

Technische Grundlagen der Informatik 2

Rechnerorganisation

Kapitel 5, Teil 1: Eintaktprozessor (Single-Cycle-Processor)

Prof. Dr. Ben Juurlink

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

SS 2014

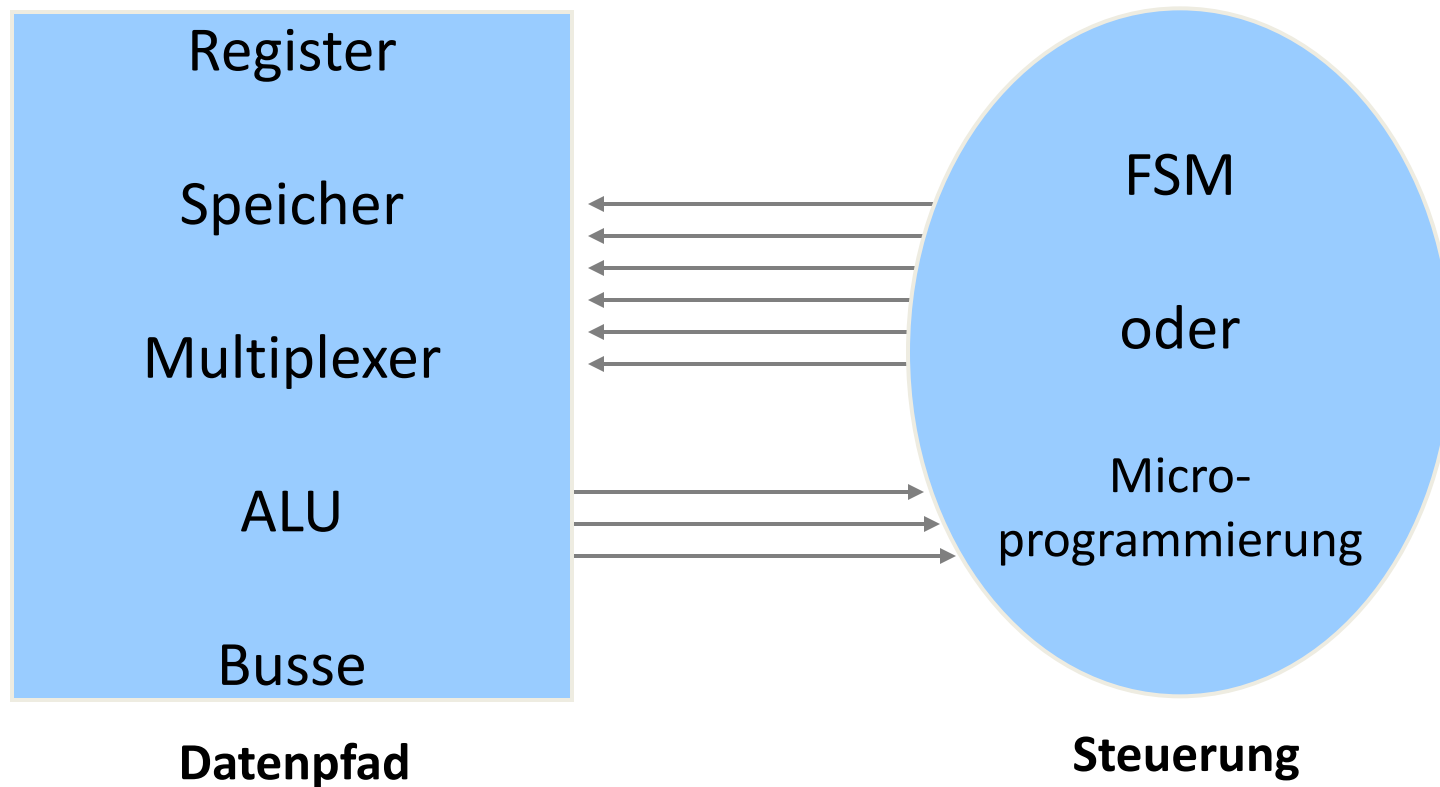
- Nach diesem Kapitel sollten Sie in der Lage sein...
 - einen Datenpfad zu entwerfen, der eine Teilmenge des MIPS-Befehlssatzes unterstützt.
 - die Kontrollsignale anzugeben, die für die Ausführung bestimmter Befehle gebraucht werden.
 - einen Befehl zu dem unterstützten Befehlssatz hinzuzufügen.
 - Hardware zu entwerfen, welche die Kontrollsignale für einen Befehlscode berechnet.
 - zu erklären, warum Eintakt-Implementierungen uneffizient und warum Mehrzyklen-Implementierungen besser sind



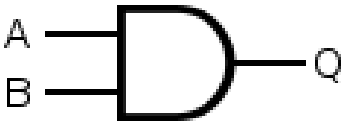

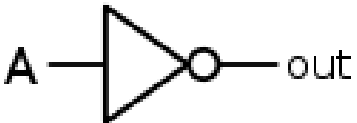

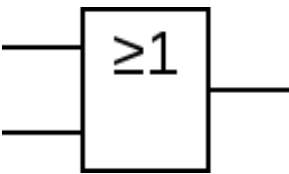
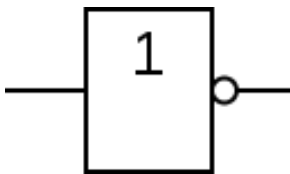
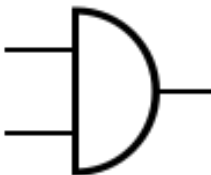
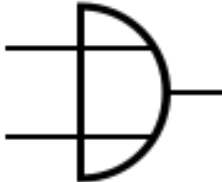
- Einleitung
- Bausteine
- Datenpfad des Eintaktprozessors
- Steuerung des Eintaktprozessors
- Leistung des Eintaktprozessors

- Wir sehen uns nun eine Implementierung des MIPS an.
- Vereinfacht, da nur folgende Befehle unterstützt werden:
 - Speicherreferenz-Befehle : `lw, sw`
 - arithmetische-logische Befehle: `add, sub, and, or, slt`
 - Kontrollfluss-Befehle: `beq, j`
- Generelle Implementierung:
 - Befehlszähler liefert die Befehlsadresse
 - Hole Befehl aus dem Speicher
 - Lese Register
 - Dekodiere den Befehl um zu entscheiden was genau zu tun ist

- **Datenpfad (datapath):** Muskeln des Prozessors
- **Steuerung (control):** Gehirn des Prozessors





	UND	ODER	NICHT
MIL/ANSI			
IEC			
DIN			



- Funktions- oder Wahrheitstabellen (truth tables):

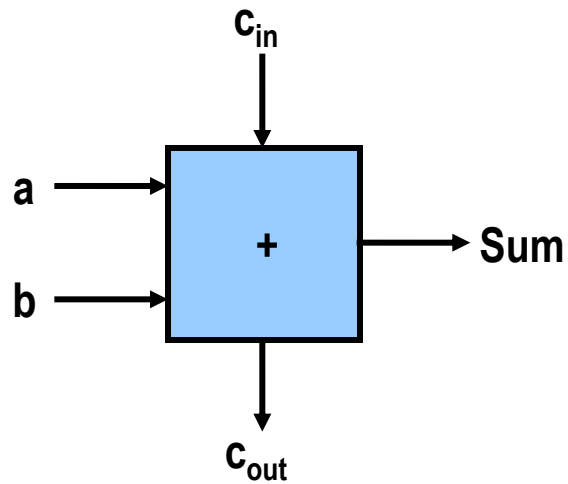
	XOR	NAND	NOR
MIL/ANSI			

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

- Blockbild:



- Funktionstabelle:

a	b	C _{in}	C _{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



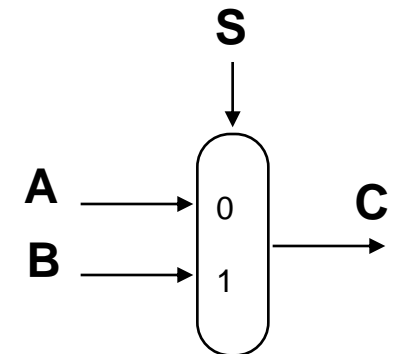
- Ein Multiplexer wählt je nach Steuersignal S einen der Inputs A oder B aus.
- Wahrheitstabelle (mit “don’t cares” (X)):

S	A	B	C
0	0	X	0
	1	X	1
1	X	0	0
	X	1	1

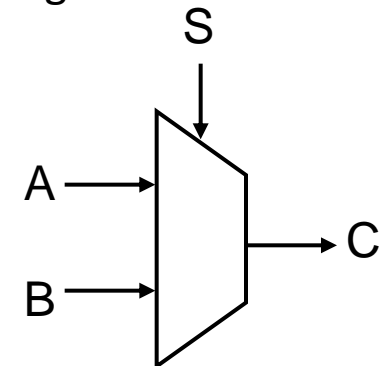
- Boole’sche Gleichung:

$$C = A\bar{S} + BS$$

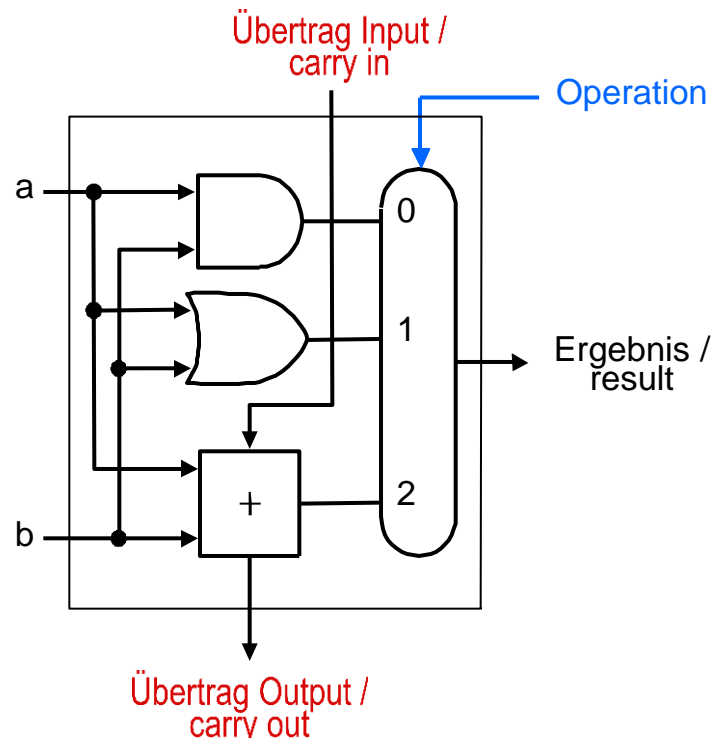
Symbol im Textbuch:



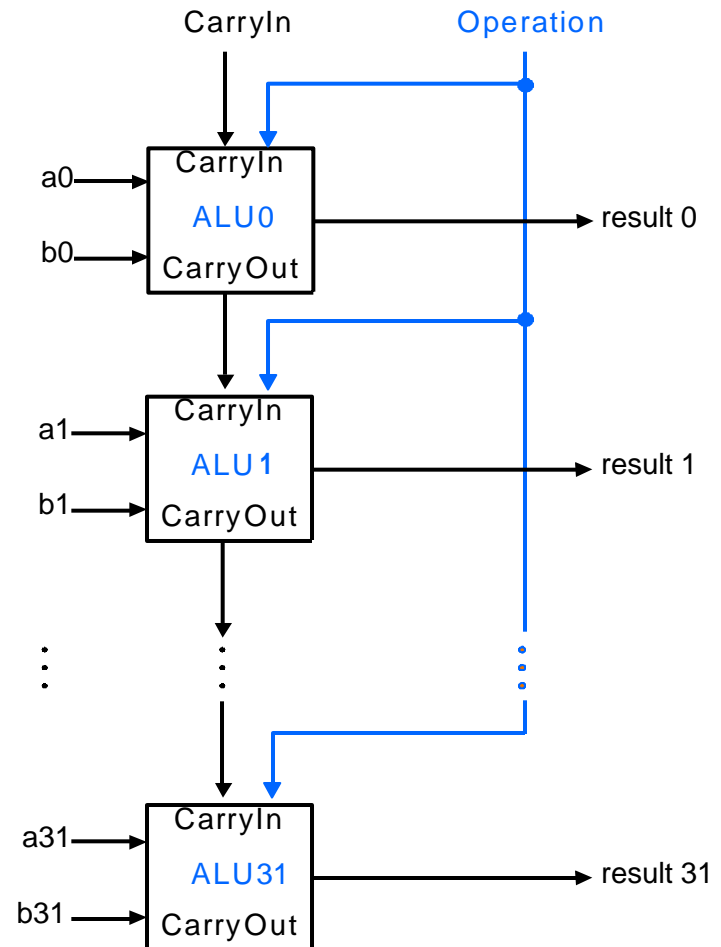
gebräuchlicher:



- MUX kann benutzt werden, um eine 1-Bit ALU zu konstruieren, die die Befehle AND, OR und + unterstützt.
- Einfach alles parallel ausführen und den richtigen Output auswählen:

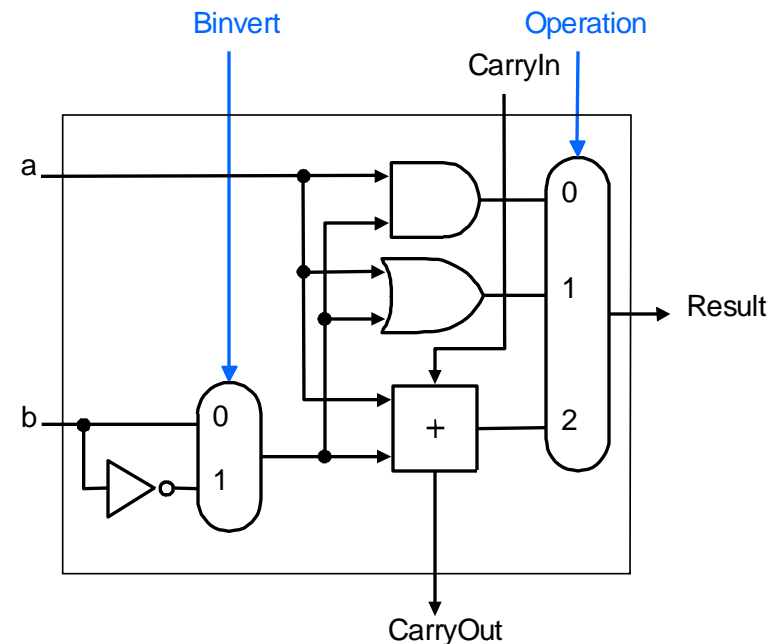


- Die 1-Bit ALU kann zur Realisierung einer 32-Bit ALU genutzt werden.
- Angenommen wird der “ripple-carry adder”, aber wir wissen, dass der “carry-look ahead adder” schneller ist.



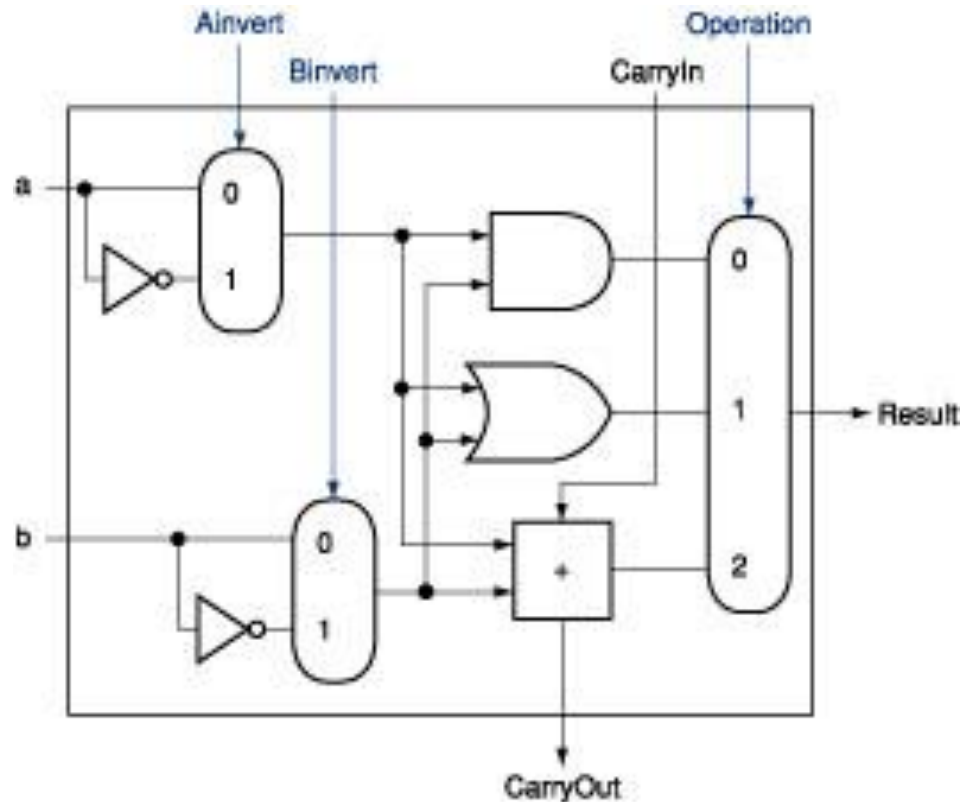
- Subtraktion verwendet 2er-Komplement:
→ einfach b negieren und addieren
- Wie negiert man b?
→ alle Bits invertieren und 1 addieren
- Eine clevere Lösung:

- Um zu subtrahieren, setze *CarryIn* von ALU0 auf 1, *Binvert* auf 1 und *Operation* auf 10





- Wahlweise kann auch a invertiert werden:

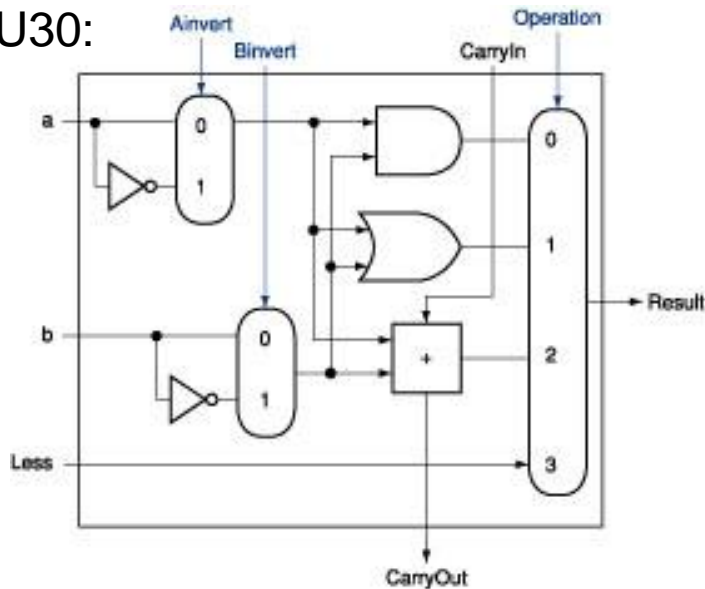


- Wie erhält man “a NOR b”?

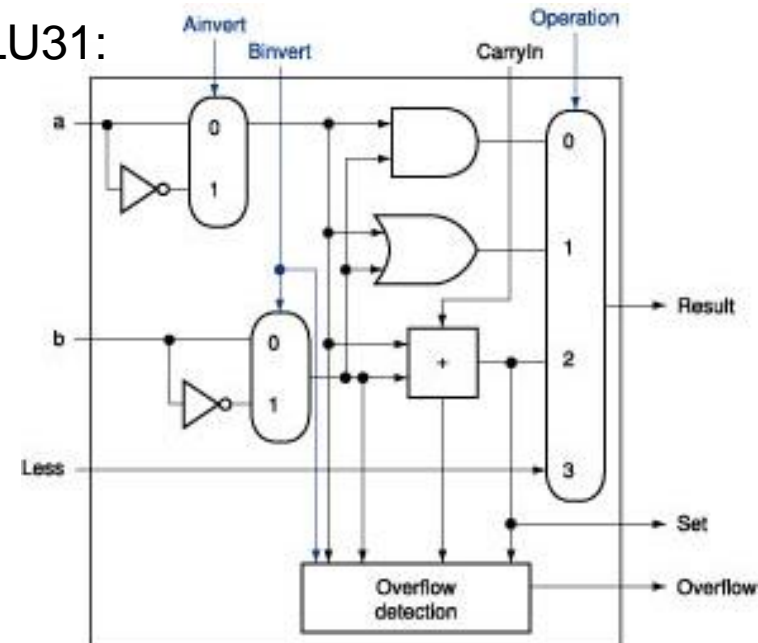
- Der Befehl `slt` “set-on-less-than” (setze wenn kleiner als) wird benötigt
 - produziert 1, wenn $rs < rt$, ansonsten 0
 - benutze Subtraktion: $(a-b) < 0$ impliziert $a < b$
- Test auf Gleichheit wird benötigt (`beq $t5, $t6, L`)
 - benutze wieder die Subtraktion: $(a-b) = 0$ impliziert $a = b$

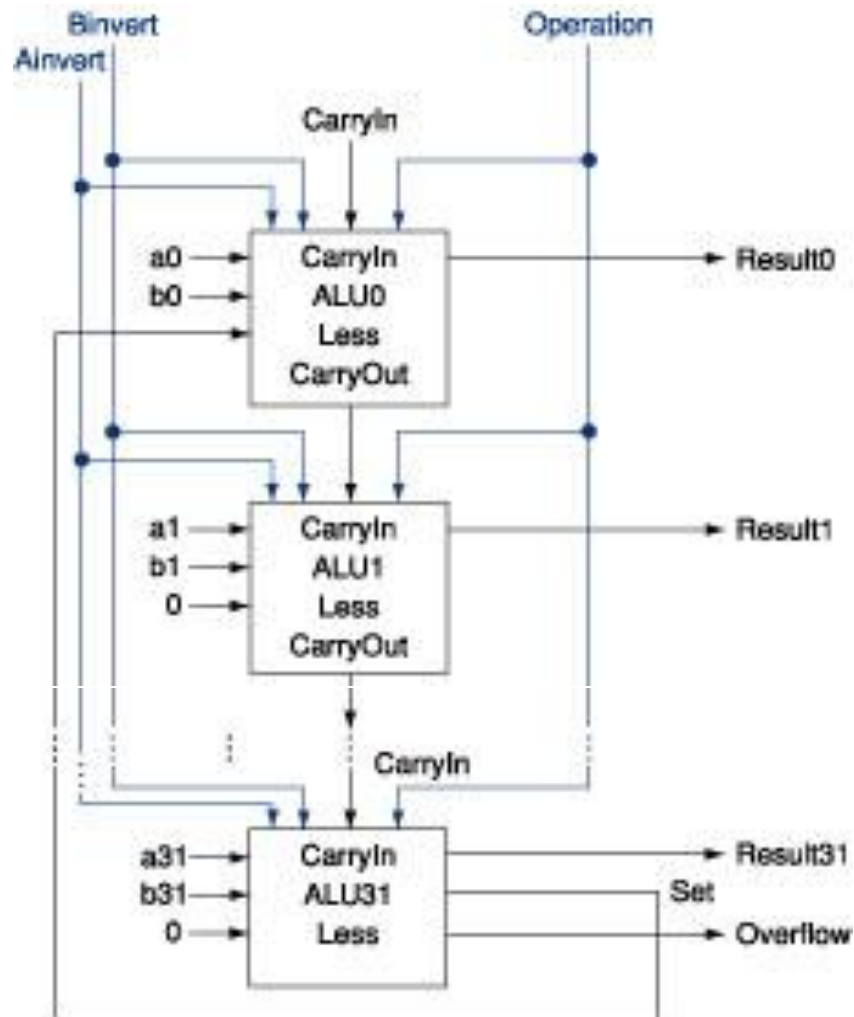
- Hinzufügen eines extra Inputs *Less*
- *Less* ist 0 für alle 1-Bit ALUs außer ALU0
- Für ALU0 ist es, setze Output von ALU31
 - *Set* ist das Vorzeichenbit vom Ergebnis der Subtraktion
- *Less*-Input wird ausgewählt, wenn *Operation*=11

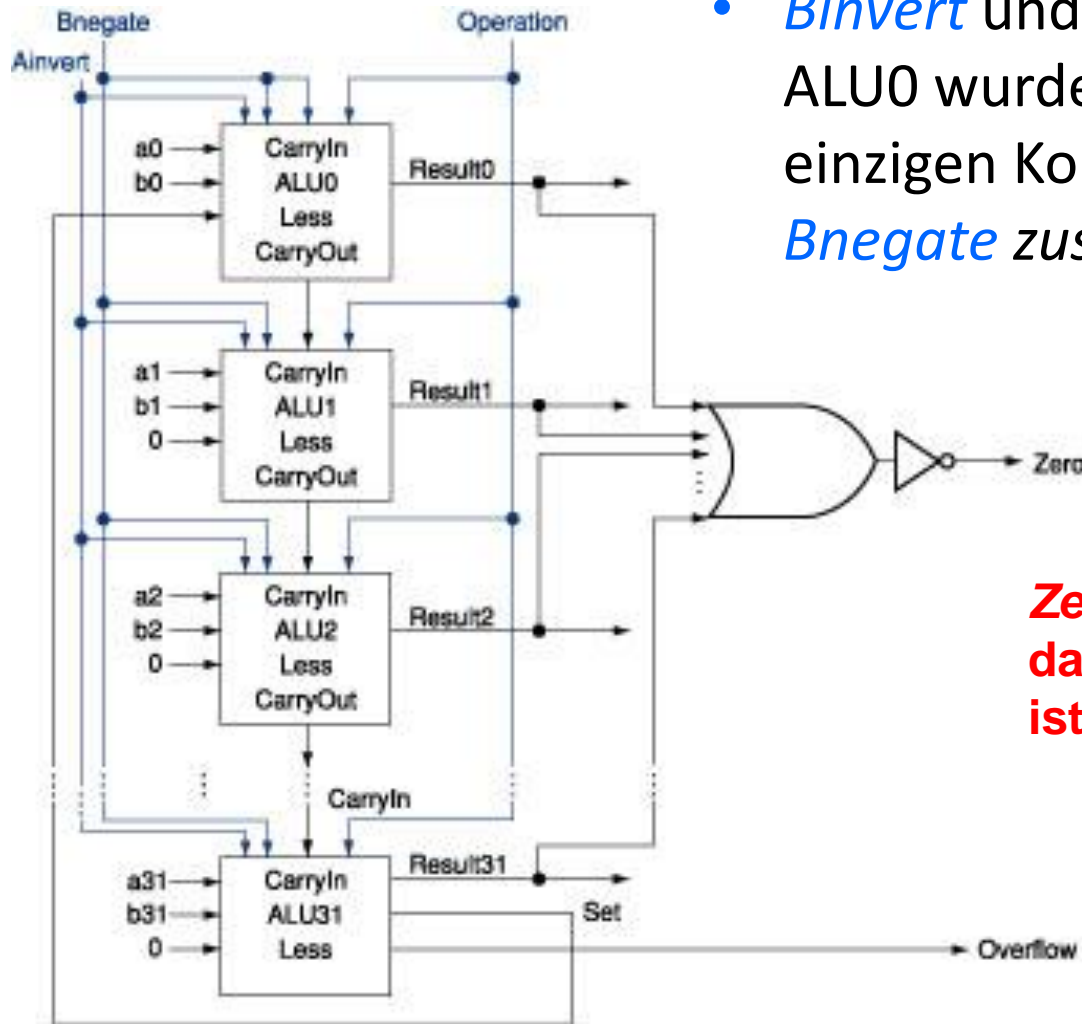
ALU0..ALU30:



ALU31:







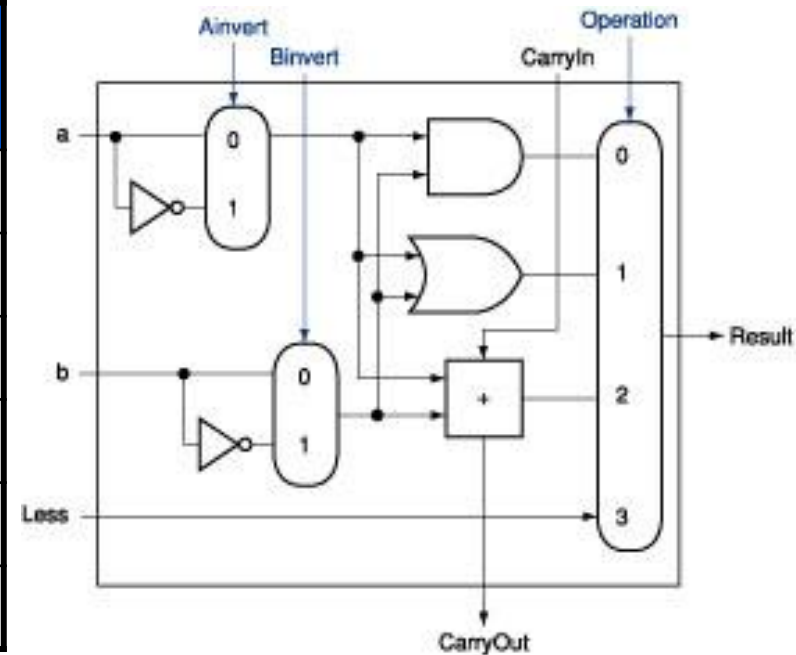
- *Binvert* und *CarryIn* der ALU0 wurden zu einem einzigen Kontrollsignal *Bnegate* zusammengefasst

Zero ist 1, wenn das Ergebnis = 0 ist!

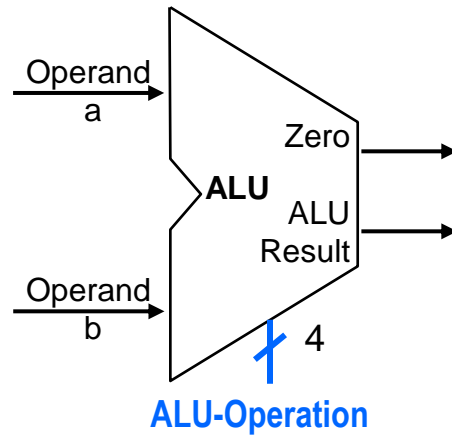


- Steuersignale um ALU-Operation zu bestimmen:

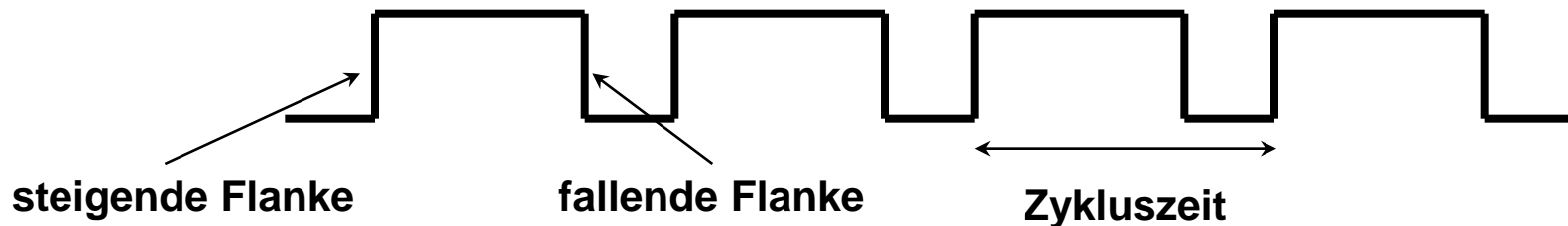
Gewünschte ALU-Aktion	<i>Ainvert</i>	<i>Bnegate</i>	<i>Operation</i>
and	0	0	00
or	0	0	01
add	0	0	10
sub	0	1	10
slt	0	1	11
nor	1	1	00



- Steuersignale haben bestimmte Bedeutung!



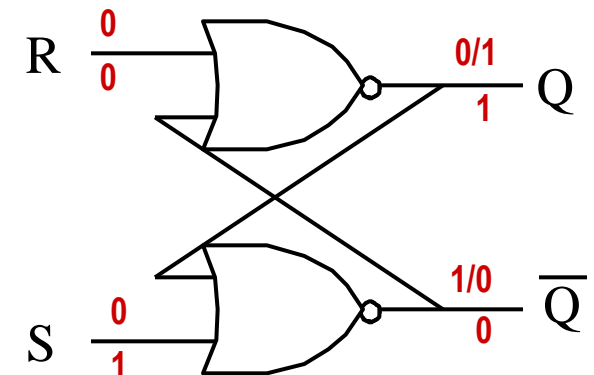
- Zwei Arten von Logikbausteinen:
 - **Schaltnetze** (**combinational logic**): Bausteine, die Daten verarbeiten. Ausgangssignale hängen nur ab von aktuellen Eingangssignale. Z. B.: ALU, MUX, ...
 - **Schaltwerke** (**sequential logic, state elements**): Baustein kann Zustand speichern, verfügt über internen Speicher. Ausgänge hängen ab von Eingängen und vom Inhalt des internen Speichers. Z.B. Register, Speicher,...
- Schaltwerke können **unclocked** oder **clocked** (durch Taktsignale synchronisiert) sein
- Taktsignale werden in synchrone Logik verwendet
 - Taktsignal gibt an, **wann** der Zustand sich ändert



- Setzen (Set) und Zurücksetzen (Reset) des Latches:
 - kreuzgekoppelte “NOR”-Gatters
 - Der Output ist abhängig vom aktuellen Input und auch von vergangenen Inputs

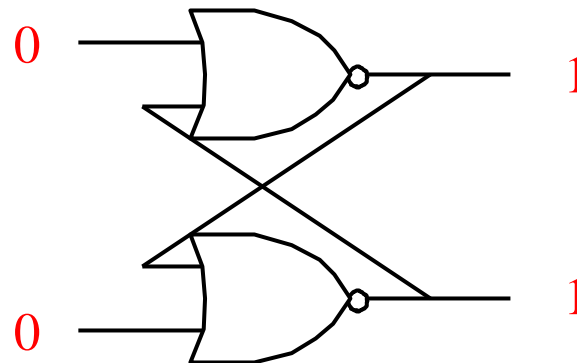
a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

- Falls Input 0, dann ist der Output das Gegenteil vom anderen Input
- Falls Input 1, dann ist der Output 0



S	R	Q	
0	0	Q	Halten
0	1	0	Wechsel
1	0	1	
1	1	?	verboten

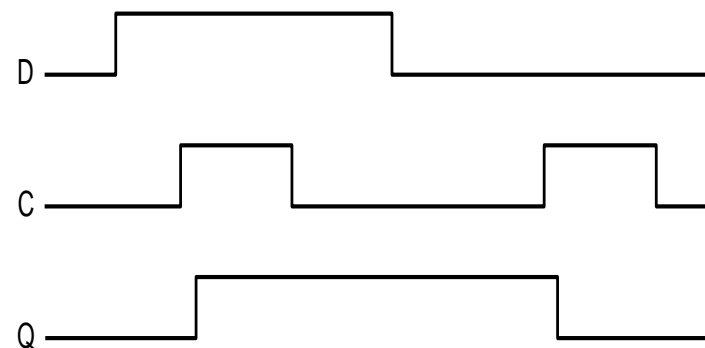
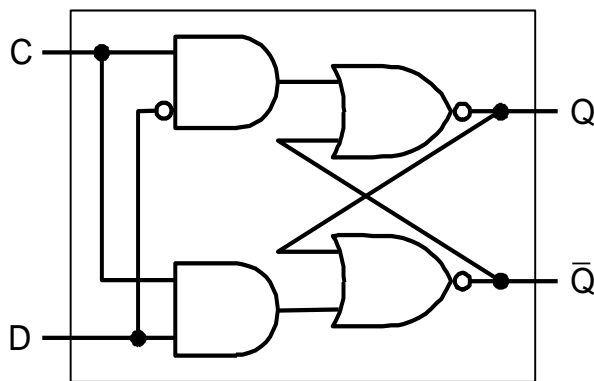
- Warum ist der Input $SR = 11$ verboten?
- Q und \bar{Q} werde beide 0
 - Was heißt das?
- Problem, falls wir zum Haltezustand zurück wollen



- Output oszilliert zwischen 0 und 1
- oder Output fest auf 1 oder 0, abhängig von den Signallaufzeit der Gatter

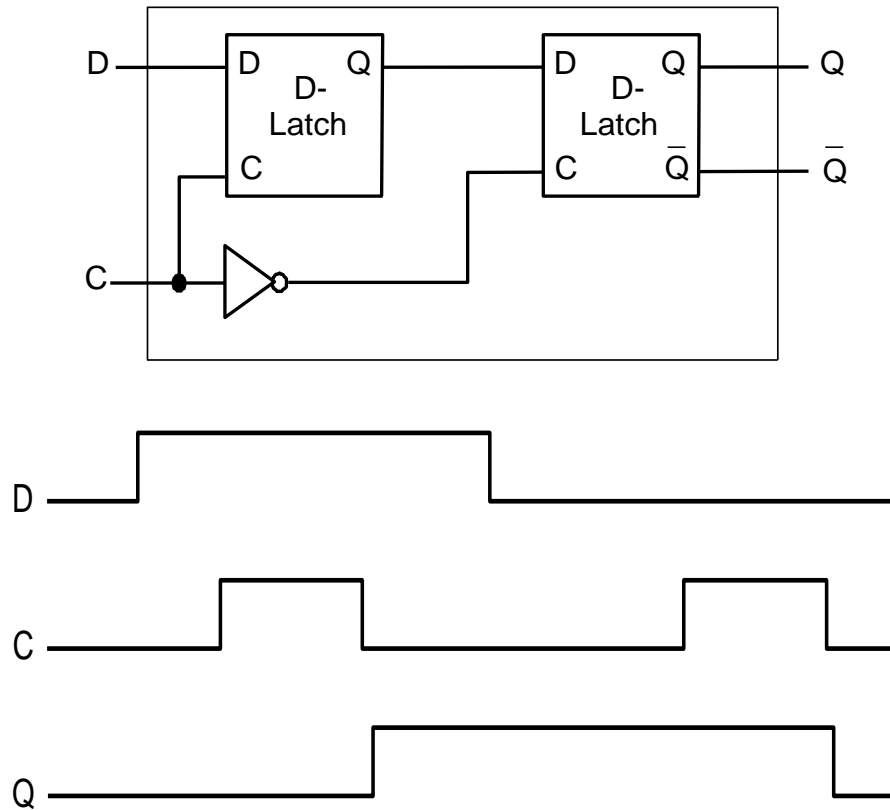
- Der Output ist gleich dem Wert, der im Element gespeichert ist
 - Es ist nicht nötig, nach Erlaubnis zu fragen, um den Wert abzurufen
- Der Wechsel des Zustands (Wert) basiert auf dem Takt (clock)
- **Latches**: Wechsel des Zustands, immer wenn der Input wechselt und “clock” = 1
- **Flip-Flop**: der Zustand wechselt nur beim Flankenwechsel der clock (**edge-triggered methodology, flankengesteuertes Taktverfahren**)
- **Clocking methodology (Taktverfahren)** definiert, **wann die Signale geschrieben werden** - ein Signal nicht zur gleichen Zeit lesen, wenn es gerade geschrieben wird

- Zwei Inputs:
 - Datenwert (D) der gespeichert werden soll
 - Clock-Signal (C) zeigt an, wenn D gespeichert werden soll
- Zwei Outputs:
 - Wert des internen Zustands (Q) und dessen Komplement
- Wahrheitstabelle nicht länger ausreichend, um das Verhalten zu definieren



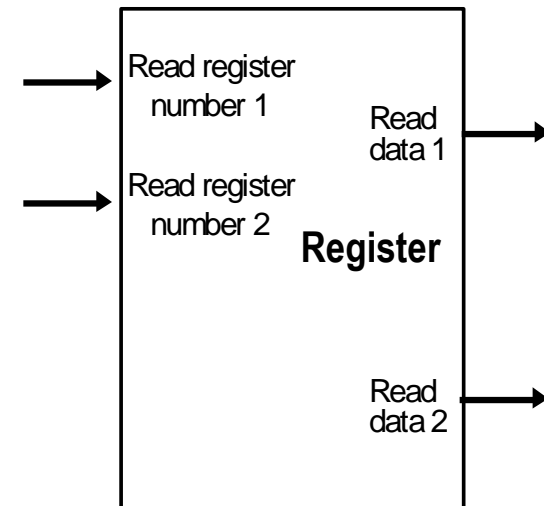
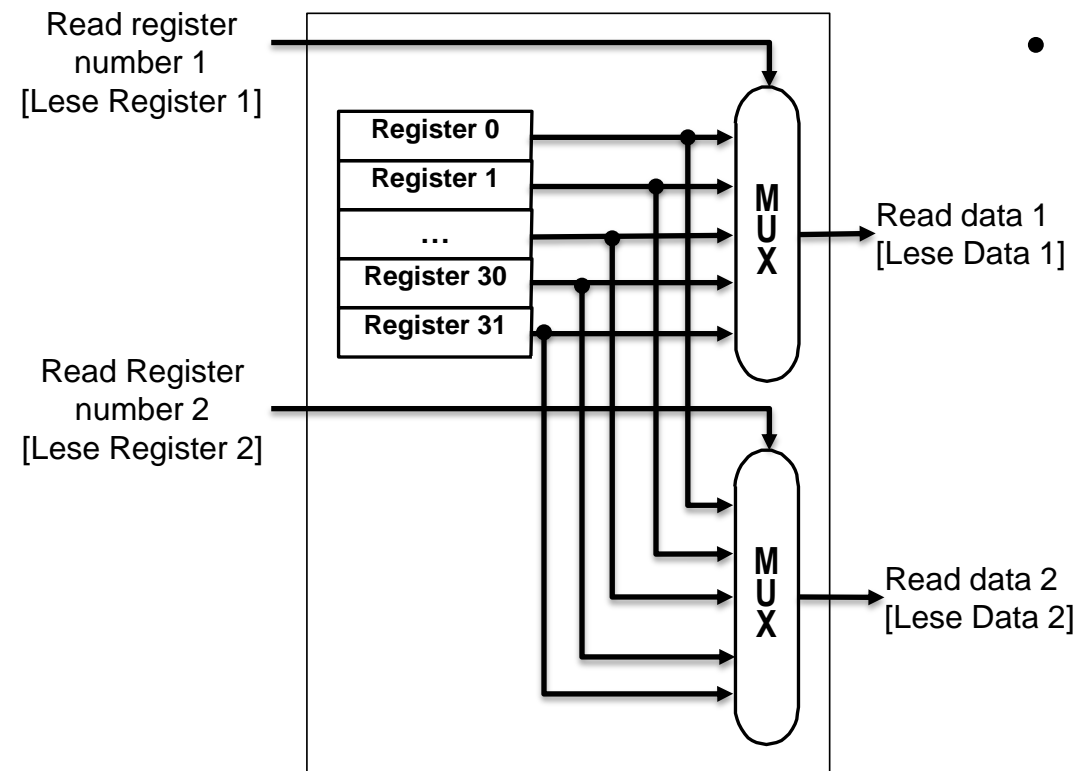


- Output wechselt nur bei fallender Taktflanke

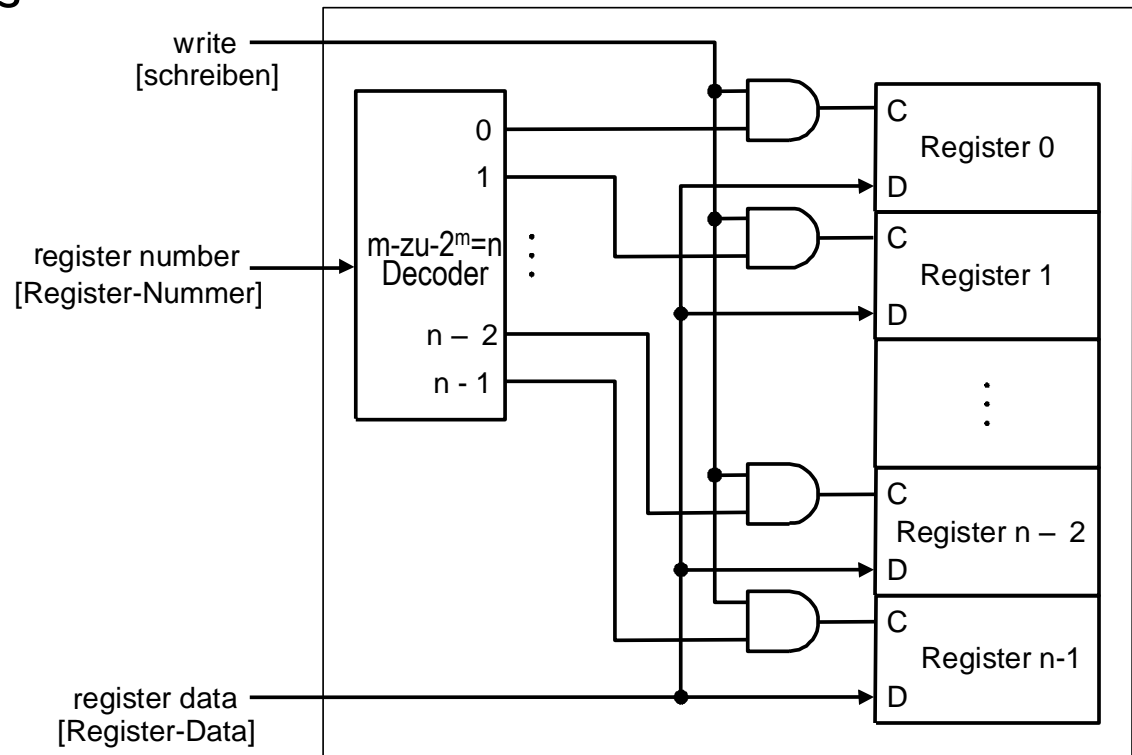


- Jedes Register besteht aus 32 D-Flip-Flops
- Implementierung der Leseports:

- Registerspeicher ohne Schreibport

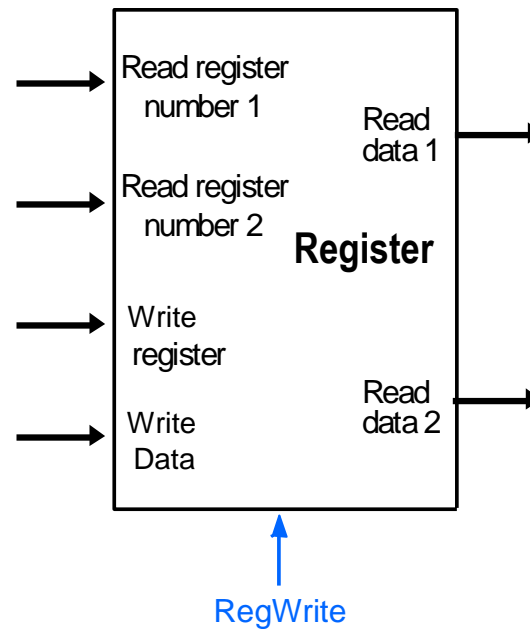


- Implementierung des Schreibports:

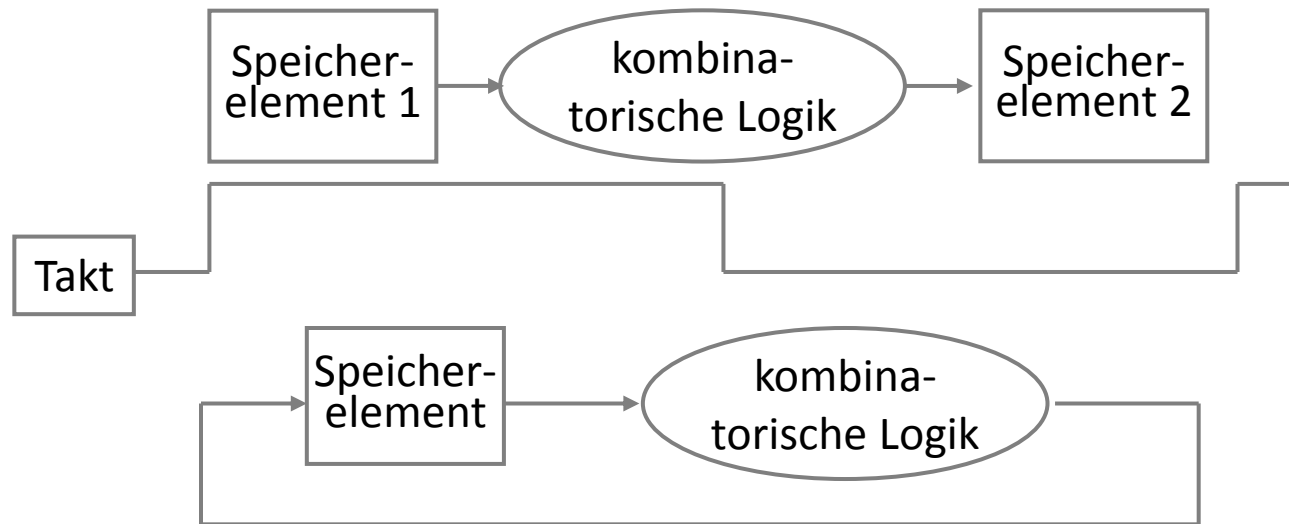


- Achtung: Der Takt (clock) wird nicht angezeigt, aber wir benutzen den Takt ,um zu bestimmen wann geschrieben werden soll

- Registerspeicher



- Ein flankengesteuertes Model
- Typische Ausführung:
 - “lese” Inhalt einiger Speicherelemente
 - sende Werte durch kombinatorische Logik
 - schreibe Ergebnisse in einen oder mehrere Speicherelemente bei der fallenden Taktflanke

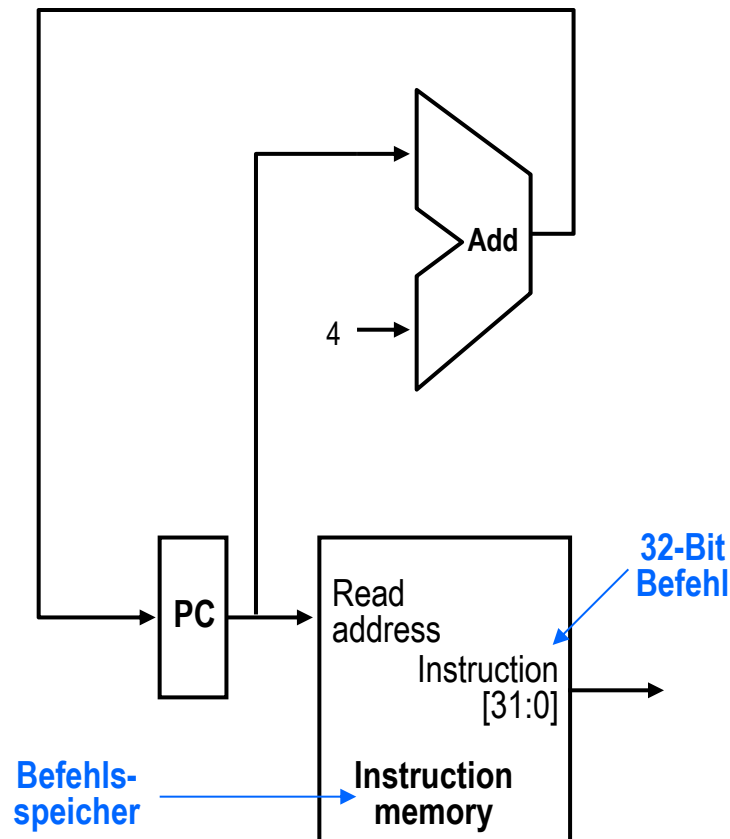


- Jetzt sind wir wirklich an dem Punkt uns die Implementierung des MIPS anzusehen
- Vereinfacht, da nur folgende Befehle unterstützt werden:
 - Speicherreferenz-Befehle: `lw, sw`
 - arithmetische-logische Befehle: `add, sub, and, or, slt`
 - Kontrollfluss-Befehle: `beq, j`
- MIPS Befehlsformate:

	6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit
R-Format	op	rs	rt	rd	shamt	func
I-Format	op	rs	rt	16-Bit Konstante		
J-Format	op	26-Bit Wort-Adresse				

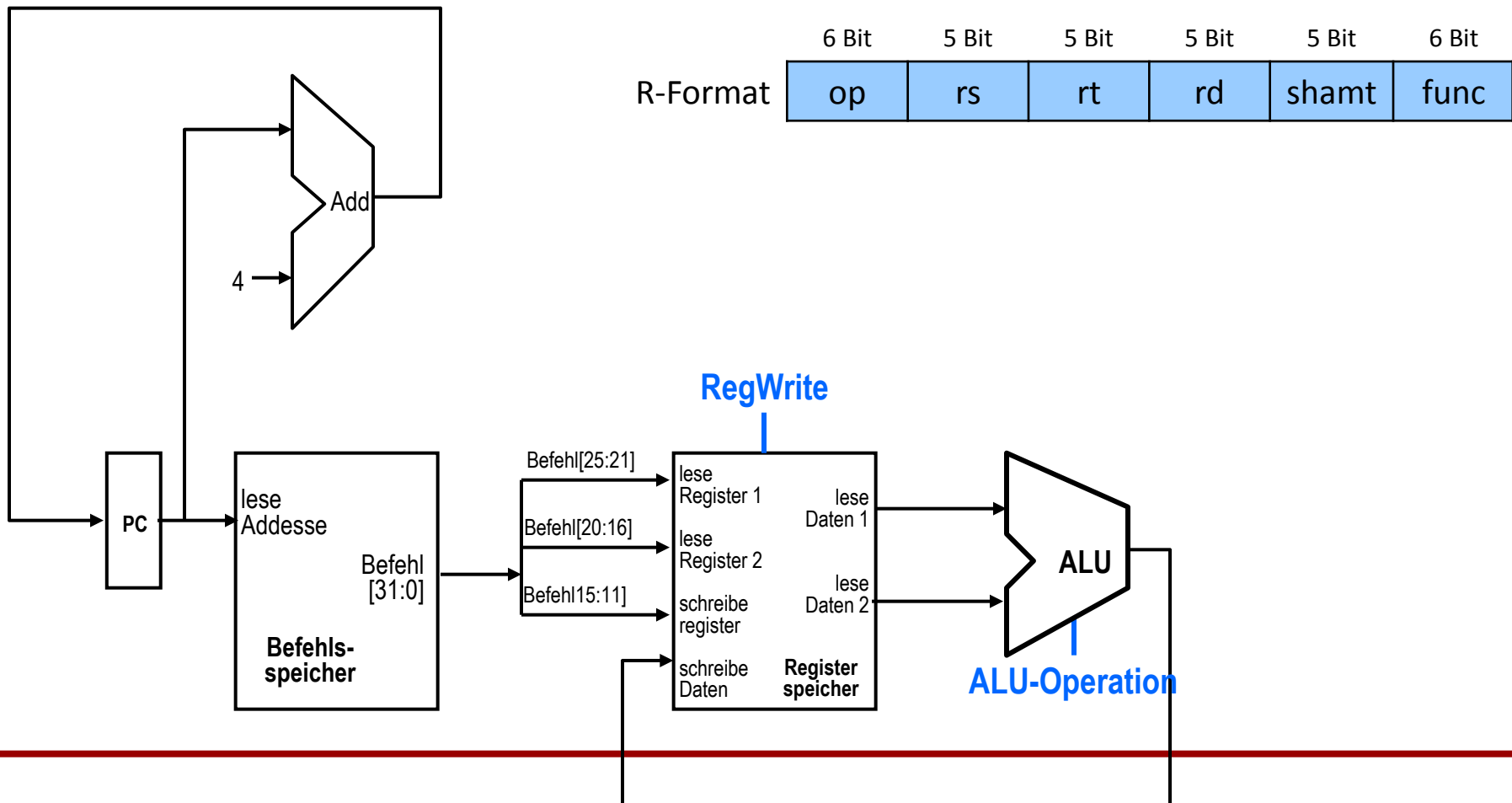
- **R-Format:** `add $1,$2,$3; sub, and, or, slt`
- **I-Format:** `addi $1,$2,100; lw $1, 40($2); beq $1,$2,100`
- **J-Format:** `j 100`

- Welche Bausteine werden benötigt, um den nächsten Befehl zu laden (und die nächste Befehlsadresse zu berechnen)?

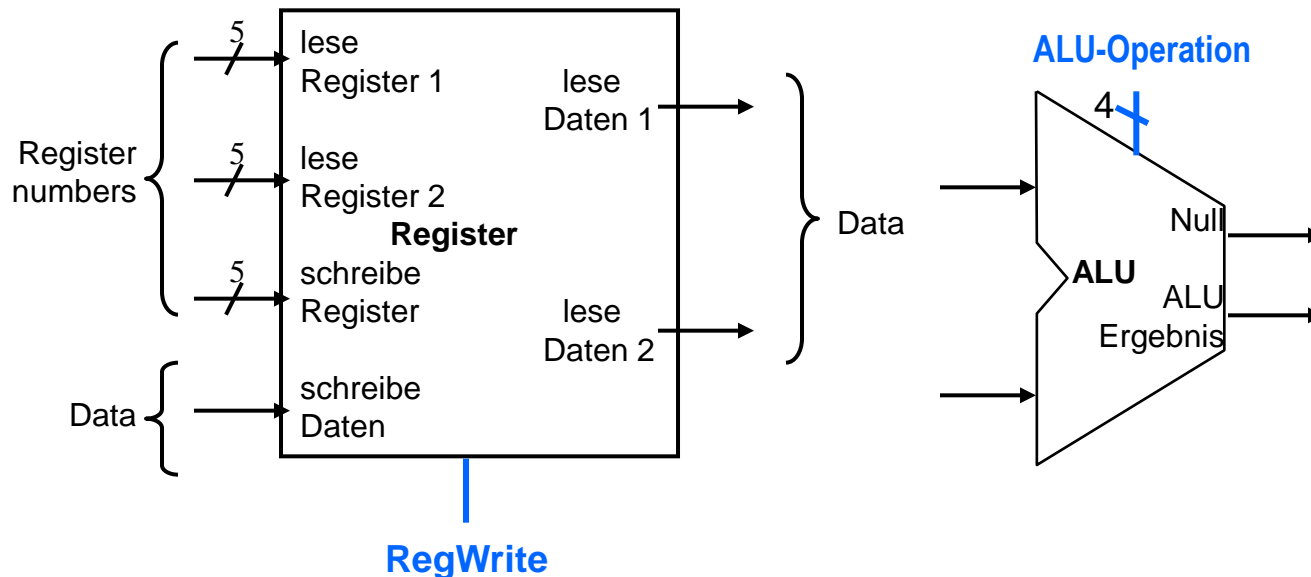


- Neuer PC wird zur fallenden Taktflanke gespeichert.

- Was brauchen wir nun, nachdem der Befehle abgerufen wurden, um die R-Type Befehle zu realisieren?

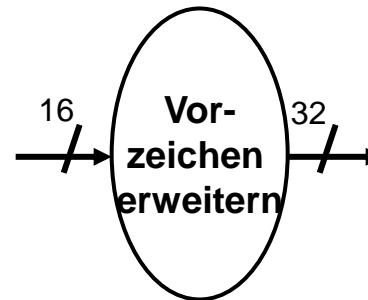
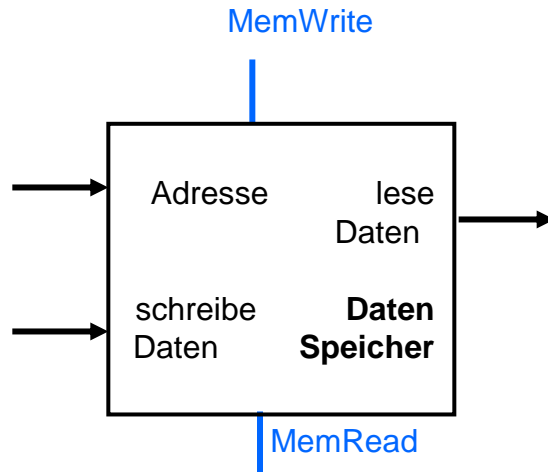
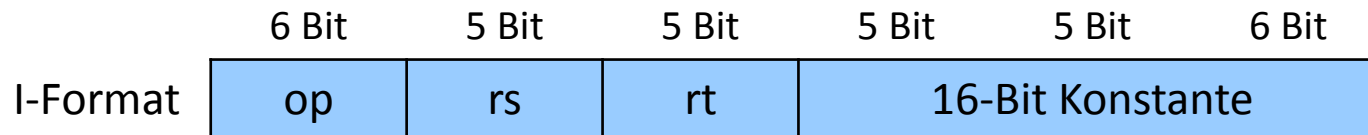


- Registerspeicher hat das Signal **RegWrite**
 - Einige Befehle schreiben den Registersatz **nicht**
- ALU hat 4 Steuersignale, die die ALU Funktion spezifizieren
 - Berechnet vom opcode und vom **möglichen** Funktionsfeld



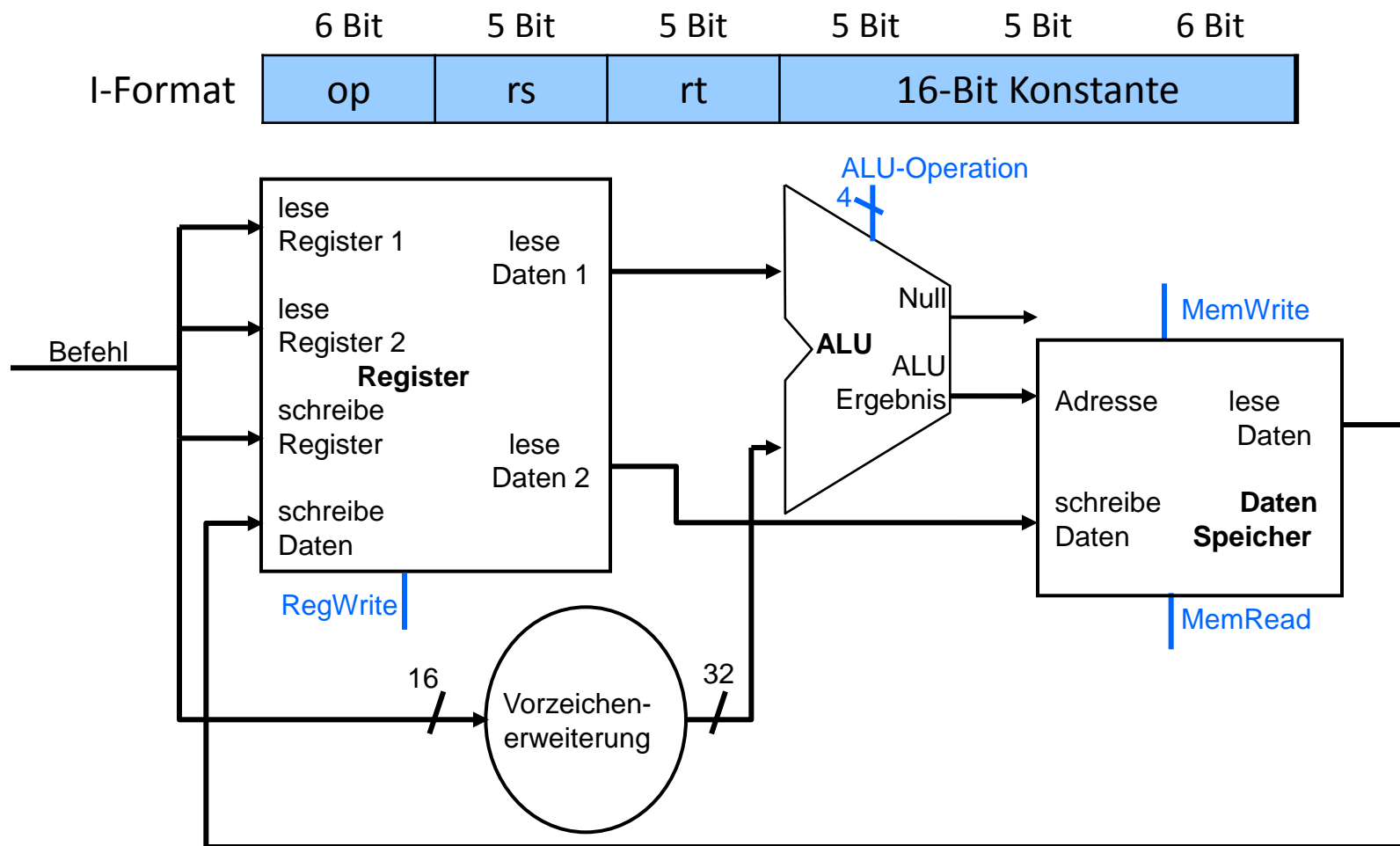
ALU-Operation	Funktion
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT
1100	NOR

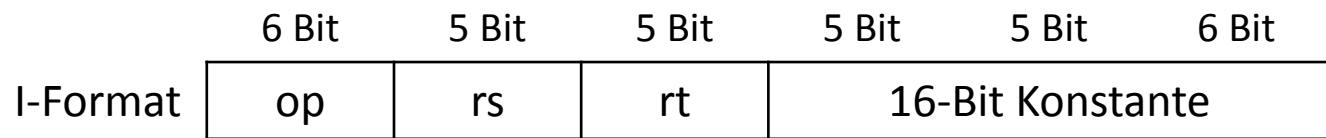
- Was brauchen wir noch zusätzlich zum Registersatz und zur ALU, um **lw** und **sw** zu implementieren?



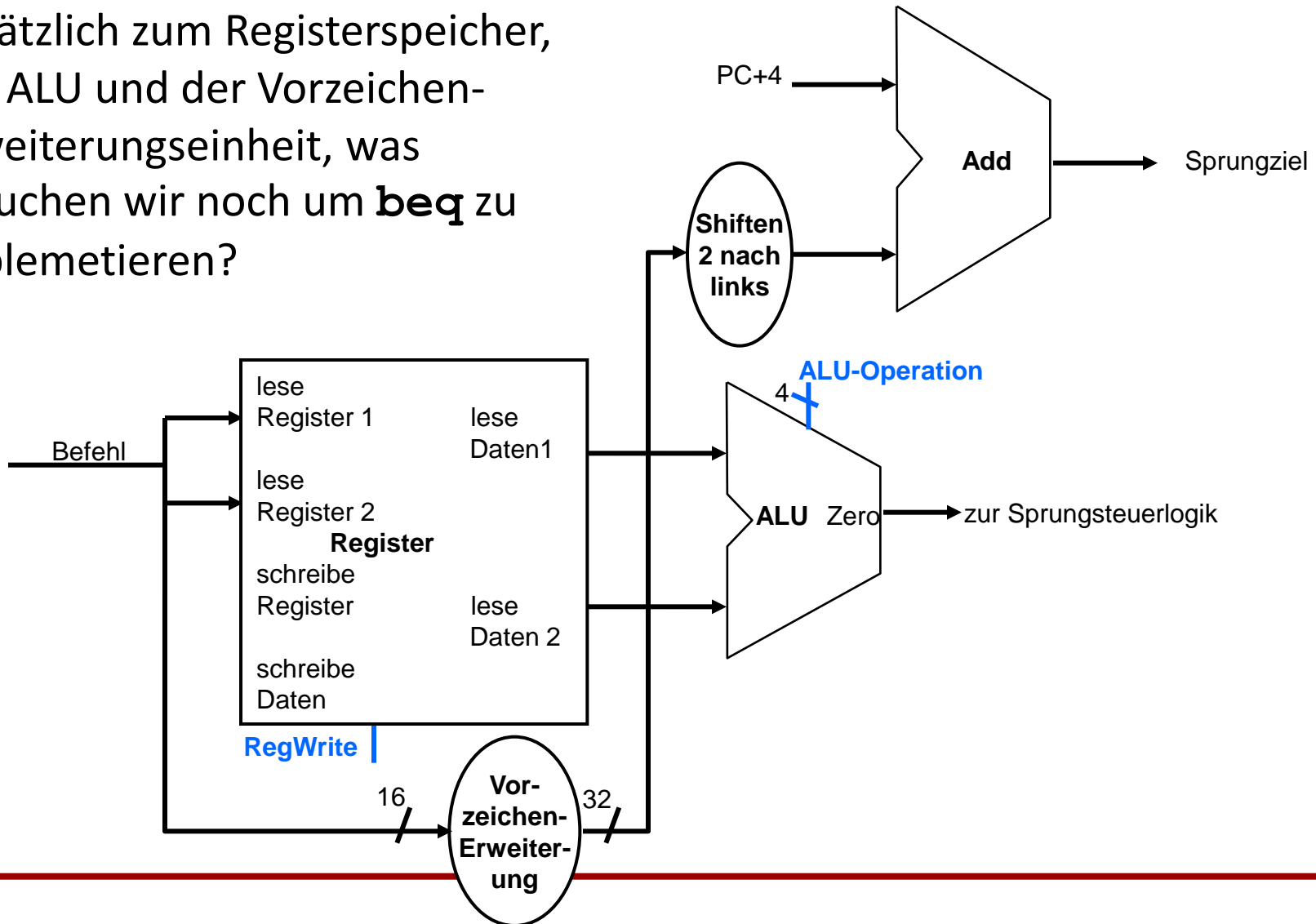
Vorzeichenerweiterung:
 FF FF wird zu FF FF FF FF
 7F FF wird zu 00 00 7F FF

- Teil des Datenpfads für Lade- und Speicherbefehle:

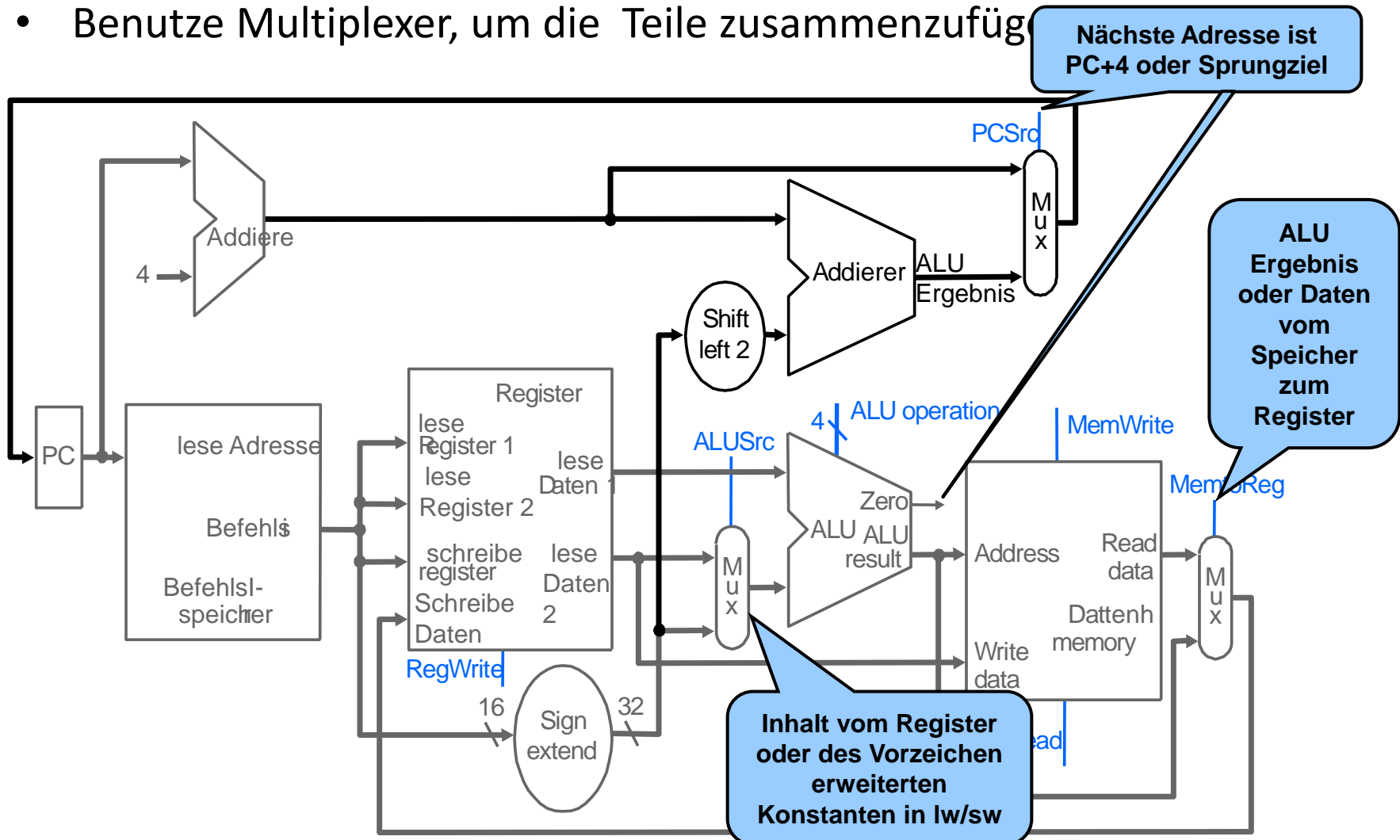




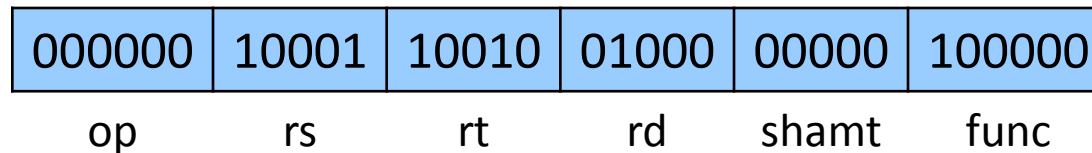
- Zusätzlich zum Registerspeicher, der ALU und der Vorzeichen-erweiterungseinheit, was brauchen wir noch um **beq** zu implementieren?



- Benutze Multiplexer, um die Teile zusammenzufügen



- Wählt die auszuführende Operation (ALU, lese/schreibe, etc.)
- Kontrolliert den Datenfluss (Multiplexer Ausgänge)
- Die Information kommt von den 32 Bits des Befehls
- Beispiel: **add \$8, \$17, \$18**
 - Befehlsformat:

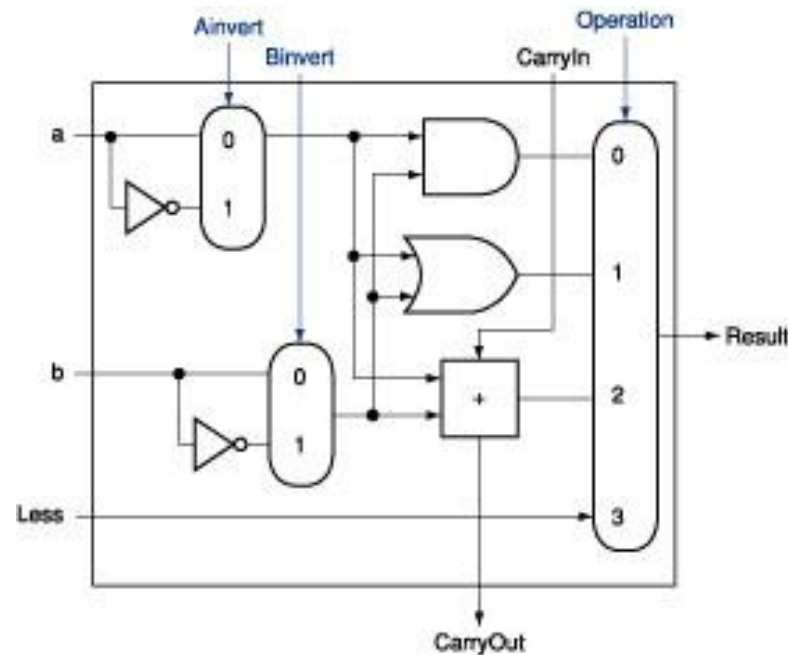


- Zuerst legen wir die Kontrolle der ALU fest und dann die Steuereinheit
 - ALU Operation basiert auf dem opcode und möglichen function code

- Was muss die ALU erledigen um
 - Lade/Speicher-Operationen?
 - **beq**?
 - R-Typ Befehle?

- ALU Steuerignale:

ALU Kontroll	Funktion
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT
1100	NOR



- Können Sie sich erinnern, warum die Steuersignale für die Subtraktion 0110 und nicht 0011 sind?

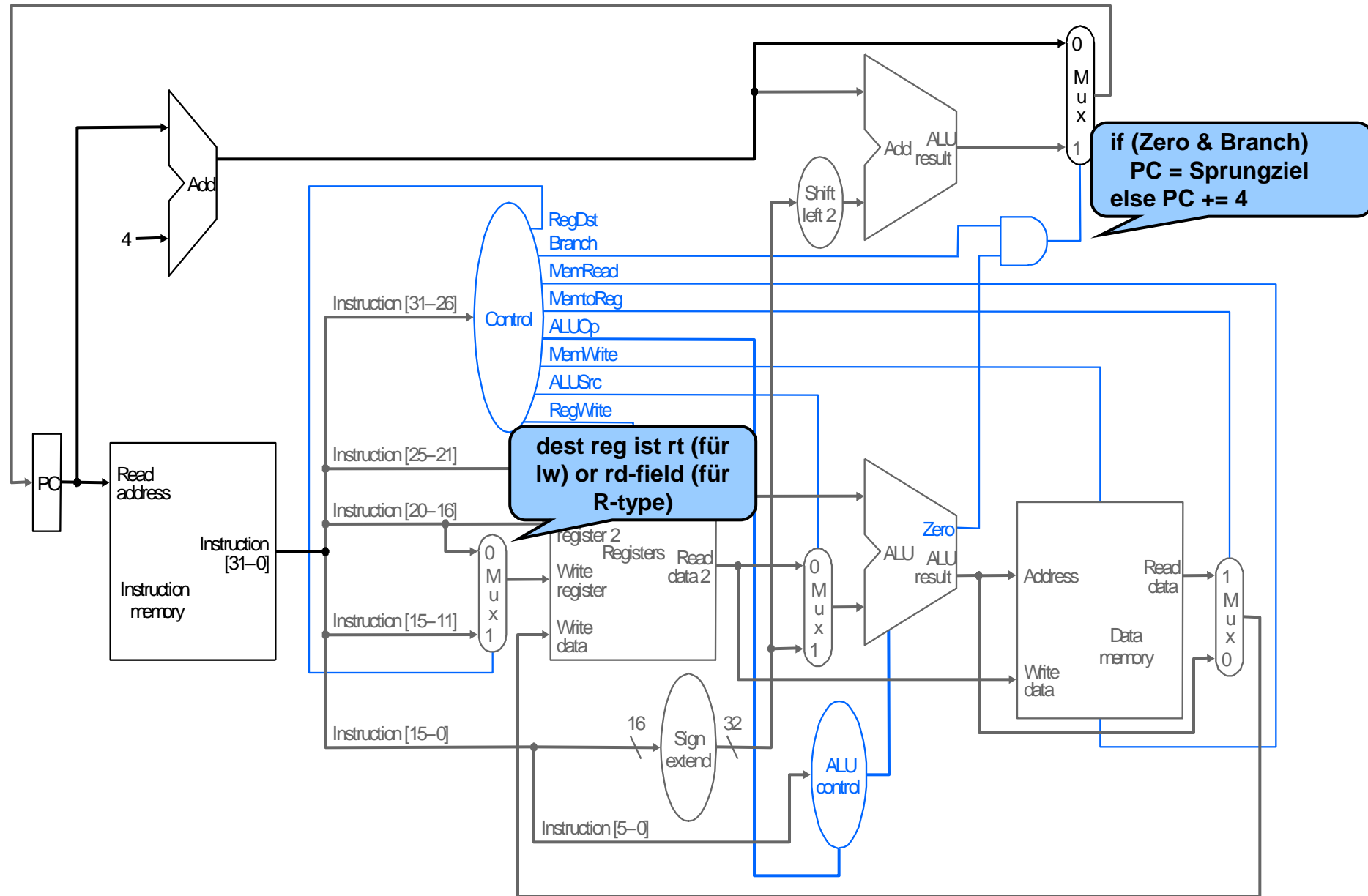


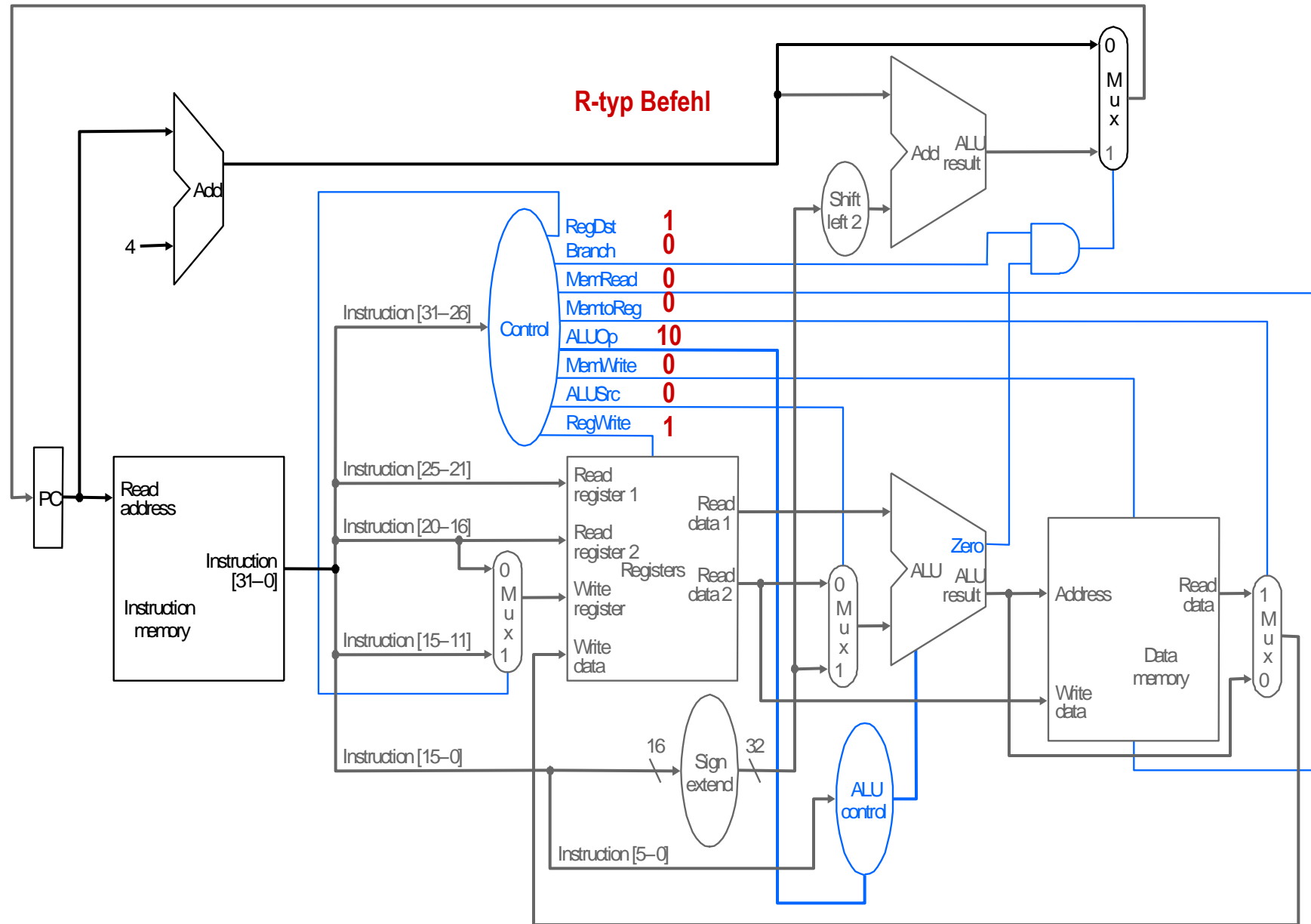
- Die 4 ALU Steuersignale werden vom opcode und vom (möglichen) funct-Feld berechnet
- Um die Steuerung zu vereinfachen, verwenden wir eine kleine Steuereinheit (**ALU control**), die die ALU Steuersignale generiert vom
 - 2-Bit Steuersignal genannt **ALUOp**
 - generiert durch die Hauptsteuereinheit
 - spezifiziert die Operation, die ausgeführt werden soll:
 - **Addition** für Lade/Speicheroperationen → $ALUOp = 00$
 - **Substraktion** für **beq** → $ALUOp = 01$
 - Bestimmt durch funct-Feld der R-type-Befehle → $ALUOp = 10$
 - Funct-Feld

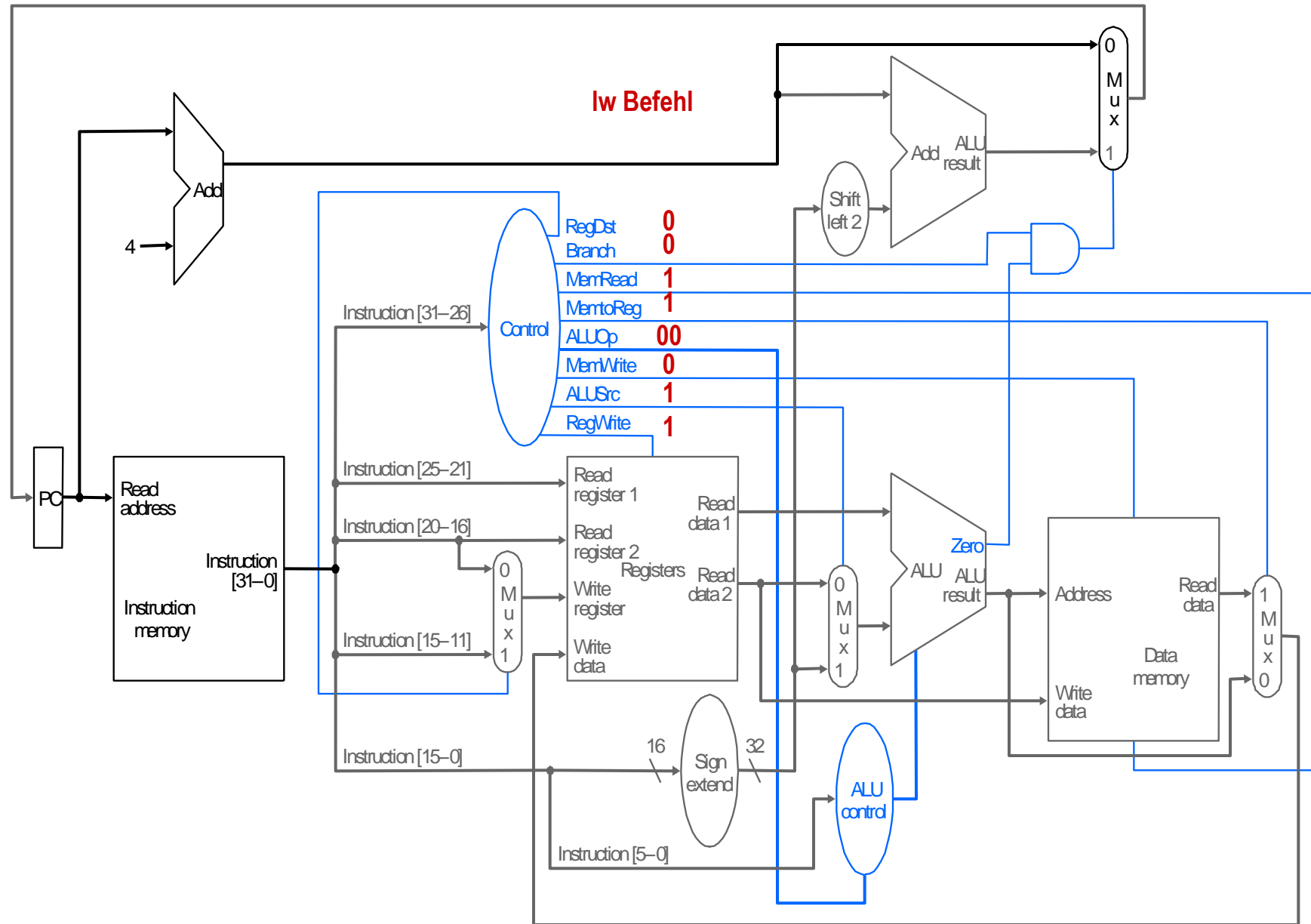


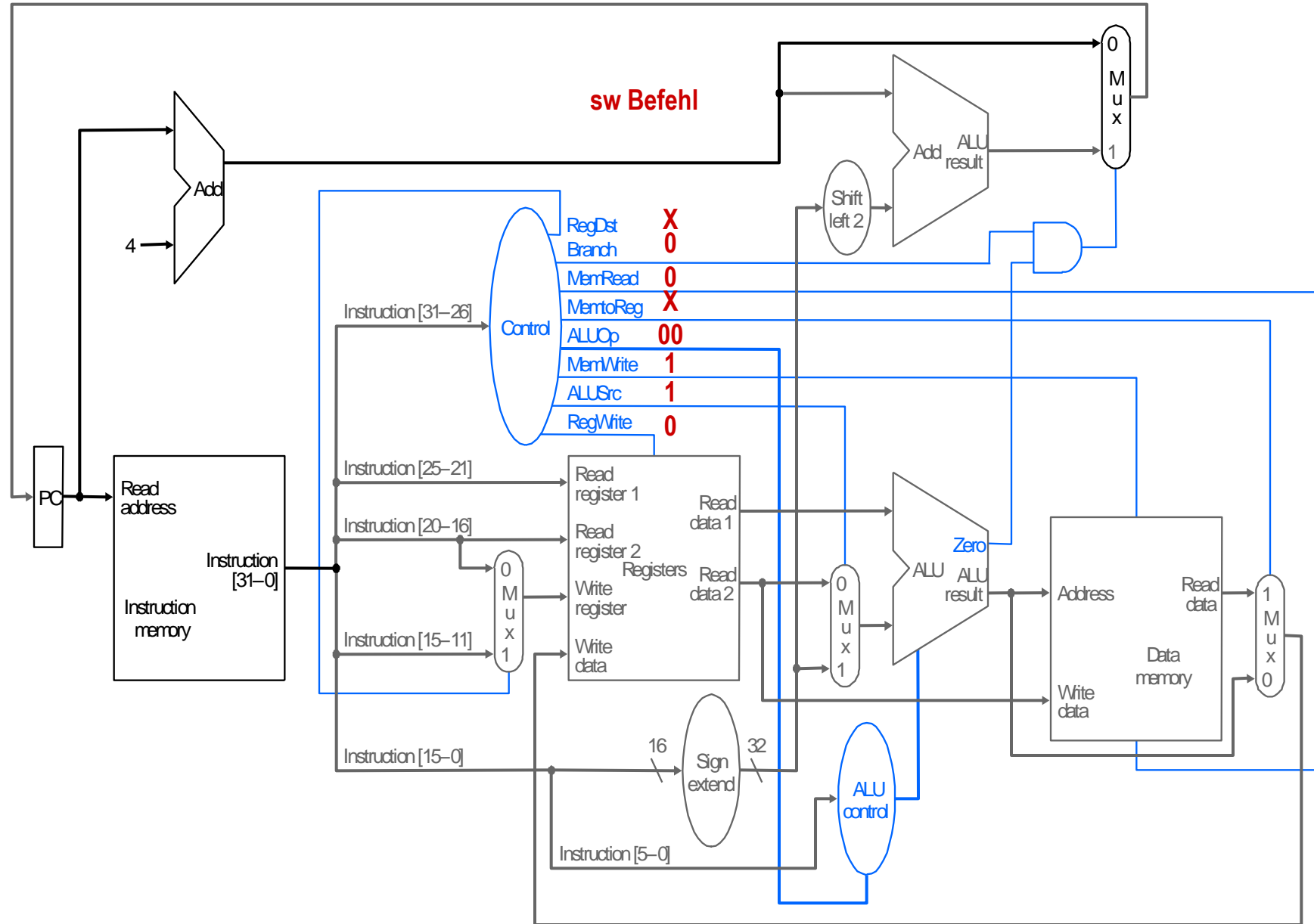
- wird durch eine Wahrheitstabelle beschrieben, welche auf Gatter abgebildet werden kann:

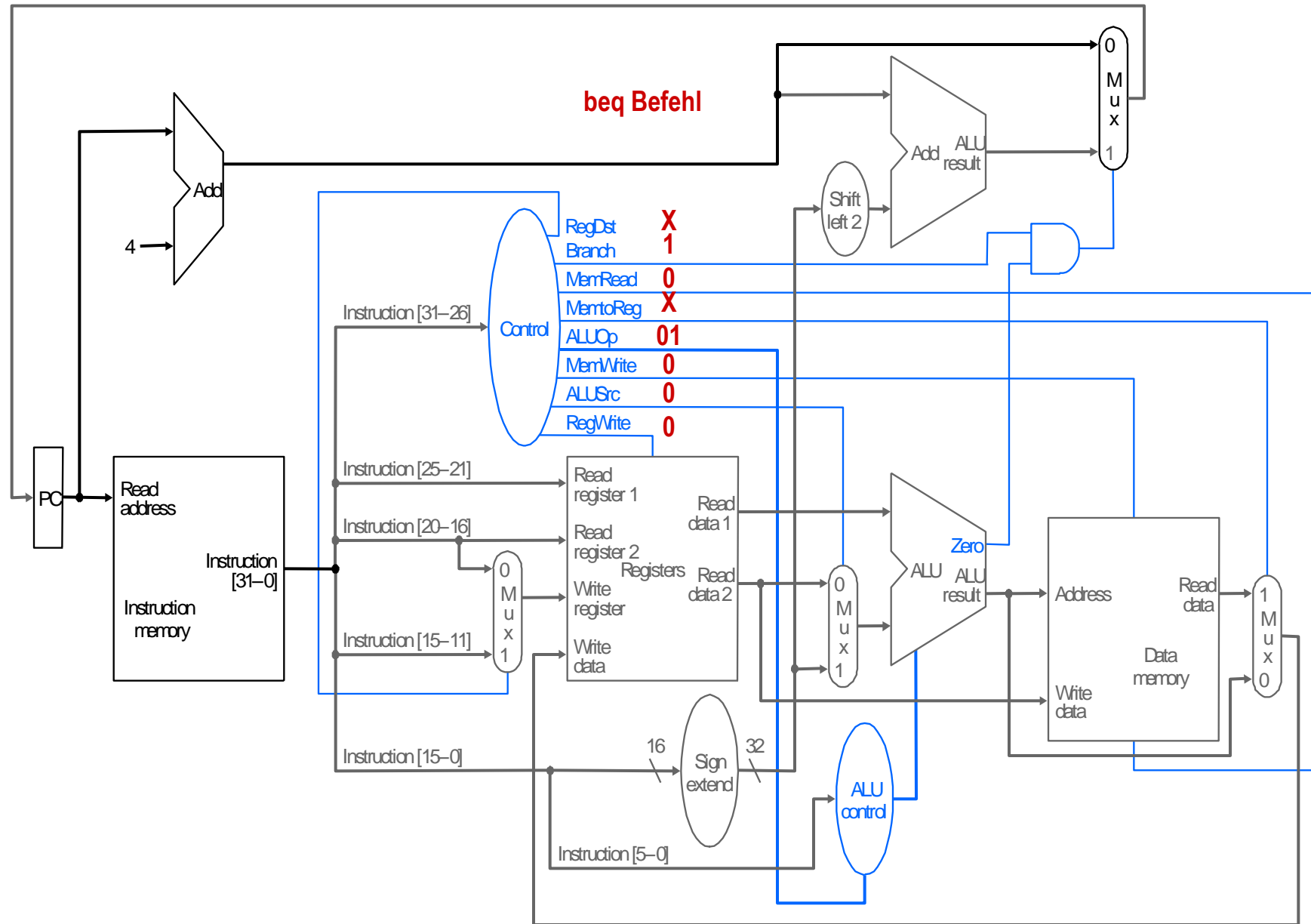
opcode	funct	ALUOp		Funct field						ALU control signals	Desired ALU action
		ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
lw/sw	NA	0	0	X	X	X	X	X	X	0010	add
beq	NA	0	1	X	X	X	X	X	X	0110	subtract
R-type	add	1	0	X	X	0	0	0	0	0010	add
R-type	sub	1	0	X	X	0	0	1	0	0110	subtract
R-type	and	1	0	X	X	0	1	0	0	0000	and
R-type	or	1	0	X	X	0	1	0	1	0001	or
R-type	slt	1	0	X	X	1	0	1	0	0111	slt











- Benutze “don't cares” wenn möglich, spezifiziere die Steuersignale für

– R-Typ Befehle

0	rs	rt	rd	shamt	func
---	----	----	----	-------	------

– lw

35	rs	rt	16-Bit Adresse
----	----	----	----------------

– sw

43	rs	rt	16-Bit Adresse
----	----	----	----------------

– beq

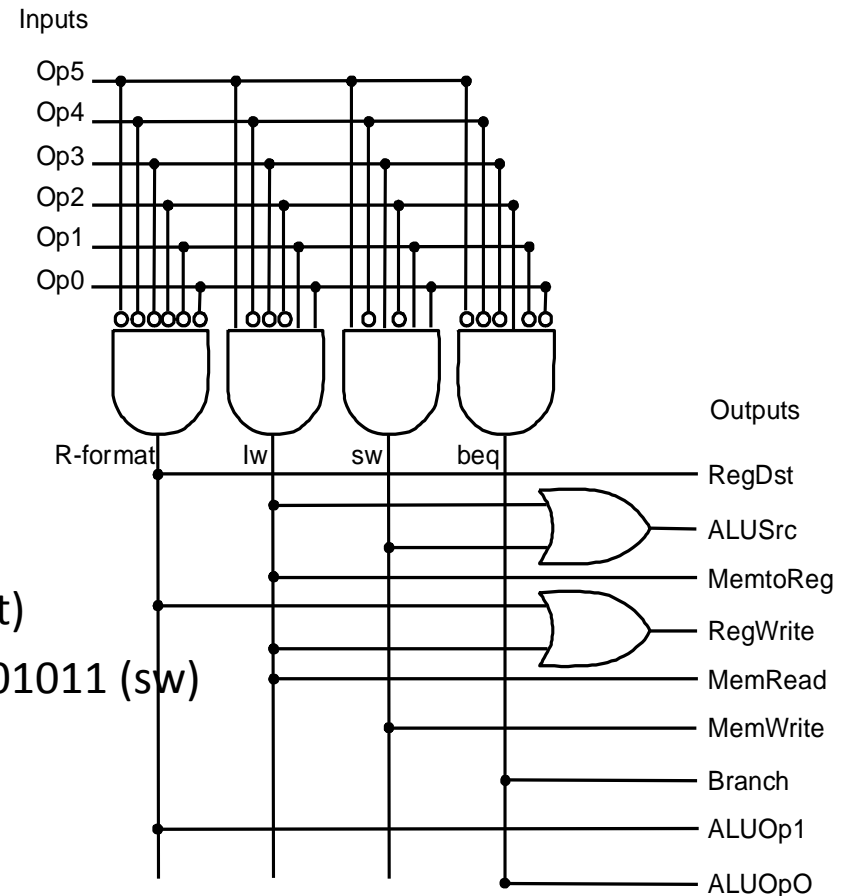
4	rs	rt	16-Bit Adresse
---	----	----	----------------

Befehl	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
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

- Wahrheitstabelle kann in **Programmable Logic Array (PLA, programmierbare logische Anordnung)** abbilden, welche die Steuersignale berechnet:

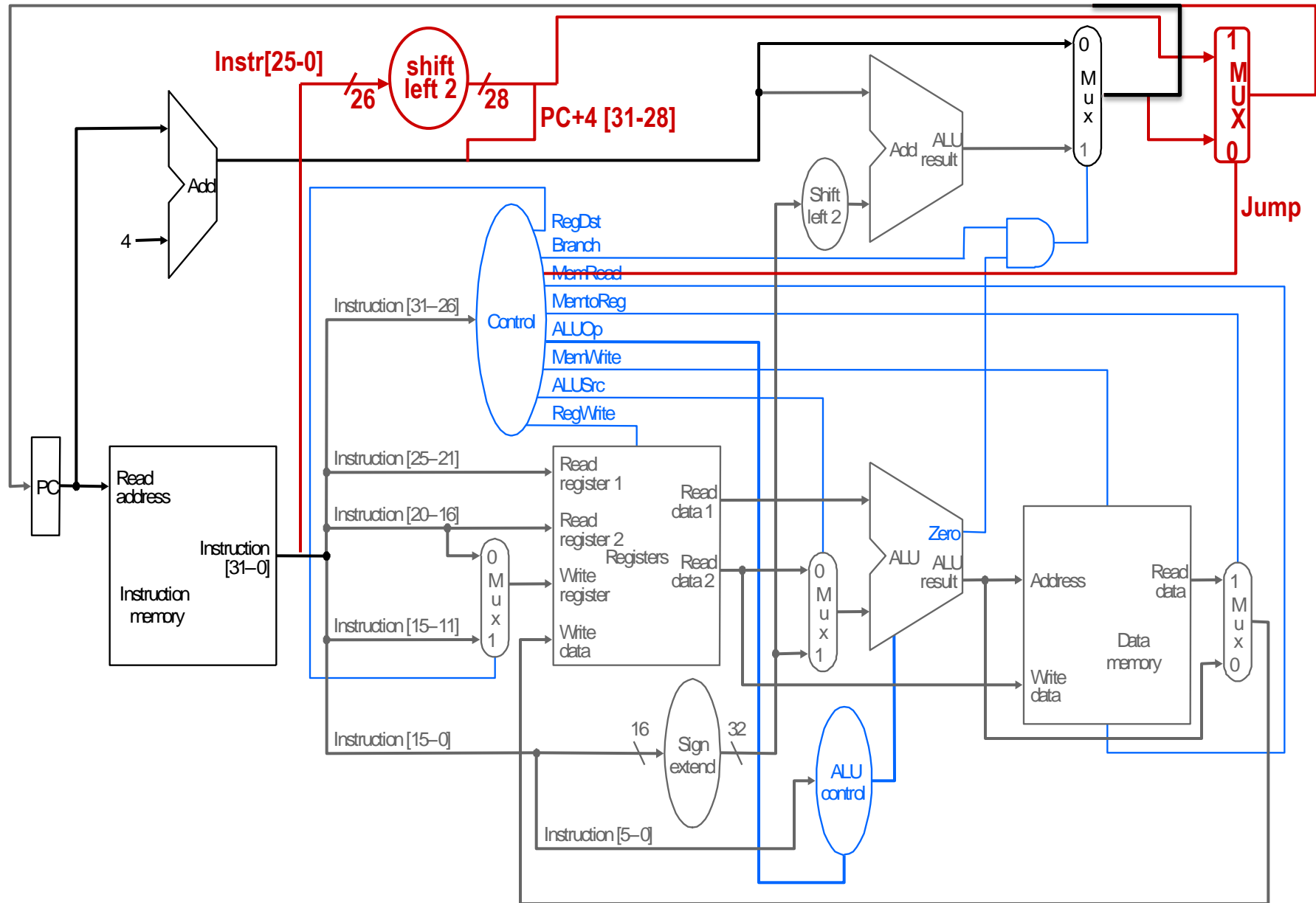
Befehl	Opcode
R-format	000000
lw	100011
sw	101011
beq	000100

- Beispiele:
 - RegDst=1 if Opcode=000000 (R-format)
 - ALUSrc=1 if Opcode=100011 (lw) or 101011 (sw)
 - usw.



- Wie müssen wir den Eintakt-Prozessor erweitern, um einen Sprung-Befehl zu implementieren?
- Sprung-Befehlsformat:

2	26-Bit Adresse
---	----------------
- Ziel-Adresse ist eine Verknüpfung von:
 - Höchstwertigsten 4 Bits von PC+4
 - 26-Bit Adresse
 - die Bits 00 (Wortadresse)
- Modifizierung des Eintakt-Prozessors:
 - Zusätzlicher MUX zum selektieren zwischen Sprungziel und (PC+4 oder branch Ziel)
 - Zusätzlicher links Shifter um 2 Stellen, da die Befehlsadresse eine Wortadresse ist und wir jedoch eine Byte Adresse brauchen
 - Zusätzliches Sprungsteuersignal welches aktiviert wird, wenn es sich um einen Sprungbefehl handelt



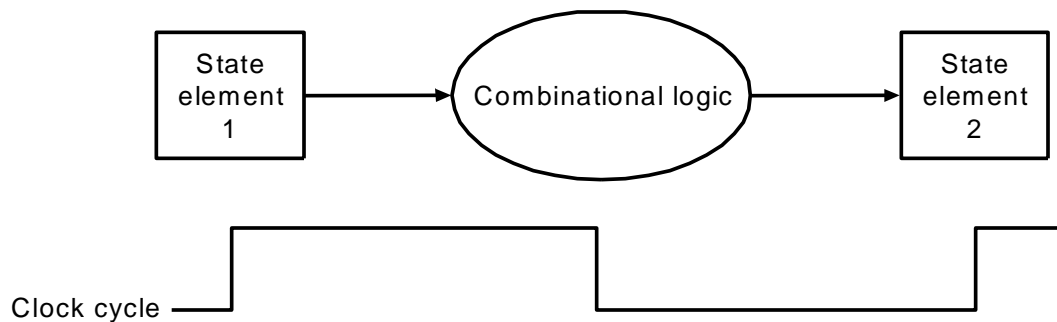
- Bestimmung der Steuersignale für den Sprung . Verwende “don’t cares” wenn möglich
 - **RegDst=X**; da j nicht zum Registerspeicher schreibt, kann **RegDst** X sein
 - **Branch=X**; für j wird der anderen Input des neuen MUX benötigt
 - **RegWrite=0**; j schreibt nicht in den Registerspeicher
 - **Jump=1**; selbstverständlich
 - **MemRead=0**; selbst wenn die geladenen Daten verworfen werden, soll kein cache miss, page fault oder andere üble Dinge (nächstes Kapitel) generierte werden
 - **Mem2Reg=X**; j schreibt nicht in den Registerspeicher
 - **ALUOp=XX**; unerheblich was die ALU für eine Operation ausführt; das Ergebnis wird verworfen
 - **MemWrite=0**; Befehl sollte nicht in den Speicher schreiben (Achtung: auch wenn die ALU Operation ausser Acht lassen wird, wird von ihr ein Ergebnis produziert
 - **ALUSrc=X**; ALU Ergebnis wird verworfen, ALU Operation und Input irrelevant
 - **RegWrite=0**; j schreibt nicht in die Registerspeicher

- Wir wollen den Befehl **seq** (*set-if-equal*) zum Befehlssatz des Eintakt-MIPS-Prozessors hinzufügen.
- Beispiel:

seq \$t1,\$t2,\$t3# \$t1 = (\$t2==\$t3)

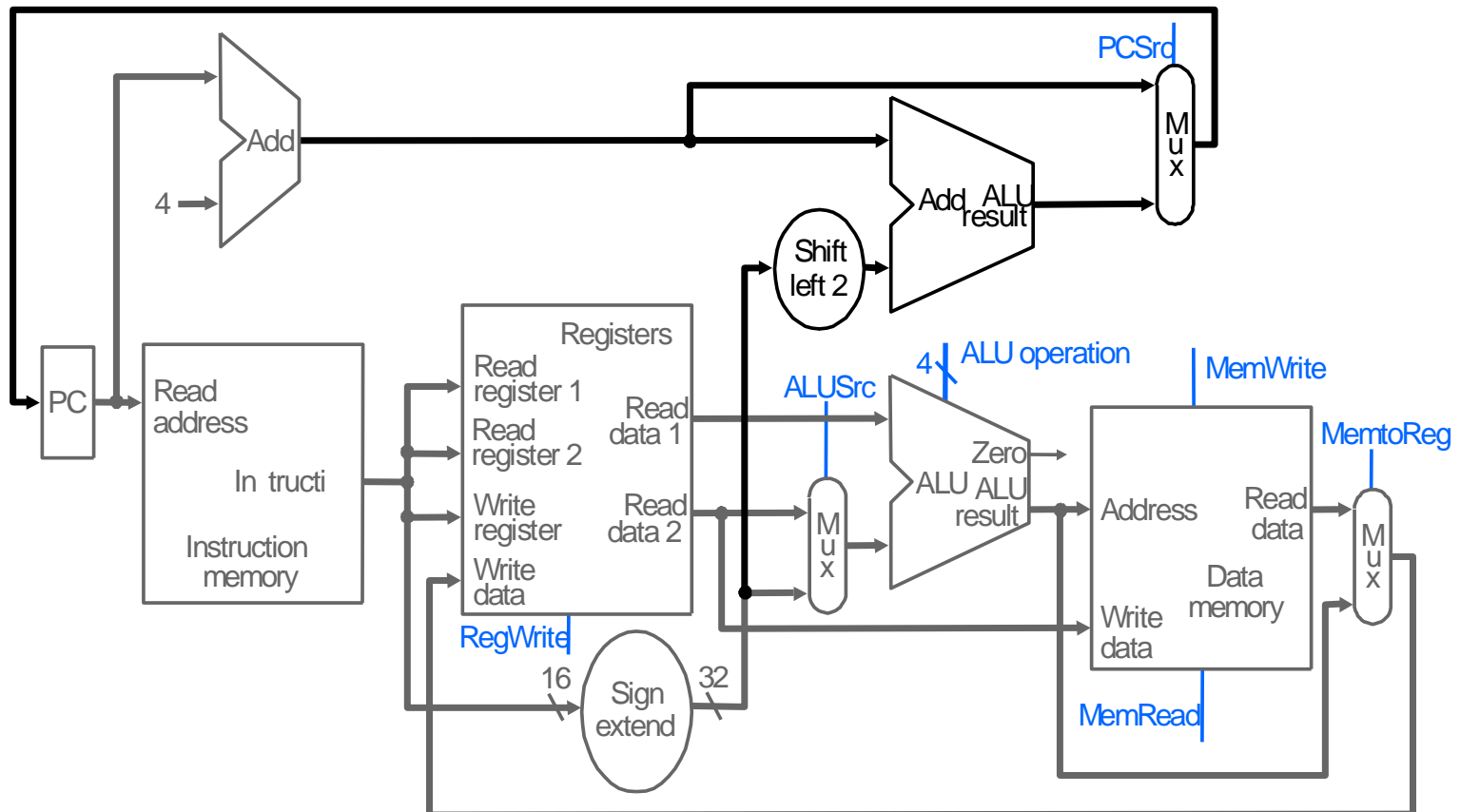
- Geben Sie ein passendes Befehlsformat an
- Modifizieren Sie den Datenpfad so, dass er den **seq**-Befehl ausführen kann
- Geben Sie alle Steuersignale (alte genauso wie neue) an

- Jede Steuerlogik ist kombinatorisch
 - Kontrollsignale sind nur abhängig von den aktuellen Befehl (opcode und möglichen funct-Code)
- Wir warten bis alles eingeschwungen ist und das Richtige getan wird
 - ALU produziert nicht gleich die “richtige Antwort”
 - wir benutzen Schreib-Signale **ob** und die Clock **wann** geschrieben wird
- Taktzeit wird durch den längsten Pfad bestimmt



Wir ignorieren Details wie “setup” und “hold times” !

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



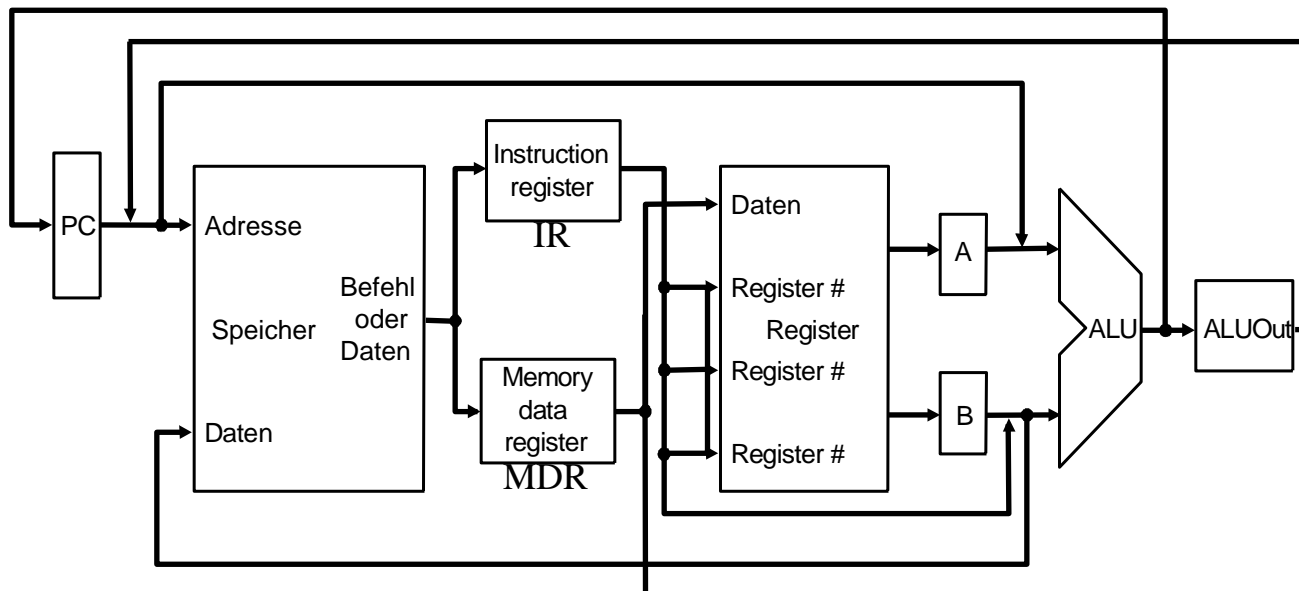
- Speicher (200ps), ALU & Addierer (100ps), Registerspeicher (50ps)

R-Typ	IM	RF	ALU	RF	400 ps	
lw	IM	RF	ALU	DM	RF	600 ps
sw	IM	RF	ALU	DM	550 ps	
beq	IM	RF	ALU	350 ps		
j	IM	200 ps				

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

- Wenn wir davon ausgehen, dass wir eine Clock mit variabler Taktzeit haben können, (was nicht realistisch ist, sondern ein Gedanken-experiment) benötigen die Befehle folgende Zeiten:
 - R-type: 400 ps
 - Load: 600 ps
 - Store: 550 ps
 - Branch: 350 ps
 - Jump: 200 ps
- kalkuliere “*durchschnittliche Befehlszeit*” in der Annahme, dass:
 - 25% loads, 10% stores, 45% R-type, 15% branches, 5% jumps
- $AIT = 0.25 \times 600 + 0.1 \times 550 + 0.45 \times 400 + 0.15 \times 350 + 0.05 \times 200 = 447.5 \text{ ps}$
- Implementierung mit einer Clock mit variabler Taktzeit: $600/447.5 = 1.34$ Mal schneller

- Eintakt-Prozessor Probleme:
 - Was wäre, wenn wir komplexere Befehl wie “floating point” Instruktionen hätten ?
 - Verschwendung des Chipfläche: **es werden keine Hardware Ressourcen gemeinsam genutzt**
- mögliche Lösung: nutze “geringere” Taktzeit, geringere Befehl benötigen eine unterschiedliche Anzahl von Takten,
Mehrtaktdatenpfad:





- Wir können einen Eintakt-Prozessor bauen, indem wir Hardware Element nutzen wie z.B. ALU, Registerdatei, Speicher, Flip-Flops,...
- Wir können einen **flankengesteuerte Taktverfahren** benutzen, um zu kontrollieren, **wann** die Daten geschrieben werden
- Wir benutzen **Multiplexer** um die Teile des Datenpfades zusammen zu fügen
- **Steuersignale** wählen die Operation aus und kontrollieren den Datenfluss (MUX Outputs)
- Steuersignale werden vom Opcode und dem (möglichen) funct-Feld berechnet
- Eintakt-Implementierung ist **einfach** aber eher **langsam** und **ineffizient**
- Als nächstes: Mehrzyklen-Implementierung



Optional

9 Steuersignale, generiert durch die Hauptsteuereinheit :

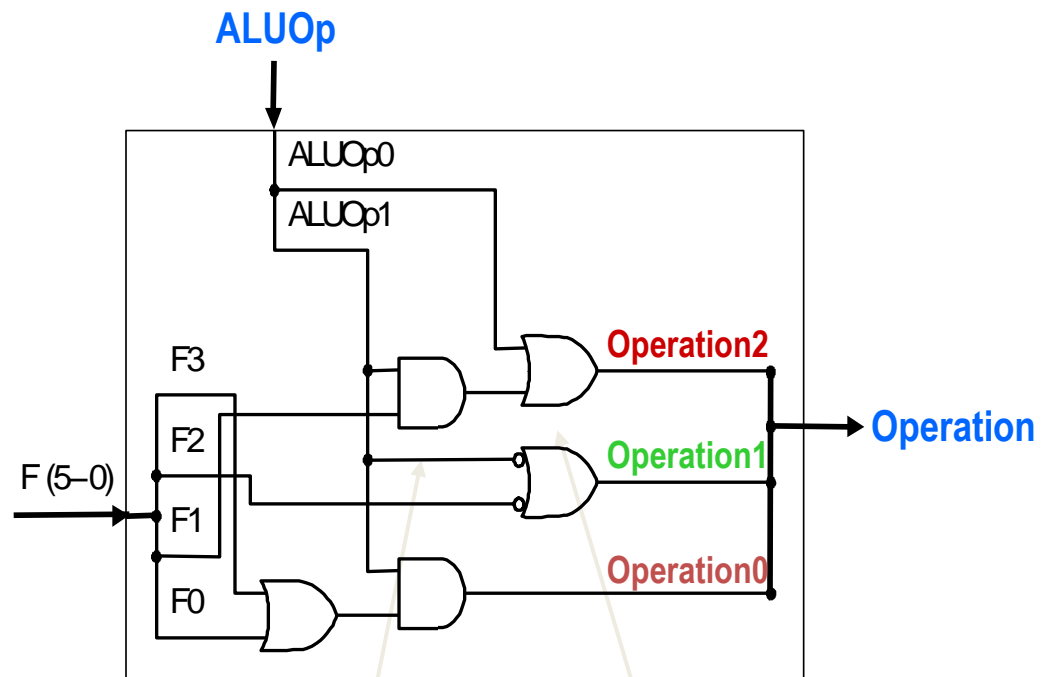
Befehl	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-Format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Diese Steuersignale werden direkt vom Opcode bestimmt:

Befehl	Opcode	
R-Format	000000	0
lw	100011	35
sw	101011	43
beq	000100	4

- Schaltung für den ALU Steuerblock:

Instr.	ALUOp	F(3-0)	ALU Kontroll-signale
lw/sw	00	XXXX	0010
beq	01	XXXX	0110
add	10	0000	0010
sub	10	0010	0110
and	10	0100	0000
or	10	0101	0001
slt	10	1010	0111



wenn (AluOp1 & F1) | AluOp0 dann
negierte es den 2^{ten} ALU Input (→ **beq**,
sub or **slt**)

Achtung: ALU Steuersignale haben ganz
links immer eine 0 und werden ignoriert

- passendes Befehlsformat:

- R-Typ

0	rs	rt	rd	shamt	func
---	----	----	----	-------	------

- Veränderungen des Datenpfads:

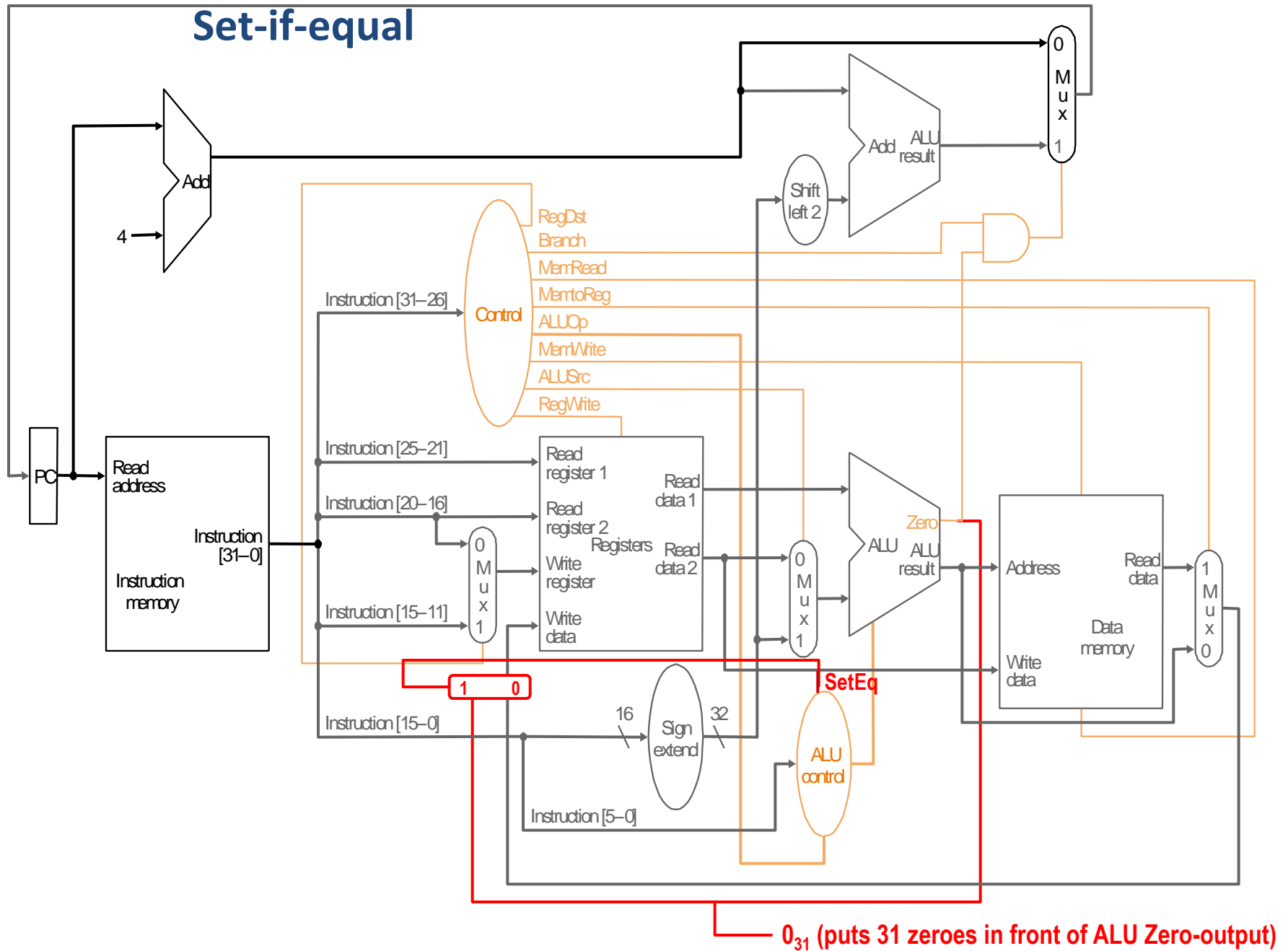
- Benutze Zero-Output der ALU und lass ALU subtrahieren
 - Steuersignal vom Zero-Output zum Schreib-Port des Registerspeichers
 - Siehe nächste Folie

- Steuersignale:

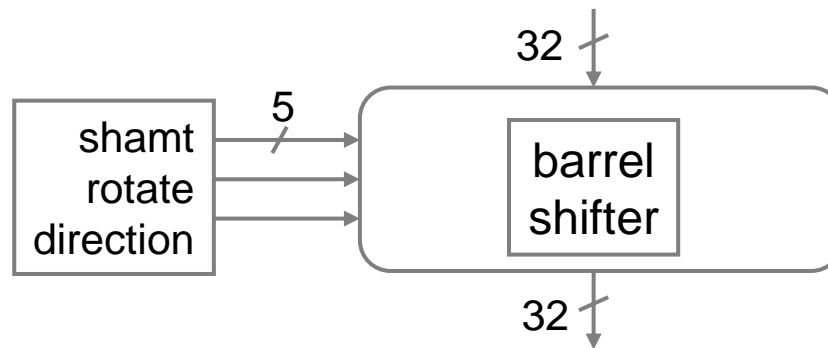
Befehl	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp	SetEq
R-format	1	0	0	1	0	0	0	10	1

- +ALU Kontrollblock muss modifiziert werden, so dass der die ALU Steuersignale 0110 (subtrahiert) für den Funktionscode des **seq** generiert

Set-if-equal



- Shift-Operationen werden im Allgemeinen nicht durch die ALU ausgeführt, sondern durch einen sogenannten barrel shifter.
- Einen 32-Bit barrel shifter hat einen 32-Bit Dateninput, einen 32-Bit Datenoutput und 7 Steuersignale. Ein Steuersignal (rotate) gibt an, ob Input rotiert oder geshiftet wird, ein weiteres (direction) bestimmt die Shiftrichtung und 5 Steuersignale (shamt/shift amount) bestimmen die Shift-Weite.
- Wir benutzen die folgenden Symbole für ein 32-Bit barrel shifter



- Modifiziere den Datenpfad, so dass es der sll Befehl implementiert werden kann. Das Befehlsformat von sll rd,rt,shamt ist:

6	5	5	5	5	6
0	0	rt	rd	shamt	0

- Gebe die Steuersignale (alte genauso wie neue) an