Rechnerarchitekturen 1*

Rechnerarithmetik

Prof. Dr. Alexander Auch

*Teilweise entnommen aus “Mikrocomputercomputertechnik 1” von Prof.Dr-Ing. Ralf Stiehler, Rechnerarchitektur von Dr. Leonard Stiegler, Patterson&Hennessy

Ziele der Veranstaltung

- **Rechnerentwurf**:
  - Prozessor, Speicher, Ein-/Ausgabe
  - Entwurfs- und Optimierungsmöglichkeiten
- **Prozessorentwurf**:
  - Befehlsverarbeitung
  - Entwurfs- und Optimierungsmöglichkeiten
- **Assemblerprogrammierung**:
  - im MIPS-Simulator MARS

Rechnerarithmetik: Zahlendarstellungen

- Zahlensysteme: binär; dezimal; hexadezimal
- Negative Zahlen: Vorzeichen
- Zweier-Komplement: Einfachere *Binär*-Arithmetik mit Vorzeichen
- Rationale/Reelle Zahlen: Fest-/Gleitkommazahlen

Rechnerarithmetik: Zahlendarstellungen

- Dezimal-System: Basis=10
  - Symbolmenge = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - zB: $234 = 2x10^2 + 3x10^1 + 4x10^0$

- Binär-System: Basis=2
  - Symbolmenge = {0, 1}
  - zB: 234_{10}

  $= 1x2^7 + 1x2^6 + 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0$

  $= 2^7 + 2^6 + 2^5 + 2^3 + 2^1$

  $= 1110\ 1010$

- Hexadezimal-System: Basis=16
  - Symbolmenge = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}
  - zB: $234_{10} = 1110\ 1010_2 = EA_{16} = 0xEA$

Umrechnung von Zahlsystemen

Beispiel Basis 16 :

Ziffernumfang {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

$(3EC9)_{16} = 9 \cdot 16^0 + 12 \cdot 16^1 + 14 \cdot 16^2 + 3 \cdot 16^3 = (3EC9)_{16} = (3EC9)_{hex}$

$= 9 \cdot 1 + 12 \cdot 16 + 14 \cdot 256 + 3 \cdot 4096$

$= 9 + 192 + 3584 + 12288$

$= 16073$

Beispiel Basis 60 :

Ziffernumfang {0,1,2,3,4,5,6,7,8,9,a,b,c……Zeichenvorratsproblem}

$(321)_{60} = 1 \cdot 60^0 + 2 \cdot 60^1 + 3 \cdot 60^2$

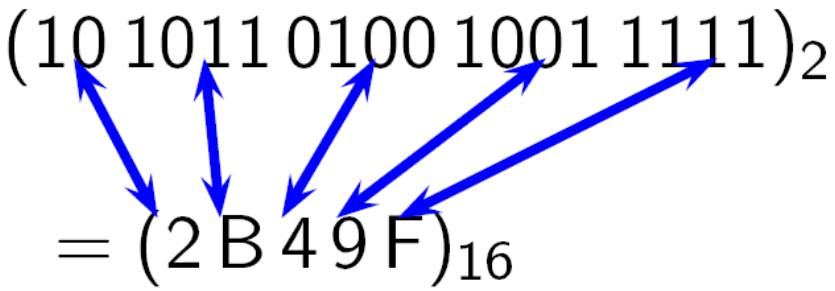
$= 1 \cdot 1 + 2 \cdot 60 + 3 \cdot 3600$

$= 1 + 120 + 10800$

$= 10921$

Umrechnung Binär ⇔ Hexadezimal

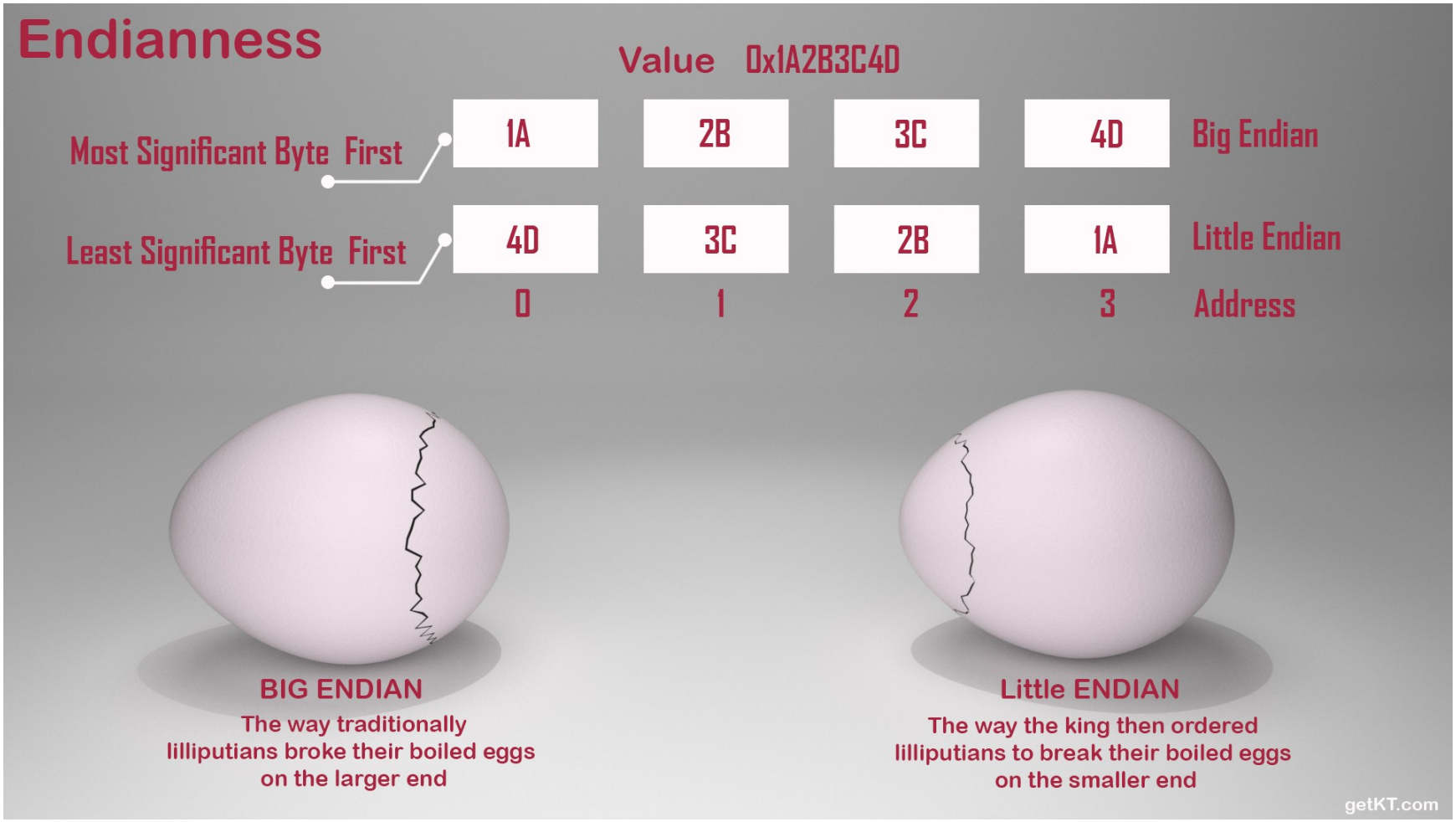
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111



$2^4 = 16$

⇒ Man braucht **4** Binärzahlen, um eine der **16** Hex-Zahlen darstellen zu können.
⇒ 4-er Gruppen bilden
⇒ 4-er Gruppen einzeln umcodieren
⇒ dabei Big/Little Endian beachten

Endianness



Quelle: https://getkt.com/blog/endianness-little-endian-vs-big-endian/

Ganzzahl-Arithmetik

Rechnerarithmetik: Binär-System

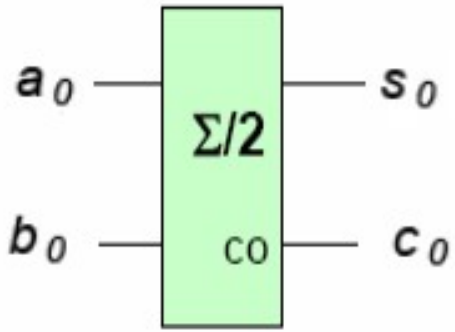
- Addition: bitweise Addition; XOR
  - zB: 234 = 1110 1010
    +34 = 0010 0010
    268 = 1 0000 1100

- Negative Zahlen: Vorzeichen durch das linke Bit bestimmt
  - negativ: zB -234 = 1 1110 1010
  - positiv: zB +234 = 0 1110 1010
  - Wichtig: normalerweise wird Zweierkomplement benutzt! (siehe später)
  - ALU soll möglichst **nur bitweise addieren** müssen !

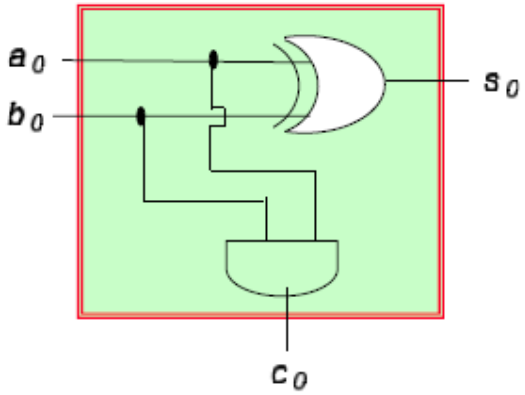
Einführung einfacher Kombinatorischer Grundschaltungen

Addition von Binärzahlen (1)

Halbaddierer : Addition von zwei 1 Bit-Zahlen



a_0	b_0	s_0	c_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Summe $s_0 = a_0$ XOR b_0

Carry $c_0 = a_0$ AND b_0

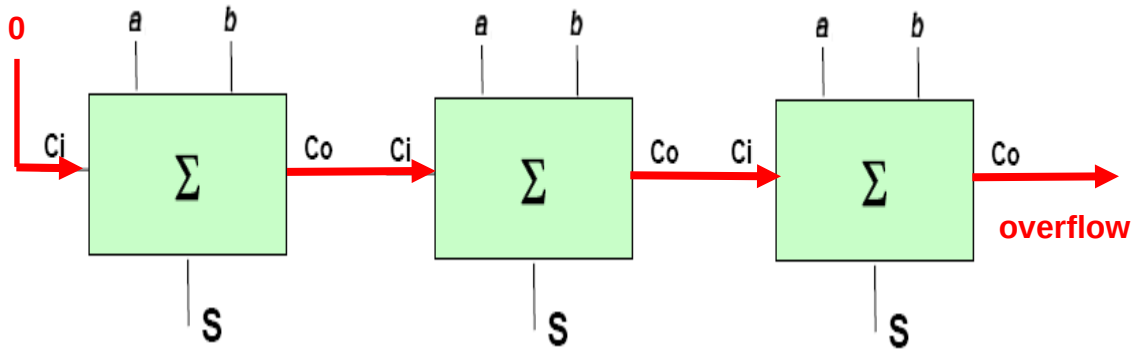
⇒ **ein Überlauf (Carry) entsteht bei Bereichsüberschreitung !**

⇒ **Wie addiert man mehrstellige Binärzahlen ?**

Addition von Binärzahlen (2)

⇒ **bei mehrstelligen Binärzahlen braucht man einen weiteren Eingang, der das Carry-Bit verarbeitet**

Volladdierer : Addition von mehrstelligen Binärzahlen



Summe, Carry = f (a_0 b_0 c_{in})

Wahrheitstabelle Volladdierer

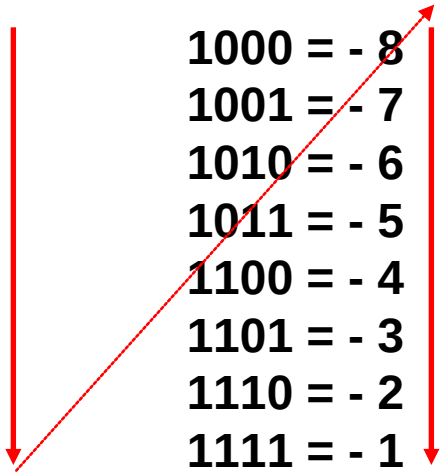
a	b	c_i	s	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Zweierkomplementdarstellung (2k-Zahlen)

⇒ ist die üblichste Darstellung, praktisch alle Rechner verwenden sie

⇒ man durchläuft erst alle positiven Zahlen und dann die negativen Zahlen in umgekehrter Reihenfolge

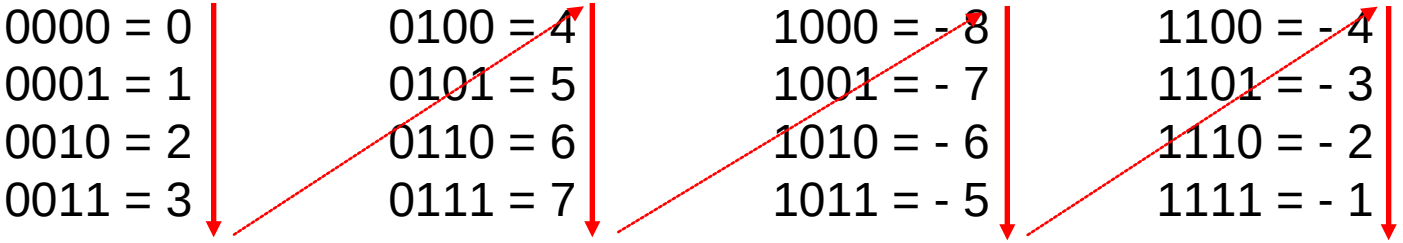
0000 = 0	1000 = - 8
0001 = 1	1001 = - 7
0010 = 2	1010 = - 6
0011 = 3	1011 = - 5
0100 = 4	1100 = - 4
0101 = 5	1101 = - 3
0110 = 6	1110 = - 2
0111 = 7	1111 = - 1



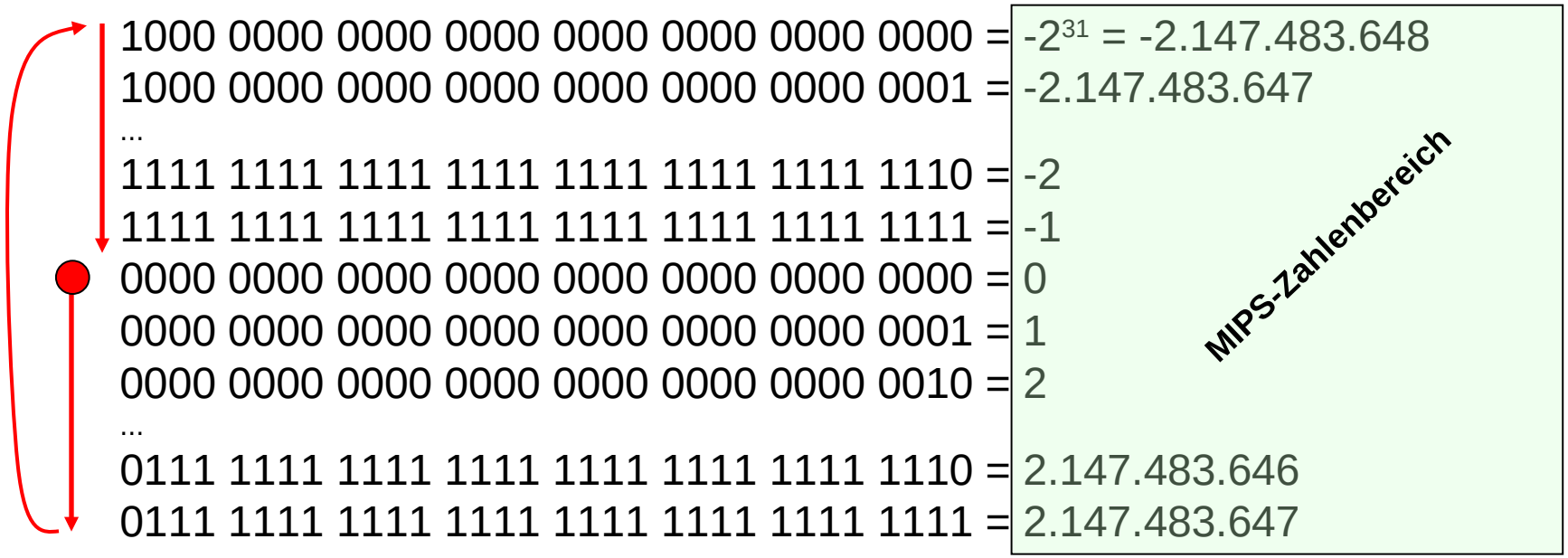
⇒ das erste Bit gibt das Vorzeichen an

⇒ es gibt eine eindeutige Darstellung für Null

Zweierkomplement für n=3



Zweierkomplement für n=31



Definition des Zweierkomplements

⇒ **Die Bitfolge $z_n \; z_{n-1} \; z_{n-2} \ldots z_1 \; z_0$ repräsentiert die Binärzahl**

$$- z_n \cdot 2^n + z_{n-1} \cdot 2^{n-1} + z_{n-2} \cdot 2^{n-2} + \ldots + z_1 \cdot 2^1 + z_0$$

⇒ **Negieren durch bitweises Komplementieren und Addition von 1**

⇒ **asymmetrischer Zahlenbereich: $[-(2^n) \,..\, (2^n-1)]$**

⇒ **Es gibt eine eindeutige Darstellung von Null**

⇒ **Bereichserweiterung durch Auffüllen mit dem Vorzeichenbit**

⇒ **Addition mit Standardaddierwerk, aber besondere Überlaufdetektion !**

Umrechnung von Zweierkomplementzahlen in Dezimalzahlen

Beispiel 5 :

$5 = (0101)_{2k}$ $= - 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$= 0 + 4 + 0 + 1$

$= 5$

Beispiel 7 :

$7 = (0111)_{2k}$ $= - 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

$= 0 + 4 + 2 + 1$

$= 7$

Beispiel -7 :

$-7 = (1001)_{2k}$ $= - 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$= -8 + 0 + 0 + 1$

$= - 7$

Zweierkomplementdarstellung – Negation und Subtraktion

⇒ **addiert man zu einer Zahl ihre Komplementärzahl, erhält man eine Reihe von Einsen, was der Zahl -1 entspricht**

Zahl	6	0	1	1	0
Komplement dazu	-7	1	0	0	1
Additionsergebnis	-1	1	1	1	1

⇒ **deswegen negiert man eine Zahl, indem man 1 zu ihrer Komplementärzahl addiert**

Zahl	6	0	1	1	0
Komplement dazu	-7	1	0	0	1
Komplement + 1	-6	1	0	1	0

⇒ **Subtraktion erfolgt durch Negation des zweiten Operanden und anschliessende Addition**

Wahrheitstabelle Volladdierer

a	*b*	c_i	*s*	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Addition/Subtraktion von 2K-Zahlen und Überlaufdetektion

Zahl 1	90		0	1	0	1	1	0	1	0
minus Zahl 2	67		0	1	0	0	0	0	1	1
Komplement von 67	188		1	0	1	1	1	1	0	0
(Komplement von 67) + 1	-67		1	0	1	1	1	1	0	1
Carry-Bit		1	1	1	1	1	0	0	0	0
Zahl 1 + Komplement von Zahl 2	**23**		0	0	0	1	0	1	1	1

Wahrheitstabelle Volladdierer

a	*b*	c_i	*s*	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

⇒ **90 - 67 = 23 => Ergebnis stimmt trotz Überlauf !**

⇒ **Keine Bereichsüberschreitung liegt vor, d.h. beide Summanden und das Ergebnis sind als 8 Bit 2k Binärzahlen darstellbar**

⇒ **binärer Overflow kann auftreten (oder nicht) , wird für die 2k-Addition ignoriert**

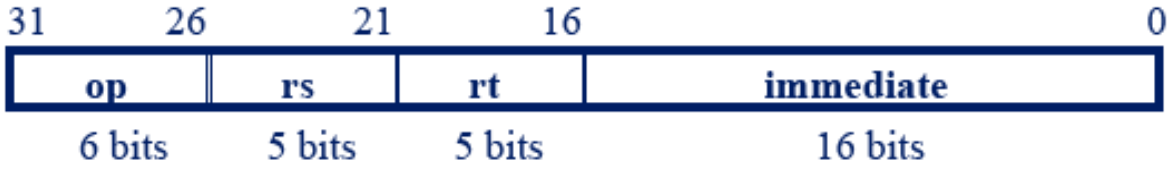
⇒ **Aber auch bei 2k-Binärzahlen sind Bereichsüberschreitungen möglich, aber der Overflow ist anders definiert**

MIPS-Light : Wir bauen einen Computer

⇒ **„Großes Ziel“ ist der Aufbau eines Prozessors MIPS-Light, der die folgende Teilmenge an MIPS-Instruktionen verarbeiten kann.**

- `add, sub, and, or, slt`

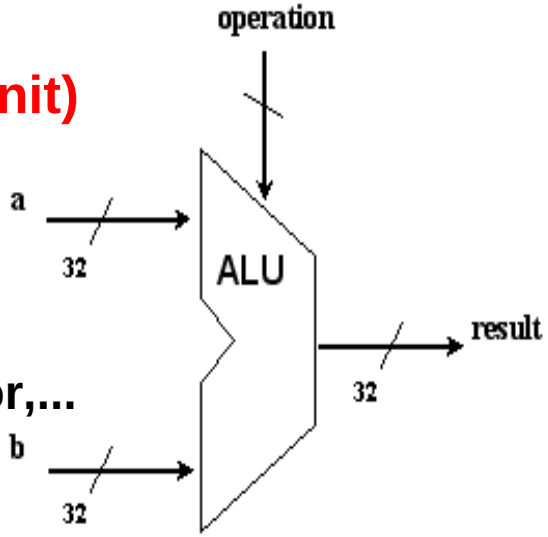
- `lw, sw, beq`



- `j`

⇒ **Erster Schritt ist der Aufbau einer einfachen ALU (Arithmetic Logical Unit)**

Aufbau einer einfachen ALU (Arithmetic Logical Unit)



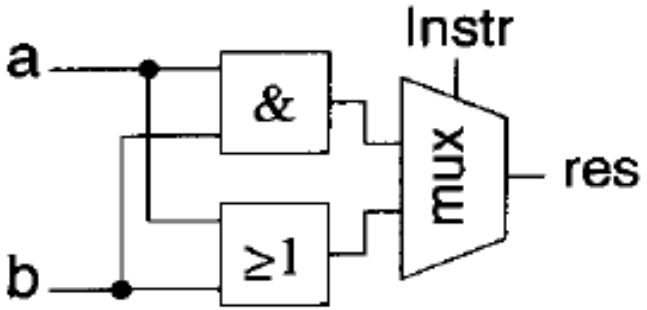
Zur Realisierung einer MIPS - Light- ALU, die einfache arithmetische und logische MIPS-Light-Operationen durchführen soll braucht man nur :

- ⇒ **ein paar einfache logische Gatter : and/or/nand/nor,xor,...**
- ⇒ **ein paar Volladdierer**
- ⇒ **ein paar Multiplexer und Demultiplexer bzw. Encoder/Decoder**
- ⇒ **um später die ALU in einen MIPS-Light Prozessor zu integrieren, benötigt man dann zusätzlich noch ein paar in Reihe geschaltete Flipflops =„Register“**
- ⇒ **immer klarer wird auch :**
  - ⇒ **Wir müssen dazu die grundlegenden ISA-Befehlsstrukturen kennen**
  - ⇒ **Warum ?**
    - ⇒ **Befehle liegen in Hardware-Registern (Flipflops), die wir „irgendwie“ mir der ALU und anderen Prozessorkomponenten verbinden müssen**

Aufbau einer einfachen ALU (Arithmetic Logical Unit)

⇒ **Grundgedanke ist, zunächst eine 1 Bit-ALU zu konstruieren und diese dann auf 32 Bit zu erweitern**

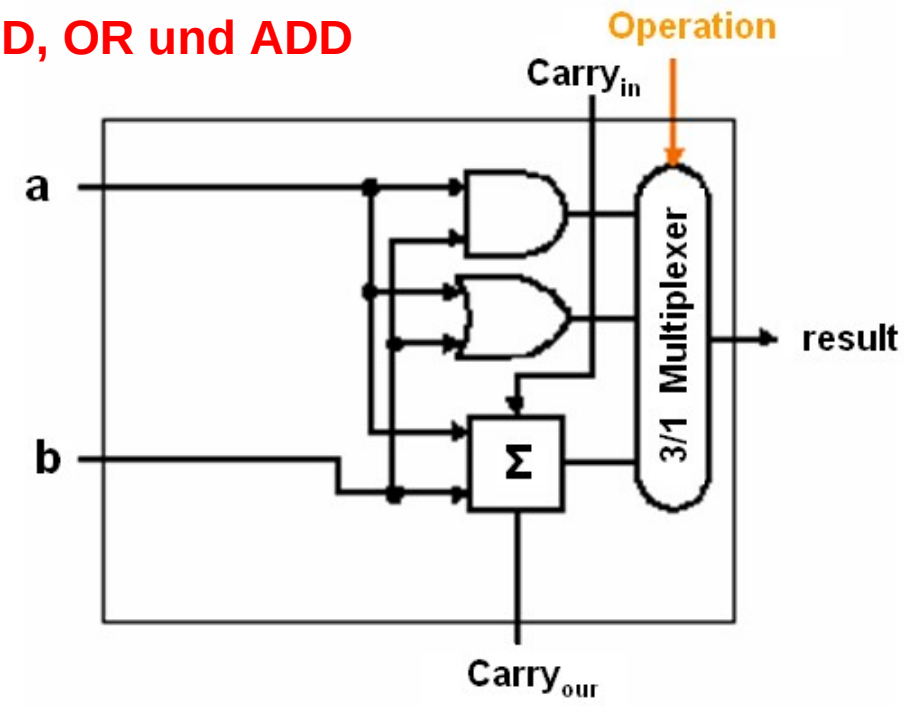
⇒ **Betrachten wir zunächst das Blockschaltbild einer einfachen 1- bit ALU, die nur die Operationen AND und OR ausführt:**



⇒ **Beide Eingänge werden auf die entsprechenen Gatter geführt und das Ergebnis per 1Bit - Multiplexer (die “Befehlsauswahl”) auf den Ausgang gelegt => trivial !**

⇒ **Nächster Schritt : Erweiterung der 1- Bit ALU für den ADD-Befehl**

1-Bit ALU für AND, OR und ADD



⇒ die Schaltung wird um einen Volladdierer VA erweitert.

⇒ man braucht nun wegen des VA-Ausgangs einen 3 → 1 Multiplexer

⇒ ld(3)=1.58 → 2 Bit für die Befehlsauswahl nötig

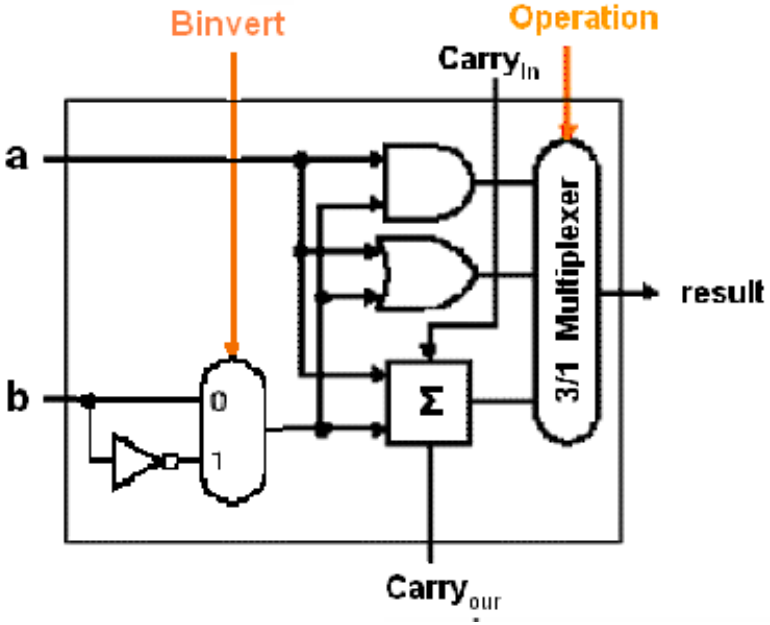
⇒ Nächster Schritt : Erweiterung der 1- Bit ALU für den SUB-Befehl

1-Bit ALU für AND, OR, ADD und SUB

⇒ Subtraktion = Addition mit dem Zweierkomplement des zweiten Summanden
- ⇒ Bitweise Komplementierung des Operanden
- ⇒ Addition von 1 zum komplementierten Operanden

⇒ Wir brauchen einen Inverter zur Komplementierung des Bits

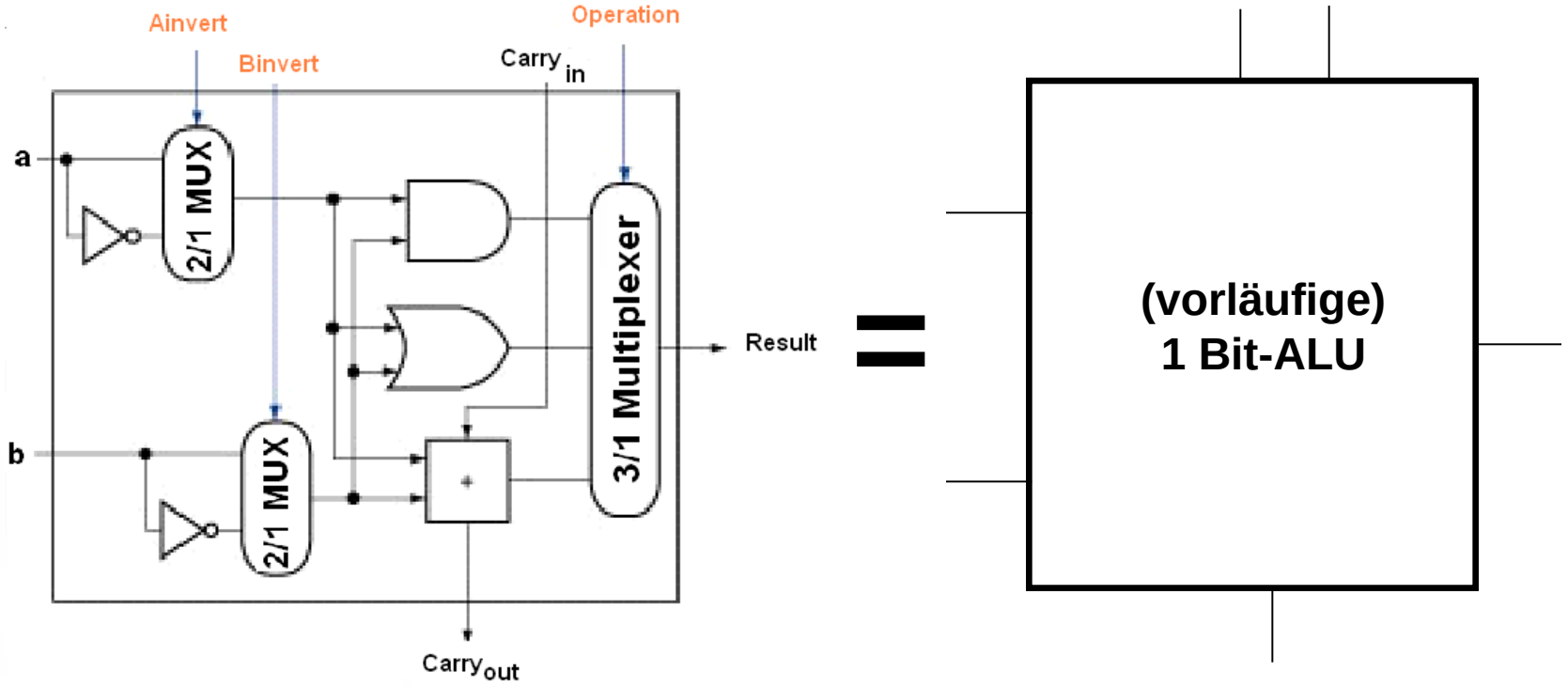
⇒ Wir brauchen einen Multiplexer (mit Steuereingang), damit man an einem Eingang des VA zwischen negiertem und nicht negiertem Wert wählen kann



Binvert = Wahl des Zweierkomplements

Die noch fehlende Addition von 1 wird später über das Carry-In-Bit implementiert !

Erweiterung um NOR Funktion und Zusammenfassung



⇒ **die Schaltung wird um eine NOR-Funktion erweitert.**

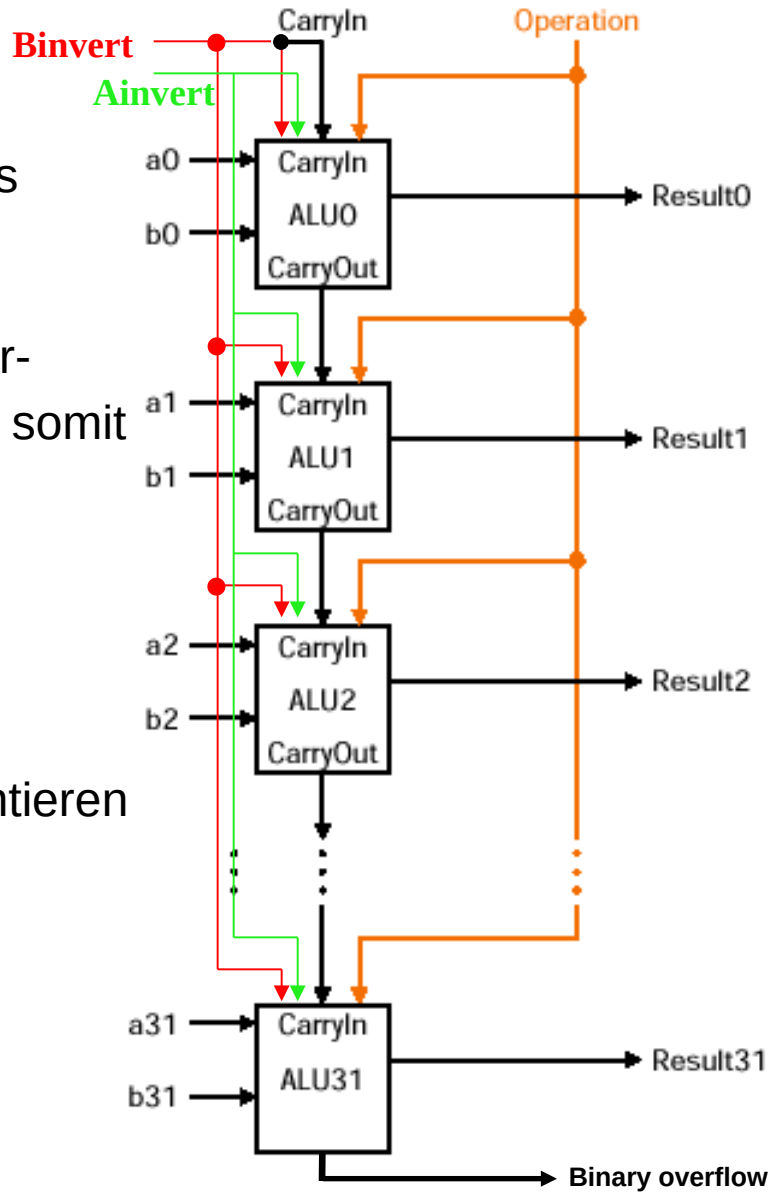
⇒ **hierzu fügt man einen Inverter und MUX am verbleibenden Eingang hinzu**

$$\text{NOR}: \bar{A} \wedge \bar{B} = A \bar{\vee} B$$

⇒ **Nächster Schritt: Kombination von 32 1- Bit ALUs zu einer einzigen 32 Bit ALU**

32-Bit ALU für AND, OR, ADD, SUB, NOR

⇒ Ainvert, Binvert, Operation gehen auf alle ALUs
⇒ Carries werden weitergereicht
⇒ $Carryout_{31}$ = binary Overflow (nicht benötigt)
⇒ $Carryin_0$ dient der Addition von 1 bei der Zweier-komplementbildung der Subtraktion und kann somit mit Binvert verbunden werden



Was fehlt ?

⇒ 2k-Overflow-Detektion

⇒ SLT, BEQ BNE kann man zusätzlich implementieren (2 Register werden verglichen und je nach Ergebnis ein Register auf 0 oder 1 gesetzt)

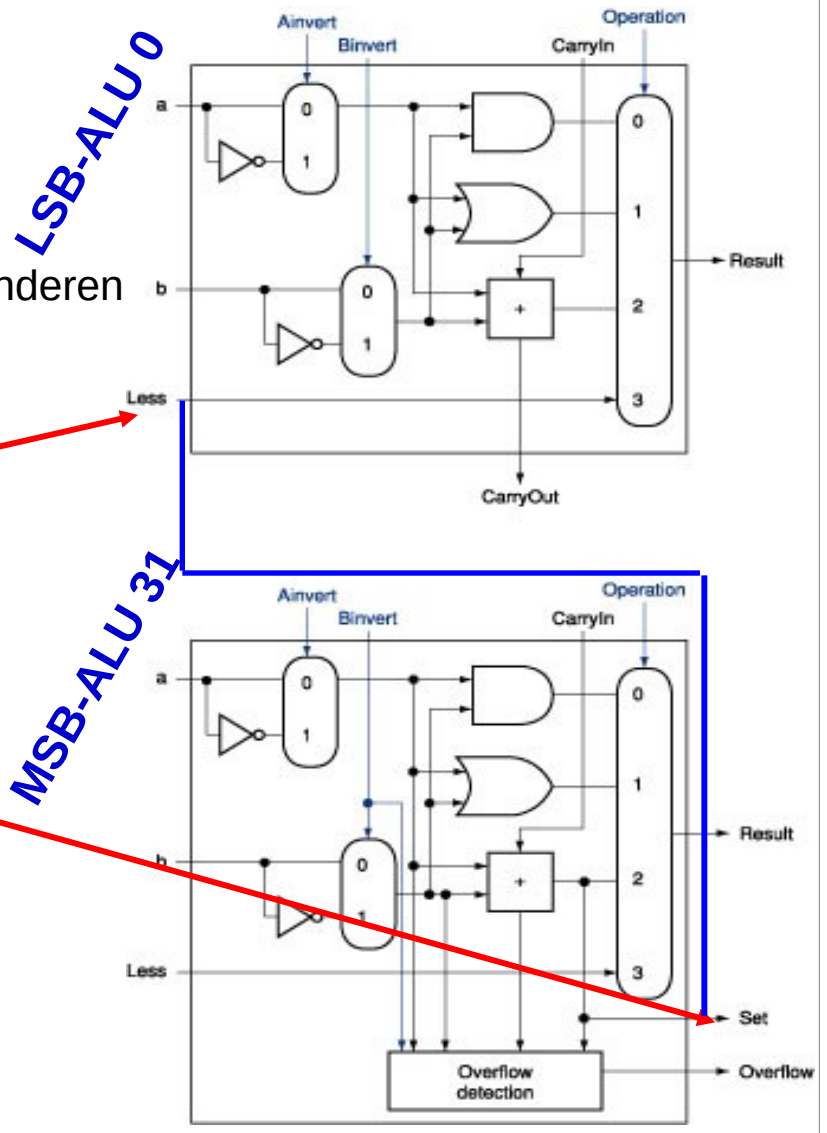
⇒ Nächster Schritt : Overflow, SLT, BEQ, BNE implementieren !

Implementierung von SLT

- ⇒ slt $t0, $s1, $s2 :
- ⇒ Berechne $s1-$s2.
- ⇒ Wenn negativ, setze LSB von $t0 auf 1 und alle anderen Bits von $t0 auf 0, wenn positiv, setze alle Bits von $t0 auf 0
- ⇒ Wie setzt man diese Bits ?
  - ⇒ Neues Signal “Less”

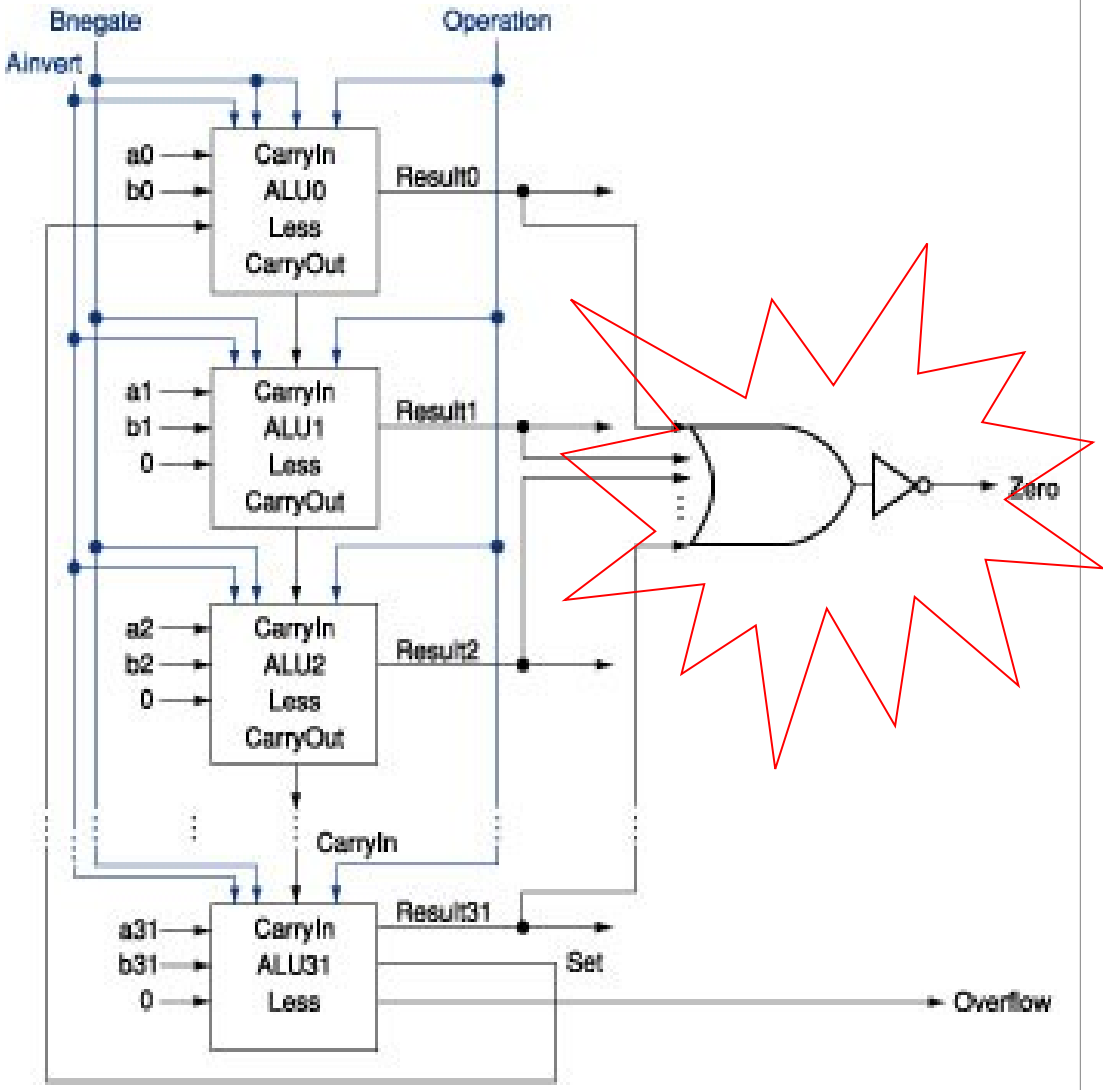
Hardware- Implementierung:
Vorzeichenbit (an ALU31) mit
Less verbinden (an ALU0) verbinden
Less von ALU 1...ALU31 auf “0”

Overflow detection :
Auswertung von a, b, Carry an ALU 31



Implementierung von BEQ BNE

- ⇒ MIPS Befehl
  beq $S1, $s2, label
- ⇒ Alle Bits von $s1 und $s2 über XOR (Volladdierer) in der ALU vergleichen und dann über ein OR auswerten
- ⇒ Zero-Flag setzen



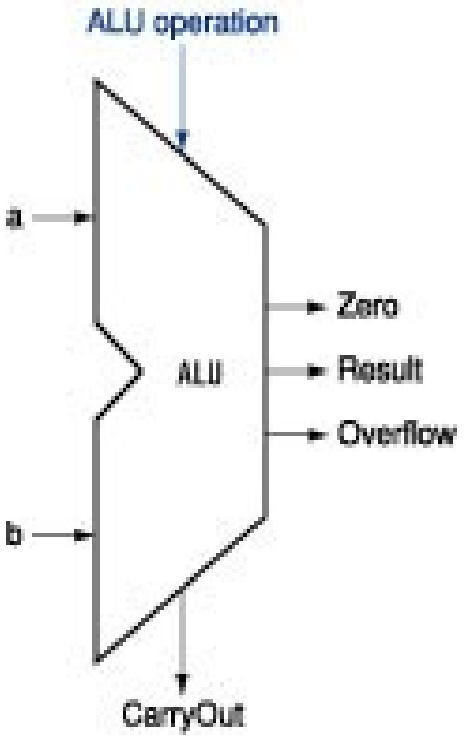
Fertige 32-Bit MIPS Light ALU

Operationen : and, or, nor, add, sub, slt, beq/bne

ALU Steuersignale: **2 Leitungen für and, or, add, und slt**

2 Leitungen für sub, nor, and slt

ALU Control Lines	Function
0000	And
0001	Or
0010	Add
0110	Sub
0111	Slt
1100	NOR



Rechnerarithmetik: Binär-System

- Zweier-Komplement: zur Addition negativer Zahlen
  - 1. alle Bits invertieren/negieren (Einer-Komplement)
  - 2. eine 1 addieren (Zweier-Komplement)
  - 3. links ein Vorzeichenbit anhängen
  - zB: 234 = 1110 1010
    -234 = 0001 0101 + 1 = 1 0001 0110

- Subtraktion: negative Zahlen in deren Zweier-Komplement miteinander addieren
  - zB: -234 = 1 0001 0110 (Zweier-Komplement)
    +234 = 0 1110 1010
    0 0000 0000 = 0

  - zB: 34 = 0010 0010; -34 = 1 1101 1101 + 1 = 1 1101 1110
    234 = 0 1110 1010
    -34 = 1 1101 1110 (Zweier-Komplement)
    200 = 0 1100 1000

Rechnerarithmetik: Binär-System

- Multiplikation: n_{10} x 2^k = n_2 k-mal links verschieben
  - zB: 234 x **2** = 1110 1010 << **1**
    = 1 1101 0100 = 468
  - zB: 234 x **4** = 1110 1010 << **2**
    = 11 1010 1000 = 936
- Multipliziererwerk: parallele Multiplikation beliebiger Zahlen
  - Basisoperationen: Zweier-Komplement; links verschieben

Multiplikation

- Einfache Multiplikationslogik:

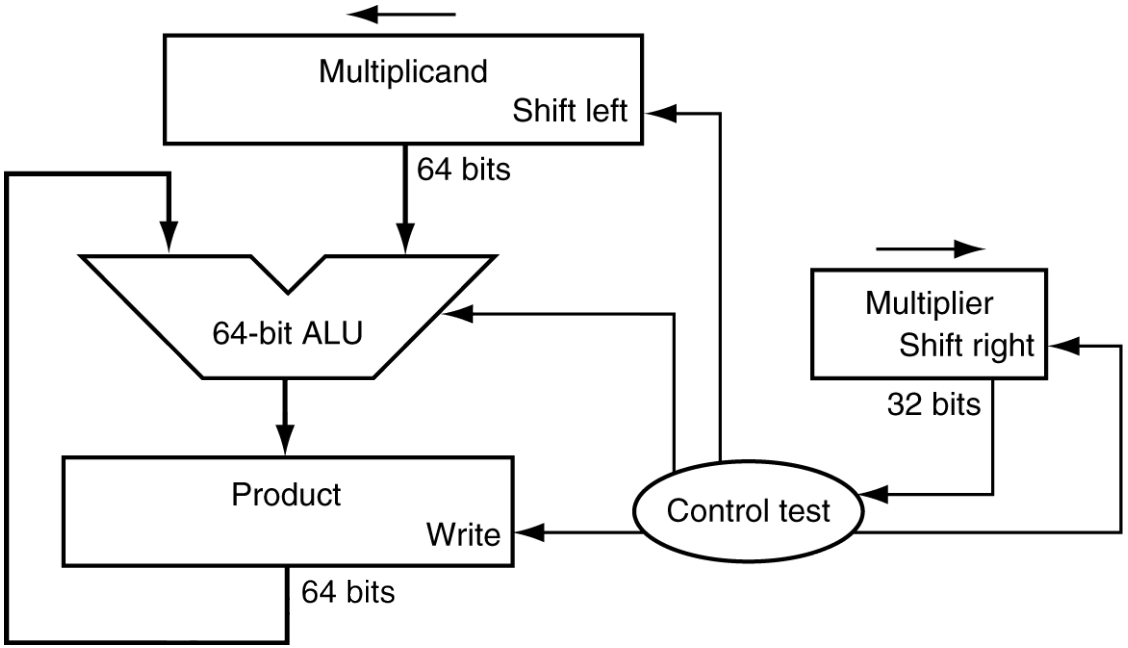
multiplicand

multiplier

product

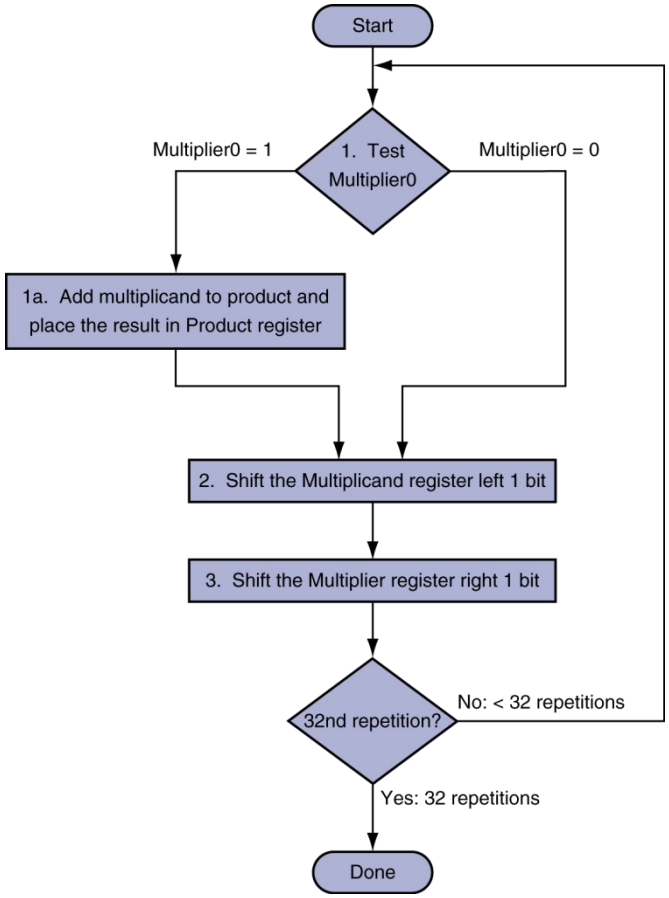
```
  1000
× 1001
------
  1000
 0000
0000
1000
------
1001000
```

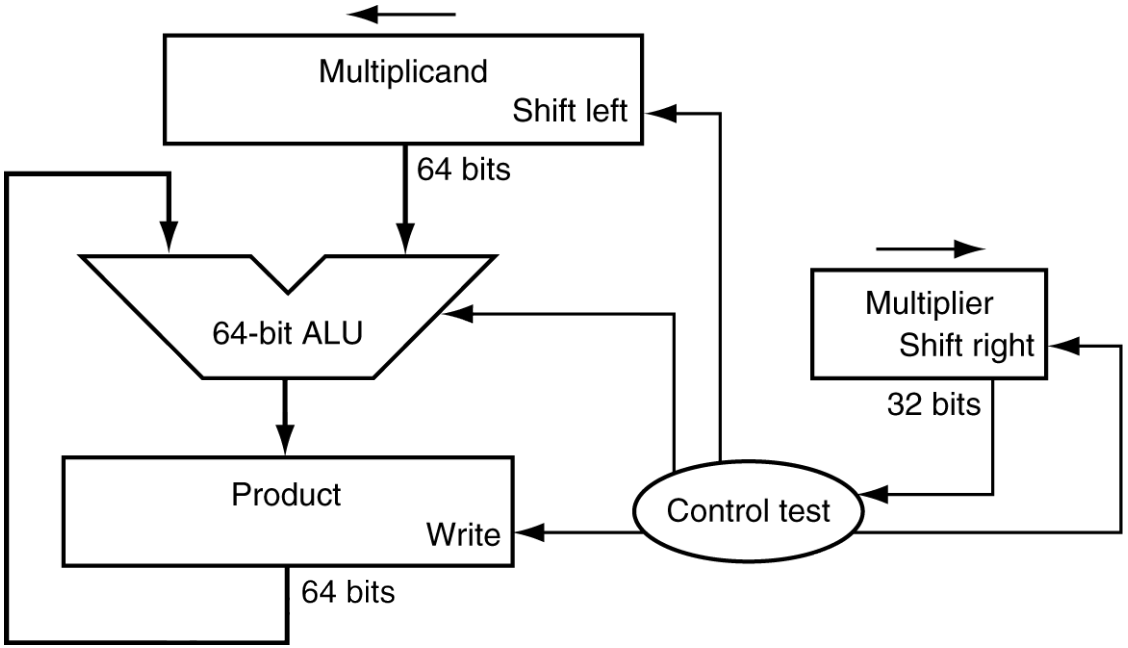
Length of product is the sum of operand lengths



Patterson/Hennessy

Multiplikation

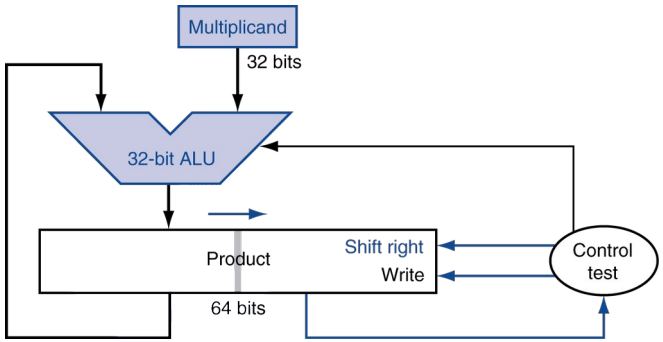


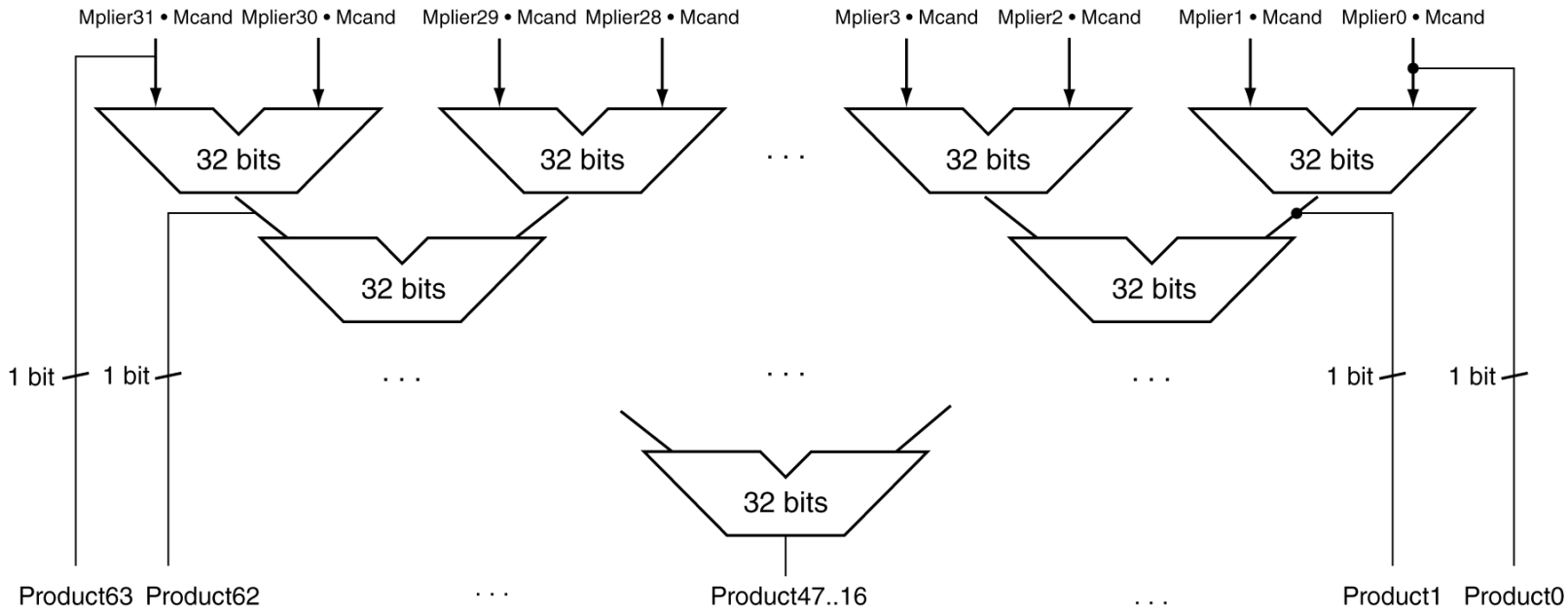


Patterson/Hennessy

Schnellere Multiplikation durch mehr Hardware

- Add/Shift parallel ausführen
- Mehrere Addierer parallel benutzen





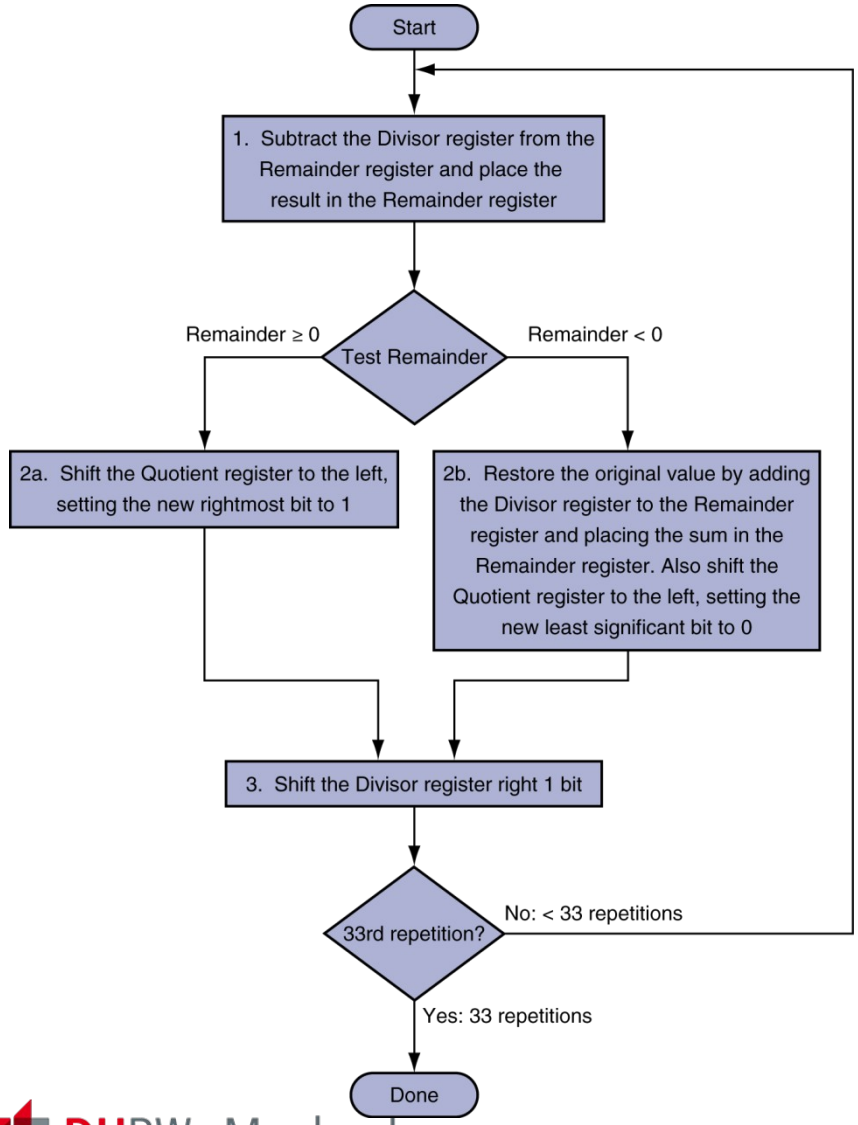
Patterson/Hennessy

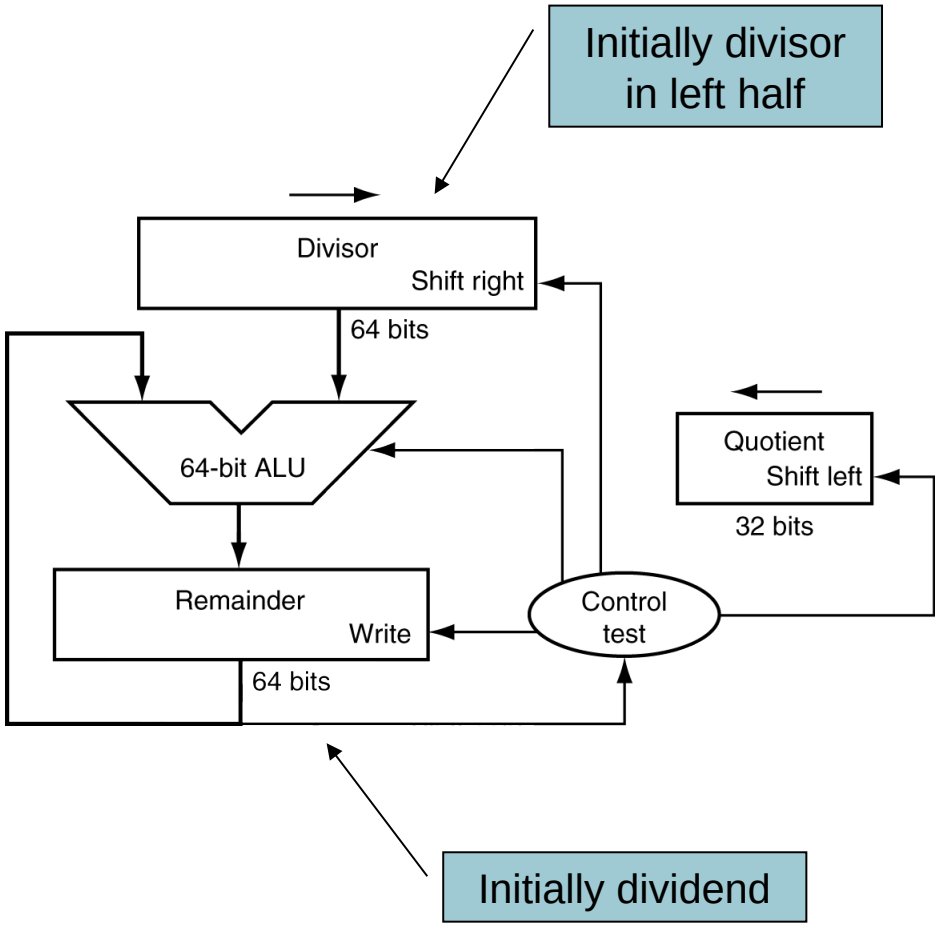
MIPS-Multiplikation

- Das Produkt wird in zwei speziellen 32-bit-Registern gespeichert:
  - HI: MSB (obere 32 bit)
  - LO: LSB (untere 32 bit)

- Instruktionen:
  - Zuerst Multiplikation t0 * t1 → HI/LO
    `mult t0, t1` oder `multu t0, t1`
  - HI nach t3 kopieren, LO nach t2
    ```
    mfhi t3
    mflo t2
    ```
  - Der Assembler unterstützt Makro mul für t2 = LSB (t0 * t1)
    `mul t2, t0, t1`

Division





Patterson/Hennessy

MIPS-Division

- Das Ergebnis wird wieder in zwei speziellen 32-bit-Registern gespeichert:
  - HI: 32bit Restwert
  - LO: 32bit Quotient

- Instruktionen:
  - Division t0 / t1 → HI = t0 mod t1, LO = t0 / t1

    ```
    div t0, t1
    ```
    oder `divu t0, t1`

  - Kein Overflow-Check oder Check auf Division durch 0!

Gleitkommazahlen

Rechnerarithmetik: Binär-System

- Rationale Zahl: umrechnen in Fest- oder Gleitkommazahl

- Festkommazahl: konstante Vor- und Nachkommastellen
  - festgelegt bei Definition des Datentyps
  - Prozessor-Unterstützung: nur wenn Gleitkommazahl nicht unterstützt wird
  - SW-Unterstützung: Unterschiedliche Notationen
  - Arithmetische Operationen komplizierter als bei Gleitkommazahlen

- Q-Notation: Qm.f ist Zweierkomplement Integer mit Vorzeichen
  - zB Q1.30:
    - 1 Bit Vorzeichen
    - m: 1 Bit Vorkommastellen = {0, 1}
    - f: 30 Bits Nachkommastellen im Zweierkomplement

- Standard IEEE 754

Rechnerarithmetik: Binär-System

- Gleitkommazahl: IEEE 754 Standard
  - Single Precision, 32 bit
  - Double Precision, 64 bit
- Format: $x = (-1)^s \times 2^e \times m$
  - ➔ S: Vorzeichen
  - ➔ E: Exponent (wird gespeichert mit „Bias“ +127 bei Single Precision oder +1023 bei Double Precision → negative Exponenten)
    Exponenten mit 1...1 sind reserviert für NaN und +/- Infinity
  - ➔ M: Mantisse / Fraction

➔ Durch Normalisierung ist 1. Bit immer 1, wird weggelassen

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Rechnerarithmetik: Binär-System

- IEEE 754: spezielle Gleitkommazahlen
- Floating Point Unit (FPU): implementiert IEEE 754

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0	0	0	0	true zero (0)
0	nonzero	0	nonzero	± denormalized number
± 1-254	anything	± 1-2046	anything	± floating point number
± 255	0	± 2047	0	± infinity
255	nonzero	2047	nonzero	not a number (NaN)

Floating-Point-Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

Patterson/Hennessy

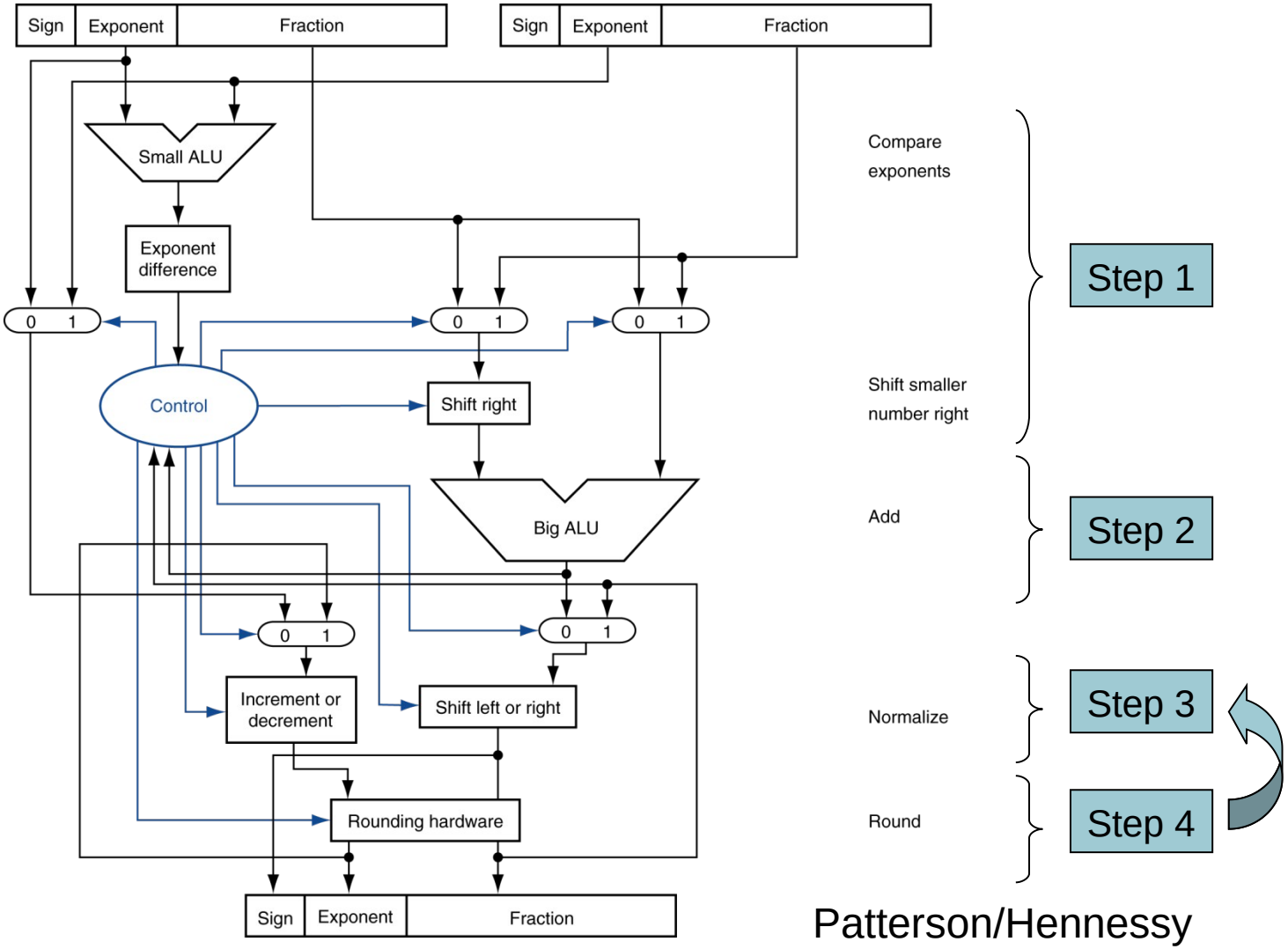
Floating-Point-Multiplikation

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Patterson/Hennessy

FP-Addition in Hardware

- Sehr kompliziert, braucht mehrere Clock-Cycles



Patterson/Hennessy

FP in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, ...)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Patterson/Hennessy

FP in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

Patterson/Hennessy

Rechnerarithmetik: Binär-Logik

Wahrheitstwerte für Junktoren der Binär-Logik										
Name										
Variable		Negation		Kon-junktion	Dis-junktion	Sub-junktion	Bij-unktion	NAND	NOR	XOR
x	y	¬x	¬y	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \leftrightarrow y$	$x \barwedge y \Leftrightarrow \neg x \vee \neg y$	$x \barvee y \Leftrightarrow \neg x \wedge \neg y$	$x \veebar y \Leftrightarrow \neg(x \leftrightarrow y)$
1	1	0	1	1	1	1	1	0	0	0
1	0	0	0	0	1	0	0	1	0	1
0	1	1	1	0	1	1	0	1	0	1
0	0	1	1	0	0	1	1	1	1	0

ASCII Tabelle (American Standard Code for Information Interchange)

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]