Klausur Informatik 2
25. Juli 2002

Gorlatch / Flik
Jucknath / Neunast

Name: ....

Vorname: ....

Matr.-Nr.: ....

Bearbeitungszeit: 120 Minuten

- Benutzen Sie für die Lösung der Aufgaben nur das mit diesem Deckblatt ausgeteilte Papier. **Lösungen, die auf anderem Papier geschrieben werden, können nicht gewertet werden!**
- Schreiben Sie Ihre Lösungen auf das Aufgabenblatt der jeweiligen Aufgabe. Verwenden Sie auch die Rückseiten. **Schreiben Sie keine Lösungen einer Aufgabe auf ein Blatt, das nicht zu dieser Aufgabe gehört!** Wenn Sie zusätzliche, von uns ausgegebene Blätter verwenden, geben Sie unbedingt an, zu welcher Aufgabe die Lösung gehört!
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen werden nicht gewertet!
- Schreiben Sie nur in **blau** oder **schwarz**. Lösungen, die mit Bleistift geschrieben sind, können nicht gewertet werden!
- Die Java-Implementierungen müssen nicht dokumentiert/kommentiert werden.
- Erscheint Ihnen eine Aufgabe mehrdeutig, wenden Sie sich an die Betreuer.
- Tragen Sie jetzt (vor Beginn der Bearbeitungszeit) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.

Aufgabe	Punkte	erreicht
1	15	
2	9	
3	11	
4	5	
5	4	
6	8	

1. Aufgabe (15 Punkte): Algorithmen und Datenstrukturen

1.1. InsertionSort auf Listen (7 Punkte) Gegeben sei die folgende Datenstruktur IntList, mit deren Hilfe Sequenzen von int-Variablen in Listen-Form verwaltet werden können.

```
public class IntList {
    protected class IntListElem {
        IntListElem next;
        int data;
    }
    IntListElem first;
}
```

Schreiben Sie unter Verwendung dieser Datenstruktur eine Methode insertionSort(IntList l), die das Verfahren InsertionSort auf der übergebenen Liste implementiert. Auf Listen ist InsertionSort besonders einfach zu implementieren, indem nacheinander jeweils das erste Element der Liste entnommen und sortiert in eine temporäre Liste eingefügt wird. Zum Schluss muss diese temporäre Liste als Ergebnis zurückgegeben werden.

Lösung:

```
// Implementation von InsertionSort auf IntList. Iterative Variante 1:
// Es wird parallel eine zweite (sortierte) Liste generiert, die als
// Ergebnis zurückgegeben wird.
public static void insertionSort(IntList l) {
    if (l == null) return;
    // curr enthält neu einzusortierendes Element
    // sorted enthält die bereits sortierte Teilliste
    IntListElem curr = null, sorted = null;
    // Entnehme ein Element nach dem nächsten der Liste und füge es in
    // die sortierte Teilliste ein.
    while (l.first != null) {
        curr = l.first;
        l.first = l.first.next;
        sorted = insertSorted(curr, sorted);
    }
    // Nachdem die ursprüngliche Liste leer ist, enthält die sortierte
    // Teilliste alle Elemente in sortierter Reihenfolge, diese wird
    // zurückgeliefert.
    l.first = sorted;
}

// Hilfsfunktion: Füge Element in sortierte Teilliste ein.
protected static IntListElem insertSorted(IntListElem el, IntListElem l) {
    if (el == null)
        return l;
    // Spezialfall: Leere Liste oder einfügen ganz vorne
    if ((l == null) || (l.data > el.data)) {
        el.next = l;
        return (el);
    }
    // Einfügen in Mitte oder am Ende: Erst Suchen des Platzes
    IntListElem iterator = l;
    while ((iterator.next != null) && (iterator.next.data <= el.data)) {
        iterator.next = l;
    }
    // Einfügen bzw. Anhängen von el
    el.next = iterator.next;
    iterator.next = el;
    // Zurückliefern der neuen Liste
    return (l);
}
```

1.2. Exceptions (2 Punkte) Um einer aufrufenden Methode Fehler während des Programmablaufs (Ungültige Argumente, Division durch 0, Zugriff über null-Referenz, ...) anzuzeigen, gibt es mehrere Möglichkeiten. Welche Vorteile bietet die Verwendung von Exceptions gegenüber anderen Verfahren, beispielsweise der Rückgabe eines eigentlich ausserhalb des Wertebereichs der Funktion liegenden Werts, oder der Verwendung einer globalen Fehlervariablen?

Lösung:

Exceptions verschieben die Fehlerverarbeitung, für die sonst oft allein der Programmierer/Bibliotheksersteller zuständig ist, ein wenig in Richtung der Compilerzuständigkeit. Vorteile, die andere Verfahren zum Teil nicht oder nur mit entsprechendem Aufwand bieten, sind:

- *Trennung von „normalem“ und Fehlerbehandlungs-Code, was die Übersicht und Verständlichkeit von Programmen erhöht.*
- *Die Möglichkeit, verschiedene Fehlerklassen zu definieren, je nach Art der Exception.*
- *Aus der Schnittstelle einer Funktion wird ersichtlich, welche Fehlersituationen auftreten können.*
- *Die Flexibilität bei der Behandlung von Fehlern: Ignorieren, Reagieren, Sammeln verschiedener Klassen oder gezieltes Behandeln einzelner Fehlerursachen, finally-Blöcke, ...*
- *Thread-Sicherheit; d.h. auch bei paralleler Programmabarbeitung beeinflusst ein Fehler in einem Thread nicht die anderen (es sei denn, dies ist gewollt, und so implementiert ...).*
- *Es kann nicht „mit undefinierten Ergebnissen weitergerechnet werden“; es muss nicht bei jedem zurückgelieferten Funktionsergebnis geprüft werden, ob eine Fehlersituation vorliegt. Insbesondere ist es manchmal unmöglich, einen besonderen Wert aus dem Wertebereich einer Funktion vorzusehen, der einen Fehler anzeigen soll.*

1.3. Stabilität (4 Punkte) Wenn ein Sortieralgorithmus *stabil* ist, bedeutet dies, dass an sich verschiedene Elemente, die aber durch das Sortierkriterium als gleich eingestuft werden, nach dem Sortieren untereinander die gleiche Reihenfolge haben, wie vor dem Sortieren. Relevant ist dies besonders, wenn z.B. Personen nur nach Nachnamen sortiert werden; Personen mit gleichem Nach-, aber unterschiedlichem Vornamen, sollen dann in der gleichen Reihenfolge wie vor dem Sortieren erscheinen.

Kann MergeSort stabil implementiert werden? Wie sieht es mit HeapSort aus? Begründen Sie Ihre Antwort, bzw. geben Sie ein Gegenbeispiel. Hinweis: Für das Verständnis kann es hilfreich sein, beide Verfahren auf einer Beispielliste, die ein Element mehrfach enthält, zu simulieren.

Lösung:

Bei entsprechender Abstimmung des „Split“ und des „Merge“ ist MergeSort stabil. Beispielsweise könnte Split die Liste immer in eine vordere und eine hintere Hälfte trennen. Wenn diese (stabil) sortiert wurden, muss Merge bei Gleichheit der ersten Elemente beider Listen zuerst das entsprechende Element aus der vorderen Liste nehmen (da dies ja auch bei der ursprünglichen Liste vor dem anderen Element stand).

HeapSort ist im allgemeinen nicht stabil. Ein Gegenbeispiel: die Sequenz $(1_1, 1_2, 1_3)$ *(der Index dient zur Unterscheidung der eigentlich gleichen „1“en) ist bereits ein Heap, wird also beim Aufbau des Heaps nicht modifiziert. Beim Entnehmen der Elemente aus dem Heap wird zuerst das Element an der Wurzel (eigentlich das größte ...) genommen, und ganz hinten in die Ergebnissequenz gepackt. Damit ergibt sich als Ergebnis des Sortierens die Sequenz* $(1_2, 1_3, 1_1)$*, die Elemente stehen also nicht mehr in der ursprünglichen Reihenfolge und HeapSort ist im allgemeinen nicht stabil.*

1.4. Abstrakte Datentypen (2 Punkte) Was sind abstrakte Datentypen, und weshalb verwendet man sie, insbesondere in größeren Softwareprojekten?

Lösung:

Abstrakte Datentypen sind Datentypen, von denen nur die Schnittstelle gegeben ist, nicht aber deren Implementierung. Die Schnittstelle enthält Konstruktoren, Destruktoren, Elementzugriffs- und -modifikationsmethoden, Zustandsabfragen, sowie für jede Methode eine Spezifikation (welche Aufgabe sie erledigt, nicht aber, wie diese erledigt wird).

Gründe für ihren Einsatz können sein:

- *Modularisierung von Programmen; besonders bei Einsatz vieler Programmierer, um Zuständigkeiten zu regeln*
- *Ermöglicht, Teile einer Implementation auszutauschen, ohne die darauf aufbauenden Teile anpassen zu müssen (sofern die Schnittstelle nicht geändert wird).*
- *Dokumentation und Übersichtlichkeit von Programmen werden verbessert; die Fehlerhäufigkeit sinkt.*

2. Aufgabe (9 Punkte): Aufwandsabschätzung

2.1. $\mathcal{O}$-Kalkül (2 Punkte) Beweisen oder widerlegen Sie (Durch Argumentation, Beweis, oder Gegenbeispiel): Wenn u eine obere Schranke von f ist, und v eine obere Schranke von g ist, dann ist $\frac{u}{v}$ eine obere Schranke von $\frac{f}{g}$. Oder formal:

$$\Big(f \in \mathcal{O}(u) \wedge g \in \mathcal{O}(v)\Big) \Longrightarrow \frac{f}{g} \in \mathcal{O}\Big(\frac{u}{v}\Big)$$

Lösung:

Allgemein ist dies nicht der Fall. Ein Gegenbeispiel sind $f(n) = u(n) = v(n) = n^2$ und $g(n) = n$. Dann ist $n^2 \in \mathcal{O}(n^2)$ und $n \in \mathcal{O}(n^2)$, aber $\frac{n^2}{n} = n \notin \mathcal{O}(\frac{n^2}{n^2}) = \mathcal{O}(1)$.

2.2. Die Klassen $\mathcal{O}(g)$, $\Omega(g)$, und $\Theta(g)$ (2 Punkte) Bestimmen Sie für die im folgenden angegebenen Funktionen, ob f Element der Klassen $\mathcal{O}(g)$, $\Omega(g)$, und $\Theta(g)$ ist:

- $f(n) = 3n^{\frac{5}{2}} + 4n - 5$
  $g(n) = n^2$

Lösung:

- $f(n) = 3n^{\frac{5}{2}} + 4n - 5$, $g(n) = n^2$:

$$\lim_{n\to\infty} \frac{3n^{\frac{5}{2}} + 4n - 5}{n^2} = \lim_{n\to\infty} \frac{3n^2\sqrt{n}}{n^2} + \lim_{n\to\infty} \frac{4n}{n^2} + \lim_{n\to\infty} \frac{-5}{n^2} = \lim_{n\to\infty} 3\sqrt{n} + \lim_{n\to\infty} \frac{4}{n} - 0 = \lim_{n\to\infty} 3\sqrt{n} + 0 \longrightarrow \infty$$

Der Grenzwert f/g divergiert. Damit ist $f \in \Omega(g)$, aber $f \notin \mathcal{O}(g)$ und damit auch $f \notin \Theta(g)$.

2.3. Aufwand von CountingSort (5 Punkte) Im folgenden finden Sie eine Implementation des Sortierverfahrens CountingSort. Geben Sie an, von welchen Größen der Aufwand dieser Methode abhängt, leiten Sie die Aufwandsfunktion (für den „Worst Case“) in Abhängigkeit dieser Variablen her, und bestimmen Sie anschließend die Aufwandsklasse. Sorgen Sie dafür, dass Ihre Herleitungen nachvollziehbar sind (Zeilenbezug, verwendete Regeln, ...).

```
static public int[] countingSort(int[] arr, int maxvalue) {
    int[] counts = new int[maxvalue];
    int[] res = new int[arr.length];
    for (int i=0; i<maxvalue; ++i) {
        counts[i] = 0;
    }
    for (int i=0; i<arr.length; ++i) {
        ++counts[arr[i]];
    }
    for (int i=1; i<maxvalue; ++i) {
        counts[i] += counts[i-1];
    }
    for (int i=arr.length-1; i>=0; --i) {
        res[--counts[arr[i]]] = arr[i];
    }
    return (res);
}
```

Lösung:

Mit der Vereinbarung $n :=$ arr.length und der Feststellung, dass der Aufwand sowohl von der Länge des Arrays

n, als auch dem Wert der Variablen maxvalue abhängt, ergibt sich für die Aufwandsfunktion:

$$\begin{aligned}
A_{countingSort}(n, \mathit{maxvalue}) &= \underbrace{2}_{2} + \underbrace{3}_{3} + \underbrace{2}_{4} + \sum_{i=0}^{\mathit{maxvalue}-1} \Big(\underbrace{2}_{4} + \underbrace{2}_{5}\Big) + \underbrace{3}_{7} + \sum_{i=0}^{n-1}\Big(\underbrace{3}_{7} + \underbrace{3}_{8}\Big) + \\
&\quad \underbrace{2}_{10} + \sum_{i=1}^{\mathit{maxvalue}-1} \Big(\underbrace{2}_{10} + \underbrace{4}_{11}\Big) + \underbrace{4}_{13} + \sum_{i=0}^{n-1}\Big(\underbrace{2}_{13} + \underbrace{6}_{14}\Big) + \underbrace{1}_{16} \\
&= 17 + 4 + \sum_{i=1}^{\mathit{maxvalue}-1} (10) + \sum_{i=0}^{n-1}(14) \\
&= 21 + \big((\mathit{maxvalue} - 1) - (1) + 1\big) \cdot 10 + \big((n-1) - 0 + 1\big) \cdot 14 \\
&= 11 + 10\mathit{maxvalue} + 14n
\end{aligned}$$

Regel 9, denn $k = 1$ ($a_1 = 10 > 0$ bzw. $a_1 = 14 > 0$):

$$A_{countingSort}(n) \in \mathcal{O}(n)$$

betrachtet in Abhängigkeit von n, bzw.

$$A_{countingSort}(\mathit{maxvalue}) \in \mathcal{O}(\mathit{maxvalue}),$$

wenn die Abhängigkeit von maxvalue betrachtet wird. Insgesamt kann man so schreiben:

$$A_{countingSort}(n, \mathit{maxvalue}) \in \mathcal{O}(n + \mathit{maxvalue})$$

Damit ist der Aufwand linear sowohl in n, als auch in dem Wert des größten vorkommenden Elements.

3. Aufgabe (11 Punkte): Programmverifikation

3.1. Formalisierung von Aussagen (4 Punkte) Formalisieren Sie die folgenden Aussagen (Hilfsprädikate sind mit zu definieren):

1. m ist die mittlere von drei ganzen Zahlen a, b, c.
2. Der char[] s ist in einem char[] str enthalten.

Lösung:

1. $M = \{a, b, c\} \wedge m \in M \wedge \exists x, y \in M : \big(x < m \wedge y > m\big)$ ***oder***
   $M = \{a, b, c\} \wedge (m \in M) \wedge \big(|\{i \in M | i < m\}| \leq 1\big) \wedge \big(|\{i \in M | i > m\}| \leq 1\big)$
2. $s \in \mathit{char}^N \wedge str \in \mathit{char}^M \wedge \exists i \in \{0, \ldots, M - N\} : \forall j \in \{0, \ldots, N - 1\} : s_j = str_{i+j}$

3.2. Partielle Korrektheit (8 Punkte) Im folgenden finden Sie eine Methode fibonacci, die einen Array der ersten n Fibonacci-Zahlen zurückgibt. Bestimmen Sie die für die POST-Condition notwendige PRE-Condition und beweisen Sie damit die Korrektheit der Methode (in der Vorgabe auf der folgenden Seite). Dabei können Sie die folgende Definition der Fibonacci-Zahlen benutzen:

$$\mathrm{fib}(n) = \begin{cases} 1; & n = 0 \\ 1; & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2); & n \geq 2 \end{cases}$$

```
public static int [] fib (int n) {
    // PRE: ?
    int [] arr = new int[n];
    arr [0] = 1;
    arr [1] = 1;
    int i = 2;
    while (i < n) {
        arr[i] = arr[i-1] + arr[i-2];
        i = i + 1;
    }
    // arr ∈ int^n ∧ arr_0 = 1 ∧ arr_1 = 1 ∧ ∀k ∈ {2,...n−1} : arr_k = arr_{k−1} + arr_{k−2}
    // POST: arr ∈ int^n ∧ ∀k ∈ {0,...n−1} : arr_k = fib(k)
    return arr;
}
```

Lösung:

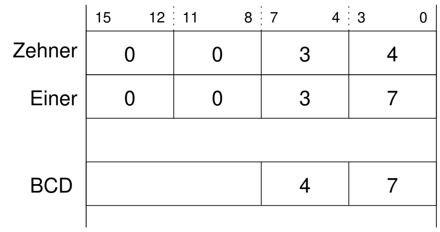
Wir verwenden die folgende Abkürzung: $S \equiv arr \in \mathit{int}^n \wedge arr_0 = 1 \wedge arr_1 = 1$:

```
public static int[] fib(int n) {
    // PRE: $2 \leq n$
    // new int[n] $\in$ int$^n \wedge 2 \leq n$
    int[] arr = new int[n];
    // $arr \in$ int$^n \wedge 2 \leq n$
    // $arr \in$ int$^n \wedge 1 = 1 \wedge 2 \leq n$
    arr[0] = 1;
    // $arr \in$ int$^n \wedge arr_0 = 1 \wedge 2 \leq n$
    // $arr \in$ int$^n \wedge arr_0 = 1 \wedge 1 = 1 \wedge 2 \leq n$
    arr[1] = 1;
    // $arr \in$ int$^n \wedge arr_0 = 1 \wedge arr_1 = 1 \wedge 2 \leq n$
    // $S \wedge 2 \leq n \wedge \forall k \in \{2, \ldots 2-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    int i = 2;
    // $I : S \wedge i \leq n \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    while (i < n) {
        // $B : (i < n) \wedge I : \big(S \wedge i \leq n \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}\big)$
        // $S \wedge i < n \wedge arr_{i-1} + arr_{i-2} = arr_{i-1} + arr_{i-2} \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}$
        arr[i] = arr[i-1] + arr[i-2];
        // $S \wedge i < n \wedge arr_i = arr_{i-1} + arr_{i-2} \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}$
        // $S \wedge i+1 \leq n \wedge \forall k \in \{2, \ldots (i+1)-1\} : arr_k = arr_{k-1} + arr_{k-2}$
        i = i + 1;
        // $I : S \wedge i \leq n \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    }
    // $\neg B : (i \geq n) \wedge I : \big(S \wedge i \leq n \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}\big)$
    // $S \wedge i = n \wedge \forall k \in \{2, \ldots i-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    // $S \wedge \forall k \in \{2, \ldots n-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    // $arr \in$ int$^n \wedge arr_0 = 1 \wedge arr_1 = 1 \wedge \forall k \in \{2, \ldots n-1\} : arr_k = arr_{k-1} + arr_{k-2}$
    // POST: $arr \in$ int$^n \wedge \forall k \in \{0, \ldots n-1\} : arr_k =$ fib$(k)$
    return arr;
}
```

4. Aufgabe (5 Punkte): Assemblerprogrammierung

Hinweis: Gegeben ist die weiter hinten stehende VIP-Befehlstabelle.

4.1. (3 Punkte) Im Hauptspeicher befindet sich an den symbolischen Adressen `Zehner` und `Einer` je eine ASCII-codierte Dezimalziffer (jeweils ein 16-Bit-Wort).
Schreiben Sie ein vollständiges VIP-Assemblerprogramm, das die beiden Ziffern in gepackter Form, d.h. als zweistellige BCD-Zahl in der niederwertigen Worthälfte der Speicherzelle `BCD` ablegt. Beispiel:




```
        ORG   0
Zehner  DAT   '4'
Einer   DAT   '7'
BCD     RES   1
```

Lösung:

```
        ORG   0
Zehner  DAT   '4'
Einer   DAT   '7'
BCD     RES   1
Start   LDA   Zehner
        AND   #0x0F
        LSL
        LSL
        LSL
        LSL
        STA   BCD
        LDA   Einer
        AND   #0x0F
        OR    BCD
        STA   BCD
        HLT
        END   Start
```

4.2. Adressierungsarten (2 Punkte) Gegeben ist das folgende VIP-Assemblerprogramm. Geben Sie die nach Ausführung des Programms in den Speicherzellen mit den Adressen 0 bis 3 befindlichen Inhalte an.

Hinweis: Beachten Sie, daß hier numerische Adressen verwendet sind und keine symbolischen!
Alle Zahlenangaben sind im Dezimalsystem.

```
ORG  4
LDX  #3
LDA  #2
STX  1
STA  @1
ADD  0[IX]
STA  0
SUBX 3
STX  1[IX]
HLT
END  4
```

Lösung:

```
ORG  4
LDX  #3    ; IX := 3
LDA  #2    ; AC := 2
STX  1     ; RAM[1] := IX = 3
STA  @1    ; RAM[RAM[1]] = RAM[3] := AC = 2
ADD  0[IX] ; AC := AC + RAM[0+IX] = 2 + RAM[3] = 4
STA  0     ; RAM[0] := AC = 4
SUBX 3     ; IX := IX - RAM[3] = 3 - 2 = 1
STX  1[IX] ; RAM[1+IX] = RAM[2] := IX = 1
HLT
END  4
```

Adresse	*Inhalt*
0	*4*
1	*3*
2	*1*
3	*2*

Befehl	Code (hex.)	SR-Inhalt z	n	c	v	Adressierung	#	@	[IX]	Wirkung
HLT	00	-	-	-	-	-	-	-	-	Anhalten
NOP	01	-	-	-	-	-	-	-	-	Nichts
NOT	02	x	x	0	0	-	-	-	-	AC := $\neg$ AC, 1-Komplement
NEG	03	x	x	x	x	-	-	-	-	AC := 0 - AC, 2-Komplement
ASL	04	x	x	AC_{15}	x	-	-	-	-	AC := AC $\times$ 2
ASR	05	x	x	AC_0	0	-	-	-	-	AC := AC $\times$ 2^{-1}, AC_{15} := AC_{15}
LSL	06	x	x	AC_{15}	0	-	-	-	-	AC := AC $\times$ 2
LSR	07	x	x	AC_0	0	-	-	-	-	AC := AC $\times$ 2^{-1}, AC_{15} := 0
JMP	10	-	-	-	-	✓	-	-	-	PC := Op
BPL	11	-	-	-	-	✓	-	-	-	PC := Op, falls $\geq$ 0
BMI	12	-	-	-	-	✓	-	-	-	PC := Op, falls $<$ 0
BCC	13	-	-	-	-	✓	-	-	-	PC := Op, falls kein Übertrag
BCS	14	-	-	-	-	✓	-	-	-	PC := Op, falls Übertrag
BVC	15	-	-	-	-	✓	-	-	-	PC := Op, falls kein Überlauf
BVS	16	-	-	-	-	✓	-	-	-	PC := Op, falls Überlauf
BEQ	17	-	-	-	-	✓	-	-	-	PC := Op, falls = (Null)
BNE	18	-	-	-	-	✓	-	-	-	PC := Op, falls $\neq$ (Null)
BGT	19	-	-	-	-	✓	-	-	-	PC := Op, falls $>$
BGE	1A	-	-	-	-	✓	-	-	-	PC := Op, falls $\geq$
BLE	1B	-	-	-	-	✓	-	-	-	PC := Op, falls $\leq$
BLT	1C	-	-	-	-	✓	-	-	-	PC := Op, falls $<$
BGTU	1D	-	-	-	-	✓	-	-	-	PC := Op, falls $>$ (vorzeichenlos)
BGEU	13	-	-	-	-	✓	-	-	-	PC := Op, falls $\geq$ (vorzeichenlos)
BLEU	1E	-	-	-	-	✓	-	-	-	PC := Op, falls $\leq$ (vorzeichenlos)
BLTU	14	-	-	-	-	✓	-	-	-	PC := Op, falls $<$ (vorzeichenlos)
LDA	20..23	x	x	0	0	✓	✓	✓	✓	AC := Op
LDX	28..2B	x	x	0	0	✓	✓	✓	✓	IX := Op
STA	30..33	-	-	-	-	✓	-	✓	✓	Op := AC
STX	38..3B	-	-	-	-	✓	-	✓	✓	Op := IX
ADD	40..43	x	x	x	x	✓	✓	✓	✓	AC := AC + Op
ADDX	48..4B	x	x	x	x	✓	✓	✓	✓	IX := IX + Op
SUB	50..53	x	x	x	x	✓	✓	✓	✓	AC := AC - Op
SUBX	58..5B	x	x	x	x	✓	✓	✓	✓	IX := IX - Op
CMP	60..63	x	x	x	x	✓	✓	✓	✓	AC - Op
CMPX	68..6B	x	x	x	x	✓	✓	✓	✓	IX - Op
AND	70..73	x	x	0	0	✓	✓	✓	✓	AC := AC $\wedge$ Op (bitpaarweise)
OR	80..83	x	x	0	0	✓	✓	✓	✓	AC := AC $\vee$ Op (bitpaarweise)
XOR	90..93	x	x	0	0	✓	✓	✓	✓	AC := AC $\oplus$ Op (bitpaarweise)

Tabelle 1: VIP-Befehlssatz-Referenz

5. Aufgabe (4 Punkte): Erweiterung des VIP

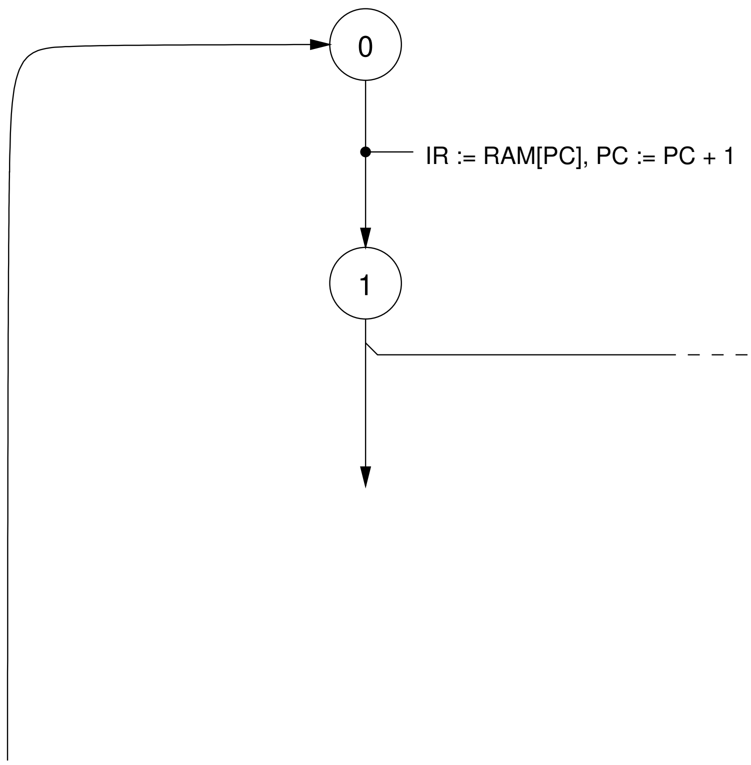
Der VIP soll um einen Befehl "**D**ecrement inde**X** and **B**ranch if **N**ot equal" `DBNX m` mit der Codierung `0x1F` erweitert werden. Er soll die gleiche Wirkung wie die nachstehende Befehlsfolge haben:

```
SUBX  #1
BNE   m
```

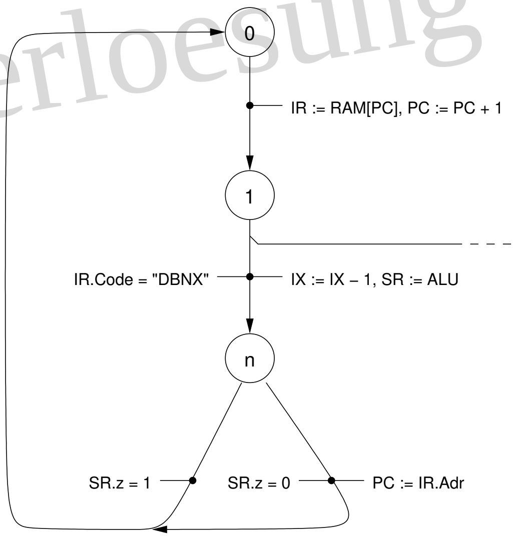
Hinweis: Ausschnitt aus der ALU-Steuertabelle (X: linker Operand, Y: rechter Operand, Z: Resultat):

$s_{0\ldots4}$	ALU-Operation
0 0 1 1 0	$Z = X + u_0$
1 0 0 1 0	$Z = X + Y + u_0$
1 0 1 0 1	$Z = Y$
1 1 0 0 0	$Z = X - 1 + u_0$
1 1 1 1 0	$Z = X \cdot 2 + u_0$

5.1. **(1 Punkte)** Vervollständigen Sie den Zustandsgraphen für den "`DBNX`"-Befehl mit minimaler Anzahl von Zuständen (unter Ausnutzung paralleler ausführbarer Operationen).



Lösung:

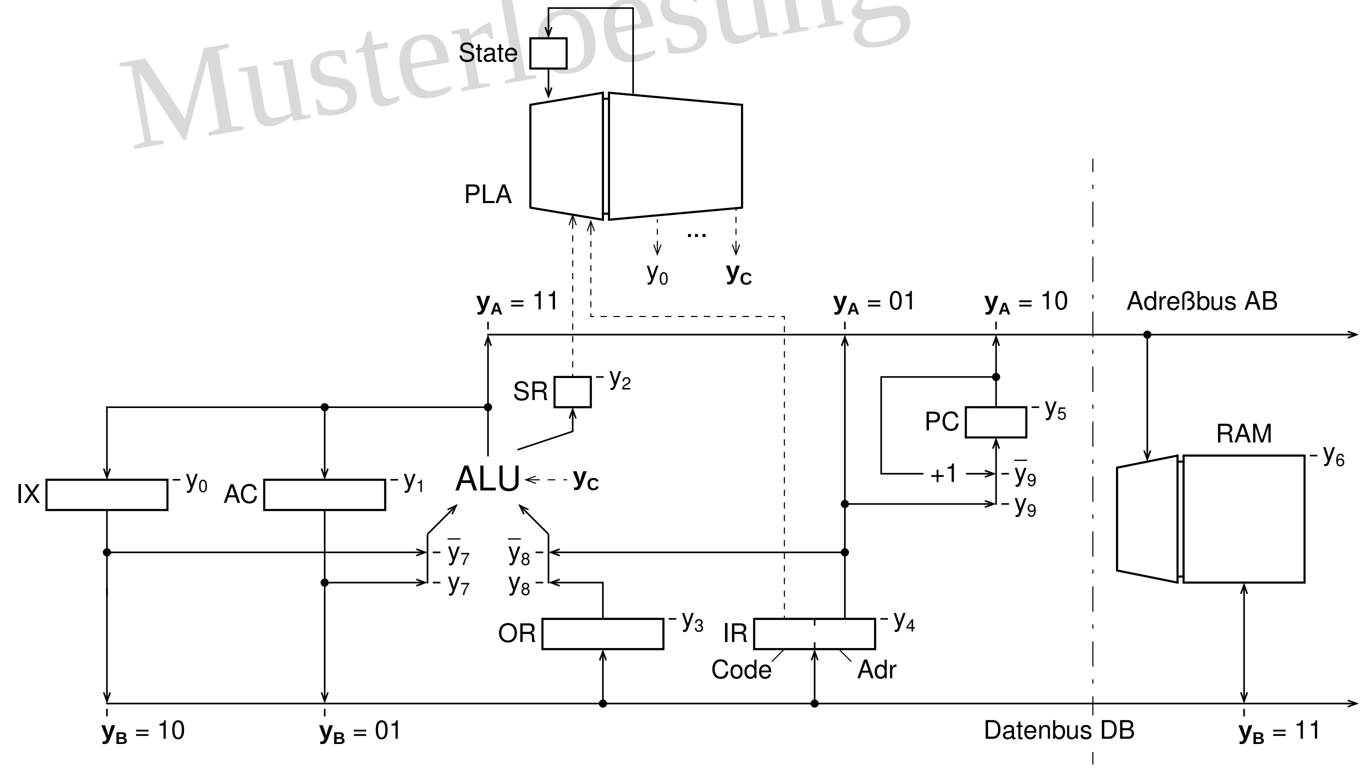


5.2. (3 Punkte) Füllen Sie die PLA-Tabelle gemäß des Zustandsgraphen der vorangehenden Teilaufgabe aus. Irrelevante Tabellenplätze dürfen freigelassen werden.

	SR				IR		Register/Speicher							Multiplexer					ALU
Z_i	z	n	c	v	Code	Z_j	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_A	y_B	$y_C = s_{0\ldots4}, u_0$
0	-	-	-	-	-	1	0	0	0		1	1	0			0	10	11	

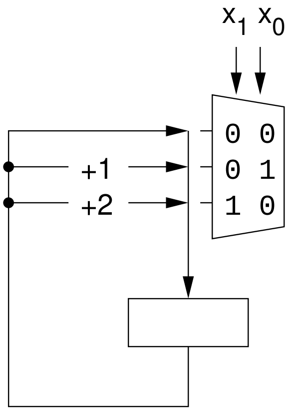
Lösung:

	SR				*IR*		*Register/Speicher*							*Multiplexer*					*ALU*
Z_i	*z*	*n*	*c*	*v*	*Code*	Z_j	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_A	y_B	$y_C = s_{0\ldots4}, u_0$
0	-	-	-	-	-	*1*	*0*	*0*	*0*		*1*	*1*	*0*			*0*	*10*	*11*	
1	-	-	-	-	*0x1F*	*n*	*1*	*0*	*1*			*0*	*0*	*0*					*110000*
n	*0*	-	-	-	-	*0*	*0*	*0*	*0*			*1*	*0*			*1*			
n	*1*	-	-	-	-	*0*	*0*	*0*	*0*			*0*	*0*						

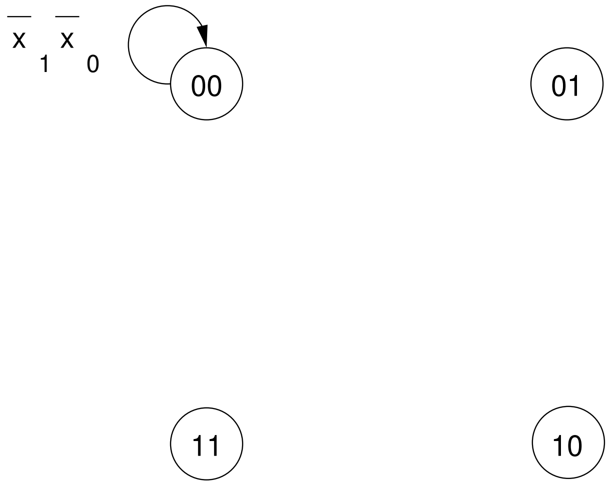

State
PLA
y_0
y_C
$y_A = 11$
$y_A = 01$
$y_A = 10$
Adreßbus AB
SR
y_2
PC
y_5
RAM
y_6
IX
y_0
AC
y_1
ALU
y_C
+1
$\bar{y}_9$
y_9
$\bar{y}_7$
y_7
$\bar{y}_8$
y_8
OR
y_3
IR
y_4
Code
Adr
$y_B = 10$
$y_B = 01$
Datenbus DB
$y_B = 11$

6. Aufgabe (8 Punkte): Schaltwerksentwurf

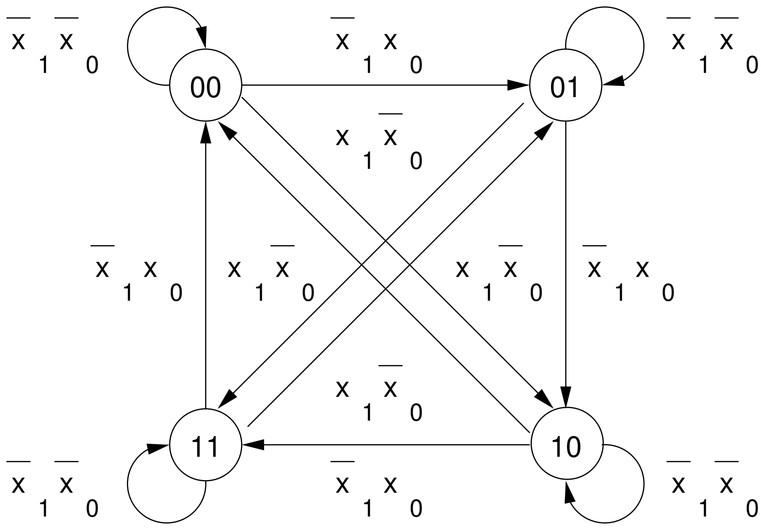
Es soll ein Modulo-4-Zähler mit zwei Flipflops u_1, u_0 entworfen werden, der in Abhängigkeit von zwei Eingangssignalen x_1, x_0 wie folgt zählt:



6.1. (2 Punkte) Vervollständigen Sie den Zustandsgraphen.



Lösung:



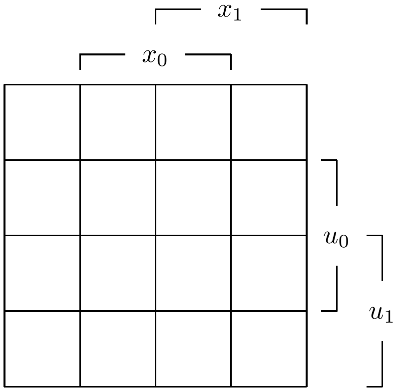
6.2. (3 Punkte) Der Zähler soll mit zwei unterschiedlichen Flipflop-Typen realisiert werden, einem JK-Flipflop u_1 mit den Eingängen j_1, k_1 und einem D-Flipflop u_0 mit dem Eingang d_0.

Ergänzen Sie die folgende Tabelle um die fehlenden Einträge.

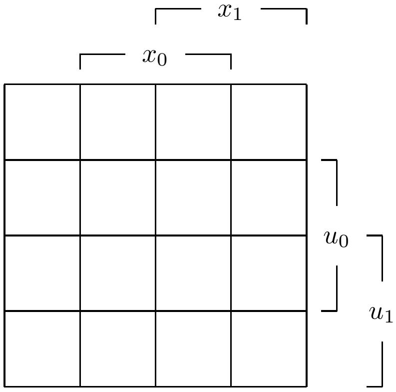
v $\rightarrow$ v$^+$	j	k	d
$0 \rightarrow 0$	0	*	0
$0 \rightarrow 1$	1	*	1
$1 \rightarrow 0$	*	1	0
$1 \rightarrow 1$	*	0	1

u_1	u_0	x_1	x_0	u_1^+	u_0^+	j_1	k_1	d_0
0	0	0	0					
0	0	0	1					
0	0	1	0					
0	0	1	1	*	*	*	*	*
0	1	0	0					
0	1	0	1					
0	1	1	0					
0	1	1	1	*	*	*	*	*
1	0	0	0					
1	0	0	1					
1	0	1	0					
1	0	1	1	*	*	*	*	*
1	1	0	0					
1	1	0	1					
1	1	1	0					
1	1	1	1	*	*	*	*	*

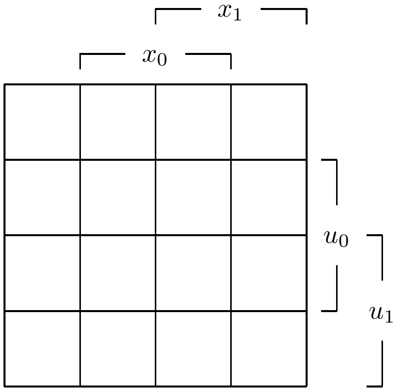
j_1:



k_1:



d_0:

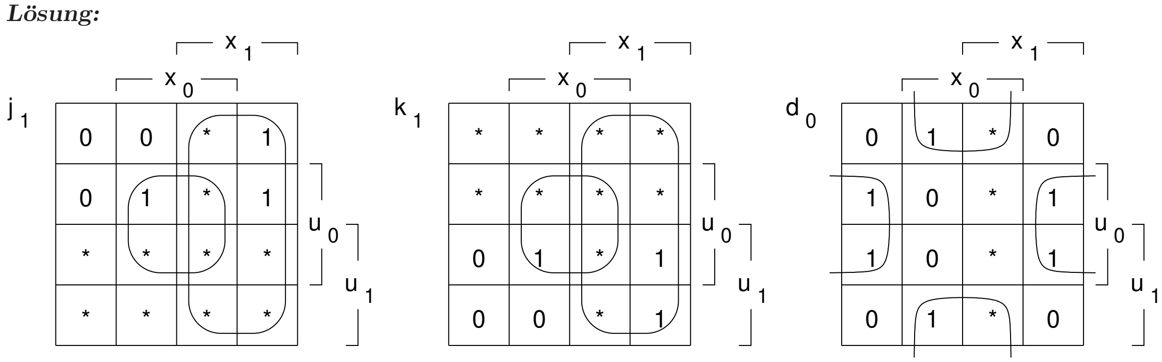


Lösung:

u_1	u_0	x_1	x_0	u_1^+	u_0^+	j_1	k_1	d_0
0	0	0	0	0	0	0	*	0
0	0	0	1	0	1	0	*	1
0	0	1	0	1	0	1	*	0
0	0	1	1	*	*	*	*	*
0	1	0	0	0	1	0	*	1
0	1	0	1	1	0	1	*	0
0	1	1	0	1	1	1	*	1
0	1	1	1	*	*	*	*	*
1	0	0	0	1	0	*	0	0
1	0	0	1	1	1	*	0	1
1	0	1	0	0	0	*	1	0
1	0	1	1	*	*	*	*	*
1	1	0	0	1	1	*	0	1
1	1	0	1	0	0	*	1	0
1	1	1	0	0	1	*	1	1
1	1	1	1	*	*	*	*	*

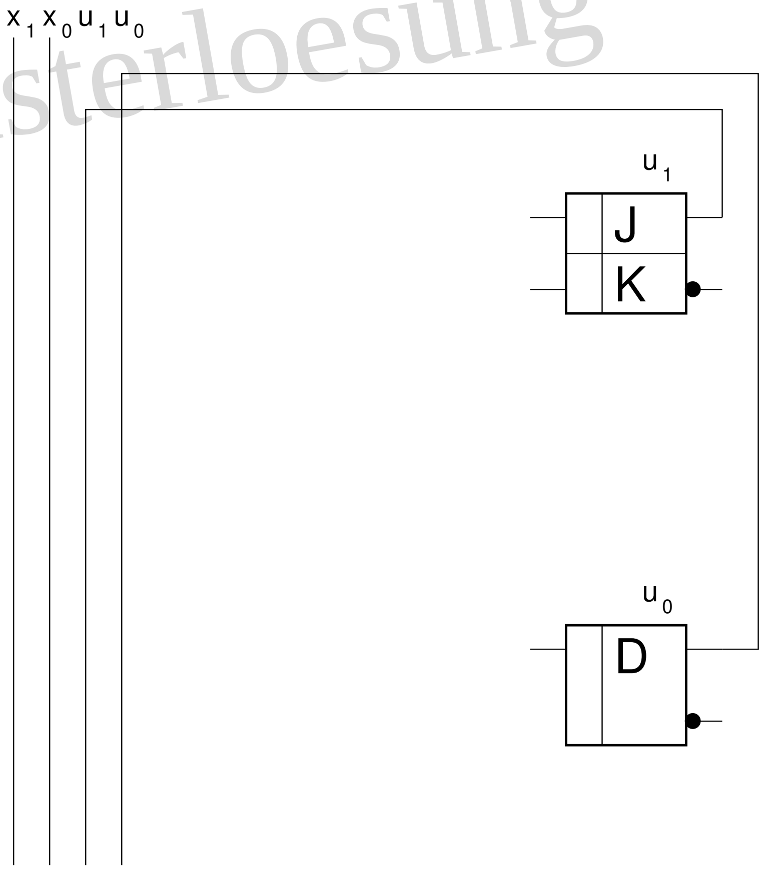
6.3. (2 Punkte) Ermitteln Sie die minimalen Ansteuergleichungen der beiden Flipflops mit Hilfe der obigen KV-Tafeln.

Lösung:



$$j_1 = x_0 \cdot u_0 + x_1$$
$$k_1 = x_0 \cdot u_0 + x_1$$
$$d_0 = ø x_0 \cdot u_0 + x_0 \cdot ø u_0$$

6.4. (1 Punkte) Vervollständigen Sie das Blockbild zur Realisierung des Schaltwerks.



Lösung: