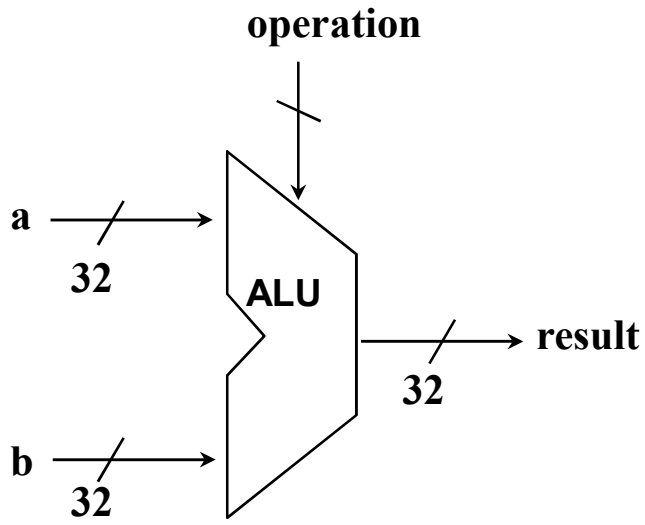
4. Computerarithmetik

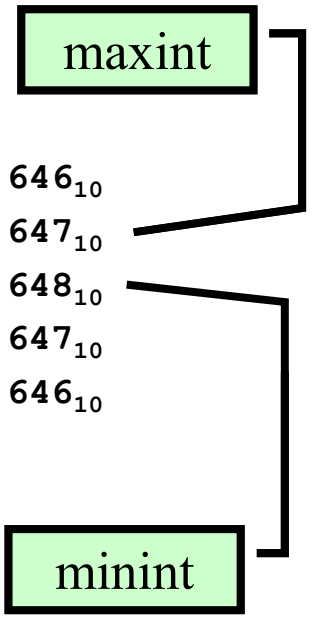
- **Arithmetische Operationen**
  - werden in ALU (Arithmetic Logic Unit) durchgeführt
  - **`operation`** wählt die durchzuführende Operation aus



MIPS Zweierkomplementdarstellung

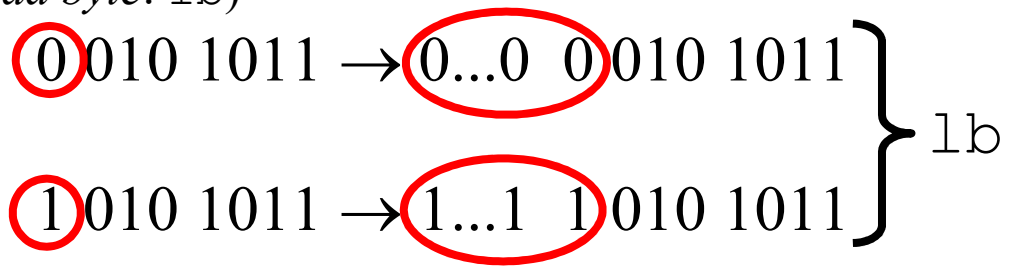
32 bit signed integers:

```
0000 0000 0000 0000 0000 0000 0000 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = + 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ = + 2₁₀
...
0111 1111 1111 1111 1111 1111 1111 1110₂ = + 2,147,483,646₁₀
0111 1111 1111 1111 1111 1111 1111 1111₂ = + 2,147,483,647₁₀
1000 0000 0000 0000 0000 0000 0000 0000₂ = − 2,147,483,648₁₀
1000 0000 0000 0000 0000 0000 0000 0001₂ = − 2,147,483,647₁₀
1000 0000 0000 0000 0000 0000 0000 0010₂ = − 2,147,483,646₁₀
...
1111 1111 1111 1111 1111 1111 1111 1101₂ = − 3₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = − 2₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = − 1₁₀
```

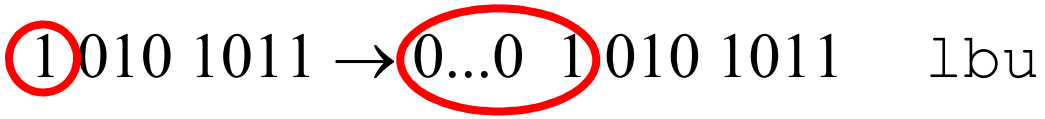


Zweierkomplement-Operationen

- Negieren eine Zweierkomplement-Zahl
  - invertiere alle Bits und addiere eine 1
  - merke: “negieren” und “invertieren” sind etwas ganz Verschiedenes!
- Konvertieren von *n*-bit Zahlen in mehr als *n* Bits
  - MIPS 16 bit immediate Zahlen werden in 32 bit für Arithmetik konvertiert
  - Kopiere das höchstwertige Bit (das Vorzeichenbit) in die dazukommenden Bits (z.B. bei *load byte*: `lb`)



  - „Vorzeichenerweiterung" für vorzeichenlose Zahlen (z.B bei *load byte unsigned*: `lbu`)



Addition & Subtraktion

- Zweierkomplement Addition
  - normale binäre Addition
  - ein evtl. entstehendes Carry-Bit wird ignoriert
- Subtraktion
  - Addition der negierten Zahlen
    - statt

```
  0111
- 0110
------
  0001
```

rechnet man

```
  0111          0111
+ 1010  bzw.  + 1001
------        +    1
 10001        ------
               10001
```

Addition & Subtraktion (2)

- Überlauf (*overflow*, Ergebnis zu groß für endliches Computerwort)
  - die Addition zweier *n*-bit Zahlen ist nicht immer mit *n* bit darstellbar:
  - Beispiel

```
  0111   7          1001   -7
+ 0001  +1        + 1011   -5
  1000  -8          10100    4
```

- Beachte
  - Bei der Addition zweier positiver Zahlen kann eine negative entstehen, bei der Addition zweier negativer eine positive.
  - Entsprechend bei der Subtraktion zweier Zahlen mit verschiedenen Vorzeichen

Überlauf-Erkennung

- kein Überlauf
  - Addition: Operanden haben unterschiedliche Vorzeichen
    - das Ergebnis liegt zwischen der negativen und der positiven Zahl, ist also darstellbar
  - Subtraktion: Operanden haben dasselbe Vorzeichen
    - wegen A-B = A + (-B) auf Addition zurückgeführt
- Überlauf entsteht durch Übertrag in das Vorzeichenbit hinein
- Überlaufbedingungen

Operation	Operand A	Operand B	Ergebnis
A+B	**≥ 0**	**≥ 0**	**< 0**
A+B	**< 0**	**< 0**	**≥ 0**
A-B	**≥ 0**	**< 0**	**< 0**
A-B	**< 0**	**≥ 0**	**≥ 0**

Überlauf-Erkennung (2)

- **Überlauferkennung mithilfe von**
  - Vorzeichen der Operanden
  - Art der Operation
  - Vorzeichen des Ergebnisses
  - bei vorzeichenlosen Zahlen ist Überlauf am Carry-Bit erkennbar

  $\Rightarrow$ einfaches Schaltnetz löst das Problem

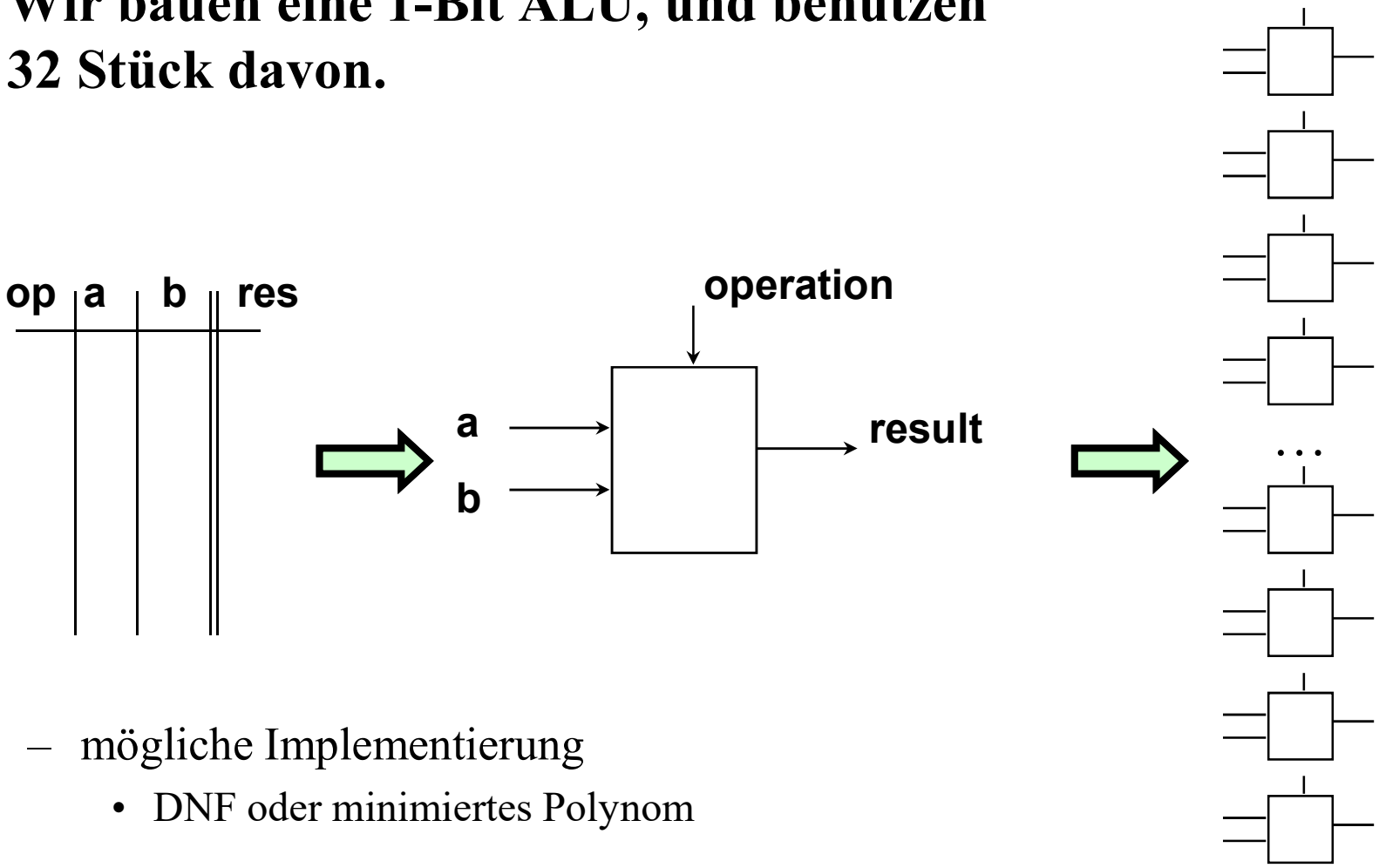
- **Betrachte die Operationen A + B und A – B.**
  - Kann Überlauf entstehen, falls B = 0 ?
  - Kann Überlauf entstehen, falls A = 0 ?

Was passiert bei Überlauf?

- Eine Ausnahmebehandlung (*exception*) wird angestoßen (Details s.u.).
  - Kontrolle springt an vordefinierte Adresse.
  - Rücksprungadresse wird gespeichert, damit dort nach Fehlerbehandlung evtl. weitergearbeitet werden kann.
  - Details hängen von Software (Betriebssystem/Programmiersprache) ab.
    - Nicht immer soll ein Überlauf auch entdeckt werden.
- Neue MIPS Instruktionen (`u` = *unsigned*)
  - `addu, addiu, subu, sltu, sltiu`
    - 32 bit Operanden werden als vorzeichenlose, also nicht-negative 32 bit Zahlen interpretiert
    - wie `add, addi, sub, slt, slti` nur ohne Exceptions
    - Achtung: `addiu` macht immer noch Vorzeichenerweiterung für den immediate Operanden! (Zweierkomplementzahl)
  - Beachte: es gibt keine MIPS Instruktionen `subi` oder `subiu`
    - `addi` bzw. `addiu` können statt dessen negative Zahlen addieren

ALU (*Arithmetic Logic Unit*)

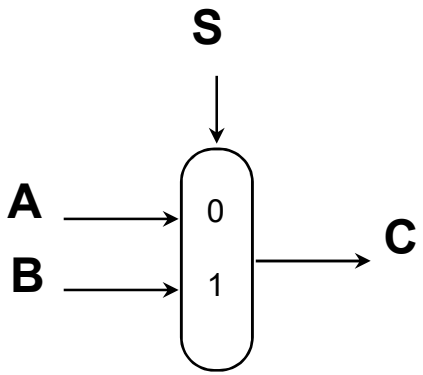
- **Wir bauen eine 1-Bit ALU, und benutzen 32 Stück davon.**



  - mögliche Implementierung
    - DNF oder minimiertes Polynom

Wiederholung: Multiplexer

- selektiert einen der Eingänge und reicht ihn zum Ausgang weiter, je nach Wertekombination an den Steuerungseingängen
- hier vereinfachtes Blockschaltbild

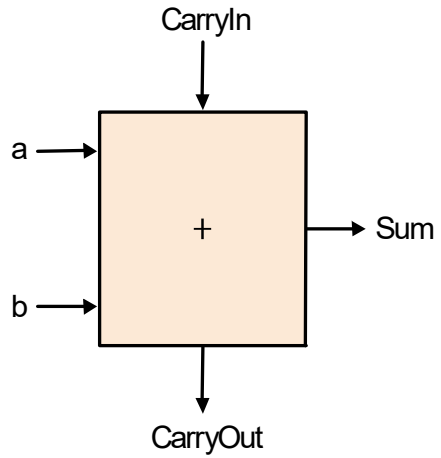


beachte: dies ist ein 2:1 MUX, obwohl er drei Eingänge hat

- Wir bauen unsere ALU mithilfe eines MUX.

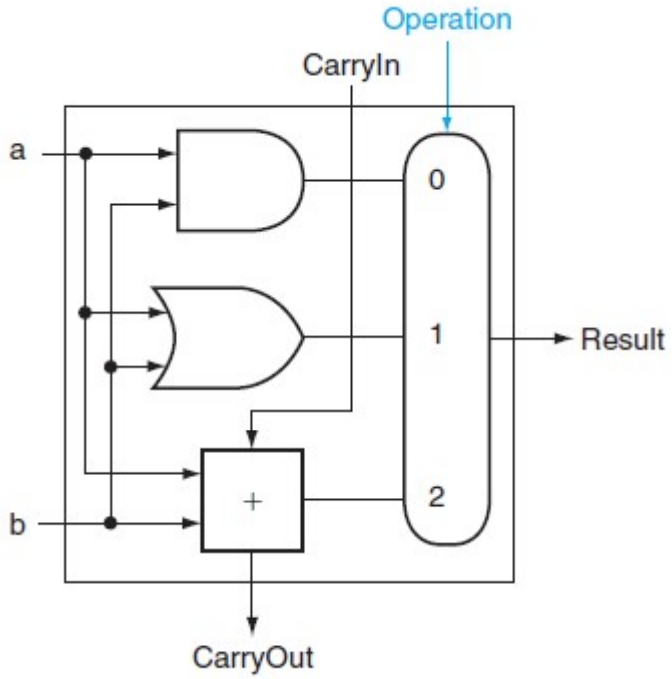
Verschiedene Implementierungen

- **es ist nicht einfach, die „beste“ Variante zu finden**
  - möglichst geringe Tiefe der Schaltung (Durchlaufverzögerung)
  - möglichst geringe Anzahl von Gattern (Kosten)
  - möglichst geringe Zahl von Eingängen an den Gattern (Kosten)
- **Übersichtlichkeit ist für uns im Moment am wichtigsten**
- **Volladdierer: 1-Bit ALU für Addition**

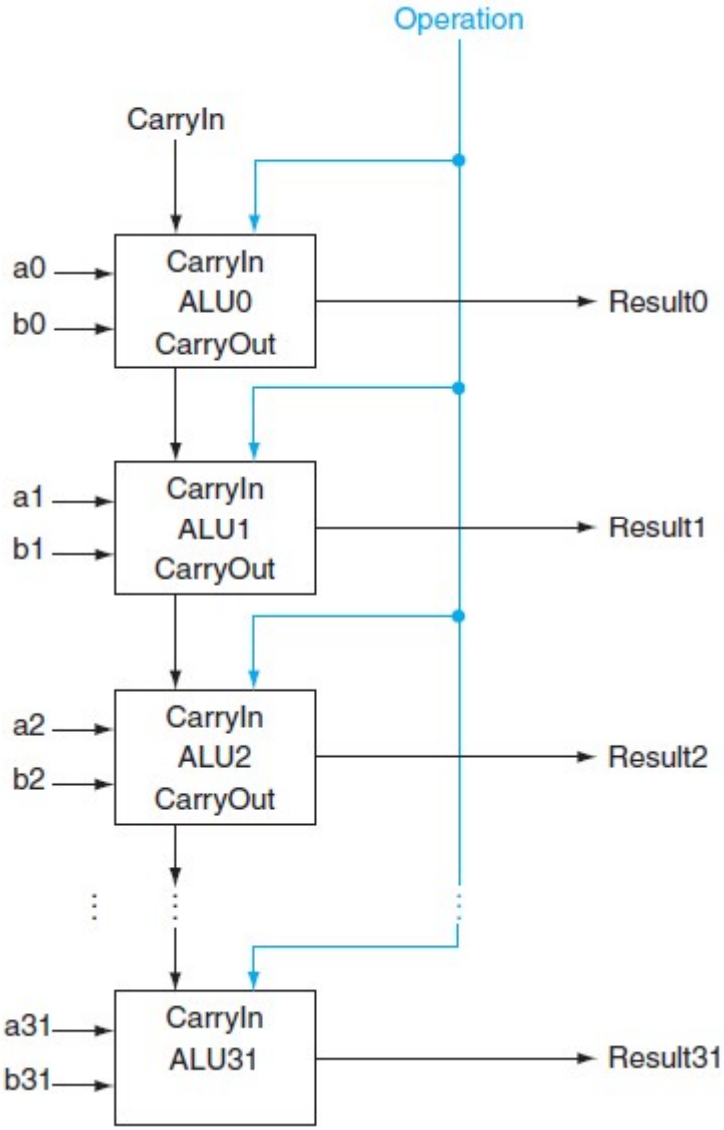


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

Konstruktion einer 32-Bit ALU

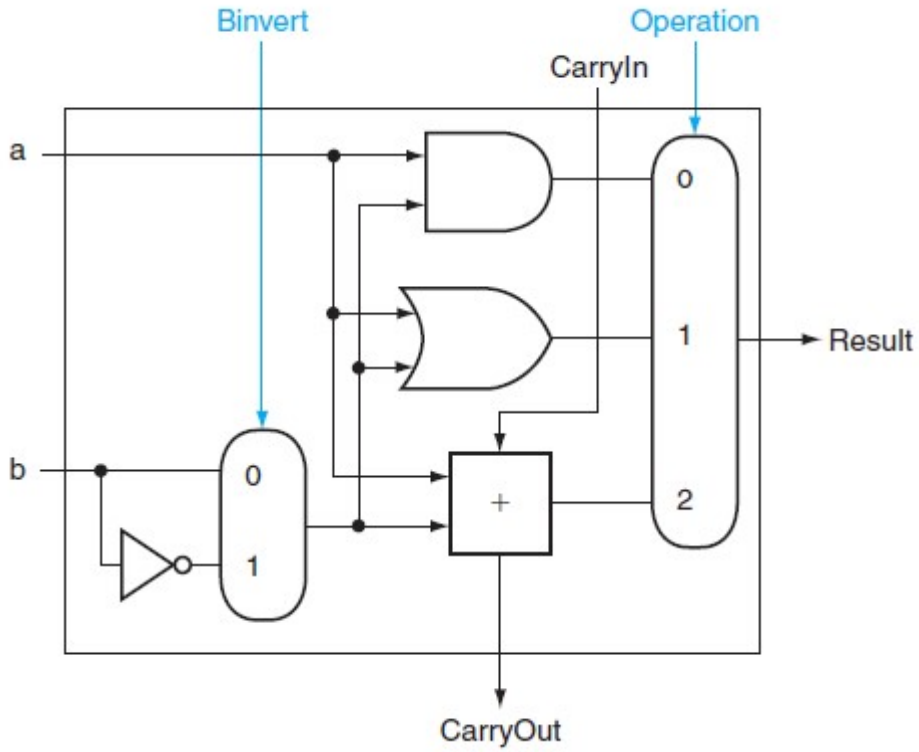


1-Bit ALU für AND, OR, ADD



Subtraktion (a – b)

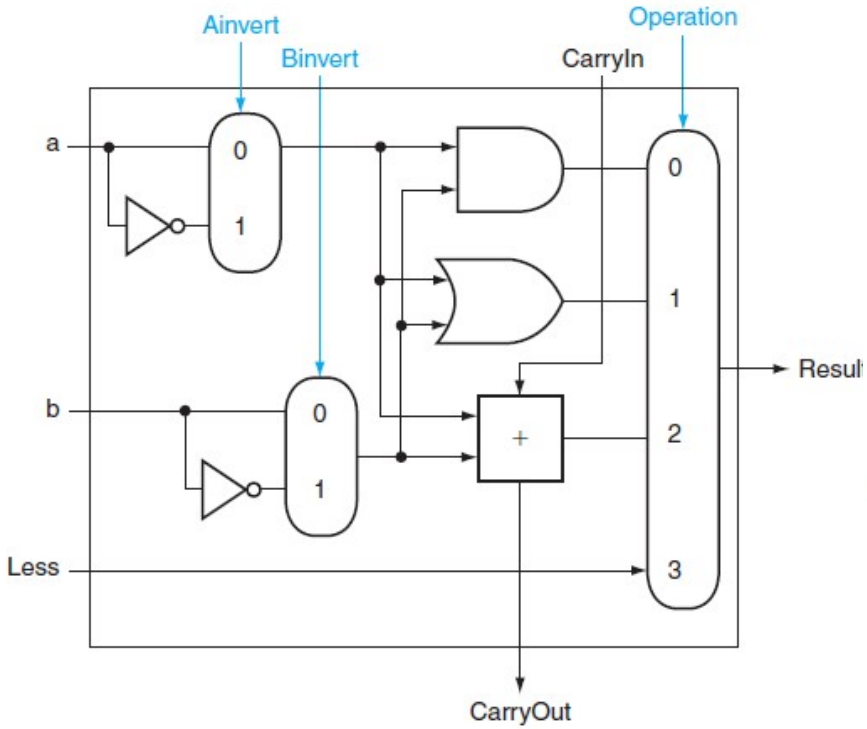
- benutze Zweierkomplement: negiere b und addiere zu a
- Wie wird negiert?
  - invertieren
  - 1 addieren (benutze CarryIn = 1)



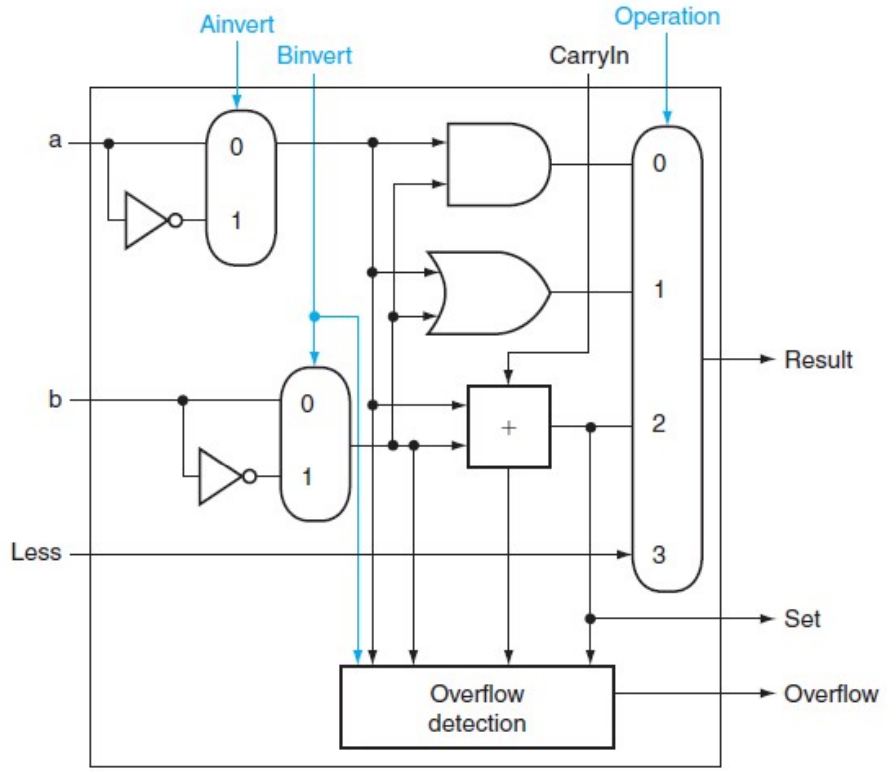
Anpassen der ALU an MIPS

- Unterstützung der set-on-less-than Instruktion
  - `slt $t1, $t2, $t3`
  - Erinnerung: `slt` ist eine arithmetische Instruktion
  - erzeugt eine 1 falls `$t2` < `$t3` und 0 sonst
  - benutze Subtraktion: (a-b) < 0 ist äquivalent zu a < b
- Unterstützung für Test auf Gleichheit
  - `beq $t5, $t6, 25`
  - benutze Subtraktion: (a-b) = 0 ist äquivalent zu a = b

Unterstützung von `slt`

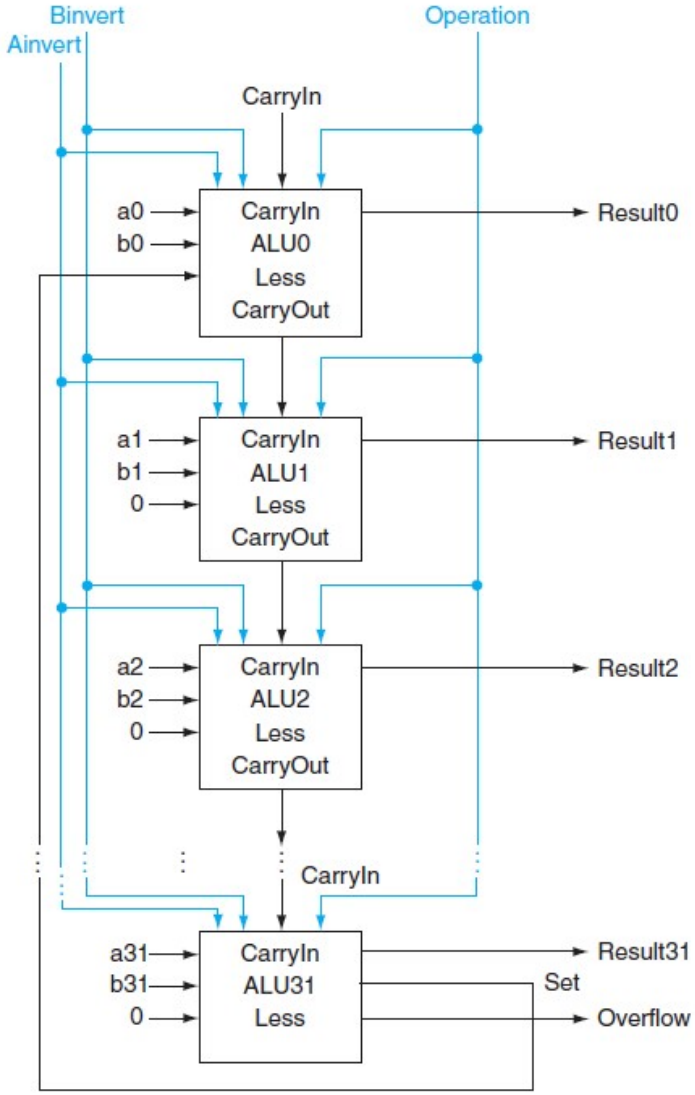


1-Bit ALU für die unteren 31 Bits
Extraeingang für das Resultat der `slt` Instruktion



1-Bit ALU für höchstwertiges Bit
(benötigen Vorzeichenbit des Resultats zum Setzen des niederwertigsten Bits)

32-Bit ALU mit Unterstützung für `slt`



Test auf Gleichheit: vollständige ALU

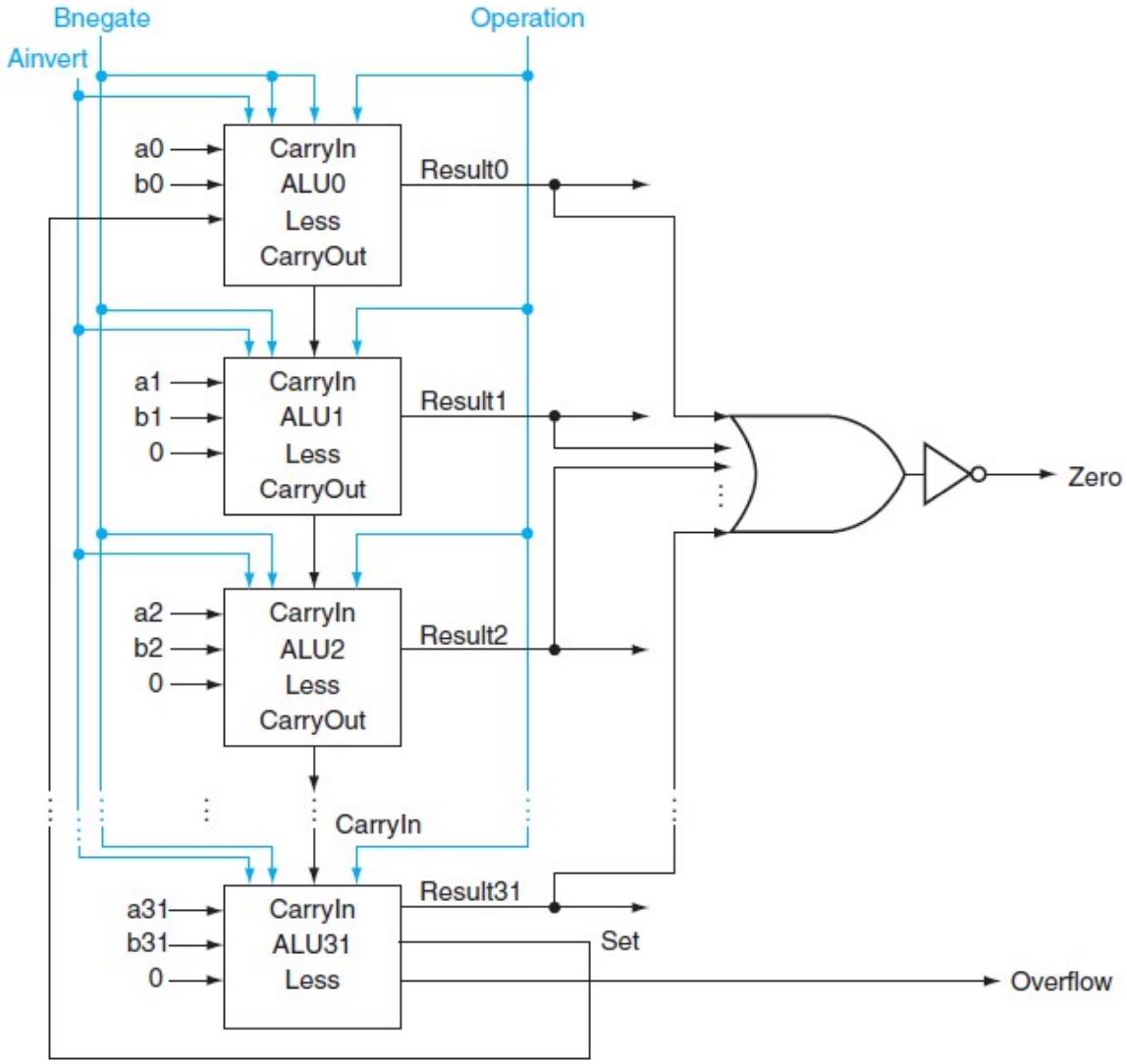
- **Steuerleitungen**

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

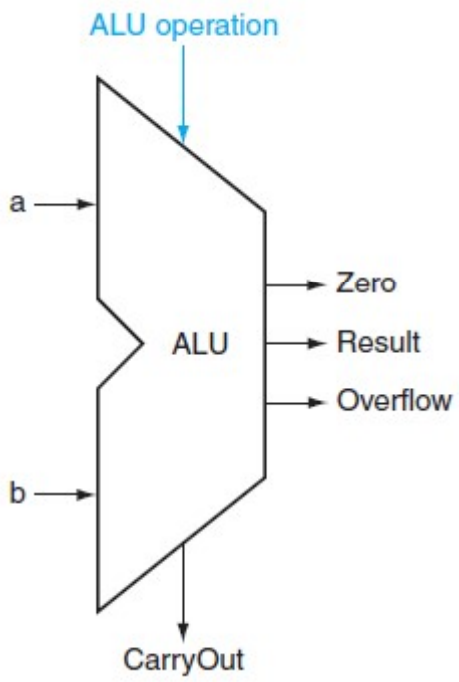
Operation

Bnegate

Ainvert



MIPS ALU



- **ALU operation:**

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

Abschließende Bemerkungen

- **Primäres Ziel: Verständnis**
  - einfache Architektur mit Multiplexer und Ripple-Carry Adder
  - nur wenige Instruktionen
- **Reale ALU**
  - Addition
    - Carry Look Ahead Addierer statt Ripple Carry Addierer
  - Multiplikation
    - Addierer-Baum mit Carry Save Addierer und einem Carry Look Ahead Addierer am Ende
  - viele weitere Instruktionen

Multiplikation in MIPS

- getrenntes Paar von 32-Bit Registern `Hi` und `Lo`
- zusammen ergeben sie das 64 Bit Produkt, das als Ergebnis der Multiplikation zweier 32 Bit Zahlen entsteht
- zwei Multiplikations-Instruktionen
  - `mult`: *multiply*
  - `multu`: *multiply unsigned*
- Zugriff auf Ergebnis über
  - *move from Lo*: `mflo`
  - *move from Hi*: `mfhi`
- MIPS Assembler enthält Pseudoinstruktion `mul` für Multiplikation mit drei normalen Registern, die `mflo` benutzt
  - sinnvoll bei kleinen Zahlen, deren Produkt mit 32 Bits darstellbar ist
  - dabei gibt es keinen Test auf Overflow
  - benutze selbst `mfhi`, um festzustellen, ob Überlauf stattgefunden hat, oder um die oberen 32 Bits des Ergebnisses weiter zu verwenden

Division in MIPS

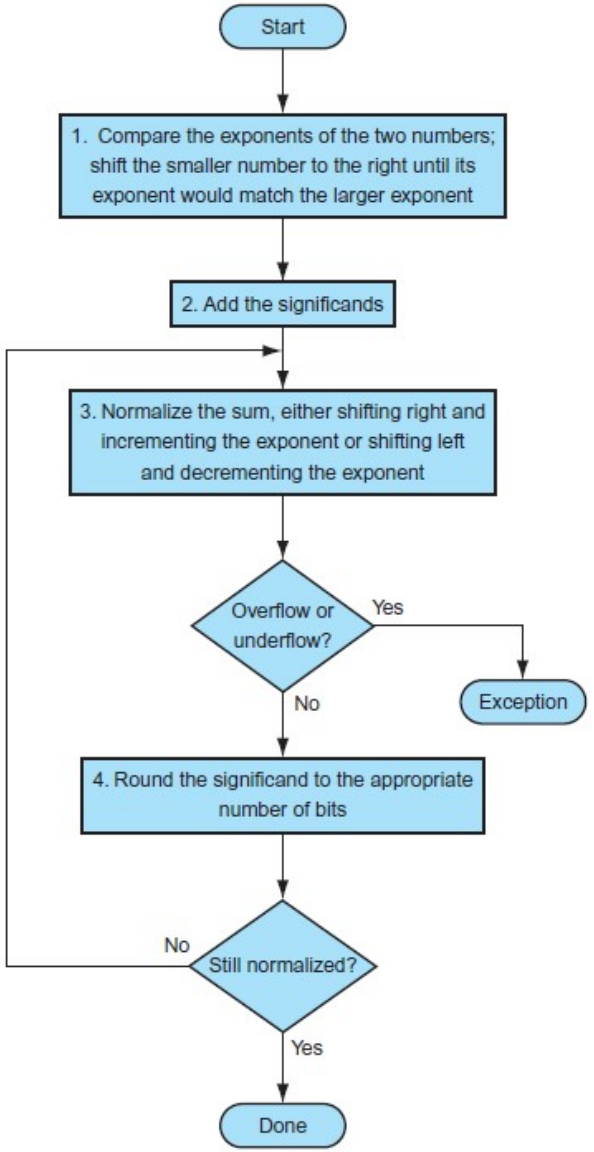
- **sehr ähnlich zur Multiplikation**
  - wie schriftliches Dividieren
    - Quotient- und Rest-Register
    - Subtraktion statt Addition
  - Besonderheit: es gibt Operanden, für die kein Ergebnis berechnet werden kann
    - Division durch 0 (dann Exception, s.u.)
- **MIPS**
  - Hardware identisch zur Hardware für die Multiplikation
  - Instruktionen: `div` und `divu`
    - `Hi` enthält den Rest
    - `Lo` den Quotienten

Gleitkomma-Operationen

- **Rechenoperationen sind kompliziert**
  - zusätzlich zu *overflow* können wir auch *underflow* haben
  - viele Sonderfälle
    - positive Zahl dividiert durch 0 ergibt "*infinity*"
    - 0 dividiert durch 0 ergibt "*not a number*"
- **Genauigkeit kann ein großes Problem sein**
  - Daten müssen wegen der Darstellung mit Signifikand immer normalisiert werden
  - nach dem Normalisieren muss gerundet werden
  - vier verschiedene Rundungs-Arten
  - beim Runden kann die Zahl wieder denormalisiert werden
  - erneutes Normalisieren notwendig

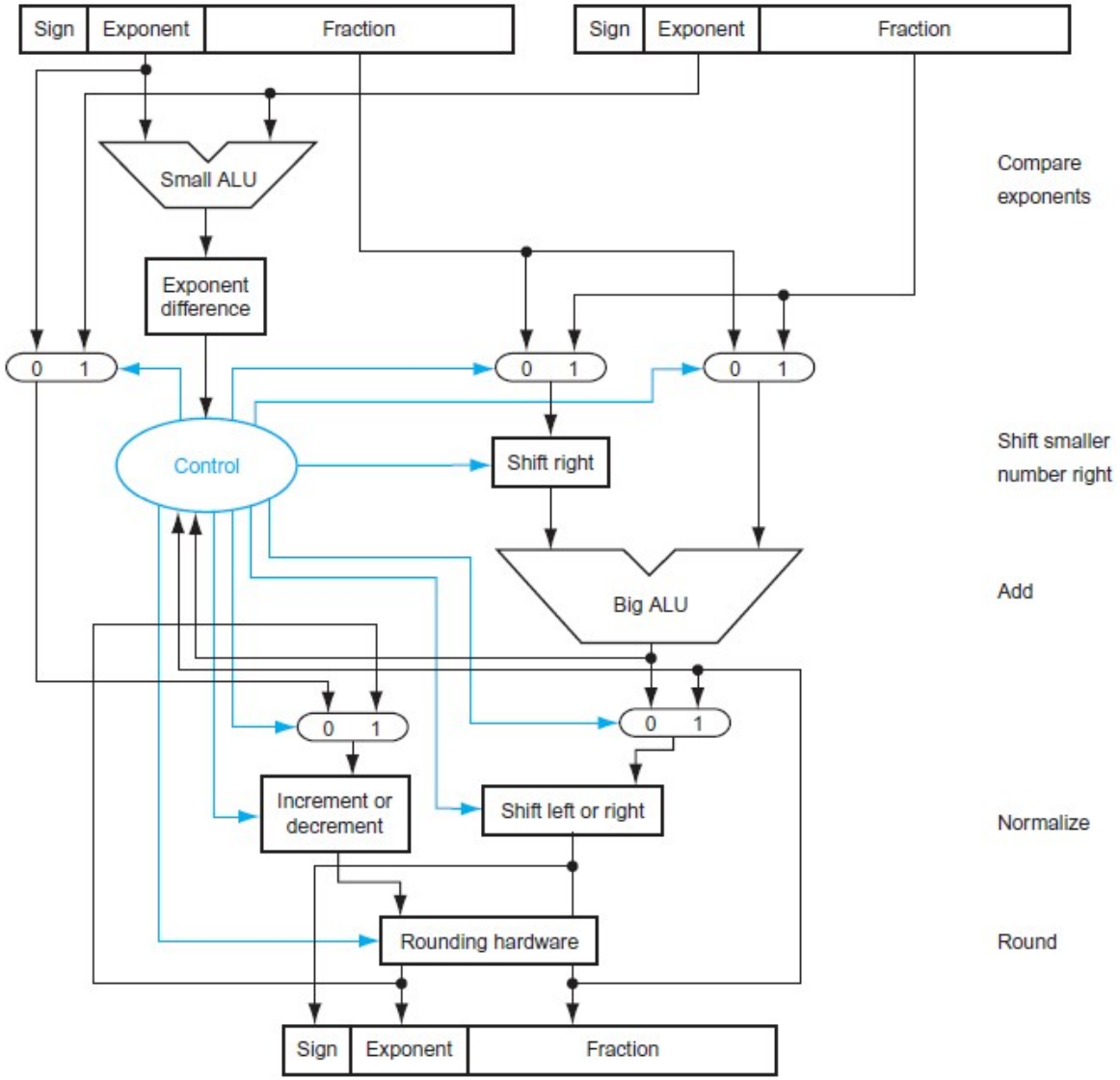
Addition von Gleitpunkt-Zahlen

- Normalerweise werden Schritte 3 und 4 nur einmal durchlaufen.
- Wenn nach dem Runden die Zahl nicht mehr normalisiert ist, muss noch einmal normalisiert werden.



Addition von Gleitpunkt-Zahlen (2)

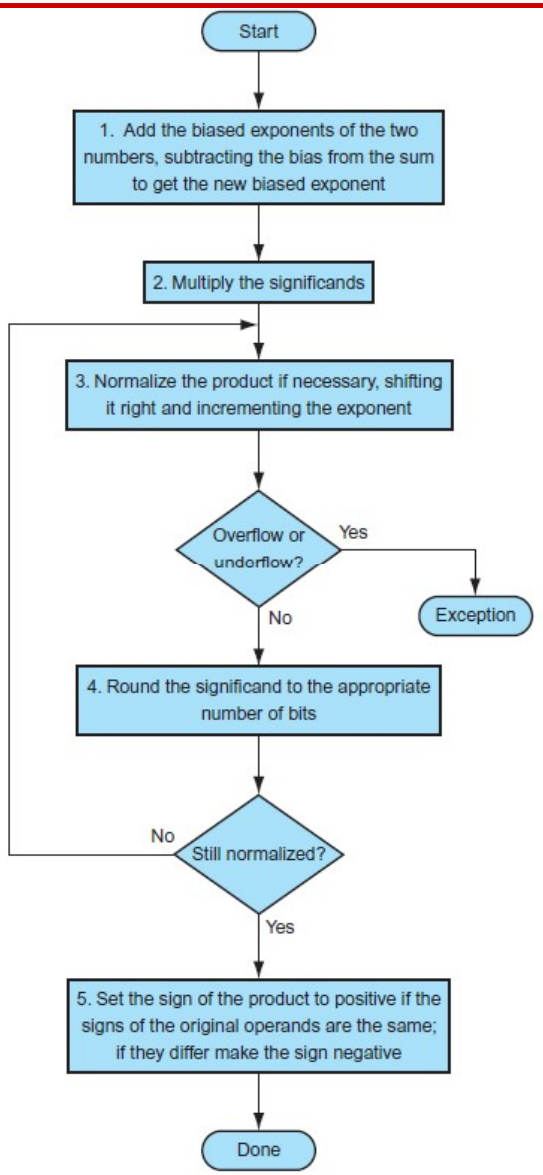
- Kleine ALU bestimmt Differenz der Exponenten.
- Multiplexer wählen aus:
  - größeren Exponenten
  - Signifikand der kleineren Zahl
  - Signifikand der größeren Zahl
  - kleinere Zahl wird nach rechts geschoben
- Normalisierung schiebt Summe nach rechts oder links und passt Exponent an



Multiplikation von Gleitpunkt-Zahlen

- das Schieben vor der Rechenoperation entfällt
  - Signifikanden werden direkt multipliziert
  - Exponenten werden addiert
- normalerweise werden Schritte 3 und 4 nur einmal durchlaufen
- wenn nach dem Runden die Zahl nicht mehr normalisiert ist, muss noch einmal normalisiert werden

- **Hardware wird analog aufgebaut**



Gleitkommazahlen in MIPS

- **Register**
  - eigene floating point register `$f0, …, $f31`
  - Paare von single precision Registern werden für double precision benutzt
    - z.B. `$f0, $f1`
- **single und double precision werden unterstützt, z.B.**
  - `add.s, add.d`
  - `sub.s, sub.d`
  - `mul.s, mul.d`
  - `div.s, div.d`
  - diverse Vergleichsoperationen
  - Sprünge, die von Vergleichen zweier Gleitkomma-Zahlen abhängen

Zusammenfassung

- Computer Arithmetik ist beschränkt durch begrenzte Genauigkeit
- Bitmuster haben keine inhärente Bedeutung, aber es existieren Standards
  - Vorzeichenlose Zahlen
  - Zweierkomplement
  - IEEE 754 Gleitkommazahlen
- Computer-Instruktionen bestimmen “Bedeutung” der Bitmuster
- Performance und Genauigkeit sind wichtig, daher sind reale Maschinen recht komplex (wegen der Standard-konformen Implementierung der Rechenoperationen)