5. Datenpfad und Steuerung

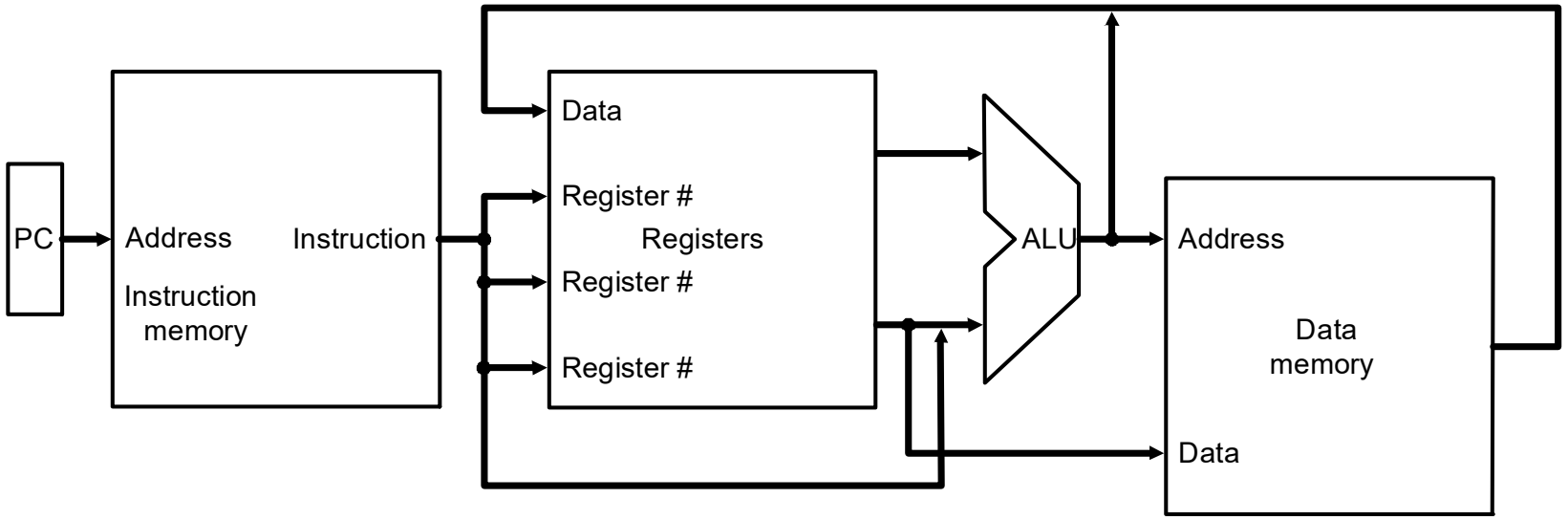
- **Englisch: *datapath and control***
- **Implementierung einer vereinfachten Version von MIPS**
  - Speichertzugriff-Instruktionen
    - `lw, sw`
  - Arithmetisch-logische Instruktionen
    - `add, sub, and, or, nor, slt`
  - Kontroll-Fluss-Instruktionen
    - `beq, j`

Datenpfad und Steuerung (2)

- **generische Implementierung**
  - benutze Befehlszähler (*program counter*, PC), um die Adresse der nächsten Instruktion zu liefern
  - hole Instruktion vom Speicher
  - lese Operanden aus Register
  - benutze Instruktion, um zu entscheiden, was genau zu tun ist
- **alle Instruktionen benutzen eine ALU nach dem Lesen der Register**
  - Adressberechnung für Speicher-Zugriff
  - Arithmetische Operationen
  - Adressberechnung für Kontroll-Fluss (Sprungbefehle)

Implementierungs-Details

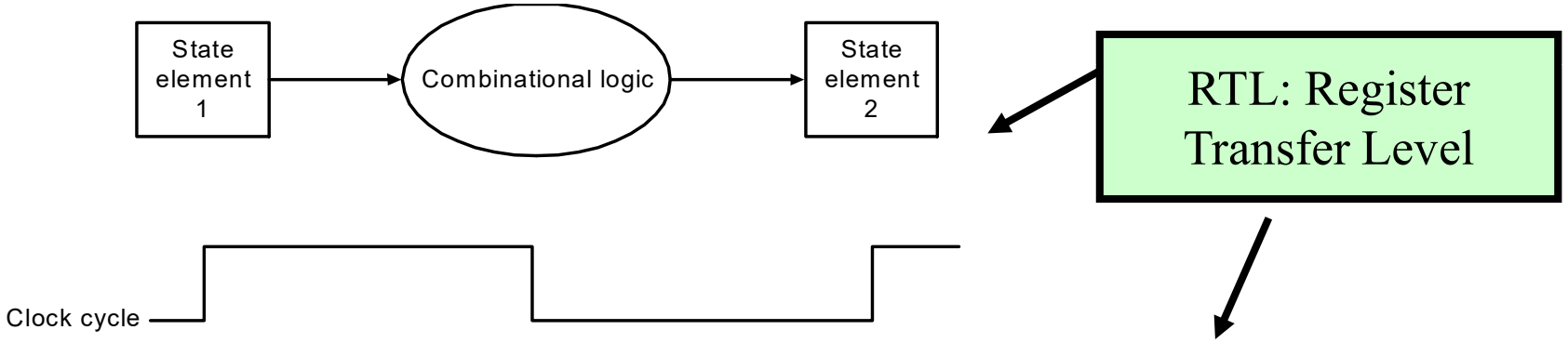
- **abstrakte / vereinfachte Sicht**



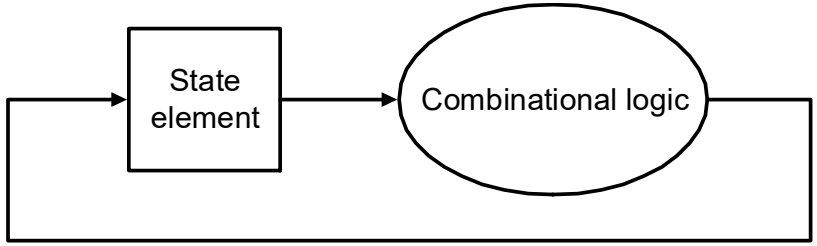
- **zwei Typen von funktionalen Einheiten**
  - Elemente, die auf Daten operieren (z.B. ALU)
    - kombinatorisch: Schaltnetz
  - Elemente, die einen Zustand haben (z.B. Register)
    - getaktet: Schaltwerk bzw. Flip-Flops
    - bei uns flankengetriggert mit der positiven Taktflanke

Unsere Implementierung

- **typische Ausführung**
  - lese Inhalt eines oder mehrerer Zustands-Elemente (*State elements*: Flip-Flops, Register, Speicher)
  - sende Werte durch kombinatorische Logik (*Combinatorial logic*, Schaltnetz)
  - schreibe Resultate in ein oder mehrere Zustands-Elemente



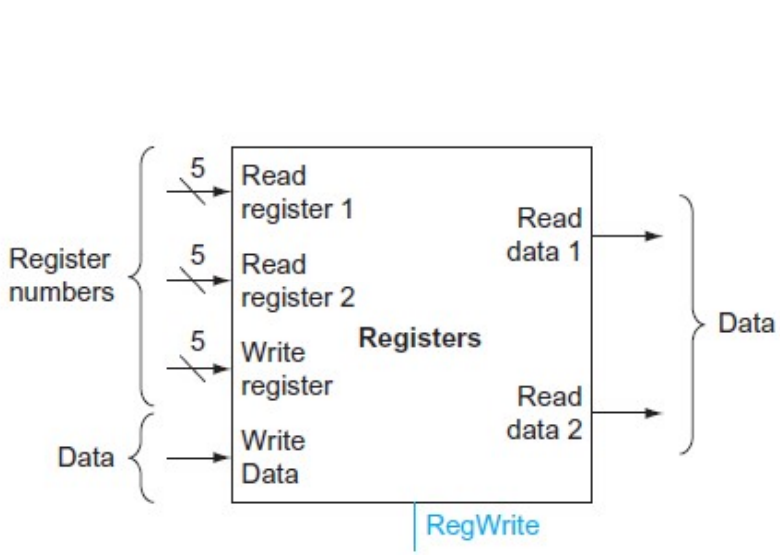
Das funktioniert auch, wenn die beiden Zustands-Elemente identisch sind:

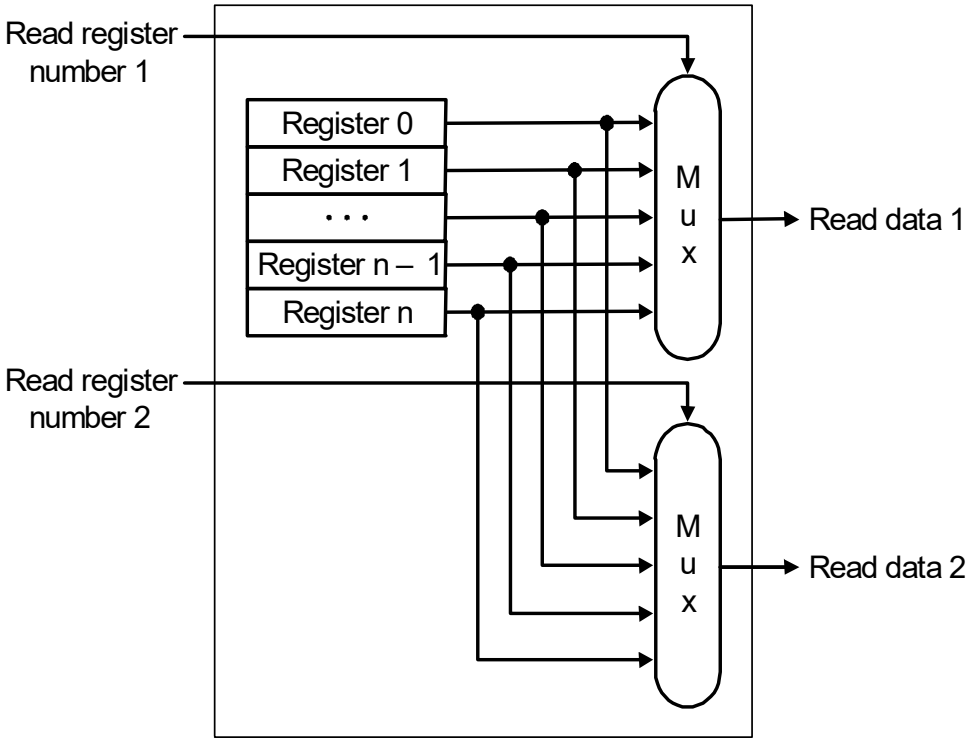


Register File

- **mit D-Flip-Flops gebaut**

Lesen: kombinatorisch
(Takt nicht notwendig)



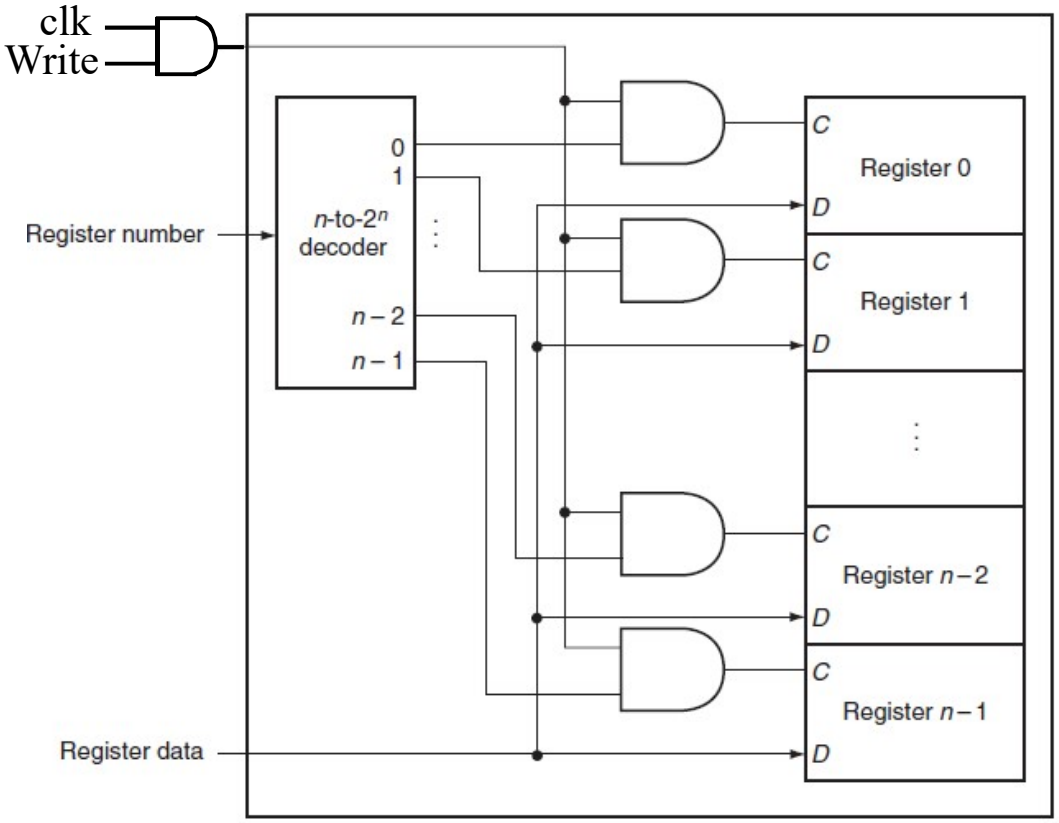


zwei Leseports

Register File (2)

- beachte: es wird ein Taktsignal benötigt, um den Zeitpunkt des Schreibens festzulegen
  - Write ist gleichzeitig Taktsignal, das nur beim Schreiben aktiv wird
  - mit der steigenden Flanke von Write werden die Daten abgespeichert
  - besser: eigenes clk-Eingangssignal
  - noch besser: Latch (siehe Vorlesung "Technische Informatik")

Schreiben: getaktet

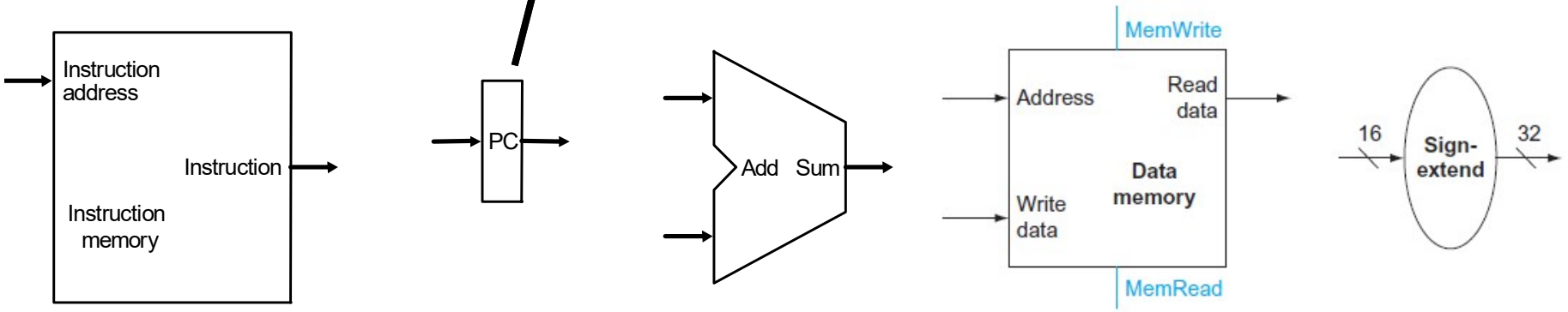


ein Schreibport

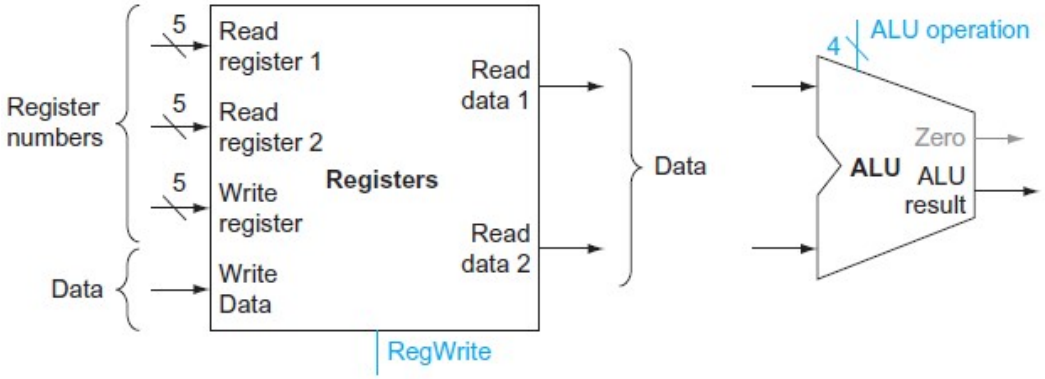
Konstruktion des Datenpfades

- **funktionale Einheiten**

Einfaches Register, kann z.B. nicht selbst eine 4 addieren

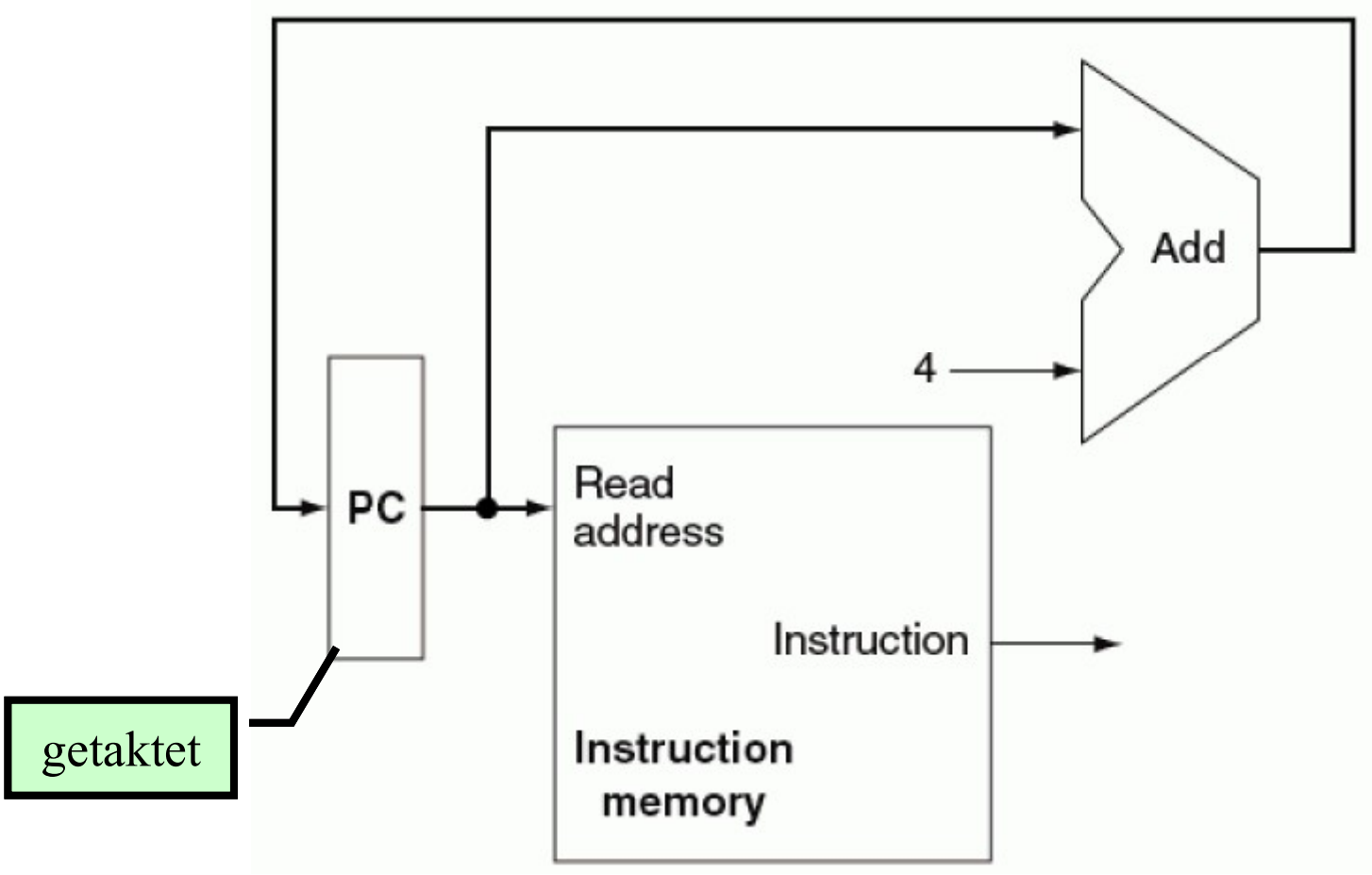


a. Instruction memory b. Program counter c. Adder

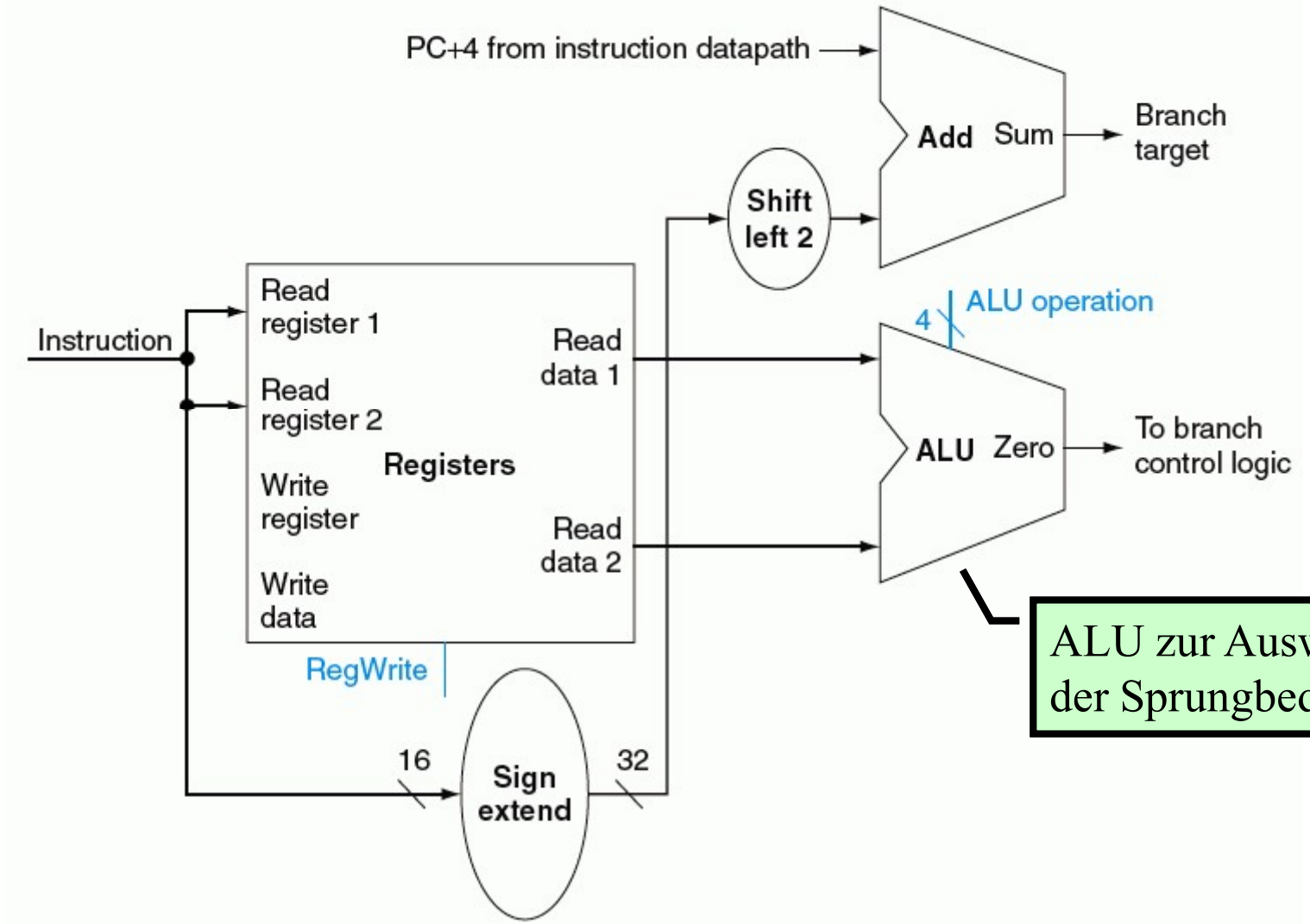


Verbinde die Einheiten zu einem Datenpfad, benutze Multiplexer, wenn verschiedene Pfade benutzt werden sollen

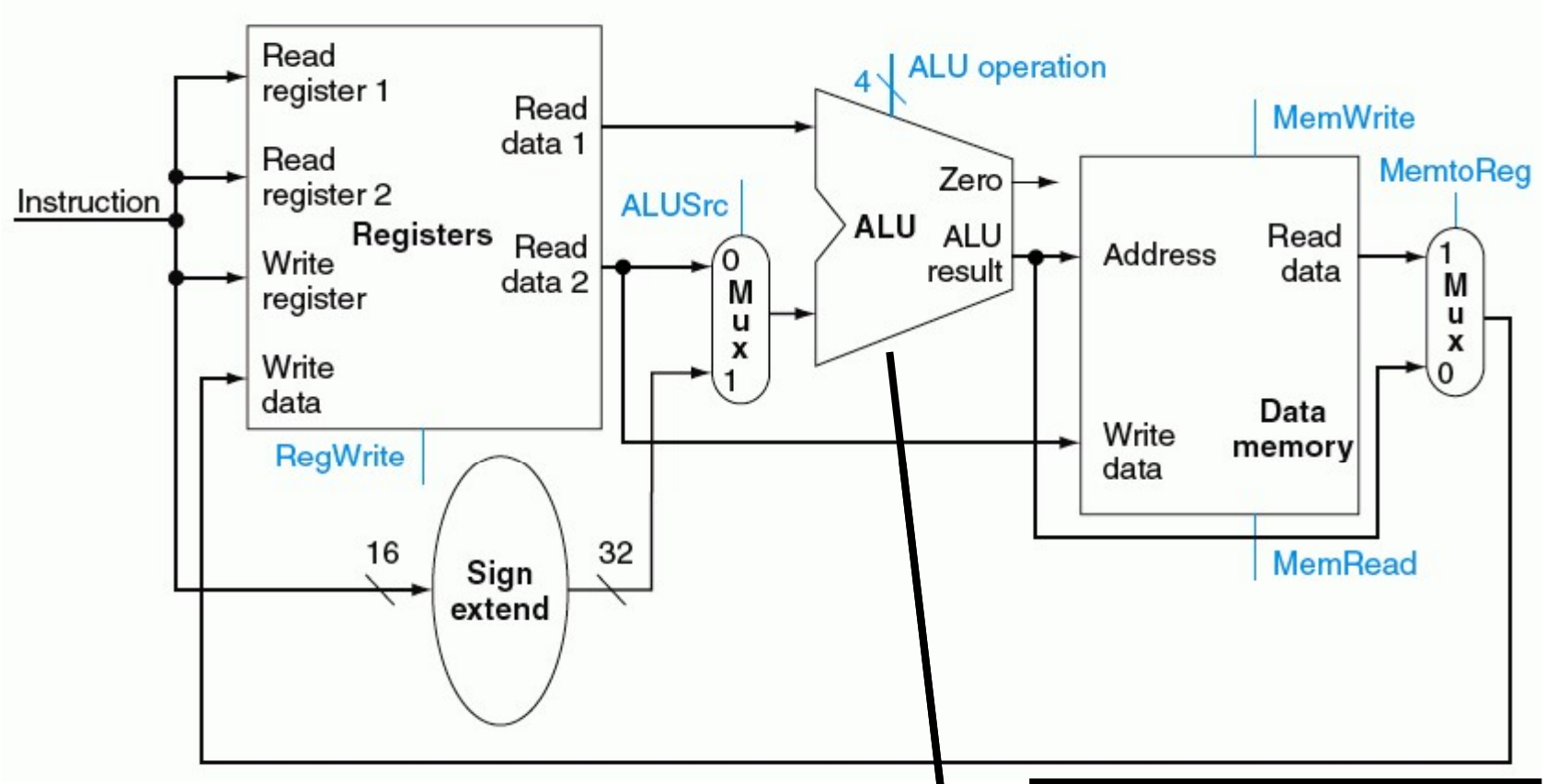
Instruktion holen und PC inkrementieren



Branches

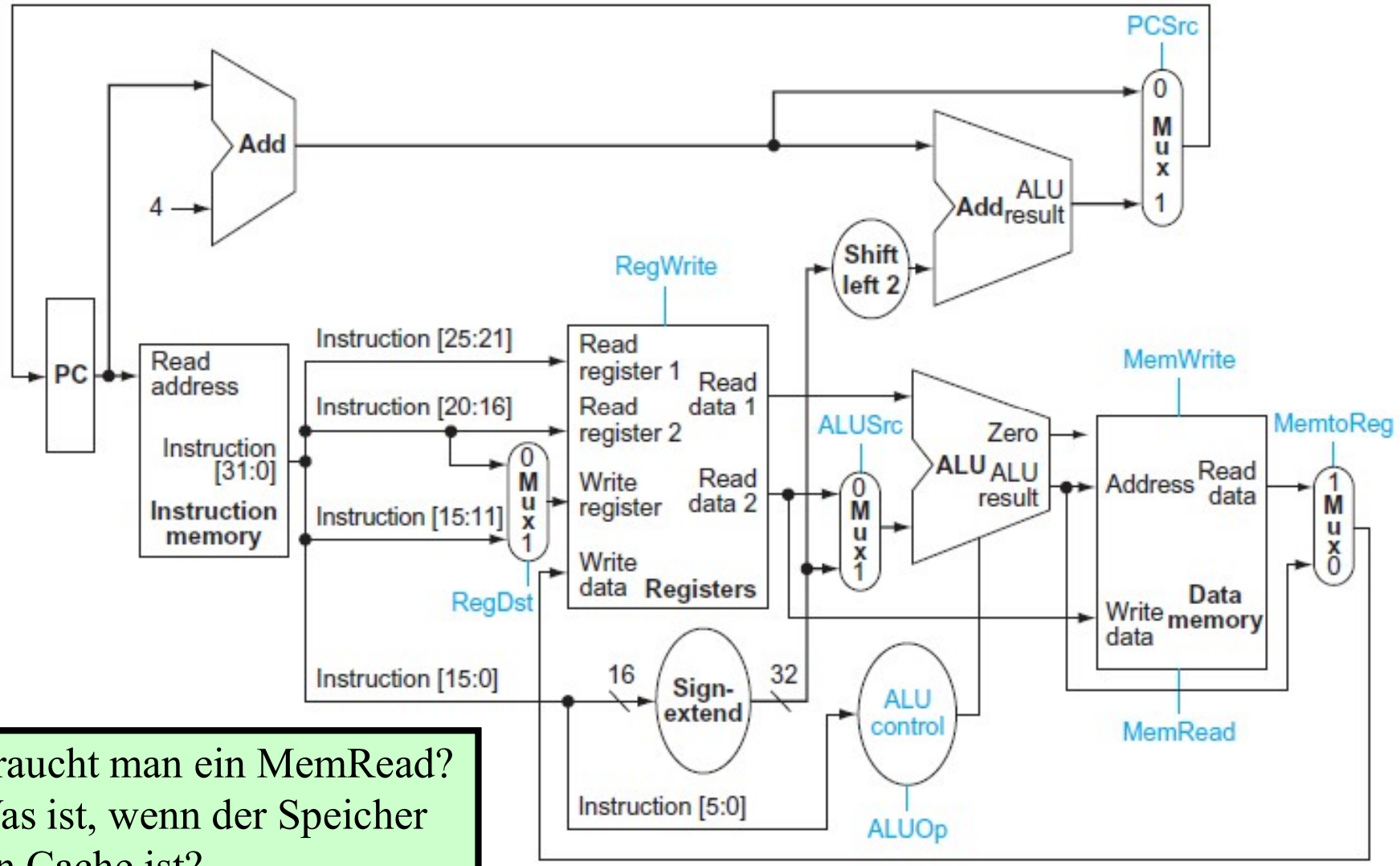

PC+4 from instruction datapath
Add
Sum
Branch target
Shift left 2
ALU operation
4
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
To branch control logic
RegWrite
16
Sign extend
32
ALU zur Auswertung der Sprungbedingung

Datenpfad für Speicher und arithm. Instruktionen



ALU für arithmetische Instruktionen und Adressberechnung

Vollständiger Datenpfad (aber noch ohne j)

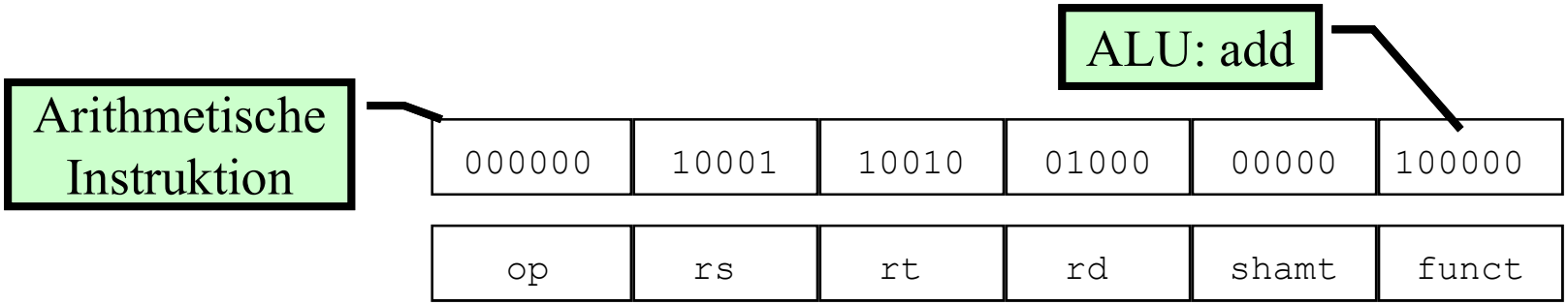


Braucht man ein MemRead?
Was ist, wenn der Speicher
ein Cache ist?

Steuerung

- **Was muss alles gesteuert werden?**
  - Funktionale Einheiten (ALU, Memory read/write, etc.)
  - Steuerung des Datenflusses (Multiplexer)
- **Information kommt von den 32 Bits des Instruktionswortes**
- **Beispiel**

add $8, $17, $18



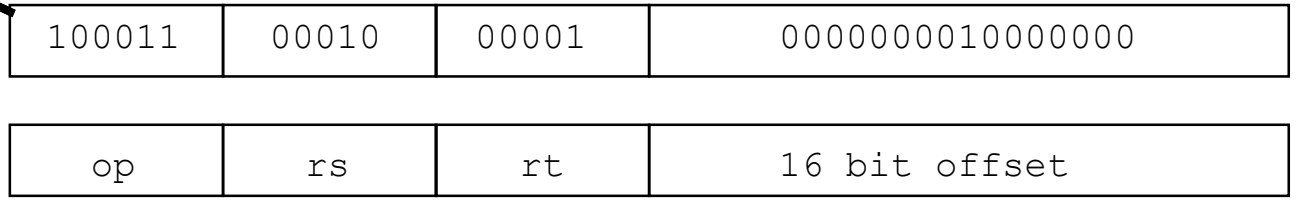
Steuerung (2)

- ALU-Operation hängt vom Instruktionstyp und vom Funktionscode ab.
- ALU wird auch manchmal benötigt, wenn keine arithmetische Instruktion durchgeführt wird.

- **Beispiel**

lw $1, 128($2)

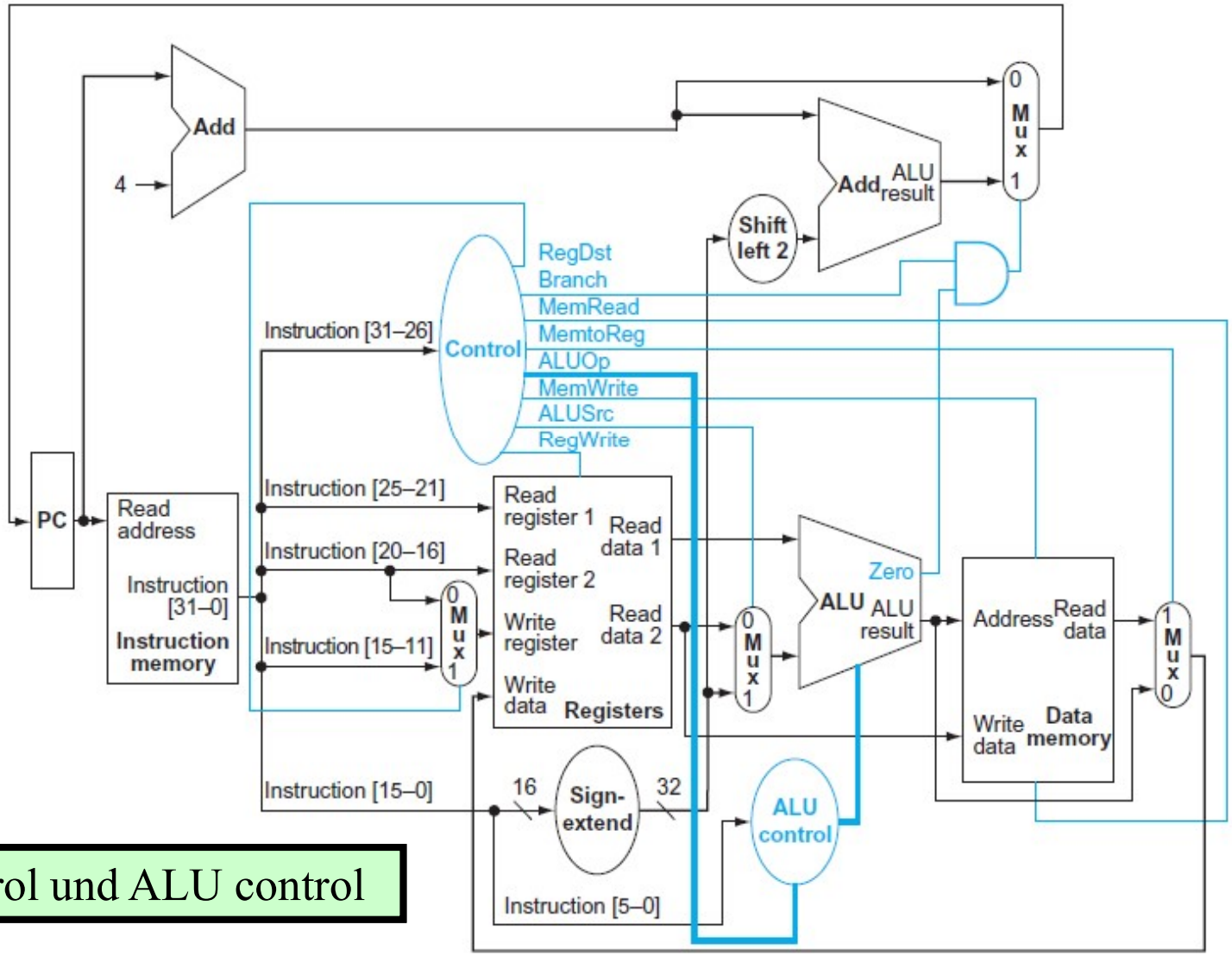
Load word
ALU: add



- ALU Steuereingänge

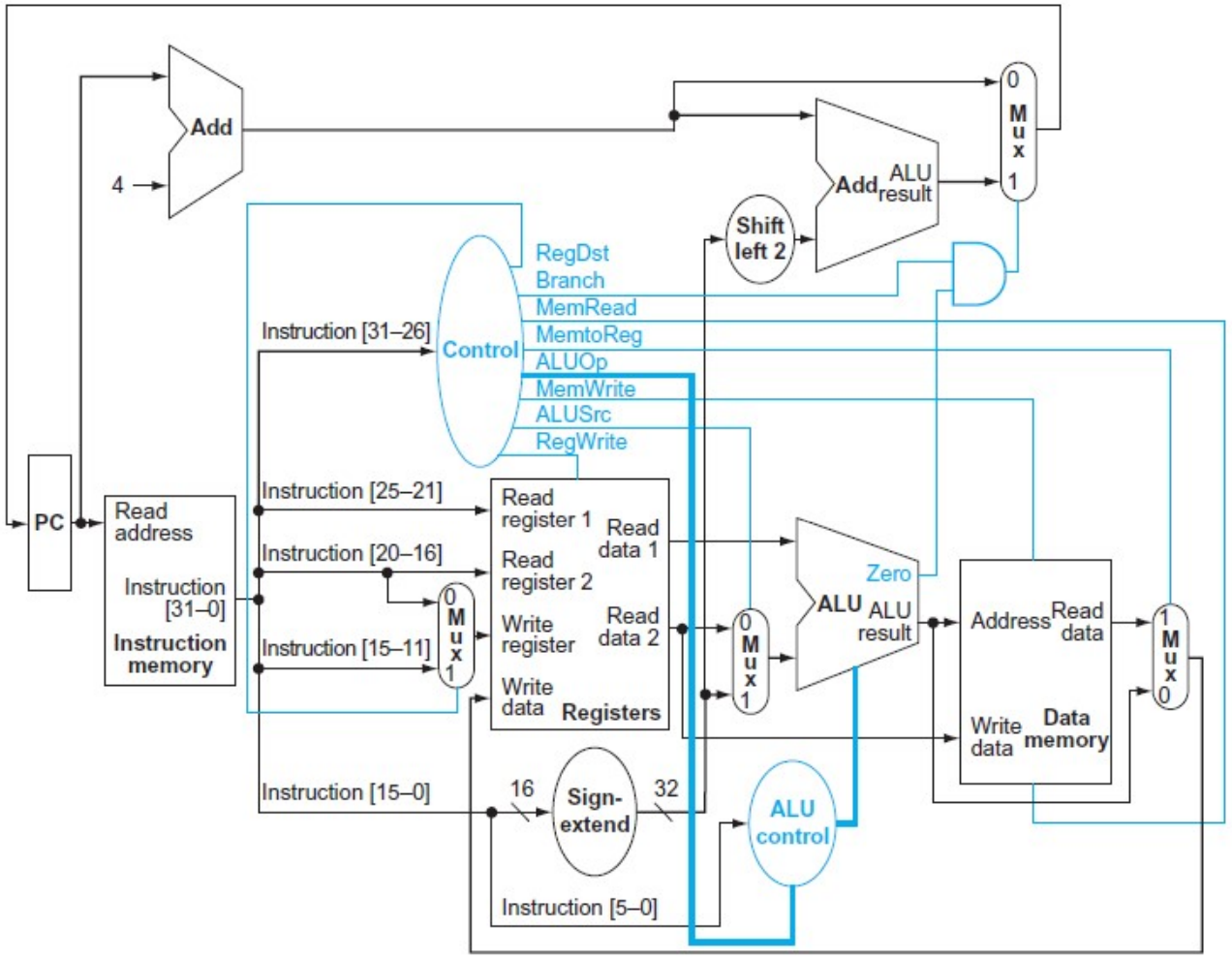
```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = nor
```

Aufteilung der Steuerung



Control und ALU control

Steuerung (5)



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Steuerung (Control)

- **Wertetabelle**
  - also ein einfaches Schaltnetz

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Steuerung (ALU Control)

- gegebener Instruktionstyp (X = don‘t care)
  - 00 = lw, sw
  - 01 = beq
  - 10 = arithmetic

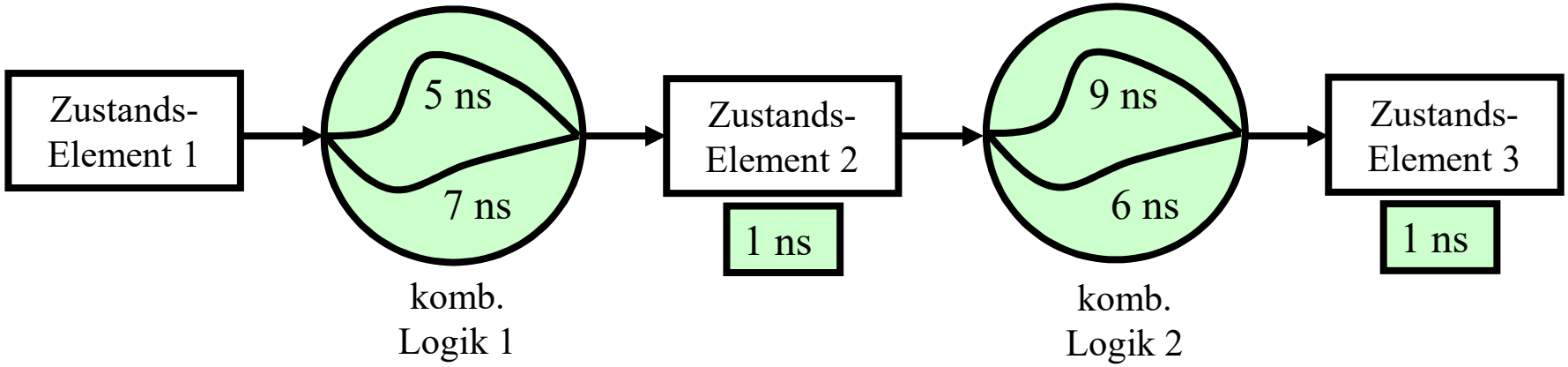
  } **ALUOp**
  nur vom Opcode abhängig
- Funktionscode für Arithmetik
- benutze Wahrheitstabelle zur Festlegung der ALU-Steuersignale
  - Also wieder einfaches Schaltnetz

	ALUOp		Funct field						Operation
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw, sw	0	0	X	X	X	X	X	X	0010
beq	0	1	X	X	X	X	X	X	0110
add	1	X	1	0	0	0	0	0	0010
sub	1	X	1	0	0	0	1	0	0110
and	1	X	1	0	0	1	0	0	0000
or	1	X	1	0	0	1	0	1	0001
slt	1	X	1	0	1	0	1	0	0111
nor	1	X	1	0	0	1	1	1	1111

ALU Operation:
0000 and
0001 or
0010 add
0110 subtract
0111 set-on-less-than
1111 nor

Struktur unserer einfachen Steuerung

- komplette Logik ist kombinatorisch
- das Abklingen von Hazards muss abgewartet werden
  - Schreib- und Takt-Signale werden benutzt, um zu entscheiden, wann geschrieben werden kann
- minimale Zykluszeit wird durch den längsten Pfad zwischen zwei Zustands-Elementen bestimmt + Verzögerungszeit des Zustandselementes selbst (*Setup Time*)



$\Rightarrow$ Zykluszeit minimal 10 ns
oder Taktfrequenz maximal 100 MHz

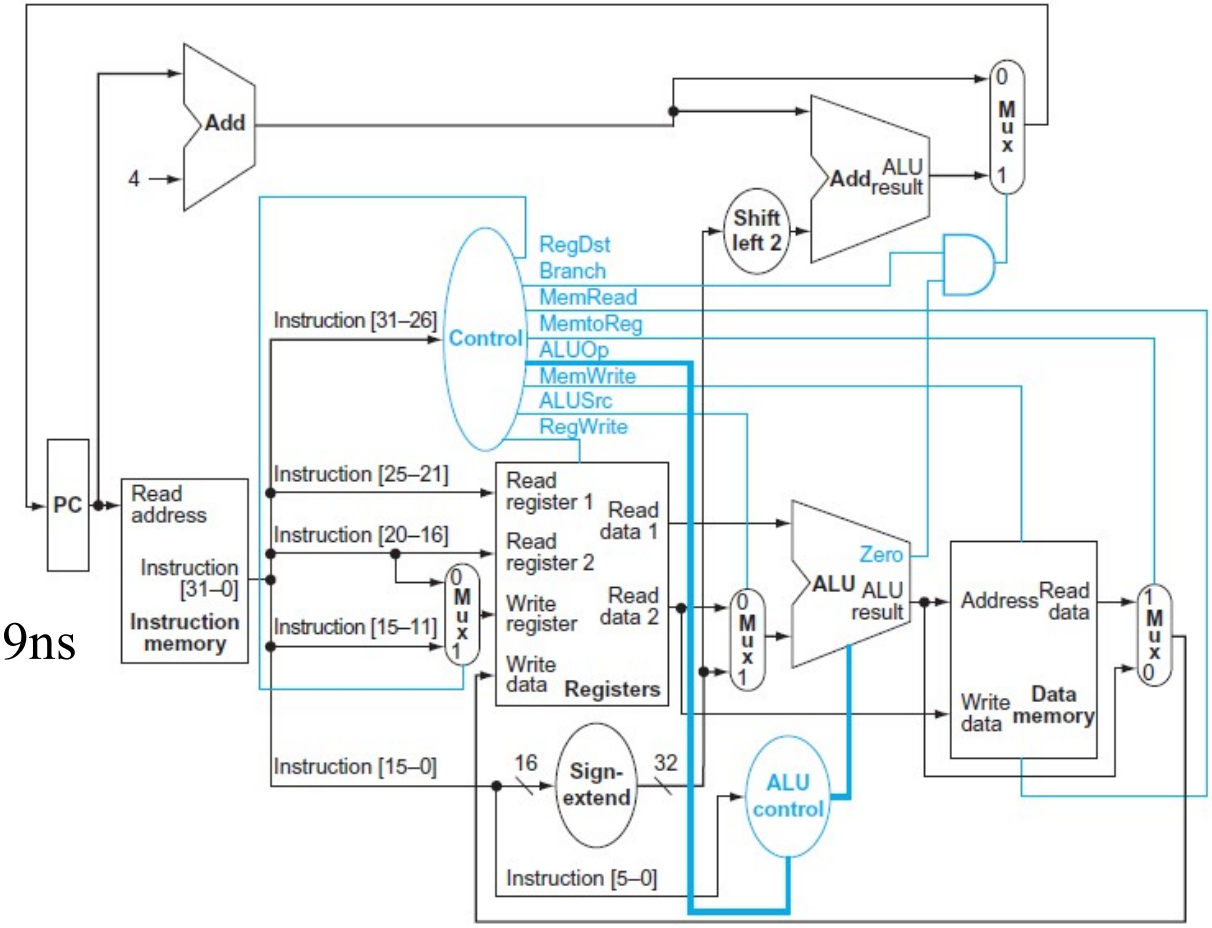
Ein-Zyklus Implementierung

Berechne Zykluszeit (nur Beispiel)

PC-Zugriff: 1ns
Speicher: 2ns
ALU und Addierer: 2ns
Registerzugriff: 1ns
Setup-Zeit Register: 1ns

Rest: vernachlässigbar

⇒ minimale Zykluszeit: 9ns



Wie geht es weiter?

- **Probleme der Ein-Zyklus Implementierung**
  - Was machen wir bei komplexeren Instruktionen, z.B. Gleitpunktoperationen?
    - würde sehr lange Zykluszeit erfordern
    - Zykluszeit abhängig von Instruktion?
      - würde zwar etwas bringen, ist aber technisch viel zu aufwendig
  - Funktionale Einheiten sind mehrfach vorhanden.
    - Kann man eine ALU nicht noch für andere Zwecke (Adressberechnungen) nutzen?
- **Mögliche Lösung**
  - Benutze "kleinere" Schritte (jeder Schritt benötigt einen Zyklus).
  - Verschiedene Instruktionen können unterschiedliche Anzahl von Schritten benötigen
  - “Multizyklus”-Datenpfad

Multizyklus Datenpfad

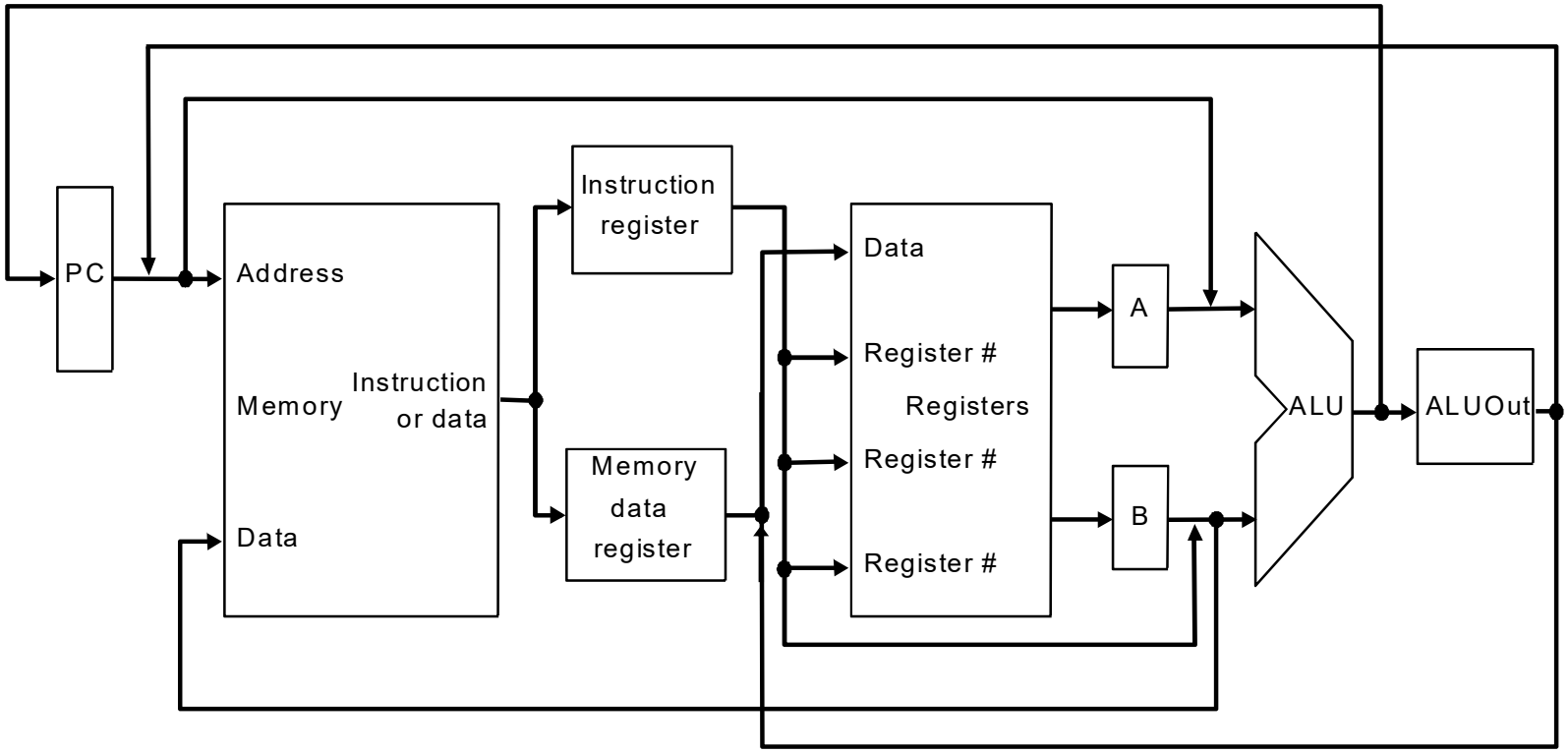
- **Funktionale Einheiten werden mehrfach genutzt**
  - ALU berechnet Adressen und inkrementiert PC
  - Speicher enthält Instruktionen und Daten
- **Steuersignale werden nicht nur durch die Instruktion festgelegt**
  - Was macht die ALU alles bei einer *subtract* Instruktion?
  - ALU benötigt in verschiedenen Phasen der Verarbeitung verschiedene Steuersignale
- **Benutze Schaltwerk (*finite state machine*, FSM) für die Steuerung**

Multizyklus Datenpfad (2)

- **Unterteile die Instruktionen in Schritte, jeder Schritt benötigt einen Takt**
  - die Menge an Arbeit sollte in jedem Schritt in etwa gleich sein (Zykluszeiten sind konstant)
  - jeder Schritt sollte nur ***eine*** große funktionale Einheit benutzen
- **am Ende jedes Zyklus**
  - speichere Werte, die im nächsten Zyklus benötigt werden
  - dazu müssen zusätzliche "interne" Register nach jeder großen funktionalen Einheit eingebaut werden

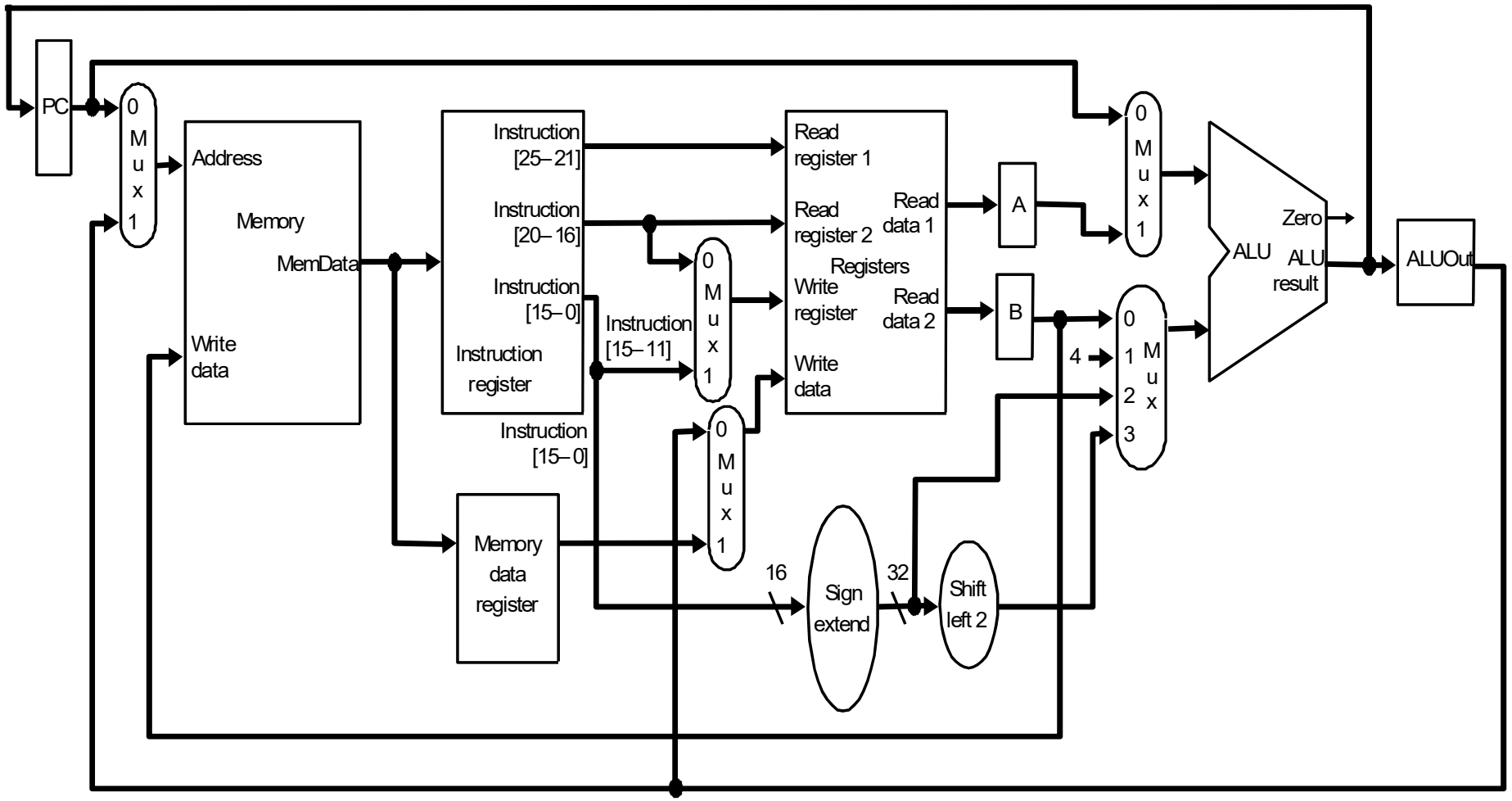
Multizyklus Datenpfad (3)

- **Überblick (ohne Details wie Multiplexer)**



Multizyklus Datenpfad (4)

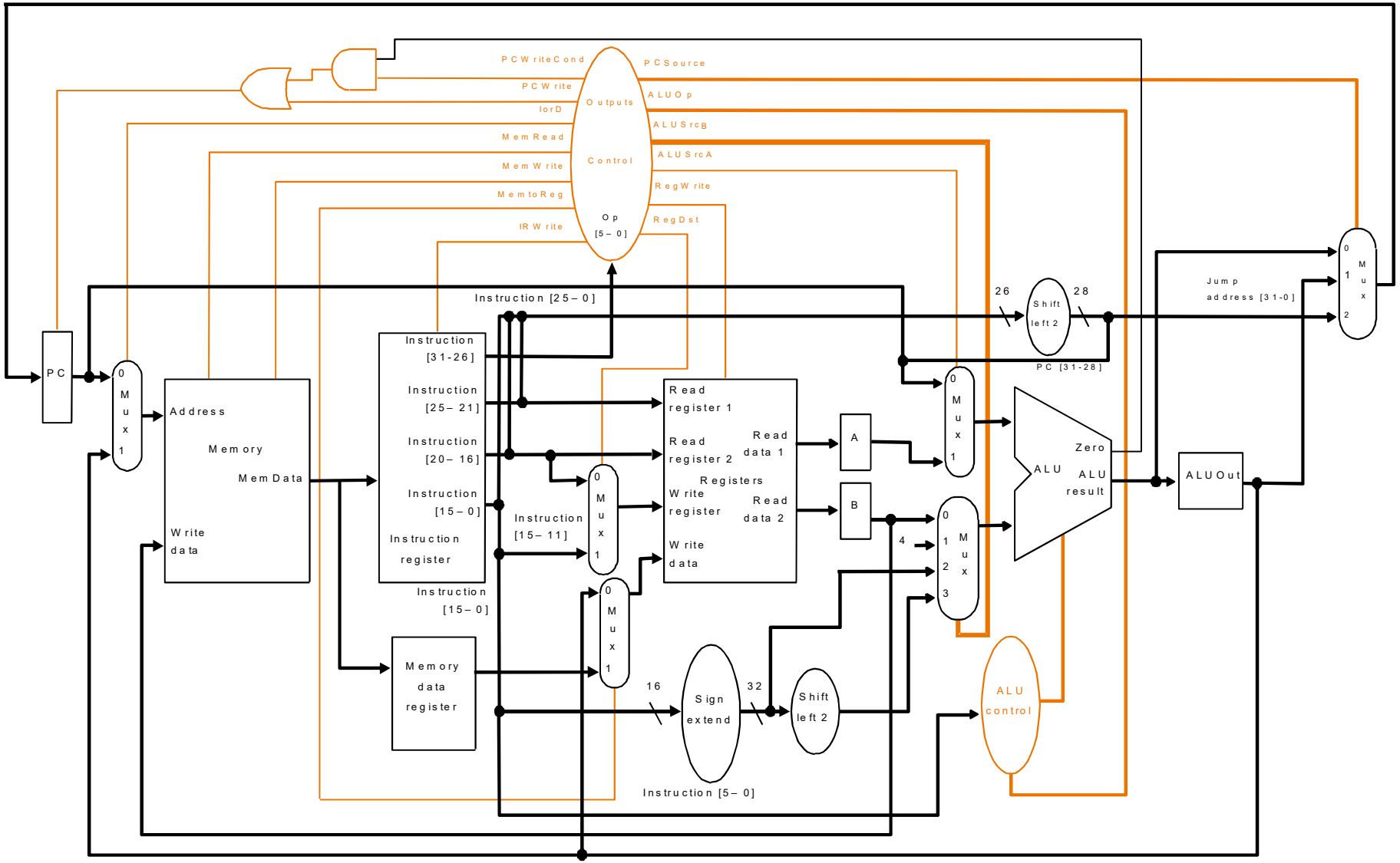
JKA4



JKA4 Für beq fehlt noch ein Multiplexer, damit ALUOut zum PC zurückgeführt werden kann

Prof. Dr. Joachim K. Anlauf; 09.07.2002

Datenpfad mit beq und j und Steuerung



Fünf Ausführungsschritte

1. **Instruktion holen**
2. **Instruktion dekodieren und Register holen**
3. **Instruktions-Ausführung, Speicher-Adressberechnung oder Branch-Fertigstellung**
4. **Speicherzugriff oder R-Typ Instruktions-Fertigstellung**
5. **LW-Fertigstellung**

Instruktionen benötigen 3 - 5 Zyklen!

Schritt 1

- **Instruktion holen**
  - Benutze PC, um die Instruktion zu holen und schreibe sie in das Instruction Register IR.
  - Inkrementiere den PC um 4 und schreibe das Ergebnis zurück in den PC.
  - Kann präzise mithilfe der RTL (*Register-Transfer Language*) beschrieben werden:

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

    - Was ist der Vorteil, den PC jetzt schon zu inkrementieren?

Schritt 2

- **Instruktion dekodieren und Register holen**
  - Hole Lese-Register `rs` und `rt` für den Fall, dass wir sie brauchen.
  - Berechne die Sprungzieladresse für den Fall, dass die Instruktion ein *branch* ist.
  - RTL

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

  - Wir setzen noch keine Steuerleitungen in Abhängigkeit vom Instruktionstyp (wir sind noch damit beschäftigt, die Instruktion zu "dekodieren").

Schritt 3

- **abhängig von der Instruktion**
  - ALU führt eine von drei Funktionen aus, abhängig vom Instruktionstyp
- **Memory Reference**

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

- **R-type**

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

- **Branch (Fertigstellung)**

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

Schritt 4

- **R-Typ oder Speicherzugriff**
  - Loads und stores greifen auf den Datenspeicher zu

    ```
    MDR = Memory[ALUOut];
    ```

    oder

    ```
    Memory[ALUOut] = B;
    ```

  - R-Typ Instruktionen (Fertigstellung)

    ```
    Reg[IR[15-11]] = ALUOut;
    ```

Das Schreiben findet am Ende des Zyklus mit der aktiven Flanke statt.

Schritt 5

- **LW (Fertigstellung)**

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

Was ist mit den anderen Instruktionen?

Zusammenfassung

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC]			
	PC = PC + 4			
Instruction	A = Reg [IR[25-21]]			
decode/register fetch	B = Reg [IR[20-16]]			
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address	ALUOut = A op B	ALUOut = A + sign-extend	if (A ==B) then	PC = PC [31-28] II
computation, branch/		(IR[15-0])	PC = ALUOut	(IR[25-0]<<2)
jump completion				
Memory access or R-type	Reg [IR[15-11]] =	Load: MDR = Memory[ALUOut]		
completion	ALUOut	or		
		Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

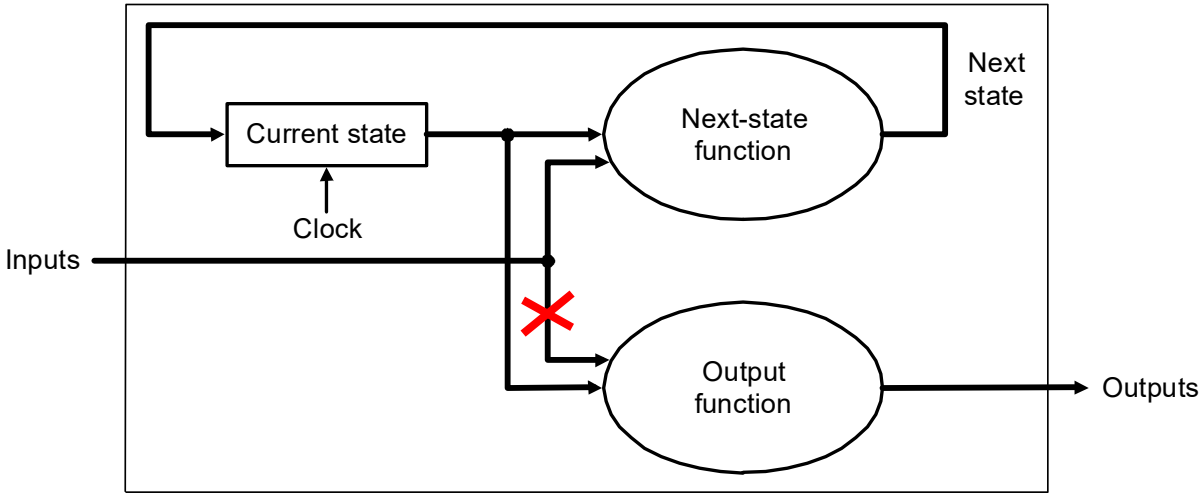
Implementierung der Steuerung

- **Werte der Steuersignale hängen davon ab,**
  - welcher Schritt in welcher Instruktion gerade ausgeführt wird
- **Benutze die Informationen, die wir gesammelt haben, um ein Schaltwerk zu spezifizieren**
  - Zustandsübergangsdiagramm (grafische Darstellung einer FSM)
  - Mikroprogrammierung
- **Implementierung kann dann von der Spezifikation abgeleitet werden**

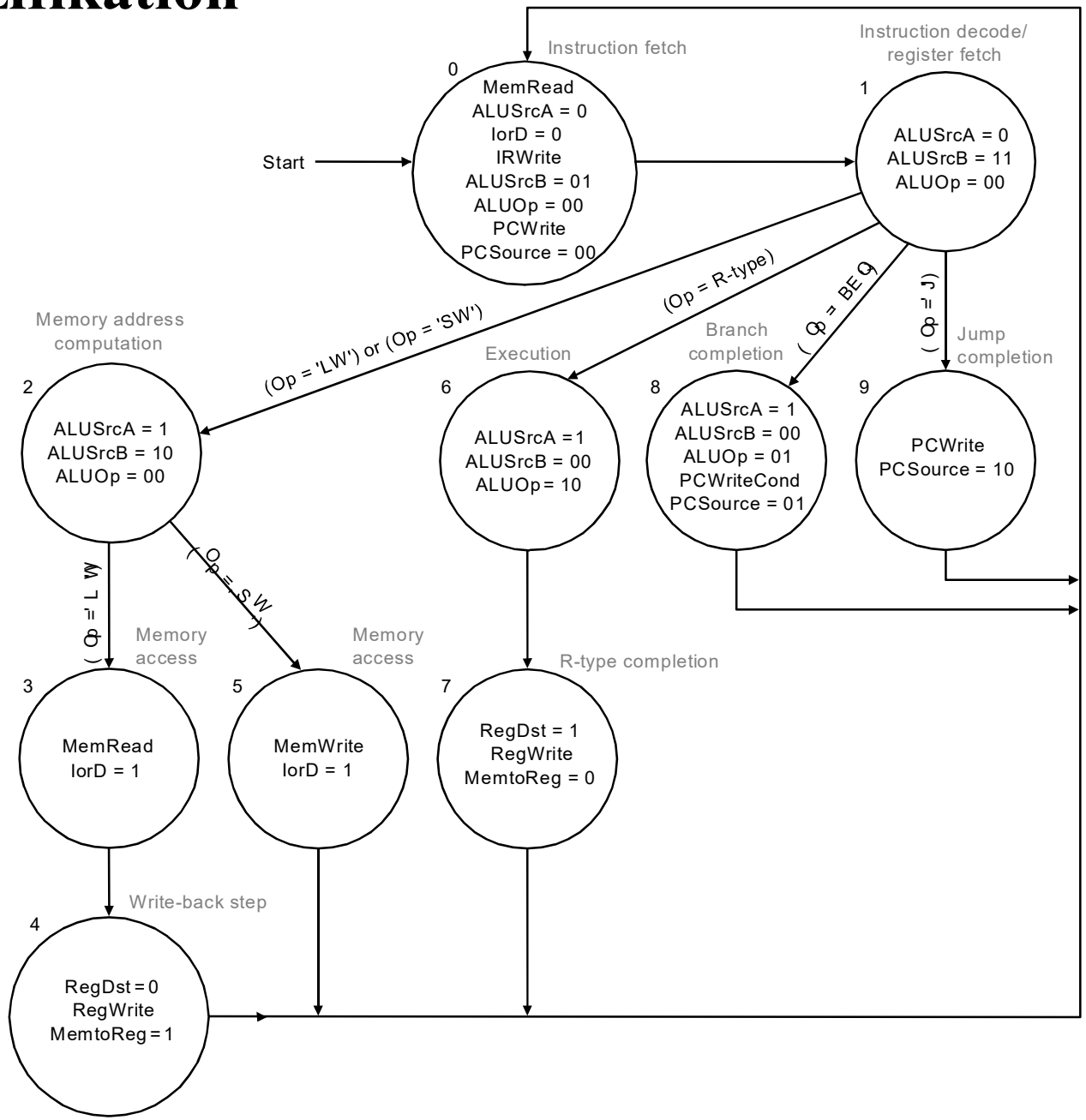
Wiederholung: Schaltwerke

- Satz von Zuständen
- Übergangsfunktion, die den nächsten Zustand berechnet (abhängig vom momentanen Zustand und den Eingängen)
- Ausgabefunktion (abhängig vom momentanen Zustand und möglicherweise den Eingängen)

Wir benutzen ein Moore Schaltwerk:
Ausgabe nur abhängig
vom Zustand



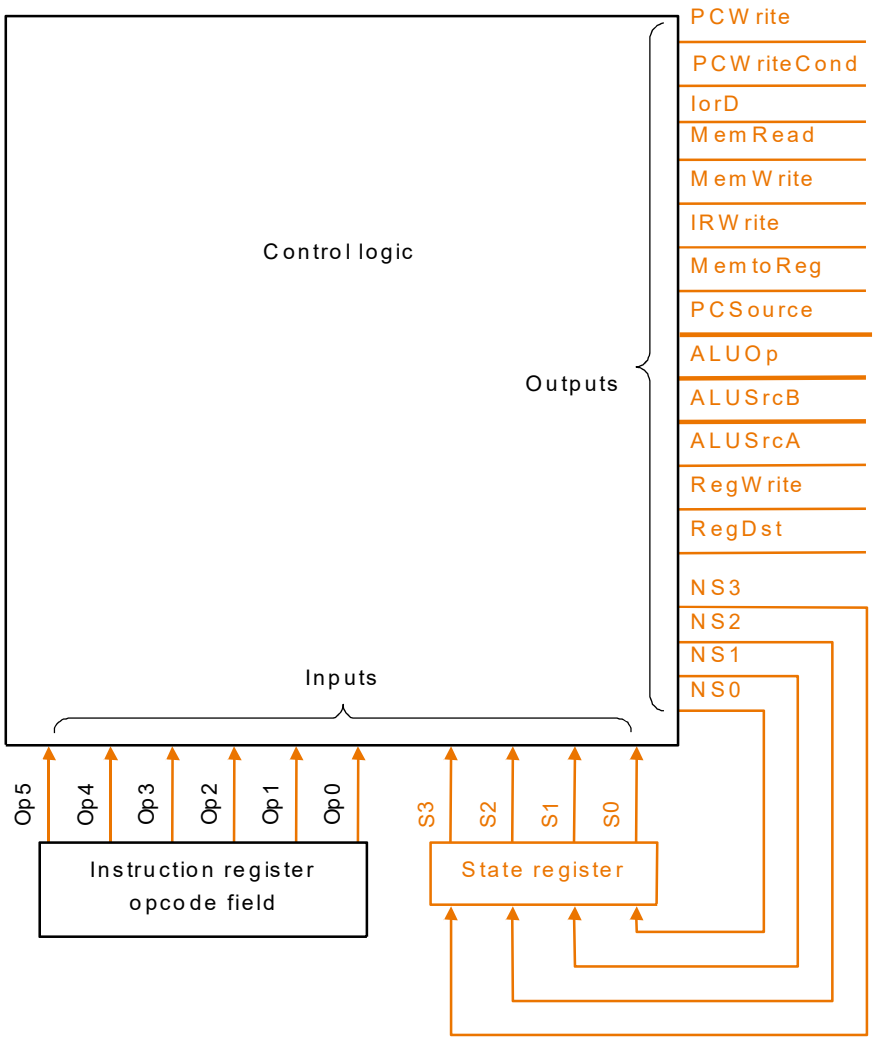
Grafische Spezifikation einer FSM



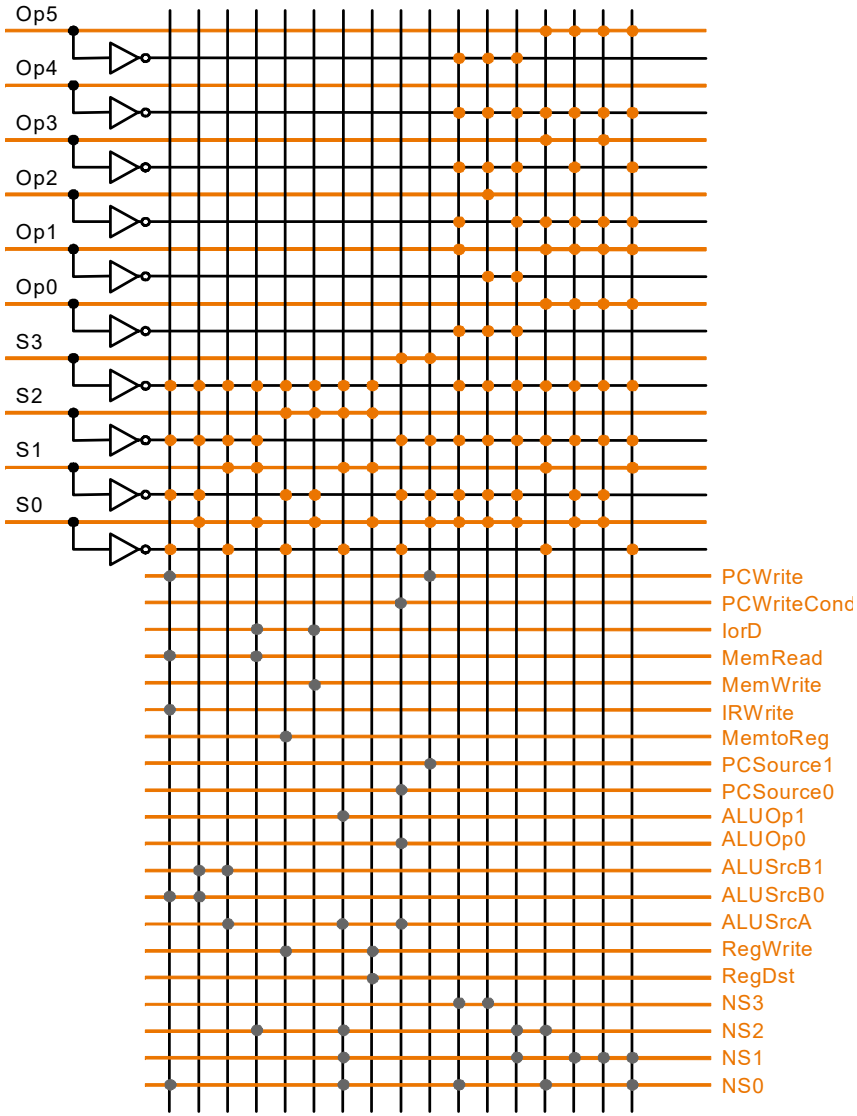
Wie viele Zustandsbits werden benötigt?

Finite State Machine für Steuerung

- **Implementierung**

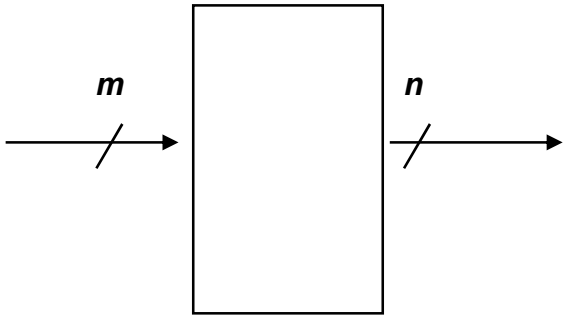


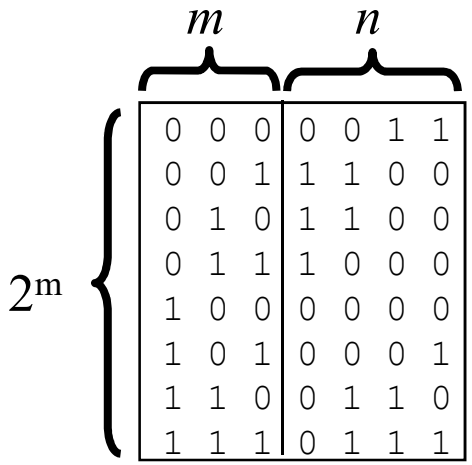
PLA Implementierung



ROM Implementierung

– Ein ROM kann eine Wahrheitstabelle implementieren.
  - mit meiner m-Bit Adresse können 2^m Einträge angesteuert werden
  - die Ausgaben sind die Bits der Daten, auf die die Adresse zeigt





ROM Implementierung (2)

- Wie viele Eingänge benötigen wir?
  - 6 Bits für Opcode, 4 Bits für Zustand = 10 Adressleitungen
  - (d.h. 2^{10} = 1024 verschiedene Adressen)
- Wie viele Ausgänge benötigen wir?
  - 16 Datenpfad-Steuerleitungen, 4 Zustandsbits = 20 Ausgänge
- ROM hat 2^{10} x 20 = 20 kbit (und eine sehr unübliche Größe)
- sehr verschwenderisch, da für sehr viele Eingangskombinationen, die Ausgänge identisch sind
  - Steuersignale hängen nur vom Zustand ab
  - Opcode wird ignoriert
  - wird nur zur Wahl des nächsten Zustandes benutzt

ROM Implementierung (3)

- **Aufteilen der Wahrheitstabelle in zwei Teile**
  - 4 Zustandsbits erzeugen die 16 Ausgänge (Steuersignale)

    ROM mit 2^4 x 16 Bits
  - 10 Bits erzeugen die nächsten 4 Zustandsbits

    ROM mit 2^{10} x 4 Bits
  - Insgesamt: ROM mit 4.3 kbit

ROM vs. PLA

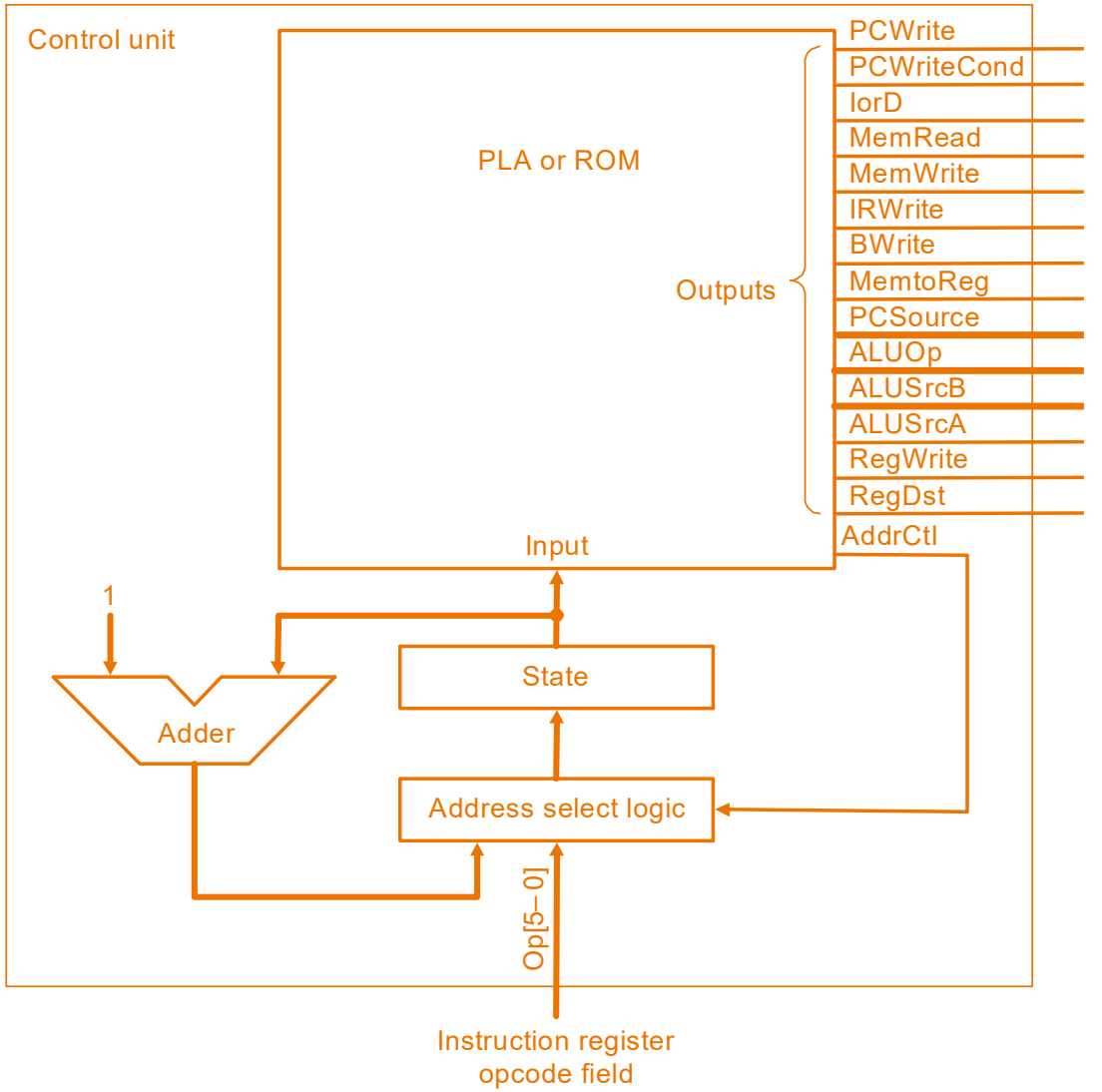
- ROM
  - enthält alle Minterme
  - realisiert vollständige Wertetabelle
- PLA ist viel kleiner
  - implementiert minimierte DNF
  - enthält nur Produktterme, die eine 1 am Ausgang erzeugen
- Größe ist

  (#Eingänge * #Produktterme) + (#Ausgänge * #Produktterme)
- in unserem Beispiel

  (2 * 10 * 17) + (20 * 17) = 680 PLA Zellen
- PLA Zelle hat gewöhnlich die Größe einer ROM Zelle (vielleicht etwas größer)

Alternativer Implementierungs-Stil

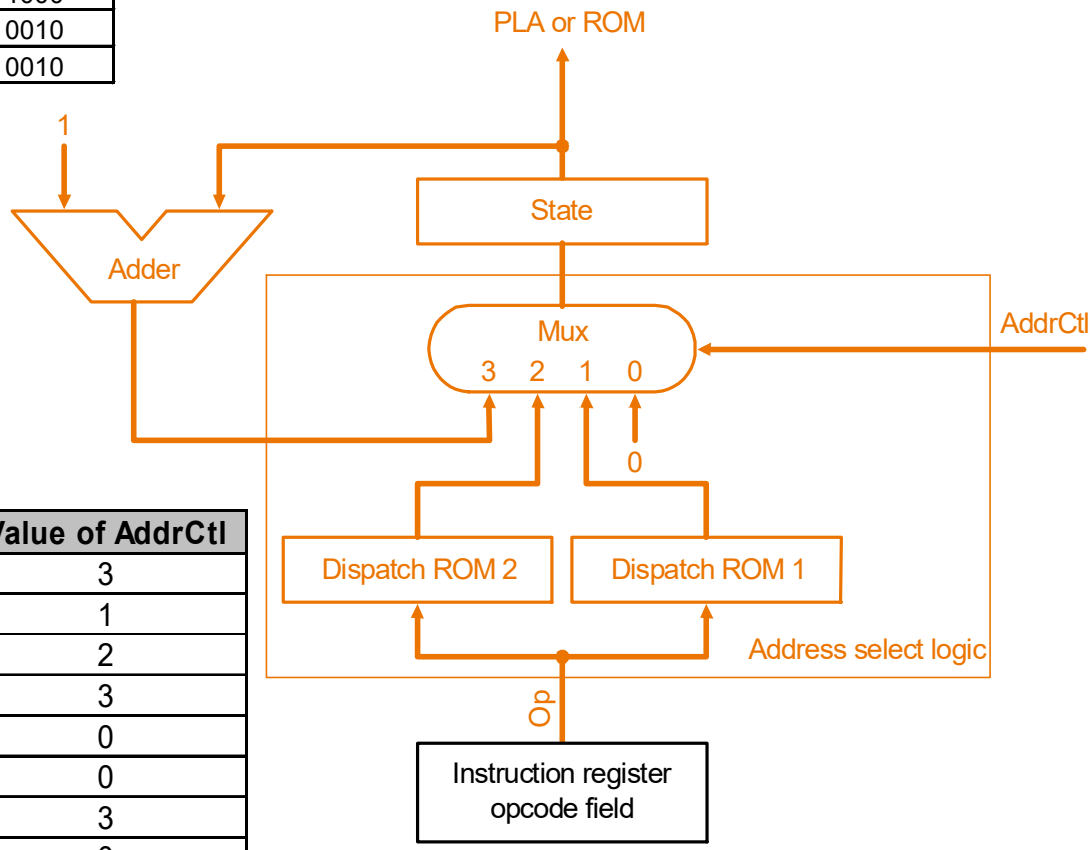
- **Komplexe Instruktionen**
  - der "nächste Zustand" ist häufig der "momentane Zustand + 1"



Details

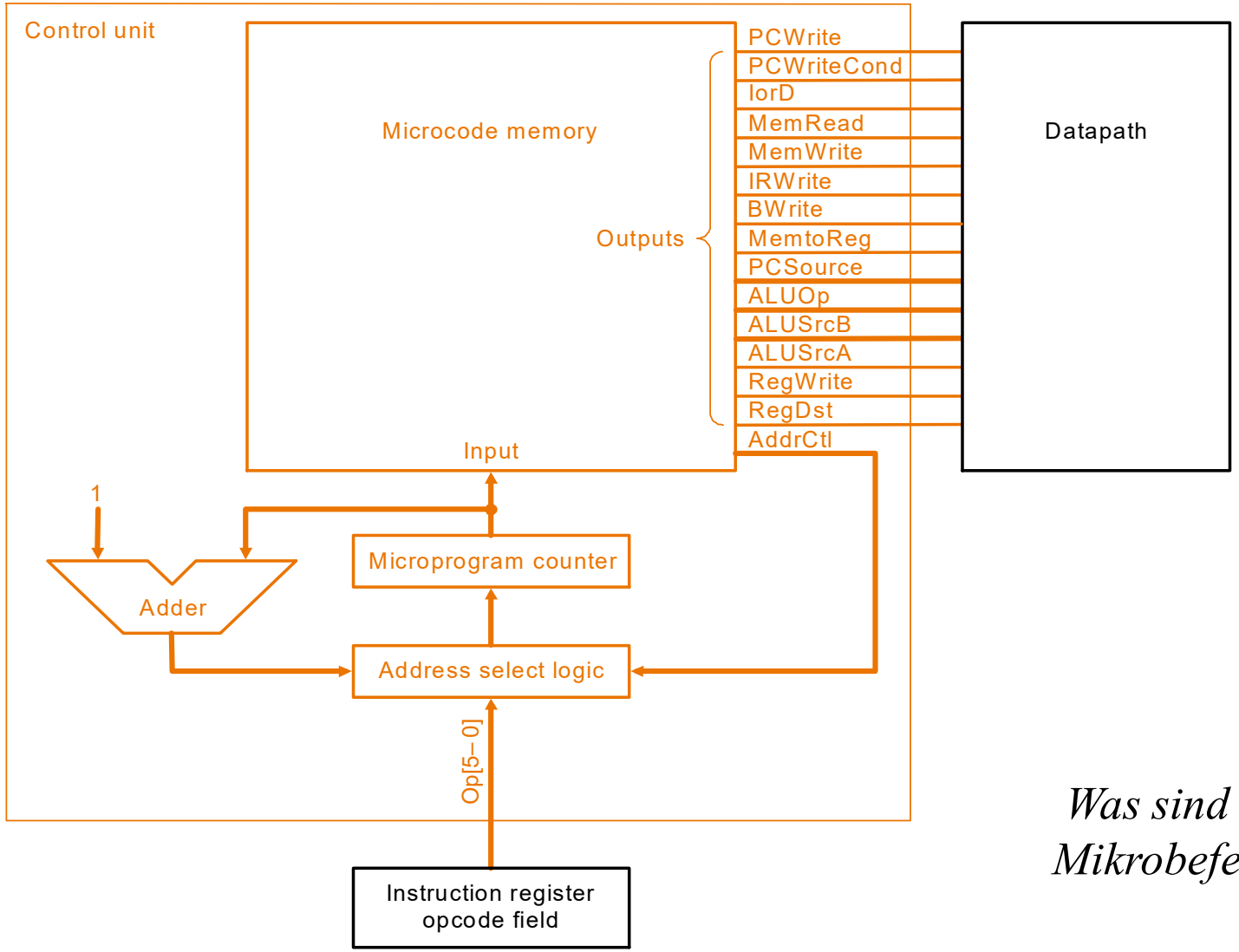
Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	`jmp`	1001
000100	`beq`	1000
100011	`lw`	0010
101011	`sw`	0010

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



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Mikroprogrammierung



Was sind die Mikrobefehle?

Mikroprogrammierung (2)

- **Eine Spezifikations-Methode**
  - brauchbar auch für Hunderte von Opcodes, Modes, Zyklen, etc.
  - Signale werden symbolisch spezifiziert indem man Mikroinstruktionen benutzt

Mikroprogrammierung (3)

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- ***Werden zwei Implementierungen derselben Architektur denselben Mikrocode haben?***
- ***Was wird ein Mikroassembler tun?***

Mikroinstruktions-Format

Field name	Value	Signals active	Comment
	Add	ALUOp = 00	Cause the ALU to add.
ALU control	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for
			branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
	B	ALUSrcB = 00	Register B is the second ALU input.
SRC2	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
	Read		Read two registers using the rs and rt fields of the IR as the register
			numbers and putting the data into registers A and B.
	Write ALU	RegWrite,	Write a register using the rd field of the IR as the register number and
Register		RegDst = 1,	the contents of the ALUOut as the data.
control		MemtoReg = 0	
	Write MDR	RegWrite,	Write a register using the rt field of the IR as the register number and
		RegDst = 0,	the contents of the MDR as the data.
		MemtoReg = 1	
	Read PC	MemRead,	Read memory using the PC as address; write result into IR (and
		IorD = 0	the MDR).
Memory	Read ALU	MemRead,	Read memory using the ALUOut as address; write result into MDR.
		IorD = 1	
	Write ALU	MemWrite,	Write memory using the ALUOut as address, contents of B as the
		IorD = 1	data.
	ALU	PCSource = 00	Write the output of the ALU into the PC.
		PCWrite	
PC write control	ALUOut-cond	PCSource = 01,	If the Zero output of the ALU is active, write the PC with the contents
		PCWriteCond	of the register ALUOut.
	jump address	PCSource = 10,	Write the PC with the jump address from the instruction.
		PCWrite	
	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
Sequencing	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Exceptions

- **Kontrollfluss**
  - der normale Kontrollfluss wird durch verschiedene Ursachen verändert
    - Branches und Jumps
      - hier wird durch den Befehl selbst in erwarteter Weise der Kontrollfluss verändert
    - Interrupts und Exceptions
      - der Kontrollfluss wird durch besondere Ereignisse verändert
  - Exceptions
    - Ereignis aus dem Inneren eines Prozessors
      - arithmetischer Überlauf
      - undefinierte Instruktion
  - Interrupts
    - Ereignis, das von außerhalb des Prozessors kommt
      - I/O-Geräte kommunizieren so mit dem Prozessor, z.B Tastatur, Netzwerkkarte, etc.

Exceptions (2)

- Exceptions zu erkennen und die notwendigen Maßnamen zu ergreifen liegt immer auf dem kritischen Timing Pfad in der Steuerung des Prozessors
- daher müssen Exceptions beim Entwurf der Steuereinheit von vornherein mit berücksichtigt werden
- ein späteres Hinzufügen zu einer aufwendigen Steuereinheit führt fast immer zu einer Verlangsamung des Taktes

Exceptions (3)

- **Behandlung von Exceptions**
  - bisher nur: Überlauf und undefinierte Instruktion
  - Aktionen
    - die Adresse der Instruktion, die eine Exception auslöst, muss abgespeichert werden
      - neues Register: `EPC` (*exception program counter*)
    - Kontrolle wird an eine bestimmte Adresse im Betriebssystem übertragen
      - Betriebssystem kann nun geeignete Maßnahmen ergreifen
        - » geeignete Reparaturmaßnahmen
        - » Warnung, Fehlermeldung
        - » Programmabbruch
    - Betriebssystem kann EPC benutzen, um das Programm an der unterbrochenen Stelle wieder fortzusetzen

Exceptions (4)

- das Betriebssystem muss auch die Ursache für die Exception erfahren
- zwei Hauptmethoden
  - vectored interrupts
    - je nach Ursache wird zu einer anderen Adresse verzweigt
    - Adressen der Routinen stehen in einer Tabelle im Speicher
    - oder die Einsprungpunkte sind in einem festen Abstand, z.B 8 Befehlsworte, voneinander entfernt
  - Status Register
    - Methode, die MIPS verwendet (Name des Registers: `Cause`)
    - Ursache steht in einem Feld im Status Register des Prozessors

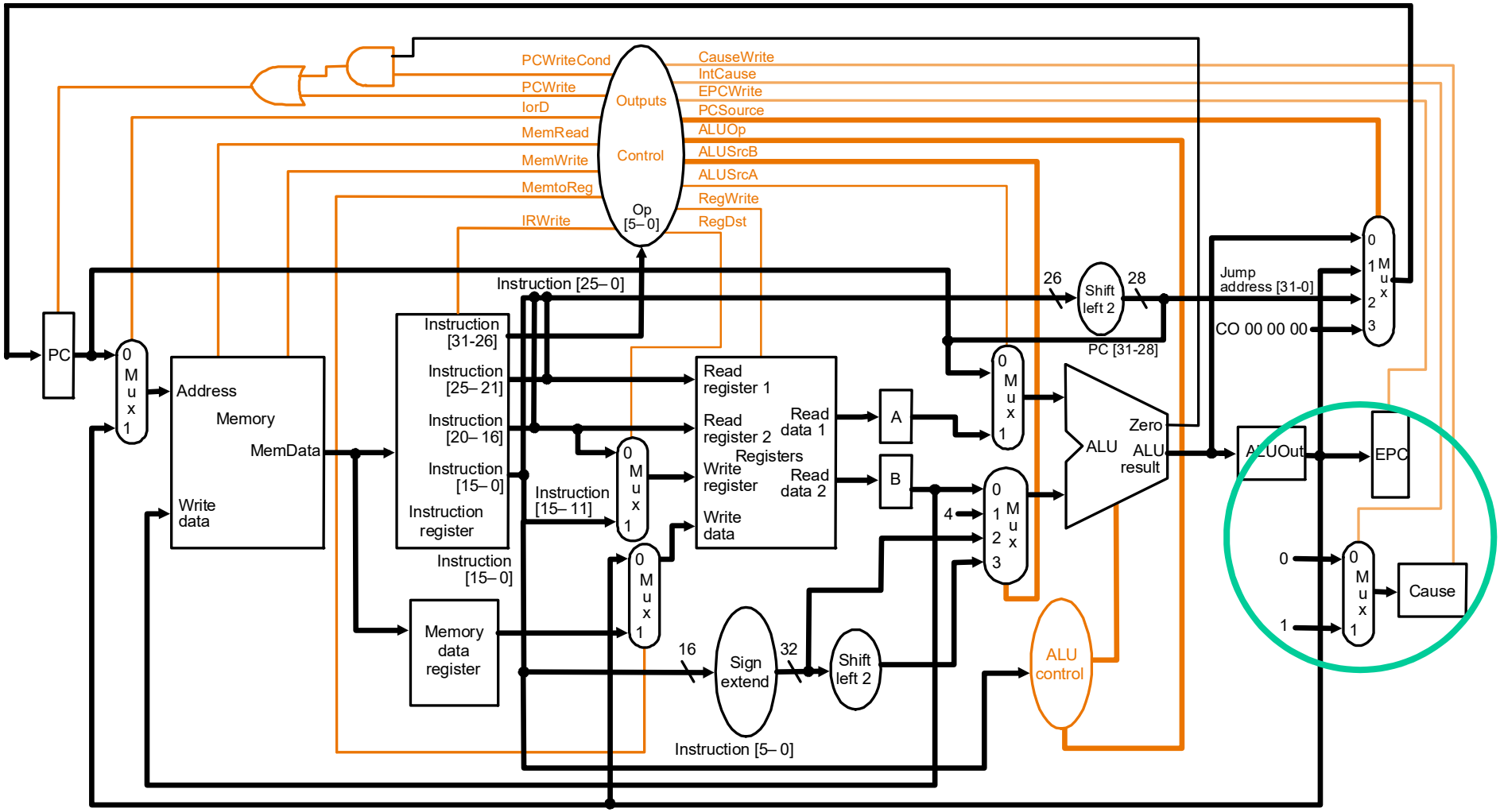
Exceptions (5)

- **Implementierung in MIPS**
  - EPC
    - 32-bit Register
    - speichert die Adresse der betroffenen Instruktion
  - Cause
    - 32-bit Register ("einige" Bits nicht benutzt)
    - hält die Ursache der Exception
      - nur das niederwertigste Bit wird hier benötigt

        undefined instruction: 0

        arithmetic overflow: 1
  - Steuersignale
    - EPCWrite: Schreiben des EPC
    - CauseWrite: Schreiben des Cause Registers
    - IntCause: 1-bit Steuersignal (Daten für das niederwertigste Bit)
    - exception address: $(C000\ 0000)_{hex}$ Adresse des Exception Handlers

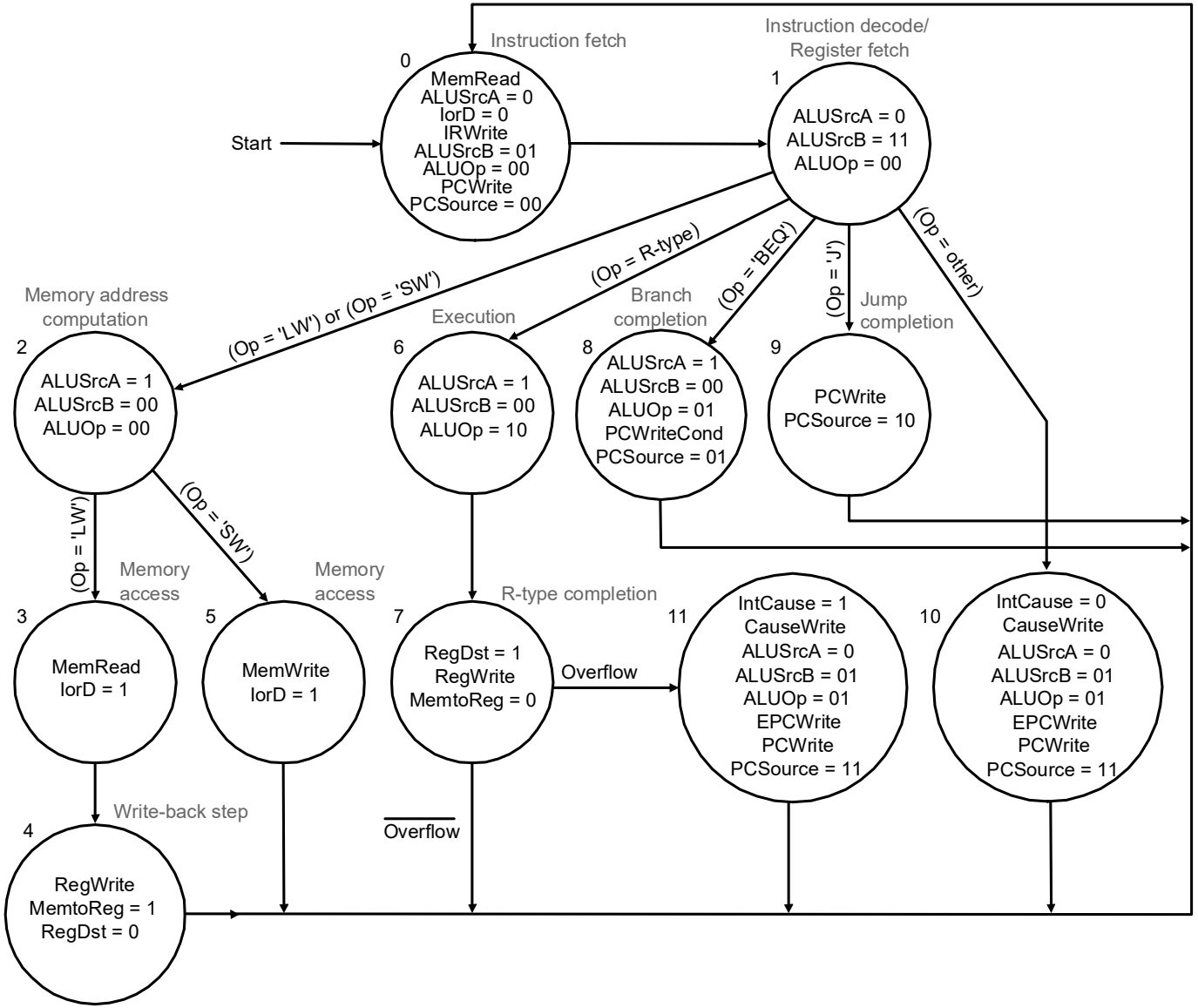
Datenpfad mit Exceptions



Exceptions (6)

- **Implementierung der Steuerung**
  - zwei neue Zustände
    - *undefined instruction*: Zustand 10
      - Übergang, wenn im Zustand 1 ein anderer Opcode anliegt, als alle bekannten
    - *arithmetic overflow*: Zustand 11
      - Übergang, wenn im Zustand 7 (*R-type completion*) die ALU ein Overflow-Signal generiert

Steuerung mit Exceptions



Zusammenfassung Mehrzyklus-Datenpfad

- **Datenpfad**
  - weniger funktionale Einheiten als beim Ein-Zyklus-Datenpfad
    - Mehrfachnutzung der Einheiten in verschiedenen Schritten
  - dafür zusätzliche Register nach den funktionalen Einheiten
- **Steuerung**
  - festverdrahtetes Schaltwerk (FSM) oder mikroprogrammiertes Steuerwerk

Zusammenfassung Mehrzyklus-Datenpfad (2)

- **Warum ist das Ganze schneller als beim Ein-Zyklus-Datenpfad?**
  - Taktfrequenz ist knapp 5 mal so hoch wie beim Einzyklus-Datenpfad
    - Im Wesentlichen ist in jedem Schritt nur ein Fünftel der Arbeit zu leisten, daher Taktfrequenz im Idealfall 5 mal so hoch, wie im Einzyklusdatenpfad.
    - Zwei Ursachen für etwas kleinere Taktfrequenz
      - zusätzliche Register verbrauchen zusätzliche Zeit
      - ungleichmäßige Aufteilung der Gesamtarbeit auf die 5 Schritte führt zu kleinerer Taktfrequenz, die sich ja nach dem längsten Verarbeitungsschritt richten muss
  - ein einzelner Befehl benötigt zwischen 3 und 5 Takte
    - `lw` braucht als einziger Befehl mit 5 Schritten sogar etwas länger als im Einzyklusdatenpfad
    - alle anderen Befehle brauchen nur 3 oder 4 Takte und sind damit schneller