# Technische Informatik

## 4 – Prozessor Einzeltaktimplementierung

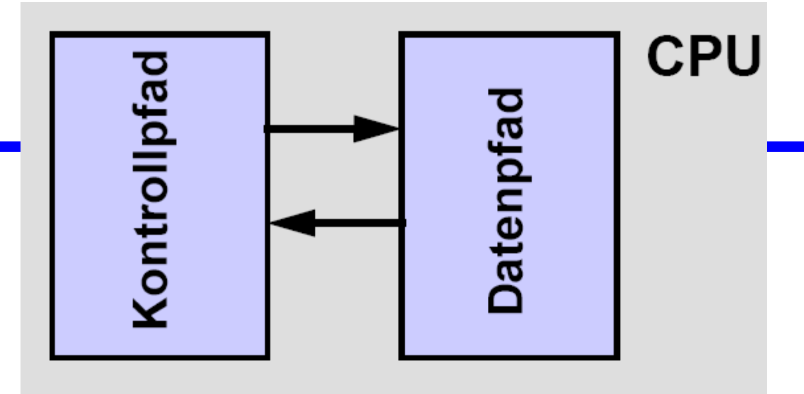
© Lothar Thiele

Computer Engineering and Networks Laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Vorgehensweise

# Prinzipieller Aufbau




***Datenpfad:***

- Verarbeitung und Transport von Instruktionen und Daten.
- Datenpfad muss alle Operationen und Datentransporte unterstützten.
- Viele Optionen sind möglich, z.B. Trennung von Instruktions- und Datenverarbeitung, Fliessbandverarbeitung (Pipelining), parallele arithmetische Einheiten, ... .

***Kontrollpfad:***

- Verarbeitung und Transport von Steuerungsdaten.
- Viele Optionen möglich, z.B. rein kombinatorische Steuerung, zustandsbasierte Steuerung, Mikroprogrammsteuerung, ... .

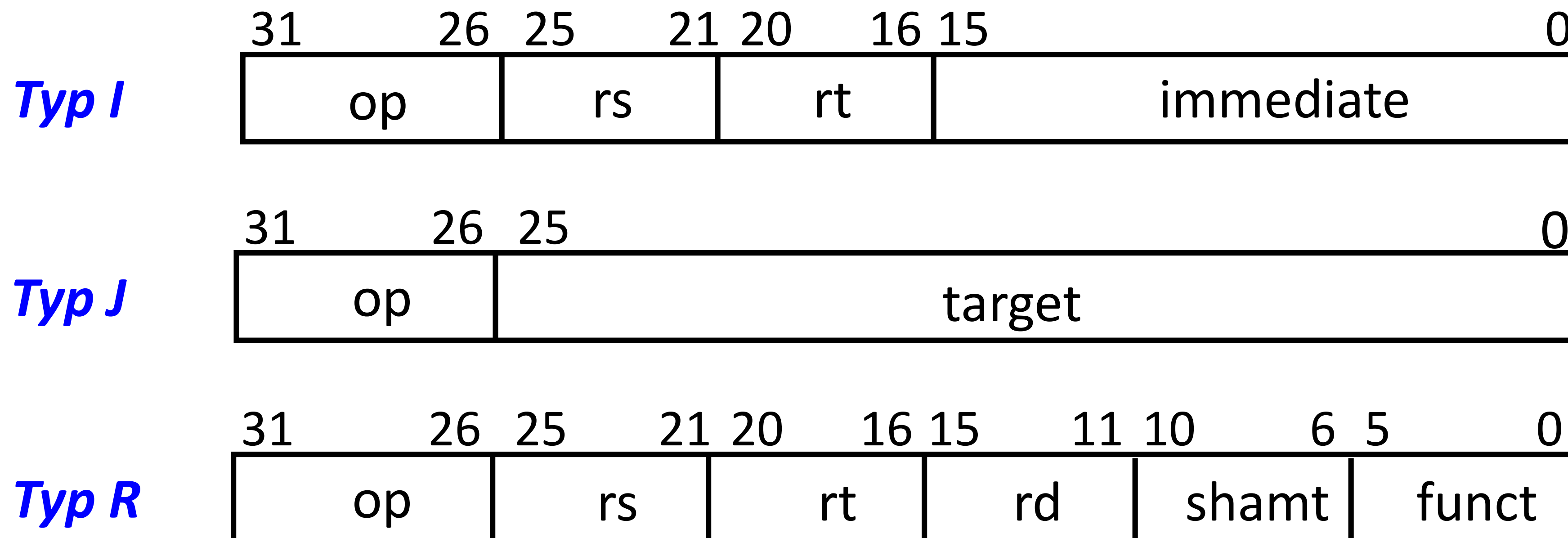
# Verfeinerung des Instruktionssatzes

- Instruktionen werden i.a. auf der Hardware *interpretiert.*
- Hier wird *beispielhaft* die Implementierung einiger MIPS-Instruktionen dargestellt, mit unterschiedlichen Implementierungsvarianten:
  - ein Takt pro Instruktion „single cycle“,
  - Fliessbandverarbeitung „ pipelining“,
  - instruktionsparallele Ausführung.

- Die *Verfeinerung der Instruktionen* zu Operationen, die direkt in der Hardware ausgeführt werden können, basiert auf einer abstrakten Darstellung der jeweils notwendigen Teiloperationen (*Daten/Kontrollflussgraph).*

# Instruktionskodierung (Wiederholung)

- Unter ***Instruktionskodierung*** versteht man die Umsetzung einer Instruktion in ein Maschinenwort.
- Der MIPS Prozessor benutzt ausschliesslich 32-Bit Kodierungen (alle Instruktionen besitzen eine feste Länge).
- Man unterscheidet die 3 Typen I, J und R:



# Instruktionskodierung (Wiederholung)

| Abkürzung | Bedeutung |
|---|---|
| I | immediate (direkt) |
| J | jump (Sprung) |
| R | register (Register) |
| op | 6 Bit Kodierung der Operation |
| rs | 5 Bit Kodierung eines Quellenregister |
| rt | 5 Bit Kodierung eines Quellenregisters oder Zielregisters |
| immediate | 16 Bit direkter Wert oder Adressverschiebung |
| target | 26 Bit Sprungadresse |
| rd | 5 Bit Kodierung des Zielregisters |
| shamt | 5 Bit Grösse einer Verschiebung |
| funct | 6 Bit Kodierung der Funktion (Ergänzung des Feldes op) |

# Die MIPS Teilmenge

- ***R-Typ Instruktionen:***

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| add | add rd, rs, rt | 000000 | rs | rt | rd | 00000 | 100000 |
| subtract | sub rd, rs, rt | 000000 | rs | rt | rd | 00000 | 100010 |
| AND | and rd, rs, rt | 000000 | rs | rt | rd | 00000 | 100100 |
| OR | or rd, rs, rt | 000000 | rs | rt | rd | 00000 | 100101 |
| set less than | slt rd, rs, rt | 000000 | rs | rt | rd | 00000 | 101010 |

- ***Speicherinstruktionen***

| | | | | | |
|---|---|---|---|---|---|
| load word | lw rt, imm(rs) | 100011 | rs | rt | imm |
| store word | sw rt, imm(rs) | 101011 | rs | rt | imm |

- ***Verzweigungsinstruktionen***

| | | | | | |
|---|---|---|---|---|---|
| branch on equal | beq rs, rt, imm | 000100 | rs | rt | imm |
| jump | j target | 000010 | target | | |

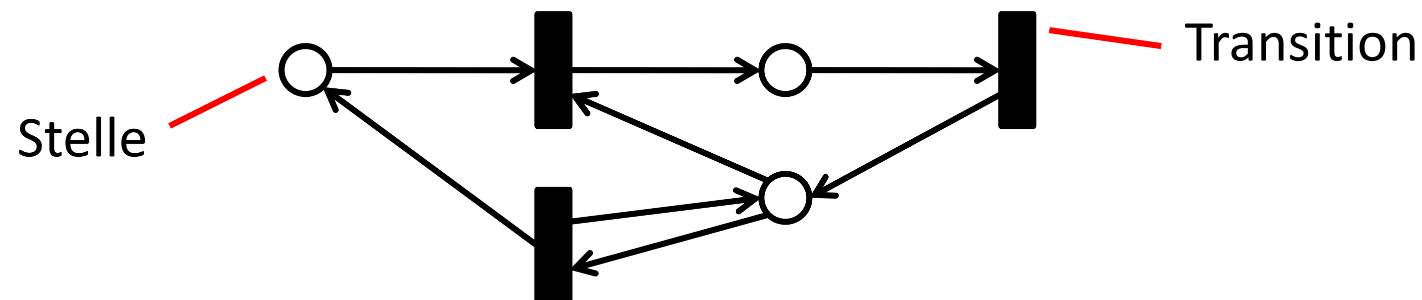
# Petri-Netze

# Petri-Netz

Wir werden die Verfeinerung der Instruktionen mit Petri-Netzen darstellen. Petri-Netze sind eine Standard-Notation zur Darstellung von parallelen und verteilten Operationen.

***Statische Repräsentation:***

- Petri-Netze bestehen aus *Stellen* und *Transitionen*, die durch Kanten miteinander verbunden sind.
- Ein Petri-Netz Graph muss bipartit sein, d.h. keine Stellen dürfen direkt miteinander verbunden sein, das gleiche gilt auch für Transitionen.

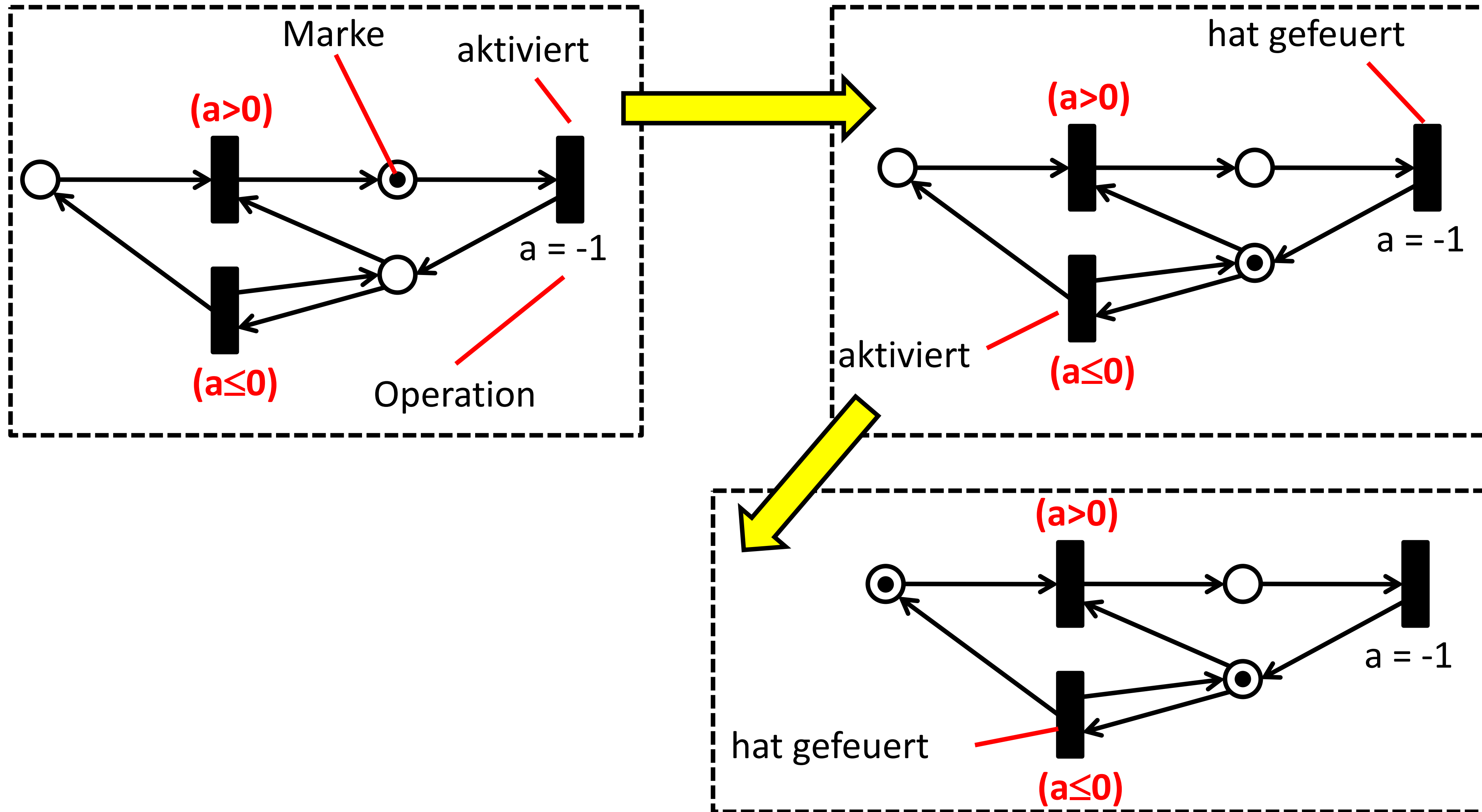


# Petri-Netz

## *Dynamische Repräsentation*

- Den *Stellen* sind *Marken* (Token) zugeordnet. Sie beschreiben den derzeitigen Zustand in der Abfolge der Operationen.
- *Marken* werden über Transitionen nach bestimmten Regeln “transportiert”:
  - Eine Transition ist *“aktiviert”*, falls (a) die der Transition zugeordnete Bedingung erfüllt ist und (b) in *jeder* Eingangsstelle mindestens eine Marke liegt.
  - Eine aktivierte Transition kann *“feuern”*. Dabei wird aus *jeder* Eingangsstelle eine Marke entfernt und *jeder* Ausgangsstelle eine Marke zugefügt. Die der Transition zugeordnete Operation wird ausgeführt.

# Petri-Netz



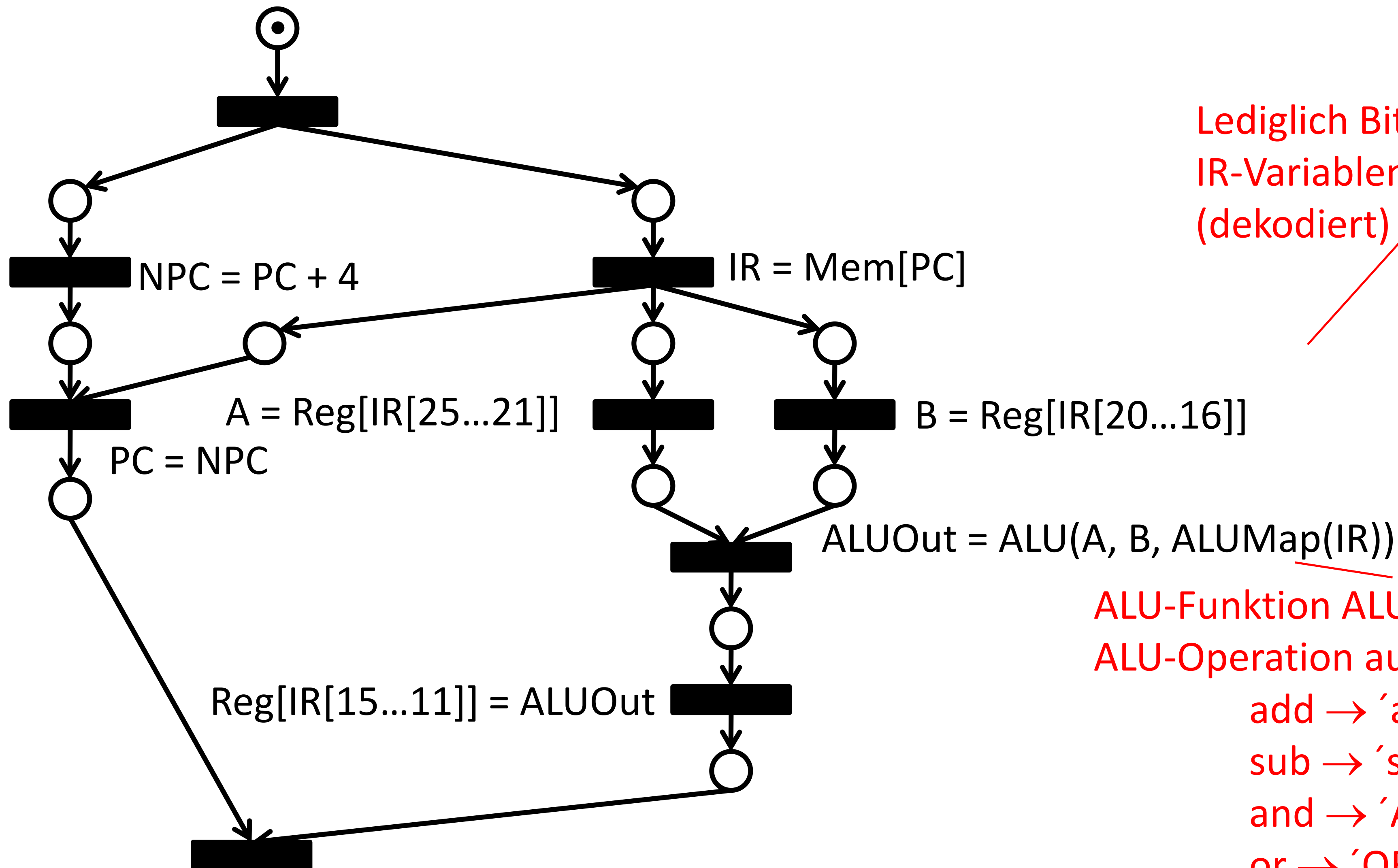
# Modellierung der Instruktionen

# Elementaroperationen

**Der Prozessor stellt die folgenden Elementaroperationen und Variablen zur Verfügung:**

- *Zwischenvariablen* zum Speichern von 32 Bit Daten und Instruktionen: A, B, ALUOut, Target, PC (program counter), NPC (next program counter), IR (instruction register)
- *Interne Register* des Prozessors: Reg[i], 0 ≤ i ≤ 31
- *Hauptspeicher* an der Adresse i: Mem[i]
- *Verschiebung* eines Wortes um 2 Bit nach links und auffüllen mit ‘00’ :
  ‘01110’ << 2 = ‘0111000’
- *Aneinanderhängen* von Bits: ‘001’ <> ‘110’ = ‘001110’
- *Erweiterung* eines Halbwortes auf ein Wort mit/ohne Vorzeichenerweiterung: SignExt(), ZeroExt()
- *Arithmetische Operation* ALU(a, b, op), wobei a und b die Argumente sind und op∈{´add´, ´subtract´, ´AND´, ´OR´, ´setOnLessThan´}:

# Beispiel: R-Instruktion




Lediglich Bits 16…20 der IR-Variablen werden verwendet (dekodiert)

ALU-Funktion ALUMap (IR) bestimmt ALU-Operation aus einer R-Instruktion:

add → ´add´

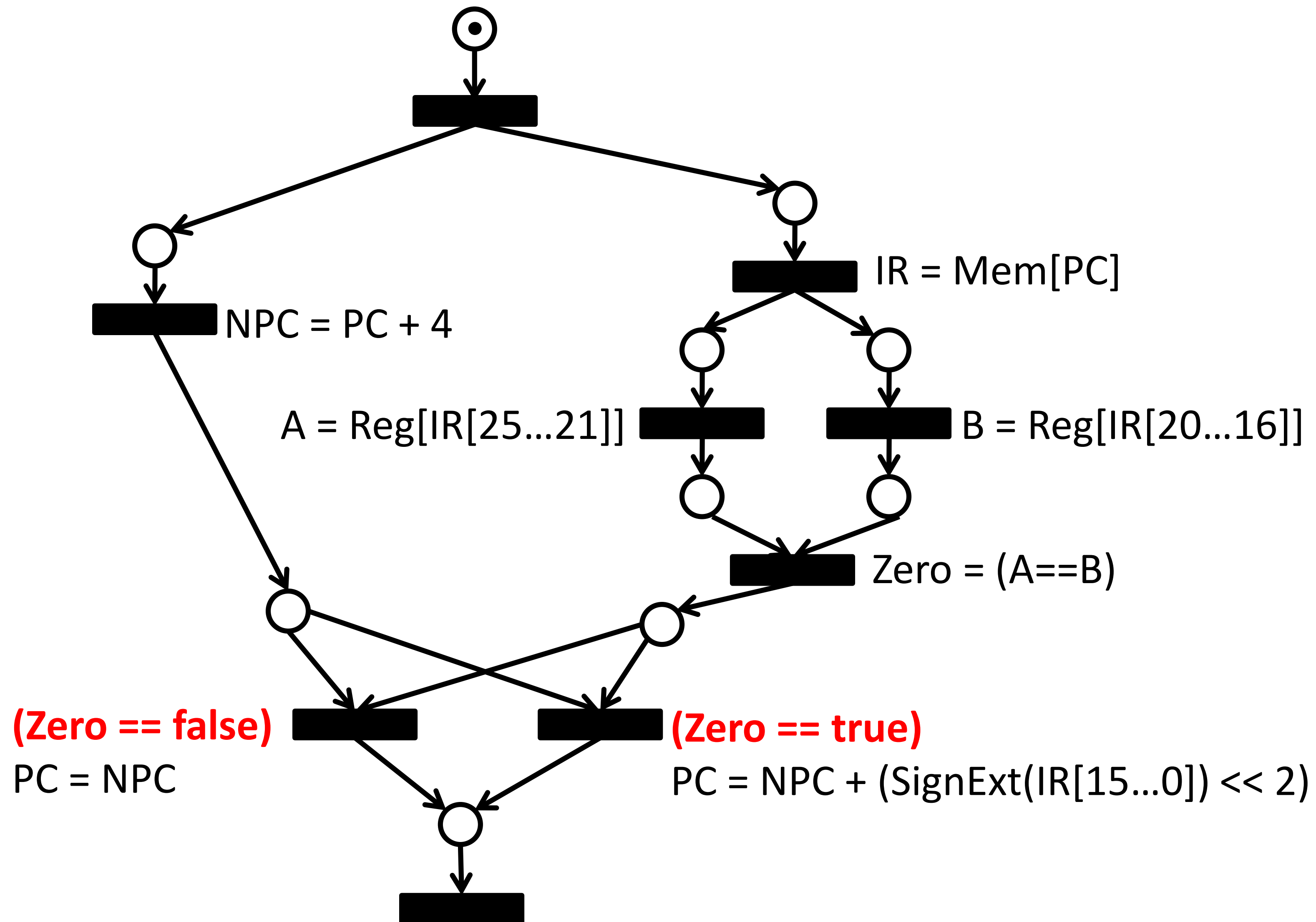
sub → ´subtract´

and → ´AND´

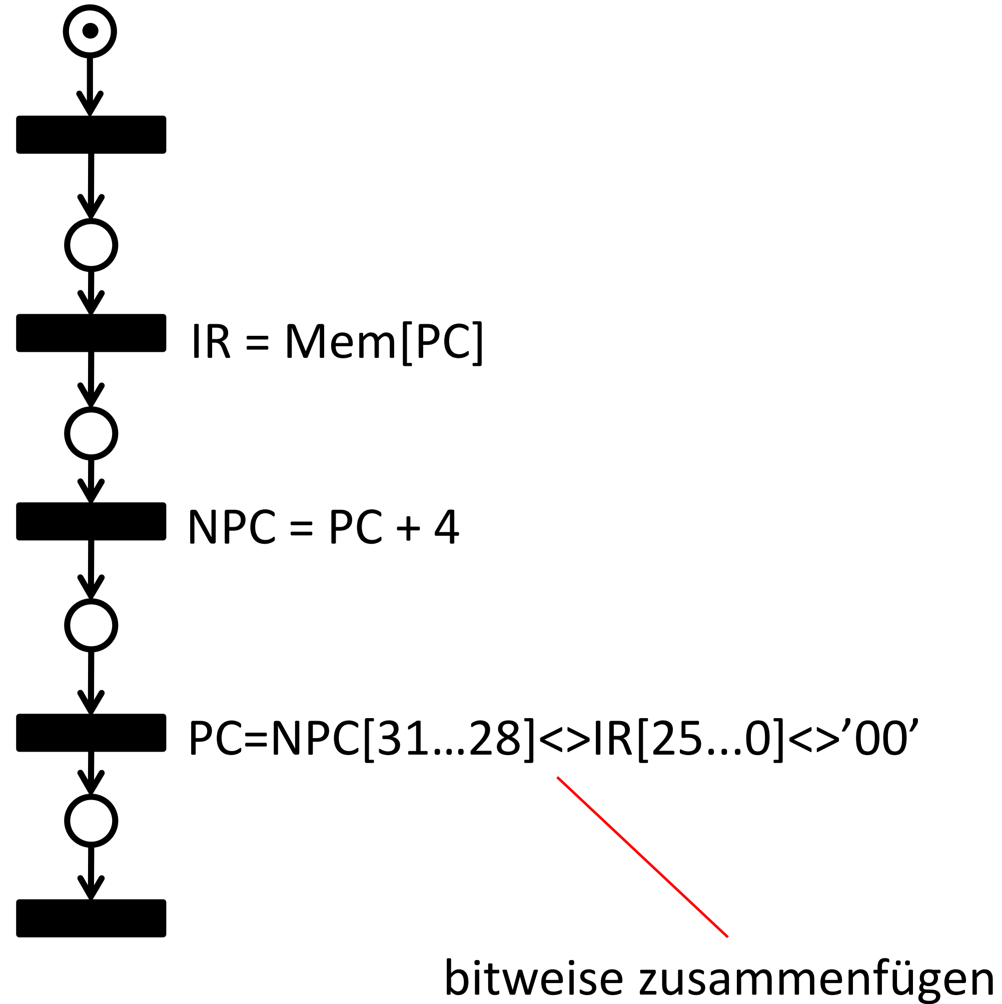
or → ´OR´

slt → ´setOnLessThan´

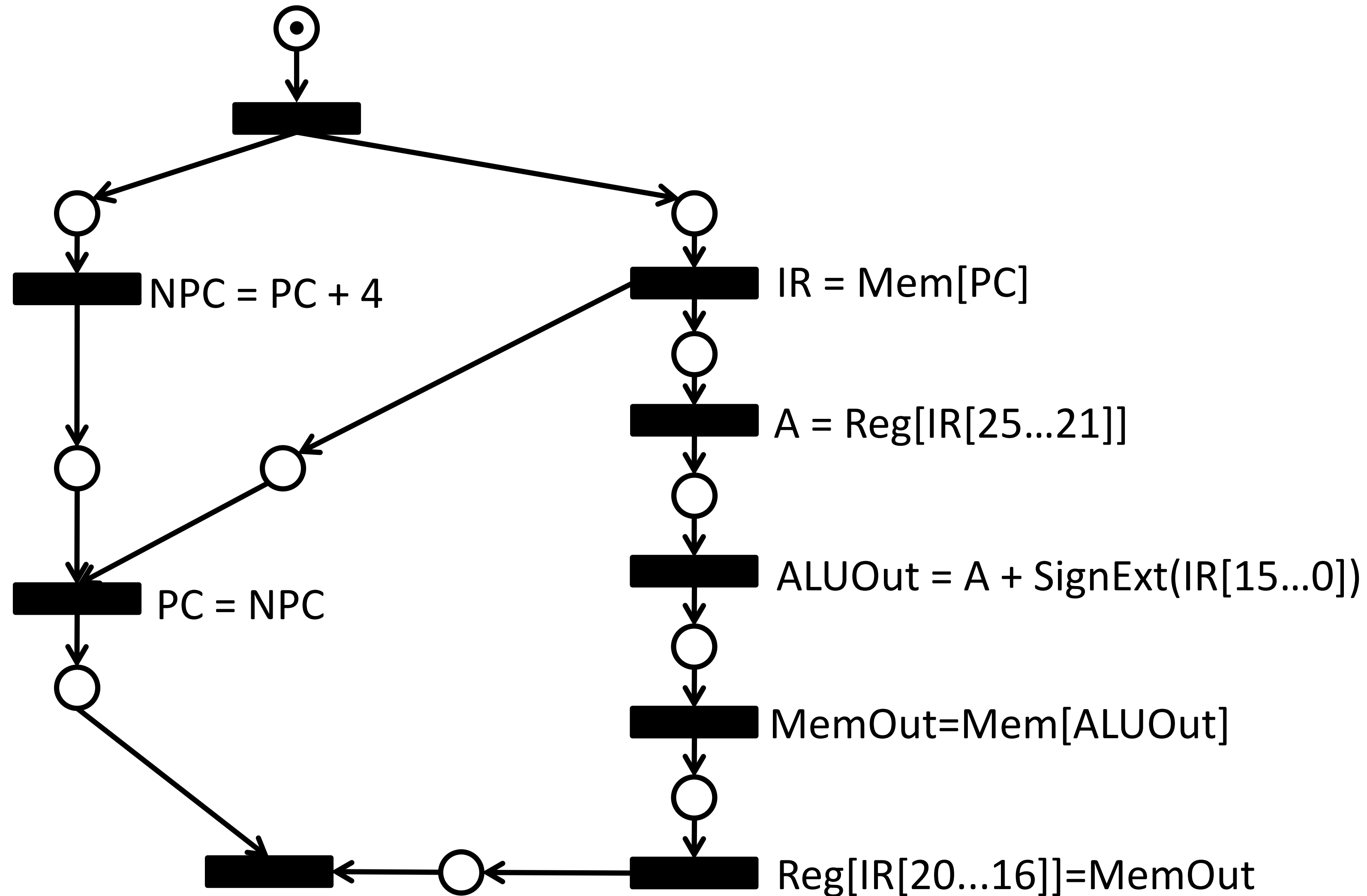
# Beispiel: beq-Instruktion



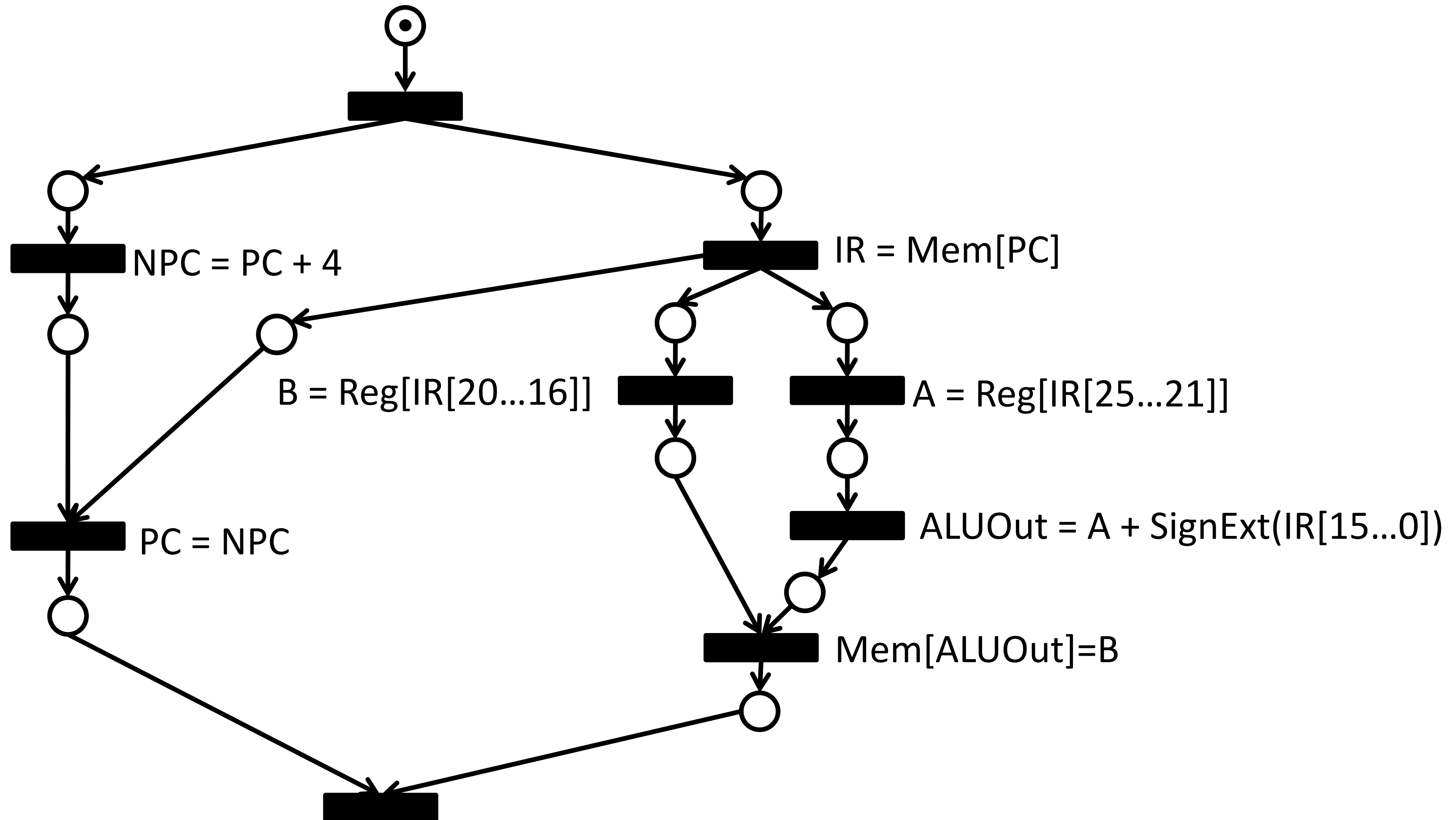
# Beispiel: j-Instruktion



# Beispiel: lw-Instruktion



# Beispiel: sw-Instruktion

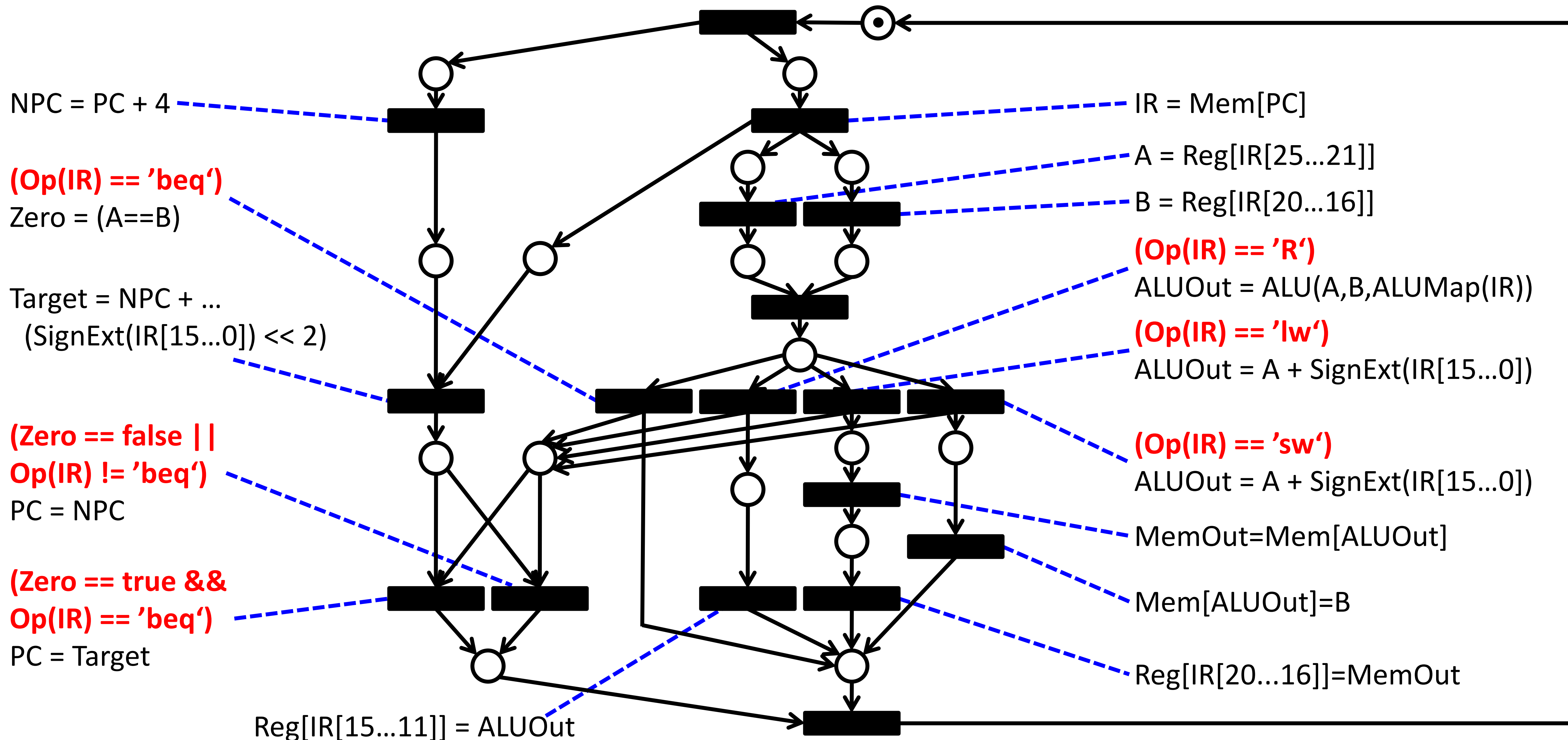


# Modellierung der MIPS Teilmenge

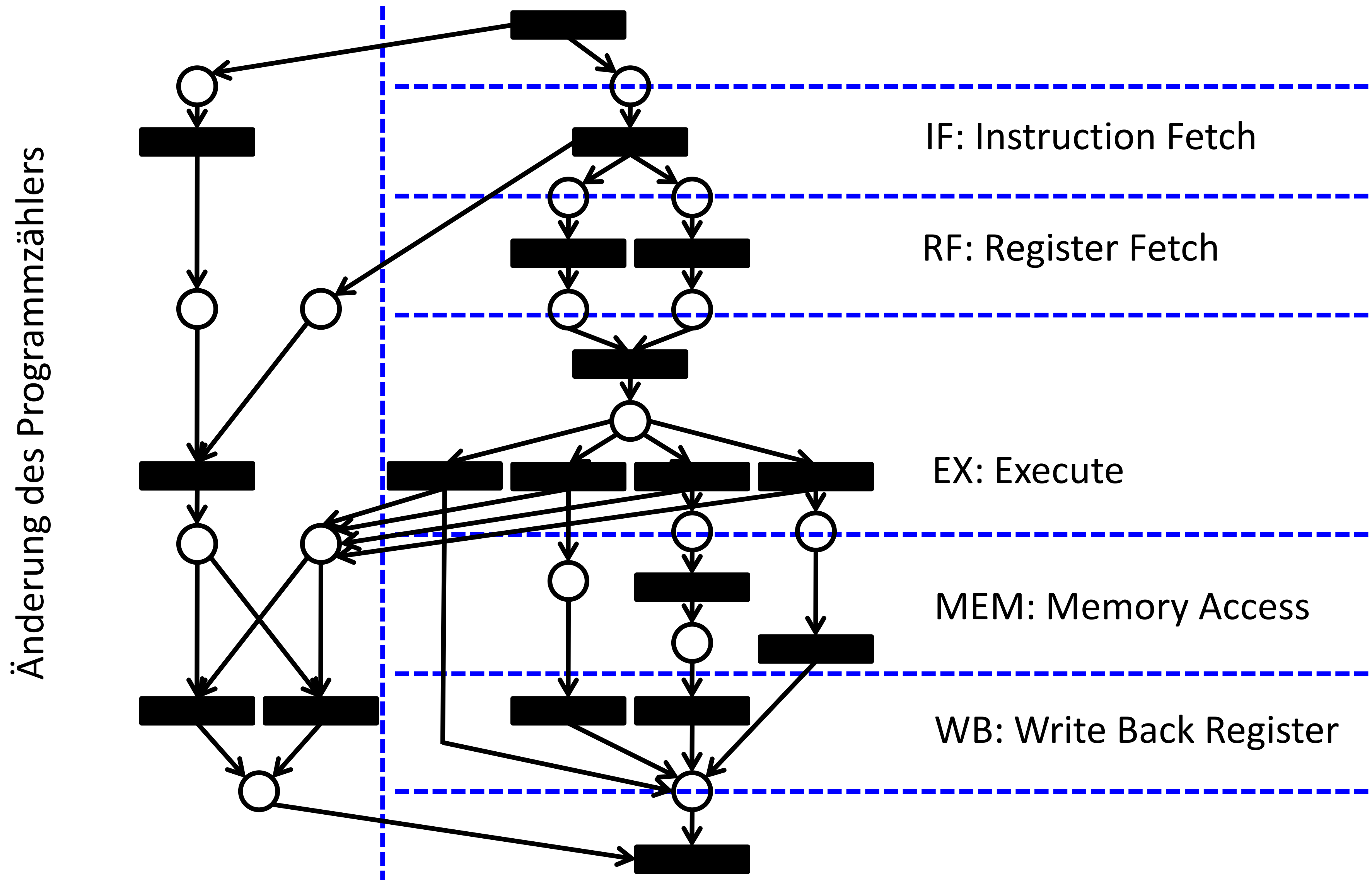
***Vereinigter Kontrollflussgraph:***

- Zusammenfügen aller Instruktionen der MIPS Teilmenge.
- Verschiebung von Operationen zur Vereinfachung der Steuerung.
- Definition einer weiteren Zwischenvariablen ‘*Target*’*.*
- Verbindung der ersten und letzten Transition zur zyklischen Abarbeitung.
- O*p[IR] ∈ {‘R‘, ‘lw‘, ‘ sw‘, ‘ beq‘, ‘ j‘}* gibt den Typ der Instruktion an und wird aus IR[31..26] bestimmt.

# MIPS Verfeinerung (ohne j-Instruktion)



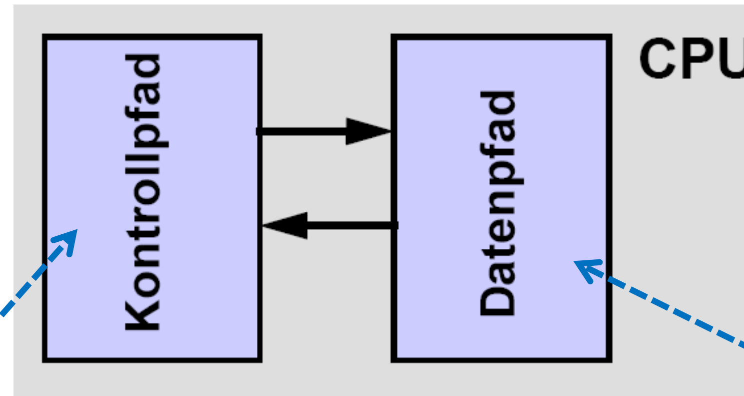
# Prinzpieller Ablauf



# Digitale Implementierung

# Einzeltakt-Implementierung

**Alle Teiloperationen werden in einem Takt ausgeführt.**

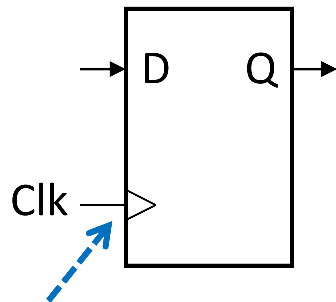



Kombinatorische Schaltung (keine Register); implementiert die Struktur des Petri-Netzes.

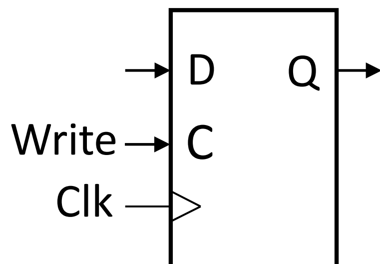
Implementiert alle Operationen, die den Transitionen im Petri-Netz zugeordnet sind.

# Synchroner Schaltungsentwurf

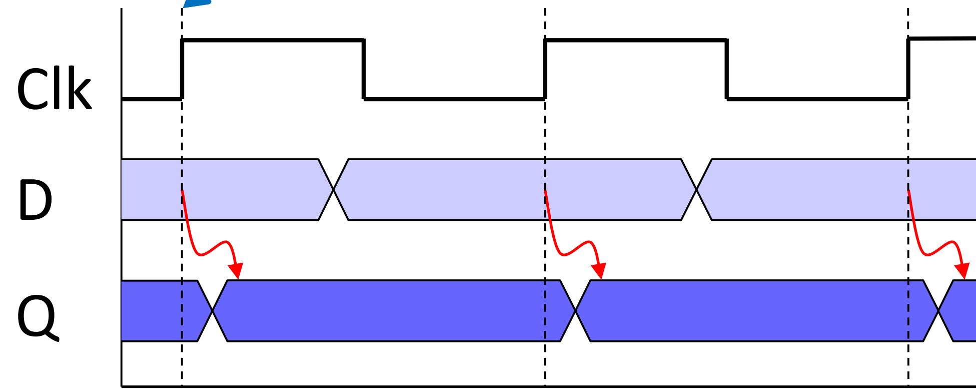
## *Register*



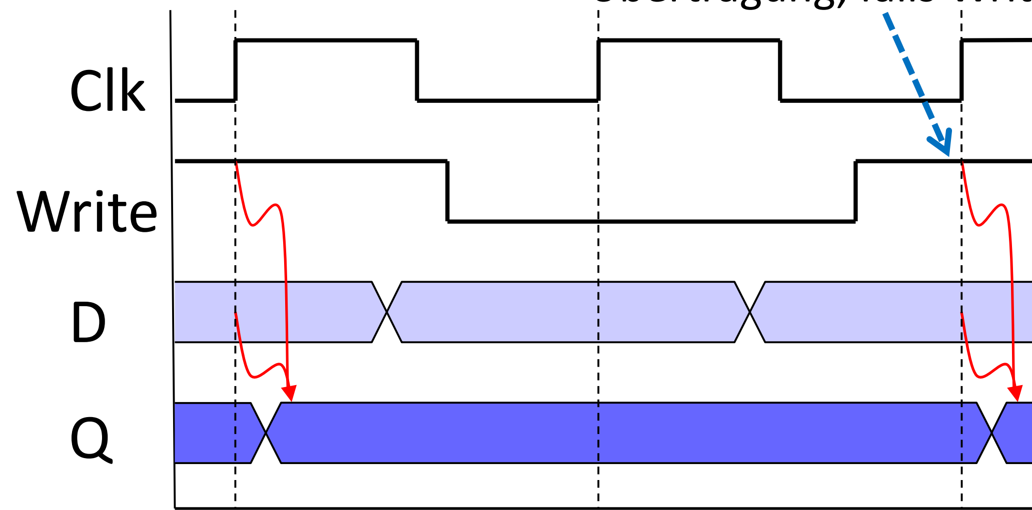

Taktsignal wird in den folgenden Darstellungen oft weggelassen.




Zur steigenden Taktflanke wird der Wert an D auf den Ausgang Q übertragen.



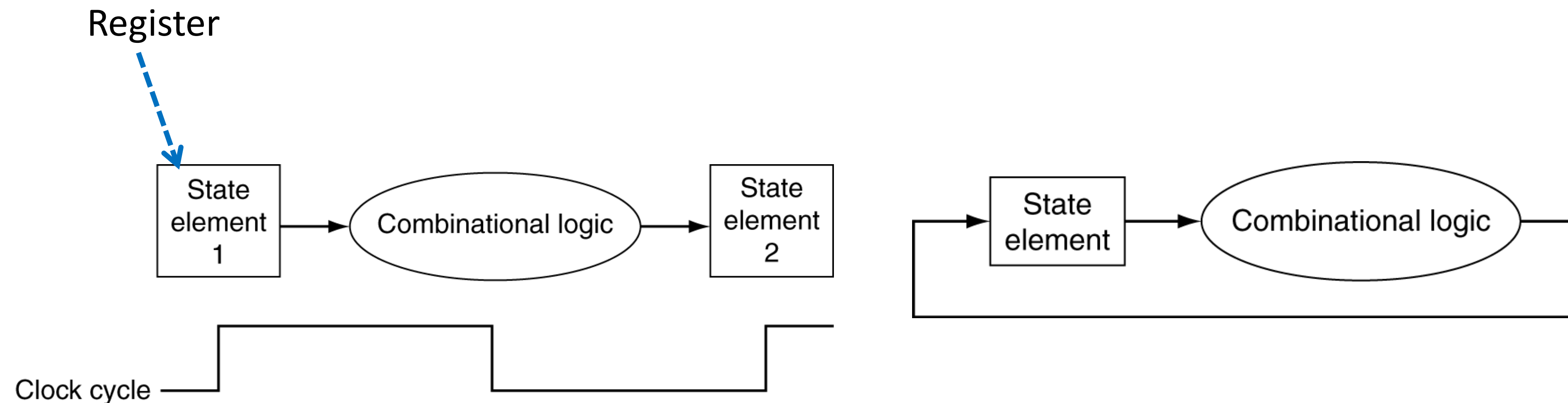

Übertragung, falls Write == 1



# Synchroner Schaltungsentwurf

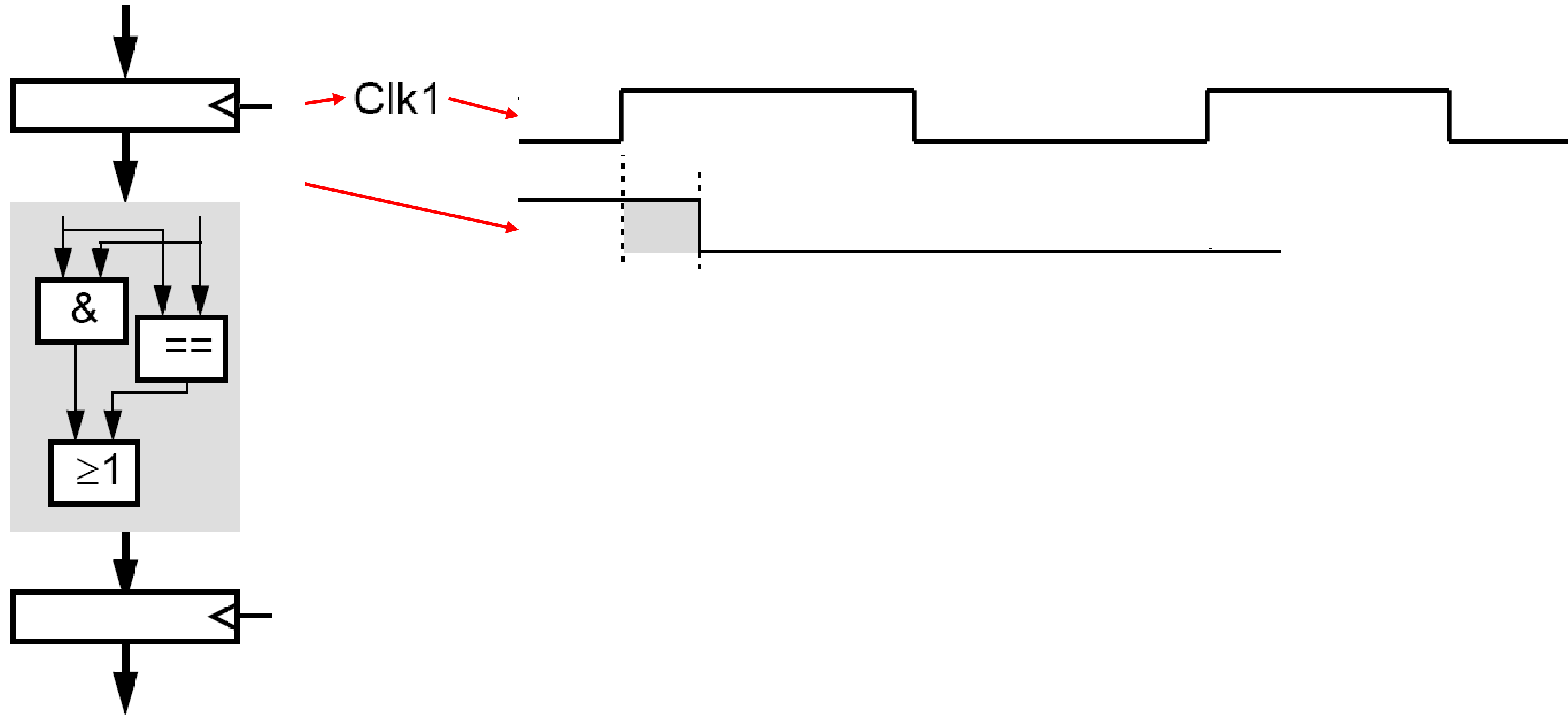
## *Einzeltaktsystem*

- Kombinatorische Logik verarbeitet Daten zwischen den aktiven Taktflanken.
- Die längste Verzögerungszeit zwischen Registerausgängen und Registereingängen bestimmt die minimale Taktperiode.



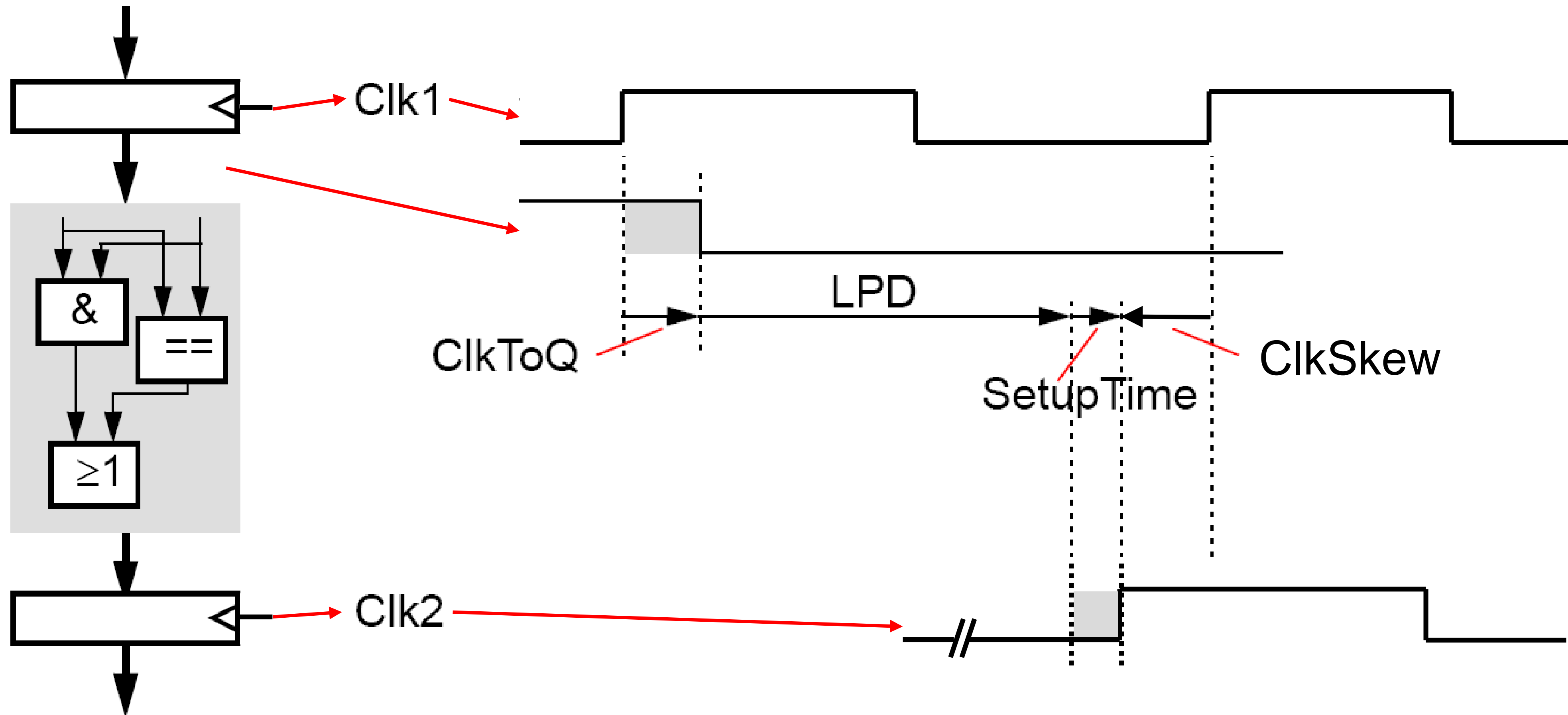
# Synchroner Schaltungsentwurf

***Bestimmung der minimalen Taktperiode:***



# Synchroner Schaltungsentwurf

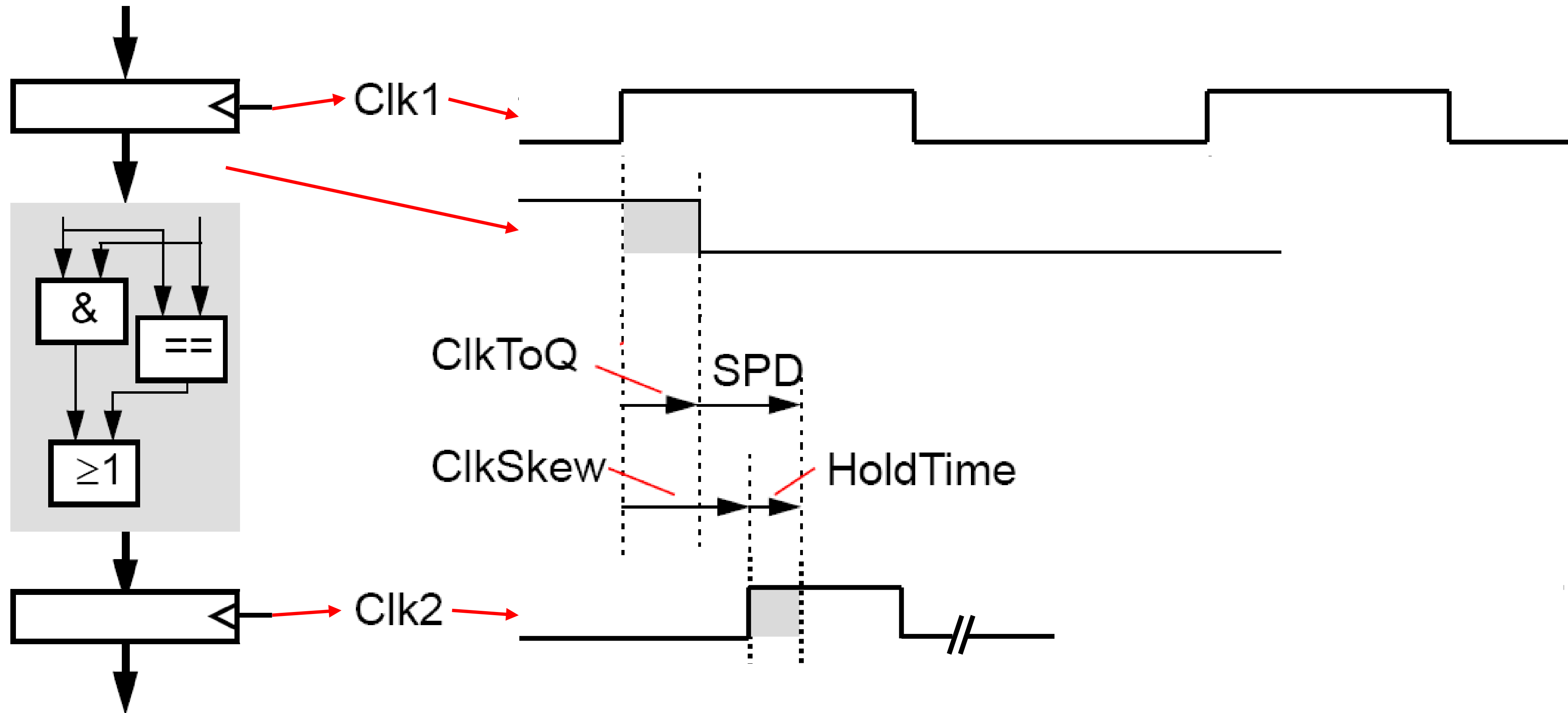
***Bestimmung der minimalen Taktperiode:***




$$ClkPeriod \geq ClkToQ + LPD + SetupTime - ClkSkew$$

# Synchroner Schaltungsentwurf

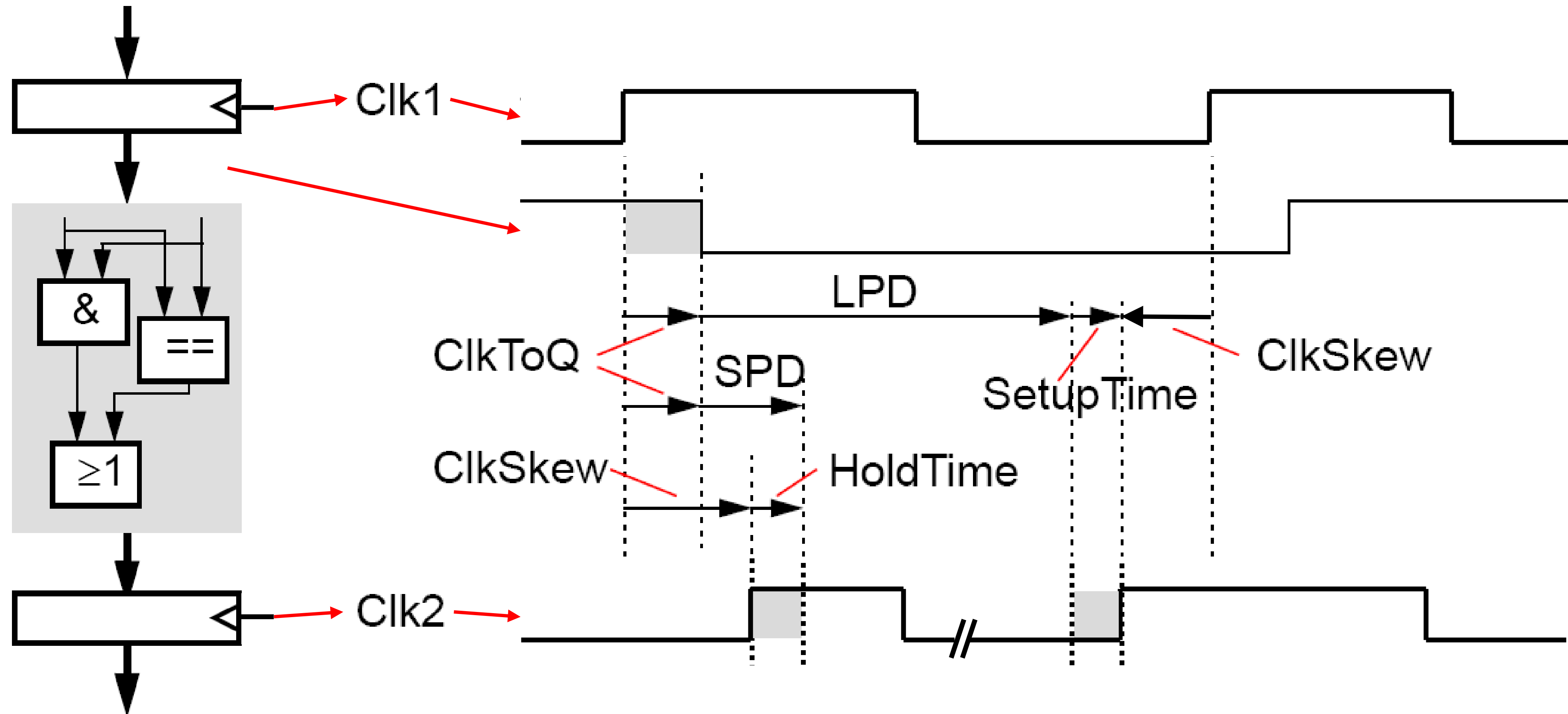
***Bestimmung der minimalen Taktperiode:***




$$ClkToQ + SPD \geq ClkSkew + HoldTime$$

# Synchroner Schaltungsentwurf

***Bestimmung der minimalen Taktperiode:***




$$ClkToQ + SPD \geq ClkSkew + HoldTime$$

$$ClkPeriod \geq ClkToQ + LPD + SetupTime - ClkSkew$$

# Synchroner Schaltungsentwurf

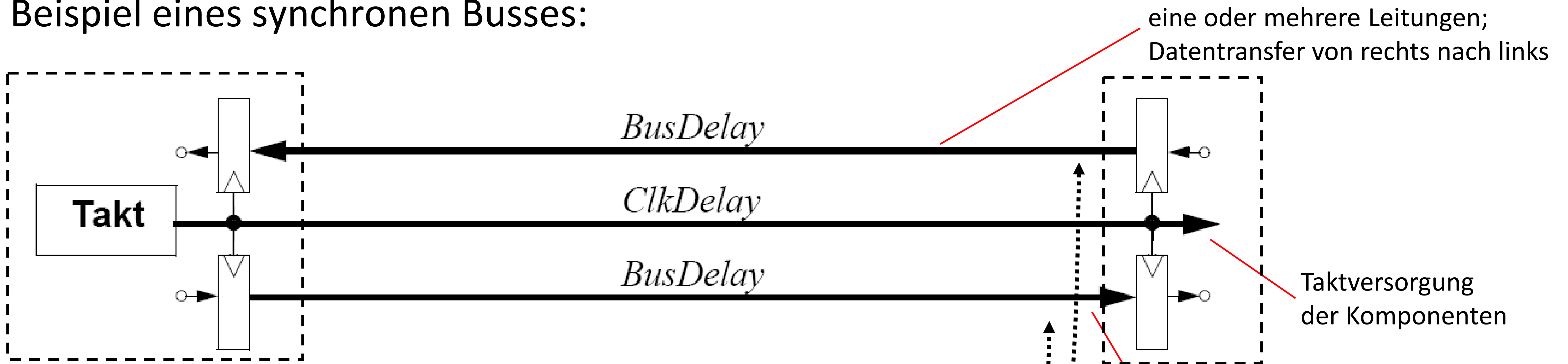
***Bezeichnungen im Zeitablauf*:**

- *ClktoQ*: Verzögerung zwischen aktiver Taktflanke und gültigem Registerausgangssignal
- *ClkSkew*: Betrag der zeitlichen Verschiebung des Taktsignals zwischen Registern
- *ClkPeriod*: Periode des Taktsignals
- *SetupTime, HoldTime*: Zeitintervalle vor bzw. nach der aktiven Taktflanke, in denen das Eingangssignal eines Registers gültig sein muss
- *SPD, LPD*: Kürzeste bzw. längste Verzögerungszeit zwischen Ausgang und Eingang von Registern über kombinatorische Komponenten

# Anwendung auf Bus

**Ein *Bus ist eine gemeinsam genutzte* Kommunikationsverbindung, z.B. zur Verbindung von Prozessor, Speicher und Ein- und Ausgabeeinheiten.**

Beispiel eines synchronen Busses:

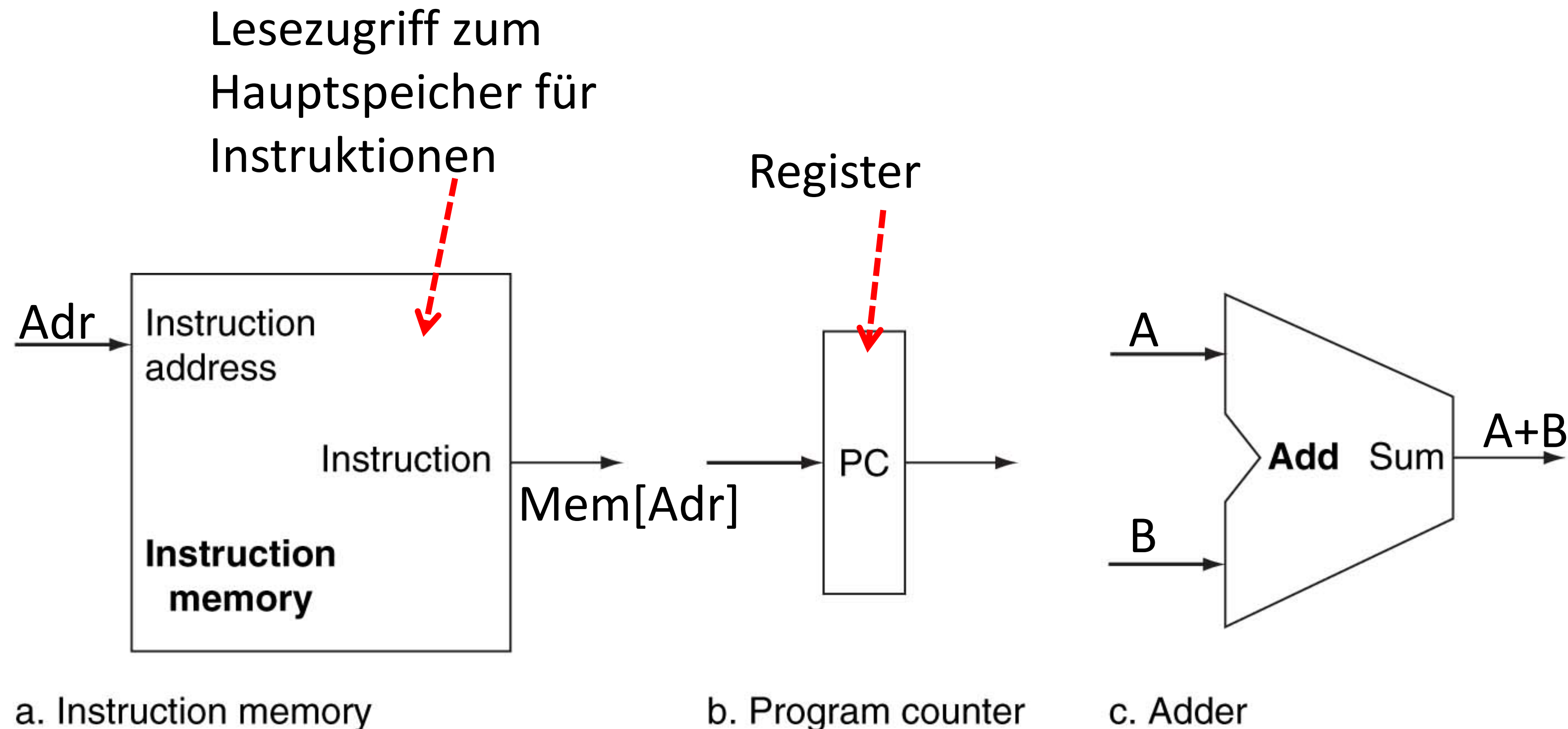



eine oder mehrere Leitungen;
Datentransfer von links nach rechts

$$ClkToQ + BusDelay - HoldTime \geq ClkDelay$$
$$ClkToQ + BusDelay + SetupTime - ClkPeriod \leq ClkDelay$$

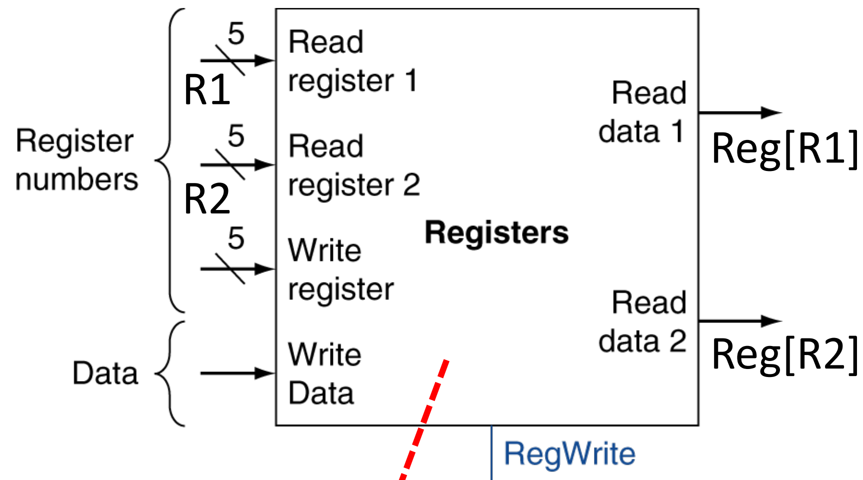
$$HoldTime - ClkToQ - BusDelay \leq ClkDelay$$
$$ClkPeriod - ClkToQ - BusDelay - SetupTime \geq ClkDelay$$

# Komponenten im Datenpfad

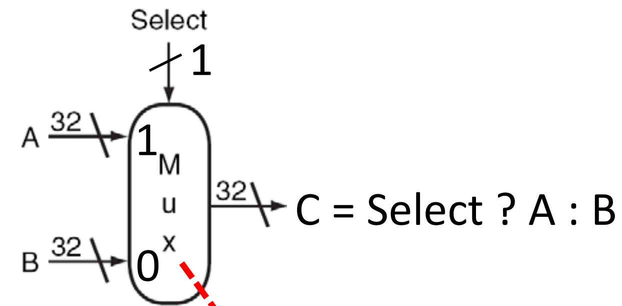



- Alle Verbindungen haben eine Breite von 32 Bit (Wortbreite) wenn nicht anders angegeben.
- Instruktionsspeicher und Addierer sind als kombinatorische Schaltungen modelliert.

# Komponenten im Datenpfad



a. Registers



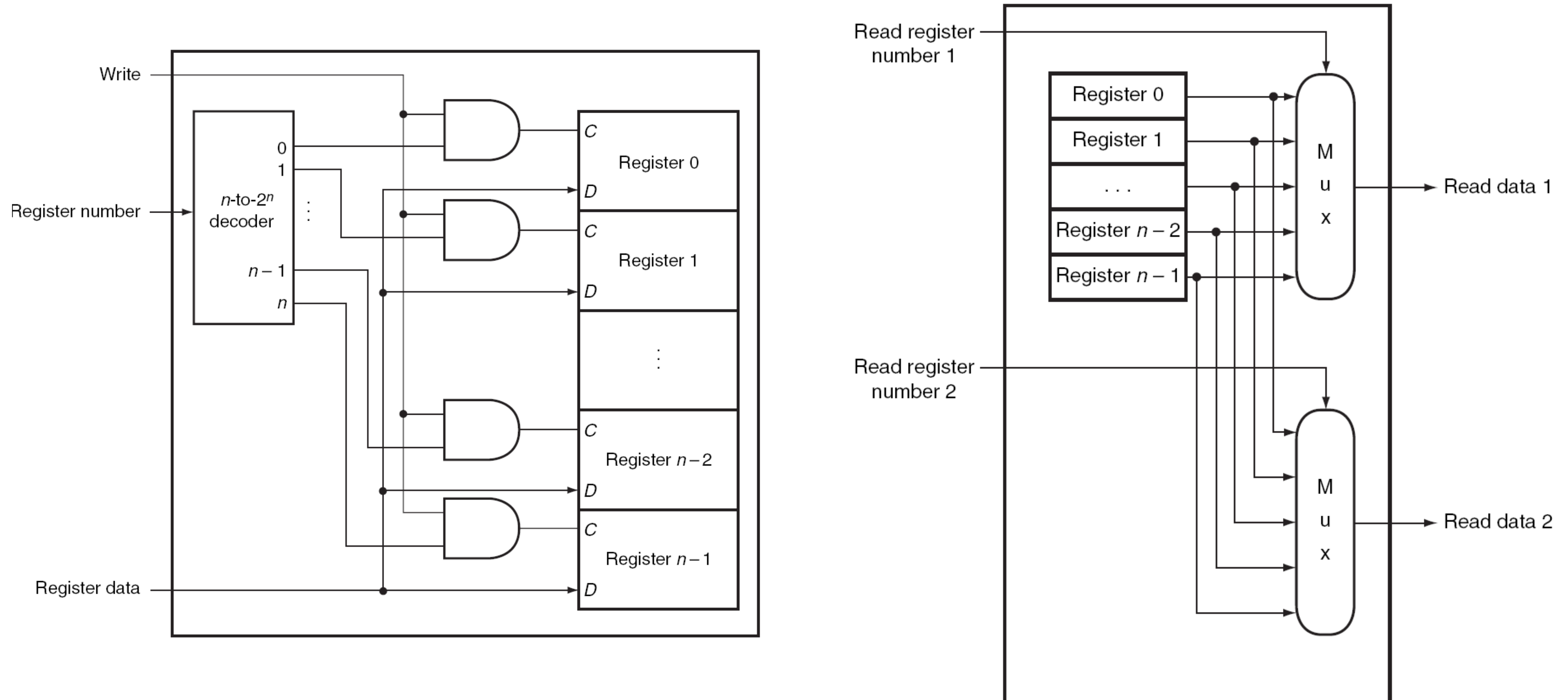
b. Multiplexer

***Registerfeld:***
kombinatorisches Lesen, getaktetes Schreiben

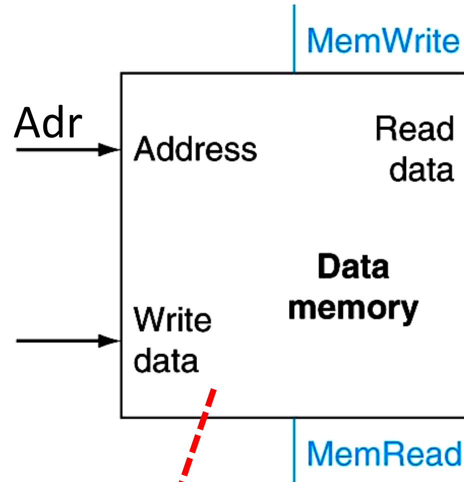
***Multiplexer:***
Auch mehr als 2 Eingänge möglich; dann erfolgt eine Binärkodierung des ausgewählten Eingangs.

# Komponenten im Datenpfad

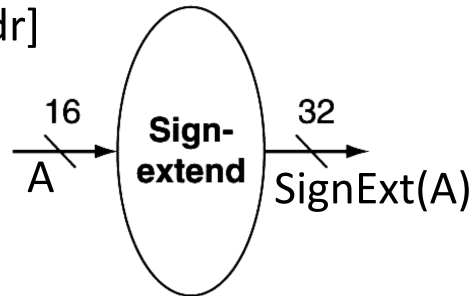
## Detaillierte Schaltung des Registerfeldes:



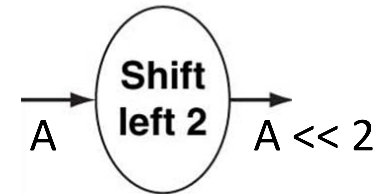
# Komponenten im Datenpfad




a. Data memory unit



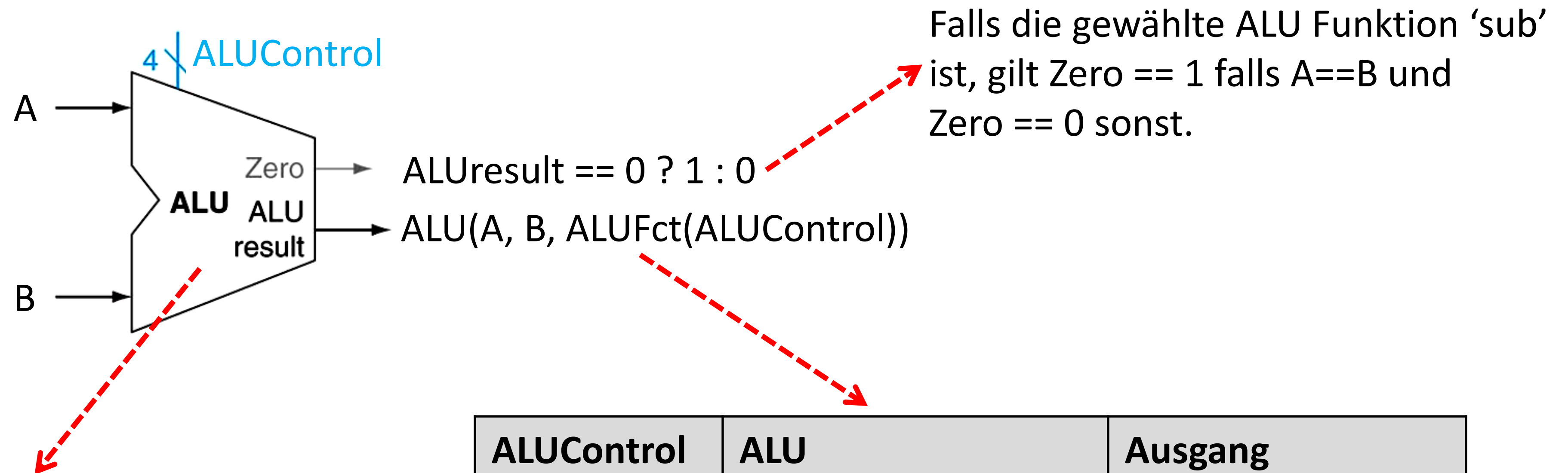

b. Sign extension unit




c. Shift

***Hauptspeicher:***
kombinatorisches Lesen falls MemRead==1,
getaktetes Schreiben falls MemWrite==1

# Komponenten im Datenpfad

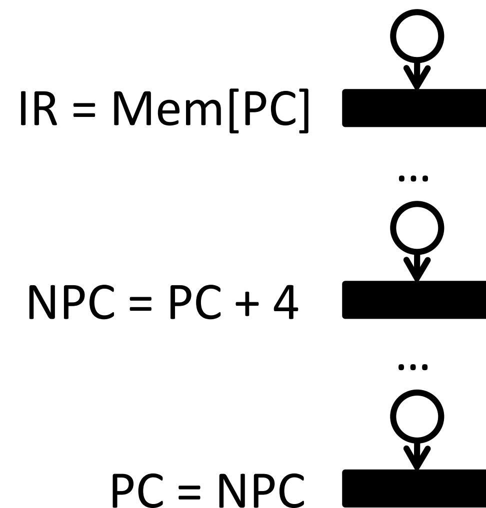


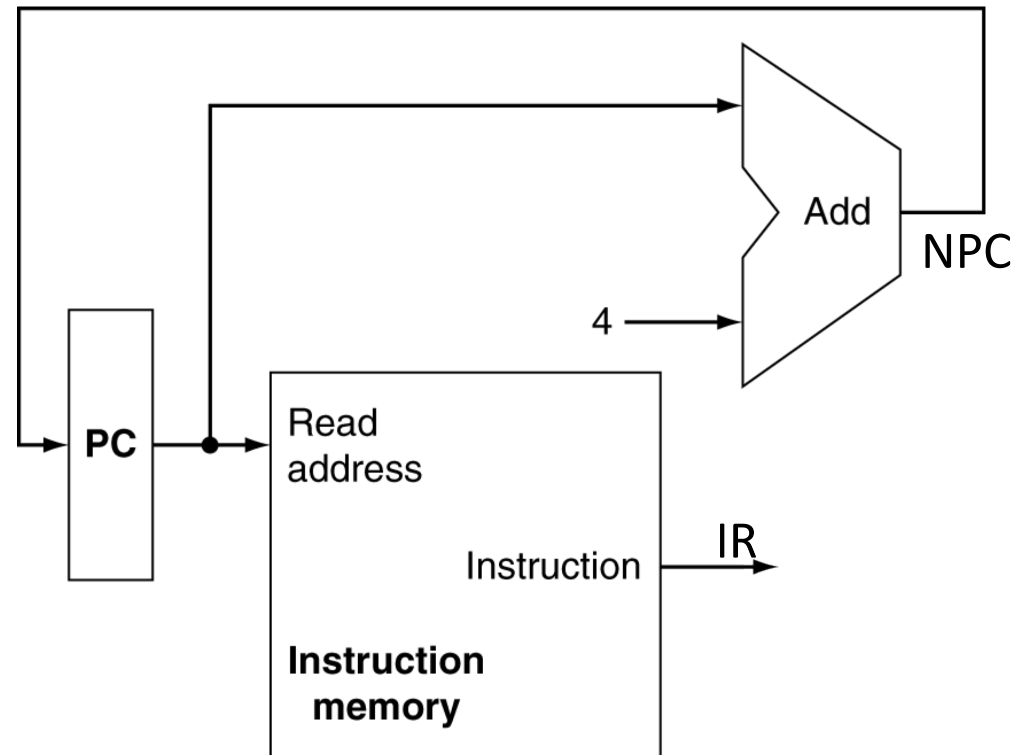

***ALU:***
Arithmetisch Logische Einheit

| ALUControl Eingang | ALU Funktion | Ausgang |
|---|---|---|
| 0010 | ´add´ | A+B |
| 0110 | ´sub´ | A-B |
| 0000 | ´AND´ | A & B (bitweise) |
| 0001 | ´OR´ | A \| B (bitweise) |
| 0111 | ´setOnLessThan´ | A<B ? 1 : 0 |

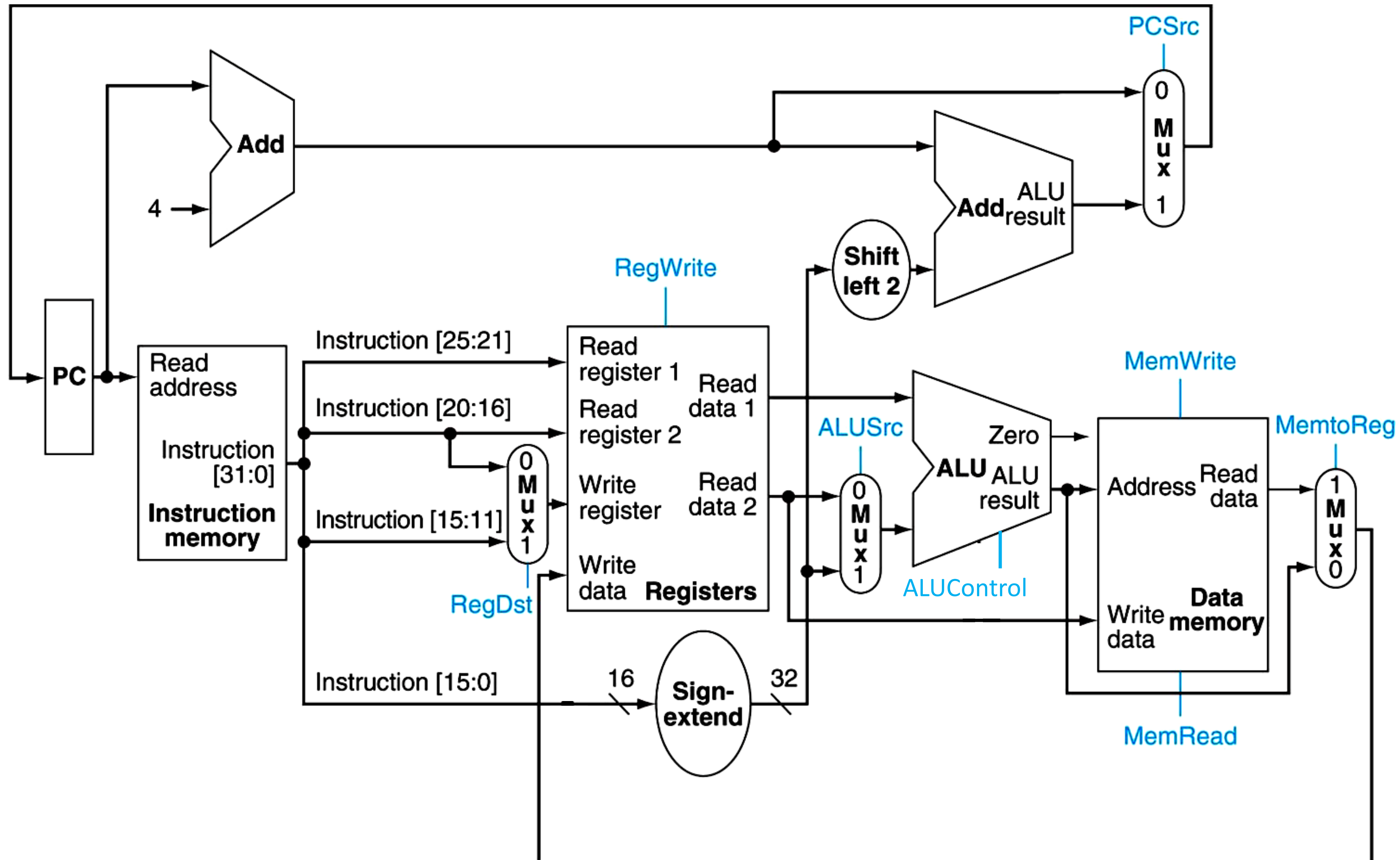
# Konstruktion des Datenpfades

- Schrittweise Erweiterung der Schaltung für jede Operation im Petri-Netz. Signale entsprechen den Variablen.
- Einfügen von Multiplexern beim Zusammenlegen von Signalen.
- *Beispiel:* Instruktion-Fetch (IF)

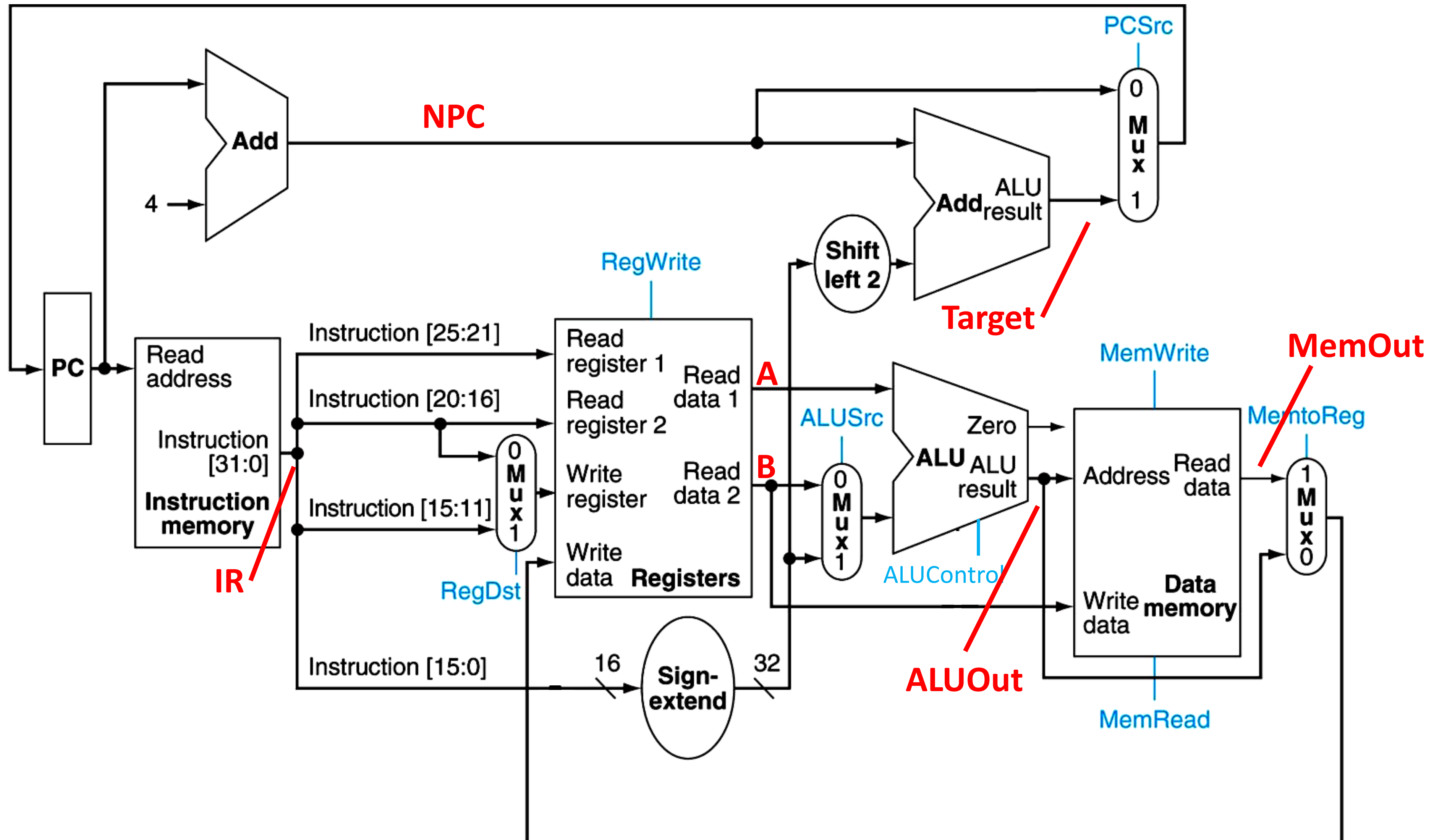


# Einzeltakt-Datenpfad



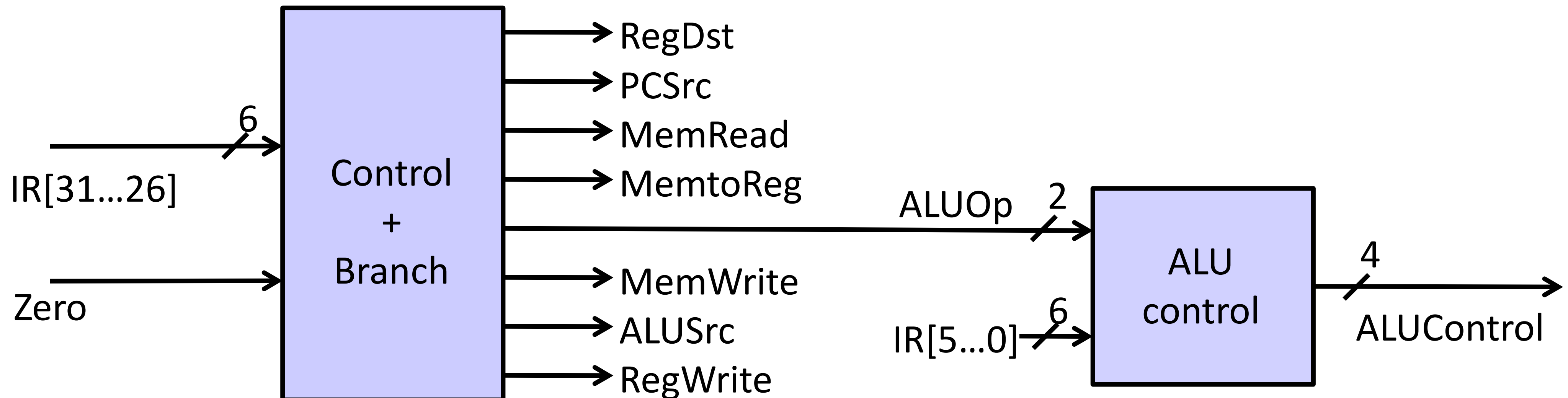
# Einzeltakt-Datenpfad



# Kontrollpfad

- Der  Kontrollpfad bildet die Struktur des Petri-Netzes ab. Er ist dafür verantwortlich, dass der Datenpfad die korrekten Operationen ausführt.
- Bei der Einzeltakt-Implementierung ist der Kontrollpfad zustandslos.

*Blockdiagramm:*



# Kontrollpfad

- Der Baustein ***Control*** bestimmt die Steuerungssignale für den Datenpfad, die Zwischensignale für den Baustein *ALU Control* sowie das Signal „Branch“.
- Das Steuerungssignal „PCSrc“ ergibt sich aus `PCSrc = Branch && Zero` .

| Instruktion | IR[31…26] | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp |
|---|---|---|---|---|---|---|---|---|---|
| R-Typ | 000000 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 10 |
| lw | 100011 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 00 |
| sw | 101011 | X | 1 | X | 0 | 0 | 1 | 0 | 00 |
| beq | 000100 | X | 0 | X | 0 | 0 | 0 | 1 | 01 |

Eingangssignale (IR[31…26]) — Ausgangssignale (RegDst … ALUOp)

# Kontrollpfad

- Der Baustein ***ALU Control*** bestimmt die Steuersignale “ALUControl” für die ALU. Dazu kodiert der Baustein *Control* den Operationscode der Instruktion (IR[31…26]) in das Signal “ALUOp”.
- Mit Hilfe des Funktionscodes der Instruktion (IR[5…0]) werden daraus die Steuersignale “ALUControl” generiert:

*Control* *ALU Control*

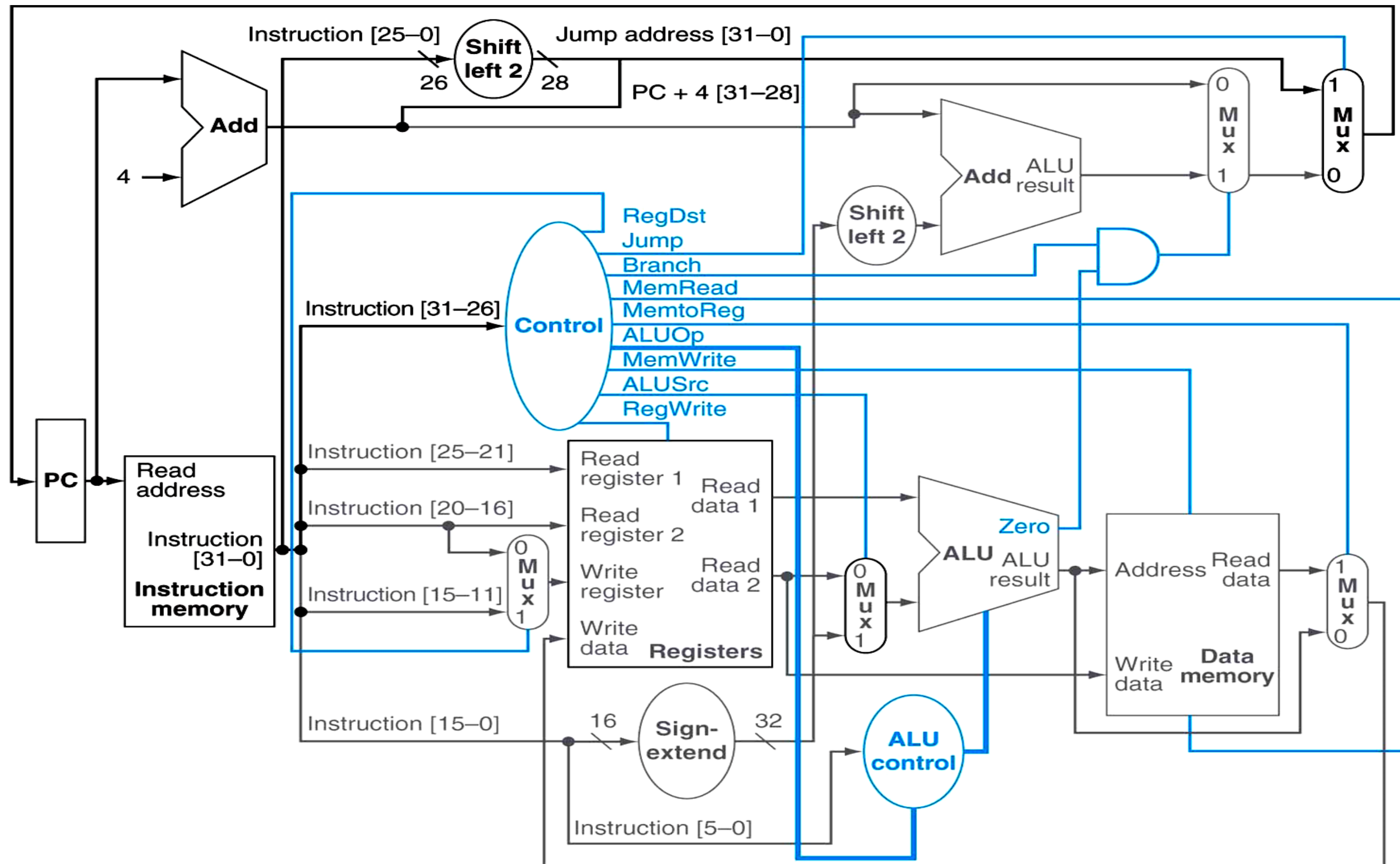
| Instruktion | opcode IR[31…26] | ALUOp | functcode IR[5…0] | ALU Funktion | ALUControl |
|---|---|---|---|---|---|
| load word (lw) | 100011 | 00 | XXXXXX | ´add´ | 0010 |
| store word (sw) | 101011 | 00 | XXXXXX | ´add´ | 0010 |
| branch equal (beq) | 000100 | 01 | XXXXXX | ´subtract´ | 0110 |
| add (add) | 000000 | 10 | 100000 | ´add´ | 0010 |
| subtract (sub) | | | 100010 | ´subtract´ | 0110 |
| and (and) | | | 100100 | ´AND´ | 0000 |
| or (or) | | | 100101 | ´OR´ | 0001 |
| set-on-less-than (slt) | | | 101010 | ´setOnLessThan´ | 0111 |

# Datenpfad und Kontrollpfad

Add
4
Add ALU result
Shift left 2
M u x 0 1
PCSrc
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign-extend
32
ALU control
ALUControl
Instruction [5–0]

# Datenpfad und Kontrollpfad

***Erweiterung*** mit der Instruktion jump (j):



# Rechenleistung

- Der längste Pfad bestimmt die minimale Taktperiode.
  - Die lw-Instruktion dauert länger als alle anderen:
    Instruction Fetch
    Register Fetch
    Execute on ALU
    Memory Access
    Register Write
- Aus technischen Gründen kann man die Taktperiode nicht dynamisch anpassen.
- Effizienzsteigerung durch ***Pipelining***.