Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

# Musterlösung 2
# Arithmetik 2.0 & Mips

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

## 1 Arithmetik

### Aufgabe 2.0: Bereichserweiterung

Betrachten Sie die Bitfolgen aus Tabelle 1. Ergänzen Sie die Tabelle indem sie die gegebenen Bitfolgen $b$ zu einer Folge $b'$ erweitern, sodass $b'$ eine Breite von 6 Bit hat. Beachten Sie hierbei, dass jeweils die Bedingungen über den entsprechenden Spalten gelten.

| $b$ | $\langle b \rangle = \langle b' \rangle$ | $[b] = [b']$ | $\langle b \rangle = [b']$ |
|---|---|---|---|
| 0010 | | | |
| 1001 | | | |
| 0111 | | | |
| 1111 | | | |
| 1011 | | | |

Tabelle 1: Zu erweiternde Bitfolgen.

### Lösung

| $b$ | $\langle b \rangle = \langle b' \rangle$ | $[b] = [b']$ | $\langle b \rangle = [b']$ |
|---|---|---|---|
| 0010 | 000010 | 000010 | 000010 |
| 1001 | 001001 | 111001 | 001001 |
| 0111 | 000111 | 000111 | 000111 |
| 1111 | 001111 | 111111 | 001111 |
| 1011 | 001011 | 111011 | 001011 |

Tabelle 2: Zu erweiternde Bitfolgen.

## Aufgabe 2.1: Shifts

Wie Sie in der Vorlesung gelernt haben, lässt sich die Multiplikation einer Bitfolge mit 2 durch einen Links-Shift ausdrücken. Ebenso lässt sich auch die ganzzahlige Division mit Shifts ausdrücken.

1. Geben Sie zu den gegebenen Bitfolgen jeweils eine Bitfolge $b'$, sodass $\langle b' \rangle = \langle b \rangle \cdot 4$ ist.

| $b$ | $\langle b' \rangle = \langle b \rangle \cdot 4$ | $\langle b' \rangle = \langle b \rangle \cdot 8$ |
|---|---|---|
| 0001001 | | |
| 0000011 | | |
| 0001110 | | |

2. Vervollständigen Sie unten stehende Tabelle.

| $b$ | $\langle b' \rangle = \lfloor \langle b \rangle \div 4 \rfloor$ | $[b'] = \lfloor [b] \div 4 \rfloor$ |
|---|---|---|
| 101101 | | |
| 010011 | | |
| 101010 | | |
| 111111 | | |

## Lösung

1. Vorgehen: Shiften Sie die Bitfolge um 2 (bzw. 3) nach links.

| $b$ | $\langle b' \rangle = \langle b \rangle \cdot 4$ | $\langle b' \rangle = \langle b \rangle \cdot 8$ |
|---|---|---|
| 0001001 | 0100100 | 1001000 |
| 0000011 | 0001100 | 0011000 |
| 0001110 | 0111000 | 1110000 |

2. Vorgehen: Shiften Sie die Bitfolge um 2 nach Rechts. Für vorzeichenbehaftete Bitfolgen bleibt das höchstwertige Bit erhalten, dh. es muss abhängig vom höchstwertigen Bit mit 0 oder 1 aufgefüllt werden.

| $b$ | $\langle b' \rangle = \lfloor \langle b \rangle \div 4 \rfloor$ | $[b'] = \lfloor [b] \div 4 \rfloor$ |
|---|---|---|
| 101101 | 001011 | 111011 |
| 010011 | 000100 | 000100 |
| 101010 | 001010 | 111010 |
| 111111 | 001111 | 111111 |

## Aufgabe 2.2: Vorzeichenbehaftete/Vorzeichenlose Arithmetik (signed/unsigned)

1. Geben Sie die Dezimaldarstellung der folgenden Binärzahlen an:
   a) $\langle 1010 \rangle$
   b) $[1010]$
   c) $\langle 010 \rangle$
   d) $[010]$
   e) $\langle 1111 \rangle$
   f) $[1111]$
   g) $[01111]$

2. Berechnen Sie:
   a) $\langle 111 \rangle + \langle 1110 \rangle$
   b) $\langle 111 + 1110 \rangle$
   c) $\langle 111 +_4 1110 \rangle$
   d) $[1110] - [1100]$
   e) $[1110 - 1100]$
   f) $[1110 -_4 1100]$
   g) $[100110] + [10000]$
   h) $[100110 + 10000]$
   i) $[100110 +_5 10000]$
   j) $[100110 +_6 10000]$

3. Berechnen Sie $[1101] + [0011]$, $[1101 + 0011]$ und $[1101 +_4 0011]$
   Was fällt Ihnen auf? Wie kommen diese Ergebnisse zustande?

## Lösung

1. a) $\langle 1010 \rangle = 10$
   b) $[1010] = -6$
   c) $\langle 010 \rangle = 2$
   d) $[010] = 2$
   e) $\langle 1111 \rangle = 15$
   f) $[1111] = -1$
   g) $[01111] = 15$

2. Berechnen Sie:
   a) $\langle 111 \rangle + \langle 1110 \rangle = 7 + 14 = 21$
   b) $\langle 111 + 1110 \rangle = \langle 10101 \rangle = 21$
   c) $\langle 111 +_4 1110 \rangle = \langle 0101 \rangle = 5$
   d) $[1110] - [1100] = -2 - -4 = 2$
   e) $[1110 - 1100] = [1110 + 0100] = [10010] = -14$
   f) $[1110 -_4 1100] = [0010] = 2$
   g) $[100110] + [10000] = -26 + -16 = -42$
   h) $[100110 + 10000] = [110110] = -10$
   i) $[100110 +_5 10000] = [10110] = -10$
   j) $[100110 +_6 10000] = [110110] = -10$

3. Berechnen Sie:
   $[1101] + [0011] = -3 + 3 = 0$
   $[1101 + 0011] = [10000] = -16$
   $[1101 +_4 0011] = [0000] = 0$
   kongruent modulo 16
   Overflow bei $+_4$

## Aufgabe 2.3: Rechnen

Wir betrachten die Interpretation von Binärzahlen als vorzeichenlose und vorzeichenbehaftete Zahlen. Vervollständigen Sie folgende Tabelle:

| $b \in \mathbb{B}^8$ | Hex | $\langle b \rangle$ | $[b]$ |
|---|---|---|---|
| 01101111 | | | |
| | 0xab | | |
| | | 127 | |
| | | | -128 |
| 11001001 | | | |
| | 0xff | | |
| | | 64 | |
| | | | 127 |

## Lösung

| $b \in \mathbb{B}^8$ | Hex | $\langle b \rangle$ | $[b]$ |
|---|---|---|---|
| 01101111 | 0x6f | 111 | 111 |
| 10101011 | 0xab | 171 | -85 |
| 01111111 | 0x7f | 127 | 127 |
| 10000000 | 0x80 | 128 | -128 |
| 11001001 | 0xc9 | 201 | -55 |
| 11111111 | 0xff | 255 | -1 |
| 01000000 | 0x40 | 64 | 64 |
| 01111111 | 0x7f | 127 | 127 |

## Aufgabe 2.4: Shifts

1. Geben Sie einen Ausdruck von Bitoperationen an, der, gegeben eine Bitfolge $a_{n-1} \cdots a_0$ und $i < j < n$, die Bitfolge $a_j \dots a_i$ extrahiert, d.h. die Bitfolge $0^{n-(j-i+1)} a_j \dots a_i$ produziert.

2. Geben Sie einen Ausdruck an, der aus zwei Bitfolgen $a_{n-1} \dots a_0$, und $b_{n-1} \dots b_0$ sowie $i \in [0, n-1]$ mit die Bitfolge $a_{i-1} \dots a_0 b_{n-1} \dots b_i$ liefert. Verwenden die ausschließlich Bitoperationen auf Bitfolgen der Länge $n$.

3. Geben Sie jeweils einen Ausdruck an, der eine Bitfolge $a$ mit 4, 5, und 31 multipliziert.

## Lösung

1. Da $n > j > i$, hat die Bitfolge die Gestalt $a_{n-1} \dots a_j \dots a_i \dots a_0$ Das Ziel ist nun, die gewünschte Bitfolge an den rechten Rand zu schieben. Vorher muss aber sichergestellt werden, dass „unerwünschte“ Bits auf der linken Seite entfernt werden. Daher schieben wir zunächst nach links, und zwar so lange, bis $a_j$ ganz links steht. Das erreichen wir durch den Ausdruck (a bezeichne die gesamte Bitfolge) $(a \ll_n (n-1-j))$. Jetzt haben wir alles Überflüssige abgeschnitten und können die Bitfolge an den rechten Rand schieben. Dabei müssen wir den vorherigen Shift wieder rückgängig machen. Wir erhalten also

$$\underbrace{(a \ll_n (n-1-j))}_{\text{vorheriges Ergebnis}} \gg_n (\underbrace{(n-1-j)}_{\text{vorheriger Shift}} + \underbrace{i}_{\text{eigentlicher Rechtsshift}})$$

2. In der Bitfolge mit a soll nach links geschoben werden, also wird wieder multipliziert, diesmal um $n - i$ Stellen. Analog dazu muss in der Bitfolge von b nach rechts geshiftet werden, und zwar um $i$ Stellen. Dann

hängt die Teilfolge von a auf der linken Seite und die Teilfolge von b auf der rechten Seite, ohne dass sie sich überschneiden. Um sie zusammenzufügen, reicht es, das bitweise Oder zu benutzen, da die Reste jeweils mit 0 aufgefüllt wurden. Im Ganzen erhält man also

$$(a \ll_n n - i) \mid_n (b \gg_n i)$$

3. Die Idee ist hier jeweils, die gewünschte Multiplikation in eine Summe von Multiplikationen mit Zweierpotenzen zu zerlegen, weil die durch Bitshifts durchgeführt werden können.

   - Hier ist fast nichts zu tun, da 4 schon eine Zweierpotenz ist. Mit $4 = 2^2$ entspricht die Multiplikation dann einem arithmetischen Linksshift um 2. Die Lösung lautet dann: $a \ll_n 2$
   - 5 lässt sich mit $5 = 4+1 = 2^2+1$ zerlegen, es muss also einmal der Shift um 2 nach links durchgeführt werden und dann muss die Bitfolge noch einmal dazuaddiert werden. Die Lösung lautet demnach: $(a \ll_n 2) +_n a$
   - Mit $31 = 16 + 8 + 4 + 2 + 1 = 2^4 + 2^3 + 2^2 + 2^1 + 1$ ergeben sich also insgesamt 4 Shifts und die eigentliche Bifolge, die zusammenaddiert werden müssen. Die Lösung lautet also:

     $$(a \ll_n 4) +_n (a \ll_n 3) +_n (a \ll_n 2) +_n (a \ll_n 1) +_n a$$

     Kürzer ist eine Sequenz, die $-$ nutzt, da $31 = 32 - 1$:

     $$(a \ll_n 5) -_n a$$

## Aufgabe 2.5: Das Vorzeichenbit

Zeigen Sie, dass für alle Bitfolgen $b_{n-1} \cdots b_0 \in \mathbb{B}^n$ gilt:

$$b_{n-1} = 1 \quad \text{genau dann wenn} \quad [b] < 0$$

## Lösung

Sei $b \in \mathbb{B}^n$.

**Hinrichtung:** $b_{n-1} = 1 \Rightarrow [b] < 0$

$$\begin{aligned} [b] &= -\langle b_{n-1} \rangle \times 2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle \\ &= -2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle && |b_{n-1} = 1 \\ &< 0 && |\text{Lemma 1.2} \end{aligned}$$

Lemma 1.2: $1 + \langle \underbrace{1 \cdots 1}_{n-1} \rangle = 2^{n-1}$, daraus folgt $\langle b_{n-2} \cdots b_0 \rangle < 2^{n-1}$.
Alternativ kann man auch direkt Lemma 1.3 anwenden.

**Rückrichtung:** $b_{n-1} = 1 \Leftarrow [b] < 0$
Wenn $[b] < 0$, dann muss $b_{n-1} = 1$, da $b_{n-1}$ der einzige negative Summand ist.

## Aufgabe 2.6: ASCII

Der ASCII-Zeichensatz umfasst die folgenden 95 druckbaren Zeichen[1]

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

und weitere 33 nicht druckbare Steuerzeichen. Insgesamt gibt es 128 ASCII-Zeichen. Typischerweise kodiert man diese mit je einem Byte, die definierende Tabelle findet sich im Anhang des Skripts.
Einen Text, auch Zeichenkette genannt, stellt man als Reihung von Bytes dar, deren Wert dem ASCII-Wert des jeweiligen Buchstabens entspricht. Um das Ende einer Zeichenkette zu kennzeichnen, ist eine mögliche und gängige Art, sie mit einem Byte mit dem Wert 0 abzuschließen. Eine so kodierte Zeichenkette nennt man *C string* oder ASCIIZ-Zeichenkette.

1. Wozu benötigt man Zeichenkodierungen?
2. Welche Zeichenkodierungen kennen Sie?
3. Wird ein gesamtes Byte benötigt, um ein ASCII-Zeichen zu speichern, oder reichen bereits weniger Bits aus?
4. Sie erhalten folgende ASCIIZ-Zeichenkette:

   ```
   072 097 108 108 111 032 087 101 108 116 033 013 010 000
   ```

   Übertragen Sie diese Zeichenkette in ein lesbares Format. Sie können dazu die ASCII-Tabelle im Anhang des Skripts verwenden.
5. Wir fassen die Kodierung als Bijektion $\mathbb{A} \to [0, 127]$ zwischen der Menge der ASCII Zeichen $\mathbb{A}$ und dem Intervall $[0, 127]$ auf.

   Geben Sie einen Ausdruck an, der …

   (a) abhängig von einer Zahl $d$ im Intervall $[0, 9]$ den zugehörigen ASCII-Code liefert.

   (b) abhängig von der Position (A ist Position 0) $p$ eines Buchstabens im Alphabet den ASCII-Code des zugehörigen Großbuchstabens liefert.

   (c) abhängig vom ASCII-Code eines Großbuchstabens den ASCII-Code des zugehörigen Kleinbuchstabens liefert.

   (d) ASCII-Code eines Buchstabens die Groß-/Kleinschreibung wechselt. Verwenden Sie weder Addition, Subtraktion noch Konditional.

## Lösung

1. - Notwendigkeit der Zuordnung von z. B. Bitfolgen zu einer verständlichen Bedeutung
   - Standardisierte Kommunikation
2. ASCII, UTF-8, Unicode, Morsezeichen, QR-Code, …
3. Es reichen 7 Bit für ein ASCII-Zeichen aus.
4. “Hallo Welt!”
   Hinweis: *CR LF* wird auf DOS- & Windows-Betriebssystemen verwendet, um einen Zeilenumbruch zu kodieren. Unix- bzw. Linux-basierte Betriebssysteme sowie macOS (seit Mac OS X) verwenden hingegen nur *LF*, um einen Zeilenumbruch zu kodieren[2].

[1] Quelle: `https://de.wikipedia.org/wiki/ASCII`
[2] Quelle: `https://de.wikipedia.org/wiki/Zeilenumbruch#ASCII_und_EBCDIC`

5. (a) Ansatz: Die Zahlen 0–9 beginnen in der ASCII-Tabelle ab ASCII-Code 48. Wir addieren also 48 zu unserer Eingabe und erhalten den entsprechenden ASCII-Code:

$$d + 48 \text{ mit } d \in [0, 9]$$

(b) Ansatz: Die Großbuchstaben beginnen in der ASCII-Tabelle ab ASCII-Code 65. Wir addieren also 65 (Index beginnt bei 0) zu unserer Eingabe und erhalten den entsprechenden ASCII-Code:

$$p + 65 \text{ mit } p \in [0, 25]$$

(c) Ansatz: Die Kleinbuchstaben beginnen in der ASCII-Tabelle genau 32 Stellen nach Beginn der Großbuchstaben. Wir addieren also 32 zu unserer Eingabe und erhalten den entsprechenden ASCII-Code:

$$f(b) = b + 32 \text{ mit } b \in [65, 90]$$

(d) Ansatz: Betrachtet man die Binärdarstellung der ASCII-Codes, dann fällt auf, dass sich die Darstellungen von Groß- und Kleinbuchstaben nur durch den Zustand von Bit 5 unterscheiden.
Sei $b$ ein gültiger ASCII-Code eines Buchstabens, dann gilt:

- $b \,\&\, 32 = 00000000 \implies b$ ist ein Großbuchstabe.
- $b \,\&\, 32 = 00100000 \implies b$ ist ein Kleinbuchstabe.

Um nun zwischen den Darstellungen von Groß- und Kleinbuchstaben zu wechseln invertieren wir Bit 5 mithilfe von $\hat{}$:

$$b \hat{} 32$$

# 2 Mips

## Aufgabe 2.7: Mips - Erste Schritte mit MARS

Machen Sie ihre ersten Schritte mit MARS:

1. Laden Sie die Zahl 5 in das Register `$t0`.
2. Laden Sie die Zahl 11 in das Register `$t1`.
3. Speichern Sie das Ergebnis der Addition der beiden Zahlen in `$t2`.
4. Subtrahieren Sie die Zahl im Register `$t0` von der im Register `$t1`. Speichern Sie das Ergebnis in `$t3`.
5. Subtrahieren Sie die Zahlen nun in umgekehrter Reihenfolge. Speichern Sie das Ergebnis in `$t4`.
6. Geben Sie das letzte Ergebnis aus
7. Überschreiben Sie die Register `$t2` - `$t4` mit dem Wert 0, ohne *li* zu bentzen. Finden Sie zwei verschiedene Möglichkeiten.

## Lösung

1. `li $t0 5`
2. `li $t1 11`
3. `add $t2 $t1 $t0`
4. `sub $t3 $t1 $t0`
5. `sub $t4 $t0 $t1`

6. 
```
li $v0 1
move $a0 $t4
syscall
```

7. 3 Möglichkeiten:

```
move $t2 $zero
addiu $t3 $zero 0
sub $t4 $t4 $t4
```

## Aufgabe 2.8: Iterativer Euklidischer Algorithmus

Übersetzen Sie das Folgende, im Pseudo-Code geschriebene Unterprogramm in Mips-Code:

```
1 EUCLID(a,b)
2     solange b != 0
3         h = a mod b
4         a = b
5         b = h
6     return a
```

Bei Schwierigkeiten findest Sie hier [3] ein Erklärvideo zu dem Übersetzen von Pseudo-Code in Mips-Code.

## Lösung

Vorgehen:

1. Zunächst wird hier die Funktion `EUCLID` definiert. Sie nimmt zwei Argumente.
2. In der Funktion werden insgesamt drei Variablen verwendet. Diese werden im Mips-Code in Registern gespeichert. Wir weisen den Variablen nun Registern zu, und zwar wie folgt:

| | |
|---|---|
| a | $a0 |
| b | $a1 |
| h | $t0 |

   Da `a` und `b` Argumente sind, müssen sie in den entsprechenden Registern stehen. `h` ist eine lokale Variable und damit in einem der für temporäre Werte vorgesehenen Registern unterzubringen.
3. Dann gibt es eine Schleife - wir müssen also Code schreiben, der dafür sorgt, dass der "Inhalt" der Schleife wiederholt ausgeführt wird, solange die Bedingung im "Kopf" der Schleife wahr ist. Dafür gibt es zwei Möglichkeiten, einmal kann man die Überprüfung der Bedingung vor den Inhalt setzen, einmal dahinter. Wenn man sie dahinter setzt, muss man vor dem ersten Durchlauf der Schleife zunächst dahin springen. Im Mips-Code wird dies mit `beqz` gelöst.
4. Im Inhalt der Schleife muss man dann die Modulorechnung umsetzen. In Mips wird das mit den Befehlen `div` und `mfhi` gelöst. Man muss beachten, die richtigen Register zu verwenden.
5. Anschließend werden noch die Neuzuweisungen von `a` und `b` umgesetzt - dies geschieht mit einfachen `move`-Answeisungen.
6. Final ist wichtig, aus der Funktion wieder zurückzukehren. Dabei muss auch der Rückgabewert richtig in das Register `$v0` verschoben werden.

[3] How-To-Pseudocode: `https://www.youtube.com/watch?v=kZvtncmwIJo`

Schleifenbedingung am Anfang:

```
1 EUCLID:
2 loop:
3   beqz $a1 end
4   div $a0 $a1
5   mfhi $t0
6   move $a0 $a1
7   move $a1 $t0
8   b loop
9 end:
10    move $v0 $a0
11  jr $ra
```

Schleifenbedingung am Ende:

```
1 EUCLID:
2   b condition
3 loop:
4   div $a0 $a1
5   mfhi $t0
6   move $a0 $a1
7   move $a1 $t0
8 condition:
9   bnez $a1 loop
10 end:
11  move $v0 $a0
12  jr $ra
```

## Aufgabe 2.9: Ausführungsprotokolle

Betrachten Sie das folgende Mips-Programm :

```
1   .text
2   # 6 in a0
3   # 3 in a1
4
5   start:
6     beq $a0 $zero final
7     beq $a1 $zero final
8     bgt $a0 $a1   mid
9     sub $a1 $a1   $a0
10    b   start
11  mid:
12    sub $a0 $a0   $a1
13    b   start
14  final:
15    add $a0 $a0   $a1
16    and $v0 $v0   $zero
17    or  $v0 $v0   $a0
```

Da Sie Ihren Mips-Interpreter zur Zeit nicht zur Hand haben, Sie aber dringend wissen müssen, was dieses Programm berechnet, sind Sie nun gezwungen ein Ausführungsprotokoll anzugeben.

| **Schritt** | **Register** | | | **Befehlszähler** |
|---|---|---|---|---|
| | $v0 | $a0 | $a1 | pc |
| 1. | | 6 | 3 | 0x00400000 |
| 2. | | … | … | … |

Abbildung 1: Anfang des Ausführungsprotokolls für den oben stehenden Code.

Der Code wurde an der Adresse 0x00400000 in den Speicher geladen.

1. Vervollständigen Sie dazu das Ausführungsprotokoll aus Abbildung 1. Das heißt, geben Sie alle Inhalte der relevanten Register, sowie den aktuellen Wert des Befehlszählers an.

2. Geben Sie eine mathematische Definition der Form

$$f(a,b) = \begin{cases} \dots \\ \\ \end{cases}$$

an, die dem Programm zugrunde liegt.

## Lösung

1.

| **Schritt** | **Register** | | | **Befehlszähler** |
|---|---|---|---|---|
| | $v0 | $a0 | $a1 | pc |
| 1. | | 6 | 3 | 0x00400000 |
| 2. | | 6 | 3 | 0x00400004 |
| 3. | | 6 | 3 | 0x00400008 |
| 4. | | 6 | 3 | 0x00400014 |
| 5. | | 3 | 3 | 0x00400018 |
| 6. | | 3 | 3 | 0x00400004 |
| 7. | | 3 | 3 | 0x00400008 |
| 8. | | 3 | 3 | 0x0040000c |
| 9. | | 3 | 0 | 0x00400010 |
| 10. | | 3 | 0 | 0x00400004 |
| 11. | | 3 | 0 | 0x00400020 |
| 12. | 0 | 3 | 0 | 0x00400024 |
| 13. | 3 | 3 | 0 | 0x00400028 |

Abbildung 2: Ausführungsprotokoll zu Aufgabe 2.

2. Euklidischer Algorithmus zur Bestimmung des größten gemeinsamen Teilers:

$$ggT(a,b) = \begin{cases} a & \text{wenn } b = 0 \\ b & \text{wenn } a = 0 \\ ggT(a-b,b) & \text{wenn } a > b \\ ggT(a,b-a) & \text{wenn } b \geq a \end{cases}$$