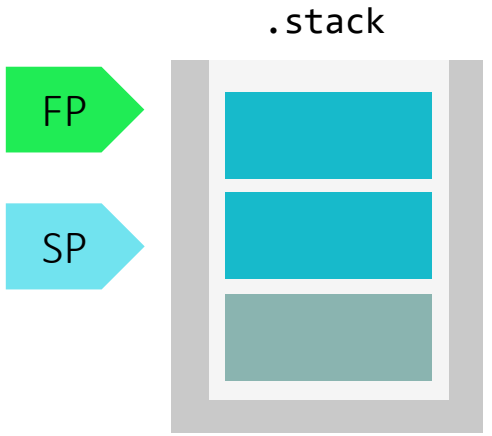
Einführung in die Programmierung



Tutorium 8

Wintersemester 2025/2026

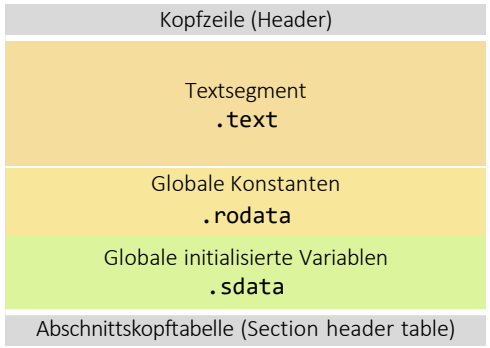
Arbeitsgruppe Systemsoftware
Angewandte Informatik 12

technische universität dortmund

Arbeitsgruppe Systemsoftware

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

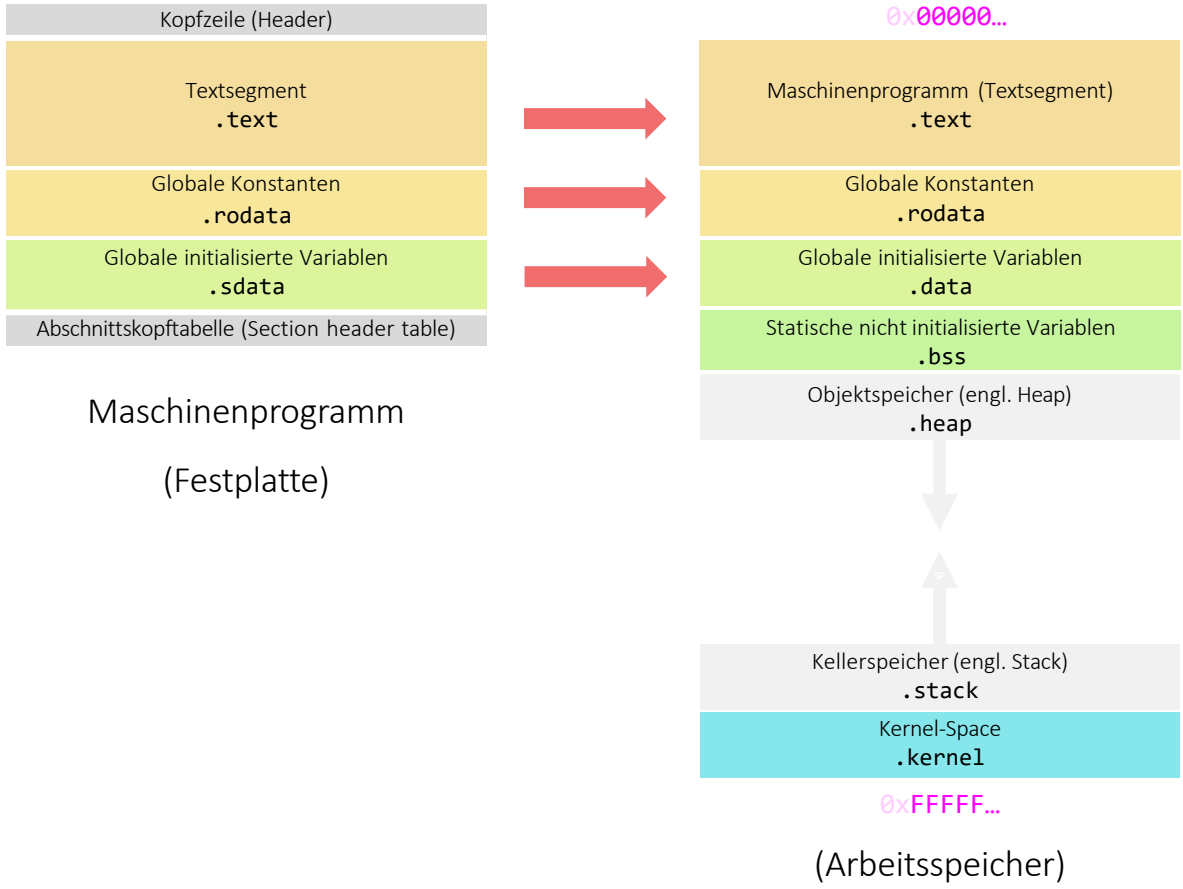
Speicherlayout - Idee

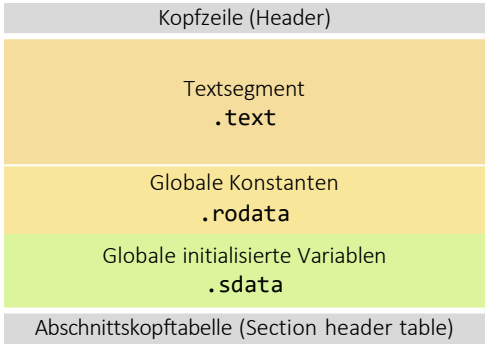


Maschinenprogramm

(Festplatte)

Speicherlayout - Idee





Maschinenprogramm

(Festplatte)

Statischer Aufbau eines Programms, welches auf einer Festplatte liegt.

Kopfzeile / Header

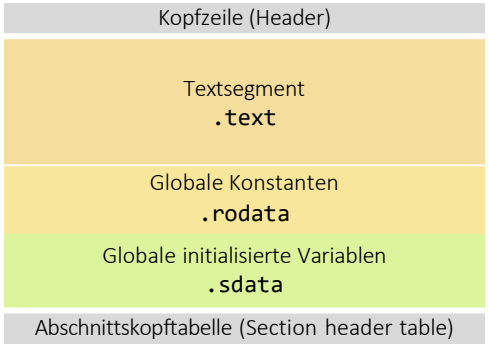
Metadaten für Betriebssystem und Loader

Textsegment (.text)

Maschinencode

Globale Konstanten (.rodata)

Global sichtbare, aber schreibgeschützte Daten



Maschinenprogramm

(Festplatte)

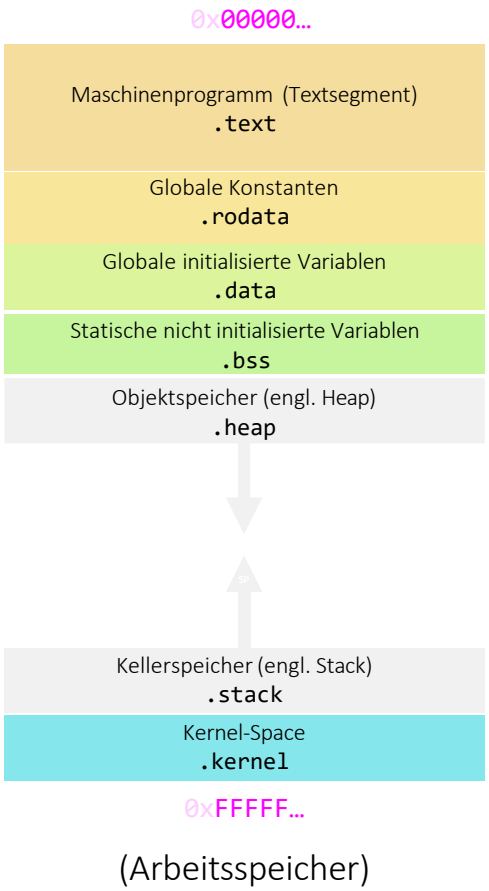
Statischer Aufbau eines Programms, welches auf einer Festplatte liegt.

Globale initialisierte Variablen (.sdata)

Initialisierte globale Variablen

Abschnittskopftabelle (Section header table)

Inhaltsverzeichnis der Segmente



Dynamischer Aufbau eines Prozesses, der im Arbeitsspeicher ausgeführt wird.

Maschinenprogramm / Textsegment (.text)

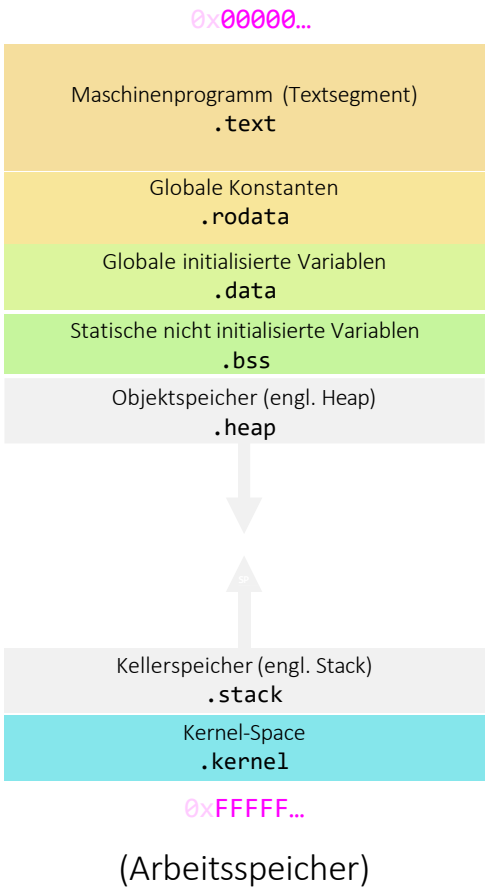
Maschinencode

Globale Konstanten (.rodata)

Global sichtbare, aber schreibgeschützte Daten

Globale initialisierte Variablen (.data)

Initialisierte globale Variablen



Dynamischer Aufbau eines Prozesses, der im Arbeitsspeicher ausgeführt wird.

Statische nicht initialisierte Variablen (.bss)

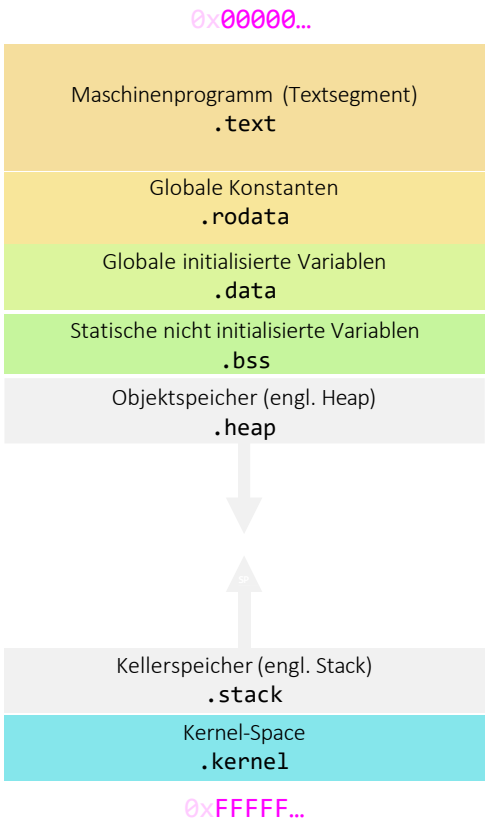
Uninitialisierte globale Variablen

Objektspeicher / Heap (.heap)

Dynamischer Speicher

Kellerspeicher / Stack (.stack)

Lokale Variablen und Funktionsaufrufe



(Arbeitsspeicher)

Dynamischer Aufbau eines Prozesses, der im Arbeitsspeicher ausgeführt wird.

Kernel-Space (.kernel)

Reservierter Speicherbereich für Betriebssystem

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

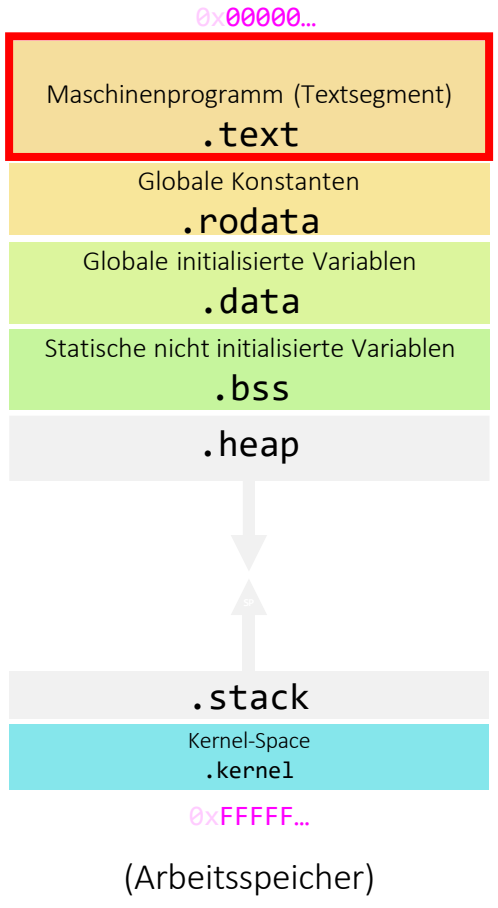
.text

Enthält den ausführbaren, unveränderlichen Maschinencode aller Funktionen des Programms.

Speicherlayout - Maschinenprogramm / Textsegment (.text)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

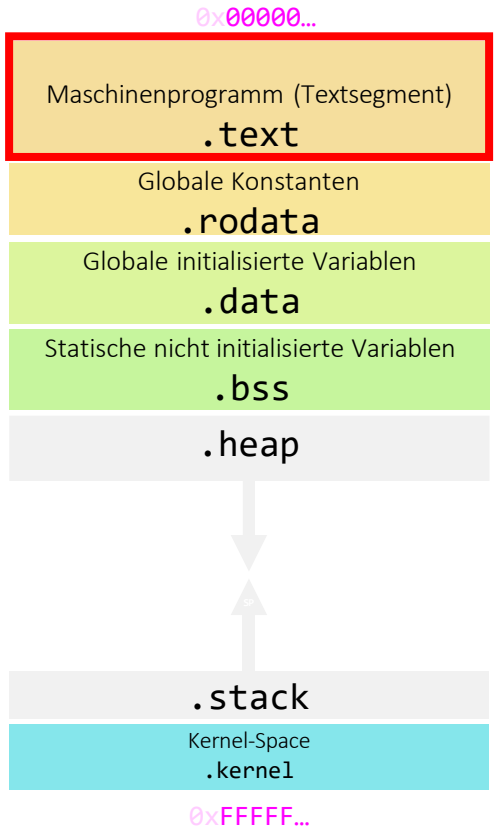


(Arbeitsspeicher)

Speicherlayout - Maschinenprogramm / Textsegment (.text)

Code

```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```

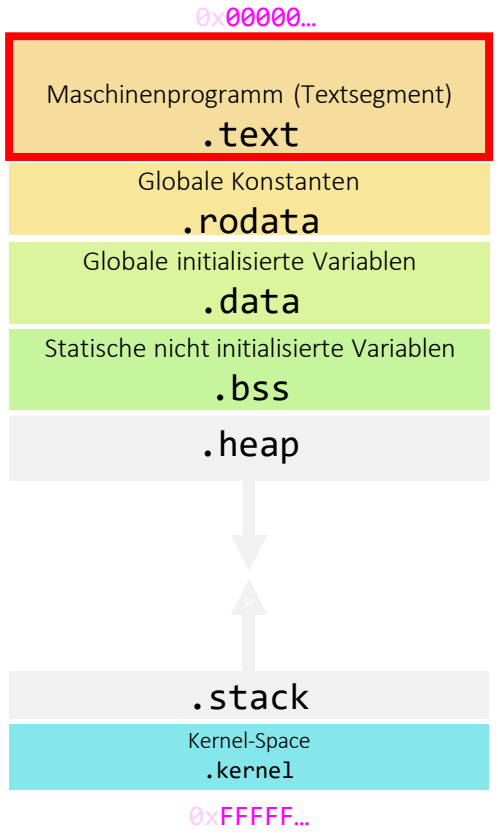


(Arbeitsspeicher)

Speicherlayout - Maschinenprogramm / Textsegment (.text)

Code

```
32          double* volume = malloc(sizeof(double));
33          double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```



(Arbeitsspeicher)

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

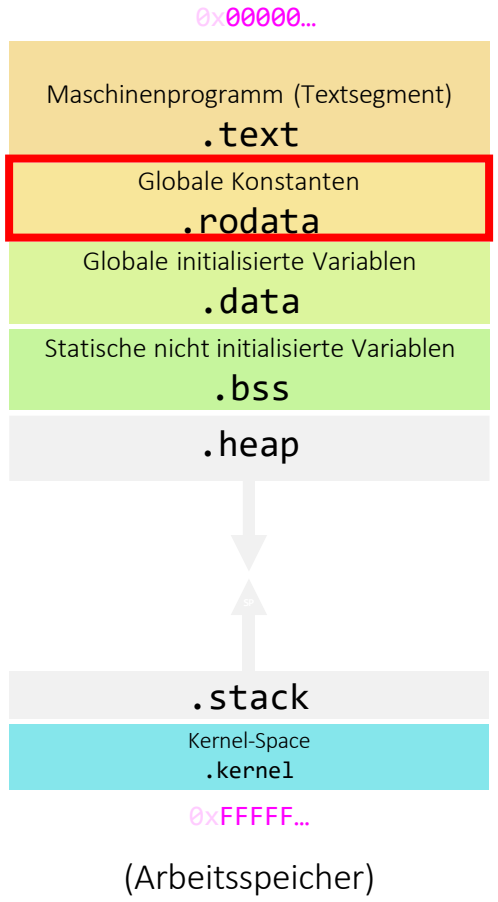
.rodata

Speichert schreibgeschützte globale Daten wie Konstanten und feste Zeichenketten (String-Literale).

Speicherlayout - Globale Konstanten (.rodata)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

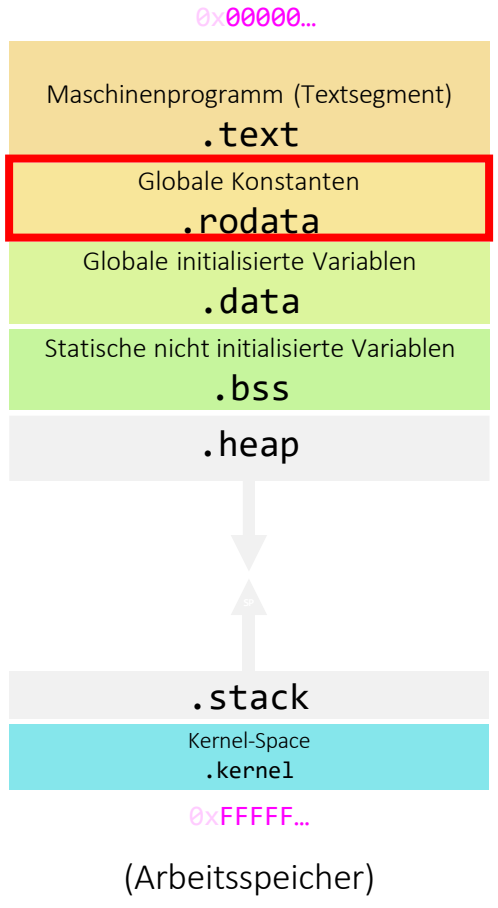


(Arbeitsspeicher)

Speicherlayout - Globale Konstanten (.rodata)

Code

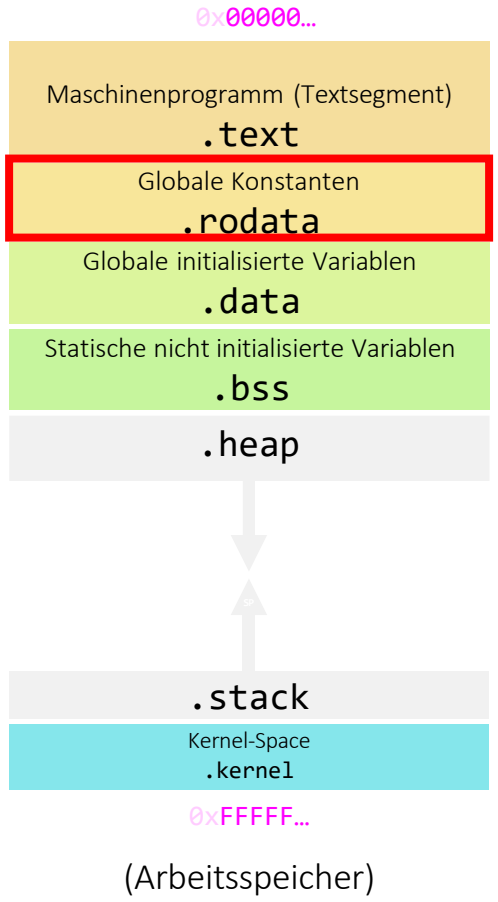
```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```



Speicherlayout - Globale Konstanten (.rodata)

Code

```
32          double* volume = malloc(sizeof(double));
33          double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```



(Arbeitsspeicher)

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

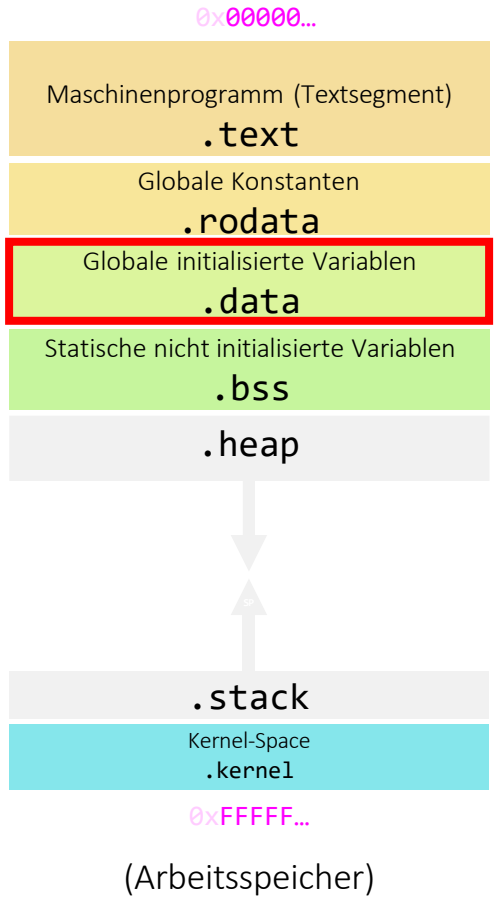
.data

Beinhaltet globale und statische Variablen, die bereits im Code mit einem festen Startwert initialisiert wurden.

Speicherlayout - Global initialisierte Variablen (.data)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

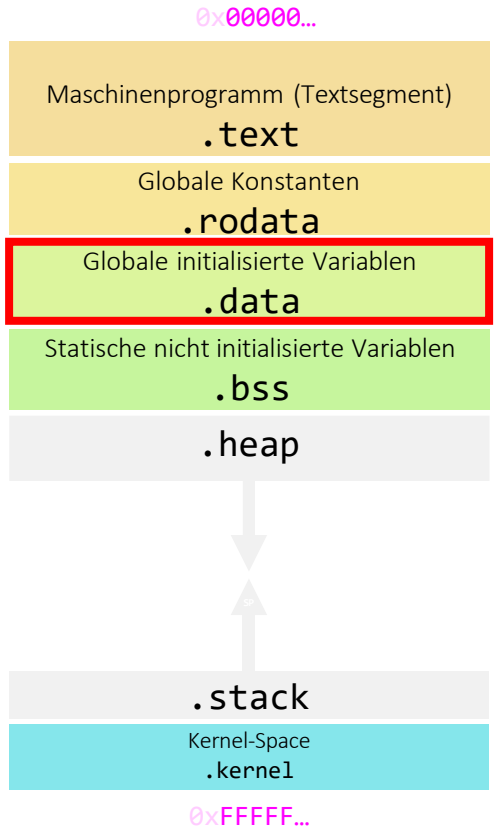


(Arbeitsspeicher)

Speicherlayout - Global initialisierte Variablen (.data)

Code

```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```

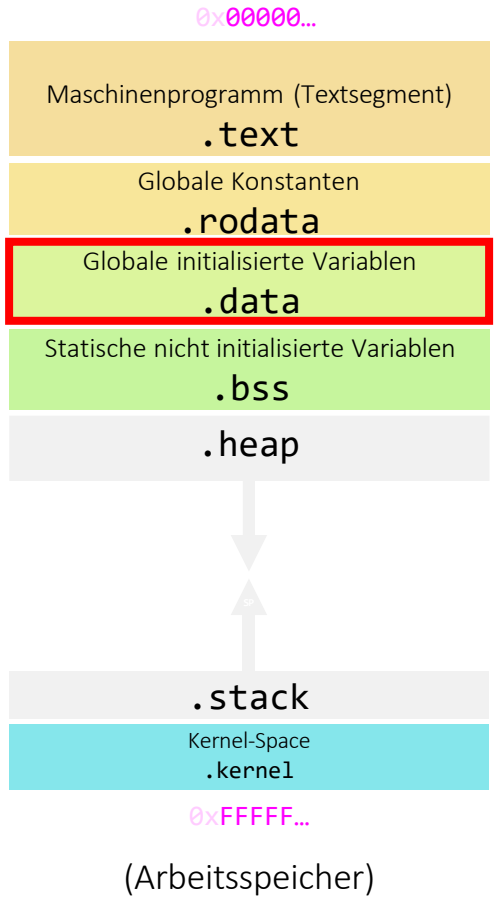


(Arbeitsspeicher)

Speicherlayout - Global initialisierte Variablen (.data)

Code

```
32          double* volume = malloc(sizeof(double));
33          double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```



(Arbeitsspeicher)

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

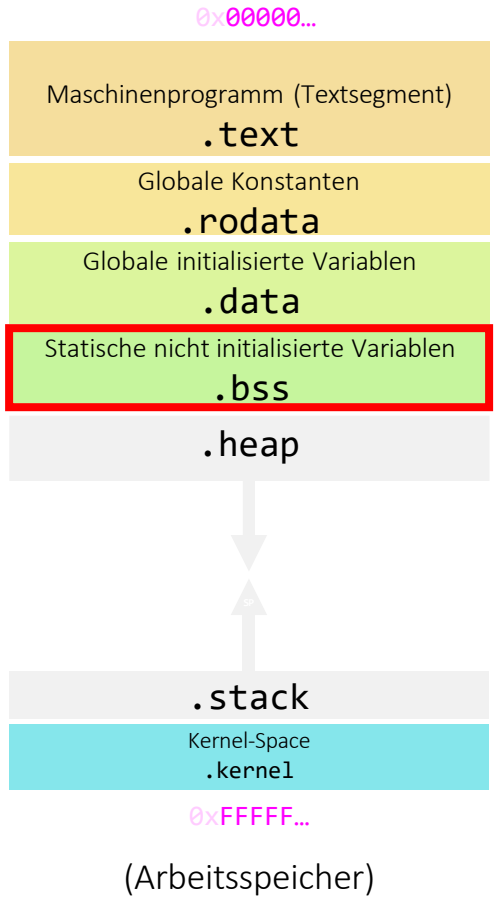
.bss

Reserviert Speicher für globale Variablen ohne expliziten Startwert, die beim Programmstart automatisch genullt werden.

Speicherlayout – Statische nicht initialisierte Variablen (.bss)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

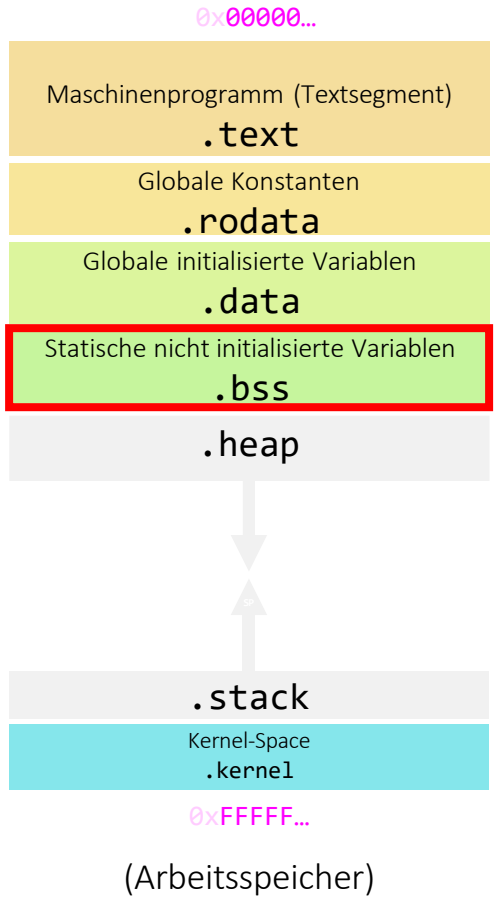


(Arbeitsspeicher)

Speicherlayout – Statische nicht initialisierte Variablen (.bss)

Code

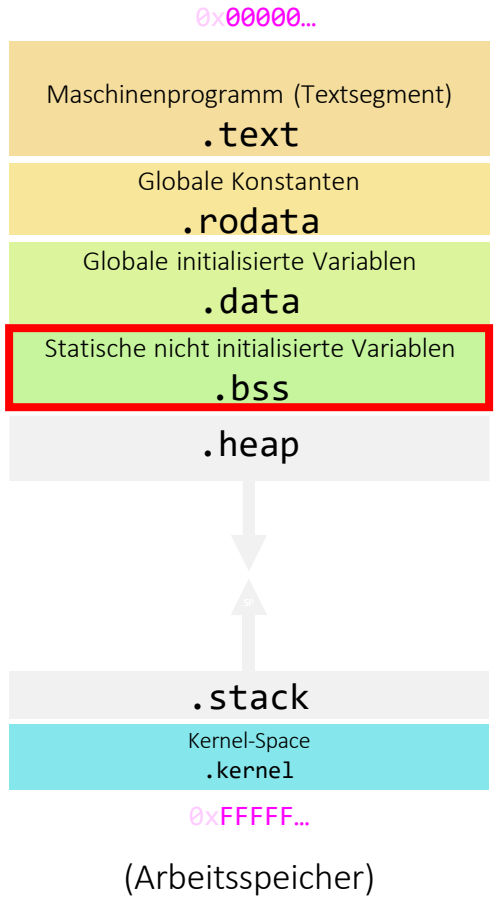
```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```



Speicherlayout – Statische nicht initialisierte Variablen (.bss)

Code

```
32          double* volume = malloc(sizeof(double));
33          double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```



(Arbeitsspeicher)

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

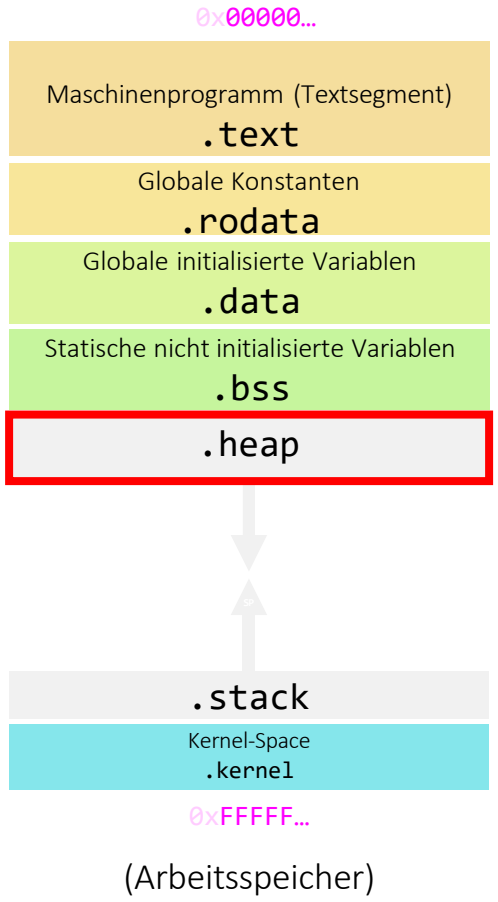
.heap

Dient der manuellen Verwaltung von dynamischem Speicher, der zur Laufzeit explizit angefordert wird (z. B. via malloc).

Speicherlayout - Objektspeicher (.heap)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

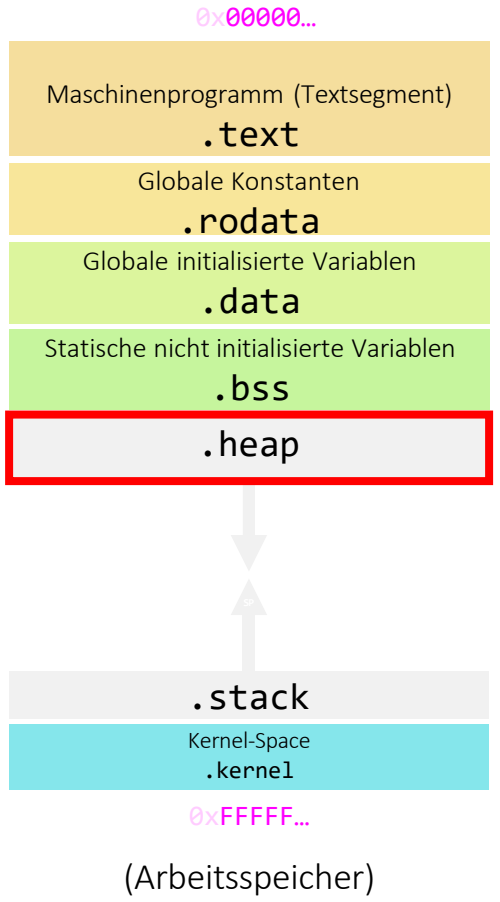


(Arbeitsspeicher)

Speicherlayout - Objektspeicher (.heap)

Code

```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```

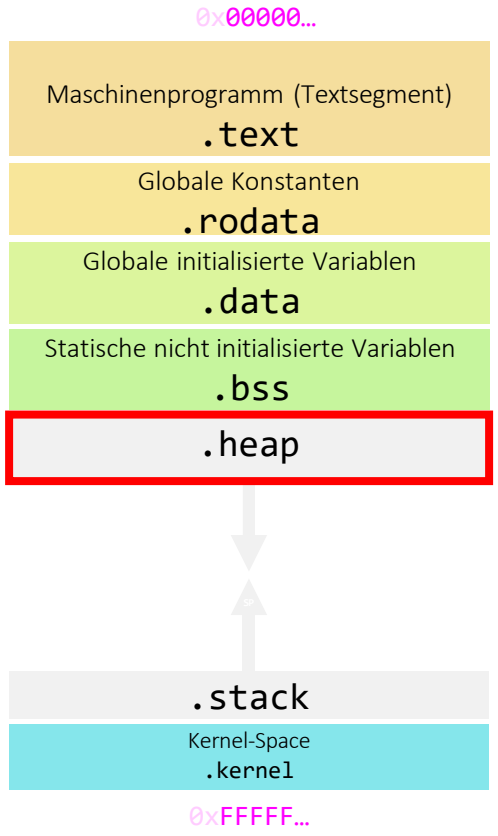


(Arbeitsspeicher)

Speicherlayout - Objektspeicher (.heap)

Code

```
32          double* volume = malloc(sizeof(double));
33          double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```



(Arbeitsspeicher)

Ordnen Sie alle Bezeichner aus sphere.c gemäß ihrer Speicherart und Lebensdauer den Segmenten .text, .rodata, .data, .bss, .heap und .stack zu und listen Sie diese alphabetisch sortiert in der Datei layout.txt auf.

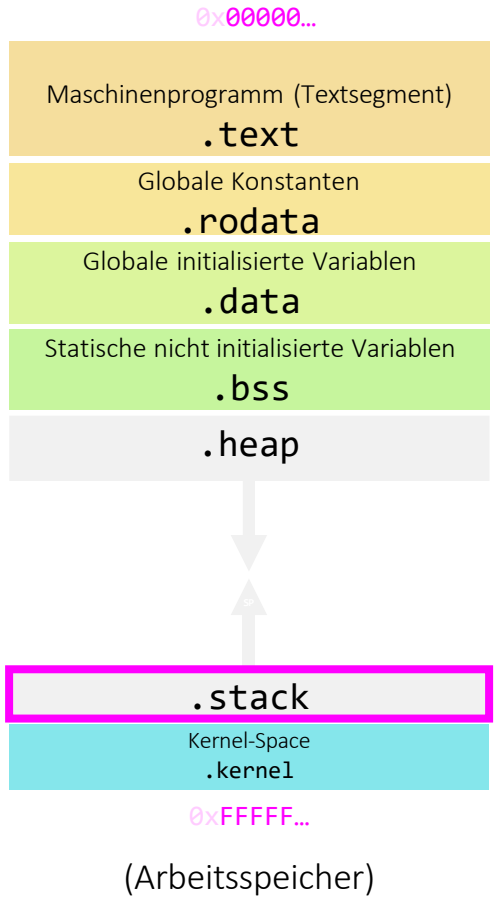
.stack

Verwaltet automatisch lokale Variablen und Funktionsparameter, die nur temporär während eines Funktionsaufrufs existieren.

Speicherlayout – Kellerspeicher (.stack)

Code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <math.h>
4
5   #define PI 3.14159
6
7   double half = 1.0;
8   double global_radius;
9   const char message[] =
    "Calculation of the sphere volume surface area";
10  const double E = 2.17828;
11
12  double calcVolume(double radius) {
13      return (4.0 / 3.0) * PI * pow(radius, 3);
14  }
15
16  ...
```

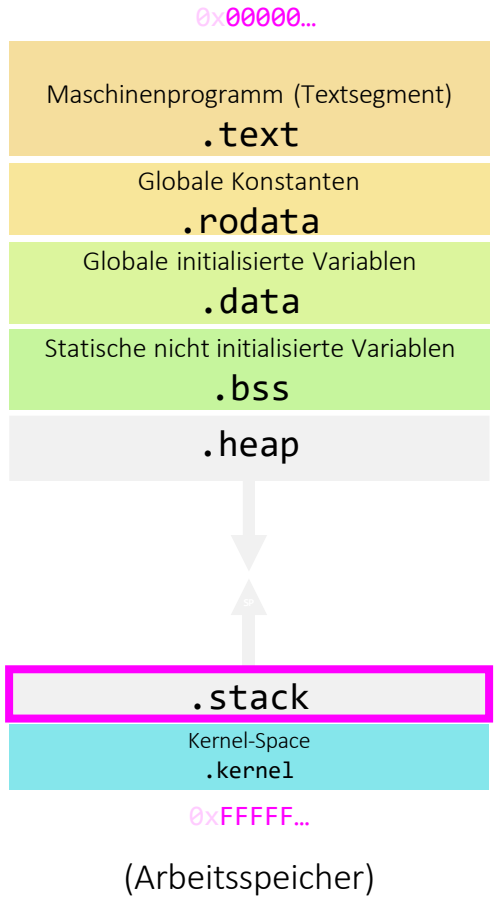


(Arbeitsspeicher)

Speicherlayout – Kellerspeicher (.stack)

Code

```
16      double calcSurface(double myRadius) {
17          return 4 * PI * pow(myRadius, 2);
18      }
19
20      int main(void) {
21          double local_diameter;
22
23          printf("%s\n", message);
24          printf("d = ");
25          scanf("%lf", &local_diameter);
26
27          half = half / 2.0;
28          global_radius = local_diameter * half;
29
30          printf("r = %f\n", global_radius);
31      ...
```



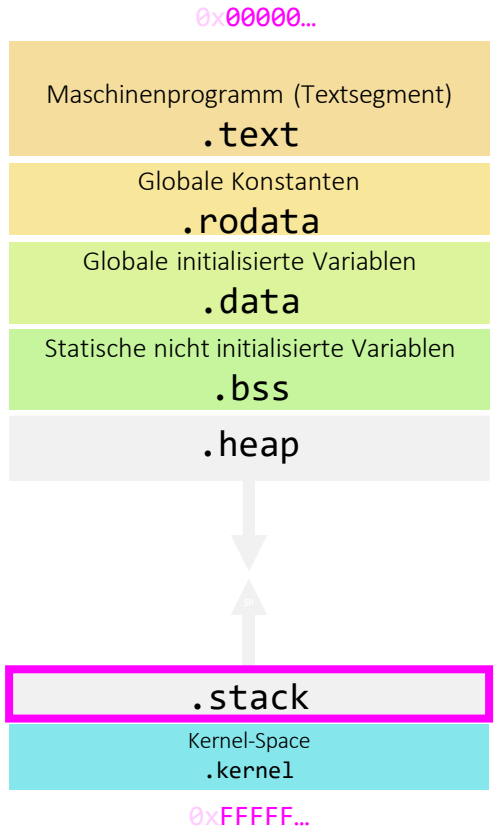
(Arbeitsspeicher)

Speicherlayout – Kellerspeicher (.stack)

Code

```
32      double* volume = malloc(sizeof(double));
33      double* surface = malloc(sizeof(double));
34
35          *volume = calcVolume(global_radius);
36          *surface = calcSurface(global_radius);
37
38          printf("V = %f m^3\n", *volume);
39          printf("A = %f m^2\n", *surface);
40
41          free(volume);
42          free(surface);
43
44          return 0;
45      }
```

Die beiden Zeiger werden streng genommen auch im Stack abgelegt, nach den Zeigern wurde in dieser Aufgabe allerdings nicht gefragt.



(Arbeitsspeicher)

Implementieren Sie die rekursive Karazuba-Multiplikation nur mittels Bit-Operationen und ohne den *-Operator.

- Effiziente Methode zur Multiplikation zweier sehr großer Binärzahlen
  $\vec{x} := \mathrm{int}(\vec{x})$, wobei $\vec{x} = (x_n, \dots, x_1)^\mathsf{T}$, und
  $\vec{y} := \mathrm{int}(\vec{y})$, wobei $\vec{y} = (y_n, \dots, y_1)^\mathsf{T}$,
  jeweils mit der Länge n.

- Verwendung des Teile-und-Herrsche-Prinzips,
  Zerlegung eines komplexen Problems in kleinere einfachere Probleme

- **Zerlegung**
  Jede Zahl $\vec{x}$ und $\vec{y}$ wird in eine obere Hälfte (High)
  $\overrightarrow{x_H} = (x_{n-1}, \ \ldots, x_{\frac{n}{2}})$ mit $x_H = \mathbf{int}(\overrightarrow{x_H})$ und
  $\overrightarrow{y_H} = (y_{n-1}, \ \ldots, y_{\frac{n}{2}})$ mit $y_H = \mathbf{int}(\overrightarrow{y_H})$

  und eine untere Hälfte (Low)
  $\overrightarrow{x_L} = (x_{\frac{n}{2}-1}, \ \ldots, x_0)$ mit $x_L = \mathbf{int}(\overrightarrow{x_L})$ und
  $\overrightarrow{y_L} = (y_{\frac{n}{2}-1}, \ \ldots, y_0)$ mit $y_L = \mathbf{int}(\overrightarrow{y_L})$
  aufgeteilt.

- **Herleitung**

  Jede Zahl x und y kann nun wie folgt dargestellt werden:

$$\mathrm{x} = \underbrace{\mathrm{int}(\overrightarrow{x_H})2^{\frac{n}{2}}}_{x_H} + \underbrace{\mathrm{int}(\overrightarrow{x_L})}_{x_L}$$

$$\mathrm{y} = \underbrace{\mathrm{int}(\overrightarrow{y_H})2^{\frac{n}{2}}}_{y_H} + \underbrace{\mathrm{int}(\overrightarrow{y_L})}_{y_L}$$

  dementsprechend gilt

$$x \cdot y = \left(x_H 2^{\frac{n}{2}} + x_L\right) \cdot \left(y_H 2^{\frac{n}{2}} + y_L\right)$$

- **Herleitung**

$$x \cdot y = \left(x_H 2^{\frac{n}{2}} + x_L\right) \cdot \left(y_H 2^{\frac{n}{2}} + y_L\right)$$

$$= \left(x_H 2^{\frac{n}{2}} + x_L\right) \cdot y_H 2^{\frac{n}{2}} + \left(x_H 2^{\frac{n}{2}} + x_L\right) \cdot y_L$$

$$= x_H 2^{\frac{n}{2}} \cdot y_H 2^{\frac{n}{2}} + x_L \cdot y_H 2^{\frac{n}{2}} + x_H 2^{\frac{n}{2}} \cdot y_L + x_L \cdot y_L$$

$$= 2^n \, x_H \, y_H + 2^{\frac{n}{2}} \left(x_L \, y_H + x_H \, y_L\right) + x_L \, y_L$$

Ⓘ Zeithungrige Variante mit 4 Multiplikationen von Binärzahlen

- **Herleitung**

$$= 2^n x_H\, y_H + 2^{\frac{n}{2}} (x_L\, y_H + x_H\, y_L) + x_L\, y_L$$

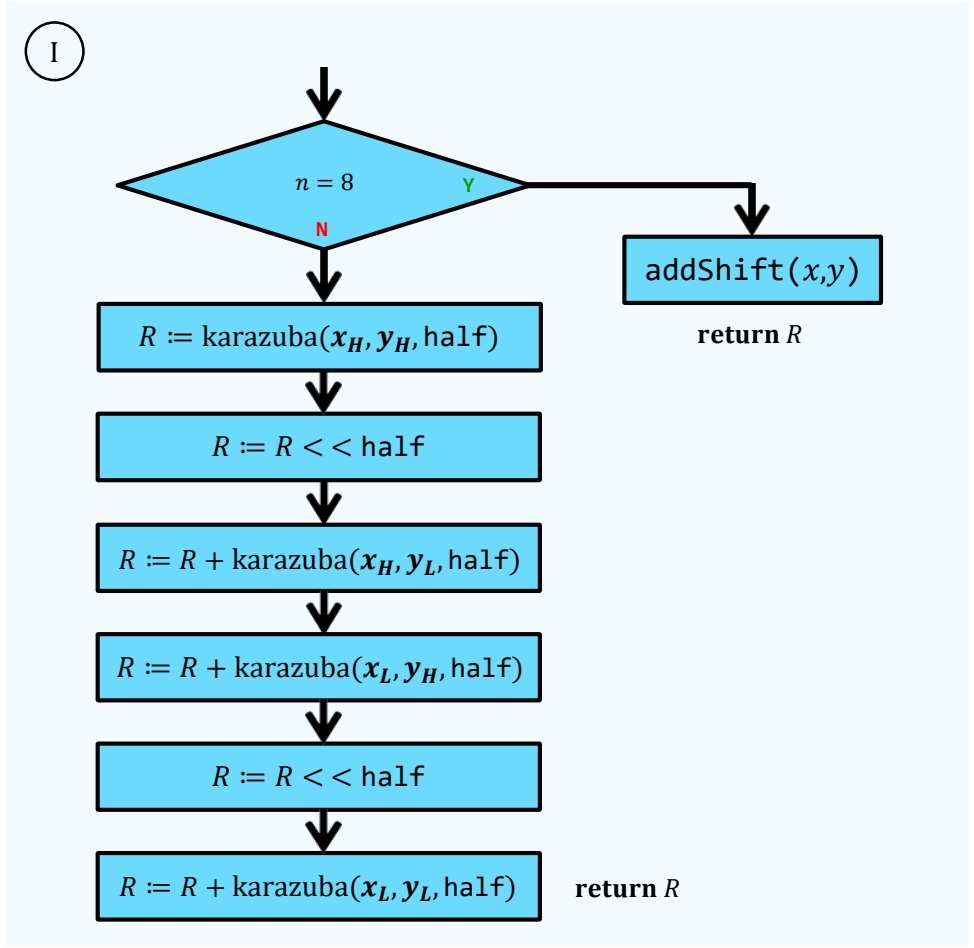
$$= 2^n x_H\, y_H +$$
$$2^{\frac{n}{2}} \Big(x_L\, y_H + x_H\, y_L + \underbrace{(x_L\, y_L + x_H\, y_H) - (x_L\, y_L + x_H\, y_H)}_{\text{Nulladdition}}\Big) +$$
$$x_L\, y_L$$
$$= \ \ldots$$

- **Herleitung**

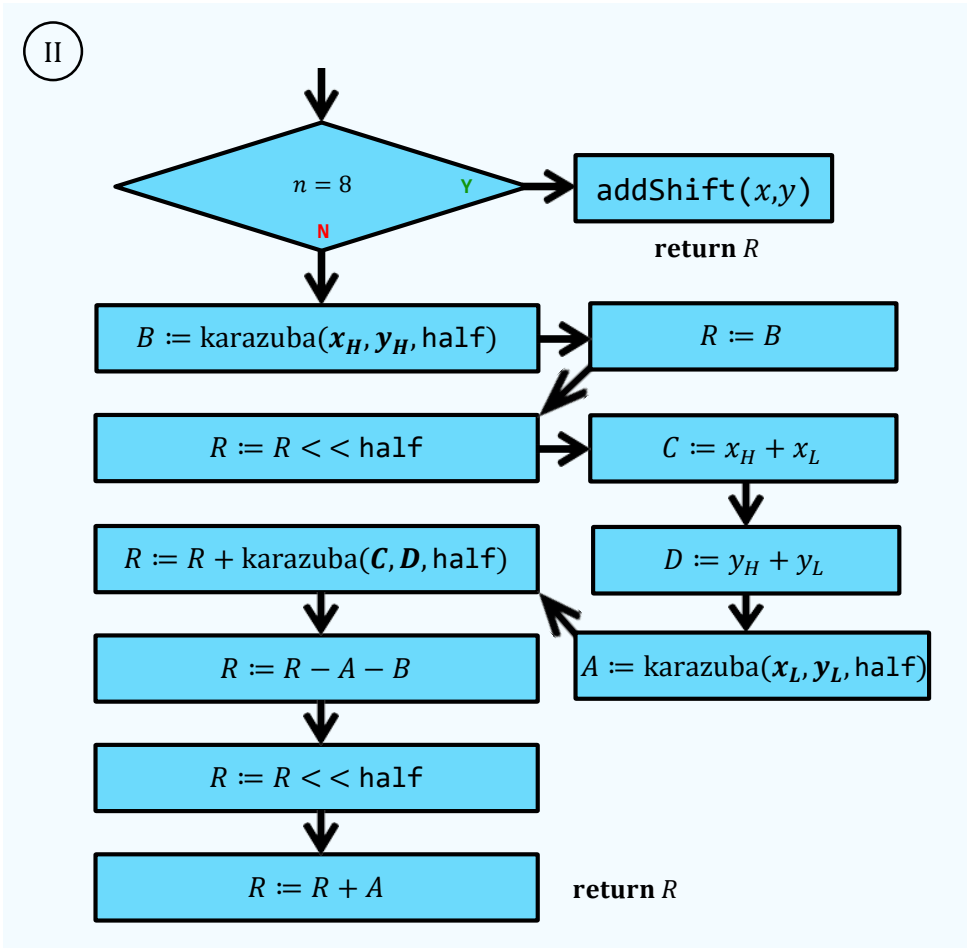
$= \ \ldots$

$$= 2^n \, x_H y_H + \, 2^{\frac{n}{2}} \left((x_H + \, x_L)(y_H + \, y_L) - \, x_L y_L \, - \, x_H y_H \right) + \, x_L y_L$$

(II) Speicherhungrige Variante mit 3 Multiplikationen von Binärzahlen

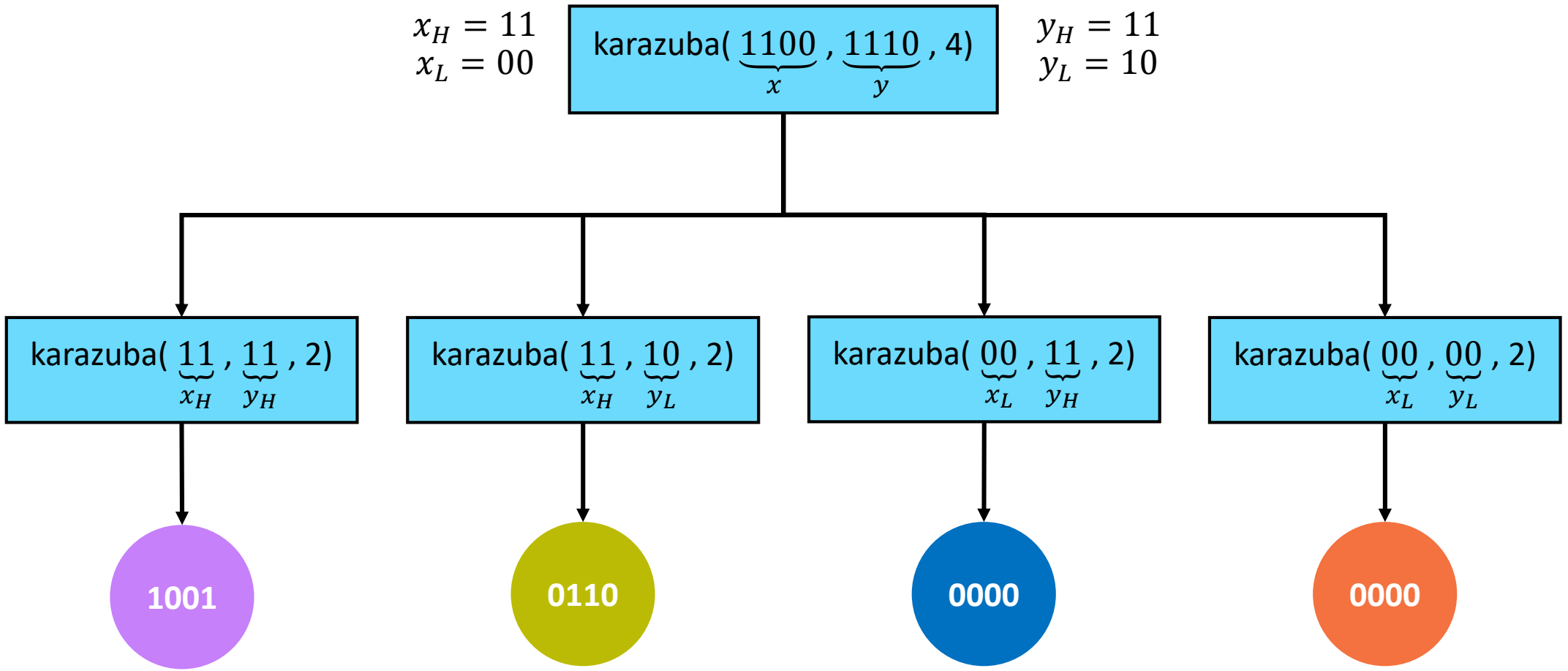


Zeithungrige Variante

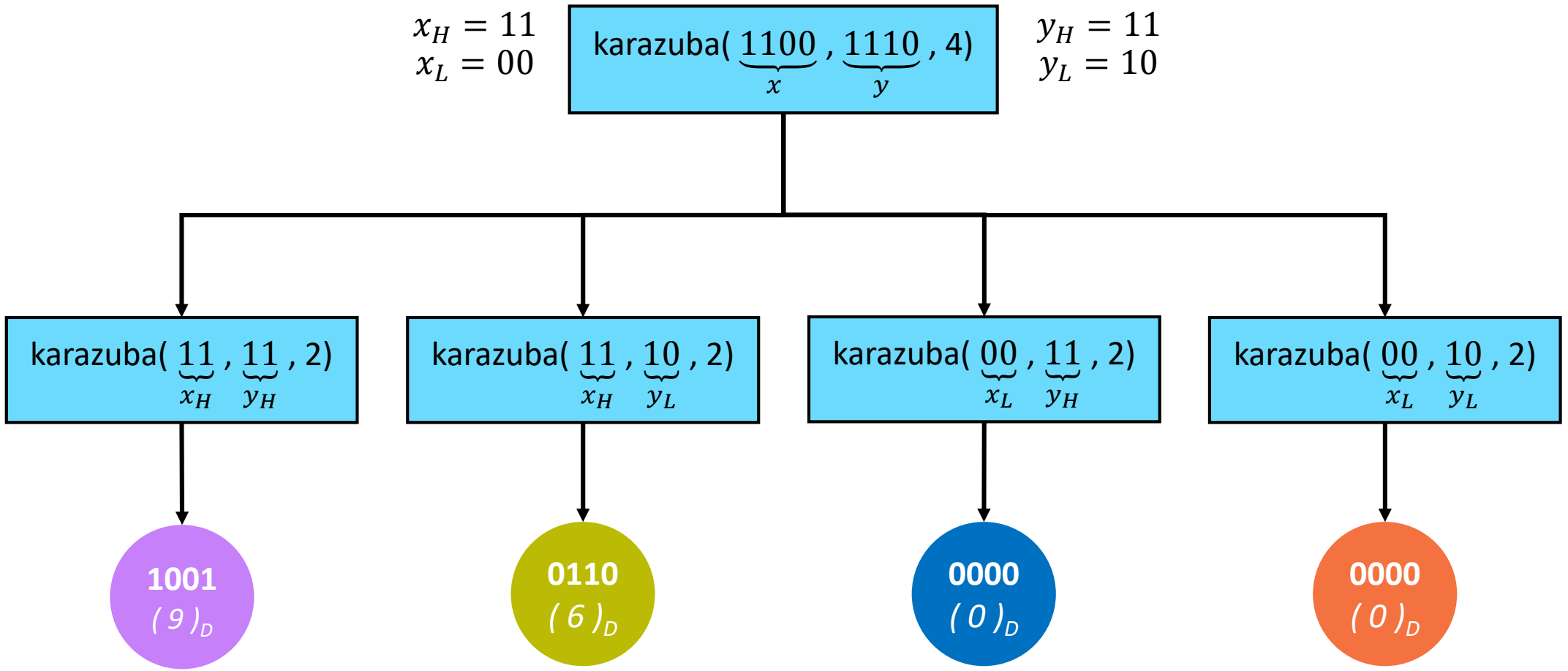


Speicherhungrige Variante

$$2^n \, x_H \, y_H + 2^{\frac{n}{2}} \, (x_L \, y_H + x_H \, y_L) + x_L \, y_L$$



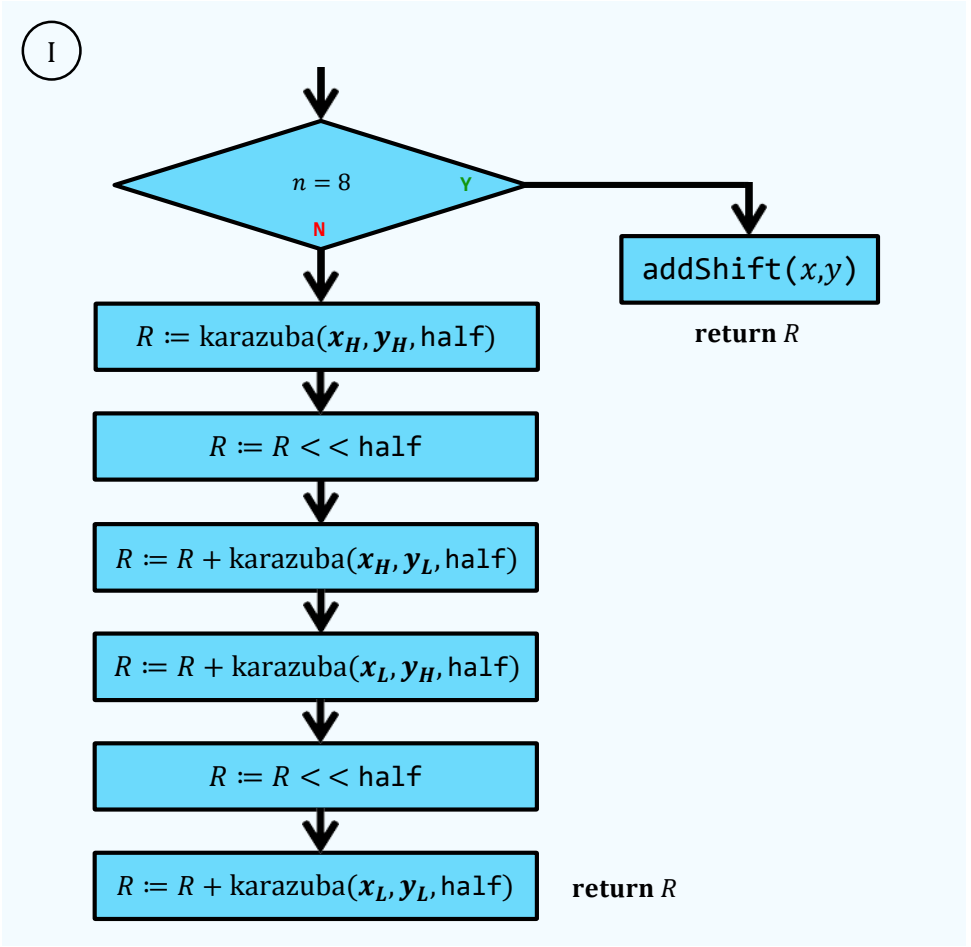
$$2^4 \cdot 9 + 2^{\frac{4}{2}}(0 + 6) + 0 = 144 + 24 + 0 = 168$$



- **Basisfall**
  Bei $n \leq 8$ stoppt Rekursion und **`addShift`** berechnet Ergebnis direkt

Code

```
28  // if the given numbers are short enough (<= 8 bits)
29  if (n <= 8) {
30    // `addShift` is used
31    return addShift((uint8_t)(x), (uint8_t)(y));
32  }
```



Zeithungrige Variante

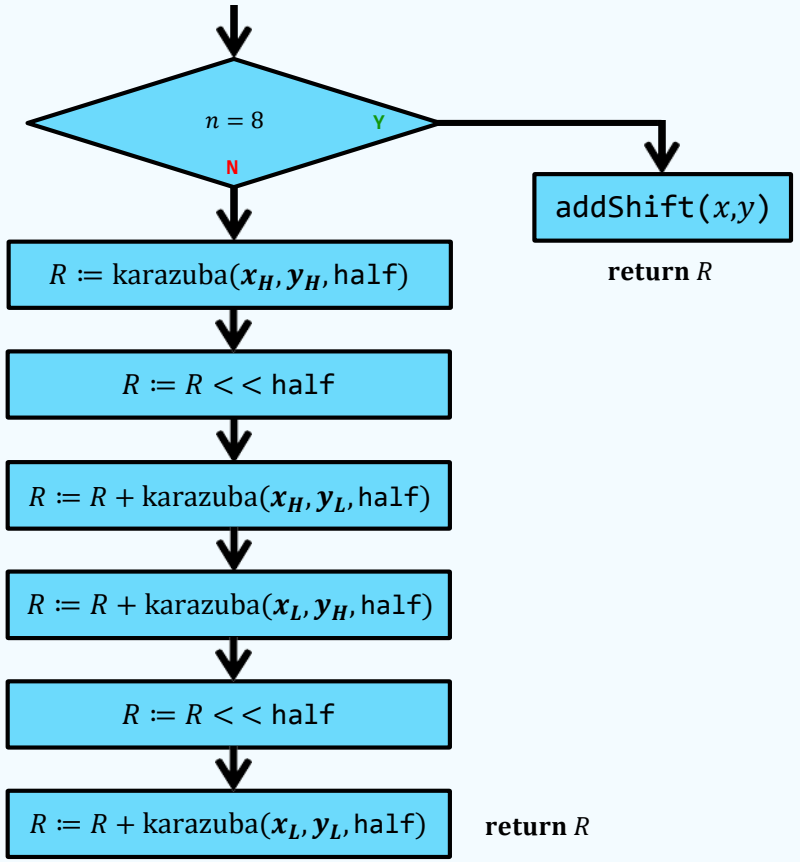
$2^n \, x_H \, y_H + 2^{\frac{n}{2}} \, (x_L \, y_H + x_H \, y_L) + x_L \, y_L$

- **Zerlegung der Zahlen**
  Aufteilung obere (H) und untere (L) Hälfte

Code

```
30  // determine half
31  size_t halfL = n / 2;
32  // if n is even: = halfL, else: = halfL + 1
33  size_t halfH = (n - halfL);
34  // construct masks for higher (H) and lower (L) halves
35  uint64_t maskL = (1ULL << halfL) - 1;
36  uint64_t maskH = (-1ULL) ^ maskL;
37  // extract higher (H) and lower (L) halves
38  uint64_t xL = x & maskL;
39  uint64_t xH = (x & maskH) >> halfL;
40  uint64_t yL = y & maskL;
41  uint64_t yH = (y & maskH) >> halfL;
42  …
```

I



Zeithungrige Variante

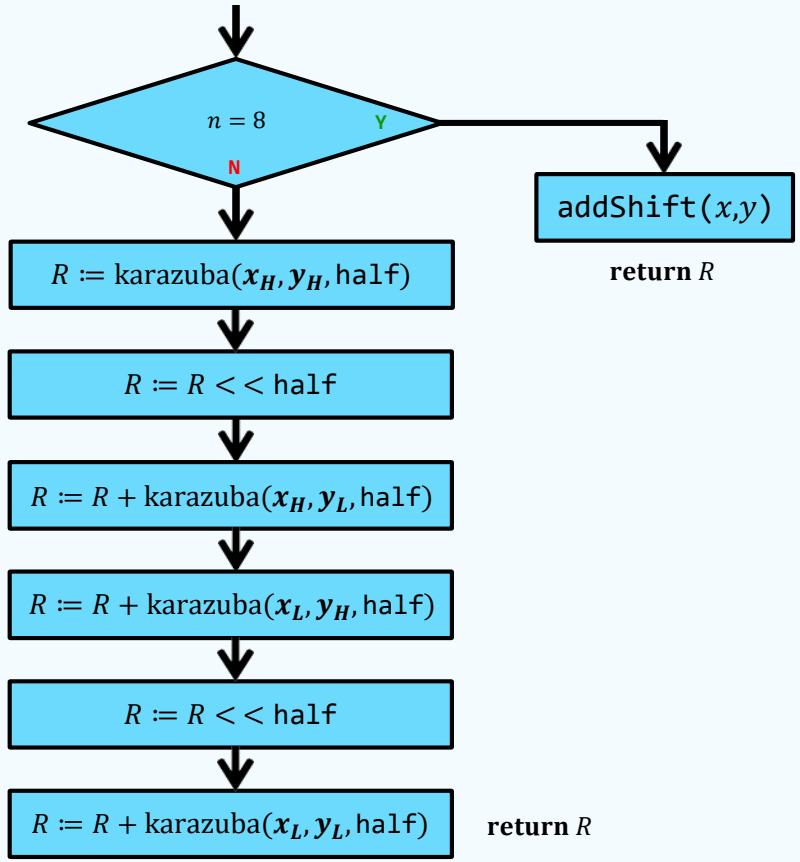
$$2^n\, x_H\, y_H + 2^{\frac{n}{2}} (x_L\, y_H + x_H\, y_L) + x_L\, y_L$$

Rekursive Berechnung Teilprodukte

Code

```
42  // formula: 2^n*xH*yH + 2^(n/2)(xL*yH + xh*yL) + xL*yL
43  // xH*yH
44  result = karazuba(xH, yH, halfH);
45  // (xH * yH) * 2^(halfL)
46  result <<= halfL;
47  // add (xH * yL) + (xL * yH)
48  result += karazuba(xH, yL, halfH);
49  result += karazuba(xL, yH, halfH);
50  // high term moves to 2^n, middle terms move to 2^(halfL)
51  result <<= halfL;
52  // add xL*yL
53  result += karazuba(xL, yL, halfL);
```

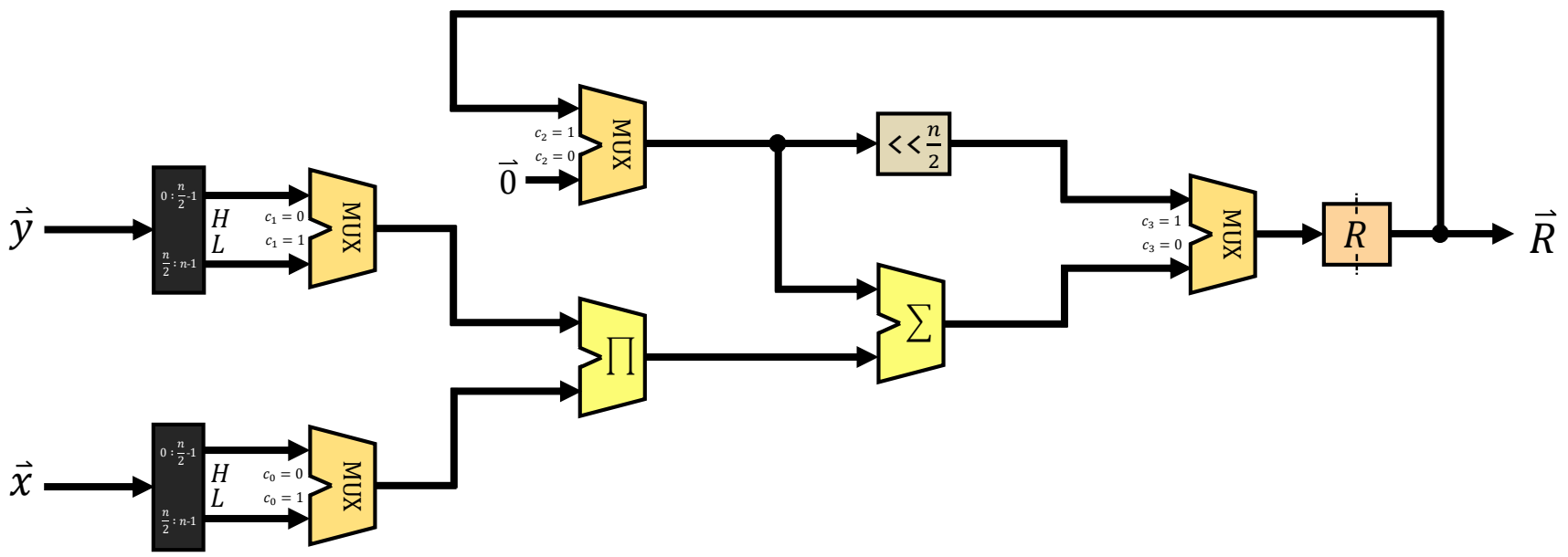
I

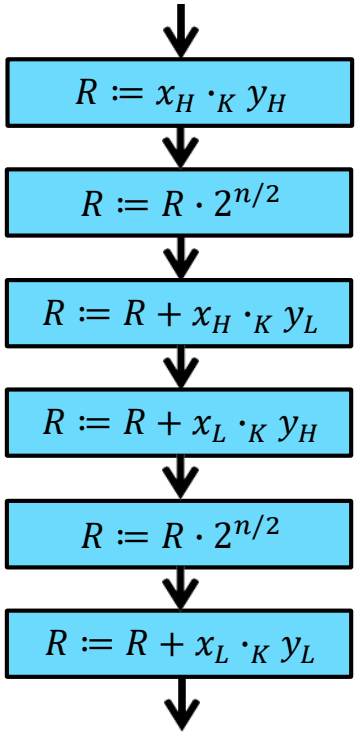


Zeithungrige Variante

$$2^n x_H y_H + 2^{\frac{n}{2}} (x_L y_H + x_H y_L) + x_L y_L$$

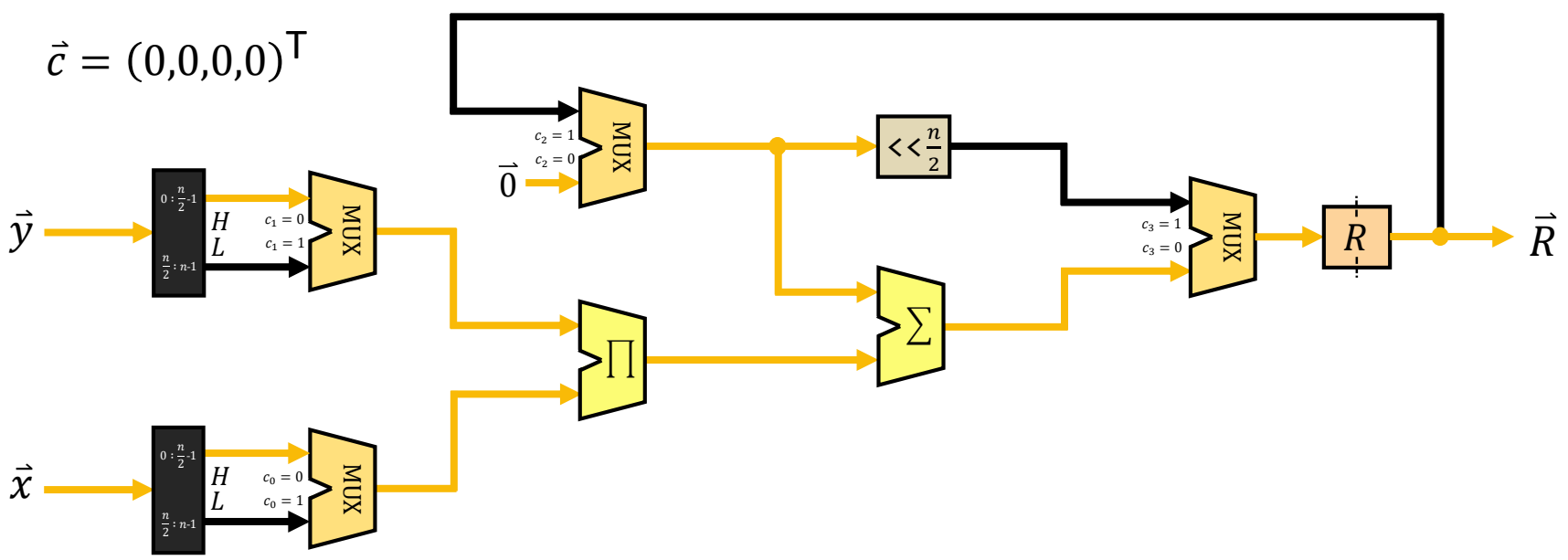
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:

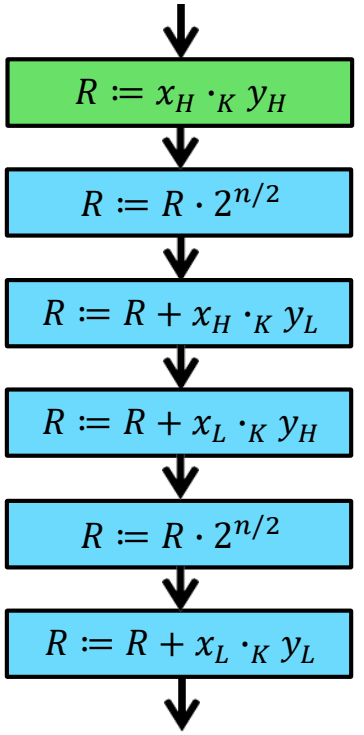




Einzelner Rekursionsschritt des Karazuba-Algorithmus im Datenflussdiagramm I

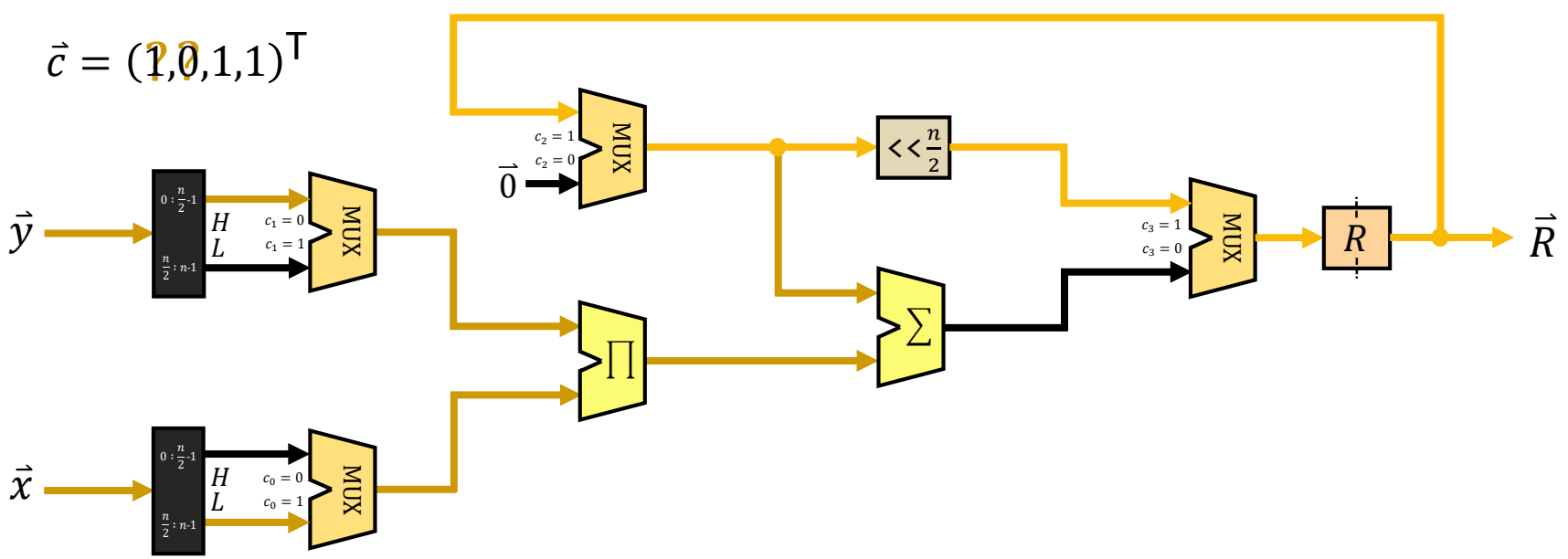
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:

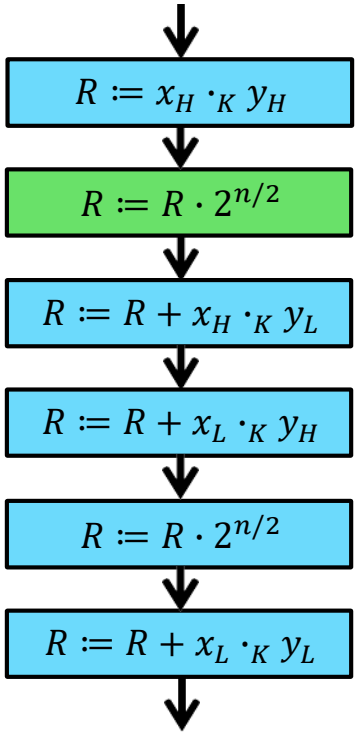




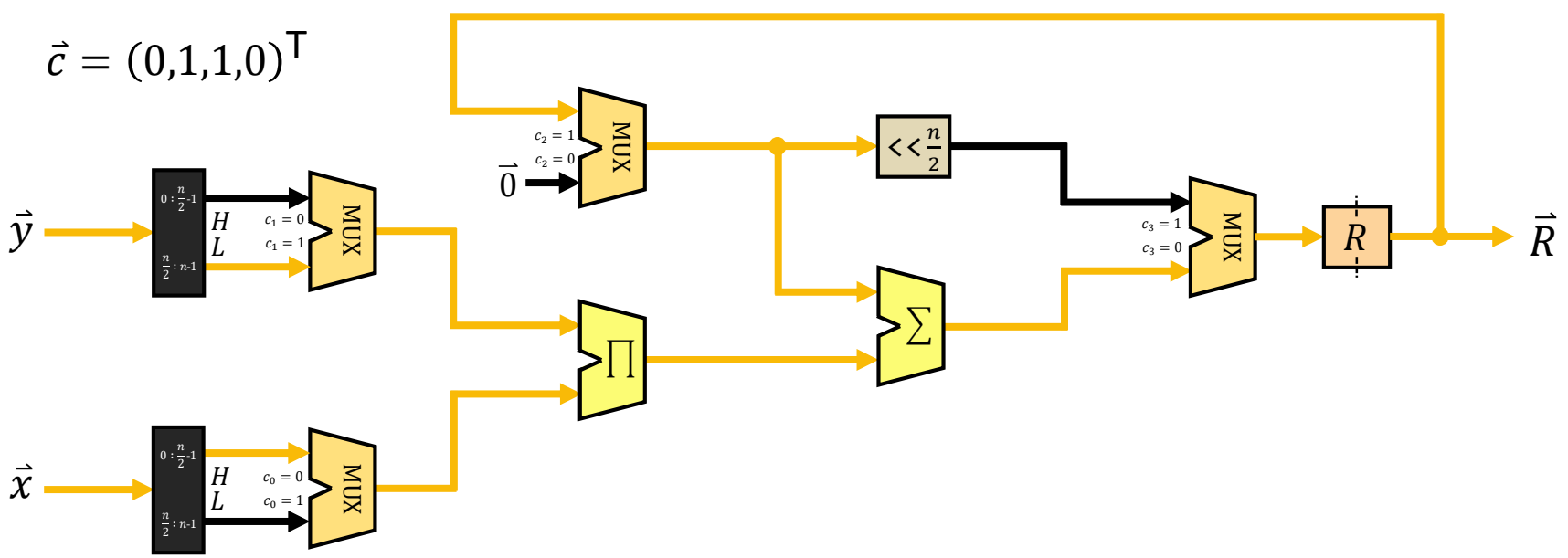
Einzelner Rekursionsschritt des Karazuba-Algorithmus im Datenflussdiagramm I

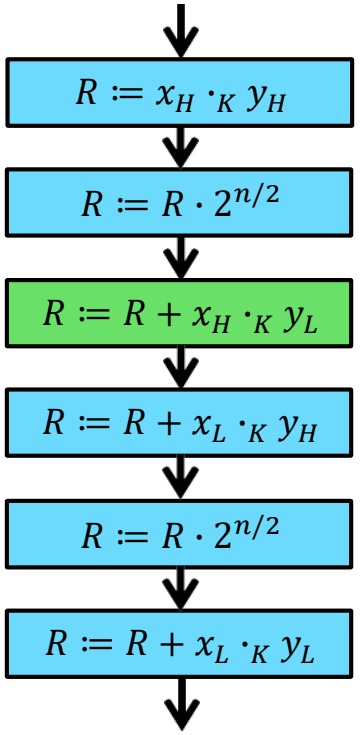
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:



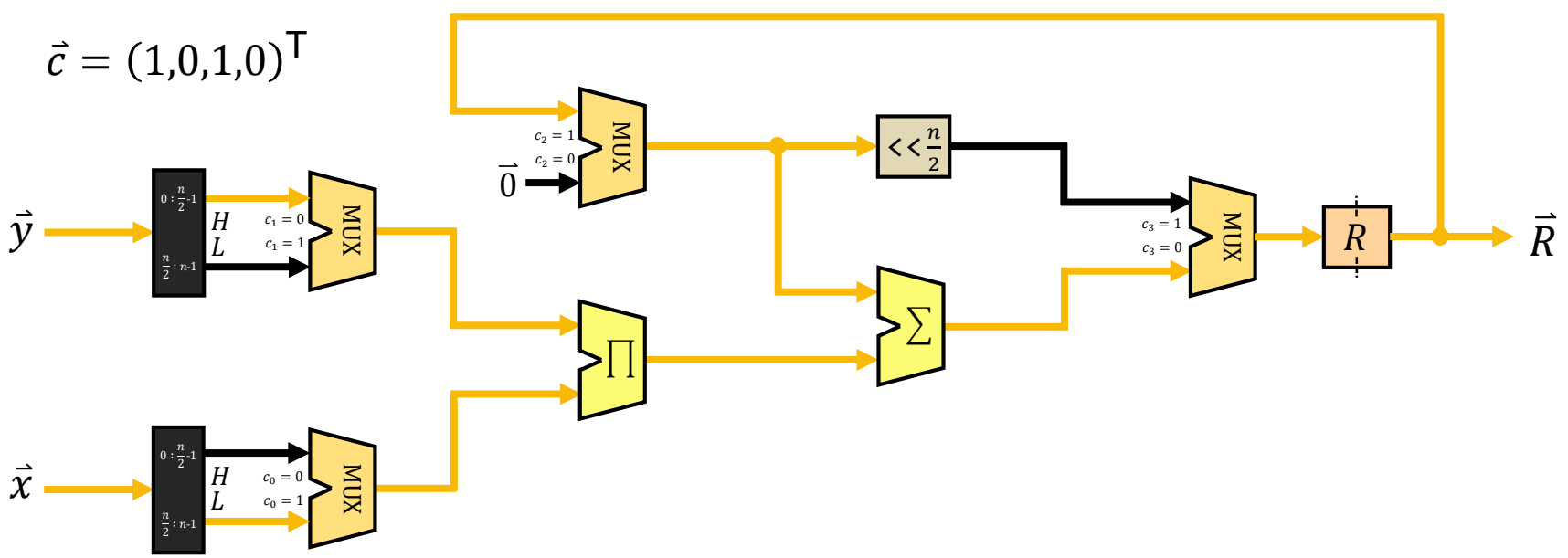


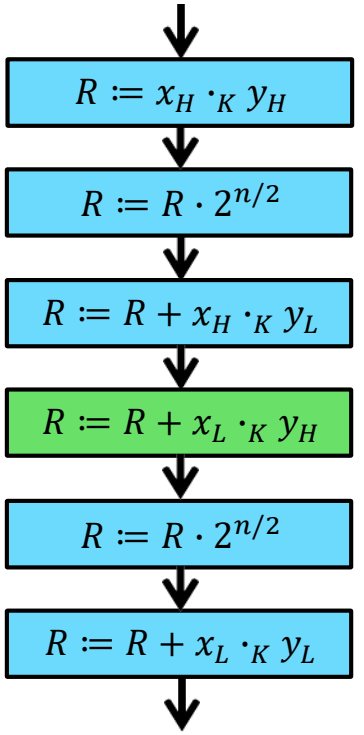
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:



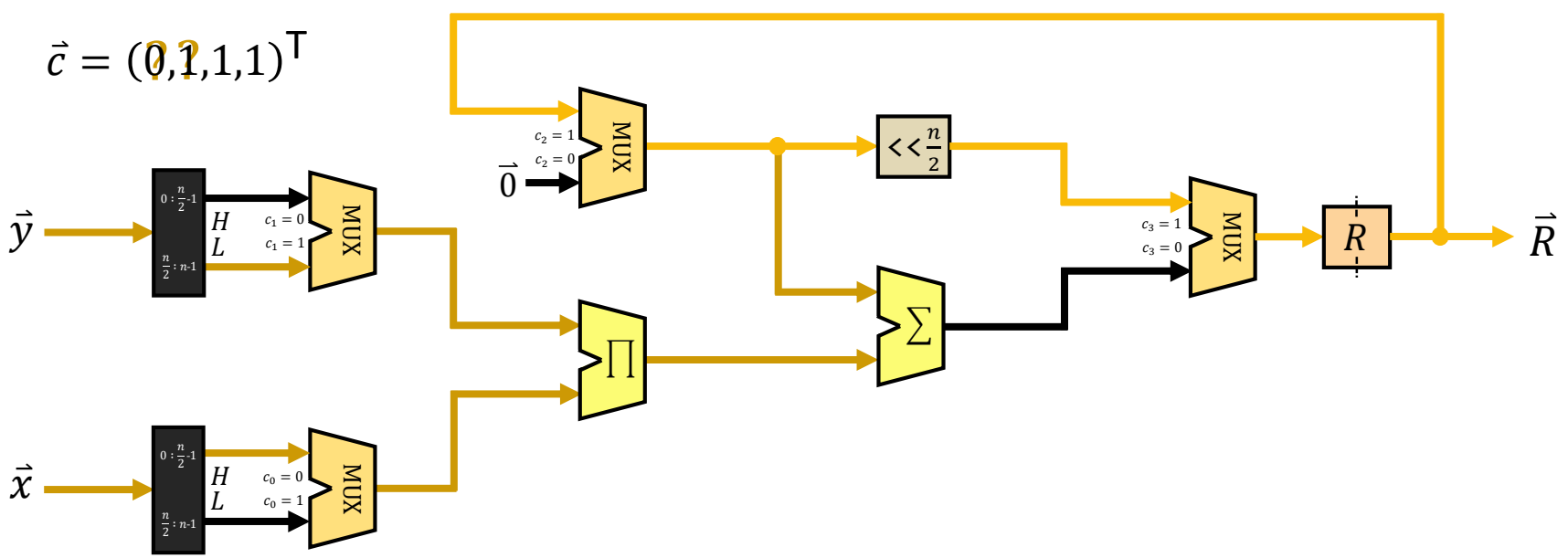


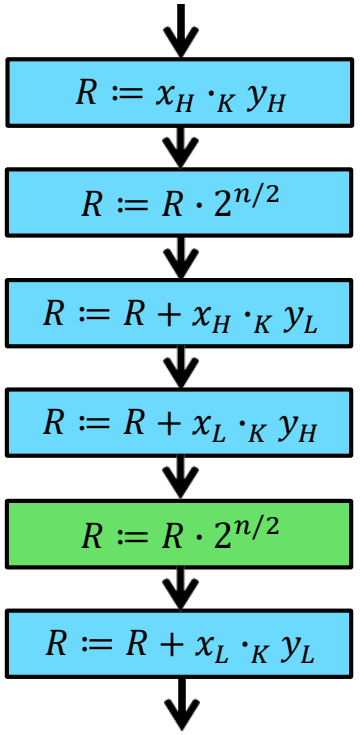
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:



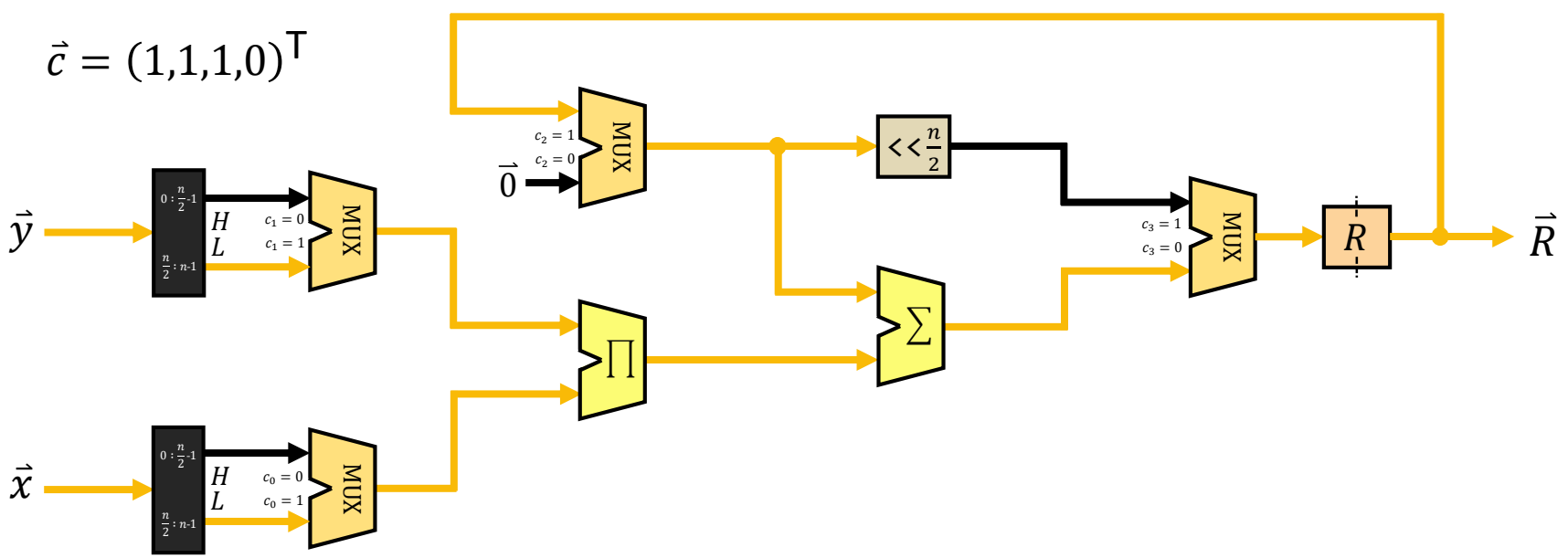


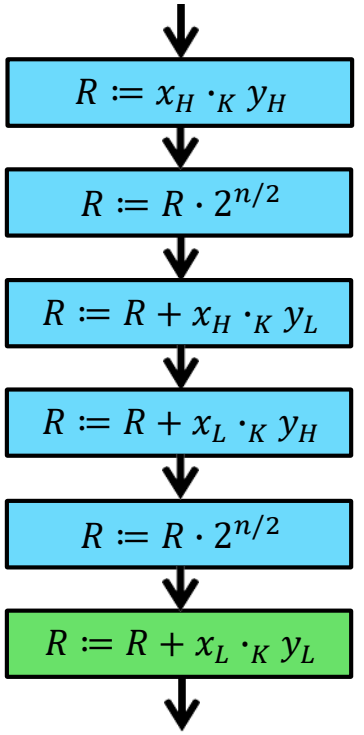
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:

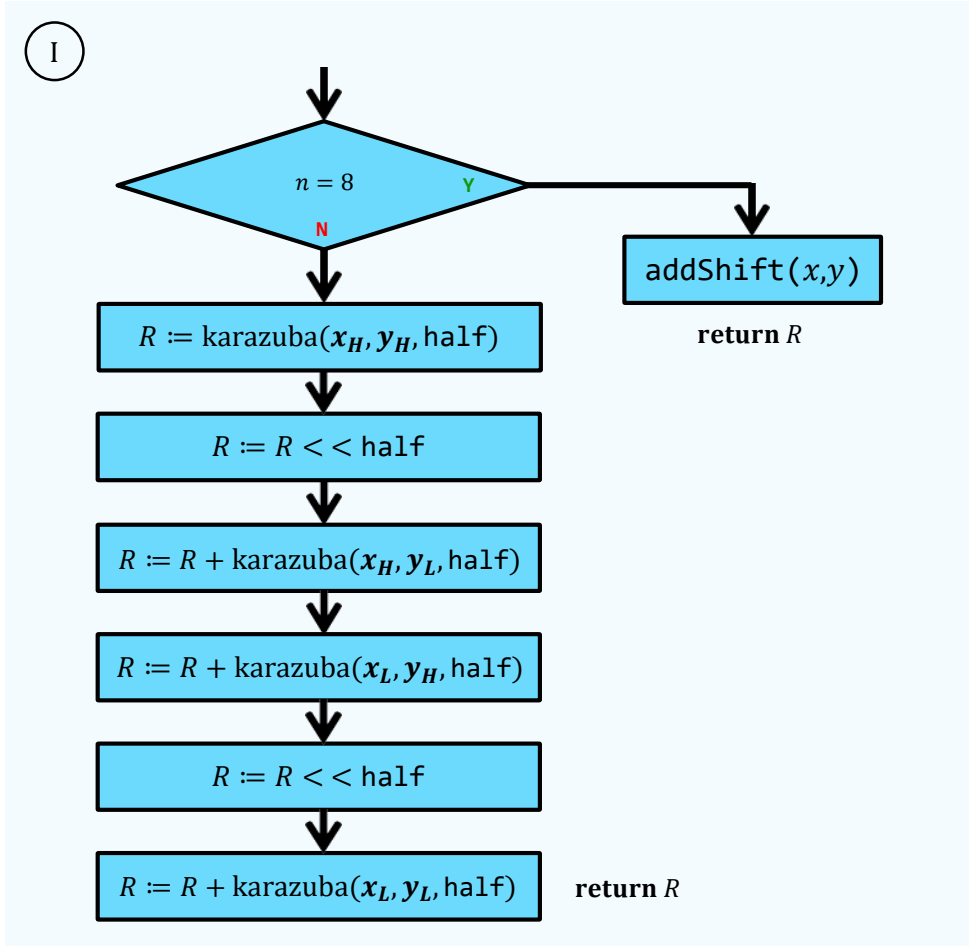




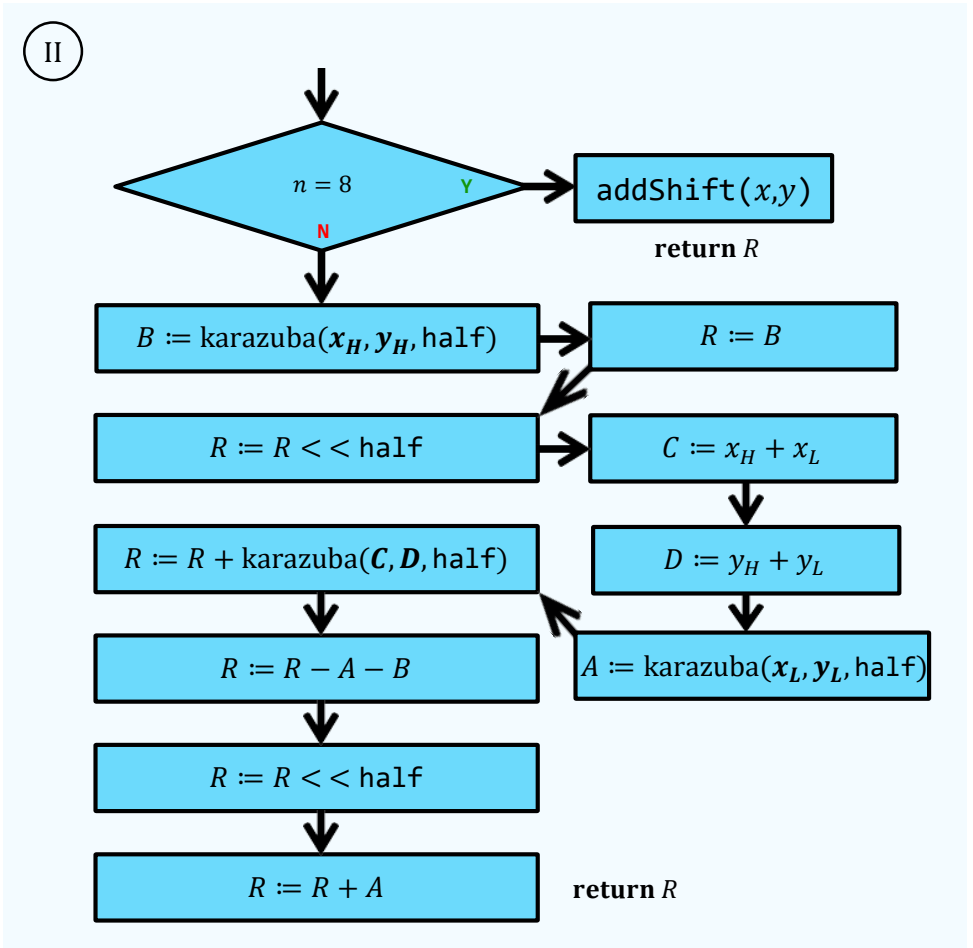
Das **Datenflussdiagramm** stellt für jeden Rechenschritt innerhalb eines einzelnen Rekursionsschritt die Datenflüsse dar:





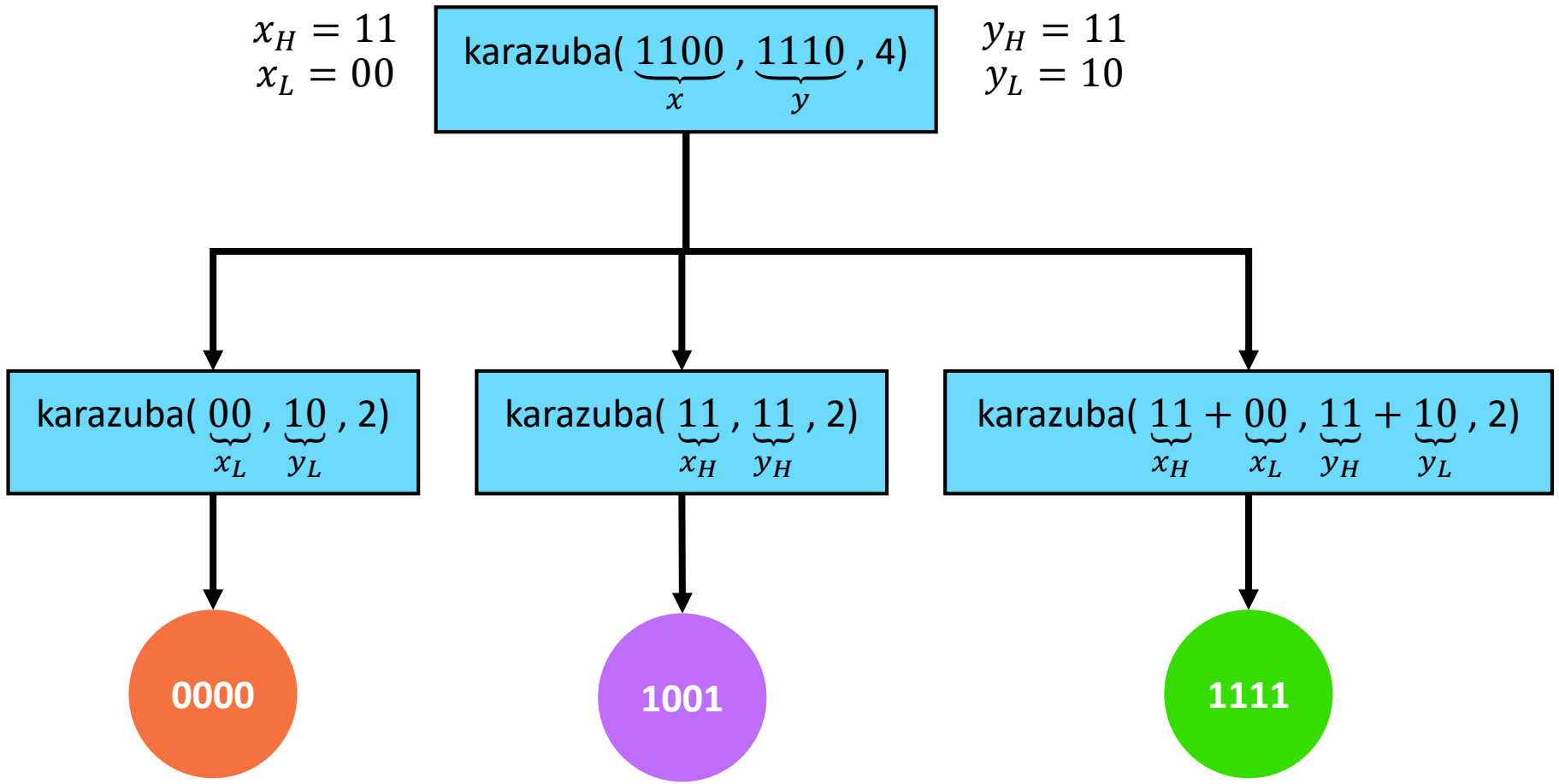


Zeithungrige Variante

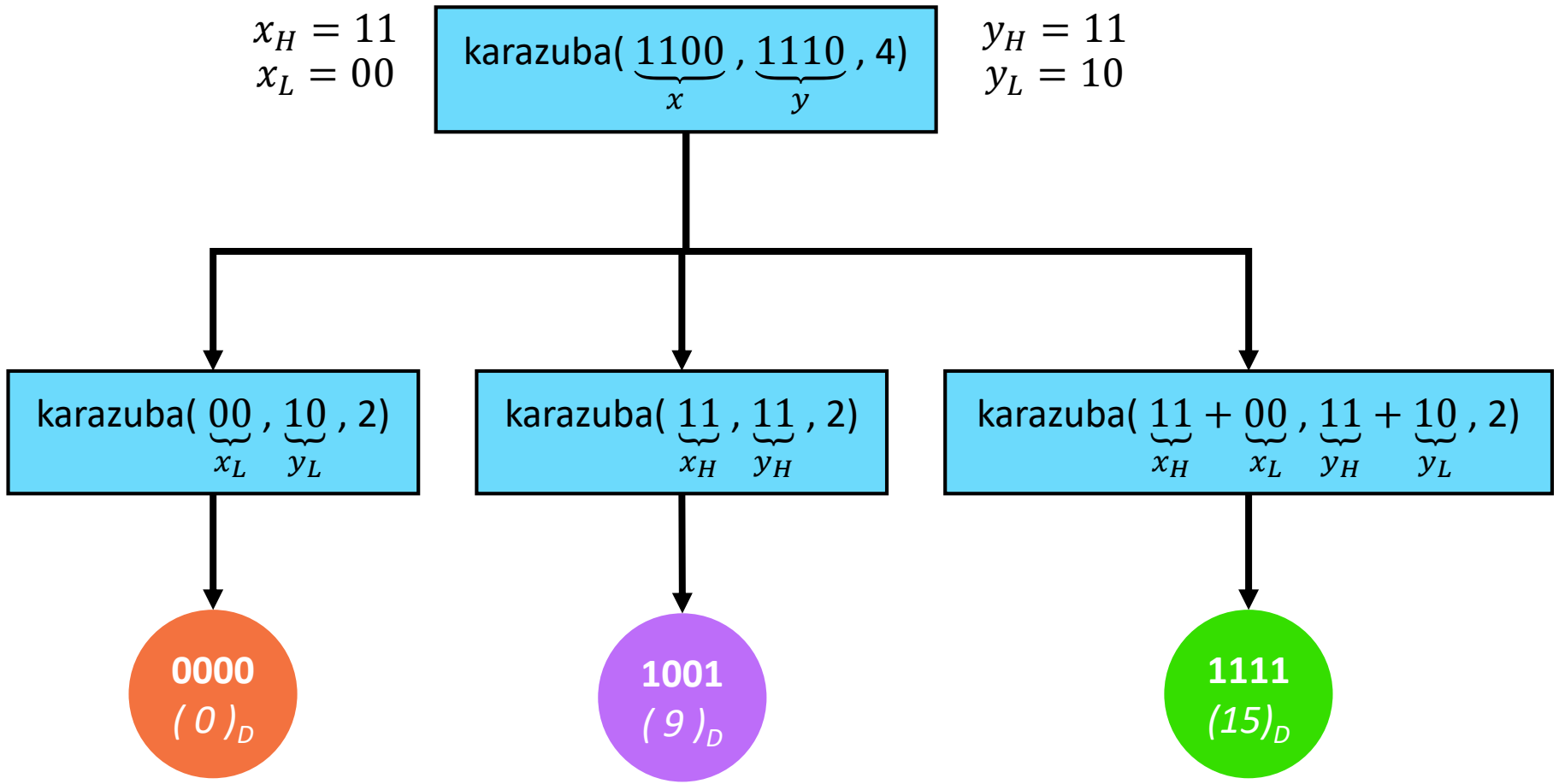


Speicherhungrige Variante

$$2^n\, x_H y_H + 2^{\frac{n}{2}} \left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H \right) + x_L y_L$$



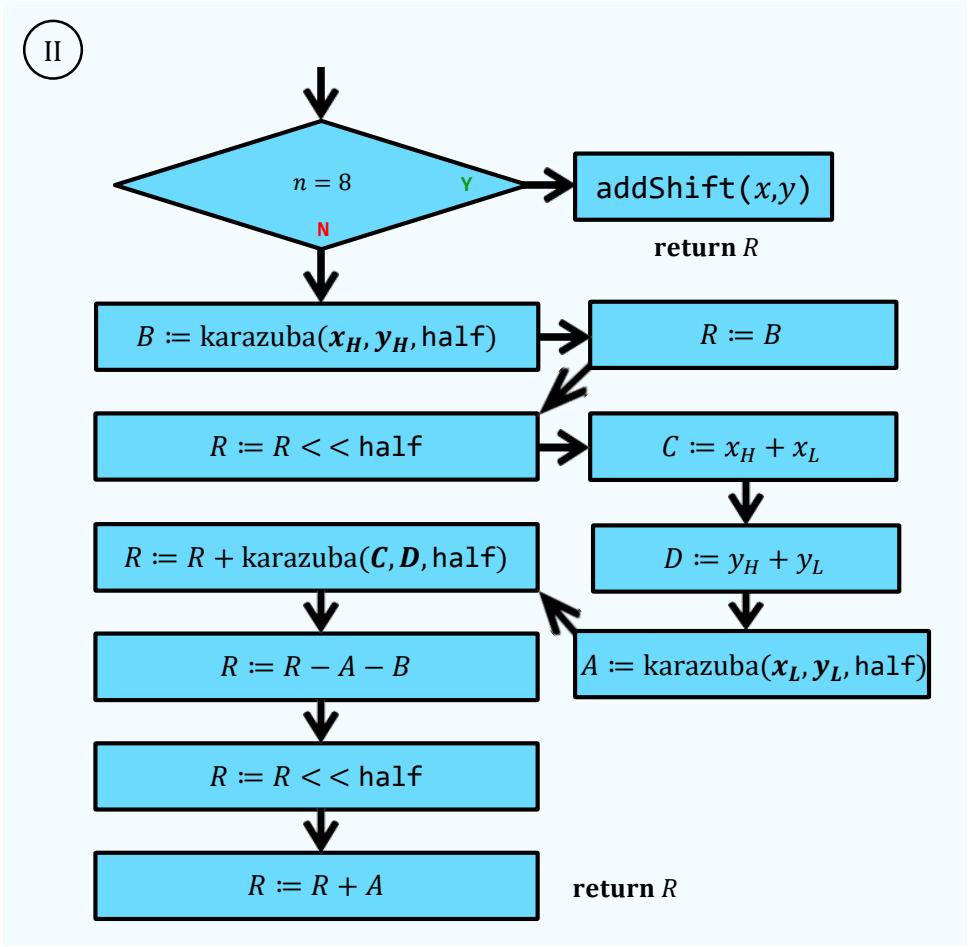
$$2^4 \cdot 9 + 2^{\frac{4}{2}}(15 - 0 - 9) + 0 = 144 + 24 + 0 = 168$$



- **Basisfall**
  Bei n ≤ 8 stoppt Rekursion und addShift berechnet Ergebnis direkt

Code

```
28  // if the given numbers are short enough (<= 8 bits)
29  if (n <= 8) {
30    // `addShift` is used
31    return addShift((uint8_t)(x), (uint8_t)(y));
32  }
```



Speicherhungrige Variante

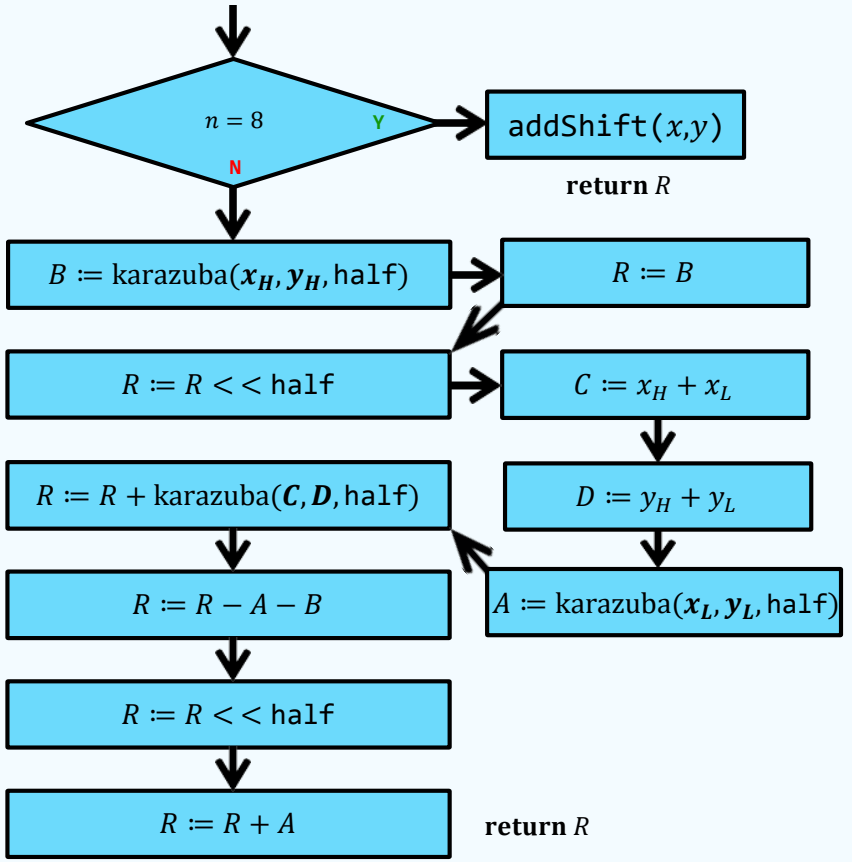
$$2^n x_H y_H + 2^{\frac{n}{2}}\left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H\right) + x_L y_L$$

- **Zerlegung der Zahlen**
  Aufteilung obere (H) und untere (L) Hälfte

Code

```
36  // determine half
37  size_t half = n / 2;
38  // create mask for lower bits
39  // e.g. n=8, half = 4, 1 << 4 is 10000, 10000 - 1 = 01111
40  uint64_t mask = (1ULL << half) - 1ULL;
41  // filter out lower half x & y
42  uint64_t xL = x & mask;
43  uint64_t yL = y & mask;
44  // filter out upper half x & y
45  // e.g. … 1100 0011 >> 4 is … 0000 1100
46  uint64_t xH = x >> half;
47  uint64_t yH = y >> half;
```

II



Speicherhungrige Variante

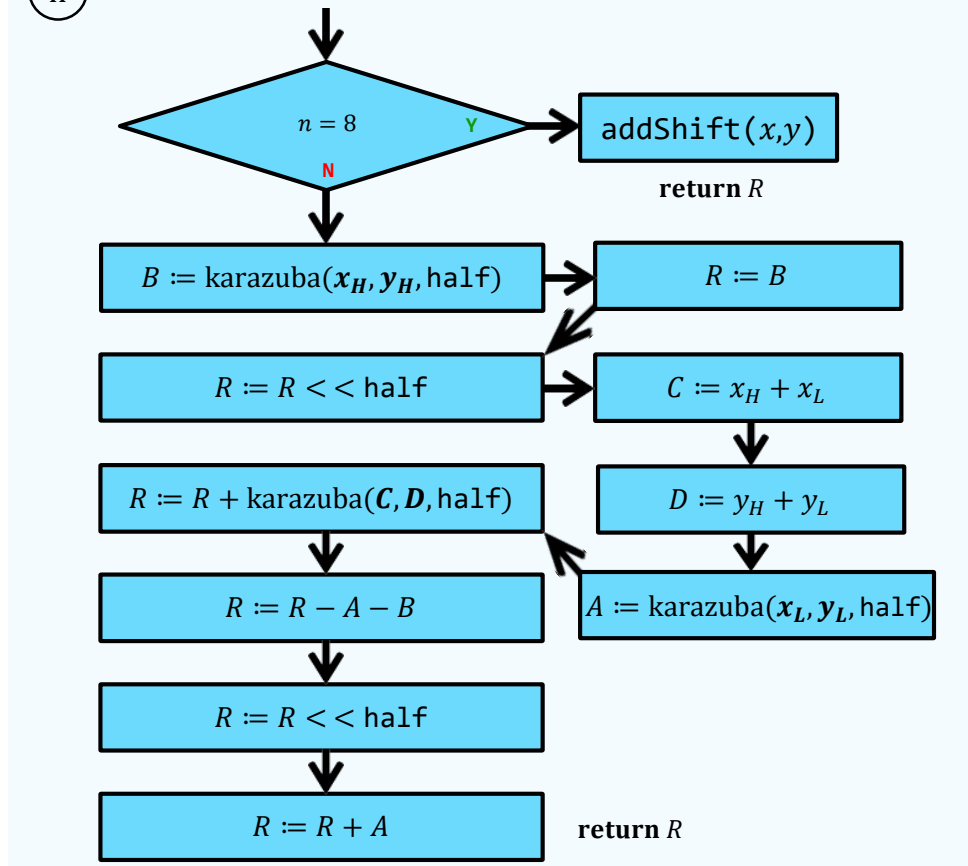
$$2^n x_H y_H + 2^{\frac{n}{2}}\left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H\right) + x_L y_L$$

- **Rekursive Berechnung Teilprodukte (1)**
  Berechne Teilprodukte A, B und C

Code

```
48  // formula: B*2^n + (C_prod - A - B)*2^(n/2) + A
49  // product of lower half (xL * yL)
50  uint64_t A = karazuba(xL, yL, half);
51  // product of upper half (xH * yH)
52  uint64_t B = karazuba(xH, yH, half);
53  // sums for 3rd partial product
54  uint64_t C = (xH + xL);
55  uint64_t D = (yH + yL);
56  …
```

II



Speicherhungrige Variante

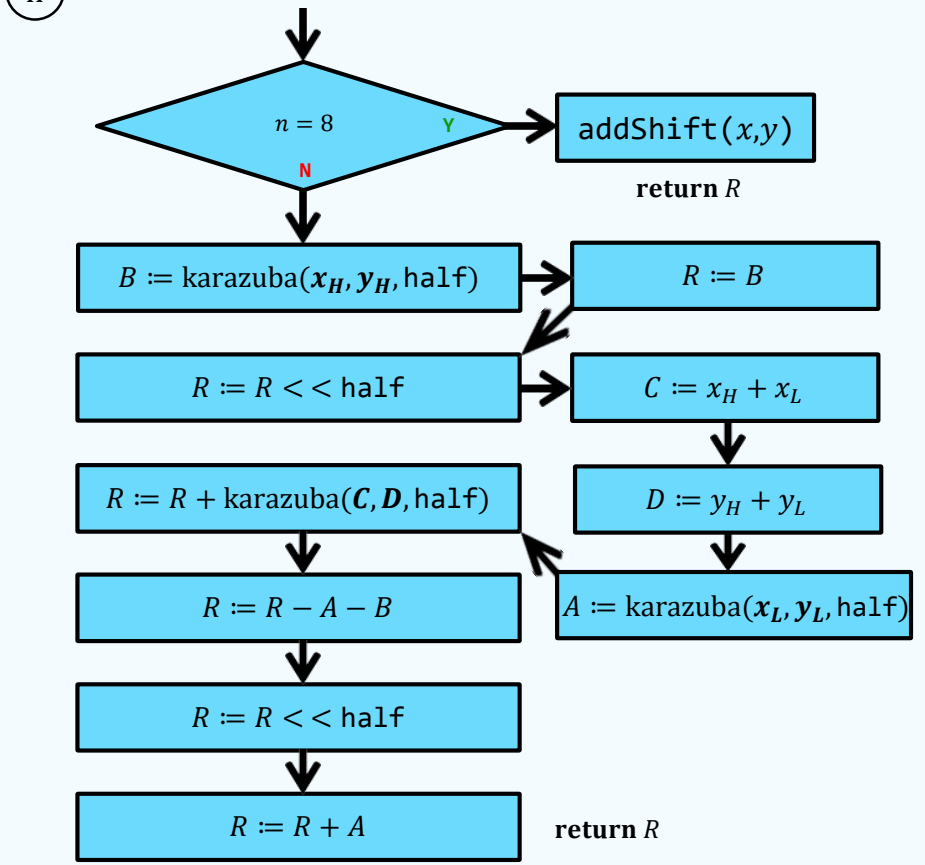
$$2^n x_H y_H + 2^{\frac{n}{2}}\left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H\right) + x_L y_L$$

- **Rekursive Berechnung Teilprodukte (2)**
  Berechne Teilprodukte A, B und C

Code

```
56  // formula: B*2^n + (C_prod - A - B)*2^(n/2) + A
57  // B*2^n
58  result = B;
59  result <<= half;
60  // (xH + xL) * (yH + yL) respectively C * D
61  // adding two n/2-bit numbers can result (n/2 + 1)-bit
        number (Overflow)
62  if (((C >> half) & 1) || ((D >> half) & 1)) {
63    // overflow:recurse with double size; prevent data loss
64    result += karazuba((xH + xL), (yH + yL), half << 1);
65  else {
66    // no overflow: recurse standard size (half)
67    result += karazuba((xH + xL), (yH + yL), half);
68  }
69  …
```

II



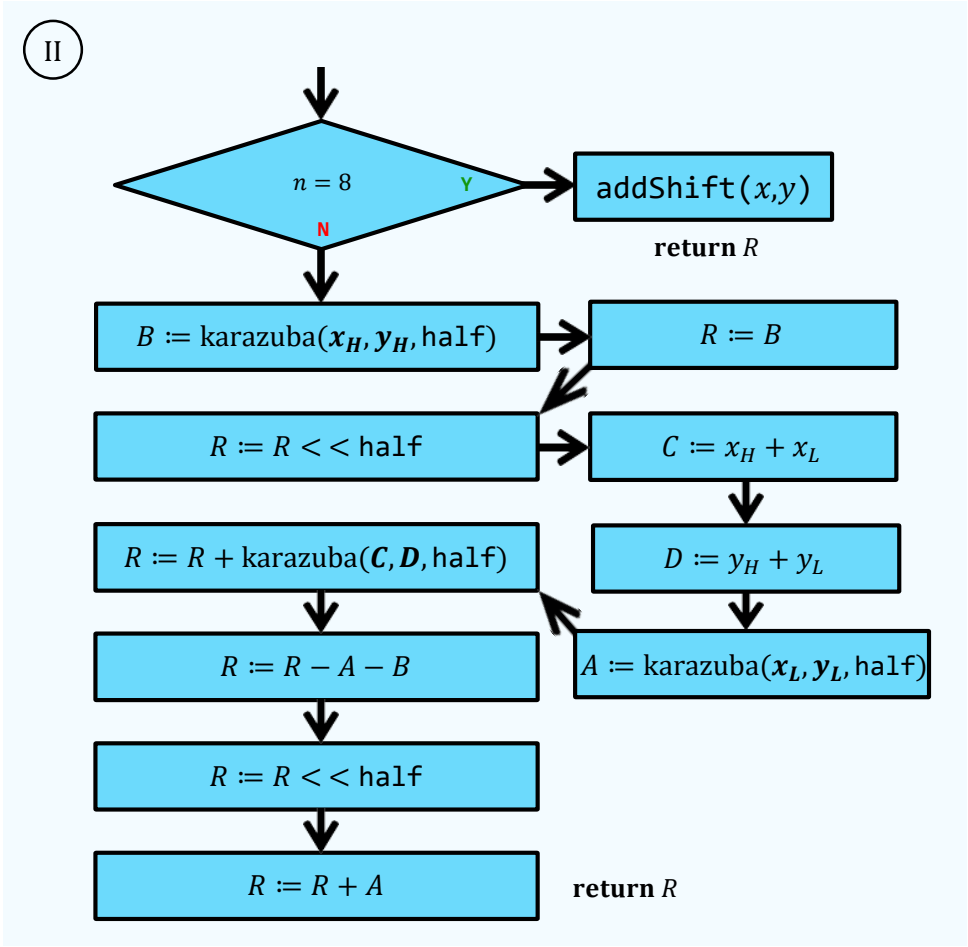
Speicherhungrige Variante

$$2^n x_H y_H + 2^{\frac{n}{2}}\left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H\right) + x_L y_L$$

- **Rekursive Berechnung Teilprodukte (3)**
  Berechne Teilprodukte A, B und C

Code

```
69  // formula: B*2^n + (C_prod - A - B)*2^(n/2) + A
70  // (C_prod - A - B)
71  result = result - A - B;
72  // (C_prod - A - B)*2^(n/2)
72  result <<= half;
73  // + A
75  result += A;
```



Speicherhungrige Variante

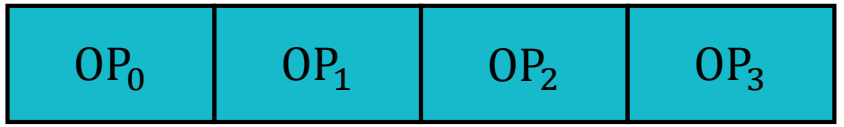
$$2^n x_H y_H + 2^{\frac{n}{2}}\left((x_H + x_L)(y_H + y_L) - x_L y_L - x_H y_H\right) + x_L y_L$$

Zerlegung `memcopy` und Endianess

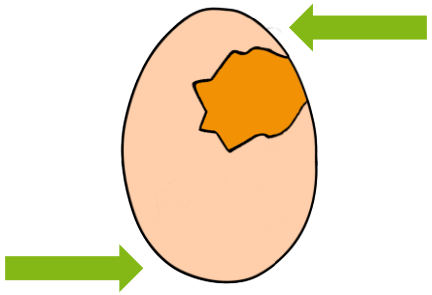
Big-Endian

vs

Little-Endian




```
if (is_little_endian()) {
    memcpy(&xL, px, half / 8);
    memcpy(&yL, py, half / 8);
    memcpy(&xH, px + (half / 8), half / 8);
    memcpy(&yH, py + (half / 8), half / 8);
} else {
    memcpy(&xH, px, half / 8);
    memcpy(&yH, py, half / 8);
    memcpy(&xL, px + (half / 8), half / 8);
    memcpy(&yL, py + (half / 8), half / 8);
}
```



Die Vermutung von Kolmogorov

Kolmogorov vermutete um 1956 folgendes:
Zur Berechnung des Produktes zweier Binärzahlen existieren keine Algorithmen die schneller sind als $\Theta(n^2)$

Kolmogorov vermutete um 1956 folgendes:
Zur Berechnung des Produktes zweier Binärzahlen existieren keine Algorithmen die schneller sind als $\Theta(n^2)$

- Tatsächlich widerlegt der Karazuba-Algorithmus diese Vermutung!
- Der Karazuba-Algorithmus besitzt eine Laufzeit von $\Theta\left(n^{\log_2 3}\right) \approx \Theta(n^{1.58})$
- Nach dem Karazuba-Algorithmus wurden viele weitere noch schnellere Algorithmen für die Multiplikation zweier Zahlen gefunden

- Toom-Cook-**Algorithmus** (1965): Bitvektoren werden nicht nur in zwei, sondern in mehrere Teile zerlegt → Eine **Optimale Zerlegung** erfordert dann für $v + 1$ Teile gerade einmal $2v + 1$ Multiplikationen
  - **Variante mit fester Teilung**: $\mathcal{O}\left(n^{c'+1}\right)$ mit von Teilung abhängigem c'
  - **Variante mit variabler Teilung**: $\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}}_{+}\right)$ im Worst-Case

Zerlege x und y in k Teile

```
37  for (size_t i = 0; i < k; i++) {
38    xParts[i] = (x >> addShift(i, partSize)) & mask;
39    yParts[i] = (y >> addShift(i, partSize)) & mask;
40  }
```

Berechne die Interpolationspunkte

```
41  for (size_t i = 0; i < k; i++) {
42    for (size_t j = 0; j < k; j++) {
43      p[i + j] += toomCook(xParts[i], yParts[j], partSize);
44    }
45  }
```

Rekonstruiere das Ergebnis durch Interpolation

```
46  for (size_t i = 0; i < v; i++) {result += (p[i]<<addShift(i, partSize)); }
```

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$
1969	Toom-Cook-Algorithmus (Knuth)	$\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}}_{+}\right)$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$
1969	Toom-Cook-Algorithmus (Knuth)	$\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}}_{+}\right)$
1971	Schönehagen-Strassen-Algorithmus	$\mathcal{O}(n \log n \log\log n)$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$
1969	Toom-Cook-Algorithmus (Knuth)	$\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}_{+}}\right)$
1971	Schönehagen-Strassen-Algorithmus	$\mathcal{O}(n \log n \log\log n)$
2007	Fürer-Algorithmus	$\mathcal{O}\left(n \log n \cdot K^{\log^* n}\right)$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$
1969	Toom-Cook-Algorithmus (Knuth)	$\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}}_{+}\right)$
1971	Schönehagen-Strassen-Algorithmus	$\mathcal{O}(n \log n \log\log n)$
2007	Fürer-Algorithmus	$\mathcal{O}\left(n \log n \cdot K^{\log^* n}\right)$
2017	Van der Hoeven-Lecerf-Algorithmus	$\mathcal{O}\left(n \log n \cdot 8^{\log^* n}\right)$

Jahr	Algorithmus	Laufzeit
?	Russische Bauernmultiplikation (Add-Shift)	$\mathcal{O}(n^2)$
1962	Karazuba-Algorithmus	$\mathcal{O}\left(n^{\log_2 3}\right) \approx \mathcal{O}(n^{1.58})$
1969	Toom-Cook-Algorithmus (Knuth)	$\mathcal{O}\left(n \cdot \log_2(n) \cdot 2^{\sqrt[2]{2\log_2 n}_{+}}\right)$
1971	Schönehagen-Strassen-Algorithmus	$\mathcal{O}(n \log n \log\log n)$
2007	Fürer-Algorithmus	$\mathcal{O}\left(n \log n \cdot K^{\log^* n}\right)$
2017	Van der Hoeven-Lecerf-Algorithmus	$\mathcal{O}\left(n \log n \cdot 8^{\log^* n}\right)$
2019/21	Van der Hoeven-Algorithmus	$\Theta(n \log n)$

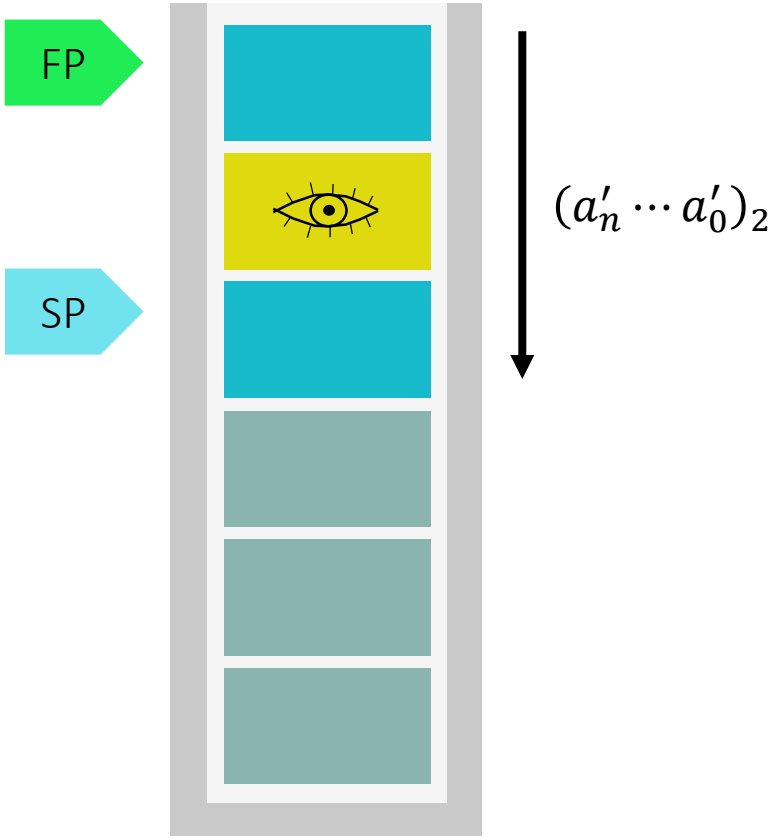
Arbeitsweise des Stacks

- Schauen Sie sich die Stack-Animation (`stack.pdf`) im Moodle an
- Welchen Bereich grenzen der **Stackpointer** und **Framepointer** im Stack ein?

- Wodurch wird die maximale Tiefe einer Rekursion beschränkt?

- Schauen Sie sich die Stack-Animation (`stack.pdf`) im Moodle an
- Welchen Bereich grenzen der **Stackpointer** und **Framepointer** im Stack ein?
  - Stackpointer und Framepointer grenzen jeweils den Arbeitsbereich der aktuell gerufenen Funktion ein
  - Dieser Bereich wird als **Stapelrahmen** (**Stackframe**) bezeichnet
  - In den Arbeitsbereichen liegen jeweils die lokalen Variablen der Funktion
- Wodurch wird die maximale Tiefe einer Rekursion beschränkt?

Stapelrahmen (Stackframe)



- Schauen Sie sich die Stack-Animation (`stack.pdf`) im Moodle an
- Welchen Bereich grenzen der **Stackpointer** und **Framepointer** im Stack ein?
  - Stackpointer und Framepointer grenzen jeweils den Arbeitsbereich der aktuell gerufenen Funktion ein
  - Dieser Bereich wird als **Stapelrahmen** (**Stackframe**) bezeichnet
  - In den Arbeitsbereichen liegen jeweils die lokalen Variablen der Funktion
- Wodurch wird die maximale Tiefe einer Rekursion beschränkt?

- Schauen Sie sich die Stack-Animation (`stack.pdf`) im Moodle an
- Welchen Bereich grenzen der **Stackpointer** und **Framepointer** im Stack ein?
  - Stackpointer und Framepointer grenzen jeweils den Arbeitsbereich der aktuell gerufenen Funktion ein
  - Dieser Bereich wird als **Stapelrahmen** (**Stackframe**) bezeichnet
  - In den Arbeitsbereichen liegen jeweils die lokalen Variablen der Funktion
- Wodurch wird die maximale Tiefe einer Rekursion beschränkt?
  - Die Rekursionstiefe ist schlicht durch die Tatsache beschränkt, dass der Stack irgendwann ggf. in den Heap hineinwächst und dort Daten überschreibt → **Stapelüberlauf** (**Stack overflow**)

Stapelüberlauf