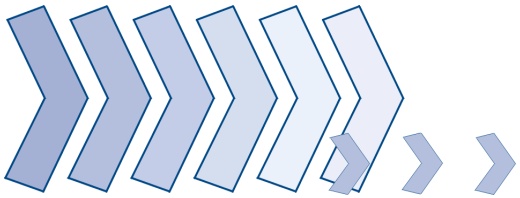
HUMBOLDT-UNIVERSITÄT ZU BERLIN



Digitale Systeme
Prozessoren

Dr.-Ing. Siegmar Sommer

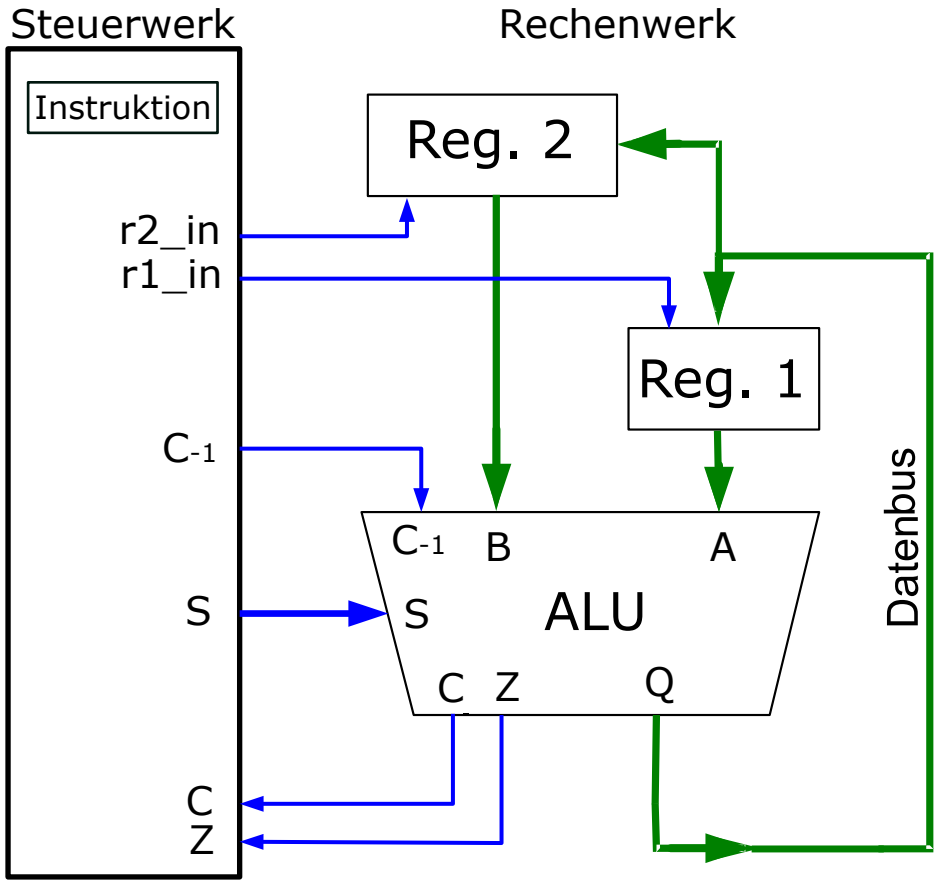
Sommersemester 2023

Prozessoren

- Bisher: Einzelne „Bausteine“ eines Rechners
  - ALU zum Durchführen von Berechnungen
  - Register zum Speichern einzelner Datenworte
  - Automaten zum Durchlaufen von Zustandsfolgen
  - adressierbare Speicher für größere Datenmengen
  - Busse
  - …
- Jetzt: Zusammensetzen von Komponenten und Festlegung ihrer Zusammenarbeit zur Ausführung von *Instruktionen*
- Instruktion (oder *Befehl*) : ein Bitmuster, das zur Erzeugung von Steuersignalen für die Komponenten eines Rechners, ausgewertet wird

Ein einfacher Prozessor

- Eine ALU und zwei Register werden zu einem *Rechenwerk* zusammen geschaltet
- Ein *Steuerwerk* beinhaltet kombinatorische und sequentielle Logik zur Steuerung der Abläufe im Prozessor → Erzeugung von Steuersignalen
- Die Steuersignale bestimmen
  - welche Operation die ALU durchführt (S, C_{-1})
  - ob das Rechenergebnis *Q* in das Register *R1* (*r1_in*) und/oder in das Register *R2* (*r2_in*) übernommen wird
- *C* (*Carry*)- und *Z (Zero*) - Signale signalisieren dem Steuerwerk Überlauf bzw. ein Null-Ergebnis
- Hier nicht gezeigt: Speicher und Ein-/Ausgabe sowie das Laden von Instruktionen in das Steuerwerk



Beispiel: Addition

- Die Instruktion bewirkt den Ablauf

  Reg.1 = Reg.1+Reg.2

- Dazu generiert das Steuerwerk die Signale

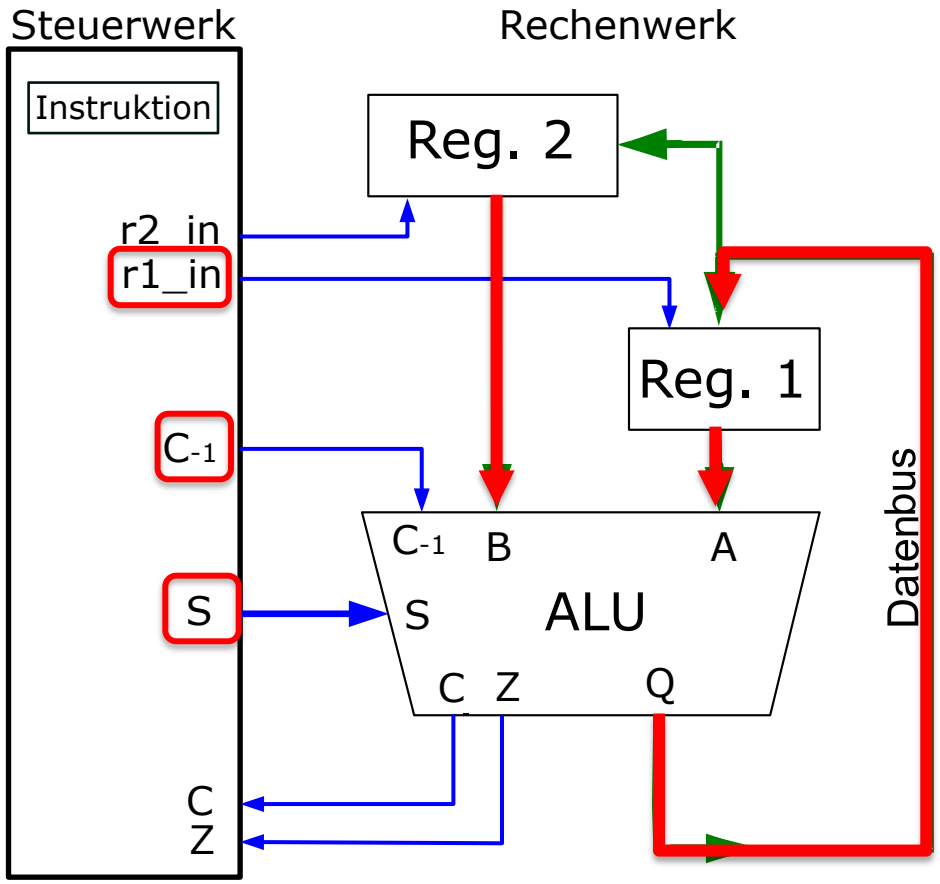
  C_{-1} = 0

  S = 00B (Festlegung: *Q=A+B*)

  r1_in = 1

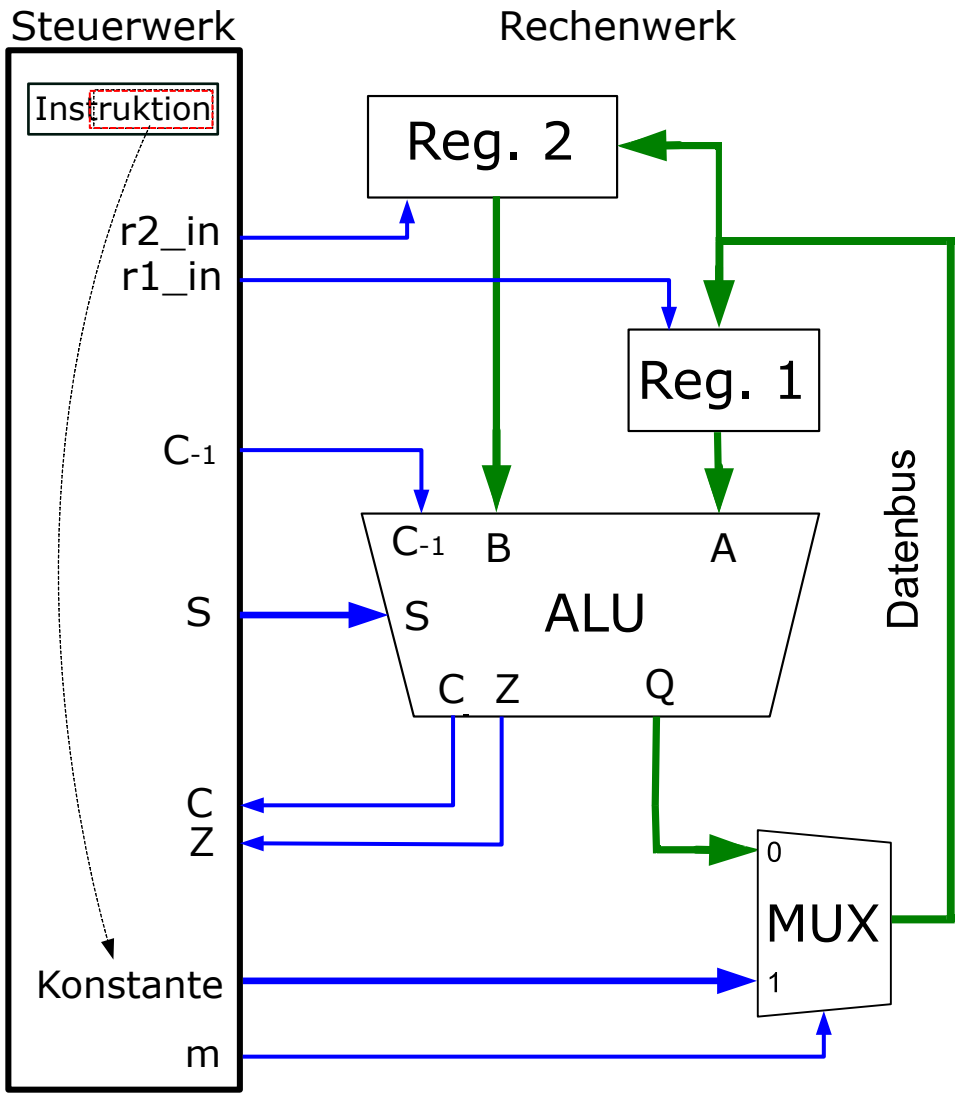
  r2_in = 0

- Hinweis: das scheinbar gleichzeitige Lesen und Schreiben des Registers 1 wird schaltungstechnisch realisiert: die Inhalte der Register sind ständig für die Operationen der ALU verfügbar, das Ergebnis Q wird allerdings erst durch eine Flanke von r1_in bzw. r2_in in eines der Register übernommen



Erweiterung

- Nun erfolgt eine Erweiterung, so dass eine Zahl (Konstante) in eines der Register geladen werden kann
- Die Konstante sei in der Instruktion enthalten
- Durch Einsatz eines Multiplexer kann, statt des ALU-Ergebnisses, die Konstante auf den Datenbus gelegt werden



Beispiel: Laden einer Konstanten

- Die Instruktion bewirkt den Ablauf

  Reg.1 = Konstante

- Dazu generiert das Steuerwerk die Signale

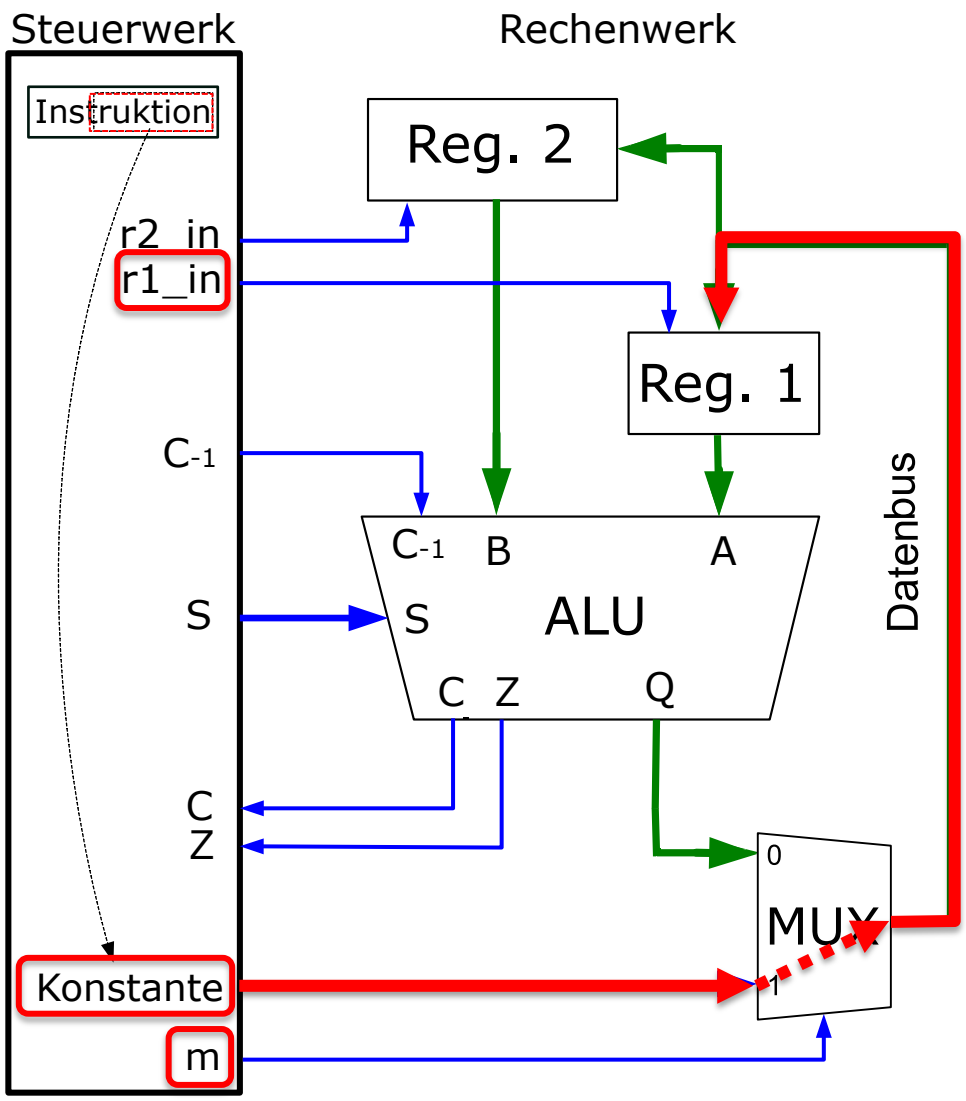
  C_{-1} = 0 (ist wirkungslos, daher eigentlich beliebig)

  S = 00B (ist wirkungslos, daher eigentlich beliebig)

  r1_in = 1

  r2_in = 0

  m = 1



Instruktionssätze

- Folgen von Steuersignalen werden durch Instruktionen (auch genannt: **Maschinenbefehle**) festgelegt
- Ein Prozessor kann eine – von einem Entwickler festgelegte – Menge von Instruktionen ausführen
- Diese Menge von möglichen Instruktionen nennt man **Instruktionssatzarchitektur** (Instruction Set Architecture, ISA)
- Um eine Instruktion auszuführen sind i.A. mehrere Teilschritte notwendig
- Eine **Steuereinheit** erzeugt aufgrund der Instruktion Folgen von Steuersignalen zur Ausführung der einzelnen Schritte

Steuereinheit

- Eine Steuereinheit muss
  - …feststellen, welches die nächste Instruktion im abzuarbeitenden Programm (Folge von Instruktionen) ist
  - …innerhalb einer instruktionsabhängigen Anzahl von Takten die entsprechenden Steuerleitungen im Prozessor aktivieren
- Eine Steuereinheit ist demnach ein Automat, der abhängig von der derzeit auszuführenden Instruktion eine Folge von Schritten abarbeitet

Steuereinheit

Zu klärende Fragen:

1. Wie werden (Folgen von) Instruktionen gespeichert, damit die Steuereinheit sie abarbeiten kann?
2. Wo werden die Instruktionen gespeichert und wie wird darauf zugegriffen?
3. Wie kann eine Steuereinheit aufgebaut sein?

Instruktionsformate

- Für die Steuereinheit muss aus der Instruktion ersichtlich sein,
  - *welche* Operation ausgeführt werden soll und
  - *mit welchen Daten* (**Operanden**)
- Deswegen besteht die Kodierung einer Instruktion (das *Befehlsformat*) üblicherweise aus
  - einem **Opcode**, der die Art der Operation angibt, und
  - einer codierten Liste von Operanden
- Die Ausdrucksmöglichkeiten für Operationen und Operanden (und damit die Komplexität des Instruktionen) sind von Prozessor zu Prozessor unterschiedlich

Beispielbefehl des Z80

- Beispielsweise kennt der 8-Bit-Prozessor Z80 der ehemaligen Fa. Zilog einen Befehl, der den Inhalt des Registers *C* in das Register *B* kopiert
- Dieser Befehl ist 1 Byte lang und lautet binär codiert

01 000 001

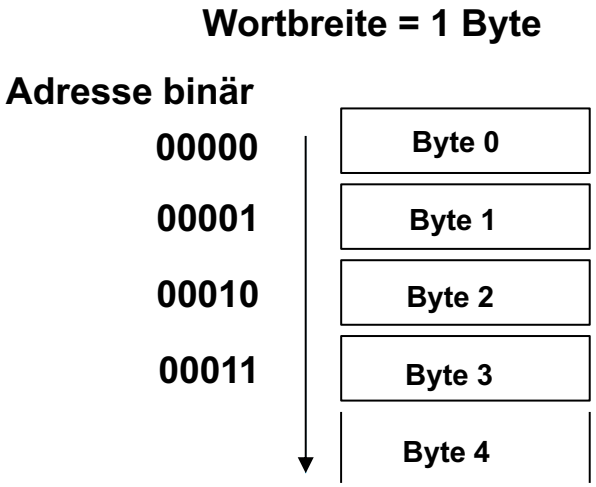
- Bedeutung der Bits:
  - 01: Opcode „LD“ - Lade Operand 1 mit Operand 2
  - 000: Operand 1 – Inhalt von Register B (*000* ist Nummer (Adresse) des Registers B)
  - 001: Operand 2 - Inhalt von Register C (*001* ist Nummer (Adresse) des Registers C)

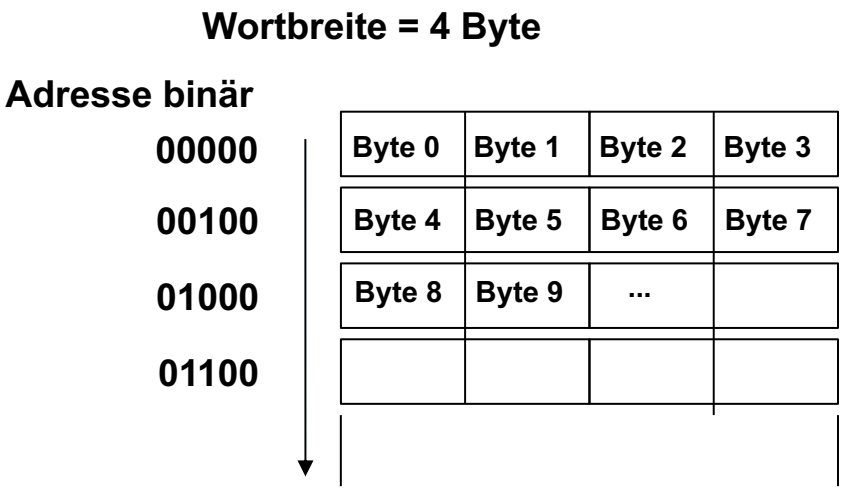
Länge der kodierten Instruktionen

- Die Länge der Instruktion ist abhängig von der Anzahl auszuführender Befehle, Steuerinformationen und Länge der Operanden
- Bei variabler Instruktionslänge werden häufig vorkommende Instruktionen mit einer kleineren Zahl von Bits kodiert
- Feste Instruktionslängen bieten weniger Flexibilität – vereinfachen dafür aber die Steuereinheit
- Oft bei Großrechnern: Breite einer Speicherzelle entspricht Anzahl der Bits des Befehls (=Wortbreite)
- Weit verbreitet: Befehle werden in Vielfachen von Byte organisiert und in einer oder aufeinanderfolgenden Speicherzellen eines wort-breiten Speichers gespeichert
  - Wort: $2n$ Byte ; $1<=n<=4$
- Die Wortbreite ist oft gleichzeitig die Verarbeitungsbreite des Prozessors (Breite der Register und der ALU)

Organisation von Instruktionen und Daten im Speicher

- Die kleinste logisch adressierbare Datenmenge ist üblicher Weise 1 Byte
- Speicher wird meist wortweise adressiert





Instruktionsformate

Befehl ohne Operand:

Art des Befehls	Steuerinformationen

Befehl mit Operanden:

Art des Befehls	Steuerinform.	Operanden

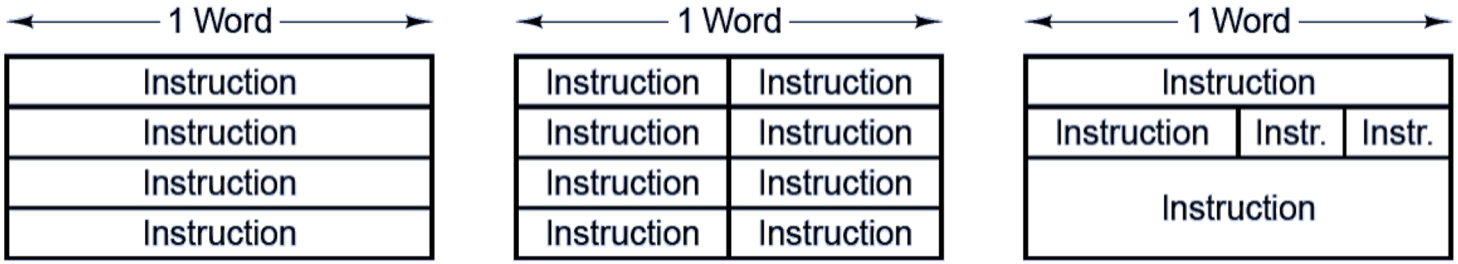
Wort-organisierter Befehl mit Operand:

Art des Befehls	Steuerinformationen
Operand	

Wort-organisierter Befehl mit mehreren Operanden:

Art des Befehls	Steuerinf.	Operand1
Operand2		
Operand3		

Varianten:

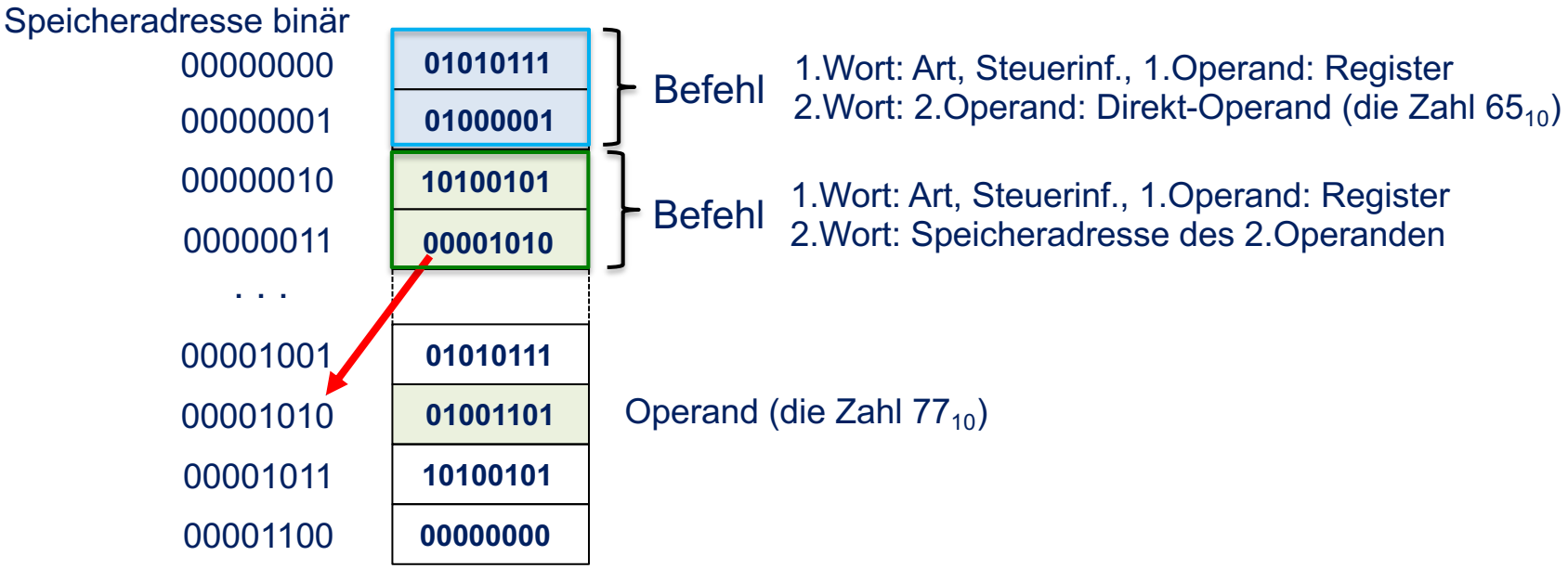


Operanden

- Operanden können von unterschiedlichem Typ sein
  - Inhalte von Registern - die Nummer des Registers ist im Befehl enthalten
  - feste Werte (sog. *Immediate*- oder Direkt-Operanden) - der Operand ist im Befehl enthalten
  - Inhalte von Speicherstellen - die Adresse der Speicherzelle ist im Befehl enthalten
  - verschiedene Varianten und Kombinationen davon
- Operanden können entweder in der kodierten Instruktion angegeben sein (wie beim Z80-Beispiel gesehen) oder sie können für eine bestimmte Operation fest vorgegeben sein
  - beispielsweise könnte es sein, dass eine bestimmte Operation immer mit demselben, festgelegten Register ausgeführt wird (dann muss dieses Register in der kodierten Instruktion nicht angegeben werden)
- Welche Kombinationen von Operanden für eine bestimmte Operation möglich sind, legt die ISA fest

Adressierung von Operanden

- Hier beispielhaft dargestellt (wird später noch genauer behandelt):
  - in der Instruktion 1 ist ein sogenannter Direktoperand enthalten (Operand ist eine „Konstante“)
  - in der Instruktion 2 ist die Speicheradresse des Operanden enthalten (Operand ist eine „Variable“ (falls der Speicher beschreibbar ist))

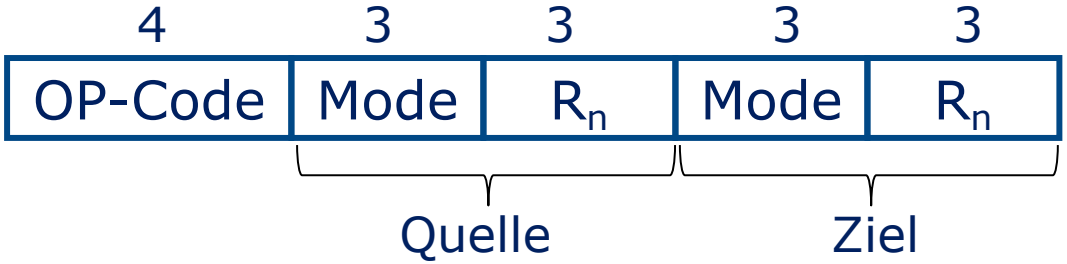


Beispiel: Instruktionsformat DEC PDP-11

1-Adress-Format:

Mode: Typ des Operanden
R_n: Register n

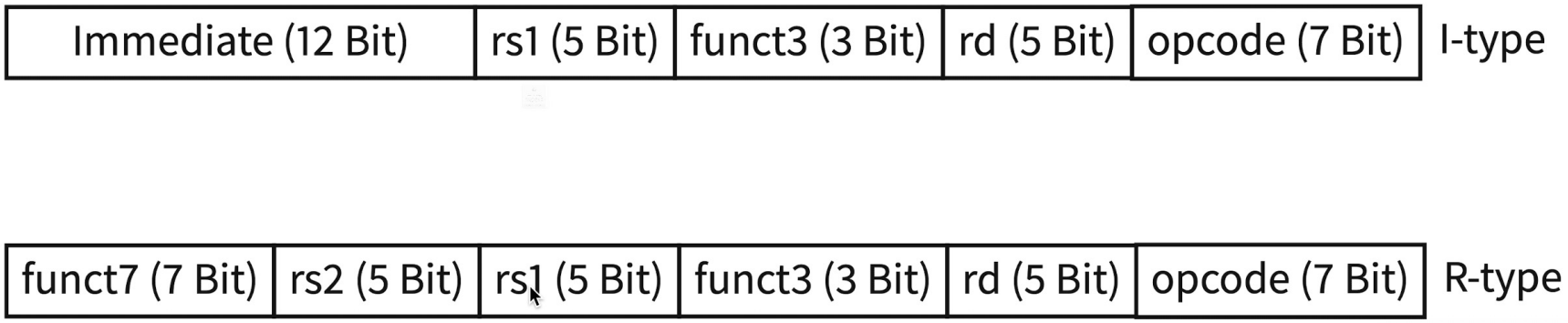
2-Adress-Format:



Sprung:

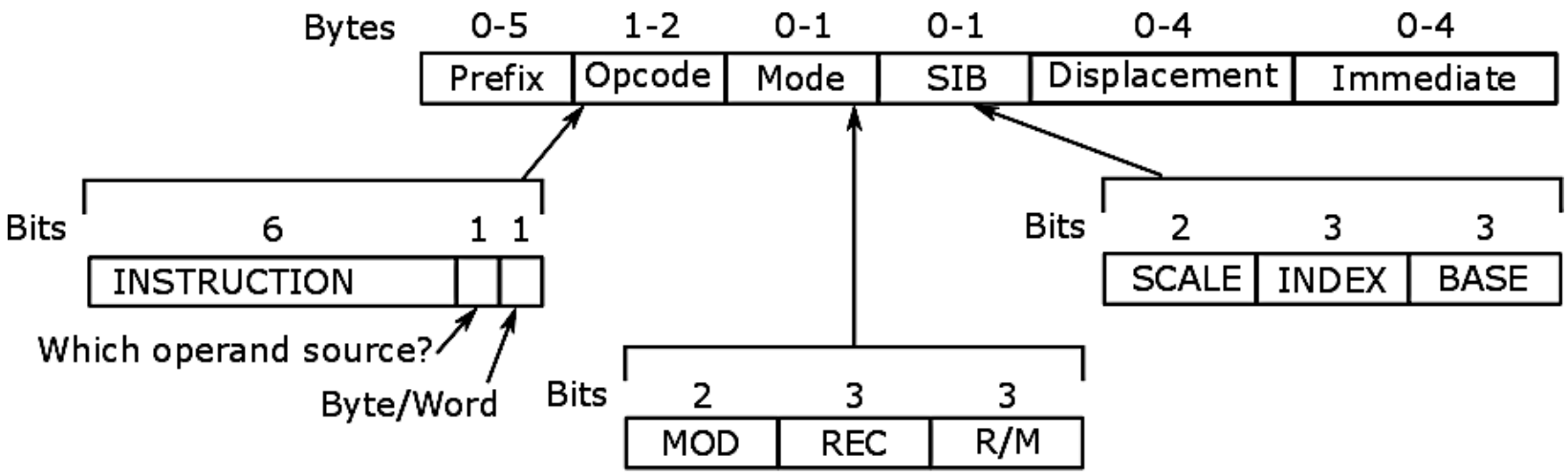
Offset: Anzahl der zu überspringenden Befehlsworte

Beispiel: RISC-V Instruktionsformat



- *rs1*, *rs2* geben Quellregister an, *rd* das Zielregister (es gibt 32)
- *opcode*, *funct3*, *funct7* kodieren den Befehl
- *I-type*: Integer Register-Immediate Instruktionen
  - z. B *ADDI rd, rs1, Immediate*
- *R-type*: Integer Register- Register Instruktionen
  - z. B. *AND rd, rs1, rs2*
- Weitere Formate sind z. B. *B-type* (Verzweigungen), *J-type* (unbedingte Sprünge)

Beispiel: Instruktionsformat Intel Pentium 4



- Prefix: Modifikation der Wirkung des Opcodes
- Opcode
  - Byte/Word: Länge der Operanden
  - Source: legt fest, welcher Operand Quelle ist
- Mode: Beschreibung der Operanden + Erweiterung durch SIB
  - REC: Register (oder Opcode-Teile), MOD: Adressierungsart für R/M, M/R: Register- oder Speicher-Operanden im Zusammenhang mit SIB
- Unübersichtlich da mit den Prozessorgenerationen erweiterte Struktur

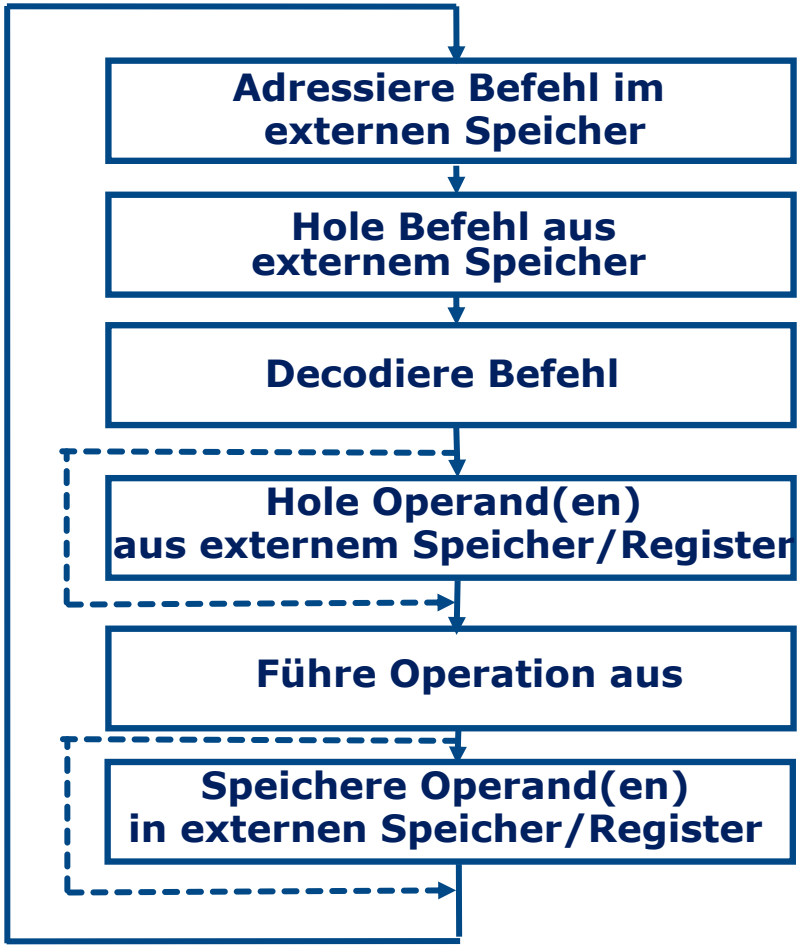
Steuerwerk

- Aufgabe: Lesen einer Instruktion, Decodierung der Instruktion und Erzeugung von Folgen von Steuersignalkombinationen (Steuersignalworten) zur Ausführung der Instruktion; Bestimmung der Lage der nächsten Instruktion im Programmspeicher
- Für den Entwurf eines Steuerwerks (Steuereinheit) muss Folgendes festliegen:
  - Anzahl der benötigten Register
  - benötigte Funktionsgruppen
  - Anzahl der Worte je Instruktion
  - Anzahl und Art der einzelnen Schritte zur Ausführung der Instruktion

Struktur eines Steuerwerks

- Zu verwaltende Informationen:
  - Adresse des Speicherwortes, in dem die nächste auszuführende Instruktion abgelegt ist
  - diese Adresse wird in einem (Spezial-) Register, genannt **Instruction Pointer** (**IP**) oder auch (eigentlich unzutreffend) **Program Counter** (**PC**)
  - des Weiteren wird ein Spezialregister (**Instruction Register**, **IR**) verwendet, das die zurzeit ausgeführte Instruktion beinhaltet
- Ein Automat organisiert Folgendes
  1. Lesen der Instruktion aus der Speicherzelle die durch den Inhalt des *IP* adressiert wird und kopieren der Instruktion in das *IR*
  2. Erhöhen des *IP* um eine Wortlänge
  3. Aktivieren von Steuersignalen in Abhängigkeit der im *IR* enthaltenen Bitmuster (Bitmuster = Instruktion)

CPU-Verhalten dargestellt als geschlossener Kreislauf



Schritt 1

Schritt 2

Schritt 3

Schritt 4

Schritt 5

Schritt 6

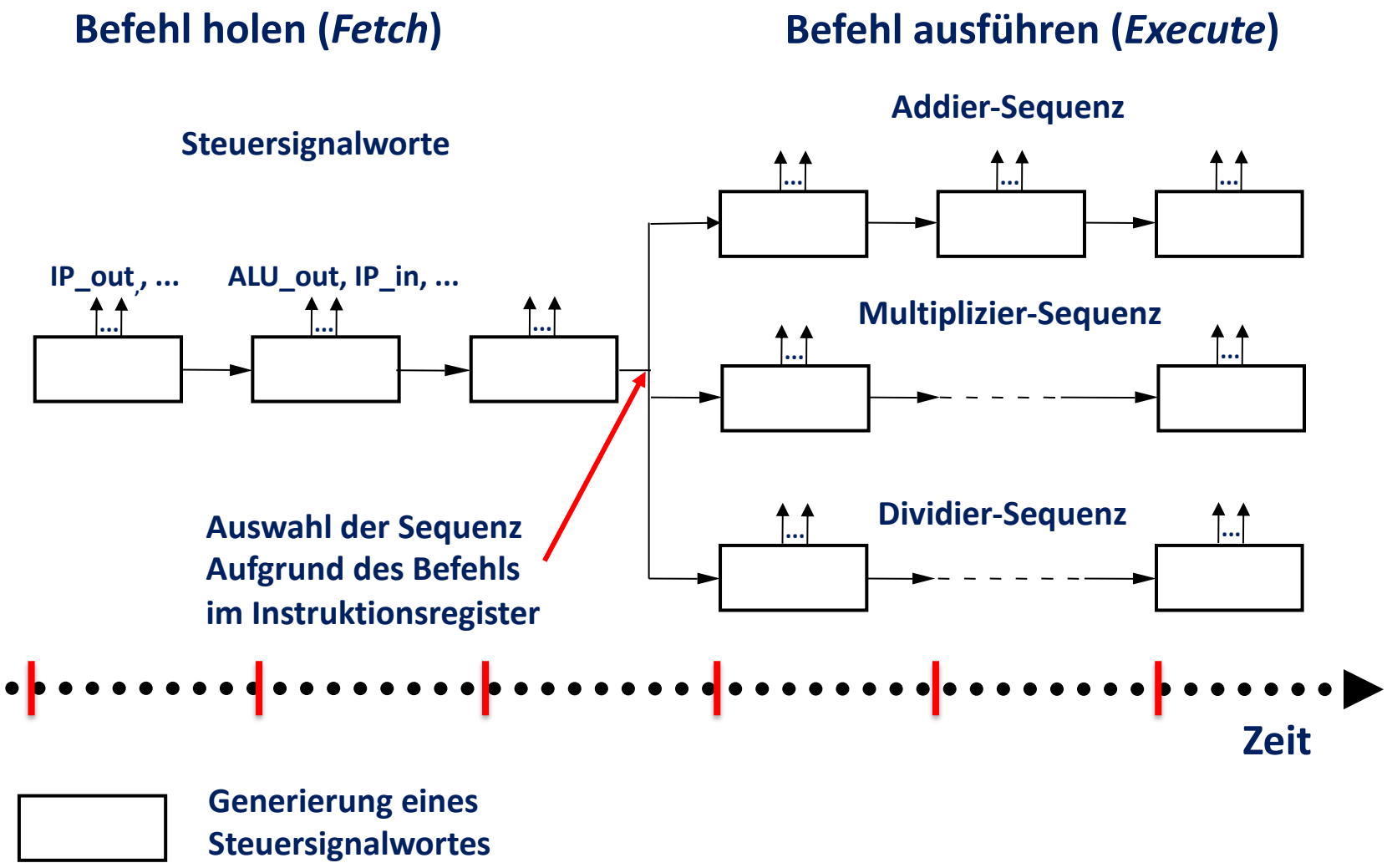
Fetch

Decode

Execute

externer Speicher: oft *Hauptspeicher* genannt, manchmal getrennt in Programmspeicher und Datenspeicher

Erzeugung von Folgen von Steuersignalworten



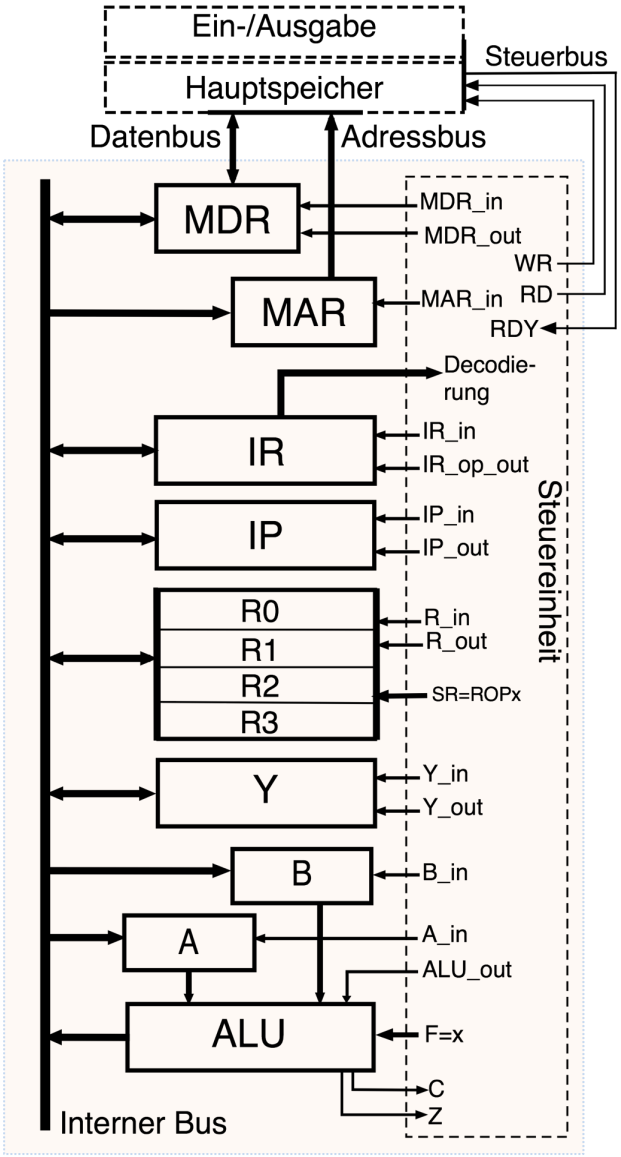
Komplexere Beispielarchitektur

- Für eine weitere Prozessorarchitektur wird folgendes 1-Wort-Befehlsformat angenommen:

- *ROP1*: Adresse des ersten Registeroperanden
- *ROP2*: Adresse des zweiten Registeroperanden
- *Operand3*: Bitmuster, ist durch das Signal *IRop_out* aus dem IR entnehmbar und kann z.B. als Speicheradresse, Sprungoffset oder Konstante verwendet werden

Komplexere Beispielarchitektur

- Register IP und IR: gehören zur Steuereinheit
- Register MAR (Memory Address Register) und MDR (Memory Data Register) unterstützen Speicherzugriffe
- R0...R3: Allzweckregister, Y: Hilfsregister
- A, B: Eingangsregister der ALU
- Signale:
  - x_in: übernimmt Wert vom Bus ins Register
  - x_out: legt (Register-) Inhalt auf den Bus
  - RD/WR: RD – Speicher/EA lesen, WR - ... schreiben
  - RDY: Speicher signalisiert abgeschlossenen Zugriff
  - F: ALU-Funktion auswählen (z.B. +, -, OR, ...)
  - SR: Register auswählen (Adresse des Registeroperanden (ROPx) in Instruktion, x abhängig davon ob 1. oder 2. Operand)

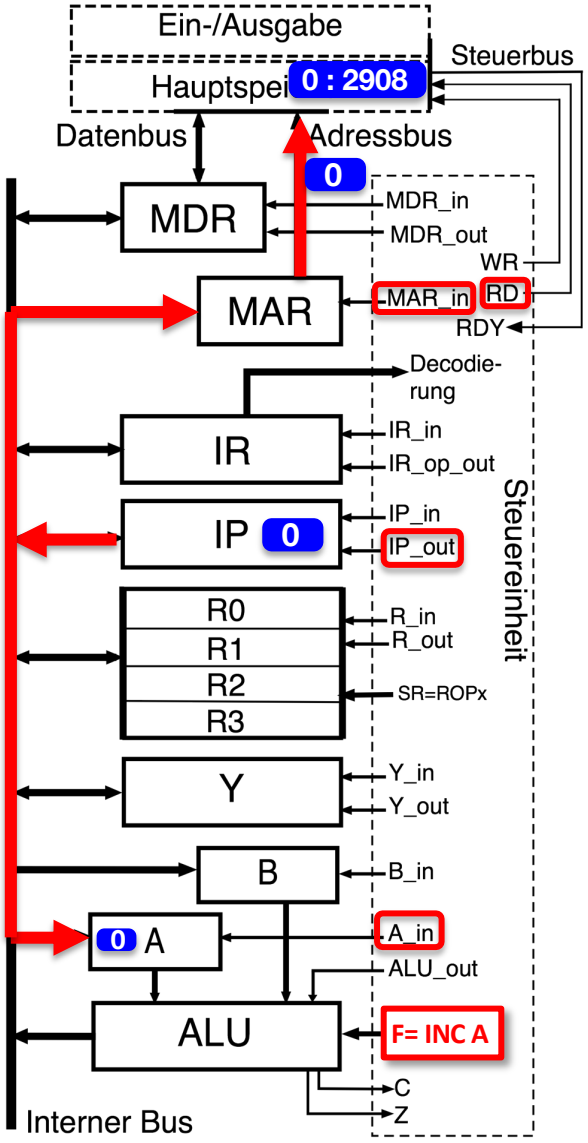


Steuersequenz zum Holen eines Befehls

- Hauptspeicherzugriffe benötigen i.A. mehr Zeit als 1 Schritt der Steuereinheit
- Deshalb: Abarbeitung der Steuereinheit wird blockiert, bis der Speicher das *RDY*-Signal aktiviert
- Aktivierung des Wartens mit Beginn des nächsten Schritts durch das Steuersignal *WRDY* - Wait for memory ReaDY (internes Signal der Steuereinheit)
- Das Signal *DECODE* leitet die Dekodierung (und damit die Ausführung) des Befehls ein

Schritt | aktive Steuersignale (Steuersignalwort)

1. IP_out, MAR_in, RD, A_in, F = INC A
2. ALU_out, IP_in, WRDY
3. MDR_out, IR_in, DECODE

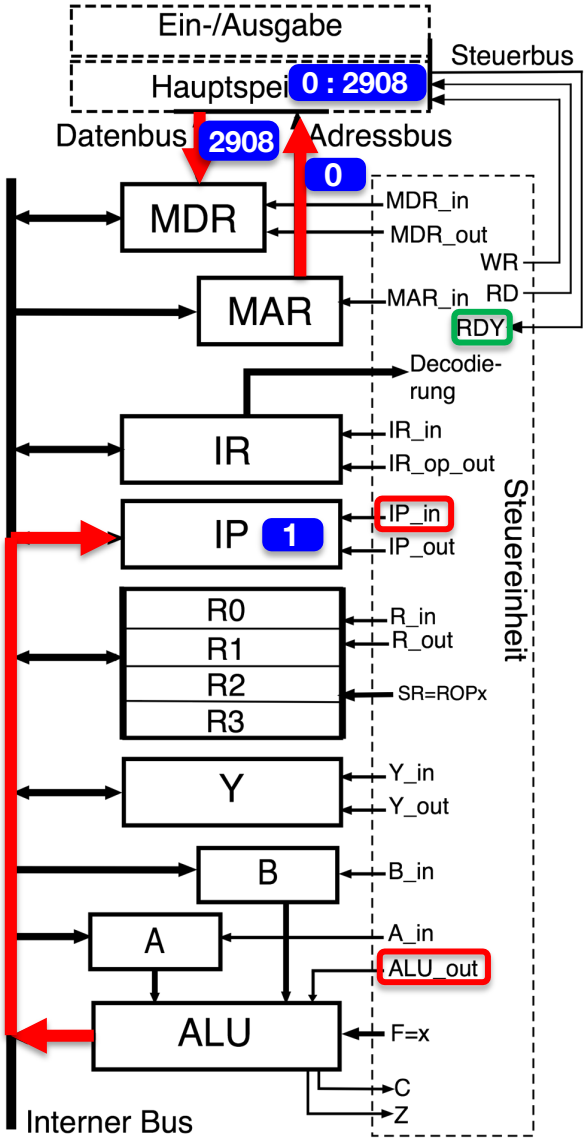


Steuersequenz zum Holen eines Befehls

- Hauptspeicherzugriffe benötigen i.A. mehr Zeit als 1 Schritt der Steuereinheit
- Deshalb: Abarbeitung der Steuereinheit wird blockiert, bis der Speicher das *RDY*-Signal aktiviert
- Aktivierung des Wartens mit Beginn des nächsten Schritts durch das Steuersignal *WRDY* - Wait for memory ReaDY (internes Signal der Steuereinheit)
- Das Signal *DECODE* leitet die Dekodierung (und damit die Ausführung) des Befehls ein

Schritt | aktive Steuersignale (Steuersignalwort)

1. IP_out, MAR_in, RD, A_in, F = INC A
2. ALU_out, IP_in, WRDY (dieser Schritt wird im Zeitschatten des Speicherzugriffs ausgeführt)
3. MDR_out, IR_in, DECODE

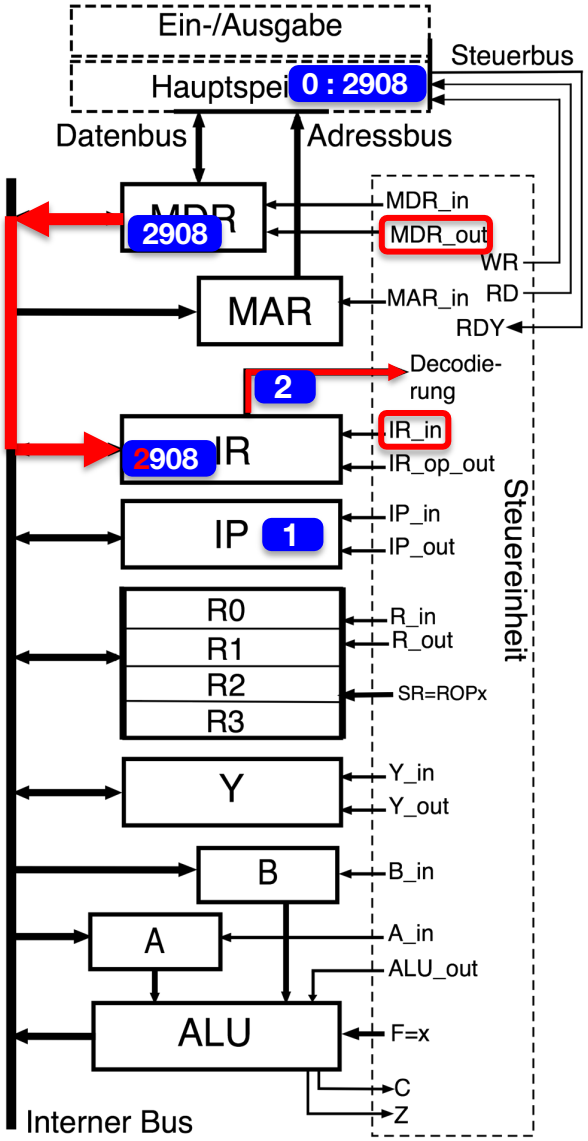


Steuersequenz zum Holen eines Befehls

- Hauptspeicherzugriffe benötigen i.A. mehr Zeit als 1 Schritt der Steuereinheit
- Deshalb: Abarbeitung der Steuereinheit wird blockiert, bis der Speicher das *RDY*-Signal aktiviert
- Aktivierung des Wartens mit Beginn des nächsten Schritts durch das Steuersignal *WRDY* - Wait for memory ReaDY (internes Signal der Steuereinheit)
- Das Signal *DECODE* leitet die Dekodierung (und damit die Ausführung) des Befehls ein

Schritt | aktive Steuersignale (Steuersignalwort)

1. IP_out, MAR_in, RD, A_in, F = INC A
2. ALU_out, IP_in, WRDY
3. MDR_out, IR_in, DECODE (dieser Schritt wird erst ausgeführt, wenn der Hauptspeicher *RDY* aktiviert hat)



Ausführen des Befehls

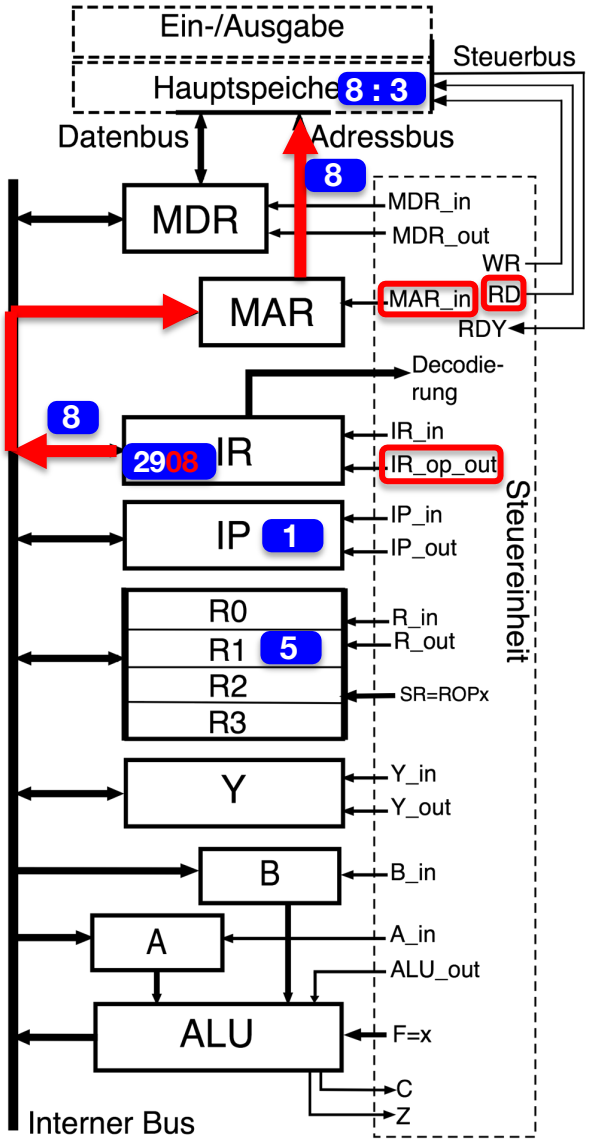
- Jetzt befindet sich der Befehl im *IR*, hier als Beispiel ein 16-Bit-Maschinenbefehl

Art des Befehls	ROP1	ROP2	Operand3
0 0 1 0	1 0	0 1	0 0 0 0 1 0 0 0

- Angenommen, es handelt sich um den Befehl
  ADD R2, R1, 8 *mit der Syntax: ROP1 = ROP2 + MEM*
- Der Befehl addiert zu *ROP2* (*R1*) den Inhalt der Speicherstelle mit der Adresse *8* und legt das Ergebnis in *ROP1* (*R2*) ab
- Die Speicheradresse von Operand3 ist ein Teil des Befehls im *IR* und kann mit dem Signal *IR_op_out* auf den Bus gelegt werden

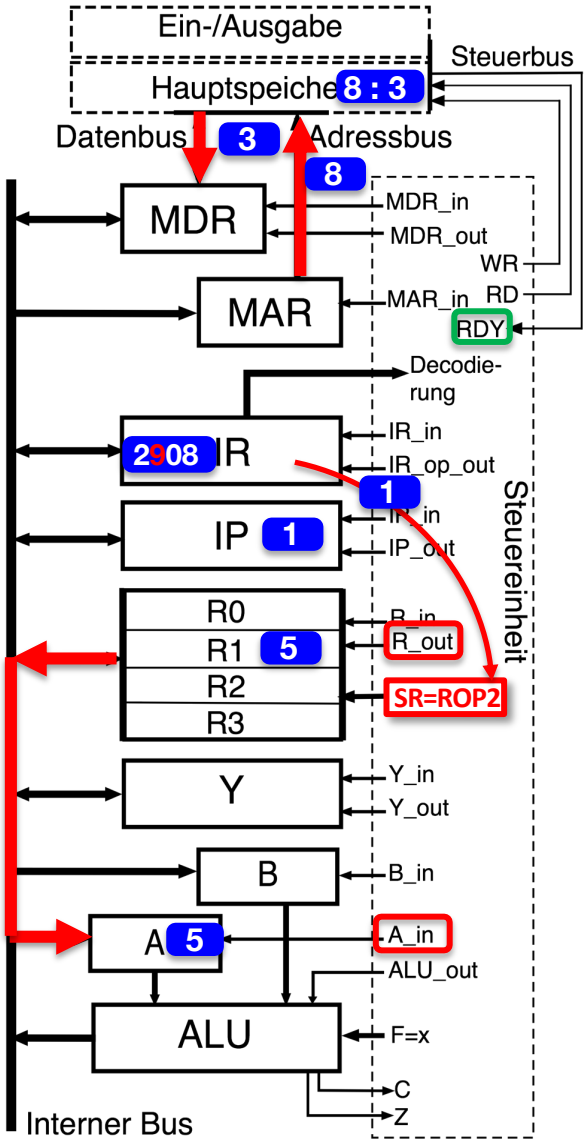
Schritt | aktive Steuersignale (Steuersignalwort)

4. IR_op_out, MAR_in, RD
5. SR=ROP2, R_out, A_in, WRDY
6. MDR_out , B_in, F=A+B
7. ALU_out , SR=ROP1, R_in



Ausführen des Befehls

- Jetzt befindet sich der Befehl im *IR*
- Angenommen, es handelt sich um den Befehl
  ADD R2, R1, 8
- *mit der Syntax: ROP1 = ROP2 + MEM*
- Der Befehl addiert zu *ROP2* (*R1*) den Inhalt der Speicherstelle mit der Adresse *8* und legt das Ergebnis in *ROP1* (*R2*) ab
- Die Speicheradresse von Operand3 ist ein Teil des Befehls im *IR* und kann mit dem Signal *IR_op_out* auf den Bus gelegt werden



Schritt |aktive Steuersignale (Steuersignalwort)

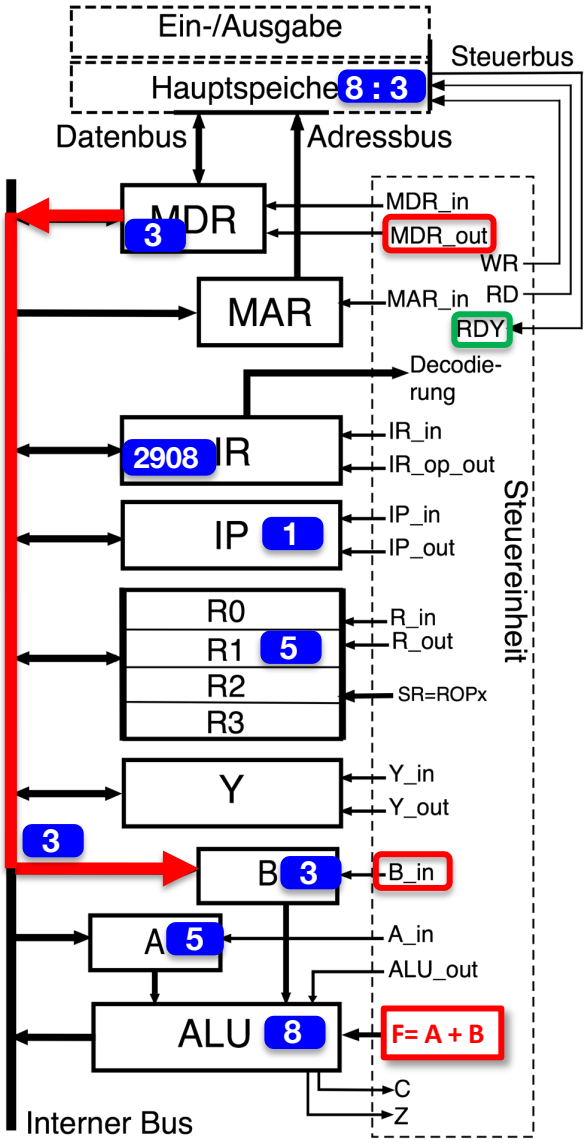
4. IR_op_out, MAR_in, RD
5. SR=ROP2, R_out, A_in, WRDY
6. MDR_out , B_in, F=A+B
7. ALU_out , SR=ROP1, R_in

Ausführen des Befehls

- Jetzt befindet sich der Befehl im *IR*
- Angenommen, es handelt sich um den Befehl
  ADD R2, R1, 8
- *mit der Syntax: ROP1 = ROP2 + MEM*
- Der Befehl addiert zu *ROP2* (*R1*) den Inhalt der Speicherstelle mit der Adresse *8* und legt das Ergebnis in *ROP1* (*R2*) ab
- Die Speicheradresse von Operand3 ist ein Teil des Befehls im *IR* und kann mit dem Signal *IR_op_out* auf den Bus gelegt werden

Schritt | aktive Steuersignale (Steuersignalwort)

4. IR_op_out, MAR_in, RD
5. SR=ROP2, R_out, A_in, WRDY
6. MDR_out , B_in, F=A+B
7. ALU_out , SR=ROP1, R_in

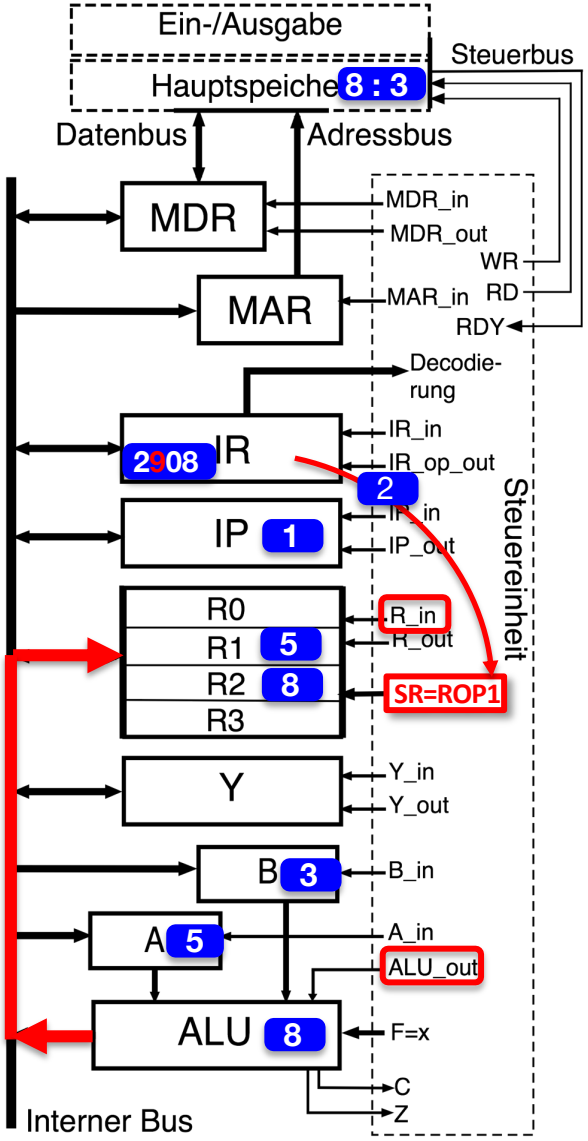


Ausführen des Befehls

- Jetzt befindet sich der Befehl im *IR*
- Angenommen, es handelt sich um den Befehl
  ADD R2, R1, 8
- *mit der Syntax: ROP1 = ROP2 + MEM*
- Der Befehl addiert zu *ROP2* (*R1*) den Inhalt der Speicherstelle mit der Adresse *8* und legt das Ergebnis in *ROP1* (*R2*) ab
- Die Speicheradresse von Operand3 ist ein Teil des Befehls im *IR* und kann mit dem Signal *IR_op_out* auf den Bus gelegt werden

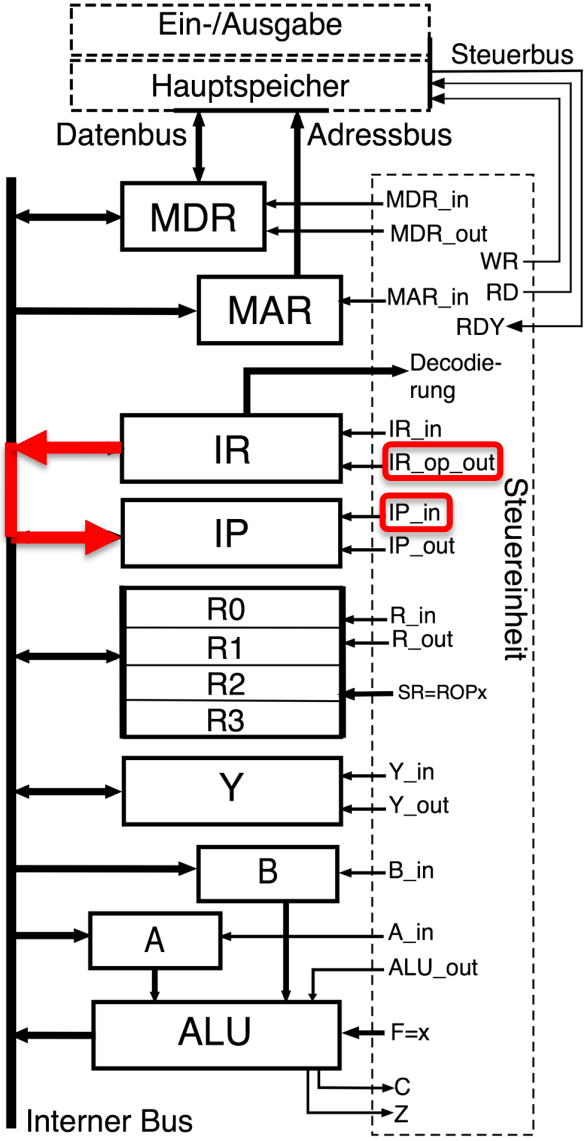
Schritt | aktive Steuersignale (Steuersignalwort)

4. IR_op_out, MAR_in, RD
5. SR=ROP2, R_out, A_in, WRDY
6. MDR_out , B_in, F=A+B
7. ALU_out , SR=ROP1, R_in



Ausführen von Sprungbefehlen

- Bisher: Anweisungen werden genau einmal in genau der Reihenfolge abgearbeitet, wie sie im Speicher stehen
- Einen wesentlichen Bestandteil von Programmen stellen jedoch Sprünge, Schleifen und Verzweigungen dar
- Ausführung eines Sprungs:
  - Sprung = überschreiben des Inhalts (=Adresse) des Instruction Pointer (IP) mit einem neuen Wert
  - Der nächste auszuführende Befehl wird von dieser neuen Adresse geholt
  - Steuersignalwort dafür „*IR_op_out, IP_in*“, wobei die Sprungzieladresse im Befehlswort enthalten ist (daher *IR_op_out*)
  - das Sprungziel könnte ebenso in einem extra Wort des Befehls enthalten sein, welches dann noch geladen werden müsste



Ausführen von Sprungbefehlen

- Ein wichtiges Mittel zur Steuerung des Programmflusses: **bedingte** Sprünge, wie z.B.:

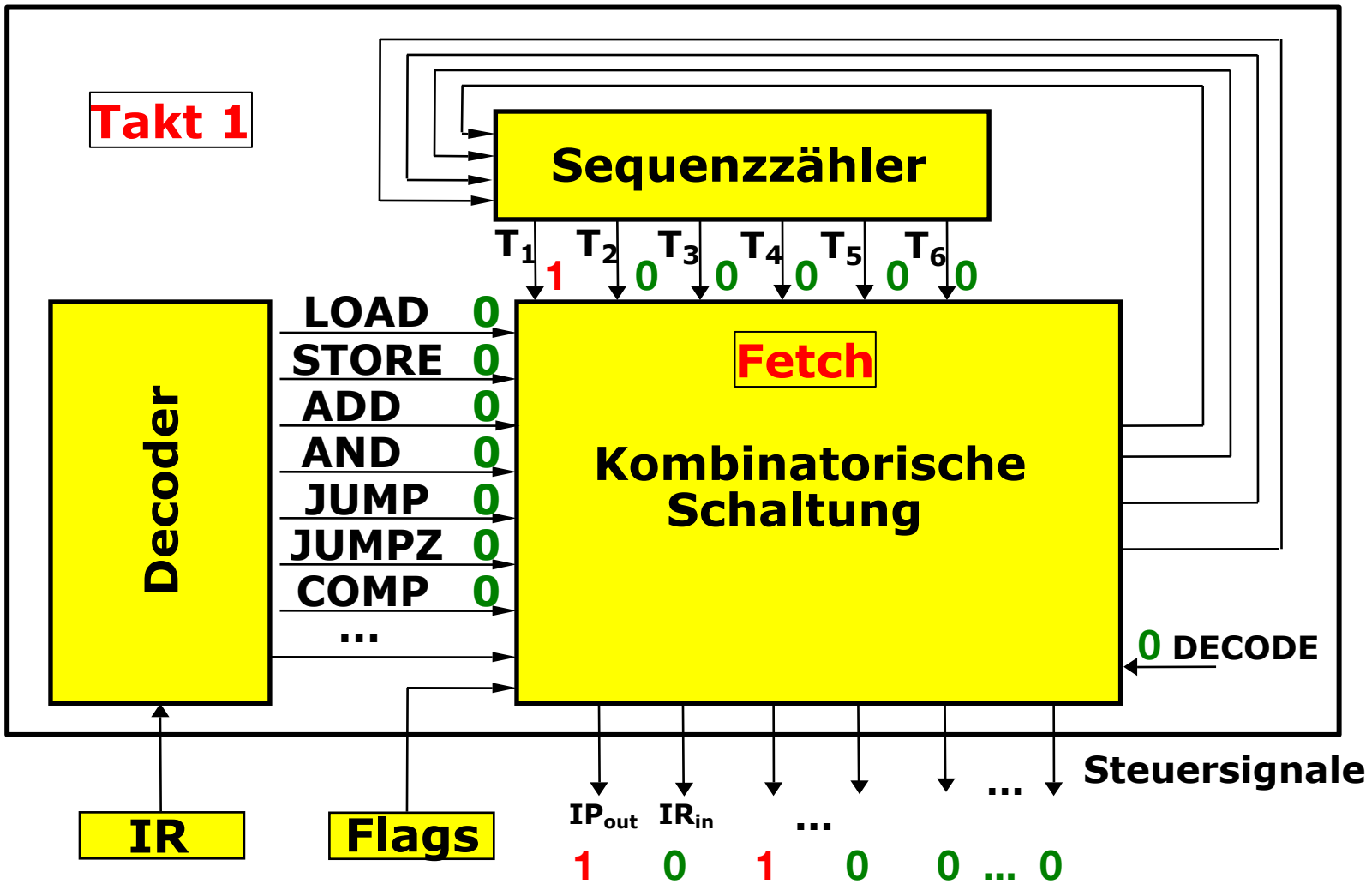
  Verzweige im Programmfluss (Sprung) zu Befehl x, falls R1==R2

- Zunächst Ausführung des Tests auf „R1==R2“
- Abhängig vom Ergebnis des Tests werden bestimmte Flags gesetzt
  - siehe ALU weiter vorne: das Z-Flag wird gesetzt, wenn das Ergebnis der Operation gleich 0 ist
- Bedingter Sprung = Übernehmen der neue Adresse in den IP, falls ein bestimmtes Flag gesetzt ist
- Dafür ist zusätzliche Hardware im Kontroll- und Datenpfad notwendig
- Im Beispiel also zwei Schritte:
  1. Subtrahiere R1-R2 (ohne ein Register zu verändern!)
  2. Kopiere Sprungzieladresse in IP, falls das Z-Flag gesetzt ist

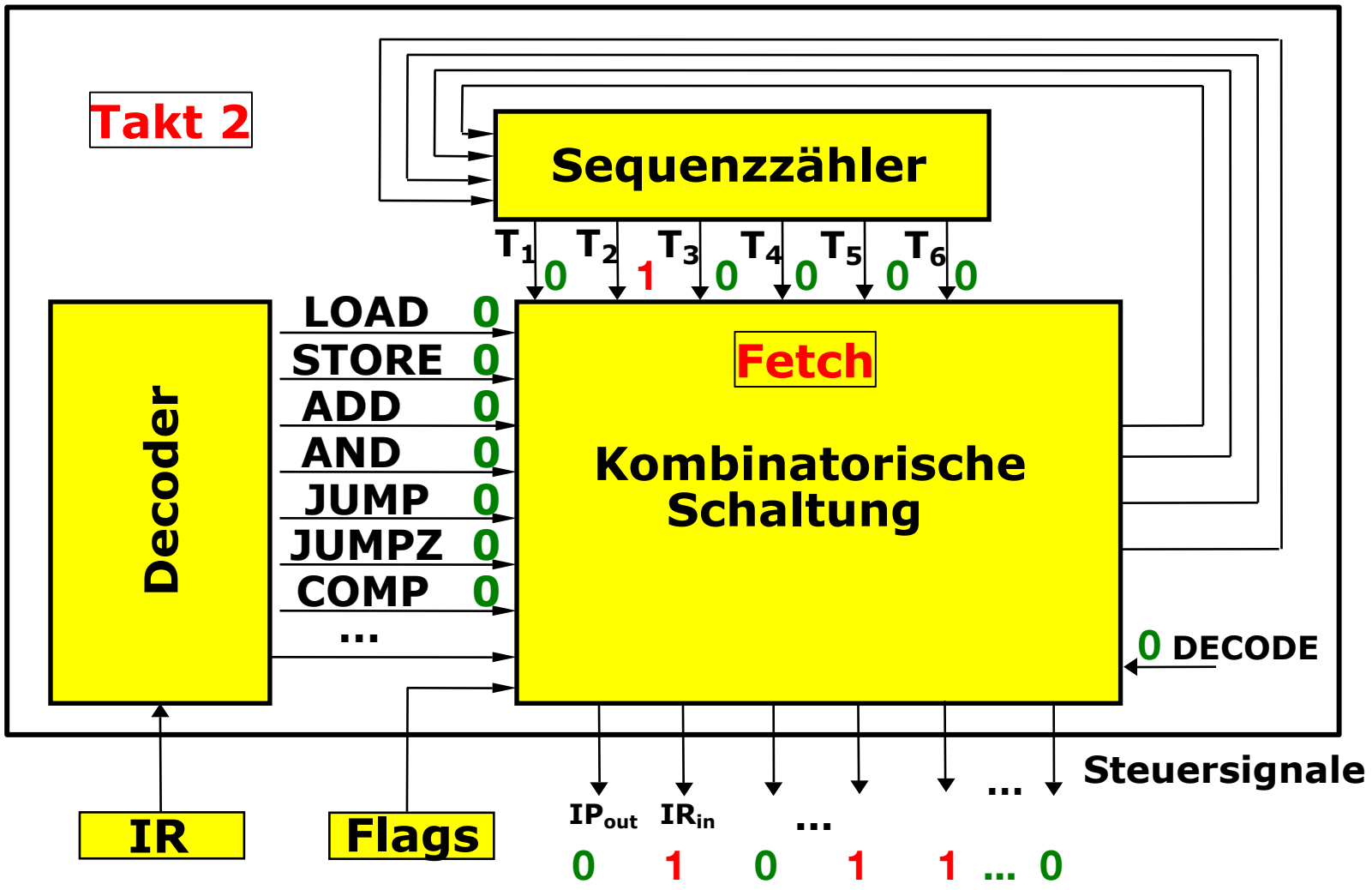
Konstruktionsprinzipien für Steuereinheiten

- Bisherige Betrachtung: die Steuereinheit ist fest „verdrahtet“
- ... ist ein festes Schaltwerk, das die Funktion des Automaten zum Erzeugen der Sequenz von Steuersignalworten erfüllt
- Für einfache Steuerwerke ist dies durchaus ein gangbarer Weg
  - gut optimierbar
  - wenige Bauelemente
  - kurze Signallaufzeiten
- Für komplexere Instruktionssätze kann ein solcher Entwurf jedoch sehr aufwändig werden

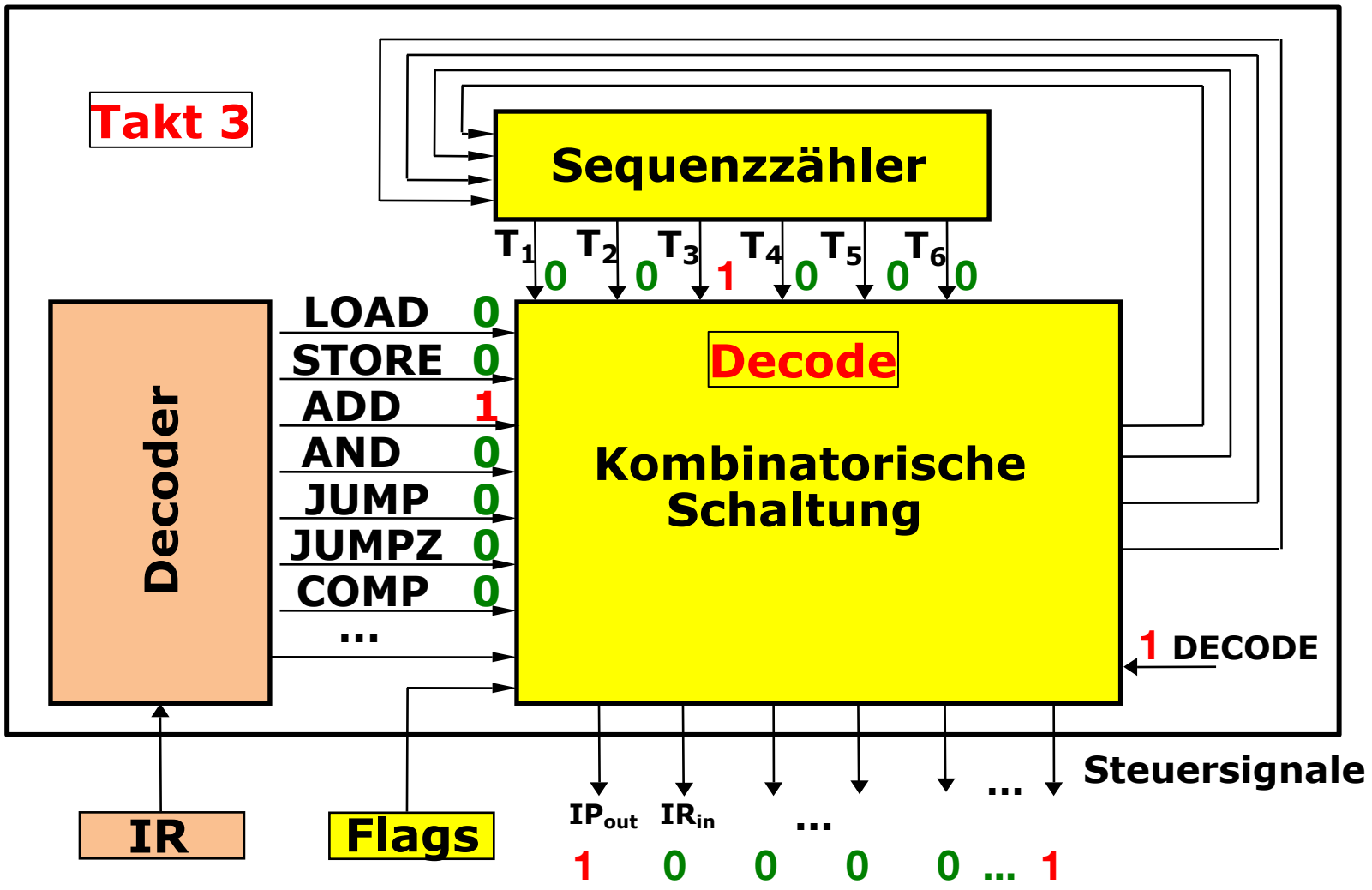
Funktionsweise einer Steuereinheit



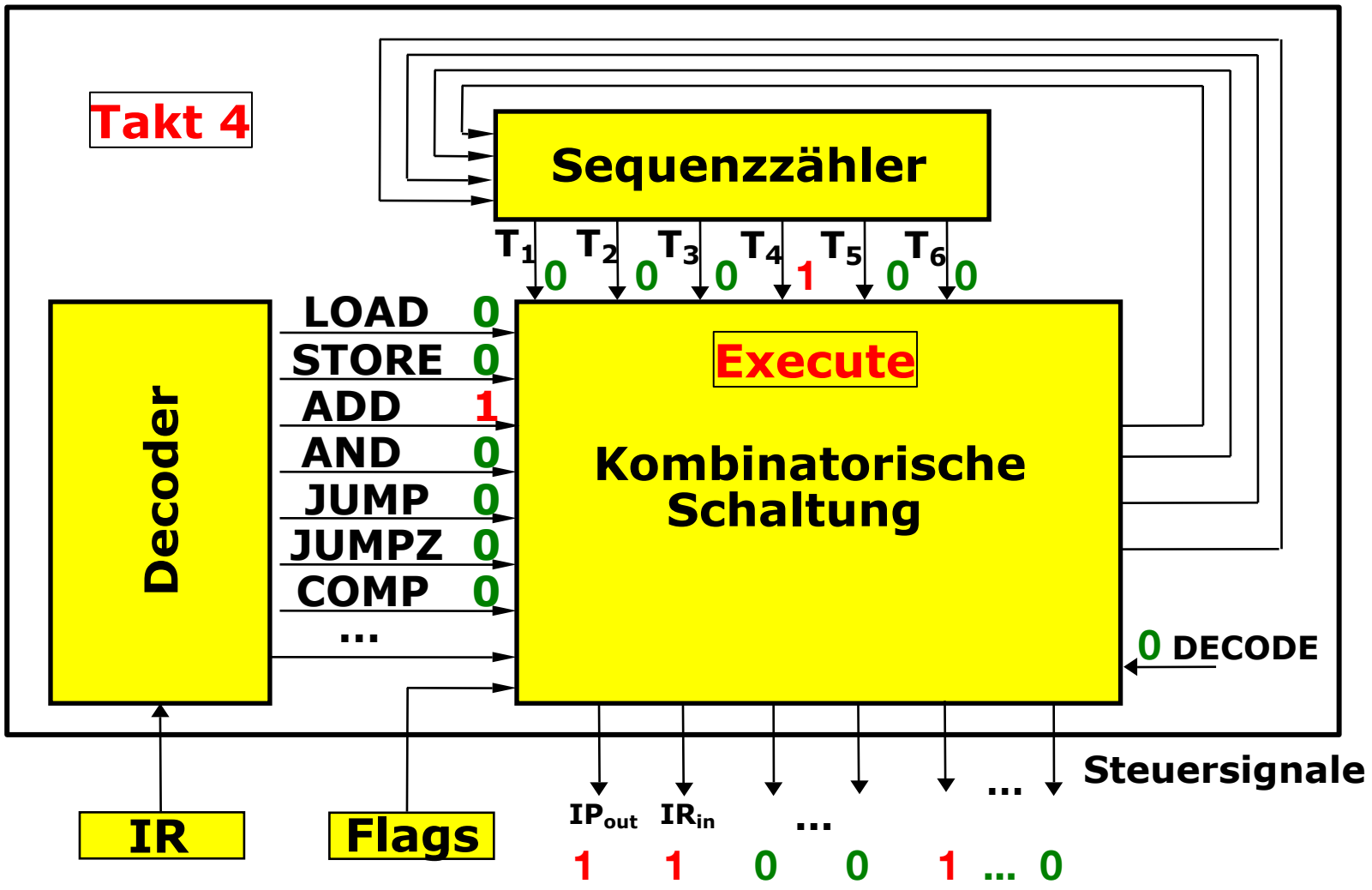
Funktionsweise einer Steuereinheit



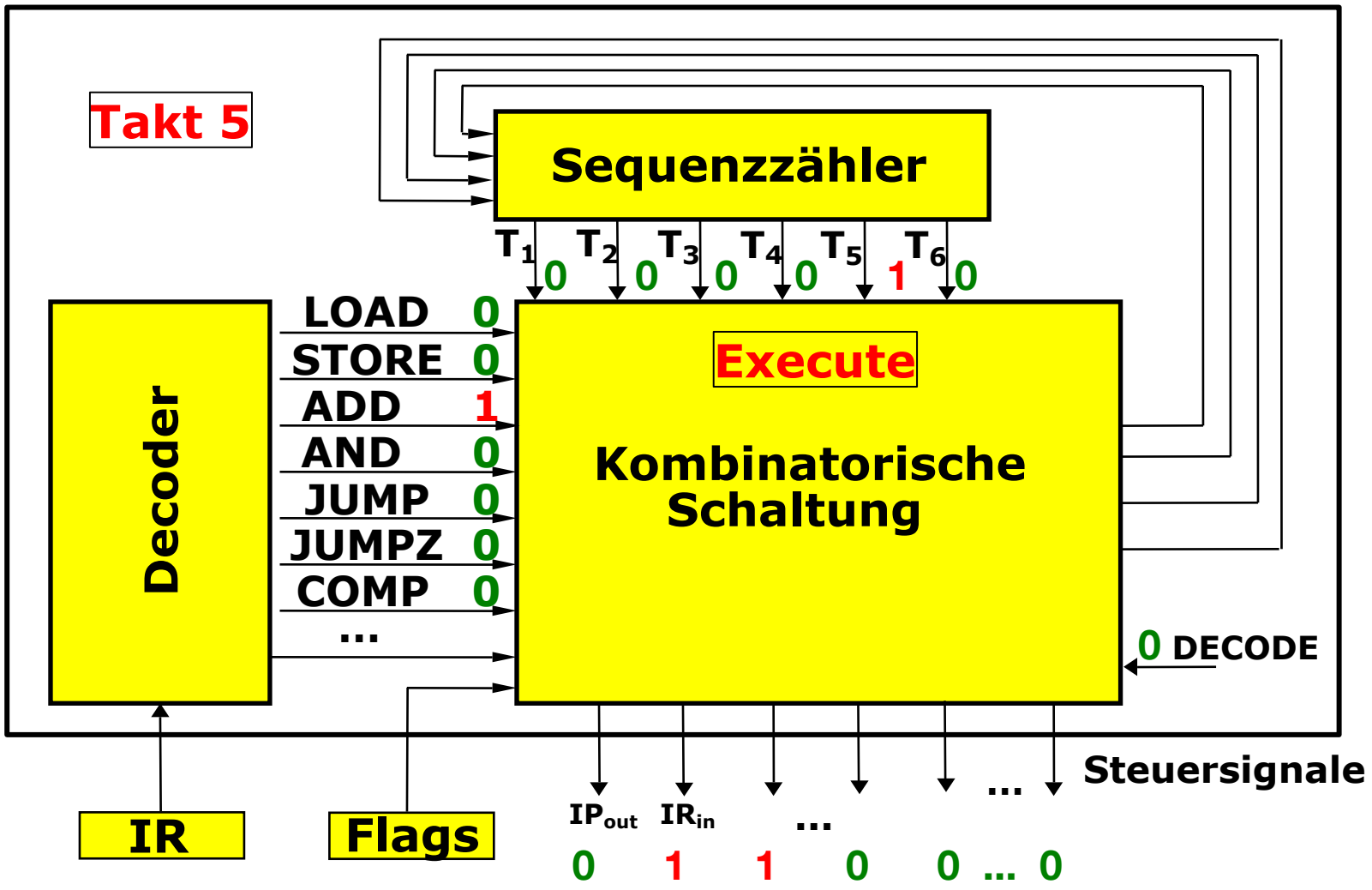
Funktionsweise einer Steuereinheit



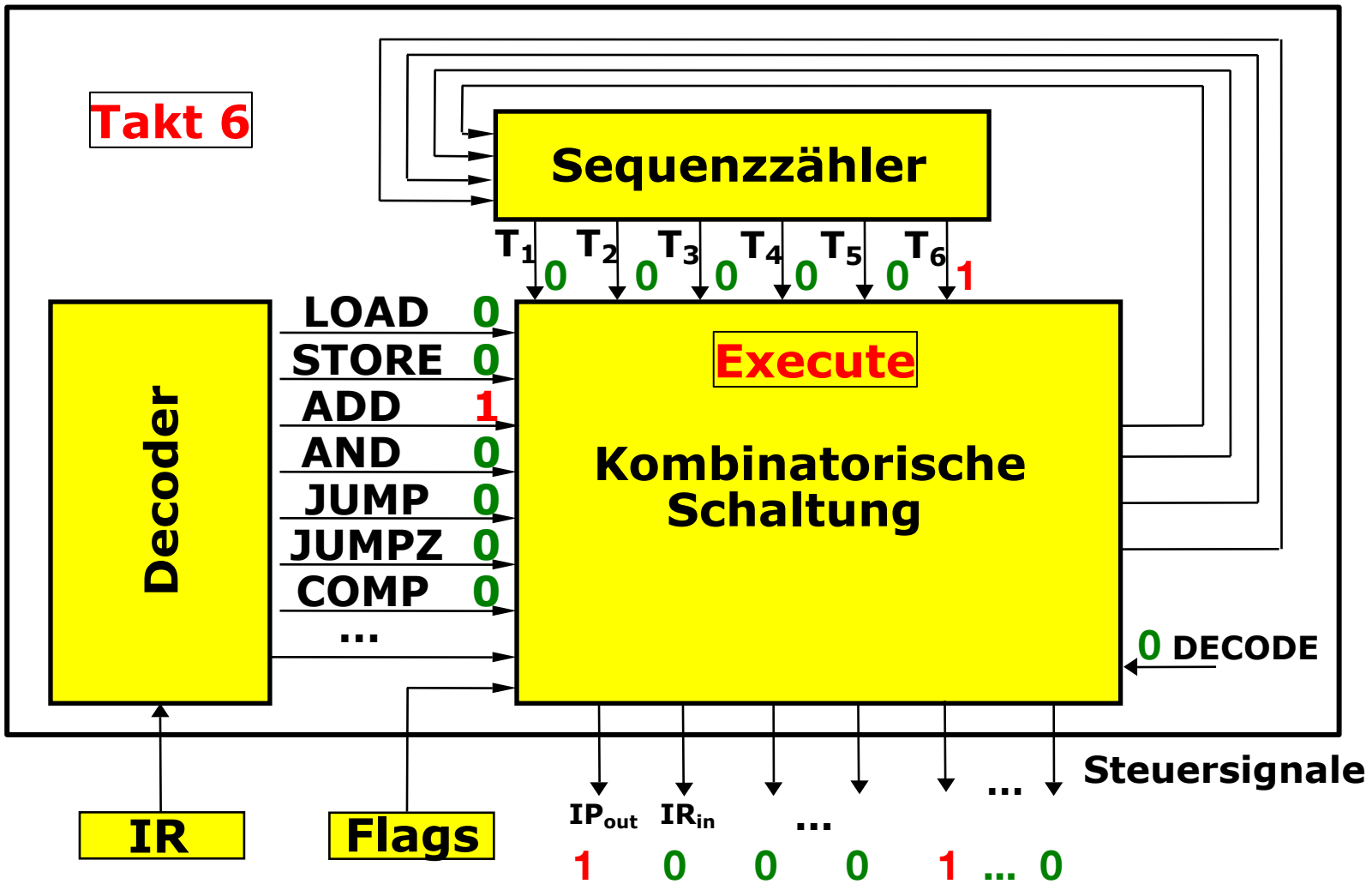
Funktionsweise einer Steuereinheit



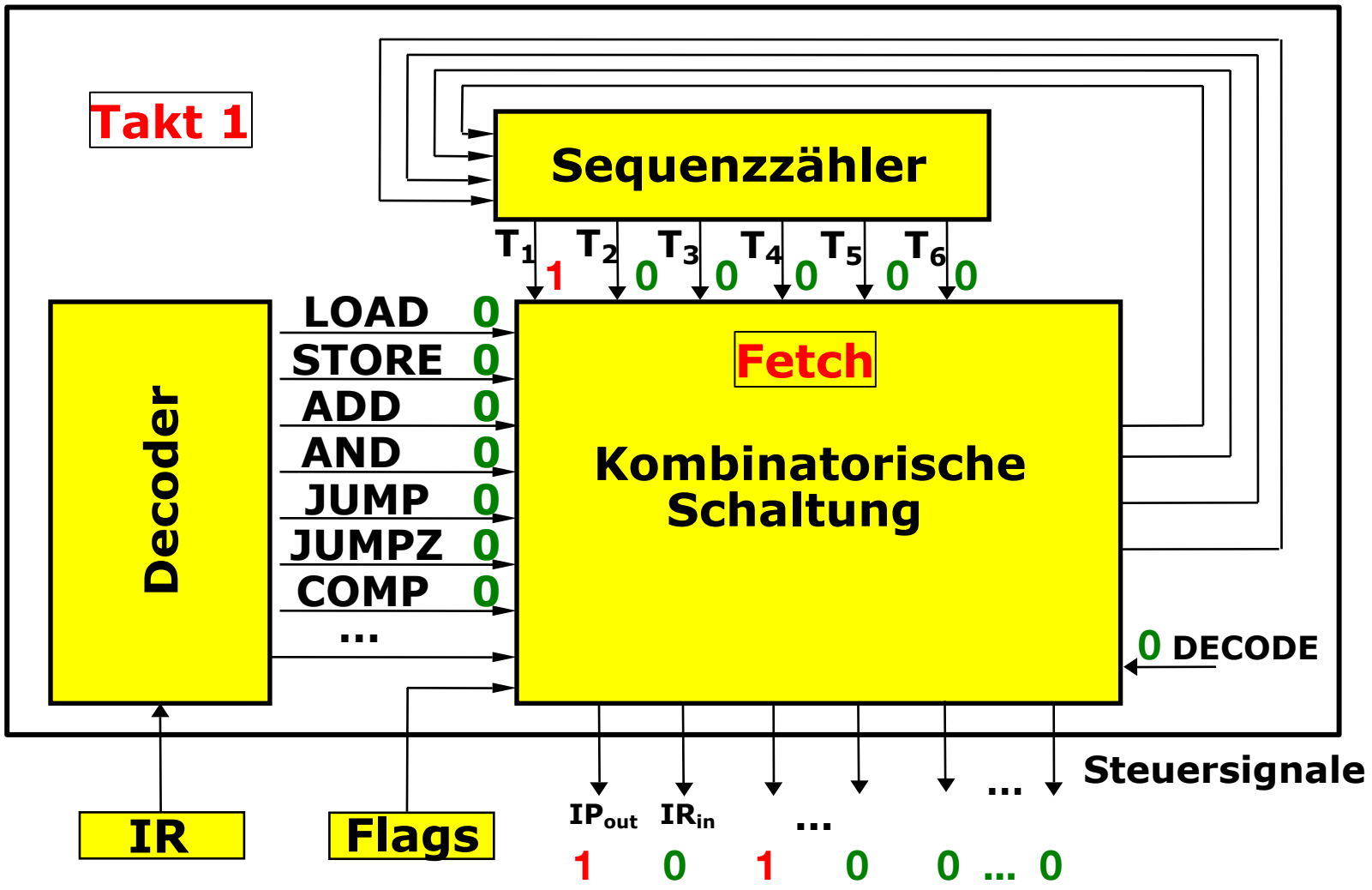
Funktionsweise einer Steuereinheit



Funktionsweise einer Steuereinheit

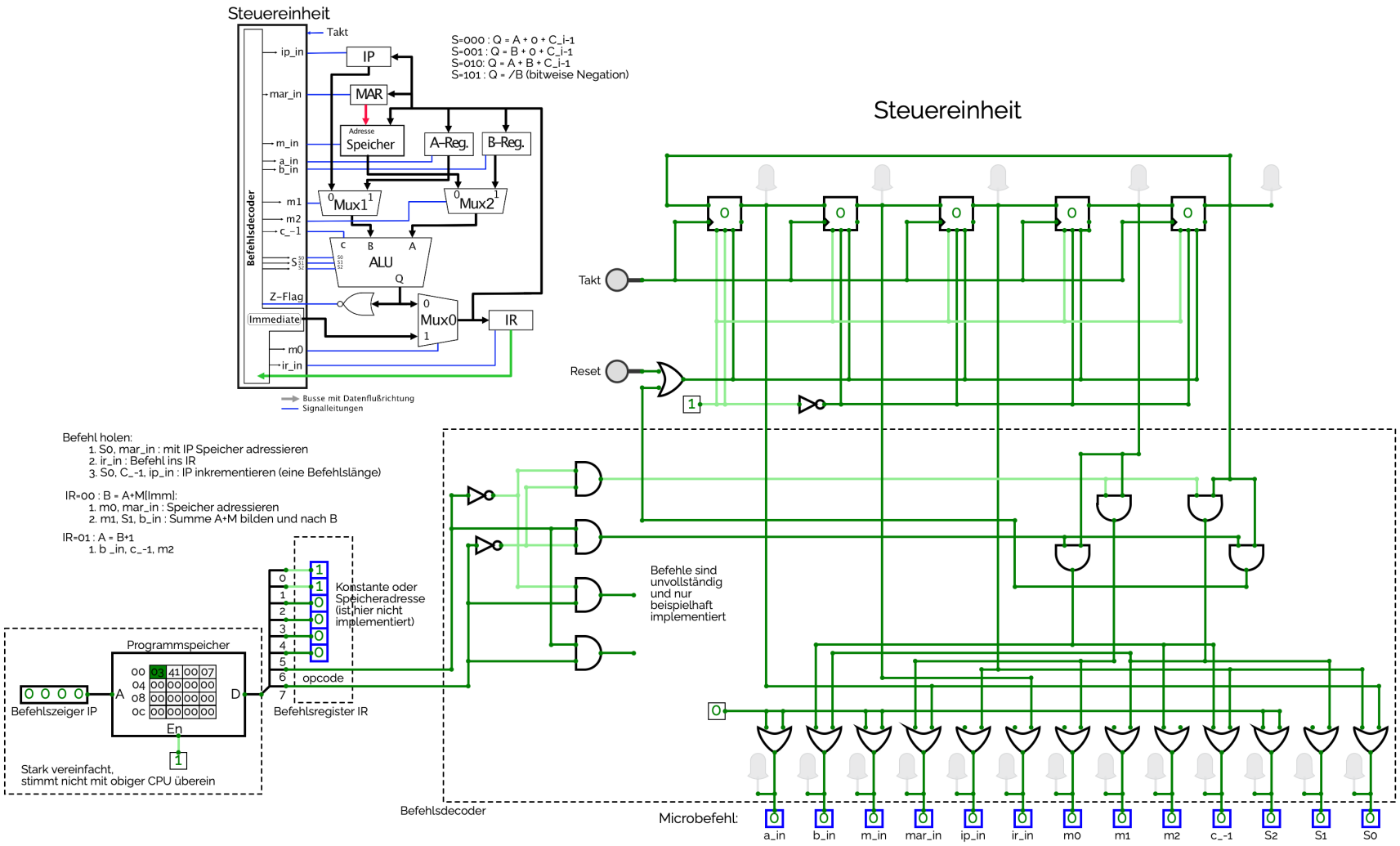


Funktionsweise einer Steuereinheit



Simulation der Steuereinheit einer einfachen CPU

https://circuitverse.org/simulator/embed/cpu-e2633e99-d16f-4abd-b694-72ae4af2dd8c

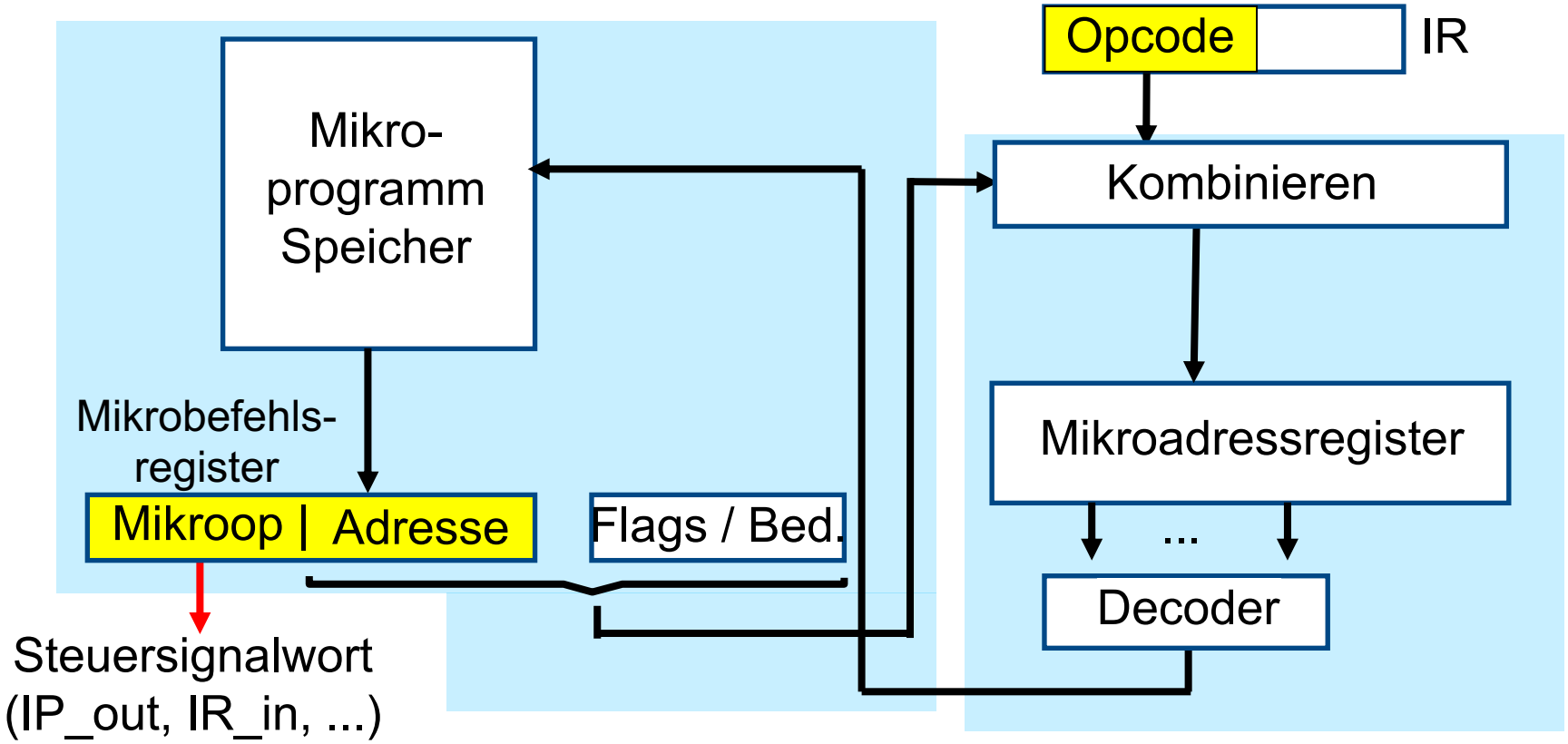


Mikroprogrammierte Steuereinheit

- Verfahren für einen systematischeren Entwurf der Steuereinheit: **Mikroprogrammierung**
- Erstmals vorgeschlagen von Maurice Wilkes (1951)
- Jeder Maschinenbefehl wird als eine Folge von „Mikrobefehlen“ realisiert
  - Mikrobefehl = Kombination von Steuersignalen (Steuersignalwort)
- Mikroprogramm = Gesamtheit der Schritte, die notwendig sind, um eine (Maschinen-)Instruktion auszuführen

Mikroprogrammierte Steuereinheit

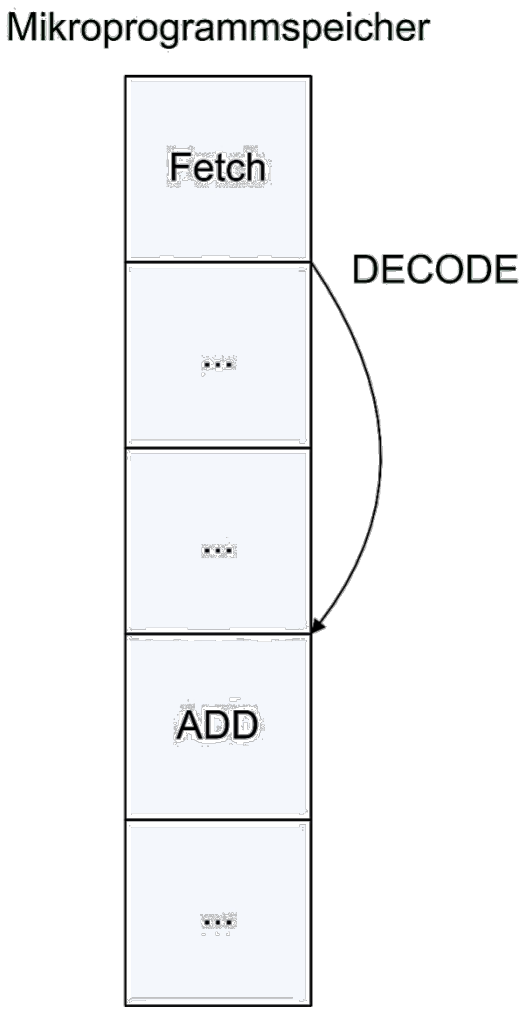
Der Mikroprogrammspeicher enthält Mikroprogramme zur Ausführung der ISA-Instruktionen



Beispiel: Größe der Mikroprogrammspeicher (in Bit) der PDP-11 je nach Modell: 0.25k × 40 Bit; 0.25k × 56; 0.25k × 65; IBM 370/165: 4k x 105

Mikroprogrammausführung

1. Mikroprogramm zum Laden eines Befehls aus dem Speicher („Fetch Instruction“) wird ausgeführt; Befehlscode steht danach im Befehlsregister
2. In Abhängigkeit vom Befehlscode Verzweigung (DECODE) in das entsprechende Mikro(unter)programm
3. Mikroprogramm für Befehl wird ausgeführt
4. Sprung zu 1.



Mikroprogrammausführung

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Holen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

Ausführen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Holen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Ausführen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Holen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

Ausführen

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Holen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

Ausführen

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Holen

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Mikroprogrammausführung

Ausführen

(Maschinen-) Programmspeicher

	...
10	ADD R2,R1
12	SUB R1,#2
14	SUB R2,#3
16	JP #20
18	SUB R1,#4
20	ADD R1,R2
	...

Mikroprogrammspeicher der CPU

0	IP_out, MAR_in, A_in, RD, F=INCA, NA[1]	Instruction Fetch: IR= instruction
1	ALU_out, IP_in, WRDY, NA[2]	
2	MDR_out, IR_in, DECODE[3,6,7]	
3	SR=ROP1, R_out, A_in, NA[4]	ADD: ROP1= ROP1+ROP2
4	SR=ROP2, R_out, B_in, F=A+B, NA[5]	
5	SR=ROP1, R_in, ALU_out, NA[0]	
6	IRop_out, IP_in, NA[0]	JP: IP=ADDR
7	IRop_out, B_in, NA[8]	SUB: ROP1= ROP1-#OP2
8	SR=ROP1, R_out, A_in, F=A-B, NA[9]	
9	SR=ROP1, R_in, ALU_out, NA[0]	
10	...	

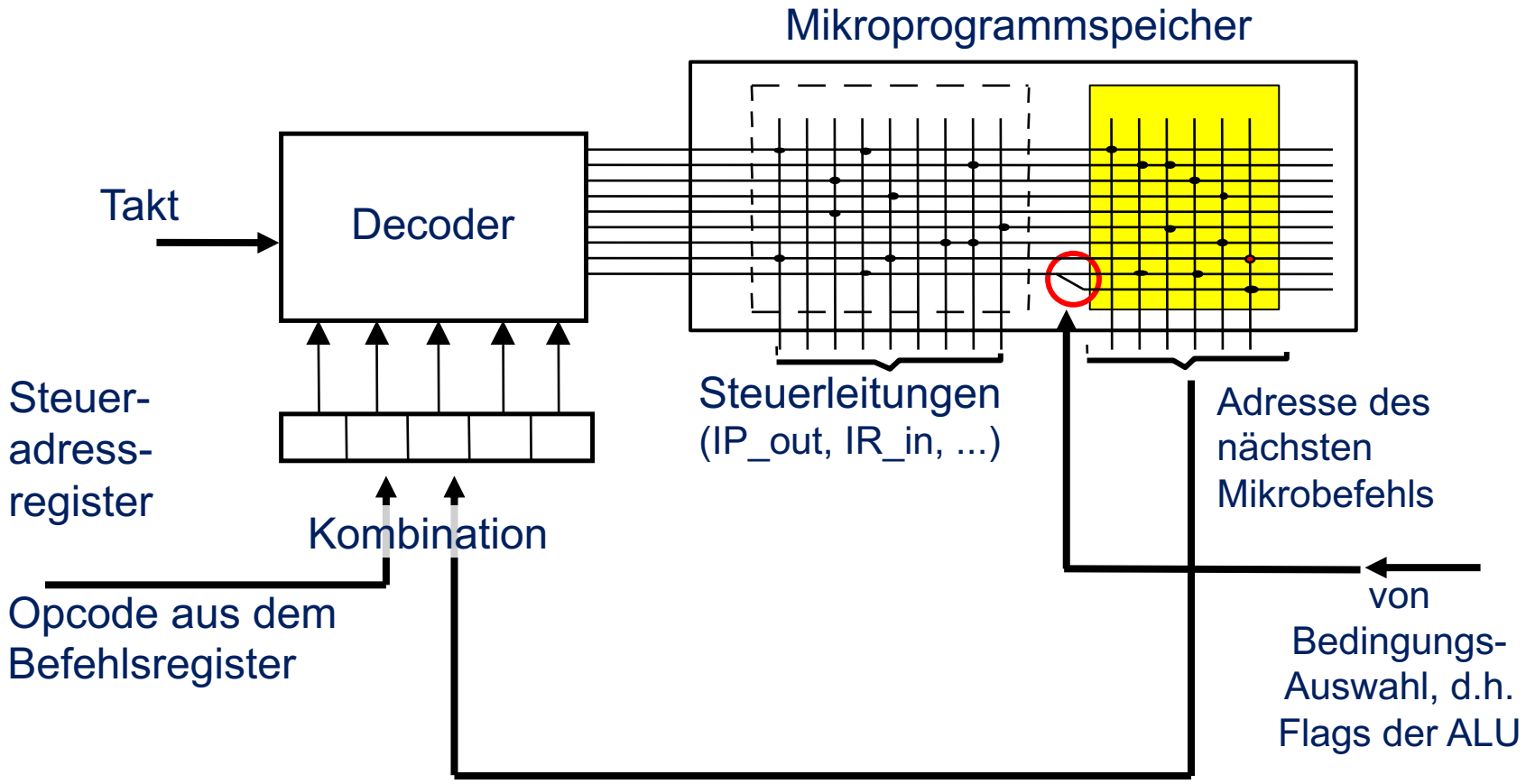
- Basis ist die vorher verwendete Beispielarchitektur
- Maschinen- und Mikroprogramm sind hier symbolisch notiert, real bestehen beide aus binären Wörtern
- im Mikroprogrammspeicher sind, wie üblich, nur die im dem entsprechenden Mikrobefehl aktiven Signale dargestellt

- Wortbreite des Programmspeichers: 2 Byte
- Die Signalgruppe „NA“ beinhaltet die Adresse des nächsten Mikrobefehls
- das Signal „DECODE“ veranlasst die Dekodierung des Befehls, d.h. verzweigt zur Mikroprogrammadr.: ADD: 3 , JP: 6, SUB: 7

Auswahl der Adresse des nächsten Mikrobefehls

- Die Adresse des nächsten Mikrobefehls kann auf unterschiedliche Weise bestimmt werden:
  - ist im aktuellen Mikrobefehl enthalten und kann unter Umständen (z.B. für Verzweigungen) modifiziert werden (Originalentwurf von Wilkes)
  - ist in einem Register enthalten (*micro instruction pointer, µIP*) und wird dem Verlauf der Ausführung des µProgramms entsprechend fortgeführt
  - wird über eine komplexe Logik mittels zusätzlichem Adressspeicher ausgewählt

Wilkes Implementierung der Mikroprogrammsteuerung



Mikroprogrammierte Steuereinheit

- Im Prinzip enthält eine mikroprogrammierte Steuereinheit selbst wiederum ein einfaches Steuerwerk für die Abarbeitung der Mikroprogramme
- Vorteile:
  - einfacher, systematischer Entwurf auch für komplizierte Befehle
  - leichte Änderung im Entwurfsprozess möglich
  - einfach zu erreichende Kompatibilität unterschiedlich aufgebauter CPUs zueinander
  - bei beschreibbarem Mikroprogrammspeicher: nachträgliche Korrektur von Fehlern möglich
- Nachteile: Hoher Platzbedarf, relativ langsame Abarbeitung, schlechte Optimierbarkeit

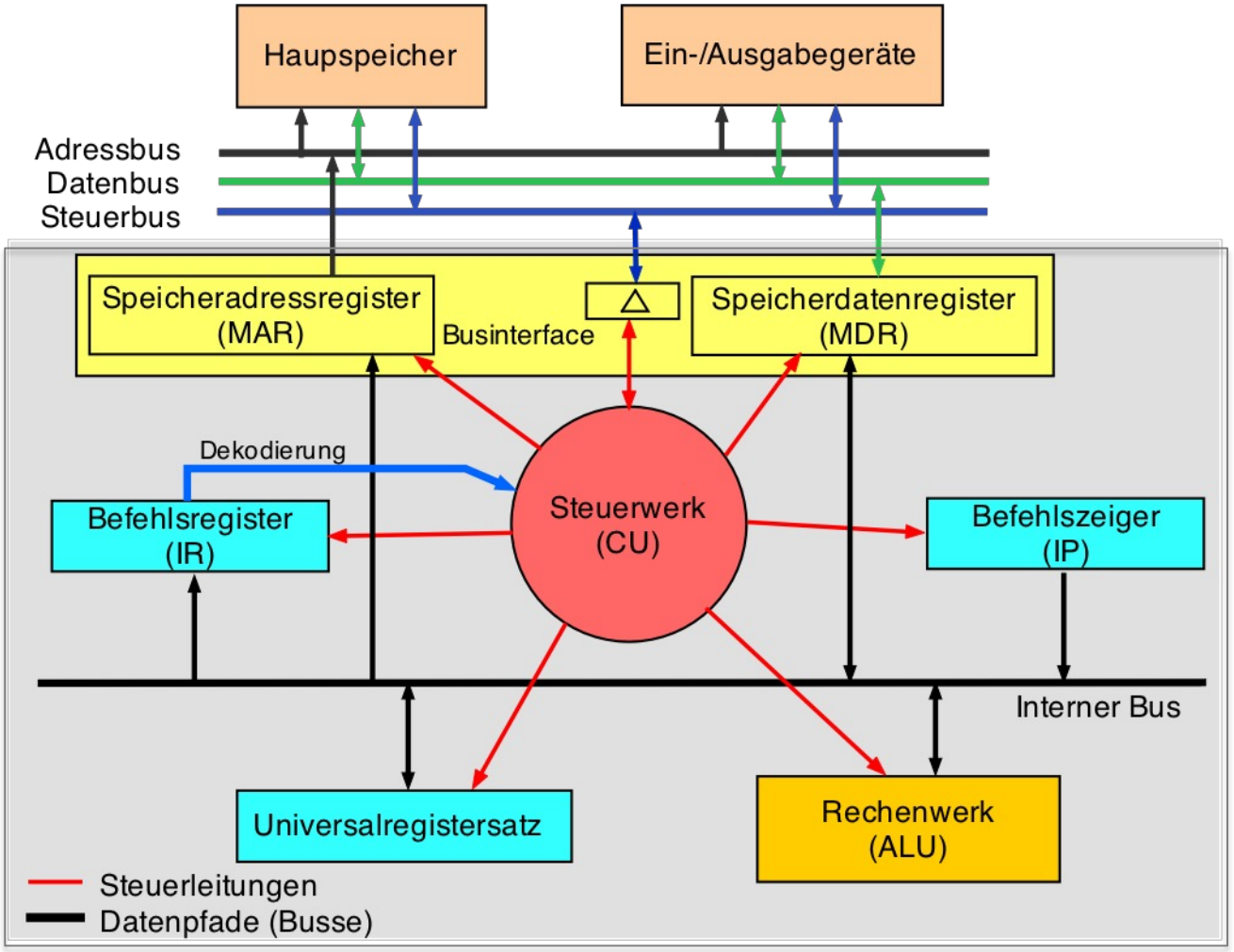
Architekturmodelle

- Bisher nicht betrachtet: der Speicherort für Instruktionen und Daten
- Instruktionen und Daten können gemeinsam oder in separaten Programm- und Datenspeichen gespeichert sein
- Bei der **Von-Neumann-Architektur** wird ein gemeinsamer Programm- und Datenspeicher verwendet
- Die **Harvard-Architektur** nutzt hingegen getrennte Programm- und Datenspeicher

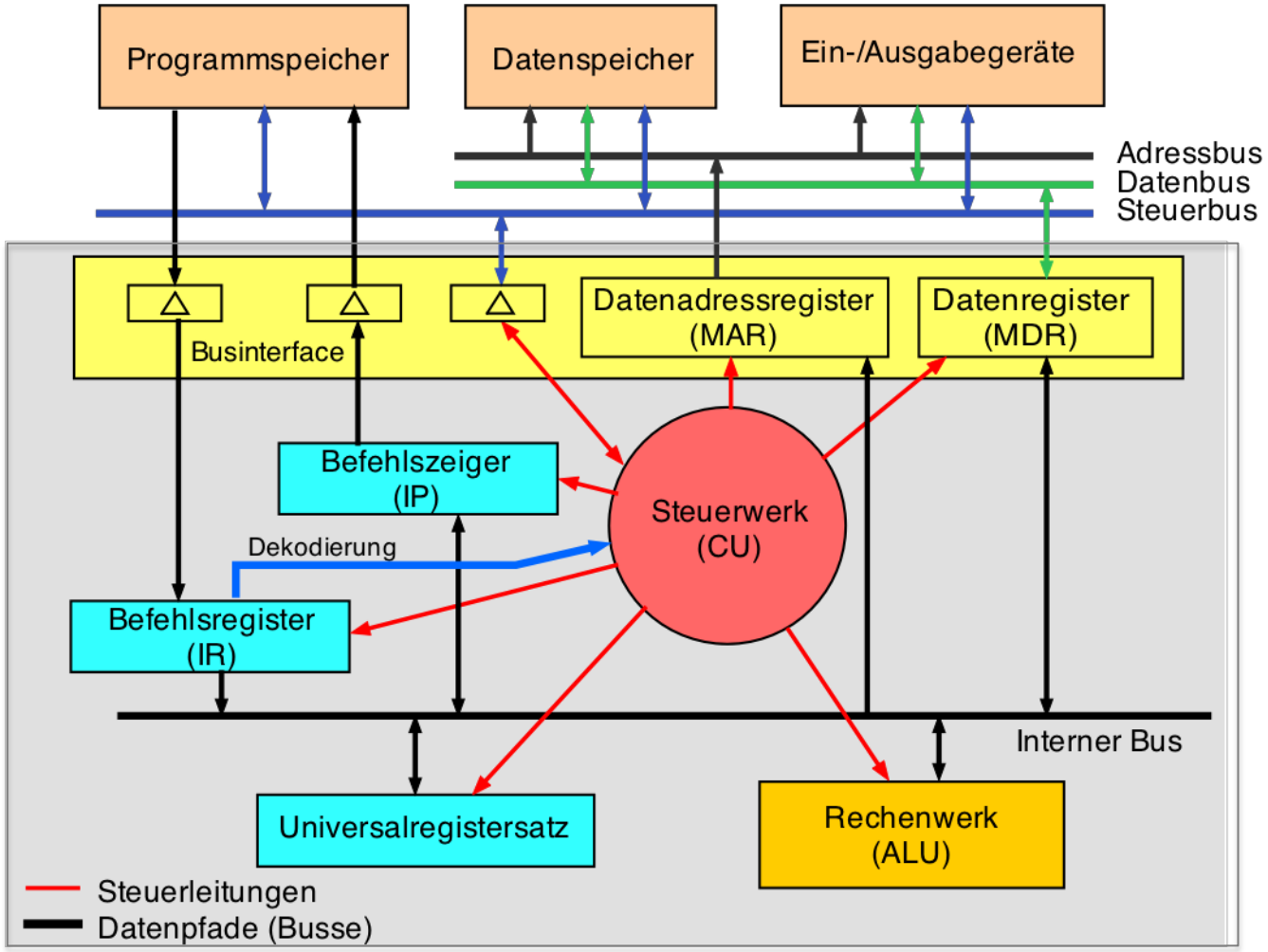
Von-Neumann-Architektur

- Konzept zur Gestaltung eines universellen Rechners nach John von Neumann (Neumann János Lajos):
  - Struktur ist **unabhängig von der Problemlösung**
  - Rechenwerk, Steuerwerk, Speicher, Ein- und Ausgabewerk
  - in einem Speicher werden Programme und Daten **binär codiert** gehalten
  - Speicherzellen werden **durchnummeriert**, erhalten so eindeutige Adressen
  - aufeinanderfolgende Befehle liegen in **fortlaufenden Speicherstellen**
  - eine Änderung der Abarbeitungsfolge ist durch **Sprungbefehle** möglich
  - Verarbeitung durch **Transportbefehle, Arithmetik- und Logikbefehle**
  - die Steuerung erfolgt durch **binäre Schaltwerke**

Von-Neumann-Architektur



Harvard-Architektur



Vor- und Nachteile der beiden Architekturen

- Von-Neumann-Architektur:
  - freier Programmspeicher kann für Daten genutzt werden und umgekehrt
  - einfache Verbindung der Funktionsgruppen
  - gewisse Nachteile hinsichtlich IT-Sicherheit (Angreifer können leichter bösartigen Programmcode einschleusen - später mehr dazu)
  - die meisten modernen Prozessorarchitekturen nutzen dieses Modell - jedoch unter Verwendung bestimmter Ansätze der Harvard-Architektur
- Harvard-Architektur:
  - auf Programm- und Datenspeicher kann gleichzeitig zugegriffen werden (kein Speicher-“Engpass“, „Flaschenhals“)
  - größerer Aufwand
  - häufig eingesetzt in digitalen Signalprozessoren oder Microcontrollern (verwendet in eingebetteten Systemen (embedded systems))

Endianness

- Problem: Ablage von numerischen Daten die aus mehr als einem Byte bestehen, in Byte-adressierten Speicher (in vielen Systemen üblich)
- Z.B. Ablage der 32-Bit-Ganzzahl

  00001111 11110000 00000000 11111111

  ab Speicheradresse 13 aufsteigend
- Eine Möglichkeit: an der *niedrigsten* Speicheradresse liegt das *höchstwertige* Byte der Zahl - genannt **Big Endian**

Adresse:	13	14	15	16
	00001111	11110000	00000000	11111111

- Andere Möglichkeit: An der *niedrigsten* Speicheradresse liegt das *niedrigstwertige* Byte - genannt **Little Endian**

Adresse:	13	14	15	16
	11111111	00000000	11110000	00001111

Endianness

- Es existieren CPUs, die Big Endian verwenden (z.B. Sparc, PowerPC, Mainframes (Großrechner)) und solche, die Little Endian verwenden (z.B. Intel x86)
- Endianness wird wichtig, wenn:
  - auf einzelne Bytes/Worte einer zuvor abgelegten, größeren Zahl separat zugegriffen wird; Beispiel:
    - speichern eine 32-Bit-Zahl ab Adresse 13
    - kopieren des niedrigstwertigen Bytes aus dem Speicher an andere Stelle
      - → Welche ist die Adresse des niedrigstwertigsten Bytes?
      - Little Endian: 13 ; Big Endian: 16
  - Daten zwischen Rechnern ausgetauscht werden, die unterschiedliche Endianness verwenden
    - z.B. beim Versenden von Daten über ein Netzwerk oder beim Ablegen von Daten auf einem gemeinsam genutzten Datenträger

Endianness

- Es gibt in vielen Softwarebibliotheken (z.B. der C-Standardbibliothek) Funktionen, die einen Wert von „Host Byte Order“ in eine festgelegte Standard-Bytereihenfolge („Network Byte Order“) und zurück umwandeln
  - übliche Namen für diese Funktionen sind *hton()* (für „Host to Network“, also konvertieren von der lokalen in die festgelegte Standard-Endianness) und *ntoh()* („Network to Host“)
  - in modernen Computer-Netzwerken (z.B. dem „Internet“) wird Big Endian als „Network Byte Order“ verwendet
- Abhängig von der Prozessorarchitektur wird bei Aufruf einer dieser Funktionen
  - nichts getan (Pozessor verwendet Big Endian) oder
  - die Bytereihenfolge umgekehrt

RISC vs. CISC

- Es gibt zwei mögliche Herangehensweisen bei der Entwicklung einer ISA:
- Complex Instruction Set Computer (CISC)
  - viele aufwendige ISA-Befehle
  - z.B. Intel x86
- Reduced Instruction Set Computer (RISC)
  - wenige schlanke ISA-Befehle
  - z.B. Sun Sparc
- CISC wurden zuerst entwickelt, später RISC
- Moderne Prozessoren verwenden meist Ideen beider Entwicklungsrichtungen

CISC

- Grundidee von CISC-Architekturen:
  - Erleichterung der maschinennahen Programmierung
  - Unterstützung von Konstrukten höherer Programmiersprachen
  - u.U. Unterstützung von Konstrukten für Betriebssysteme
- Entstehungsgründe für umfangreiche Maschinenbefehlssätze:
  - Geschwindigkeitsunterschied zwischen CPU und Hauptspeicher
  - Mikroprogrammierung
  - kompakter Code
  - Unterstützung höherer Programmiersprachen
  - Marktstrategie

RISC

- Grundidee von RISC-Architekturen:
  - oft verwendete, einfache Befehle so schnell wie möglich ausführen
    - Softwareanalyse-Statistiken zeigen: nur wenige Instruktionen eines Prozessors werden häufig verwendet
  - Ausführung dieser Befehle möglichst in einem einzigen Takt
  - keine Mikroprogrammierung
  - nur Load/Store und Register-Register-Befehle: weniger Adressierungsarten und schnelle Ausführung
- Sprachkonstrukte höherer Programmiersprachen werden bereits von Compilern in Sequenzen von einfachen Befehlen der Zielprozessoren übersetzt → Verringerung des Aufwands bei der Befehlsdekodierung des Prozessors → Erhöhung des Aufwands beim Compiler
- Operanden werden nach Möglichkeit in großen Registersätzen gehalten:
  - schneller Zugriff
  - schnelle Verarbeitung
- Einheitliche Befehlsformate, dadurch schnelle Decodierung
- Ausführungszeit u.U. kürzer als bei CISC (obwohl RISC-Programme u.U. länger sind)

RISC

- Geblieben ist heute von der RISC-Idee im Wesentlichen:
  - die Load/Store-Architektur
  - ein großer Registersatz; typische Größen:
    - 32 allgemeine und
    - 32 Fließkomma-Register
  - einheitliche Befehlsformate von 32 oder 64 Bit
  - die Verwendung weniger Adressierungsarten
  - der Verzicht auf Mikroprogrammierung
- Anmerkung: Die Kerne moderner Mikroprozessoren folgen im Wesentlichen RISC-Prinzipien – allerdings für *Mikro*instruktionen

Zusammenfassung

- Konstruktion eines „kompletten“ Prozessors aus den zuvor zusammengetragenen Einzelteilen
- Datenpfad (Rechenwerk) zum Durchführen der Berechnungen
- Kontrollpfad (Steuerwerk) zum Erzeugen der zur Ausführung von Instruktionen notwendigen Steuersignale
- Implementation von Steuerwerken
  - in festverdrahteter Form
  - mittels Mikroprogrammierung
- Freiheitsgrade beim Rechnerentwurf, zum Beispiel
  - Architekturmodelle (von Neumann, Harvard)
  - Endianness
  - ISA-Implementierungen: CISC, RISC