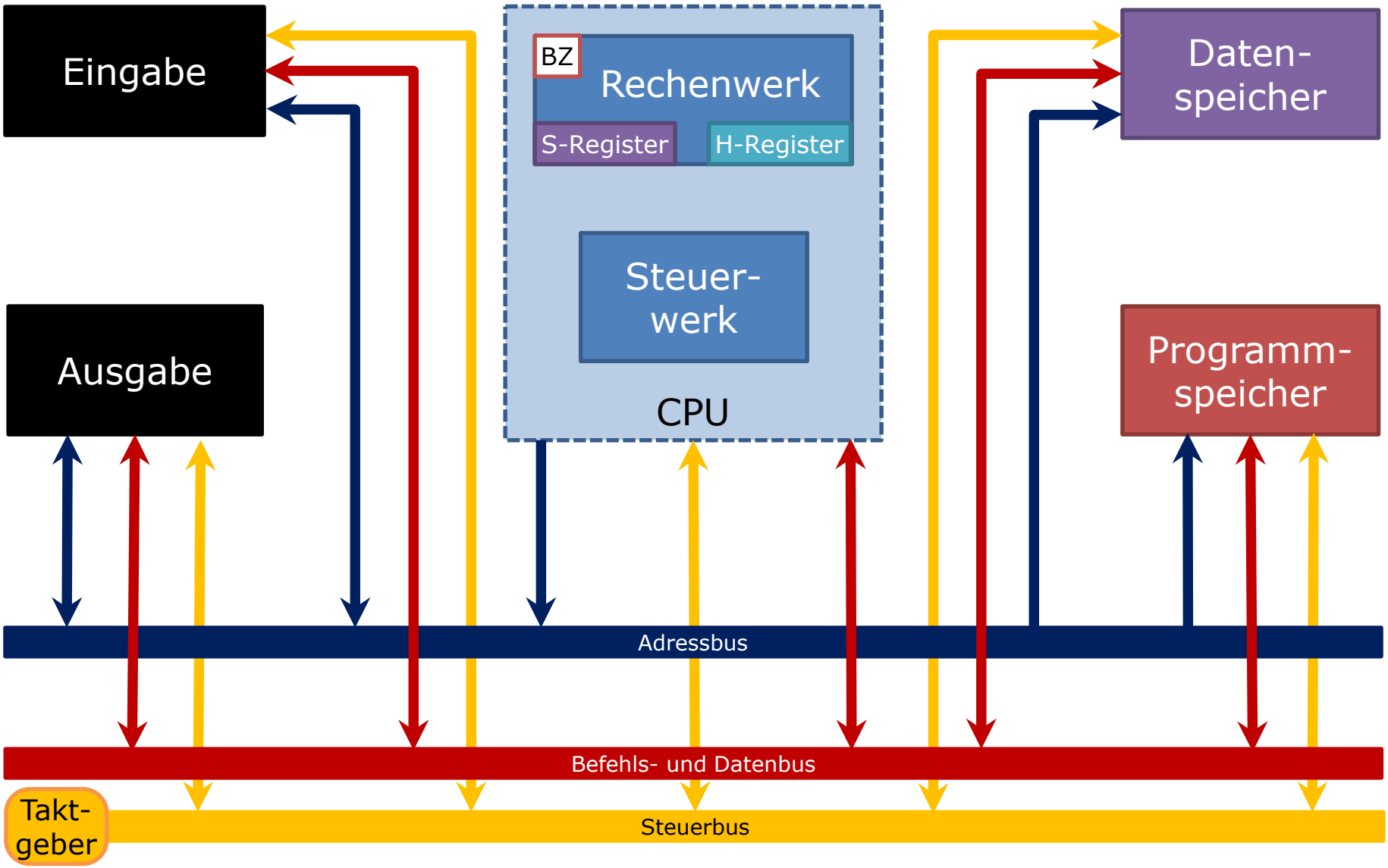
KLASSISCHE HARVARD-ARCHITEKTUR (STATISCHE PROGRAMME)



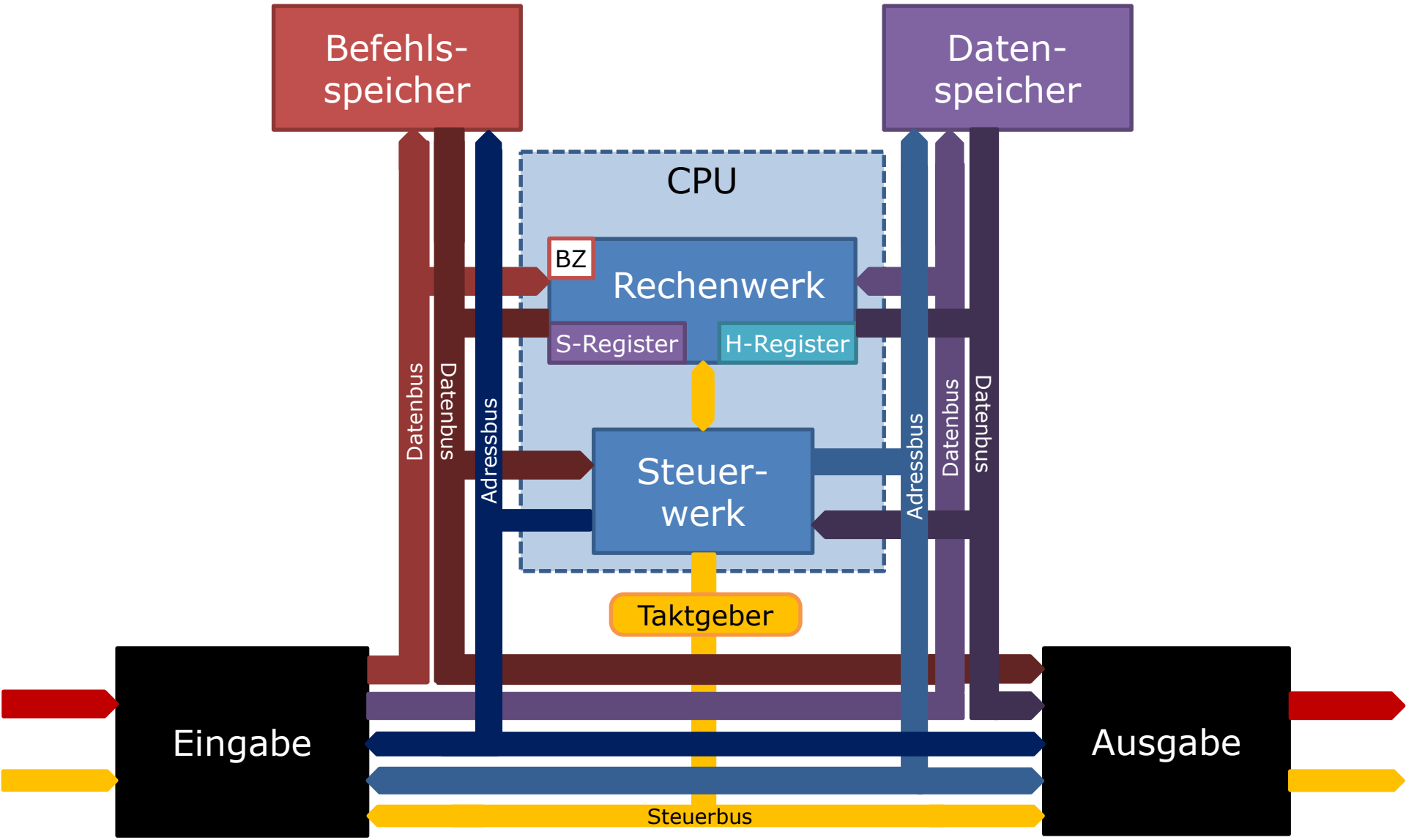
Merkmale der Harvard-Architektur

- getrennte Speicher für Programmkode und Daten
- getrennte Daten- und Steuerbusse für Programmkode und Daten
- erlaubt somit das gleichzeitige Lesen eines Befehlskodeworts und das Lesen oder Schreiben eines Datenworts
- erlaubt begrenzte Parallelisierung der Befehlskodeabarbeitung
- Mehrwortbefehle (bspw. Read-Modify-Write) auf Daten verhindern, dass Befehle innerhalb eines Speicherzyklus abgearbeitet werden
- Befehle ohne Datenspeicherzugriffe werden gegenüber von-Neumann nicht beschleunigt
- die strikte Trennung von Befehls- und Datenbus ist außer in Spezialfällen unüblich, da nur fertige Programme aus ROM ausführbar wären

KLASSISCHE HARVARD-ARCHITEKTUR (DYNAMISCHE PROGRAMME)



Super Harvard-Architektur (SHArc)

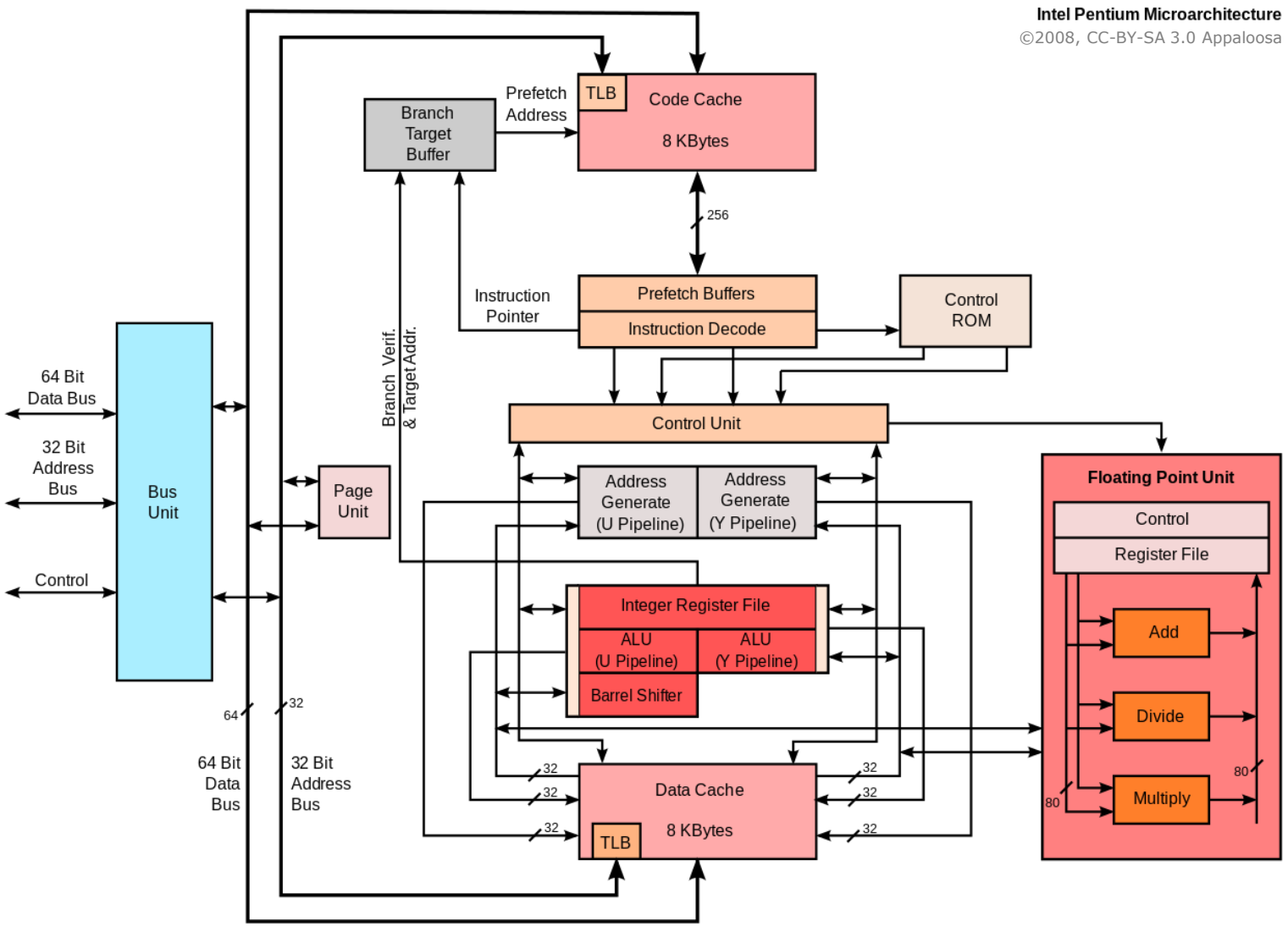


Merkmale der SHArc

- sehr gut geeignet für digitale Signalprozessoren (DSP)
  mit zwei oder vier Bussystemen
- oft verbunden mit Direktspeicherzugriff und Video-Controllern unter Aufweichung der Trennung der Bussysteme:
  jeder Bus kann sowohl Steuerinformationen, Kode und Daten liefern während der Speicher unabhängig von der CPU adressierbar ist

modernere Architekturen

Intel Pentium



Kombination aus von-Neumann- und Harvard-Architektur

- Miniaturisierungsgrenzen verhindern noch schnellere/kleinere Rechner
  → Lichtgeschwindigkeit, Wärmeentwicklung
  → nur mit Parallelisierung lösbar
- moderne Prozessoren verwenden Mischformen aus Harvard- (bzw. SHArc) und von-Neumann-Architektur
- innerhalb des Prozessorchips werden Daten und Programm voneinander getrennt verwaltet werden
- separate Puffer und Speicherverwaltungseinheiten für Daten und Programm über getrennte interne Busse
- gemeinsamer externer Speicher für Daten und Programm
- CPU-Pipelining wird dadurch ermöglicht
  (einzelne Pipelinestufen können bzgl. Speicherzugriffe getrennt werden)

Befehlstypen und Speicherzugriff

Flynn'sche Klassifikation

SISD

Single Instruction on Single Data

- traditionelle Einkernprozessoren
- Abarbeitung von Aufgaben sequentiell
- klassische von-Neumann- und Harvard-Architektur gehört hier rein

MISD

Multiple Instructions on Single Data

- fehlertolerante Systeme
- Makropipelining
- Schachcomputer

SIMD

Single Instruction on Multiple Data

- Feld- und Vektor-Prozessoren
- Ausführung gleichartiger Operationen auf mehrere Daten(ströme)
- modernen Architekturen (ARM, Power, x86, AMD64) haben SIMD-Einheiten

MIMD

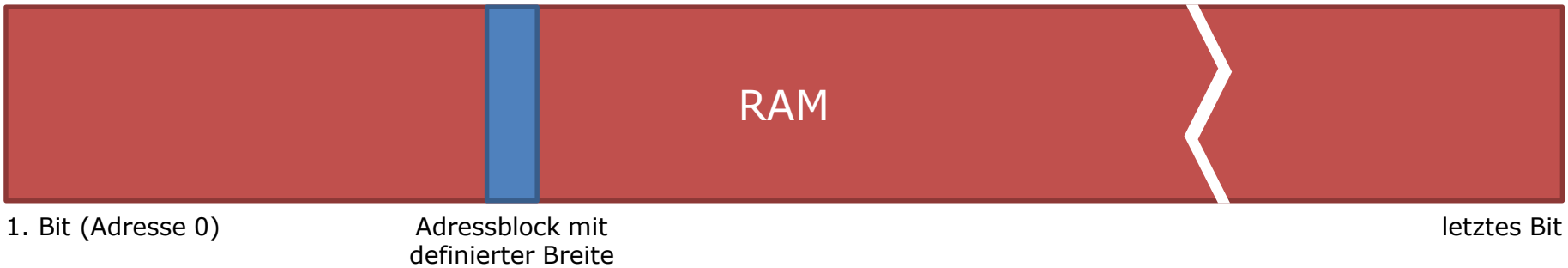
Multiple Instructions on Multiple Data

- Multiprozessorsysteme ohne Datentrennung (jeder Kern hat Zugriff)
- unterschiedliche Operationen auf mehreren Daten(ströme)
- UMA-, NUMA- und COMA-Systeme

Befehlssatzkomplexität

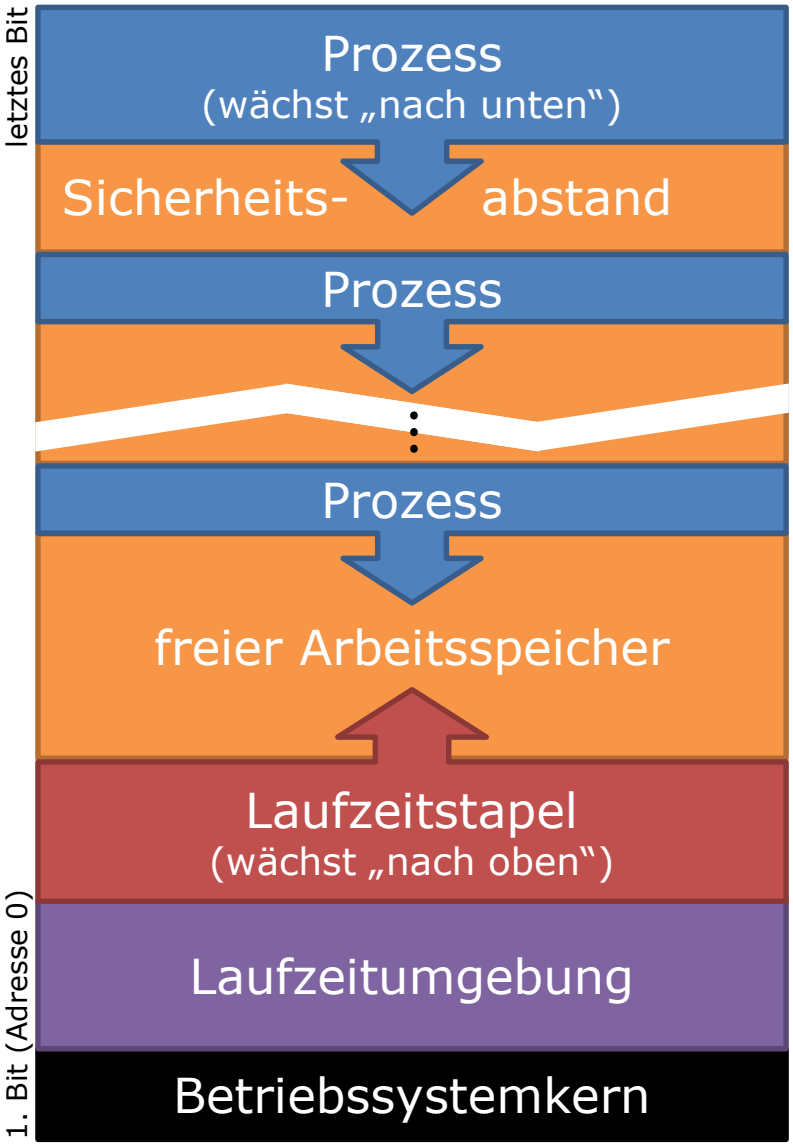
CISC Complex Instruction Set Computer	RISC Reduced Instruction Set Computer
• historische Rechner (68000, DX80386, Z80, …) aber in Teilen auch in AMD und x86	• fast alle modernen Rechner, insbesondere ARM, MIPS, POWER und SPARC
• umfangreicher Befehlssatz für komplexe (Spezial)Aufgaben (bspw. MULTSTR)	• auf Basisoperationen reduzierter Befehlssatz
• wenige Register, arbeitet direkt mit möglichst wenigen Zugriffen auf Arbeitsspeicher	• mehr Register, arbeitet mit Puffern (L1/L2) um Arbeitsspeicherzugriffe zu reduzieren
• pro Befehl 2 bis 15 Taktzyklen	• je Taktzyklus möglichst ein Befehl; im Mittel 1,5 Taktzyklen pro Befehl
• arbeitet mit Mikrokode zur Übersetzung der CISC-Befehle in ALU-Operationen	• arbeitet mit festverdrahtetem ALU- Befehlssatz
• arbeiten i.d.R. ohne Pipeline	• hochgradig gepipelinet

Arbeitsspeicherzugriff (1/2)



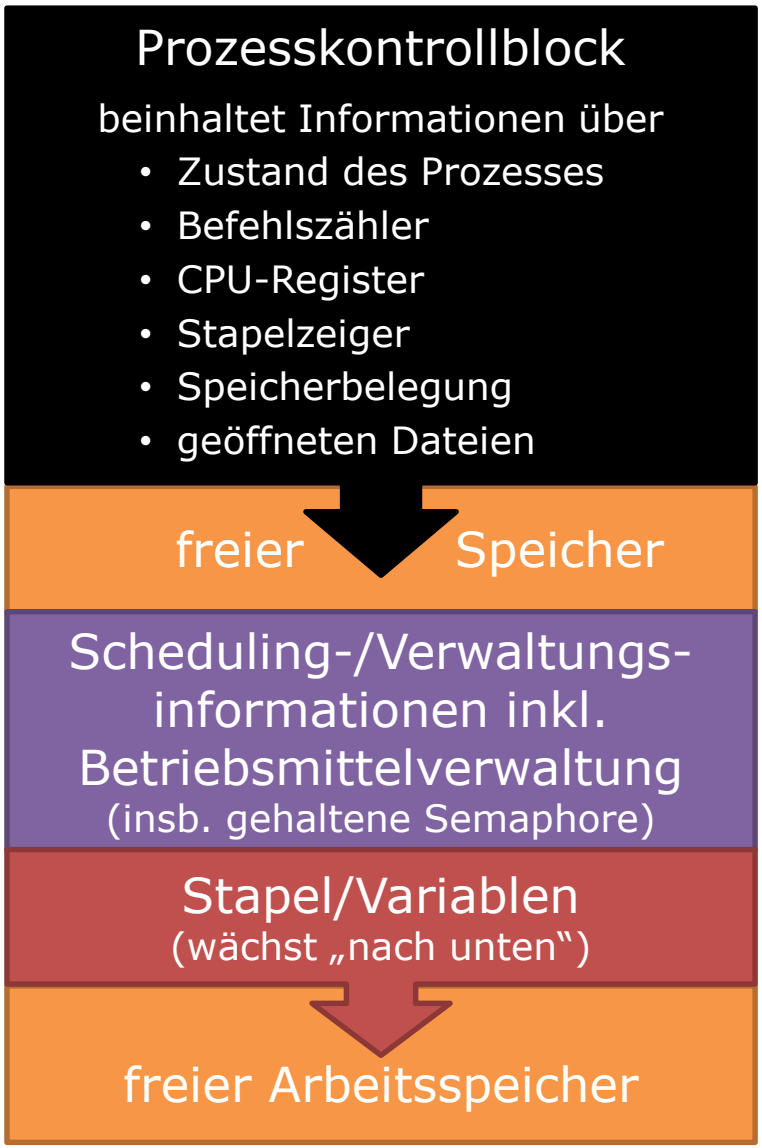
- Arbeitsspeicher kann nicht in einzelnen Bits adressiert werden
  - ⇢ Einteilung in Blöcke gleicher Größe
  - ⇢ Einteilung in Bereiche mit bestimmten Aufgaben
- Beispiel: 64-Bit-Architektur
  - Befehlsbreite 8 Bit
  - Operandenbreite 8 Bit bis 32 Bit
  - Adressbreite 32 Bit → maximal 4.294.967.296 Adressen adressierbar
  - ⇢ bei 16GB Arbeitsspeicher entspricht dies 32 Bit je Adresse

Arbeitsspeicherzugriff (2/2)



- Arbeitsspeicher wird in Bereiche eingeteilt, die jeder für sich eigene Adressierungsstrategien verfolgen
- feste Bereiche (Betriebssystemkern, Laufzeitumgebung)
- dynamisch wachsende/schrumpfende Bereiche (Prozesse, Laufzeitstapel)
- Bereiche dürfen sich nicht kreuzen
  → besonders heutzutage wegen ASLR u.s.w. problematisch
  → benötigt Verdrängungsstrategien, bspw. „LFU mit 80:20-Regel“ (mehr dazu im Modul 3IM-BERN-40)

Prozesse



- folgt ähnlichem Prinzip wie RAM
- fester Bereiche für die Kontrolle durch das Betriebssystem (Scheduling, Verwaltung, Semaphore)
- dynamisch wachsende/schrumpfende Bereiche (Kontrollblock, Stapel)
- Bereiche dürfen sich nicht kreuzen
  → problematisch mit Sprungbefehlen

  → benötigt Verdrängungs- und Erweiterungsstrategien