



Technische Grundlagen der Informatik 2

Rechnerorganisation

Kapitel 5, Teil 2:

Multicycle Processor (Mehrzyklenprozessor)

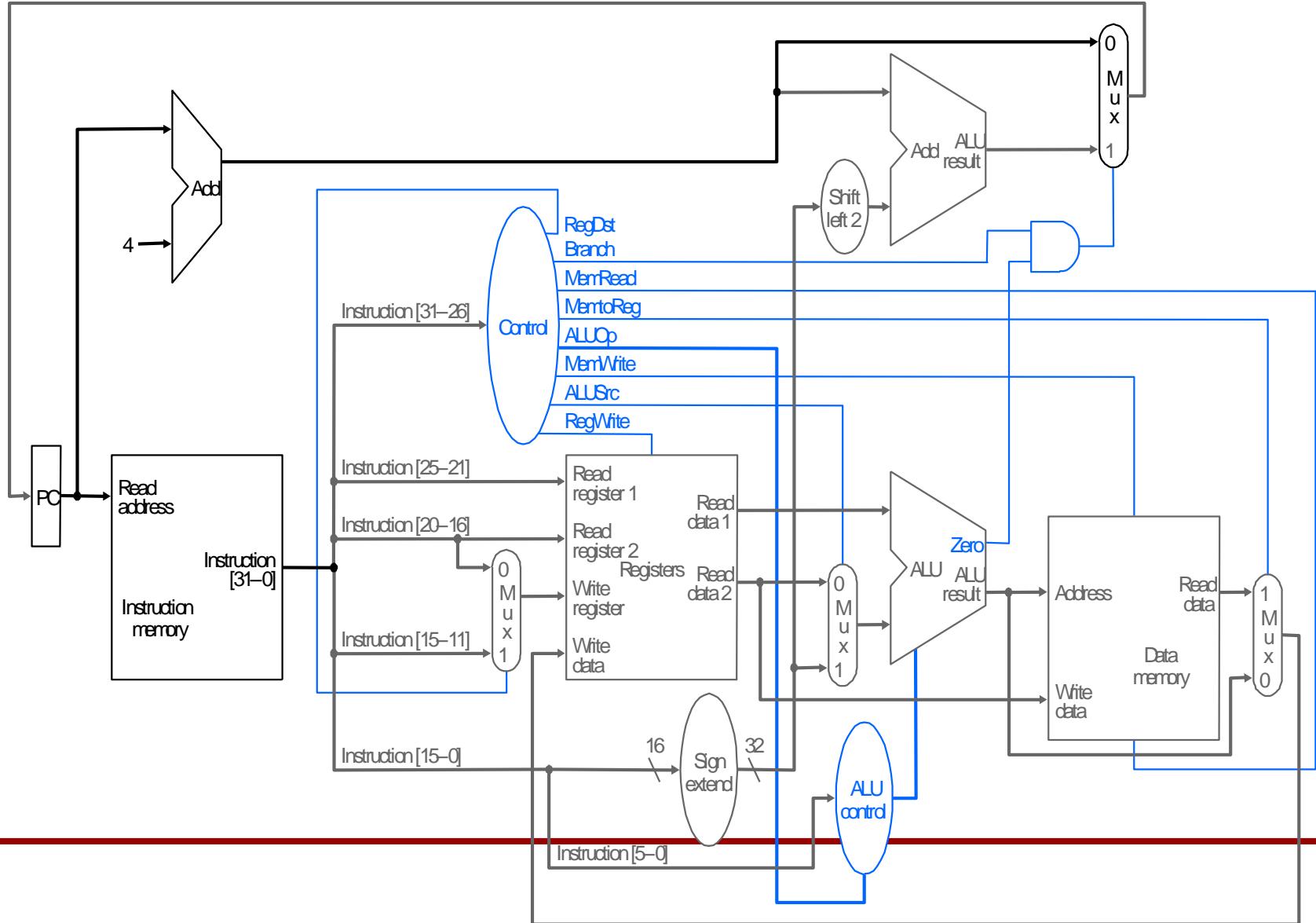
Prof. Dr. Ben Juurlink

Fachgebiet: Architektur eingebetteter System
Institut für Technische Informatik und Mikroelektronik
Fak. IV – Elektrotechnik und Informatik

SS 2014

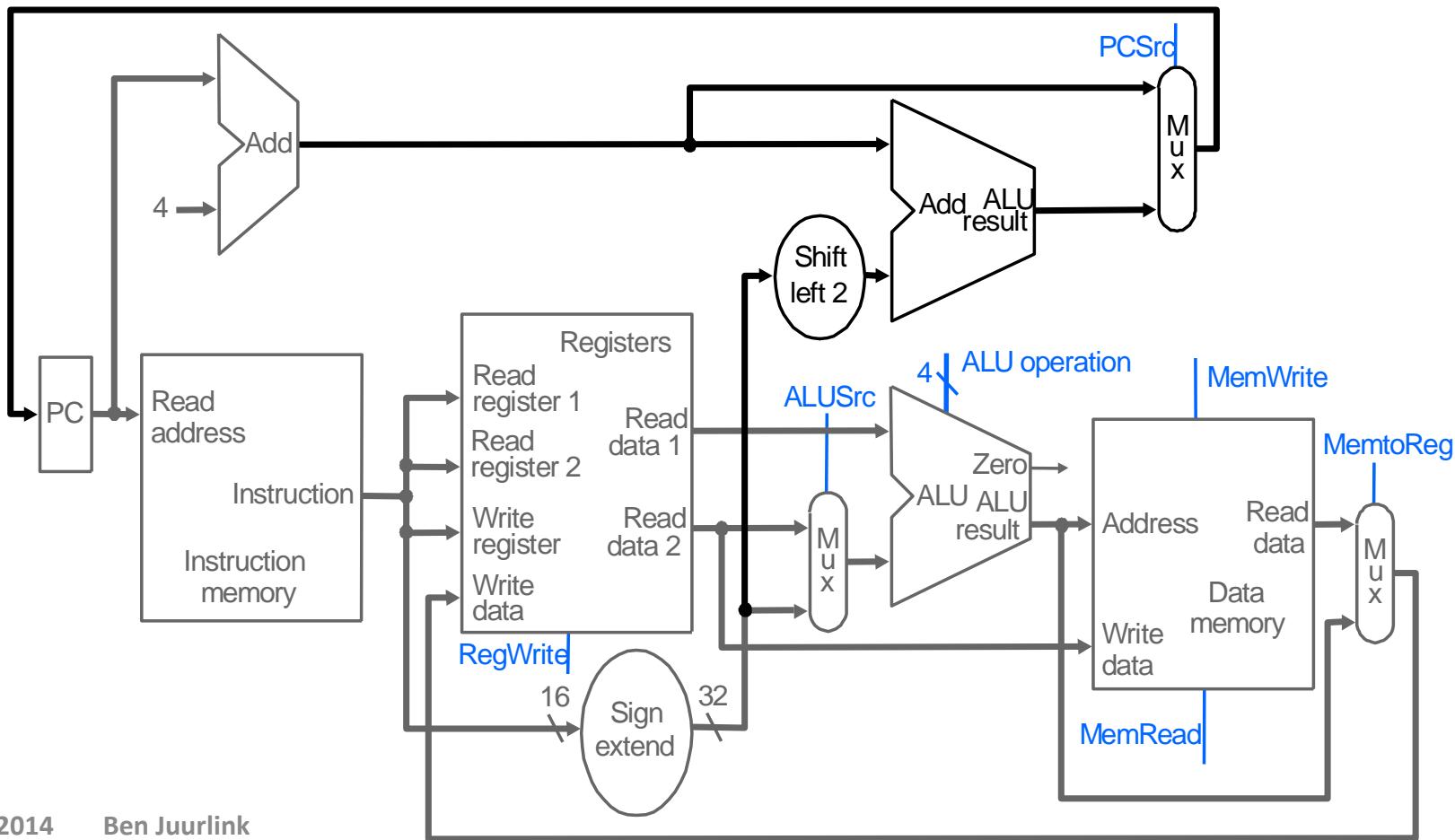


- Nach dieser Vorlesung sollten Sie in der Lage sein:
 - zu erklären, warum die Eintakt-Implementierung langsam und ineffizient ist
 - zu erklären, warum die Mehrzyklen-Implementierung besser ist
 - Mehrzyklen-Datenpfad zu erstellen und erweitern
 - Erstellen der Steuerung der Mehrzyklen-Implementierung mit einer FSM und/ oder Microcode



- Jeder Befehl braucht einen Takt
 - Taktzeit muss lang genug sein, um den langsamsten Befehl zu bearbeiten
 - Was ist der CPI-Wert des Eintakt-Prozessors?
- Verschwendung von Chipfläche: **kein Teilen von Hardware Ressourcen**
 - getrennte Addierer/ALU für
 - Inkrementieren des PC
 - Berechnung der Branch Zieladresse
 - Ausführung von Operationen/ Berechnung der effektiven Adresse
 - separater Befehls- und Datenspeicher

- Kalkulierte die Taktzeit (Verzögerungen können bis auf folgenden ignoriert werden):
 - Speicher(200ps), ALU und Addierer (100ps), Registerspeicherzugriff (50ps)



- Speicher (200ps), ALU & Addierer (100ps), Registerspeicherzugriff (50ps)

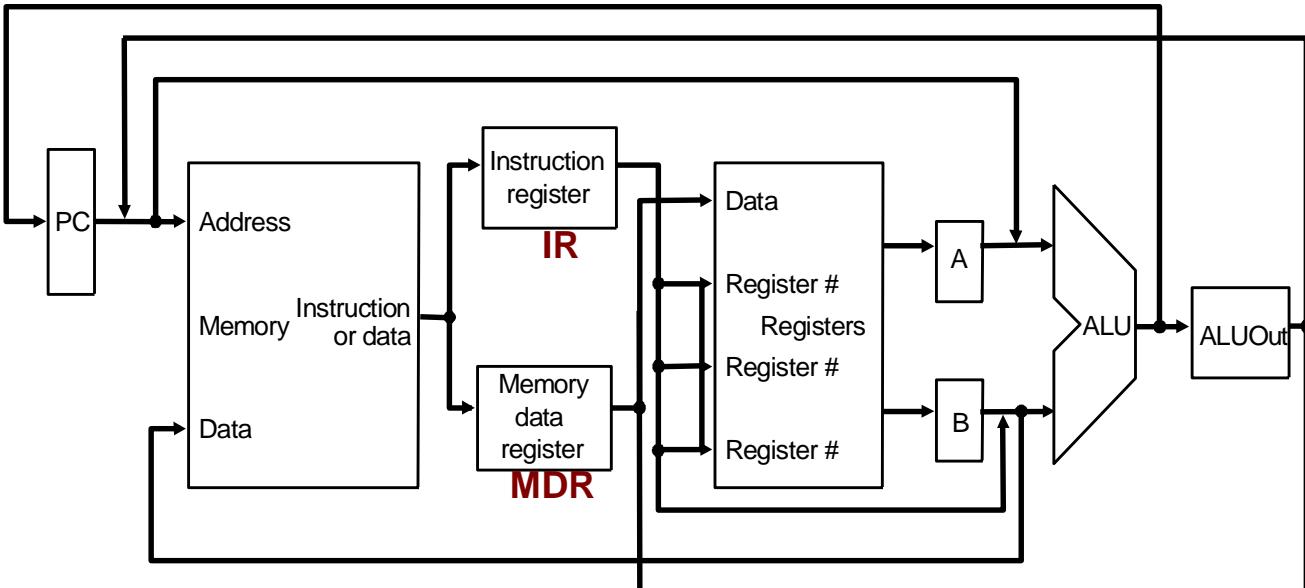


- langsamster Befehl ist **lw**, der 600 ps (1.67GHz) benötigt

- Angenommen wir haben ein **Clock mit variabler Taktzeit** (nicht realistisch, nur Übung), so dass Befehle folgende Zeiten brauchen:
 - R-type: 400 ps
 - Load: 600 ps
 - Store: 550 ps
 - Branch: 350 ps
 - Jump: 200 ps
- Kalkuliere durchschnittliche Befehlszeit (DB), unter folgender Prämisse:
 - 25% Lade-, 10% Speicher-, 45% R-type, 15% Verzweigung-, 5% Sprung-Befehle
- $DB = 0.25 \times 600 + 0.1 \times 550 + 0.45 \times 400 + 0.15 \times 350 + 0.05 \times 200 = 447.5 \text{ ps}$
- Variable-Taktzeit-Implementierung ist $600/447.5 = 1.34x$ schneller
- Was wäre wenn wir kompliziertere Befehle, wie Gleitkomma Operationen hätten?

- Kürzere Taktzeit nutzen
- Verschiedene Befehle benötigen unterschiedliche Anzahl von Schritten.
 - Jeder Schritt einen Takt.
- Eine Einheit kann mehrmals pro Befehl benutzt werden, **zu verschiedenen Takten**.

Abstrakte Darstellung der Mehrzyklus-Implementierung:

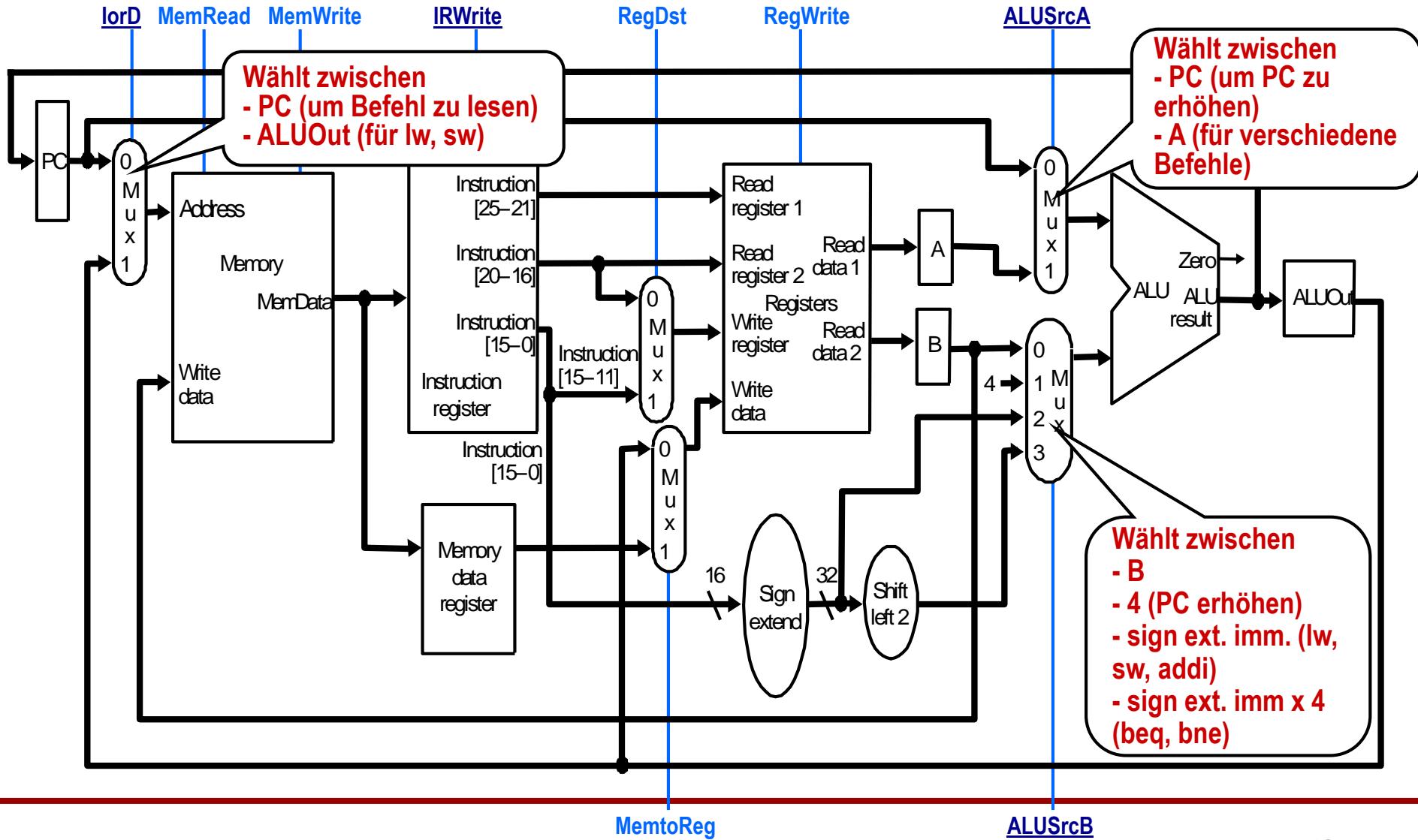


- Was benötigen wir, um Befehlsausführung in mehrere Schritte aufzuteilen?

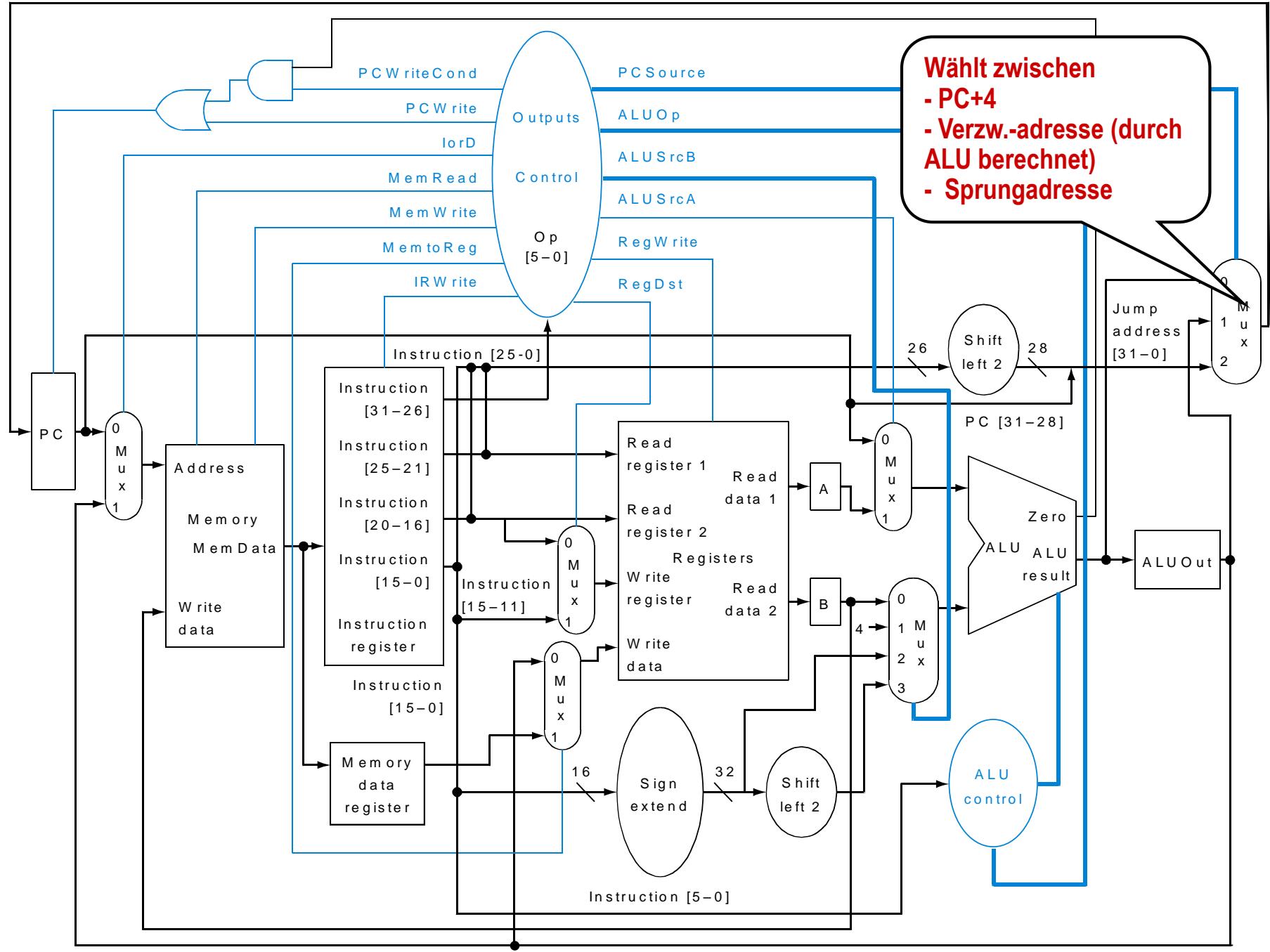
- Hauptunterschiede zur Eintakt-Implementierung
 - Ein Speicher für Befehle sowie für Daten
 - Eine ALU, anstelle von einer ALU und zwei Addierer
 - führt Operationen aus
 - berechnet die effektive Adresse
 - inkrementiert den Programmzähler
 - berechnet Sprungziel-Adresse
 - jedoch in verschiedenen Schritten/Takten
 - Ein oder mehrere Register nach jeder Funktionseinheit (FU) um den Output der FU zu halten, bis er in einem darauffolgenden Takt benutzt wird
- Steuersignale werden nicht nur durch den Befehl bestimmt, sondern auch durch den aktuellen Ausführungstakt
 - Wir werden eine FSM (*finite state machine* / endlichen Zustandsautomaten) oder Microcode für die Steuerung benutzen.

- Teile den Befehl in Schritte auf, jeder Schritt benötigt 1 Takt
 - Verteile die zu erledigenden Arbeit
 - In jedem Takt darf nur eine Hauptfunktionseinheit verwendet werden
- Am Ende des Taktzyklus
 - speichere Werte für Gebrauch in späteren Takten (einfachste Lösung)
 - führe zusätzliche „interne“ Register ein:
 - Befehlsregister (IR = *Instruction Register*) und Datenspeicherregister (MDR = *Memory Data Register*) sichern jeweils den gelesenen Befehl und die gelesenen Daten.
 - Zwei Register, weil beide Werte im gleichen Zyklus benötigt werden.
 - A und B speichern die Daten gelesen aus dem Registersatz
 - ALUOut speichert den Output der ALU

- Wir unterscheiden zwischen:
 - **Prozessorstatus:** Register, die für den Programmierer sichtbar sind.
 - Enthalten Daten die durch **darauffolgende Befehle** in einem späteren Zyklus gebraucht werden.
 - Im Falle eines Prozesswechsels (*context switch*), müssen diese Daten gesichert werden
 - **Interner Status:** Register, die für den Programmierer nicht sichtbar sind (IR, MDR, A, B, and ALUout).
 - Enthalten Daten, die durch den **gleichen Befehl** in einem späteren Zyklus gebraucht werden



- MDR, A, B und ALUout benötigen kein Schreibsteuersignal, weil sie Daten nur zwischen aufeinanderfolgenden Taktzyklen halten müssen
- IR benötigt Schreibsteuersignal, weil es den Befehl bis zum Ende der Ausführung halten muss
- Vorheriges Bild berücksichtigt nicht:
 - Bedingte Sprungbefehle (Verzweigungen)
 - Sprungbefehle
 - Alle Steuersignale und Steuerlogik
- Gesamtbild ist auf der nächsten Folie zu sehen
- $t_{cycle} \geq \max(\text{ALU Verzögerung}, \text{Speicherzugriffszeit}, \text{Registerspeicherzugriffszeit})$



Wählt zwischen

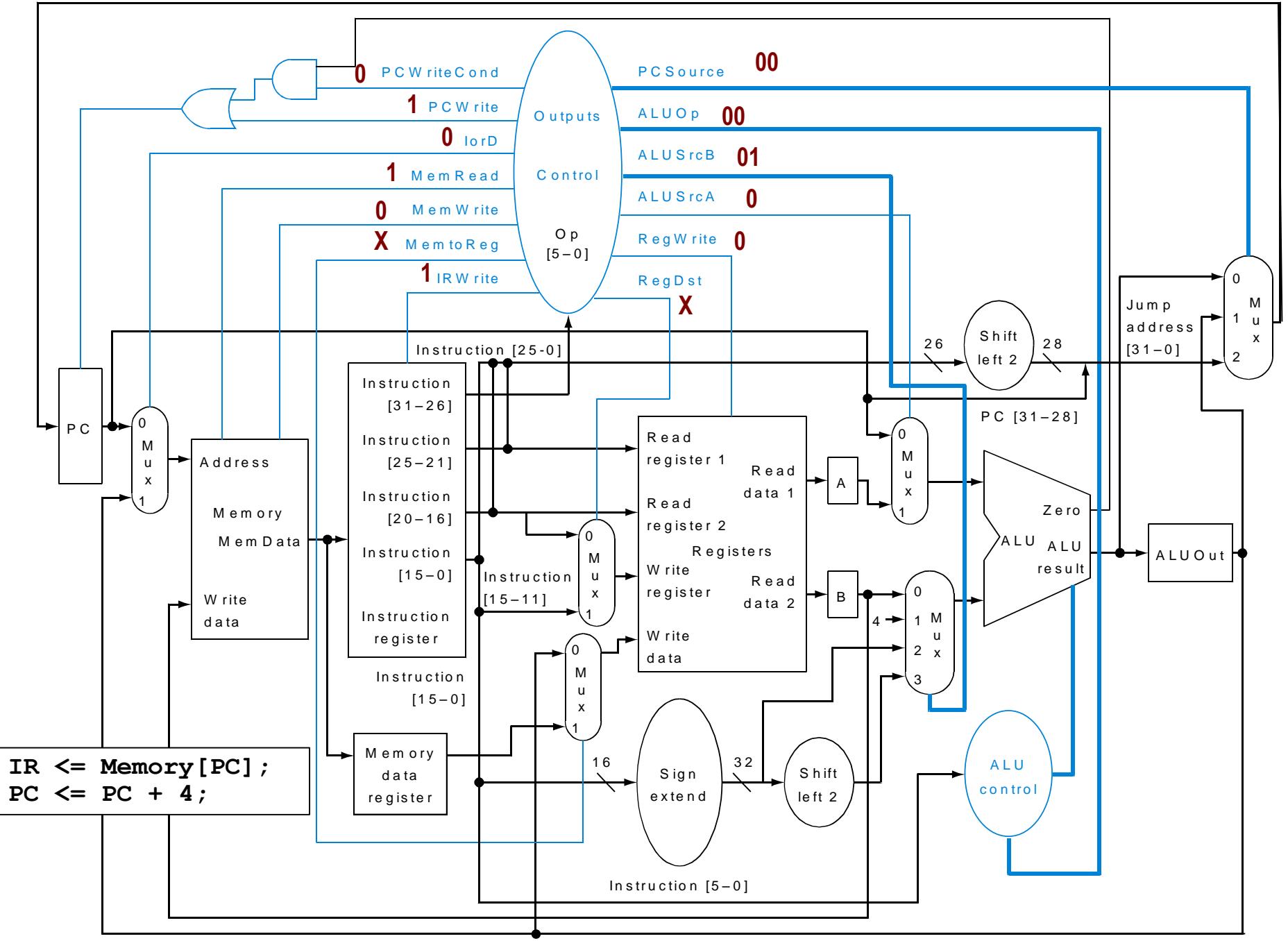
- PC+4
- Verzw.-Adresse (durch ALU berechnet)
- Sprungadresse

1. Befehlsholschritt (*Instruction Fetch*)
2. Befehlsentschlüsselungs- und Registerholschritt (*Instruction Decode/Register Fetch*)
3. Ausführung, Berechnung der Speicheradresse oder Sprungausführung (*Execute, memory address computation, or branch completion*)
4. Speicherzugriff oder R-Befehlabschlussschritt (*Memory Access or R-type instruction completion*)
5. Speicherleseabschlusschritt (*memory read completion*)

➤ Befehle dauern zwischen 3 und 5 Taktschritte

- „Send“ PC zum Speicher um den Befehl zu laden und ins Instruktionsregister zu speichern
- Inkrementiere PC mit 4 und speichere Ergebnis zurück in den PC
- Kann kurz und bündig durch eine **RTL (Register-Transfer Language/Registertransfersprache)** beschrieben werden:

```
IR <= Memory [PC] ;  
PC <= PC + 4 ;
```
- Was ist der Vorteil den PC in diesen Schritt zu aktualisieren?
- Welche Werte sollen die Steuersignale haben?





1. Schritt: Befehlsholschritt

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

- Welche Werte sollen die Steuersignale haben?
 - RegWrite, RegDst, MemtoReg:0, X, X
 - ALUSrcA: 0 (wähle PC)
 - ALUSrcB: 01 (wähle Konstanten 4)
 - ALUOp: 00 (addiere)
 - PCSource: 00 (wähle PC+4 berechnet durch ALU)
 - MemRead, MemWrite: 1, 0 (lese Befehl aus Speicher, schreibe nicht)
 - lorD, IRWrite: 0, 1 (sende PC zum Speicher, schreibe IR)
 - PCWriteCond, PCWrite: 0, 1 (PCWriteCond kann X sein, da PCWrite=1)

- Lese die Register **rs** und **rt**, falls benötigt
- Berechne Sprungadresse, falls benötigt
- Diese Aktionen wurden **spekulativ** ausgeführt!!! Warum?
- RTL:

```
A <= Reg[IR[25:21]] ;  
B <= Reg[IR[20:16]] ;  
ALUOut <= PC + (sign-extend(IR[15:0]) << 2) ;
```
- Bemerke: Wir setzen keine Steuersignale basierend auf dem Befehlstyp (wir „decodieren“ den Befehl z. Z. in der Steuerlogik)

ALU führt eine von vier Funktionen aus, abhängig vom Befehlstyp

- Berechnung der Speicheradresse:

$$\text{ALUOut} \leq A + \text{sign-extend}(\text{IR}[15:0]);$$

- R-type:

$$\text{ALUOut} \leq A \text{ op } B;$$

- Branch:

$$\text{if } (A==B) \text{ PC } \leq \text{ALUOut};$$

- Sprungzieladresse in ALUOut wurde im vorigen Schritt berechnet

- Jump:

$$\text{PC } \leq \{\text{PC}[31:28], (\text{IR}[25:0] \ll 2)\}$$

- Lade und Speicherbefehle greifen auf den Speicher zu:
 - Laden:
 $MDR \leq Memory[ALUOut];$
 - Speichern:
 $Memory[ALUOut] \leq B;$
 - Bemerke: B wird eigentlich zweimal gelesen, im 2. und 3. Schritt. Kein Problem da IR nicht geändert.
- R-Befehle werden abgeschlossen:
 $Reg[IR[15-11]] \leq ALUOut;$

- Ladebefehle werden abgeschlossen:

$$\text{Reg}[\text{IR}[20-16]] \leq \text{MDR};$$

Was ist mit den anderen Befehlen?

Schritte erforderlich für die Ausführung aller Befehlsklassen:

Schritt	R-Typ Befehle	Speicher- befehle	Verzwei- gungen	Sprünge	
Befehlsholschritt		$IR = \text{Memory}[PC]$ $PC = PC + 4$			
Befehlsentschlüsselungs- und Registerholschritt		$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $ALUOut = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$			
Ausführung oder Adressberechnung oder Sprungausführung	$ALUOut = A \text{ op } B$	$ALUOut = A + \text{sign-extend}(IR[15-0])$	if ($A == B$) then $PC = ALUOut$	$PC = PC[31-28] \parallel (IR[25-0]\ll 2)$	
Speicherzugriff oder R- Befehlabschlusschritt	$\text{Reg}[IR[15-11]] = ALUOut$	Load: $MDR = \text{Mem}[ALUOut]$ Store: $\text{Mem}[ALUOut] = B$			
Speicherleseabschluss		Load: $\text{Reg}[IR[20-16]] = MDR$			

- Wie viele Taktzyklen wird es dauern, um diesen Code auszuführen?

lw \$t2,0(\$t3)

lw \$t3,4(\$t3)

beq \$t2,\$t3,L1 # nehme an, nicht genommen

add \$t5,\$t2,\$t3

sw \$t5,8(\$t3)

L1: . . .

– $5 + 5 + 3 + 4 + 4 = 21$ Taktzyklen

- Was passiert im 8. Ausführungszyklus?

– 3. Ausführungszyklus des 2. **lw** → ALU berechnet die Adresse

- In welchem Zyklus findet die eigentliche Addition von \$t2 und \$t3 statt?

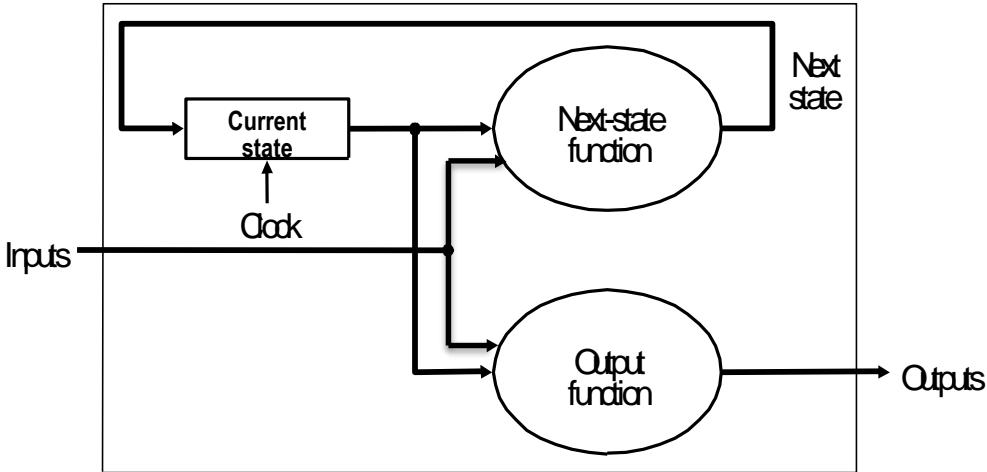
– 3. Zyklus von add = 16. Ausführungzyklus insgesamt

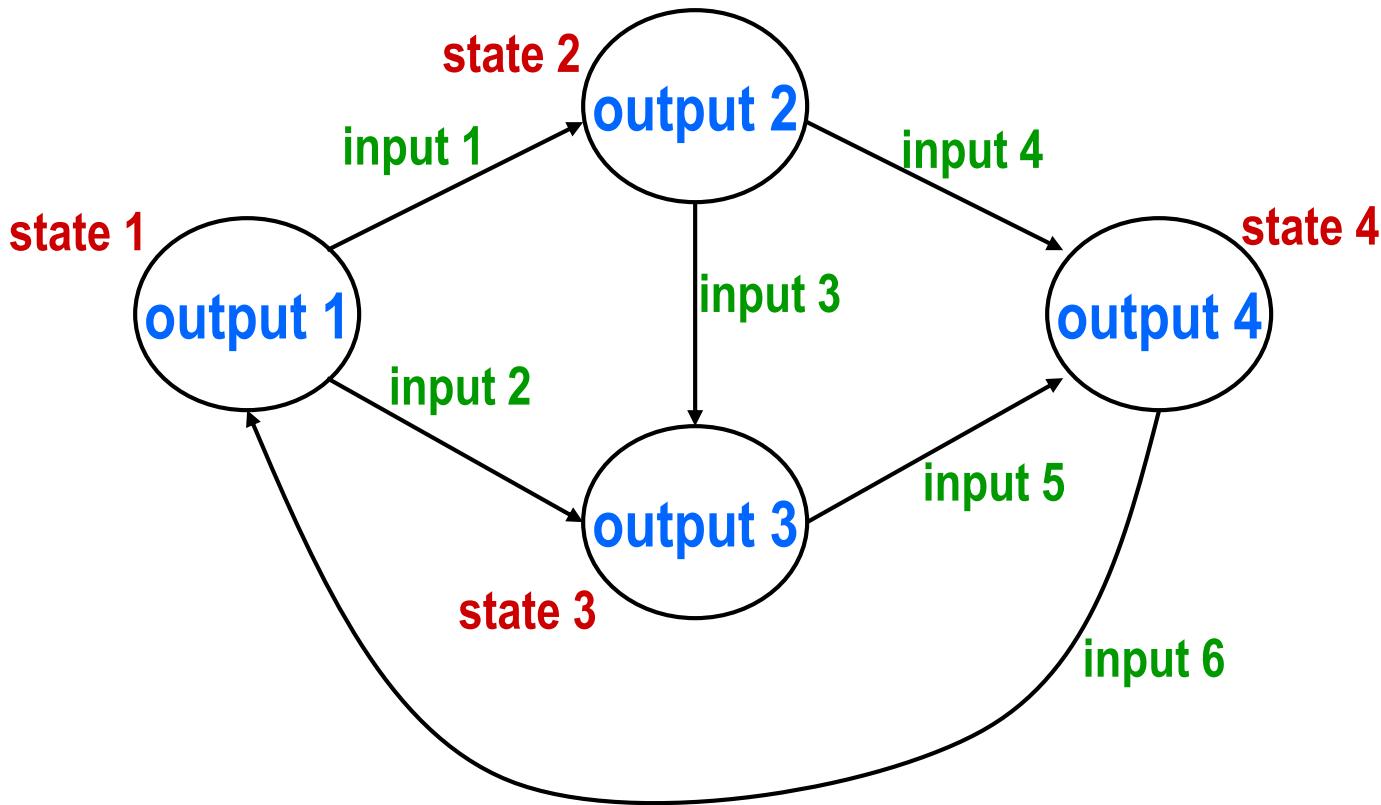


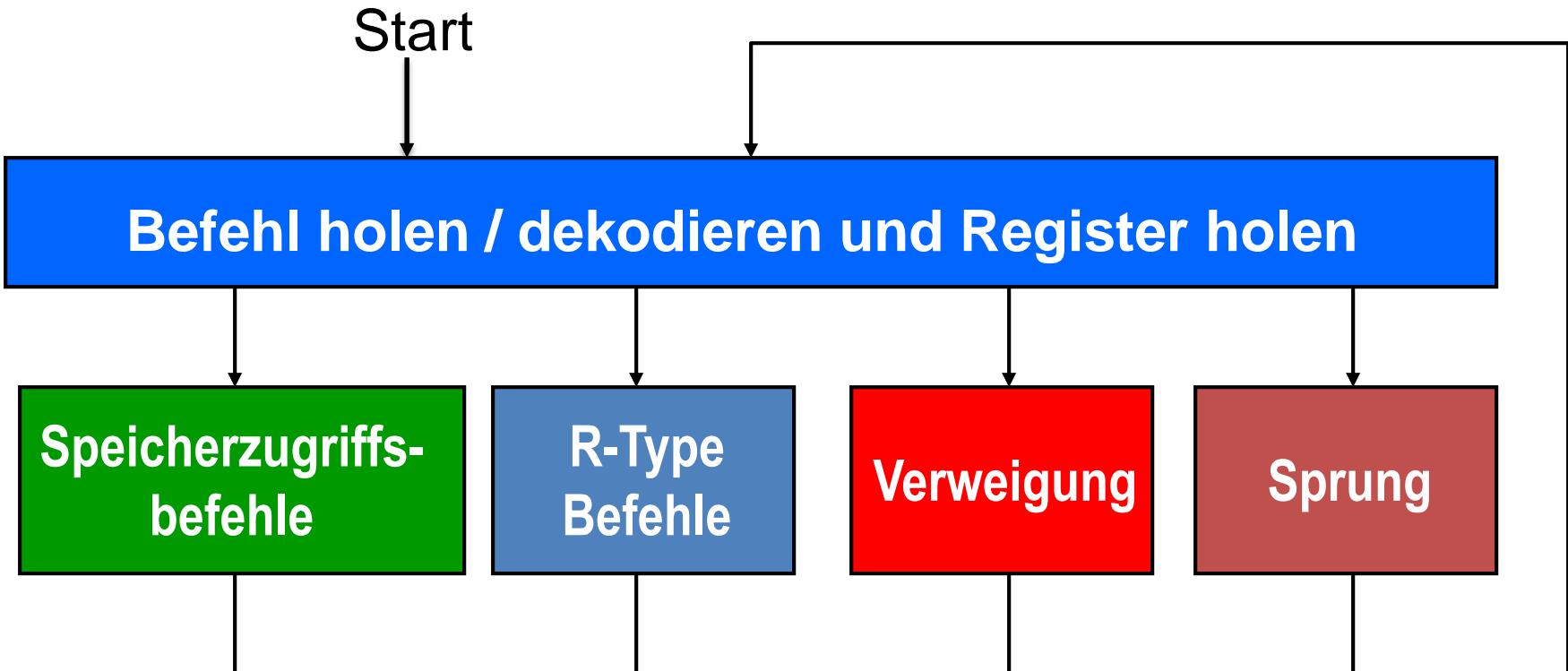
- SPECINT2000 hat folgender Befehlsmix:
 - 25% Ladebefehle
 - 10% Speicherbefehle
 - 11% Verzweigungen
 - 2% Sprünge
 - 52% ALU Befehle
- Was ist der durchschnittliche CPI der Mehrzyklenimplementierung?
 - $\text{CPI} = 0.25 \times 5 + 0.1 \times 4 + 0.11 \times 3 + 0.02 \times 3 + 0.52 \times 4 = 4.12$

- Steuersignale hängen ab:
 - Welcher Befehl wird ausgeführt
 - Aktueller Ausführungsschritt der Instruktion
- Benutze die vorhandene Information um einen endlichen Zustandsautomaten (*finite state machine, FSM*) zu spezifizieren:
 - Spezifizierte FSM **graphisch**, oder
 - Benutze **Mikroprogrammierung**
- Implementierung kann von der Spezifikation abgeleitet werden.

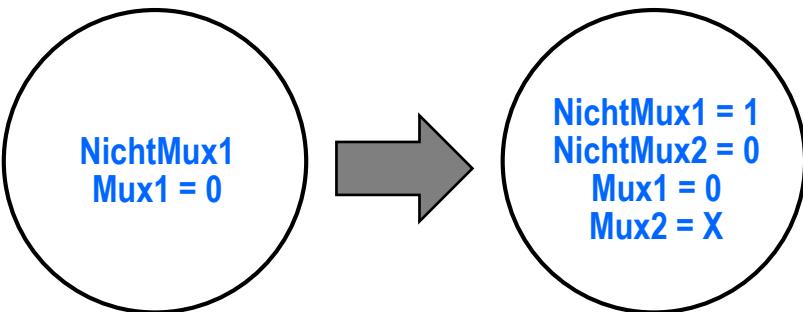
- Endlicher Zustandsautomat:
 - Menge Zustände (S)
 - Zustandabbildungsfunktion (*next-state function*) ($S \times I \rightarrow S$)
 - nächster Zustand abhängig vom aktuellen Zustand und Eingabe
 - Ausgabefunktion (*output function*) ($S \times I \rightarrow O, S \rightarrow O$)
 - Ausgabe abhängig vom aktuellen Zustand und möglich Eingabe)
- Wir werden Moore Automaten verwenden
 - Ausgabe hängt nur vom aktuellen Zustand ab

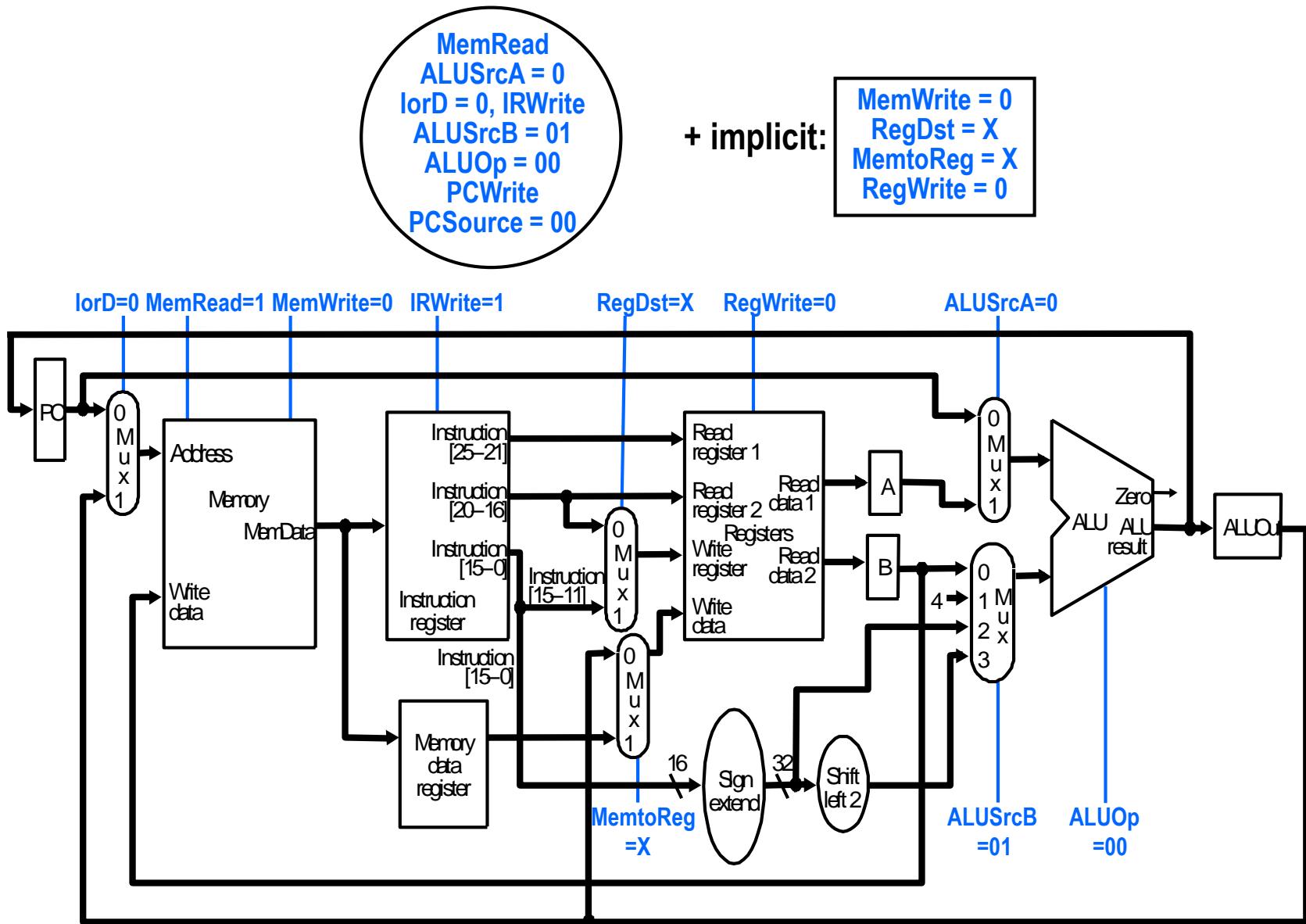






- Nicht-MUX-Steuersignale (z. B. RegWrite) die angegeben werden, sind logisch 1
- Nicht-MUX-Steuersignale die nicht angegeben werden, sind logisch 0
- MUX-Steuersignale (z. B. MemtoReg) die nicht angegeben werden, sind Don't-Cares
- Beispiel:
 - 4 Steuersignale: NichtMux1, NichtMux2, Mux1, Mux2

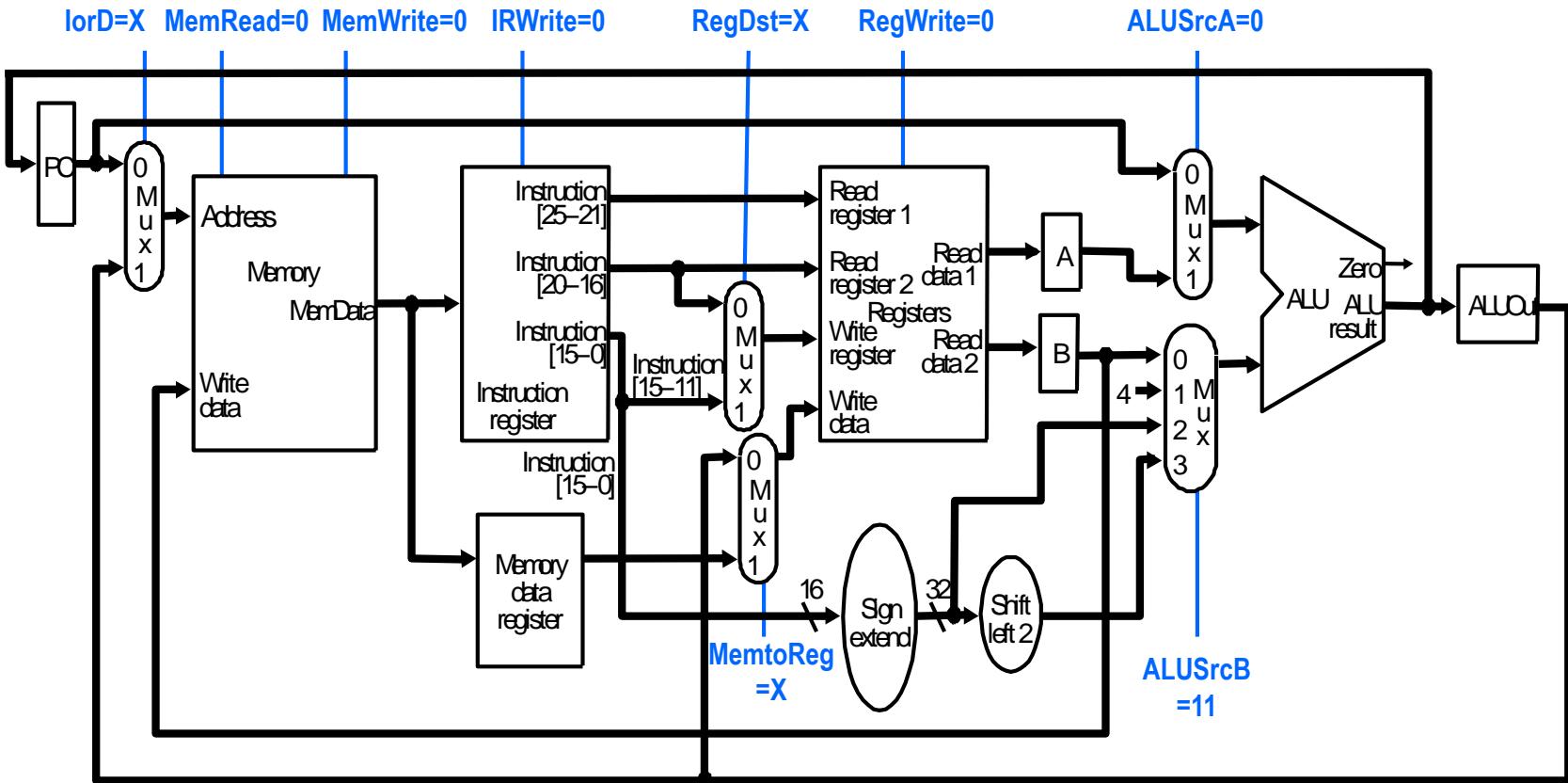


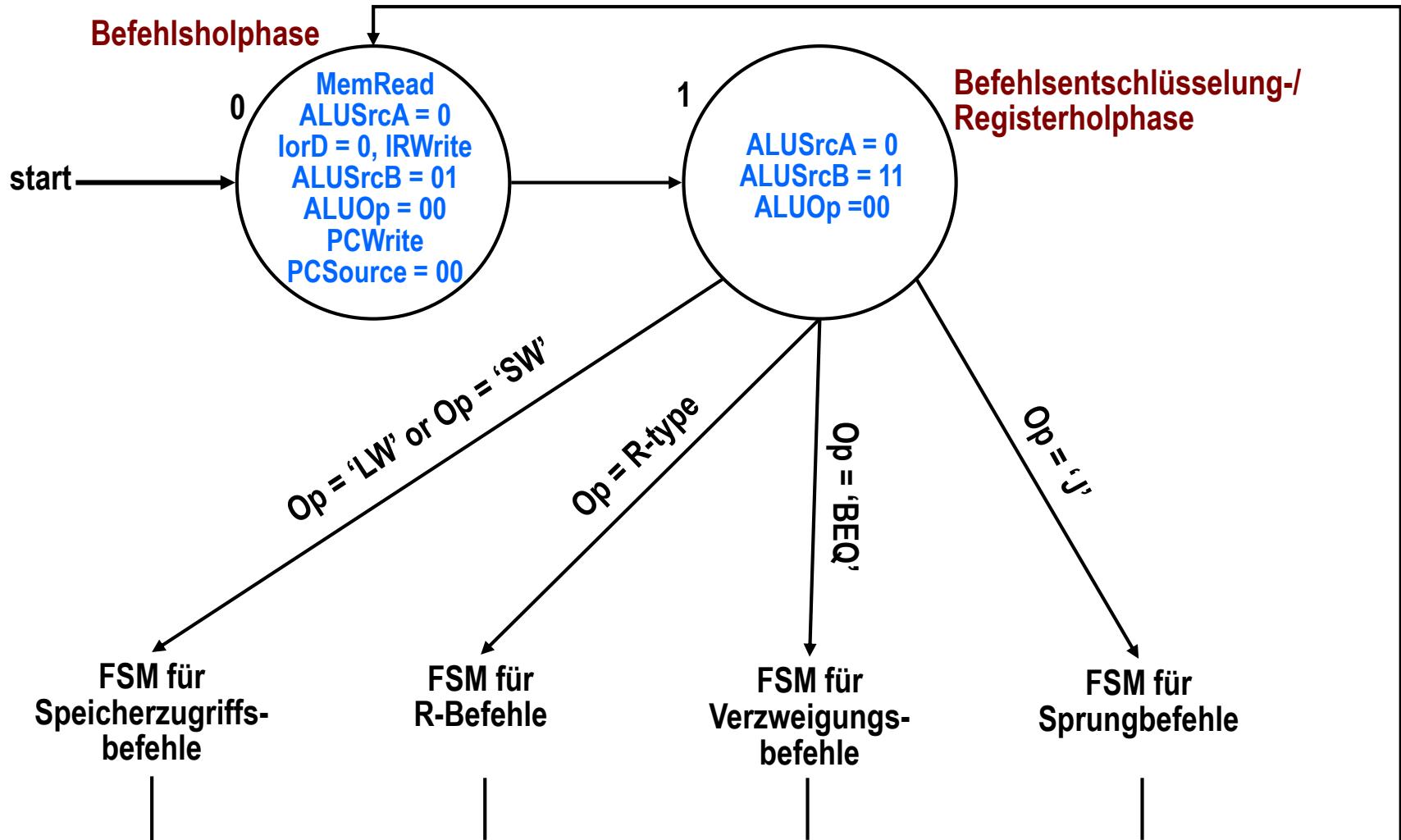


ALUSrcA = 0
 ALUSrcB = 11
 ALUOp = 00

```

A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
    
```





Berechnung der Speicheradresse

Von Zustand 1

$Op = 'LW'$ or $Op = 'SW'$

2
 $ALUSrcA = 1$
 $ALUSrcB = 10$
 $ALUOp = 00$

$Op = 'SW'$

Speicherzugriff

3
 Speicherzugriff
 $MemRead$
 $IorD = 1$

5
 Speicherzugriff
 $MemWrite$
 $IorD = 1$

4
 Speicherlese-abschluss
 $RegWrite$
 $MemtoReg = 1$
 $RegDst = 0$

Zu Zustand 0

Laden/Speichern:
 $ALUOut = A +$
 $\text{sign-extend}(IR[15-0])$

Laden:
 $MDR \leq \text{Memory}[ALUOut]$;
 $\text{Reg}[IR[20-16]] \leq MDR$

Speichern:
 $\text{Memory}[ALUOut] \leq B$;

Berechnung der Speicheradresse

Von Zustand 1

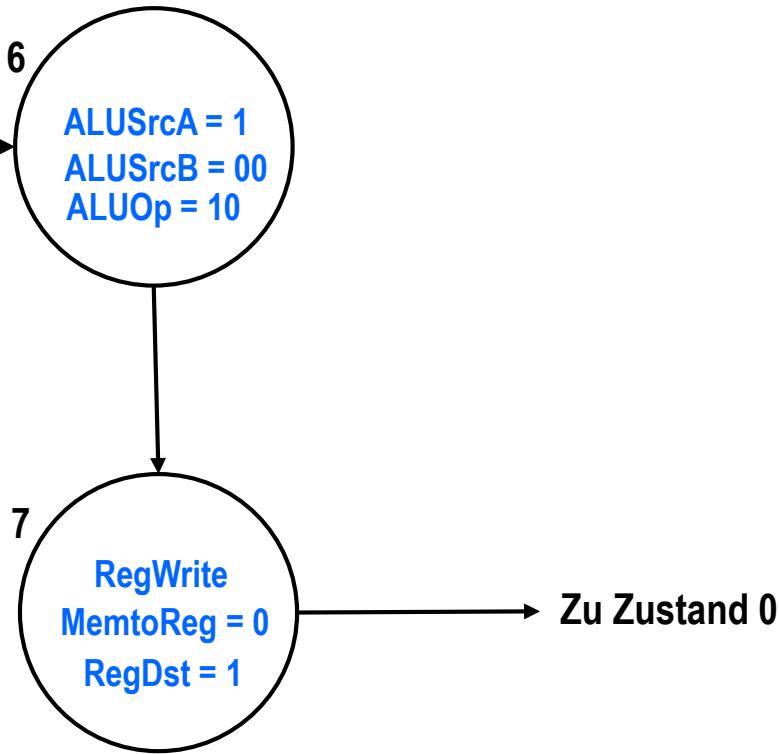
Op = 'R-Type'

Ausführung:

$ALUOut \leq A \text{ op } B;$

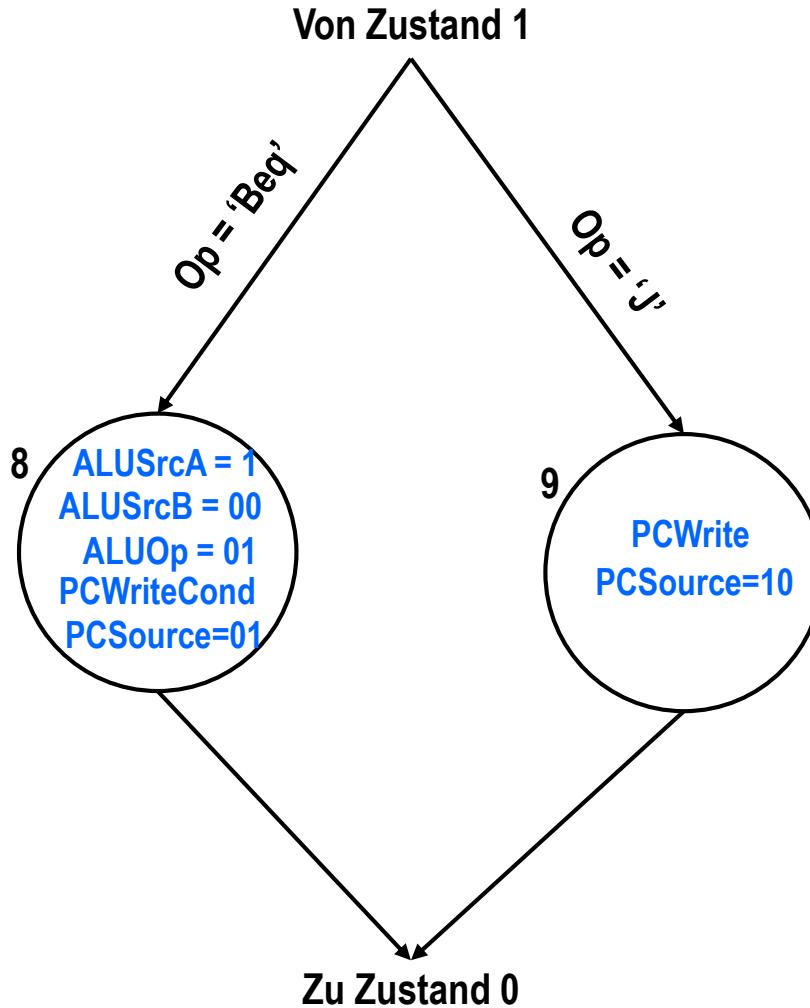
Abschluss:

$Reg[IR[15-11]] \leq ALUOut;$



Berechnung der Speicheradresse

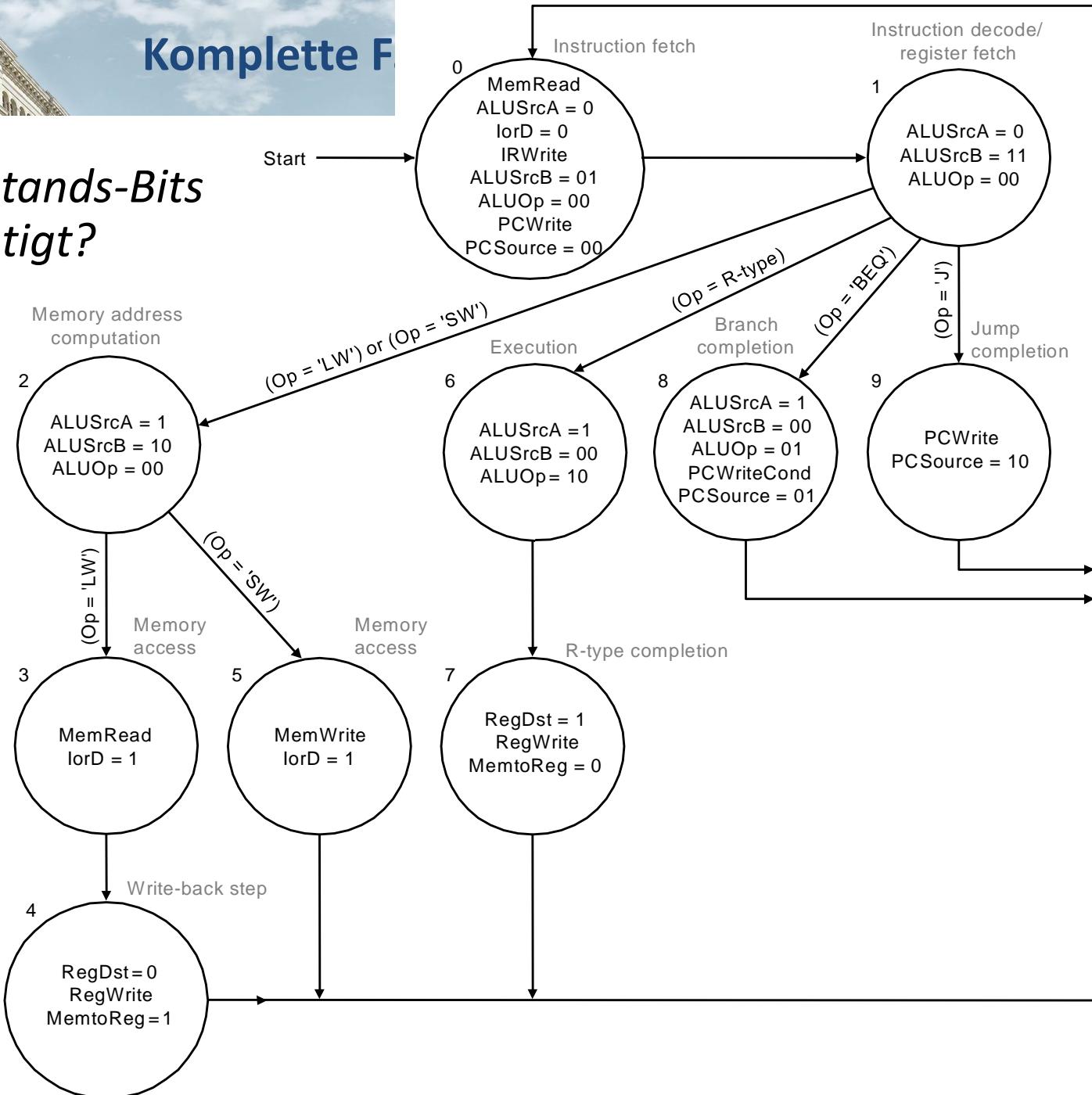
```
Branch:  
if (A==B) PC <= ALUOut;  
  
Jump:  
PC <=  
{PC[31:28], (IR[25:0]<<2)}
```



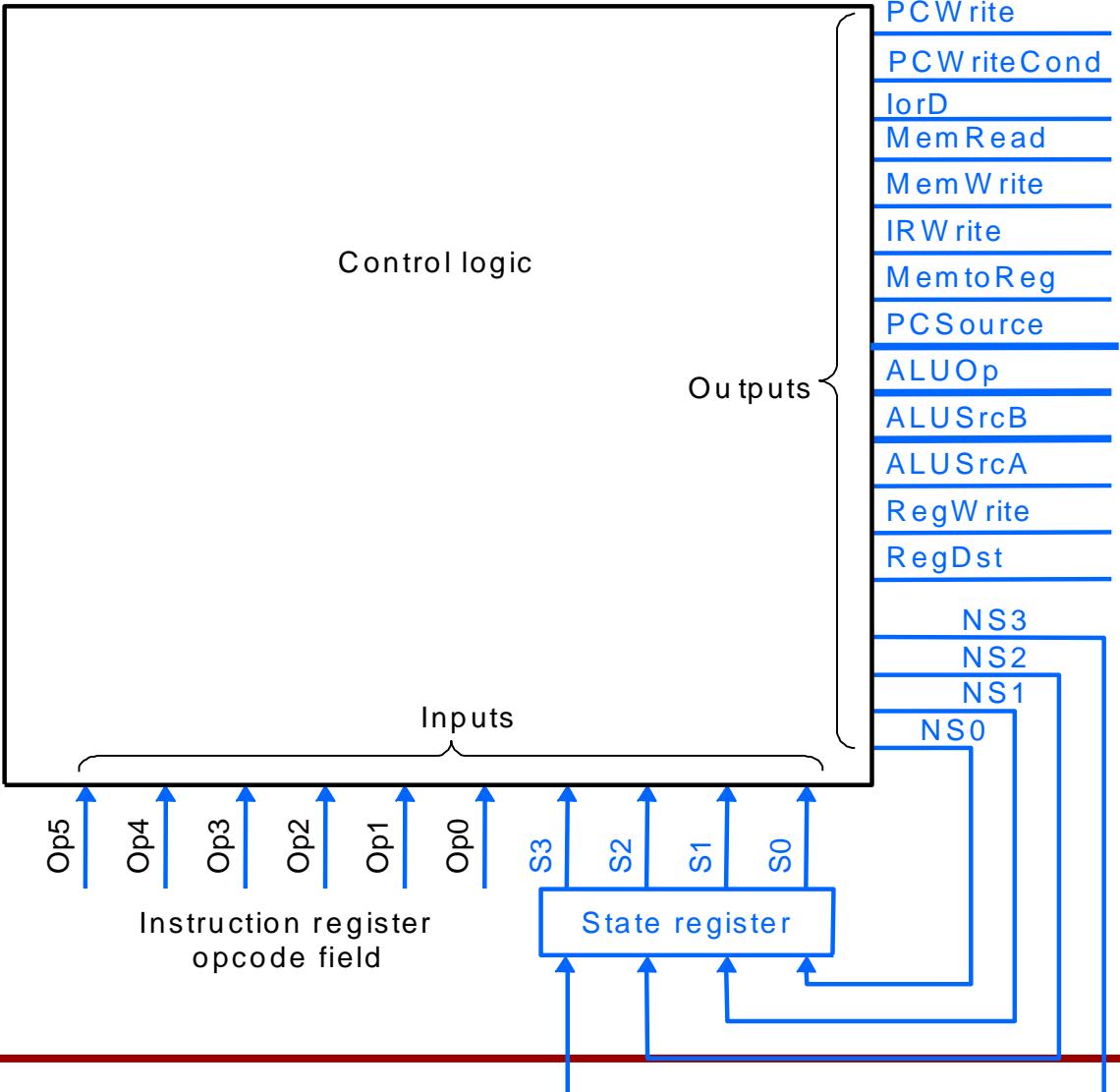


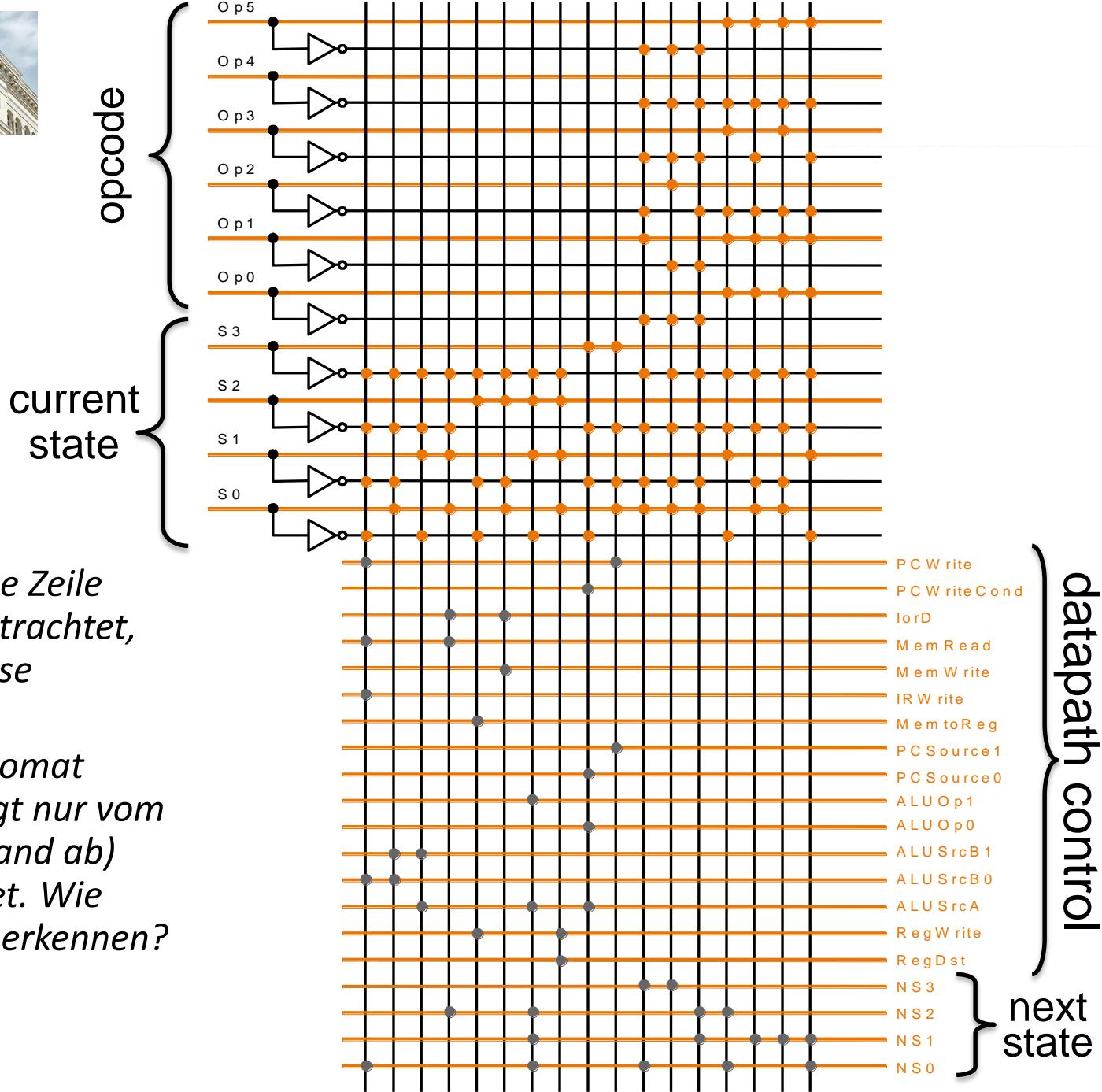
Komplette F

- Wie viele Zustands-Bits werden benötigt?



- Typische Implementierung einer FSM-Steuerung:
kombinatorischer Logikblock + Zustandsregister





- Wenn man eine Zeile oder Spalte betrachtet, können Sie diese erklären?
- Ein Moore Automat (Ausgabe hängt nur vom aktuellen Zustand ab) wird verwendet. Wie kann man das erkennen?

- Echte Prozessoren verfügen über viele Befehle → komplexe FSM mit vielen Zuständen
 - graphische Spezifikation schwierig wenn nicht sogar unmöglich
- Spezifizierte die Steuerung mit Befehlen
 - **Mikro-Befehle**
 - Besteht aus separaten Feldern (für die Steuerung der ALU, SRC1, SCR2, etc.)

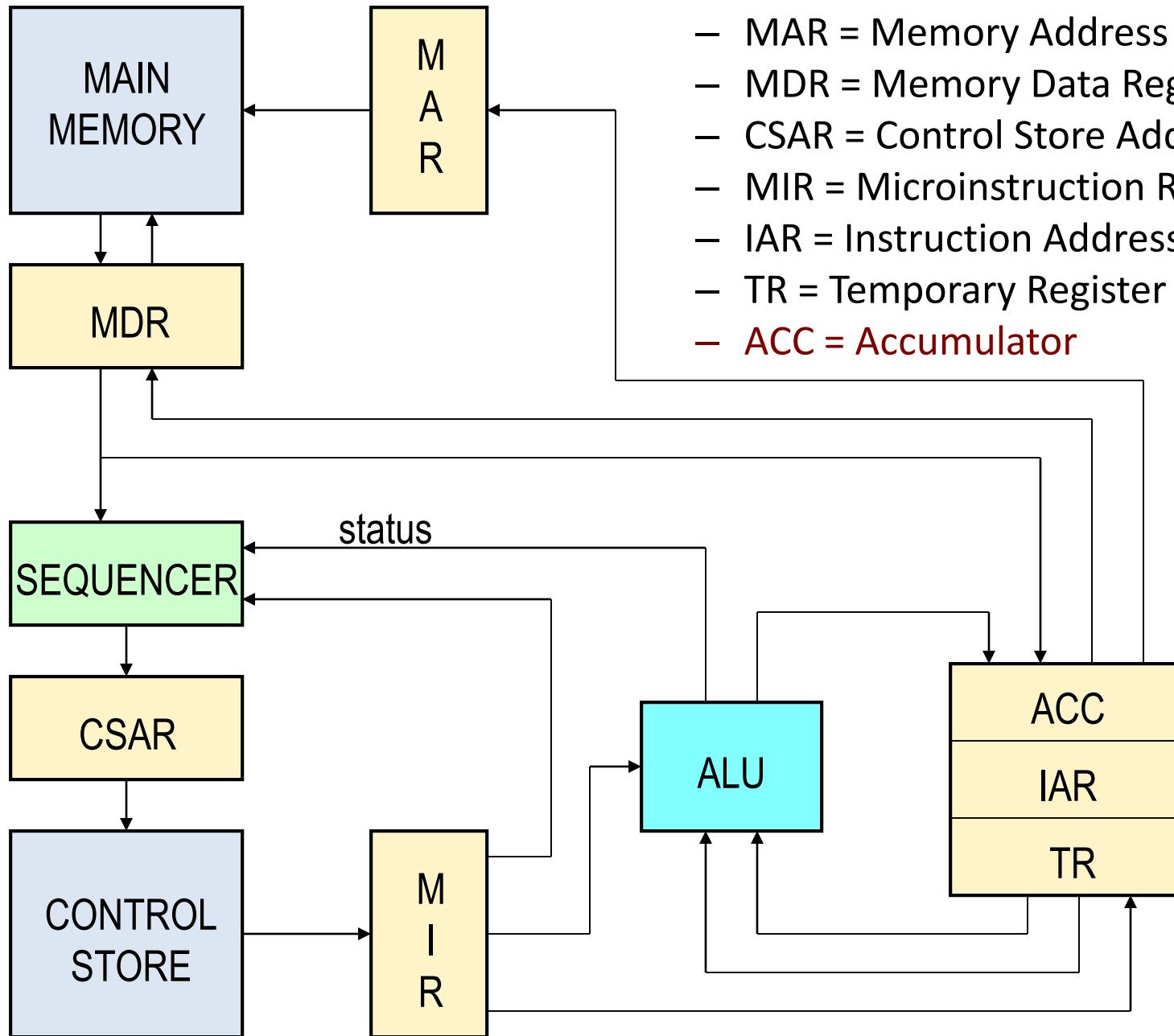
*“... consider the **control** proper, that is, the part of the machine which supplies the pulses for operating the gates associated with the arithmetical and control registers. The designer of this part of a machine usually proceeds in an **ad hoc manner** [...]. I would like to suggest a way in which the control can be made systematic, and therefore less complex.*

*Each operation called for by an **order** in the order code of the machine involves a sequence of steps which may include transfers from the store to control or arithmetical registers, or vice versa, and transfers from one register to another. Each of these steps is achieved by pulsing certain wires associated with the control and arithmetical registers, and I will refer to it as a “**micro-operation**.” Each true machine operation is thus made up of a sequence or “**microprogramme**” of micro-operations.”*

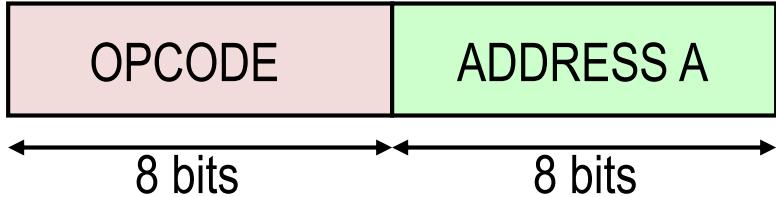
- Maurice Wilkes (1951)

- Nächste Folien basiert auf: *Microprogramming: A Tutorial and Survey of Recent Developments*, T.G. Rauscher und P.M. Adams, IEEE Trans. on Computers, C-29(1), Jan. 1980.

Ein einfacher mikroprogrammierter Computer



- MAR = Memory Address Register
- MDR = Memory Data Register
- CSAR = Control Store Address Register
- MIR = Microinstruction Register
- IAR = Instruction Address Register
- TR = Temporary Register
- ACC = Accumulator



Name	Bedeutung
Add	$ACC \leftarrow ACC + Memory[A]$
Sub	$ACC \leftarrow ACC - Memory[A]$
Load	$ACC \leftarrow Memory[A]$
Store	$Memory[A] \leftarrow ACC$
Jump	goto A
Jump if zero	goto A if $ACC == 0$

Mikrobefehle

- Registertransfers:

MDR \leftarrow **Register**

Register \leftarrow **MDR**

MAR \leftarrow **Register**

- Speicherzugriffe:

Read # **MDR** = **Memory** [**MAR**]

Write # **Memory** [**MAR**] = **MDR**

- Sequencing-Operationen:

CSAR \leftarrow **CSAR+1** (default)

```
    CSAR  $\leftarrow$  decoded MDR # goto subroutine that
                                # implements the operation
    CSAR  $\leftarrow$  constant      # CSAR $\leftarrow$ 0 is return to
        start
    SKIP      # add 2 to CSAR if ACC=0, add 1 otherwise
```

- ALU-Operationen:

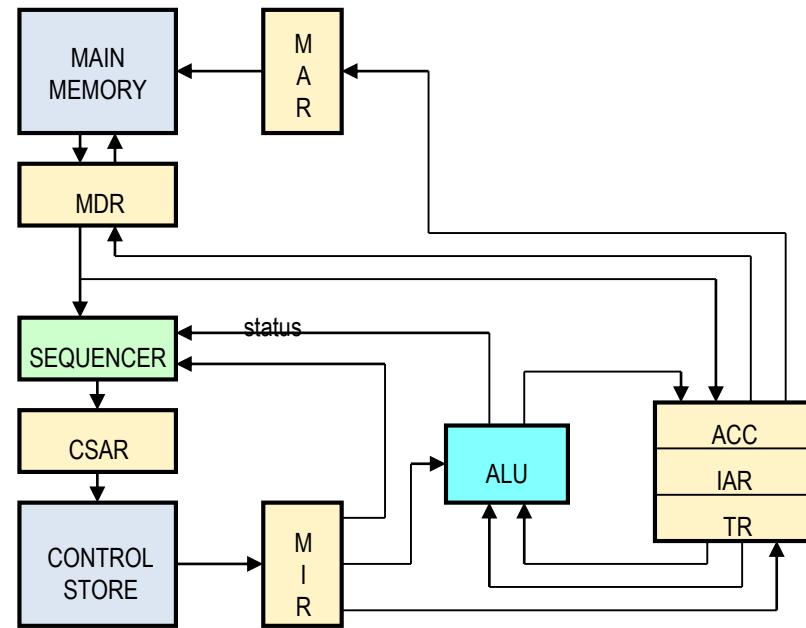
ACC \leftarrow **ACC+Register**

ACC \leftarrow **ACC-Register**

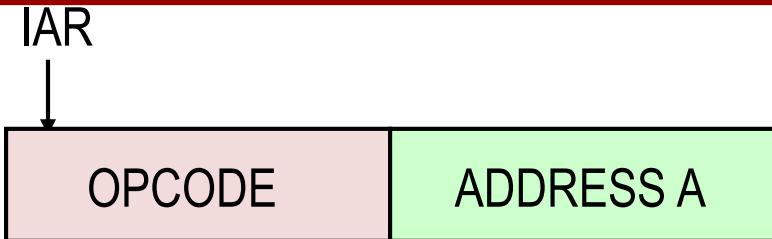
ACC \leftarrow **Register**

Register \leftarrow **ACC**

ACC \leftarrow **Register+1**



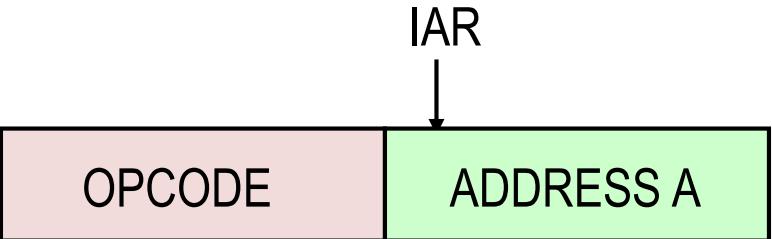
*Warum ist z.B.
Register \leftarrow **MAR**
kein Mikrobefehl?*



Befehlsholphase:

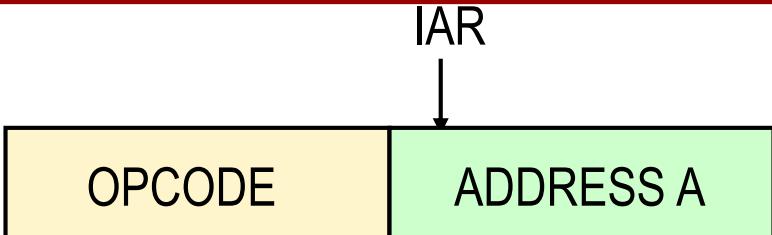
```

MAR ← IAR           ; move instruction
Read                ; read instr. opcode into MDR
ACC ← IAR+1         ; increment instr. address register
IAR ← ACC           ; (IAR now points to address A)
CSAR ← decoded MDR ; goto routine that performs
                      ; operation
    
```



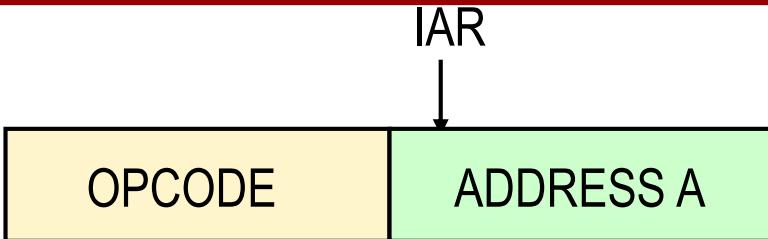
```

MDR ← Register
Register ← MDR
MAR ← Register
Read   # MDR = Memory[MAR]
Write  # Memory[MAR] = MDR
CSAR ← CSAR+1 (default)
CSAR ← decoded MDR
CSAR ← constant
SKIP
ACC ← ACC+Register
ACC ← ACC-Register
ACC ← Register
Register ← ACC
ACC ← Register+1
    
```



Addiere Speicher zum Akkumulator (`add A # ACC` \leftarrow `ACC+Memory[A]`):

<code>MAR ← IAR</code>	<code>; move address of address A to MAR</code>
<code>Read</code>	<code>; read address A into MDR</code>
<code>ACC ← IAR+1</code>	<code>} kann nicht korrekt sein, alte Daten in ACC gehen</code>
<code>IAR ← ACC</code>	<code>} verloren! Sichere ACC in TR und stelle es wieder her</code>
<code>TR ← MDR</code>	<code>; load address A into TR</code>
<code>MAR ← TR</code>	<code>; load address A into MAR</code>
<code>Read</code>	<code>; read data into MDR</code>
<code>TR ← MDR</code>	<code>; move data to TR</code>
<code>ACC ← ACC+TR</code>	<code>; add data to ACC</code>
<code>CSAR ← 0</code>	<code>; return to instruction fetch</code>

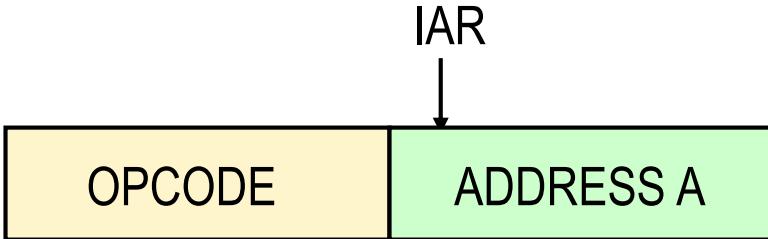


Subtrahiere Speicher vom Akkumulator (weggelassen)

Lade Akkumulator vom Speicher (**load A # ACC ← Memory [A]**):

MAR ← IAR	; move address of address A to MAR
Read	;
ACC ← IAR+1	;
IAR ← ACC	;
TR ← MDR	;
MAR ← TR	;
Read	;
into MDR	; read data pointed to by A
ACC ← MDR	;
CSAR ← 0	;

Auch ein Fehler?



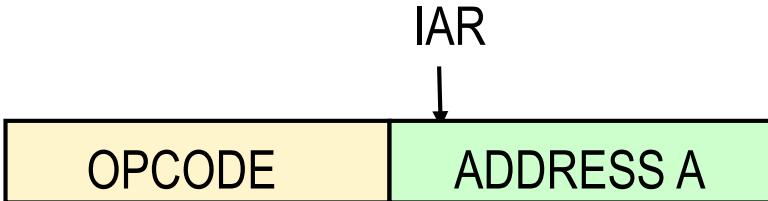
Speichere Akkumulator in Speicher (weggelassen)

Sprung (**jump A**):

```
MAR ← IAR ; move address of address A
to MAR

Read ; MDR = Memory [A]

IAR ← MDR ;
CSAR ← 0 ;
```



Springe wenn Null (`jz A # if (ACC==0) goto A`):

```

MAR ← IAR      ; move address of target address to MAR
Read           ; read target address into MDR
SKIP           ; skip next instruction if ACC==0
CSAR ← 0       ; return to instruction fetch if ACC!=0
IAR ← MDR     ; load target address into IAR
CSAR ← 0       ; return to instruction fetch
  
```

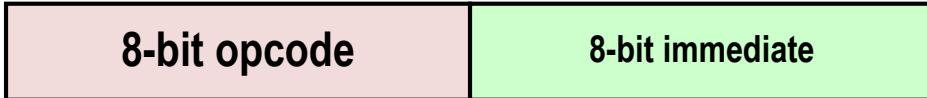
Was ist der Fehler hier?

Falls $ACC!=0$, IAR zeigt noch auf Adresse A
 Füge ein:

```

TR ← ACC
ACC ← IAR+1
IAR ← ACC
ACC ← TR
  
```

- Wir wollen den Befehl “*add immediate*” dem mikroprogrammierten Computer hinzufügen.
- Das Format dieses Befehls ist:



Name	Bedeutung
Addi	$\text{ACC} \leftarrow \text{ACC} + \text{imm8}$

- Gebe die Mikroprogrammroutine, die diese Instruktion ausführt.

- Abschnitt 5.7 (auf CD)
- Komplexer als vorheriges Beispiel
 - **Horizontale Mikro-Befehle**, die mehrere Ressourcen parallel kontrollieren
 - Vorherige Beispiel war **vertikale Mikro-Befehle**, die einzelne Operationen beeinflussen

- Jeder Mikrobefehl besteht aus verschiedenen Feldern:
 - **ALU control**: bestimmt die ALU Operation, wie vorher
 - **SRC1**: wählt den 1. ALU Input
 - **SRC2**: wählt den 2. ALU Input
 - **Register control**: bestimmt, ob die Registerdatei gelesen oder geschrieben wird (mit ALUOut (for R-type) oder MDR (für \perp_w))
 - **Memory**: bestimmt, ob der Speicher gelesen wird (wobei PC oder ALUOut (für \perp_w) als Adresse genutzt werden) oder geschrieben wird (für s_w)
 - **PC Write control**: bestimmt den nächsten Wert des PC
 - **Sequencing**: bestimmt den nächsten Mikrobefehl

Field	Value	Active signals	Remark
ALU control	Add	ALUOp = 00	Cause ALU to add
	Subt	ALUOp = 01	Cause ALU to subtract
	Func code	ALUOp = 10	Use instruction's function code to determine ALU control
SRC1	PC	ALUSrcA = 0	Use PC as first ALU input
	A	ALUSrcA = 1	Register A is first ALU input
SRC2	B	ALUSrcB = 00	Register B is second ALU input
	4	ALUSrcB = 01	4 is second ALU input
	Extend	ALUSrcB = 10	output of sign extension unit is 2 nd ALU input
	Extshft	ALUSrcB = 11	output of shift-by-2 unit is 2 nd ALU input
Register control	Read		Read registers rs and rt and put data into A and B
	Write ALU	RegWrite, RegDst=1, MemtoReg=0	Write data in ALUOut to register rd
	Write MDR	RegWrite, RegDst=0, MemtoReg=1	Write data in MDR to register rt

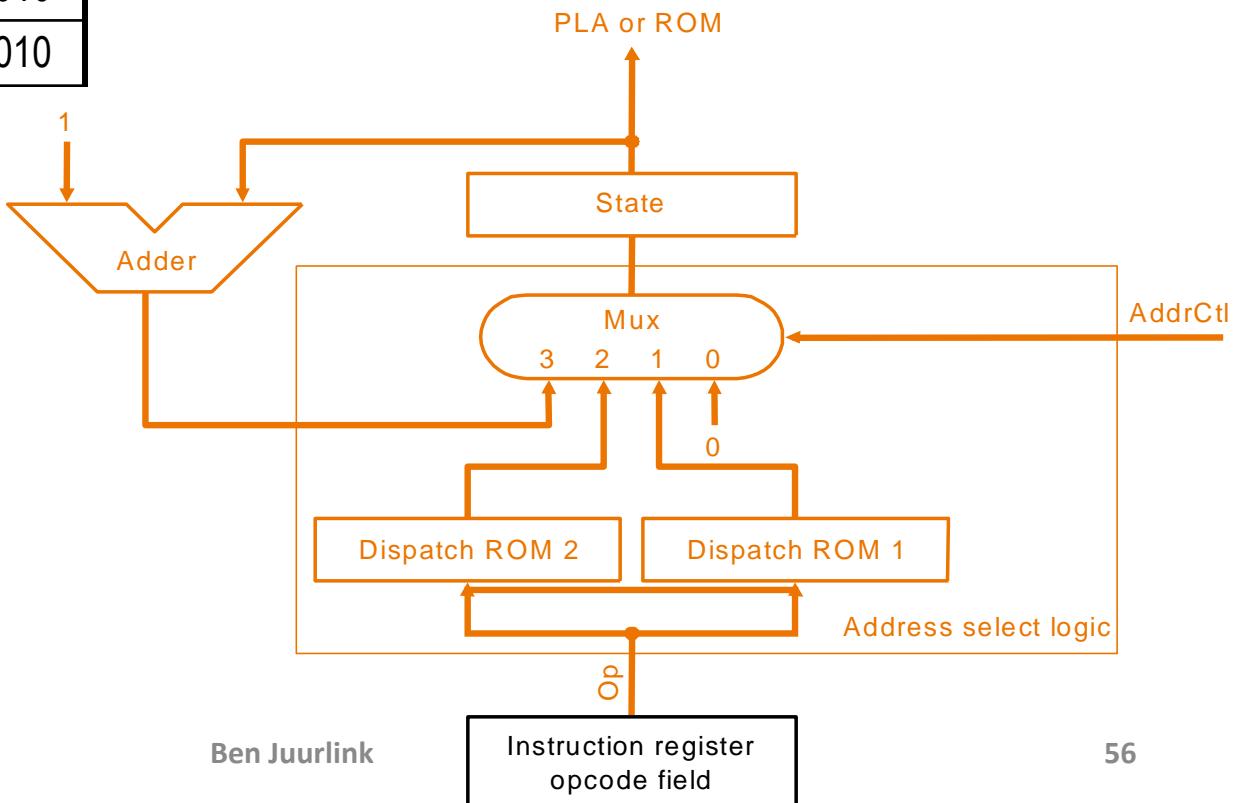
Field	Value	Active signals	Remark
Memory	Read PC	MemRead, lorD=0	IR (and MDR) \leftarrow Memory[PC]
	Read ALU	MemRead, lorD=1	MDR \leftarrow Memory[ALUOut]
	Write ALU	MemWrite, lorD=1	Memory[ALUOut] \leftarrow B
PC write control	ALU	PCSsource=00, PCWrite	Write ALU output into PC
	ALUOut-cond	PCSsource=01, PCWriteCond	if (ALU Zero output) PC \leftarrow ALUOut
	jump address	PCSsource=10, PCWrite	Write jump address from instr. into PC
Sequencing	Seq	AddrCtl=11	Choose next microinstruction sequentially
	Fetch	AddrCtl=00	Go to first microinstruction to begin new instruction
	Dispatch 1	AddrCtl=01	Dispatch using ROM 1
	Dispatch 2	AddrCtl=10	Dispatch using ROM2

Address	Label	ALU control	SRC 1	SRC2	Register control	Memory	PCWrite control	Sequencing
0000	Fetch	Add	PC	4		Read PC	ALU	Seq
0001		Add	PC	Extshft	Read			Dispatch 1
0010	Mem1	Add	A	Extend				Dispatch 2
0011	LW2					Read ALU		Seq
0100					Write MDR			Fetch
0101	SW2					Write ALU		Fetch
0110	RFormat1	Func code	A	B				Seq
0111					Write ALU			Fetch
1000	BEQ1	Subt	A	B			ALUOut-cond	Fetch
1001	JUMP1						Jump address	Fetch

Dispatch ROM 1		
Opcode	Opcode name	Value
000000	R-format	0110
000010	j	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Opcode	Opcode name	Value
100011	lw	0011
101011	sw	0101

Dispatch ROMs werden durch den Opcode indiziert und beinhalten die Startadressen der Mikroprogramm-Routinen die die Maschinenbefehle ausführen



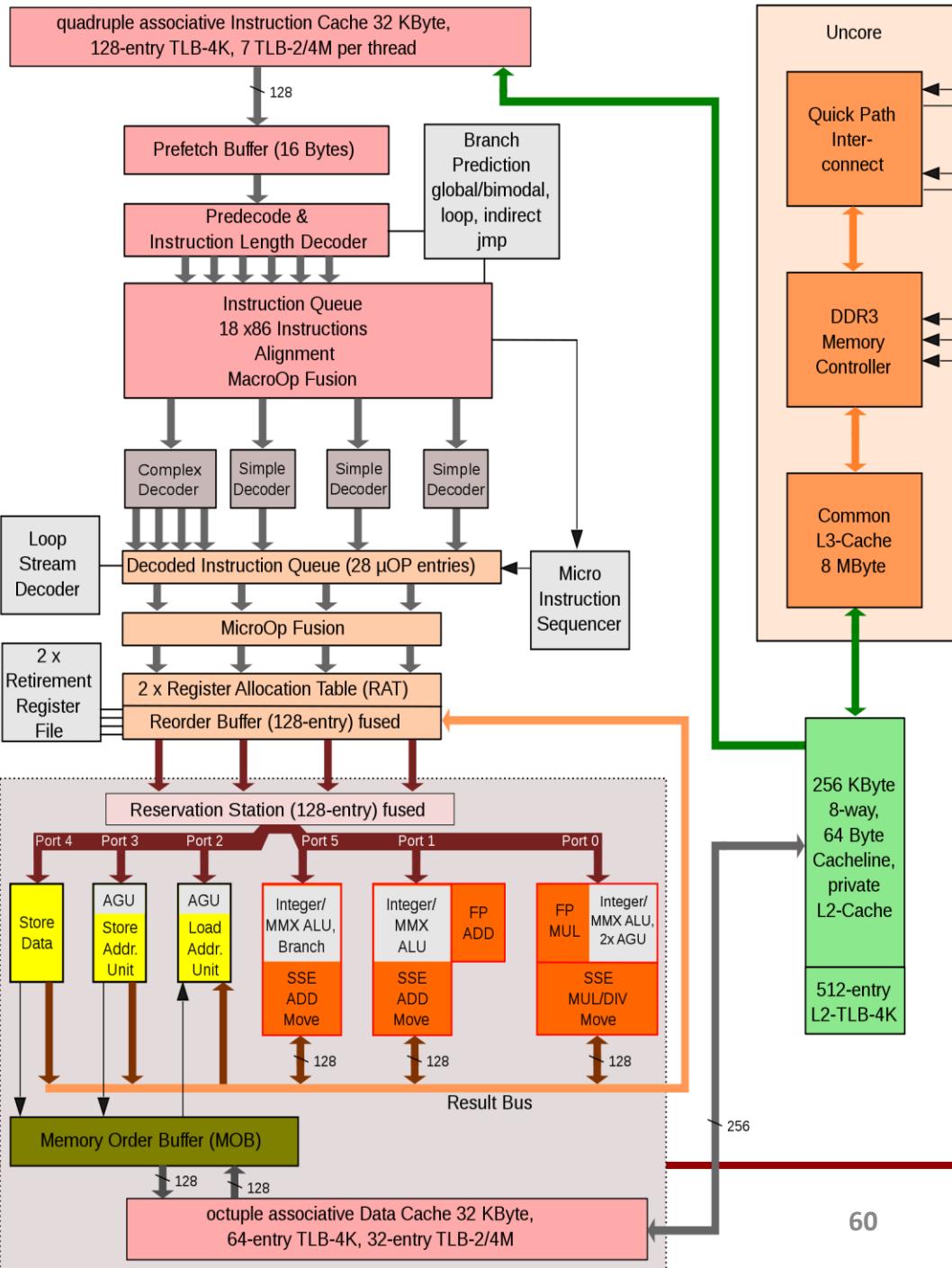
- Keine Kodierung:
 - ein Bit für jedes Kontrollsignal
 - schneller, braucht aber mehr Speicherkapazität
 - Wurde für VAX 780 verwendet - 400KB Mikroprogrammspeicher!
- Viel Kodierung:
 - sende Mikrobefehle durch eine Logik, um Steuersignale zu erhalten
 - Braucht weniger Speicher, ist aber langsamer
- Vertikale versus horizontale Mikro-Befehle
 - Vertikale Mikrobefehle führen einzelne Operationen aus (mein Beispiel)
 - Horizontale Mikrobefehle steuern viele Ressourcen parallel (Textbuch)
- Historischer Kontext des **CISC (Complex Instruction Set Computer)**:
 - Zu viel Logik um die Steuerung + Rest auf einen Chip unterzubringen
 - Nutzten ROM (oder auch RAM) um Mikrobefehle zu speichern
 - Einfach, neue Befehle hinzuzufügen

SRZ	Subtract & Reset to Zero
EXP	Execute Programmer
DIH	Disable Interrupts and Hang
DIG	Disable Gravity
DIE	Disable Everything
CMD	Compare Meaningless Data
WNAM	We Need A Miracle
DOV	Divide and Overflow
WWR	Write Wring Disk
SAI	Skip All Instructions
RPM	Read Programmer's Mind



- Mikroprogrammierung: eine Spezifikationsmethodik
 - Angemessen wenn es hunderte von Opcodes, Modi, Zyklen etc. gibt
 - Signale werden symbolisch durch Mikrobefehle dargestellt
- Spezifikationsvorteile:
 - relativ einfach zu designen und zu schreiben
 - Architektur und Mikro-Befehle können parallel entwickelt werden
- Implementierungsvorteile:
 - Einfach zu ändern, da Werte im Speicher
 - kann andere Architekturen nachbilden
 - kann internen Registern ausnutzen
- Implementierungsnachteile, **langsamer** da jetzt:
 - Steuerung auf dem gleichen Chip wie der Prozessor
 - ROM nicht schneller als RAM
 - Unnötig zurückzugehen und Änderungen vorzunehmen

- 1. Einsatz: Core i7
- 2008
- 45 nm
- Kombiniert festverdrahtete (FSM) Steuerung für einfache Befehle mit mikrocodierter Steuerung für komplexe Befehle (seit 80486)
- Bis zu vier Befehle werden pro Takt übersetzt in Mikro-Operationen
- Komplexe x86 Instruktionen wurden durch einen Mikroprogramm abgewickelt





- Eintaktprozessor ist unwirtschaftlich:
 - Taktperiode muss so lang dauern, wie die längste Instruktion benötigt
 - Komponenten arbeiten jeweils nur einen kleinen Teil der Gesamtzeit
- Mehrzyklenprozessor löst das Problem, indem
 - verschiedene Befehle verschiedene Anzahl Schritte dauern können
 - Einheiten mehrmals pro Befehl verwendet werden können, jedoch in unterschiedlichen Schritten/Taktzyklen
- Um dies zu erreichen, müssen wir
 - zusätzliche interne Register einführen
 - MUXs hinzufügen oder bestehende vergrößern
- Mehrzyklen-Steuerung kann spezifiziert werden durch
 - Endlicher Zustandsautomat (FSM)
 - Mikrocode
- Als nächstes: Steigerung der Leistung mit **Pipelining**

- Warum ist die Mehrzyklen-Implementierung schneller als die Eintakt-Implementierung (gleicher ISA, gleiche Technologie, etc.)?
 - a. Mehrzyklen-Implementierung hat einen niedrigeren CPI
 - b. Mehrzyklen-Implementierung führt weniger Instruktionen aus
 - c. Mehrzyklen-Implementierung ist einfacher
 - d. Mehrzyklen-Implementierung hat eine höhere Taktfrequenz