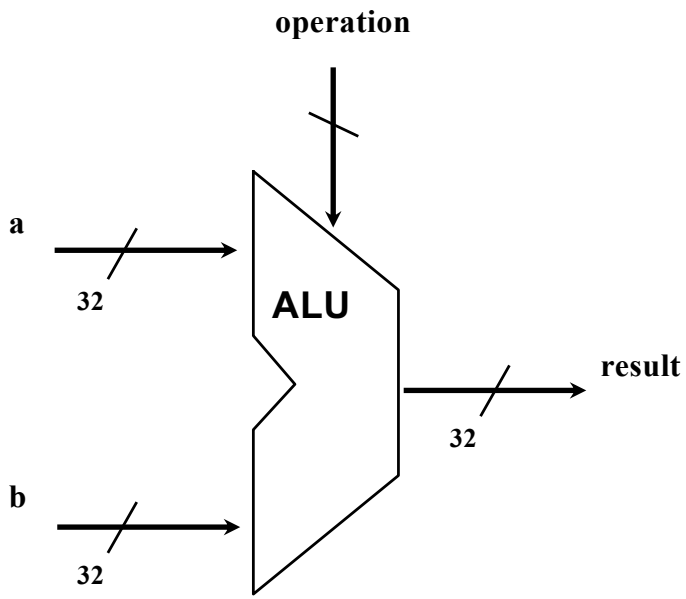
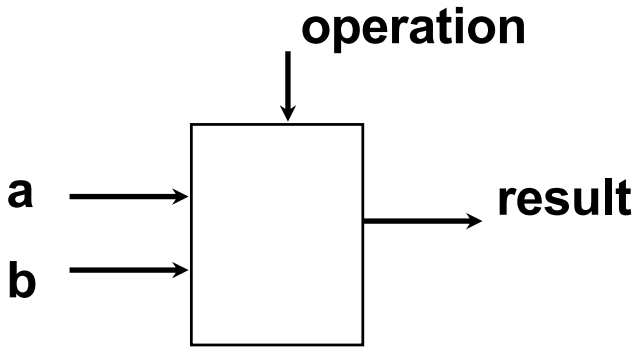
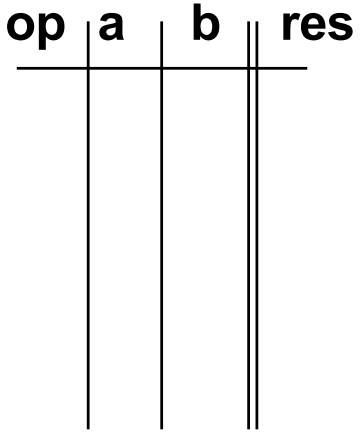
Lasst uns einen Prozessor bauen

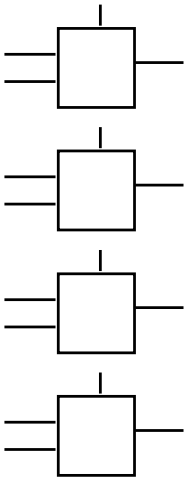


Eine ALU (arithmetic logic unit)

- Start: Lasst uns eine ALU bauen welche die **zwei** Befehle unterstützt: `andi` und `ori`.
- Bauen **1-Bit-**ALU und nutzen 32 Stück davon

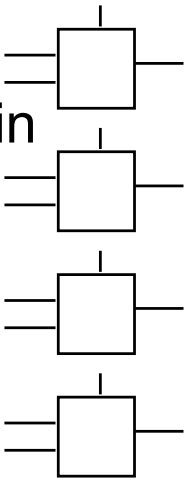






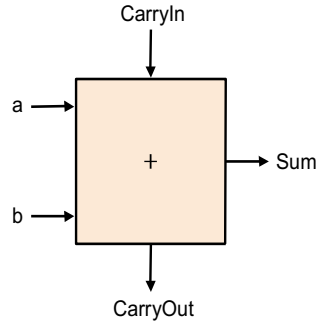
- Mögliche Implementation: Programmierbarer Logikbaustein
  - **PLA**: **P**rogrammierbare **L**ogische **A**nordnung
  - Nur einmal programmierbare Matrix (Brennvorgang)
  - Ermöglicht Formeln der Aussagenlogik (boolesche Gleichungen) zu realisieren (DNF: Disjunktive Normalform).

    DNF: $(\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge C)$



Verschiedene Implementationen

- Nicht triviale Entscheidung: Wie am “besten” etwas bauen
  - Möchten nicht zu viele Inputs an einem einzigen (Logik-)Gatter
  - Möchten nicht durch zu viele Gatter gehen müssen
  - Für unsere Ansprüche, ist ein einfaches Verständnis wichtig
- Eine 1-Bit ALU für Additionen:



$c_{out} = (a \vee b) \wedge (a \vee c_{in}) \wedge (b \vee c_{in})$

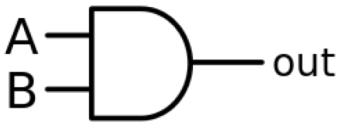
$sum = a \text{ xor } b \text{ xor } c_{in}$

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

- Wie bauen wir eine 1-bit ALU für **add**, **and**, und **or**?
- Wie bauen wir eine 32-bit ALU?

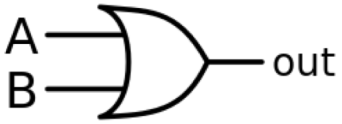
Typen von Logikgattern und Symbolik

- Und-Gatter



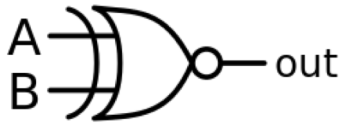
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

- Oder-Gatter



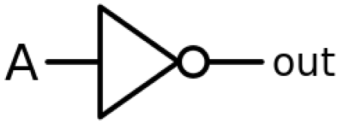
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

- XOR-Gatter



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

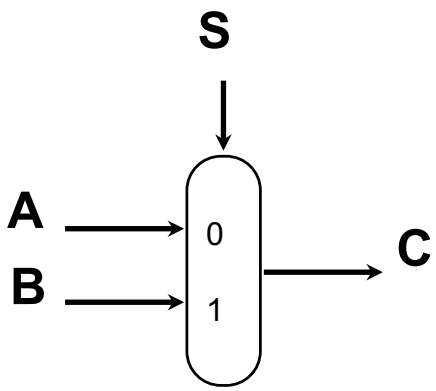
- Nicht-Gatter



A	Y
0	1
1	0

Noch ein Bauteil: Der Multiplexer

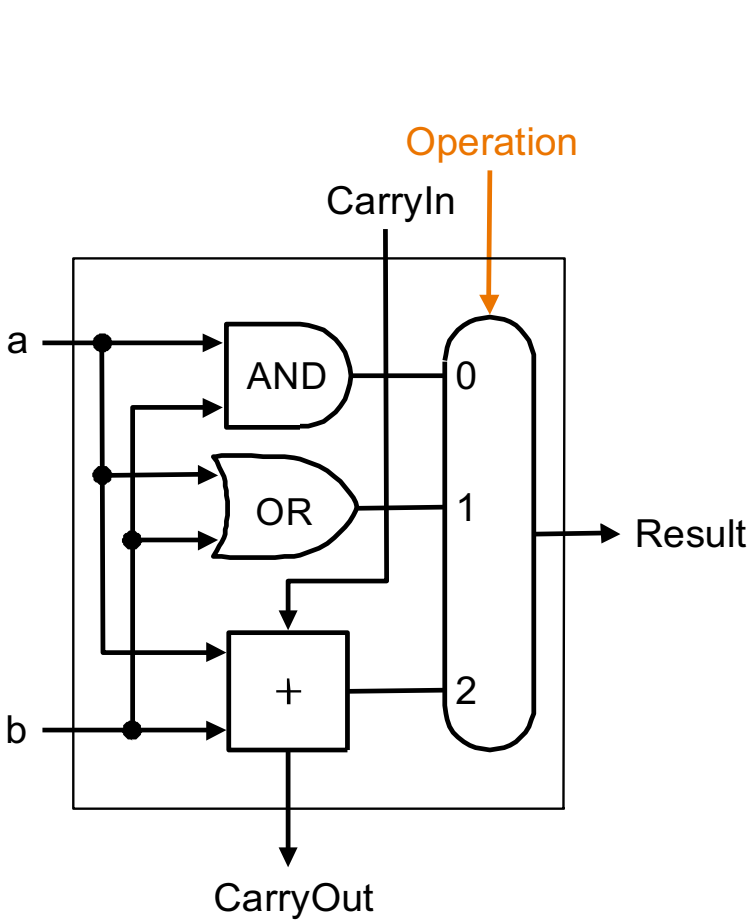
- Wählt, basierend auf dem Steuereingang, einen Input als Output,

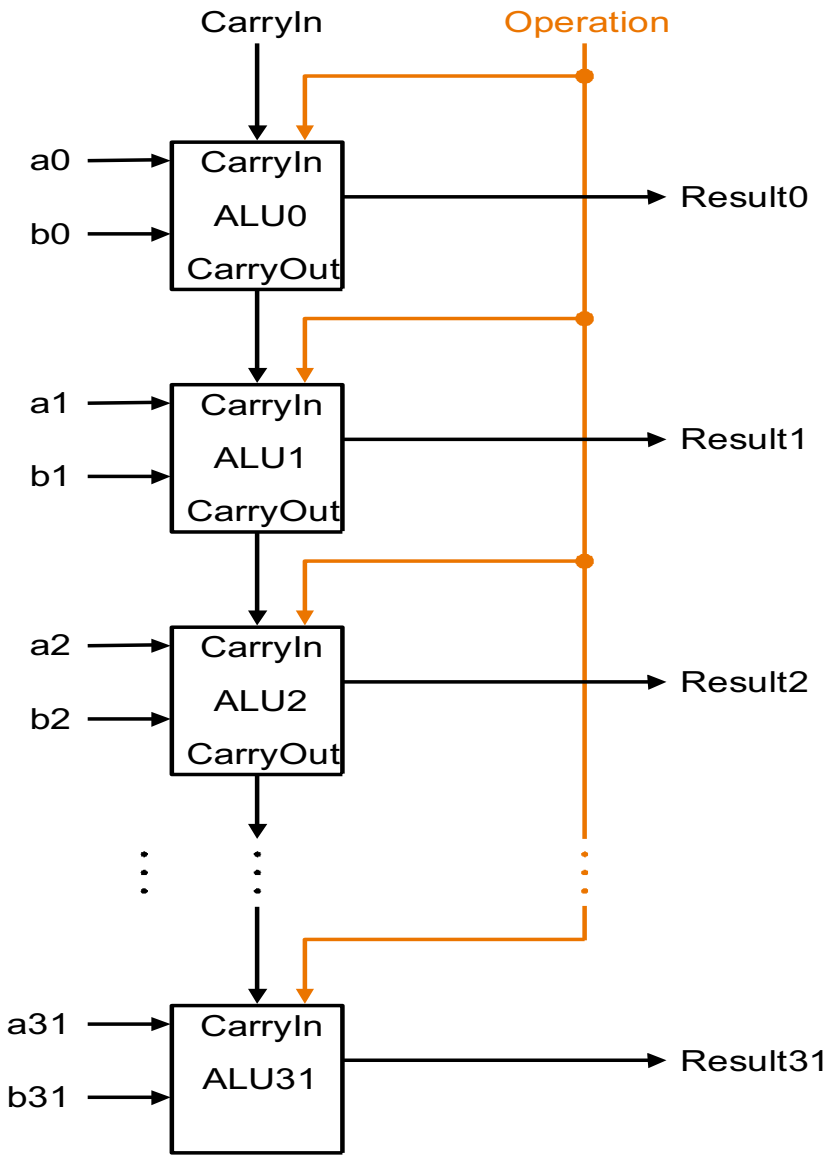


Hinweis: Wir nennen das einen 2-Input Mux auch wenn es 3 Inputs sind

- Bauen wir unsere ALU mit einem MUX:

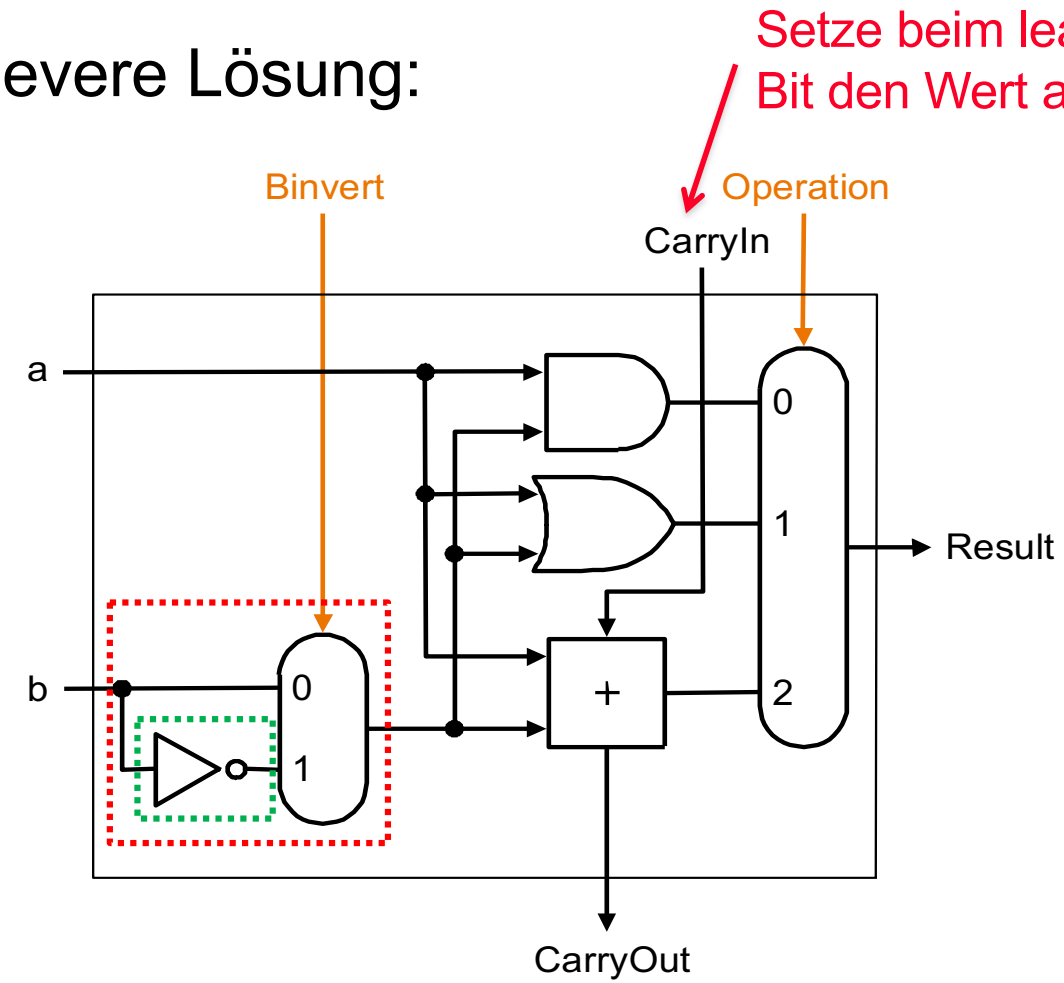
Bauen einer 32 bit ALU | Carry-Ripple-Addierer





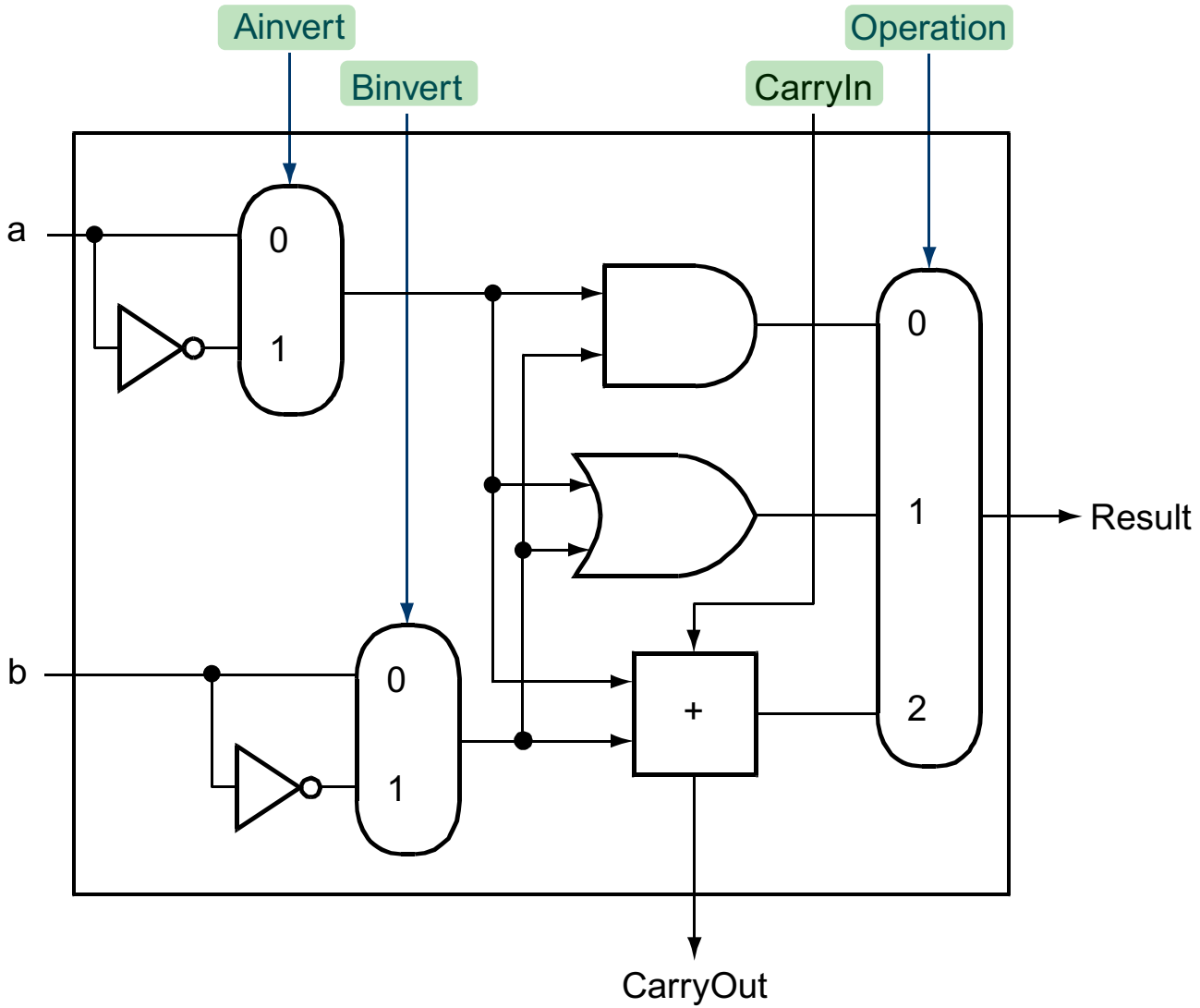
Wie sieht es mit Substraktionen aus (a – b) ?

- Zweierkomplement Ansatz: Invertiere b und addiere a
- Aber wie Invertieren?
- Eine clevere Lösung:



NOR Funktion hinzufügen

- Können noch a invertieren. Wie erhalten wir “a NOR b” ?

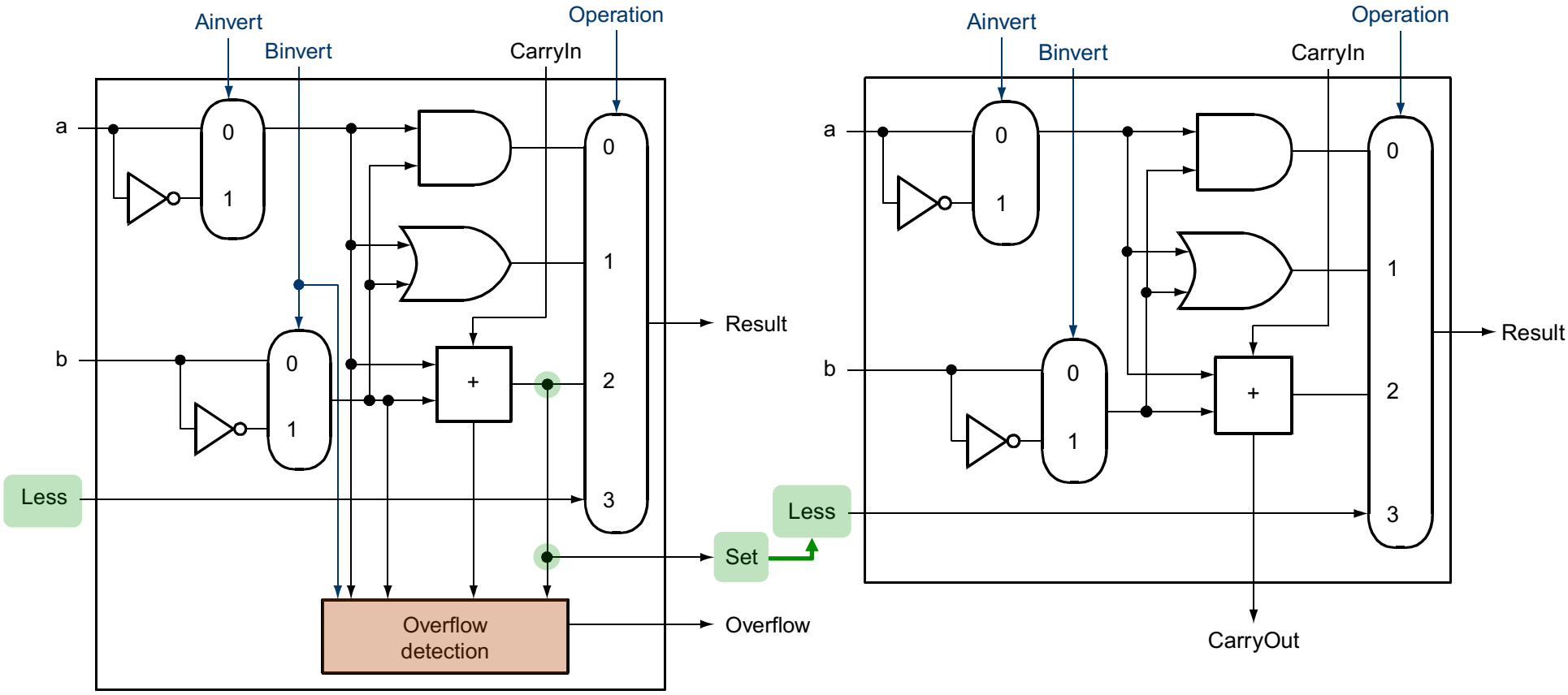


Anpassen der ALU für MIPS

- Benötigen den set-on-less-than Befehl (slt)
  - Zu Erinnerung: slt ist ein arithmetischer Befehl
  - Erzeugt ein 1 wenn rs < rt, sonst 0
  - Nutzt Subtraktion : (a-b) < 0 impliziert a < b
- Benötigen noch Test auf Gleichheit (beq $t5, $t6, Lbl)
  - Nutzen Subtraktion : (a-b) = 0 impliziert a = b

Unterstützung von slt

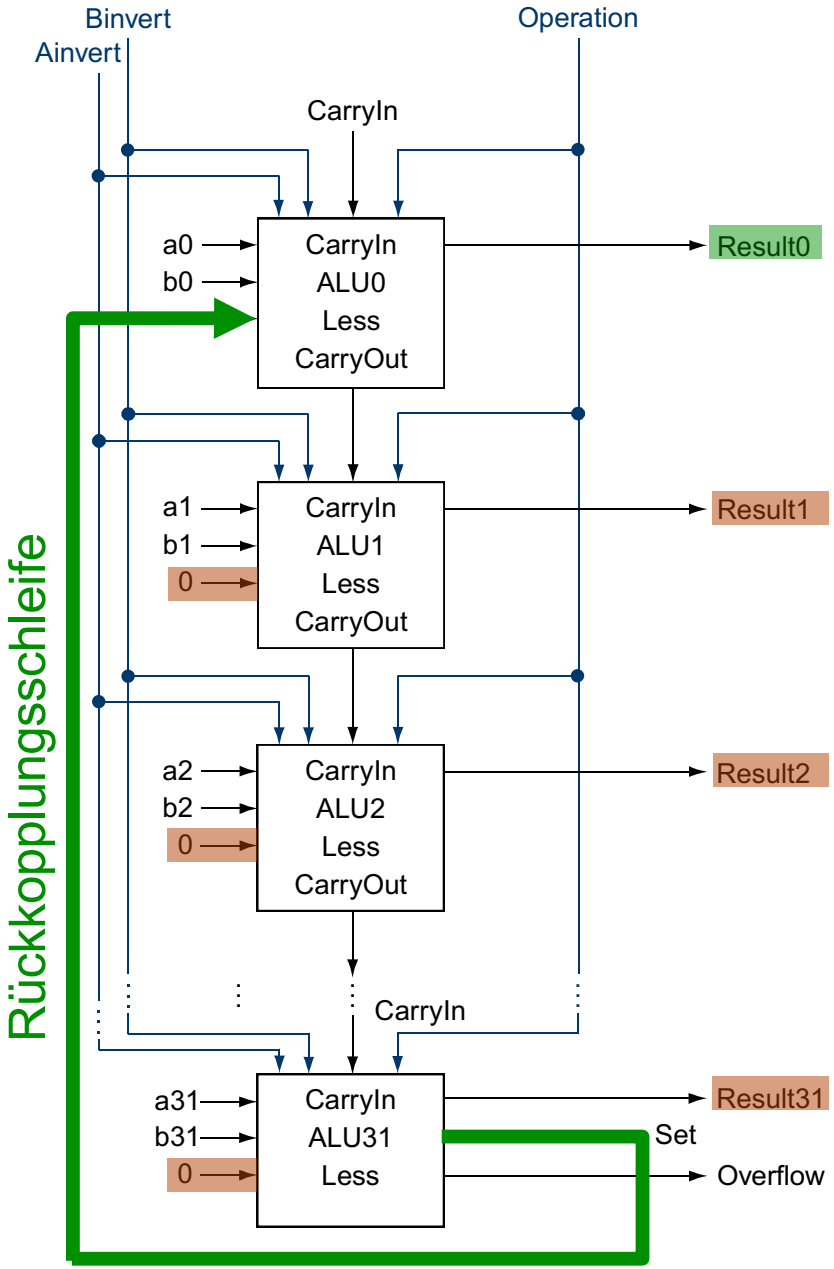
- Erkennen wir die Idee dahinter?



Nutzen diese ALU für das *most significant* Bit

Diese für alle anderen Bits

Unterstützung von slt

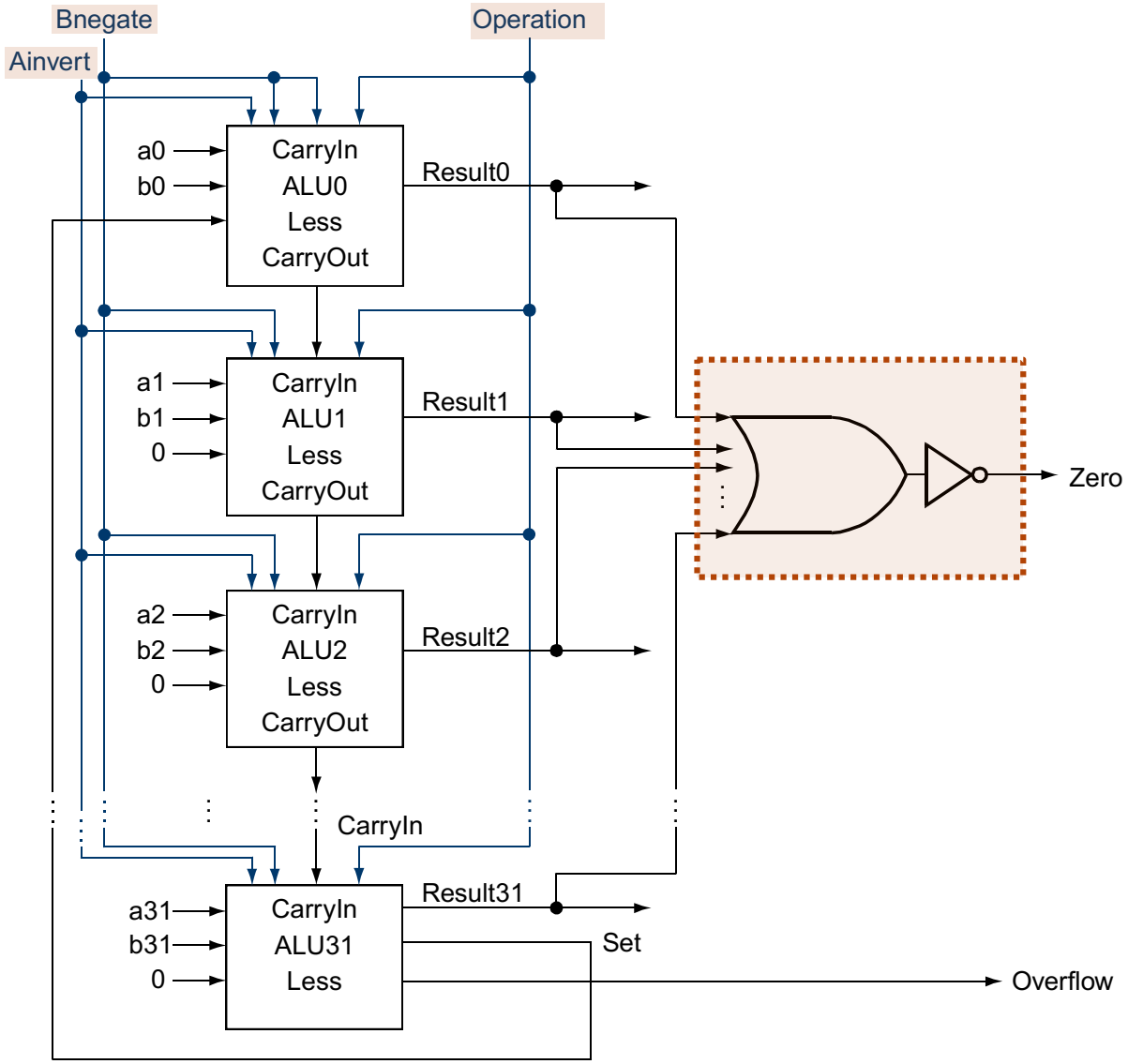


Test auf Gleichheit

- Steuerleitung:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

- Notiz: *Zero* ist 1 wenn alle Resultate 0 sind.



Fazit

- Wir können eine ALU bauen welche das MIPS Befehlsset unterstützt:
  - Schlüsselidee: Multiplexer zur Auswahl des benötigten Outputs
  - Effiziente Subtraktion durch Nutzen der Zweierkomplements
  - Wir können mit 1-bit ALUs eine a 32-bit ALU bauen (Replikation)
- Wichtige Punkte zur Hardware
  - Es sind immer alle Gatter in Betrieb
  - Die Anzahl Inputs beeinflusst die Geschwindigkeit eines Gatters
  - Die Anzahl in Serie geschalteter Gatter beeinflusst die Geschwindigkeit der Schaltung. (“kritischer Pfad” oder “deepest level of logic”)
- Unser Hauptfokus: Verstehen, dennoch:
  - Kluge Anpassungen können die Performanz verbessern (Ähnlich dem nutzen besserer Algorithmen in der Software)

ALU Zusammenfassung

- Wir können eine ALU bauen welche die MIPS Addition unterstützt
- User Fokus ist Verstehen, nicht Performance
- Echte Prozessoren nutzen ausgeklügeltere Techniken für die Arithmetik (e.g. nicht den «ripple carry adder»)
- Wo Performance nicht kritisch ist erlaubt Hardwarebeschreibungs-sprache eine Automatisierung der Herstellung von Hardware

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

Basic MIPS Architecture Review

[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

Review: DIE Performance Gleichung

- Unsere Grundgleichung für die Performance ist nun:

$$\text{(User) CPU Zeit} = \text{IC} \times \text{CPI} \times \text{Taktperiode}$$

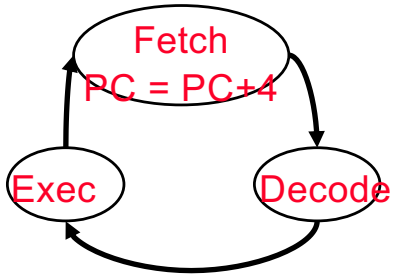
or

$$\text{(User) CPU Zeit} = \frac{\text{IC} \times \text{CPI}}{\text{Taktrate}}$$

- Diese Gleichungen zeigen die drei Schlüsselfaktoren welche Einfluss auf die Performance haben
  - Messen der **CPU Zeit** durch ausführen des Programm
  - Die **Taktrate** ist normalerweise gegeben
  - Messen der gesamten Anzahl an Befehlen (**IC**) durch Profiler/Simulatoren ohne alle Implementationsdetails zu kennen
  - **CPI** variiert je nach Befehlstyp und ISA Implementation. Hier müssen wir die Implementationsdetails kennen

Der Prozessor: Datenpfad & Steuerung (Control)

- Unsere MIPS Implementation ist vereinfacht:
  - Speicherbefehle/Datentransferbefehle: `lw, sw`
  - Arithmetische & logische Befehle: `add, sub, and, or, slt`
  - Sprung (und Verzweigungsbefehle): `beq, j`
- Generische Implementation
  - Nutzen des Befehlszähler (PC) zum liefern der Befehlsadresse und «**fetch**» des Befehls vom Hauptspeicher (und update des PC)
  - **«Decode»** des Befehls (und 2 Register lesen)
  - **«Execute»** Befehls

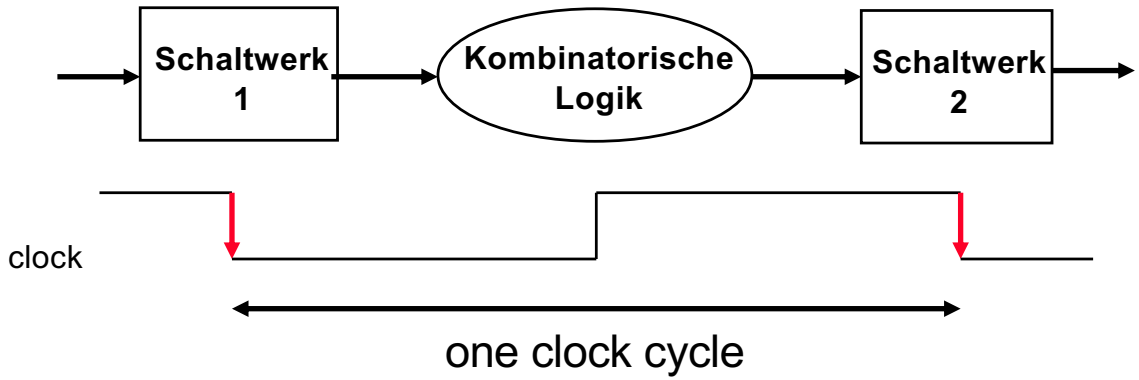


- Alle Befehle (ausser `j`) nutzen die ALU nach dem lesen der Register

Wie nutzen die Befehle der drei Befehlskategorien die ALU?

Taktverfahren (clocking methodology)

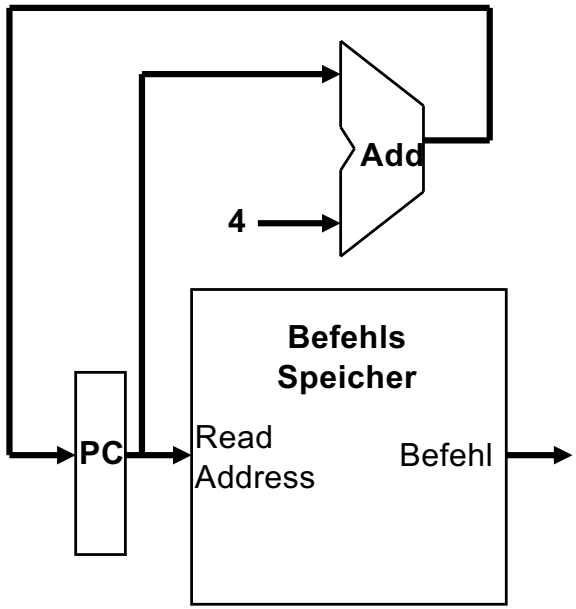
- Das (systemweite) Taktsignal definiert wann Signale gelesen und wann geschrieben werden können
  - Eine flankengesteuerte Methode
- Typische Ausführung (MIPS: 2 verschiedenen Arten von Logikbausteinen)
  - Lese Inhalt eines **Schaltwerk** (State Element)
  - Sende durch **Schaltnetz** (kombinatorische Logikschaltung)
  - Schreibe Resultat in ein oder mehrere Schaltwerke



- Schaltwerke werden bei jedem Taktzyklus überschrieben.
  - Schreiben passiert nur wenn und bei fallender Signalflanke
  - Schreiben muss vor dem nächsten Zyklus abgeschlossen sein

Fetching (von Befehlen)

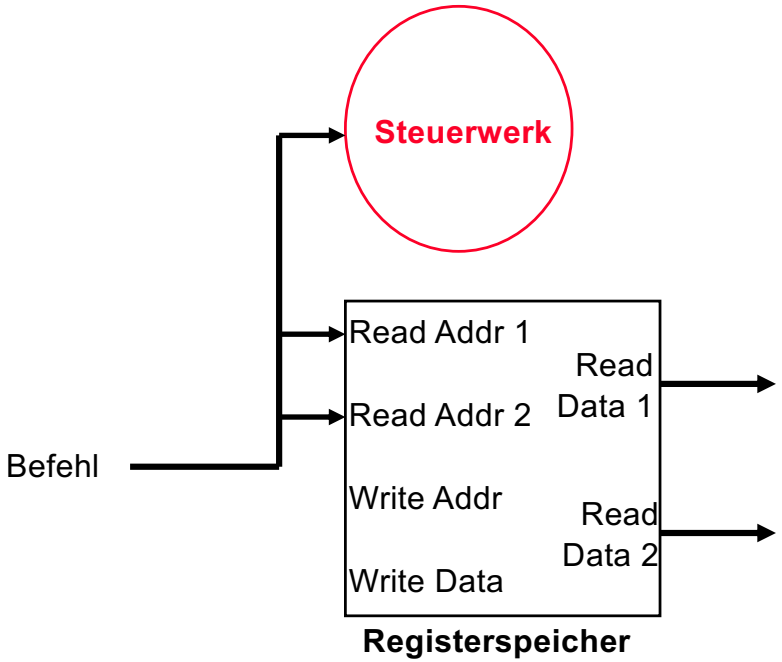
- Das «Fetching» von Befehlen schliesst folgendes ein:
  - Lese den Befehl vom Befehlsspeicher
  - Update den PC mit der Adresse des nächsten Befehls



  - PC wird jedem Taktzyklus aktualisiert. Daher wird kein explizites Steuersignal zum schreiben benötigt.
  - Der Befehlsspeicher wird bei jedem Taktzyklus gelesen. Daher wird kein explizites Steuersignal zum lesen benötigt.

Befehle decodieren

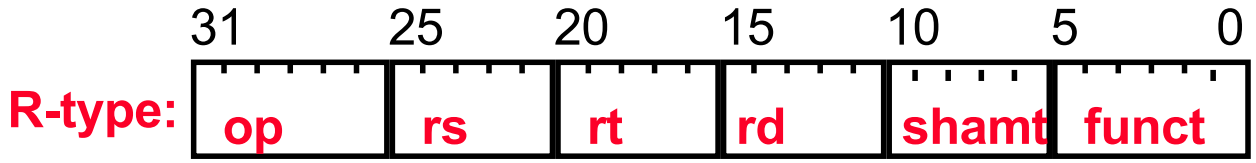
- Decoding von Befehlen beinhaltet:
  - Vom gelesen Befehl die Bits der Felder “**op**” und “**func**” zum **Steuerwerk** weiterleiten



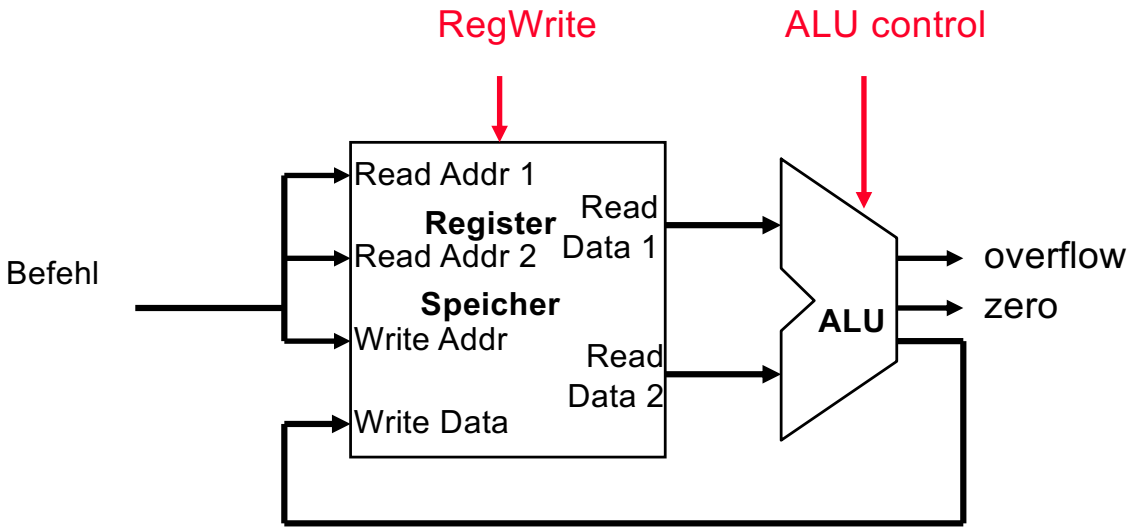
  - Lesen zwei Werte vom Registerspeicher (aka Registersatz)
    - Die Registerspeicheradressen befinden sich im Befehl

Ausführen von Operationen im R Befehlsformat

- R Format Operationen (`add, sub, slt, and, or`)



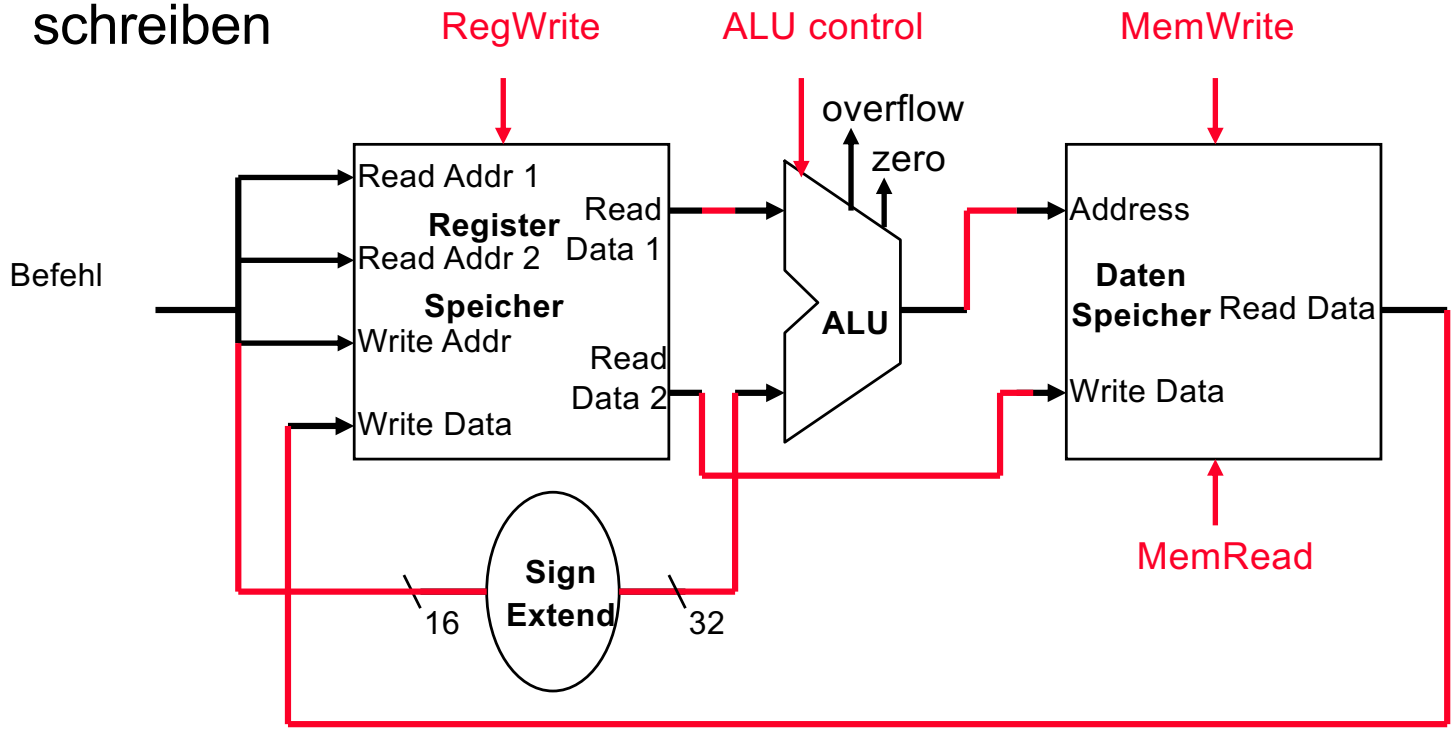
- Führe (op und funct) Operation mit den Werten in rs und rt aus
- Schreibe Resultat zurück in den Registerspeicher (in Adresse rd)



- Da nicht bei jedem Taktzyklus in den Registerspeicher geschrieben wird (e.g. `sw`), benötigen wir ein explizites Steuersignal für den Registerspeicher

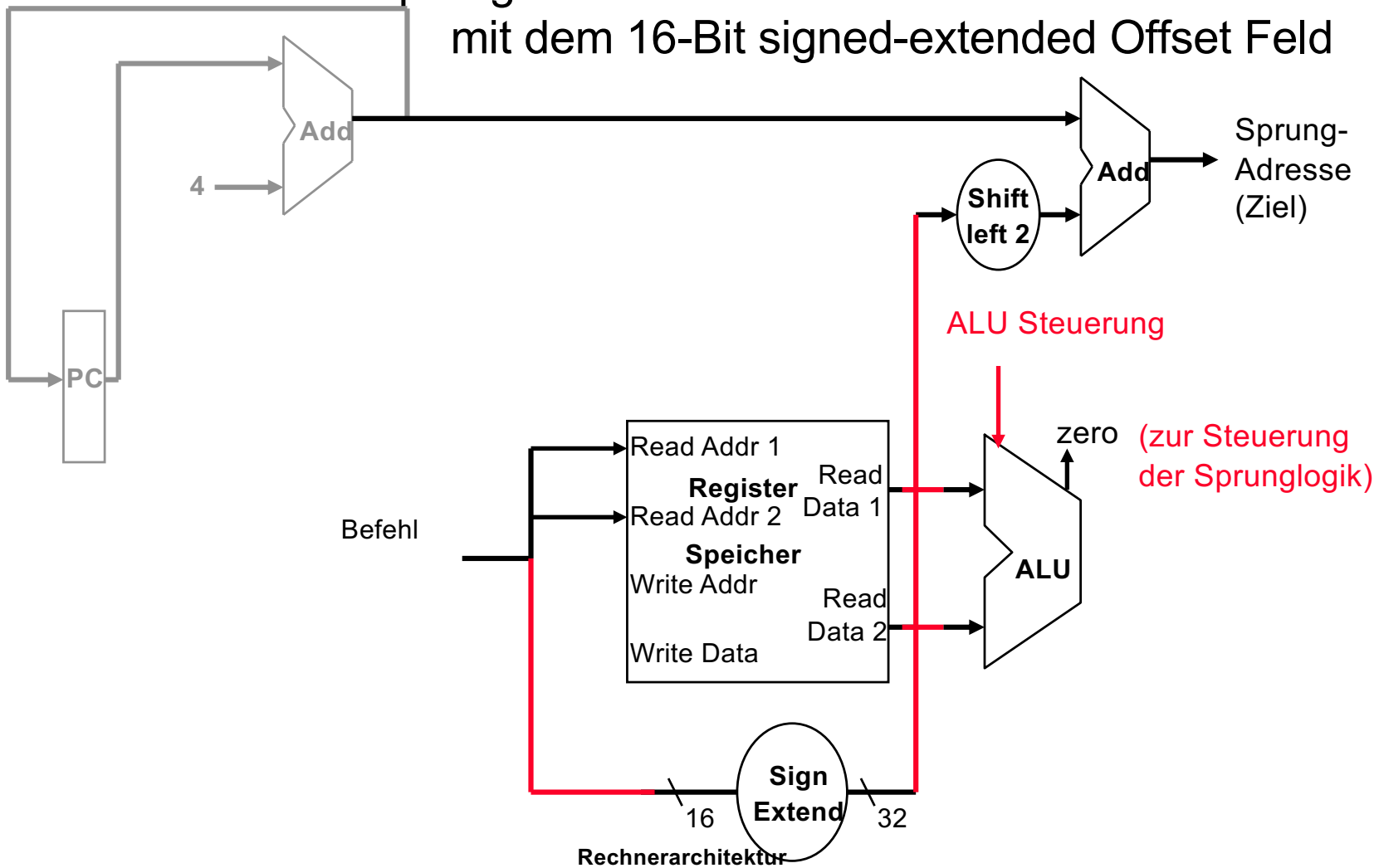
Ausführen von Lese und Schreib Befehlen

- Lese- und Schreibbefehle beinhalten
  - Berechne die Speicheradresse durch addieren des Basisregister (gelesen von Registerspeicher beim decoding) zum16-Bit signed-extended Offset Feld im Befehl
  - Store Wert (gelesen von Registerspeicher beim decoding) in den Datenspeicher schreiben
  - Load Wert, gelesen vom Datenspeicher, in Registerspeicher schreiben



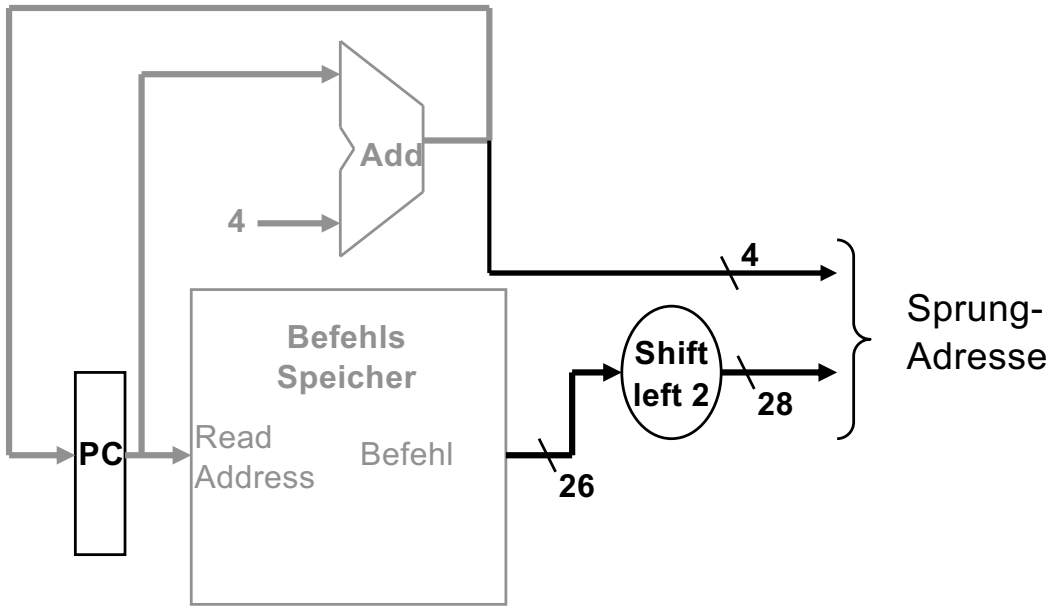
Ausführen von bedingten Sprungbefehlen

- Bedingte Sprungbefehle beinhalten
  - Vergleiche die aus dem Registerspeicher gelesenen Operanden (**`zero`** **`Ausgang der`** ALU)
  - Berechne die Sprungadresse durch addieren des aktualisierten PC mit dem 16-Bit signed-extended Offset Feld



Ausführen von des jump Befehls

- Jump (unbedingter Sprung) Befehl beinhaltet
  - Ersetzte die 28 unteren Bits des PC mit den 26 Bits des “fetched” Befehl welcher 2 Bits nach links geschiftet wurde



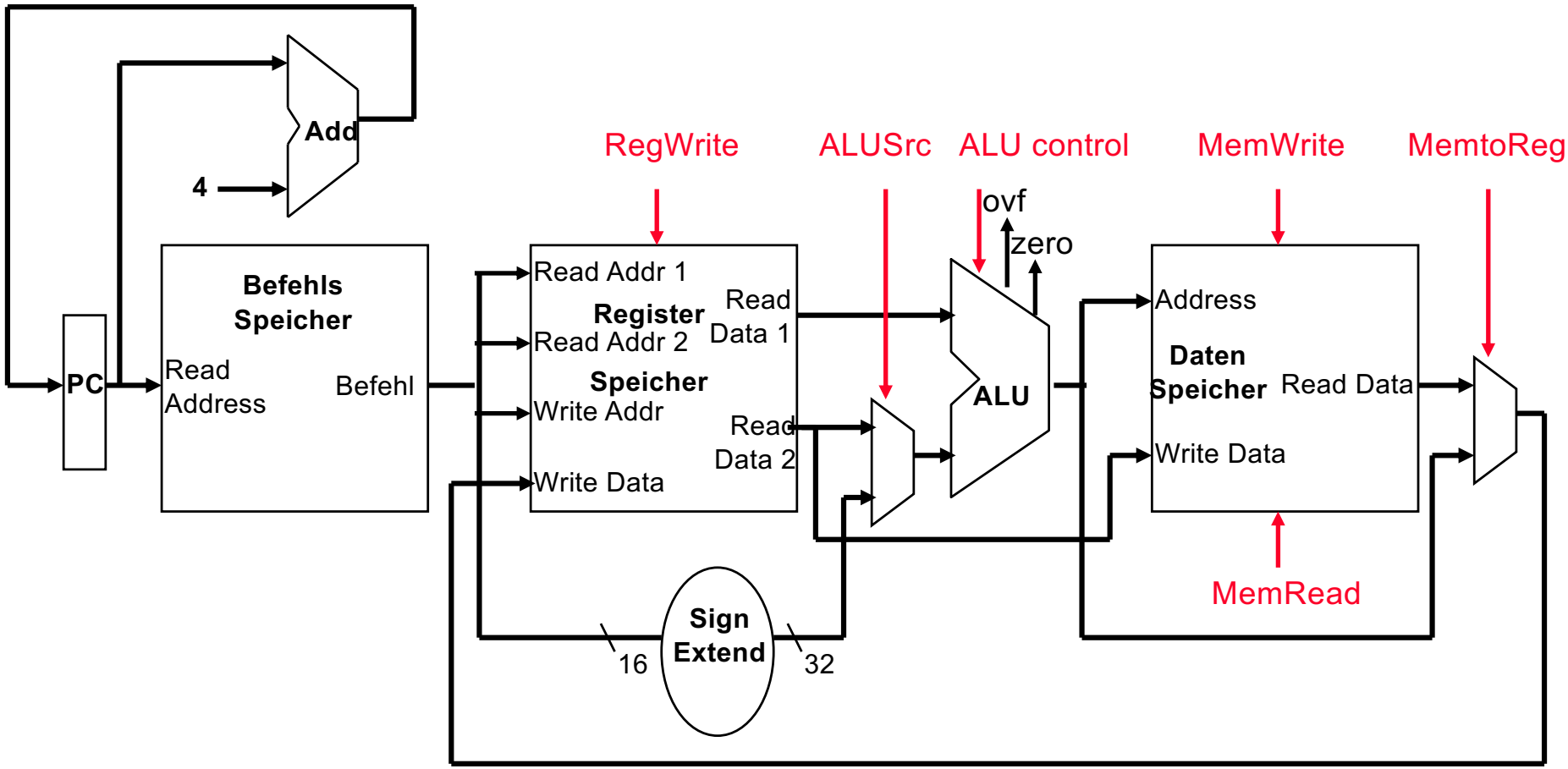
Aufbau eines einfachen Datenpfads

- Datenpfad Segmente, Steuerleitungen und Multiplexer nach Bedarf zusammenfügen
- Ein-Takt (single cycle) Design – fetch, decode und execute in **einem** Taktzyklus
  - Keine Datenpfad Ressource kann pro Befehl mehr als einmal benutzt werden. Deshalb müssen einige dupliziert werden (e.g., separate Befehls- und Datenspeicher, verschiedene Addierer)
  - Benötigen Multiplexer mit einer Steuerleitung um bei mehreren möglichen Eingängen eine Auswahl zu treffen
  - Steuersignale zum steuern des schreiben in den Register- und Datenspeicher
- Die Taktzeit wird durch die Länge des längsten Pfads determiniert

Beispiel: Datenpfad bauen

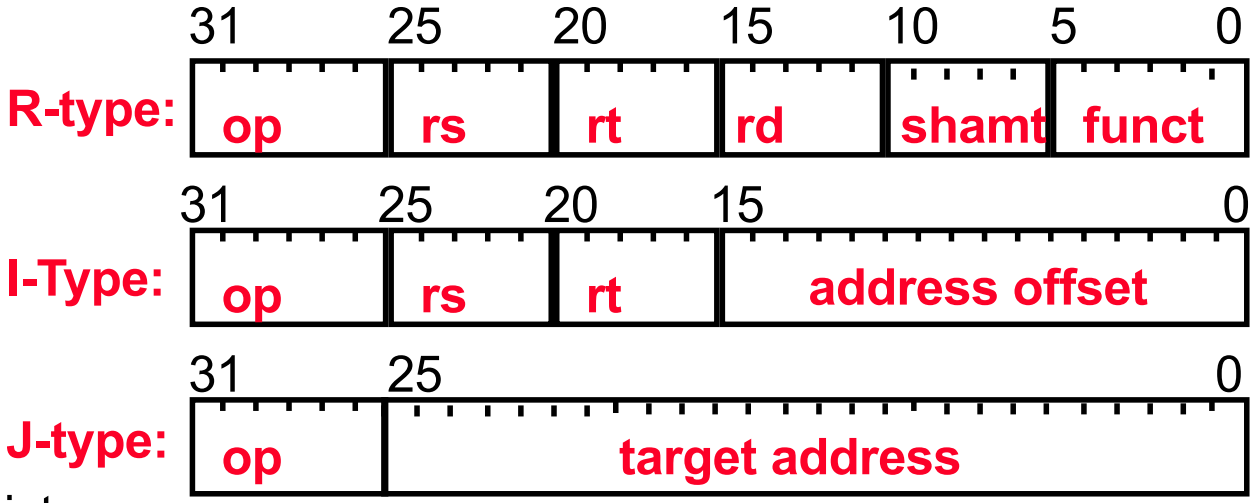
- Arithmetisch-logische Befehle im R-Format und Speicherbefehle sind ähnlich
- Unterschied:
  - R-Format nutzt ALU mit Inputs von Register; Speicherbefehle nutzen die ALU ebenfalls, aber der zweite Input ist der Offset
  - Der Wert (Ziel) welcher gespeichert wird, kommt aus der ALU (R-Format) oder Speicher (Für ein Load)
- Bauen eines Datenpfads für diese Befehle mit einem einzigen Registerspeicher und einer einzigen ALU (multiplexer können benutzt werden)

Fetch, R-Befehle, und (einige) Speicherzugriffe



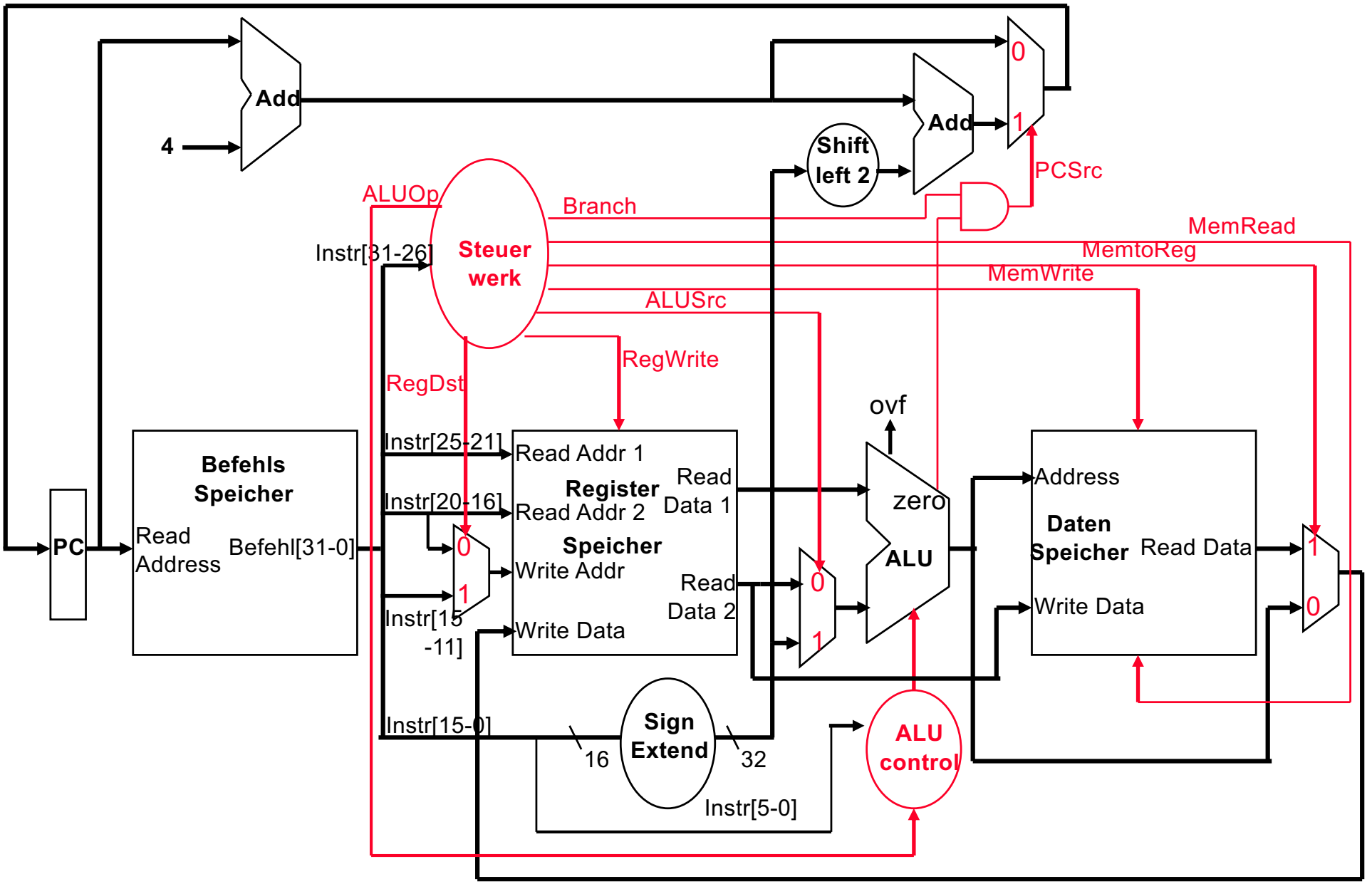
Steuerwerk hinzufügen

- Bestimmt die auszuführende Operation (ALU, Register-speicher und Hauptspeicher read/write)
- Steuerung des Datenfluss (Multiplexer Inputs)

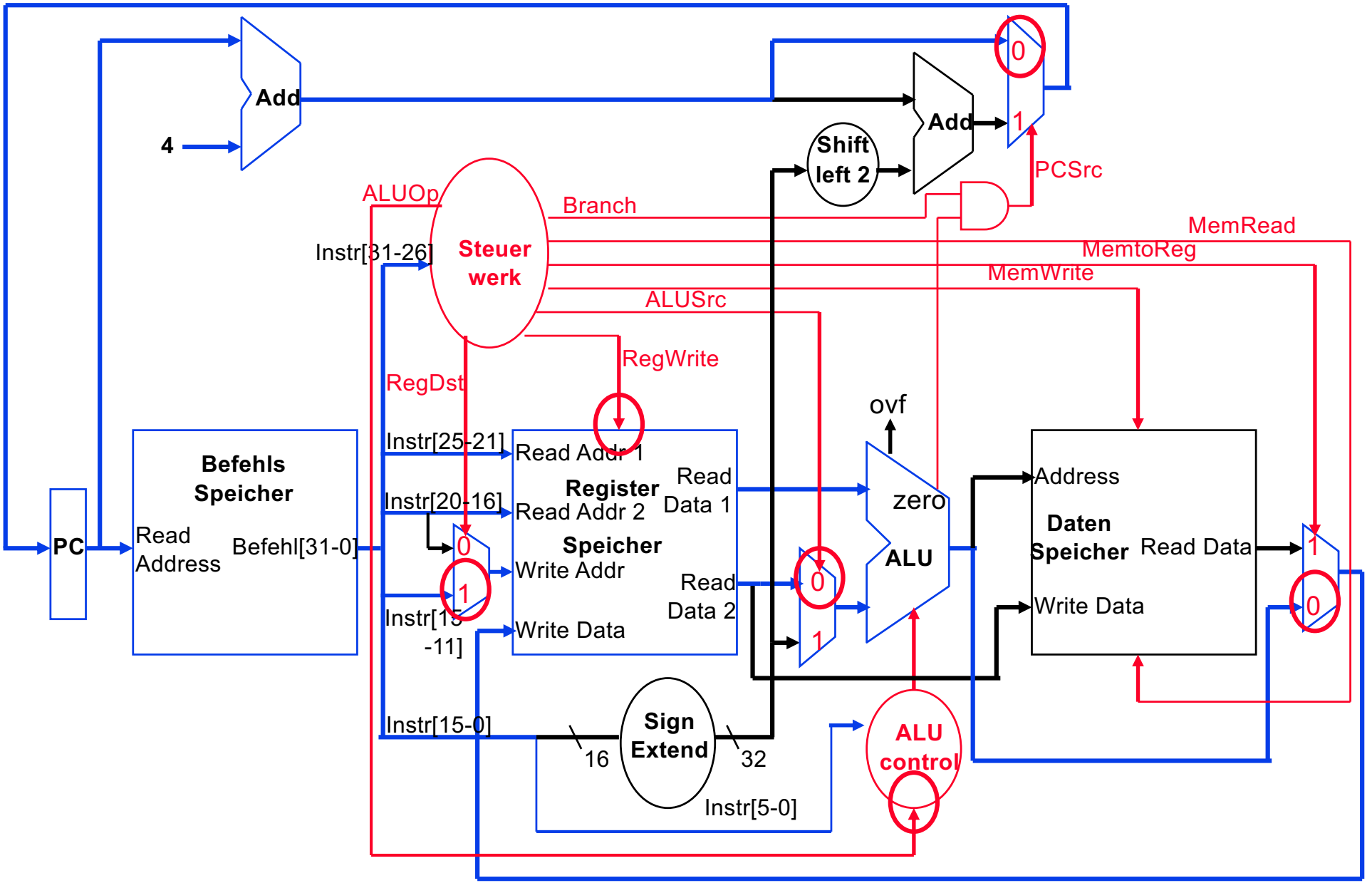


- Beobachtungen
  - **op** Feld immer Bits 31-26
  - Adressen der zu zu lesenden Register sind immer spezifiziert durch die Felder **rs** (Bits 25-21) und **rt** (Bits 20-16); für **lw** und **sw** ist **rs** das Basisregister
  - Zu beschreibende Registeradresse ist entweder – in rt (Bits 20-16) für **lw** oder in **rd** (Bits 15-11) für Befehle im R-Format
  - Offset für **beq**, **lw**, und **sw** immer in Bits 15-0

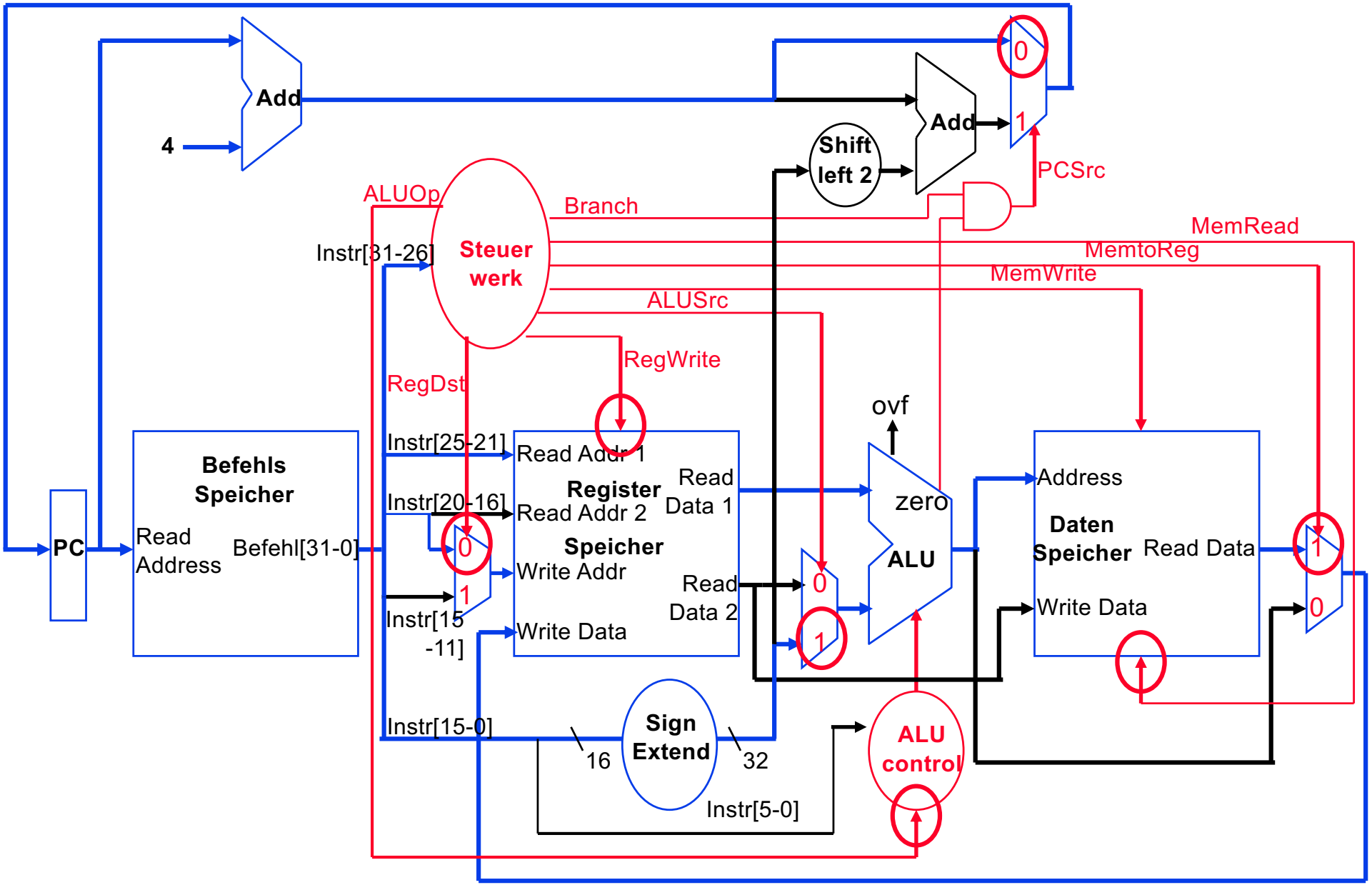
Single Cycle Datenpfad mit Steuerwerk



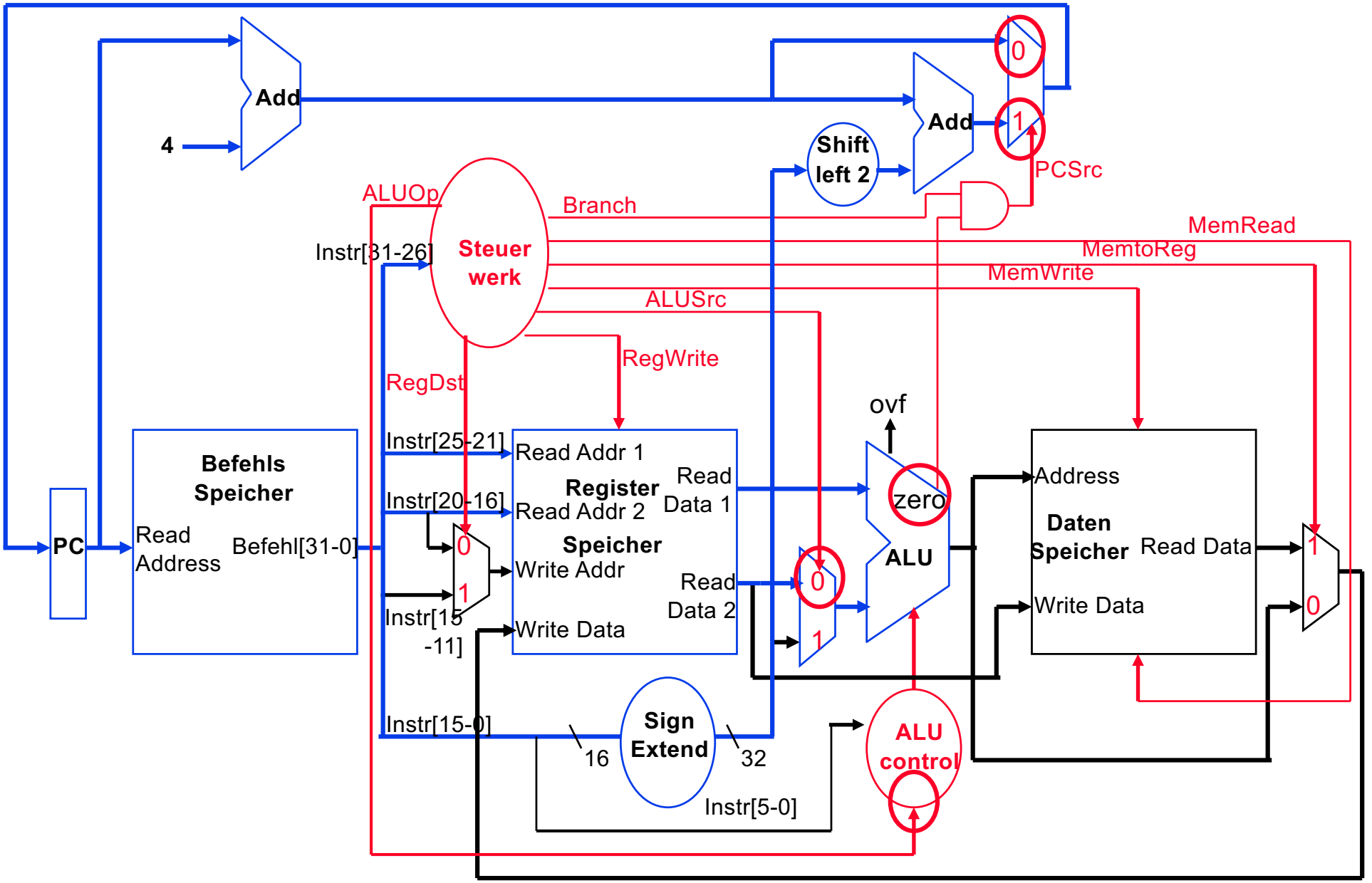
R-Format Befehle Datenfluss und Steuerung



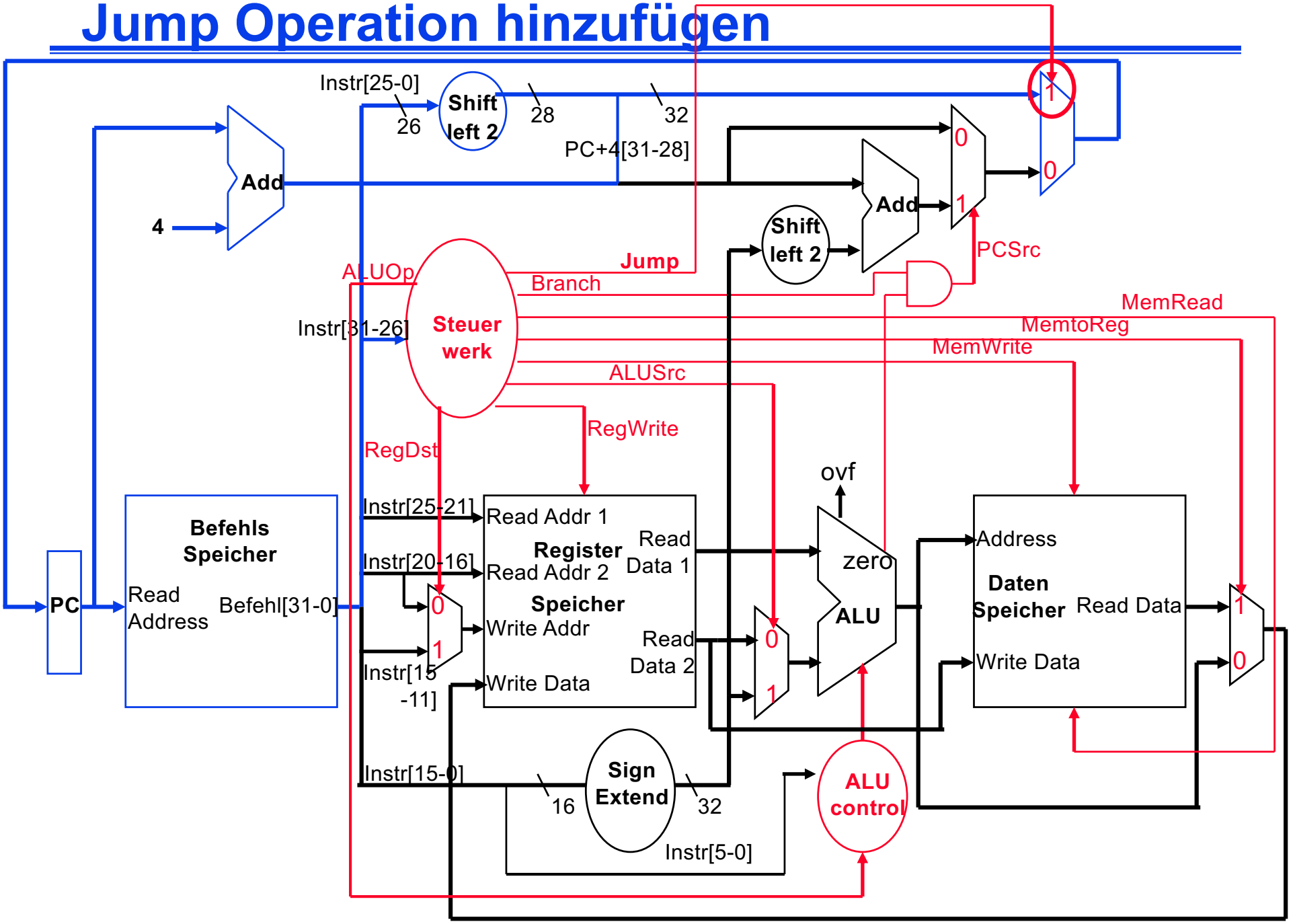
Load Word Befehl Datenfluss und Steuerung



Sprungbefehle Datenfluss und Steuerung

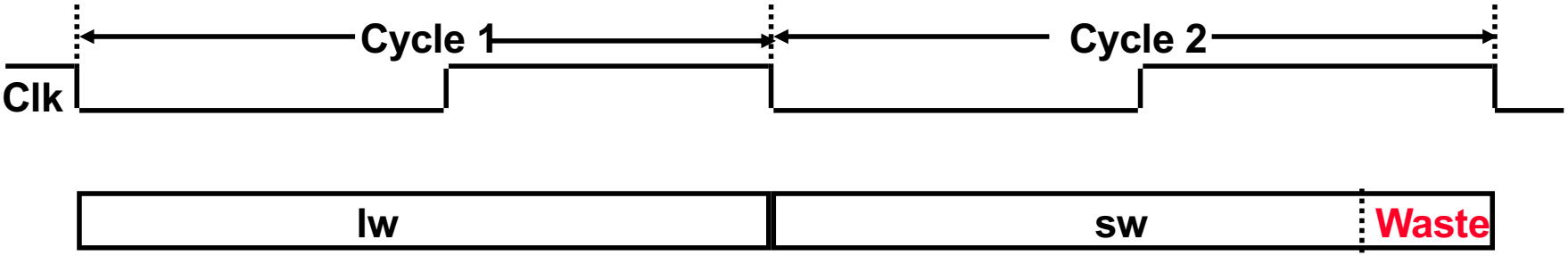


Jump Operation hinzufügen



Vor- & Nachteile vom Single Cycle Datenpfad

- Nutzt den Takt ineffizient – der Taktzyklus muss dem langsamsten Befehl unterstützen
  - Besonders problematisch für komplexere Befehle wie die für die Multiplikation von Gleitkommazahlen



- Verschwendung von Platz für Komponenten (e.g., Addierer) welche mehrfach vorkommen müssen da diese diese innerhalb eines Taktes nicht geteilt werden können

Aber

- Simpel und einfach zu verstehen

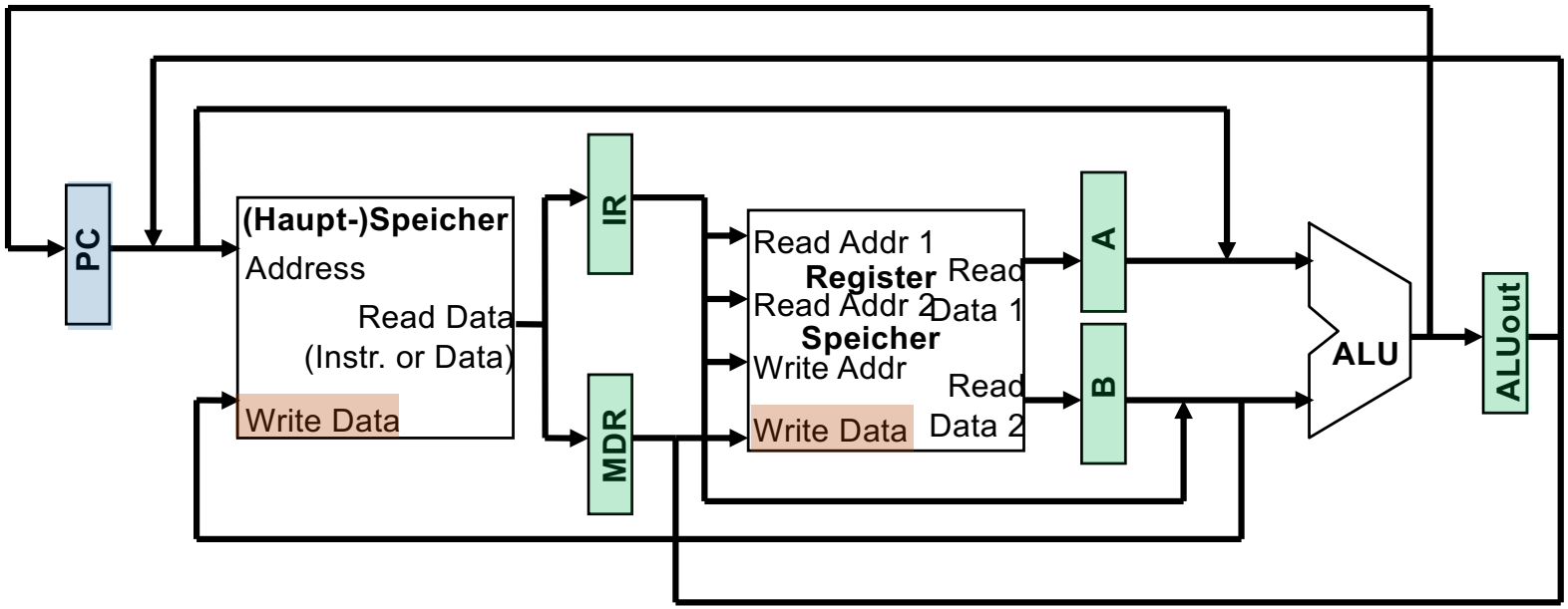
Multicycle Datenpfad Ansatz

- Ein Befehl darf mehr als ein Taktzyklus benötigen
  - Teilen von Befehlen in einzelne **Schritte** welche jeder ein Taktzyklus benötigt.
    - Jeder Schritt soll eine ähnliche Menge Arbeit verrichten
    - Jeder Schritt soll nur eine wesentliche Funktionseinheit nutzen
  - Befehl benötigten eine unterschiedliche Anzahl Taktzyklen

- Zusätzlich zu schnelleren Taktzyklen, erlaubt der Multicycle Ansatz ein wiederverwenden von Komponenten solange diese in verschiedenen Taktzyklen genutzt werden. Resultat:
  - Benötigen nur einen Speicher – aber nur einen Speicherzugriff pro Taktzyklus
  - Benötigen nur eine ALU/Adder – aber nur ein ALU Operation pro Taktzyklus

Multicycle Datenpfad Ansatz

- Am Ende eines Taktzyklus
  - Gespeicherte Werte die von aktuellen Befehl in späteren Taktzyklen noch benötigt werden, werden in temporären Register gespeichert (nicht sichtbar für uns). Alle (ausser IR) halten Daten nur in angrenzen Taktzyklen.
    (Es wird kein Steuersignal zu schreiben benötigt)



IR – Befehls Register **MDR** – Speicherdatenregister

A, B – Zusätzliche Register **ALUout** – ALU Output Register

- Daten für nachfolgende Befehle werden in für uns (Programmierer) sichtbaren Registern gespeichert (i.e., Registerspeicher, PC, Datenspeicher)

Befehle aus Sicht Befehlssatzarchitektur (ISA)

- Beachte jeden Befehl aus Sicht der Befehlssatzarchitektur.
- **Add** als Beispiel:
  - Der **add** Befehl ändert ein Register.
  - Dieses Register ist definiert durch die Bits 15:11 des Befehls.
  - Befehl definiert durch den Befehlszähler (PC).
  - Neuer Wert ist die Summe ("op") zweier Register.
  - Register definiert durch Bits 25:21 und 20:16 des Befehls.

```
Reg[Memory[PC][15:11]] <-
    Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]
```

  - Um das umzusetzen teilen wir den Befehl die **einzelne Schritte** auf. (Wie beim Einführen von Variablen beim Programmieren)

Aufteilen eines Befehls

- Befehlssatzarchitektur Arithmetik Definition:

```
Reg[Memory[PC][15:11]] <-
    Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]
```

- Kann aufgeteilt werden:
  - `IR <- Memory[PC]`
  - `A <- Reg[IR[25:21]]`
  - `B <- Reg[IR[20:16]]`
  - `ALUOut <- A op B`
  - `Reg[IR[15:11]] <- ALUOut`

- Einen wichtiger Teil der Arithmetik Definition fehlt noch!
  - `PC <- PC + 4`

Idee hinter dem Multicycle Ansatz

- Wir definieren jeden Befehl aus Sicht der Befehlssatz-architektur. Macht das mal selber als (freiwillige) Übung.
- Aufteilen in Schritte welche unserer Regel folgen:
  - Daten fliessen durch höchstens eine Haupt-Funktionseinheit
  - Verteile die Arbeit dabei gleichmässig auf die Schritte)
- Führe, wo nötig, neue Register ein (A, B, ALUOut, MDR, ...)
- Versuche so viel wie möglich in einem Schritt zu erledigen
  - Vermeidet unnötige Taktzyklen
- Versuche ebenfalls Schritte gemeinsam zu nutzen
  - Minimiert Steuerung. Hilft Lösung simpel zu halten.
- Resultat: Multicycle Implementation unseres Buches!

Fünf Ausführungsschritte

- Befehlsholschritt (Fetch)
  - *IF*: instruction fetch
- Befehlscodierung und Registerholschritt
  - *ID*: instruction decode / register fetch
- Ausführung / Effektivadress-Schritt (Execution)
  - *EX*: execution
- Speicherzugriff oder R-Typ Befehl Komplettierungsschritt
  - *MEM*: Memory Access
- Speicherleseabschluss
  - *WB*: write-back

Befehle benötigen 3 - 5 Taktzyklen

Schritt 1: Befehlsholschritt

- Nutzen PC um den Befehl in den Befehlsregister (IR) zu laden.
- Inkrement des PC um 4. Resultat zurück in PC schreiben.
- Nutzen der RTL "Register-Transfer Language"

```
IR <= Memory[PC];
PC <= PC + 4;
```

Können wir die Steuersignale bestimmen?

Was ist der Vorteil den PC jetzt zu aktualisieren?

Schritt 2: Befehlscodierung und Registerholschritt

- Lese Register rs und rt falls diese benötigt werden
- Berechne Sprungadresse für den Fall eines Sprungbefehls
- RTL:

```
A <= Reg[IR[25:21]];

B <= Reg[IR[20:16]];

ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- Wir setzten noch keine Steuerleitungen basierend auf dem Befehlstyp.

Schritt 3: (Befehls abhängig)

- ALU führt eine von drei Befehlen aus, basierend auf dem Befehlstyp

- Speicher Referenz:

```
ALUOut <= A + sign-extend(IR[15:0]);
```

- R-type:

```
ALUOut <= A op B;
```

- Sprung:

```
if (A==B) PC <= ALUOut;
```

Schritt 4: (R-type oder Speicherzugriff)

- Speicherzugriffe, laden und speichern

```
MDR <= Memory[ALUOut];
      or
Memory[ALUOut] <= B;
```

- R-type Befehl beenden

```
Reg[IR[15:11]] <= ALUOut;
```

Schritt 5: Speicherleseabschluss

- `Reg[IR[20:16]] <= MDR;`

Nur load Befehle benötigen diesen Schritt

Zusammenfassung

Tab. 5.7 Übersicht über die für die Ausführung aller Befehlsklassen erforderlichen Schritte bzw. Befehlsausführungsphasen. Die Befehle beanspruchen zwischen drei und fünf Ausführungsphasen. Die ersten beiden Phasen sind von der Befehlsklasse unabhängig. Nach diesen Phasen muss der Befehl je nach Befehlsklasse zwischen einem und drei weiteren Zyklen ausführen. Die leeren Einträge für die Speicherzugriffsphase oder die Speicherleseabschlussphase weisen darauf hin, dass die jeweilige Befehlsklasse weniger Zyklen beansprucht. In einer Mehrzyklenimplementierung wird ein neuer Befehl gestartet, sobald der aktuelle Befehl abgeschlossen ist, so dass sich keine Zyklen im Leerlauf befinden oder vergeudet werden. Wie bereits erwähnt, wird der Registersatz bei jedem Zyklus gelesen. Solange das IR unverändert bleibt, sind die aus dem Registersatz gelesenen Werte jedoch identisch. Insbesondere der während der Befehlsentschlüsselungsphase aus Register B gelesene Wert für eine Verzweigung oder einen R-Befehl ist mit dem Wert identisch, der während der Ausführungsphase in Register B gespeichert und anschließend in der Speicherzugriffsphase für eine `store-word`-Befehl verwendet wurde.

Befehlsausführungs-phase	Aktion bei R-Befehlen	Aktion bei Speicherreferenzbefehlen	Aktion bei Verzweigungen	Aktion bei Sprüngen
Befehlsholphase	IR <= Memory[PC] PC <= PC + 4			
Befehlsentschlüsselungs-/ Registerholphase	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC +(sign-extend (IR[15:0]) << 2)			
Ausführung, Adressberechnung, Verzweigungs-/ Sprungausführung	ALUOut <= A op B	ALUOut <=A + sign-extend (IR[15:0])	if (A == B) PC <=ALUOut	PC <= PC [31:28], (IR[25:0]],2'b00)
Speicherzugriff oder R-Befehlsausführung	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Speicherleseabschluss		Load: Reg[IR[20:16]] <= MDR		

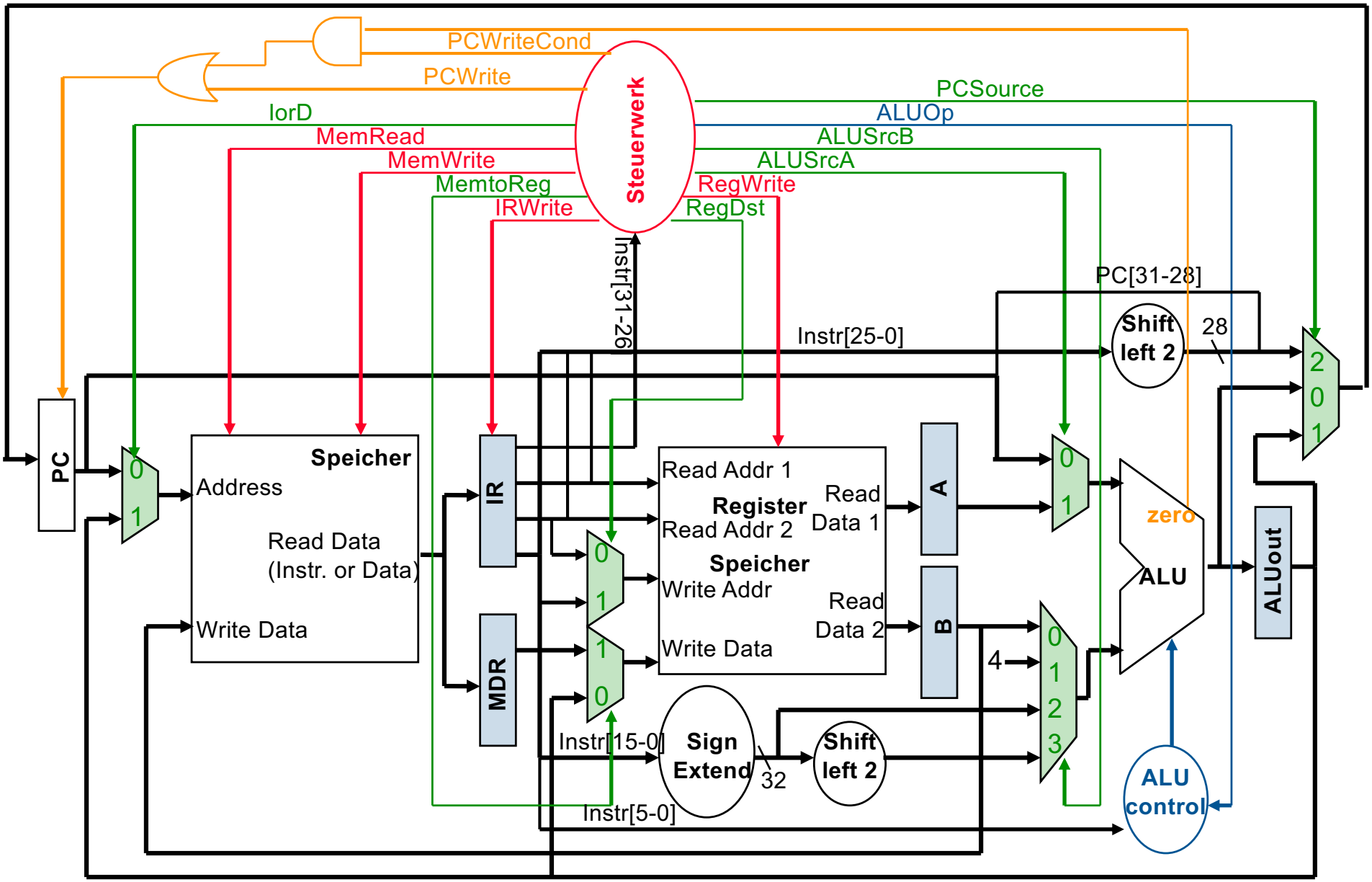
Einfache Fragen

- Wieviele Taktzyklen benötigt dieser Code?

```
          lw $t2, 0($t3)
          lw $t3, 4($t3)
          beq $t2, $t3, Label #assume not
          add $t5, $t2, $t3
          sw $t5, 8($t3)
Label:    ...
```

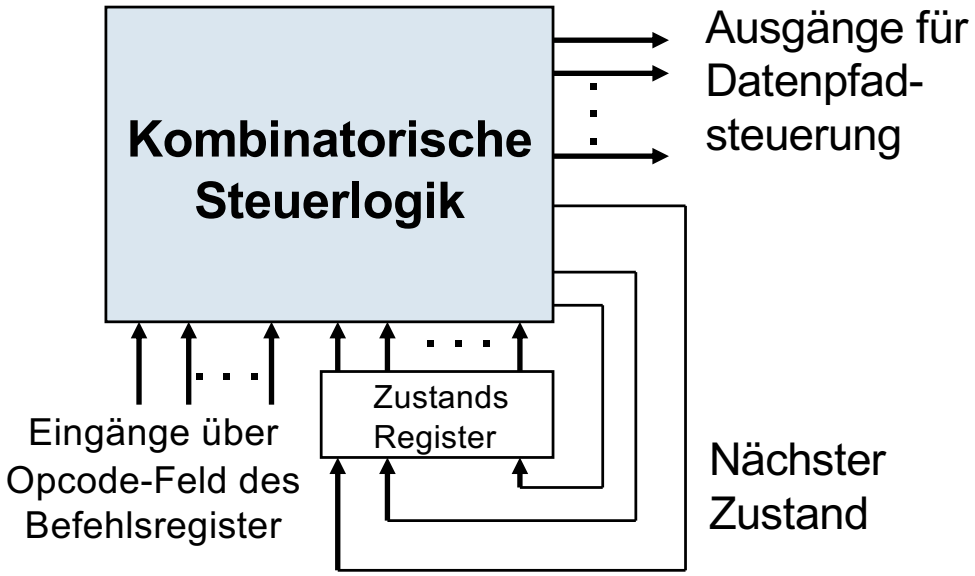
- Was passiert bei der Ausführen des Taktzyklus Nr. 8?
- In welchem Taktzyklus wird `$t2` und `$t3` addiert?

Der Multicycle Datenpfad mit Steuersignalen

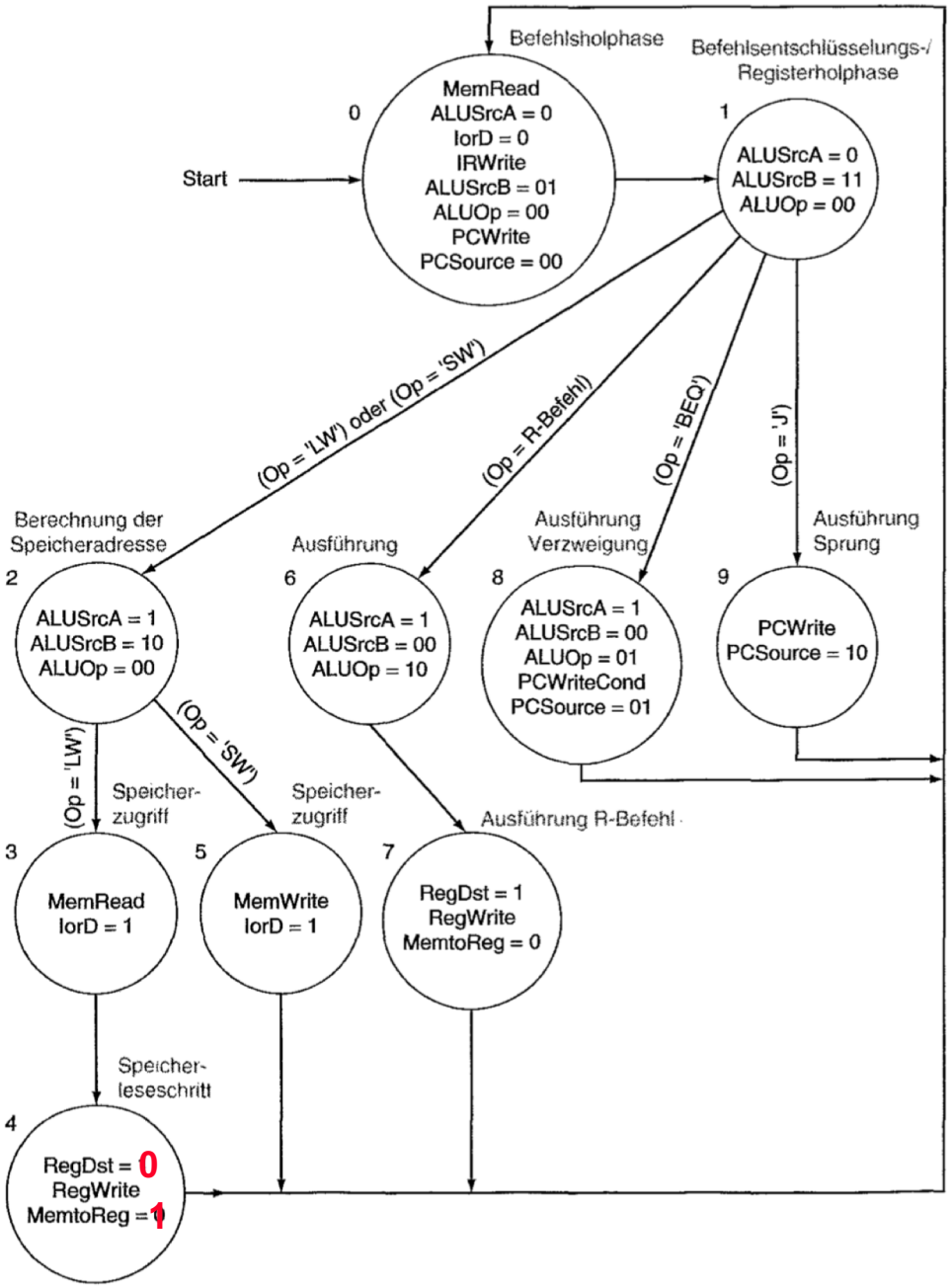


Multicycle Steuerwerk

- Multicycle Datenpfad Steuersignale sind nicht nur durch die Bits im Befehl determiniert
  - Z.B., Die op Code Bits zeigen der ALU was sie tun soll, aber *nicht* welcher Befehl als nächstes ausgeführt werden soll
- Benötigen einen endlichen Automat (FSM) zur Steuerung
  - Eine Menge an Zuständen (aktueller Zustand im State Register)
  - Nächste Zustandsfunktion (determiniert durch aktuellen Zustand und Input)
  - Output Funktion (aktuellen Zustand und Input)

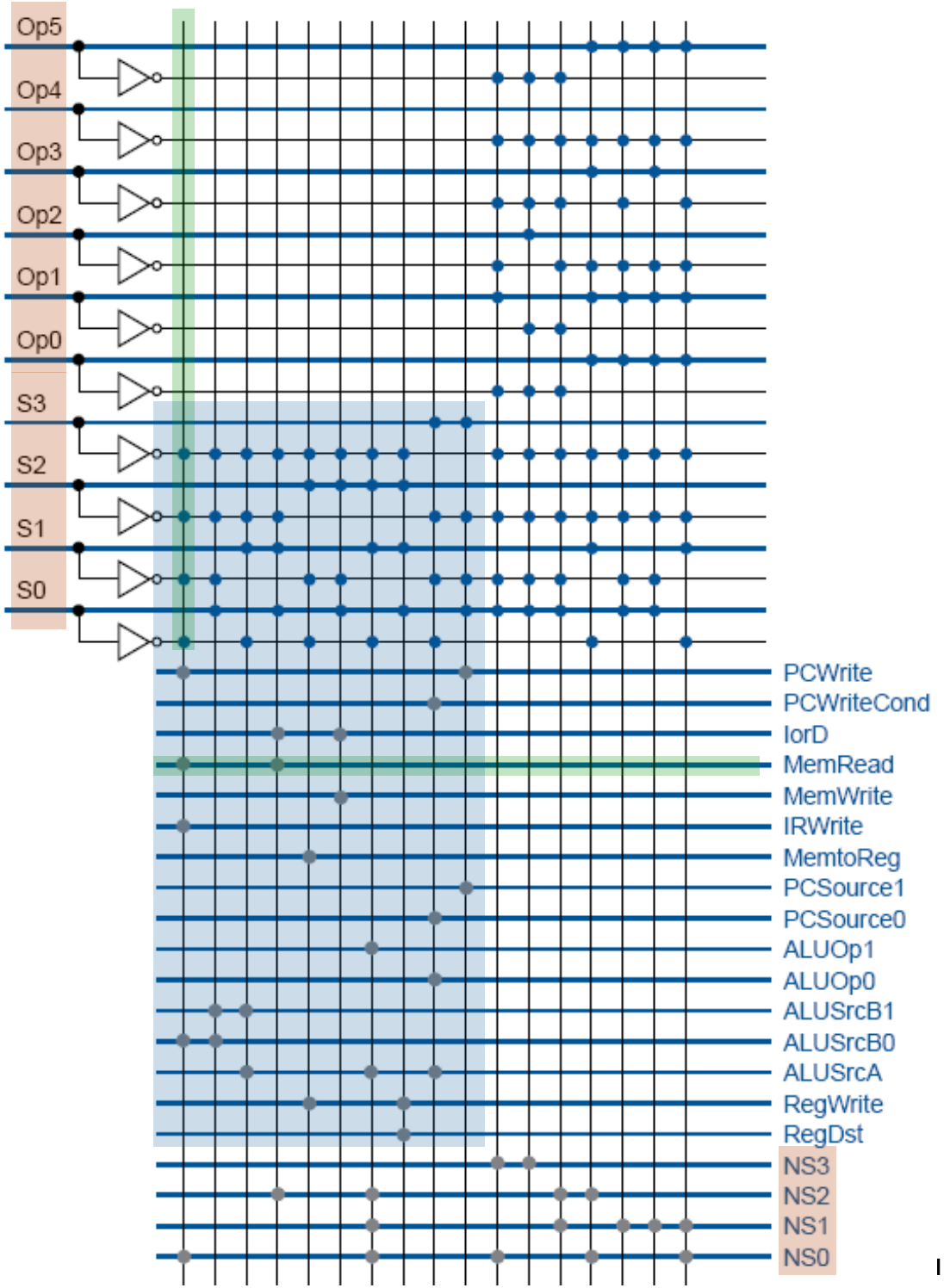


Multicycle Steuerwerk: Endlicher Automat (FSM)

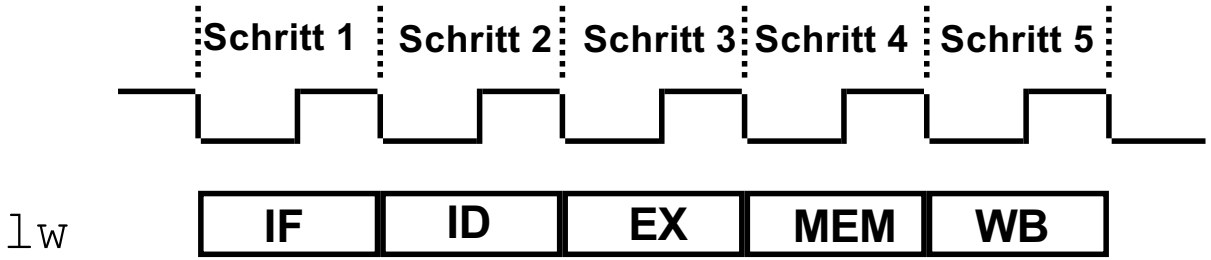


Steuerwerk: PLA

PLA Implementation des endlichen Automaten für das Steuerwerk



Die fünf Schritte des Load Befehls

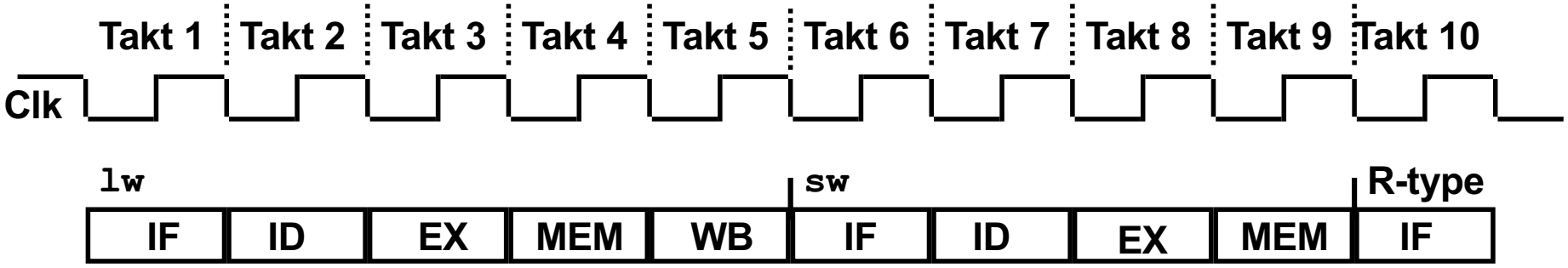


- IF: Befehl laden und PC aktualisieren
- ID: Befehl decodieren, Register lesen, Sign Extend Offset
- EX: Ausführen R-type; Berechne Speicheradresse; Werte vergleichen für Sprung; Sprung Fertigstellung
- MEM: Memory Read; Memory Write Fertigstellung; R-type Fertigstellung (Register Speicher schreiben)
- WB: Memory Read Fertigstellung (Register Speicher schreiben)

Befehle benötigen 3 - 5 Taktzyklen!

Multicycle Vor- & Nachteile

- Nutzt die Taktzyklen effizienter – die Länge des Taktzyklus richtet sich nach dem langsamsten Befehls-Schritt



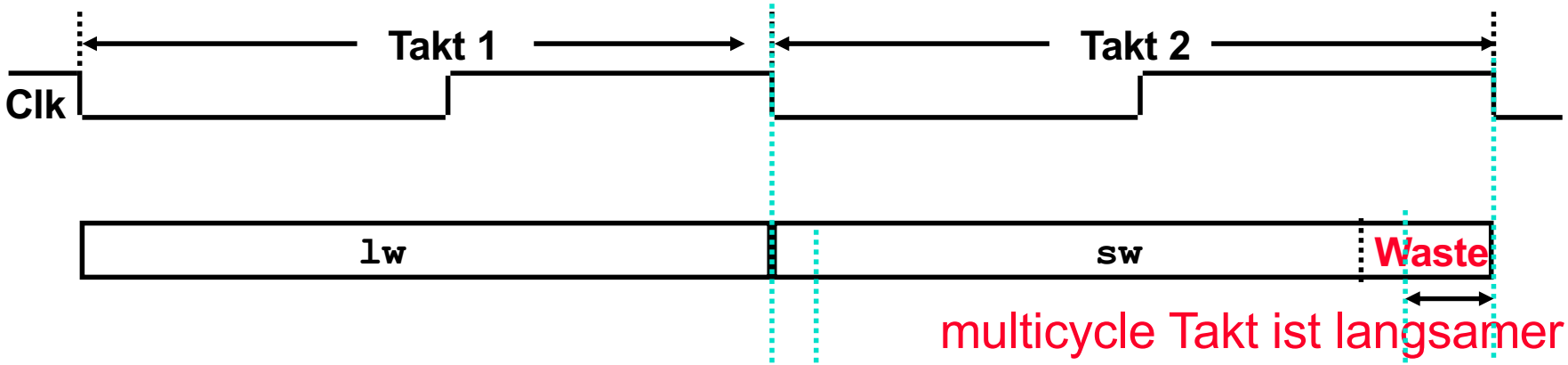
- Multicycle Implementationen erlauben es Funktionseinheiten mehrfach pro Befehl zu nutzen, solange diese in verschiedenen Taktzyklen benutzt werden

Aber, wir zahlen einen Preis dafür

- Benötigen zusätzliche interne temporäre Register, mehr Multiplexer, und komplexere (FSM) Steuerung

Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:



multicycle Takt ist langsamer als 1/5 des single cycle Takt aufgrund des Overhead durch die temporären Register

Multiple Cycle Implementation: