Klausur Informatik 2
22. März 2000

Biedl / Lemke
Schöner / Tschirley

Name: ....

Vorname: ....

Matr.-Nr.: ....

Bearbeitungszeit: 120 Minuten

Wichtige Hinweise:

- 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 Lösung auch auf die Rückseiten der Blätter! Weiteres Papier kann bei den Tutoren angefordert werden.
- Tragen Sie jetzt (vor Beginn der eigentlichen Bearbeitungszeit!) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen können nicht gewertet werden.
- Schreiben Sie **nicht** mit Bleistift!
- Schreiben Sie nur in **blau** oder **schwarz**!

Viel Erfolg!

Aufgabe	Punkte	erreicht
1	10	
2	10	
3	13	
4	4	
5	13	

Aufgabe 1 (10 Punkte): Hoare-Kalkül

1. (4 Punkte) Für zwei Prozeduren sind im folgenden Aufgabenbeschreibungen und Deklarationen gegeben. Formalisieren Sie diese Beschreibungen zu Post-Conditions (in Form von logisch-mathematischen Aussagen über die Eingabe-/Ausgabevariablen) der entsprechenden Prozedur, die für einen Korrektheitsbeweis mit dem Hoare-Kalkül verwendet werden könnte.

1. **PROCEDURE** Sort(**VAR** arr: **ARRAY OF CARDINAL**);
   Sort sortiert das Array arr (die Indizes reichen von 0 bis **HIGH**(arr)) aufsteigend. Das Array arr muss dazu modifiziert werden. Sie können die freie Variable ARR benutzen, um in der POST-Condition auf den ursprünglichen Zustand des Arrays zuzugreifen (PRE-Condition $arr = ARR$).
2. **PROCEDURE** CardinalSquareRoot(n: **CARDINAL**; **VAR** res: **CARDINAL**);
   CardinalSquareRoot gibt in res den ganzzahligen Anteil der positiven Wurzel der natürlichen Zahl n zurück. **Hinweis:** Eine Post-Condition wie $\lfloor\sqrt{n}\rfloor$ ($\sqrt{n}$ abgerundet) ist keine gültige Lösung, da es bei der Umwandlung von Variablen des Datentyps REAL nach CARDINAL zu Rundungsfehlern kommen kann, wodurch die Verwendung des Datentyps REAL ausscheidet.

2. (4 Punkte) Beweisen Sie mit Hilfe des Hoare-Kalküls, dass das folgende Codefragment die angegebene Spezifikation erfüllt, d.h. dass nach der Ausführung der Prozedur die POST-Condition gilt, sofern vor ihrer Ausführung die angegebene PRE-Condition galt ($\prod_{k=1}^{N} k$ ist für $N = 0$ per Definition gleich 1, ansonsten ist es das Produkt der Zahlen k von $k = 1$ bis $k = N$). Neben dem Beweis der partiellen Korrektheit sollen Sie auch einen Terminationsbeweis angeben, um die *totale* Korrektheit der Prozedur zu zeigen. Geben Sie dabei jeweils an, welche Regeln des Hoare-Kalküls Sie verwendet haben (verwenden Sie den Vordruck auf der nächsten Seite zum Eintragen der Zusicherungen):

```
PROCEDURE Fac(n: CARDINAL; VAR res: CARDINAL);
VAR i: CARDINAL;
BEGIN
  (* PRE: n ≥ 0 *)
  i   := 0;
  res := 1;
  WHILE (i < n) DO
    i   := i+1;
    res := i*res;
  END;
  (* POST: res = ∏_{k=1}^{n} k *)
END;
```

3. (2 Punkte) Ausgehend von dem folgenden Codefragment:

```
(* P *)
WHILE (B) DO
  (* Anweisungsfolge S *)
END;
(* Q *)
```

Welche Beziehungen bestehen zwischen

- der Zusicherung P und der Schleifeninvariante I?
- der Bedingung B der **WHILE**-Schleife, der Invariante I und der Zusicherung Q?

Bei der Beantwortung dieser Frage könnte es hilfreich sein, sich zu überlegen, an welchen Stellen die Invariante I eigentlich gelten muss.

```
PROCEDURE Fac(n: CARDINAL; VAR res: CARDINAL);
VAR i: CARDINAL;
BEGIN
  (* PRE: n >= 0 *)

  i   := 0;

  res := 1;

  WHILE (i < n) DO

    i   := i+1;

    res := i*res;

  END;

  (* POST: res = PROD_{k=1}^{n} k *)
END;
```

Lösung:

1. (a) **PROCEDURE** Sort(**VAR** arr: **ARRAY OF CARDINAL**);
   Sort *sortiert das Array* arr *(die Indizes reichen von* 0 *bis* **HIGH**(arr)*) aufsteigend.*

   (∗ *PRE:* $arr = ARR$ ∗)
   (∗ *POST:* $\forall i, j \in [0..HIGH(arr)] : j > i \Rightarrow arr_j \geq arr_i$
   $\wedge\, arr \in Permutations(ARR)$ ∗)

   (b) **PROCEDURE** CardinalSquareRoot(n: **CARDINAL**; **VAR** res: **CARDINAL**);
   CardinalSquareRoot *gibt in* res *den ganzzahligen Anteil der positiven Wurzel der natürlichen Zahl* n *zurück.*

   (∗ *POST:* $(res \geq 0) \wedge \big(res^2 \leq n < (res+1)^2\big) \wedge res \in \mathbb{N}$ ∗)

2. *Der Teil der folgenden Aussagen, die sich auf den Terminationsbeweis bezieht, ist fett gedruckt.*

```
PROCEDURE Fac(n: CARDINAL; VAR res: CARDINAL);
VAR i: CARDINAL;
BEGIN
  (* PRE: $n \geq 0$ *)
  (* Logik, Mathematik: $n \geq 0 \wedge 0 = 0$ *)
  i   := 0;
  (* Zuweisungsaxiom:    $n \geq 0 \wedge i = 0$ *)
  (* Logik, Mathematik: $n \geq 0 \wedge i = 0 \wedge 1 = 1$ *)
  res := 1;
  (* Zuweisungsaxiom:    $n \geq 0 \wedge i = 0 \wedge res = 1$ *)
  (* Logik, Mathematik: $I : \big(res = \prod_{k=1}^{i} k \wedge i \leq n\big)$ *)
  WHILE (i < n) DO
    (* Schleife:           $I : \big(res = \prod_{k=1}^{i} k \wedge i \leq n\big) \wedge B : (i < n) \wedge \tau \overset{\text{def}}{=} \mathbf{n - i = x}$ *)
    (* Logik, Mathematik: $res = \prod_{k=1}^{i+1-1} k \wedge (i+1 < n+1) \wedge \mathbf{n - (i+1) = x - 1}$ *)
    i   := i+1;
    (* Zuweisungsaxiom:    $res = \prod_{k=1}^{i-1} k \wedge (i < n+1) \wedge \mathbf{n - i = x - 1}$ *)
    (* Logik, Mathematik: $i \cdot res = \prod_{k=1}^{i} k \wedge (i \leq n) \wedge \mathbf{n - i = x - 1}$ *)
    res := i*res;
    (* Zuweisungsaxiom:    $I : \big(res = \prod_{k=1}^{i} k \wedge i \leq n\big) \wedge \tau = \mathbf{n - i = x - 1 < x}$ *)
  END;
  (* Schleife:           $I : \big(res = \prod_{k=1}^{i} k \wedge i \leq n\big) \wedge \neg B : (\neg(i < n))$ *)
  (* Logik, Mathematik: $res = \prod_{k=1}^{i} k \wedge i = n$ *)
  (* POST: $res = \prod_{k=1}^{n} k$ *)
END;
```

Zur Terminierung:

- *Die Terminierungsfunktion, der mit jedem Schleifendurchlauf kleiner werdende INTEGER-Ausdruck:* $\tau \overset{\text{def}}{=} n - i$
- τ *wird in jedem Schleifendurchlauf echt kleiner, wie im Listing gezeigt wird.*
- *Weiterhin gilt (mit* $B : i < n$*):* $I \wedge (\tau \leq 0) \Rightarrow \neg B$, da $\tau \leq 0 \Rightarrow n \leq i \Rightarrow \neg B$
- *Da weiterhin der Schleifenrumpf terminiert, da er selbst keine Schleife enthält, terminiert auch die gesamte WHILE-Schleife, und somit auch die Prozedur Fac*

3. *Welche Beziehungen bestehen zwischen*
   - *der Zusicherung* P *und der Schleifeninvariante* I*?*
     Mögliche Antworten: $P \Rightarrow I$, aus P folgt I, P impliziert I, I enthält P
   - *der Bedingung* B *der* **WHILE**-*Schleife, der Invariante* I *und der Zusicherung* Q*?*
     Mögliche Antworten: $(I \wedge \neg B) \Rightarrow Q$, aus I und nicht B folgt Q, I und nicht B impliziert Q, Q enthält I und nicht B

Aufgabe 2 (10 Punkte): $\mathcal{O}$-Kalkül

1. (5 Punkte) Zeigen oder widerlegen Sie mit Hilfe der Definition des $\mathcal{O}$-Kalküls jeweils, ob $f(n) \in \mathcal{O}(g(n))$. Falls dies zutrifft, $g(n)$ also eine obere Schranke von $f(n)$ ist, bestimmen Sie zusätzlich, ob diese Schranke dicht ist. Achten Sie darauf, Ihren Ansatz und Beweisweg klar zu zeigen.

1. $f(n) = n^{3/2}$

   $g(n) = n \cdot \log(n)$

2. $f(n) = 3 \cdot n + \log(n^n)$

   $g(n) = n \cdot \log(n)$

Hinweis: Für hinreichend große n gilt: $\sqrt{n} \geq \log n$.

2. (5 Punkte) Bestimmen Sie eine Formel für den Aufwand des folgenden Code-Fragments. Dabei soll der Zeitaufwand jedes Operators (Zuweisung, Addition, Vergleich, etc.) mit der Einheit 1 abgeschätzt werden. Anschliessend geben Sie die dazugehörige Aufwandsklasse an. Dabei dürfen Sie entweder die Definition des $\mathcal{O}$-Kalküls oder die unten angegebenen Rechenregeln benutzen. Geben Sie den benutzten Rechenweg detailliert an.

```
DoTask1(x);
i := 0;
WHILE (i < x) DO
  j := i;
  WHILE (j = i) DO
    DoTask2(x);
    j := j+1;
  END;
  i := i+1;
END;
```

Der Aufwand der verwendeten Funktionen ist bekannt und im folgenden gegeben:

- DoTask1(n) $\in \mathcal{O}(\mathrm{n}^2)$
- DoTask2(n) $\in \mathcal{O}(\log(\mathrm{n}))$

Definition ($\mathcal{O}$-Kalkül):
Sind f und g zwei positiv-reellwertige Funktionen über $\mathbb{N}$, dann bedeutet $f \in \mathcal{O}(g)$, dass es eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}$ gibt, sodass für alle $n > n_0$ gilt: $f(n) \leq c * g(n)$. g ist dann eine obere Schranke für f.

Rechenregeln ($\mathcal{O}$-Kalkül):

(1)	$f \in \mathcal{O}(f)$	
(2)	$k * \mathcal{O}(f) = \mathcal{O}(f)$,	k konstant
(3)	$\mathcal{O}(\mathcal{O}(f)) = \mathcal{O}(f)$	
(4)	$\mathcal{O}(f) * \mathcal{O}(g) = \mathcal{O}(f * g)$	
(5)	$\mathcal{O}(f * g) = f * \mathcal{O}(g)$	
(6)	$\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g)$	
(7)	$f + \mathcal{O}(g) \subseteq \mathcal{O}(f + g)$	
(8)	$\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f)$,	falls $\mathcal{O}(g) \subseteq \mathcal{O}(f)$
(9)	$a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 \in \mathcal{O}(n^k)$,	$a_k > 0$

Lösung:

1. *Zeigen oder widerlegen Sie* $f(n) \in \mathcal{O}(g(n))$*:*
   (a) $f(n) = n^{3/2}$, $g(n) = n \cdot \log(n)$

$f(n) \notin \mathcal{O}(g(n))$, da :

$$n^{3/2} \leq c \cdot n \log n \quad ; n > n_0$$
$$\sqrt{n} \leq c \cdot \log n$$

nicht erfüllbar, da keine c, n_0 *existieren, die diese Bedingung erfüllen*

   (b) $f(n) = 3 \cdot n + \log(n^n)$, $g(n) = n \cdot \log(n)$

Schranke:

$$\begin{aligned} 3n + \log(n^n) &\leq c \cdot n \log n \quad ; n > n_0 \\ 3 + \log(n) &\leq c \cdot \log n \\ \frac{3}{\log(n)} + 1 &\leq c \end{aligned}$$

erfüllt für: $c = 6, n_0 = 1$

Dichte Schranke:

$$\begin{aligned} & f(n) \in \mathcal{O}(v(n)) \Rightarrow g(n) \in \mathcal{O}(v(n)) \\ & f(n) \in \mathcal{O}(v(n)) \\ & \Rightarrow 3n + \log(n^n) \leq c_f \cdot v(n) \quad ; n > n_f \\ & \quad \Rightarrow \log(n^n) \leq c_f \cdot v(n) \\ & \quad \Rightarrow \log(n^n) \leq c_g \cdot v(n) \quad ; n > n_g, c_g = c_f, n_g = n_f \\ & g(n) \in \mathcal{O}(v(n)) \end{aligned}$$

2. *Bestimmen Sie eine Formel für den Aufwand des folgenden Code-Fragments:*

```
1  DoTask1(x);
2  i := 0;
3  WHILE (i < x) DO
4     j := i;
5     WHILE (j = i) DO
6        DoTask2(x);
7        j := j+1;
8     END;
9     i := i+1;
10 END;
```

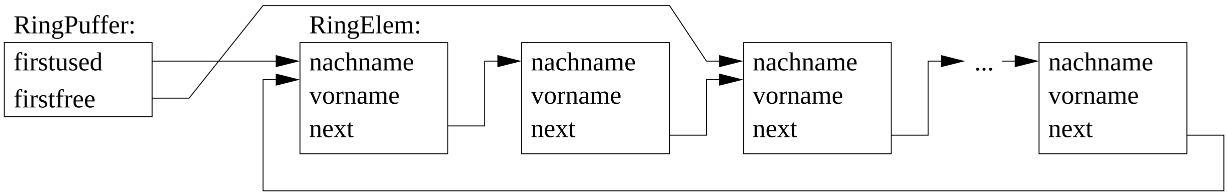
Der Zeilenbezug ist unter den geschweiften Klammern angegeben. Zeilen 3 und 5 haben einen zusätzlichen, fehlschlagenden Vergleich nach der letzten Iteration der entsprechenden Schleife, welcher ausserhalb des entsprechenden Summenzeichens auftaucht.

$$\begin{aligned} A(x) = {} & \underbrace{\mathcal{O}(x^2)}_{1} + \underbrace{1}_{2} + \underbrace{1}_{3} + \sum_{i=0}^{x-1} \Bigg(\underbrace{1}_{3} + \underbrace{1}_{4} + \underbrace{1}_{5} + \sum_{j=i}^{i} \Big(\underbrace{1}_{5} + \underbrace{\mathcal{O}(\log x)}_{6} + \underbrace{2}_{7} + \underbrace{0}_{8} \Big) \\ & + \underbrace{2}_{9} + \underbrace{0}_{10} \Bigg) \\ = {} & \mathcal{O}(x^2) + 1 + 1 + \sum_{i=0}^{x-1} \big(1 + 1 + 1 + 1 + \mathcal{O}(\log x) + 2 + 0 + 2 + 0\big) \\ = {} & \mathcal{O}(x^2) + x \cdot \mathcal{O}(\log x) + 8x + 2 \end{aligned}$$

$$\begin{aligned} \mathcal{O}\big(A(x)\big) & \overset{(5)}{=} \mathcal{O}(x^2) + \mathcal{O}(x \log x) + 8x + 2 \\ & \overset{(9)}{=} \mathcal{O}(x^2) + \mathcal{O}(x \log x) \\ & \overset{(8)}{=} \mathcal{O}(x^2), \qquad \text{da } x \log x \in \mathcal{O}(x^2) \end{aligned}$$

Aufgabe 3 (13 Punkte): Modula Programmierung

In einer Arzt-Praxis soll ein neues Patientenverwaltungssystem eingeführt werden. Zu diesem System gehört ein Modul, das die gerade im Wartezimmer befindlichen Personen verwaltet. Nötig ist dafür eine Datenstruktur, die es erlaubt, neue Patienten einzufügen und den Patienten herauszusuchen, der als nächstes an der Reihe ist. Diese Datenstruktur soll ein Ringpuffer sein, der wie in der Abbildung gezeigt aufgebaut ist.

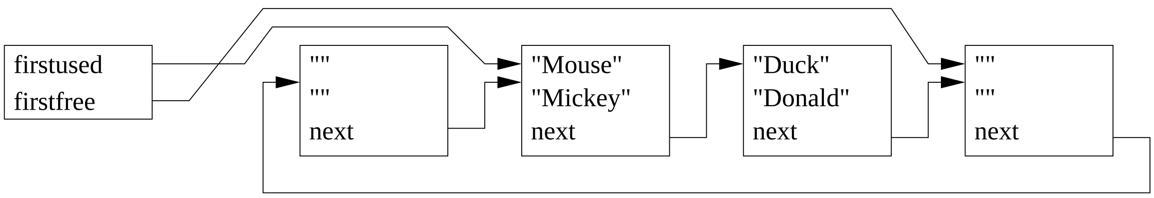


RingPuffer ist eine Verwaltungsstruktur, über die ein Zugriff auf das zuerst eingefügte belegte Element (firstused, der als nächstes aufgerufene Patient) und auf das erste leere Element (firstfree) möglich ist. Die Elemente selbst sind vom Typ RingElem. In ihnen werden die Namen der Wartenden eingetragen. Weiterhin erlauben sie den Zugriff auf das jeweils nächste Element. Dabei ist das letzte Element mit dem ersten verbunden, um einen Ring zu bilden. Wenn ein neuer Patient kommt, wird er in dem ersten leeren Element eingetragen und der Zeiger auf das erste leere Element ein Element weiter gesetzt. Wird ein Patient aufgerufen, muss der Zeiger auf das erste belegte Element (nach Auslesen des Namens) um ein Element weitergesetzt werden.
Die folgenden Prozeduren/Funktionen werden neben anderen zum fertigen System gehören:

- RingCreate(ring): Erzeugen eines leeren Ringpuffers. Die übergebene Verwaltungsstruktur ring vom Typ RingPuffer wird initialisiert (beide Pointer auf NIL gesetzt), aber es werden noch keine Elemente angelegt.
- RingInsert(ring,anzahl): Im Ring (Wartezimmer) wird für anzahl neue Patienten Platz gemacht (z.B. werden soviele neue Stühle in das Wartezimmer gestellt). Dafür werden anzahl (leere) Elemente in den Ring eingefügt, und zwar in dem bisher nicht schon von Patienten belegten Bereich.
- RingNewPatient(ring,nachname,vorname): Ein neuer Patient wird in dem Ring gespeichert. Falls keine belegbaren Elemente mehr existieren (firstfree^.next = firstused), wird RingInsert ausgeführt, um Platz zu schaffen. Die Bedingung firstfree^.next = firstused für einen vollen Ring bedeutet insbesondere, dass immer mindestens ein Element des Rings „frei“ sein muss, d.h. keinen Patienten enthält, ansonsten wäre der Fall eines vollen nicht von dem eines leeren Rings zu unterscheiden.
- RingNextPatient(ring,nachname,vorname): Der Name des nächsten Patienten wird zurückgegeben. Das von ihm im Ringpuffer belegte Element wird durch weitersetzen von firstused für die weitere Verwendung freigegeben. Falls sich kein Patient im Wartezimmer befindet (firstfree = firstused), werden leere Strings anstatt Nach- und Vornamen zurückgegeben.

Lösen Sie folgende Aufgaben:

1. **(3 Punkte)** Definieren Sie in Modula-2 die Datenstrukturen, die nötig sind, um die in den Abbildungen gezeigte Datenstruktur repräsentieren zu können. Dazu gehören RingPuffer und RingElem und evtl. benötigte Hilfstypen, wie Pointer.
2. **(7 Punkte)** Implementieren Sie die Prozedur RingInsert.
3. **(2 Punkte)** Geben Sie eine mögliche Anweisungsfolge an, um den unten gezeigten Ring zu erzeugen. Sie können auf die vier oben angegebenen Prozeduren/Funktionen zurückgreifen.
4. **(1 Punkte)** Wessen Name wird beim nächsten Aufruf von RingNextPatient zurückgegeben, und wie sieht die Datenstruktur danach aus?



Lösung:

1. *Definieren Sie die in Modula-2 nötigen Datenstrukturen:*

```
TYPE PRingElem = POINTER TO RingElem;

     RingPuffer = RECORD
       firstused, firstfree: PRingElem;
     END;

     RingElem = RECORD
       nachname, vorname: ARRAY [0..29] OF CHAR;
       next             : PRingElem;
     END;
```

2. *Definieren Sie die Prozedur RingInsert.*

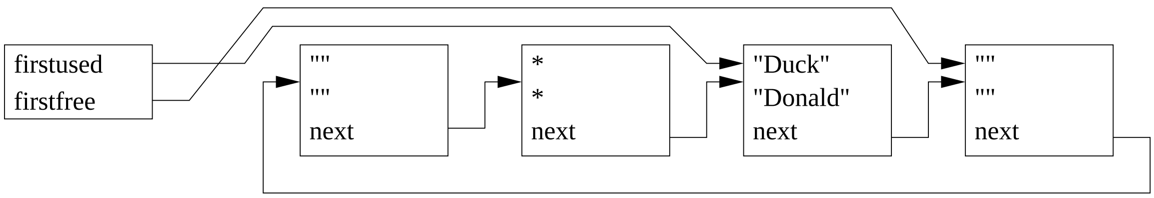
```
PROCEDURE RingInsert(VAR ring: RingPuffer; anzahl:CARDINAL);
  VAR tmpPtr, newElem: PRingElem;
  BEGIN
    IF (anzahl = 0) THEN RETURN; END;
    IF (ring.firstfree = NIL) THEN
      NEW(ring.firstfree);
      ring.firstused           := ring.firstfree;
      ring.firstfree^.next     := ring.firstfree;
      ring.firstfree^.nachname := "";
      ring.firstfree^.vorname  := "";
      DEC(anzahl);
    END;
    WHILE (anzahl > 0) DO
      NEW(tmpPtr);
      tmpPtr^.next         := ring.firstfree^.next;
      ring.firstfree^.next := tmpPtr;
      tmpPtr^.nachname     := "";
      tmpPtr^.vorname      := "";
      DEC(anzahl);
    END;
  END RingInsert;
```

3. *Geben Sie eine mögliche Anweisungsfolge an, um den unten gezeigten Ring zu erzeugen.*

```
RingCreate(ring);
RingInsert(ring, 4);
RingNewPatient(ring, "Mouse", "Mickey");
RingNewPatient(ring, "Duck", "Donald");
```

4. *Wessen Name wird beim nächsten Aufruf von RingNextPatient zurückgegeben, und wie sieht die Datenstruktur danach aus?*

Mickey Mouse

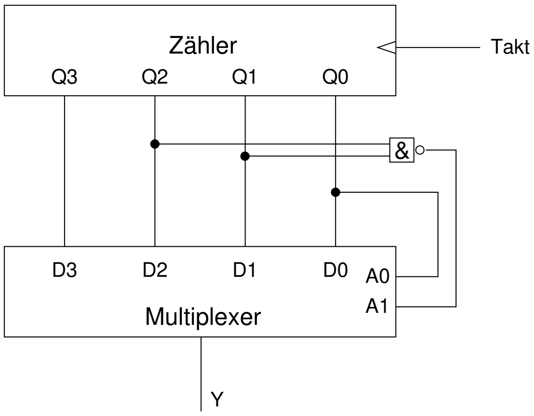


Aufgabe 4 (4 Punkte): Schaltwerke

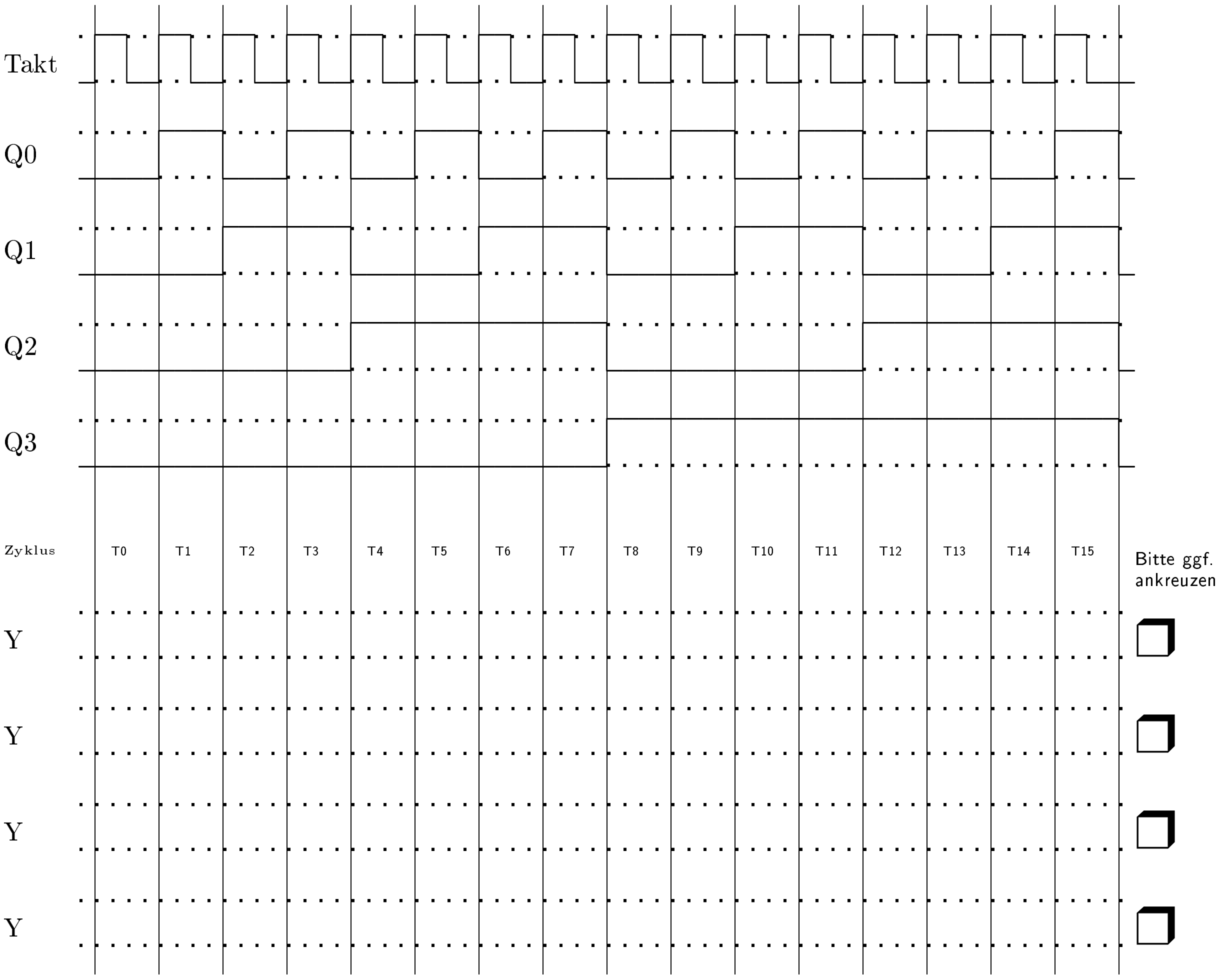
Gegeben sei die unten abgebildete Schaltung, bestehend aus einem synchronen Vorwärts-Binärzähler und einem Multiplexer mit den Dateneingängen `D0` bis `D3` und den Adreßeingängen `A0` und `A1`. Ab Takt Null wird die Taktleitung an einen Taktgenerator angeschlossen.

1. **(2 Punkte)** Wie lautet die Adreßbeschaltung des Multiplexers? Tragen Sie die zeitlichen Belegungen von A1 und A0, sowie den daraus resultierenden durchgeschalteten Eingang des Multiplexers in die Tabelle ein.

2. **(2 Punkte)** Wie sieht das vom Multiplexer erzeugte Ausgangssignal `Y` aus? Tragen Sie den Signalverlauf in das Diagramm ein. Falls Sie sich verschreiben und eine neue Zeile verwenden, **markieren Sie das Signal, welches gewertet werden soll, mit einem Kreuz in dem Kästchen rechts**!

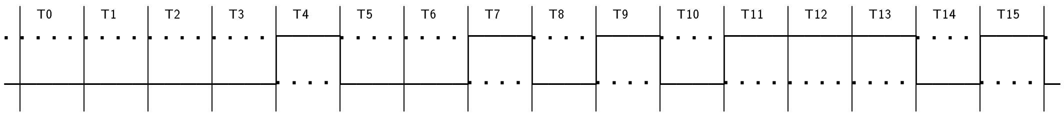


Q3	Q2	Q1	Q0	A1	A0	Eingang
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			



Lösung:

Q3	***Q2***	***Q1***	***Q0***	***A1***	***A0***	***Eingang***
0	*0*	*0*	*0*	*1*	*0*	*D2*
0	*0*	*0*	*1*	*1*	*1*	*D3*
0	*0*	*1*	*0*	*1*	*0*	*D2*
0	*0*	*1*	*1*	*1*	*1*	*D3*
0	*1*	*0*	*0*	*1*	*0*	*D2*
0	*1*	*0*	*1*	*1*	*1*	*D3*
0	*1*	*1*	*0*	*0*	*0*	*D0*
0	*1*	*1*	*1*	*0*	*1*	*D1*
1	*0*	*0*	*0*	*1*	*0*	*D2*
1	*0*	*0*	*1*	*1*	*1*	*D3*
1	*0*	*1*	*0*	*1*	*0*	*D2*
1	*0*	*1*	*1*	*1*	*1*	*D3*
1	*1*	*0*	*0*	*1*	*0*	*D2*
1	*1*	*0*	*1*	*1*	*1*	*D3*
1	*1*	*1*	*0*	*0*	*0*	*D0*
1	*1*	*1*	*1*	*0*	*1*	*D1*

Y


Aufgabe 5 (13 Punkte): Assembler und SCRAM

Ein String ist eine Folge von ASCII-codierten Zeichen (Tabelle siehe unten). Das Ende eines Strings wird stets mit dem Terminalsymbol *nil* markiert. Das Terminalsymbol selbst gehört immer zum String dazu.

1. (8 Punkte) Schreiben Sie ein Assemblerprogramm, das Strings filtert. Ihr Programm soll einen String derart filtern, daß nach Abarbeitung nur noch das Leerzeichen *spc*, Großbuchstaben, sowie das Terminalsymbol in ihm vorkommen (diese Zeichen sind in der Tabelle der ASCII-Codes **fett** gedruckt). Alle anderen Zeichen sollen entfernt werden, wodurch sich der String verkürzen oder gleich lang bleiben wird.

Beispiel: Der String `INFORMA!ablaTIK IST @#%&^! gar nicht TOLL` würde durch die Filterung zu dem String `INFORMATIK IST TOLL` werden (das Terminalsymbol ist hier nicht sichtbar).

Gehen Sie bei der Entwicklung des Programms davon aus, daß der String bereits im Speicher des SCRAM vorhanden ist und sein erstes Zeichen an der Adresse `String` steht. Der neue String soll nach Abarbeitung des Programms ebenfalls an dieser Adresse beginnen, verwenden Sie also keinen zusätzlichen Speicherbereich!

Hinweis: Prüfen Sie nicht alle Großbuchstaben einzeln ab! Um zu Überprüfen, ob ein Zeichen ein Großbuchstabe ist, reicht es aus, zu überprüfen, ob sein ASCII-Code größer oder gleich dem Code des `A`, sowie kleiner oder gleich dem Code des `Z` ist!

Lösung:

```
        ORG     0
        LDA     #String         ; Adresse des Strings laden
        STA     read            ; Pointer zum Lesen
        STA     write           ; Pointer zum Schreiben

loop                            ; Schleifenbeginn
        LDA     @read           ; nächstes Zeichen lesen
        JEQ     end             ; wenn NULL dann Programm beenden
        SUB     space           ; prüfen, ob Zeichen == Space
        JEQ     copy            ; wenn ja, dann gleich kopieren
        LDA     @read           ; nochmal Zeichen lesen
        SUB     A               ; Zeichencode kleiner als 65?
        AND     Neg             ;
        JNE     skip            ; dann überspringen
        LDA     @read           ; nochmals Zeichen lesen
        SUB     Z               ; Code größer als 90?
        AND     Neg             ;
        JEQ     skip            ; dann überspringen

copy                            ; Kopieren
        LDA     @read           ; Zeichen lesen
        STA     @write          ; Zeichen schreiben
        LDA     write           ; Schreib-Pointer weiterschieben
        ADD     #1
        STA     write
skip
        LDA     read            ; Lese-Pointer weiterschieben
        ADD     #1
        STA     read
        JMP     loop
end
        STA     @write          ; Terminalsymbol nicht vergessen!
        HLT

read    RES     1
write   RES     1
String  DAT     20,32,33,10,98,0
Neg     DAT     32768
Space   DAT     32
A       DAT     65
Z       DAT     90
```

```
        END
```

2. (2 Punkte) Der SCRAM soll erweitert werden. Bisher können bedingte Sprünge nur mit Hilfe des Prädikats *Wert des AC gleich Null?* unter Verwendung der Befehle `JNE` und `JEQ` durchgeführt werden, da vom AC lediglich die Steuerleitung `z` in die CLU zurückführt. In manchen Fällen ist es wünschenswert, wenn eine weitere Steuerleitung `n` existieren würde, die signalisierte, ob die letzte Rechenoperation ein negatives Ergebnis lieferte, um dann weitere bedingte Sprünge durchführen zu können.

- Welche Komponenten des SCRAM würde zweckmäßigerweise das Signal `n` generieren, das mit der Steuerleitung `n` in die CLU geleitet wird? Wie würde die Komponente das Signal erzeugen?

  Lösung:
  Eine solche Steuerleitung kann im AC realisiert werden. Das Signal der Steuerleitung entspricht im einfachsten Fall dem Zustand des höchstwertigen Bits. Dabei gilt zu beachten, daß der SCRAM 8- und 16-Bit-Zahlen verarbeitet. Es muß dafür gesorgt werden, daß intern immer mit 16 Bit gerechnet wird. Wenn sich Adressen im Akkumulator befinden, wird ebenfalls die n-Steuerleitung gesetzt, was jedoch dem Sinn (Zahl im AC negativ) nicht ganz entspricht, da Adressen nicht negativ sein können. Die Abfrage Jump on negative set *ist hier unzutreffend und müßte lauten* Jump on upper half memory address.

- Entwerfen Sie die RTL für einen neuen Befehl `JNS` (*jump on negative set*), der einen Sprung an die als Operand übergebene Adresse durchführt, falls die Steuerleitung `n` gesetzt ist.

  Lösung:

```
  t0 qE : MAR <- PC
  t1 qE : MBR <- MEM, PC <- PC +1
  t2 qE : IR <- MBR
n t3 qE : PC <- IR(0), uPC <- 0
  t4 qE : uPC <- 0
```

3. (3 Punkte) Prüfen Sie Ihr Programm aus Aufgabenteil 1 dahingehend, wie es von dem neuen Befehl größtmöglichen Nutzen ziehen könnte. Versehen Sie ihr Programm aus Teil 1 an entsprechenden Zeilen mit Zeilennummern und notieren Sie die *deutlich* neuen Assembler-Anweisungen.

Lösung:

```
        ORG     0
        LDA     #String         ; Adresse des Strings laden
        STA     read            ; Pointer zum Lesen
        STA     write           ; Pointer zum Schreiben

loop                            ; Schleifenbeginn
        LDA     @read           ; nächstes Zeichen lesen
        JEQ     end             ; wenn NULL dann Programm beenden
        SUB     space           ; prüfen, ob Zeichen == Space
        JEQ     copy            ; wenn ja, dann gleich kopieren
        LDA     @read           ; nochmal Zeichen lesen
        SUB     A               ; Zeichencode kleiner als 65?
        JNS     skip            ; dann überspringen
        LDA     Z               ; nochmals Zeichen lesen
        SUB     @read           ; Code größer als 90?
        JNS     skip            ; dann überspringen

copy                            ; Kopieren
        LDA     @read           ; Zeichen lesen
        STA     @write          ; Zeichen schreiben
        LDA     write           ; Schreib-Pointer weiterschieben
        ADD     #1
        STA     write
skip
        LDA     read            ; Lese-Pointer weiterschieben
        ADD     #1
        STA     read
        JMP     loop
end
        STA     @write          ; Terminalsymbol nicht vergessen!
        HLT
```

```
read    RES       1
write   RES       1
String  DAT       20,32,33,10,98,0
Neg     DAT       32768
Space   DAT       32
A       DAT       65
Z       DAT       90

        END
```

ASCII-Codes

Der hexadezimale Code eines Zeichens berechnet sich aus der Summe der Spalten- und Zeilenbezeichnungen.

Bsp: `spc = 0x20 + 0x0 = 0x20` was der dezimalen 32 entspricht.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	**nil**	soh	stz	etx	eot	enq	ack	bel	bs	ht	lf	vt	ff	cr	so	si
0x10	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
0x20	**spc**	!	”	#	$	%	&	’	(	)	*	+	,	–	.	/
0x30	0	1	2	3	4	5	6	7	8	9	;	;	<	=	>	?
0x40	@	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**
0x50	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**	[	\	]	↑	←
0x60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

Tabelle nach: N. Wirth: *Algorithmen und Datenstrukturen mit Modula-2*, 4. Aufl., B. G. Teubner, Stuttgart 1986

Name: ....

Matr.-Nr. ....

A5

SCRAM-Befehle

Code	Mnemonik	Zyklen	Bedeutung der Operation
$ 00 00	HLT	4	Halt, Ende der Programmabarbeitung.
$ m1 op	LDA *A*	5 6 8	Laden des Akkumulators.
$ m2 op	STA *A*	- 5 8	Speichern des Akkumulators.
$ m3 op	ADD *A*	5 7 9	Addiere den Wert zum Akkumulator.
$ m4 op	SUB *A*	5 7 9	Subtrahiere den Wert vom Akkumulator.
$ m5 op	MUL *A*	5 7 9	Multipliziere Akkumulator mit dem Wert.
$ m6 op	DIV *A*	5 7 9	Dividiere Akkumulator durch den Wert.
$ m7 op	AND *A*	5 7 9	Bitweises UND des Akkus mit dem Wert.
$ m8 op	OR *A*	5 7 9	Bitweises ODER des Akkus mit dem Wert
$ m9 op	XOR *A*	5 7 9	Bitweises Exklusiv–ODER des Akkus
$ 0A 00	NOT	5	Bitweise Negation des Akkumulators.
$ 0B op	JMP *A*	4	Unbedingter Sprung an die Adresse *A*.
$ 0C op	JEQ *A*	5	Bedingter Sprung an die Adresse *A*, wenn Akku den Wert Null enthält.
$ 0D op	JNE *A*	5	Bedingter Sprung an die Adresse *A*, wenn Akku ungleich Null.
$ 0F 00	NOP	4	Die “No Operation”.

Die Bezeichnung *op* beim Befehlscode steht dabei für die Operandenadresse, bzw. den unmittelbaren Wert. In der Spalte *Zyklen* steht die Anzahl der Taktzyklen bei unmittelbarer-, direkter- und indirekter Adressierung. Die mit *m* gekennzeichneten Bits enthalten die Adressierungsart.

SCRAM Adressierungsarten

Bit 15	14	Adr.-art	Bsp.
0	0	Direkt	LDA A
0	1	Unmittelbar	LDA #A
1	0	Indirekt	LDA @A

SCRAM Schaltbild