

Rechnerorganisation

Einstein-Prof. Dr.-Ing. Friedel Gerfers



Kapitel 5: Der Eintaktprozessor (Single-Cycle-Processor)

Überblick des bisherigen Stoffs

- Welche Information versteht ein Rechner?
 - Zahlendarstellung
 - Maschinensprache
- Wie kommt man von Hochsprachen (C, C++, Java...) zu dieser Darstellung?
 - MIPS Assembler
 - Maschinensprache
- Wer verarbeitet die Daten und berechnet Ergebnisse?
 - ALU

Frage dieser Vorlesung:

- Wie wird Maschinensprache in Hardware ausgeführt?
 - Datenpfad + Steuerung

Datenpfad und Steuerung /1

- **Datenpfad (*datapath*)**

- Hardwarekomponenten (ALU, Counter wie PC), die Operationen ausführen und deren Verbindungen durch Register festlegen
 - **Muskeln** des Prozessors

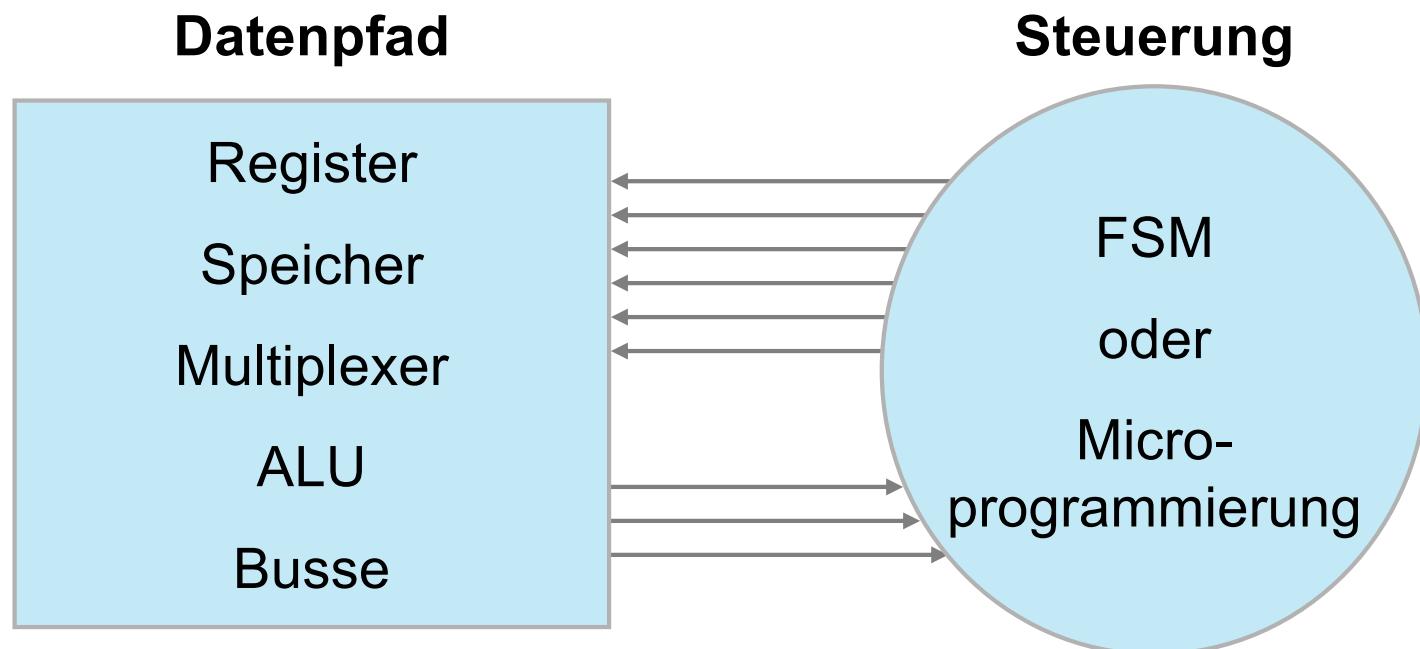


- **Steuerung (*control*):**

- Teil des Prozessors
 - Steuert den Ablauf der Befehlsverarbeitung mithilfe von der Steuersignalen des Datenpfads
 - **Gehirn** des Prozessors



Datenpfad und Steuerung /2



Ziele

- Nach diesem Kapitel sind Sie in der Lage
 - einen Datenpfad zu entwerfen, der einen Teil des MIPS-Befehlssatzes implementiert
 - die Steuersignale anzugeben, die für die Ausführung bestimmter Befehle gebraucht werden
 - einen Befehl zu dem unterstütztem Befehlssatz hinzuzufügen
 - Hardware zu entwerfen, welche die Steuersignale für verschiedene Befehle berechnet
 - zu erklären, warum Eintakt-Implementierungen ineffizient und warum Mehrzyklen-Implementierungen besser sind



Gliederung

- ALU-Erweiterung
- Datenpfad des Eintaktprozessors
- Steuerung des Eintaktprozessors
- Leistung des Eintaktprozessors

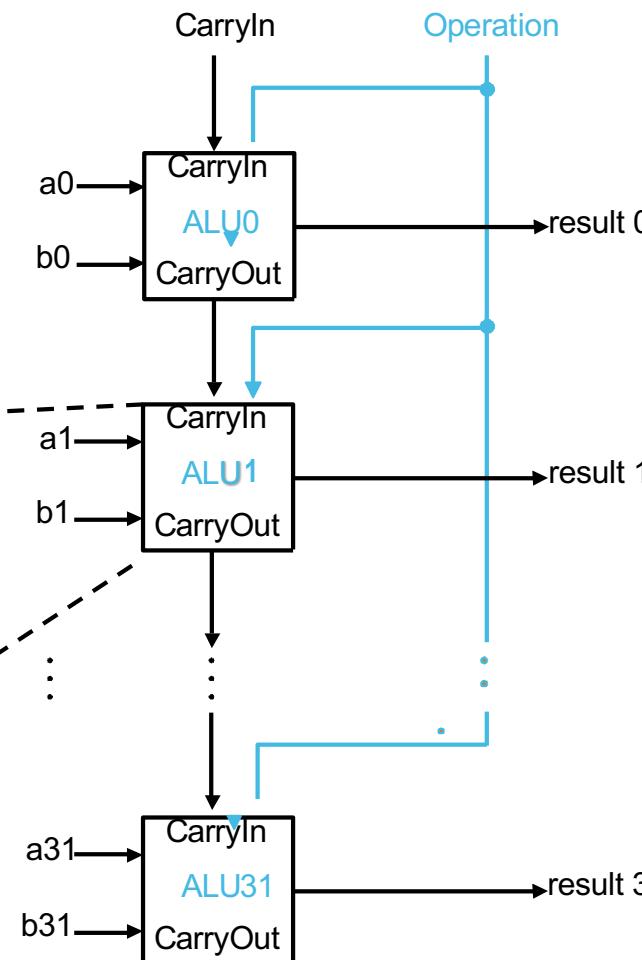
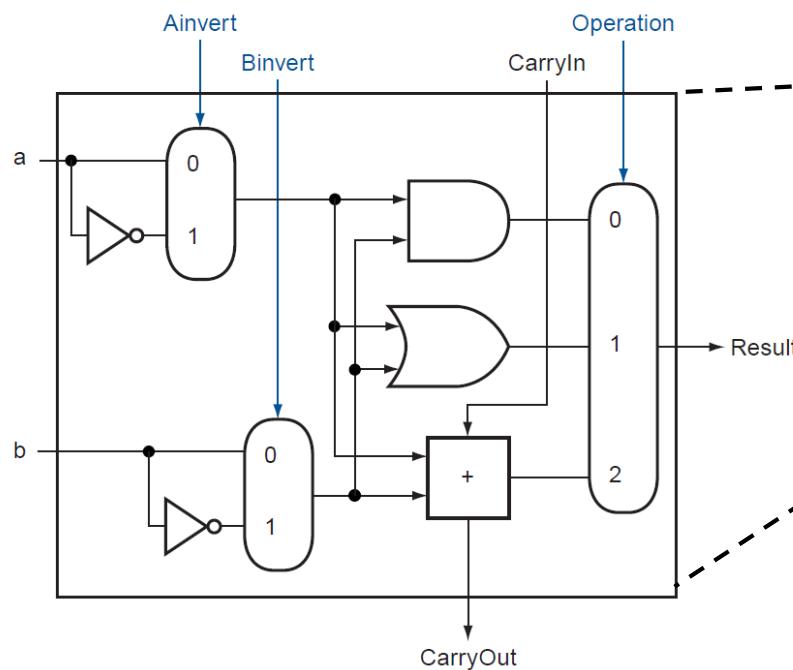
Der Prozessor in dieser VL wird **Eintaktprozessor** genannt, weil jeder Befehl in einem Taktzyklus ausgeführt wird

Eine einfache MIPS-Implementierung

- Wir sehen uns nun eine **Implementierung des MIPS** an
- Diese ist **vereinfacht**, unterstützt nur folgende Befehle:
 - Speicherreferenz-Befehle : **lw, sw**
 - arithmetisch-logische Befehle: **add, sub, and, or, slt**
 - Kontrollfluss-Befehle: **beq, j**
- Generelle Implementierungserweiterung damit folgende Funktionen realisiert werden ...
 - Sende Befehlszähler an den Speicher, der den Code enthält
 - Hole Befehl aus diesem Speicher
 - Lese Register
 - Dekodierte den Befehl, um zu entscheiden was zu tun ist

ALU-Erweiterung

- Die ALU muss erweitert werden, damit die Instruktionen **slt** und **beq** auch implementiert werden können
- Bisherige ALU:



Erweiterung der ALU um slt und beq

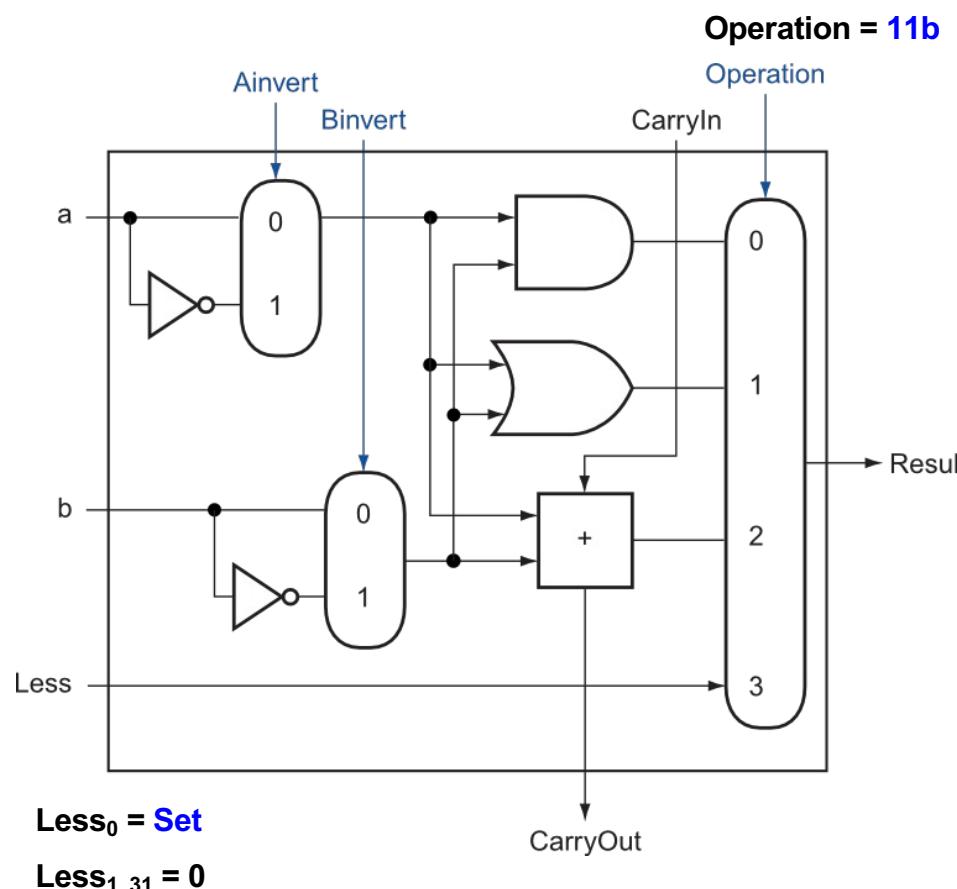
- Für s_{lt} (Set Less Than) wird getestet, ob der Inhalt eines Registers < (kleiner) dem Inhalt eines anderen ist (R-Reg. Format)
 - s_{lt rd,rs,rt} produziert rd == 1, wenn rs < rt, ansonsten rd == 0
 - Verwende Subtraktion: (a-b) < 0 → a < b
 - D.h.: (rs - rt) < 0 → rs < rt
- Für beq (Branch On Equal) wird getestet, ob der Inhalt eines Registers = (gleich) dem Inhalt eines anderen ist (I-Reg. Format)
 - Verwende wieder Subtraktion: (a-b) = 0 → a = b
 - D.h.: (rs - rt) = 0 → rs = rt
- Bedenke: Subtraktion → 2-Komplement + add

Erweiterung der ALU um s1t /1

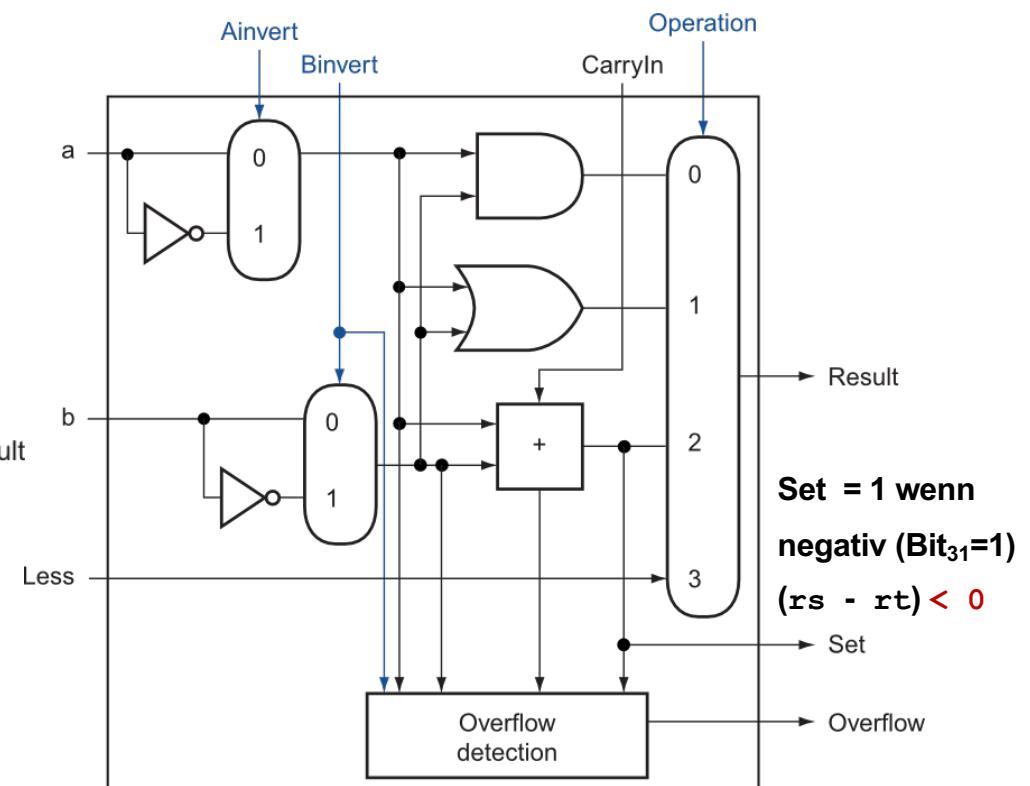
- Ein extra ALU Input *Less* wird hinzugefügt, sowie für ALU_{31} ein Ausgang *Set*
- Input *Less* wird auf 0 gesetzt für alle 1-Bit ALUs **außer ALU_0**
- Der ALU_0 Input *Less*: Wird mit dem *Set*-Output der ALU_{31} verbunden
 - Set-Output ALU_{31} = **Vorzeichenbit** des Ergebnis der Subtraktion
 - Für die Ausführung der Subtraktionsfunktion muss einer der Eingänge a (oder b) noch invertiert werden
- *Less*-Input wird ausgewählt, wenn ***Operation = 11***

Erweiterung der ALU um slt /2

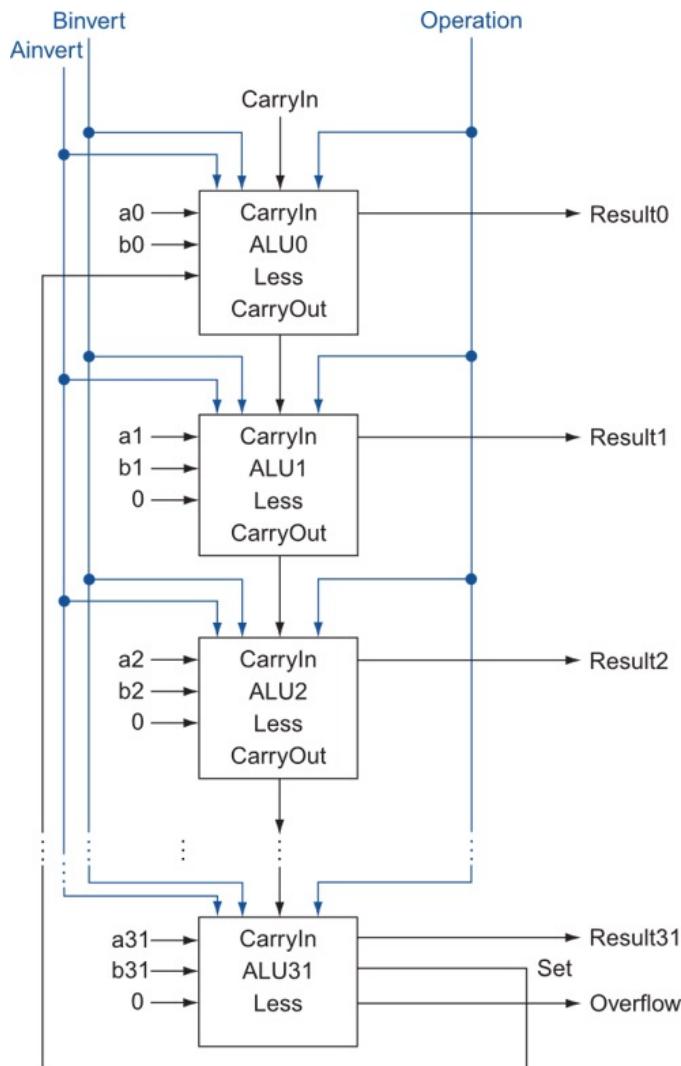
ALU_{0..ALU₃₀}:



ALU₃₁:



Erweiterung der 32-Bit ALU um slt



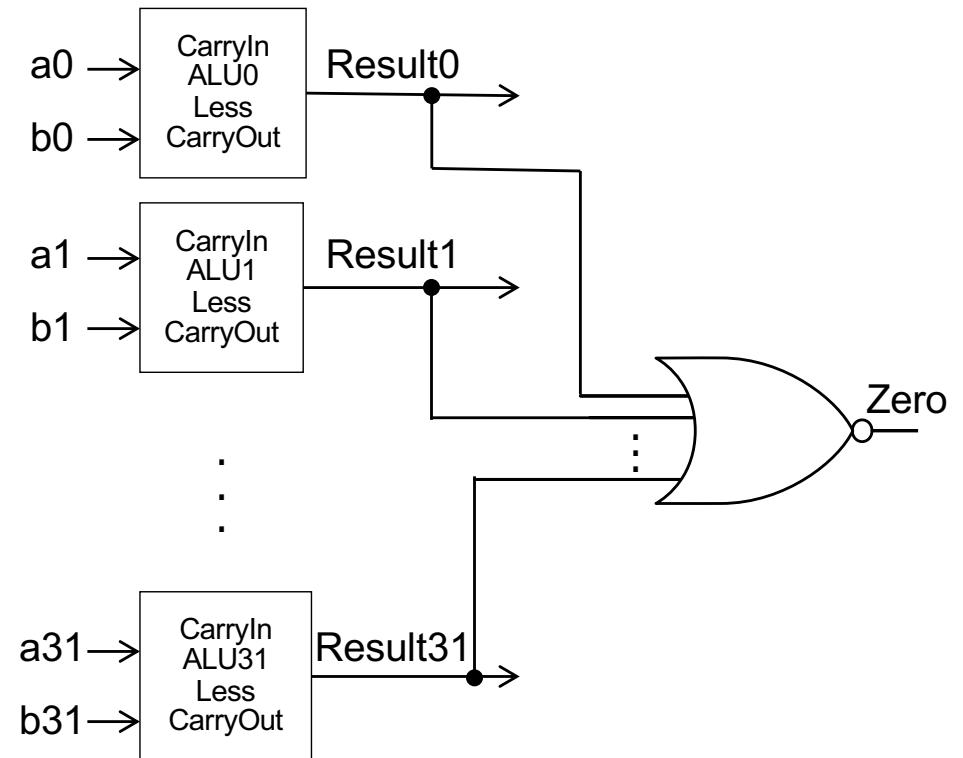
ALU-Steuersignale für **slt**:

- **Ainvert:** 0
- **Binvert:** 1
- **CarryIn:** 1
 - subtrahiere $b_{31:0}$ von $a_{31:0}$
- **Operation:** 11
 - selektiere *Less*-Input

Erweiterung der ALU um beq

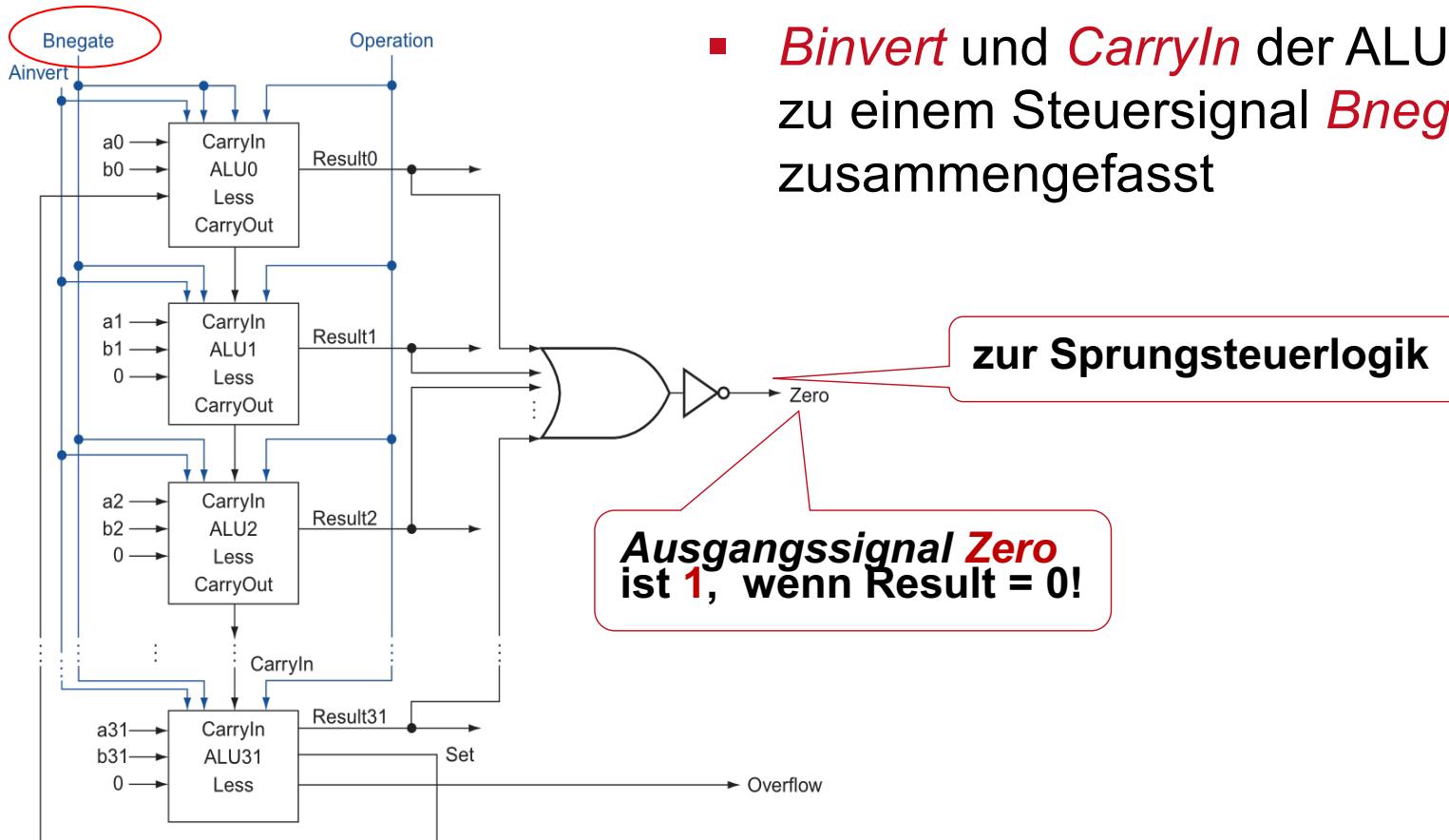
`beq rs,rt,offset # if (Reg[rs]==Reg[rt]) PC += 4+4xoffset`

- Müssen testen ob $\text{Reg}[rs] == \text{Reg}[rt]$
- Benutze Subtraktion:
 - $\text{Reg}[rs] == \text{Reg}[rt] \leftrightarrow \text{Reg}[rs] - \text{Reg}[rt] == 0$
 - $\text{Reg}[rs] == \text{Reg}[rt] \leftrightarrow \text{Result}_0 + \text{Result}_1 + \dots + \text{Result}_{31} == 0$
 - Gleiche Steuersignale wie Subtraktion:
 $\text{Ainvert} = 0;$
 $\text{Binvert} = \text{CarryIn}=1;$
 $\text{Operation} = 10 \text{ (Adder-Output)}$



NUR wenn alle Eingänge des NOR 0 sind liefert das NOR eine 1

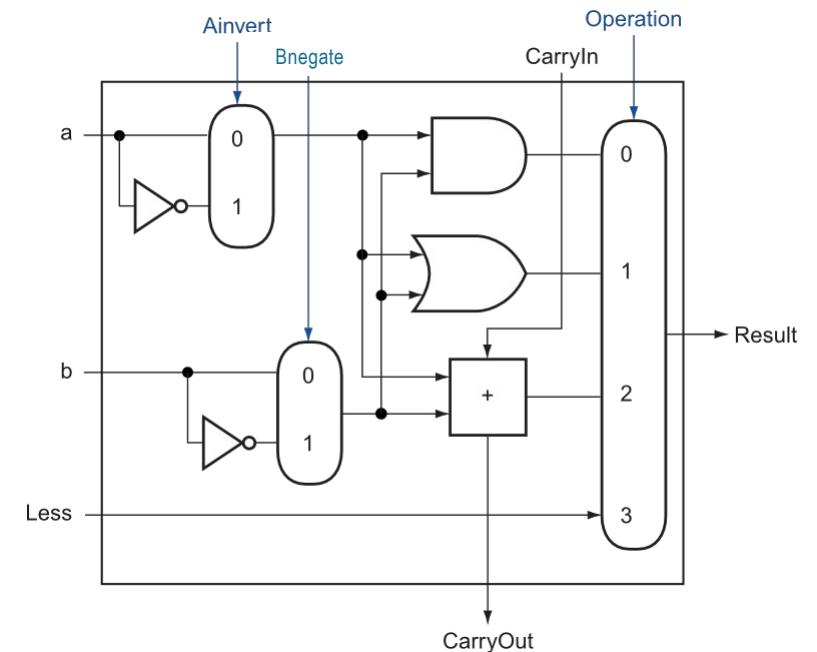
Erweiterung der ALU um beq



ALU Steuersignale

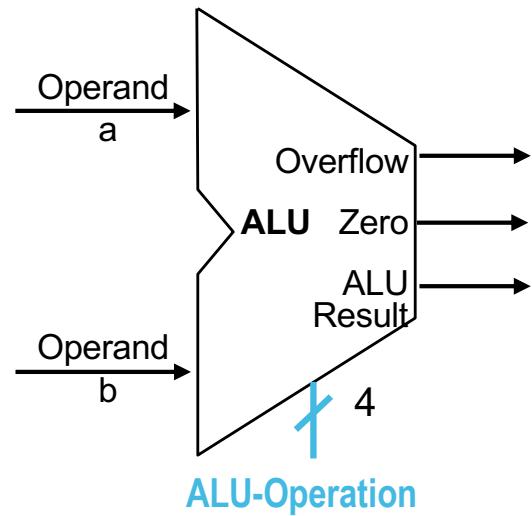
- Steuersignale um ALU-Operation zu bestimmen:

Gewünschte ALU-Aktion	Ainvert	Bnegate	Operation
and	0	0	00
or	0	0	01
add	0	0	10
sub	0	1	10
slt	0	1	11
nor	1	1	00
nand	1	1	01



- 4 Bit Steuersignal notwendig
- Steuersignale haben festgelegte Bedeutung!

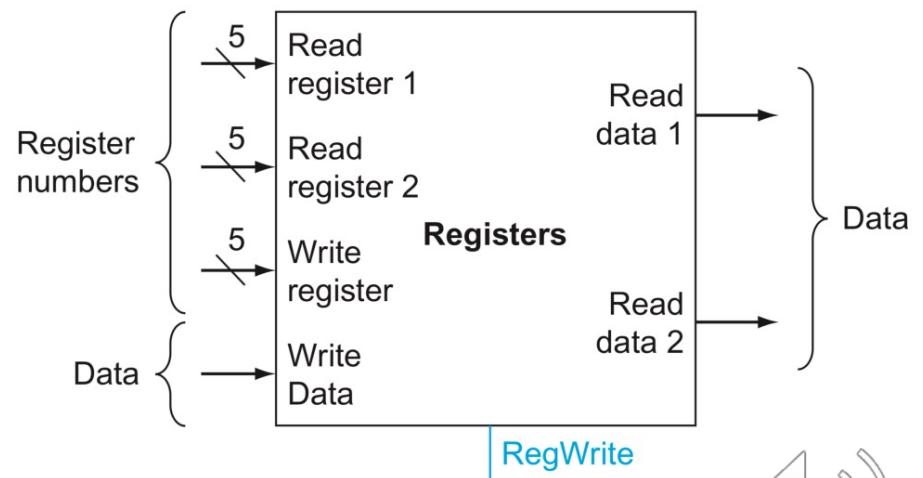
ALU Blockschaltbild



ALU-Operation Steuersignale	ALU Operation
0000	and
0001	or
0010	add
0110	sub
0111	slt
1100	nor

Rückblick: Registersatz (*register file*)

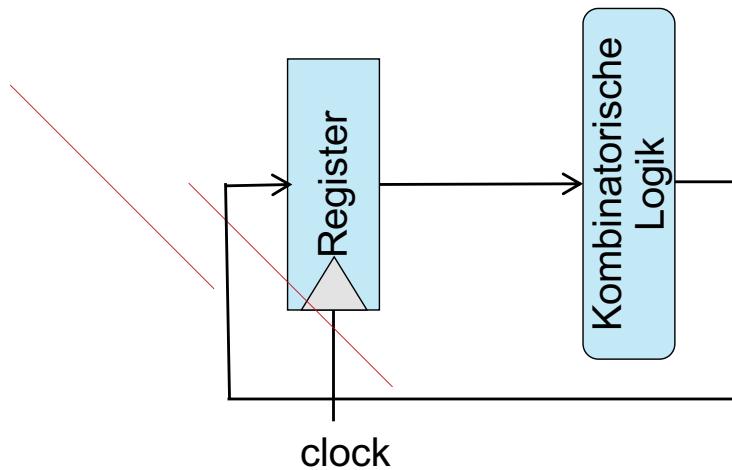
- Registerspeicher:
- Funktion:
 - LeseDaten1 = Register[LeseRegister1] # immer
 - Lesedaten2 = Register[LeseRegister2] # immer
 - Register[SchreibeRegister] = SchreibeDaten
 - nur wenn Steuersignal **RegWrite** (== 1)!
 - am Ende des Taktzyklus!
(bei fallender Taktflanke)
 - also wie z. B. beim **add** Befehl
 $Reg[rd] = Reg[rs]+Reg[rt]$ wird das Ergebnis in $Reg[rd]$ abgelegt



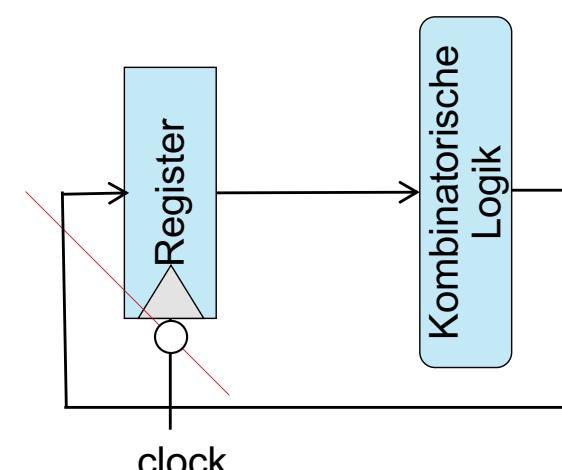
Flankengesteuertes Taktverfahren

- Flankengesteuertes Taktverfahren (*edge triggered clocking methodology*): alle Zustandsänderungen erfolgen **ausschließlich** während einer (fallender oder steigender) Taktflanke
 - Damit kann ein Schaltwerk in einem Taktzyklus gelesen und geschrieben werden, ohne dass es zu undefinierten Datenwerten führt

steigende Taktflanke



fallende Taktflanke



Rückblick: MIPS-Befehlsformate

- Jetzt wirklich an dem Punkt eine Implementierung des MIPS anzusehen
- Vereinfacht, nur folgende Befehle:
 - 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

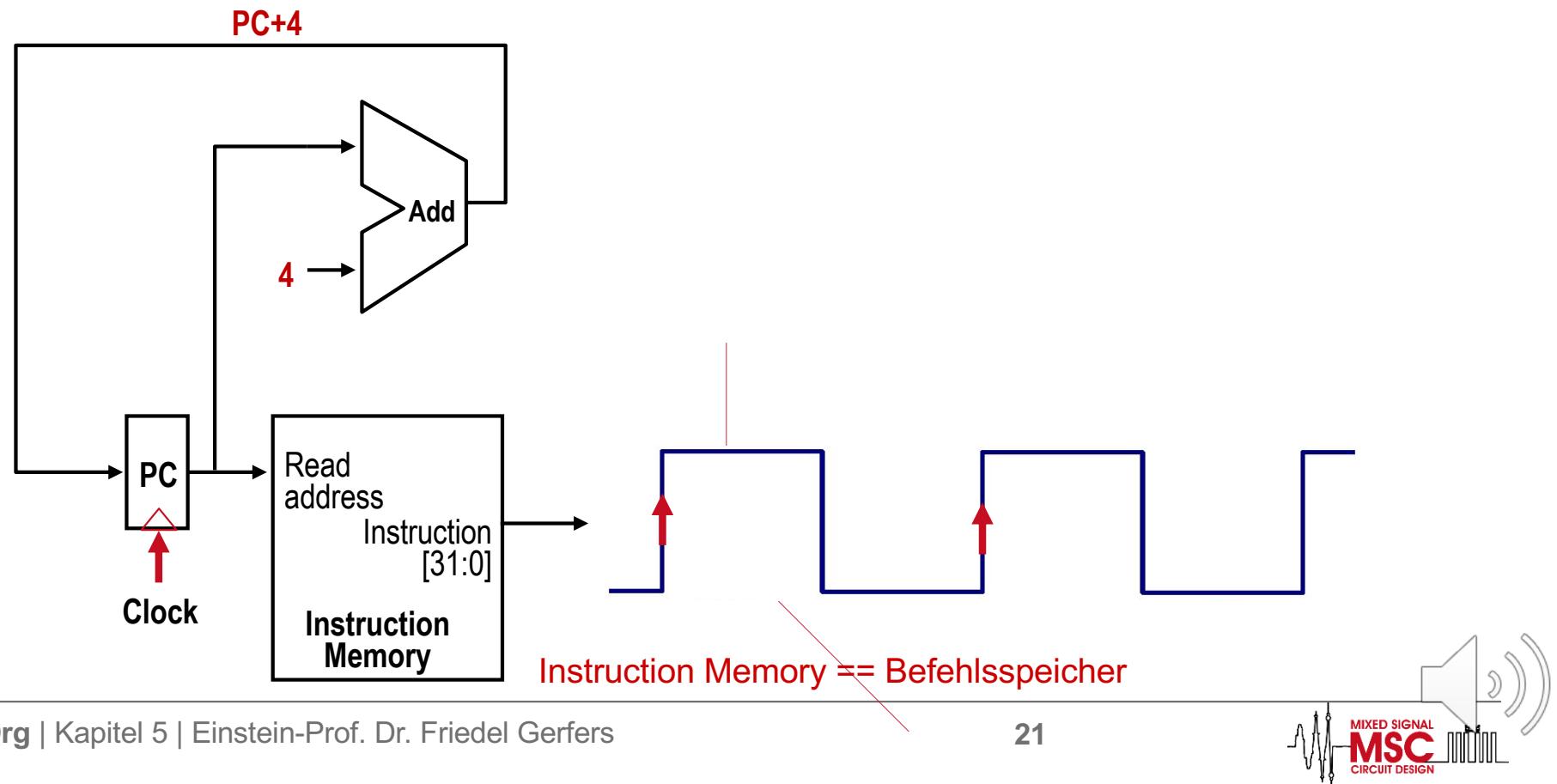
I-Format: **addi \$1,\$2,100;**

lw \$1,40(\$2); beq \$1,\$2,100

J-Format: **j 100**

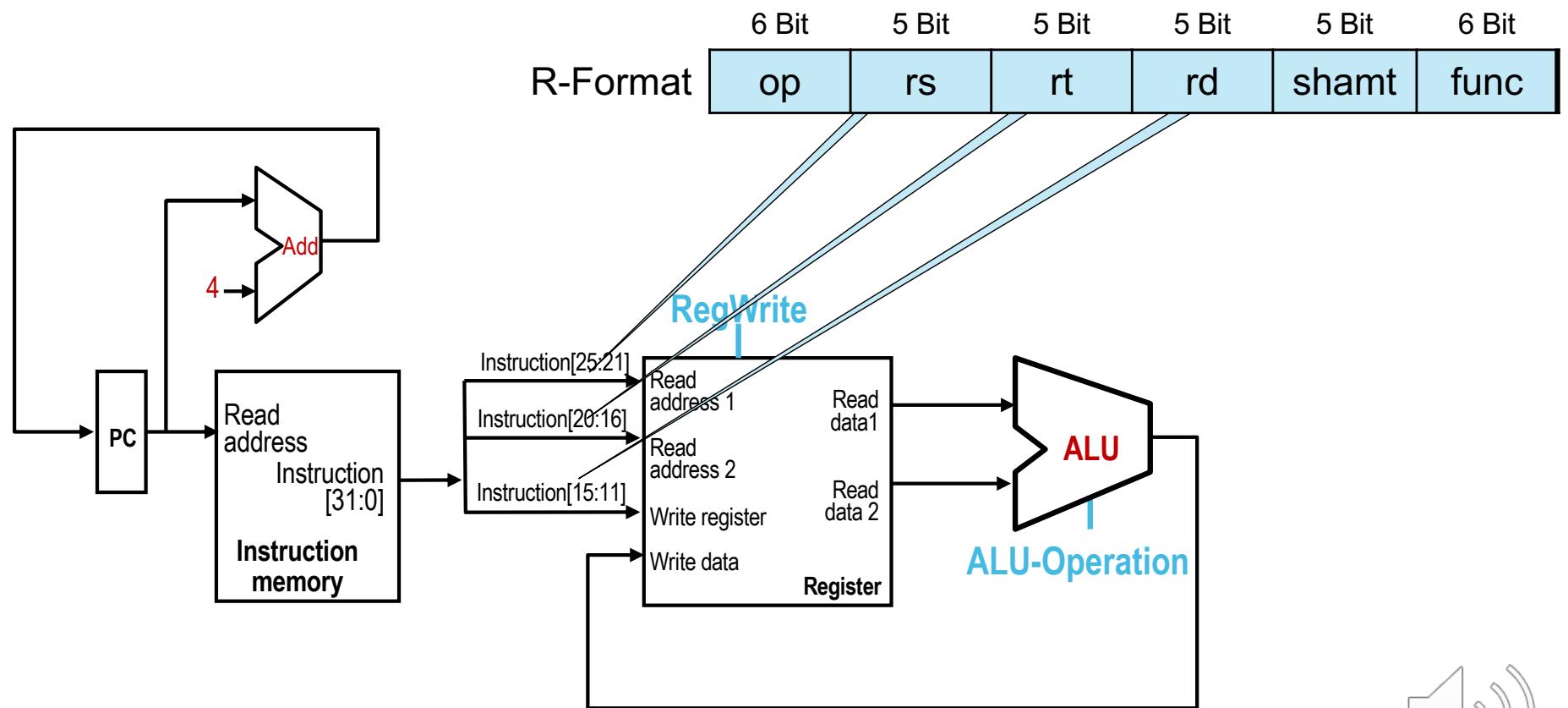
1. Schritt: lade den Befehl

- Welche HW-Bausteine benötigen wir, um den nächsten Befehl zu laden (und die nächste Befehlsadresse zu berechnen)?
- $PC + 4 \rightarrow$ nächste Befehlsadresse berechnet
- Neuer PC wird zur fallenden Taktflanke gespeichert



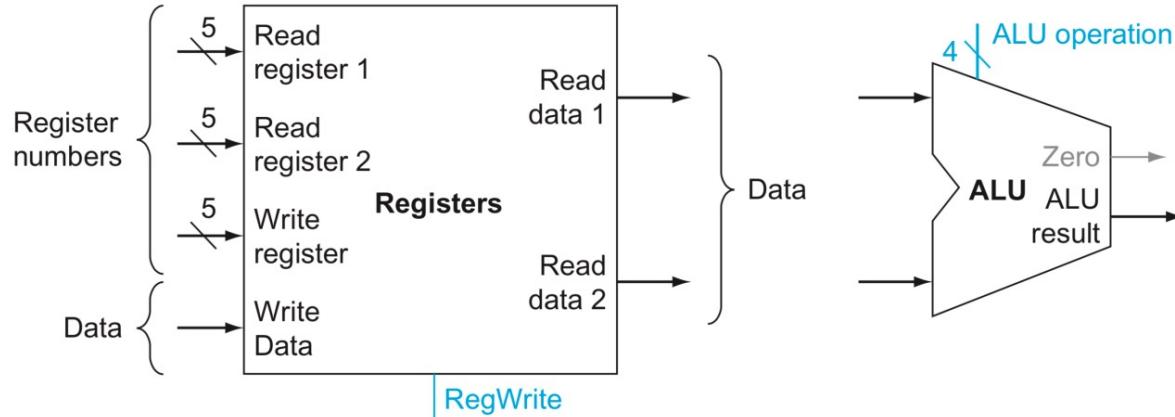
R-Type Befehle

- Was brauchen wir nun, nachdem der Befehl geholt wurde, um R-Typ Befehle zu realisieren?



Steuersignale

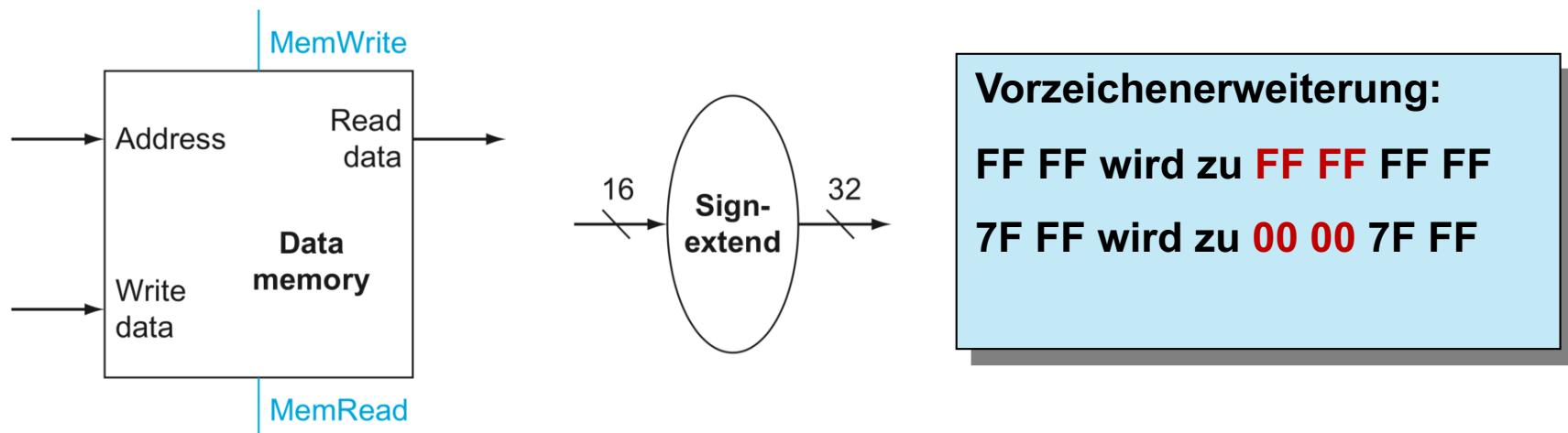
- Registerspeicher hat das Steuersignal **RegWrite**
 - Einige Befehle schreiben **nicht** zum Registersatz
- ALU hat 4 Steuersignale, die die ALU Funktion spezifizieren
 - Bestimmt vom **Opcode** und **manchmal auch** vom Funktionsfeld



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

Speicherzugriffsbefehle /1

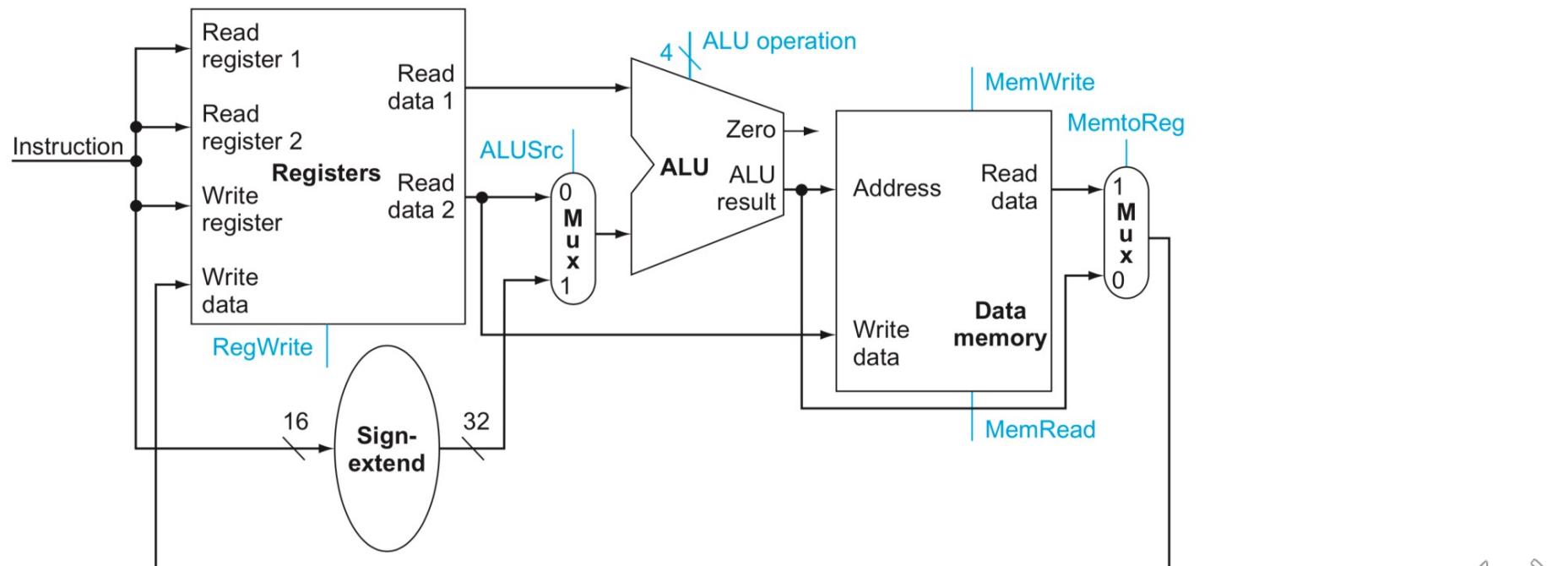
- **lw** und **sw** verwenden die ALU, um die Adresse zu berechnen
- Was brauchen wir noch zusätzlich zum Registersatz (Data Memory) und ALU, um **lw** und **sw** zu implementieren?



Speicherzugriffsbefehle /2



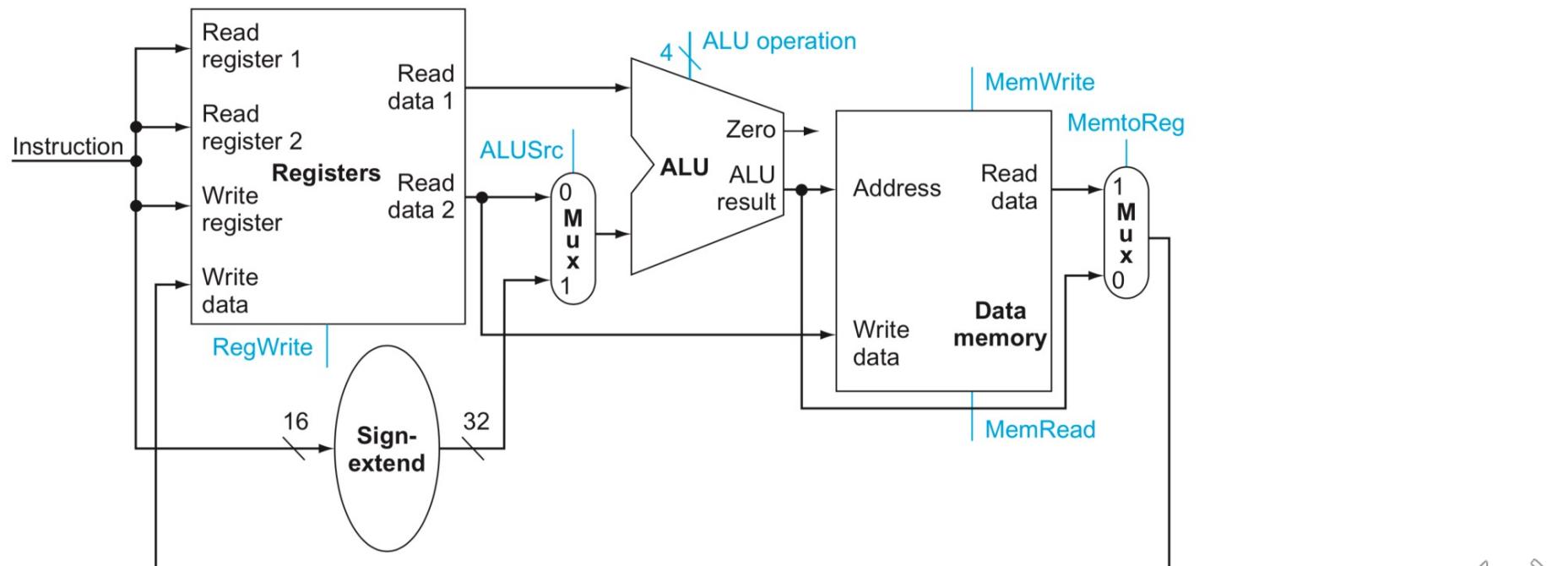
- Teil des Datenpfads für Lade- und Speicherbefehle:



Speicherzugriffsbefehle /3



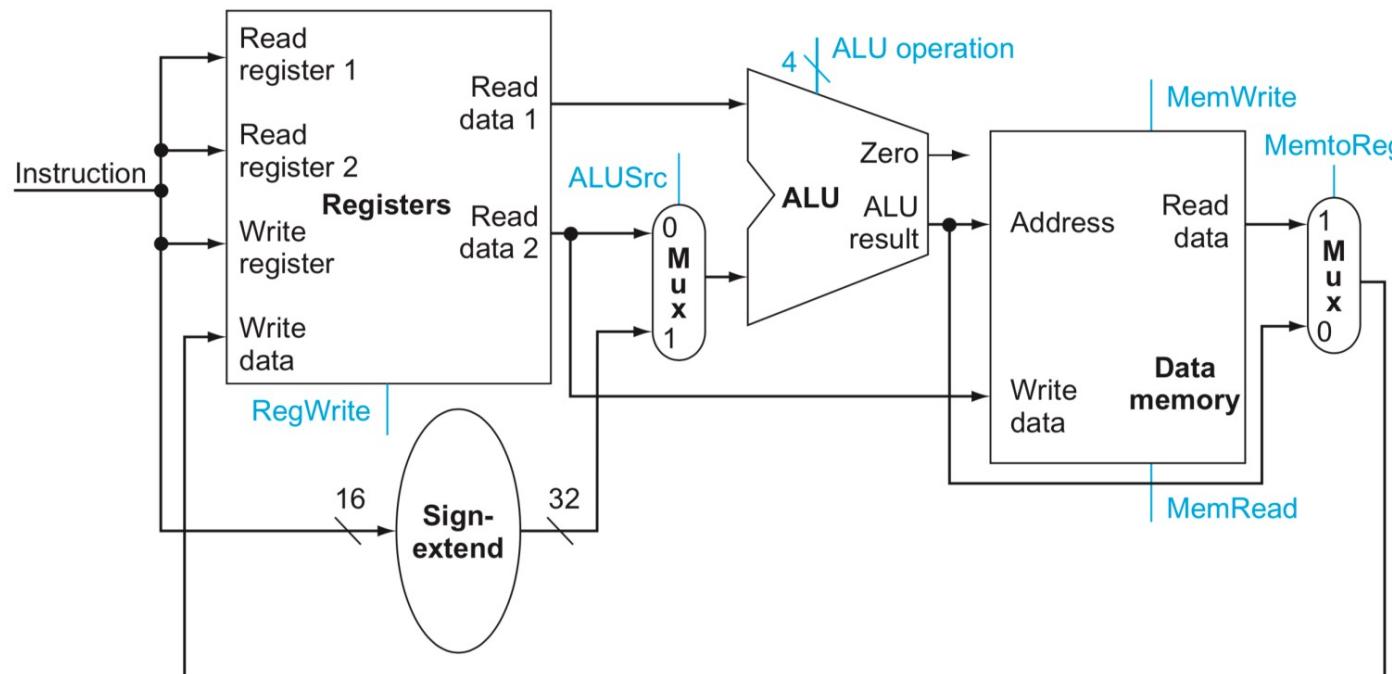
- ALU verwendet Registeroperanden für arithmetische Anweisungen oder einen vorzeichenerweiterten Immediate für **lw** und **sw**



Speicherzugriffsbefehle /4



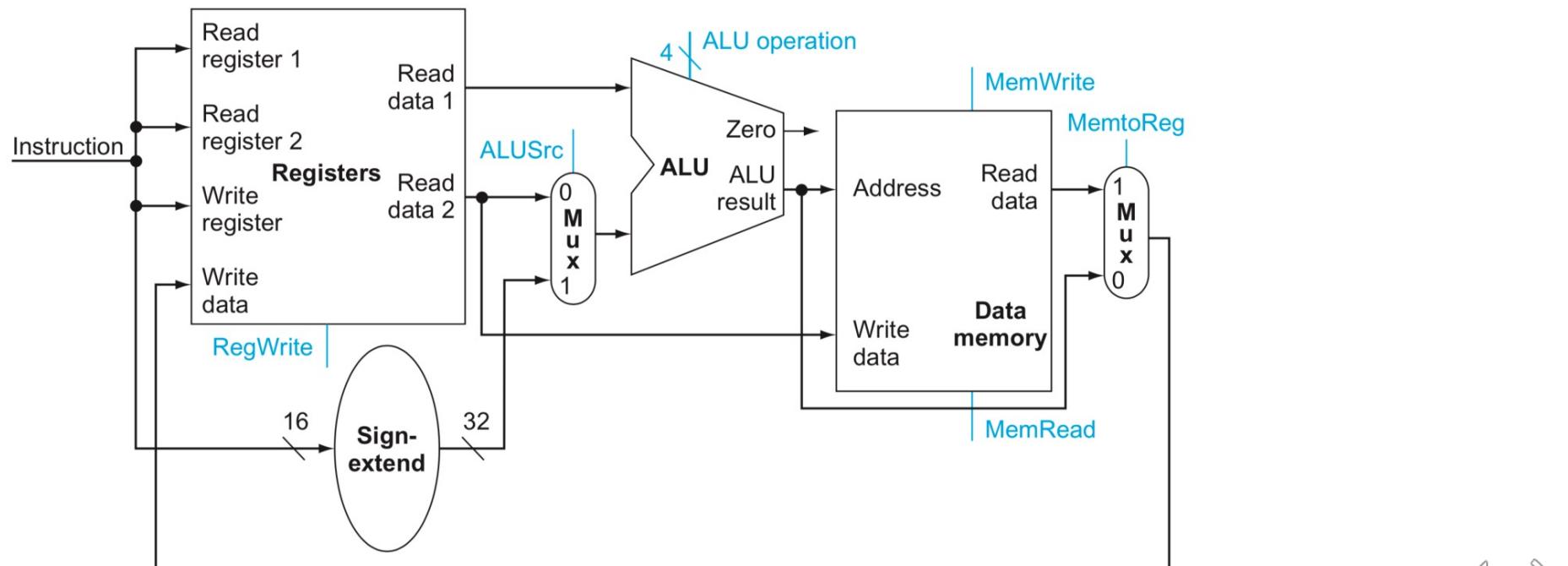
- Registeroperanden bei **ALUSrc=0** oder vorzeichenerweiterten Immediate **ALUSrc=1**



Speicherzugriffsbefehle /5

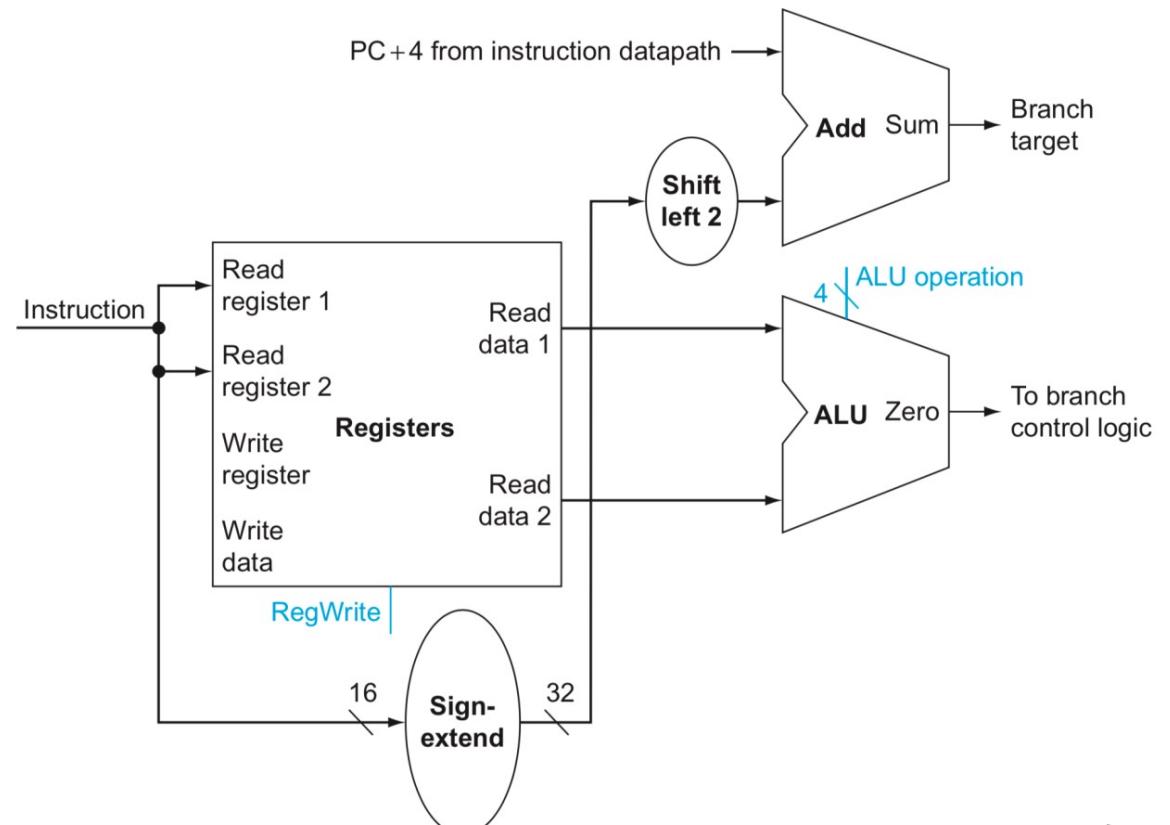
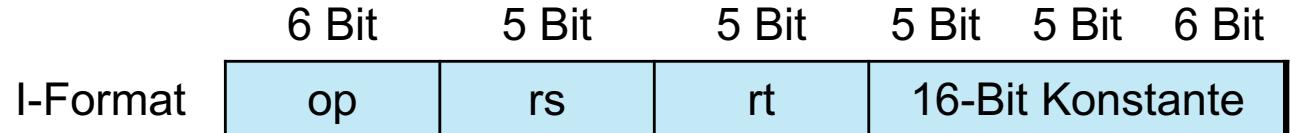


- Speichern des ALU-Ergebnisses bei $\text{MemToReg}=0$ oder des Datenspeicherausgangs ($\text{MemToReg}=1$) in das Registerfile



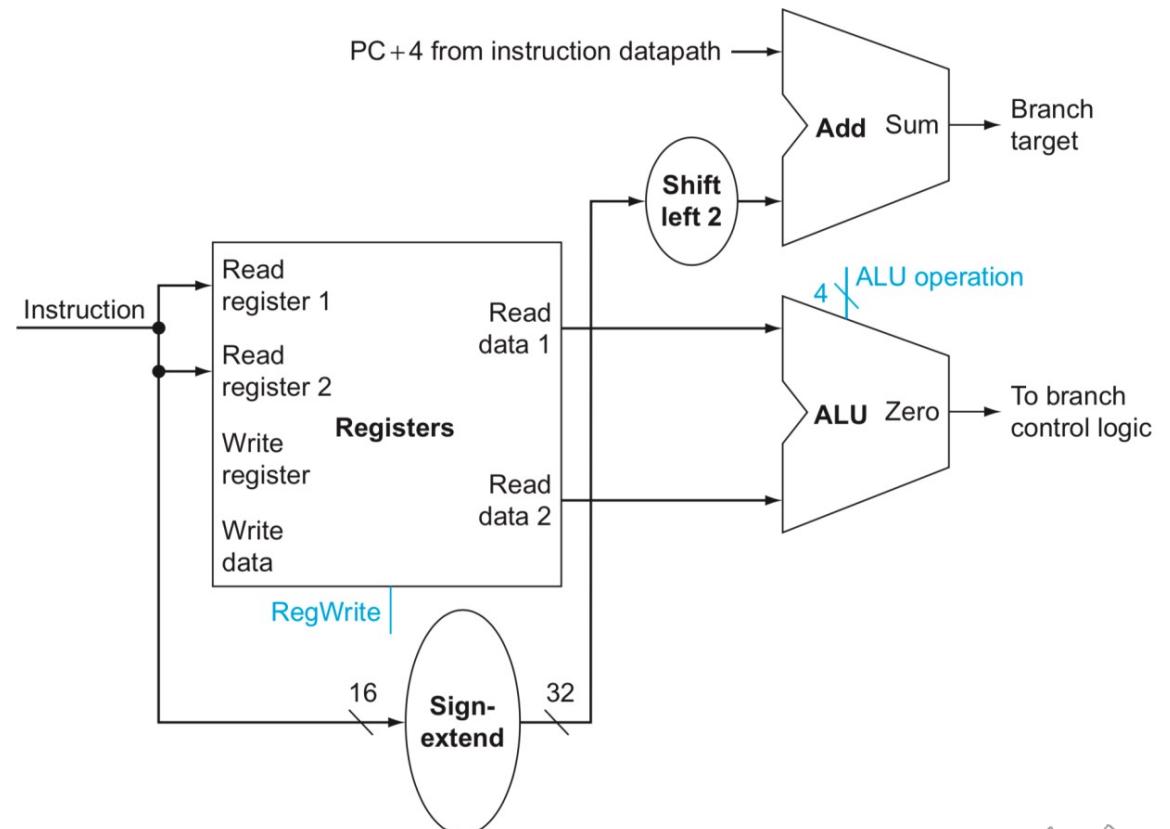
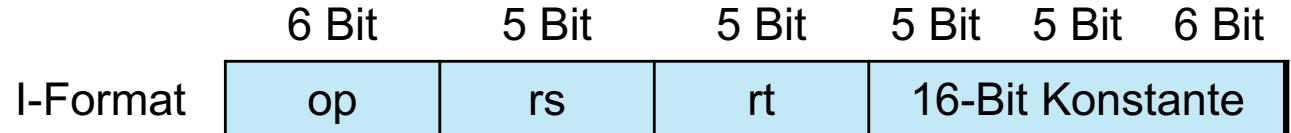
Verzweigung /1

- **beq** verwendet die ALU um auf Gleichheit zu testen (**Subtraktion**)
- Lesen der Quellregister rs und rt aus Registerfile
- Benötigen die ALU und die Vorzeichenerweiterungseinheit
- Was brauchen wir noch um **beq** zu implementieren?



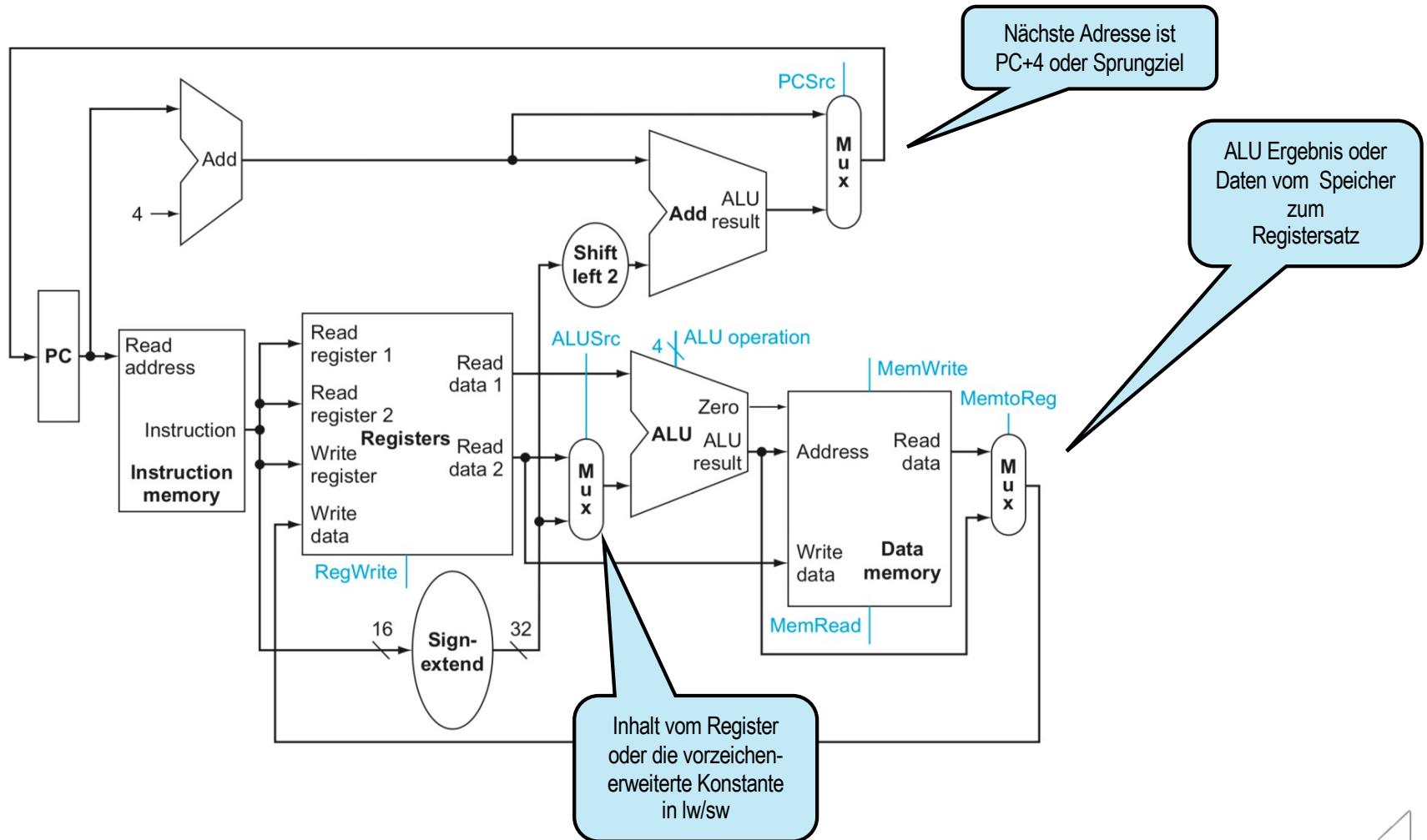
Verzweigung /2

- Ist Subtraktionsergebnis 0 →
 - Quelloperanden waren gleich
 - die Verzweigungsadresse mit dem vorzeichenverweiterten Immediate berechnen
 - zusätzlicher Addierer
 - Neue Zieladresse ergibt sich aus dem Wert PC+4 Wert und dem um 2 (x4) stellen geshiftet Offset
- Andernfalls keine Verzweigung und PC + 4 für nächsten Befehl



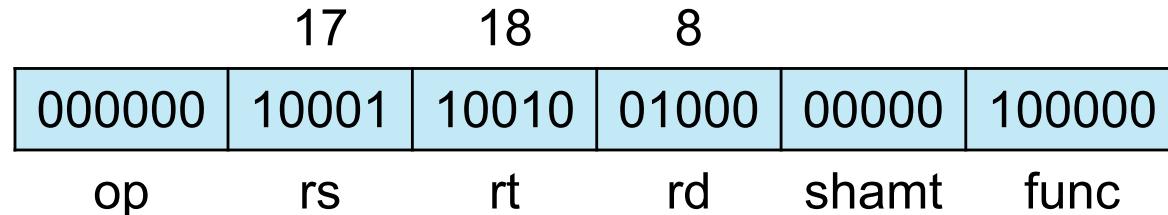
Teile zusammenfügen

- Benutze Multiplexer, um die Teile zusammenzufügen



Steuerung

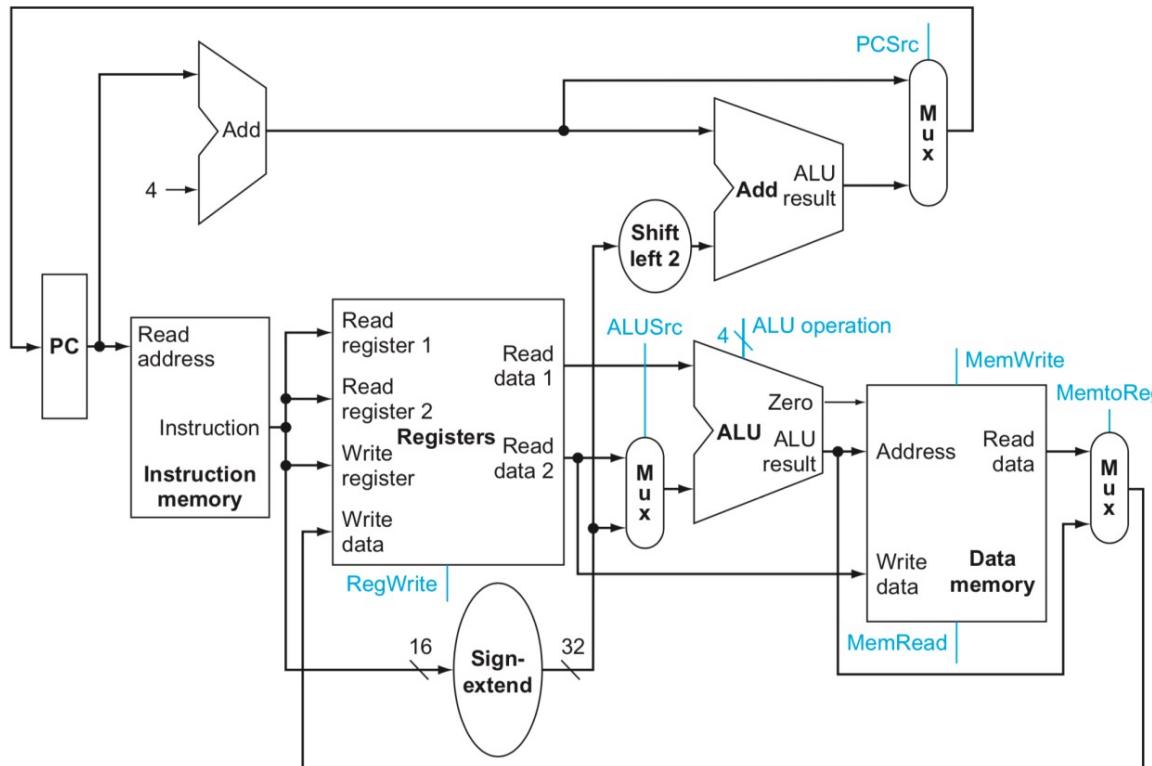
- Wählt die auszuführende Operation (ALU, lese/schreibe, etc.)
- Kontrolliert den Datenfluss (Multiplexer Ausgänge)
- Die Information kommt von den 32 Bits des Befehls
- Beispiel: **add \$8, \$17, \$18**
 - Befehlsformat:



- Zuerst legen wir die ALU-Steuerung fest und dann die Hauptsteuereinheit
 - ALU Operation hängt ab vom *Opcode* (`lw, sw, beq, addi, ...`) und manchmal auch *Function Code* (`add, sub, slt, ...`)

ALU-Steuerung

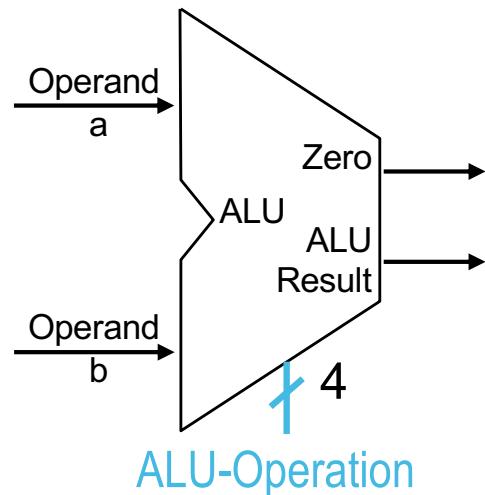
- Was muss die ALU machen für
 - Lade/Speicher-Befehle? ➤ addieren für die Adressberechnung
 - beq? ➤ Subtrahieren mit Vergleich auf Null
 - R-Typ Befehle? ➤ hängt vom Funktionscode ab



- Ziel → Datenpfad, um die ALU-Steuerung ergänzen
- Hauptsteuer-einheit steuert nicht in allen Fällen direkt die ALU

ALU und ihre Steuerung

- ALU-Steuersignalbelegung sowie die entsprechende ALU-Operation zusammengefasst



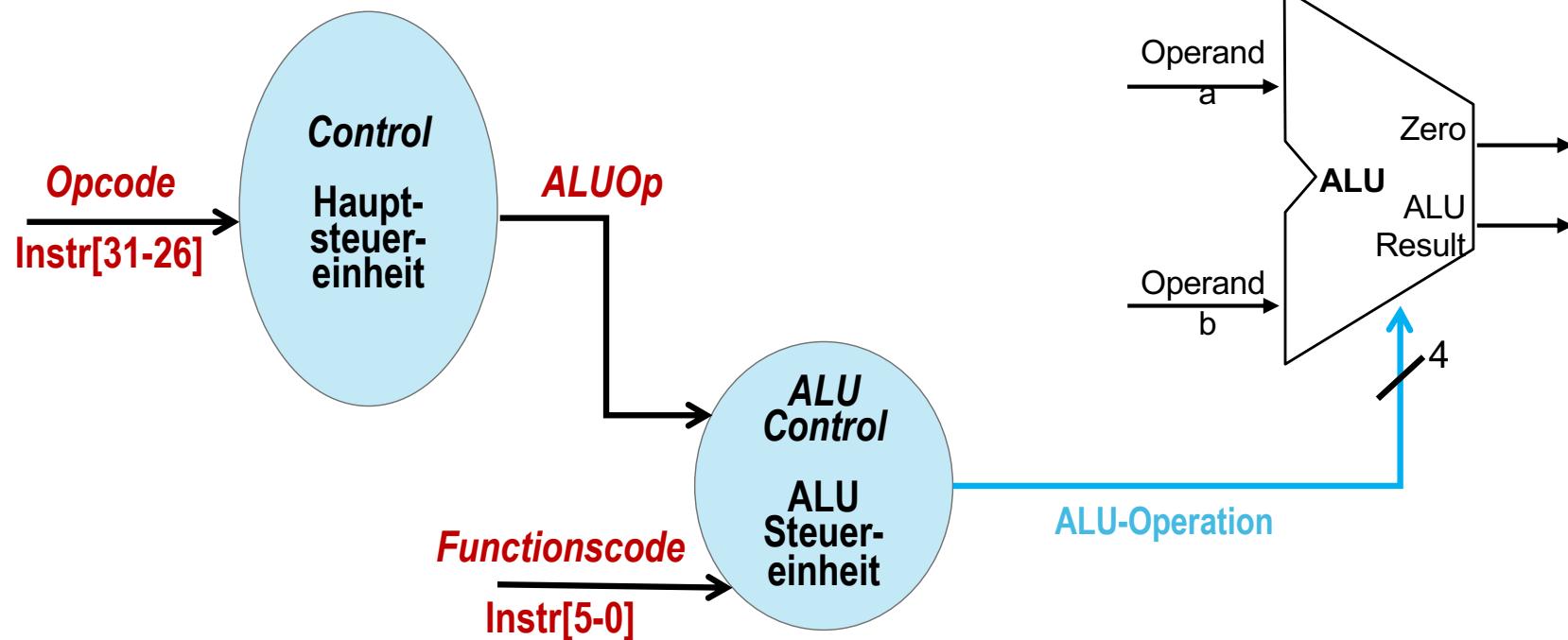
ALU-Operation Steuersignale	ALU Operation
00 00	and
00 01	or
00 10	add
01 10	sub
01 11	slt
11 00	nor

ALU Steuersignale

- Die 4 ALU Steuersignale werden vom Opcode und manchmal auch vom Funktionscode berechnet
- Um die Steuerung zu vereinfachen, verwenden wir eine kleine Steuereinheit (genannt **ALU control**), die die ALU Steuersignale berechnet aus:
 - 2-Bit Steuersignal genannt **ALUOp**
 - berechnet vom Hauptsteuereinheit
 - spezifiziert entweder die Operation, die ausgeführt werden soll, oder gibt an, dass die Operation durch Funktionsfeld bestimmt wird
 - **Addition** für Lade/Speicheroperationen → **ALUOp = 00**
 - **Subtraktion** für **beq** → **ALUOp = 01**
 - Bestimmt durch Funct-Feld der R-type-Befehle → **ALUOp = 10**
 - Funktionscode, sofern vorhanden (R-Typ)

ALU-Steuereinheit /1

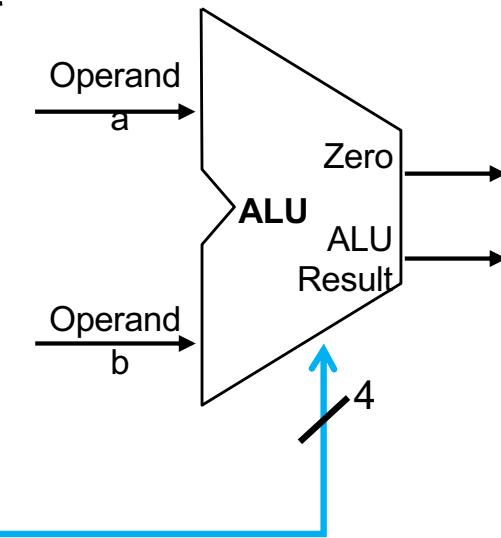
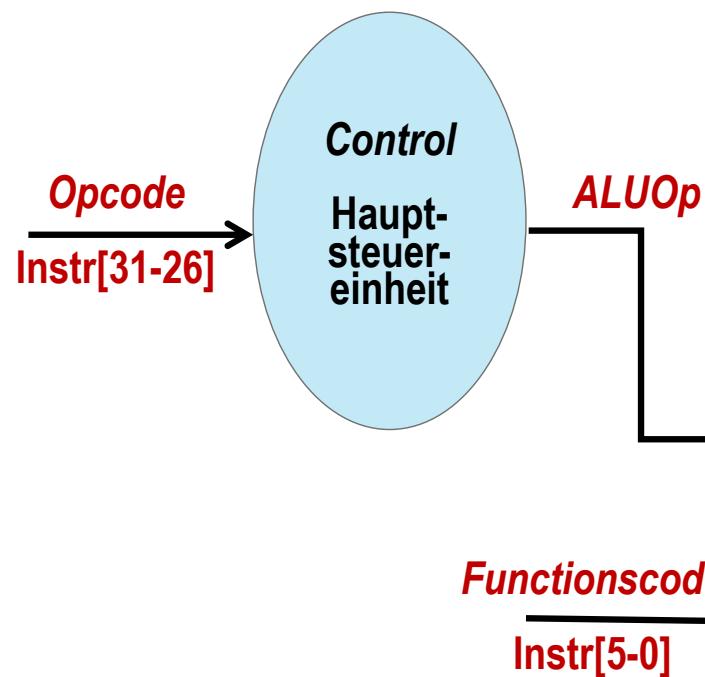
- Aufbau der gesamten Steuereinheit bestehend aus der Hauptsteuereinheit und der ALU-Steuereinheit



- 4bit breite Ausgangssignal der ALU-Steuereinheit definiert mittels des Signals **ALU-Operation** die Funktion der ALU
- Die Funktion der ALU-Steuereinheit wird durch die 2 Eingänge **ALUOp** und den 6bit **Funktionscode** festgelegt

ALU-Steuereinheit /2

- Aufbau der gesamten Steuereinheit bestehend aus der Hauptsteuereinheit und der ALU-Steuereinheit



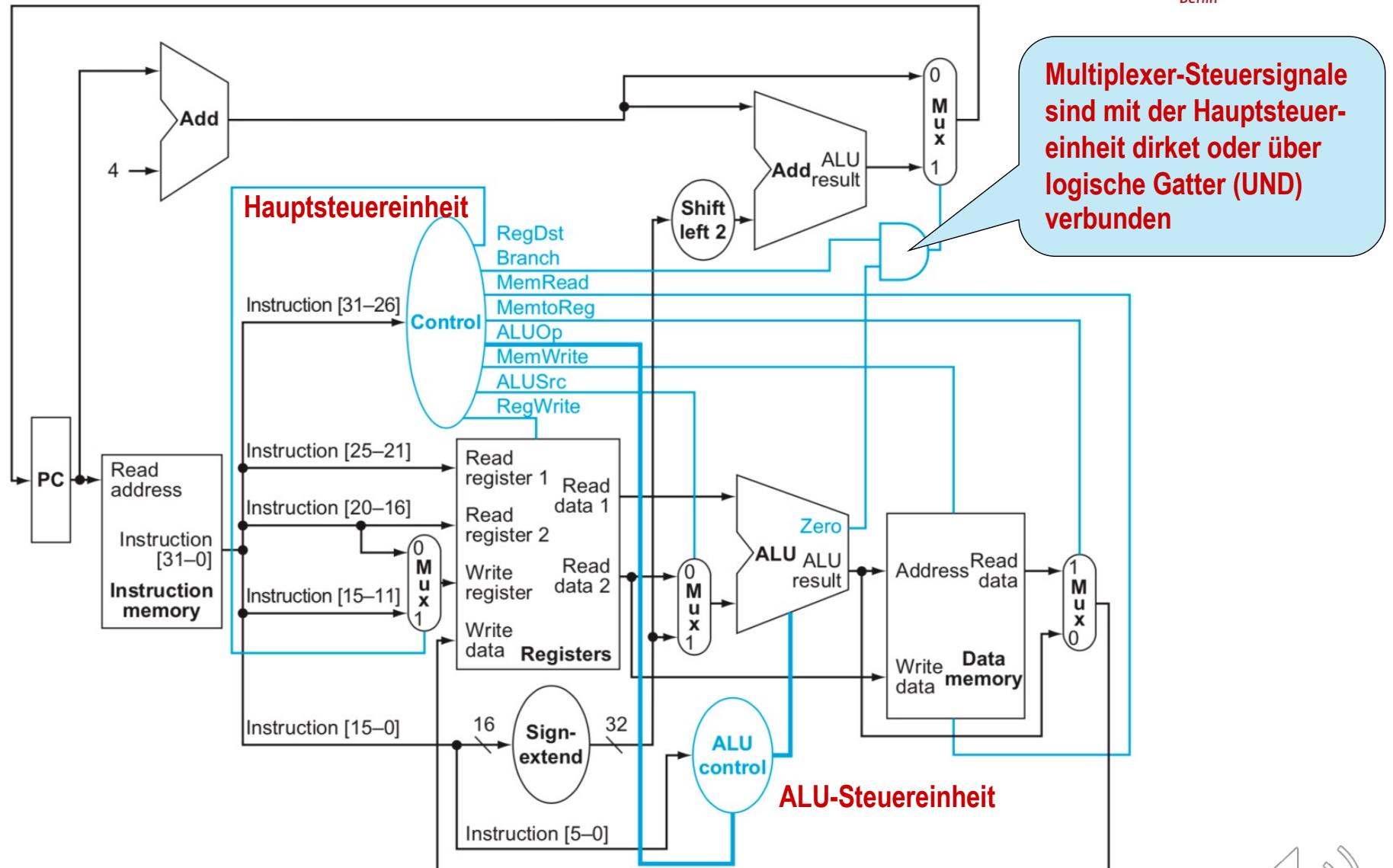
- Die Hauptsteuereinheit liest wiederum die 6bit des OP-Code der Instruktion aus

Spezifikation ALU Steuereinheit

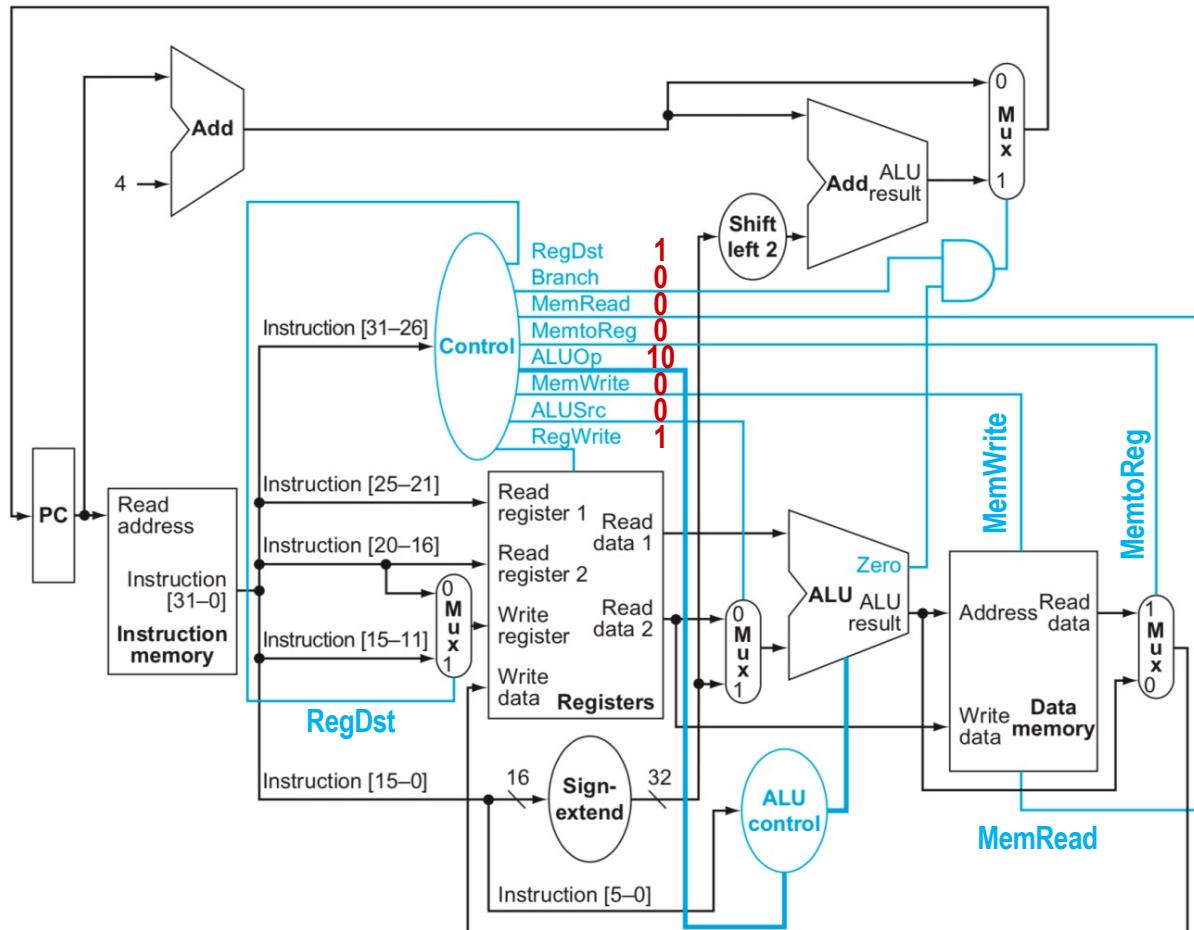
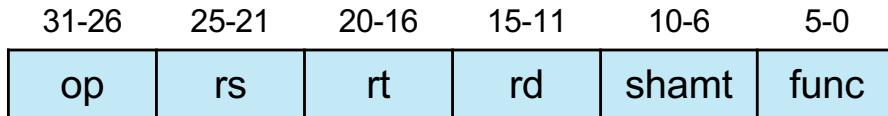
- ALU-Steuereinheit ist durch Wahrheitstabelle spezifiziert
- Realisierung durch Logik-Gatter

opcode	funct	ALUOp		Funct field						ALU control signals	Desired ALU action
		ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
lw/sw	n.z.	0	0	X	X	X	X	X	X	0010	add
beq	n.z.	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

Datenpfad mit Steuerung



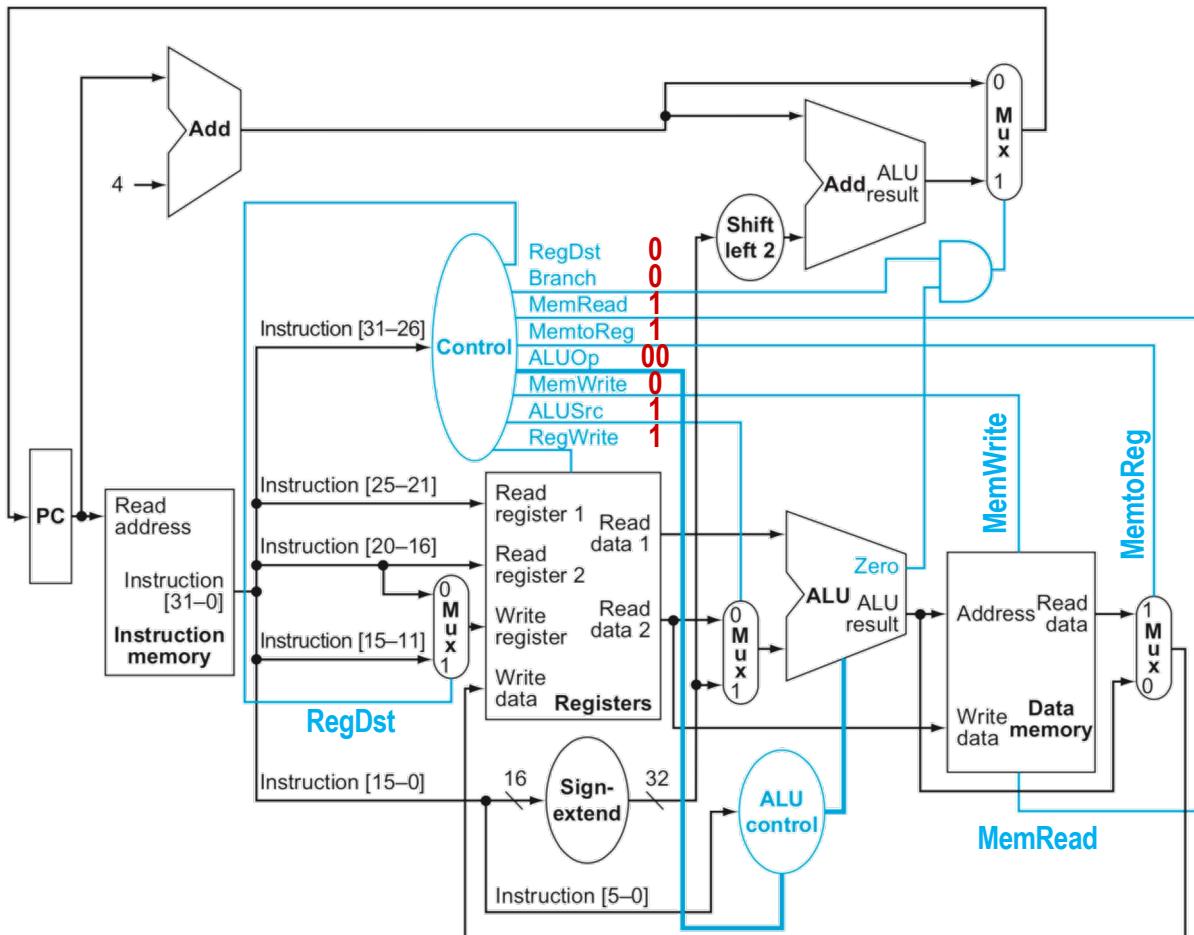
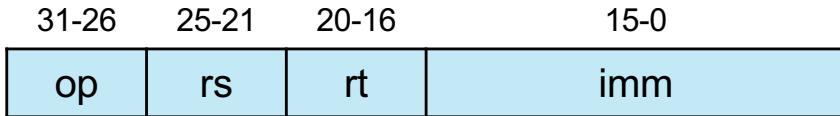
Steuersignale für R-Type Befehle



Steuersignale für das R-Befehlsformat
(**add** Befehl: Inhalte der Register rs & rt addieren und Ergebnis in rd abgelegen)

- 1) Reg Destination = 1 setzen, damit wir das Register rd schreiben können
- 2) Branch=0 gesetzt so das das UND Gatter eine 0 liefert und wir PC+4 propagieren
- 3) MemRead=0 da wir nicht aus dem Datenspeicher lesen möchten
- 4) MemtoReg=0 da wir das ALU-Ergebnis propagieren
- 5) ALUOp = 10 wie bereits für das R-Type Befehlsformat definiert
- 6) MemWrite=0 da wir nicht in den Datenspeicher schreiben wollen
- 7) ALUSrc=0 da wir den 2 Operanden aus dem Registersatz propagieren möchten
- 8) RegWrite=1 da wir das ALU Ergebnis in das Register rd schreiben möchten

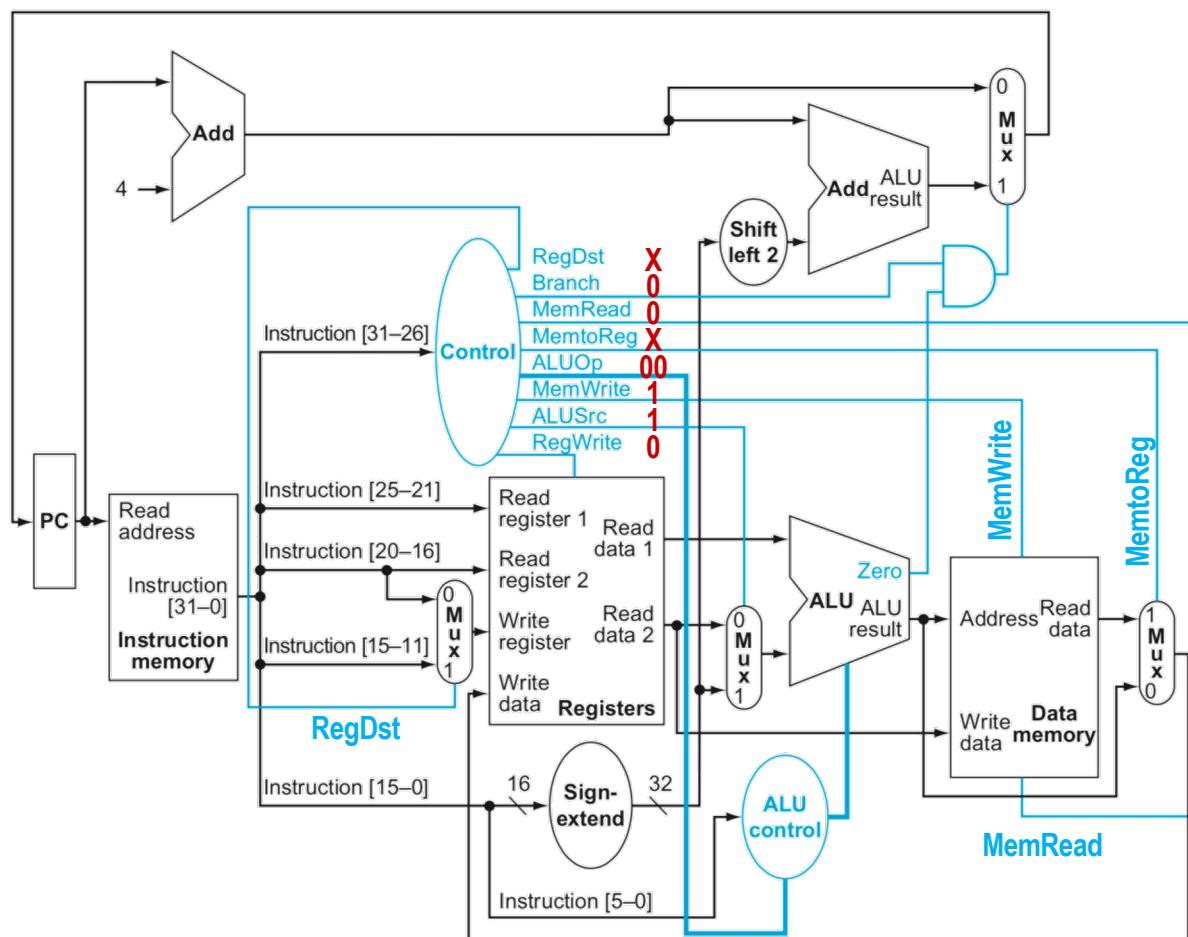
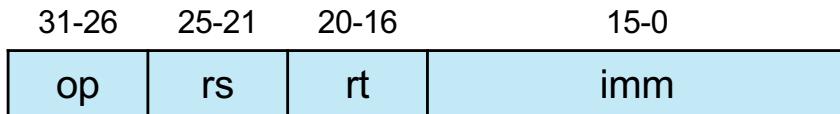
Steuersignale für Iw



Steuersignale für das I-Befehlsformat
(Iw Befehl: laden eines Wortes aus dem Speicher und ablegen des Wortes im Registersatz)

- 1) Reg Destination = 0 setzen, damit wir das Register rt schreiben können
- 2) Branch=0 da wir nicht verzweigen möchten
- 3) MemRead=1 da wir aus dem Datenspeicher lesen möchten
- 4) MemtoReg=1 da wir das Datum aus dem Datenspeicher propagieren wollen
- 5) ALUOp = 00 für die Ausführung einer Addition
- 6) MemWrite=0 da wir nicht in den Datenspeicher schreiben wollen
- 7) ALUSrc=1 da wir den das vorzeichenerweiterte Immediate propagieren möchten
- 8) RegWrite=1 damit das Datum aus dem Datenspeicher in das Register rt geschrieben wird

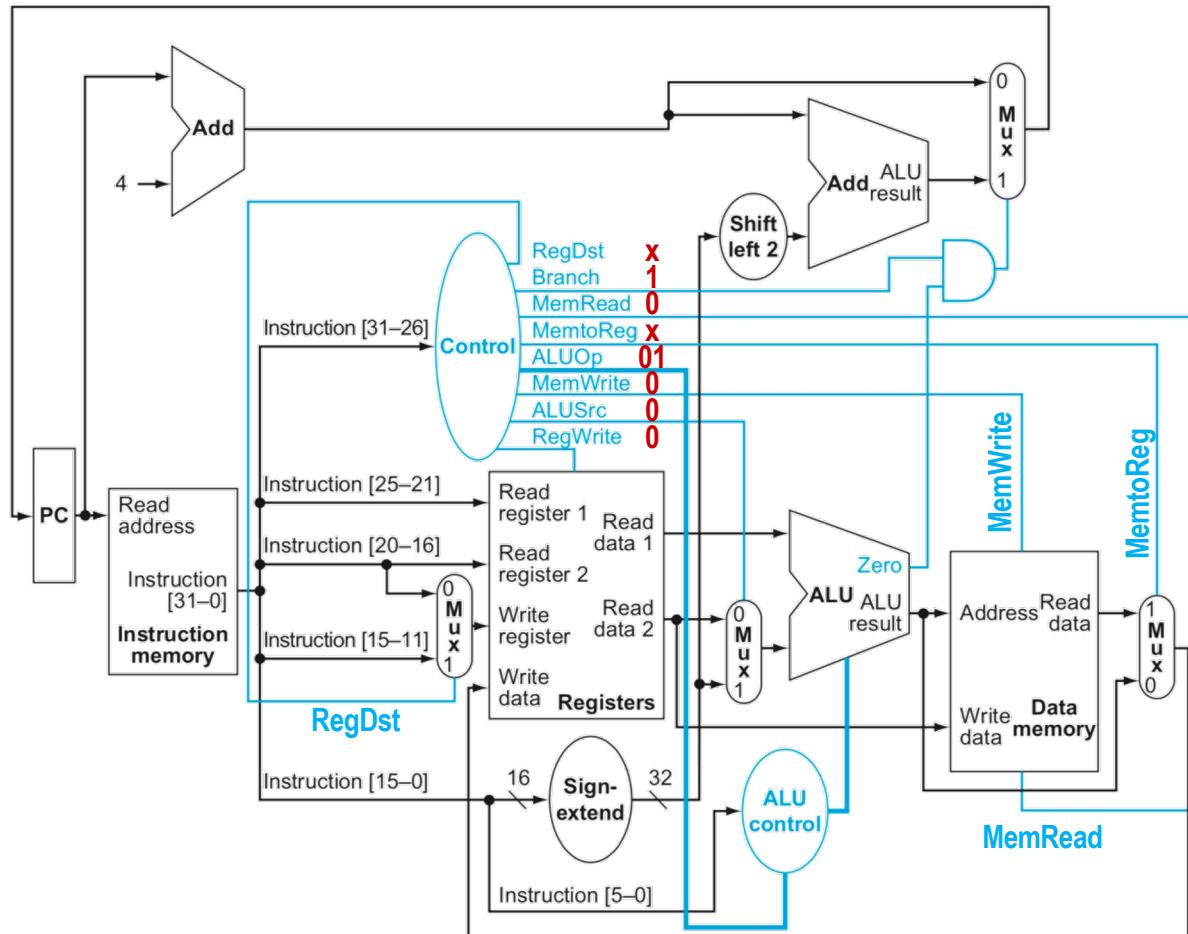
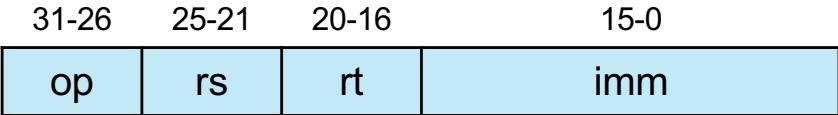
Steuersignale für sw



Steuersignale für das I-Befehlsformat
(**sw** Befehl: Store Word speichert ein Wort aus dem Registersatz in den Datenspeicher)

- 1) RegWrite=0 da wir NICHT in den Registersatz schreiben möchten, somit
- 2) Reg Destination = X
- 3) MemtoReg= X
- 4) Branch=0 da wir nicht verzweigen möchten
- 5) MemRead=0 da wir nicht aus dem Datenspeicher lesen möchten
- 6) ALUOp = 00 für die Ausführung einer Addition
- 7) MemWrite=1 da wir in den Datenspeicher schreiben wollen
- 8) ALUSrc=1 da wir das vorzeichenverweiterte Immediate propagieren möchten

Steuersignale für beq



Steuersignale für das I-Befehlsformat

(**beq** Befehl: wir möchten eine Subtraktion durchführen und bei einem Ergebnis von 0 verzweigen an die berechnete Verzweigungsadresse)

- 1) **RegWrite=0** da wir NICHT in den Registersatz schreiben möchten, somit
- 2) **Reg Destination = X**
- 3) **MemtoReg=X**
- 4) **Branch=1** da wir verzweigen möchten, wenn zusätzlich noch das Ergebnis der Subtraktion 0 (zero 1 ist) und das UND Gatter eine 1 liefert
- 5) **MemRead=0** da wir nicht aus dem Datenspeicher lesen möchten
- 6) **ALUOp = 01** für die Ausführung einer Subtraktion
- 7) **MemWrite=0** da wir nicht in den Datenspeicher schreiben wollen
- 8) **ALUSrc=0** da wir den Inhalt von rt propagieren möchten

Spezifikation der Hauptsteuereinheit /1

- Belegung der Steuerleitungen der Hauptsteuereinheit wird ausschließlich durch den Opcode des Befehls bestimmt

– R-Typ Befehle	0	rs	rt	rd	shamt	func
– lw	35	rs	rt	16-Bit Konstante		
– sw	43	rs	rt	16-Bit Konstante		
– beq	4	rs	rt	16-Bit Konstante		

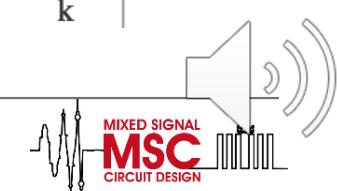
OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci- mal	Hexa- decim- al	ASCII Char- acter	Deci- mal	Hexa- decim- al	ASCII Char- acter
(1)	sll	add,f	00 0000	0	0	NUL	64	40	@
		sub,f	00 0001	1	1	SOH	65	41	A
j	srl	mul,f	00 0010	2	2	STX	66	42	B
jal	sra	div,f	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt,f	00 0100	4	4	EOT	68	44	D

| lw subu | 10 0011 35 23 # | 99 63 c |

| sw sltu | 10 1011 43 2b + | 107 6b k |

(1) opcode(31:26) == 0



Spezifikation der Hauptsteuereinheit /2

- Folgende Tabelle gibt an, wie die Steuersignale für die einzelnen Opcode-Werte gesetzt werden müssen

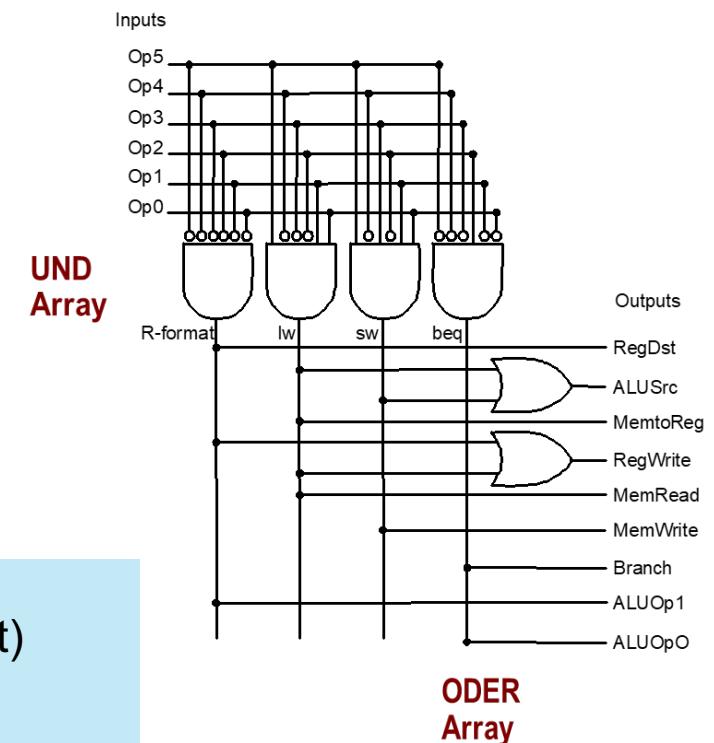
Befehl	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALU-Op1	ALU-Op0
R-Format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Mögliche Implementierung der Hauptsteuereinheit

- Tabelle kann auf eine *Programmable Logic Array* (PLA, programmierbare logische Anordnung) abgebildet werden, welche die Steuersignale berechnet:

Befehl	Opcode	RegDst	ALUSrc	...
R-format	000000	1	0	.
lw	100011	0	1	.
sw	101011	X	1	.
beq	000100	X	0	.

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



gdw = genau dann wenn

Erweiterung um Sprung Instruktion

- Wie soll der Eintakt-Prozessor erweitert werden, um den Sprung-Befehl (**j**) zu implementieren?

- Sprung-Befehlsformat:

2	26-Bit Adresse
---	----------------

- Zieladresse ist eine Verknüpfung von:

- Höchstwertigsten 4 Bits von **PC+4**
 - 26-Bit Adresse im Befehl
 - die **LSB** Bits 00 (da Wortadresse)

- Wie setzen wir das in Hardware um?

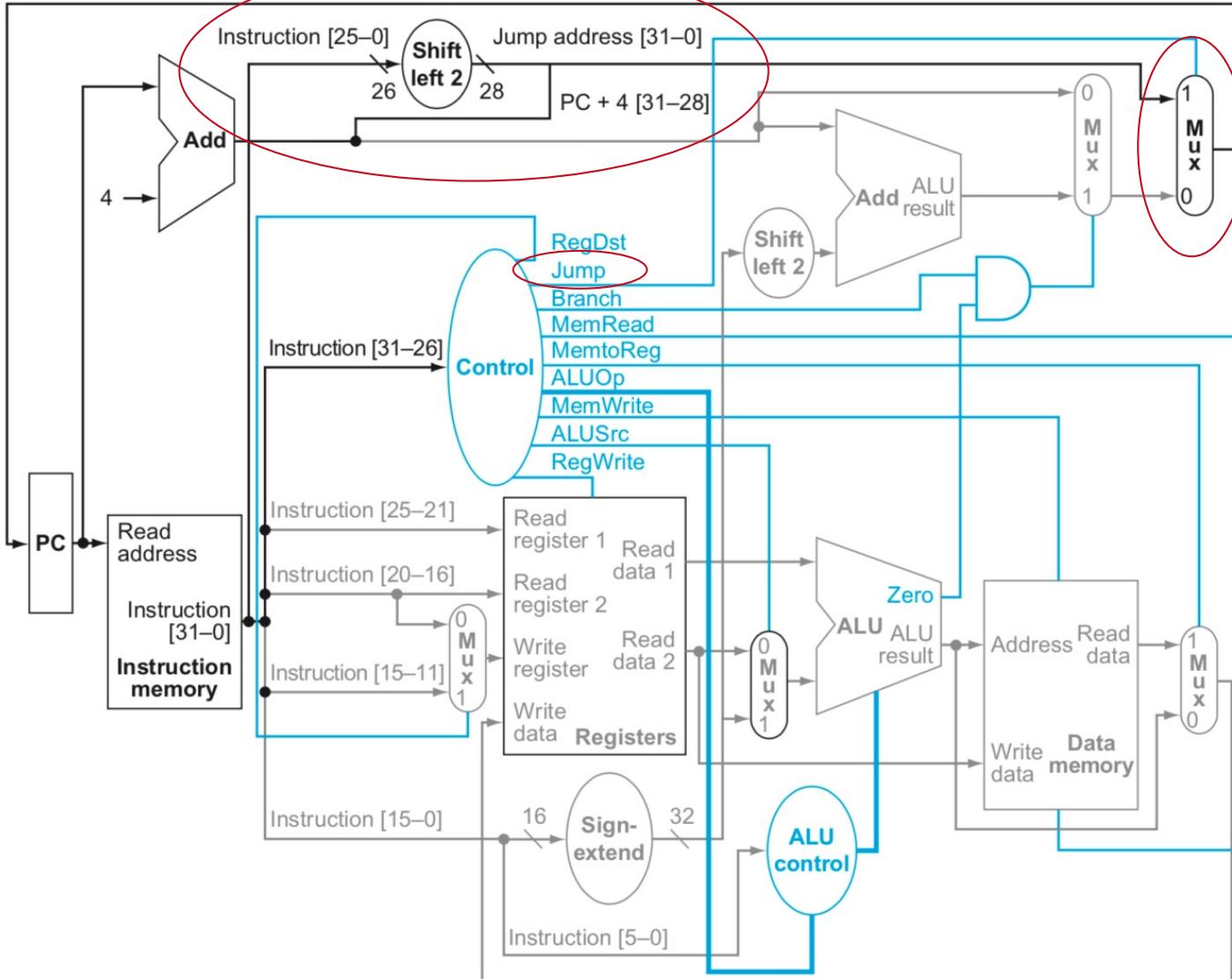
- Zieladresse berechnen: $(PC+4)[31-28] \# (Instr[25-0] \ll 2)$
 - Selektieren zwischen nächster PC und Zieladresse wird gesteuert von einem neuen Steuersignal **Jump**
 - Sonst?

Kann erzielt werden durch Leitungen richtig zu verknüpfen

„2-Stellen-nach-links-Shifter“

MUX

Eintaktprozessor erweitert um Jump

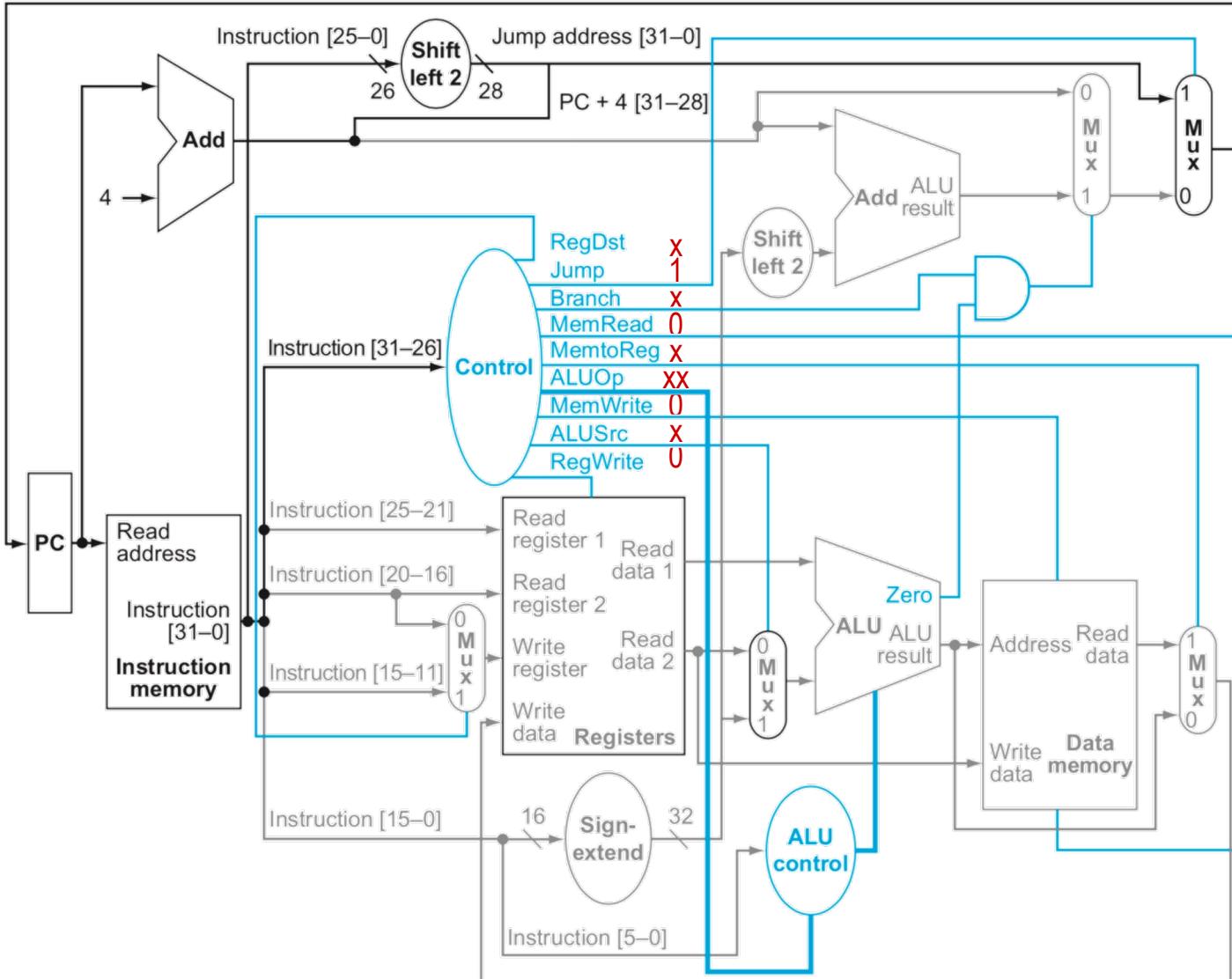


Hauptsteuereinheit um das Steuersignal **Jump** erweitert

Die Steuerleitung **Jump** treibt einen weiteren Multiplexer, der bei **Jump = 1** die neu berechnete PC Adresse propagiert

Für die Berechnung der PC Adresse wird eine zusätzliche Shift Left Unit benötigt die die 26bit LSBs der Instruktion um 2bit shiftet und den 4 MSBs des PC+4 Wert verkettet

Steuersignale für Jump /1



Steuersignale für das J-Befehlsformat
(**j** Befehl: wir möchten verzweigen an die Verzweigungsadresse)

- 1) RegWrite=0 da wir NICHT in den Registersatz schreiben möchten, somit
- 2) Reg Destination = X
- 3) MemtoReg = X

Da wir keine ALU-Berechnung benötigen und nur die neue Jumпадresse propagieren

- 4) Branch=X
- 5) ALUOp = X
- 6) ALUSrc=X
- 7) MemRead=0 da wir nicht aus dem Datenspeicher lesen möchten
- 8) MemWrite=0 da wir nicht in den Datenspeicher schreiben wollen
- 9) Jump=1 Jumпадresse propagieren

Steuersignale für Jump /2

- Bestimme 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 Input des neuen MUX benötigt
 - **RegWrite=0**; j schreibt nicht in den Registerspeicher
 - **Jump=1**; damit Jump Adresse propagiert wird
 - **MemRead=0**; selbst wenn die geladenen Daten verworfen werden, soll kein cache miss, page fault oder andere üble Dinge (nächstes Kapitel) generiert werden

Steuersignale für Jump /3

- Mem2Reg=X; j schreibt nicht in den Registerspeicher
- ALUOp=X; unerheblich was die ALU für eine Operation ausführt; das Ergebnis wird verworfen
- ALUSrc=X; ALU Ergebnis wird verworfen, ALU Operation und Input irrelevant (Achtung: auch wenn die ALU Operation außer Acht gelassen wird, produziert diese ein Ergebnis)
- MemWrite=0; Befehl sollte nicht in den Datenspeicher schreiben

Frage aus einer alten Prüfung

- 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 (bereits vorhandene und neu hinzuzufügende) an

Setzen – Wenn-Gleich (Set-If-Equal) /1

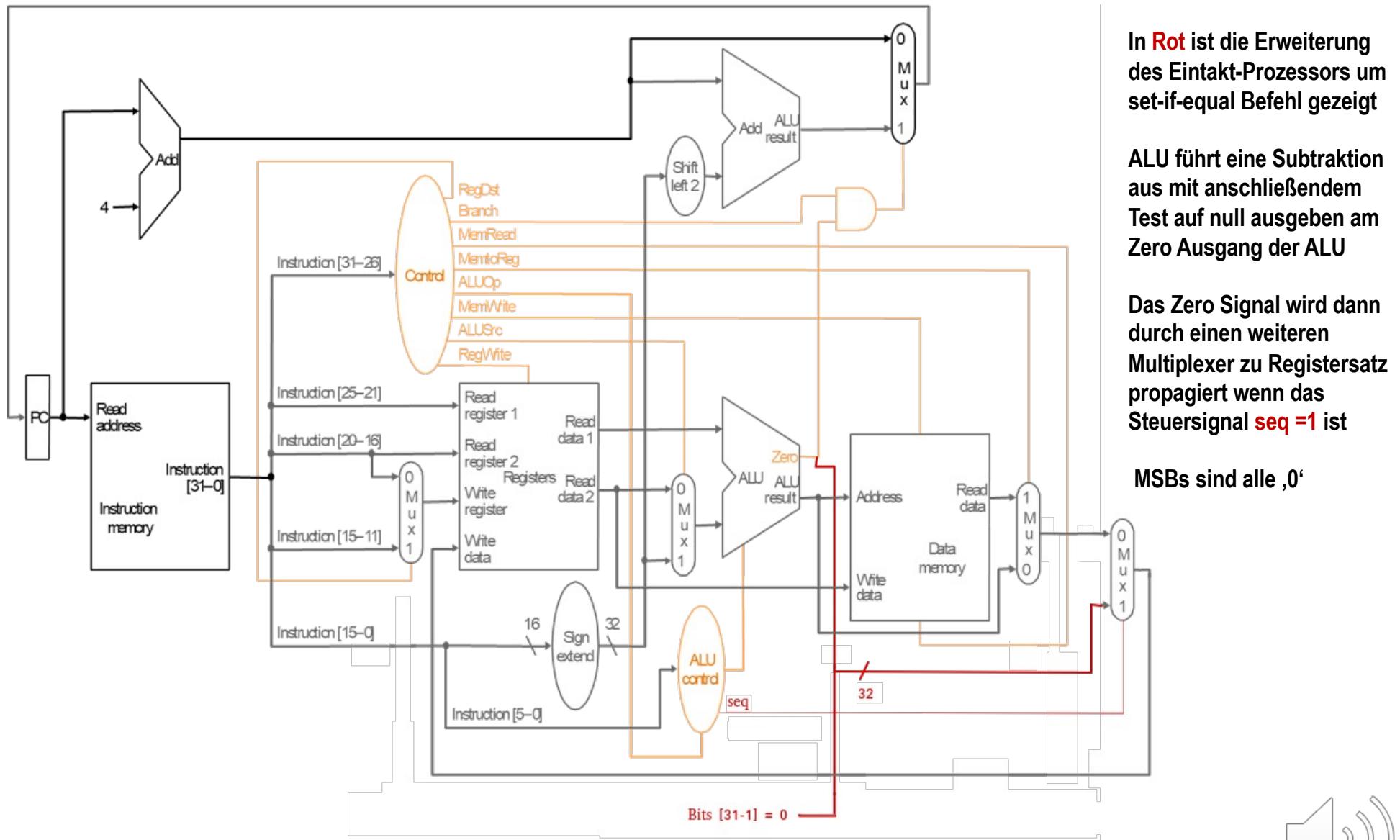
- Passendes Befehlsformat (ähnlich zum **beq** Befehl):

- R-Typ

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

- Setzen des rd Registers ($rd=1$), wenn $rs = rt$
- Veränderungen des Datenpfads:
 - Zero-Output der ALU benutzen und ALU **subtrahieren** lassen
 - Steuersignal vom Zero-Output zum Schreib-Port des Registerspeichers

Set-If-Equal



In Rot ist die Erweiterung des Eintakt-Prozessors um set-if-equal Befehl gezeigt

ALU führt eine Subtraktion aus mit anschließendem Test auf null ausgeben am Zero Ausgang der ALU

Das Zero Signal wird dann durch einen weiteren Multiplexer zu Registersatz propagiert wenn das Steuersignal seq =1 ist

MSBs sind alle ,0‘

Setzen – Wenn-Gleich (Set-If-Equal) /2

- Steuersignale:

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

- ALU Control muss modifiziert werden:

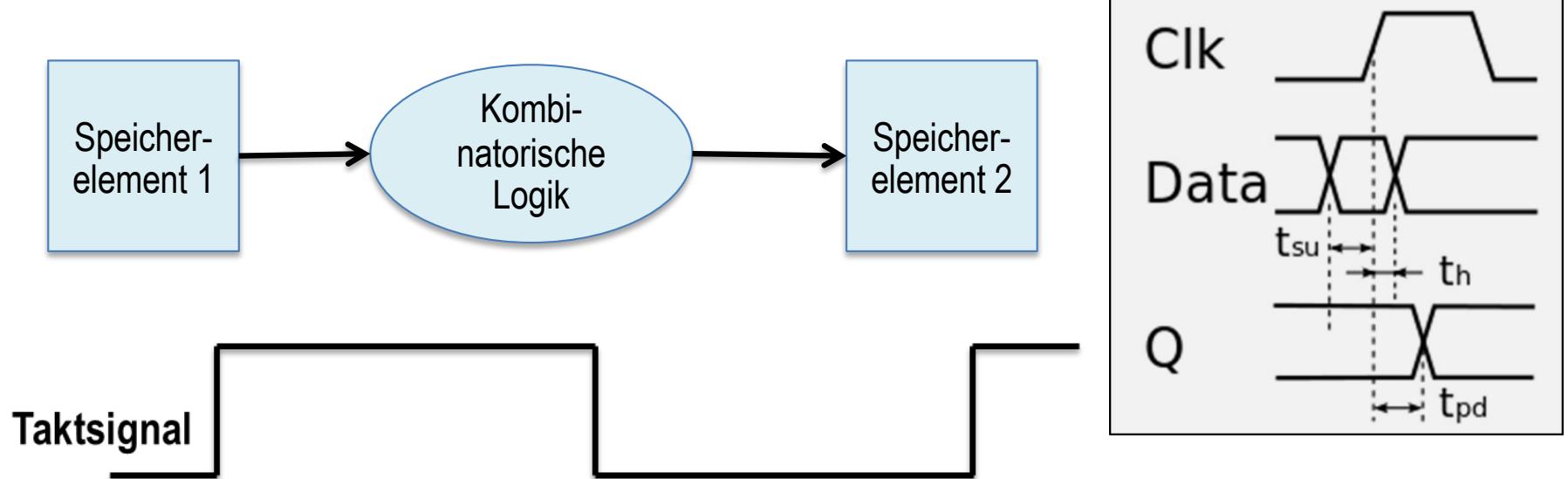
Für Funktionscode von **seq** sollen die ALU-Steuersignal 0110 (Subtraktion) generiert werden

Struktur der Eintakt-Steuerung /1

- Jede Steuerlogik ist kombinatorisch
 - Steuersignale nur abhängig vom aktuellen Befehl (Opcode und ev. auch Funktionscode)
- Wir warten, bis alles eingeschwungen ist und das Richtige getan wird
 - ALU produziert die „richtige Antwort“ nicht sofort
 - Schreib-Signale bestimmen **ob** geschrieben wird,
 - Das Taktsignal (Clock) bestimmt **wann** geschrieben wird

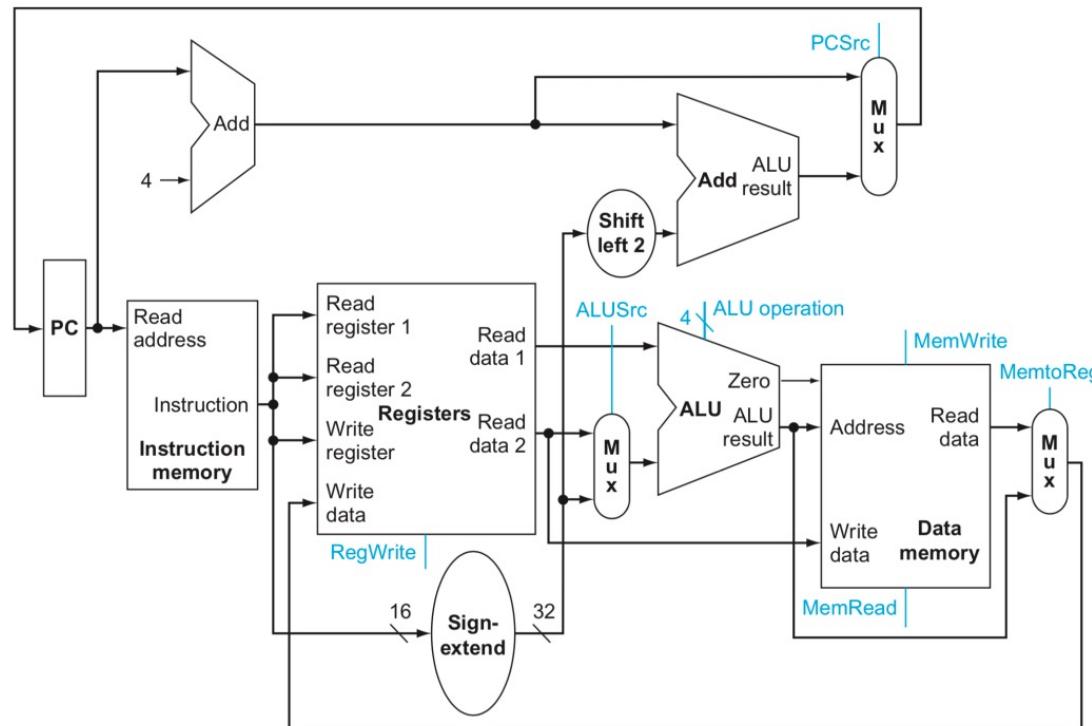
Struktur der Eintakt-Steuerung /2

- Taktzeit wird durch den längsten Pfad bestimmt
 - Dabei ignorieren wir Details wie **Setup** und **Hold Time**!



Taktzeit der Eintakt-Implementierung

- Kalkuliere die Taktzeit des Eintaktprozessors
- Annahme: Verzögerungen können bis auf folgende ignoriert werden:
 - Speicher (200ps), ALU und Addierer (100ps), Registerspeicher (50ps)
 - $1 \text{ ps} = 10^{-12} \text{ s}$



Taktzeit der Eintakt-Implementierung

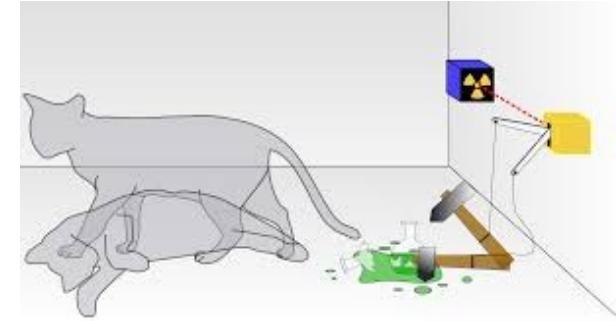
- Speicher (IM/DM, 200 ps), ALU & Addierer (100 ps), Registerspeicher (RF, 50 ps)

R-Typ.	IM Instruction Memory	RF	ALU	RF	400 ps
Iw	IM	RF	ALU	DM Data Memory	RF
sw	IM	RF	ALU	DM	550 ps
beq	IM	RF	ALU		350 ps
j	IM				200 ps

- Langsamster Befehl ist **Iw** der 600 ps (1.67GHz) benötigt

Leistung der Eintakt-Implementierung

- Annahme: Clock mit variabler Taktzeit (nicht realistisch, nur Gedankenexperiment) → Instruktionen benötigen folgende Zeiten:
 - R-type: 400 ps
 - Load: 600 ps
 - Store: 550 ps
 - Branch: 350 ps
 - Jump: 200 ps
- „*durchschnittliche Befehlszeit*“ (DB) für folgenden Instruktionsmix:
 - 25% loads, 10% stores, 45% R-type, 15% branches, 5% jumps
- $DB = 0.25 \times 600 + 0.1 \times 550 + 0.45 \times 400 + 0.15 \times 350 + 0.05 \times 200 = 447.5 \text{ ps}$
- Implementierung mit einer Clock mit variabler Taktzeit wäre
 - $600/447.5 = 1.34$ Mal schneller

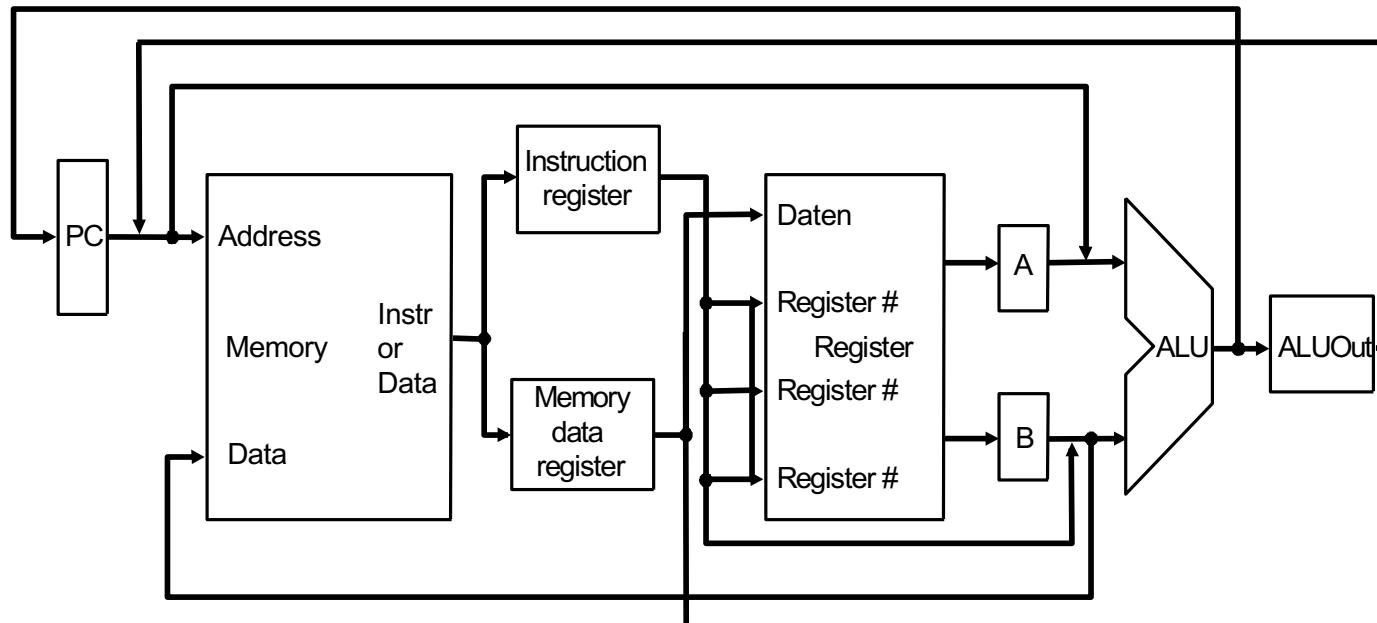


Warum Eintaktausführung nicht mehr verwendet wird /1

- Probleme des Eintaktprozessors:
 - Komplexere Befehle wie Gleitkomma-Instruktionen würden **Taktzeit** erhöhen, max. Takt verringern
 - **Verschwendungen der Chipfläche:** Hardware-Ressourcen (z. B. ALU + Addierer) müssen repliziert werden

Warum Eintaktausführung nicht mehr verwendet wird /2

- Mögliche Lösung:
 - Kürzere Taktzeit
 - Unterschiedliche Befehle dauern unterschiedliche Anzahl von Taktzyklen
 - Mehrzyklenprozessor:



Zusammenfassung

- Wir können einen Eintaktprozessor bauen, indem wir Hardwareelemente nutzen wie z.B. ALU, Registerdatei, Speicher, Flip-Flops,...
- Wir können ein **flankengesteuertes Taktverfahren** benutzen, um zu kontrollieren, **wann** die Daten geschrieben werden
- Wir benutzen **Multiplexer** um die Teile des Datenpfades zusammenzufügen
- **Steuersignale** wählen die Operation aus und kontrollieren den Datenfluss (MUX Outputs)
- Steuersignale werden vom Opcode und ev auch vom Funktionsfeld berechnet
- Eintaktprozessor ist **einfach** aber eher **langsam** und **ineffizient**
- **Als nächstes: Rechenleistung**