Fachbereich
Informatik

Klausur Informatik 2
5. Oktober 1999

Name: ....

Vorname: ....

Matr.-Nr. ....

Bearbeitungszeit: 120 Minuten

Spielregeln (**Jetzt lesen!**):

- Benutzen Sie für die Lösung der Aufgaben **nur** das mit diesem Deckblatt ausgeteilte Papier. Lösungen, die auf anderem Papier geschrieben werden, können **nicht** bewertet werden. Schreiben Sie Ihre Lösung auch auf die Rückseiten der Blätter; benötigen Sie für eine Lösung mehr als ein Blatt, finden Sie am Ende der Klausur Leerblätter. Zusätzliches Papier können Sie von den Tutoren bekommen.
- Tragen Sie jetzt (vor Beginn der eigentlichen Bearbeitungszeit!) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.
- Schreiben Sie deutlich! Unleserliche oder zweideutige Lösungen können nicht gewertet werden.
- Schreiben Sie **nicht** mit Bleistift!
- Schreiben Sie nur in **blau** oder **schwarz**!

Viel Erfolg!

Bewertung
(bitte offenlassen :–)

Aufgabe	Punkte	Erreichte Punkte
1	12	
2	12	
3	9	
4	17	
Summe	50	

• **AUFGABE 1 (12 Punkte) Programmverifikation, Hoare-Kalkül.**

Jede natürliche Zahl kann in ein Produkt von Primzahlen zerlegt werden. So gilt z.B. $60 = 2 \cdot 2 \cdot 3 \cdot 5$. Für jede natürliche Zahl gibt es genau eine solche Zerlegung.

Die unten abgebildete Prozedur ermittelt die Primfaktoren einer übergebenen Zahl `x` und legt sie im Array `faktor` ab. Sie prüft, ob die Zahl durch eine Primzahl teilbar ist. Falls nicht, wird mit der nächsten Primzahl weiterprobiert. Andernfalls wird die Primzahl abgespeichert und der Algorithmus mit dem Ergebnis der Division weitergeführt. Der Algorithmus bricht ab, wenn durch die letzte Primzahl dividiert wurde, d.h. `quotient=1`. Für `x=60` würde dann gelten `faktor={2,2,3,5}`. Das Array `primzahl` enthält bei Eintritt in die Prozedur alle Primzahlen von 2 bis `MAX(CARDINAL)` in aufsteigender Reihenfolge.

1. (3 Punkte) Geben Sie jeweils eine geeignete Vorbedingung, Nachbedingung und Invariante für die `WHILE`-Schleife an!
2. (6 Punkte) Beweisen Sie die **partielle** Korrektheit des Rumpfes mit Hilfe des Hoare-Kalküls! Sie können den Programmvordruck auf der nächsten Seite benutzen.

   Hinweis: Falls `q=a DIV b` und `r=a MOD b`, dann gilt `q·b+r=a`.
3. (2 Punkte) Für den Korrektheitsbeweis von `WHILE`-Schleifen müssen Invarianten gefunden werden. Invariant heißt soviel wie „nicht veränderlich", trotzdem ändert sich ein Aspekt einer Invarianten beim Schleifendurchlauf. Welcher Aspekt ist veränderlich und welcher Aspekt bleibt tatsächlich invariant?
4. (1 Punkt) Die Regeln und Axiome des Hoare-Kalküls können auch auf andere Programmiersprachen wie z.B. C, Pascal u.s.w. übertragen werden. Wäre eine Übertragung auf OPAL ebenfalls möglich? Erläutern Sie!

```
PROCEDURE PrimFac( x:CARDINAL; primzahl:ARRAY OF CARDINAL; VAR faktor:ARRAY OF CARDINAL );
(* Zerlegt x in Primfaktoren und legt diese in faktor ab. primzahl enthält Primzahlen *)
(* in aufsteigender Folge, d.h. primzahl[0]=2, primzahl[1]=3, primzahl[2]=5 u.s.w.    *)

VAR f, p, n, quotient, rest: CARDINAL;

BEGIN
f:= 0; p := 0; n := x;                  (* Initialisierung der Laufindizes          *)
WHILE n # 1 DO
    quotient := n DIV primzahl[p];
    rest := n MOD primzahl[p];
    IF rest = 0 THEN                    (* n ist ohne Rest durch primzahl[p] teilbar  *)
        faktor[f] := primzahl[p];       (* => Primzahl abspeichern                    *)
        f := f+1;
        n := quotient;                  (* weitermachen mit Ergebnis der Division     *)
    ELSE                                (* n ist nicht durch primzahl[p] teilbar      *)
        p := p+1;                       (* => nächste Primzahl probieren              *)
    END; (*IF*)
END; (*WHILE*)
END PrimFac;
```

{PRE:

```
f := 0; p := 0; n := x;

WHILE n # 1 DO

    quotient := n DIV primzahl[p];

    rest := n MOD primzahl[p];

    IF rest = 0 THEN

        faktor[f] := primzahl[p];

        f := f+1;

        n := quotient;

    ELSE

        p := p+1;

    END; (*IF*)

END; (*WHILE*)
```

{POST:

• **AUFGABE 2 (12 Punkte) Aufwandsberechnungen, $\mathcal{O}$-Kalkül.**

1. (7 Punkte) Bestimmen Sie den **Worst-Case-Aufwand A(n)** in Abhängigkeit von `n`, der Anzahl der Punkte des im folgenden gegebenen Algorithmus'.

   Nehmen Sie zur Aufwandsberechnung bitte an, daß alle Elementaroperationen, wie Zuweisung, Vergleich, Addition, Parameterübergabe bei Prozeduraufruf, eine Zeiteinheit benötigen. Die Funktion `Angle(p1, p2: Point)` wird in konstanter Zeit $\mathcal{O}(1)$ ausgeführt.

   Sorgen Sie dafür, daß Ihre Lösung nachvollziehbar ist, insbesondere daß die einzelnen Terme Ihrer Lösung einzelnen Zeilen im Programm zugeordnet werden können.

2. (1 Punkt) Bitte bestimmen Sie nun die Aufwandsklasse des in der vorigen Teilaufgabe entwickelten Ausdrucks für den Aufwand. Dokumentieren Sie Ihren Lösungsweg.

3. (4 Punkte) Finden Sie heraus, ob die in Teilaufgabe 2 bestimmte Aufwandsklasse eine dichte obere Schranke des von Ihnen in Teilaufgabe 1 berechneten Aufwands ist.

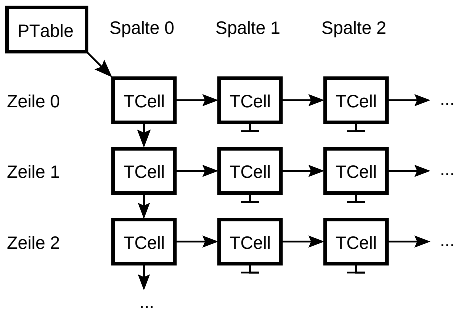
```
CONST nMax = 1000;
TYPE Point = RECORD x,y: INTEGER END;
PROCEDURE ConvexHullWrap(VAR p: ARRAY[0..nMax] OF Point; n: CARDINAL);
VAR i, min, m:      INTEGER;
    minangle, v, a: REAL;
    t:              Point;
BEGIN
  min := 0;

  FOR i := 1 TO n-1 DO             (* Finde Punkt mit kleinstem 'y'-Wert (Startpunkt)      *)
      IF p[i].y < p[min].y THEN min := i; END;
  END;

  m := -1;
  p[n] := p[min];                  (* Marke fuer den Abbruch der REPEAT-Schleife:          *)
  minangle := 0.0;

  REPEAT                           (* Wiederhole, bis man wieder auf den Startpunkt trifft *)
      m := m+1;
      t := p[m]; p[m] := p[min]; p[min] := t;      (* Swap(p[M], p[min])                   *)
      v := minangle; minangle := 360.0;
      min := n;

      FOR i := m+1 TO n DO         (* Suche Punkt für minimalen Winkel                     *)
          a := Angle(p[m], p[i]);
          IF (a > v) AND (a < minangle) THEN
              min := i; minangle := a;
          END; (* IF *)
      END; (* FOR *)
  UNTIL min = n;
END ConvexHullWrap;
```

Rechenregeln (O-Kalkül):

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

• AUFGABE 3 (9 Punkte) Imperative Programme, Rekursive Datenstrukturen in Modula.

Ein Tabellenkalkulationsprogramm soll Tabellen beliebiger Größe speichern und bearbeiten können. Bei der Erzeugung der Tabelle werden die Anzahl der Zeilen und der Spalten angegeben. Es soll möglich sein, später sowohl Zeilen als auch Spalten effizient einzufügen bzw. zu entfernen. Deshalb soll die Tabelle als eine Liste von Listen gespeichert werden: Eine Liste enthält die Felder der ersten Spalte der Tabelle, jedes Element dieser Liste ist der Anfang einer weiteren Liste, in der alle Felder dieser Zeile enthalten sind.



Außerdem soll jedes einzelne Feld einer Tabelle in der Lage sein, einen von drei Datentypen zu speichern: entweder eine Zeichenkette (String) oder eine Ganze Zahl (Integer) oder eine Reelle Zahl (Real). Je nach Eingabe des Benutzers kann sich das Datum und der Datentyp jedes Feldes separat zur Laufzeit ändern.

1. (3 Punkte) Definieren Sie in Modula einen geeigneten Datentyp `TCell` für die einzelnen Felder der Tabelle, der in der Lage ist, die drei verschiedenen Datentypen zu speichern. Außerdem soll mit diesem Datentyp die oben beschriebene Liste von Listen aufgebaut werden können.

   Definieren Sie weiterhin einen Datentyp `PTable` für eine ganze Tabelle.

2. (1 Punkt) Wieviele Bytes belegt ein einzelnes Tabellenfeld im Speicher? Gehen Sie bei der Berechnung von folgenden Größen für die dem Modula-Compiler bekannten Datentypen aus:

`INTEGER`	4 Byte	`SHORTINT`	2 Byte
`CARDINAL`	4 Byte	`SHORTCARD`	2 Byte
`REAL`	4 Byte	`LONGINT`	4 Byte
`BOOLEAN`	1 Byte	`LONGCARD`	4 Byte
`CHAR`	1 Byte	`BYTE`	1 Byte
`String`	256 Byte	`WORD`	1 Byte
Aufzählungstyp	1 Byte	`LONGREAL`	8 Byte
Pointertyp	4 Byte	`ADDRESS`	4 Byte

   Die Ausrichtung der Daten an Wortgrenzen (Alignment) muß nicht berücksichtigt werden.

3. (3 Punkte) Definieren Sie eine Prozedur `NewTable` in Modula, die die initiale Größe der Tabelle als Argument erhält und einen Zeiger auf die Tabelle zurückliefert. Die Tabellenfelder sollen von dem in Teilaufgabe 1 entwickelten Datentyp sein und initial Integer-Zahlen speichern.

```
PROCEDURE NewTable( Width, Height:CARDINAL ) : PTable;
```

4. (2 Punkte) Definieren Sie eine Prozedur `PutString` in Modula, die in das Feld an der Position `x,y` einen String `s` schreibt. Die Prozedur bekommt einen Zeiger auf eine Tabelle, die Koordinaten `x,y` und den String `s` als Argument übergeben.

```
PROCEDURE PutString( Table:PTable; x,y:CARDINAL; s:String );
```

• AUFGABE 4 (17 Punkte) SCRAM-Erweiterung, Assembler.

Gegeben ist der in der Vorlesung behandelte SCRAM-Rechner und seine Assemblerbefehle. Die zur Lösung notwendigen Unterlagen sind der Klausur beigefügt.

Assemblerprogrammierung Teil 1 (7 Punkte)

1. (7 Punkte) Schreiben Sie ein Assemblerprogramm, das für eine Konstantenliste den arithmetischen Mittelwert bestimmt und das Ergebnis in einem Wort `Erg` abspeichert.

   Die Anzahl der Konstanten sei im ersten Wort der Liste enthalten. Die folgende Konstantenliste ist dabei nur als Beispiel gedacht. Ihr Programm soll auch für jede andere Konstantenliste des gleichen Typs funktionieren.

   Beispielliste:

```
Anzahl   DAT  5     ; Anzahl der Einträge
Feld     DAT  -20   ; 1. Eintrag
         DAT  -4    ; 2. Eintrag
         DAT  13    ; 3. Eintrag
         DAT  -10   ; 4. Eintrag
         DAT  -2    ; 5. Eintrag
```

SCRAM (7 Punkte)

Der SCRAM ist bislang eine Einadreßmaschine: alle Instruktionen des SCRAM arbeiten mit maximal einem Operanden. Das bedeutet, daß alle Rechenoperationen eine Verknüpfung von Operand und Akkumulator sind.

Im Rahmen dieser Aufgabe soll der SCRAM die Möglichkeit erhalten, Mehradreßinstruktionen auszuführen. Während beispielsweise eine Addition bislang bedeutet, daß der Operand zu dem Wert im Akkumulator addiert werden soll und das Ergebnis im Akkumulator liegt, soll bei einer Mehradreßaddition der erste Operand mit dem zweiten Operanden summiert und das Ergebnis sowohl im zweiten Operanden als auch im Akkumulator abgelegt werden. Die Operation `ADDX op1, op2` bewirkt beispielsweise, daß der Wert der Variablen `op1` zu dem Wert der Variablen `op2` addiert wird und das Ergebnis sowohl in der Variablen `op2`, als auch im Akkumulator hinterlegt wird. Im Speicher belegt diese Instruktion 2 Worte:

0001 0011	op1
0000 0000	op2

2. (5 Punkte) Entwickeln Sie die RTL für eine Zweiadreßaddition, wobei beide Operanden *direkt* adressiert werden sollen. Verwenden Sie den Opcode `$13` und die Adressierungsart `$00`

3. (2 Punkte) Welche Teile des SCRAM müssen hardwareseitig erweitert werden? Welche Konventionen müssen zusätzlich vereinbart werden (beachten Sie den Operationscode)?

Assemblerprogrammierung Teil 2 (3 Punkte)

Der SCRAM wurde erweitert. Fast alle Befehle haben nun ein Zweiadreß-Äquivalent, mit dem zwei Operanden gleichzeitig verarbeitet werden können. Ihre Mnemonischen Bezeichner erhalten ein `X` als letztes Zeichen, wie `ADDX` als Zweiaddreßäquivalent von `ADD`. Beide Operanden können wahlfrei in allen drei bislang bekannten Adressierungsarten adressiert werden.

4. (2 Punkte) Modifizieren Sie ihr Programm aus Teilaufgabe 1 derart, daß es größtmöglichen Nutzen aus den neuen Zweiadreß-Befehlen zieht. Alle neuen Zweiadreßbefehle sind in der unten stehenden Tabelle abgedruckt.

5. (1 Punkt) Vergleichen Sie die Laufzeit Ihrer beiden Programme. Wie groß schätzen Sie den erwarteten Geschwindigkeitsvorteil? Wovon hängt der Geschwindigkeitsvorteil ab?

Material zu Aufgabe 4

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”.

SCRAM-Adressierungsarten

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

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 Schaltbild

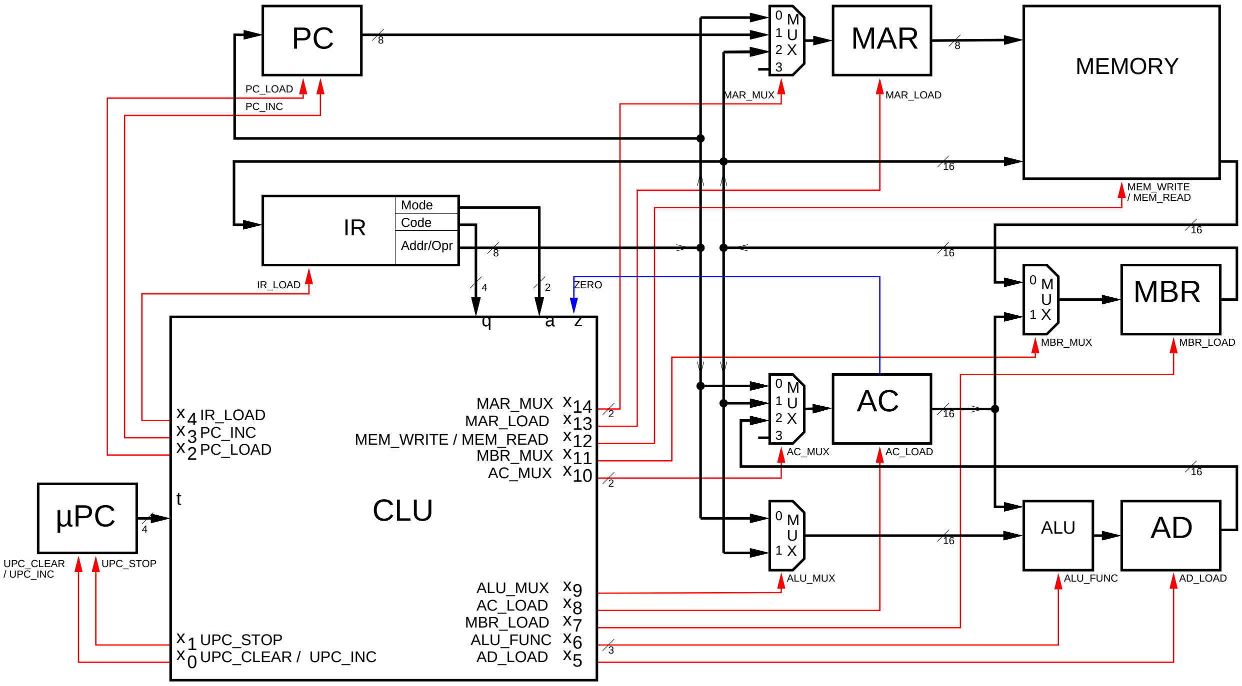


Tabelle der erweiterten Zweiadreßbefehle (nur für Aufgabenteil 3)

Mnemonik	Bedeutung der Operation
ADDX	Addiere op1 zu op2
SUBX	Subtrahiere op2 von op1
MULX	Multipliziere op1 und op2
DIVX	Dividiere op1 durch op2
ANDX	Führe bitweises UND von op1 und op2 durch
ORX	Führe bitweises ODER von op1 und op2 durch
XORX	Führe bitweises Exklusiv-ODER von op1 und op2 durch