Musterloesung

Fakultät IV

Klausur Informatik 2
26. Februar 2002

Biedl / Flik
Schöner / 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	9	
4	6	
5	6	
6	5	

1. Aufgabe (15 Punkte): Algorithmen und Datenstrukturen

Musterloesung

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.

```
1 public class IntList {
2     protected class IntListElem {
3         IntListElem next;
4         int data;
5     }
6     IntListElem first ;
7 }
```

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:

```
1   // Implementation von InsertionSort auf IntList. Iterative Variante 1:
2   // Es wird parallel eine zweite (sortierte) Liste, die als Ergebnis
3   // zurückgegeben wird.
4   public static void insertionSort( IntList l ) {
5       if (l == null) return;
6       // curr enthält neu einzusortierendes Element
7       // sorted enthält die bereits sortierte Teilliste
8       IntListElem curr = null, sorted = null;
9       // Entnehme ein Element nach dem nächsten der Liste und füge es in
10      // die sortierte Teilliste ein.
11      while (l. first != null) {
12          curr = l. first ;
13          l. first = l. first .next;
14          sorted = insertSorted(curr, sorted);
15      }
16      // Nachdem die ursprüngliche Liste leer ist, enthält die sortierte
17      // Teilliste alle Elemente in sortierter Reihenfolge, diese wird
18      // zurückgeliefert.
19      l. first = sorted;
20  }
21
22  // Hilfsfunktion: Füge Element in sortierte Teilliste ein.
23  protected static IntListElem insertSorted(IntListElem el, IntListElem l) {
24      if (el == null)
25          return l;
26      // Spezialfall: Leere Liste oder einfügen ganz vorne
27      if ((l == null ) || ( l.data > el.data)) {
28          el.next = l;
29          return (el);
30      }
31      // Einfügen in Mitte oder am Ende: Erst Suchen des Platzes
32      IntListElem iterator = l;
33      while (( iterator .next != null) && ( iterator .next.data <= el.data)) {
34          iterator .next = l;
35      }
36      // Einfügen bzw. Anhängen von el
37      el.next = iterator .next;
38      iterator .next = el;
39      // Zurückliefern der neuen Liste
40      return (l);
41  }
```

Musterlösung

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/Bibliothekersteller 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

Musterloesung

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, ...).

```
1  static public int [] countingSort(int [] arr , int maxvalue) {
2      int [] counts = new int[maxvalue];
3      int [] res = new int[arr.length];
4      for (int i=0; i<maxvalue; ++i) {
5          counts[i ] = 0;
6      }
7      for (int i=0; i<arr.length; ++i ) {
8          ++counts[arr[i ]];
9      }
10     for (int i=1; i<maxvalue; ++i) {
11         counts[i ] += counts[i-1];
12     }
13     for (int i=arr.length-1; i>=0; --i) {
14         res[--counts[arr[i ]]] = arr [i ];
15     }
16     return (res);
17 }
```

Lösung:

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

Musterloesung

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

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

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

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

betrachtet in Abhängigkeit von n, bzw.

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

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

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

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

3. Aufgabe (9 Punkte): Programmverifikation

Musterloesung

Gegeben sei folgende Methode zur Berechnung der Fakultät einer natürlichen Zahl sowie ein (fragwürdiger) Korrektheitsbeweis.

```
1  public static int factorial (int n) {
2      // n = n
3      // n = ∏_{i=n}^{n} i
4      int k = n;
5      // n = ∏_{i=k}^{n} i
6      int f = n;
7      // I : f = ∏_{i=k}^{n} i
8      while (k > 1) {
9          // (B : k > 1) ∧ (I : f = ∏_{i=k}^{n} i)
10         // f = ∏_{i=(k−1)+1}^{n} i
11         k = k − 1;
12         // f = ∏_{i=k+1}^{n} i
13         // f · k = k · ∏_{i=k+1}^{n} i
14         // f = f ∧ f · k = ∏_{i=k}^{n} i
15         f = f * k;
16         // f = f · k ∧ f · k = ∏_{i=k}^{n} i
17         // I : f = ∏_{i=k}^{n} i
18     }
19     // (¬B : k ≤ 1) ∧ (I : f = ∏_{i=k}^{n} i)
20     // POST: f = ∏_{i=1}^{n} i
21     return f;
22 }
```

3.1. Verletzung der Spezifikation (2 Punkte) Für welche Werte von n ist factorial(n) $\neq n!$, d.h. für welche Eingangswerte verstößt factorial gegen seine Spezifikation?

Lösung:

factorial(0) $= 0 \neq 1 = 0!$. Außerdem ist factorial(n) $= n, \forall\ n < 0$, wofür die Fakultät eigentlich nicht definiert ist. Ein Korrektheitsbeweis müsste also $[-\infty, 0]$ aus dem Definitionsbereich ausschließen.

3.2. Beweisfehler (4 Punkte) Inwiefern ist der Beweis fehlerhaft oder unvollständig? Welche Zeilen folgen nicht aus der vorhergehenden (Hinweis: 2 Zeilen sind fehlerhaft)?

Korrigieren bzw. ergänzen Sie den Beweis (nicht das Programm), sodass er in sich schlüssig und die Zusicherung der Nachbedingung korrekt wird. Wie lautet die dafür notwendige Vorbedingung? Benutzen Sie für Ihre Korrekturen den Vordruck auf der folgenden Seite.

Lösung:

Die POST-Bedingung folgt nicht aus der negierten Schleifenbedingung und der (hier angegebenen) Invariante. Es wird bei diesem Übergang die Annahme $k = 1$ gemacht, aus $\neg B \wedge I$ folgt aber nur $k \leq 1$. Als Konsequenz muss der Term $k \geq 1$ mit in die Invariante aufgenommen werden (die Bedingung kann nicht geändert werden, da eine Änderung des Codes in der Aufgabenstellung verboten wurde, und die Bedingung ergibt sich direkt aus dem Code). Um diesen Term mit in die Invariante aufzunehmen, muss $k \geq 1$ aber auch schon vor der Schleife gelten; da diese Zusicherung nirgends direkt hergestellt wird, sondern durch die Zuweisung in Zeile 4 lediglich k auf den Wert n gesetzt wird, muss $n \geq 1$ in die PRE-Bedingung aufgenommen werden.

Weiterhin kann die Zeile 16, außer für den im allgemeinen nicht zutreffenden Spezialfall $k = 1$, nicht erfüllt sein, da $f = f \cdot k$ entweder $f = 0$ oder $k = 1$ impliziert, was beides an dieser Stelle nicht zugesichert werden kann. Durch einfaches Streichen dieser Zeile und des Terms $f = f$ in Zeile 14 wird dieser Teil des Beweises korrekt (der Rest der Zeile 14 ergibt sich, wenn das Zuweisungsaxiom auf die Zeilen 15 und 17 angewendet wird).

Insgesamt ergibt sich damit als „korrekter" Korrektheitsbeweis:

Musterloesung

```
1  public static int factorial (int n) {
2      // n ≥ 1
3      // n = ∏_{i=n}^{n} i ∧ (n ≥ 1)
4      int k = n;
5      // n = ∏_{i=k}^{n} i ∧ (k ≥ 1)
6      int f = n;
7      // I : (f = ∏_{i=k}^{n} i ∧ (k ≥ 1))
8      while (k > 1) {
9          // (B : k > 1) ∧ (I : f = ∏_{i=k}^{n} i ∧ (k ≥ 1))
10         // f = ∏_{i=(k−1)+1}^{n} i ∧ (k − 1 ≥ 1)
11         k = k − 1;
12         // f = ∏_{i=k+1}^{n} i ∧ (k ≥ 1)
13         // f · k = k · ∏_{i=k+1}^{n} i ∧ (k ≥ 1)
14         // f · k = ∏_{i=k}^{n} i ∧ (k ≥ 1)
15         f = f * k;
16         // I : (f = ∏_{i=k}^{n} i ∧ (k ≥ 1))
17     }
18     // (¬B : k ≤ 1) ∧ (I : f = ∏_{i=k}^{n} i ∧ (k ≥ 1))
19     // POST: f = ∏_{i=1}^{n} i
20     return f;
21 }
```

3.3. Zuweisungsaxiom (3 Punkte) Benutzen Sie die Regeln des Hoare-Kalküls, um die für die Erfüllung der Nachbedingung nötige Vorbedingung herzuleiten. Halten Sie sich genau an die Regeln und Axiome des Hoare-Kalküls. Schreiben Sie nur eine Zusicherung pro Zeile, und geben Sie die zur Herleitung dieser Zeile benutzte Regel an. Verwenden Sie den Vordruck auf der folgenden Seite.

```
1 j = j + 1;
2 i = j;
3 // POST: i = 2
```

Lösung:

```
1  // PRE: j = 1  [Verstärkungs-/Abschwächungsregel]
2  // j + 1 = 2
3  j = j + 1;      // [Zuweisungsaxiom]
4  // j = 2
5  i = j;          // [Zuweisungsaxiom]
6  // POST: i = 2
```

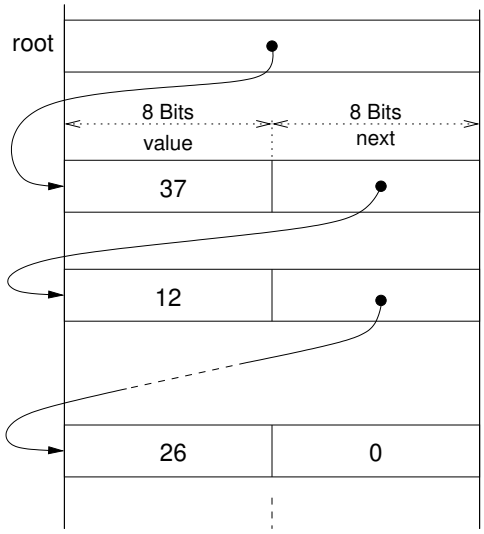
Außerdem wurde die ***Regel der sequentiellen Komposition*** *angewendet, um die (übereinstimmende) Post-Bedingung der Zuweisung in Zeile 3 an die PRE-Bedingung der Zuweisung in Zeile 5 aneinanderzufügen, und so die Bedingungen der Zeilen 1 und 6 zu den Vor- bzw. Nachbedingungen des gesamten Codeblocks zu machen.*

4. Aufgabe (6 Punkte): Assemblerprogrammierung

Musterloesung

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

4.1. **(4 Punkte)** Das folgende Speicherabbild zeigt die Realisierung einer einfach verketteten Liste. Jedes Listenglied entspricht einem 16-Bit-Speicherwort und besteht aus einem Zahlenwert *value* und einem Zeiger *next* mit der Adresse des nächsten Listengliedes. Das letzte Glied der Liste wird durch $next = 0$ gekennzeichnet.
Die Speicherzelle mit der symbolischen Adresse *root* enthält den Zeiger auf das erste Listenglied. Mit dem Zeigerwert 0 wird eine leere Liste angezeigt.



Schreiben Sie ein VIP-Assemblerprogramm, das die Listenlänge, d.h. die Anzahl der Listenglieder ermittelt. Verwenden Sie das Indexregister IX zur Zählung der Listenglieder.

Hinweis: Eine Java-Implementierung könnte so aussehen:

```
class List {
  int value;
  List next;
}

int length ( List root) {
  int IX = 0;
  List AC = root;

  while (AC != null) {
    AC = AC.next;
    IX++;
  }
  return IX;
}
```

Lösung:

```
        ORG  0
root    DAT  100
ptr     RES  1
length  RES  1
mask    DAT  0xff

start   LDX  #0
        LDA  root
```

Musterloesung

```
loop    AND  mask
        BEQ  stop
        STA  ptr
        LDA  @ptr
        ADDX #1
        JMP  loop
        STX  length
stop    HLT
        END  start
```

4.2. **(2 Punkte)** Gegeben ist das folgende VIP-Assemblerprogramm. Beschreiben Sie in Anlehnung an die Befehlstabelle die Wirkung eines jeden Maschinenbefehls mit Registerzuweisungen. Ersetzen Sie dabei "Op" entsprechend der Adressierungsart, und geben Sie wann immer möglich konkrete Zahlen an.

```
        ORG  100

M       RES  1

N       EQU  2

P       DAT  100

Q       DAT  4

Start   LDX  #N

        LDA  M[IX]

        SUB  Q

        STA  @P

        HLT

        END  Start
```

Lösung:

```
        ORG  100
M       RES  1
N       EQU  2
P       DAT  100
Q       DAT  4
Start   LDX  #N       ; IX := 2
        LDA  M[IX]    ; AC := RAM[100+2]
        SUB  Q        ; AC := AC - RAM[102]
        STA  @P       ; RAM[100] := AC
        HLT
        END  Start
```

Musterloesung

Befehle ohne Operanden:

Operation	Code (hex)	SR-Inhalt z	n	c	v	Adressierung	#	@	[IX]	Wirkung
HLT	00	-	-	-	-	-	-	-	-	Anhalten
NOP	01	-	-	-	-	-	-	-	-	Nichts
NOT	02	x	x	-	-	-	-	-	-	AC := ¬ AC, 1-Komplement
NEG	03	x	x	x	x	-	-	-	-	AC := 0 - AC, 2-Komplement
ASL	04	x	x	x	x	-	-	-	-	AC := AC × 2, c := Bit 15
ASR	05	x	x	x	-	-	-	-	-	AC := AC × 2^{-1}, c := Bit 0
LSL	06	x	x	x	-	-	-	-	-	AC := AC × 2, c := Bit 15
LSR	07	x	x	x	-	-	-	-	-	AC := AC × 2^{-1}, c := Bit 0, Bit 15 := 0

Befehle mit Operanden:

Operation	Code (hex)	SR-Inhalt z	n	c	v	Adressierung	#	@	[IX]	Wirkung
JMP	10	-	-	-	-	✓	-	-	-	PC := Op
BPL	11	-	-	-	-	✓	-	-	-	PC := Op, falls positiv
BMI	12	-	-	-	-	✓	-	-	-	PC := Op, falls negativ
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 nicht Null (≠)
BGT	19	-	-	-	-	✓	-	-	-	PC := Op, falls >
BGE	1A	-	-	-	-	✓	-	-	-	PC := Op, falls ≥
BLE	1B	-	-	-	-	✓	-	-	-	PC := Op, falls ≤
BLT	1C	-	-	-	-	✓	-	-	-	PC := Op, falls <
BGTU	1D	-	-	-	-	✓	-	-	-	PC := Op, falls > (vorzeichenlos)
BGEU	13	-	-	-	-	✓	-	-	-	PC := Op, falls ≥ (vorzeichenlos)
BLEU	1E	-	-	-	-	✓	-	-	-	PC := Op, falls ≤ (vorzeichenlos)
BLTU	14	-	-	-	-	✓	-	-	-	PC := Op, falls < (vorzeichenlos)
LDA	20 + (0..3)	x	x	0	0	✓	✓	✓	✓	AC := Op
LDX	28 + (0..3)	x	x	0	0	✓	✓	✓	✓	IX := Op
STA	30 + (0..3)	-	-	-	-	✓	-	✓	✓	Op := AC
STX	38 + (0..3)	-	-	-	-	✓	-	✓	✓	Op := IX
ADD	40 + (0..3)	x	x	x	x	✓	✓	✓	✓	AC := AC + Op
ADDX	48 + (0..3)	x	x	x	x	✓	✓	✓	✓	IX := IX + Op
SUB	50 + (0..3)	x	x	x	x	✓	✓	✓	✓	AC := AC - Op
SUBX	58 + (0..3)	x	x	x	x	✓	✓	✓	✓	IX := IX - Op
CMP	60 + (0..3)	x	x	x	x	✓	✓	✓	✓	AC - Op
CMPX	68 + (0..3)	x	x	x	x	✓	✓	✓	✓	IX - Op
AND	70 + (0..3)	x	x	-	-	✓	✓	✓	✓	AC := AC ∧ Op (bitpaarweise)
OR	80 + (0..3)	x	x	-	-	✓	✓	✓	✓	AC := AC ∨ Op (bitpaarweise)
XOR	90 + (0..3)	x	x	-	-	✓	✓	✓	✓	AC := AC ⊕ Op (bitpaarweise)

Tabelle 1: VIP-Befehlssatzreferenz

5. Aufgabe (6 Punkte): Erweiterung des VIP

Musterloesung

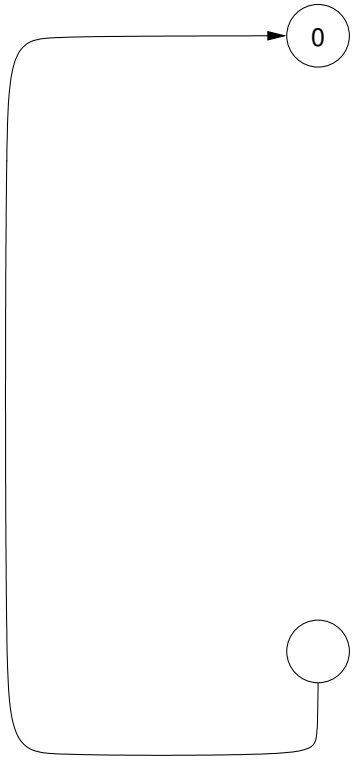
Der VIP soll um einen unbedingten Sprungbefehl mit indirekter Adressierung des Sprungziels erweitert werden, d.h. um einen Befehl, der den Inhalt einer Speicherzelle M in den PC lädt und das Statusregister unbeeinflußt läßt: `JMP @M`. Als Befehlcodierung wird 0x1F verwendet.

5.1. (2 Punkte) Beschreiben Sie den Befehlszyklus mit Registerzuweisungen.

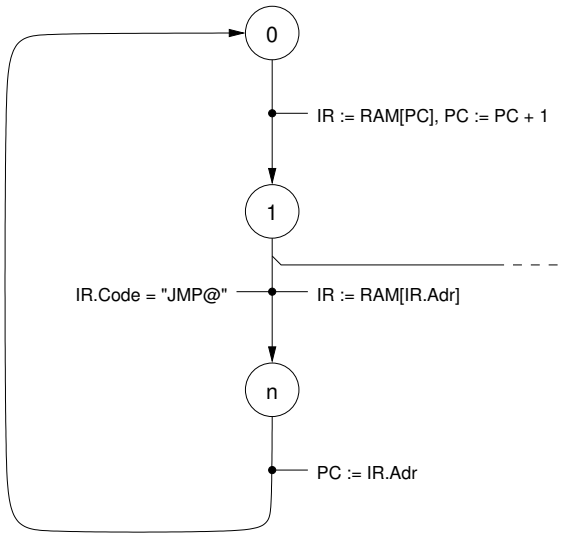
Lösung:

```
IR := RAM[PC], PC := PC+1
IR := RAM[IR.Adr]
PC := IR.Adr
```

5.2. (1 Punkte) Vervollständigen Sie den Zustandsgraphen für den "`JMP @M`"-Befehl mit minimaler Anzahl von Zuständen (unter Ausnutzung parallel ausführbarer Operationen).



Lösung:



Musterlösung

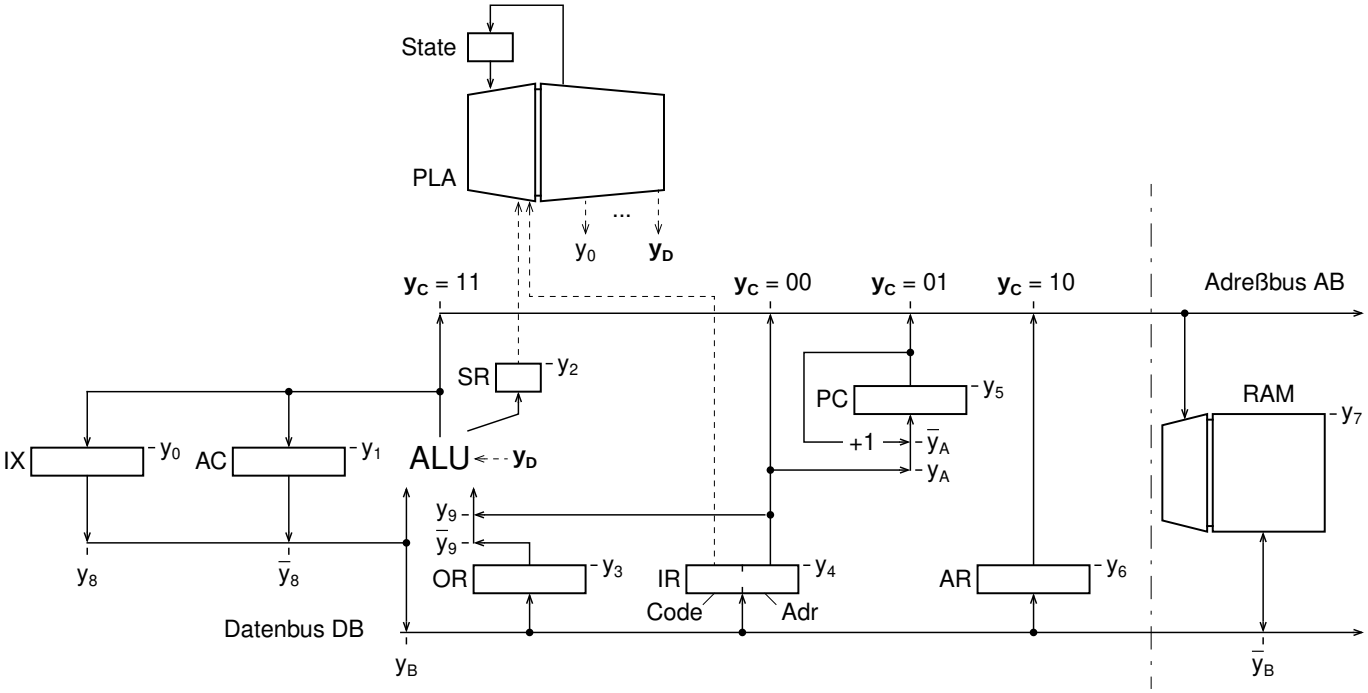
5.3. **(3 Punkte)** Füllen Sie die PLA-Tabelle gemäß des Zustandsgraphen der vorangehenden Teilaufgabe aus. Irrelevante Werte können freigelassen werden.

	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	y_D
0	-	-	-	-	-	1														
1	-	-	-	-	0x1F															

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	y_D
0	-	-	-	-	-	*1*	*0*	*0*	*0*		*1*	*1*		*0*			*0*	*0*	*01*	
1	-	-	-	-	*0x1F*	*n*	*0*	*0*	*0*		*1*	*0*		*0*				*0*	*00*	
n	-	-	-	-	-	*0*	*0*	*0*	*0*			*1*		*0*			*1*			

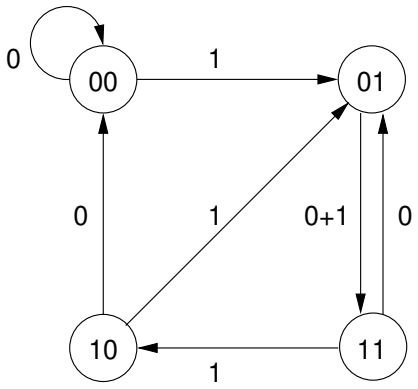
Musterloesung



6. Aufgabe (5 Punkte): Schaltwerksentwurf

Musterloesung

Gegeben ist der abgebildete Zustandsgraph eines Schaltwerks mit den beiden Zustandsvariablen u_1, u_0 und dem Eingangssignal e.



6.1. (2 Punkte) Ergänzen Sie die Tabelle der Übergangs- und Ansteuerfunktionen zur Realisierung mit einem SR-Flipflop für u_1 und einem JK-Flipflop für u_0.

v → v$^+$	s	r	j	k
0 → 0	0	*	0	*
0 → 1	1	0	1	*
1 → 0	0	1	*	1
1 → 1	*	0	*	0

u_1	u_0	e	u_1^+	u_0^+	s_1	r_1	j_0	k_0
0	0	0						
0	0	1						
0	1	0						
0	1	1						
1	0	0						
1	0	1						
1	1	0						
1	1	1						

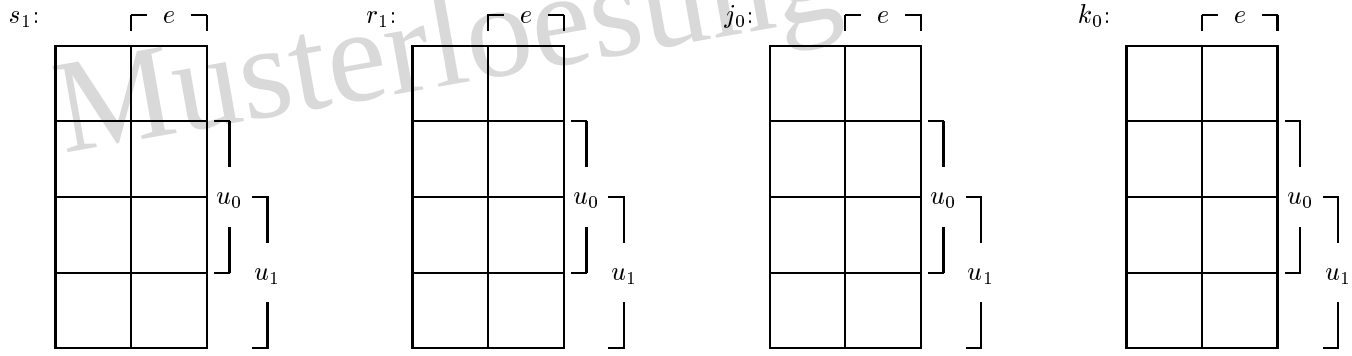
Lösung:

u_1	u_0	e	u_1^+	u_0^+	s_1	r_1	j_0	k_0
0	*0*	*0*	*0*	*0*	*0*	*	*0*	*
0	*0*	*1*	*0*	*1*	*0*	*	*1*	*
0	*1*	*0*	*1*	*1*	*1*	*0*	*	*0*
0	*1*	*1*	*1*	*1*	*1*	*0*	*	*0*
1	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*
1	*0*	*1*	*0*	*1*	*0*	*1*	*1*	*
1	*1*	*0*	*0*	*1*	*0*	*1*	*	*0*
1	*1*	*1*	*1*	*0*	*	*0*	*	*1*

6.2. (2 Punkte) Ermitteln Sie die minimalen Ansteuergleichungen der beiden Flipflops mit Hilfe der KV-Tafeln.

Musterloesung

s_1: r_1: j_0: k_0:



Lösung:

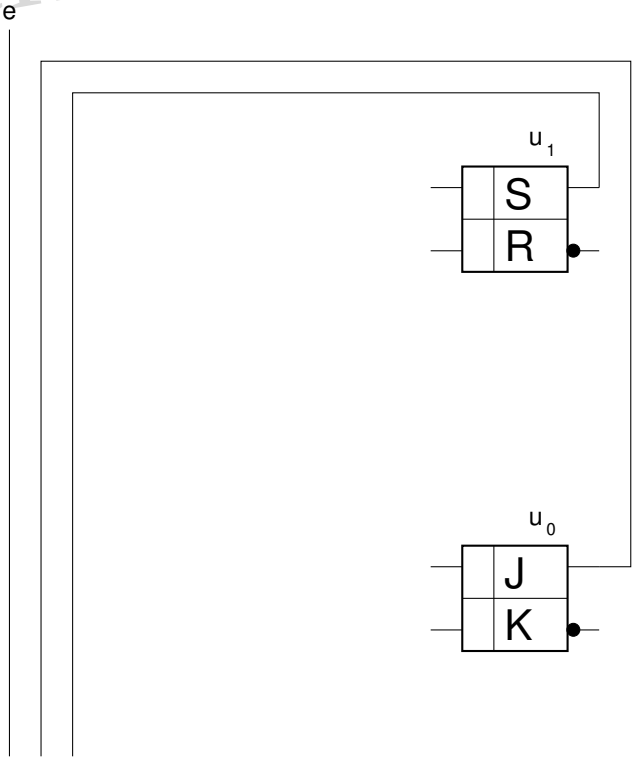
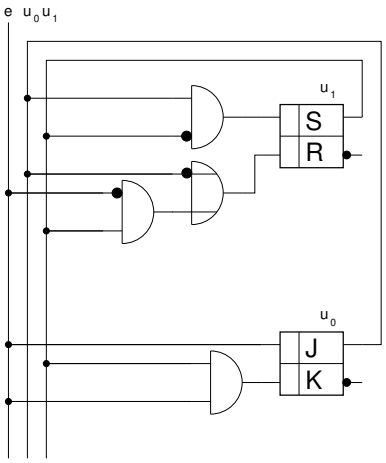
s1: r1: j0: k0:

s1: e		r1: e		j0: e		k0: e	
0	0	*	*	0	1	*	*
1	1	0	0	*	*	0	0
0	*	1	0	*	*	0	1
0	0	1	1	0	1	*	*
u0	u1	u0	u1	u0	u1	u0	u1

$$
\begin{aligned}
s_1 &= u_0 \cdot \overline{u_1} \\
r_1 &= \overline{u_0} + u_1 \cdot \overline{e} \\
j_0 &= e \\
k_0 &= u_1 \cdot e
\end{aligned}
$$

Musterloesung

6.3. **(1 Punkte)** Vervollständigen Sie das Blockbild.



Lösung: